Thesis for The Degree of Licentiate of Engineering

Foundations of Information-Flow Control and Effects

Carlos Tomé Cortiñas

CHALMERS

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2022

Foundations of Information-Flow Control and Effects
Carlos Tomé Cortiñas

Unit of Information Security
Division of Computing Science
Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Gothenburg, Sweden 2022.

Abstract

In programming language research, information-flow control (IFC) is a technique for enforcing a variety of security aspects, such as confidentiality of data, on programs. This Licenciate thesis makes novel contributions to the theory and foundations of IFC in the following ways: Chapter A presents a new proof method for showing the usual desired property of noninterference; Chapter B shows how to securely extend the concurrent IFC language MAC with asynchronous exceptions; and, Chapter C presents a new and simpler language for IFC with effects based on an explicit separation of pure and effectful computations.

Acknowledgements

First, I would like to thank Alejandro Russo for his supervision. I would also like to extend my gratitude to my coauthors and collaborators for a fruitful collaboration, which in particular has resulted in this thesis.

A huge thanks to my former and present colleagues and friends at the department for providing such a friendly and fun work environment. Thanks Fabian, Nachi, Irene, Víctor, Abhiroop, Matti, Sandro, Agustín, Elisabet, Alejandro (el otro), Ivan, Mohammad, Benjamin, Liam, and Robert. If you feel that I have forgotten you, then your name probably deserves to be in this list, my apologies.

Special thanks to my friends in Gothenburg, back home in Spain, and the rest of the world for making life most enjoyable. Thanks Μαρία, Ειρήνη, Antonio, Duarte, Arturo, Rachele, and the many of you whom I may have forgotten. Last but not least, I cannot begin to express my thanks to my family for their unconditional love and support throughout these years. Y gracias a mis padres Élida y Eduardo por estar siempre ahí.

Contents

Overview

Introduction

This Licenciate thesis investigates static information-flow control (IFC) on higher-order programming languages with effects.

I.1. Information-Flow Control

Language-based IFC [SM03] is an approach to security in computing systems that aims to protect confidentiality and integrity of users' data at the level of programming languages. Typically, information-flow control programming languages track *where* and *how* information propagates during the execution of programs, and enforce that all possible flows of information, within and out of the program, abide by a given security policy.

Security policies usually consists of a group of principals, i.e. those actors whose information is at stake, represented by security levels, and a specification of how information is allowed to flow among them. For example, the principals, say Alice, Bob and Charlie, could be employees of the same company, and thus, the security policy would consist of at least three security levels: `Alice`, `Bob` and `Charlie`. Furthermore, if Alice trusts Bob but not Charlie with her private data, then the security policy would specify that information is allowed to flow from `Alice` to `Bob` but not from `Alice` to `Charlie`.

Programming languages for IFC usually let programmers assign security levels, from a given security policy, to the different parts of the program that manipulate users' data. For example, if the program reads data inputted by Charlie, then there would be a channel, which programs can use, that has label `Charlie`. A sensible question to ask then is: what does it mean for the program to abide by the security policy? This question is usually answered in the form of a formal security condition that combines (a subclass of) programs, certain security policies, and the "execution" semantics of the programming

language, i.e. what do programs compute and what can be observed from their execution at some security levels. When programs satisfy the security condition we say that they are secure.

IFC enforcement mechanisms broadly fall in two categories: *static*, where programs are analysed before execution, i.e. at compile time, and they are allowed to run only if they comply with the security policy; and *dynamic*, where all programs are executed, but their execution will be aborted or modified at the point where they, if so, are about to violate the security policy. Additionally these approaches might be combined to yield *hybrid* enforcements. Note that enforcement mechanisms will always over approximate the security property, i.e. all accepted programs are secure, but will not be complete, i.e. some programs deemed as "insecure" are in fact secure. Whether a program is secure with respect to a nontrivial security property is in general undecidable [Ric53].

To summarize, language-based IFC approaches to security consist in

- a framework for specifying security policies;
- a programming language;
- a formal security property expressing what does it mean for programs to be secure with respect to security policies; and
- an enforcement mechanism that ensures programs comply with security policies.

In the rest of this section, we describe in more detail some of the points above.

I.1.1. Security Policies

Security policies are commonly formalized using lattices [Den76]. The elements of a lattice, or security levels, sometimes also called labels, or sensitivities, are an abstraction over the distinct principals (or sets of), and the relation specifies the allowed flows of information. Security levels and principals need not correspond to each other one-to-one; for instance, in the previous example there would be a security level `Alice or Bob`, for data belonging to either Alice or Bob, but this level doesn't correspond to any of the principals.

In the simplest scenario, the security policy defines two security labels `L`, for *public*, and `H`, for *secret*, and the relation states that every flow of information is allowed except from `H` to `L`—i.e. flows to equal or more sensitivity are allowed, e.g. from public to secret, but flows from more to less sensitivity are forbidden, e.g. from secret to public. The so-called attacker frequently belongs to the security level `L`, i.e. the attacker is modelled as part of the system. In principle,

the attacker can only observe public data, in accordance with the security policy, however their (malicious) intention is to also observe secret data, which clearly violates the security policy.

The public-secret situation generalizes to more complex scenarios, i.e. those with nontrivial security lattices; the attacker belongs to an arbitrary security level, "public," and confidential data to another level, "secret," such that flows of information from the later to the former are disallowed.

I.1.2. Security Properties

Security properties connect security policies and the execution semantics of programs. Usually these properties are variations of the classical noninterference [GM82] which states that secret data can not influence what attackers observe from the public outputs of a program:

Definition I.1.1 (Noninterference)**.** A program satisfies *noninterference* if its public outputs are independent of its secret inputs.

What constitutes program's secret inputs and public outputs, and hence what is the precise statement of noninterference, varies with respect to

- the features of the programming language under consideration, and thus what behaviour programs can exhibit, e.g. general recursion, printing, memory references; and
- the observational power of the attackers, i.e. what can they observe from the execution of programs, e.g. distinguishing termination from divergence, timing output events, inspecting the intermediate contents of memory.

An approach to formally show that a program satisfies noninterference is by considering the public outputs that it generates in several executions for distinct secret input values. If varying the secret input does not affect the public outputs of the program, that is, the public outputs agree on all executions, then it must the case that those outputs are truly independent of the secret inputs, and hence the program is noninterferent.

I.1.3. Enforcement Mechanisms

Enforcement mechanisms can roughly be divided in two categories, *static* and *dynamic*, according to whether the enforcement happens before the execution of programs, i.e. at compile time, or during their execution, i.e. at runtime. Other mechanisms, so-called *hybrid*, combine static and dynamic enforcement.

- Static enforcement mechanisms consists in statically analysing programs to determine whether all the flows of information are permitted by a given security policy. Static analysis for IFC appear in several veins, on the traditional side the static analyser takes a program as input and outputs whether the program is secure or not. Modern approaches built the analysis into the programming language by means of a type system, in the sense, that all well-typed programs are secure.

- Dynamic enforcement mechanisms consists in a runtime monitor that executes along with programs and halts or modifies the execution of the program in case some flow of information violates the security policy.

II

Contributions

This thesis contributes to the foundations of IFC with effects in two areas: proof methods for noninterference and IFC languages with effects.

Proof Methods This thesis presents a new method for proving noninterference for higher-order IFC programming languages. In the literature noninterference has been typically proved different by appealing to an array of techniques: term erasure (e.g. [LZ10; RCH08]), logical relations (e.g. [TZ04; BA15]), parametricity (e.g. [Alg18]), and adequate denotational semantics (e.g. [Aba+99; Kav19]). In contrast to these, the method presented in this thesis arises from the insight that to understand whether programs are secure it is enough to look at a representative class of them for which noninterference is trivially true, e.g. programs of certain type are equivalent to a program that does not mention *syntactically* a secret input. This thesis uses normalization to obtain the representative class of programs, and hence the technique is dubbed *noninterference by normalization*.

IFC Languages with Effects Effects are an important feature of practical programming languages, and hence practical IFC languages should incorporate them in some form. However one must be careful as effects open new and unintended channels that can be used to leak secret data to the public.

First, this thesis presents a new design for an extension of MAC [Vas+18], a concurrent language for static IFC, with *asynchronous exceptions*. Asynchronous exceptions are a useful feature that permit threads to communicate with each other and interact with the runtime system. In the IFC setting they allow thread safe communication among threads that operate with data at different security levels. As a side-effect, this enables new programming patterns that would have been insecure in MAC, for instance, speculative execution.

Furthermore, this thesis presents a new approach for designing IFC languages with effects. Usually, these languages are designed and implemented in a monolithic fashion (e.g. MAC). Although there are plenty effect-free languages for static IFC in the literature (e.g. [Aba+99; SI08]), when it comes to incorporating effects it is typically the job of the researcher to modify the type system just enough so that one can express some interesting secure programs that perform effects. In this thesis we propose a new mechanism, which a sort of *distributivity* construct, that can be used to seamless integrate languages for effects, for example in the form of graded monads, and the aforementioned effect-free IFC languages.

Figure II.1 summarizes the specific results included in this thesis, relates them to the areas mentioned above, and indicates the venues where research articles have been published.

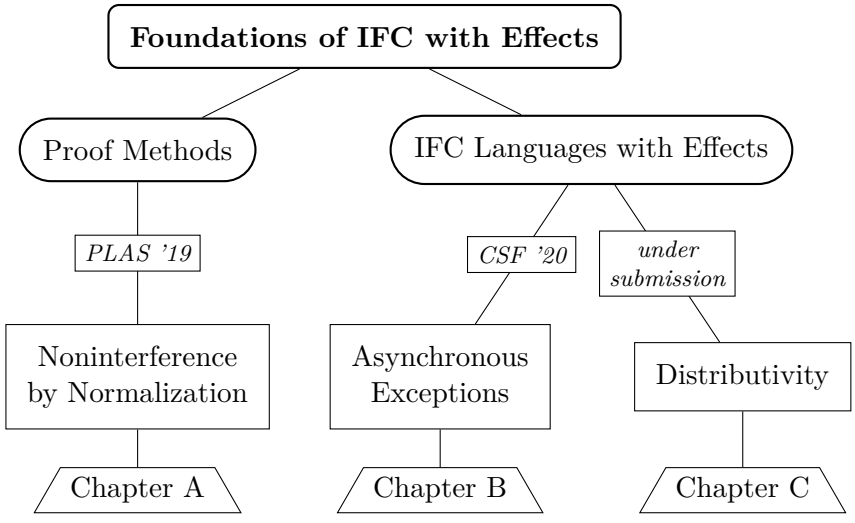


Figure II.1.: Thesis' contributions overview

In the rest of this chapter, we describe the articles that compose this thesis in more detail.

Simple Noninterference by Normalization

Carlos Tomé Cortiñas and Nachiappan Valliappan

The main contribution of this paper consists in a novel proof technique for proving noninterference. This technique exploits the insight that to reason about information-flow properties of a programming language, it is not necessary

to consider all possible programs, where programs might be arbitrarily complex, but that it is enough to consider a smaller (and simpler) class that in some sense represents all of them.

Based on this idea, this paper presents a new proof of noninterference for a static IFC calculus based on a terminating fragment of the calculus that formalizes the Haskell IFC library SecLib [RCH08]. The calculus is a simply-typed λ-calculus (STLC) extended with unit, product, and sum types, and a family of monads for each security level. The simpler class of programs, from which a noninterference property follows rather directly, is characterized as the normal forms of a standard equational theory which includes, among others, β, η and δ equations for monadic types.

Specifically, the paper describes in detail i) the class of normal forms ii) a procedure, based on NbE, for obtaining normal forms for arbitrary terms iii) a proof that indeed normal forms faithfully represent classes of equivalent terms, and iv) a proof of noninterference based on analysing the normal forms. The results in this paper have been mechanized in the proof assistant Agda [Abe+05].

Statement of Contributions This paper was coauthored with Nachiappan Valliappan. Both Nachiappan and Carlos contributed equally to the formalization, the Agda mechanization, and the writing of the paper.

This paper was published in the Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security, CCS 2019, London, United Kingdom, November 11-15, 2019 [TV19]. A reformatted version appears as Chapter A of this thesis.

Securing Asynchronous Exceptions

Carlos Tomé Cortiñas, Marco Vassena, and Alejandro Russo

This paper explores how to extend a concurrent language for static IFC, in this case based on MAC [Vas+18], with asynchronous exceptions.

Asynchronous exceptions, sometimes called interrupts, are a useful language feature that enables different threads of execution to communicate and interact with each other and the runtime system. The naive combination of asynchronous exceptions with existing features of IFC languages (e.g. concurrency and synchronization variables) can, in principle, open up new possibilities of information leakage.

The main contribution of this paper is to show how to securely incorporate asynchronous exceptions to concurrent MAC. For that, the paper presents an

extension to the syntax of MAC with new primitives that i) permit threads to refer to other threads, a feature that is not available in concurrent MAC, and ii) permit threads to send asynchronous exceptions to other threads in the system. This paper further present an operational semantics for this extension of MAC, inspired by the work on asynchronous exceptions in Haskell [Mar+01]. In this paper it is also proved that the resulting language satisfies a strong notion of security, namely progress-sensitive noninterference (PSNI). The results in this paper have been mechanized in the proof assistant Agda.

Statement of Contributions This paper was coauthored with Marco Vassena and Alejandro Russo. Carlos suggested the idea of studying asynchronous exceptions in the context of IFC. Carlos devised the calculus and its formal semantics, and mechanized them along with the security guarantees in Agda. Carlos wrote the technical sections of the paper.

This paper was published in the 33rd IEEE Computer Security Foundations Symposium, CSF 2020, Boston, MA, USA, June 22-26, 2020 [TVR20]. A reformatted version appears as Chapter B of this thesis.

Pure Information-Flow Control with Effects Made Simple

Carlos Tomé Cortiñas and Alejandro Russo

The main contribution of this paper is a novel primitive distr that permit us to extend existing languages for effect-free IFC (e.g. DCC [Aba+99], λ_{SC} [SI08]) with effects in a principled manner.

To evidence this, this paper describes a number of extensions of the sealing calculus (SC) [SI08] with different forms of effects, e.g. printing or global store. These extensions consist on combining naively SC with a graded monad for effects, which tracks fine-grainedly concrete effects, and the primitive distr. In this paper it is also proved that the resulting languages from these extensions are secure, i.e. they satisfy suitable versions of termination-insensitive noninterference (TINI). This paper also presents an implementation of this idea as a proof-of-concept library in Haskell. The results in this paper have been partially mechanized in the proof assistant Agda.

Statement of Contributions This paper was coauthored with Alejandro Russo. Carlos devised the idea of splitting effect-free and effectful IFC. Carlos formalized and mechanized the calculi presented in the paper. Carlos wrote most of the paper.

This paper is under submission. The latest version of the manuscript appears as Chapter C of this thesis.

Bibliography

[Aba+99] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. "A Core Calculus of Dependency". In: *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*. Ed. by Andrew W. Appel and Alex Aiken. ACM, 1999, pp. 147–160. DOI: 10.1145/292540.292555. URL: https://doi.org/10.1145/292540.292555 (cit. on pp. 7, 8, 10).

[Abe+05] Andreas Abel, Guillaume Allais, Jesper Cockx, Nils Anders Danielsson, Philipp Hausmann, Fredrik Nordvall Forsberg, Ulf Norell, Víctor López Juan, Andrés Sicard-Ramírez, and Andrea Vezzosi. *Agda 2*. 2005–. URL: https://wiki.portal.chalmers.se/agda/pmwiki.php (cit. on p. 9).

[Alg18] Maximilian Algehed. "A Perspective on the Dependency Core Calculus". In: *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. Ed. by Mário S. Alvim and Stéphanie Delaune. ACM, 2018, pp. 24–28. DOI: 10.1145/3264820.3264823. URL: https://doi.org/10.1145/3264820.3264823 (cit. on p. 7).

[BA15] William J. Bowman and Amal Ahmed. "Noninterference for free". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. Ed. by Kathleen Fisher and John H. Reppy. ACM, 2015, pp. 101–113. DOI: 10.1145/2784731.2784733. URL: https://doi.org/10.1145/2784731.2784733 (cit. on p. 7).

[Den76] Dorothy E. Denning. "A Lattice Model of Secure Information Flow". In: *Commun. ACM* 19.5 (1976), pp. 236–243. DOI: 10.1145/360051.360056. URL: https://doi.org/10.1145/360051.360056 (cit. on p. 4).

[GM82] Joseph A. Goguen and José Meseguer. "Security Policies and Security Models". In: *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*. IEEE Computer Society, 1982, pp. 11–20. DOI: 10.1109/SP.1982.10014. URL: https://doi.org/10.1109/SP.1982.10014 (cit. on p. 5).

[Kav19] G. A. Kavvos. "Modalities, cohesion, and information flow". In: *Proc. ACM Program. Lang.* 3.POPL (2019), 20:1–20:29. DOI: 10.1145/3290333. URL: https://doi.org/10.1145/3290333 (cit. on p. 7).

[LZ10] Peng Li and Steve Zdancewic. "Arrows for secure information flow". In: *Theor. Comput. Sci.* 411.19 (2010), pp. 1974–1994. DOI: 10.1016/j.tcs.2010.01.025. URL: https://doi.org/10.1016/j.tcs.2010.01.025 (cit. on p. 7).

[Mar+01] Simon Marlow, Simon L. Peyton Jones, Andrew Moran, and John H. Reppy. "Asynchronous Exceptions in Haskell". In: *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*. Ed. by Michael Burke and Mary Lou Soffa. ACM, 2001, pp. 274–285. DOI: 10.1145/378795.378858. URL: https://doi.org/10.1145/378795.378858 (cit. on p. 10).

[RCH08] Alejandro Russo, Koen Claessen, and John Hughes. "A library for light-weight information-flow security in haskell". In: *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*. Ed. by Andy Gill. ACM, 2008, pp. 13–24. DOI: 10.1145/1411286.1411289. URL: https://doi.org/10.1145/1411286.1411289 (cit. on pp. 7, 9).

[Ric53] H. G. Rice. "Classes of recursively enumerable sets and their decision problems". In: *Trans. Amer. Math. Soc.* 74 (1953), pp. 358–366. ISSN: 0002-9947. DOI: 10.2307/1990888. URL: https://doi.org/10.2307/1990888 (cit. on p. 4).

[SI08] Naokata Shikuma and Atsushi Igarashi. "Proving Noninterference by a Fully Complete Translation to the Simply Typed Lambda-Calculus". In: *Log. Methods Comput. Sci.* 4.3 (2008). DOI: 10.2168/LMCS-4(3:10)2008. URL: https://doi.org/10.2168/LMCS-4(3:10)2008 (cit. on pp. 8, 10).

[SM03] Andrei Sabelfeld and Andrew C. Myers. “Language-based information-flow security”. In: *IEEE J. Sel. Areas Commun.* 21.1 (2003), pp. 5–19. DOI: 10.1109/JSAC.2002.806121. URL: https://doi.org/10.1109/JSAC.2002.806121 (cit. on p. 3).

[TV19] Carlos Tomé Cortiñas and Nachiappan Valliappan. “Simple Noninterference by Normalization”. In: *Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security, CCS 2019, London, United Kingdom, November 11-15, 2019*. Ed. by Piotr Mardziel and Niki Vazou. ACM, 2019, pp. 61–72. DOI: 10.1145/3338504.3357342. URL: https://doi.org/10.1145/3338504.3357342 (cit. on p. 9).

[TVR20] Carlos Tomé Cortiñas, Marco Vassena, and Alejandro Russo. “Securing Asynchronous Exceptions”. In: *33rd IEEE Computer Security Foundations Symposium, CSF 2020, Boston, MA, USA, June 22-26, 2020*. IEEE, 2020, pp. 214–229. DOI: 10.1109/CSF49147.2020.00023. URL: https://doi.org/10.1109/CSF49147.2020.00023 (cit. on p. 10).

[TZ04] Stephen Tse and Steve Zdancewic. “Translating dependency into parametricity”. In: *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, Snow Bird, UT, USA, September 19-21, 2004*. Ed. by Chris Okasaki and Kathleen Fisher. ACM, 2004, pp. 115–125. DOI: 10.1145/1016850.1016868. URL: https://doi.org/10.1145/1016850.1016868 (cit. on p. 7).

[Vas+18] Marco Vassena, Alejandro Russo, Pablo Buiras, and Lucas Waye. “MAC A verified static information-flow control library”. In: *Journal of Logical and Algebraic Methods in Programming* 95 (2018), pp. 148–180. ISSN: 2352-2208. DOI: https://doi.org/10.1016/j.jlamp.2017.12.003. URL: https://www.sciencedirect.com/science/article/pii/S235222081730069X (cit. on pp. 7, 9).

Papers

A

Simple Noninterference by Normalization

Carlos Tomé Cortiñas and Nachiappan Valliappan

Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security, CCS 2019, London, United Kingdom, November 11-15, 2019

Abstract Information-flow control (IFC) languages ensure programs preserve the confidentiality of sensitive data. *Noninterference*, the desired security property of such languages, states that public outputs of programs must not depend on sensitive inputs. In this paper, we show that noninterference can be proved using normalization. Unlike arbitrary terms, normal forms of programs are well-principled and obey useful syntactic properties—hence enabling a simpler proof of noninterference. Since our proof is syntax-directed, it offers an appealing alternative to traditional semantic based techniques to prove noninterference.

In particular, we prove noninterference for a static IFC calculus, based on Haskell's SecLib library, using normalization. Our proof follows by straightforward induction on the structure of normal forms. We implement normalization using *normalization by evaluation* and prove that the generated normal forms preserve semantics. Our results have been verified in the Agda proof assistant.

A.1. Introduction

Information-flow control (IFC) is a security mechanism which guarantees confidentiality of sensitive data by controlling how information is allowed to flow in a program. The guarantee that programs secured by an IFC system do not leak sensitive data is often proved using a property called *noninterference*. Noninterference ensures that an observer authorized to view the output of a program (pessimistically called the attacker) cannot infer any sensitive data handled by it. For example, suppose that the type $\mathsf{Int}_{\mathtt{H}}$ denotes a secret integer and $\mathsf{Bool}_{\mathtt{L}}$ denotes a public Boolean. Now consider a program f with the following type:

$$f : \mathsf{Int}_{\mathtt{H}} \to \mathsf{Bool}_{\mathtt{L}}$$

For this program, noninterference ensures that f outputs the same Boolean for any given integer.

To prove noninterference, we must show that the public output of a program is not affected by varying the secret input. This has been achieved using many techniques including *term erasure* based on dynamic operational semantics [LZ10; RCH08; Ste+11; VR16], denotational semantics [Aba+99; Kav19], and *parametricity* [TZ04; BA15; Alg18]. In this paper, we show that noninterference can also be proved by normalizing programs using the static or *residualising* semantics [Lin05] of the language.

If a program returns the same output for any given input, it must be the case that it does not depend on the input to compute the output. Thus proving noninterference for a program which receives a secret input and produces a public output, amounts to showing that the program behaves like a *constant* program. For example, proving noninterference for the program f consists of showing that it is equivalent to either $\lambda\, x\,.\, \mathit{true}$ or $\lambda\, x\,.\, \mathit{false}$; it is immediately apparent that these functions do not depend on the secret input x. But how can we prove this for *any* arbitrary definition of f?

The program f may have been defined as the simple function $\lambda\, x\,.\, (\mathit{not}\ \mathit{false})$ or perhaps the more complex function $\lambda\, x\,.\, ((\lambda\, y\,.\, \mathit{snd}\ (x\ ,\ y))\ \mathit{true})$. Observe, however, that both these programs can be normalized to the equivalent function $\lambda\ x\ .\ \mathit{true}$. In general, although terms in the language may be arbitrarily complex, their *normal forms* (such as $\lambda\, x\,.\, \mathit{true}$) are not. They are simpler, thus well-suited for showing noninterference.

The key idea in this paper is to normalize terms, and prove noninterference by simple structural induction on their normal forms. To illustrate this, we prove noninterference for a static IFC calculus, which we shall call λ_{SEC}, based on Haskell's SecLib library by Russo, Claessen, and Hughes We present the

typing rules and static semantics for λ_{SEC} by extending Moggi's *computational metalanguage* [Mog91] (Section A.2). We identify normal forms of λ_{SEC}, and establish syntactic properties about a normal form's dependency on its input (Section A.3). Using these properties, we show that the normal forms of program f are $\lambda\, x\,.\, \mathit{true}$ or $\lambda\, x\,.\, \mathit{false}$—as expected (Section A.4).

To prove noninterference for all terms using normal forms, we implement normalization for λ_{SEC} using *normalization by evaluation* (NbE) [BS91] and prove that it preserves the static semantics (Section A.5). Using normalization, we prove noninterference for program f and further generalize this proof to *all* terms in λ_{SEC} (Section A.6)—including, for example, a program which operates on both secret and public values such as $\mathsf{Bool}_{\mathsf{L}} \times \mathsf{Bool}_{\mathsf{H}} \to \mathsf{Bool}_{\mathsf{L}} \times \mathsf{Bool}_{\mathsf{H}}$. Finally, we conclude by discussing related work and future directions (Section A.7).

Unlike earlier proofs, our proof shows that noninterference is an inherent property of the normal forms of λ_{SEC}. Since the proof is primarily type and syntax-directed, it provides an appealing alternative to typical semantics based proof techniques. All the main theorems in this paper have been mechanized in the proof assistant Agda.[1]

A.2. The λ_{SEC} Calculus

In this section we present λ_{SEC}, a static IFC calculus that we shall use as the basis for our proof of noninterference. It models the pure and terminating fragment of the IFC library SecLib[2] for Haskell, and is an extension of the calculus developed by Russo, Claessen, and Hughes [RCH08] with sum types. SecLib is a lightweight implementation of static IFC which allows programmers to incorporate untrusted third-party code into their applications while ensuring that it does not leak sensitive data. Below, we recall the public interface (API) of SecLib:

$$
\begin{aligned}
&\mathsf{data}\ S\ (l :: \mathit{Lattice})\ a \\
&\mathit{return} :: a \to S\ l\ a \\
&(\gg\!=) \quad :: S\ l\ a \to (a \to S\ l\ b) \to S\ l\ b \\
&\mathit{up} \quad\quad :: l_{\mathsf{L}} \sqsubseteq l_{\mathsf{H}} \Rightarrow S\ l_{\mathsf{L}}\ a \to S\ l_{\mathsf{H}}\ a
\end{aligned}
$$

Similar to other static IFC libraries in Haskell such as LIO [Ste+11] or MAC [Vas+18], SecLib's security guarantees rely on exposing the API to the

[1] https://github.com/carlostome/ni-nbe

[2] `https://hackage.haskell.org/package/seclib`

programmer while hiding the underlying implementation. Programs written against the API and the *safe* parts of the language [Ter+12] are guaranteed to be *secure-by-construction*; the library enforces security statically through types. As an example, suppose that we have the two-point security lattice (see [Den76]) $\{\mathtt{L}, \mathtt{H}\}$ where the only disallowed flow is from secret ($\mathtt{H}$) to public ($\mathtt{L}$), denoted $\mathtt{H} \not\sqsubseteq \mathtt{L}$. The following program written using the SecLib API is well-typed and—intuitively—secure:

$$
\begin{aligned}
&\mathit{example} :: S\ \mathtt{L}\ \mathsf{Bool} \rightarrow S\ \mathtt{H}\ \mathsf{Bool} \\
&\mathit{example}\ p = \mathit{up}\ (p \gg\!= \lambda\, b \rightarrow \mathit{return}\ (\mathit{not}\ b))
\end{aligned}
$$

The function *example* negates the Bool that it receives as input and upgrades its security level from public to secret. On the other hand, had the program tried to downgrade the secret input to public—clearly violating the policy of the security lattice—the typechecker would have rejected the program as ill-typed.

The Calculus λ_{SEC} is a simply typed λ-calculus (STLC) with a base (uninterpreted) type, unit type, product and sum types, and a security monad type for every security level in a set of labels (denoted by Label). The set of labels may be a lattice, but our development only requires it to be a preorder on the relation $\sqsubseteq$. Throughout the rest of this paper, we use the labels $l_{\mathtt{L}}$ and $l_{\mathtt{H}}$ and refer to them as *public* and *secret*, although they represent levels in an arbitrary security lattice such that $l_{\mathtt{H}} \not\sqsubseteq l_{\mathtt{L}}$. Figure A.1 defines the syntax of terms, types and contexts of λ_{SEC}.

$$
\begin{aligned}
&\mathsf{Label}\ l\ ,\ l_{\mathtt{H}}\ ,\ l_{\mathtt{L}} \\
&\mathsf{Context}\ \Gamma\ \Delta\ \Sigma ::= \emptyset \mid \Gamma\ ,\ x : \tau \\
&\mathsf{Type}\ \tau\ \tau_1\ \tau_2 ::= \tau_1 \Rightarrow \tau_2 \mid \iota \mid () \\
&\qquad\qquad \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \\
&\qquad\qquad \mid \mathsf{S}\ l\ \tau \\
&\mathsf{Term}\ t\ s\ u \quad ::= x \mid \lambda\, x.t \mid t\ s \mid () \\
&\qquad\qquad \mid < t\ ,\ s > \mid \mathsf{fst}\ t \mid \mathsf{snd}\ t \\
&\qquad\qquad \mid \mathsf{left}\ t \mid \mathsf{right}\ t \\
&\qquad\qquad \mid \mathsf{case}\ t\ (\mathsf{left}\ x_1 \rightarrow s)\ (\mathsf{right}\ x_2 \rightarrow u) \\
&\qquad\qquad \mid \mathsf{return}\ t \mid \mathsf{let}\ x = t\ \mathsf{in}\ u \mid \mathsf{up}\ t
\end{aligned}
$$

Figure A.1.: The λ_{SEC} calculus

$\boxed{\Gamma \vdash t : \tau}$

$$\textsc{Return}\ \frac{\Gamma \vdash t : \tau}{\Gamma \vdash \mathsf{return}\ t : \mathsf{S}\ l\ \tau} \qquad \textsc{Up}\ \frac{\Gamma \vdash t : \mathsf{S}\ l_{\mathtt{L}}\ \tau \qquad l_{\mathtt{L}} \sqsubseteq l_{\mathtt{H}}}{\Gamma \vdash \mathsf{up}\ t : \mathsf{S}\ l_{\mathtt{H}}\ \tau}$$

$$\textsc{Let}\ \frac{\Gamma \vdash t : \mathsf{S}\ l\ \tau_1 \qquad \Gamma\,,\, x : \tau_1 \vdash s : \mathsf{S}\ l\ \tau_2}{\Gamma \vdash \mathsf{let}\ x = t\ \mathsf{in}\ s : \mathsf{S}\ l\ \tau_2}$$

Figure A.2.: Type system of λ_{SEC} (excerpts)

In addition to the standard introduction and elimination constructs for unit, products and sums in STLC, λ_{SEC} uses the constructs return, let and up for the security monad $\mathsf{S}\ l\ \tau$, which mirrors S from SecLib. Note that our presentation favours let, as in Moggi [Mog89], over the Haskell bind ($\gg\!=$), although both presentations are equivalent—i.e. $t \gg\!= \lambda\, x\,.\, u$ can be encoded as $\mathsf{let}\ x = t\ \mathsf{in}\ u$.

The typing rules for return and let, shown in Figure A.2, ensure that computations over labelled values in the security monad $\mathsf{S}\ l\ \tau$ do not leak sensitive data. The construct return allows the programmer to tag a value of type τ with security label l; and bind enforces that sequences of computations over labelled values stay at the same security level.

Further, the calculus models the *up* combinator in SecLib as the construct up. Its purpose is to relabel computations to higher security levels. The rule Up, shown in Figure A.2, statically enforces that information can only flow from $l_{\mathtt{L}}$ to $l_{\mathtt{H}}$ in agreement with the security policy $l_{\mathtt{L}} \sqsubseteq l_{\mathtt{H}}$. The rest of the typing rules for λ_{SEC} are standard [Pie02], and thus omitted here. For a full account we refer the reader to our Agda formalization.

For completeness, the function *example* from earlier can be encoded in the λ_{SEC} calculus as follows:[3]

$$\mathit{example} \;=\; \lambda\, s\,.\,\mathsf{up}\ (\mathsf{let}\ b = s\ \mathsf{in}\ \mathsf{return}\ (\mathit{not}\ b))$$

Static Semantics The static semantics of λ_{SEC} is defined as a set of equations relating terms of the same type typed under the same environment. The equations characterize pairs of λ_{SEC} terms that are equivalent based on β-reduction, η-expansion and other monadic operations. We present the equations for return

[3] In λ_{SEC}, the type Bool is encoded as $() + ()$ with $\mathit{false} = \mathsf{left}\ ()$ and $\mathit{true} = \mathsf{right}\ ()$.

and let constructs of the monadic type S (à la Moggi [Mog91]) in Figure A.3, and further extend this with equations for the up primitive in Figure A.4. The remaining equations—including β and η rules for other types, and permutation rules for commuting case conversions—are fairly standard [Lin05; AS19], and can be found in the Agda formalization. As customary, we use the notation $t_1\ [x/t_2]$ for capture-avoiding substitution of the term t_2 for variable x in term t_1.

$\boxed{\Gamma \vdash t_1 \approx t_2 : \tau}$

$$\beta\text{-S}\ \frac{\Gamma \vdash t_1 : \tau \qquad \Gamma\ ,\ x : \tau \vdash t_2 : \mathsf{S}\ l\ \tau}{\Gamma \vdash \mathsf{let}\ x = (\mathsf{return}\ t_1)\ \mathsf{in}\ t_2 \approx t_2\ [x/t_1] : \mathsf{S}\ l\ \tau}$$

$$\eta\text{-S}\ \frac{\Gamma \vdash t : \mathsf{S}\ l\ \tau}{\Gamma \vdash t \approx \mathsf{let}\ x = t\ \mathsf{in}\ (\mathsf{return}\ x) : \mathsf{S}\ l\ \tau}$$

$$\gamma\text{-S}\ \frac{\Gamma \vdash t_1 : \mathsf{S}\ l\ \tau_1 \qquad \Gamma\ ,\ x : \tau_1 \vdash t_2 : \mathsf{S}\ l\ \tau_2 \qquad \Gamma\ ,\ x : \tau_1\ ,\ y : \tau_2 \vdash t_3 : \mathsf{S}\ l\ \tau_3}{\Gamma \vdash \mathsf{let}\ x = (\mathsf{let}\ y = t_1\ \mathsf{in}\ t_2)\ \mathsf{in}\ t_3 \approx \mathsf{let}\ y = t_1\ \mathsf{in}\ (\mathsf{let}\ x = t_2\ \mathsf{in}\ t_3) : \mathsf{S}\ l\ \tau_3}$$

Figure A.3.: Static semantics of λ_{SEC} (return and let)

The up primitive induces equations regarding its interaction with itself and other constructs in the security monad. In Figure A.4, we make the auxiliary condition of up and the label of return explicit using subscripts for better clarity. These equations can be understood as follows:

- Rule δ_1-S. applying up over let is equivalent to distributing it over the subterms of let.
- Rule δ_2-S. applying up on an term labelled as return t is equivalent to relabelling t with the final label.
- Rule δ_{trans}-S. applying up twice is equivalent to applying it once using the transitivity of the relation $\sqsubseteq$.
- Rule δ_{refl}-S. applying up using the reflexive relation $l \sqsubseteq l$ is equivalent to not applying it.

$\boxed{\Gamma \vdash t_1 \approx t_2 : \tau}$

$$\frac{\Gamma \vdash t : \mathsf{S}\ l_{\mathtt{L}}\ \tau_1 \qquad \Gamma\,,\, x : \tau_1 \vdash u : \mathsf{S}\ l_{\mathtt{L}}\ \tau_2 \qquad p : l_{\mathtt{L}} \sqsubseteq l_{\mathtt{H}}}{\Gamma \vdash \mathsf{up}_p\ (\mathsf{let}\ x = t\ \mathsf{in}\ u) \approx \mathsf{let}\ x = (\mathsf{up}_p\ t)\ \mathsf{in}\ (\mathsf{up}_p\ u) : \mathsf{S}\ l_{\mathtt{H}}\ \tau}\ \delta_1\text{-}\mathsf{S}$$

$$\frac{\Gamma \vdash t : \tau \qquad p : l_{\mathtt{L}} \sqsubseteq l_{\mathtt{H}}}{\Gamma \vdash \mathsf{up}_p\ (\mathsf{return}_{\mathtt{L}}\ t) \approx \mathsf{return}_{\mathtt{H}}\ t : \mathsf{S}\ l_{\mathtt{H}}\ \tau}\ \delta_2\text{-}\mathsf{S}$$

$$\frac{\Gamma \vdash t : \mathsf{S}\ l_{\mathtt{L}}\ \tau \qquad p : l_{\mathtt{L}} \sqsubseteq l_{\mathtt{M}} \qquad q : l_{\mathtt{M}} \sqsubseteq l_{\mathtt{H}} \qquad r = \text{trans-}\sqsubseteq\ p\ q}{\Gamma \vdash \mathsf{up}_q\ (\mathsf{up}_p\ t) \approx \mathsf{up}_r\ t : \mathsf{S}\ l_{\mathtt{H}}\ \tau}\ \delta_{\text{TRANS}}\text{-}\mathsf{S}$$

$$\frac{\Gamma \vdash t : \mathsf{S}\ l\ \tau \qquad p : l \sqsubseteq l}{\Gamma \vdash \mathsf{up}_p\ t \approx t : \mathsf{S}\ l\ \tau}\ \delta_{\text{REFL}}\text{-}\mathsf{S}$$

Figure A.4.: Static semantics of λ_{SEC} (up)

A.3. Normal Forms of λ_{SEC}

As discussed in Section A.1, our proof of noninterference utilizes syntactic properties of normal forms, and hence relies on normalizing terms in the language. Normal forms are a restricted subset of terms in the λ_{SEC} calculus which intuitively corresponds to terms that cannot be normalized further. The syntax of normal forms is defined using two well-typed interdependent syntactic categories: *neutral* forms as $\Gamma \vdash_{\text{ne}} t : \tau$ (Figure A.5) and normal forms as $\Gamma \vdash_{\text{nf}} t : \tau$ (Figure A.6). Neutral forms are a special case of normal forms which depend entirely on the typing context (e.g. a variable).

Since the definition of neutral and normal forms are merely a syntactic restriction over terms, they can be embedded back into terms of λ_{SEC} using a *quotation* function $\ulcorner n \urcorner$. This embedding can be implemented for neutrals and normal forms by simply mapping them to their term counterparts.

Neutral Forms The neutral forms are terms which are characterized by a property called *neutrality*, which is stated as follows:

Property A.3.1 (Neutrality)**.** For a given neutral form of type $\Gamma \vdash_{\text{ne}} \tau$,

$\boxed{\Gamma \vdash_{\text{ne}} t : \tau}$

$$\textsc{Var}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash_{\text{ne}} x : \tau} \qquad \textsc{App}\ \frac{\Gamma \vdash_{\text{ne}} t : \tau_1 \Rightarrow \tau_2 \qquad \Gamma \vdash_{\text{nf}} s : \tau_1}{\Gamma \vdash_{\text{ne}} t\ s : \tau_2} \qquad \textsc{Fst}\ \frac{\Gamma \vdash_{\text{ne}} t : \tau_1 \times \tau_2}{\Gamma \vdash_{\text{ne}} \mathsf{fst}\ t : \tau_1}$$

$$\textsc{Snd}\ \frac{\Gamma \vdash_{\text{ne}} t : \tau_1 \times \tau_2}{\Gamma \vdash_{\text{ne}} \mathsf{snd}\ t : \tau_2}$$

Figure A.5.: Neutral forms

neutrality states that the type τ must occur as a *subformula* of a type in the context Γ.

For instance, given a neutral form $\Gamma \vdash_{\text{ne}} n : \mathsf{Bool}$, neutrality states that the type Bool must occur as a subformula of some type in the typing context Γ. An example of such a context is $\Gamma = [x : () \Rightarrow \mathsf{Bool}\ ,\ y : \mathsf{S}\ l_{\mathtt{H}}\ \iota]$. The notion of a subformula, originally defined for logical propositional formulas in proof theory [TS00], can also be defined for types as follows:

Definition A.3.1 (Subformula)**.** For some types τ, τ_1 and τ_2; a subformula of a type is defined as:

- τ is a subformula of τ
- τ is a subformula of $\tau_1 \otimes \tau_2$ if τ is a subformula of τ_1 or τ is a subformula of τ_2, where $\otimes$ denotes the binary type operators $\times$, $+$ and $\Rightarrow$.

The type Bool occurs as a subformula in the typing context $[() \Rightarrow \mathsf{Bool}\ ,\ \mathsf{S}\ l_{\mathtt{H}}\ \iota]$ since the type Bool is a subformula of the type $() \Rightarrow \mathsf{Bool}$. Note, however, that the type ι does not occur as a subformula in this context since ι is not a subformula of the type $\mathsf{S}\ l_{\mathtt{H}}\ \iota$ by the above definition.

Normal Forms Intuitively, normal forms of type $\Gamma \vdash_{\text{nf}} \tau$ are characterized as terms of type $\Gamma \vdash \tau$ that cannot be *reduced* further using the static semantics. Precisely, a normal form is a term obtained by systematically applying the equations defined by the relation $\approx$ in a specific order to a given term. We leave the exact order of applying the equations unspecified since we only require that there *exists* a normal form for every term—we prove this later in Section A.5.

$\boxed{\Gamma \vdash_{\mathrm{nf}} t : \tau}$

$$\textsc{Unit}\ \frac{}{\Gamma \vdash_{\mathrm{nf}} () : ()} \qquad \textsc{Lam}\ \frac{\Gamma\,,\, x : \tau_1 \vdash_{\mathrm{nf}} t : \tau_2}{\Gamma \vdash_{\mathrm{nf}} \lambda\, x\,.\, t : \tau_1 \Rightarrow \tau_2} \qquad \textsc{Base}\ \frac{\Gamma \vdash_{\mathrm{ne}} t : \iota}{\Gamma \vdash_{\mathrm{nf}} t : \iota}$$

$$\textsc{Ret}\ \frac{\Gamma \vdash_{\mathrm{nf}} t : \tau}{\Gamma \vdash_{\mathrm{nf}} \mathsf{return}\ t : \mathsf{S}\ l\ \tau}$$

$$\textsc{LetUp}\ \frac{l_{\mathtt{L}} \sqsubseteq l_{\mathtt{H}} \qquad \Gamma \vdash_{\mathrm{ne}} t : \mathsf{S}\ l_{\mathtt{L}}\ \tau_1 \qquad \Gamma\,,\, x : \tau_1 \vdash_{\mathrm{nf}} s : \mathsf{S}\ l_{\mathtt{H}}\ \tau_2}{\Gamma \vdash_{\mathrm{nf}} \mathsf{let}{\uparrow}\ x = t\ \mathsf{in}\ s : \mathsf{S}\ l_{\mathtt{H}}\ \tau_2}$$

$$\textsc{Left}\ \frac{\Gamma \vdash_{\mathrm{nf}} t : \tau_1}{\Gamma \vdash_{\mathrm{nf}} \mathsf{left}\ t : \tau_1 + \tau_2} \qquad \textsc{Right}\ \frac{\Gamma \vdash_{\mathrm{nf}} t : \tau_2}{\Gamma \vdash_{\mathrm{nf}} \mathsf{right}\ t : \tau_1 + \tau_2}$$

$$\textsc{Case}\ \frac{\Gamma \vdash_{\mathrm{ne}} t : \tau_1 + \tau_2 \qquad \Gamma\,,\, x_1 : \tau_1 \vdash_{\mathrm{nf}} t_1 : \tau \qquad \Gamma\,,\, x_2 : \tau_2 \vdash_{\mathrm{nf}} t_2 : \tau}{\Gamma \vdash_{\mathrm{nf}} \mathsf{case}\ t\ (\mathsf{left}\ x_1 \to t_1)\ (\mathsf{right}\ x_2 \to t_2) : \tau}$$

Figure A.6.: Normal forms

The normal forms in Figure A.6 extend the β-short η-long forms in simply-typed λ-calculus (STLC) [BCF04; AS19] with return and let↑. Note that, unlike neutrals, arbitrary normal forms do not obey neutrality since they may also construct values which do not occur in the context. For example, the normal form left () (which denotes the value *false*) of type $\emptyset \vdash_{\text{nf}} \mathsf{Bool}$ constructs a value of the type Bool in the empty context $\emptyset$.

The reader may have noticed that the let↑ construct in normal forms does not directly resemble a term, and hence it is not immediately obvious how it should be quoted. Normal forms constructed by let↑ can be quoted by first applying up to the quotation of the neutral and then using let. The reason let↑ represents both let and up in the normal forms is to prevent reducibility of the normal forms. Had we added up separately to normal forms, then this may trigger further reductions. For example, the term up (return ()) can be reduced further to the term return (). Disallowing up-terms directly in normal forms removes the possibility of this reduction in normal forms. Similarly, adding up to neutral forms is also equally worse since it breaks neutrality.

The syntactic characterization of neutral and normal forms provides us with useful properties in the proof of noninterference. For example, there cannot exist a neutral of type $\emptyset \vdash_{\text{ne}} \tau$ for any type τ. By neutrality, if such a neutral form exists, then τ must be a subformula of the empty context $\emptyset$, but this is impossible! Similarly, the η-long form of normal forms guarantee that a normal form of a function type must begin with either a λ or case—hence reducing the number of possible cases in our proof. In the next section, we utilize these properties to show that the program f (from earlier) behaves as a constant.

A.4. Normal Forms and Noninterference

The program $f : \mathsf{Int}_{\mathsf{H}} \to \mathsf{Bool}_{\mathsf{L}}$ from Section A.1 can be generalized in λ_{SEC} as a term[4] $\emptyset \vdash f : \mathsf{S}\ l_{\mathsf{H}}\ \tau \Rightarrow \mathsf{S}\ l_{\mathsf{L}}\ \mathsf{Bool}$ marking the secret input and public output through the security monad. Noninterference for this term—which Russo, Claessen, and Hughes [RCH08] refer to as a "noninterference-like" property for λ_{SEC}—states that given two levels l_{L} (*public*) and l_{H} (*secret*) such that the flow of information from secret to public is disallowed as $l_{\mathsf{H}} \not\sqsubseteq l_{\mathsf{L}}$; for any two possibly different secrets s_1 and s_2, applying f to s_1 is equivalent to applying it to s_2. In other words, it states that varying the secret input must *not interfere* with the public output.

As explained before, for $\emptyset \vdash f : \mathsf{S}\ l_{\mathsf{H}}\ \tau \Rightarrow \mathsf{S}\ l_{\mathsf{L}}\ \mathsf{Bool}$ to satisfy noninterference,

[4] λ_{SEC} does not have polymorphic types, in this case τ represents an arbitrary but concrete type, for instance unit ().

it must be equivalent to the constant function whose body is return *true* or return *false* independent of the input. For an arbitrary program f it is not possible to conclude so just from case analysis—as programs may be fairly complex—however, for normal forms of the same type it is possible. In the Lemma below, we materialize this intuition:

Lemma A.4.1 (Normal forms of f are constant)**.** *For any normal form* $\emptyset \vdash_{nf} f : \mathsf{S}\ l_{\mathsf{H}}\ \tau \Rightarrow \mathsf{S}\ l_{\mathsf{L}}\ \mathsf{Bool}$*, either* $f \equiv \lambda\, x\,.\,(\mathsf{return}\ \mathit{true})$ *or* $f \equiv \lambda\, x\,.\,(\mathsf{return}\ \mathit{false})$

Note that the equality relation $\equiv$ denotes syntactic (or propositional) equality, which means that the normal forms on both sides must be syntactically identical. The proof follows by direct case analysis on the normal forms of type $\emptyset \vdash_{\mathrm{nf}} f : \mathsf{S}\ l_{\mathsf{H}}\ \tau \Rightarrow \mathsf{S}\ l_{\mathsf{L}}\ \mathsf{Bool}$:

Proof of Lemma A.4.1. Upon closer inspection of the normal forms of λ_{SEC} (Figure A.6), the reader may notice that at function type $\emptyset \vdash_{\mathrm{nf}} \mathsf{S}\ l_{\mathsf{H}}\ \tau \Rightarrow \mathsf{S}\ l_{\mathsf{L}}\ \mathsf{Bool}$ there exists only two possibilities: a case or a λ construct. The former, can be easily dismissed by neutrality because it requires the scrutinee—a neutral form of sum type $\tau_1 + \tau_2$—to appear in the empty context. In the latter case, the λ construct extends typing context of the body with the type of the argument, and thus refines the normal form to have the shape $\lambda\, x\,.\,\_$ where $\emptyset\ ,\ x : \mathsf{S}\ l_{\mathsf{H}}\ \tau \vdash_{\mathrm{nf}} \_ : \mathsf{S}\ l_{\mathsf{L}}\ \mathsf{Bool}$.

Considering the normal forms of type $\emptyset\ ,\ x : \mathsf{S}\ l_{\mathsf{H}}\ \tau \vdash_{\mathrm{nf}} \mathsf{S}\ l_{\mathsf{L}}\ \mathsf{Bool}$, we realize that there are only three possible candidates: the case construct again, the monadic return or let. As before, case is discharged because it requires the scrutinee of sum type to occur in the context $\emptyset\ ,\ x : \mathsf{S}\ l_{\mathsf{H}}\ \tau$. Analogously, the monadic let with a neutral term of type $\mathsf{S}\ l_{\mathsf{L}}\ \tau$, expects this type to occur in the same context—but it does not, since $\mathsf{S}\ l_{\mathsf{L}}\ \tau$ is not a subformula of $\mathsf{S}\ l_{\mathsf{H}}\ \tau$. The remaining case, return, can be further refined, where the only possibilities leave us with $\lambda\, x\,.\,(\mathsf{return}\ \mathit{true})$ or $\lambda\, x\,.\,(\mathsf{return}\ \mathit{false})$. □

In order to show that noninterference holds for arbitrary programs of type $\emptyset \vdash f : \mathsf{S}\ l_{\mathsf{H}}\ \tau \Rightarrow \mathsf{S}\ l_{\mathsf{L}}\ \mathsf{Bool}$ using this lemma, we must link the behaviour of a program with that of its normal form. In the next section we develop the necessary normalization machinery and later complete the proof of noninterference in Section A.6.

A.5. From λ_{SEC} to Normal Forms

The goal of this section is to implement a normalization algorithm that bridges the gap between terms and their normal forms. For this purpose, we employ

Normalization by Evaluation (NbE).

Normalization based on rewriting techniques [Pie02] perform syntactic transformations of a term to produce a normal form. NbE, on the other hand, normalizes a term by evaluating it in a host language, and then extracting a normal form from the (semantic) value in the host language. Evaluation of a term is implemented by an interpreter function eval, and the extraction of normal forms, called *reification*, is implemented by an inverse function reify. Normalization is implemented as a function from terms to normal forms by composing these functions:

$$\begin{aligned}&\mathsf{norm} : (\Gamma \vdash \tau) \;\to\; (\Gamma \vdash_{\mathrm{nf}} \tau)\\ &\mathsf{norm}\ t \;=\; \mathsf{reify}\ (\mathsf{eval}\ t)\end{aligned}$$

The function eval and reify have the following types in the host language:

$$\begin{aligned}&\mathsf{eval} : (\Gamma \vdash \tau) \;\to\; (\llbracket\, \Gamma \,\rrbracket \;\to\; \llbracket\, \tau \,\rrbracket)\\ &\mathsf{reify} : (\llbracket\, \Gamma \,\rrbracket \;\to\; \llbracket\, \tau \,\rrbracket) \;\to\; (\Gamma \vdash_{\mathrm{nf}} \tau)\end{aligned}$$

In these types, the function $\llbracket\, \_ \,\rrbracket$ interprets types and contexts in λ_{SEC} as types in the host language. That is, the type $\llbracket\, \tau \,\rrbracket$ denotes the interpretation of the (λ_{SEC}) type τ in the host language, and similarly for $\llbracket\, \Gamma \,\rrbracket$. On the other hand, the function $\llbracket\, \Gamma \,\rrbracket \;\to\; \llbracket\, \tau \,\rrbracket$—a function between the interpretations in the host language—denotes the interpretation of the term $\Gamma \vdash \tau$.

The advantages of using NbE over a rewrite system are two-fold: first, it serves as an actual implementation of the normalization algorithm; second, and most importantly, when implemented in a proof system like Agda, it makes normalization amenable to formal reasoning. For example, since Agda ensures that all functions are total, we are assured that a normal form must exist for every term in λ_{SEC}. Similarly, we also get a proof that normalization terminates for free since Agda ensures that all functions are terminating.

We implement the functions eval and reify for terms in λ_{SEC} using Agda as the host language. Note that, however, the implementation of our algorithm—and NbE in general—is not specific to Agda. It may also be implemented in other programming languages such as Haskell [DRR01] or Standard ML [BCF04].

In the remainder of this section, we will denote the typing derivations $\Gamma \vdash_{\mathrm{nf}} \tau$ and $\Gamma \vdash_{\mathrm{ne}} \tau$ as $\mathsf{Nf}\ \tau$ and $\mathsf{Ne}\ \tau$ respectively. We leave the context Γ implicit to avoid the clutter caused by contexts and their *weakenings* [AHS95; McB18]. Similarly, we will represent variables of type $\tau \in \Gamma$ as $\mathsf{Var}\ \tau$, leaving Γ implicit. Although we use de Bruijn indices in the actual implementation of variables, we will continue to use named variables here to ease presentation. We encourage the curious reader to see the formalization in Agda for further details.

A.5.1. NbE for Simple Types

To begin with, we implement evaluation and reification for the types ι, $()$, $\times$ and $\Rightarrow$. The implementation for sums is more technical, and hence deferred to Appendix I. Note that the implementation of NbE for simple types is entirely standard [AHS95; BCF04]. Their interpretation as Agda types is defined as follows:

$$\begin{array}{lcl} [\![\ \iota\ ]\!] & = & \mathsf{Nf}\ \iota \\ [\![\ ()\ ]\!] & = & \top \\ [\![\ \tau_1 \times \tau_2\ ]\!] & = & [\![\ \tau_1\ ]\!] \times [\![\ \tau_2\ ]\!] \\ [\![\ \tau_1 \Rightarrow \tau_2\ ]\!] & = & [\![\ \tau_1\ ]\!] \to [\![\ \tau_2\ ]\!] \end{array}$$

The types $()$, $\times$ and $\Rightarrow$ are simply interpreted as their counterparts in Agda. For the base type ι, however, we cannot provide a counterpart in Agda since we do not know anything about this type. Instead, since the type ι is not constructed or eliminated by any specific construct in λ_{SEC}, we simply require a normal form as an evidence for producing a value of type ι—and thus interpret it as $\mathsf{Nf}\ \iota$.

Typing contexts map variables to types, and hence their interpretation is an execution environment (or equivalently, a semantic substitution) defined like-wise:

$$\begin{array}{lcl} [\![\ \emptyset\ ]\!] & = & \emptyset \\ [\![\ \Gamma\ ,\ x : \tau_1\ ]\!] & = & [\![\ \Gamma\ ]\!]\ [\ \mathsf{Var}\ \tau_1 \mapsto [\![\ \tau_1\ ]\!]\ ] \end{array}$$

For example, a value γ which inhabits the interpretation $[\![\ \Gamma\ ]\!]$ denotes the execution environment for evaluating a term typed in the context Γ.

Given these definitions, evaluation is implemented as a straightforward interpreter function:

$$\begin{array}{lcl} \mathsf{eval}\ x & \gamma & = \mathsf{lookup}\ x\ \gamma \\ \mathsf{eval}\ () & \gamma & = \mathsf{tt} \\ \mathsf{eval}\ (\mathsf{fst}\ t) & \gamma & = \pi_1\ (\mathsf{eval}\ t\ \gamma) \\ \mathsf{eval}\ (\mathsf{snd}\ t) & \gamma & = \pi_2\ (\mathsf{eval}\ t\ \gamma) \\ \mathsf{eval}\ (< t_1\ ,\ t_2\ >) & \gamma & = (\mathsf{eval}\ t_1\ \gamma\ ,\ \mathsf{eval}\ t_2\ \gamma) \\ \mathsf{eval}\ (\lambda\ x\,.\,t) & \gamma & = \lambda\ v\ \to\ \mathsf{eval}\ t\ (\gamma\ [x \mapsto v]) \\ \mathsf{eval}\ (t\ s) & \gamma & = (\mathsf{eval}\ t\ \gamma)\ (\mathsf{eval}\ s\ \gamma) \end{array}$$

Note that γ is an execution environment for the term's context; lookup, π_1 and π_2 are Agda functions; and tt is the constructor of the unit type $\top$. For the case of $\lambda\ x\,.\,t$, evaluation is expected to return an equivalent semantic function.

We compute the body of this function by evaluating the body term t using the substitution γ extended with a mapping which assigns the value v to the variable x—denoted $\gamma\,[x \mapsto v]$.

Reification, on the other hand, is implemented using two helper functions reflect and reifyVal. The function reflect converts neutral forms to semantic values, while the dual function reifyVal converts semantic values to normal forms. These functions are implemented as follows:

```
reifyVal : ⟦ τ ⟧ → Nf τ
reifyVal {ι} n          = n
reifyVal {()} tt        = ()
reifyVal {τ₁ × τ₂} p    =
  < reifyVal {τ₁} (π₁ p) , reifyVal {τ₂} (π₁ p) >
reifyVal {τ₁ ⇒ τ₂} f    =
  λ x . reifyVal {τ₂} (f (reflect {τ₁} x))  |  fresh x


reflect : Ne τ → ⟦ τ ⟧
reflect {ι} n           = n
reflect {()} n          = tt
reflect {τ₁ × τ₂} n     =
  (reflect {τ₁} (fst n) , reflect {τ₂} (snd n))
reflect {τ₁ ⇒ τ₂} n     =
  λ v → reflect {τ₂} (n (reifyVal {τ₁} v))
```

Note that the argument inside the braces { } denotes an implicit parameter, which is the type of the corresponding neutral/value argument of reflect/reifyVal here.

Reflection is implemented by performing a type-directed translation of neutral forms to semantic values by induction on types. The interpretation of types, defined earlier, guides our implementation. For example, reflection of a neutral with a function type must produce a function value since the type $\Rightarrow$ is interpreted as an Agda function. For this purpose, we are given the argument value in the semantics and it remains to construct a function body of the appropriate type. We produce the body of this function by recursively reflecting a neutral application of the function and (the reification of) the argument value. The function reifyVal is also implemented in a similar fashion by induction on types.

To implement reification, recollect that the argument to reify is a function that results from partially applying the eval function with a term. If the term has type $\Gamma \vdash \tau$, then the argument, say f, must have the type $[\![\,\Gamma\,]\!] \to [\![\,\tau\,]\!]$.

Thus, to apply f, we need an execution environment of the type $[\![\ \Gamma\ ]\!]$. This environment can be generated by simply reflecting the variables in the context as follows:

$$\begin{aligned}
&\mathsf{genEnv} : (\Gamma : \mathsf{Ctx}) \ \to\ [\![\ \Gamma\ ]\!] \\
&\mathsf{genEnv}\ \emptyset \qquad\quad\ \ = \ \emptyset \\
&\mathsf{genEnv}\ (\Gamma\ ,\ x : \tau) \ = \ \mathsf{genEnv}\ \Gamma\ [\ x \mapsto \mathsf{reflect}\ x\ ]
\end{aligned}$$

Finally, we can now implement reify as follows:

$$\mathsf{reify}\ \{\Gamma\}\ f \ = \ \mathsf{let}\ \gamma \ = \ \mathsf{genEnv}\ \Gamma\ \mathsf{in}\ \mathsf{reifyVal}\ (f\ \gamma)$$

We generate an environment γ to apply the semantic function f, and then convert the resulting semantic value to a normal form by applying reifyVal.

A.5.2. NbE for the Security Monad

To interpret a type $\mathsf{S}\ l\ \tau$, we need a semantic counterpart in the host language which is also a monad. Suppose that we define such a monad as an inductive data type T parameterized by a label l and some type a (which would be $[\![\ \tau\ ]\!]$ in this case). Evidently this monad must allow the implementation of the semantic counterparts of the terms return, let and up in λ_{SEC} as follows:

$$\begin{aligned}
&\mathsf{return} : a \ \to\ T\ l\ a \\
&\mathsf{bind} \quad : T\ l\ a \ \to\ (a \ \to\ T\ l\ b) \ \to\ T\ l\ b \\
&\mathsf{up} \quad\ : (l_{\mathtt{L}} \ \sqsubseteq\ l_{\mathtt{H}}) \ \to\ T\ l_{\mathtt{L}}\ a \ \to\ T\ l_{\mathtt{H}}\ a
\end{aligned}$$

To satisfy this specification, we define the data type T in Agda with the following constructors:

$$\textsc{Return}\ \frac{x : a}{\mathsf{return}\ x : T\ l\ a} \qquad \textsc{BindN}\ \frac{p : l_{\mathtt{L}} \ \sqsubseteq\ l_{\mathtt{H}} \qquad n : \mathsf{Ne}\ \mathsf{S}\ l_{\mathtt{L}}\ \tau \qquad f : \mathsf{Var}\ \tau \ \to\ T\ l_{\mathtt{H}}\ a}{\mathsf{bindNe}\ p\ n\ f : T\ l_{\mathtt{H}}\ a}$$

The constructor return returns a semantic value in the monad, while bindNe registers a binding of a neutral to monadic value. These constructors are the semantic equivalent of return and let↑ in the normal forms, respectively. The constructor bindNe is more general than the required function bind in order to allow the definition of up, which is defined by induction as follows:

$$\begin{aligned}
&\mathsf{up}\ p\ (\mathsf{return}\ v) \ = \\
&\quad \mathsf{return}\ v
\end{aligned}$$

```
up p (bindNe q n f) =
  bindNe (trans q p) n (λ x → up p (f x))
```

To understand this implementation, suppose that $p : l_M \sqsubseteq l_H$ for some labels l_M and l_H. A monadic value of type $T\ l_M\ a$ which is constructed by a return can be simply relabelled to $T\ l_H\ a$ since return can be used to construct a monadic value on any label. For the case of bindNe $q\ n\ f$, we have that $q : l_L \sqsubseteq l_M$ and n : Ne S $l_L\ \tau_1$, hence $l_L \sqsubseteq l_H$ by transitivity, and we may simply use bindNe to register n and recursively apply up on the continuation f to produce the desired result of type $T\ l_H\ a$.

Using the type T in the host language, we may now interpret the monad in λ_{SEC} as follows:

$$[\![\ \mathsf{S}\ l\ \tau\ ]\!] = T\ l\ [\![\ \tau\ ]\!]$$

Having mirrored the monadic primitives in λ_{SEC} using semantic counterparts, evaluation is rather simple:

```
eval (return t) γ = return (eval t γ)
eval (up p t)   γ = up p (eval t γ)
eval (let x = t in s) γ =
  bind (eval t γ) (λ v → eval s (γ [x ↦ v]))
```

For implementing reflection, we can use bindNe to register a neutral binding and recursively reflect the given variable:

```
reflect {S l τ} n =
  bindNe refl n (λ x → return (reflect {τ} x))
```

Since we do not need to increase the sensitivity of the neutral to bind it here, we simply provide the "reflexive flow" refl : $l \sqsubseteq l$.

The function reifyVal, on the other hand, is rather straightforward since the constructors of T are essentially semantic counterparts of the normal forms, and can hence be translated to it:

```
reifyVal {S l τ} (return v) =
  return (reifyVal {τ} v)
reifyVal {S l τ} (bindNe {p} n f) =
  let↑ {p} x = n in reifyVal {τ} (f x)
```

A.5.3. Preservation of Semantics

To prove that normalization preserves static semantics of λ_{SEC}, we must show that the normal form of term is equivalent to the term. Since normal forms and terms belong to different syntactic categories, we must first quote normal forms to state this relationship using the term equivalence relation $\approx$. This property, called *consistency* of normal forms, is stated as follows:

Theorem A.5.1 (Consistency of normal forms)**.** *For any term* $\Gamma \vdash t : \tau$ *we have that* $\Gamma \vdash t \approx \ulcorner \mathit{norm}\ t \urcorner : \tau$

An attempt to prove consistency by induction on the terms or types fails quickly since the induction principle alone is not strong enough to prove this theorem. To solve this issue we must establish a notion of equivalence between a term and its interpretation using *logical relations* [Plo80]. Using these relations, we can prove that evaluation is consistent by showing that it is *related* to applying a substitution in the syntax. Following this, we can also prove the consistency of reification by showing that reifying a value related to a term, yields a normal form which is equivalent to the term when quoted. The consistency of evaluation and reification yields the proof of consistency for normal forms.

This proof follows the style of the consistency proof of NbE for STLC using Kripke logical relations by Coquand [Coq93]. As is the case for sums, NbE for the security monad uses an inductively defined data type to implement the semantic monad. Hence, we are able to leverage the proof techniques used to prove the consistency of NbE for sums [VR19] to prove the same for the security monad. We skip the details of the proof here, but encourage the curious reader to see the Agda mechanization of this theorem.

A.6. Noninterference for λ_{SEC}

After developing the necessary machinery to normalize terms in the calculus, we are ready to state and prove noninterference for λ_{SEC}. First, we complete the proof of noninterference for the program f from Section A.4.

A.6.1. Special Case of Noninterference

Theorem A.6.1 (Noninterference for f)**.** *Given security levels* $l_{\mathtt{L}}$ *and* $l_{\mathtt{H}}$ *such that* $l_{\mathtt{H}} \not\sqsubseteq l_{\mathtt{L}}$ *and a function* $\emptyset \vdash f : S\ l_{\mathtt{H}}\ \tau \Rightarrow S\ l_{\mathtt{L}}\ \mathit{Bool}$ *then* $\forall\ s_1\ s_2 : S\ l_{\mathtt{H}}\ \tau.\ f\ s_1 \approx f\ s_2$

The proof of Theorem A.6.1 relies upon two key ingredients: Lemma A.4.1 (Section A.4), which characterizes the shape of the normal forms of f; and consistency of normal forms, Theorem A.5.1 (Section A.5.3), which links the semantics of f with that of its normal forms.

Proof of Theorem A.6.1. To show that a function $\emptyset \vdash f : \mathsf{S}\ l_{\mathrm{H}}\ \tau \Rightarrow \mathsf{S}\ l_{\mathrm{L}}\ \mathsf{Bool}$ is equivalent when applied to two different secret inputs s_1 and s_2, first, we instantiate Lemma A.4.1 with the normal form of f, denoted by $\mathsf{norm}\ f$. In this manner, we obtain that the normal forms of f are exactly the constant function that returns *true* or *false* wrapped in the return. In the former case, by correctness of normalization we have that $f \approx \ulcorner\ \mathsf{norm}\ f\ \urcorner \approx \lambda\, x\,.\,\mathsf{return}\ \mathit{true}$. By β-reduction and congruence of term-level function application, we have that $\forall\ t.\ (\lambda\, x\,.\,\mathsf{return}\ \mathit{true})\ t \approx \mathsf{return}\ \mathit{true}$. Therefore, $f\ s_1 \approx f\ s_2$. The case when $\mathsf{norm}\ f \equiv \lambda\, x\,.\,\mathsf{return}\ \mathit{false}$ follows a similar argument. □

The noninterference property proven above characterizes what it means for a concrete class of programs, i.e. those of type $\emptyset \vdash f : \mathsf{S}\ l_{\mathrm{H}}\ \tau \Rightarrow \mathsf{S}\ l_{\mathrm{L}}\ \mathsf{Bool}$, to be secure: the attacker cannot even learn one bit of the secret from using program f. Albeit interesting, this property does not scale to more complex programs; for instance if the function f was typed in a non empty context the proof of the above lemma would not hold. The rest of this section is dedicated to generalize and prove noninterference from the program f to arbitrary programs written in λ_{SEC}. As will become clear, normal forms of λ_{SEC} play a crucial role towards proving noninterference.

A.6.2. General Noninterference Theorem

In order to discuss general noninterference for λ_{SEC}, we must first specify what are the *secret* (l_{H}) inputs of a program and its *public* (l_{L}) output with respect to an attacker at level l_{L}. The attacker can only learn information of a program by running it with different secret inputs and then observing its public output. Because the attacker can only observe outputs at their security level, we restrict the security condition to only consider programs where outputs are fully observable, i.e. *transparent* and *ground*, to the attacker.

Definition A.6.1 (Transparent type)**.**

- $()$ is transparent at any level l.
- ι is transparent at any level l.
- $\tau_1 \Rightarrow \tau_2$ is transparent at l iff τ_2 is transparent at l.

- $\tau_1 + \tau_2$ is transparent at l iff τ_1 and τ_2 are transparent at l.
- $\tau_1 \times \tau_2$ is transparent at l iff τ_1 and τ_2 are transparent at l.
- $\mathsf{S}\ l'\ \tau$ is transparent at l iff $l' \sqsubseteq l$ and τ is transparent at l.

Definition A.6.2 (Ground type)**.**

- $()$ is ground.
- ι is ground.
- $\tau_1 + \tau_2$ is ground iff τ_1 and τ_2 are ground.
- $\tau_1 \times \tau_2$ is ground iff τ_1 and τ_2 are ground.
- $\mathsf{S}\ l\ \tau$ is ground iff τ is ground.

A type τ is transparent at security level $l_{\mathtt{L}}$ if the type does not include the security monad type over a higher security level $l_{\mathtt{H}}$. A ground type, on the other hand, is a first order type, i.e. a type that does not contain a function type. These simplifying restrictions over the output type of a program allow us to state a generic noninterference property over terms and perform induction on the normal forms.

These restrictions do not hinder the generality of our security condition: a program producing a partially public output, for instance of product type $\mathsf{S}\ l_{\mathtt{L}}\ \mathsf{Bool} \times \mathsf{S}\ l_{\mathtt{H}}\ \mathsf{Bool}$, can be transformed to produce a fully public output by applying the snd projection. We return to this example later at the end of the section. Also note that previous work on proving noninterference for static IFC languages [Aba+99; MI04] impose similar restrictions.

Departing from the traditional view of programs as closed terms, i.e. terms without free variables, in the λ_{SEC} calculus we consider all terms for which a typing derivation exists. This includes terms that contain free variables—unknowns—typed by the context, which we identify as the program inputs. Note that open terms are more general since they can always be closed as a function by abstracting over the free variables.

Now, we state what it means for a context to be secret at level l. These definitions, dubbed l-sensitivity, force the types appearing in the context to be at least as sensitive as l.

Definition A.6.3 (Context sensitivity)**.**

A context Γ is l-sensitive if and only if for all types $\tau \in \Gamma$, τ is l-sensitive. A type τ is l-sensitive, on the other hand, if and only if:

- τ is the function type $\tau_1 \Rightarrow \tau_2$ and τ_2 is l-sensitive.
- τ is the product type $\tau_1 \times \tau_2$ and τ_1 and τ_2 are l-sensitive.
- τ is the monadic type $\mathsf{S}\ l'\ \tau_1$ and $l \sqsubseteq l'$.

Next, we define substitutions[5], which lay at the core of β-reduction rules in the λ_{SEC} calculus. Substitutions map free variables in a term to other terms possibly typed in a different context.

$$\textit{Substitution}\ \sigma ::= \sigma_\emptyset \mid \sigma\ [x \mapsto t]$$

$\boxed{\Gamma \vdash_{\text{sub}} \sigma : \Delta}$

$$\frac{\Gamma \vdash_{\text{sub}} \sigma : \Delta \qquad \Gamma \vdash t : \tau}{\Gamma \vdash_{\text{sub}} \sigma\ [\ x \mapsto t\ ] : \Delta\ ,\ x : \tau}\ (25) \qquad \frac{}{\Gamma \vdash_{\text{sub}} \sigma_\emptyset : \emptyset}\ (26)$$

Figure A.7.: Substitutions for λ_{SEC}

A substitution is either empty, $\sigma_\emptyset$, or is the substitution σ extended with a new mapping from the variable $x : \tau$ to term t. We denote $t\ [\ \sigma\ ]$ the application of substitution σ to term t. Its definition is standard by induction on the term structure, thus we omit it here and refer the reader to the Agda formalization.

Substitutions, in general, provide a mix of terms of secret and public type to fill the variables in the context Γ of a program. However, for noninterference we need to fix the public part of the substitution and allow the secret part to vary. We do so by splitting a substitution σ into the composition of a public substitution, $\Gamma \vdash_{\text{sub}} \sigma_{l_\mathbb{L}} : \Delta$, that fixes the public inputs, and a secret substitution $\Delta \vdash_{\text{sub}} \sigma_{l_\mathbb{H}} : \Sigma$, that restricts Δ to be $l_\mathbb{H}$-sensitive. The composition of both, denoted $\Gamma \vdash_{\text{sub}} (\sigma_{l_\mathbb{L}}\ ;\ \sigma_{l_\mathbb{H}}) : \Sigma$, maps variables in context Γ to terms typed in Σ: first, $\sigma_{l_\mathbb{L}}$ maps variables from Γ to terms in Δ, subsequently, $\sigma_{l_\mathbb{H}}$ maps variables in Δ to terms typed in Σ. Below, we state $l_\mathbb{L}$-equivalence of substitutions:

Definition A.6.4 (Low equivalence of substitutions)**.**

Two substitutions σ_1 and σ_2 are $l_\mathbb{L}$-equivalent , written $\sigma_1 \approx_{l_\mathbb{L}} \sigma_2$, if and only if for all $l_\mathbb{H}$ such that $l_\mathbb{H} \not\sqsubseteq l_\mathbb{L}$, there exists a public substitution $\sigma_{l_\mathbb{L}}$, and two secret substitutions $\sigma^1_{l_\mathbb{H}}$ and $\sigma^2_{l_\mathbb{H}}$, such that $\sigma_1 \equiv \sigma_{l_\mathbb{L}}\ ;\ \sigma^1_{l_\mathbb{H}}$ and $\sigma_2 \equiv \sigma_{l_\mathbb{L}}\ ;\ \sigma^2_{l_\mathbb{H}}$

[5] In Section A.2 we purposely left capture-avoiding substitutions underspecified, we amend that here.

Informally, noninterference for λ_{SEC} states that applying two low equivalent substitutions to an arbitrary term whose type is ground and transparent yields two equivalent programs. As previously explained, intuitively a program satisfies such property if it is equivalent to a *constant* program: i.e. a program where the output does not depend on the input—in this case the variables in the typing context. As in Section A.4, instead of defining and proving this on arbitrary terms, we achieve this using normal forms.

Constant Terms and Normal Forms We prove the noninterference theorem by showing that terms of a type at level $l_{\mathtt{L}}$, typed in a $l_{\mathtt{H}}$-sensitive context, must be constant. We achieve this in turn by showing that the normal forms of such terms are constant. Below, we state when a term is constant:

Definition A.6.5 (Constant term)**.**
A term $\Gamma \vdash t : \tau$ is said to be constant if, for any two substitutions σ_1 and σ_2, we have that $t\,[\,\sigma_1\,] \approx t\,[\,\sigma_2\,]$.

Similarly, we must define what it means for a normal form to be constant. However, we cannot state this for normal forms directly using substitutions since the result of applying a substitution to a normal form may not be a normal form. For example, the result of substituting the variable x in the normal form $x : \iota \Rightarrow \iota\ ,\ y : \iota \vdash_{\text{nf}} x\ y : \iota$ by the identity function is not a normal form—and *cannot* be derived syntactically as a normal form using $\vdash_{\text{nf}}$. Instead, we lean on the shape of the context to state the property.

If a normal form $\Gamma \vdash_{\text{nf}} n : \tau$ is constant, then there must exist a syntactically identical derivation $\emptyset \vdash_{\text{nf}} n' : \tau$ such that $n \equiv n'$. However, since n and n' are typed in different contexts, Γ and $\emptyset$, it is not possible to compare them for syntactic equality. We solve this problem by *renaming* the normal form n' to add as many variables as mentioned in context Γ. The signature of the renaming function is the following:

$$\mathsf{ren} : \{\Gamma \leqslant \Delta\} \to (\Gamma \vdash_{\text{nf}} \tau) \to (\Delta \vdash_{\text{nf}} \tau)$$

The relation $\leqslant$ between contexts Γ and Δ indicates that the variables appearing in Δ are at least those present in Γ. This relation, called *weakening*, is defined as follows:

- $\emptyset \leqslant \emptyset$
- If $\Gamma \leqslant \Delta$, then $\Gamma \leqslant \Delta\ ,\ x : \tau$
- If $\Gamma \leqslant \Delta$, then $\Gamma\ ,\ x : \tau \leqslant \Delta\ ,\ x : \tau$

The function ren can be defined by simple induction on the derivation of the normal forms. Note that terms can also be renamed in the same fashion.

Definition A.6.6 (Constant normal form)**.** A normal form $\Gamma \vdash_{\text{nf}} n : \tau$ is constant if there exists a normal form $\emptyset \vdash_{\text{nf}} n' : \tau$ such that $\mathsf{ren}\ (n') \equiv n$.

Further, we need a lemma showing that if a term is constant, then so is its normal form.

Lemma A.6.1 (Constant plumbing lemma)**.** *If the normal form n of a term $\Gamma \vdash t : \tau$ is constant, then so is t.*

The proof follows by induction on the normal forms:

Proof of Lemma A.6.1. If n is constant, then there must exist a normal form $\emptyset \vdash_{\text{nf}} n' : \tau$ such that $\mathsf{ren}\ (n') \equiv n$. Let the quotation of this normal form $\ulcorner n' \urcorner$ be some term $\emptyset \vdash t' : \tau$. Recall from earlier that terms can also be renamed, hence we have $\mathsf{ren}\ (t') \approx \mathsf{ren}\ (\ulcorner n' \urcorner)$ by correctness of n'. Since it can be shown that $\mathsf{ren}\ (\ulcorner n' \urcorner) \equiv \ulcorner\ \mathsf{ren}\ (n')\ \urcorner$, we have that $\mathsf{ren}\ (\ulcorner n' \urcorner) \equiv \ulcorner\ n\ \urcorner$, and by correctness of n, we also have $\mathsf{ren}\ (t') \approx t$ — (1).

A substitution σ maps free variables in a term to terms. The empty substitution, denoted $\sigma_\emptyset$, is the unique substitution, such that $\Delta \vdash t'\ [\ \sigma_\emptyset\ ] : \tau$ for any Δ. That is, applying the empty substitution simply renames the term. We can show that $t'\ [\ \sigma_\emptyset\ ] \equiv \mathsf{ren}\ (t')$, and hence, by (1), we have $t'\ [\ \sigma_\emptyset\ ] \approx t$ — (2). Since $\sigma_\emptyset$ renames a term typed in the empty context, we can show that for any substitution σ, we have $(t'\ [\ \sigma_\emptyset\ ])\ [\ \sigma\ ] \approx t'\ [\ \sigma_\emptyset\ ]$. Because $\sigma_\emptyset$ is also unique, for any two substitutions σ_1 and σ_2, we have $(t'\ [\ \sigma_\emptyset\ ])\ [\ \sigma_1\ ] \approx (t'\ [\ \sigma_\emptyset\ ])\ [\ \sigma_2\ ]$ by transitivity of $\approx$. As a result, from (2), we achieve the desired result, $t\ [\ \sigma_1\ ] \approx t\ [\ \sigma_2\ ]$, therefore t must be constant. □

The key insight of our noninterference proof is reflected in the following lemma which shows how normal forms of λ_{SEC} typed in a sensitive context are either constant or the flow between the security level of the context and the output type is permitted. Below we include the proof to showcase how it follows by straightforward induction on the shape of the normal forms.

Lemma A.6.2 (Normal forms do not leak)**.** *Given a normal form $\Gamma \vdash_{nf} n : \tau$, where the context Γ is l_i-sensitive, and τ is a ground and transparent type at level l_o, then either n is constant or $l_i \sqsubseteq l_o$.*

Proof. By induction on the structure of the normal form n. Note that λ and case normal forms need not be considered since the preconditions ensure that τ cannot be a function type (dismisses λ), and Γ cannot contain a variable of a sum type (dismisses case).

- **Case 1** ($\Gamma \vdash_{\text{nf}} () : ()$). The normal form $()$ is constant.
- **Case 2** ($\Gamma \vdash_{\text{nf}} n : \iota$). In this case, we are given the neutral n by the [Base] rule in Figure A.6. It can be shown by induction that for all neutrals of type $\Gamma \vdash_{\text{ne}} \tau$, if Γ is l_i-sensitive and τ is transparent at l_o, then $l_i \sqsubseteq l_o$. Hence, n gives us that $l_i \sqsubseteq l_o$.
- **Case 3** ($\Gamma \vdash_{\text{nf}} \mathsf{return}\ n : \mathsf{S}\ l\ \tau$). By applying the induction hypothesis on the normal form n, we have that n is either constant or $l_i \sqsubseteq l_o$. In the latter case, we are done since we already have $l_i \sqsubseteq l_o$. In the former case, there exists a normal form n' such that $\mathsf{ren}\ (n') \equiv n$. By congruence of the relation $\equiv$, we get that $\mathsf{return}\ (\mathsf{ren}\ (n')) \equiv \mathsf{return}\ n$. Note that the function ren is defined as $\mathsf{ren}\ (\mathsf{return}\ n') \equiv \mathsf{return}\ (\mathsf{ren}\ n')$, and hence by transitivity of $\equiv$, we have that $\mathsf{ren}\ (\mathsf{return}\ (n')) \equiv \mathsf{return}\ n$. Thus, the normal form $\mathsf{return}\ n$ is also constant.
- **Case 4** ($\Gamma \vdash_{\text{nf}} \mathsf{let}{\uparrow}\ x = n\ \mathsf{in}\ m : \mathsf{S}\ l_2\ \tau_2$). For this case, we have a neutral $\Gamma \vdash_{\text{ne}} n : \mathsf{S}\ l_1\ \tau_1$ such that $l_1 \sqsubseteq l_2$, by the [LetUp] rule in Figure A.6. Similar to case 2, we have that $l_i \sqsubseteq l_1$ from the neutral n. Hence, $l_i \sqsubseteq l_2$ by transitivity of the relation $\sqsubseteq$. Additionally, since $\mathsf{S}\ l_2\ \tau$ is transparent at l_o, it must be the case that $l_2 \sqsubseteq l_o$ by definition of transparency. Therefore, once again by transitivity, we have $l_i \sqsubseteq l_o$.
- **Case 5** ($\Gamma \vdash_{\text{nf}} \mathsf{left}\ n : \tau_1 + \tau_2$). Similar to return.
- **Case 6** ($\Gamma \vdash_{\text{nf}} \mathsf{right}\ n : \tau_1 + \tau_2$). Similar to return.

□

The last step to noninterference is an ancillary lemma which shows that terms typed in l_{H}-sensitive contexts are constant:

Lemma A.6.3. *Given a term $\Gamma \vdash t : \tau$, where the context Γ is l_{H}-sensitive, and τ is a ground type transparent at l_{L}. If $l_{\text{H}} \not\sqsubseteq l_{\text{L}}$, then t is constant.*

The proof follows from Lemmas A.6.2 and A.6.1.

Finally, we are ready to formally state and prove the noninterference property for programs written in λ_{SEC}, which effectively demonstrates that programs do not leak sensitive information. The proof follows from the previous lemmas, which characterize the behaviour of programs by the syntactic properties of their normal forms.

Theorem A.6.2 (Noninterference for λ_{SEC})**.** *Given security levels l_{L} and l_{H} such that $l_{\mathsf{H}} \not\sqsubseteq l_{\mathsf{L}}$; an attacker at level l_{L}; two l_{L}-equivalent substitutions σ_1 and σ_2 such that $\sigma_1 \approx_{l_{\mathsf{L}}} \sigma_2$; and a type τ that is ground and transparent at l_{L}; then for any term $\Gamma \vdash t : \tau$ we have that $t\,[\,\sigma_1\,] \approx t\,[\,\sigma_2\,]$.*

Proof of Theorem A.6.2. Low equivalence of substitutions $\sigma_1 \approx_{l_{\mathsf{L}}} \sigma_2$ gives that $\sigma_1 = \sigma_{l_{\mathsf{L}}} ; \sigma^1_{l_{\mathsf{H}}}$ and $\sigma_2 = \sigma_{l_{\mathsf{L}}} ; \sigma^2_{l_{\mathsf{H}}}$. After applying the public substitution $\sigma_{l_{\mathsf{L}}}$ to the term $\Gamma \vdash t : \tau$, we are left with a term typed in a l_{H}-sensitive context Δ, $\Delta \vdash t\,[\,\sigma_{l_{\mathsf{L}}}\,] : \tau$. By Lemma A.6.3, $t\,[\,\sigma_{l_{\mathsf{L}}}\,]$ is constant which means that $(t\,[\,\sigma_{l_{\mathsf{L}}}\,])\,[\,\sigma^1_{l_{\mathsf{H}}}\,] \approx (t\,[\,\sigma_{l_{\mathsf{L}}}\,])\,[\,\sigma^2_{l_{\mathsf{H}}}\,]$. By readjusting substitutions using composition we obtain $t\,([\,\sigma_{l_{\mathsf{L}}} ; \sigma^1_{l_{\mathsf{H}}}\,]) \approx t\,([\,\sigma_{l_{\mathsf{L}}} ; \sigma^2_{l_{\mathsf{H}}}\,])$, which yields $t\,[\,\sigma_1\,] \approx t\,[\,\sigma_2\,]$. □

A.6.3. Follow-up Example

To conclude this section, we briefly show how to instantiate the theorem of noninterference for λ_{SEC} for programs of type $\emptyset \vdash t : \mathsf{S}\ l_{\mathsf{L}}\ \mathsf{Bool} \times \mathsf{S}\ l_{\mathsf{H}}\ \mathsf{Bool} \Rightarrow \mathsf{S}\ l_{\mathsf{L}}\ \mathsf{Bool} \times \mathsf{S}\ l_{\mathsf{H}}\ \mathsf{Bool}$, which are the recurring example for explaining noninterference in the literature [RCH08; BA15]. Adapted to the notion of noninterference based on substitutions, the corollary we aim to prove is the following:

Corollary A.6.1 (Noninterference for t)**.** *Given security levels l_{L} and l_{H} such that $l_{\mathsf{H}} \not\sqsubseteq l_{\mathsf{L}}$ and a program $x : \mathsf{S}\ l_{\mathsf{L}}\ \mathsf{Bool} \times \mathsf{S}\ l_{\mathsf{H}}\ \mathsf{Bool} \vdash t : \mathsf{S}\ l_{\mathsf{L}}\ \mathsf{Bool} \times \mathsf{S}\ l_{\mathsf{H}}\ \mathsf{Bool}$ then $\forall\ p : \mathsf{S}\ l_{\mathsf{L}}\ \mathsf{Bool}\ ,\ s_1\ s_2 : \mathsf{S}\ l_{\mathsf{H}}\ \mathsf{Bool}$. we have that $t\,[\,x \mapsto (p\ ,\ s_1)\,] \approx t\,[\,x \mapsto (p\ ,\ s_2)\,]$.*

Because the main noninterference theorem requires the output to be fully observable by the attacker, we transform t to the desired shape by applying the snd projection. This is justified because the first component of the output is protected at level l_{H}, which the attacker cannot observe. Below we prove noninterference for $x : \mathsf{S}\ l_{\mathsf{L}}\ \mathsf{Bool} \times \mathsf{S}\ l_{\mathsf{H}}\ \mathsf{Bool} \vdash \mathsf{snd}\ t : \mathsf{S}\ l_{\mathsf{H}}\ \mathsf{Bool}$:

Proof of Corollary A.6.1. To apply Theorem A.6.2 we have to show that both substitutions are low equivalent, $[\,x \mapsto (p\ ,\ s_1)\,] \approx_{l_{\mathsf{L}}} [\,x \mapsto (p\ ,\ s_2)\,]$ The key idea is that the substitution $[\,x \mapsto (p\ ,\ s_1)\,]$ can be decomposed into a public substitution $\sigma_{l_{\mathsf{L}}} \equiv [\,x \mapsto (p\ ,\ y)\,]$ and two different secret substitutions where each replaces the variable y by a different secret, $\sigma^1_{l_{\mathsf{H}}} \equiv [\,y \mapsto s_1\,]$ and $\sigma^2_{l_{\mathsf{H}}} \equiv [\,y \mapsto s_2\,]$. Now, the proof follows directly from Theorem A.6.2. □

A.7. Conclusions and Future Work

In this paper we have presented a novel proof of noninterference for the λ_{SEC} calculus (based on Haskell's IFC library SecLib) using normalization. The simplicity of the proof relies upon the normal forms of the calculus, which as opposed to arbitrary terms, are well-principled. To obtain normal forms from terms, we have implemented normalization using NbE, and shown that normal forms obey useful syntactic properties such as neutrality and $\beta\eta$-long form. Most of the auxiliary lemmas and definitions towards proving noninterference build on these properties. Because normal forms are well-principled, many cases of the proofs follow directly by structural induction.

An important difference between our work and previous proofs based on term erasure is that our proof utilizes the static semantics of the language instead of the dynamic semantics. Specifically, our proof of noninterference is not tied to any particular evaluation strategy, such as call-by-name or call-by-value, assuming the strategy is adequate with respect to the static semantics.

Perhaps the closest to our line of work is the proof of noninterference by Miyamoto and Igarashi [MI04] for a modal lambda calculus using normalization. The main novelty of our proof is that it works for standard extensions of the simply typed lambda calculus and does not change the typing rules of the underlying calculus (as presented and implemented by Russo, Claessen, and Hughes [RCH08]). This makes our proof technique applicable even in the presence of other useful normalization-preserving extensions of STLC. For example, it should be possible to extend our proof for λ_{SEC} further with exceptions and other *computational* effects (à la Moggi [Mog89]) since our security monad is already an instance of this. Moreover, our proof relies on syntactic properties of normal forms in an open typing context since normalization is based on the static semantics of the language.

In this work we have only considered a calculus which models terminating computations. This opens up a question of whether our proof technique is applicable to languages which support general recursion, where computations need not necessarily terminate. The extensibility of this technique to recursion relies directly upon the choice of static semantics for normalizing recursion. For example, it may be possible to extend the proof for λ_{SEC} with a fixpoint combinator by treating it as an uninterpreted constant during normalization. That is, it may be sufficient to normalize the body of the function by ignoring the recursive application, because if the body does not leak a secret, then its recursive call must not either. Since complete normalization is not strictly needed for our purposes, we believe that our technique can also be extended to general recursion.

Our NbE implementation for λ_{SEC} extends NbE for Moggi's computational metalanguage [Fil01; Lin05] with a family of monads parameterized by a preordered set of labels. This resembles the parameterization of monads by effects specified by a preordered monoid, also known as *graded monads* [WT03; OP14], and thus indicates the extensibility of our NbE algorithm to calculi with graded monads. It would be interesting to see if our proof technique can be used to prove noninterference for static enforcement of IFC using graded monads.

Using static semantics means that our work lays a foundation for static analysis of noninterference-like security properties. This opens up a plethora of exciting opportunities for future work. For example, one possibility would be to use type-directed partial evaluation [Dan98] to simplify programs and inspect the resulting programs to verify if they violate security properties. Another arena would be the extension of our proof to more expressive IFC calculi such as dependency core calculus (DCC) or MAC [Vas+18]. The main challenge here would be to identify the appropriate static semantics of the language, as they may not always have been designed with one in mind.

Acknowledgements

We thank Alejandro Russo, Fabian Ruch, Sandro Stucki and Maximilian Algehed for the insightful discussions on normalization and noninterference. We would also like to thank Irene Lobo Valbuena, Claudio Agustin Mista and the anonymous reviewers at PLAS'19 for their comments on earlier drafts of this paper. This work was funded by the Swedish Foundation for Strategic Research (SSF) under the projects WebSec (Ref. RIT17-0011) and Octopi (Ref. RIT17-0023).

Bibliography

[Aba+99] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. "A Core Calculus of Dependency". In: *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*. Ed. by Andrew W. Appel and Alex Aiken. ACM, 1999, pp. 147–160. DOI: 10.1145/292540.292555. URL: https://doi.org/10.1145/292540.292555 (cit. on pp. 21, 38).

[AHS95] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. "Categorical Reconstruction of a Reduction Free Normalization Proof". In: *Category Theory and Computer Science, 6th International Conference, CTCS '95, Cambridge, UK, August 7-11, 1995, Proceedings*. Ed. by David H. Pitt, David E. Rydeheard, and Peter T. Johnstone. Vol. 953. Lecture Notes in Computer Science. Springer, 1995, pp. 182–199. DOI: 10.1007/3-540-60164-3_27. URL: https://doi.org/10.1007/3-540-60164-3%5C_27 (cit. on pp. 31, 32).

[Alg18] Maximilian Algehed. "A Perspective on the Dependency Core Calculus". In: *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. Ed. by Mário S. Alvim and Stéphanie Delaune. ACM, 2018, pp. 24–28. DOI: 10.1145/3264820.3264823. URL: https://doi.org/10.1145/3264820.3264823 (cit. on p. 21).

[AS19] Andreas Abel and Christian Sattler. "Normalization by Evaluation for Call-By-Push-Value and Polarized Lambda Calculus". In: *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*. Ed. by Ekaterina Komendantskaya. ACM, 2019, 3:1–3:12. DOI: 10.1145/3354166.3354168. URL: https://doi.org/10.1145/3354166.3354168 (cit. on pp. 25, 29, 51).

[BA15] William J. Bowman and Amal Ahmed. "Noninterference for free". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. Ed. by Kathleen Fisher and John H. Reppy. ACM, 2015, pp. 101–113. DOI: 10.1145/2784731.2784733. URL: https://doi.org/10.1145/2784731.2784733 (cit. on pp. 21, 43).

[BCF04] Vincent Balat, Roberto Di Cosmo, and Marcelo P. Fiore. "Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums". In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*. Ed. by Neil D. Jones and Xavier Leroy. ACM, 2004, pp. 64–76. DOI: 10.1145/964001.964007. URL: https://doi.org/10.1145/964001.964007 (cit. on pp. 29, 31, 32).

[BS91] Ulrich Berger and Helmut Schwichtenberg. "An Inverse of the Evaluation Functional for Typed lambda-calculus". In: *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*. IEEE Computer Society, 1991, pp. 203–211. DOI: 10.1109/LICS.1991.151645. URL: https://doi.org/10.1109/LICS.1991.151645 (cit. on p. 22).

[Coq93] Catarina Coquand. "From Semantics to Rules: A Machine Assisted Analysis". In: *Computer Science Logic, 7th Workshop, CSL '93, Swansea, United Kingdom, September 13-17, 1993, Selected Papers*. Ed. by Egon Börger, Yuri Gurevich, and Karl Meinke. Vol. 832. Lecture Notes in Computer Science. Springer, 1993, pp. 91–105. DOI: 10.1007/BFb0049326. URL: https://doi.org/10.1007/BFb0049326 (cit. on p. 36).

[Dan98] Olivier Danvy. "Type-Directed Partial Evaluation". In: *Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School, Copenhagen, Denmark, June 29 - July 10, 1998*. Ed. by John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann. Vol. 1706. Lecture Notes in Computer Science. Springer, 1998, pp. 367–411. DOI: 10.1007/3-540-47018-2_16. URL: https://doi.org/10.1007/3-540-47018-2%5C_16 (cit. on p. 45).

[Den76] Dorothy E. Denning. "A Lattice Model of Secure Information Flow". In: *Commun. ACM* 19.5 (1976), pp. 236–243. DOI: 10.1145/360051.360056. URL: https://doi.org/10.1145/360051.360056 (cit. on p. 23).

[DRR01] Olivier Danvy, Morten Rhiger, and Kristoffer Høgsbro Rose. "Normalization by evaluation with typed abstract syntax". In: *J. Funct. Program.* 11.6 (2001), pp. 673–680. DOI: 10.1017/S0956796801004166. URL: https://doi.org/10.1017/S0956796801004166 (cit. on p. 31).

[Fil01] Andrzej Filinski. "Normalization by Evaluation for the Computational Lambda-Calculus". In: *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001, Krakow, Poland, May 2-5, 2001, Proceedings*. Ed. by Samson Abramsky. Vol. 2044. Lecture Notes in Computer Science. Springer, 2001, pp. 151–165. DOI: 10.1007/3-540-45413-6_15. URL: https://doi.org/10.1007/3-540-45413-6%5C_15 (cit. on p. 45).

[Kav19] G. A. Kavvos. "Modalities, cohesion, and information flow". In: *Proc. ACM Program. Lang.* 3.POPL (2019), 20:1–20:29. DOI: 10.1145/3290333. URL: https://doi.org/10.1145/3290333 (cit. on p. 21).

[Lin05] Sam Lindley. "Normalisation by evaluation in the compilation of typed functional programming languages". PhD thesis. University of Edinburgh, UK, 2005. URL: https://hdl.handle.net/1842/778 (cit. on pp. 21, 25, 45).

[LZ10] Peng Li and Steve Zdancewic. "Arrows for secure information flow". In: *Theor. Comput. Sci.* 411.19 (2010), pp. 1974–1994. DOI: 10.1016/j.tcs.2010.01.025. URL: https://doi.org/10.1016/j.tcs.2010.01.025 (cit. on p. 21).

[McB18] Conor McBride. "Everybody's Got To Be Somewhere". In: *Proceedings of the 7th Workshop on Mathematically Structured Functional Programming, MSFP@FSCD 2018, Oxford, UK, 8th July 2018*. Ed. by Robert Atkey and Sam Lindley. Vol. 275. EPTCS. 2018, pp. 53–69. DOI: 10.4204/EPTCS.275.6. URL: https://doi.org/10.4204/EPTCS.275.6 (cit. on p. 31).

[MI04] Kenji Miyamoto and Atsushi Igarashi. "A modal foundation for secure information flow". In: *In Proceedings of IEEE Foundations of Computer Security (FCS)*. 2004, pp. 187–203 (cit. on pp. 38, 44).

[Mog89] Eugenio Moggi. "Computational Lambda-Calculus and Monads". In: *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*. IEEE Computer Society, 1989, pp. 14–23. DOI: 10.1109/LICS.1989.39155. URL: https://doi.org/10.1109/LICS.1989.39155 (cit. on pp. 24, 44).

[Mog91] Eugenio Moggi. "Notions of Computation and Monads". In: *Inf. Comput.* 93.1 (1991), pp. 55–92. DOI: 10.1016/0890-5401(91)90052-4. URL: https://doi.org/10.1016/0890-5401(91)90052-4 (cit. on pp. 22, 25).

[OP14] Dominic A. Orchard and Tomas Petricek. "Embedding effect systems in Haskell". In: *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*. Ed. by Wouter Swierstra. ACM, 2014, pp. 13–24. DOI: 10.1145/2633357.2633368. URL: https://doi.org/10.1145/2633357.2633368 (cit. on p. 45).

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002. ISBN: 978-0-262-16209-8 (cit. on pp. 24, 31).

[Plo80] Gordon D. Plotkin. "Lambda-definability in the full type hierarchy". In: *To H. B. Curry: essays on combinatory logic, lambda calculus and formalism*. Academic Press, London-New York, 1980, pp. 363–373 (cit. on p. 36).

[RCH08] Alejandro Russo, Koen Claessen, and John Hughes. "A library for light-weight information-flow security in haskell". In: *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*. Ed. by Andy Gill. ACM, 2008, pp. 13–24. DOI: 10.1145/1411286.1411289. URL: https://doi.org/10.1145/1411286.1411289 (cit. on pp. 21, 22, 29, 43, 44).

[Ste+11] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. "Flexible dynamic information flow control in Haskell". In: *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011*. Ed. by Koen Claessen. ACM, 2011, pp. 95–106. DOI: 10.1145/2034675.2034688. URL: https://doi.org/10.1145/2034675.2034688 (cit. on pp. 21, 22).

[Ter+12] David Terei, Simon Marlow, Simon L. Peyton Jones, and David Mazières. "Safe haskell". In: *Proceedings of the 5th ACM SIGPLAN Symposium on Haskell, Haskell 2012, Copenhagen, Denmark, 13 September 2012*. Ed. by Janis Voigtländer. ACM, 2012, pp. 137–148. DOI: 10.1145/2364506.2364524. URL: https://doi.org/10.1145/2364506.2364524 (cit. on p. 23).

[TS00] Anne Sjerp Troelstra and Helmut Schwichtenberg. *Basic proof theory, Second Edition*. Vol. 43. Cambridge tracts in theoretical computer science. Cambridge University Press, 2000. ISBN: 978-0-521-77911-1 (cit. on p. 27).

[TV19] Carlos Tomé Cortiñas and Nachiappan Valliappan. "Simple Noninterference by Normalization". In: *Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security, CCS 2019, London, United Kingdom, November 11-15, 2019*. Ed. by Piotr Mardziel and Niki Vazou. ACM, 2019, pp. 61–72. DOI: 10.1145/3338504.3357342. URL: https://doi.org/10.1145/3338504.3357342 (cit. on p. 19).

[TZ04] Stephen Tse and Steve Zdancewic. "Translating dependency into parametricity". In: *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, Snow Bird, UT, USA, September 19-21, 2004*. Ed. by Chris Okasaki and Kathleen Fisher. ACM, 2004, pp. 115–125. DOI: 10.1145/1016850.1016868. URL: https://doi.org/10.1145/1016850.1016868 (cit. on p. 21).

[Vas+18] Marco Vassena, Alejandro Russo, Pablo Buiras, and Lucas Waye. "MAC A verified static information-flow control library". In: *Journal of Logical and Algebraic Methods in Programming* 95 (2018), pp. 148–180. ISSN: 2352-2208. DOI: https://doi.org/10.1016/j.jlamp.2017.12.003. URL: https://www.sciencedirect.com/science/article/pii/S235222081730069X (cit. on pp. 22, 45).

[VR16] Marco Vassena and Alejandro Russo. "On Formalizing Information-Flow Control Libraries". In: *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2016, Vienna, Austria, October 24, 2016*. Ed. by Toby C. Murray and Deian Stefan. ACM, 2016, pp. 15–28. DOI: 10.1145/2993600.2993608. URL: https://doi.org/10.1145/2993600.2993608 (cit. on p. 21).

[VR19] Nachiappan Valliappan and Alejandro Russo. "Exponential Elimination for Bicartesian Closed Categorical Combinators". In: *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*. Ed. by Ekaterina Komendantskaya. ACM, 2019, 20:1–20:13. DOI: 10.1145/3354166.3354185. URL: https://doi.org/10.1145/3354166.3354185 (cit. on p. 36).

[WT03] Philip Wadler and Peter Thiemann. "The marriage of effects and monads". In: *ACM Trans. Comput. Log.* 4.1 (2003), pp. 1–32. DOI: 10.1145/601775.601776. URL: https://doi.org/10.1145/601775.601776 (cit. on p. 45).

Appendices

I. NbE for Sums

It is tempting to interpret sums component-wise like products and functions as: $[\![\tau_1 + \tau_2]\!] = [\![\tau_1]\!] \uplus [\![\tau_2]\!]$. However, this interpretation makes it impossible to implement reflection faithfully: should the reflection of a variable $x : \tau_1 + \tau_2$

be a semantic value of type $[\![\ \tau_1\ ]\!]$ (left injection) or $[\![\ \tau_2\ ]\!]$ (right injection)? We cannot make this decision since the value which substitutes x may be either of these cases. The standard solution to this issue is to interpret sums using *decision trees* [AS19]. A decision tree allows us to defer this decision until more information is available about the injection of the actual value.

As in the previous case for the monadic type T, a decision tree can be defined as an inductive data type D parameterized by some type interpretation a with the following constructors:

$$\textsc{Leaf}\ \frac{x : a}{\mathsf{leaf}\ x : D\ a} \qquad \textsc{Branch}\ \frac{n : \mathsf{Ne}\ (\tau_1 + \tau_2) \qquad f : \mathsf{Var}\ \tau_1\ \to\ D\ a \qquad g : \mathsf{Var}\ \tau_2\ \to\ D\ a}{\mathsf{branch}\ n\ f\ g : D\ a}$$

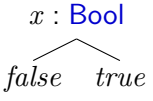
The leaf constructor constructs a leaf of the tree from a semantic value, while the branch constructor constructs a tree which represents a suspended decision over the value of a sum type. The branch constructor is the semantic equivalent of case in normal forms.

Decision trees allow us to model semantic sum values, and hence allow the interpretation of the sum type as follows:

$$[\![\ \tau_1 + \tau_2\ ]\!]\ =\ D\ ([\![\ \tau_1\ ]\!]\ \uplus\ [\![\ \tau_2\ ]\!])$$

We interpret a sum type (in λ_{SEC}) as a decision tree which contains a value of the sum type (in Agda).

As an example, the term *false* of type Bool, implemented as left (), will be interpreted as a decision tree leaf (inj_1 tt) of type $D\ [\![\ \mathsf{Bool}\ ]\!]$ since we know the exact injection. The Agda constructor inj_1 denotes the left injection in Agda, and inj_2 the right injection. For a variable x of type Bool, however, we cannot interpret it as a leaf since we don't know the actual injection that may substitute it. Instead, it is interpreted as a decision tree by branching over the possible values as branch x ($\lambda\ \_\ \to$ leaf (inj_1 tt)) ($\lambda\ \_\ \to$ leaf (inj_2 tt))[6]—which intuitively represents the following tree:



In light of this interpretation of sums, the implementation of evaluation for injections is straightforward since we only need to wrap the appropriate injection inside a leaf:

[6]We ignore the argument (as $\lambda\ \_$) here since it has the uninteresting type ()

```
eval (left t)  γ = leaf (inj₁ (eval t γ))
eval (right t) γ = leaf (inj₂ (eval t γ))
```

For evaluating case however, we must first implement a decision procedure since case is used to make a choice over sums.

To make a decision over a tree of type $D\ [\![\ \tau\ ]\!]$, we need a function $\mathsf{mkDec} : D\ [\![\ \tau\ ]\!] \to [\![\ \tau\ ]\!]$. It can be implemented by induction on the type τ using monadic functions fmap and join on trees, which can in turn be implemented by straightforward structural induction on the tree. Additionally, we will also need a function which converts a decision over normal forms to a normal form: $\mathsf{convert} : D\ (\mathsf{Nf}\ \tau) \to \mathsf{Nf}\ \tau$. The implementation of this function is made possible by the fact that branch resembles case in normal forms, and can hence be translated to it. We skip the implementation of these functions here, but encourage the reader to see the Agda implementation.

Using these definitions, we can now complete evaluation as follows:

```
eval (case t (left x₁ → t₁) (right x₂ → t₂)) γ =
  mkDec (fmap match (eval t γ))
  where
    match : ([[ τ₁ ]] ⊎ [[ τ₂ ]]) → [[ τ ]]
    match (inj₁ v) = eval t₁ (γ [x₁ ↦ v])
    match (inj₂ v) = eval t₂ (γ [x₂ ↦ v])
```

We first evaluate the term t of type $\tau_1 + \tau_2$ to obtain a tree of type $D\ ([\![\ \tau_1\ ]\!] \uplus [\![\ \tau_2\ ]\!])$. Then, we map the function match which eliminates the sum inside the decision tree to $[\![\ \tau\ ]\!]$, to produce a tree of type $D\ [\![\ \tau\ ]\!]$. Finally, we run the decision procedure mkDec on the resulting decision tree to produce the desired value of type $[\![\ \tau\ ]\!]$.

Reflection for a neutral of a sum type can now be implemented using branch as follows:

```
reflect {τ₁ + τ₂} n =
  branch n
    (leaf (λ x₁ → inj₁ (reflect {τ₁} x₁)))
    (leaf (λ x₂ → inj₂ (reflect {τ₂} x₂)))
```

As discussed earlier, we construct the decision tree for neutral n using branch. The subtrees represent all possible semantic values of n and are constructed by reflecting the variables x_1 and x_2.

The function reifyVal, on the other hand, is implemented similar to evaluation by eliminating the sum value inside the decision tree into normal forms as follows:

```
reifyVal {τ1 + τ2} tr = convert (fmap matchNf tr)
  where
    matchNf : ([[ τ1 ]] + [[ τ2 ]]) → Nf (τ1 + τ2)
    matchNf (inj1 x) = left (reifyVal {τ1} x)
    matchNf (inj2 y) = right (reifyVal {τ2} y)
```

With this function, we have completed the implementation of NbE for sums.

B

Securing Asynchronous Exceptions

Carlos Tomé Cortiñas, Marco Vassena, and Alejandro Russo

33rd IEEE Computer Security Foundations Symposium, CSF 2020, Boston, MA, USA, June 22-26, 2020

Abstract Language-based information-flow control (IFC) techniques often rely on special purpose, ad hoc primitives to address different covert channels that originate in the runtime system, beyond the scope of language constructs. Since these piecemeal solutions may not compose securely, there is a need for a unified mechanism to control covert channels. As a *first step* towards this goal, we argue for the design of a general interface that allows programs to safely interact with the runtime system and the available computing resources. To coordinate the communication between programs and the runtime system, we propose the use of asynchronous exceptions (*interrupts*), which, to the best of our knowledge, have not been considered before in the context of IFC languages. Since asynchronous exceptions can be raised at any point during execution—often due to the occurrence of an external event—threads must temporarily *mask* them out when manipulating locks and shared data structures to avoid deadlocks and, therefore, breaking program invariants. Crucially, the naive combination of asynchronous exceptions with existing features of IFC languages (e.g. concurrency and synchronization variables) may open up new possibilities of information leakage. In this paper, we present MAC$_{\text{ASYNC}}$, a concurrent, statically enforced IFC language that, as a novelty, features asynchronous exceptions. We show how asynchronous exceptions easily enable (out of the box) useful programming patterns like speculative execution and some degree of resource management. We prove that programs in MAC$_{\text{ASYNC}}$ satisfy progress-sensitive noninterference and mechanize our formal claims in the Agda proof assistant.

B.1. Introduction

Information-flow control (IFC) [SM03] is a promising approach for preserving confidentiality of data. It tracks how data of different sensitivity levels (e.g. public or sensitive) flows within a program, and raises alarms when confidentiality might be at stake. This technology has been previously used to secure operating systems (e.g. [Zel+06; Van+07]), web browsers (e.g. [Ste+14; Yip+09]), and several programming languages (e.g. [Hed+14; Mye+06; RCH08]).

Most language-based approaches for IFC reason about constructions found in programs (e.g. variables, branches, and data structures), while often ignoring aspects of runtime systems which might create *covert channels* (e.g. [BR13; PA17; Vas+19]) capable of producing leaks, e.g. through caches, parallelism, resource usage, etc. To deal with this problem, researchers have proposed security-aware runtime system designs [Vas+19; PA19]. However, building runtime systems is a major endeavour and these proposals have yet to be implemented. A more lightweight approach to securing runtime systems relies on *special-purpose language constructs* that coordinate the execution of programs with different components of the runtime—e.g. the garbage collector [PA17], the scheduler [RS06], timeouts [RS09], lazy evaluation [VBR17] and caches [Fer+18].[1] While a step in the right direction, designing ad hoc constructs every time that some coordination with the runtime system is needed feels rather unsatisfactory—an observation that has also been made outside the security arena [Li+07; Siv+16; FF04]. In fact, implementing *hooks* in an existing runtime system requires specific knowledge of its internals and considerable expertise. Even worse, the composition of piecemeal security solutions may weaken or even break the security guarantees of the runtime system as a whole. These issues suggest the need for a unified mechanism to close covert channels in the runtime system. As a first step towards this goal, we believe that runtime systems should expose a general *IFC-aware interface* that allow IFC languages to systematically control and secure components of the runtime system. How should programs coordinate with the runtime system through this interface?

In the 70s, Unix-like operating systems conceived *signals* as a limited form of inter process communication (IPC).[2] Signals are no more than asynchronous notifications sent to processes in order to notify them of the occurrence of events, where the origin of signals is either the kernel or other processes. Furthermore, when receiving a signal, process execution can be interrupted

[1] In this last case, we are abusing the term runtime to denote "the rest of the system."
[2] https://standards.ieee.org/content/ieee-standards/en/standard/1003_1-2017.html

during any *nonatomic* instruction—and if the process has previously registered a signal handler, then that routine gets executed. If we think of the kernel as "the runtime" and of processes as our "programs", signals are exactly the mechanism needed to implement the interface that we need! In fact, and generally speaking, the idea of OS-signals have been already internalized by programming languages in the form of *asynchronous exceptions*.

Asynchronous exceptions are raised as a result of external events and can occur at *any point* of the program. As a result, they are considered so difficult to master that many languages (e.g. Python [FM02] and Java [Ora20]) either restrict or completely forbid programmers from using them. The main reason is that interrupting a program at any point might break, for instance, a datastructure invariant or result in holding a lock indefinitely—and it is not that clear how to get out of such situation.

Despite not being widely adopted in its full expressive power, asynchronous exceptions enable very useful programming patterns: *speculative execution* (i.e. a thread can spawn a child thread and later decide that it does not need the result and kill it), *timeouts*, and *resource management*.

Our Contributions

In this work, we present MAC$^{\text{Async}}$, a Haskell IFC library that extends the concurrent version of MAC [Rus15; Vas+18] with asynchronous exception. We formally prove progress-sensitive noninterference (PSNI) [HS12] for MAC$^{\text{Async}}$ and provide mechanized proofs in Agda [Abe+05] of all our claims as supplementary material to this work. We believe that the extension presented in this paper and its formal security guarantees extend to other Haskell IFC libraries (e.g. LIO [Ste+11b]).

The semantics for asynchronous exceptions in MAC$^{\text{Async}}$ is inspired by how asynchronous exception are modelled in Haskell [Mar+01]—where a mechanism of *masking/unmasking* marks regions of code where asynchronous exceptions can be safely raised. However, allowing untrusted code to mask exceptions *arbitrarily* poses other security risks. For example, a rouge thread could abuse the masking mechanism to exhaust all available computing resources and starve other threads in the system without the risk of being terminated. To avoid that, we propose a fine-grained (selective) masking/unmasking mechanism instead of the traditional *all-or-nothing* approaches, which disable all asynchronous exceptions inside handlers [Rep90; GM84]. Furthermore, in contrast with [Mar+01], our design forbids raising multiple exceptions at the same time, which, we believe, can too easily disrupt programs in unpredictable ways. While an exception is raised, our language does not raise incoming exceptions, which

are, instead, stored in a queue of pending exceptions and raised only when the current one has been handled.

From the security perspective, asynchronous exceptions follow the *no write-down* security check for IFC: when throwing an asynchronous exception, the security level of the source thread should flow to the security level of the recipient. The caveats are, however, in the formalization of masking/unmasking mechanisms and the noninterference proof. For example, it is important for security that asynchronous exception are *deterministically* inserted into the queue of pending exceptions. We utilize *term erasure* as the proof technique and leverage *double-step erasure* to deal with the complexity of our semantics (i.e. concurrency, synchronization variables and asynchronous exceptions), like in previous existing work (e.g. [Vas+18; Vas+16; VR16]).

In summary, our list of contributions includes:

- An extension to MAC, called MACASYNC, to handle IFC-aware asynchronous exceptions in the presence of concurrency.
- Formal semantics, enforcement, and progress-insensitive noninterference guarantees for MACASYNC.
- Mechanized proofs of all our claims in approximately 3,000 lines of AGDA.[3]
- We showcase MACASYNC and the new programming patterns enabled by asynchronous exceptions with two examples, in which we implement secure versions of (i) a speculative execution combinator, and (ii) a load-balancing controller for sensitive worker threads, respectively.

To the best of our knowledge, this work is the first account for asynchronous exceptions in concurrent IFC-systems. The rest of the paper is organized as follows. In Section B.2, we revisit MAC's API. Section B.3 presents MACASYNC by example. In Section B.4, we extend MAC's semantics to track asynchronous exceptions. In Section B.5, we introduce asynchronous exceptions and the masking/unmasking mechanisms. Section B.6 presents our security guarantees. Section B.7 describes related work and Section B.8 concludes.

B.2. The MAC IFC Library

To help readers get familiar with the MAC IFC library [Rus15], we give a brief overview of its API and programming model.

[3]Available at `https://bitbucket.org/carlostome/mac-async`.

```
  -- Abstract types
data Labeled l τ
data MAC l τ
  -- Monadic structure for computations
instance Monad (MAC l)
  -- Core operations
label   :: l_L ⊑ l_H ⇒ τ → MAC l_L (Labeled l_H τ)
unlabel :: l_L ⊑ l_H ⇒ Labeled l_L τ → MAC l_H τ
```

Figure B.1.: Core API for MAC

Security Lattice The information flow policies enforced by MAC are specified by a security lattice [Den76], which defines a partial order between security levels (*labels*). These labels represent the sensitivity of program inputs and outputs and the *order* between them dictates which flows of information are allowed in a program. For example, the classic two-point lattice $L = (\{\mathtt{L}, \mathtt{H}\}, \sqsubseteq)$ classifies data as either *public* (L) or *secret* (H) and only prohibits sending secret inputs into public outputs, i.e. $\mathtt{H} \not\sqsubseteq \mathtt{L}$. In MAC, the security lattice is embedded in Haskell using standard features of the type system [Rus15]. In particular, each security label is represented by an abstract datatype and valid flows of information (the $\sqsubseteq$ relation between labels) are encoded using typeclass constructs—see Figure B.1.

Security Types MAC enforces security statically by means of special types annotated with security labels. The abstract type *Labeled* l τ associates label l with data of type τ. For example, *pwd* :: *Labeled* H *String* is a secret string and *score* :: *Labeled* L *Int* is a public integer. The abstract type *MAC* l τ represents a side-effectful computation that manipulates data labelled with l and whose result has type τ. MAC provides a monadic interface to help programmers write secure code. The basic primitives of the interface are *return* and *bind* (written as the infix operator $\gg\!=$). Primitive *return* :: $\tau \to$ *MAC* l τ creates a computation that simply returns a value of type τ without causing side-effects. Primitive $(\gg\!=)$ :: *MAC* l $\tau_1 \to (\tau_1 \to$ *MAC* l $\tau_2) \to$ *MAC* l τ_2 chains two computations (at the same security level l) together, in a *sequence*. Specifically, program $m \gg\!= f$ takes the result obtained by executing m and *binds* it to function f, which produces the rest of the computation. Our examples use **do**-notation, Haskell syntactic sugar for monadic computations. For instance,

we write **do** $x \leftarrow m$; *return* $(x + 1)$ for the program $m \gg\!= \lambda x \rightarrow$ *return* $(x + 1)$, which increments by one the result returned by m.

Flows of Information In order to enforce information flow policies, MAC regulates the interaction between *MAC* computations and *Labeled* data. Computations cannot *write* and *read* labelled data directly, but must use special functions *label* and *unlabel* (Figure B.1). These functions create and read labelled data as long as these operations comply with specific security rules, known as *no write-down* and *no read-up* [BL96]. Intuitively, function *label* writes some data into a fresh, $l_{\mathtt{H}}$-labelled value as long as the decision to do so depends on less sensitive data, i.e. the computation is labelled with $l_{\mathtt{L}}$ such that $l_{\mathtt{L}} \sqsubseteq l_{\mathtt{H}}$. (To help readers, we use subscripts in metavariables $l_{\mathtt{L}}$ and $l_{\mathtt{H}}$ to indicate that $l_{\mathtt{L}} \sqsubseteq l_{\mathtt{H}}$). Dually, function *unlabel* allows $l_{\mathtt{H}}$-labelled computations to read data from lower security levels, i.e. data labelled with $l_{\mathtt{L}}$ such that $l_{\mathtt{L}} \sqsubseteq l_{\mathtt{H}}$. In the type signatures of these functions, the precondition $l_{\mathtt{L}} \sqsubseteq l_{\mathtt{H}}$ is a typeclass constraint, which must be statically satisfied when type checking programs. As a result, programs that attempt to leak secret data, e.g. via *implicit flows*, are ill-typed and rejected by the compiler. In particular, programs cannot branch on *secret* H-labelled data directly, but must use *unlabel* first to extract its content. Once unlabelled, secret data can only be manipulated within a computation labelled with H thanks to the type of *unlabel* and *bind*. Then, trying to use function *label* to create *public* L-labelled data triggers a type error that represents a violation of the no write-down rule. Specifically, an attempt to create public data from within a secret context generates an *unsatisfiable* type constraint $\mathtt{H} \sqsubseteq \mathtt{L}$, arising from the use of *label*.

MAC incorporates other kinds of resources (e.g. references and network sockets) in a similar way. Resources are encapsulated in *labelled* resources handlers and the API exposed to labelled computations is designed so that the read and write side-effects of each operation respect the *no read-up* and *no write-down* rules.

Concurrency Extending IFC languages with concurrency is a delicate task because threads provide attackers with new means to leak data. For example, the possibility of executing computations concurrently magnifies the bandwidth of the termination covert channel [Ste+12]. This channel enables brute force attacks in which threads try to guess the secret and enter into a loop to suppress public outputs if they succeed. Even worse, the combination of concurrency and shared resources can introduce subtle *internal-timing channels* [SV98]. This covert channel is exploited by attacks that influence the (public) outcome of *data*

$$
\begin{aligned}
&\mathit{fork} :: l_{\mathtt{L}} \sqsubseteq l_{\mathtt{H}} \Rightarrow \mathit{MAC}\ l_{\mathtt{H}}\ () \rightarrow \mathit{MAC}\ l_{\mathtt{L}}\ ()\\
&\textbf{data}\ \mathit{MVar}\ l\ \tau\\
&\mathit{newMVar} :: l_{\mathtt{L}} \sqsubseteq l_{\mathtt{H}} \Rightarrow \tau \rightarrow \mathit{MAC}\ l_{\mathtt{L}}\ (\mathit{MVar}\ l_{\mathtt{H}}\ \tau)\\
&\mathit{putMVar}\ \ :: \mathit{MVar}\ l\ \tau \rightarrow \tau \rightarrow \mathit{MAC}\ l\ ()\\
&\mathit{takeMVar} :: \mathit{MVar}\ l\ \tau \rightarrow \mathit{MAC}\ l\ \tau
\end{aligned}
$$

Figure B.2.: Concurrent API for MAC

races with secret data [BR13; Ste+12]. To support concurrency securely, MAC: (i) decouples computations that manipulate secret data from computations that can generate public outputs, and (ii) prevents threads (labelled computations) from affecting data races between threads at lower security levels. Primitive *fork* (Figure B.2) allows a $l_{\mathtt{L}}$-labelled computation to fork a thread at a higher security level, i.e. labelled with $l_{\mathtt{H}}$ such that $l_{\mathtt{L}} \sqsubseteq l_{\mathtt{H}}$. Intuitively, forking constitutes a *write* operation and thus the type of *fork* enforces the no write-down rule. MAC does not implements threads directly, but relies on Haskell *green* (lightweight user-level) threads. These threads are managed by the GHC runtime system running a round-robin scheduler, which is compatible with the security guarantees of MAC [VR16; Vas+18].

Synchronization Variables MAC supports shared mutable state in the form of synchronization variables, following the style of Concurrent Haskell [JGF96]. The abstract type $\mathit{MVar}\ l\ \tau$ (Figure B.2) represents a synchronization variable that can be either *empty* or *full* with a value of type τ at security level l. Threads can create and atomically access synchronization variables with functions *newMVar*, *putMVar* and *takeMVar*. Function *newMVar* creates a synchronization variable initially full with the given value. (Like *label* and *fork*, function *newMVar* performs a write side-effect, thus its type signature has a similar security check). Functions *putMVar* and *takeMVar* allow threads to write and read shared variables *synchronously*. In particular, these functions *block* threads trying to read or write variables in the wrong "state". For example, function *putMVar* writes a value into an *empty* variable and blocks the thread if the variable is full. Dually, function *takeMVar* empties a *full* variable and returns its content and blocks the caller otherwise. Notice that *putMVar* and *takeMVar* perform both *read* and *write* side-effects: they must always read the variable to determine whether the caller should be blocked. Then, the *no read-up* and *no write-down* security rules imply that these functions are secure

only when they operate within the same security level, i.e. both the variable and the computation are labelled with l [Rus15].

B.3. MACasync by Example

Asynchronous exceptions enable useful programming patterns that, to our knowledge, cannot be coded securely in any existing IFC language. We illustrate some of these idioms in MACASYNC, which extends MAC with three new primitives *throwTo*, *mask* (*unmask*), and *catch*. These primitives allow threads to (1) send signals to threads at higher security levels by throwing exceptions *asynchronously*, (2) *suppress* (*enable*) exceptions in specific regions of code, and (3) *react* to exceptions by running their corresponding exception handler, respectively.

Example B.3.1 (Speculative execution)**.** Imagine two implementations of the same algorithm whose performance depends on the input. Instead of settling for one, we could run both concurrently and just return the output of the first that finishes. At that point the thread computing the other algorithm may be killed since its result is no longer necessary. We can implement such a combinator for speculative execution in MACASYNC using asynchronous exceptions. First, we declare $Kill :: Exception$ as a new exception, and define $kill\ t$, a function that sends exception *Kill* asynchronously to the thread identified by t.[4]

```
1 data Exception = Kill | ...
2 kill t = throwTo t Kill
```

Then, we define the combinator *speculate*, which receives two computations c_1 and c_2 to run speculatively.

```
3  speculate :: MAC l a → MAC l a → MAC l a
4  speculate c1 c2 = do
5    m ← newEmptyMVar
6    t1 ← fork (c1 >>= putMVar m)
7    t2 ← fork (c2 >>= putMVar m)
8    r  ← takeMVar m
9    kill t1; kill t2
10   return r
```

The combinator creates an *empty* synchronization variable m (line 5) and forks two threads (6–7), which run computations c_1 and c_2 concurrently and then

[4] In MACASYNC, primitive *fork* returns the identifier of the child thread to the parent.

write the result to m. When the combinator reads variable m (8), it *blocks* until either thread terminates and fills it with the result. When this happens, the combinator resumes, kills the children threads (one may still be running) (9), and returns the result (10).

Example B.3.2 (Thread pool)**.** This example presents the code of a *controller* thread that maintains a pool of *worker* threads to perform computations on a stream of incoming (sensitive) inputs. In this scheme, the controller thread manages the worker threads in the pool by reacting to asynchronous exceptions sent by other (public and secret) threads in the system. For example, when some secret input becomes available, a thread can send an exception $Input_{\mathrm{H}}$ *secret* to the controller thread, which extracts the secret data and forwards it to the first available worker thread to process it. Similarly, when the thread pool is no longer needed, it can be deallocated by sending the exception *Kill* to the controller, which then kills each worker thread in the pool. In the same way, the controller could be programmed to react to specific exceptions and carry out even more tasks (e.g. dynamically resizing the thread pool).

To set up this scheme, a thread calls function *initTP* (Figure B.3) to initialize the thread pool and start the controller thread. Function *initTP n f* allocates an empty synchronization variable m (line 6), forks a pool of n worker threads executing function f (line 7), collects their identifiers *ts*, and passes it to the controller thread (line 8). As new input becomes available, the controller writes it to the shared variable m, which is then read by one of the workers and its content processed via function f (line 11). To avoid getting killed in the middle of a computation, worker threads *mask* exception *Kill* while processing data, thus ensuring that they always complete on-going computations without aborting prematurely. It may seem erroneous to mask also instruction *takeMVar*: can this cause a worker thread to *block indefinitely* waiting for new input? No, in Concurrent Haskell, and MACasync, operations that can block indefinitely (like *takeMVar*) are *interruptible*, i.e. they can receive and raise asynchronous exceptions even in masked blocks [Mar+01].

Function *controller ts m* implements a controller thread for the thread pool *ts* sharing variable m. As long as it receives no exception, the controller thread simply waits on an always-empty synchronization variable via *wait* (line 15). When the thread receives an exception, it resumes and executes the corresponding code in the list of exception handlers. In particular, when new secret input becomes available ($Input_{\mathrm{H}}$ *secret*), it opens the secret (line 19) and writes it to variable m (line 20), so that the worker threads can process it. Notice that if variable m is *full* at this point, then some previous input is still waiting to be processed (all workers threads are busy) and the controller just

```
1  type Data = ...
2  data Exception = Kill | Input_l (Labeled l Data) | ...
3  type Size = Int
4  initTP :: Size → (Data → MAC H ()) → MAC L (TId H)
5  initTP n f = do
6    m ← newEmptyMVar
7    ts ← forM [1 .. n] (λ_ → fork (worker f m))
8    fork (controller ts m)
9  worker :: (Data → MAC H ()) → MVar H Data → MAC H ()
10 worker f m = do
11   mask [Kill] (takeMVar m >>= f)
12   worker f m
13 controller :: [TId H] → MVar H Data → MAC H ()
14 controller ts m =
15   let wait = newEmptyMVar >>= takeMVar in
16   catch wait
17     [(Input_H secret,
18      mask [Input_H, Kill]
19           (do s ← unlabel secret
20               putMVar m s
21               unmask [Input_H, Kill] (controller ts m)))
22     ,(Kill,
23      mask [Input_H, Kill] (forM ts kill)) ]
```

Figure B.3.: Thread pool example

waits on the variable. As soon as a worker thread completes, it empties the variable containing the pending input, and the controller resumes by writing the variable; then it continues to wait for further exceptions. To avoid dropping any input, the controller thread masks exceptions *Kill* and $Input_H$ (line 18) while processing requests. For example, if exceptions were not properly masked in that block of code, the controller could receive an exception, e.g. *Kill*, which would terminate the thread while trying to feed the last input received to the workers. Once done, the controller unmasks the exceptions again (line 21) and continues to wait for new input. After receiving and *eventually* raising the exception *Kill*, the controller thread propagates it to all the workers in the pool (line 23) and then terminates. Also in this case, the controller thread masks the

other exceptions, which could otherwise prematurely terminate the controller and leave some of the worker threads alive.

The example, however, has a catch! Primitive *putMVar* may also block the controller indefinitely like *takeMVar*, and thus may likewise be *interrupted* and raise an exception, even if that exception is masked. As a result, the controller thread could also be interrupted on line 20 and drop the current input. To fix the program, we introduce the combinator *retry killed m ss*, which repeatedly attempts to fill variable *m* with the inputs pending in list *ss* while handling other exceptions.

```
24 retry :: Bool → [TId H] → MVar H Data
25                          → [Data] → MAC H ()
26 retry killed ts m [] =
27   if killed then forM ts kill; exit else return ()
28 retry killed m (s : ss) =
29   catch (putMVar m s)
30         [(Input_H secret,
31          do s' ← unlabel secret
32             if killed
33                then retry killed ts m (s : ss)
34                else retry killed ts m ((s : ss) ++ [s'])
35         ,(Kill, retry True ts m (s : ss))]
```

If further inputs are received while executing *retry*, the function appends them to list *ss* to avoid dropping them, and thus ensuring that they will eventually be delivered to the workers. If the controller receives exception *Kill* while retrying, the Boolean flag *killed* is switched on and further inputs are discarded. In this case, when all the inputs received before *Kill* are dispatched, the controller kills the worker threads and terminates with *exit* (line 27)—the function *retry* assumes exceptions *Input* H and *Kill* are masked so this operation will not be interrupted. In conclusion, to repair the code of *controller*, we simply replace *putMVar m s* (line 20) with *retry False ts m* [*s*].

Even though relatively simple, these examples cannot be coded in IFC languages without support for asynchronous communication like MAC. In these languages, synchronous primitives (e.g. *MVar*) must be restricted to operate within a single security level for security reasons, as explained in Section B.2 and Vassena, Russo, Buiras, and Waye [Vas+18]. For instance, if only synchronous communication was available, then the *controller* thread from our second example could not receive commands from public (L-labelled)

threads.[5]

B.4. Formal Semantics

B.4.1. Core of MACasync

From a security perspective, the interaction between synchronization variables, asynchronous exceptions, and exception masking is a delicate matter. MACasync implements these primitives on top of those provided by Concurrent Haskell, whose runtime is not designed with security in mind. For example, the fact that a thread may be able to resume another by sending an asynchronous exception [Mar+01] (as explained in the second example above) may introduce subtle internal timing covert channels that weaken the security guarantees of MAC. To rule that out, we extend the small-step semantics of MAC from Vassena, Russo, Buiras, and Waye [Vas+18] with asynchronous exceptions and perform a rigorous, comprehensive security analysis of the whole language.

The core of MACasync is the standard call-by-name λ-calculus with Boolean and unit type (Figure B.4). We specify the side-effect free semantics of the core λ-calculus (e.g. function abstraction, application) as a small-step reduction relation, $t_1 \rightsquigarrow t_2$, which denotes that term t_1 reduces in one step to t_2. These reduction rules are standard and we completely omit them in this presentation.

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

Figure B.4.: Core syntax

The defining feature of MACasync is the security monad *MAC*, which encapsulates computations that may produce side-effects. Figure B.5 specifies the syntax and part of the semantics for the side-effectful constructs of the language. The small-step relation $t_1 \longrightarrow t_2$ denotes a single *sequential* step that brings term t_1 of type $\mathit{MAC}\ l\ \tau$ to t_2.

Rule (Unlabel$_1$) reduces term *unlabel* t_1 to *unlabel* t_2 by evaluating the argument through a *pure* semantics step $t_1 \rightsquigarrow t_2$. When the argument is

[5] In MAC, a public thread could technically communicate asynchronously with a secret thread by updating a secret, mutable reference. However, these labelled references would inevitably introduce serious data races and thus do not represent a viable alternative.

evaluated, rule (UNLABEL$_2$) extracts the content of the labelled value and returns it in the security monad.

$$\begin{array}{lll} \text{Types:} & \tau ::= \cdots \mid \mathit{MAC}\ l\ \tau \mid \mathit{Labeled}\ l\ \tau \\ \text{Values:} & v ::= \cdots \mid \mathit{Labeled}\ t \mid \mathit{return}\ t \\ \text{Terms:} & t ::= \cdots \mid \mathit{label}\ t \mid \mathit{unlabel}\ t \mid t_1 \gg\!= t_2 \end{array}$$

$$\frac{t_1 \rightsquigarrow t_2}{\mathit{unlabel}\ t_1 \longrightarrow \mathit{unlabel}\ t_2}\ (\text{UNLABEL}_1) \qquad \mathit{unlabel}\ (\mathit{Labeled}\ t) \longrightarrow \mathit{return}\ t\ (\text{UNLABEL}_2)$$

Figure B.5.: Syntax and semantics of MACASYNC (excerpts)

B.4.2. Synchronization Variables

Figure B.6 extends MACASYNC with synchronization variables. The store Σ is partitioned by label into separate memory segments S, each consisting of a list of memory cells c, which can be either *full* with a term ($(\!|\, t \,|\!)$) or *empty* ($\otimes$). A value $\mathit{MVar}_l\ n$ denotes a synchronization variable that refers to the n-th cell of the l-labelled memory segment in the store.[6]

In rule (NEW), primitive $\mathit{newMVar}_l\ t$ allocates a new memory cell containing term t in the l-labelled segment of the store, at *fresh* address $n = |\Sigma(l)|$, i.e. $\Sigma[(l, n) \mapsto (\!|\, t \,|\!)]$, and returns the corresponding synchronization variable $\mathit{MVar}_l\ n$. Term $\mathit{putMVar}_l\ t_1\ t_2$ writes term t_2 into the *empty* cell pointed by the synchronization variable t_1. To do that, rule (PUT$_1$) starts evaluating the variable t_1 through a pure semantics step $t_1 \rightsquigarrow t_1'$. When the variable is fully evaluated, e.g. $\mathit{MVar}_l\ n$, rule (PUT$_2$) takes over and writes the given term t in the cell identified by (l, n), i.e. $\Sigma[(l, n) \mapsto (\!|\, t \,|\!)]$. Notice that the term steps only if the cell in the store Σ is initially *empty*, i.e. $(l, n) \mapsto \otimes \in \Sigma$. If the cell is *full*, the term cannot be reduced by any other rule of the semantics and gets *stuck*, capturing the intended *blocking* behaviour of synchronization variables. We omit the rules for *takeMVar*, which follow a similar pattern [Vas+18].

B.4.3. Concurrency

Unlike previous concurrent incarnations of MAC, threads in MACASYNC can communicate with each other by sending signals in the form of asynchronous

[6] Some terms in the calculus carry a label annotation that is inferred from its type. For example, the label l in $\mathit{MVar}_l\ n$ comes from its type $\mathit{MVar}\ l\ \tau$.

$$
\begin{array}{lll}
\text{Store:} & \Sigma \in \mathit{Label} \to \mathit{Memory} \\
\text{Memory:} & S ::= [\,] \mid c : S \\
\text{Cell:} & c ::= \otimes \mid (\!|\, t \,|\!) \\
\text{Addresses:} & n \in \mathbb{N} \\
\text{Types:} & \tau ::= \cdots \mid \mathit{MVar}\ l\ \tau \\
\text{Values:} & v ::= \cdots \mid \mathit{MVar}_l\ n \\
\text{Terms:} & t ::= \cdots \mid \mathit{newMVar}_l\ t \mid \mathit{takeMVar}\ t \\
 & \qquad \mid \mathit{putMVar}\ t_1\ t_2
\end{array}
$$

$$
\frac{n = |\Sigma(l)|}{\Sigma, \mathit{newMVar}_l\ t \longrightarrow \Sigma[(l,n) \mapsto (\!|\, t \,|\!)], \mathit{return}\ (\mathit{MVar}_l\ n)}\ (\textsc{New})
$$

$$
\frac{t_1 \rightsquigarrow t_1'}{\Sigma, \mathit{putMVar}\ t_1\ t_2 \longrightarrow \Sigma, \mathit{putMVar}\ t_1'\ t_2}\ (\textsc{Put}_1)
$$

$$
\frac{(l,n) \mapsto \otimes \in \Sigma \qquad \Sigma' = \Sigma[(l,n) \mapsto (\!|\, t \,|\!)]}{\Sigma, \mathit{putMVar}\ (\mathit{MVar}_l\ n)\ t \longrightarrow \Sigma', \mathit{return}\ ()}\ (\textsc{Put}_2)
$$

Figure B.6.: Syntax and semantics for synchronization variables

exceptions. To enable this form of communication, the runtime system assigns a unique thread identifier to each thread of the system. Thread identifiers are *opaque* to avoid leaking secret data through the number of threads in the system, and *labelled* to prevent sensitive threads from sending exceptions to threads at lower security levels. MACasync incorporates thread identifiers with values $\mathit{TId}_l\ n$ of the new primitive type $\mathit{TId}\ l$, whose label l represents the static security level of the thread identified by n. Thread identifiers are also *unforgeable* and only generated automatically by the runtime system each time a new thread is forked.

$$\mathit{fork} :: l_{\mathtt{L}} \sqsubseteq l_{\mathtt{H}} \Rightarrow \mathit{MAC}\ l_{\mathtt{H}}\ () \to \mathit{MAC}\ l_{\mathtt{L}}\ (\mathit{TId}\ l_{\mathtt{H}})$$

Figure B.7 extends the sequential calculus of MACasync with concurrency primitives. To simplify our security analysis, term $\mathit{fork}_l\ t$ is annotated with the security label l of thread t of type $\mathit{MAC}\ l\ ()$. Similarly to Vassena, Russo, Buiras, and Waye [Vas+18], we decorate the sequential reduction relation from

above with *events*, which inform the top-level scheduler of the execution of sequential commands that have global effects. For example, event $\mathbf{fork}_l(t)$ indicates that thread t at security level l has been forked and event **step** denotes an uninteresting (*silent*) sequential step. Later, we extend the category of events to keep track of asynchronous exceptions as well. Sequential steps are also parameterized by a *thread id map* ϕ, which represents a source of fresh thread identifiers for each security level. The use of this map is exemplified by rule (Fork). Whenever a new thread is forked, e.g. $fork_l\ t$, we use the label annotation l to generate a fresh identifier $n = \phi(l)$, which is then returned in the monad wrapped in the constructor of thread identifiers, i.e. $TId_l\ n$.

Events:	$e ::= \mathbf{step} \mid \mathbf{fork}_l(t)$
Thread Id:	$n \in \mathbb{N}$
Thread Id Map	$\phi \in \mathit{Label} \to \mathit{Thread\ Id}$
Types:	$\tau ::= \cdots \mid \mathit{TId}\ l$
Values:	$v ::= \cdots \mid \mathit{TId}_l\ n$
Terms:	$t ::= \cdots \mid \mathit{fork}_l\ t$

(Fork)

$$\frac{n = \phi(l)}{\Sigma, \mathit{fork}_l\ t \xrightarrow{\mathbf{fork}_l(t)}_{\phi} \Sigma, \mathit{return}\ (\mathit{TId}_l\ n)}$$

Figure B.7.: Syntax and semantics of *fork*

Figure B.8 introduces the top-level semantics relation that formalizes how concurrent configurations evolve. Concurrent configurations are pairs $\langle \Sigma, \Theta \rangle$ consisting of the concurrent store Σ and a map of thread pools Θ. The thread pool map Θ maps each label of the lattice to the list of threads T_s at that security level, currently in the system. Each rule of the concurrent semantics constructs the source of fresh thread identifiers ϕ from the thread pool map Θ of the initial configuration by means of the function $\mathsf{nextId}(\Theta) = \lambda l.|\Theta(l)|$.

A concurrent step $l, n \vdash c_1 \hookrightarrow c_2$ indicates that configuration c_1 steps to c_2, while running the thread identified (l, n), i.e. the n-th thread of the l-labelled thread pool. The particular scheduler used to determine which thread runs at every step is not very relevant for our discussion, therefore we omit it in our semantics. It suffices to say that the security guarantees of MAC$_{\text{Async}}$ carry over for a wide range of deterministic schedulers [Vas+18] (as witnessed by our mechanized proofs) and include the Round Robin scheduler adopted in Concurrent Haskell. The concurrent rules rely on sequential

Configuration: $C ::= \langle \Sigma, \Theta \rangle$
Thread Pool Map: $\Theta \in \mathit{Label} \to \mathit{Thread\ Pool}$
Thread Pool: $T_s ::= [\,] \mid (th, T_s)$
Thread State: $th ::= t$

$$\frac{\phi = \mathsf{nextId}(\Theta) \qquad \Sigma_1, t_1 \xrightarrow{\mathbf{step}}_{\phi} \Sigma_2, t_2}{l, n \vdash \langle \Sigma_1, \Theta[(l,n) \mapsto t_1] \rangle \hookrightarrow \langle \Sigma_2, \Theta[(l,n) \mapsto t_2] \rangle} \text{(Seq)}$$

$$\frac{\begin{array}{c}\phi = \mathsf{nextId}(\Theta_1) \\ n' = \phi(l') \qquad \Theta_2 = \Theta_1[(l,n) \mapsto t_2] \qquad \Sigma, t_1 \xrightarrow{\mathbf{fork}_{l'}(t)}_{\phi} \Sigma, t_2\end{array}}{l, n \vdash \langle \Sigma, \Theta_1[(l,n) \mapsto t_1] \rangle \hookrightarrow \langle \Sigma, \Theta_2[(l',n') \mapsto t] \rangle} \text{(Fork)}$$

Figure B.8.: Syntax and semantics of concurrent MACasync

events to determine which step to take. For example, rule (Seq) extracts the running thread from the thread pool, i.e. $\Theta[(l,n) \mapsto t_1]$, which steps *silently*, i.e. generating event **step**, and thus the rule only reinserts the thread term in the thread pool, i.e. $\Theta[(l,n) \mapsto t_2]$. In contrast, event $\mathbf{fork}_{l'}(t)$ in rule (Fork) indicates that the running thread has forked, therefore the rule reinserts the parent thread in the pool, i.e. $\Theta_2 = \Theta_1[(l,n) \mapsto t_2]$, and also adds its child at the corresponding security level l' and fresh index $n' = \phi(l')$, i.e. $\Theta_2[(l',n') \mapsto t]$.

B.5. Asynchronous Exceptions

MACasync supports sending and handling asynchronous exceptions by means of two new primitives *throwTo* and *catch*, see Figure B.9. Primitive *throwTo* $t\ \xi$ raises the exception ξ of abstract type χ *asynchronously* in the thread with identifier t. Intuitively, this operation constitutes a write effect, therefore MACasync restricts its API according to the *no write-down* rule to enforce security. To this end, the API ensures that the security label of the receiver thread ($l_{\mathtt{H}}$) is at least as sensitive as the label of the sender ($l_{\mathtt{L}}$) through the type constraint $l_{\mathtt{L}} \sqsubseteq l_{\mathtt{H}}$. Once delivered and raised, asynchronous exceptions behave like synchronous exceptions. They disrupt the execution of the receiving thread in the usual way, with the exception bubbling up in the code of the thread and, if uncaught, eventually crashing it. Threads can recover from exceptions

by wrapping regions of code in a *catch* block. The same mechanism, allows threads to *react* to asynchronous signals by handling exceptions appropriately. Primitive *catch t hs* takes as a parameter a computation t and a list containing pairs of exceptions and handlers. Then, if an exception ξ is raised during the execution of t, the handler corresponding to the first exception *matching* ξ in the list *hs*, if there is one, gets executed.

$$\begin{aligned} &\mathit{throwTo} :: l_{\mathtt{L}} \sqsubseteq l_{\mathtt{H}} \Rightarrow \mathit{TId}\ l_{\mathtt{H}} \to \chi \to \mathit{MAC}\ l_{\mathtt{L}}\ () \\ &\mathit{catch} \quad\ :: \mathit{MAC}\ l\ \tau \to [(\chi, \mathit{MAC}\ l\ \tau)] \to \mathit{MAC}\ l\ \tau \end{aligned}$$

Figure B.9.: MACasync API for asynchronous exceptions

Figure B.10 extends the calculus with value *raise* ξ, which indicates that the computation is in an *exceptional state*, and a new event $\mathbf{throw}_l(\xi, n)$, which instructs the runtime to deliver exception ξ to the thread identified by (l, n). To model how asynchronous exceptions propagate precisely, we add new rules both to the sequential and concurrent semantics. Rule (ThrowTo$_1$) evaluates the thread identifier in term *throwTo* t_1 ξ, which reduces to *throwTo* t_2 ξ through the pure step $t_1 \rightsquigarrow t_2$. (For simplicity, our model assumes that exceptions are already evaluated in terms, thus the rules do not need to reduce them). When the thread identifier is fully evaluated, i.e. it is of the form $\mathit{TId}_l\ n$, rule (ThrowTo$_2$) generates event $\mathbf{throw}_l(\xi, n)$ and returns unit. The rule reflects the *nonblocking* behaviour of *throwTo*, which always succeeds as soon as the thread identifier is evaluated and regardless of the state of the receiving thread. This design decision has important security implications that we discuss further in Section B.5.3. Rule (Catch$_1$) executes the computation t_1 in term *catch* t_1 *hs*. If during the execution of t_1 the computation receives some exception ξ, and the exception propagates up to the exception handler, then the term reduces to *catch* (*raise* ξ) *hs* and rules (Catch$_{\xi 1}$) and (Catch$_{\xi 2}$) determine whether the exception gets handled or not. In these rules, function $\mathsf{first}(hs, \xi)$ searches for a handler corresponding to exception ξ in the list *hs*. To do so, the function traverses the list of exception-handler pairs *hs* left-to-right until it finds a pair whose left component is equal to exception ξ. If a handler for that exception is in the list, i.e. $h = \mathsf{first}(hs, \xi)$, then (Catch$_{\xi 1}$) passes control to it. If no handler matches the exception, i.e. $\emptyset = \mathsf{first}(hs, \xi)$, then rule (Catch$_{\xi 2}$) simply propagates the exception.

$$
\begin{array}{lll}
\text{Exceptions:} & \xi & \\
\text{Events:} & e & ::= \cdots \mid \mathbf{throw}_l(\xi, n) \\
\text{Values:} & v & ::= \cdots \mid \mathit{raise}\ \xi \\
\text{Handlers:} & \mathit{hs} & ::= [\,] \mid (\xi, t) : \mathit{hs} \\
\text{Terms:} & t & ::= \cdots \mid \mathit{throwTo}\ t\ \xi \mid \mathit{catch}\ t\ \mathit{hs}
\end{array}
$$

$$
(\textsc{ThrowTo}_1)\quad \frac{t_1 \rightsquigarrow t_2}{\Sigma, \mathit{throwTo}\ t_1\ \xi \xrightarrow{\mathbf{step}}_\phi \Sigma, \mathit{throwTo}\ t_2\ \xi}
$$

$$
(\textsc{ThrowTo}_2)\quad \frac{}{\Sigma, \mathit{throwTo}\ (\mathit{TId}_l\ n)\ \xi \xrightarrow{\mathbf{throw}_l(\xi,n)}_\phi \Sigma, \mathit{return}\ ()}
$$

$$
(\textsc{Catch}_1)\quad \frac{\Sigma_1, t_1 \xrightarrow{e}_\phi \Sigma_2, t_2}{\Sigma_1, \mathit{catch}\ t_1\ \mathit{hs} \xrightarrow{e}_\phi \Sigma_2, \mathit{catch}\ t_2\ \mathit{hs}}
\qquad
(\textsc{Catch}_{\xi 1})\quad \frac{h = \mathsf{first}(\mathit{hs}, \xi)}{\Sigma, \mathit{catch}\ (\mathit{raise}\ \xi)\ \mathit{hs} \xrightarrow{\mathbf{step}}_\phi \Sigma, h}
$$

$$
(\textsc{Catch}_{\xi 2})\quad \frac{\emptyset = \mathsf{first}(\mathit{hs}, \xi)}{\Sigma, \mathit{catch}\ (\mathit{raise}\ \xi)\ \mathit{hs} \xrightarrow{\mathbf{step}}_\phi \Sigma, \mathit{raise}\ \xi}
$$

Figure B.10.: Syntax and semantics for asynchronous exceptions

B.5.1. Masking Exceptions

Asynchronous exceptions are typically sent in response to external events such as user interrupts and exceeding resource limits. These exceptions can disrupt threads unpredictably, at any moment during their execution, and end up breaking code invariants and leaving shared data structures in an inconsistent state. For example, an incoming exception may crash a thread inside a critical section and cause it to hold a lock indefinitely, without the possibility of cleaning up. Therefore, writing robust code in the presence of asynchronous exceptions requires a mechanism to *temporarily* suppress exceptions in critical sections that must not be interrupted. Inspired by Marlow, Jones, Moran, and Reppy [Mar+01], MACasync sports two *scoped* combinators, *mask* and *unamsk*, to

disable and enable specific exceptions in a given code region, respectively.[7]

$$\begin{aligned} &\mathit{mask} &&:: \chi \to \mathit{MAC}\ l\ \tau \to \mathit{MAC}\ l\ \tau \\ &\mathit{unmask} &&:: \chi \to \mathit{MAC}\ l\ \tau \to \mathit{MAC}\ l\ \tau \end{aligned}$$

Intuitively, primitive $\mathit{mask}\ \xi\ t$ runs computation t with exceptions ξ disabled. If such an exception is received during the execution of t, the exception is *not* dropped, but stored in a buffer of pending exceptions ξ_s and raised once the execution goes past the *mask* instruction. Term $\mathit{unmask}\ \xi\ t$ works the other way around and enables exceptions ξ while executing t. In general, whether an exception received by a thread should be raised immediately or temporarily suppressed depends on the *masking context* of the thread. Intuitively, the masking context at each execution point depends on the (nested) *mask* and *unmask* instructions crossed up to that point. For instance, if program $\mathit{unmask}\ \xi\ (\mathit{mask}\ \xi'\ t)$ receives exception ξ while executing t, and if $\xi \not\equiv \xi'$ and t does not contain any $\mathit{mask}\ \xi$ instruction, then the exception gets raised, i.e. $\mathit{unmask}\ \xi\ (\mathit{mask}\ \xi'\ (\cdots\ \mathit{raise}\ \xi\ \cdots))$.

Figure B.11 presents the sequential semantics of *mask* and *unmask*. The masking context M is a map from exceptions to Booleans, representing a bit vector that indicates which exceptions can be raised in the reduction steps. To keep track of exceptions, the sequential relation carries the list of pending exceptions ξ_s, on the left, and the list of remaining exceptions ξ_s' on the right of the arrow. Further, the arrow is annotated with the masking context of the thread (M). Rules (Mask) and (Unmask) modify the masking context accordingly via functions $\mathsf{mask}(M, \xi) = \lambda\xi'.\xi \equiv \xi' \ \vee\ M(\xi')$ and $\mathsf{unmask}(M, \xi)$ (analogous). In particular, the rules reduce term $\mathit{mask}\ \xi\ t$ (respectively $\mathit{unmask}\ \xi\ t$) by executing term t with modified mask M_2 obtained from disabling (enabling) exception ξ in the current mask M_1, i.e. $M_2 = \mathsf{mask}(M_1, \xi)$ ($M_2 = \mathsf{unmask}(M_1, \xi)$). When a nested, *masked* computation has completed, either successfully ($\mathit{return}\ t$) or not ($\mathit{raise}\ \xi'$), rules (Mask_1) and (Mask_ξ) simply propagate the result.

The masking context M and the list of pending exceptions ξ_s determine whether any exception in the list should be raised or not. To reflect that, we need to adapt the semantics rules for most constructs of the calculus. Figure B.12 shows the modifications for the monadic bind ($\gg\!=$). (The rules for the other constructs are modified in a similar way, we refer the reader to the Agda mechanization for details).

[7] Even though these primitives take only a single exception as an argument, they are equivalent to the multi-exception variants used in Section B.3, i.e. $\mathit{mask}\ [\xi_1, \xi_2]\ t$ behaves exactly like $\mathit{mask}\ \xi_1\ (\mathit{mask}\ \xi_2\ t)$.

Mask: $M \in \chi \to \mathit{Bool}$
Terms: $t ::= \cdots \mid \mathit{mask}\ \xi\ t \mid \mathit{unmask}\ \xi\ t$
Exception list: $\xi_s ::= [\,] \mid (\xi : \xi_s)$

(Mask)
$$\frac{M_2 = \mathsf{mask}(M_1, \xi) \qquad \Sigma_1, t_1, \xi_s \xrightarrow{e}_{(\phi, M_2)} \Sigma_2, t_2\ \xi_s'}{\Sigma_1, \mathit{mask}\ \xi\ t_1, \xi_s \xrightarrow{e}_{(\phi, M_1)} \Sigma_2, \mathit{mask}\ \xi\ t_2, \xi_s'}$$

(Unmask)
$$\frac{M_2 = \mathsf{unmask}(M_1, \xi) \qquad \Sigma_1, t_1, \xi_s \xrightarrow{e}_{(\phi, M_2)} \Sigma_2, t_2, \xi_s'}{\Sigma_1, \mathit{unmask}\ \xi\ t_1, \xi_s \xrightarrow{e}_{(\phi, M_1)} \Sigma_2, \mathit{unmask}\ \xi\ t_2, \xi_s'}$$

(Mask$_1$)
$$\Sigma, \mathit{mask}\ \xi\ (\mathit{return}\ t), \xi_s \xrightarrow{\mathbf{step}}_{(\phi, M)} \Sigma, \mathit{return}\ t, \xi_s$$

(Mask$_\xi$)
$$\Sigma, \mathit{mask}\ \xi\ (\mathit{raise}\ \xi'), \xi_s \xrightarrow{\mathbf{step}}_{(\phi, M)} \Sigma, \mathit{raise}\ \xi', \xi_s$$

Figure B.11.: Syntax and semantics of *mask* (*unmask* is similar)

First, we define function $\mathsf{unmasked}_{\xi_s'}(M, \xi_s)$, which extracts from the list of pending exceptions ξ_s the *first* exception ξ that is *unmasked* in M, i.e. such that $\neg\, M(\xi)$, The function walks down the list recursively and accumulates the exceptions ξ that are *masked* in M, i.e. such that $M(\xi)$, in the list ξ_s', which is then returned together with the rest of the list ξ_s, when an unmasked exception is found. If all the exceptions in the list are masked, the function simply returns $\emptyset$.

$$\mathsf{unmasked}_{\xi_s'}(M, \xi_s) = \begin{cases} \emptyset & \text{if } \xi_s = [\,] \\ (\xi, \xi_s' +\!\!+ \xi_s'') & \text{if } \xi_s = \xi : \xi_s'' \text{ and } \neg\, M(\xi) \\ \mathsf{unmasked}_{(\xi_s' +\!\!+ [\xi])}(M, \xi_s'') & \text{if } \xi_s = \xi : \xi_s'' \text{ and } M(\xi) \end{cases}$$

In the *elimination* rules of the semantics, we apply function $\mathsf{unmasked}_{[\,]}(M, \xi_s)$ to determine whether a pending exception should be raised. For example, if all exceptions are masked, i.e. $\mathsf{unmasked}_{[\,]}(M, \xi_s) = \emptyset$, then rule (Bind$_2$) steps as usual. In contrast, if an unmasked exception is pending, i.e. $\mathsf{unmasked}_{[\,]}(M, \xi_s) =$

$$\frac{\mathsf{unmasked}_{[]}(M, \xi_s) = (\xi, \xi_s')}{\Sigma, \mathit{return}\ t_1 \gg\!= t_2, \xi_s \xrightarrow{\mathbf{step}}_{(\phi,M)} \Sigma, \mathit{raise}\ \xi, \xi_s'} \quad (\textsc{Bind-Raise})$$

$$\frac{\mathsf{unmasked}_{[]}(M, \xi_s) = \emptyset}{\Sigma, \mathit{return}\ t_1 \gg\!= t_2, \xi_s \xrightarrow{\mathbf{step}}_{(\phi,M)} \Sigma, t_2\ t_1, \xi_s} \quad (\textsc{Bind}_2)$$

$$\Sigma, \mathit{raise}\ \xi \gg\!= t, \xi_s \xrightarrow{\mathbf{step}}_{(\phi,M)} \Sigma, \mathit{raise}\ \xi, \xi_s \quad (\textsc{Bind}_\xi)$$

Figure B.12.: *Masking* semantics of bind ($\gg\!=$)

(ξ, ξ_s'), rule (Bind-Raise) raises it, i.e. *raise* ξ, and the thread steps with buffer ξ_s' where exception ξ has been removed.

Once raised, exceptions propagate *unconditionally* via rule ($\textsc{Bind}_\xi$), i.e. no further exceptions are raised until the current one is handled.

B.5.2. Concurrency and Synchronization Variables

The modifications needed for supporting asynchronous exceptions in the concurrent semantics are minimal. Figure B.14 extends the thread state *th* with the list of pending exceptions ξ_s and the initial masking context M. When a thread forks, the child thread *inherits* the masking context of the parent thread and runs with an initially empty list of exceptions. New rule (Throw) processes event $\mathbf{throw}_{l'}(\xi, n')$ by delivering exception ξ to the thread (t, ξ_s', M') identified by (l', n'). Since exceptions are processed in the same order as they are delivered, the rule inserts ξ at the end of the buffer ξ_s', i.e. $\xi_s' +\!\!+ [\xi]$.

Next, we introduce new rules that capture precisely the interaction between synchronization variables and asynchronous exceptions. As we explained before, Concurrent Haskell by design allows specific *blocking* operations to be *interrupted* by asynchronous exceptions, even if they are masked [Mar+01]. Therefore, our semantics resumes threads stuck on synchronization variables if *any* exception is pending. The rules in Figure B.13 formalize this requirement for primitive *takeMVar*, the rules for *putMVar* are symmetric. Rule ($\textsc{Take}_1$) covers the case where no unmasked exception is pending, i.e. $\mathsf{unmasked}_{[]}(M, \xi_s) = \emptyset$, and the thread can step because the variable is full, i.e. $(l, n) \mapsto (\!|\, t \,|\!) \in \Sigma$, and

$$\frac{(l,n) \mapsto (\!|\, t \,|\!) \;\in\; \Sigma \qquad \mathsf{unmasked}_{[]}(M, \xi_s) = \emptyset \qquad \Sigma' = \Sigma[(l,n) \mapsto \otimes]}{\Sigma, \mathit{takeMVar}\ (\mathit{MVar}_l\ n), \xi_s \xrightarrow{\mathbf{step}}_{(\phi,M)} \Sigma', \mathit{return}\ t, \xi_s} \quad (\textsc{Take}_1)$$

$$\frac{(l,n) \mapsto c \;\in\; \Sigma \qquad \mathsf{unmasked}_{[]}(M, \xi_s) = (\xi, \xi_s')}{\Sigma, \mathit{takeMVar}\ (\mathit{MVar}_l\ n), \xi_s \xrightarrow{\mathbf{step}}_{(\phi,M)} \Sigma, \mathit{raise}\ \xi, \xi_s'} \quad (\textsc{Take-Raise-Unmasked})$$

$$\frac{(l,n) \mapsto \otimes \;\in\; \Sigma \qquad \mathsf{unmasked}_{[]}(M, \xi_s) = \emptyset \qquad \xi_s = \xi : \xi_s'}{\Sigma, \mathit{takeMVar}\ (\mathit{MVar}_l\ n), \xi_s \xrightarrow{\mathbf{step}}_{(\phi,M)} \Sigma, \mathit{raise}\ \xi, \xi_s'} \quad (\textsc{Take-Raise-Masked})$$

Figure B.13.: Synchronization variables and exceptions

Thread: $th ::= (t, \xi_s, M)$

$$\frac{\begin{array}{c}\phi = \mathsf{nextId}(\Theta_1) \qquad \Sigma, t_1, \xi_s \xrightarrow{\mathbf{throw}_{l'}(\xi,n')}_{(\phi,M)} \Sigma, t_2, \xi_s \\ \Theta_2 = \Theta_1[(l,n) \mapsto (t_2, \xi_s, M)] \qquad (l',n') \mapsto (t, \xi_s', M') \;\in\; \Theta_2\end{array}}{l, n \vdash \langle \Sigma, \Theta_1[(l,n) \mapsto (t_1, \xi_s, M)] \rangle \hookrightarrow \langle \Sigma, \Theta_2[(l',n') \mapsto (t, \xi_s' +\!\!+ [\xi], M')] \rangle} \quad (\textsc{Throw})$$

Figure B.14.: Extended concurrent semantics for asynchronous exceptions

thus the rule returns its content t and empties the variable, i.e. $\Sigma[(l,n) \mapsto \otimes]$. On the other hand, in rule (Take-Raise-Unmasked), an unmasked exception is pending, i.e. $\mathsf{unmasked}_{[]}(M, \xi_s) = (\xi, \xi_s')$, thus, regardless of the variable being full or empty, i.e. $(l,n) \mapsto c \;\in\; \Sigma$, the rule aborts the computation and raises the exception ξ without modifying the store. Lastly, in rule (Take-Raise-Masked), the variable is *empty*, i.e. $(l,n) \mapsto \otimes \;\in\; \Sigma$, and the thread should block and get stuck. However, an exception ξ is pending in the buffer ξ_s, i.e. $\xi_s = \xi : \xi_s'$, therefore—*regardless* of the masking context M—the thread is resumed by raising exception ξ. In the rule, the condition $\mathsf{unmasked}_{[]}(M, \xi_s) = \emptyset$ reveals that the pending exception that gets raised is *masked* and ensures that the semantics is *deterministic*. Without this premise, a thread with both unmasked and masked exceptions pending in its buffer could

either step via rule (Take-Raise-Unmasked) and raise the first unmasked exception, or via rule (Take-Raise-Masked) and raise the first masked exception. The condition above removes the nondeterminism: if both masked and unmasked exceptions are pending, the first *unmasked* exception is raised via rule (Take-Raise-Unmasked).

B.5.3. Design Choices and Security

In this part we motivate some of the design choices that are key to the security guarantees of MACasync and that, we believe, can help programmers to write code that is more robust to asynchronous exceptions.

API of *throwTo* The type of *throwTo* (Figure B.9) restricts how threads are permitted to communicate asynchronously with each other to enforce security. Imagine an unrestricted version of *throwTo* called $\mathit{throwTo}^{\mathbf{leaky}}$, which—in clear violation of the *no write-down* security rule—allows secret threads to send exceptions to public threads. If MACasync exposed this leaky primitive, then an attacker could exploit it to leak secret data to a public thread through classic *implicit flows* attacks:

$$
\begin{array}{l}
\mathbf{do}\ \mathit{tid}_{\mathtt{L}} \leftarrow \mathit{fork}_{\mathtt{L}}\ (\mathbf{do}\ \mathit{catch}\ \mathit{loop}\ [(\xi, \mathit{print}_{\mathtt{L}}\ 1)]) \\
\quad \_ \quad \leftarrow \mathit{fork}_{\mathtt{H}}\ (\mathbf{do}\ s \leftarrow \mathit{unlabel}\ \mathit{secret} \\
\qquad\qquad\qquad\qquad \mathbf{if}\ s\ \ \mathbf{then}\ \mathit{throwTo}^{\mathbf{leaky}}\ \mathit{tid}_{\mathtt{L}}\ \xi \\
\qquad\qquad\qquad\qquad\quad \mathbf{else}\ \ \mathit{return}\ ())
\end{array}
$$

The code above forks two threads, a public thread that waits for an asynchronous exception in a loop, and a secret thread that branches on secret data and sends an exception to the public thread in one branch. Since the secret thread sends an exception to the public thread only when the secret is true, the attacker can easily learn its value by simply monitoring the public output of the public thread, which prints 1 only when an exception is raised. MACasync rejects such attacks by statically enforcing the no write-down rule in the API of *throwTo*, which would make the code above ill-typed.

Asynchronous *throwTo* In MACasync, primitive *throwTo* is itself an asynchronous operation. As rule (ThrowTo$_2$) in Figure B.10 shows, primitive *throwTo* always returns immediately, without waiting for the receiver thread to raise the exception. This design choice follows a previous line of work on asynchronous exceptions for Haskell [Mar+01], where the authors argue that the asynchronous semantics is easier to implement. Maybe surprisingly, the

current implementation of Concurrent Haskell with asynchronous exception in the GHC runtime provides only a *synchronous* version of *throwTo*.[8] Would MACasync be secure with a synchronous primitive $throwTo^{\mathbf{sync}}$? No, unfortunately the possibility of two threads synchronizing by throwing exceptions opens a new covert channel. Consider the following example, where $throwTo^{\mathbf{sync}}$ has synchronous semantics, i.e. $throwTo^{\mathbf{sync}}$ *blocks* the sender thread until the exception is raised in the receiver thread.

$$
\begin{array}{l}
\textbf{do } tid_{\mathtt{H}} \leftarrow fork_{\mathtt{H}}\ (\textbf{do } s \leftarrow unlabel\ secret \\
\qquad\qquad\qquad\qquad\quad \textbf{if } s \textbf{ then } mask\ \xi\ loop \textbf{ else } loop) \\
\quad no\_ops \\
\quad throwTo^{\mathbf{sync}}\ tid_{\mathtt{H}}\ \xi \\
\quad print_{\mathtt{L}}\ 0
\end{array}
$$

In the code above, the main public thread forks a secret thread, which branches on secret data and in one branch enters the masked block $mask\ \xi\ loop$. After waiting for a sufficient amount of time through *no_ops*, the public thread sends exception ξ *synchronously* to the secret thread. If the secret thread is looping in the masked block, the exception ξ will never be raised, causing the public thread to block forever on $throwTo^{\mathbf{sync}}$ and thus suppressing the final public output $print_{\mathtt{L}}\ 0$. Then, the attacker can learn the value of the secret by simply observing (the lack of) the output 0 on the public channel.

As discussed in Section B.2 for *MVar*, synchronous communication primitives perform both read *and* write side-effects, therefore $throwTo^{\mathbf{sync}}$ cannot operate securely between threads at different security levels. Even though Concurrent Haskell provides only the equivalent of $throwTo^{\mathbf{sync}}$, we can still derive a secure asynchronous implementation for *throwTo* by internally forking an isolated thread that calls the unsafe $throwTo^{\mathbf{sync}}$, i.e. we define $throwTo\ t\ \xi$ as $fork\ (throwTo^{\mathbf{sync}}\ t\ \xi\ \gg return\ ())$.

Reliable Exception Delivery In MACasync, threads store the received exceptions in the buffer where they remain until raised. Importantly, the exceptions are raised following the order in which they have been delivered, thus enabling threads to react to signals in the same order as they arise. Even though multiple exceptions can be pending in the buffer, our semantics ensures that new exceptions are not raised while the thread is in an exceptional state. This choice eliminates, by design, the risk of multiple simultaneous exceptions disrupting critical code in unpredictable ways. Once handled via the matching exception

[8] `https://hackage.haskell.org/package/base-4.12.0.0/docs/Control-Exception.html#v:throwTo`

handler, the code resumes normal execution and any other pending exception may be raised. This ensures that all remaining exceptions, if not masked, will eventually be raised.

B.5.4. Relation to MAC

MACasync extends MAC with asynchronous exceptions [Vas+18]. MAC features exception-handling primitives and classic exceptions, but these operate within individual threads and are intended to signal and recover from exceptional conditions arising only *internally*, due to the current state of the computation. Asynchronous exceptions are more expressive than regular exceptions. In addition to signaling (external) exceptional conditions, they enable a flexible signal-based communication mechanism. In MAC, threads can communicate with each other only synchronously and *indirectly*, through synchronization variables. Though possible, this communication mechanism is too cumbersome to use as it would require programmers to establish an appropriate communication protocol and change their code heavily, for example to ensure that all threads that need communicating share the same synchronization variable. Even worse, communication in this style is limited between threads at the same security level. In contrast, threads in MACasync can communicate *directly*, by sending exceptions to the identifier of the intended receiver thread, and also to threads at a different, more sensitive security level. MACasync leverages the mechanization of MAC in its security proofs. Modelling the semantics of asynchronous exceptions presented in this paper required substantial changes to the existing artifact. These changes include extending the syntax and semantics of the previous model with our new primitives, as well as carefully adapting the existing semantics rules to capture the semantics of *interruptible exceptions*.

B.6. Security Guarantees

This section shows that MACasync satisfies progress-sensitive noninterference (PSNI). We begin by describing our proof technique based on *term erasure*. Then, we present two lemmas that are key to the progress-sensitive guarantees of the calculus and sketch the noninterference proof. We refer the interested reader to the Agda mechanization for detailed proofs.

B.6.1. Term Erasure

Term erasure is a widely used technique to prove noninterference properties of information-flow control (IFC) languages (e.g. [LZ10; Ste+11b; Ste+12;

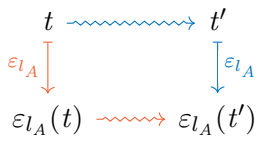


Figure B.15.: Single-step simulation

Heu+15; BVR15; VR16; Vas+18]). The technique takes its name from the *erasure function*, which removes secret data *syntactically* from program terms. To this end, the erasure function, written $\varepsilon_{l_A}(t)$, rewrites the subterms of t above the attacker's security level l_A to special term $\bullet$, which only reduces to itself. Once this function is defined, the technique relies on establishing a core property, a *simulation* between the execution of terms (and later configurations as well) and their erased counterpart. The simulation diagram in Figure B.15 illustrates this property for pure terms. The diagram shows that erasing the confidential parts of term t and then reducing the erased term $\varepsilon_{l_A}(t)$ along the orange path leads to the same term $\varepsilon_{l_A}(t')$ obtained along the blue path by first stepping from term t to term t' and then applying erasure, i.e. the diagram *commutes*. Intuitively, if term t leaked while stepping to t', then some data above security level l_A would remain in the erased term $\varepsilon_{l_A}(t')$, but it would be erased along the other path, in which t is first erased and then reduced, and thus the diagram would not commute.

B.6.2. Erasure Function

We define the erasure function for terms in Figure B.16. Since the sensitivity of many terms is determined by their type, the definition of the erasure function is type driven, i.e. we write $\varepsilon_{l_A}(t :: \tau)$ for the erasure of term t of type τ. (We omit the type of the term when it is irrelevant). Ground values are unaffected by the erasure function, e.g. $\varepsilon_{l_A}(()) = ()$, and most terms are erased homomorphically, e.g. $\varepsilon_{l_A}(t_1\ t_2) = \varepsilon_{l_A}(t_1)\ \varepsilon_{l_A}(t_2)$. The content of *secret* labelled values is removed, i.e. $\varepsilon_{l_A}(\mathit{Labeled}\ t{::}\mathit{Labeled}\ l\ t) = \mathit{Labeled}\ \bullet$ if $l \not\sqsubseteq l_A$, or erased homomorphically otherwise, i.e. $\varepsilon_{l_A}(\mathit{Labeled}\ t :: \mathit{Labeled}\ l\ t) = \mathit{Labeled}\ \varepsilon_{l_A}(t : \tau)$ if $l \sqsubseteq l_A$. Notice that terms of type $\mathit{MAC}\ l\ \tau$ (e.g. *mask*, *unmask*) are also erased homomorphically, despite the fact that the computation may be sensitive, i.e. even if $l \not\sqsubseteq l_A$. (The special erasure for primitive *throwTo* is explained below). Should not erasure rewrite these constructs to $\bullet$? Intuitively, these terms represent a *description* of a sensitive computation, which cannot leak data until it is inserted in a sequential configuration and executed. Since these

terms can only execute when fetched from a thread pool, it is then sufficient to erase thread pools appropriately.

We define the erasure function for configurations, stores, thread pools, and thread states in Figure B.17. Configurations are erased component-wise, i.e. $\varepsilon_{l_A}(\langle \Sigma, \Theta \rangle) = \langle \varepsilon_{l_A}(\Sigma), \varepsilon_{l_A}(\Theta) \rangle$. Thread pools Θ containing secret threads are entirely removed by the erasure function, i.e. $\varepsilon_{l_A}(\Theta)(l) = \bullet$ if $l \not\sqsubseteq l_A$, while those containing thread pools are erased homomorphically, i.e. $\varepsilon_{l_A}(\Theta)(l) = \varepsilon_{l_A}(\Theta(l))$ if $l \sqsubseteq l_A$, where $\varepsilon_{l_A}([\,]) = [\,]$ and $\varepsilon_{l_A}(th, T_s) = (\varepsilon_{l_A}(th), \varepsilon_{l_A}(T_s))$. (The erasure function for memory stores and segments is similar). When some secret thread gets scheduled from an erased thread pool $\bullet$, a *dummy* thread $(\bullet, [\,], \lambda\_.\mathit{False})$ runs instead and simply loops. However, rewriting secret thread pools to $\bullet$ can disrupt operations involving thread identifiers. For example, an erased public thread using primitive *throwTo* to communicate with a secret thread gets *stuck*, since rule (Throw) would try to deliver an exception into thread pool $\bullet$. To recover from this situation, we apply the *two-step erasure* technique [VR16]. This technique rewrites problematic terms, e.g. *throwTo*, to special, $\bullet$-annotated erased terms added to the calculus, i.e. $\mathit{throwTo}_\bullet$. The semantics of these *new* terms is then defined precisely to re-establish the core simulation property fundamental for security (Figure B.15). For example, term $\mathit{throwTo}_\bullet\ t\ \xi$ reduces just like *throwTo* in rules (ThrowTo$_1$) and (ThrowTo$_2$), thus respecting the simulation property of the sequential semantics. However, instead of generating a regular event $\mathbf{throw}_{l_H}(\xi, n)$, which would get the concurrent configuration stuck in rule (Throw), it generates a new event $\mathbf{throw}_{\bullet l_H}(\xi)$. Similarly, this event is handled by a new rule of the concurrent semantics, which simply drops the exception (no thread labelled $l_H \not\sqsubseteq l_A$ can receive it), thus completing the simulation diagram of the concurrent step (Throw).

B.6.3. Progress-Sensitive Noninterference

The proof of progress-sensitive noninterference (PSNI) builds on two key properties of the concurrent relation: *deterministic reduction* and *erased simulation*.

Lemma B.6.1 (Deterministic Reduction)**.** *If* $c_1 \hookrightarrow c_2$ *and* $c_1 \hookrightarrow c_3$*, then* $c_2 \equiv c_3$*.*

The symbol $\equiv$ above denotes syntactic equality up to α-renaming, in our mechanized proofs we use De Bruijn indexes and syntactic equality. Determinism of the concurrent semantics is important for security, because it eliminates scheduler refinement attacks [RS06].

The second lemma that we prove relates the reduction step of a thread in the concurrent semantics with the corresponding erased thread. If the security level

$$
\begin{aligned}
&\varepsilon_{l_A}(()) = () \\
&\varepsilon_{l_A}(t_1\ t_2) = \varepsilon_{l_A}(t_1)\ \ \varepsilon_{l_A}(t_2) \\
&\varepsilon_{l_A}(\mathit{Labeled}\ t :: \mathit{Labeled}\ l\ t) \\
&\qquad = \begin{cases} \mathit{Labeled}\ \varepsilon_{l_A}(t) & \text{if } l_{\mathtt{H}} \sqsubseteq l_A \\ \mathit{Labeled}\ \bullet & \text{otherwise} \end{cases} \\
&\varepsilon_{l_A}(\mathit{mask}\ \xi\ t) = \mathit{mask}\ \xi\ \ \varepsilon_{l_A}(t) \\
&\varepsilon_{l_A}(\mathit{unmask}\ \xi\ t) = \mathit{unmask}\ \xi\ \ \varepsilon_{l_A}(t) \\
&\varepsilon_{l_A}(\mathit{throwTo}\ (t :: \mathit{TId}\ l_{\mathtt{H}})\ \xi :: \mathit{MAC}\ l_{\mathtt{L}}\ ()) \\
&\qquad = \begin{cases} \mathit{throwTo}\ \varepsilon_{l_A}(t)\ \ \xi & \text{if } l_{\mathtt{H}} \sqsubseteq l_A \\ \mathit{throwTo}_{\bullet}\ \varepsilon_{l_A}(t)\ \ \xi & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure B.16.: Erasure of terms (excerpts)

l of the thread is below the level of the attacker, i.e. $l \sqsubseteq l_A$, then we construct a simulation diagram similar to that of Figure B.15, but for concurrent steps. Instead, if the security level of the thread is *not* observable by the attacker, i.e. $l \not\sqsubseteq l_A$, then the configurations before and after the step are *indistinguishable* to the attacker. This indistinguishability relation is called l_A-equivalence, written $c_1 \approx_{l_A} c_2$, and defined as the kernel of the erasure function (Figure B.17), i.e. $\varepsilon_{l_A}(c_1) \equiv \varepsilon_{l_A}(c_2)$.

Lemma B.6.2 (Erased Simulation)**.** *Given a concurrent reduction step* $l, n \vdash c_1 \hookrightarrow c_1'$ *then*

- $l, n \vdash \varepsilon_{l_A}(c_1) \hookrightarrow \varepsilon_{l_A}(c_1')$*, if* $l \sqsubseteq l_A$*, or*
- $c_1 \approx_{l_A} c_1'$*, if* $l \not\sqsubseteq l_A$

Using Lemmas B.6.1 and B.6.2, we prove PSNI, where symbol $\hookrightarrow^*$ denotes the transitive reflexive closure of $\hookrightarrow$ as usual.

Theorem B.6.1 (Progress-Sensitive Noninterference)**.** *Given two well-typed concurrent configurations* c_1 *and* c_2*, such that* $c_1 \approx_{l_A} c_2$*, and a reduction step* $l, n \vdash c_1 \hookrightarrow c_1'$*, then there exists a configuration* c_2' *such that* $c_1' \approx_{l_A} c_2'$ *and* $c_2 \hookrightarrow^* c_2'$*.*

Proof. By cases on $l \sqsubseteq l_A$.

$$\varepsilon_{l_A}(\langle \Sigma, \Theta \rangle) = \langle \varepsilon_{l_A}(\Sigma), \varepsilon_{l_A}(\Theta) \rangle$$
$$\varepsilon_{l_A}(\Sigma)(l) = \begin{cases} \varepsilon_{l_A}(S) & \text{if } l \sqsubseteq l_A \text{ and } S = \Sigma(l) \\ \bullet & \text{otherwise} \end{cases}$$
$$\varepsilon_{l_A}(\Theta)(l) = \begin{cases} \varepsilon_{l_A}(T_s) & \text{if } l \sqsubseteq l_A \text{ and } T_s = \Theta(l) \\ \bullet & \text{otherwise} \end{cases}$$
$$\varepsilon_{l_A}((t, \xi_s, M)) = (\varepsilon_{l_A}(t), \xi_s, M)$$

Figure B.17.: Erasure of configurations (excerpts)

If $l \sqsubseteq l_A$ then in the configuration c_2 there is an l_A-equivalent thread identified by (l, n). Before that thread runs, however, there can be a *finite* number of high threads in c_2 scheduled before (l, n). After the high threads run, i.e. $c_2 \hookrightarrow^* c_2''$, for some configuration c_2'', the low thread is scheduled again, i.e. $l, n \vdash c_2'' \hookrightarrow c_2'$, for some other configuration c_2'. From Lemma B.6.2 (*erased simulation*) applied to the first set of steps, we obtain $c_2 \approx_{l_A} c_2''$ (all these steps involve threads above the attacker's level) and then $c_1 \approx_{l_A} c_2''$ follows by transitivity of the l_A-equivalence relation. Then, we apply Lemma B.6.2 again and conclude that $l, n \vdash \varepsilon_{l_A}(c_2'') \hookrightarrow \varepsilon_{l_A}(c_2')$, since $l \sqsubseteq l_A$ as well as $l, n \vdash \varepsilon_{l_A}(c_1) \hookrightarrow \varepsilon_{l_A}(c_1')$. By definition of l_A-equivalence, we derive $\varepsilon_{l_A}(c_1) \equiv \varepsilon_{l_A}(c_2'')$ from $c_1 \approx_{l_A} c_2''$ and from Lemma B.6.1 (*deterministic reduction*) we conclude that $\varepsilon_{l_A}(c_1') \equiv \varepsilon_{l_A}(c_2')$, i.e. $c_1' \approx_{l_A} c_2'$.

If $l \not\sqsubseteq l_A$, then we apply Lemma B.6.2 and obtain $c_1 \approx_{l_A} c_1'$, thus $c_1' \approx_{l_A} c_2'$ for $c_2' = c_2$ by reflexivity and transitivity of l_A-equivalence. □

B.7. Related Work

Asynchronous Exceptions Mechanisms Asynchronous exceptions and signals allow developers to implement key functionalities of real world systems (e.g. speculative computation, timeouts, user interrupts, and enforcing resources bounds) robustly [Mar+01; Mar17]. Surprisingly, support for asynchronous exceptions in concurrent programming languages differ considerably. For example, Java has deprecated fully asynchronous methods to stop, suspend, and resume threads because they can too easily break programs invariants without hope of recovery [Ora20]. Similarly, the interaction between synchronous exceptions and signals makes it hard to write robust signal handlers in Python [FM02]. Lacking robust asynchronous primitives, several programming languages and

operating systems (e.g. JAVA, MODULA3, and POSIX-compliant OS's) rely on *semi-asynchronous* communication as a workaround. With this approach, a thread sends a signal to another by setting special flags that must be polled periodically by the receiver. Even though programming in this model is less convenient, we believe that the principles proposed in this paper could be adapted for semi-asynchronous communication. The Standard ML of New Jersey (SML/NJ) features asynchronous signaling mechanisms based on first-class continuations [Rep90]. When a thread receives a signal, control is passed to the corresponding handler together with the interrupted state of the thread as a continuation. Then, the handler may decide to resume the execution of the interrupted thread or pass control to another thread. Erlang implements a special kind of asynchronous signaling. Threads can monitor each other through *bidirectional links*, which propagate the exit code of a thread to the other [Arm03]. Multicore OCAML support asynchronous exceptions through algebraic effects and effects handlers [Dol+17]. Syme, Petricek, and Lomov [SPL11] extend F# with an asynchronous modality that changes the semantics of control-flow operators to use continuations, thus sparing programmers from writing asynchronous code in continuation-passing style. Bierman, Russo, Mainland, Meijer, and Torgersen [Bie+12] port this approach to C# and additionally formalize it with a direct operational semantics and prove type-safety. Inoue, Aotani, and Igarashi [IAI18] provide interruptible executions in Scala for context-aware (reactive) programming via an embedded domain specific language based on workflows. Concurrent HASKELL supports asynchronous exceptions with scoped (un)masking combinators [Mar+01] and MACASYNC relies on them to provide secure API to untrusted code.

Semantics of Asynchronous Exceptions Jones, Reid, Henderson, Hoare, and Marlow [Jon+99] present a semantics framework for reasoning about the correctness of compiler optimizations in the presence of (imprecise) exceptions for HASKELL. Their framework can capture asynchronous exceptions as well, but it is based on denotational semantics and thus not suitable for reasoning about covert channels. Marlow, Jones, Moran, and Reppy [Mar+01] were the first to develop an operational semantics for asynchronous exceptions, which inspired ours and which we have extended to model fine-grained exception handlers. Their semantics is based on evaluation contexts [FH92], while ours is small-step to leverage the existing formalization and mechanization of MAC [Vas+18; VR16; VBR17]. Hutton and Wright [HW07] study an operational semantics for interrupts in the context of a basic terminating language without concurrency and I/O. Their goal is exploratory: they want to formally justify the source

level semantics with respect to its compilation to a low-level language. Harrison, Allwein, Gill, and Procter [Har+08] identify asynchronous exceptions as a computational effect and formalize them in a modular monadic model.

Covert Channels and Countermeasures Several runtime system features create subtle covert channels that weaken and sometimes completely break the security guarantees of IFC languages. When memory is shared between computations at different security levels, *garbage collection* cycles leak information via timing, even across network connections [PA17]. To close this channel, memory should be partitioned by security level and each memory partition should be managed by an independent timing-sensitive garbage collector (see the garbage collector implemented in Zee for an example [PA19]). *Lazy evaluation* introduces a software level cache in the runtime system which creates an internal timing channel in concurrent Haskell IFC libraries [BR13]. To close this channel, Vassena, Breitner, and Russo [VBR17] design a runtime system primitive that *restricts sharing* between threads at different security levels. The same primitive can close the lazy covert channel in MACasync. General purpose runtime system automatically balance computing resources (CPU time, memory and cores) between running threads to achieve fairness, but, by doing so on multicore systems, they also internalize many external timing covert channels [Vas+19]. LIO_{PAR} is a runtime system design that recovers security in multicore systems by making resource management hierarchical and explicit at the programming language level. Even though in LIO_{PAR} parent threads send asynchronous signals to kill children and reclaim computing resources, LIO_{PAR} does not support fine-grained exception handlers and masking primitives. Language-based predictive mitigation is a general technique to bound the leakage of timing channels (e.g. arising due to hardware caches) in programs [ZAM12]. Thibault and Askarov [TA17] optimize this technique for a sequential programming language with asynchronous I/O, but their approach does not consider concurrency and asynchronous exceptions. Interruptible enclaves have been the target of several interrupt-based attacks [BPS17; He+18; BPS18] and Busi, Noorman, Bulck, Galletta, Degano, Mühlberg, and Piessens [Bus+20] propose *full abstraction* [Aba99] as the desirable security criterion for extending processor with interruptible enclaves securely. Our security criterion (PSNIe) is simpler to prove and aligns with the expected security guarantees of MACasync. Intuitively, Busi, Noorman, Bulck, Galletta, Degano, Mühlberg, and Piessens [Bus+20] prove a more complex criterion because it ensures that the extended processor has no more vulnerabilities than the original, but that does not imply that neither processor satisfies some specific security property.

Secure Runtime Systems and Abstractions Systems that by design run untrusted programs (e.g. mobile code and plugins) must place adequate security mechanisms to impede buggy or malicious code from exhausting all available computing resources. KaffeOS is an extension of the JAVA runtime system that isolates *processes* and manage their computing resources (memory and CPU time) to prevent abuse [BH05]. When a process exceeds its resource budget, KaffeOS kills it and reclaims its resources without affecting the integrity of the system. Cinder is an operating system for mobile devices that provides *reserves* and *taps* abstractions for storing and distributing energy [Roy+11]. Using these abstractions, applications can delegate and subdivide their energy quota while maintaining energy isolation. Yang and Mazières [YM14] extend GHC runtime systems with *resource containers*, an abstraction that enforce dynamic space limits according to an allocator-pays semantics. None of these systems enforce information flow policies except for Cinder, but we believe that secure API for asynchronous exceptions like those of MACASYNC could represent a basic building block to enforce them reliably.

Zee is an IFC language for implementing secure (timing-sensitive) runtime systems [PA19]. The lack of asynchronous exceptions in Zee complicates the implementation of certain runtime system components, but we believe that Zee could support them by applying the insights from this work. An interesting line of work aims at exposing safe high-level API to allow users to reprogram features of the runtime systems, e.g. concurrency primitives [Li+07], multicore schedulers [Siv+16], and kill-safe abstractions [FF04]. We believe that the primitives designed to remove covert channels in GHC and other runtime systems discussed above could be implemented following this approach.

B.8. Conclusions and Future Work

This work presents the first IFC language that support asynchronous exceptions securely. Embedded in HASKELL, the IFC library MACASYNC provides primitives for fine-grained masking and unmasking of asynchronous exceptions, which enable useful programming patterns, that we showcased with two examples. We have formalized MACASYNC in 3,000 lines of AGDA and proved progress-sensitive noninterference.

As future work, we plan to use MACASYNC to reason about the delivery of OS signals to threads. Specially, we will explore OS signals dedicated to alert about exhaustion of resources that cannot be easily partitioned (e.g. the battery in an IoT board). This scenario will demand the OS—which can be thought as just another thread—to send signals from higher to lower levels in

the security lattice, thus opening up an information leakage channel which, we believe, needs to be mitigated.

Another direction for future work consists on using MACasync to build realistic systems. For instance, we expect MACasync to be able to provide an IFC-aware interface for GHC to control CPU usage by leveraging on previous work [Li+07; Siv+16]. Moreover, building realistic systems often involves mutually distrusts principals, where we expect *privileges* [Ste+11a; Way+15] to restrict untrusted code from abusing our selective mask mechanism.

Acknowledgements

This work was funded by the Swedish Foundation for Strategic Research (SSF) under the project WebSec (Ref. RIT17-0011) and Octopi (Ref. RIT17-0023) as well as the Swedish research agency Vetenskapsrådet. This work was partially supported by the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity.

Bibliography

[Aba99] Martín Abadi. "Protection in Programming-Language Translations". In: *Secure Internet Programming, Security Issues for Mobile and Distributed Objects*. Ed. by Jan Vitek and Christian Damsgaard Jensen. Vol. 1603. Lecture Notes in Computer Science. Springer, 1999, pp. 19–34. DOI: 10.1007/3-540-48749-2_2. URL: https://doi.org/10.1007/3-540-48749-2%5C_2 (cit. on p. 86).

[Abe+05] Andreas Abel, Guillaume Allais, Jesper Cockx, Nils Anders Danielsson, Philipp Hausmann, Fredrik Nordvall Forsberg, Ulf Norell, Víctor López Juan, Andrés Sicard-Ramírez, and Andrea Vezzosi. *Agda 2*. 2005–. URL: https://wiki.portal.chalmers.se/agda/pmwiki.php (cit. on p. 58).

[Arm03] Joe Armstrong. "Making reliable distributed systems in the presence of hardware errors". PhD thesis. The Royal Institute of Technology Stockholm, Sweden, 2003 (cit. on p. 85).

[BH05] Godmar Back and Wilson C. Hsieh. "The KaffeOS Java runtime system". In: *ACM Trans. Program. Lang. Syst.* 27.4 (2005), pp. 583–630. DOI: 10.1145/1075382.1075383. URL: https://doi.org/10.1145/1075382.1075383 (cit. on p. 87).

[Bie+12] Gavin M. Bierman, Claudio V. Russo, Geoffrey Mainland, Erik Meijer, and Mads Torgersen. "Pause 'n' Play: Formalizing Asynchronous C#". In: *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*. Ed. by James Noble. Vol. 7313. Lecture Notes in Computer Science. Springer, 2012, pp. 233–257. DOI: 10.1007/978-3-642-31057-7_12. URL: https://doi.org/10.1007/978-3-642-31057-7%5C_12 (cit. on p. 85).

[BL96] David Elliott Bell and Leonard J. LaPadula. "Secure Computer Systems: A Mathematical Model, Volume II". In: *J. Comput. Secur.* 4.2/3 (1996), pp. 229–263 (cit. on p. 61).

[BPS17] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control". In: *Proceedings of the 2nd Workshop on System Software for Trusted Execution, SysTEX@SOSP 2017, Shanghai, China, October 28, 2017*. ACM, 2017, 4:1–4:6. DOI: 10.1145/3152701.3152706. URL: https://doi.org/10.1145/3152701.3152706 (cit. on p. 86).

[BPS18] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. Ed. by David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang. ACM, 2018, pp. 178–195. DOI: 10.1145/3243734.3243822. URL: https://doi.org/10.1145/3243734.3243822 (cit. on p. 86).

[BR13] Pablo Buiras and Alejandro Russo. "Lazy Programs Leak Secrets". In: *Secure IT Systems - 18th Nordic Conference, NordSec 2013, Ilulissat, Greenland, October 18-21, 2013, Proceedings*. Ed. by Hanne Riis Nielson and Dieter Gollmann. Vol. 8208. Lecture Notes in Computer Science. Springer, 2013, pp. 116–122. DOI: 10.1007/978-3-642-41488-6_8. URL: https://doi.org/10.1007/978-3-642-41488-6%5C_8 (cit. on pp. 57, 62, 86).

[Bus+20] Matteo Busi, Job Noorman, Jo Van Bulck, Letterio Galletta, Pierpaolo Degano, Jan Tobias Mühlberg, and Frank Piessens. "Provably Secure Isolation for Interruptible Enclaved Execution on Small Microprocessors". In: *33rd IEEE Computer Security Foundations Symposium, CSF 2020, Boston, MA, USA, June 22-26, 2020*. IEEE, 2020, pp. 262–276. DOI: 10.1109/CSF49147.2020.00026. URL:

https://doi.org/10.1109/CSF49147.2020.00026 (cit. on p. 86).

[BVR15] 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, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. Ed. by Kathleen Fisher and John H. Reppy. ACM, 2015, pp. 289–301. DOI: 10.1145/2784731.2784758. URL: https://doi.org/10.1145/2784731.2784758 (cit. on p. 81).

[Den76] Dorothy E. Denning. "A Lattice Model of Secure Information Flow". In: *Commun. ACM* 19.5 (1976), pp. 236–243. DOI: 10.1145/360051.360056. URL: https://doi.org/10.1145/360051.360056 (cit. on p. 60).

[Dol+17] Stephen Dolan, Spiros Eliopoulos, Daniel Hillerström, Anil Madhavapeddy, K. C. Sivaramakrishnan, and Leo White. "Concurrent System Programming with Effect Handlers". In: *Trends in Functional Programming - 18th International Symposium, TFP 2017, Canterbury, UK, June 19-21, 2017, Revised Selected Papers*. Ed. by Meng Wang and Scott Owens. Vol. 10788. Lecture Notes in Computer Science. Springer, 2017, pp. 98–117. DOI: 10.1007/978-3-319-89719-6_6. URL: https://doi.org/10.1007/978-3-319-89719-6%5C_6 (cit. on p. 85).

[Fer+18] 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 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. Ed. by David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang. ACM, 2018, pp. 1583–1600. DOI: 10.1145/3243734.3243743. URL: https://doi.org/10.1145/3243734.3243743 (cit. on p. 57).

[FF04] Matthew Flatt and Robert Bruce Findler. "Kill-safe synchronization abstractions". In: *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*. Ed. by William W. Pugh and Craig Chambers. ACM, 2004, pp. 47–58. DOI: 10.1145/996841.

996849. URL: https://doi.org/10.1145/996841.996849 (cit. on pp. 57, 87).

[FH92] Matthias Felleisen and Robert Hieb. "The Revised Report on the Syntactic Theories of Sequential Control and State". In: *Theor. Comput. Sci.* 103.2 (1992), pp. 235–271. DOI: 10.1016/0304-3975(92)90014-7. URL: https://doi.org/10.1016/0304-3975(92)90014-7 (cit. on p. 85).

[FM02] Stephen N. Freund and Mark P. Mitchell. *Safe Asynchronous Exceptions For Python*. Tech. rep. Williams College, 2002 (cit. on pp. 58, 84).

[GM84] Richard P. Gabriel and John McCarthy. "Queue-based Multi-processing Lisp". In: *Proceedings of the 1984 ACM Conference on LISP and Functional Programming, LFP 1984, Austin, Texas, USA, August 5-8, 1984*. Ed. by Robert S. Boyer, Edward S. Schneider, and Guy L. Steele Jr. ACM, 1984, pp. 25–44. DOI: 10.1145/800055.802019. URL: https://doi.org/10.1145/800055.802019 (cit. on p. 58).

[Har+08] William L. Harrison, Gerard Allwein, Andy Gill, and Adam M. Procter. "Asynchronous Exceptions as an Effect". In: *Mathematics of Program Construction, 9th International Conference, MPC 2008, Marseille, France, July 15-18, 2008. Proceedings*. Ed. by Philippe Audebaud and Christine Paulin-Mohring. Vol. 5133. Lecture Notes in Computer Science. Springer, 2008, pp. 153–176. DOI: 10.1007/978-3-540-70594-9_10. URL: https://doi.org/10.1007/978-3-540-70594-9%5C_10 (cit. on p. 86).

[He+18] Wenjian He, Wei Zhang, Sanjeev Das, and Yang Liu. "SGXlinger: A New Side-Channel Attack Vector Based on Interrupt Latency Against Enclave Execution". In: *36th IEEE International Conference on Computer Design, ICCD 2018, Orlando, FL, USA, October 7-10, 2018*. IEEE Computer Society, 2018, pp. 108–114. DOI: 10.1109/ICCD.2018.00025. URL: https://doi.org/10.1109/ICCD.2018.00025 (cit. on p. 86).

[Hed+14] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. "JSFlow: tracking information flow in JavaScript and its APIs". In: *Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014*. Ed. by Yookun Cho, Sung Y. Shin, Sang-Wook Kim, Chih-Cheng Hung, and Jiman Hong. ACM, 2014,

pp. 1663–1671. DOI: 10.1145/2554850.2554909. URL: https://doi.org/10.1145/2554850.2554909 (cit. on p. 57).

[Heu+15] Stefan Heule, Deian Stefan, Edward Z. Yang, John C. Mitchell, and Alejandro Russo. "IFC Inside: Retrofitting Languages with Dynamic Information Flow Control (Extended Version)". In: *CoRR* abs/1501.04132 (2015). arXiv: 1501.04132. URL: http://arxiv.org/abs/1501.04132 (cit. on p. 81).

[HS12] Daniel Hedin and Andrei Sabelfeld. "A Perspective on Information-Flow Control". In: *Software Safety and Security - Tools for Analysis and Verification*. Ed. by Tobias Nipkow, Orna Grumberg, and Benedikt Hauptmann. Vol. 33. NATO Science for Peace and Security Series - D: Information and Communication Security. IOS Press, 2012, pp. 319–347. DOI: 10.3233/978-1-61499-028-4-319. URL: https://doi.org/10.3233/978-1-61499-028-4-319 (cit. on p. 58).

[HW07] Graham Hutton and Joel J. Wright. "What is the meaning of these constant interruptions?" In: *J. Funct. Program.* 17.6 (2007), pp. 777–792. DOI: 10.1017/S0956796807006363. URL: https://doi.org/10.1017/S0956796807006363 (cit. on p. 85).

[IAI18] Hiroaki Inoue, Tomoyuki Aotani, and Atsushi Igarashi. "Context-Workflow: A Monadic DSL for Compensable and Interruptible Executions". In: *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*. Ed. by Todd D. Millstein. Vol. 109. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 2:1–2:33. DOI: 10.4230/LIPIcs.ECOOP.2018.2. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2018.2 (cit. on p. 85).

[JGF96] Simon L. Peyton Jones, Andrew D. Gordon, and Sigbjørn Finne. "Concurrent Haskell". In: *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*. Ed. by Hans-Juergen Boehm and Guy L. Steele Jr. ACM Press, 1996, pp. 295–308. DOI: 10.1145/237721.237794. URL: https://doi.org/10.1145/237721.237794 (cit. on p. 62).

[Jon+99] Simon L. Peyton Jones, Alastair Reid, Fergus Henderson, C. A. R. Hoare, and Simon Marlow. "A Semantics for Imprecise Exceptions". In: *Proceedings of the 1999 ACM SIGPLAN Conference on Program-*

ming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, May 1-4, 1999. Ed. by Barbara G. Ryder and Benjamin G. Zorn. ACM, 1999, pp. 25–36. DOI: 10.1145/301618.301637. URL: https://doi.org/10.1145/301618.301637 (cit. on p. 85).

[Li+07] Peng Li, Simon Marlow, Simon L. Peyton Jones, and Andrew P. Tolmach. "Lightweight concurrency primitives for GHC". In: *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2007, Freiburg, Germany, September 30, 2007*. Ed. by Gabriele Keller. ACM, 2007, pp. 107–118. DOI: 10.1145/1291201.1291217. URL: https://doi.org/10.1145/1291201.1291217 (cit. on pp. 57, 87, 88).

[LZ10] Peng Li and Steve Zdancewic. "Arrows for secure information flow". In: *Theor. Comput. Sci.* 411.19 (2010), pp. 1974–1994. DOI: 10.1016/j.tcs.2010.01.025. URL: https://doi.org/10.1016/j.tcs.2010.01.025 (cit. on p. 80).

[Mar+01] Simon Marlow, Simon L. Peyton Jones, Andrew Moran, and John H. Reppy. "Asynchronous Exceptions in Haskell". In: *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*. Ed. by Michael Burke and Mary Lou Soffa. ACM, 2001, pp. 274–285. DOI: 10.1145/378795.378858. URL: https://doi.org/10.1145/378795.378858 (cit. on pp. 58, 64, 67, 73, 76, 78, 84, 85).

[Mar17] Simon Marlow. *Asynchronous Exceptions in Practice*. Jan. 2017. URL: https://simonmar.github.io/posts/2017-01-24-asynchronous-exceptions.html (cit. on p. 84).

[Mye+06] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. *Jif: Java information flow*. 2006. URL: https://www.cs.cornell.edu/jif (cit. on p. 57).

[Ora20] Oracle. *Java Thread Primitive Deprecation*. Oracle. 2020. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html (cit. on pp. 58, 84).

[PA17] Mathias V. Pedersen and Aslan Askarov. "From Trash to Treasure: Timing-Sensitive Garbage Collection". In: *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 2017, pp. 693–709. DOI: 10.1109/

SP.2017.64. URL: https://doi.org/10.1109/SP.2017.64 (cit. on pp. 57, 86).

[PA19] Mathias Vorreiter Pedersen and Aslan Askarov. "Static Enforcement of Security in Runtime Systems". In: *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*. IEEE, 2019, pp. 335–350. DOI: 10.1109/CSF.2019.00030. URL: https://doi.org/10.1109/CSF.2019.00030 (cit. on pp. 57, 86, 87).

[RCH08] Alejandro Russo, Koen Claessen, and John Hughes. "A library for light-weight information-flow security in haskell". In: *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*. Ed. by Andy Gill. ACM, 2008, pp. 13–24. DOI: 10.1145/1411286.1411289. URL: https://doi.org/10.1145/1411286.1411289 (cit. on p. 57).

[Rep90] John H. Reppy. *Asynchronous Signals is Standard ML*. Tech. rep. USA, 1990 (cit. on pp. 58, 85).

[Roy+11] Arjun Roy, Stephen M. Rumble, Ryan Stutsman, Philip Alexander Levis, David Mazières, and Nickolai Zeldovich. "Energy management in mobile devices with the cinder operating system". In: *European Conference on Computer Systems, Proceedings of the Sixth European conference on Computer systems, EuroSys 2011, Salzburg, Austria, April 10-13, 2011*. Ed. by Christoph M. Kirsch and Gernot Heiser. ACM, 2011, pp. 139–152. DOI: 10.1145/1966445.1966459. URL: https://doi.org/10.1145/1966445.1966459 (cit. on p. 87).

[RS06] Alejandro Russo and Andrei Sabelfeld. "Securing Interaction between Threads and the Scheduler". In: *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006), 5-7 July 2006, Venice, Italy*. IEEE Computer Society, 2006, pp. 177–189. DOI: 10.1109/CSFW.2006.29. URL: https://doi.org/10.1109/CSFW.2006.29 (cit. on pp. 57, 82).

[RS09] Alejandro Russo and Andrei Sabelfeld. "Securing Timeout Instructions in Web Applications". In: *Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009, Port Jefferson, New York, USA, July 8-10, 2009*. IEEE Computer Society, 2009, pp. 92–106. DOI: 10.1109/CSF.2009.16. URL: https://doi.org/10.1109/CSF.2009.16 (cit. on p. 57).

[Rus15] Alejandro Russo. "Functional pearl: two can keep a secret, if one of them uses Haskell". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. Ed. by Kathleen Fisher and John H. Reppy. ACM, 2015, pp. 280–288. DOI: 10.1145/2784731.2784756. URL: https://doi.org/10.1145/2784731.2784756 (cit. on pp. 58–60, 63).

[Siv+16] K. C. Sivaramakrishnan, Tim Harris, Simon Marlow, and Simon Peyton Jones. "Composable scheduler activations for Haskell". In: *J. Funct. Program.* 26 (2016), e9. DOI: 10.1017/S0956796816000071. URL: https://doi.org/10.1017/S0956796816000071 (cit. on pp. 57, 87, 88).

[SM03] Andrei Sabelfeld and Andrew C. Myers. "Language-based information-flow security". In: *IEEE J. Sel. Areas Commun.* 21.1 (2003), pp. 5–19. DOI: 10.1109/JSAC.2002.806121. URL: https://doi.org/10.1109/JSAC.2002.806121 (cit. on p. 57).

[SPL11] Don Syme, Tomas Petricek, and Dmitry Lomov. "The F# Asynchronous Programming Model". In: *Practical Aspects of Declarative Languages - 13th International Symposium, PADL 2011, Austin, TX, USA, January 24-25, 2011. Proceedings*. Ed. by Ricardo Rocha and John Launchbury. Vol. 6539. Lecture Notes in Computer Science. Springer, 2011, pp. 175–189. DOI: 10.1007/978-3-642-18378-2_15. URL: https://doi.org/10.1007/978-3-642-18378-2%5C_15 (cit. on p. 85).

[Ste+11a] Deian Stefan, Alejandro Russo, David Mazières, and John C. Mitchell. "Disjunction Category Labels". In: *Information Security Technology for Applications - 16th Nordic Conference on Secure IT Systems, NordSec 2011, Tallinn, Estonia, October 26-28, 2011, Revised Selected Papers*. Ed. by Peeter Laud. Vol. 7161. Lecture Notes in Computer Science. Springer, 2011, pp. 223–239. DOI: 10.1007/978-3-642-29615-4_16. URL: https://doi.org/10.1007/978-3-642-29615-4%5C_16 (cit. on p. 88).

[Ste+11b] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. "Flexible dynamic information flow control in Haskell". In: *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011*. Ed. by Koen Claessen. ACM, 2011, pp. 95–106. DOI: 10.1145/2034675.2034688. URL:

https://doi.org/10.1145/2034675.2034688 (cit. on pp. 58, 80).

[Ste+12] Deian Stefan, Alejandro Russo, Pablo Buiras, Amit Levy, John C. Mitchell, and David Mazières. "Addressing covert termination and timing channels in concurrent information flow systems". In: *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*. Ed. by Peter Thiemann and Robby Bruce Findler. ACM, 2012, pp. 201–214. DOI: 10.1145/2364527.2364557. URL: https://doi.org/10.1145/2364527.2364557 (cit. on pp. 61, 62, 80).

[Ste+14] Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, David Herman, Brad Karp, and David Mazières. "Protecting Users by Confining JavaScript with COWL". In: *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*. Ed. by Jason Flinn and Hank Levy. USENIX Association, 2014, pp. 131–146. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/stefan (cit. on p. 57).

[SV98] Geoffrey Smith and Dennis M. Volpano. "Secure Information Flow in a Multi-Threaded Imperative Language". In: *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*. Ed. by David B. MacQueen and Luca Cardelli. ACM, 1998, pp. 355–364. DOI: 10.1145/268946.268975. URL: https://doi.org/10.1145/268946.268975 (cit. on p. 61).

[TA17] Jérémy Thibault and Aslan Askarov. *Language-based predictive mitigation for systems with asynchronous I/O*. Tech. rep. 2017 (cit. on p. 86).

[TVR20] Carlos Tomé Cortiñas, Marco Vassena, and Alejandro Russo. "Securing Asynchronous Exceptions". In: *33rd IEEE Computer Security Foundations Symposium, CSF 2020, Boston, MA, USA, June 22-26, 2020*. IEEE, 2020, pp. 214–229. DOI: 10.1109/CSF49147.2020.00023. URL: https://doi.org/10.1109/CSF49147.2020.00023 (cit. on p. 55).

[Van+07] Steve Vandebogart, Petros Efstathopoulos, Eddie Kohler, Maxwell N. Krohn, Cliff Frey, David Ziegler, M. Frans Kaashoek, Robert Tappan Morris, and David Mazières. "Labels and event processes in the Asbestos operating system". In: *ACM Trans. Comput. Syst.*

25.4 (2007), p. 11. DOI: 10.1145/1314299.1314302. URL: https://doi.org/10.1145/1314299.1314302 (cit. on p. 57).

[Vas+16] Marco Vassena, Pablo Buiras, Lucas Waye, and Alejandro Russo. "Flexible Manipulation of Labeled Values for Information-Flow Control Libraries". In: *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I*. Ed. by Ioannis G. Askoxylakis, Sotiris Ioannidis, Sokratis K. Katsikas, and Catherine A. Meadows. Vol. 9878. Lecture Notes in Computer Science. Springer, 2016, pp. 538–557. DOI: 10.1007/978-3-319-45744-4_27. URL: https://doi.org/10.1007/978-3-319-45744-4%5C_27 (cit. on p. 59).

[Vas+18] Marco Vassena, Alejandro Russo, Pablo Buiras, and Lucas Waye. "MAC A verified static information-flow control library". In: *Journal of Logical and Algebraic Methods in Programming* 95 (2018), pp. 148–180. ISSN: 2352-2208. DOI: https://doi.org/10.1016/j.jlamp.2017.12.003. URL: https://www.sciencedirect.com/science/article/pii/S235222081730069X (cit. on pp. 58, 59, 62, 66–70, 80, 81, 85).

[Vas+19] 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 - 8th International Conference, POST 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*. Ed. by Flemming Nielson and David Sands. Vol. 11426. Lecture Notes in Computer Science. Springer, 2019, pp. 1–28. DOI: 10.1007/978-3-030-17138-4_1. URL: https://doi.org/10.1007/978-3-030-17138-4%5C_1 (cit. on pp. 57, 86).

[VBR17] Marco Vassena, Joachim Breitner, and Alejandro Russo. "Securing Concurrent Lazy Programs Against Information Leakage". In: *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*. IEEE Computer Society, 2017, pp. 37–52. DOI: 10.1109/CSF.2017.39. URL: https://doi.org/10.1109/CSF.2017.39 (cit. on pp. 57, 85, 86).

[VR16] Marco Vassena and Alejandro Russo. "On Formalizing Information-Flow Control Libraries". In: *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS*

2016, Vienna, Austria, October 24, 2016. Ed. by Toby C. Murray and Deian Stefan. ACM, 2016, pp. 15–28. DOI: 10.1145/2993600.2993608. URL: https://doi.org/10.1145/2993600.2993608 (cit. on pp. 59, 62, 81, 82, 85).

[Way+15] Lucas Waye, Pablo Buiras, Dan King, Stephen Chong, and Alejandro Russo. "It's My Privilege: Controlling Downgrading in DC-Labels". In: *Security and Trust Management - 11th International Workshop, STM 2015, Vienna, Austria, September 21-22, 2015, Proceedings*. Ed. by Sara Foresti. Vol. 9331. Lecture Notes in Computer Science. Springer, 2015, pp. 203–219. DOI: 10.1007/978-3-319-24858-5_13. URL: https://doi.org/10.1007/978-3-319-24858-5%5C_13 (cit. on p. 88).

[Yip+09] Alexander Yip, Neha Narula, Maxwell N. Krohn, and Robert Tappan Morris. "Privacy-preserving browser-side scripting with BFlow". In: *Proceedings of the 2009 EuroSys Conference, Nuremberg, Germany, April 1-3, 2009*. Ed. by Wolfgang Schröder-Preikschat, John Wilkes, and Rebecca Isaacs. ACM, 2009, pp. 233–246. DOI: 10.1145/1519065.1519091. URL: https://doi.org/10.1145/1519065.1519091 (cit. on p. 57).

[YM14] Edward Z. Yang and David Mazières. "Dynamic space limits for Haskell". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. Ed. by Michael F. P. O'Boyle and Keshav Pingali. ACM, 2014, pp. 588–598. DOI: 10.1145/2594291.2594341. URL: https://doi.org/10.1145/2594291.2594341 (cit. on p. 87).

[ZAM12] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. "Language-based control and mitigation of timing channels". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. Ed. by Jan Vitek, Haibo Lin, and Frank Tip. ACM, 2012, pp. 99–110. DOI: 10.1145/2254064.2254078. URL: https://doi.org/10.1145/2254064.2254078 (cit. on p. 86).

[Zel+06] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. "Making Information Flow Explicit in HiStar". In: *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*. Ed. by Brian N. Bershad and Jeffrey C. Mogul. USENIX Association, 2006, pp. 263–278.

URL: `http://www.usenix.org/events/osdi06/tech/zeldovich.html` (cit. on p. 57).

C

Pure Information-Flow Control with Effects Made Simple

Carlos Tomé Cortiñas and Alejandro Russo

Manuscript

Abstract Information-flow control (IFC) is a promising technology to protect data confidentiality. The foundational work on the dependency core calculus (DCC) positions monads as a suitable abstraction for enforcing IFC. Pure functional languages with effects, like Haskell, can provide IFC as a library (e.g. MAC and LIO), a minor task compared to implementing compilers for IFC from scratch.
Previous works on IFC as a library introduce ad hoc primitives to type programs whose effects do not depend on the sensitive data in context. In this work, we start afresh and ask ourselves: what would we need to extend an effect-free language for IFC with secure effects? The answer turns out to be elegant and simple. In a pure language with effects there is a natural place where information flows from sensitive data to effects need to be restricted, and when effects are tracked in a fine-grained fashion, for instance, with a graded monad, then a *single* primitive is enough to allow secure flows! To support our insight, we present and prove secure several IFC enforcement mechanisms based on extensions of the sealing calculus (SC) with effects using a graded monad. Effects that depend on sensitive data are secured through a novel primitive distr. Our security guarantees are mechanized in the Agda proof assistant. Moreover, we provide an implementation of these mechanisms as a new Haskell library for IFC. Our implementation amounts to less than 10 lines of code for the effect-free part and less than 30 lines for the part with effects. Lastly, we demonstrate that our library is capable of encoding previous Haskell libraries for static IFC.

C.1. Introduction

Information-flow control (IFC) [SM03; HS12] is a promising technology to protect data confidentiality. Many IFC approaches are designed to prevent sensitive data from influencing what attackers can observe from a program's public behaviour—a security property known as noninterference [GM82]. IFC mechanisms specify the sensitivity of data via *labels*, and then enforce security by controlling that the flows of information abide by the security policy. In the simplest scenario, there are two labels (alternatively, security levels or sensitivities) `L`, for *public*, and `H`, for *secret*, and the security policy specifies that every flow is allowed except from `H` to `L`—i.e. flows from more to less sensitivity are forbidden. In static approaches to IFC, the sensitivity of data is known a priori, e.g. by specifying it using types, and the enforcement statically decides, e.g. during type checking, whether the program will leak information upon execution. To maintain confidentiality of data, IFC mechanisms need to protect against two kinds of potentially malicious flows: 1) *explicit flows*, when public data directly depends on secret data; and 2) *implicit flows*, when the control flow and public outputs of the program are indirectly influenced by secret data, e.g. due to branching on a `H`-labelled Boolean.

In recent years, the use of pure functional languages has been proliferating for tackling different IFC challenges (e.g. Vassena, Russo, Garg, Rajani, and Stefan [Vas+19], Parker, Vazou, and Hicks [PVH19], Polikarpova, Stefan, Yang, Itzhaky, Hance, and Solar-Lezama [Pol+20], and Rajani and Garg [RG20]). From a pragmatic perspective, pure functional languages can provide IFC security via libraries [LZ06; RCH08; Ste+11], which is less demanding than building compilers from scratch (e.g. Simonet [Sim03] and Myers, Zheng, Zdancewic, Chong, and Nystrom [Mye+06]).

Pure languages are particularly well suited for controlling information flows because of their abstraction facilities and strong encapsulation of effects. For instance, the popular dependency core calculus (DCC) [Aba+99] utilizes the abstract type $\mathsf{T}_{\mathtt{H}}\,A$ to label pieces of data of type A with sensitivity `H` and then the type system ensures that data can only be eliminated into—or flow to—data of equal or higher sensitivity. DCC's security guarantees ensure that programs without effects are secure. A different strand of work aims to provide security in pure languages with effects by restricting the interplay between sensitive data and public effects (e.g. LIO [Ste+11], MAC [Rus15] and HLIO [BVR15]). In a pure language like Haskell, the *only* programs that can produce effects and thereby interact with the external world have to be of type $\mathsf{IO}\,A$, for some type A. In this light, in order to protect against implicit flows through effects it is enough to control which programs of IO type are safe to execute.

Let us consider the Haskell library MAC as an example. MAC replaces the IO monad with a custom monad MAC_l of computations that is indexed by a type-level label l. The label l has two purposes: (i) akin to DCC, it is an *upper bound* on the sensitivity of the information "going into" the monad, as well as (ii) it is a *lower bound* on the observers' effects—restricting information "leaving" the monad. Concretely, a computation of type $\mathsf{MAC}_{\mathtt{L}}\,\mathsf{Bool}$ *cannot* branch on secret values but can perform public and secret effects; in contrast, a computation of type $\mathsf{MAC}_{\mathtt{H}}\,\mathsf{Bool}$ can branch on secret data but *cannot* perform public effects.

However, not everything in the garden is rosy. The label l in the MAC monad MAC_l does too many things at once. This leads to situations where the programmer needs to go through some contortions or use "special purpose" primitives like join[1]: $\mathsf{MAC}_l\,\mathsf{Unit} \Rightarrow \mathsf{MAC}_{l'}\,\mathsf{Unit}$ (restricted to $l' \sqsubseteq l$) [Vas+16]. To illustrate this point, we extend the two-point security policy with two incomparable labels $\mathtt{A}$, for *Alice*, and $\mathtt{B}$, for *Bob*, such that $\mathtt{L} \sqsubseteq \mathtt{A} \sqsubseteq \mathtt{H}$ and $\mathtt{L} \sqsubseteq \mathtt{B} \sqsubseteq \mathtt{H}$ *but* neither $\mathtt{A} \sqsubseteq \mathtt{B}$ nor $\mathtt{B} \sqsubseteq \mathtt{A}$ (the relation $\sqsubseteq$ specifies the permitted flows). With that in mind, let us consider the following program in MAC which receives an $\mathtt{A}$-sensitive Bool, i.e. *alice_sec* : $\mathsf{MAC}_{\mathtt{A}}\,\mathsf{Bool}$,[2] and prints a string on the $\mathtt{A}$-observable channel $\mathtt{Ch}_{\mathtt{A}}$:

$$
\begin{aligned}
&\mathit{prog}_1 : \mathsf{MAC}_{\mathtt{A}}\,\mathsf{Bool} \Rightarrow \mathsf{MAC}_{\mathtt{A}}\,\mathsf{Unit}\\
&\mathit{prog}_1\ \mathit{alice\_sec} = \mathit{alice\_sec} \gg\!= \lambda b.\ \mathsf{if}\ b\ \mathsf{then}\ \mathsf{print}_{\mathtt{Ch}_{\mathtt{A}}}\ \texttt{"Alice is here!"}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \mathsf{else}\ \mathsf{return}\ \mathsf{unit}
\end{aligned}
$$

In the above program, the information flows according to the security policy (from the $\mathtt{A}$-sensitive value to Alice's channel)—i.e. the program is secure. Consider now a different program that combines prog_1 with printing on the channel $\mathtt{Ch}_{\mathtt{B}}$:

$$
\begin{aligned}
&\mathit{prog}_2 : \mathsf{MAC}_{\mathtt{A}}\mathsf{Bool} \Rightarrow \mathsf{MAC}\ \boxed{?}\ \mathsf{Unit}\\
&\mathit{prog}_2\ \mathit{alice\_sec} = \mathit{prog}_1\ \mathit{alice\_sec} \gg\!= \lambda\, x.\ \mathsf{print}_{\mathtt{Ch}_{\mathtt{B}}}\ \texttt{"Hi Bob"}
\end{aligned}
$$

Clearly, prog_2 is still secure since the decision to print to Bob's channel and what is printed does not depend on the contents of the $\mathtt{A}$-sensitive value *alice_sec*. What label then should replace $\boxed{?}$ in its type? First, the types of the computations on both sides of the bind ($\gg\!=$) have to match. By (i) and (ii) the type of prog_1 *sec* has to be $\mathsf{MAC}_{\mathtt{A}}\,\mathsf{Unit}$, and by (ii) $\mathsf{print}_{\mathtt{Ch}_{\mathtt{B}}}\ \texttt{"Hi Bob"}$ has to

[1] The type of join is more general but this simplified form suffices for our purposes.

[2] To the reader familiar with MAC: our point applies equally if one uses the $\mathsf{Labeled}_l$ type to protect sensitive Booleans.

be of type $\mathsf{MAC}_{\mathtt{B}}\,\mathsf{Unit}$. There is a type mismatch! Since the label `L` is a lower bound of `A` and `B`, we can use MAC's join to fix the program:

$$\begin{aligned}&\mathit{prog}_2 : \mathsf{MAC}_{\mathtt{A}}\,\mathsf{Bool} \Rightarrow \mathsf{MAC}_{\mathtt{L}}\,\mathsf{Unit}\\&\mathit{prog}_2\ \mathit{alice\_sec} = \mathsf{join}(\mathit{prog}_1\ \mathit{alice\_sec}) \gg\!= \lambda\, x.\ \mathsf{join}(\mathsf{print}_{\mathtt{Ch_B}}\ \texttt{"Hi Bob"})\end{aligned}$$

Now, something unexpected happened. To assign a type to prog_2, which only mentions labels `A` and `B`, we *have to* resort to another label `L`. What is worse, in the type of prog_2 only the label `A` appears and does so in an argument position which means we know nothing about the program's possible effects. The design decision of indexing the monad MAC by a *single* label, while enabling a simple implementation of IFC as a library (cf. [Rus15]), requires the application of special purpose primitives (like join) to mitigate over-approximations of how information flows in and out of computations.

In this work, we take a step back and ask ourselves: what would it take to allow arbitrary effects in a pure language with an already existing mechanism for effect-free IFC? The answer turns out to be elegant and simple. We observe that enforcing IFC in a pure language with effects can be reduced to the *single* point—which we will explain below—where effects and sensitive data interact. Based on this observation, we present a novel IFC mechanism which is arguably simpler than existing IFC libraries, allows to assign more natural types to programs; and overcomes the programming contortions discussed above.

To briefly present our idea, let us assume that effect-free IFC is achieved using a DCC-style abstract type $\mathsf{T}_l\,A$, and we have at our disposal a more refined type constructor Eff of effectful programs akin to IO but annotated with concrete information about *observable* effects. For example, Eff could be annotated with the set of channels where the program might print, the set of exceptions the program might raise, or the set of memory references the program might modify. Moreover, we assume that the security policy specifies the sensitivity of each effect.

Consider, for instance, a scenario with two output channels, namely $\mathtt{Ch_L}$ and $\mathtt{Ch_H}$, that are assigned sensitivities `L` and `H`, respectively. In this scenario, a program $\mathit{prog}_3 : \mathsf{T}_{\mathtt{A}}\,(\mathsf{Eff}_{\{\mathtt{Ch_B}\}}\,\mathsf{Unit})$ is a computation that might print to `B`'s channel $\mathtt{Ch_B}$ depending on `A`-sensitive data. This is where sensitive data and effects interact! Assuming $\mathit{alice\_sec} : \mathsf{T}_{\mathtt{A}}\,\mathsf{Bool}$ is in scope, prog_3, for instance,

could have the following implementation:[3]

$$
\begin{aligned}
&\mathit{prog}_3 : \mathsf{T}_{\mathtt{A}}\,(\mathsf{Eff}_{\{\mathtt{Ch_B}\}}\,\mathsf{Unit})\\
&\mathit{prog}_3 = \mathsf{bind}\,b = \mathit{alice\_sec}\\
&\qquad \mathsf{in}\ \mathsf{return}_{\mathtt{A}}(\mathsf{if}\,b\,\mathsf{then}\ \mathsf{print}_{\mathtt{Ch_B}}\ \texttt{"So it was true, huh"}\\
&\qquad\qquad \mathsf{else}\ \mathsf{return}\,\mathsf{unit})
\end{aligned}
$$

In order to run the effects of the inner computation of type $\mathsf{Eff}_{\{\mathtt{B}\}}\,\mathsf{Unit}$ we have to "extract" it first from the $\mathsf{T}_{\mathtt{A}}$ value—recall that the language is pure. In general, extracting anything from $\mathsf{T}_{\mathtt{A}}$ is prohibited as it would render the IFC enforcement unsound: in prog_3, the computation of type $\mathsf{Eff}_{\{\mathtt{Ch_B}\}}\,\mathsf{Unit}$ would become executable, and if the $\mathtt{A}$-sensitive Boolean *alice_sec* is true, it will print `"So it was true, huh"` on Bob's channel $\mathtt{Ch_B}$—a flow that violates the security policy. On the contrary, let us consider a program of a different type, $\mathit{prog}_4 : \mathsf{T}_{\mathtt{A}}\,(\mathsf{Eff}_{\{\mathtt{Ch_H}\}}\,\mathsf{Unit})$. A possible implementation of prog_4 could be:

$$
\begin{aligned}
&\mathit{prog}_4 : \mathsf{T}_{\mathtt{A}}\,(\mathsf{Eff}_{\{\mathtt{Ch_H}\}}\,\mathsf{Unit})\\
&\mathit{prog}_4 = \mathsf{bind}\,b = \mathit{alice\_sec}\\
&\qquad \mathsf{in}\ \mathsf{return}_{\mathtt{A}}(\mathsf{if}\,b\,\mathsf{then}\ \mathsf{print}_{\mathtt{Ch_H}}\ \texttt{"It is true!"}\\
&\qquad\qquad \mathsf{else}\ \mathsf{print}_{\mathtt{Ch_H}}\ \texttt{"It is false!"})
\end{aligned}
$$

In this program, the computation, i.e. $\mathsf{if}\,b\ \mathsf{then}\ \mathsf{print}_{\mathtt{Ch_H}}\ \texttt{"It is true!"}\ \mathsf{else}\ \mathsf{print}_{\mathtt{Ch_H}}\ \texttt{"It is false!"}$, is safe to run since it prints $\mathtt{A}$-sensitive data on the channel $\mathtt{Ch_H}$—a flow permitted by the security policy. In fact, the underlying computation of any program of type $\mathsf{T}_{\mathtt{A}}\,(\mathsf{Eff}_{\{\mathtt{Ch_H}\}}\,\mathsf{Unit})$ is *definitely* safe to run since the only possible effects it might produce are printing to the channel $\mathtt{Ch_H}$, which we know from the type $\mathsf{Eff}_{\{\mathtt{Ch_H}\}}\,\mathsf{Unit}$, and the decision on what to print depends on data of at most sensitivity $\mathtt{A}$, which we know from the type constructor $\mathsf{T}_{\mathtt{A}}$. Securing effectful programs then amounts to allowing any such program prog_4 to extract the computation of type $\mathsf{Eff}_{\{\mathtt{Ch_H}\}}\,\mathsf{Unit}$ from $\mathsf{T}_{\mathtt{A}}$ and run its effects while forbidding prog_3 from doing so. To achieve this, we introduce a novel primitive distr which systematically permits computations to be extracted, and hence, executed only when they are known to depend on data less sensitive than the sensitivity of their observers. With distr we can turn prog_4 into an executable program: $\mathsf{distr}(\mathit{prog}_4) : \mathsf{Eff}_{\{\mathtt{Ch_H}\}}\,\mathsf{Unit}$. We have reduced the enforcement of IFC in pure languages with effects to a *single* primitive; this gives us *modularity*, *clarity* and *simplicity* in the language design and its possible implementations.

[3] bind is DCC's eliminator for the type T_l.

To conclude, we provide an alternative implementation of $prog_2$ with a more natural type using the primitive distr:

$$
\begin{aligned}
&prog_2' : \mathsf{T}_{\mathtt{A}}\ \mathsf{Bool} \Rightarrow \mathsf{Eff}_{\{\mathtt{Ch_A},\mathtt{Ch_B}\}}\ \mathsf{Unit}\\
&prog_2'\ alice\_sec = \mathsf{distr}\ (\mathsf{bind}\ b = alice\_sec\\
&\qquad\qquad\qquad\qquad \mathsf{in\ return}(\mathsf{if}\ b\ \mathsf{then}\ \mathsf{print}_{\mathtt{Ch_A}}\ \texttt{"Alice is here!"}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else\ return\ unit}))\\
&\qquad\qquad\qquad \gg\!= \lambda\, x.\, \mathsf{print}_{\mathtt{Ch_B}}\ \texttt{"Hi Bob"}
\end{aligned}
$$

Our Contributions In this work, we show that IFC in the context of pure languages with effects can be achieved through the combination of the following features: 1. an enforcement for effect-free IFC, 2. a type for tracking observable effects in a fine-grained fashion, and 3. a primitive distr which selectively permits to execute effectful computations which depend on sensitive data.

We present our idea through an IFC enforcement mechanism in the form of a security-type system for programs written in the programming language λ_{REC}, which is a call-by-name variant of the simply-typed λ-calculus (STLC) extended with recursive functions and Booleans. The security-type system for λ_{REC}, which we dub λ_{SC}, is an adaptation of the sealing calculus (SC) of Shikuma and Igarashi [SI08], which is more expressive than DCC (cf. [TZ04; SI08]). We then extend λ_{REC} in two different directions by adding printing and global store effects in the form of explicit graded monadic effects [Kat14]. We name these extensions $\lambda_{\text{REC}}^{\text{PRINT}}$ and $\lambda_{\text{REC}}^{\text{STORE}}$, respectively. Although the orthogonal extensions of λ_{SC} for printing and global store readily enforce IFC by not permitting any interaction between labelled values and effects, we additionally extend λ_{SC} with our novel primitive distr. This allows to type check more of those effectful programs that are secure.

Along with our informal argumentation for why our idea yields secure IFC enforcement mechanisms for the different effects, we have mechanized proofs in the Agda proof assistant [Abe+05] about the security guarantees that the programs (in the terminating fragment of the languages) satisfy, namely termination-insensitive noninterference (TINI). Our proofs are based on the technique of step-indexed logical relations [Ahm06].

Finally, we realize our idea in the form of a proof-of-concept Haskell library which we call SCLib. The conciseness of our implementation illustrates the elegance and simplicity of our insight: less than 10 lines of code for the effect-free fragment and less than 30 for the part with effects. In order to implement the effect-free IFC mechanism we also present a novel implementation of SC using an encoding of contextual information as type-level capabilities. Our library is at least as expressive as previous work on libraries for IFC in Haskell, which

we evidence by showing implementations of SecLib [RCH08], DCC (in its presentation by Algehed [Alg18]) and MAC in terms of SCLib's interface.

In summary, the technical contributions of this paper are:

- A reformulation of SC as security-type system, λ_{SC}, for λ_{REC} (Section C.2)
- Two extensions of λ_{REC} and λ_{SC} for enforcing IFC in pure languages with effects via graded monads. As examples, we consider printing (Section C.3.1) and global store (Section C.3.2) effects
- We present distr, a single primitive that can control the interaction of sensitive data and effects
- Security guarantees and proofs of TINI based on logical relations for all the enforcements (Section C.4)
- A Haskell implementation using a novel encoding of contextual information as capabilities together with evidence that SCLib can encode existing monadic security libraries (Section C.5)
- Mechanized proofs of our security guarantees (Agda code submitted as accompanying material)

C.2. Effect-Free Information-Flow Control

In this section, we briefly recall the sealing calculus (SC) [SI08], introduce the programming language on which we wish to enforce IFC, and explain our adaptation of SC as a security-type system.

The Sealing Calculus SC utilizes an abstract type $\mathsf{S}_l\ A$ for protecting sensitive data.[4] A value of type $\mathsf{S}_l\ A$ is "sealed" at sensitivity l in the sense that it is only available to observers with sensitivity at least as high as l. Values of type $\mathsf{S}_l\ A$ are introduced and eliminated using the primitives seal_l and unseal_l. SC enforces IFC by restricting in which *contexts* a sealed value can be "unsealed". For example, the A-sensitive Boolean $sec :: \mathsf{S}_{\mathtt{A}}\ \mathsf{Bool}$ can only be unsealed in contexts of at least sensitivity A.

Let us illustrate SC with a program that receives two Booleans with sensitivities A and B, respectively, and computes their conjunction with sensitivity H

[4] In the original presentation, the authors use the notation $[A]^l$ instead.

($\mathsf{and} : \mathsf{Bool} \times \mathsf{Bool} \Rightarrow \mathsf{Bool}$ implements conjunction):

$$
\begin{aligned}
&\mathit{and}' : \mathsf{S}_{\mathtt{A}}\,\mathsf{Bool} \Rightarrow \mathsf{S}_{\mathtt{B}}\,\mathsf{Bool} \Rightarrow \mathsf{S}_{\mathtt{H}}\,\mathsf{Bool} \\
&\mathit{and}' = \lambda\, \mathit{sb}_1\ \mathit{sb}_2.\, \mathsf{seal}_{\mathtt{H}}\,(\mathsf{and}\,(\mathsf{unseal}_{\mathtt{A}}\ \mathit{sb}_1)\,(\mathsf{unseal}_{\mathtt{B}}\ \mathit{sb}_2))
\end{aligned}
$$

In the above program, the term $\mathsf{seal}_{\mathtt{H}}$ provides the context in which sb_1 and sb_2 can be unsealed: the term $\mathsf{unseal}_{\mathtt{A}}\ \mathit{sb}_1$, for instance, is only well-typed because its label $\mathtt{A}$ flows to $\mathtt{H}$, which is the highest label.

The Language λ_{REC} is a call-by-name variant of the STLC with Unit and Bool as the only primitive types and extended with recursive function definitions. We work directly with intrinsically-typed terms, and thus we consider that typing derivations $\Gamma \vdash t : a$ *are* terms. Terms of the form $\mu\, f.\, x.\, t$ define recursive functions, where the bound variable f refers to the function being defined. When the function is not recursive we use the usual λ-abstraction $\lambda\, x.\, t = \mu\, f.\, x.\, t$. The small-step semantics is specified by a relation $t \to t'$ on closed terms, i.e. $\cdot \vdash t : a$, and we call values those terms t for which $t \not\to$. For reference we include the complete definition in Appendix I. Since λ_{REC} is well-understood we omit any further explanations.

The Security-Type System In Figure C.1 we introduce λ_{SC}, the security-type system for programs written in λ_{REC}. The syntax is parameterized by a security policy specified in the form of a lattice structure on the set of labels $(L, \sqsubseteq)$. The types reflect those in λ_{REC} and include a new type constructor S_l for each label l. Typing judgements are of the form $\pi\ ;\ \Gamma \vdash^{\text{SC}} t : A$ where: π is a *finite* set of labels drawn from L, i.e. $\pi \subseteq L$; Γ is an λ_{SC} typing context; and A is an λ_{SC} type. The component π is analogous to protection contexts from related work by Tse and Zdancewic [TZ04].

The set of labels π in the typing judgement represents the sensitivities of all the data on which the program may depend. In order to clarify the role of π, let us consider a program with a typing derivation indexed by the set of labels $\pi_1 \coloneqq \{\mathtt{H}\}$, $\pi_1\ ;\ \cdot \vdash^{\text{SC}} p : A$. This program may depend on data labelled at types $\mathsf{S}_{\mathtt{L}}\,A$ and $\mathsf{S}_{\mathtt{H}}\,A$ by unsealing. If, instead, p is indexed by the set $\pi_2 \coloneqq \{\mathtt{L}\}$, i.e. $\pi_2\ ;\ \cdot \vdash^{\text{SC}} p : A$, then the only terms that the program can depend on by unsealing are of type $\mathsf{S}_{\mathtt{L}}\,A$. This mechanism ensures that the flows of information are secure. It is useful to think that the labels that belong to π act as a kind of type-level *key* whose "possession" permits access to information at most as sensitive as the label itself.

The typing rules of the λ_{REC} fragment of λ_{SC}, i.e. Rules Fun, App and If, are rather standard: they simply propagate the set of labels π to their premises.

$$\begin{array}{rl} \text{Sets of labels} & \pi \subseteq L \\ \text{Types} & A, B ::= \mathsf{Unit} \mid \mathsf{Bool} \mid A \Rightarrow B \mid \mathsf{S}_l\, A \\ \text{Typing contexts} & \Gamma ::= \cdot \mid \Gamma, x : A \end{array}$$

$\boxed{\pi \,;\, \Gamma \vdash^{\mathrm{SC}} t : A}$

$$\textsc{Var}\ \frac{(x : A) \in \Gamma}{\pi \,;\, \Gamma \vdash^{\mathrm{SC}} x : A} \qquad \textsc{Fun}\ \frac{\pi \,;\, \Gamma, f : A \Rightarrow B, x : A \vdash^{\mathrm{SC}} t : B}{\pi \,;\, \Gamma \vdash^{\mathrm{SC}} \mu\, f.\, x.\, t : A \Rightarrow B}$$

$$\textsc{App}\ \frac{\pi \,;\, \Gamma \vdash^{\mathrm{SC}} t : A \Rightarrow B \qquad \pi \,;\, \Gamma \vdash^{\mathrm{SC}} u : A}{\pi \,;\, \Gamma \vdash^{\mathrm{SC}} \mathsf{app}\, t\, u : B}$$

$$\textsc{Unit}\ \frac{}{\pi \,;\, \Gamma \vdash^{\mathrm{SC}} \mathsf{unit} : \mathsf{Unit}} \qquad \textsc{True}\ \frac{}{\pi \,;\, \Gamma \vdash^{\mathrm{SC}} \mathsf{true} : \mathsf{Bool}} \qquad \textsc{False}\ \frac{}{\pi \,;\, \Gamma \vdash^{\mathrm{SC}} \mathsf{false} : \mathsf{Bool}}$$

$$\textsc{If}\ \frac{\pi \,;\, \Gamma \vdash^{\mathrm{SC}} t : \mathsf{Bool} \qquad \pi \,;\, \Gamma \vdash^{\mathrm{SC}} u_1 : A \qquad \pi \,;\, \Gamma \vdash^{\mathrm{SC}} u_2 : A}{\pi \,;\, \Gamma \vdash^{\mathrm{SC}} \mathsf{ifte}\, t\, u_1\, u_2 : A}$$

$$\textsc{Seal}\ \frac{\pi \cup \{l\} \,;\, \Gamma \vdash^{\mathrm{SC}} t : A}{\pi \,;\, \Gamma \vdash^{\mathrm{SC}} \mathsf{seal}_l\, t : \mathsf{S}_l\, A} \qquad \textsc{Unseal}\ \frac{\pi \,;\, \Gamma \vdash^{\mathrm{SC}} t : \mathsf{S}_l\, A \qquad \exists l' \in \pi.\, l \sqsubseteq l'}{\pi \,;\, \Gamma \vdash^{\mathrm{SC}} \mathsf{unseal}_l\, t : A}$$

Figure C.1.: Types and intrinsically-typed terms of λ_{SC}

Observe that one has to explicitly unseal sensitive Booleans, i.e. of type $\mathsf{S}_l\, \mathsf{Bool}$, in order to branch on them using the Rule If. Rules Unseal and Seal are the most interesting since they enforce that information flows to the appropriate places. Rule Seal serves a double purpose: from premise to conclusion, it introduces terms of type $\mathsf{S}_l\, A$; and, from conclusion to premise, it extends the set of labels with the label l, i.e. $\pi \cup \{l\}$. The typing derivation above the premise can then unseal any term of type $\mathsf{S}_{l'}\, A$ such that its label l' can flow to l. Rule Unseal allows unsealing a term with type $\mathsf{S}_l\, A$ if the set π in its conclusion contains at least a label l' such that $l \sqsubseteq l'$. Continuing with the intuition of labels in π as keys, a key $l' \in \pi$ can be used to unseal terms of type $\mathsf{S}_l\, A$ exactly when $l \sqsubseteq l'$.

In order to use λ_{SC} as a security-type system for λ_{REC} programs, we define a family of erasure functions from λ_{SC} types, typing contexts, and terms $\pi\ ;\ \Gamma \vdash^{\text{SC}} t : A$ to λ_{REC} types, typing contexts, and programs $\varepsilon(\Gamma) \vdash \varepsilon(t) : \varepsilon(A)$:

$$\begin{array}{ll} \varepsilon(\mathsf{Unit}) = \mathsf{Unit} & \varepsilon(\mu\, f.\, x.\, t) = \mu\, f.\, x.\, \varepsilon(t) \\ \varepsilon(\mathsf{Bool}) = \mathsf{Bool} & \varepsilon(\mathsf{app}\, t\, u) = \mathsf{app}\, \varepsilon(t)\, \varepsilon(u) \\ \varepsilon(A \Rightarrow B) = \varepsilon(A) \Rightarrow \varepsilon(B) & \varepsilon(\mathsf{seal}_l\, t) = \varepsilon(t) \\ \varepsilon(\mathsf{S}_l\, A) = \varepsilon(A) & \varepsilon(\mathsf{unseal}_l\, t) = \varepsilon(t) \end{array}$$

$$\begin{array}{l} \varepsilon(\cdot) = \cdot \\ \varepsilon(\Gamma, x : A) = \varepsilon(\Gamma), x : \varepsilon(A) \end{array}$$

To clarify this point further, the noninterference property enforced by a λ_{SC} term $\{\mathtt{L}\}\ \ ;\ sec : \mathsf{S}_{\mathtt{H}}\, \mathsf{Bool} \vdash^{\text{SC}} t : \mathsf{Bool}$ on its underlying λ_{REC} program $sec : \mathsf{Bool} \vdash \varepsilon(t) : \mathsf{Bool}$ is that for all $\cdot \vdash s_1, s_2 : \mathsf{Bool}$ whenever both $\varepsilon(t)[s_1/sec]$ and $\varepsilon(t)[s_2/sec]$ terminate then they do so with the same Boolean. In this way, λ_{SC} types and typing contexts play the role of security specifications, and λ_{SC} terms of evidence that the underlying programs are secure, i.e. they satisfy the security specification. In contrast with SC, λ_{SC} terms do not come equipped with an operational semantics, only their underlying λ_{REC} programs do. However, when convenient we identify λ_{SC} terms with their erased λ_{REC} programs. To conclude, we observe that the security-type system is closed under the operational semantics of λ_{REC}:

Lemma C.2.1. *Given a term $\pi\ ;\ \cdot \vdash^{SC} t : A$ such that $\varepsilon(t) \to p$ then there exists a term $\pi\ ;\ \cdot \vdash^{SC} t' : A$ such that $\varepsilon(t') = p$.*

C.3. Effectful Information-Flow Control

In this section, we present the main contribution of this paper: the observation that a single primitive distr is enough to enforce IFC in pure languages with effects. We study two extensions of the programming language and the security-type system from Section C.2 with printing and global store.

In these extensions, we treat effects *explicitly* in the style of Haskell's IO monad [JW93], Moggi's monadic metalanguage [Mog91], or Katsumata's explicit subeffecting calculus (EFe) [Kat14, Section 5]: the only programs that can perform effects are of type $\mathsf{Eff}_C\, a$ for some effect annotation C and type a, and sequencing of effects is made explicit through the primitive bind. Specifically, we consider printing and global store effects, Sections C.3.1 and C.3.2, respectively, as suitable representatives of the two kinds of effects that need to be secured:

Printing Effects. Printing on a channel can be observed externally to the program by the channel's observers. Observers can infer information about the program's input from what is being printed to the channel. To secure printing effects one must ensure that the decision to print and what is being printed only depends on data less sensitive than the channel's observers.

Global Store Effects. Reading from the store cannot be observed directly. However, reading effects need to be secured because what is read may influence the program's subsequent behaviour. To secure reading effects one must ensure that what has been read is tracked as sensitive data.

In contrast to HASKELL and the metalanguage, we use a graded monad [Kat14] whose effect annotation C tracks precisely to which channels a computation might print and which store locations a computation might access.

C.3.1. Printing Effects

In Figure C.2 we present the extension of λ_{REC} which allows programs to perform printing effects. We dub this language $\lambda_{\text{REC}}^{\text{PRINT}}$. We assume that the set of printing channels Ch is fixed a priori, i.e. the channels are statically known. The set of types is extended with a new type for computations $\mathsf{Eff}_C\, a$ that is indexed by a set of channels $C \subseteq Ch$. A program of type $\mathsf{Eff}_C\, a$ when executed might *only* print to the channels that appear in C and return a result of type a.

Statics The typing rules of the standard monadic operations, return and bind, are as expected: in Rule RETURN, the computation does not perform any effects thus the type is indexed by the empty set of channels; and, in Rule BIND, the type is indexed by the union of the channels on which the computations $\Gamma \vdash t : \mathsf{Eff}_{C_1}\, a$ and $\Gamma \vdash u : a \Rightarrow \mathsf{Eff}_{C_2}\, b$ might print, that is, $C_1 \cup C_2$. We include subeffecting—casting from a smaller to a larger set of channels—as the term subeff in the language, see Rule SUBEFF. Printing is performed via a family of primitive operations, print_{ch}, one for each available channel $ch \in Ch$ (Rule PRINT). We assume, for simplicity, that only Boolean values can be printed. Further, observe that the resulting monadic type is indexed by the singleton set that only contains the channel on which the printing is performed, i.e. $\mathsf{Eff}_{\{ch\}}\, \mathsf{Unit}$.

Dynamics The operational semantics of $\lambda_{\text{REC}}^{\text{PRINT}}$ is defined as the combination of the small-step operational semantics of λ_{REC}—see Appendix I for more details—and the small-step operational semantics of computations defined in

$$
\begin{array}{rl}
\text{Sets of channels} & C, C_1, C_2 \subseteq Ch \\
\text{Outputs} & o, o_1, o_2 \in \mathrm{List}\,(Ch \times \mathbb{2}) \\
\text{Types} & a, b ::= \ \ldots \mid \mathsf{Eff}_C\, a \\
\text{Typing contexts} & \Gamma ::= \ \ldots
\end{array}
$$

$\boxed{\Gamma \vdash t : a}$

$$\textsc{Return}\ \frac{\Gamma \vdash t : a}{\Gamma \vdash \mathsf{return}\, t : \mathsf{Eff}_\emptyset\, a} \qquad \textsc{Bind}\ \frac{\Gamma \vdash t : \mathsf{Eff}_{C_1} a \qquad \Gamma \vdash u : a \Rightarrow \mathsf{Eff}_{C_2} b}{\Gamma \vdash \mathsf{bind}\, t\, u : \mathsf{Eff}_{C_1 \cup C_2} b}$$

$$\textsc{Subeff}\ \frac{\Gamma \vdash t : \mathsf{Eff}_{C_1} a \qquad C_1 \subseteq C_2}{\Gamma \vdash \mathsf{subeff}\, t : \mathsf{Eff}_{C_2}\, a} \qquad \textsc{Print}\ \frac{\Gamma \vdash t : \mathsf{Bool}}{\Gamma \vdash \mathsf{print}_{ch}\, t : \mathsf{Eff}_{\{ch\}}\, \mathsf{Unit}}$$

$\boxed{t \rightsquigarrow u, o \text{ with } \cdot \vdash t : \mathsf{Eff}_C\, a \text{ and } \cdot \vdash u : \mathsf{Eff}_C\, a}$

$$\textsc{Bind}\ \frac{t \rightsquigarrow t', o}{\mathsf{bind}\, t\, u \rightsquigarrow \mathsf{bind}\, t'\, u, o} \qquad \textsc{Bind-Subeff}\ \frac{\mathrm{value}\, t}{\mathsf{bind}\, (\mathsf{subeff}\, t)\, u \rightsquigarrow \mathsf{subeff}\, (\mathsf{bind}\, t\, u), \epsilon}$$

$$\textsc{Bind-Ret}\ \frac{}{\mathsf{bind}\, (\mathsf{return}\, t)\, u \rightsquigarrow \mathsf{app}\, u\, t, \epsilon} \qquad \textsc{Subeff}\ \frac{t \rightsquigarrow t', o}{\mathsf{subeff}\, t \rightsquigarrow \mathsf{subeff}\, t', o}$$

$$\textsc{Print}\ \frac{t \to t'}{\mathsf{print}_{ch}\, t \rightsquigarrow \mathsf{print}_{ch}\, t', \epsilon} \qquad \textsc{Print-Val}\ \frac{\mathrm{value}\, b}{\mathsf{print}_{ch}\, b \rightsquigarrow \mathsf{return}\, \mathsf{unit}, [(ch, b)]} \qquad \textsc{EffectFree}\ \frac{t \to t'}{t \rightsquigarrow t', \epsilon}$$

Figure C.2.: Types, well-typed terms and small-step semantics of $\lambda^{\text{PRINT}}_{\text{REC}}$ (omitting the unchanged rules of Appendix I)

Figure C.2. The semantics of effect-free terms, inherited from λ_{REC}, $t \to u$, treats monadic terms, such as $\mathsf{return}\, t$, as values even when their subterms are not values, e.g. $\mathsf{return}\,(\mathsf{app}\,(\mu\, f.\, x.\, x)\,\mathsf{true}) \not\to$. The semantics of computations (alternatively, monadic semantics) is of the form $t \rightsquigarrow u, o$ and is interpreted as follows: program $\cdot \vdash t : \mathsf{Eff}_C\, a$ evaluates in one step to program $\cdot \vdash u : \mathsf{Eff}_C\, a$ and produces output o. The output is a list of pairs of channels and Boolean values $o \in \mathrm{List}\,(Ch \times 2)$, and it represents the outputs of the program during execution.

We now briefly explain the semantics. Rule Bind reduces the left subterm of bind and executes its effects o. Once the left subterm is a value, i.e. $\mathsf{return}\, t$, Bind-Ret applies the rest of the computation u to the underlying value of type a, i.e. t. Applying the continuation u does not produce effects—recall that we are in a pure language—thus the step contains the empty output on the right, i.e. ϵ. Rule Print reduces the argument t of $\mathsf{print}_{ch}\, t$ until it is a value of type Bool, either true or false, and then Rule Print-Val prints the corresponding Boolean on the output channel ch. The output $[(ch, v)]$ is the singleton list with only one pair. Observe that these rules make $\mathsf{print}_{ch}\, t$ strict in its argument. Rule EffectFree serves to lift effect-free reductions to the level of computations. Since by definition effect-free reductions do not produce effects, the right hand side of the effect contains the empty output ϵ. To complete the picture, we denote by $t \rightsquigarrow^* u, o$ the reflexive-transitive closure of the monadic reduction relation. To combine effects, we use the monoid structure on $\mathrm{List}\,(Ch \times 2)$.

Two Reduction Relations While it might seem unnecessary to define the semantics using the combination of a small-step relation of effect-free programs and a small-step relation of computations, it is a natural form of expressing the operational semantics of pure languages with effects [WT03]. The effect-free relation evaluates programs that *cannot* perform effects, whilst the relation for computations evaluates programs which can, and computes those effects.

To better clarify this point, let us consider the execution on the following program, which we show in prettified syntax[5]:

$$
\begin{aligned}
&\mathit{prog}_5 : \mathsf{Eff}_{\{ch_1, ch_2\}}\, \mathsf{Unit} \\
&\mathit{prog}_5 = \mathsf{if}\ \mathsf{true}\ \mathsf{then} \\
&\qquad\quad \mathsf{print}_{ch_1}\ \mathsf{false}\ \mathsf{else}\ \mathsf{return}\ \mathsf{unit} \\
&\qquad\quad \gg\!= \lambda\, x.\, \mathsf{print}_{ch_2}\ \mathsf{true}
\end{aligned}
$$

The program evaluates as follows: 1. the effect-free semantics reduces the

[5]When possible we use Haskell-like syntax and leave the term subeff implicit.

term on the left of the bind, i.e. $\mathsf{if}\,\mathsf{true}\,\mathsf{then}\,\mathsf{print}_{ch_1}\,\mathsf{false}\,\mathsf{else}\,\mathsf{return}\,\mathsf{unit}$, to the value $\mathsf{print}_{ch_1}\,\mathsf{false}$ (Rule If-True through Bind) and this causes no effect; 2. the monadic semantics performs the effect, i.e. printing true on ch_1, (Rule Print-Val); 3. the rest of the computation $\lambda\, x.\,\mathsf{print}_{ch_2}\,\mathsf{true}$ is applied to the resulting value, unit (Rule Bind-Ret); 4. the effect-free semantics reduces the application (Rule Beta through EffectFree) and finally; 5. the monadic semantics performs the effects of $\mathsf{print}_{ch_2}\,\mathsf{true}$.

To conclude the presentation of $\lambda^{\text{PRINT}}_{\text{REC}}$, we enunciate the following lemma which states that the index C in the type of computations $\mathsf{Eff}_C\,a$ is a sound approximation of the set of channels where a program $\cdot \vdash t : \mathsf{Eff}_C\,a$ may print, i.e. t does not produce output in any channel not in C. In the security literature (e.g. [VIS96]) it is usually called *confinement*. Let us denote by $o|_{ch}$ the projection from o to the list consisting of all those pairs whose first component is the channel ch.

Lemma C.3.1 (Confinement for $\lambda^{\text{PRINT}}_{\text{REC}}$)**.** *For any λ^{PRINT}_{REC} program of type $\cdot \vdash p : \mathsf{Eff}_C\,a$, λ^{PRINT}_{REC} value $\cdot \vdash v : \mathsf{Eff}_C\,a$, and output o, if $p \rightsquigarrow^* v, o$ then $\forall ch \in Ch.\, ch \notin C \Rightarrow o|_{ch} = []$.*

Security-Type System After having defined the programming language, we are in position to turn our attention to the extension of λ_{SC} that enforces IFC on $\lambda^{\text{PRINT}}_{\text{REC}}$. We assume that the security policy specifies the sensitivity of each printing channel, i.e. the greatest lower bound of the sensitivities of all its observers, in the form of a function label $\in Ch \to L$. In Figure C.3 we present the extension of λ_{SC} that accommodates printing effects. We name it $\lambda^{\text{PRINT}}_{\text{SC}}$ hereafter. The types are the same as those in λ_{SC} with the addition of a new type constructor Eff_C of computations. The typing rules for the $\lambda^{\text{PRINT}}_{\text{REC}}$ fragment of $\lambda^{\text{PRINT}}_{\text{SC}}$, i.e. Rules Return, Bind, Subeff and Print, simply propagate the set of labels π to their premises. Observe, again, that one has to explicitly unseal sensitive Booleans, i.e. of type $\mathsf{S}_l\,\mathsf{Bool}$, to apply the Rule Print. Before detailing Rule Distr, we extend the family of erasure functions ε from $\lambda^{\text{PRINT}}_{\text{SC}}$-terms to $\lambda^{\text{PRINT}}_{\text{REC}}$ programs in the obvious way: i.e. the type former Eff_C erases to "itself" and, analogously to seal_l and unseal_l, distr is a no-op.

$$\begin{array}{ll} \varepsilon(\dots) = \dots & \varepsilon(\dots) = \dots \\ \varepsilon(\mathsf{Eff}_C\,A) = \mathsf{Eff}_C\,\varepsilon(A) & \varepsilon(\mathsf{distr}\,t) = \varepsilon(t) \end{array}$$

Rule Distr introduces one of the novelties of our work; an enforcement mechanism that selectively permits to execute the effects of computations that depend on sensitive data. In $\lambda^{\text{PRINT}}_{\text{SC}}$, a term of type $\mathsf{S}_l\,(\mathsf{Eff}_C\,A)$ describes a

$$
\begin{array}{rl}
\text{Sets of channels} & C, C_1, C_2 \subseteq Ch \\
\text{Sensitivity of channels} & \mathsf{label} \in Ch \to L \\
\text{Sets of labels} & \pi \subseteq L \\
\text{Types} & A, B ::= \ \ldots \mid \mathsf{Eff}_C\, A \\
\text{Typing contexts} & \Gamma ::= \ \ldots
\end{array}
$$

$\boxed{\pi \ ; \ \Gamma \vdash^{\mathrm{SC}} t : A}$

$$\textsc{Return}\quad \frac{\pi \ ; \ \Gamma \vdash^{\mathrm{SC}} t : A}{\pi \ ; \ \Gamma \vdash^{\mathrm{SC}} \mathsf{return}\, t : \mathsf{Eff}_{\emptyset}\, A}$$

$$\textsc{Bind}\quad \frac{\pi \ ; \ \Gamma \vdash^{\mathrm{SC}} t : \mathsf{Eff}_{C_1}\, A \qquad \pi \ ; \ \Gamma \vdash^{\mathrm{SC}} u : A \Rightarrow \mathsf{Eff}_{C_2}\, B}{\pi \ ; \ \Gamma \vdash^{\mathrm{SC}} \mathsf{bind}\, t\, u : \mathsf{Eff}_{C_1 \cup C_2}\, B}$$

$$\textsc{Subeff}\quad \frac{\pi \ ; \ \Gamma \vdash^{\mathrm{SC}} t : \mathsf{Eff}_{C_1}\, A \qquad C_1 \subseteq C_2}{\pi \ ; \ \Gamma \vdash^{\mathrm{SC}} \mathsf{subeff}\, t : \mathsf{Eff}_{C_2}\, A}$$

$$\textsc{Print}\quad \frac{\pi \ ; \ \Gamma \vdash^{\mathrm{SC}} t : \mathsf{Bool}}{\pi \ ; \ \Gamma \vdash^{\mathrm{SC}} \mathsf{print}_{ch}\, t : \mathsf{Eff}_{\{ch\}}\, \mathsf{Unit}}$$

$$\textsc{Distr}\quad \frac{\pi \ ; \ \Gamma \vdash^{\mathrm{SC}} t : \mathsf{S}_l\, (\mathsf{Eff}_C\, A)}{\pi \ ; \ \Gamma \vdash^{\mathrm{SC}} \mathsf{distr}\, t : \mathsf{Eff}_C\, (\mathsf{S}_l\, A)}\ (\forall ch \in C.\ l \sqsubseteq \mathrm{label}(ch))$$

Figure C.3.: Types and intrinsically-typed terms of $\lambda_{\mathrm{SC}}^{\mathrm{PRINT}}$ (omitting the unchanged rules of Figure C.1)

computation that might *only* print on the channels in C and what is printed and the decision to print potentially depends on data of sensitivity l. Then, it is natural to ask, when it is secure to execute the effects of the inner computation of type $\mathsf{Eff}_C\, A$? Clearly, whenever the sensitivities of the computation's effects, i.e. the channels in the set C, are as high as the sensitivity of the data used to decide to perform the effects, i.e. sensitivity l. The side-condition of the rule exactly captures this condition: $\forall ch \in C.\ l \sqsubseteq \mathrm{label}(ch)$.

To illustrate DISTR in action, let us consider the following term in $\lambda_{\mathrm{SC}}^{\mathrm{PRINT}}$ that prints Alice's sensitive input to the H-sensitive channel:

$$
\begin{aligned}
&\mathit{prog}_6 : \mathsf{S}_{\mathtt{A}}\,\mathsf{Bool} \Rightarrow \mathsf{Eff}_{\{\mathtt{Ch}_{\mathtt{H}}\}}\,\mathsf{Unit}\\
&\mathit{prog}_6 = \lambda\, sb.\, \mathsf{distr}\,(\mathsf{seal}_{\mathtt{A}}\,(\mathsf{print}_{\mathtt{Ch}_{\mathtt{H}}}\,(\mathsf{unseal}_{\mathtt{A}}\, sb))) \gg\!= \lambda\, x.\, \mathsf{return}\,\mathsf{unit}
\end{aligned}
$$

The primitive distr permits to execute the effects of the computation inside the term $\mathsf{seal}_{\mathtt{A}}\,(\mathsf{print}_{\mathtt{Ch}_{\mathtt{H}}}\,(\mathsf{unseal}_{\mathtt{A}}\, sb))$. The term distr protects the return type of the computation at the same sensitivity as the premise's type $\mathsf{S}_l\, A$. This requirement is necessary to enforce IFC because the trivial computation that performs no effects and just returns has access to sensitive data, as exemplified by the following program:

$$
\begin{aligned}
&\mathit{prog}_7 : \mathsf{S}_{\mathtt{A}}\,\mathsf{Bool} \Rightarrow \mathsf{S}_{\mathtt{B}}\,\mathsf{Bool} \Rightarrow \mathsf{Eff}_{\{\mathtt{Ch}_{\mathtt{H}}\}}\,(\mathsf{S}_{\mathtt{H}}\,\mathsf{Bool})\\
&\mathit{prog}_7 =\\
&\quad \lambda\, sb_1\, sb_2.\, \mathsf{distr}\,(\mathsf{seal}_{\mathtt{H}}\,(\mathsf{print}_{\mathtt{Ch}_{\mathtt{H}}}\,(\mathsf{unseal}_{\mathtt{A}}\, sb_1)) \gg\!= \lambda\, x.\, \mathsf{return}\,(\mathsf{unseal}_{\mathtt{B}}\, sb_2))
\end{aligned}
$$

C.3.2. Global Store Effects

We turn our attention to global store effects which combines printing effects from the previous section with reading from locations in the store. Following the same steps, in Figure C.4 we present the extension of λ_{REC} (Appendix I) in which programs have access to a global store and can read from and write to it. We name this language $\lambda_{\mathrm{REC}}^{\mathrm{STORE}}$. Our development rests on two assumptions: (1) the set of locations in the store *Loc* is fixed during execution, and (2) only terms of ground type, i.e. Bool and Unit, can be stored. This helps simplify the presentation of the language and the security-type system, and, as we will show in the next section, the construction of the logical relation from which noninterference follows. At the end of this section, however, we briefly discuss how to lift these assumptions.

$\lambda_{\mathrm{REC}}^{\mathrm{STORE}}$ extends λ_{REC} with a type of computations $\mathsf{Eff}_S\, a$, which is indexed by a set of store locations $S \subseteq \mathit{Loc}$, and a type of references $\mathsf{Ref}_s\, r$, which is indexed by store locations $s \in S$. A program of type $\mathsf{Eff}_S\, a$ when executed might *only* write to the references mentioned in the set S and finally return

a result of type a. A term of type $\mathsf{Ref}_s\, r$ is a reference in the store s that contains terms of ground type r. References permit both "printing" effects via writing—like channels in $\lambda_{\text{REC}}^{\text{PRINT}}$—but also reading effects. Note that the annotation S in the type of computations Eff_S *does not* mention the locations on the store from which the program might read. This asymmetry stems from how the execution of programs performing these effects interact with their environment: writing alters the store whilst reading does not.

Statics Typing judgements are of the form $\Sigma, \Gamma \vdash t : a$ where the store typing Σ determines the shape of the store, i.e. what types it contains and in what locations. The rest of components are like those in λ_{REC}. The typing rules of the monadic operations return, bind and subeff follow the same pattern as in $\lambda_{\text{REC}}^{\text{PRINT}}$ (Figure C.2), thus we do not discuss them any further. The term ref_s (Rule Ref) is the runtime representation of references. Reading and writing is achieved via primitives read and write (Rules Read and Write respectively). Observe that in Read, the type of the computation $\mathsf{Eff}_\emptyset\, r$ in the conclusion of the rule is indexed by the empty set of locations, while in Write is indexed by the singleton set $\{s\}$.

Dynamics The operational semantics of the language is defined as in $\lambda_{\text{REC}}^{\text{PRINT}}$; a combination of the small-step semantics for λ_{REC} that treats effectful primitives as values, and a small-step semantics for computations. Given a store typing Σ, a store θ is a function from locations to typed terms according to Σ. Since the language considers a fixed-size store, we use the notation θ instead of $\theta(\Sigma)$. The semantics of computations is of the form $\theta_1, t \rightsquigarrow \theta_2, u$, and is interpreted as: program $\Sigma, \cdot \vdash t : \mathsf{Eff}_S\, a$ paired with store θ_1 evaluates in one step to program $\Sigma, \cdot \vdash u : \mathsf{Eff}_S\, a$ and store θ_2.

We now explain the semantics. Rule Read reduces the argument of read and it does not modify the store. When the argument is a store location, i.e. ref_s, Read-Ref retrieves the term t from the store, i.e. $\theta(s) = t$. The rules for writing Rules Write and Write-Ref first reduce the left subterm of $\mathsf{write}\, u\, t$ to a store location and then write the right subterm t on the store. Different from $\lambda_{\text{REC}}^{\text{PRINT}}$, we permit to write any term on the store, not only values.

To briefly illustrate $\lambda_{\text{REC}}^{\text{STORE}}$, consider the following program:

$$
\begin{aligned}
&\mathit{prog}_8 : \mathsf{Bool} \Rightarrow \mathsf{Eff}_{\{s'\}}\, \mathsf{Unit} \\
&\mathit{prog}_8\, b = \mathsf{if}\ b\ \mathsf{then}\ (\mathsf{read}\, s \gg\!= \lambda x.\, \mathsf{write}\, s'\, x)\ \mathsf{else}\ \mathsf{return}\, \mathsf{unit}
\end{aligned}
$$

Based on the Boolean input, prog_8 copies the contents of the store location s to s'. Observe that the type only mentions the location s' in its index.

$$
\begin{array}{rr}
\text{Store locations} & s \in \mathit{Loc} \\
\text{Sets of s. locations} & S, S_1, S_2 \subseteq \mathit{Loc} \\
\text{Ground types} & r ::= \mathsf{Bool} \mid \mathsf{Unit} \\
\text{Types} & a, b ::= r \mid a \Rightarrow b \mid \mathsf{Eff}_S\, a \mid \mathsf{Ref}_s\, r \\
\text{Typing contexts} & \Gamma ::= \ldots \\
\text{Store typings} & \Sigma \in \mathit{Loc} \to \text{Ground types}
\end{array}
$$

$\boxed{\Sigma, \Gamma \vdash t : a}$

$$\textsc{Ref}\ \frac{\Sigma(s) = r}{\Sigma, \Gamma \vdash \mathsf{ref}_s : \mathsf{Ref}_s\, r} \qquad \textsc{Read}\ \frac{\Sigma, \Gamma \vdash t : \mathsf{Ref}_s\, r}{\Sigma, \Gamma \vdash \mathsf{read}\, t : \mathsf{Eff}_\emptyset\, r}$$

$$\textsc{Write}\ \frac{\Sigma, \Gamma \vdash t : \mathsf{Ref}_s\, r \qquad \Sigma, \Gamma \vdash u : r}{\Sigma, \Gamma \vdash \mathsf{write}\, t\, u : \mathsf{Eff}_{\{s\}}\, \mathsf{Unit}}$$

$$\text{Stores} \quad \theta, \theta_1, \theta_2 \quad \in (\Sigma : \text{Store typing}) \to (s : \mathit{Loc}) \to \Sigma\ , \cdot \vdash t : \Sigma(s)$$

$\boxed{\theta_1(\Sigma), t \rightsquigarrow \theta_2(\Sigma), u \text{ with } \Sigma, \cdot \vdash t : \mathsf{Eff}_S\, a \text{ and } \Sigma, \cdot \vdash u : \mathsf{Eff}_S\, a}$

$$\textsc{Read}\ \frac{t \to u}{\theta, \mathsf{read}\, t \rightsquigarrow \theta, \mathsf{read}\, u} \qquad \textsc{Read-Ref}\ \frac{\theta(s) = t}{\theta, \mathsf{read}\, \mathsf{ref}_s \rightsquigarrow \theta, \mathsf{return}\, t}$$

$$\textsc{Write}\ \frac{t \to t'}{\theta, \mathsf{write}\, t\, u \rightsquigarrow \theta, \mathsf{write}\, t'\, u} \qquad \textsc{Write-Ref}\ \frac{\theta_2 = \theta_1[s \mapsto t]}{\theta_1, \mathsf{write}\, \mathsf{ref}_s\, t \rightsquigarrow \theta_2, \mathsf{return}\, \mathsf{unit}}$$

Figure C.4.: Types, well-typed terms and small-step semantics of $\lambda^{\text{STORE}}_{\text{REC}}$ (omitting the unchanged rules of Appendix I and Figure C.2)

We conclude the explanation of $\lambda^{\text{STORE}}_{\text{REC}}$ with a confinement lemma similar to that of $\lambda^{\text{PRINT}}_{\text{REC}}$ (Lemma C.3.1):

Lemma C.3.2 (Confinement for $\lambda^{\text{STORE}}_{\text{REC}}$)**.** *For any λ^{STORE}_{REC} program $\Sigma, \cdot \vdash f : \mathsf{Eff}_S\, a$, λ^{STORE}_{REC} value $\Sigma, \cdot \vdash v : \mathsf{Eff}_S\, a$, and stores $\theta_2, \theta_2 : \Sigma$, if $\theta_1, f \rightsquigarrow^* \theta_2, v$ then $\forall s \in Loc.\, s \notin S \Rightarrow \theta_1(s) = \theta_2(s)$.*

Security-Type System Now we explain the IFC enforcement mechanism $\lambda^{\text{STORE}}_{\text{SC}}$ (Figure C.5). As in $\lambda^{\text{PRINT}}_{\text{SC}}$ (Figure C.3) we assume that the security policy specifies for each store location its sensitivity as a function label $\in Loc \to L$. The typing rules for the monadic primitives are analogous to $\lambda^{\text{PRINT}}_{\text{SC}}$ thus we have omitted them. The rule for references is straightforward (Rule Ref). More interesting is the typing rule for reading from the store (Rule Read). In the conclusion of the rule, the return type of the computation is the λ_{SC} type for sensitive data $\mathsf{S}_l\, R$. The sensitivity of the location is l, i.e. $\text{label}(s) = l$ in the side-condition of the rule, thus to protect the flow of information is necessary to wrap also the return type.

The type of read diverges from usual presentations of IFC libraries (e.g. MAC [Rus15] and HLIO [BVR15] with the exception of SLIO [RG20]) in that the result of reading from the store is wrapped in the type constructor S_l. These libraries incorporate the data into their monad of computations, which keeps track of the sensitivities of the observed values.

We conclude the section with a concrete example of $\lambda^{\text{STORE}}_{\text{SC}}$ that shows that $prog_8$ is secure with respect to the following specification: $\text{label}(s) = \mathtt{A}$, $\text{label}(s') = \mathtt{H}$ and the sensitivity of the Boolean argument is $\mathtt{H}$, i.e. $sb : \mathsf{S}_{\mathtt{H}}\, \mathsf{Bool}$:

$$
\begin{aligned}
&prog'_8 : \mathsf{S}_{\mathtt{H}}\, \mathsf{Bool} \Rightarrow \mathsf{Eff}_{\{s'\}}\, (\mathsf{S}_{\mathtt{H}}\, \mathsf{Unit}) \\
&prog'_8\ sb = \mathsf{distr}(\mathsf{seal}_{\mathtt{H}}(\mathsf{if}\ \mathsf{unseal}_{\mathtt{A}}\ sb \\
&\qquad\qquad\qquad\qquad \mathsf{then}\ (\mathsf{read}\ s \gg\!= \lambda x.\, \mathsf{write}\ s'\ (\mathsf{unseal}_{\mathtt{H}}\ x)) \\
&\qquad\qquad\qquad\qquad \mathsf{else}\ \mathsf{return}\ \mathsf{unit}))
\end{aligned}
$$

The above program exemplifies how $\lambda^{\text{STORE}}_{\text{SC}}$ enforces that flows from the store to the program and back to the store are secure. Note that $\varepsilon(prog'_8) = prog_8$.

Lifting the Assumptions on the Store

To conclude this section, we briefly discuss how to lift the initial assumptions, i.e. (1) and (2), on the store. Since $\lambda^{\text{STORE}}_{\text{REC}}$ and $\lambda^{\text{STORE}}_{\text{SC}}$ already support the definition of recursive functions, removing the constraint of ground types on the store, i.e. making it higher-order, is straightforward and does not change the expressivity of the language. To permit dynamically allocated locations,

$$
\begin{array}{rr}
\text{Store locations} & s \in \mathit{Loc} \\
\text{Sets of s. locations} & S, S_1, S_2 \subseteq \mathit{Loc} \\
\text{Sensitivity of loc.} & \mathrm{label} \in \mathit{Loc} \to L \\
\text{Sets of labels} & \pi \subseteq L \\
\text{Ground types} & R ::= \ \mathsf{Bool} \mid \mathsf{Unit} \\
\text{Types} & A, B ::= \ R \mid A \Rightarrow B \mid \mathsf{Eff}_S\, A \mid \mathsf{Ref}_s\, R \\
\text{Typing contexts} & \Gamma ::= \ \ldots \\
\text{Store typings} & \Sigma \in \mathit{Loc} \to \text{Ground type}
\end{array}
$$

$\boxed{\pi \mathbin{;} \Sigma, \Gamma \vdash^{\mathrm{SC}} t : A}$

$$
\textsc{Ref}\ \frac{\Sigma(s) = R}{\pi \mathbin{;} \Sigma, \Gamma \vdash^{\mathrm{SC}} \mathsf{ref}_s : \mathsf{Ref}_s\, R} \qquad \textsc{Read}\ \frac{\pi \mathbin{;} \Sigma, \Gamma \vdash^{\mathrm{SC}} t : \mathsf{Ref}_s\, R}{\pi \mathbin{;} \Sigma, \Gamma \vdash^{\mathrm{SC}} \mathsf{read}\, t : \mathsf{Eff}_{\emptyset}\, (\mathsf{S}_l\, R)}\ (\mathrm{label}(s) = l)
$$

$$
\textsc{Write}\ \frac{\pi \mathbin{;} \Sigma, \Gamma \vdash^{\mathrm{SC}} t : \mathsf{Ref}_s\, R \qquad \pi \mathbin{;} \Sigma, \Gamma \vdash^{\mathrm{SC}} u : R}{\pi \mathbin{;} \Sigma, \Gamma \vdash^{\mathrm{SC}} \mathsf{write}\, t\, u : \mathsf{Eff}_{\{s\}}\, \mathsf{Unit}}
$$

$$
\textsc{Distr}\ \frac{\pi \mathbin{;} \Sigma, \Gamma \vdash^{\mathrm{SC}} t : \mathsf{S}_l\, (\mathsf{Eff}_S\, A)}{\pi \mathbin{;} \Sigma, \Gamma \vdash^{\mathrm{SC}} \mathsf{distr}\, t : \mathsf{Eff}_S\, (\mathsf{S}_l\, A)}\ (\forall s \in S.\ l \sqsubseteq \mathrm{label}(s))
$$

Figure C.5.: Types and intrinsically-typed terms of $\lambda_{\mathrm{SC}}^{\mathrm{STORE}}$ (omitting the unchanged rules of Figure C.1)

the type of computations $\mathsf{Eff}_S\, a$ changes so that the annotation S records memory regions, i.e. an abstraction over sets of locations, where the program might write—see e.g. Lucassen and Gifford [LG88] or Tofte and Talpin [TT97]. New references are allocated in a memory region statically assigned by the programmer, and the type of the reference tracks such region. Then, the security-type system is similar to $\lambda_{\mathrm{SC}}^{\mathrm{STORE}}$, but it assigns security labels to regions instead of concrete locations.

C.3.3. Other Effects, Combination of Effects

To conclude we note that the previous sections show how to treat, from an IFC perspective, reading and writing effects in a general sense. With such a development, our approach could be used to not only encode secure reading/writing of references but also effects involving files and network communications. However, our framework can be adapted to deal with other kinds of effects, e.g. exceptions, or combinations thereof by selecting what effects the graded monad has to track, e.g. what exceptions are thrown, and when it it secure to extract a computation from a sealed value.

C.4. Security Guarantees

In this section, we prove that the IFC mechanisms λ_{SC}, $\lambda_{\mathrm{SC}}^{\mathrm{PRINT}}$ and $\lambda_{\mathrm{SC}}^{\mathrm{STORE}}$ can be used to enforce noninterference for the programming languages λ_{REC}, $\lambda_{\mathrm{REC}}^{\mathrm{PRINT}}$ and $\lambda_{\mathrm{REC}}^{\mathrm{STORE}}$ presented in Sections C.2, C.3.1 and C.3.2, respectively. Our noninterference proofs employ the technique of step-indexed logical relations (LRs) [Ahm06]. For each language, we construct a step-indexed LR parameterized by the sensitivity of the attacker `Atk` and the security types of λ_{SC}. In the effect-free setting, the LR interprets each λ_{SC} type A as a binary relation over λ_{REC} programs of the erased type $\varepsilon(A)$. The relation captures the idea of indistinguishable programs: if the type is *public* enough, e.g. $\mathsf{S}_{\mathtt{L}}\,\mathsf{Bool}$ and `L` flows to `Atk`, then two programs are related when they evaluate to the same value. Noninterference follows as a corollary of the so called fundamental theorem of the LR.

Step-indexing is a technical device that helps in proving that programs defined as recursive functions belong to the relation by looking at their finite unrollings. To ease the explanation, we use grey to display the step-indexing machinery, and sometimes we completely ignore it.

Following previous work [RG20; Gre+21], we show that the security-type systems enforce variants of termination-insensitive noninterference (TINI). At the end of this section, we briefly discuss alternative security conditions. TINI

states that when two runs of a program with different secret inputs terminate, then its public outputs agree, and thus the attacker cannot use nontermination as a channel to infer the contents of sensitive data. In the effect-free setting, the inputs to a program are its arguments, and the output is its return value. In the effectful setting, what we need to consider as inputs and outputs changes: in $\lambda_{\text{REC}}^{\text{STORE}}$, for instance, the store must be considered an additional input to the program.

Definition C.4.1 (TINI for λ_{REC})**.** A program $sec : \mathsf{Bool} \vdash p : \mathsf{Bool}$ satisfies TINI if for any two terms $\cdot \vdash s_1, s_2 : \mathsf{Bool}$, and any two values $\cdot \vdash v_1, v_2 : \mathsf{Bool}$, if $p[s_1/sec] \to^* v_1$ and $p[s_2/sec] \to^* v_2$ then $v_1 = v_2$.

In the definition $v_1 = v_2$ denotes that v_1 and v_2 are syntactically equal values. Note that programs that diverge for any input vacuously satisfy TINI: the assumption that the substituted programs terminate will never hold.

λ_{SC} enforces TINI:

Theorem C.4.1. *Given any λ_{SC} term $\{\mathtt{Atk}\}$; $sec : \mathsf{S}_{\mathtt{H}}\,\mathsf{Bool} \vdash^{SC} t : \mathsf{S}_{\mathtt{Atk}}\,\mathsf{Bool}$ where $\mathtt{H} \not\sqsubseteq \mathtt{Atk}$, the erased program $sec : \mathsf{Bool} \vdash \varepsilon(t) : \mathsf{Bool}$ satisfies TINI.*

In practice, we are concerned that information does not flow from H-protected data to the attacker with sensitivity Atk. Thus, to show that a program $sec : \mathsf{Bool} \vdash p : \mathsf{Bool}$ does not leak information from H to Atk, it is enough to find an λ_{SC} term $\{\mathtt{Atk}\}$; $sec : \mathsf{S}_{\mathtt{H}}\,\mathsf{Bool} \vdash^{\text{SC}} t : \mathsf{S}_{\mathtt{Atk}}\,\mathsf{Bool}$ such that $p = \varepsilon(t)$.

Logical Relation In order to prove that λ_{SC} enforces TINI (Theorem C.4.1), we construct an step-indexed LR parameterized by the attacker's sensitivity Atk and the λ_{SC} types—see Figure C.6. The proof, as we will show, then falls out as a consequence of the fundamental theorem of the LR.

At each λ_{SC} type the LR defines what the attacker can observe about pairs of λ_{REC} programs of erased type—alternatively, the same program with different secrets. We split the definition depending on whether programs are evaluated, i.e. they are already values, $\mathcal{R}_{\mathcal{V}}^{\mathtt{Atk}}[\![-]\!]$ and $\mathcal{R}_{\mathcal{C}}^{\mathtt{Atk}}[\![-]\!]$, respectively. We briefly explain these definitions. At λ_{SC} type Bool, for instance, see $\mathcal{R}_{\mathcal{C}}^{\mathtt{Atk}}[\![\mathsf{Bool}]\!]$ and $\mathcal{R}_{\mathcal{V}}^{\mathtt{Atk}}[\![\mathsf{Bool}]\!]$, the LR states that if the programs terminate then the attacker can observe if they return equal values. At higher types, i.e. $A \Rightarrow B$, two functions are related if whenever they reduce to a value, see $\mathcal{R}_{\mathcal{C}}^{\mathtt{Atk}}[\![A \Rightarrow B]\!]$, they map related inputs $\mathcal{R}_{\mathcal{C}}^{\mathtt{Atk}}[\![A]\!](u_1, u_2)$ to related outputs $\mathcal{R}_{\mathcal{C}}^{\mathtt{Atk}}[\![A]\!](\mathsf{app}\,t_1\,u_1, \mathsf{app}\,t_2\,u_2)$, see $\mathcal{R}_{\mathcal{V}}^{\mathtt{Atk}}[\![A \Rightarrow B]\!]$. Lastly, at type $\mathsf{S}_l\,A$, see $\mathcal{R}_{\mathcal{V}}^{\mathtt{Atk}}[\![\mathsf{S}_l\,A]\!]$, the LR compares the sensitivity of the attacker with the label l, and in case it is less sensitive, i.e. $l \sqsubseteq \mathtt{Atk}$, the programs have to be related at λ_{SC} type A. If the label l is more

$$\mathcal{R}^{\mathtt{Atk}}_{\mathcal{V}}[\![-]\!] \in (A : \lambda_{\mathrm{SC}}\ \mathrm{type}) \to \mathbb{N} \to (t_1 : \cdot \vdash \varepsilon(A)) \to (t_2 : \cdot \vdash \varepsilon(A)) \to \mathrm{Set}$$
$$\mathcal{R}^{\mathtt{Atk}}_{\mathcal{V}}[\![\mathsf{Unit}]\!]_n(t_1, t_2) :\Leftrightarrow t_1 = t_2$$
$$\mathcal{R}^{\mathtt{Atk}}_{\mathcal{V}}[\![\mathsf{Bool}]\!]_n(t_1, t_2) :\Leftrightarrow t_1 = t_2$$
$$\mathcal{R}^{\mathtt{Atk}}_{\mathcal{V}}[\![A \Rightarrow B]\!]_n(t_1, t_2) :\Leftrightarrow \forall(m : \mathbb{N}).\, m \leqslant n.$$
$$\quad \forall(u_1, u_2 : \cdot \vdash \varepsilon(A)).\, \mathcal{R}^{\mathtt{Atk}}_{\mathcal{C}}[\![A]\!]_m(u_1, u_2) \Rightarrow \mathcal{R}^{\mathtt{Atk}}_{\mathcal{C}}[\![B]\!]_m(\mathsf{app}\, t_1\, u_1, \mathsf{app}\, t_2\, u_2)$$
$$\mathcal{R}^{\mathtt{Atk}}_{\mathcal{V}}[\![\mathsf{S}_l\, A]\!]_n(t_1, t_2) :\Leftrightarrow l \sqsubseteq \mathtt{Atk} \Rightarrow \mathcal{R}^{\mathtt{Atk}}_{\mathcal{V}}[\![A]\!]_n(t_1, t_2)$$

$$\mathcal{R}^{\mathtt{Atk}}_{\mathcal{C}}[\![A]\!]_n(t_1, t_2) :\Leftrightarrow \forall(m_1, m_2 : \mathbb{N}).\, m_1 + m_2 < n.$$
$$\quad \forall(u_1, u_2 : \cdot \vdash \varepsilon(A)).\, t_1 \to^*_{m_1} u_1 \wedge t_2 \to^*_{m_2} u_2 \Rightarrow \mathcal{R}^{\mathtt{Atk}}_{\mathcal{V}}[\![A]\!]_{n-(m_1+m_2)}(u_1, u_2)$$

$$\mathcal{R}^{\mathtt{Atk}}_{\mathcal{S}}[\![\cdot]\!]_n(\epsilon, \epsilon) :\Leftrightarrow \top$$
$$\mathcal{R}^{\mathtt{Atk}}_{\mathcal{S}}[\![\Gamma, x : A]\!]_n((\gamma_1, t_1), (\gamma_2, t_2)) :\Leftrightarrow \mathcal{R}^{\mathtt{Atk}}_{\mathcal{C}}[\![A]\!]_n(t_1, t_2) \wedge \mathcal{R}^{\mathtt{Atk}}_{\mathcal{S}}[\![\Gamma]\!]_n(\gamma_1, \gamma_2)$$

$$\mathcal{R}^{\mathtt{Atk}}_{\mathcal{T}}[\![(\Gamma, A)]\!]_n(t_1, t_2) :\Leftrightarrow$$
$$\quad \forall(\gamma_1, \gamma_2 : \cdot \vdash \varepsilon(\Gamma)).\, \mathcal{R}^{\mathtt{Atk}}_{\mathcal{S}}[\![\Gamma]\!]_n(\gamma_1, \gamma_2) \Rightarrow \mathcal{R}^{\mathtt{Atk}}_{\mathcal{C}}[\![A]\!]_n(t_1[\gamma_1], t_2[\gamma_2])$$

Figure C.6.: Logical relation for λ_{REC}-λ_{SC}

sensitive than the attacker's label, i.e. $l \not\sqsubseteq \mathtt{Atk}$ then the programs do not need to be related.

Definitions $\mathcal{R}^{\mathtt{Atk}}_{\mathcal{V}}[\![-]\!]$ and $\mathcal{R}^{\mathtt{Atk}}_{\mathcal{C}}[\![-]\!]$ work on closed terms, however, in order to prove the fundamental theorem we have to lift them to closed substitutions and open terms. A substitution assigns to each type a in a typing context Γ a closed term of that type, i.e. $\cdot \vdash t : a$. We denote substitutions by γ and use $\gamma : \cdot \vdash \Gamma$ to mean that γ is in the set of substitutions over Γ. We define the LR for substitutions, $\mathcal{R}^{\mathtt{Atk}}_{\mathcal{S}}[\![-]\!]$, by induction on λ_{SC} typing contexts. At the empty context $\cdot$ the empty substitutions (ϵ, ϵ) are trivially related—denoted by $\top$. Two nonempty substitutions (γ_1, t_1) and (γ_2, t_2) are related whenever they are pointwise related, i.e. $\mathcal{R}^{\mathtt{Atk}}_{\mathcal{C}}[\![A]\!](t_1, t_2)$ and $\mathcal{R}^{\mathtt{Atk}}_{\mathcal{S}}[\![\Gamma]\!](\gamma_1, \gamma_2)$. The LR for open terms, written $\mathcal{R}^{\mathtt{Atk}}_{\mathcal{T}}[\![(\Gamma, A)]\!]$, is indexed by a pair consisting of an λ_{SC} typing context Γ and an λ_{SC} type A. It states that two λ_{REC} terms are related if for any two related closing substitutions the substituted terms are related at type A.

The fundamental theorem of the LR states that the underlying program of

an λ_{SC} term is related to itself. Formally:

Theorem C.4.2 (Fundamental Theorem of the LR for λ_{REC}-λ_{SC})**.** *For any attacker with sensitivity* $\mathtt{Atk}$*, and* λ_{SC} *term* $\{\mathtt{Atk}\} ; \Gamma \vdash^{SC} t : A$*, it is the case that for all* $n : \mathbb{N}$*,* $\mathcal{R}_{\mathcal{T}}^{\mathtt{Atk}}[\![(\Gamma, A)]\!]_n(\varepsilon(t), \varepsilon(t))$*.*

Proof. By induction on the the typing derivation and the step-index n. □

TINI (Theorem C.4.1) follows as a corollary of the fundamental theorem:

Proof. Assume two Boolean secrets $\cdot \vdash s_1, s_2 : \mathsf{Bool}$. Since $\mathtt{H} \not\sqsubseteq \mathtt{Atk}$, the secrets are related, i.e. $\mathcal{R}_{\mathcal{C}}^{\mathtt{Atk}}[\![\mathsf{S}_{\mathtt{H}}\,\mathsf{Bool}]\!](s_1, s_2)$, and thus the two substitutions $\gamma_1 = \{sec \mapsto s_1\}$ and $\gamma_2 = \{sec \mapsto s_2\}$ are related. By the fundamental theorem, the term $\varepsilon(t)$ is related to itself, i.e. $\mathcal{R}_{\mathcal{C}}^{\mathtt{Atk}}[\![\mathsf{S}_{\mathtt{Atk}}\,\mathsf{Bool}]\!](\varepsilon(t)[\gamma_1], \varepsilon(t)[\gamma_2])$. Unfolding the definitions of we obtain that $\forall(v_1, v_2 : \cdot \vdash \mathsf{Bool}).\, \varepsilon(t)[\gamma_1] \to^* v_1 \wedge \varepsilon(t)[\gamma_2] \to^* v_2 \Rightarrow v_1 = v_2$. □

C.4.1. Noninterference for Printing Effects

In order to formalize noninterference for $\lambda_{\text{REC}}^{\text{PRINT}}$ we look at effectful programs that might print. For that, we first define indistinguishability of outputs with respect to a subset of channels:

Definition C.4.2 (Output indistinguishability)**.** Let $C \subseteq Ch$. Two outputs o_1 and o_2 are C-indistinguishable, denoted by $o_1 =_C o_2$, if the two outputs agree in C, i.e. $o_1|_C = o_2|_C$.

We define TINI for $\lambda_{\text{REC}}^{\text{PRINT}}$ for programs from Bool to $\mathsf{Eff}_C\,\mathsf{Unit}$, i.e. programs that depending on a Boolean produce output in an arbitrary set of channels C:

Definition C.4.3 (TINI for $\lambda_{\text{REC}}^{\text{PRINT}}$)**.** A program $sec : \mathsf{Bool} \vdash p : \mathsf{Eff}_C\,\mathsf{Unit}$ satisfies TINI *with respect to* $C' \subseteq C$, if for any two terms $\cdot \vdash s_1, s_2 : \mathsf{Bool}$, any two values $\cdot \vdash v_1, v_2 : \mathsf{Eff}_C\,\mathsf{Unit}$, and any two outputs o_1 and o_2, if $p[s_1/sec] \rightsquigarrow^* v_1, o_1$ and $p[s_2/sec] \rightsquigarrow^* v_2, o_2$ then $o_1 =_{C'} o_2$.

TINI is parameterized by a subset of the channels where the outputs have to agree. We will instantiate C' with the set of channels observable by the attacker.

$\lambda_{\text{SC}}^{\text{PRINT}}$ can be used to enforce TINI on $\lambda_{\text{REC}}^{\text{PRINT}}$ programs:

Theorem C.4.3. *Given any* λ_{SC}^{PRINT} *term* $\{\mathtt{Atk}\} ; sec : \mathsf{S}_{\mathtt{H}}\,\mathsf{Bool} \vdash^{SC} t : \mathsf{Eff}_C\,\mathsf{Unit}$ *where* $\mathtt{H} \not\sqsubseteq \mathtt{Atk}$ *the erased program* $sec : \mathsf{Bool} \vdash \varepsilon(t) : \mathsf{Eff}_C\,\mathsf{Unit}$ *satisfies TINI* with respect to $C_{\mathtt{Atk}}$ *where* $C_{\mathtt{Atk}} \coloneqq \{ch \mid ch \in C, \text{label}(ch) \sqsubseteq \mathtt{Atk}\}$.

$$\mathcal{R}^{\mathtt{Atk}}_{\mathcal{V}}[\![\ldots]\!]_n(t_1,t_2) :\Leftrightarrow \ldots$$
$$\mathcal{R}^{\mathtt{Atk}}_{\mathcal{V}}[\![\mathsf{Eff}_C\, A]\!]_n(t_1,t_2) :\Leftrightarrow$$
$$\forall(m_1,m_2:\mathbb{N}).\, m_1+m_2<n.\, \forall(u_1,u_2:\cdot\vdash\varepsilon(A)), o_1, o_2.$$
$$t_1 \rightsquigarrow^*_{m_1} \mathsf{return}(u_1), o_1 \wedge t_2 \rightsquigarrow^*_{m_2} \mathsf{return}(u_2), o_2$$
$$\Rightarrow \mathcal{R}^{\mathtt{Atk}}_{\mathcal{C}}[\![A]\!]_{n-(m_1+m_2)}(u_1,u_2) \wedge o_1 =_{C_{\mathtt{Atk}}} o_2$$

Figure C.7.: Logical relation for $\lambda^{\text{PRINT}}_{\text{REC}}$-$\lambda^{\text{PRINT}}_{\text{SC}}$ (omitting the unchanged clauses of Figure C.6)

Logical Relation The LR for $\lambda^{\text{PRINT}}_{\text{REC}}$ (Figure C.7) is very similar to that of λ_{REC} (Figure C.6) so we skip over the commonalities and directly discuss its definition at the type of computations. The LR relates two computations $\mathcal{R}^{\mathtt{Atk}}_{\mathcal{V}}[\![\mathsf{Eff}_C\, A]\!](t_1,t_2)$ if whenever they terminate, i.e. $t_1 \rightsquigarrow^* \mathsf{return}\, u_1, o_1$ and $t_2 \rightsquigarrow^* \mathsf{return}\, u_2, o_2$, the resulting terms are related, i.e. $\mathcal{R}^{\mathtt{Atk}}_{\mathcal{C}}[\![\tau]\!](u_1,u_2)$, and the outputs are indistinguishable by the attacker, i.e. $o_1 =_{C_{\mathtt{Atk}}} o_2$. The fundamental theorem of the LR states that erased terms are related to themselves:

Theorem C.4.4 (Fundamental Theorem of the LR for $\lambda^{\text{PRINT}}_{\text{REC}}$-$\lambda^{\text{PRINT}}_{\text{SC}}$)**.** *For any attacker with sensitivity* $\mathtt{Atk}$*, and* λ^{PRINT}_{SC} *term* $\{\mathtt{Atk}\}\,;\Gamma\vdash^{SC} t : A$*, it is the case that* $\mathcal{R}^{\mathtt{Atk}}_{\mathcal{T}}[\![(\Gamma,A)]\!](\varepsilon(t),\varepsilon(t))$*.*

Proof. By induction on the typing derivation with use of the Lemma C.3.1. □

The proof that $\lambda^{\text{PRINT}}_{\text{SC}}$ enforces TINI (Theorem C.4.3) requires the following Lemma.

Lemma C.4.1. *If* $p \rightsquigarrow^* v, o$*, then there exists a* $\rightarrow$*-value* p' *such that* $p \rightarrow^* p'$ *and* $p' \rightsquigarrow^* v, o$*.*

TINI follows as a corollary of the fundamental theorem.

Proof. Let us assume two secret Booleans $\cdot\vdash s_1, s_2 : \mathsf{Bool}$. Since $\mathtt{H} \not\sqsubseteq \mathtt{Atk}$, the secrets are related, i.e. $\mathcal{R}^{\mathtt{Atk}}_{\mathcal{C}}[\![\mathsf{S}_{\mathtt{H}}\,\mathsf{Bool}]\!](s_1,s_2)$, and thus the substitutions $\gamma_1 = \{sec \mapsto s_1\}$ and $\gamma_2 = \{sec \mapsto s_2\}$ are related. By the fundamental theorem, the term $\varepsilon(t)$ is related to itself $\mathcal{R}^{\mathtt{Atk}}_{\mathcal{C}}[\![\mathsf{Eff}_C\,\mathsf{Unit}]\!](\varepsilon(t)[\gamma_1],\varepsilon(t)[\gamma_2])$.

By assumption $\varepsilon(t)[s_1/sec] \rightsquigarrow^* v_1, o_1$ and $\varepsilon(t)[s_2/sec] \rightsquigarrow^* v_2', o_2$, and by Lemma C.4.1, there are two intermediate programs t_1' and t_2' such that: $\varepsilon(t)[s_1/sec] \rightarrow^* t_1'$ and $t_1' \rightsquigarrow^* v_1, o_1$; and $\varepsilon(t)[s_2/sec] \rightarrow^* t_2'$ and $t_2' \rightsquigarrow^* v_2, o_2$.

We apply $\mathcal{R}_{\mathcal{C}}^{\mathtt{Atk}}[\![\mathsf{Eff}_C\ \mathsf{Unit}]\!](\varepsilon(t)[s_1/sec], \varepsilon(t)[s_2/sec])$ to the two effect-free reductions which gives us that $\mathcal{R}_{\mathcal{V}}^{\mathtt{Atk}}[\![\mathsf{Eff}_C\ \mathsf{Unit}]\!](t_1', t_2')$. We apply this to the monadic reductions and obtain that $o_1 =_{C_{\mathtt{Atk}}} o_2$. □

C.4.2. Noninterference for Global Store Effects

We formalize noninterference for $\lambda_{\text{REC}}^{\text{STORE}}$ by looking at effectful programs which receive a store as input and produce a store as output. The contents of the store are possibly unevaluated $\lambda_{\text{REC}}^{\text{STORE}}$ programs of ground type (see Figure C.4). In order to compare stores, we define an indistinguishability relation for programs of ground type:

$$\mathcal{R}_{\mathcal{G}}[\![-]\!] : (r : \lambda_{\text{REC}}^{\text{STORE}} \text{ ground type}) \to \mathbb{N} \to (t_1 : \cdot \vdash r) \to (t_2 : \cdot \vdash r) \to \text{Set}$$
$$\mathcal{R}_{\mathcal{G}}[\![r]\!]_n(t_1, t_2) :\Leftrightarrow$$
$$\forall(m_1, m_2 : \mathbb{N}).\, m_1 + m_2 < n.\, \forall(v_1, v_2 : \cdot \vdash r).$$
$$t_1 \to^*_{m_1} v_1 \wedge t_2 \to^*_{m_2} v_2 \Rightarrow v_1 = v_2$$

In some sense it resembles the LR at Unit and Bool types in Figure C.6.

Stores are parameterized by store typings that determine the type of the contents at each location. Since stores neither grow nor shrink, assumption (1) (Section C.3.2), we define an indistinguishability relation for stores of the same store typing. Indistinguishability is parameterized by a subset of the locations.

Definition C.4.4 (Store indistinguishability)**.** Let $S \subseteq Loc$. Two stores θ_1 and θ_2 of store typing $\Sigma(s)$ are S-indistinguishable, denoted by $\theta_1 =_S^n \theta_2$, if they are indistinguishable at each location in S, i.e. $\forall s \in S.\ \mathcal{R}_{\mathcal{G}}[\![\Sigma(s)]\!]_n(\theta_1(s), \theta_2(s))$.

TINI for $\lambda_{\text{REC}}^{\text{STORE}}$ programs is:

Definition C.4.5 (TINI for $\lambda_{\text{REC}}^{\text{STORE}}$-$\lambda_{\text{SC}}^{\text{STORE}}$)**.** A program $\Sigma, \cdot \vdash p : \mathsf{Eff}_S\ \mathsf{Unit}$ satisfies TINI *with respect to* $S' \subseteq Loc$, if for any two stores $\theta_1, \theta_2 : \Sigma$, any two values $\cdot \vdash v_1, v_2 : \mathsf{Eff}_S\ \mathsf{Unit}$, and any two stores $\theta_1', \theta_2' : \Sigma$, if $\theta_1 =_S \theta_2$ and $\theta_1, p \rightsquigarrow^* \theta_1', v_1$ and $\theta_2, p \rightsquigarrow^* \theta_2', v_2$ then $\theta_1' =_S \theta_2'$.

Again, $\lambda_{\text{SC}}^{\text{STORE}}$ enforces TINI on $\lambda_{\text{REC}}^{\text{STORE}}$ programs:

Theorem C.4.5. *Given any* λ_{SC}^{STORE} *term* $\{\mathtt{Atk}\}\ ;\ \Sigma, \cdot \vdash^{SC} t : \mathsf{Eff}_S\ \mathsf{Unit}$ *the erased program* $\Sigma, \cdot \vdash \varepsilon(t) : \mathsf{Eff}_S\ \mathsf{Unit}$ *satisfies TINI* with respect to $S_{\mathtt{Atk}}$ *where* $S_{\mathtt{Atk}} := \{s \mid s \in Loc, \text{label}(s) \sqsubseteq \mathtt{Atk}\}$.

The LR that we construct to prove TINI (Figure C.8) is largely similar to that of $\lambda_{\text{REC}}^{\text{PRINT}}$ (Figures C.7 and C.6), with the difference that effectful programs take as argument and produce as result indistinguishable pairs of stores. TINI follows as a consequence of the fundamental theorem of the LR.

$$
\begin{aligned}
&\mathcal{R}^{\mathtt{Atk}}_{\mathcal{V}}[\![\ldots]\!]_n(t_1,t_2) :\Leftrightarrow \ldots\\
&\mathcal{R}^{\mathtt{Atk}}_{\mathcal{V}}[\![\mathsf{Eff}_C\, A]\!]_n(t_1,t_2) :\Leftrightarrow\\
&\quad \forall(m_1,m_2:\mathbb{N}).\, m_1+m_2<n.\, \forall(u_1,u_2:\cdot\vdash\varepsilon(A))(\theta_1,\theta_2,\theta_1',\theta_2':\Sigma).\\
&\qquad \theta_1 =_{S_{\mathtt{Atk}}} \theta_2 \wedge \theta_1, t_1 \rightsquigarrow^*_{m_1} \theta_1', \mathsf{return}\, u_1 \wedge \theta_2, t_2 \rightsquigarrow^*_{m_2} \theta_2', \mathsf{return}\, u_2\\
&\qquad\qquad \Rightarrow \mathcal{R}^{\mathtt{Atk}}_{\mathcal{C}}[\![A]\!]_{n-(m_1+m_2)}(u_1,u_2) \wedge \theta_1' =_{S_{\mathtt{Atk}}} \theta_2'
\end{aligned}
$$

Figure C.8.: Logical relation for $\lambda^{\text{STORE}}_{\text{REC}}$-$\lambda^{\text{STORE}}_{\text{SC}}$ (omitting the unchanged clauses of Figure C.6)

C.4.3. Other Security Properties

To conclude this section, we note that the security-type systems for $\lambda^{\text{PRINT}}_{\text{REC}}$ and $\lambda^{\text{STORE}}_{\text{REC}}$, in fact, enforce a stronger notion of security than TINI, namely progress-insensitive noninterference (PINI). Progress-insensitive noninterference (PINI) states that an attacker cannot infer the contents of sensitive data even if they have access to prefixes of the public outputs that nonterminating programs produce. In order to prove this stronger notion, the LRs require nontrivial generalizations that can deal with partial reduction sequences. We declare this line of research as future work.

C.5. Implementation

In this section, we present an implementation of λ_{SC} and $\lambda^{\text{PRINT}}_{\text{SC}}$ (Sections C.2 and C.3.1) as a Haskell library, which we call SCLib. We omit $\lambda^{\text{STORE}}_{\text{SC}}$ (Section C.3.2) for lack of space. However, its implementation is similar to that of $\lambda^{\text{PRINT}}_{\text{SC}}$. Furthermore, we demonstrate that existing Haskell libraries for static IFC can be reimplemented in terms of the interface that SCLib exposes.

The main characteristic of λ_{SC} is that typing judgements $\pi\,;\,\Gamma \vdash^{\text{SC}} t : A$ are indexed by a set of labels π. Onwards, we refer to the left part of the judgement, i.e. $\pi\,;\,\Gamma$, as the context of the term t. The set of labels plays an important role in enforcing IFC. However, individual labels are not first-class citizens: there is no type of labels and, hence, labels can neither be introduced nor eliminated. Further, some typing rules in λ_{SC} modify the set of labels in their context: e.g. the rule for unseal_l (cf. Figure C.1) augments the set in its premise with l. When shallowly embedding in Haskell any calculus that manipulates the context in this fashion, there is a natural problem to overcome: Haskell does not allow library implementors to have access to a program's context. For

```
module SCLib
  (Key (), Label (..), FlowsTo (..), S (S), seal, unseal, ...)
where
-- Enumeration of security labels for the two-point lattice
data Label = H | L
-- "Flows to" relation as a typeclass
class FlowsTo (l :: Label) (l' :: Label)
-- Instances
instance FlowsTo l l
instance FlowsTo L H
-- Type-level keys
data Key (l :: Label) = Key
-- Security type
data S l a = S (Key l -> a)
-- Sealing
seal :: (Key l -> a) -> S l a
seal = Seal
-- Unsealing
unseal :: FlowsTo l' l => Key l -> S l' a -> a
unseal k@Key (S f) = f Key
```

Figure C.9.: Implementation of λ_{SC} (e.g. for the two-point security lattice)

instance, to embed a linear type system, Bernardy, Boespflug, Newton, Jones, and Spiwack [Ber+17] need to change the compiler.

To overcome these difficulties, our implementation resorts to a combination of Haskell's module system to hide the implementation details' from users, and the use of Haskell's function space, i.e. abstraction and application, to manage the runtime representation of labels. We hope to convince the reader that our simple implementation matches the studied enforcement mechanisms, and that it can shed light on previous work on static IFC in Haskell.

C.5.1. Implementation of λ_{sc}

When we introduced λ_{SC}, we mentioned the intuition that labels in π are some sort of type-level keys whose possession permits access to sealed data. Our implementation takes this intuition literally: there is a type for keys whose elements are attached with type-level labels, and the primitive to unseal, i.e. unseal, is parameterized by a key. The elements of this type are like capabilities [DH83] which need to be explicitly exercised.

Figure C.9 shows the *complete* implementation of λ_{SC} in Haskell. Without

loss of generality, we assume the two-point security lattice. As previous work (e.g. MAC [Rus15], HLIO [BVR15], and DCC [AR17]) we represent labels as types of kind `Label` (line 5, and the use of the GHC extension `DataKinds`) and encode the "flows to" relation via a typeclass (lines 7–10). For simplicity we show the encoding of the two-point security lattice, however, this can be generalized (cf. [BVR15]). In line 12 we introduce a new datatype `Key` which is parameterized by a type `l` of kind `Label`. Then, line 14 introduces the type `S`, which is a wrapper over the function space between the types `Key l` and `a`, i.e. `Key l -> a`.

We now implement the primitives seal and unseal—Rules Seal and Unseal from Figure C.1. Rule Seal introduces a sealed value of type `S l a` from a value of type `a` that is typed in a context that has been extended by label `l`. Our implementation (lines 16 and 17), however, uses the function space `Key l -> a` to mimic the context extension by label `l`. Intuitively, the value of type `Key l` represents a proof that the label `l` is present in the context, and hence it can be used to construct the value of type `a` through `unseal`, which we explain next. Rule Unseal permits to eliminate sealed values of type `S l a` provided that the context contains a label `l'` secret enough, i.e. `l FlowsTo l'`. The combinator `unseal` (lines 19–20) allows unsealing terms of type `S l a` precisely in case we *have* a value of type `Key l'` and the label in its type, i.e. `l'`, flows to `l`—see the constraint `FlowsTo l' l`. In order to enforce IFC, it is important that the constructors of `Key` are kept abstract from the user—observe `Key ()` in the export list of the module (line 2). Otherwise, anyone could extract the underlying term of type `a` from an `l`-sensitive value `secret :: S l a` by applying `unseal Key`. Similar to Russo, Claessen, and Hughes [RCH08], the combinator `unseal` is strict in its argument (`k@Key`) in order to forbid forged keys like `undefined :: Key l`. We remark that the noninterference property—recall TINI from Definition C.4.1—rules out programs that force `undefined` and halt with error.

The implementation discussed so far consists of the trusted computing base (TCB) of SCLib. From now on, users of the library can derive functionalities from the library's interface. For example, programmers can implement the `Functor`, `Applicative` and `Monad` instances for the type `S l a`. For instance,

```
instance Functor (S l) where
  fmap f x = seal (\k -> f (unseal k x))

instance Applicative (S l) where
  pure x  = seal (\k -> x)
  f <*> a = seal (\k -> (unseal k f) (unseal k a))
```

```
instance Monad (S l) where
  return = pure
  m >>= f = seal (\k -> unseal k (f (unseal k m)))
```

Note that the programmer does not need access to the TCB in order to implement these instances—as opposed to MAC (cf. [Vas+16]). This phenomenon, we believe, is a sign of the simplicity and generality of our implementation.

C.5.2. Implementation of λ_{SC}^{PRINT}

Figure C.10 shows the implementation of λ_{SC}^{PRINT} which builds on the implementation of λ_{SC} (Figure C.9). The datatype `Eff` wraps `IO` computations and is indexed by a type-level set [OP14] `ls` of channels where the computation can write to. For simplicity, we will omit the map label (Figure C.3) from channels to labels and identify channels with labels so that the index `ls` has kind `[Label]`. This makes the implementation simpler.

In lines 15–18 we implement the return and bind of the graded monad. `returnEff` does not produce effects, thus, its type is indexed by the empty set `[]` (cf. Figure C.3). `bindEff` type is indexed by the union of the sets of labels of the computations (cf. Figure C.3). The implementation of subeffecting is the identity function (lines 20–21). Printing effects can be performed by the combinator `printEff` (lines 26–30). The argument of type `SLabel l` is a term level representation of a the type-level label of printing channel.

Lines 34–36 show the implementation of the novel primitive `distr`. The typeclass constraint `FlowsToSet l ls` (see the definitions in lines 32) ensures that $\texttt{l} \sqsubseteq \texttt{l'}$ for every label `l'` in `ls`. Its implementation is standard. The implementation uses the value `Key`, which pertains to the TCB, to unseal the `Eff` action, i.e. `unseal Key m`; and then it runs its effects, i.e. `res <- runEff (unseal Key m)`; finally it seals the result at the same label, i.e. `return (seal (\k -> res))`.

C.5.3. Implementing Existing Libraries for IFC

We conclude this section by showing that we can reimplement some of the existing libraries in Haskell for IFC. We show implementations of SecLib [RCH08], simplified dependency core calculus (SDCC) [Alg18] (an alternative presentation of DCC) and a variation of MAC [Rus15][6] using SCLib interface. In some sense the implementations help to explain the mentioned libraries. Further, this shows that the programmer can choose to write programs against SCLib's

[6] In our variation the type `Labeled` is a monad. This is "unsafe" in MAC (cf. [Vas+18, Section 9.1]).

```
module SCLib
  (..., Eff (), FlowsToSet (), pureEff, appEff, returnEff
  , bindEff, distr, subeff, printEff)
where

newtype Eff (ls :: [Label]) a = Eff { runEff :: IO a }
-- Functor
instance Functor (Eff ls) where
  fmap f (Eff io) = Eff (fmap f io)
-- Applicative
pureEff :: a -> Eff [] a
pureEff = returnEff

appEff :: Eff ls1 (a -> b) -> Eff ls2 a -> Eff (Union ls1 ls2) b
appEff (Eff ioff) (Eff ioa) = Eff (ioff <*> ioa)
-- Graded monad
returnEff :: a -> Eff [] a
returnEff a = Eff (return a)

bindEff :: Eff ls1 a -> (a -> Eff ls2 b) -> Eff (Union ls1 ls2) b
bindEff (Eff m) f = Eff (m >>= runEff . f)
-- Subeffecting
subeff :: Subset ls1 ls2 => Eff ls1 a -> Eff ls2 a
subeff (Eff m) = Eff m
-- Print
data SLabel :: Label -> * where
  SH :: SLabel H
  SL :: SLabel L

printEff :: (Show a) => SLabel l -> a -> Eff [l] ()
printEff l x = Eff (print (header l) >> print x)
  where header :: SLabel l -> String
        header SH = "Channel H:"
        header SL = "Channel L:"
-- Distr
type family FlowsToSet (l :: Label) (ls :: [Label]) :: Constraint
...

distr :: FlowsToSet l ls => S l (Eff ls a) -> Eff ls (S l a)
distr m = Eff (do res <- (runEff (unseal Key m))
                  return (seal (\k -> res)))
```

Figure C.10.: Implementation of $\lambda_{\mathrm{SC}}^{\mathrm{PRINT}}$ (omitting the unchanged code of Figure C.9)

"low-level" interface; or a more "high-level" interface, e.g. MAC; or a combination of both. We declare future work to compare the performance among implementations.

For each library we briefly explain its interface and show its implementation in terms of SCLib.

SecLib and SDCC SecLib is one of the pioneers of static IFC libraries in the context of Haskell. Its main feature is a family of security monads `Sec` indexed by labels from the security lattice, each equipped with `>>=` (bind) and `return`. SecLib's special ingredient is a combinator `up` that allows coercions from lower to higher labels in the security monad.

SDCC is an alternative presentation of DCC which favours a simple set of combinators instead of DCC's nonstandard bind and *protected at* relation. Similar to DCC, SDCC sports a family of monads indexed by security labels. These support `fmap`, `return` and `>>=` (bind). Further, SDCC implements two combinators `up` and `com`, that allow to relabel in the style of SecLib and commute terms of monadic type with different labels. SDCC's interface is strictly a superset of that of SecLib, thus we directly show the implementation of the former.

```
type T l a = S l a

instance Functor (T l) where
  ...
instance Monad (T l) where
  ...

up :: FlowsTo l l' => T l a -> T l' a
up lv = seal (\k -> unseal k lv)

com :: T l (T l' a) -> T l' (T l a)
com lv = seal (\k' -> seal (\k -> unseal k' (unseal k lv)))
```

MAC MAC which we discussed in the introduction, is one of the state-of-the-art libraries for effectful IFC in Haskell. At its core, MAC defines two types: `Labeled l a` for pure sensitive values, and `MAC l a` for secure computations. `MAC l a` is a monad for each label `l` where the label: 1. protects the data in context; and 2. restricts the permitted effects. (cf. [Vas+18]) MAC's functionality stems from the interaction between `Labeled` and `MAC` through the primitives `label` and `unlabel`. In order to label a value one needs to do so within the `MAC l` monad: the `Labeled l a` type does not export a combinator

`a -> Labeled l a`. Our implementation, however, permits to do so. Below we show MAC's implementation in terms of SCLib.

```
type Labeled l a = S l a

type MAC l a = forall ls. FlowsToSet l ls => Eff ls (S l a)

label :: FlowsTo l l' => a -> MAC l (Labeled l' a)
label a = returnEff (seal (\k -> a))

unlabel :: FlowsTo l l' => Labeled l a -> MAC l' a
unlabel lv = returnEff lv

join :: FlowsTo l l' => MAC l' a -> MAC l (Labeled l' a)
join m = bindEff m (\x -> seal (\k -> x))
```

C.6. Related Work

IFC for Effect-Free and Effectful Languages Algehed and Russo [AR17] add effects to their embedding of DCC in Haskell but argue that their approach only works for those effects that can be implemented within Haskell. Hirsch and Cecchetti [HC21] develop a formal framework based on productors and type-and-effect systems to characterize secure programs in impure languages with IFC. They give semantics to traditional security-type systems based on controlling implicit flows using program counter (PC) labels. In contrast, our approach considers from starters a pure language, where the type of effectful computation is separated from that of effect-free programs. Crary, Kliger, and Pfenning [CKP05] present a graded monad for IFC that tracks both writes and reads on the store, while ours tracks only writes. It will be interesting to understand if this two approaches are equivalent. Devriese and Piessens [DP11] add IFC mechanisms on top of existing monads for effects but don't consider the effect-free–effectful interaction.

Modalities for IFC The languages and IFC enforcement mechanisms that we present are based on the sealing calculus (SC) of Shikuma and Igarashi [SI08]. Differently from them, we think of SC terms as evidence that STLC programs satisfy noninterference. The work by Miyamoto and Igarashi [MI04] gives an informal connection between a classical type system for IFC and a certain modal logic. Their type system is very different from our enforcement mechanism in that a typing judgement has two separate variable contexts. Recently, the work by Abel and Bernardy [AB20] presents a unified treatment of modalities in

typed λ-calculi. The authors present a effect-free lambda calculus parameterized by family of modalities with certain mathematical structure, and show that many programming language analyses, including IFC, are instantiations of their framework. In contrast to our work, it is not very clear how one would implement theirs system in Haskell, since it would require a fine-grained control over the variables in the context. Kavvos [Kav19] studies modalities for IFC in the classified sets model, which they use to prove noninterference properties for a range of calculi that includes SC.

Coeffectful Type Systems for IFC A recent line of work suggests using coeffect type systems to enforce IFC. Petricek, Orchard, and Mycroft [POM14] develop a calculus to capture different granularity demands on contexts, i.e. flat whole-context coeffects (like implicit parameters [Lew+00]) or structural per-variable ones (like usage or data access patterns). The work by Gaboardi, Katsumata, Orchard, Breuvart, and Uustalu [Gab+16] expands on that and uses graded monads and comonads to combine effects and coeffects. The authors describe *distributivity laws* that are similar to our primitive distr addresses. The article suggests IFC as an application where the coeffect system captures the IFC constraints and the effect system gives semantics to effects. The distributive laws explains how both are combined. However, their work does not state neither proves a security property for their calculus. Different from it, our work does not use comonads as the underlying structure for IFC, and further considers printing and global store effects. Granule is a recent programming language [OLI19] based on *graded modal types* that impose usage constraints on the variables.

Logical Relations for Noninterference Both Heintze and Riecke [HR98] and Zdancewic [Zda02] use logical relation arguments to prove noninterference for a simply-typed security lambda calculus. Tse and Zdancewic [TZ04] use logical relations to prove soundness of a translation from DCC [Aba+99] to system F and obtain noninterference from parametricity. Unfortunately, their translation is unsound (cf. [SI08]). Bowman and Ahmed [BA15] fix this by using "open" logical relations show their translation from DCC to system F_ω is sound. Different from the cited work so far, Rajani and Garg [RG20] use logical relations to prove noninterference for a language with references. Gregersen, Bay, Timany, and Birkedal [Gre+21] extend the use of logical relation to prove noninterference for languages with impredicative polymorphism. Different from Rajani and Garg [RG20] and Gregersen, Bay, Timany, and Birkedal [Gre+21], we consider first-order references for simplicity. Otherwise, we should have had

to utilize a step-indexed Kripke-style logical-relations model, which would have introduced technical complications that are orthogonal to the main contribution of our work.

C.7. Conclusions

In this paper, we have demonstrated that to enforce IFC in pure languages with a single primitive distr suffices to securely control what information flows from sensitive data to effects. To support our claim, we have presented IFC enforcement mechanisms for several kinds of effects and proved that they satisfy noninterference. Our development rests on the insight that effect-free IFC for pure languages can already express that a computation will not leak sensitive data through its effects when executed. Then, a single primitive, distr, to execute these is enough to extend IFC to effects and retain the security guarantees. We hope that this work brings a new perspective to IFC research for pure languages with effects.

Bibliography

[AB20] Andreas Abel and Jean-Philippe Bernardy. "A unified view of modalities in type systems". In: *Proc. ACM Program. Lang.* 4.ICFP (2020), 90:1–90:28. DOI: 10.1145/3408972. URL: https://doi.org/10.1145/3408972 (cit. on p. 134).

[Aba+99] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. "A Core Calculus of Dependency". In: *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*. Ed. by Andrew W. Appel and Alex Aiken. ACM, 1999, pp. 147–160. DOI: 10.1145/292540.292555. URL: https://doi.org/10.1145/292540.292555 (cit. on pp. 103, 135).

[Abe+05] Andreas Abel, Guillaume Allais, Jesper Cockx, Nils Anders Danielsson, Philipp Hausmann, Fredrik Nordvall Forsberg, Ulf Norell, Víctor López Juan, Andrés Sicard-Ramírez, and Andrea Vezzosi. *Agda 2*. 2005–. URL: https://wiki.portal.chalmers.se/agda/pmwiki.php (cit. on p. 107).

[Ahm06] Amal J. Ahmed. "Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types". In: *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*. Ed. by Peter Sestoft. Vol. 3924. Lecture Notes in Computer Science. Springer, 2006, pp. 69–83. DOI: 10.1007/11693024_6. URL: https://doi.org/10.1007/11693024%5C_6 (cit. on pp. 107, 122).

[Alg18] Maximilian Algehed. "A Perspective on the Dependency Core Calculus". In: *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. Ed. by Mário S. Alvim and Stéphanie Delaune. ACM, 2018, pp. 24–28. DOI: 10.1145/3264820.3264823. URL: https://doi.org/10.1145/3264820.3264823 (cit. on pp. 108, 131).

[AR17] Maximilian Algehed and Alejandro Russo. "Encoding DCC in Haskell". In: *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2017, Dallas, TX, USA, October 30, 2017*. ACM, 2017, pp. 77–89. DOI: 10.1145/3139337.3139338. URL: https://doi.org/10.1145/3139337.3139338 (cit. on pp. 130, 134).

[BA15] William J. Bowman and Amal Ahmed. "Noninterference for free". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. Ed. by Kathleen Fisher and John H. Reppy. ACM, 2015, pp. 101–113. DOI: 10.1145/2784731.2784733. URL: https://doi.org/10.1145/2784731.2784733 (cit. on p. 135).

[Ber+17] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. "Linear Haskell: practical linearity in a higher-order polymorphic language". In: *CoRR* abs/1710.09756 (2017). arXiv: 1710.09756. URL: http://arxiv.org/abs/1710.09756 (cit. on p. 129).

[BVR15] 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, ICFP 2015, Vancouver,*

BC, Canada, September 1-3, 2015. Ed. by Kathleen Fisher and John H. Reppy. ACM, 2015, pp. 289–301. DOI: 10.1145/2784731.2784758. URL: https://doi.org/10.1145/2784731.2784758 (cit. on pp. 103, 120, 130).

[CKP05] Karl Crary, Aleksey Kliger, and Frank Pfenning. "A monadic analysis of information flow security with mutable state". In: *J. Funct. Program.* 15.2 (2005), pp. 249–291. DOI: 10.1017/S0956796804005441. URL: https://doi.org/10.1017/S0956796804005441 (cit. on p. 134).

[DH83] Jack B. Dennis and Earl C. Van Horn. "Programming Semantics for Multiprogrammed Computations (Reprint)". In: *Commun. ACM* 26.1 (1983), p. 29. DOI: 10.1145/357980.357993. URL: https://doi.org/10.1145/357980.357993 (cit. on p. 129).

[DP11] Dominique Devriese and Frank Piessens. "Information flow enforcement in monadic libraries". In: *Proceedings of TLDI 2011: 2011 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Austin, TX, USA, January 25, 2011*. Ed. by Stephanie Weirich and Derek Dreyer. ACM, 2011, pp. 59–72. DOI: 10.1145/1929553.1929564. URL: https://doi.org/10.1145/1929553.1929564 (cit. on p. 134).

[Gab+16] Marco Gaboardi, Shin-ya Katsumata, Dominic A. Orchard, Flavien Breuvart, and Tarmo Uustalu. "Combining effects and coeffects via grading". In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*. Ed. by Jacques Garrigue, Gabriele Keller, and Eijiro Sumii. ACM, 2016, pp. 476–489. DOI: 10.1145/2951913.2951939. URL: https://doi.org/10.1145/2951913.2951939 (cit. on p. 135).

[GM82] Joseph A. Goguen and José Meseguer. "Security Policies and Security Models". In: *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*. IEEE Computer Society, 1982, pp. 11–20. DOI: 10.1109/SP.1982.10014. URL: https://doi.org/10.1109/SP.1982.10014 (cit. on p. 103).

[Gre+21] Simon Oddershede Gregersen, Johan Bay, Amin Timany, and Lars Birkedal. "Mechanized logical relations for termination-insensitive noninterference". In: *Proc. ACM Program. Lang.* 5.POPL (2021), pp. 1–29. DOI: 10.1145/3434291. URL: https://doi.org/10.1145/3434291 (cit. on pp. 122, 135).

[HC21] Andrew K. Hirsch and Ethan Cecchetti. "Giving semantics to program-counter labels via secure effects". In: *Proc. ACM Program. Lang.* 5.POPL (2021), pp. 1–29. DOI: 10.1145/3434316. URL: https://doi.org/10.1145/3434316 (cit. on p. 134).

[HR98] Nevin Heintze and Jon G. Riecke. "The SLam Calculus: Programming with Secrecy and Integrity". In: *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*. Ed. by David B. MacQueen and Luca Cardelli. ACM, 1998, pp. 365–377. DOI: 10.1145/268946.268976. URL: https://doi.org/10.1145/268946.268976 (cit. on p. 135).

[HS12] Daniel Hedin and Andrei Sabelfeld. "A Perspective on Information-Flow Control". In: *Software Safety and Security - Tools for Analysis and Verification*. Ed. by Tobias Nipkow, Orna Grumberg, and Benedikt Hauptmann. Vol. 33. NATO Science for Peace and Security Series - D: Information and Communication Security. IOS Press, 2012, pp. 319–347. DOI: 10.3233/978-1-61499-028-4-319. URL: https://doi.org/10.3233/978-1-61499-028-4-319 (cit. on p. 103).

[JW93] Simon L. Peyton Jones and Philip Wadler. "Imperative Functional Programming". In: *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*. Ed. by Mary S. Van Deusen and Bernard Lang. ACM Press, 1993, pp. 71–84. DOI: 10.1145/158511.158524. URL: https://doi.org/10.1145/158511.158524 (cit. on p. 111).

[Kat14] Shin-ya Katsumata. "Parametric effect monads and semantics of effect systems". In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. Ed. by Suresh Jagannathan and Peter Sewell. ACM, 2014, pp. 633–646. DOI: 10.1145/2535838.2535846. URL: https://doi.org/10.1145/2535838.2535846 (cit. on pp. 107, 111, 112).

[Kav19] G. A. Kavvos. "Modalities, cohesion, and information flow". In: *Proc. ACM Program. Lang.* 3.POPL (2019), 20:1–20:29. DOI: 10.1145/3290333. URL: https://doi.org/10.1145/3290333 (cit. on p. 135).

[Lew+00] Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark Shields. "Implicit Parameters: Dynamic Scoping with Static Types". In: *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, January 19-21, 2000*. Ed. by Mark N. Wegman and Thomas W. Reps. ACM, 2000, pp. 108–118. DOI: 10.1145/325694.325708. URL: https://doi.org/10.1145/325694.325708 (cit. on p. 135).

[LG88] John M. Lucassen and David K. Gifford. "Polymorphic Effect Systems". In: *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*. Ed. by Jeanne Ferrante and Peter Mager. ACM Press, 1988, pp. 47–57. DOI: 10.1145/73560.73564. URL: https://doi.org/10.1145/73560.73564 (cit. on p. 122).

[LZ06] Peng Li and Steve Zdancewic. "Encoding Information Flow in Haskell". In: *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006), 5-7 July 2006, Venice, Italy*. IEEE Computer Society, 2006, p. 16. DOI: 10.1109/CSFW.2006.13. URL: https://doi.org/10.1109/CSFW.2006.13 (cit. on p. 103).

[MI04] Kenji Miyamoto and Atsushi Igarashi. "A modal foundation for secure information flow". In: *In Proceedings of IEEE Foundations of Computer Security (FCS)*. 2004, pp. 187–203 (cit. on p. 134).

[Mog91] Eugenio Moggi. "Notions of Computation and Monads". In: *Inf. Comput.* 93.1 (1991), pp. 55–92. DOI: 10.1016/0890-5401(91)90052-4. URL: https://doi.org/10.1016/0890-5401(91)90052-4 (cit. on p. 111).

[Mye+06] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. *Jif: Java information flow*. 2006. URL: https://www.cs.cornell.edu/jif (cit. on p. 103).

[OLI19] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. "Quantitative program reasoning with graded modal types". In: *Proc. ACM Program. Lang.* 3.ICFP (2019), 110:1–110:30. DOI: 10.1145/3341714. URL: https://doi.org/10.1145/3341714 (cit. on p. 135).

[OP14] Dominic A. Orchard and Tomas Petricek. "Embedding effect systems in Haskell". In: *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*. Ed. by Wouter Swierstra. ACM, 2014, pp. 13–24. DOI: 10.1145/2633357.2633368. URL: https://doi.org/10.1145/2633357.2633368 (cit. on p. 131).

[Pol+20] Nadia Polikarpova, Deian Stefan, Jean Yang, Shachar Itzhaky, Travis Hance, and Armando Solar-Lezama. "Liquid information flow control". In: *Proc. ACM Program. Lang.* 4.ICFP (2020), 105:1–105:30. DOI: 10.1145/3408987. URL: https://doi.org/10.1145/3408987 (cit. on p. 103).

[POM14] Tomas Petricek, Dominic A. Orchard, and Alan Mycroft. "Coeffects: a calculus of context-dependent computation". In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*. Ed. by Johan Jeuring and Manuel M. T. Chakravarty. ACM, 2014, pp. 123–135. DOI: 10.1145/2628136.2628160. URL: https://doi.org/10.1145/2628136.2628160 (cit. on p. 135).

[PVH19] James Parker, Niki Vazou, and Michael Hicks. "LWeb: information flow security for multi-tier web applications". In: *Proc. ACM Program. Lang.* 3.POPL (2019), 75:1–75:30. DOI: 10.1145/3290388. URL: https://doi.org/10.1145/3290388 (cit. on p. 103).

[RCH08] Alejandro Russo, Koen Claessen, and John Hughes. "A library for light-weight information-flow security in haskell". In: *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada, 25 September 2008*. Ed. by Andy Gill. ACM, 2008, pp. 13–24. DOI: 10.1145/1411286.1411289. URL: https://doi.org/10.1145/1411286.1411289 (cit. on pp. 103, 108, 130, 131).

[RG20] Vineet Rajani and Deepak Garg. "On the expressiveness and semantics of information flow types". In: *J. Comput. Secur.* 28.1 (2020), pp. 129–156. DOI: 10.3233/JCS-191382. URL: https://doi.org/10.3233/JCS-191382 (cit. on pp. 103, 120, 122, 135).

[Rus15] Alejandro Russo. "Functional pearl: two can keep a secret, if one of them uses Haskell". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. Ed. by Kathleen Fisher and John H. Reppy. ACM, 2015, pp. 280–288. DOI: 10.1145/

2784731.2784756. URL: https://doi.org/10.1145/2784731.2784756 (cit. on pp. 103, 105, 120, 130, 131).

[SI08] Naokata Shikuma and Atsushi Igarashi. "Proving Noninterference by a Fully Complete Translation to the Simply Typed Lambda-Calculus". In: *Log. Methods Comput. Sci.* 4.3 (2008). DOI: 10.2168/LMCS-4(3:10)2008. URL: https://doi.org/10.2168/LMCS-4(3:10)2008 (cit. on pp. 107, 108, 134, 135).

[Sim03] Vincent Simonet. *Flow Caml*. 2003. URL: http://cristal.inria.fr/~simonet/soft/flowcaml/ (cit. on p. 103).

[SM03] Andrei Sabelfeld and Andrew C. Myers. "Language-based information-flow security". In: *IEEE J. Sel. Areas Commun.* 21.1 (2003), pp. 5–19. DOI: 10.1109/JSAC.2002.806121. URL: https://doi.org/10.1109/JSAC.2002.806121 (cit. on p. 103).

[Ste+11] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. "Flexible dynamic information flow control in Haskell". In: *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011*. Ed. by Koen Claessen. ACM, 2011, pp. 95–106. DOI: 10.1145/2034675.2034688. URL: https://doi.org/10.1145/2034675.2034688 (cit. on p. 103).

[TT97] Mads Tofte and Jean-Pierre Talpin. "Region-based Memory Management". In: *Inf. Comput.* 132.2 (1997), pp. 109–176. DOI: 10.1006/inco.1996.2613. URL: https://doi.org/10.1006/inco.1996.2613 (cit. on p. 122).

[TZ04] Stephen Tse and Steve Zdancewic. "Translating dependency into parametricity". In: *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, Snow Bird, UT, USA, September 19-21, 2004*. Ed. by Chris Okasaki and Kathleen Fisher. ACM, 2004, pp. 115–125. DOI: 10.1145/1016850.1016868. URL: https://doi.org/10.1145/1016850.1016868 (cit. on pp. 107, 109, 135).

[Vas+16] Marco Vassena, Pablo Buiras, Lucas Waye, and Alejandro Russo. "Flexible Manipulation of Labeled Values for Information-Flow Control Libraries". In: *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I*. Ed. by Ioannis G. Askoxylakis, Sotiris Ioannidis, Sokratis K. Katsikas, and Catherine A. Meadows. Vol. 9878. Lecture Notes in Computer Science. Springer, 2016, pp. 538–557. DOI: 10.1007/978-3-319-

45744-4_27. URL: https://doi.org/10.1007/978-3-319-45744-4%5C_27 (cit. on pp. 104, 131).

[Vas+18] Marco Vassena, Alejandro Russo, Pablo Buiras, and Lucas Waye. "MAC A verified static information-flow control library". In: *Journal of Logical and Algebraic Methods in Programming* 95 (2018), pp. 148–180. ISSN: 2352-2208. DOI: https://doi.org/10.1016/j.jlamp.2017.12.003. URL: https://www.sciencedirect.com/science/article/pii/S235222081730069X (cit. on pp. 131, 133).

[Vas+19] Marco Vassena, Alejandro Russo, Deepak Garg, Vineet Rajani, and Deian Stefan. "From fine- to coarse-grained dynamic information flow control and back". In: *Proc. ACM Program. Lang.* 3.POPL (2019), 76:1–76:31. DOI: 10.1145/3290389. URL: https://doi.org/10.1145/3290389 (cit. on p. 103).

[VIS96] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. "A Sound Type System for Secure Flow Analysis". In: *J. Comput. Secur.* 4.2/3 (1996), pp. 167–188. DOI: 10.3233/JCS-1996-42-304. URL: https://doi.org/10.3233/JCS-1996-42-304 (cit. on p. 115).

[WT03] Philip Wadler and Peter Thiemann. "The marriage of effects and monads". In: *ACM Trans. Comput. Log.* 4.1 (2003), pp. 1–32. DOI: 10.1145/601775.601776. URL: https://doi.org/10.1145/601775.601776 (cit. on p. 114).

[Zda02] Stephan Arthur Zdancewic. *Programming languages for information security*. Cornell University, 2002 (cit. on p. 135).

Appendices

I. The Language λ_{REC}

$$\begin{array}{rrcl} \text{Types} & a, b & ::= & \mathsf{Unit} \mid \mathsf{Bool} \mid a \Rightarrow b \\ \text{Typing contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : a \end{array}$$

$\boxed{\Gamma \vdash t : a}$

$$\textsc{Var}\ \frac{(x : a) \in \Gamma}{\Gamma \vdash x : a} \qquad \textsc{Fun}\ \frac{\Gamma, f : a \Rightarrow b, x : a \vdash t : b}{\Gamma \vdash \mu\, f.\, x.\, t : a \Rightarrow b} \qquad \textsc{App}\ \frac{\Gamma \vdash t : a \Rightarrow b \qquad \Gamma \vdash u : a}{\Gamma \vdash \mathsf{app}\, t\, u : b}$$

$$\textsc{Unit}\ \frac{}{\Gamma \vdash \mathsf{unit} : \mathsf{Unit}} \qquad \textsc{True}\ \frac{}{\Gamma \vdash \mathsf{true} : \mathsf{Bool}} \qquad \textsc{False}\ \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{Bool}}$$

$$\textsc{If}\ \frac{\Gamma \vdash t : \mathsf{Bool} \qquad \Gamma \vdash u_1 : a \qquad \Gamma \vdash u_2 : a}{\Gamma \vdash \mathsf{ifte}\, t\, u_1\, u_2 : a}$$

$\boxed{t \to u \text{ with } \cdot \vdash t : a \text{ and } \cdot \vdash u : a}$

$$\textsc{App}\ \frac{t \to t'}{\mathsf{app}\, t\, u \to \mathsf{app}\, t'\, u} \qquad \textsc{Beta}\ \frac{}{\mathsf{app}\, (\mu\, f.\, x.\, t)\, u \to t[f/\mu\, f.\, x.\, t, u/x]}$$

$$\textsc{If}\ \frac{t \to t'}{\mathsf{ifte}\, t\, u_1\, u_2 \to \mathsf{ifte}\, t'\, u_1\, u_2} \qquad \textsc{If-True}\ \frac{}{\mathsf{ifte}\, \mathsf{true}\, u_1\, u_2 \to u_1} \qquad \textsc{If-False}\ \frac{}{\mathsf{ifte}\, \mathsf{false}\, u_1\, u_2 \to u_2}$$