THESIS FOR THE DEGREE OF LICENTIATE OF PHILOSOPHY

MAC
A Verified Information-Flow Control Library

MARCO VASSENA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY

Göteborg, Sweden 2017

MAC, A Verified Information-Flow Control Library
MARCO VASSENA

Technical Report 166L
ISSN 1652-876X
Department of Computer Science and Engineering
Research group: Information Security

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 1000

Printed at Chalmers
Göteborg, Sweden 2017

ABSTRACT

Information Flow Control (IFC) is a language-based security mechanism that tracks where data flows within a program and prevents leakage of sensitive data. IFC has been embedded in pure functional languages such as Haskell, in the form of a *library*, thus reducing the implementation and maintenance effort and fostering a *secure-by-construction* programming-model. **MAC** is a state-of-the-art IFC Haskell library that detects leaks *statically* and that supports many advanced programming features, such as exceptions, mutable references and concurrency. While **MAC** is an elegant functional pearl and is implemented concisely in less than 200 lines of code, it does not provide any formal security guarantee.

This thesis presents the first full-fledged verified formal model of **MAC**, which guarantees that any program written against the library's API satisfies non-interference by construction. In particular, the contributions of this work improve **MAC** in three areas: formal verification techniques, expressivity and protection against covert channels. Firstly, the thesis enriches *term erasure* with *two-steps erasure*, a novel flexible technique, which has been used to reason systematically about the security implications of advanced programming features and that greatly simplifies the non-interference proof. Secondly, this work gives a *functor* algebraic structure to *labeled values*, an abstract data type which protects values with explicit labels, thus enabling flexible manipulation of labeled data through classic functional programming patterns. Thirdly, the thesis closes the *sharing-based* internal-timing covert channel, which exploits the sharing feature of *lazy evaluation* to leak data, by affecting the timing behavior of threads racing to gain access to some shared resource. We design an unsharing primitive that disables sharing by lazily duplicating *thunks* and we apply it to restrict sharing, when needed for security reasons.

All the results presented in this thesis have been corroborated with extensive mechanized proofs, developed in the Agda proof assistant.

ACKNOWLEDGMENTS

To Alejandro Russo, for being the perfect supervisor and a dear friend. This work would have not been possible without you on my side, encouraging, challenging and pushing me to the next level. It has been so inspiring to work with you and I am glad to have this opportunity to fully appreciate your collaboration and support. I feel immensely lucky to have you as a mentor and I look forward to all the exciting work to come in the next years.

While working at Chalmers, I have met extraordinary and inspiring people, whom I am lucky I can call friends. Thank you Daniel, Danielito, Raúl, Evgeny, Mauricio, Pablo, Simon, Alexander, Per, Hamid and many others.

To my friends Pier, Ludia, Aura, Yasmine, Tugce, Tim, Carlo and Katja. Thank you so much for all the fun together: the cozy movie nights, the potluck dinners, the everlasting BBQs, the crazy parties and the trips. No matter if it is a cold, dark winter night, or a sunny, summer day you have always been there for me.

I am so glad to have met people that share with me the passion for playing music. A huge thank you to the Cougar Boys, now The Traveling Lingonberries, and whatever name we may choose next. In particular thank you Pier, Carlo, Enzo, Grischa, Evgeny for all the jamming and the gig (hopefully there will be more coming soon) and to Aura, Andreas and Vincenzo for filling in when needed.

A Laura, il mio amore più grande. Grazie per appoggiare sempre le mie scelte, nonostante questo significhi vivere a migliaia di chilometri di distanza e vedersi raramente. Grazie per essermi vicina ogni giorno: presto torneremo insieme per non lasciarci più.

Alla mia cara famiglia che ha sempre creduto in me e che mi manca sempre tantissimo. Ai miei genitori, non avrei mai raggiunto questo traguardo senza il vostro sostegno e incoraggiamento. Un grazie di cuore a mia sorella Chiara, per prenderti sempre cura di tutti. Grazie ai nonni Clara, Alba e Domenico, quando sono con voi è come se il tempo non fosse mai passato e ritorno bambino.

CONTENTS

CHAPTER

ONE

INTRODUCTION

Our digital society relies on software in virtually any aspect. Banks and financial institutions, social security and intelligent transportation systems, business, telecommunication and logistics companies use software to provide their services to clients and keep digital records of their activity in information systems. The advantages and benefits of the digitalization process are many. Firstly, digital storage is cost-effective and durable, which makes it feasible to collect and store large amounts of data. Secondly, digital data can be exactly replicated and copies are indistinguishable from the original. Data replication then improves data reliability, e.g., against data corruption, fault-tolerance, e.g., against data loss, and accessibility, as the information becomes available to users anytime and anywhere via internet. The data itself represents a precious resource for commercial companies, because it reveals trends and habits of customers—something extremely precious in a competitive market.

Nowadays, the role that software plays in handling data is even bigger due to the spread of personal computer and, in the past few years, smartphones. *App stores* distribute software for these devices in the form of *apps*, which provide users with handy functionalities, (e.g., route planning, fitness), social virtual platforms, (e.g., chats, social networks, dating apps), entertainment, (e.g., games, music, videos), etc. Often apps manipulate sensitive data and usually deliver a unique experience to each user, tailored on his or her personal data. For example a route planner requires the user's geographical location, obtained from the GPS of his device, to provide accurate directions, and a social network app suggests friends based on the device contacts list. Clearly, such personal data is sensitive and should be handled with care by apps, however users, who are not security experts, are normally not concerned about the risks of data theft and unaware of the dangers of data breach. As a result, mobile devices are an easy target for malicious software, that collects and leaks sensitive data—remember that data is valuable in the digital society!

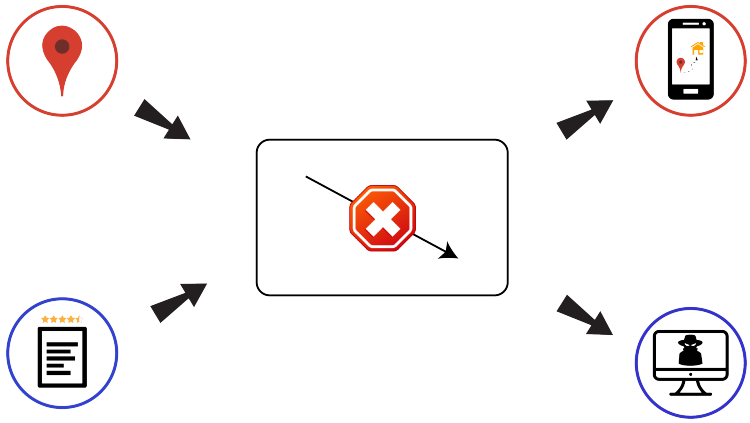

Fig. 1: Route planner app secured with IFC.

Unfortunately, the security mechanisms currently deployed to protect sensitive data in smartphones are inappropriate. In fact a user, who wishes to use an app, must first install it and grant it explicit access to any sensitive information or functionality that it requires. This security mechanism, known as *access control*, is unsatisfactory, because, once access to data and capabilities is granted, users have no control of what the app might do with it. For example, a malicious application that has access to the GPS location and network connectivity of a device can monitor and track the location of users. Since there are legitimate uses of these features, e.g., a route planner app, users have no way to distinguish between a malicious and a honest app, and therefore will likely grant access in good faith [14]. Furthermore, even honest apps may involuntarily leak sensitive information, due to security vulnerabilities or software bugs, therefore there is a need to *systematically* prevent data leakage, either intentional or accidental.

Clearly, there is a tension between the opportunities and the risks connected to digitalization. Surely, we cannot stop the digitalization process, because our society profoundly relies on software already, however the dangers connect to data theft and data breach may hinder or even outweigh the benefits. In this thesis, we investigate Information-Flow Control (IFC) [19], a promising programming language-based security mechanism, alternative to *access control*, which supports a sustainable digitalization process.

1 Information Flow Control

IFC protects the confidentiality of data by tracking *where* data at different security levels flows within a program and raises alarms when sensitive information is leaked. A security lattice [6] specifies the security levels of the system and which flows of information are insecure. For instance, Figure 2 describes a simple security lattice, that contains two levels, i.e., **Secret** and **Public** and where the

only insecure flow is from **Secret** to **Public**. Intuitively, Figure 1 describes how an IFC system secures a routing planner app, depicted as the box in the center. The inputs of a program are annotated with their sensitivity as **Secret** or **Public**, for which we use color red and blue respectively. The routing planner app has two inputs (the circles on the left of the box): the user's location, i.e., secret data, and the review of a restaurant, i.e., public data. The outputs of the app are depicted on the right and are also labeled as secret or public. It is secure to show secret data, e.g., the location, on the device display, which is considered a **Secret** channel, because only the authenticated user can view it. For example, the routing app can safely display on the user's device the route from the current location to home—see the top-right red circle. A route planner app with access to network connectivity can send the current location over the internet, i.e., a **Public** channel, to any server, which we depict as the attacker's computer in the bottom-right blue circle. Clearly, sending secret data over a public channel represents a leak and it is the goal of an IFC system to stop this insecure flow of information. The security policy, that we have just informally described, is known in literature as *non-interference* [8] and guarantees that the secrets inputs of a program shall not interfere with the public outputs.

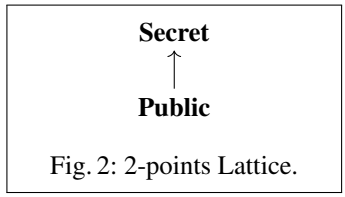

Fig. 2: 2-points Lattice.

IFC has been applied to both imperative and functional programming languages, *statically* [3, 15, 25], i.e., using a type system that rejects possibly leaky programs, *dynamically* [10, 24], i.e., adding a run-time monitor that aborts the execution of a program that is about to leak, and in a *hybrid* fashion [4], i.e., as a combination of both approaches. IFC is able to detect both *explicit* and *implicit* insecure flows of information, however the number and the bandwidth of *covert channels*, i.e., unintended communication channels that leak information by exploiting system features, challenge the applicability of IFC. Examples of covert channels, which may reveal some bits of information about a secret, include the power consumption of a program, which can be detected with power analysis, the wall-clock time that it takes to execute a program, also known as *external timing* covert channel [2, 7, 9, 12, 26] and the order of public outputs in a concurrent system, where the interleaving of racing threads depends on a secret, i.e., the *internal timing* covert channel [21].

Modern programming languages provide many sophisticated features to simplify software development. For example, exception handling primitives are an advanced control-flow structure to deal with exceptional situations, concurrency fosters modular programming, locks provide a synchronization mechanism between threads, etc. The lack of advanced features makes software development impractical, hence defeating the applicability of IFC systems, therefore it is desirable to support them. Unfortunately, the inclusion of such advanced features in IFC systems is challenging, because their elusive behavior and their subtle interaction might secretly enable data leakage. *Term erasure* [13] is a standard

proof technique used to verify IFC functional languages [4, 11, 22, 23], which requires that the same public output should be produced if secrets are erased before or after program execution. Erasure is performed by the *erasure function*, which removes any piece of sensitive data above the attacker's level from a program. Then, the technique requires to establish a *simulation* between the reductions of the original and the erased program, which are formally defined by the operational semantics of the language.

2 MAC

This thesis focuses on **MAC**, a state-of-the-art static IFC library for Haskell, a pure functional programming language [18]. **MAC** brings ideas from Mandatory Access Control into a language-based setting, fostering a programming model, where any well-typed program written against the library's API is *secure by construction*. In particular, the library guarantees data confidentiality by restricting all side-effectful operations to comply with the *no write-down* and *no read-up* security policies [1]. Intuitively, **MAC** prevents the attempted leak in Figure 1, because that operation violates the *no write-down* security policy, since it sends the location, i.e., secret data, to the attacker's server, i.e., a public channel. In Haskell, the type-system strictly separates side-effecful code, i.e., code that can perform I/O, from that that can perform none, which are also referred to as *impure* and *pure* code respectively. The rigid and syntactic distinction between pure and impure code simplifies the design and implementation of security mechanisms so much that **MAC** is implemented as a simple 200 lines of code (LOC) Haskell library. IFC systems for mainstream languages, e.g., Jif [15] and Paragon [3] for Java, JSFlow for JavaScript [10], FlowCaml for Caml [17], require instead to extend the the language, the compiler and the run-time system. **MAC** is fully embedded in Haskell: the security type system and the security lattice is encoded in Haskell's expressive type-system using standard well-established features, such as type classes. Furthermore, **MAC** is very expressive and provides many advanced programming features, such as exceptions, mutable data structures and concurrency. Russo does not give any formal security guarantees about **MAC** in his functional pearl, instead the elegance and the compactness of the code convinces the reader that the library is secure [18].

3 Contributions

Figure 3 outlines the contributions of this thesis, which revolve about improving IFC systems for pure functional languages, such as **MAC**, in three areas: formal verification techniques, expressivity and protection against covert channels. Firstly, we have devised *two-steps erasure*, a novel idea that extends and improves *term erasure*, when it fails for certain problematic language primitives.

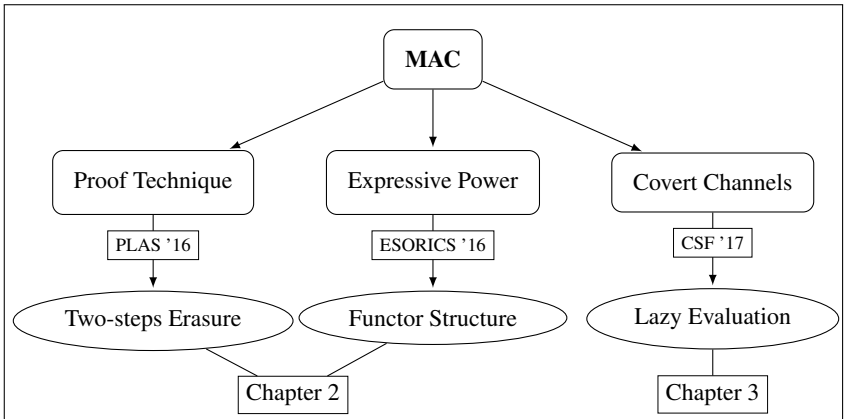

Fig. 3: Overview of the thesis.

Two-steps erasure perform term erasure in two steps, i.e., it replaces the challenging primitives with special, ad-hoc constructs, whose semantics then performs the desired term erasure. We have used this technique systematically, in order to develop the first comprehensive fully-verified formal model of **MAC**. Secondly, we have extended **MAC** with new sophisticated primitives that boost expressivity and that encourage the *functional* coding style of the host language, i.e., Haskell, therefore making the library more practical for software development. Thirdly, we have addressed the internal-timing covert channel [5], which arise from Haskell's *lazy evaluation*. Lazy evaluation combines *non-strictness* and *sharing*, two characteristics that have opposite security implications. Informally, non-strictness guarantees that function arguments are not evaluated until needed inside the function—a feature that, as we describe in the second paper, naturally stops termination leaks in IFC libraries and tools. Sharing ensures that results of evaluated terms are stored for subsequent re-utilization. Crucially, sharing represents a hidden side-effect in disguise, that eludes the security mechanisms of the libraries and that enables the internal-timing channel [5]. All the results of this thesis have been corroborated with mechanized proofs, developed in the Agda proof assistant [16], which we have made available online.

This thesis focuses on **MAC**, because it lacks formal security guarantees and because it is simpler to formalize due to its *static* nature—the presence of security labels in typing judgments simplifies proofs. Nevertheless, we remark that the ideas and the techniques developed in this thesis apply to other state-of-the-art *dynamic* and *hybrid* IFC Haskell libraries, such as **LIO** [23] and **HLIO** [4], and to other IFC systems as well.

4 Overview

In this section, we give a more detailed overview of the thesis, which is based on three peer-reviewed papers, published individually in the proceedings of peer-reviewed international conferences and workshops.

4.1 On Formalizing Information-Flow Control Libraries

The paper presents a full-fledged, mechanically-verified formal model of the **MAC** library as a simply-typed λ-calculus extended with security primitives and advanced features, such as exceptions, mutable references and concurrency. The main contribution of the paper consists of three insights, which empowers *term erasure* with new proof techniques and simplify reasoning about concurrent systems. The paper describes in detail i) *two-steps erasure*, a novel proof technique to reason about security in presence of advanced stateful features; ii) *exception masking*, a novel proof technique that simplifies reasoning about the interaction between exceptions and security primitives; iii) *scheduler parametric proofs*: the security guarantees are valid for a wide range of deterministic schedulers, that we characterize formally with precise scheduler requirements. As a result we prove that **MAC** is secure under a round-robin scheduler, by simply instantiating our main scheduler-parametric theorem. In addition, the insights of the paper and the extensive mechanized proof (4000 LOC), led us to uncover some problems in LIO's proofs and propose changes to repair its non-interference guarantees.

Statement of contributions This paper was coauthored with Alejandro Russo and published in the proceedings of the 11th Workshop on Programming Languages and Analysis for Security (PLAS), 2016. Marco and Alejandro devised the proof techniques, Marco was responsible for the mechanized formalization of the model and significantly contributed to the writing of the whole paper.

4.2 Flexible Manipulation of Labeled Values for Information-Flow Control Libraries

This paper explores the algebraic structure of *labeled data*, i.e., an abstract data type that explicitly labels a piece of data with a label, which is used in IFC libraries to enforce the *no write-down* and *no read-up* security policies. Unfortunately, programmers have to deal with these policies unnecessarily also when performing *pure* computations on labeled data, which are inherently secure, since they cannot perform any I/O. In this paper, we give a *functor* structure to labeled data, which precisely encapsulates this pattern. Furthermore, we study an applicative functor operator, which extends this feature to work on multiple labeled values combined by a multi-parameter function and a relabel primitive which securely upgrades the label of labeled values, as needed when aggregating data with heterogeneous labels. These primitives encourage the functional

programming style, typical of the host language, i.e., Haskell, and provides flexibility when manipulating labeled data with side-effect free computations, therefore fostering the secure-by-construction programming model.

Statement of contributions This paper was coauthored with Pablo Buiras, Lucas Waye, and Alejandro Russo and published in the proceedings of the 21st European Symposium on Research in Computer Security (ESORICS), 2016. Pablo and Lucas conceived the idea of adding a functor structure to labeled values. Marco identified some technical problems with that feature and devised a solution. He was responsible for the mechanized proofs and writing most of the paper.

Chapter 2 merges and revises these two papers, in order to provide a uniform, coherent, comprehensive formal model of **MAC**, to integrate more examples of the features of the library, to fix few technical inaccuracies in the semantics of the calculus and to give a full account of the scheduler-parametric PSNI theorem and simplify its proof. An extended abstract based on these papers was accepted at the 28th Nordic Workshop on Programming Theory (NWPT'16), where it has been then selected as one of the best contributions and invited to a special issue of the Journal of Logical and Algebraic Methods in Programming (JLAMP), where Chapter 2 is currently under submission as a full research article.

4.3 Securing Concurrent Lazy Programs Against Information Leakage

Lazy evaluation is a distinctive feature of Haskell, the programming language used to implement many state-of-the-art IFC libraries and tools, including **MAC**. Unfortunately, *sharing* enables data leakage via the internal timing covert channel. The paper proposes, as a counter measure, an *unsharing* primitive, which *lazily* restricts sharing from secret computations, i.e., sensitive threads, to public computations, therefore disabling any data race between public threads, which implicitly depends on secrets via *sharing*. We formally model sharing with Sestoft's abstract machine [20], extended with a mutable store, which also exhibits sharing (the first of its kind), and adapt the semantics of the calculus to duplicate *thunks*, when needed for security reasons. We show that the calculus satisfies progress-sensitive non-interference and support our results with mechanized proofs. In addition, we remark that such primitive is useful beyond security, also to close *memory leaks* in programs implemented in languages with lazy evaluation.

Statement of contributions This paper was coauthored with Joachim Breitner and Alejandro Russo and published in the proceedings of the 30th IEEE Computer Security Foundations Symposium (CSF), 2017. Joachim conceived the first version of the lazy unsharing primitive and Alejandro suggested to use it to close the sharing-based internal-timing covert channel. Marco adapted the primitive for the operational semantics of Sestoft's abstract machine and for introducing mutable references. He also extended MAC with the new primitive,

he was responsible for the mechanized proofs and writing the technical sections of the paper and the examples as well.

References

1. David E. Bell and L. La Padula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997, Rev. 1, MITRE Corporation, Bedford, MA, 1976.
2. Andrew Bortz and Dan Boneh. Exposing private information by timing web applications. In *Proc. of the 16th World Wide Web*. ACM, 2007.
3. Niklas Broberg, Bart van Delft, and David Sands. Paragon for practical programming with information-flow control. In *APLAS*, volume 8301 of *Lecture Notes in Computer Science*, pages 217–232. Springer, 2013.
4. P. Buiras, D. Vytiniotis, and A. Russo. HLIO: Mixing static and dynamic typing for information-flow control in Haskell. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP '15)*. ACM, 2015.
5. Pablo Buiras and Alejandro Russo. Lazy programs leak secrets. In *Proc. Nordic Conference in Secure IT Systems (NORDSEC)*. Springer-Verlag, 2013.
6. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.
7. Edward W. Felten and Michael A. Schneider. Timing attacks on Web privacy. In *Proc. of the 7th ACM conference on Computer and communications security*, CCS '00. ACM, 2000.
8. J.A. Goguen and J. Meseguer. Security policies and security models. In *Proc of IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1982.
9. Helena Handschuh and Howard M. Heys. A Timing Attack on RC5. In *Proc. of the Selected Areas in Cryptography*. Springer-Verlag, 1999.
10. D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. In *Proc. of the ACM Symposium on Applied Computing (SAC '14)*. ACM, March 2014.
11. Stefan Heule, Deian Stefan, Edward Z. Yang, John C. Mitchell, and Alejandro Russo. IFC inside: Retrofitting languages with dynamic information flow control. In *Conference on Principles of Security and Trust (POST)*. Springer, April 2015.
12. Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proc. of the 16th CRYPTO*. Springer-Verlag, 1996.
13. P. Li and S. Zdancewic. Arrows for secure information flow. *Theoretical Computer Science*, 411(19):1974–1994, 2010.
14. Simon Meurer and Roland Wismüller. Apefs: An infrastructure for permission-based filtering of android apps. In AndreasU. Schmidt, Giovanni Russello, Ioannis Krontiris, and Shiguo Lian, editors, *Security and Privacy in Mobile Information and Communication Systems*, volume 107. Springer Berlin Heidelberg, 2012.
15. A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java Information Flow. `http://www.cs.cornell.edu/jif`, 2001.
16. Ulf Norell. Dependently typed programming in agda. In Andrew Kennedy and Amal Ahmed, editors, *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*, pages 1–2. ACM, 2009.
17. F. Pottier and V. Simonet. Information Flow Inference for ML. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 319–330, January 2002.
18. Alejandro Russo. Functional pearl: Two can keep a secret, if one of them uses Haskell. In *Proc. of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015. ACM, 2015.

19. Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.
20. Peter Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(03), 1997.
21. G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM symposium on Principles of Programming Languages (POPL '98)*, 1998.
22. D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Maziéres. Addressing covert termination and timing channels in concurrent information flow systems. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP '12)*. ACM, 2012.
23. D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Proc. of the ACM SIGPLAN Haskell symposium (HASKELL '11)*, 2011.
24. Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in Haskell. In *Proceedings of the 4th ACM symposium on Haskell*, pages 95–106, New York, NY, USA, 2011. ACM.
25. Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.
26. Wing H. Wong. Timing attacks on RSA: revealing your secrets through the fourth dimension. *Crossroads*, 11, May 2005.

PAPER I AND II

Based on

On Formalizing Information-Flow Control Libraries,

by Marco Vassena and Alejandro Russo,

11th Workshop on Programming Languages and Analysis for Security

and

Flexible Manipulation of Labeled Values for Information-Flow Control Libraries,

by Marco Vassena, Pablo Buiras, Lucas Waye, and Alejandro Russo,

21st European Symposium on Research in Computer Security.

CHAPTER

TWO

MAC, A VERIFIED INFORMATION FLOW CONTROL LIBRARY

Abstract. The programming language Haskell plays a unique, privileged role in information-flow control (IFC) research: *it is able to enforce information security via libraries*. Many state-of-the-art IFC libraries (e.g., **LIO** and **HLIO**) support a variety of advanced features like mutable data structures, exceptions, and concurrency, whose subtle interaction makes verification of security guarantees challenging. In this work, we focus on **MAC**, a statically-enforced IFC library for Haskell. In **MAC**, like other IFC libraries, computations have a well-established algebraic structure for computations (i.e., monads) responsable to manipulate *labeled values*—values coming from an abstract data type which associates a sensitivity label to a piece of information. In this work, we enrich labeled values with a *functor* structure and provide an *applicative functor* operator which encourages a more functional programming style and simplifies code. Furthermore, we present a full-fledged, mechanically-verified model of **MAC**. Specifically, we show progress insensitive non-interference for our sequential calculus and pinpoint sufficient requirements on the scheduler to prove progress-sensitive non-interference for our concurrent calculus. For that, we study the security guarantees of **MAC** using *term erasure*, a proof technique that ensures that the same public output should be produced if secrets are erased before or after program execution. As another contribution, we extend term erasure with *two-steps erasure*, a flexible novel technique that, which greatly simplifies the non-interference proof and helps to prove many advanced features of **MAC**.

1 Introduction

Nowadays, many applications (apps) manipulate users' private data. Such apps *could have been written by anyone* and users who wish to benefit from their

functionality are forced to grant them access to their data—something that most users will do without a second thought [33]. Once apps collect users' information, there are no guarantees about how they handle it, thus leaving room for data theft and data breach by malicious apps. The key to guaranteeing security without sacrificing functionality is not granting or denying access to sensitive data, but rather ensuring that *information only flows into appropriate places*.

Information-flow control (IFC) [47] is a promising programming language-based approach to enforcing security. IFC scrutinizes how data of different sensitivity levels (e.g., public or private) flows within a program, and raises alarms when there is an unsafe flow of information. Most IFC tools require the design of new languages, compilers, interpreters, or modifications to the runtime, e.g., [8, 36, 39, 42]. In this scenario, the functional programming language Haskell plays a unique privileged role: *it is able to enforce security via libraries* [29] by using an embedded domain-specific language. Many of the state-of-the-art Haskell security libraries, namely **LIO** [52], **HLIO** [10], and **MAC** [45], bring ideas from Mandatory Access Control [5] into a language-based setting. Every computation in such libraries has a *current label* which is used to (i) approximate the sensitivity level of all the data in scope and (ii) restrict subsequent side-effects which might compromise security. From now on, we simply use the term libraries when referring to **LIO**, **HLIO**, and **MAC**. IFC uses labels to model the sensitivity of data, which are then organized in a security lattice [12] specifying the allowed flows of information, i.e., $\ell_1 \sqsubseteq \ell_2$ means that data with label ℓ_1 can flow into entities labeled with ℓ_2. Although these libraries are parameterized on the security lattice, for simplicity we focus on the classic two-point lattice with labels H and L to respectively denote secret (high) and public (low) data, and where $H \not\sqsubseteq L$ is the only disallowed flow. Figure 1 shows a graphical representation of a public computation in these libraries, i.e., a computation with current label L. The computation can read or write data in scope, which is considered public (e.g., average temperature of 17°C in the Swedish summer), and it can write to public (L-) or secret (H-) sinks. By contrast, a secret computation, i.e., a computation with current label H, can also read and write data in its scope, which is considered sensitive, but in order to prevent information leaks it can *only* write to sensitive/secret sinks. Structuring computations in this manner ensures that sensitive data does not flow into public entities, a policy known as noninterference [16]. While secure, programming in this model can be overly restrictive for users who want to manipulate differently-labeled values.

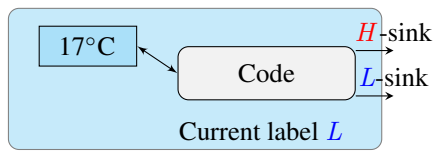

Fig. 1: Public computation

To address this shortcoming, libraries introduce the notion of a *labeled value* as an abstract data type which protects values with explicit labels, in addition to the current label. Figure 2 shows a public computation with access to both

public and sensitive pieces of information, such as a password (pwd). Public computations can freely manipulate sensitive labeled values provided that they are treated as black boxes, i.e., they can be stored, retrieved, and passed around as long as its content is not inspected. Libraries **LIO** and **HLIO** even allow public computations to inspect the contents of sensitive labeled values, raising the current label to H to keep track of the fact that a secret is in scope—this variant is known as a *floating-label* system.

Reading sensitive data usually amounts to "tainting" the entire context or ensuring the context is as sensitive as the data being observed. As a result, the system is susceptible to an issue known as *label creep*: reading too many secrets may cause the current label to be so high in the lattice that the computation can no longer perform any useful side effects. To address this problem, libraries provide a primitive which enables public computations to spawn sub-computations that access sensitive labeled values without tainting the parent. In a sequential setting, such sub-computations are implemented by special function calls. In the presence of concurrency, however, they must be executed in a different thread to avoid compromising security through *internal timing* and *termination covert channels* [51].

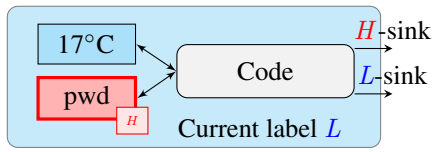

Fig. 2: Labeled values

Practical programs need to manipulate sensitive labeled values by transforming them. It is quite common for these operations to be naturally free of I/O or other side effects, e.g., arithmetical or algebraic operations, especially in applications like image processing, cryptography, or data aggregation for statistical purposes. Writing such functions, known as *pure* functions, is the bread and butter of functional programming style, and is known to improve programmer productivity, encourage code reuse, and reduce the likelihood of bugs [24]. Nevertheless, the programming model involving sub-computations that manipulate secrets forces an imperative style, whereby computations must be structured into separate compartments that must communicate explicitly. While side-effecting instructions have an underlying structure (called monad [34]), research literature has neglected studying structures for labeled values and their consequences for the programming model. To empower programmers with the simpler, functional style, we propose additional operations that allow pure functions to securely manipulate labeled values, specifically by means of a structure similar to *applicative functors* [32]. In particular, this structure is useful in concurrent settings where it is no longer necessary to spawn threads to manipulate sensitive data, thus making the code less imperative (i.e., side-effect free).

Additionally, practical programs often aggregate information from heterogeneous sources. For that, programs needs to upgrade labeled values to an upper bound of the labels being involved before data can be combined. In previous incarnations of the libraries, such relabelings require to spawn threads just for

that purpose. As before, the reason for that is libraries decoupling every computation which manipulate sensitive data—even those for simply relabeling—so that the internal timing and termination covert channels imposed no threats. In this light, we introduce a primitive to securely relabel labeled values, which can be applied irrespective of the computation's current label and does not require spawning threads.

We provide a mechanized security proof for the security library **MAC**[1] and claim our results also apply to **LIO** and **HLIO**. **MAC** has fewer lines of code and leverages types to enforce confidentiality, thus making it ideal to model its semantics in a dependently-typed language like Agda. The contributions of this paper are:

1. We develop the first exhaustive full-fledged formalization of **MAC**, a state-of-the-art library for Information-Flow Control, in a call-by-need λ-calculus and prove progress-insensitive (PINI) for the sequential calculus.
2. We enrich the calculus with scheduler-parametric concurrency and prove progress-sensitive noninterference (PSNI) [1] for a wide-range of deterministic schedulers, by formally identifying sufficient requirements on the scheduler to ensure PSNI—a novel aspect if compared with previous work [20, 51]. We leverage on the generality of our result and prove that **MAC** is secure by instantiating our PSNI theorem with a round-robin scheduler, i.e., the scheduler used by GHC's runtime system.
3. We corroborate our results with an extensive mechanized proof developed in the Agda proof assistant that counts more than 4000 lines of code. The mechanization has provided us with stimulating insights and pinpointed problems in proofs of similar works.
4. We improve and simplify the term-erasure proof technique by proposing a novel flexible technique called *two-steps* erasure, which we utilize systematically to prove that many advanced features are secure, especially those that change the security level of other terms and detect exceptions.
5. We introduce a *functor* structure, a relabeling primitive and an *applicative* operator that give flexibility to programmers, by upgrading labeled values and conveniently aggregating heterogeneously labeled data.
6. We have released a prototype of our ideas in the **MAC** library[2].

Highlights This work builds on our previous papers "Flexible Manipulation of Labeled Values for Information-Flow Control Libraries" [57] and "On Formalizing Information-Flow Control Libraries" [58], which we have blended and significantly rewritten and corrected in a few technical inaccuracies. We have integrated these works with several examples and shaped them into a uniform, coherent and comprehensive story of this line of work. We summarize the novel contributions of this article as follows:

[1] Available at `https://github.com/marco-vassena/agda-mac`
[2] Available at `https://hackage.haskell.org/package/mac`

$$
\begin{array}{l}
\textbf{data}\ \mathit{Labeled}\ \ell\ a \\
\textbf{data}\ \mathit{MAC}\ \ell\ a \\
\textbf{instance}\ \mathit{Monad}\ (\mathit{MAC}\ \ell) \\
\mathit{label}\ \ \ :: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow a \rightarrow \mathit{MAC}\ \ell_{\mathrm{L}}\ (\mathit{Labeled}\ \ell_{\mathrm{H}}\ a) \\
\mathit{unlabel} :: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow \mathit{Labeled}\ \ell_{\mathrm{L}}\ a \rightarrow \mathit{MAC}\ \ell_{\mathrm{H}}\ a \\
\mathit{run}^{\mathsf{TCB}} :: \mathit{MAC}\ \ell\ a \rightarrow \mathit{IO}\ a
\end{array}
$$

Fig. 3: Core API for **MAC**.

- Uniform, coherent and comprehensive account of a formal model of **MAC**;
- Integration of examples in the description of the features of the library;
- Fixed several technical inaccuracies in the semantics of the calculus;
- Simplification and full account of the scheduler-parametric PSNI proof.

In the following, we point out the technical differences between this article and the conference version in footnotes.

This paper is organized as follows. Section 2 gives an overview of **MAC**. Section 3 formalizes the core of **MAC** in a simply-typed call-by-need lambda-calculus. Section 4 presents a secure primitive that regulates the interaction between computations at different security levels. Sections 5 and 6 extend the calculus with other advanced practical features, namely exceptions and mutable references. Section 7 proves that the sequential calculus satisfies progress-insensitive non-interference (PINI). Section 8 extends the calculus with concurrency and Section 9 presents functor, applicative, and relabeling operations. Section 10 gives the security guarantee of the concurrent calculus, which satisfies progress-sensitive non-interference (PSNI). Section 11 gives related work and Section 12 concludes.

2 Overview

In **MAC**, each label is represented as an abstract data type. Figure 3 shows the core part of **MAC**'s API. Abstract data type $\mathit{Labeled}\ \ell\ a$ classifies data of type a with a security label ℓ. For instance, $\mathit{pwd} :: \mathit{Labeled}\ H\ \mathit{String}$ is a sensitive string, while $\mathit{rating} :: \mathit{Labeled}\ L\ \mathit{Int}$ is a public integer. (Symbol :: is used to describe the type of terms in Haskell.) Abstract data type $\mathit{MAC}\ \ell\ a$ denotes a (possibly) side-effectful secure computation which handles information at sensitivity level ℓ and yields a value of type a as a result. A $\mathit{MAC}\ \ell\ a$ computation enjoys a monadic structure, i.e., it is built using the fundamental operations $\mathit{return} :: a \rightarrow \mathit{MAC}\ \ell\ a$ and $(\gg\!=) :: \mathit{MAC}\ \ell\ a \rightarrow (a \rightarrow \mathit{MAC}\ \ell\ b) \rightarrow \mathit{MAC}\ \ell\ b$ (read as "bind"). The operation $\mathit{return}\ x$ produces a computation that returns the value denoted by x and produces no side-effects. The function $(\gg\!=)$ is used to *sequence* computations and their corresponding side-effects. Specifically, $m \gg\!= f$ takes a computation m and function f which will be applied to the *result* produced by running m and yields the resulting computation.

We sometimes use Haskell's do-notation to write such monadic computations. For example, the program $m \gg\!= \lambda x \rightarrow \mathit{return}\ (x + 1)$, which adds 1 to the value produced by m, can be written as shown in Figure 4.

```
do x ← m
   return (x + 1)
```

Fig. 4: **do**-notation

2.1 Secure Flows of Information

Generally speaking, side-effects in a $\mathit{MAC}\ \ell\ a$ computation can be seen as actions which either read or write data. Such actions, however, need to be conceived in a manner that respects the sensitivity of the computations' results as well as the sensitivity of sources and sinks of information modeled as labeled values. The functions label and $\mathit{unlabel}$ allow $\mathit{MAC}\ \ell\ a$ computations to securely interact with labeled values. To help readers, we indicate the relationship between type variables in their subindexes, i.e., we use ℓ_{L} and ℓ_{H} to attest that $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$. If a $\mathit{MAC}\ \ell_{\mathrm{L}}$ computation writes data into a sink, the computation should have at most the sensitivity of the sink itself. This restriction, known as *no write-down* [5], respects the sensitivity of the data sink, e.g., the sink never receives data more sensitive than its label. In the case of function label, it creates a fresh labeled value, which from the security point of view can be seen as allocating a fresh location in memory and immediately writing a value into it—thus, it applies the no write-down principle. In the type signature of label, what appears on the left-hand side of the symbol $\Rightarrow$ are *type constraints*. They represent properties that must be statically fulfilled about the types appearing on the right-hand side of $\Rightarrow$. Type constraint $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$ ensures that when calling $\mathit{label}\ x$ (for some x in scope), the computation creates a labeled value only if ℓ_{L}, i.e. the current label of the computation, is no more confidential than ℓ_{H}, i.e. the sensitivity of the created labeled value. In contrast, a computation $\mathit{MAC}\ \ell_{\mathrm{H}}\ a$ is only allowed to read labeled values at most as sensitive as ℓ_{H}—observe the type constraint $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$ in the type signature of $\mathit{unlabel}$. This restriction, known as *no read-up* [5], protects the confidentiality degree of the result produced by $\mathit{MAC}\ \ell_{\mathrm{H}}\ a$, i.e. the result might only involve data ℓ_{L} which is, at most, as sensitive as ℓ_{H}.

We remark that **MAC** is an embedded domain specific language (EDSL), implemented as a Haskell *library* of around 200 lines of code and programs written in **MAC** are secure-by-construction. What makes it possible to provide strong security guarantees via a library is the fact that Haskell type-system enforces a strict separation between side-effect free code, which is guaranteed *not* to perform side effects, and side-effectful code, where side-effects may occur[3]. Specifically side-effects, i.e., input-output operations, can only occur in monadic computations of type $\mathit{IO}\ a$. Crucially *pure* computations are inherently secure, while IO computations are potentially leaky. In **MAC**, a secure

[3] In the functional programming community, they are also known as *pure* and *impure* code respectively.

computation of type $MAC\ \ell\ a$ is internally represented as a wrapper around an $IO\ a$ computation, that is used to implement side-effectful features, such as references and concurrency. **MAC** provides security-by-construction because *impure* operations, i.e., those of type IO, can only be constructed using **MAC** label-annotated API, which accepts only those that are statically deemed secure. Function run^{TCB} extracts the underlying $IO\ a$ computation from a secure computation of type $MAC\ \ell\ a$. Thanks to the secure-by-construction design, the IO computation so obtained is secure and can be executed directly, without the need of additional protection mechanism, such as monitors. Note that the function run^{TCB} is part of the Trusted Computing Base (TCB), i.e., it is available only to trusted code. In what follows, we describe an example which illustrates **MAC**'s programming model, particularly the use of $label$, $unlabel$.

Example The most common use of $label$ is to classify data to be protected. As an example, consider the Haskell program listed in Figure 5, which prompts the user for a password through the terminal and then passes it to a routine to check if the password is listed on dictionaries of commonly used passwords. Observe that the program performs input-output operations: $putStrLn :: String \rightarrow IO\ ()$ prints to standard output and $getLine :: IO\ String$ reads from standard input. Clearly the content of variable pwd should be handled with care by $isWeak :: String \rightarrow IO\ Bool$. In particular a computation of type $IO\ Bool$ can also perform arbitrary output operations and potentially leak the password. One way to protect pwd is by writing all password-related operations, like $isWeak$, within **MAC**, where pwd is marked as sensitive data.

```
p :: IO Bool
p = do
  putStrLn "Choose a password:"
  pwd ← getLine
  return (isWeak pwd)
```

Fig. 5: The password is exposed in $isWeak$.

Figure 6 shows the modifications to the code to secure $isWeak$. Observe how $label$ is used to mark pwd as sensitive by wrapping it inside a labeled expression of type $Labeled\ H\ String$. After that, the labeled password is passed to function $isWeak$ by bind $(\gg\!=)$ and function run^{TCB} executes the whole computation. Fixing the type of $isWeak$ appropriately, **MAC** prevents intentional or accidental *leakage* of the password. Several secure designs are possible, depending on how $isWeak$ provides its functionality. For example a secure interface could be $isWeak :: Labeled\ H\ String \rightarrow MAC\ L\ (Labeled\ H\ Bool)$, where the outermost computation $(MAC\ L)$ accounts for reading public data, e.g., fetching online dictionaries of common passwords, while the labeled result $(Labeled\ H\ Bool)$, protects the sensitivity of this piece of information about the password, namely if it is weak or not. The type $isWeak :: Labeled\ H\ String \rightarrow MAC\ H\ Bool$ is also secure and additionally allows to read from secret channels, e.g., file `/etc/shadow`, to check that the password is not reused.

```
p :: IO Bool
p = do putStrLn "Choose a password:"
       pwd ← getLine
       let lpwd = label pwd :: MAC L (Labeled H String)
       run^TCB (lpwd >>= isWeak)
```

Fig. 6: Label H protects the password in $isWeak$.

2.2 Implicit flows

The interaction between the current label of a computation and the no write-down restriction makes implicit flow ill-typed, as shown in Figure 7. In order to branch on sensitive data, a program needs first to unlabel it, thus requiring the computation to be of type $MAC\ H\ a$ (for some type a). From that point, the computation cannot write to public data regardless of the taken branch. As **MAC** provides additional primitives responsible for producing useful side-effects like exception handling, network communication, references, and synchronization primitives—we refer the interested reader to [45] for further details.

```
impl :: Labeled H Bool →
        MAC H (Labeled L Bool)
impl secret = do
  bool ← unlabel secret
    -- H ⋢ L
  if bool then label True
          else label False
```

Fig. 7: Implicit flows are ill-typed.

3 Core Calculus

This section formalizes **MAC** as a simply typed call-by-name λ-calculus extended with unit and boolean values and security primitives.

3.1 Pure Calculus

Figure 8 shows the formal syntax of the pure calculus underlying **MAC**, where meta variables τ, v and t denote respectively types, values, and terms. The typing judgment $\Gamma \vdash t : \tau$ denotes that term t has type τ assuming typing environment Γ. The typing rules of the pure calculus are standard and therefore omitted. The small-step semantics of the the calculus is represented by the relation $t_1 \rightsquigarrow t_2$, which denotes that term t_1 reduces to t_2. Rule [BETA] indicates that the calculus has *call-by-value* semantics, because the argument of a function, evaluated to weak-head normal form by rule [APP], is not evaluated upon function application, but rather substituted in the body—we write $t_1\ [x\ /\ t_2]$ for *capture-avoiding substitution*[4]. Rule [IF$_1$] evaluates the conditional of a if-then-else expression and rules [IF$_2$,IF$_3$] take the appropriate branch.

[4] In the machine-checked proofs all variables are De Bruijn indexes.

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

$$\text{(App)}\ \frac{t_1 \rightsquigarrow t_1'}{t_1\ t_2 \rightsquigarrow t_1'\ t_2} \qquad \text{(Beta)}\ (\lambda x.t_1)\ t_2 \rightsquigarrow t_1\ [t_2\ /\ x]$$

$$\text{(If}_1\text{)}\ \frac{t_1 \rightsquigarrow t_1'}{\textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3 \rightsquigarrow \textbf{if}\ t_1'\ \textbf{then}\ t_2\ \textbf{else}\ t_3} \qquad \text{(If}_2\text{)}\ \textbf{if}\ \mathit{True}\ \textbf{then}\ t_1\ \textbf{else}\ t_2 \rightsquigarrow t_1$$

$$\text{(If}_3\text{)}\ \textbf{if}\ \mathit{False}\ \textbf{then}\ t_1\ \textbf{else}\ t_2 \rightsquigarrow t_2$$

Fig. 8: Syntax and semantics of the pure calculus.

3.2 Core of MAC

We now extend this standard calculus with the security primitives of **MAC** as shown in Figure 9. Meta variable ℓ ranges over labels, which are assumed to form a lattice $(\mathscr{L}, \sqsubseteq)$. Labels are types in **MAC** despite we place them in a different syntactic category named ℓ—this decision is made merely for clarity of exposition. The new type $\mathit{Labeled}\ \ell\ \tau$ represents a (possibly side-effect free) resource, which annotates with the security level ℓ a value $t :: \tau$ wrapped in $\mathit{Labeled}$. For example, $\mathit{Labeled}\ 42 :: \mathit{Labeled}\ L\ \mathit{Int}$ is a public integer. In the following, we introduce further forms of labeled resources, in particular mutable references in Section 6 and synchronization variables in B. The actual **MAC** implementation handles more labeled resources and provides an uniform implementation for them [45][5]. The constructor $\mathit{Labeled}$ is not available to the user, who can only use label and $\mathit{unlabel}$ to create and inspect labeled resources, respectively.

A configuration $\langle \Sigma, t \rangle$ consists of a store Σ and a term t describing a computation of type $\mathit{MAC}\ \ell\ \tau$ and represents a secure computation at sensitivity level ℓ, which yields a value of type τ as result. For the moment, we ignore the store in the configuration (explained in Section 6). In order to enforce the security invariants, functions label and $\mathit{unlabel}$ live in the MAC monad and their type signatures ensure that the label of the resource is compatible with the security level of the current computation, as explained in the previous section. Besides those primitives, computations are created using the standard monad

[5] In our conference version [57, 58], we follow the original **MAC** paper [45] and represent all labeled resources using the same labeled data type $\mathit{Res}\ t :: \mathit{Res}\ \ell\ \tau$, where $t :: \tau$ determines the kind of resource. For example $\mathit{Res}\ (\mathit{Id}\ 42) :: \mathit{Res}\ \ell\ (\mathit{Id}\ \mathit{Int})$ is a term representing a public integer. Here, for clarity of exposition, we use separate data types for each labeled resource. This design choice does not affect our results.

Label: ℓ
Store: Σ
Types: $\tau ::= \cdots \mid \mathit{MAC}\ \ell\ \tau \mid \mathit{Labeled}\ \ell\ \tau$
Configuration: $c ::= \langle \Sigma, t \rangle$
Values: $v ::= \cdots \mid \mathit{return}\ t \mid \mathit{Labeled}\ t$
Terms: $t ::= \cdots \mid t_1 \gg\!= t_2 \mid \mathit{label} \mid \mathit{unlabel}\ t$

$$\text{(LIFT)}\ \frac{t \rightsquigarrow t'}{\langle \Sigma, t \rangle \longrightarrow \langle \Sigma, t' \rangle} \qquad \text{(BIND}_1\text{)}\ \frac{\langle \Sigma, t_1 \rangle \longrightarrow \langle \Sigma', t_1' \rangle}{\langle \Sigma, t_1 \gg\!= t_2 \rangle \longrightarrow \langle \Sigma', t_1' \gg\!= t_2 \rangle}$$

$$\text{(BIND}_2\text{)}\ \langle \Sigma, \mathit{return}\ t_1 \gg\!= t_2 \rangle \longrightarrow \langle \Sigma, t_2\ t_1 \rangle \qquad \text{(LABEL)}\ \langle \Sigma, \mathit{label}\ t \rangle \longrightarrow \langle \Sigma, \mathit{return}\ (\mathit{Labeled}\ t) \rangle$$

$$\text{(UNLABEL}_1\text{)}\ \frac{t \rightsquigarrow t'}{\langle \Sigma, \mathit{unlabel}\ t \rangle \longrightarrow \langle \Sigma, \mathit{unlabel}\ t' \rangle}$$

$$\text{(UNLABEL}_2\text{)}\ \langle \Sigma, \mathit{unlabel}\ (\mathit{Labeled}\ t) \rangle \longrightarrow \langle \Sigma, \mathit{return}\ t \rangle$$

Fig. 9: Core of **MAC**.

$$\text{(LABEL}^\tau\text{)}\ \frac{\ell_L \sqsubseteq \ell_H \qquad \Gamma \vdash t : \tau}{\Gamma \vdash \mathit{label}\ t : \mathit{MAC}\ \ell_L\ (\mathit{Labeled}\ \ell_H\ \tau)} \qquad \text{(UNLABEL}^\tau\text{)}\ \frac{\ell_L \sqsubseteq \ell_H \qquad \Gamma \vdash t : \mathit{Labeled}\ \ell_L\ \tau}{\Gamma \vdash \mathit{unlabel}\ t : \mathit{MAC}\ \ell_H\ \tau}$$

Fig. 10: Typing rules for *label* and *unlabel*.

operations *return* and $\gg\!=$. For easy exposition, in the following we give the type of **MAC**'s constructs as Haskell APIs. We explain their relation with traditional typing judgments by means of an example, see Figure 10. The typing rules [LABEL$^\tau$, UNLABEL$^\tau$] are *type scheme rules*, i.e., there is such a judgment for every label ℓ_L and $\ell_H \in \mathscr{L}$, such that $\ell_L \sqsubseteq \ell_H$, where labels come from either type signatures or explicit type annotations in programs, as we showed in the previous section. The *type constraints* in the API, i.e., what appears before the symbol $\Rightarrow$, is placed as a premise of the corresponding typing rule. We remark that type constrains are built using *type classes*, a well-established feature of Haskell type system, therefore we do not discuss them any further [59].

In Figure 9, we explicitly distinguish pure-term evaluation from top-level monadic-term evaluation, similarly to [53]. In particular, the relation $c_1 \longrightarrow c_2$ denotes monadic evaluation, which extends the evaluation of pure terms, i.e., $\rightsquigarrow$, via [LIFT]. The semantics rules in Figure 9 are fairly straight-forward and follow the pattern seen in the pure semantics, where some *context-rules*, e.g.

```
savePwd :: Labeled H String → MAC L (MAC H ())
savePwd lpwd = do putStrLn^MAC "Saving new password"
                  return (passwd pwd)
```

Fig. 11: A nested computation that writes at security level L and H.

[BIND$_1$, UNLABEL$_1$] reduce a redex subterm, and then the interesting rule fires, e.g. [BIND$_2$, UNLABEL$_2$]. In particular rule [BIND$_1$] executes the computation on the left of the bind and rule [BIND$_2$] extracts the result of the computation and feeds it to the right-side argument of $(\gg\!=)$. Rule [UNLABEL$_1$] evaluates the argument to labeled expression and rule [UNLABEL$_2$] returns its content. Rule [LABEL] creates a labeled expression by wrapping the argument in *Labeled* and returns it in the security monad. It is worth noting that thanks to the static nature of **MAC**, no run-time checks are needed to prevent insecure flows of information in these rules.

4 Label Creep

Let us continue the password example from the introduction. After checking that the password is strong enough, the program replaces the old password with the new one by updating file `/etc/shadow` with the new hashed password, using primitive *passwd* :: *Labeled* H *String* $\rightarrow$ *MAC* H ()—note that the label of the computation is H, in order to unlabel the password and hash it. We consider the hash of a password as sensitive data: it should not be leaked in order to prevent *offline* dictionary attacks [19,38]. The program should also inform the user that the password is being saved by printing on the screen a message. We consider printing on the screen as a public write operation, i.e., *putStrLn*$^{\text{MAC}}$ :: *String* $\rightarrow$ *MAC* L (). Figure 11 shows the code of the discussed routine. Observe that *putStrLn*$^{\text{MAC}}$ `"Saving new password"` :: *MAC* L () and *passwd pwd* :: *MAC* H () belong to different *MAC* computations. Therefore, both operations cannot coexist together, otherwise secret data, e.g., the password, could be unlabeled and then leaked on a public channel, e.g., standard output. Specifically the program in Figure 12 is rejected as ill-typed. Programs that handle data and channels with heterogeneous labels necessarily involve *nested* *MAC* ℓ a computations in its return type. In this case, the type of *savePwd lpwd* :: *MAC* L (*MAC* H ()) indicates that it is a public computations, which prints on the screen, and that *produces* as value a sensitive computation *MAC* H *Int*, which lastly writes to the sensitive file. Obviously having to nest computations

```
putStrLn^MAC "Saving ..."
passwd lpwd
```

Fig. 12: Ill-typed ($L \not\equiv H$).

$$join :: \ell_L \sqsubseteq \ell_H \Rightarrow MAC\ \ell_H\ \tau \rightarrow MAC\ \ell_L\ (Labeled\ \ell_H\ \tau)$$

Fig. 13: Primitive $join$.

Terms: $t ::= \cdots \mid join\ t$

$$\text{(JOIN)}\quad \frac{\langle \Sigma, t\rangle \Downarrow \langle \Sigma', return\ t'\rangle}{\langle \Sigma, join\ t\rangle \longrightarrow \langle \Sigma', return\ (Labeled\ t')\rangle}$$

Fig. 14: Calculus with $join$.

complicates the programming model of **MAC** and hinders its applicability[6]. We recognize this pattern of returning nested computations as a static version of a problem known in dynamic systems as *label creep* [11,46]—which occurs when the context gets tainted to the point that no useful operations are allowed anymore. In $savePwd$, it is necessary to do all the public computation first and the all the sensitive ones. In a *sequential setting*, **MAC** provides the primitive $join$[7], which alleviates this problem by safely *integrating* more sensitive computations into less sensitive ones.

4.1 Primitive *join*

Figure 13 shows the type signature of $join$. Intuitively, function $join$ runs the computation of type $MAC\ \ell_H\ \tau$ and wraps the result into a labeled expression to protect its sensitivity. As we will show in Section 7.4, programs written using the monadic API, $label$, $unlabel$, and $join$ satisfy *progress-insensitive non-interference* (PINI), where leaks due to non-termination of programs are ignored. This design decision is similar to that taken by mainstream IFC compilers (e.g., [17,37,49]), where the most effective manner to exploit termination takes exponential time in the size (of bits) of the secret [1]. In the semantics, Figure 14 extends terms with the new primitive $join\ \ t$. Rule [JOIN] formalizes the semantics of $join$ using big-step semantics—similar to other work [45,54], we restrict ourselves to terminating computations. We write $\langle \Sigma, t\rangle \Downarrow \langle \Sigma', v\rangle$ if and only if v is a value and $\langle \Sigma, t\rangle \longrightarrow^* \langle \Sigma', v\rangle$, where relation $\longrightarrow^*$ denotes the reflexive transitive closure of $\longrightarrow$. Rule [JOIN] executes the secure computation $t \Downarrow return\ t'$ and wraps the result t in $Labeled$ to protect its sensitivity.[8]

[6] Remember that Haskell employs lazy evaluation, therefore the inner computations is not automatically evaluated, but needs to be explicitly executed. Only trusted code, using run^{TCB} can force evaluation of MAC computations.

[7] Not to be confused with the monadic $join :: Monad\ m \Rightarrow m\ (m\ a) \rightarrow m\ a$.

[8] We refrain from using $label\ t_2$ because we will soon add exceptions to secure computations.

```
savePwd :: Labeled H String → MAC L ()
savePwd lpwd = do putStrLn^MAC "Saving new password"
                  join (passwd pwd)
                  putStrLn^MAC "Password saved"
```

Fig. 15: Example revisited with *join*.

```
throw :: χ → MAC ℓ τ
catch :: MAC ℓ τ → (χ → MAC ℓ τ) → MAC ℓ τ
```

Fig. 16: API for exceptions.

Revisited Example In Figure 15 we simplify program *savePwd*, by replacing *return* with *join*. Observe that the return type of *savePwd* does not involve nested computations, therefore the execution of the sensitive computation is not suspended, but rather follows directly after the public print statement.

5 Exception Handling

Exception handling is a common programming language mechanism used to signal some anomalous condition and stop the execution of a program. It is sometimes possible to recover from such exceptional circumstances and resume execution afterwards. For instance, consider again the program *savePwd* in Figure 15. If primitive *passwd* fails due to some IO exception, e.g., file `etc/shadow` has already been opened or has not been found, the whole program crashes. Not supporting exceptions in the context of input-output operations, is not only impeding our programming model, but it is also insecure. In fact, exceptions change the control flow of a program, and an uncaught exception can propagate throughout a program and eventually crash it, potentially suppressing public events. For example, if *passwd* throws an exception, the program aborts before printing `"Password saved"` on the screen. Observe that, such behavior constitutes a leak, because the failure comes from a sensitive context, i.e., *passwd*, and therefore can depend on the value of the secret, i.e., the password. In this section, we incorporate exception handling primitives in **MAC** to remedy this situation, see Figure 16. Intuitively, *catch* t_1 t_2 runs the computation t_1 and recovers from a failure by passing the exception to the exception handler t_2. Section 5.2 discusses some subtleties between exception handling primitives and *join*, which may propagate exceptions from sensitive contexts to less sensitive ones, if neglected.

Types: $\tau ::= \cdots \mid \chi$
Values: $v ::= \cdots \mid \xi \mid \mathit{throw}\ t$
Terms: $t ::= \cdots \mid \mathit{catch}\ t_1\ t_2$

$$(\textsc{Bind}_\chi)\quad \langle \Sigma, \mathit{throw}\ t_1 \gg\!= t_2\rangle \longrightarrow \langle \Sigma, \mathit{throw}\ t_1\rangle$$

$$(\textsc{Catch}_1)\quad \frac{\langle \Sigma, t_1\rangle \longrightarrow \langle \Sigma', t_1'\rangle}{\langle \Sigma, \mathit{catch}\ t_1\ t_2\rangle \longrightarrow \langle \Sigma', \mathit{catch}\ t_1'\ t_2\rangle}$$

$$(\textsc{Catch}_2)\quad \langle \Sigma, \mathit{catch}\ (\mathit{return}\ t_1)\ t_2\rangle \longrightarrow \langle \Sigma, \mathit{return}\ t_1\rangle$$

$$(\textsc{Catch}_3)\quad \langle \Sigma, \mathit{catch}\ (\mathit{throw}\ t_1)\ t_2\rangle \longrightarrow \langle \Sigma, t_2\ t_1\rangle$$

Fig. 17: Exception handling primitives.

Values: $v ::= \cdots \mid \mathit{Labeled}_\chi\ t$

$$(\textsc{Join}_\chi)\quad \frac{\langle \Sigma, t\rangle \Downarrow \langle \Sigma', \mathit{throw}\ t'\rangle}{\langle \Sigma, \mathit{join}\ t\rangle \longrightarrow \langle \Sigma', \mathit{return}\ (\mathit{Labeled}_\chi\ t')\rangle}$$

$$(\textsc{Unlabel}_\chi)\quad \mathit{unlabel}\ (\mathit{Labeled}_\chi\ t) \rightsquigarrow \mathit{throw}\ t$$

Fig. 18: Secure interaction between join and exceptions.

5.1 Calculus

For simplicity, we consider only one exception $\xi :: \chi$, where χ denotes an exception type. In the calculus, we extend terms with ξ, $\mathit{throw}\ t$, and $\mathit{catch}\ t_1\ t_2$—see Figure 17. Term $\mathit{throw}\ t$ aborts the current MAC computation with exception t, see rule [$\textsc{Bind}_\chi$]. Term $\mathit{catch}\ t_1\ t_2$ evaluates computation t_1 via rule [$\textsc{Catch}_1$], and either it returns the result, if the computation succeeds, i.e., rule [$\textsc{Catch}_2$], or it attempts to recover a failure by running exception handler t_2, if the computation throws an exception, i.e., rule [$\textsc{Catch}_3$].

5.2 Join and Exceptions

The interplay between exceptions and join is delicate and security might be at stake if these two features were naively combined [23,53]. Observe that the type signatures in Figure 16 hint that exceptions can be thrown and caught among computations with the same label—a design decision which does not break security guarantees. Nevertheless, information can be leaked if exceptions thrown

```
fetchDict :: String → MAC L [String]
fetchDict lang = readFile "usr/share/dict" ++ "-" ++ lang

fetchCacheDict : Ref L (Map String [String]) → String → MAC L [String]
fetchCacheDict r lang = do
  dicts ← read r
  case lookup lang dicts of
    Just dict → return dict
    Nothing → do
      dict ← fetchDict lang
      write (insert dict dicts) r
      return dict
```

Fig. 19: *fetchCacheDict* is a cached version of *fetchDict*.

in sensitive computations are propagated to less sensitive ones. From now on, we refer to exceptions raised in a sensitive *MAC* computation as *sensitive exceptions*. In fact, sensitive exceptions can affect the control-flow of less sensitive computations and thus suppressing observable events, giving place to an implicit flow[9]. In our calculus, *join* is the only primitive that combines computations with different labels and thus is potentially vulnerable to this attack. In order to close leaks via exceptions, **MAC** modifies the semantics of *join* to *mask* exceptions, preventing them to propagate to less sensitive computations—this solution is similar to previous work [23, 53].

Figure 18 implements this countermeasure. Firstly it adds a new internal constructor $Labeled_\chi\ t$ denoting a labeled value (of type $Labeled\ \ell\ \tau$) which contains inside the exception $(t :: \chi)$. Rule [JOIN$_\chi$] shows the semantics for $join\ t$ when exceptions are triggered: *exceptions are not propagated further but rather returned inside a labeled expression*. Under this programming model, it is necessary to inspect the return value of *join* to determine if the computation terminated abnormally. The attacker must then *unlabel* the result to observe the exception, see rule [UNLABEL$_\chi$]. Observe that, since this operation is subject to *no read-up*, sensitive exceptions are not observable from less sensitive computations. As a consequence of this programming model, *only* sensitive computations can handle sensitive exceptions. Consider again the example from Figure 15. Note that program *savePwd* prints `"Password Saved"` even though *passwd* might have actually failed: it would be insecure to do otherwise! The only way to observe and recover from a failure of *passwd*, without compromising security, is to explicitly surround it with a *catch* block, i.e., *catch* (*passwd pwd*) *handler*.

$$
\begin{aligned}
&\textbf{data}\ \mathit{Ref}\ \ell\ \tau\\
&\mathit{new} :: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow \tau \to \mathit{MAC}\ \ell_{\mathrm{L}}\ (\mathit{Ref}\ \ell_{\mathrm{H}}\ \tau)\\
&\mathit{read} :: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow \mathit{Ref}\ \ell_{\mathrm{L}}\ \tau \to \mathit{MAC}\ \ell_{\mathrm{H}}\ \tau\\
&\mathit{write} :: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow \tau \to \mathit{Ref}\ \ell_{\mathrm{H}}\ \tau \to \mathit{MAC}\ \ell_{\mathrm{L}}\ ()
\end{aligned}
$$

Fig. 20: API for references.

6 References

Mutable references are an imperative feature often needed to boost the performance of algorithms. Following the password example from the previous sections, we might want to reject weak password that are vulnerable to dictionary attacks. To do that, in Figure 19, function *fetchDic* fetches the list of words on a dictionary available in the system—we consider the content of a dictionary to be public information therefore the computation has security level L. Depending on the local system language, we can tweak the function to pick an appropriate dictionary, for example *fetchDic* `"en"` fetches English words from dictionary `"usr/share/dict-en"`. A password-strength checker application could test a password against multiple dictionaries, which would require to call *fetchDict* multiple times. Since dictionaries are seldom changed, it is wasteful to fetch the same dictionary multiple times, therefore, using references, we implement a simple caching mechanism that avoids the overhead. Function *fetchCacheDict* takes as an extra argument a reference to a table of cached dictionaries, i.e., $\mathit{Ref}\ L\ (\mathit{Map}\ \mathit{String}\ [\mathit{String}])$. When the language *lang* dictionary is needed, the function reads the cached table (*dicts*) from the reference ($\mathit{read}\ r$) and checks if it has already been fetched ($\mathit{lookup}\ \mathit{lang}\ \mathit{dicts}$). If it is a hit (case $\mathit{Just}\ \mathit{dict}$), the dictionary is returned directly without the need of any IO operation. Otherwise (case *Nothing*), the dictionary is fetched with *fetchDict*, the result cached ($\mathit{write}\ (\mathit{insert}\ \mathit{dict}\ \mathit{dicts})\ r$) and returned.

6.1 Calculus

Figure 21 extends the calculus with mutable references, another feature available in **MAC**. Memory is compartmentalized into isolated labeled segments[10], one for each label of the lattice, and accessed exclusively through the store Σ. A memory in the category $\mathit{Memory}\ \ell$ contains terms at security level ℓ. We use the standard list interface $[\,]$, $t : t_s$ and $t_s[n]$ for the empty list, the insertion of a term into an existing list and accessing the nth-element, respectively. We write $\Sigma(\ell)[n]$ to retrieve the nth-cell in the ℓ-memory. The notation $\Sigma(\ell)[n] := t$

[9] We refer interested readers to [45] for further details about this attack.

[10] A split memory model simplifies the proofs because allocation in one segment does not affect allocation in another. We argue why this model is reasonable and discuss alternatives in Section 7.

$$
\begin{array}{lll}
\text{Store:} & \Sigma ::= (\ell : \mathit{Label}) \to \mathit{Memory}\ \ell & \\
\text{Memory } \ell & t_s ::= [\,] \mid t : t_s & \\
\text{Addresses:} & n ::= 0 \mid 1 \mid 2 \mid \cdots & \\
\text{Types:} & \tau ::= \cdots \mid \mathit{Ref}\ \ell\ \tau & \\
\text{Values:} & v ::= \cdots \mid \mathit{Ref}\ n & \\
\text{Terms:} & t ::= \cdots \mid \mathit{new}\ t \mid \mathit{read}\ t \mid \mathit{write}\ t_1\ t_2 &
\end{array}
$$

$$
\text{(NEW)}\quad \frac{|\Sigma(\ell)| = n}{\langle \Sigma, \mathit{new}\ t\rangle \longrightarrow \langle \Sigma(\ell)[n] := t, \mathit{return}\ (\mathit{Ref}\ n)\rangle}
$$

$$
\text{(WRITE}_1\text{)}\quad \frac{t_1 \rightsquigarrow t_1'}{\langle \Sigma, \mathit{write}\ t_1\ t_2\rangle \longrightarrow \langle \Sigma, \mathit{write}\ t_1'\ t_2\rangle}
$$

$$
\text{(WRITE}_2\text{)}\quad \langle \Sigma, \mathit{write}\ (\mathit{Ref}\ n)\ t\rangle \longrightarrow \langle \Sigma(\ell)[n] := t, \mathit{return}\ ()\rangle
$$

$$
\text{(READ}_1\text{)}\quad \frac{t \rightsquigarrow t'}{\langle \Sigma, \mathit{read}\ t\rangle \longrightarrow \langle \Sigma, \mathit{read}\ t'\rangle}
\qquad
\text{(READ}_2\text{)}\quad \langle \Sigma, \mathit{read}\ (\mathit{Ref}\ n)\rangle \longrightarrow \langle \Sigma, \mathit{return}\ \Sigma(\ell)[n]\rangle
$$

Fig. 21: **MAC** with references.

denotes the store obtained by performing the update $\Sigma(\ell)[n \mapsto t]$. Secure computations create, read and write references using primitives new, read and write respectively. Observe that their types are restricted according to the *no read-up* and *no write-down* rules, like those of label and $\mathit{unlabel}$—see Figure 20. A reference is represented as a value $\mathit{Ref}\ \ n :: \mathit{Ref}\ \ \ell\ \ \tau$ where n is an address[11], pointing to the n-th cell of the ℓ-memory, which contains a term of type τ. Rule [NEW] extends the ℓ-labeled memory with the new term and returns a reference to it. The notation $|t_s|$ denotes the length of a list and is used to compute the address of a new reference—memories are zero-indexed. Rule [WRITE_1] evaluates its first argument to a reference and rule [WRITE_2] overwrites the content of the memory cell pointed by the reference and returns unit. Similarly, rule [READ_2] retrieves the term stored in memory and pointed to by the reference, which is evaluated via rule [READ_1].

7 Soundness

This section formally presents the security guarantees of the sequential calculus. Section 7.1 describes the proof technique (*term erasure*), Section 7.2 defines

[11] **MAC**'s implementation of labeled reference is a simple wrapper around Haskell's type IORef . However, we denote references as a simple index into the labeled memory. This design choice does not affect our results.

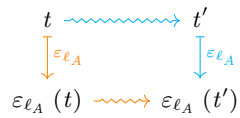

Fig. 22: Single-step simulation.

$$\varepsilon_{\ell_A}(\mathit{Labeled}\ t :: \mathit{Labeled}\ \ell_{\mathrm{H}}\ \tau) = \begin{cases} \mathit{Labeled}\ \bullet & \text{if } \ell_{\mathrm{H}} \not\sqsubseteq \ell_A \\ \mathit{Labeled}\ \varepsilon_{\ell_A}(t) & \text{otherwise} \end{cases}$$

$$\varepsilon_{\ell_A}(\mathit{label}\ t :: \mathit{MAC}\ \ell_{\mathrm{L}}\ (\mathit{Labeled}\ \ell_{\mathrm{H}}\ \tau)) = \begin{cases} \mathit{label}\ \bullet & \text{if } \ell_{\mathrm{H}} \not\sqsubseteq \ell_A \\ \mathit{label}\ \varepsilon_{\ell_A}(t) & \text{otherwise} \end{cases}$$

Fig. 23: Term erasure for labeled values.

the erasure function and Section 7.4 concludes with the *progress-insensitive non-interference* theorem (PINI).

7.1 Term Erasure

Term erasure is a proof technique to prove non-interference in functional programs. It was firstly introduced by Li and Zdancewic [30] and then used in a subsequent series of work on information-flow libraries [20, 43, 51, 53, 54]. The technique relies on an erasure function on terms, which we denote by ε_{ℓ_A}. This function essentially rewrites data above the attacker's security level, denoted by label ℓ_A, to the special syntax node $\bullet$. Once ε_{ℓ_A} is defined, the core of the proof technique consists of proving an essential relationship about the erasure function and reduction steps. The diagram in Figure 22 highlights this intuition. It shows that erasing sensitive data from a term t and then taking a step (orange path) is the same as firstly taking a step and then erasing sensitive data (cyan path), i.e., the diagram *commutes*. If term t leaks data whose sensitivity label is above ℓ_A, then erasing all sensitive data first and then taking a step might not be the same as taking a step and then erasing secret values—the leaked sensitive data in t' might remain in $\varepsilon_{\ell_A}(t')$ after all. From now on, we refer to this relationship as the *single-step simulation* between regular terms and erased ones.

7.2 Erasure Function

We proceed to define the erasure function for the pure calculus. Since security levels are at the type-level, the erasure function is type-driven. We write $\varepsilon_{\ell_A}(t :: \tau)$ for the erasure of term t with type τ of data not observable by the attacker. We omit the type annotation when it is either irrelevant or clear from the context.

$$\varepsilon_{\ell_A}(\langle \Sigma, t :: \mathit{MAC}\ \ell_H\ \tau\rangle) = \begin{cases} \langle \varepsilon_{\ell_A}(\Sigma), \bullet\rangle & \text{if } \ell_H \not\sqsubseteq \ell_A \\ \langle \varepsilon_{\ell_A}(\Sigma), \varepsilon_{\ell_A}(t)\rangle & \text{otherwise} \end{cases}$$

(a) Erasure for configuration.

$$\varepsilon_{\ell_A}(t_s :: \mathit{Memory}\ \ell_H) = \begin{cases} \bullet & \text{if } \ell_H \not\sqsubseteq \ell_A \\ \mathit{map}\ \varepsilon_{\ell_A}\ t_s & \text{otherwise} \end{cases}$$

(b) Erasure for memory.

$$\varepsilon_{\ell_A}(\mathit{Ref}\ n :: \mathit{Ref}\ \ell_H\ \tau) = \begin{cases} \mathit{Ref}\ \bullet & \text{if } \ell_H \not\sqsubseteq \ell_A \\ \mathit{Ref}\ n & \text{otherwise} \end{cases}$$

$$\varepsilon_{\ell_A}(\mathit{new}\ t :: \mathit{MAC}\ \ell_L\ (\mathit{Ref}\ \ell_H\ \tau)) = \begin{cases} \mathit{new}_\bullet\ \varepsilon_{\ell_A}(t) & \text{if } \ell_H \not\sqsubseteq \ell_A \\ \mathit{new}\ \varepsilon_{\ell_A}(t) & \text{otherwise} \end{cases}$$

$$\varepsilon_{\ell_A}(\mathit{write}\ t_1\ t_2) = \begin{cases} \mathit{write}_\bullet\ \varepsilon_{\ell_A}(t_1)\ \varepsilon_{\ell_A}(t_2 :: \mathit{Ref}\ \ell_H\ \tau) & \text{if } \ell_H \not\sqsubseteq \ell_A \\ \mathit{write}\ \varepsilon_{\ell_A}(t_1)\ \varepsilon_{\ell_A}(t_2) & \text{otherwise} \end{cases}$$

(c) Erasure for references and memory primitives.

Fig. 24: Erasure for configuration, store and memory primitives.

Ground values (e.g., True) are unaffected by the erasure function and, for most terms, the function is homomorphically applied, e.g., $\varepsilon_{\ell_A}(t_1\ t_2 :: \tau) = \varepsilon_{\ell_A}(t_1 :: \tau' \to \tau)\ \varepsilon_{\ell_A}(t_2 :: \tau')$. Figure 23 shows the definition of the erasure functions for the interesting cases. The *content* of a resource of type $\mathit{Labeled}\ \ell_H\ \tau$ is rewritten to $\bullet$ if the label is sensitive, i.e., it is not visible to the attacker's label ($\ell_H \not\sqsubseteq \ell_A$), otherwise it is erased homomorphically. Similarly the erasure function rewrites the argument of label to $\bullet$, if it gets labeled with a sensitive label or otherwise erased homomorphically. Observe that this definition respects the commutativity of the diagram in Figure 22 for rule [LABEL].

Figure 24 shows the erasure function for configuration, store and memory primitives. A configuration $\langle \Sigma, t\rangle$ is erased by erasing the store Σ and by rewriting term t to $\bullet$, if it represents a *sensitive* computation, i.e., if term t has type $\mathit{MAC}\ \ell_H\ \tau$, where ($\ell_H \not\sqsubseteq \ell_A$), and homomorphically otherwise, see Figure 24a. It is worth pointing out that, different from the conference version of this work [58], the erasure of a term $t :: \mathit{MAC}\ \ell_H\ \tau$, where $\ell_H \not\sqsubseteq \ell_A$ is homomorphic if the term is considered in *isolation*. Intuitively the term alone is just the

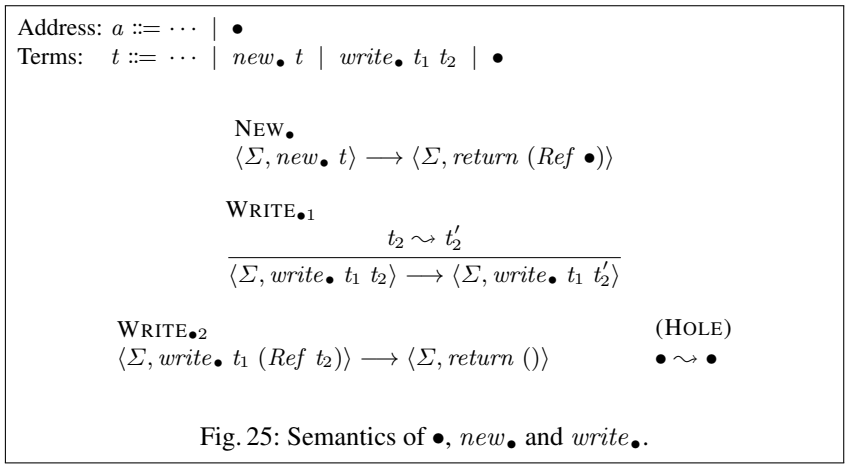

Fig. 25: Semantics of $\bullet$, $new_\bullet$ and $write_\bullet$.

description of a secure computation[12], which can be executed only if paired with a store in a configuration, where instead is aggressively erased to $\bullet$ as shown in Figure 24a. The store Σ is erased pointwise by erasing the memories at each security level, i.e., $\varepsilon_{\ell_A}(\Sigma) = \lambda\ell.\varepsilon_{\ell_A}(\Sigma(\ell))$, see Figure 24b. The erasure function collapses sensitive memories completely by rewriting them to $\bullet$ and erase non-sensitive ones homomorphically. Figure 24c shows the erasure of references, whose address is rewritten to $\bullet$ if sensitive, and primitive new and $write$, which is non-standard. Observe that these primitive perform a *write* effect and due to the *no write-down* rule they can only affect memories at least as sensitive as the current secure computation. When these operations constitute a *sensitive write*, i.e., they involve memories not visible to the attacker ($\ell_H \not\sqsubseteq \ell_A$), we employ a technique called *two-steps erasure*—a novel approach if compared with previous papers (e.g., [52]). Rather than being a pure syntactic procedure, erasure is also performed by additional evaluation rules, triggered by special constructs introduced by the erasure function. Specifically the erasure function replaces constructs new and $write$ with special constructs $new_\bullet$ and $write_\bullet$, whose semantics simulates that of the original terms with a *no-operation*—see Figure 25. In particular rule [NEW$_\bullet$] leaves the store Σ unchanged (the argument to $new_\bullet$ is ignored), and returns a dummy reference with address $\bullet$. The same principle applies to $write_\bullet$. Rule [WRITE$_{\bullet 1}$] evaluates the second argument to a reference, simulating rule [WRITE$_1$] and [WRITE$_{\bullet 2}$] skips the write and just returns unit. Note that the semantics of $new_\bullet$ and $write_\bullet$ correctly captures the unchanged observational power of an attacker performing *sensitive write* operations. We remark that $\bullet$, $new_\bullet$ and $write_\bullet$ and their semantics rules are introduced in the

[12] Observe that in [58] this was not the case, because rule [UNLABEL$_2$] and [BIND$_2$] where given as pure reductions ($\rightsquigarrow$). By separating the pure semantics from the top-level monadic semantics, we simplify the formalization of applicative functors, see Section 10.1.

Terms: $t ::= \cdots \mid \mathit{join}_\bullet\ t$

$$\varepsilon_{\ell_A}(\mathit{join}\ t :: \mathit{MAC}\ \ell_L\ (\mathit{Labeled}\ \ell_H\ \tau)) = \begin{cases} \mathit{join}_\bullet\ \varepsilon_{\ell_A}(t) & \text{if } \ell_H \not\sqsubseteq \ell_A \\ \mathit{join}\ \varepsilon_{\ell_A}(t) & \text{otherwise} \end{cases}$$

$$\varepsilon_{\ell_A}(\mathit{Labeled}_\chi\ t :: \mathit{Labeled}\ \ell_H\ \tau) = \begin{cases} \mathit{Labeled}\ \bullet & \text{if } \ell_H \not\sqsubseteq \ell_A \\ \mathit{Labeled}_\chi\ \varepsilon_{\ell_A}(t) & \text{otherwise} \end{cases}$$

$$(\textsc{Join}_\bullet)$$
$$\langle \Sigma, \mathit{join}_\bullet\ t\rangle \longrightarrow \langle \Sigma, \mathit{return}\ (\mathit{Labeled}\ \bullet)\rangle$$

Fig. 26: Erasure of join and $\mathit{Labeled}_\chi$ and semantics of $\mathit{join}_\bullet$.

calculus due to mere technical reasons (as explained above)—they are not part of the surface syntax nor **MAC**.

Figure 26 shows the erasure function for the remaining terms of the sequential calculus, that is join and $\mathit{Labeled}_\chi$. Using the same technique that we have described previously, we replace join with special term $\mathit{join}_\bullet$, when it is used to run a sensitive computation ($\ell_H \not\sqsubseteq \ell_A$). Erasure is then performed by means of rule [$\textsc{Join}_\bullet$], which immediately returns a dummy labeled value ($\mathit{Labeled}\ \bullet$) and the store unchanged. The rule captures the observational power of an attacker that runs a *terminating* sensitive computation. Observe in particular that the rule does not need to run the sensitive computation: the store can only be changed in sensitive memories (*no write-down*), which are not visible to the attacker, and the result of the computation is irrelevant—the attacker cannot unlabel it (*no read-up*), because it is marked as sensitive. What about computations that fail with an exception? In Figure 26, the erasure function not only rewrites the content of a sensitive exception to $\bullet$, as expected, but it also *masks* its exceptional nature, by replacing the constructor $\mathit{Labeled}_\chi$ with $\mathit{Labeled}$, thus ensuring that rule [$\textsc{Join}_\bullet$] simulates rule [$\textsc{Join}_\chi$] as well. Crucially, we have the freedom of choosing this definition without breaking *simulation*, because no other construct can detect, either explicitly or implicitly, the difference. For instance, rule [$\textsc{Unlabel}_\chi$] operates on labeled expressions containing exceptions. In this case, if the labeled exception is not visible to the attacker, then $\mathit{unlabel}$ must be performed in a non-visible computation as well, due to the typing rules. Operation $\mathit{unlabel}$ then gets rewritten to $\bullet$ and the step is then simulated by rule [$\textsc{Hole}$] instead. As a result of that, and unlike the approach taken by Stefan et al. in [53], there are no sensitive labeled exceptions in erased terms.

7.3 Discussion

Term Erasure We prove the single-step simulation directly over the small-step reduction relation. Instead, other works [20, 30, 43, 51, 53, 54] prove the simulation by relating operational semantics step reductions (upper part in Figure 22)

with reductions on a ℓ_A-indexed small-step relation of the form $c \longrightarrow_\ell \varepsilon_{\ell_A}(c')$, i.e., a relation which applies erasure at every reduction step. The reason for that is wired deeply in the dynamic nature of the enforcement. For instance, **LIO** considers labels as terms, which makes difficult to know what data is sensitive until run-time. In contrast, **MAC** does not need such an auxiliary construction because, due to its static nature, labels are not terms but rather type-level entities and therefore known before execution. In this light, our erasure function can safely erase any sensitive information found in labeled terms according to their type. Our small-step semantics satisfies type-preservation, i.e., reduction does not change types of terms, therefore labels are unaffected by execution—freeing us from the need to use a special small-step relation like $\longrightarrow_\ell$.

Masking Sensitive Exceptions In previous work, labeled exceptions are erased by erasing their content according to their label, but always preserving their exceptional state [53]. In contrast, we mask sensitive exceptions in erased programs. More specifically, erasing sensitive exceptions always results in erased unexceptional values—in other words, there are no sensitive exceptions in erased programs. Note that the simulation between terms and their erased counterparts guarantees that this rewriting is *sound*. In particular sensitive exception handling routines, the only routines which can distinguish exceptional from unexceptional sensitive values, gets also erased and do not occur in erased programs.

Memory It is known that dealing with dynamic allocation of memory makes it challenging to prove non-interference (e.g., [2, 18]). One manner to tackle this technicality is by establishing a bijection between public memory addresses of the two executions we want to relate and considering equality of public terms up to such notion [2]. Instead, and similar to other work [20, 52], we compartmentalize the memory into isolated labeled segments, one for each label of the lattice. This way, allocation in one segment does not affect the others. The fact that GHC's memory is non-split, does not compromise our security guarantees, because references are part of **MAC**'s internals and they cannot be inspected or deallocated explicitly. In the conference version of this work [58], we have explored an alternative way to prove *single-step* simulation for terms new and $write$ consists in extending the semantics of memory operations to node $\bullet$, i.e., by defining $\mid \bullet \mid \; = \bullet$ and $\bullet[\bullet \mapsto t] = \bullet$. Thanks to two-steps erasure, we can prove simulation as we did here, without recurring to a non-standard memory semantics. Having an split-memory model requires some care when proving non-interference, and in fact, we have identified problems with the proofs in manuscripts and articles related to **LIO** [53, 54]. We refer interested readers to Appendix B of our conference version [58] for details about the mistakes.

7.4 Progress-Insensitive Non-Interference

The sequential calculus that we have presented satisfies *progress-insensitive non-interference*. The proof of this result is based on two fundamental properties:

single-step simulation and *determinancy* of the small step semantics. In the following, we assume well-typed terms.

Proposition 1 (Single-step Simulation) *If* $c_1 \longrightarrow c_2$ *then* $\varepsilon_{\ell_A}(c_1) \longrightarrow \varepsilon_{\ell_A}(c_2)$.

Proof 1 *By induction on the reduction steps and typing judgment.*

Sensitive computations are simulated by transition $\langle \Sigma, \bullet \rangle \longrightarrow \langle \Sigma, \bullet \rangle$, obtained by lifting rule [HOLE] with [PURE]. Non-sensitive computations are simulated by the same rule that performs the non-erased transition, except when it involves some *sensitive write* operations, e.g., in rules [NEW, WRITE_1, WRITE_2, JOIN, JOIN_χ], which are simulated by rules [$\text{NEW}_\bullet$, $\text{WRITE}_{\bullet 1}$, $\text{WRITE}_{\bullet 2}$, $\text{JOIN}_\bullet$].

Proposition 2 (Determinancy) *If* $c_1 \longrightarrow c_2$ *and* $c_1 \longrightarrow c_3$ *then* $c_2 \equiv c_3$.

Proof 2 *By standard structural induction on the reductions.*

Before stating progress-insensitive non-interference, we define low-equivalence for configurations.

Definition 1 **(ℓ_A-equivalence).** *Two configurations* c_1 *and* c_2 *are indistinguishable from an attacker at security level* ℓ_A*, written* $c_1 \approx_{\ell_A} c_2$*, if and only if* $\varepsilon_{\ell_A}(c_1) \equiv \varepsilon_{\ell_A}(c_2)$.

Using Proposition 1 and 2, we show that our semantics preserves ℓ_A-equivalence.

Proposition 3 ($\approx_{\ell_A}$ Preservation) *If* $c_1 \approx_{\ell_A} c_2$, $c_1 \longrightarrow c_1'$, *and* $c_2 \longrightarrow c_2'$, *then* $c_1' \approx_{\ell_A} c_2'$.

By repeatedly applying Proposition 3, we prove progress-insensitive non-interference.

Theorem 1 (PINI) *If* $c_1 \approx_{\ell_A} c_2$, $c_1 \Downarrow c_1'$ *and* $c_2 \Downarrow c_2'$, *then* $c_1' \approx_{\ell_A} c_2'$.

8 Concurrency

Every day, millions of users around the world use concurrent applications, such as email, chat rooms, social networks, e-commerce platforms etc. These services are normally designed concurrently so that multithreaded servers can handle a large number of user requests simultaneously by running multiple instances of the same application. **MAC** features *concurrency* and *synchronization variables*, which shows that the secure-by-construction programming model, that we propose is possible even in a concurrent setting. The extension is non-trivial: the possibility to run simultaneous $MAC\ \ell$ computations provides attackers with new means to bypass security checks.

$$fork :: \ell_L \sqsubseteq \ell_H \Rightarrow MAC\ \ell_H\ () \rightarrow MAC\ \ell_L\ ()$$

Fig. 29: API for concurrency.

8.1 Termination Attack

In Section 7, we have proved that the sequential calculus satisfies *progress-insensitive* non-interference, a security condition that is too weak for concurrent systems. The key observation is the fact that a *non-terminating* sensitive computation at security level ℓ_H embedded in a non-sensitive one at security level ℓ_L via $join$, will suppress public side-effects that follows $join$. Since the embedded computation is sensitive, the suppressed public events may depend on a secret, therefore revealing a bit of secret information. To illustrate this point, we present the attack in Figure 27. We assume that there exists a function $print^{\mathsf{MAC}}$ which prints an integer on a public channel. Observe how function $leak$ may suppress subsequent public events with infinite loops.

```
leak :: Int → Labeled H Secret → MAC L ()
leak n secret = do
  join^MAC (do bits ← unlabel secret
               when (bits !! n) loop
               return True)
  print^MAC n
```

Fig. 27: Termination leak.

Unfortunately concurrency magnifies the bandwidth of the termination covert channel to be linear in the size (of bits) of secrets [51][13], which permits to leak any secret systematically and efficiently. If a thread runs $leak\ 0\ secret$, the code publishes 0 *only if* the first bit of $secret$ is 0; otherwise it loops (see function $loop$) and it does not produce any public effect—see Figure 28. Similarly, a thread running $leak\ 1\ secret$ will leak the second bit of $secret$, while a thread running $leak\ 2\ secret$ will leak the third bit of it and so on. An attacker might then leak the whole secret by spawning as many threads as bits in the secret, i.e., $|secret|$, where each thread runs the one-bit attack described above and n matches the bit being leaked (e.g., $n = 0$ for the first bit, $n = 1$ for the second one, etc.).

```
magnify :: Labeled H Secret → MAC L ()
magnify secret =
  for [0 .. |secret|]
      (λn → fork (leak n secret))
```

Fig. 28: Attack magnification.

[13] Furthermore, the presence of threads introduce the *internal timing covert channel* [50], a channel that gets exploited when, depending on secrets, the timing behavior of threads affect the order of events performed on public-shared resources. Since the same countermeasure closes both the internal timing and termination covert channels, we focus on the latter.

Scheduler state:	ω
Pool Map :	$\Phi ::= (\ell : Label) \to (Pool\ \ell)$
Thread Pool ℓ:	$t_s ::= [\,] \mid t : t_s$
Configuration:	$c ::= \langle \omega, \Sigma, \Phi \rangle$
Sequential Event ℓ:	$s ::= \varnothing \mid fork(t)$
Concurrent Event ℓ:	$e ::= Step \mid Skip \mid Done \mid Fork\ \ell\ n$
Terms:	$t ::= \cdots \mid fork\ t$

(a) Syntax of concurrent calculus.

$$\frac{\Phi(\ell)[n] = t_1 \qquad \langle \Sigma_1, \Phi(\ell) \rangle \longrightarrow_s \langle \Sigma_2, t_2 \rangle \qquad \omega_1 \xrightarrow{(\ell,n,e)} \omega_2}{\langle \omega_1, \Sigma_1, \Phi \rangle \hookrightarrow \langle \omega_2, \Sigma_2, \Phi(\ell)[n] := t_2 \rangle}$$

(b) Scheme rule for concurrent semantics.

Fig. 30: Calculus with concurrency.

(SFORK)
$$\langle \Sigma, fork\ t \rangle \longrightarrow_{fork(t)} \langle \Sigma, return\ () \rangle$$

(BIND$_1$)
$$\frac{\langle \Sigma, t_1 \rangle \longrightarrow_s \langle \Sigma', t_1' \rangle}{\langle \Sigma, t_1 \gg\!= t_2 \rangle \longrightarrow_s \langle \Sigma', t_1' \gg\!= t_2 \rangle}$$

(CATCH$_1$)
$$\frac{\langle \Sigma, t_1 \rangle \longrightarrow_s \langle \Sigma', t_1' \rangle}{\langle \Sigma, catch\ t_1\ t_2 \rangle \longrightarrow_s \langle \Sigma', catch\ t_1'\ t_2 \rangle}$$

Fig. 31: Decorated Sequential Semantics (interesting rules).

To securely support concurrency, **MAC** forces programmers to decouple MAC computations with sensitive labels from those performing observable side-effects—an approach also taken in LIO [51]. As a result, non-terminating computations based on secrets cannot affect the outcome of public events. To achieve this behavior, **MAC** replaces $join$ by $fork$—see Figure 29. Informally, it is secure to spawn sensitive computations (of type $MAC\ \ell_{\mathrm{H}}\ ()$) from non-sensitive ones (of type $MAC\ \ell_{\mathrm{L}}\ ()$) because that decision depends on data at level ℓ_{L}, which is no more sensitive ($\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$). From now on, we call *sensitive* (*non-sensitive*) *threads* those executing MAC computations with a label non-observable (observable) to the attacker. In the two-point lattice, for example, threads running $MAC\ H\ ()$ computations are sensitive, while those running $MAC\ L\ ()$ are observable by the attacker.

8.2 Calculus

Figure 30 extends the calculus from Section 3 with concurrency. It introduces *global configurations* of the form $\langle \omega, \Sigma, \Phi \rangle$ composed by an *abstract* scheduler state ω, a store Σ and a pool map Φ, see Figure 30a. Threads are secure computations of type $MAC\ \ell\ ()$ and are organized in isolated thread pools according to their security label. A pool t_s in the category $Pool\ \ell$ contains threads at security level ℓ and is accessed exclusively through the pool map. We use the same notation for thread pools and pool maps that we have defined to manipulate and extend stores and memories. Term $fork\ t$ spawns thread t and replaces $join$ in the calculus. Without $join$, constructor $Labeled_\chi$ becomes redundant and is also removed. Our calculus includes also synchronization primitives [45], we refer to B for details.

Relation $c_1 \hookrightarrow c_2$ denotes that concurrent configurations c_1 steps to c_2. Figure 30b shows the scheme rule for $c_1 \hookrightarrow c_2$ and highlights the top-level common aspects to all the rules, which we detail later on. The relation $\omega_1 \xrightarrow{(\ell,n,e)} \omega_2$ represents a transition in the scheduler, that depending on the initial state ω_1, decides to run thread identified by (ℓ, n), which is retrieved from the configuration $(\Phi(\ell)[n])$ and executed. Concurrent events inform the scheduler about the evolution of the global configuration, so that it can realize concrete scheduling policies and update its state accordingly. Event $Step$ denotes a single sequential step, event $Fork\ \ell\ n$ informs the scheduler that the current thread has forked a new thread identified by (ℓ, n), event $Done$ is generated when a thread has terminated and event $Stuck$ denotes that a thread is *stuck*, e.g., on a synchronization variable. Note that the scheduled thread determines, with its execution and with sequential event s, triggered by the decorated sequential step, i.e., $\langle \Sigma, t_1 \rangle \longrightarrow_s \langle \Sigma, t_2 \rangle$, which concurrent event e should be passed to the scheduler. Lastly, the final configuration is composed by the updated scheduler state, i.e., ω_2, the updated memory, i.e., Σ_2 and the pool map updated with the executed thread, i.e., $\Phi(\ell)[n] := t_2$.

Decorated Semantics Figure 31 shows the interesting rules of the decorated semantics. Rule [SFORK] is the only rule that explicitly generates event $fork(t)$ and context rules [BIND_1,CATCH_1] propagate the same event generated by the premise step. All the other rules generate the empty event $\varnothing$. Note that, without context rules we could have given the semantics of $fork$ in the concurrent semantics directly.

Concurrent Semantics Figure 32 shows the actual semantics of the concurrent calculus, where each rule generates the appropriate event for the scheduler depending on the state of the thread scheduled and the sequential event. Concurrent rule [STEP] sends event $Step$ to the scheduler, because the thread generates sequential event $\varnothing$, and then updates the store and the thread pool accordingly. Rule [CFORK] generates concurrent event $Fork\ \ell_{\text{H}}\ m$, because the thread generates event $fork(t)$, which is identified by label ℓ_{H} and number m. Observe that

$$\textsc{(Step)}\quad \frac{\omega \xrightarrow{(\ell,n,Step)} \omega' \qquad \langle \Sigma, \Phi(\ell)[n]\rangle \longrightarrow_{\varnothing} \langle \Sigma', t'\rangle}{\langle \omega, \Sigma, \Phi\rangle \hookrightarrow \langle \omega', \Sigma', \Phi(\ell)[n] := t'\rangle}$$

$$\textsc{(CFork)}\quad \frac{|\Phi(\ell_{\mathrm{H}})| = m \qquad \langle \Sigma, \Phi(\ell_{\mathrm{L}})[n]\rangle \longrightarrow_{fork(t)} \langle \Sigma, t'\rangle \qquad \Phi' = \Phi(\ell_{\mathrm{H}})[m \mapsto t] \qquad \omega \xrightarrow{(\ell_{\mathrm{L}},n,Fork\ \ell_{\mathrm{H}}\ m)} \omega'}{\langle \omega, \Sigma, \Phi\rangle \hookrightarrow \langle \omega', \Sigma, \Phi'(\ell_{\mathrm{L}})[n] := t'\rangle}$$

$$\textsc{(Done)}\quad \frac{\omega \xrightarrow{(\ell,n,Done)} \omega' \qquad \Phi(\ell)[n] = v}{\langle \omega, \Sigma, \Phi\rangle \hookrightarrow \langle \omega', \Sigma, \Phi\rangle} \qquad \textsc{(Stuck)}\quad \frac{\omega \xrightarrow{(\ell,n,Stuck)} \omega' \qquad \langle \Sigma, \Phi(\ell)[n]\rangle \not\longrightarrow}{\langle \omega, \Sigma, \Phi\rangle \hookrightarrow \langle \omega', \Sigma, \Phi\rangle}$$

Fig. 32: Concurrent Semantics.

$$\omega ::= (\ell, n) : \omega \mid [\,]$$

$$(\ell, n) : \omega \xrightarrow{(\ell,n,Step)}_{RR} \omega +\!\!+ [(\ell, n)] \qquad (\ell, n) : \omega \xrightarrow{(\ell,n,Skip)}_{RR} \omega +\!\!+ [(\ell, n)]$$

$$(\ell, n) : \omega \xrightarrow{(\ell,n,Done)}_{RR} \omega$$

$$(\ell_{\mathrm{L}}, n_1) : \omega \xrightarrow{(\ell_{\mathrm{L}},n_1,Fork\ \ell_{\mathrm{H}}\ n_2)}_{RR} \omega +\!\!+ [(\ell_{\mathrm{H}}, n_2), (\ell_{\mathrm{L}}, n_1)]$$

Fig. 33: Round-robin Scheduler.

the spawned thread is placed in pool $\Phi(\ell_{\mathrm{H}})$ in the *free* position $m = |\Phi(\ell_{\mathrm{H}})|$—threads are identified with their position in the pool map. The extended pool map Φ' is lastly updated with the parent thread. In rule [Done], $\Phi(\ell)[n] = v$ denotes that the scheduled thread is a value, i.e. the computation has terminated, then the rule sends event $Done$ to the scheduler and leaves the store and pool map unchanged—terminated threads remain in pool map Φ. In rule [Stuck], the notation $\Phi(\ell)[n]\rangle \not\longrightarrow$ denotes that the thread is *stuck*, i.e., it is not a value nor a redex. The scheduler is then informed by event $Stuck$ and the store Σ and pool map Φ are left unchanged.

8.3 Round-robin Scheduler

Figure 33 shows a round-robin scheduler with time-slot of one step, as an example of a scheduler that can be securely employed in our concurrent calculus. The state of the scheduler is a queue that tracks the identifiers of *alive* threads in the global configuration. A thread is uniquely identified by a pair consisting of a label, i.e., its security level, and a thread identifier, i.e., its position in the corresponding thread pool. The queue is concretely represented by a list of thread

$$
\begin{aligned}
&\mathit{fmap} :: (a \to b) \to \mathit{Labeled}\ \ell\ a \to \mathit{Labeled}\ \ell\ b\\
&(\langle * \rangle) :: \mathit{Labeled}\ \ell\ (a \to b) \to \mathit{Labeled}\ \ell\ a \to \mathit{Labeled}\ \ell\ b\\
&\mathit{relabel} :: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow \mathit{Labeled}\ \ell_{\mathrm{L}}\ a \to \mathit{Labeled}\ \ell_{\mathrm{H}}\ a
\end{aligned}
$$

Fig. 34: API for flexible manipulation of labeled values.

identifiers, whose first element identifies the next thread in the schedule. After executing one step (event Step), the current thread has used up its time slot and is enqueued. If the scheduled thread cannot execute (event Skip), it is skipped and enqueued as well. When the current thread has terminated (event Done), the thread is not alive anymore and hence removed from the queue. Message $(\ell_{\mathrm{L}}, n_1, \mathit{Fork}\ \ell_{\mathrm{H}}\ n_2)$ informs the scheduler that thread (ℓ_{L}, n_1) has spawned thread (ℓ_{H}, n_2), which is then enqueued with the current thread.

9 Flexible Labeled Values

In this section we extend the API of labeled values with new operations that allow to perform *pure* (side-effect free) computations with labeled data—see Figure 34. Observe that these primitives operate on labeled data without using label and $\mathit{unlabel}$, thus avoiding incurring in the *no read-up* and *no write-down* restrictions and *irrespectively of their security level*. For instance, a non-sensitive computation at security level ℓ_{L} can operate on sensitive labeled data at security level ℓ_{H} using fmap, without forking threads in a concurrent setting, thus introducing *flexibility* when data is processed by pure functions. We remark that, depending on the evaluation strategy of the host language (i.e. call-by-value or call-by-name), a naive implementation of these primitives is vulnerable to leaks via non-termination—we elaborate on this point later, in Section 9.3. Section 9.1 gives a broad description of these primitives, Section 9.2 shows their flexibility with an example, and Section 9.3 formalizes them in our calculus.

9.1 Functors and Relabeling

Intuitively, a functor is a container-like data structure which provides a method called fmap that applies (maps) a function over its contents, while preserving its structure. Lists are the most canonical example of a functor data-structure. In this case, fmap corresponds to the function map, which applies a function to each element of a list, e.g. $\mathit{fmap}\ (+1)\ [1,2,3] \equiv [2,3,4]$. A functor structure for labeled values allows to manipulate sensitive data without the need to explicitly extract it—see Figure 34. For instance, $\mathit{fmap}\ (+1)\ d$, where $d :: \mathit{Labeled}\ H\ \mathit{Int}$ stores the number 42, produces the number 43 as a sensitive labeled value.

To aggregate data at possibly different security levels **MAC** provides primitives $\mathit{relabel}$ and $(\langle * \rangle)$. Primitive $\mathit{relabel}$ upgrades the security level of a labeled value, which is useful to "lift" data to an upper bound of all the data involved

in a computation prior to combining them. Operator $(\langle * \rangle)$ supports function application within a labeled value, i.e. it allows to feed functions wrapped in a labeled value ($Labeled\ \ell\ (a \to b)$ with arguments also wrapped ($Labeled\ \ell\ a$), where aggregated results get wrapped as well ($Labeled\ \ell\ b$).

Discussion In functional programming, operator $(\langle * \rangle)$ is part of the *applicative functors* [32] interface, which in combination with $fmap$, is used to map functions over functors. Note that if labeled values were full-fledged applicative functors, our API would also include the primitive $pure :: a \to Labeled\ \ell\ a$. This primitive brings arbitrary values into labeled values, which might break the security principles enforced by **MAC**. Instead of $pure$, **MAC** centralizes the creation of labeled values in the primitive $label$. Observe that, by using $pure$, a programmer could write a computation $m :: MAC\ H\ (Labeled\ L\ a)$ where the *created* labeled information is sensitive rather than public. We argue that this situation ignores the no-write down principle, which might bring confusion among users of the library. More importantly, freely creating labeled values is not compatible with the security notion of *cleareance*, where secure computations have an upper bound on the kind of sensitive data they can observe and generate. This notion becomes useful to address certain covert channels [60] as well as poison-pill attacks [23]. While **MAC** does not yet currently support cleareance, it is an interesting direction for future work.

9.2 Examples

The functor API of labeled values, i.e., $fmap$, is a handy tool that functional programmers use to code simple concise functions elegantly. In Figure 35, the 1-line function $isShort$ checks whether the password is weak because it is too short. In the anonymous function, pwd is the *unlabeled* password, and the expression $|pwd| \leqslant 5$ checks if the password contains less than 5 characters. Observe that what the function computes is an attribute of the password, therefore it should be considered *sensitive*. The API of $fmap$ ensures that by preserving the label of the labeled argument, i.e., $Labeled\ H\ String$, in the resulting labeled value, i.e., $Labeled\ H\ Bool$. Compare the program in Figure 35 with the homonym program in Figure 36 written without $fmap$, but using $join$ instead. Firstly, note that the imperative program has a different signature: it must necessarily involve MAC computation in order to perform $unlabel$. Since the password $lpwd$ is sensitive, i.e., it has type $Labeled\ H\ String$, only a sensitive computation can unlabel it. Then, the program employs $join$ to convert the sensitive computation into a sensitive labeled value, which then gets wrapped in a non-sensitive computation, i.e., $MAC\ L\ (Labeled\ H\ Bool)$. In a concurrent

$$isShort :: Labeled\ H\ String \to Labeled\ H\ Bool$$
$$isShort = fmap\ (\lambda pwd \to |pwd| \leqslant 5)$$

Fig. 35: A *pure* computation on a password.

```
isShort :: Labeled H String → MAC L (Labeled H Bool)
isShort lpwd = do
  join (do
    pwd ← unlabel lpwd
    return (|pwd| ⩽ 5))
```

Fig. 36: *join* involves *MAC* also for *pure* computations.

```
isWeak :: Labeled L' (String → Bool).
```

(a) Third party API.

```
f :: Labeled H String → Labeled H Bool
f pwd = relabel isWeak ⟨∗⟩ pwd
```

(b) Embedding mistrusted code.

(c) 3-Points Lattice.

Fig. 37: Combining heterogeneously labeled data.

setting, where *join* is not available, the whole program must be completely restructured, because threads have type $MAC\ H\ ()$ and may not return any other result in a non-sensitive computation.

The strength of a password is often estimated by combining several syntactic aspects, such as its length or the presence and number of special characters and digits. Suppose now that some third-party API function provides such syntactic checks in the form of a **MAC** labeled *pure* function *isWeak*, see Figure 37a. The type system guarantees that the function is secure, because it has type $String \to Bool$, however the third party has labeled it with its own label L', because it wants to strictly control who can use it and under what terms. In order to keep the code of our password-checker isolated from that of the third party, while still providing functionality to the user, we incorporate the new label L' into the system and modify the lattice as shown in Figure 37c. The lattice reflects our mistrust over the third-party code by making L and L' incomparable elements. Thanks to **MAC**'s security guarantees, we can safely run third-party mistrusted code, i.e., *isWeak*, with the user's secret password, as shown in Figure 37b. In particular *relabel* upgrades the function to *isWeak* to security level H (observe that $L' \sqsubseteq H$ in the lattice), and then applies the function to the password (*pwd*) using the applicative functor operator, i.e., $\langle * \rangle$, which protects the final result with label H.

9.3 Calculus

In Figure 38, we extend our calculus with the primitives for flexible manipulation of labeled values, discussed in the previous section. Firstly we add terms

Terms: $t ::= \cdots \mid \mathit{fmap}\ t_1\ t_2 \mid t_1\ \langle * \rangle\ t_2 \mid \mathit{relabel}\ t$

$$\text{(FMAP)}\quad \mathit{fmap}\ t_1\ t_2 \rightsquigarrow (\mathit{Labeled}\ t_1)\ \langle * \rangle\ t_2$$

$$(\langle * \rangle_1)\quad \frac{t_1 \rightsquigarrow t_1'}{t_1\ \langle * \rangle\ t_2 \rightsquigarrow t_1\ \langle * \rangle\ t_2}$$

$$(\langle * \rangle_2)\quad \frac{t_2 \rightsquigarrow t_2'}{(\mathit{Labeled}\ t_1)\ \langle * \rangle\ t_2 \rightsquigarrow (\mathit{Labeled}\ t_1)\ \langle * \rangle\ t_2'}$$

$$(\langle * \rangle_3)\quad (\mathit{Labeled}\ t_1)\ \langle * \rangle\ (\mathit{Labeled}\ t_2) \rightsquigarrow \mathit{Labeled}\ (t_1\ t_2)$$

$$\text{(RELABEL}_1\text{)}\quad \frac{t \rightsquigarrow t'}{\mathit{relabel}\ t \rightsquigarrow \mathit{relabel}\ t'}$$

$$\text{(RELABEL}_2\text{)}\quad \mathit{relabel}\ (\mathit{Labeled}\ t) \rightsquigarrow \mathit{Labeled}\ t$$

Fig. 38: Calculus with flexible labeled values.

```
leak :: Int → Labeled H Secret → MAC L ()
leak n secret = let result = fmap loopOn secret in publish n
  where loopOn = λbits → if (bits !! n) then loop else bits
```

Fig. 39: Function *leak* attempts to leak the n-th bit of *secret*.

fmap $t_1\ t_2$, $t_1\ \langle * \rangle\ t_2$ and *relabel* t, whose types correspond to those given in Figure 34. Primitive *fmap* is implemented in terms of $\langle * \rangle$ in rule [FMAP], where the function is simply lifted to labeled value (every applicative functor is also a functor). Rules [$\langle * \rangle_1$, $\langle * \rangle_2$] evaluate the first and second argument to a labeled value respectively, which are then combined by rule [$\langle * \rangle_3$], which applies the function to the argument and wraps the result in a labeled value. Rule [RELABEL$_1$] evaluates its argument to weak-head normal form and rule [RELABEL$_2$] upgrades its label. Observe that since labels are types *relabel* leaves the content of *Labeled* unchanged. We remark that these primitives do not affect the security guarantees of the sequential calculus, where their semantics solely must be adjusted to handle exceptional values, i.e., constructor $\mathit{Labeled}_\chi$, see A for more details.

Discussion The API of flexible labeled values shown in Figure 34 might seem insecure at first sight. In particular, it might be counter-intuitive that a *public* computation might be able to manipulate a *secret* with an *arbitrary* function without introducing potential leaks. Figure 39 shows an attack that attempts to leak via *non-termination* the n-th bit of a secret. Function *leak* applies function *loopOn* on the secret using *fmap* and then performs a *non-sensitive* side-effect,

i.e., $publish\ n$, which outputs the number n on a public channel. Interestingly, depending on the evaluation strategy of the language, the attack might succeed. Specifically, under a *call-by-value* evaluation strategy, function $loopOn$ passed to $fmap$ is eagerly applied to the secret, which might introduce a loop depending on the value of the n-th bit of the secret suppressing the subsequent *public* action $publish\ n$. Under a *call-by-name* evaluation strategy, however, function $loopOn$ does not get immediately evaluated since $result$ is not needed for computing $publish\ n$. Therefore, $publish\ n$ gets executed independently of the value of the secret, i.e., no termination leaks are introduced. Instead, $loopOn$ gets evaluated when and only if $result$ is *unlabeled* and its content inspected—something that is possible only in a computation at security level at least as sensitive as H because of the *no-read up* policy, where it is secure to do so. We remark that it is possible to close this termination channel under a call-by-value semantics by defining $Labeled$ with an explicit suspension, e.g. $\mathbf{data}\ Labeled\ \ell\ a = Labeled\ (() \to a)$, and corresponding forcing operation, so that $fmap$ behaves lazily as desired.

10 Soundness of Concurrent Calculus

The concurrent calculus that we have presented satisfies *progress sensitive non-interference*. Section 10.1 extends the erasure function for the concurrent calculus and for flexible labeled values. To obtain a parametric proof of non-interference, we assume certain properties about the scheduler. Specifically, our proof is valid for deterministic schedulers which fulfill progress and non-interference themselves, i.e., schedulers cannot leverage sensitive information in threads to determine what to schedule next. Section 10.2 formalizes the requirements for such *suitable* schedulers. In Section 10.3 we prove a scheduler-parametric progress-sensitive non-interference theorem for our calculus and we constructively obtain a proof that **MAC** is secure with a round-robin scheduler by simply instantiating our main theorem.

10.1 Erasure Function

Figure 40 shows the erasure function for the concurrent calculus. A concurrent configuration $\langle \omega, \Sigma, \Phi \rangle$ is erased by erasing each component, where the erasure of the scheduler state ω is scheduler specific (Figure 40a). Similarly to store Σ, pool map Φ is erased pointwise, i.e., $\varepsilon_{\ell_A}(\Phi) = \lambda\ell.\varepsilon_{\ell_A}(\Phi(\ell))$, and sensitive thread pools are rewritten to $\bullet$ and erased homomorphically otherwise, just like memories (see Figure 40b). Observe that primitive $fork$ performs a *write* effect because it adds a new thread to a thread pool, therefore we employ our *two-steps erasure* technique, just like we did for memory primitives. Specifically, the erasure function replaces $fork$ with $fork_\bullet$ whenever it spawns a *sensitive* thread, which would write to a *sensitive* thread pool ($\ell_H \not\sqsubseteq \ell_A$), see Figure 40c. Sequential fork-events are erased similarly in order to ensure simulation,

$$\varepsilon_{\ell_A}(\langle \omega, \Sigma, \Phi \rangle) = \langle \varepsilon_{\ell_A}(\omega), \varepsilon_{\ell_A}(\Sigma), \varepsilon_{\ell_A}(\Phi) \rangle$$

(a) Erasure for concurrent configuration.

$$\varepsilon_{\ell_A}(t_s :: Pool\ \ell_{\mathrm{H}}) = \begin{cases} \bullet & \text{if } \ell_{\mathrm{H}} \not\sqsubseteq \ell_A \\ map\ \varepsilon_{\ell_A}\ t_s & \text{otherwise} \end{cases}$$

(b) Erasure for thread pool.

$$\varepsilon_{\ell_A}(fork\ t) = \begin{cases} fork_\bullet\ \varepsilon_{\ell_A}(t :: MAC\ \ell_{\mathrm{H}}\ ()) & \text{if } \ell_{\mathrm{H}} \not\sqsubseteq \ell_A \\ fork\ \varepsilon_{\ell_A}(t) & \text{otherwise} \end{cases}$$

(c) Erasure of fork.

$$\varepsilon_{\ell_A}(fork(t :: MAC\ \ell_{\mathrm{H}}\ ())) = \begin{cases} fork_\bullet(\varepsilon_{\ell_A}(t)) & \text{if } \ell_{\mathrm{H}} \not\sqsubseteq \ell_A \\ fork(\varepsilon_{\ell_A}(t)) & \text{otherwise} \end{cases}$$

(d) Erasure for *sequential* fork event.

$$\varepsilon_{\ell_A}(Fork\ \ell_{\mathrm{H}}\ n) = \begin{cases} Step & \text{if } \ell_{\mathrm{H}} \not\sqsubseteq \ell_A \\ Fork\ \ell_{\mathrm{H}}\ n & \text{otherwise} \end{cases}$$

(e) Erasure for *concurrent* fork event.

Fig. 40: Erasure function for concurrent calculus.

i.e., the erasure function rewrites $fork(t)$ to $fork_\bullet(\varepsilon_{\ell_A}(t))$ when t is sensitive—see Figure 40d. Sequential event $\varnothing$ is not affected by the erasure function. The erasure function masks spawning sensitive threads from the scheduler as well by erasing concurrent events accordingly (Figure 40e). In this case it rewrites event $Fork\ \ell_{\mathrm{H}}\ n$ to $Step$ whenever $\ell_{\mathrm{H}} \not\sqsubseteq \ell_A$—the other events are not affected by the erasure function. In the sequential calculus $fork_\bullet$ is reduced by rule [SFork$_\bullet$], defined in Figure 41, which simulates the decorated reduction of $fork$. A new concurrent rule [CFork$_\bullet$] detects the sequential event $fork_\bullet(t)$ and *skips* spawning the thread, i.e., it does not insert it in the thread pool, and sends concurrent event $Step$ to the scheduler, therefore simulating precisely rule [CFork] when a non-sensitive thread of type $MAC\ \ell_{\mathrm{L}}\ ()$ forks a sensitive thread $MAC\ \ell_{\mathrm{H}}\ ()$.

Seq. Effect: $s ::= \cdots \mid \mathit{fork}_\bullet(t)$
Terms: $t ::= \cdots \mid \mathit{fork}_\bullet\ t$

$$\text{(SFork}_\bullet\text{)}\quad \langle \Sigma, \mathit{fork}_\bullet\ t\rangle \longrightarrow_{\mathit{fork}_\bullet(t)} \langle \Sigma, \mathit{return}\ ()\rangle$$

$$\text{(CFork}_\bullet\text{)}\quad \frac{\omega \xrightarrow{(\ell_L, n, \mathit{Step})} \omega' \qquad \langle \Sigma, \Phi(\ell_L)[n]\rangle \longrightarrow_{\mathit{fork}_\bullet(t)} \langle \Sigma, t'\rangle}{\langle \omega, \Sigma, \Phi\rangle \hookrightarrow \langle \omega', \Sigma, \Phi(\ell_L)[n] := t'\rangle}$$

Fig. 41: Sequential and concurrent semantics of $\mathit{fork}_\bullet$.

$$\varepsilon_{\ell_A}(\mathit{fmap}\ t_1\ t_2 :: \mathit{Labeled}\ \ell_H\ \tau) = \begin{cases} \mathit{fmap}_\bullet\ \varepsilon_{\ell_A}(t_1)\ \varepsilon_{\ell_A}(t_2) & \text{if } \ell_H \not\sqsubseteq \ell_A \\ \mathit{fmap}\ \varepsilon_{\ell_A}(t_1)\ \varepsilon_{\ell_A}(t_2) & \text{otherwise} \end{cases}$$

$$\varepsilon_{\ell_A}(t_1\ \langle * \rangle\ t_2 :: \mathit{Labeled}\ \ell_H\ \tau) = \begin{cases} \varepsilon_{\ell_A}(t_1)\ \langle * \rangle_\bullet\ \varepsilon_{\ell_A}(t_2) & \text{if } \ell_H \not\sqsubseteq \ell_A \\ \varepsilon_{\ell_A}(t_1)\ \langle * \rangle\ \varepsilon_{\ell_A}(t_2) & \text{otherwise} \end{cases}$$

$$\varepsilon_{\ell_A}(\mathit{relabel}\ t :: \mathit{Labeled}\ \ell_H\ \tau) = \begin{cases} \mathit{relabel}_\bullet\ \varepsilon_{\ell_A}(t) & \text{if } \ell_H \not\sqsubseteq \ell_A \\ \mathit{relabel}\ \varepsilon_{\ell_A}(t) & \text{otherwise} \end{cases}$$

Fig. 42: Erasure of flexible labeled values.

Context-Aware Erasure Function A common challenge when reasoning about security of IFC libraries is that the sensitivity of a term may depend on context where they are used. Consider for instance the primitive *relabel*, i.e., which upgrades the security level of a labeled term. A public number, e.g., $\mathit{Labeled}\ 42 :: \mathit{Labeled}\ L\ \mathit{Int}$, should be treated as secret when in the context of relabeling, e.g., $\mathit{relabel}\ (\mathit{Labeled}\ 42) :: \mathit{Labeled}\ H\ \mathit{Int}$. Doing otherwise, i.e., erasing the term homomorphically, breaks simulation because sensitive data produced by *relabel* remains *after* erasure. For example $\mathit{relabel}\ (\mathit{Labeled}\ 42)$ is homomorphically erased to $\mathit{relabel}\ \varepsilon_L(\mathit{Labeled}\ 42 :: \mathit{Labeled}\ L\ \mathit{Int})$ which reduces on the orange path to $\mathit{Labeled}\ 42 \not\equiv \mathit{Labeled}\ \bullet$, obtained on the cyan path by $\varepsilon_L(\mathit{Labeled}\ 42 :: \mathit{Labeled}\ H\ \mathit{Int})$, thus breaking commutativity of rule [RELABEL$_2$].

Then, one might be tempted to stretch the definition of the erasure function to accommodate for the problematic cases shown above. Unfortunately, this approach does not work, because it will necessary break simulation for other cases. We support this statement by showing that this is the case for any arbitrary erasure function that is suitable for $\mathit{relabel}\ t :: \mathit{Labeled}\ H\ \tau$, where $t :: \mathit{Labeled}\ L\ \tau$. Observe that we need a different behavior for our erasure function for public labeled values when embedded in *relabel*, which we will capture in a different *auxiliary* erasure function ε'_L. Suppose we defined $\varepsilon_L(\mathit{relabel}\ t :: \mathit{Labeled}\ H\ \tau) =$

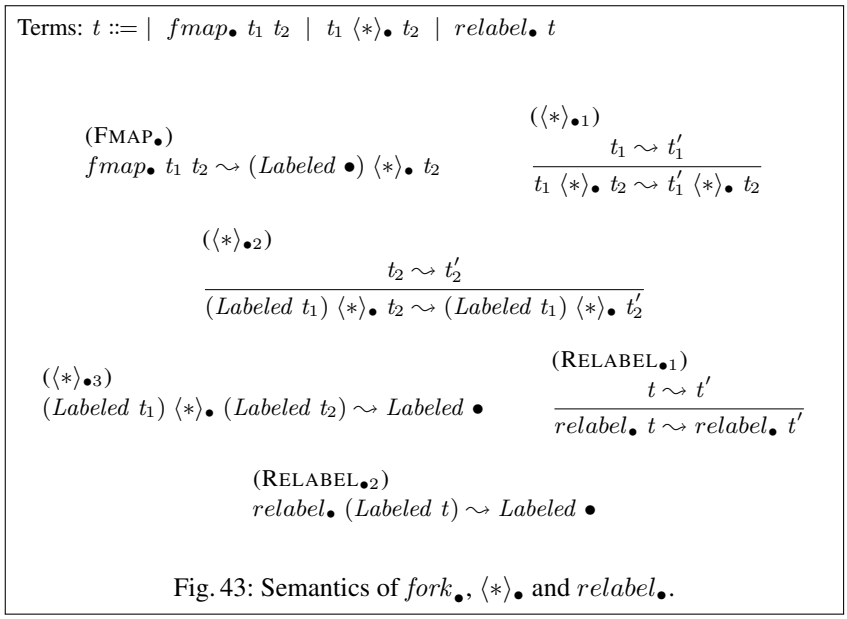

Fig. 43: Semantics of $fork_\bullet$, $\langle * \rangle_\bullet$ and $relabel_\bullet$.

$relabel\ \varepsilon'_L(t :: Labeled\ L\ \tau)$, for some suitable ε'_L that exhibits the desired behavior, e.g., $\varepsilon'_L(Labeled\ 42 :: Labeled\ L\ Int) = Labeled\ \bullet$. Alas, while this definition respects simulation for step [RELABEL$_2$], introducing a *different* erasure function in a *context-sensitive* way is fatal for simulation of beta reductions. More precisely, the original erasure function is *no longer homomorphic over substitution*, i.e., $\varepsilon_{\ell_A}([x\ /\ t_1]\ t_2) \not\equiv [x\ /\ \varepsilon_{\ell_A}(t_1)]\ \varepsilon_{\ell_A}(t_2)$—an essential property of the erasure function [20, 30, 43, 53, 54], without which step [BETA] does not commute anymore. Essentially, function ε_{ℓ_A} is oblivious to the context in which some term will be substituted inside the body of a function, thus breaking simulation. As a counterexample, consider term $(\lambda x.relabel\ x)\ t$, which is erased *homomorphically*, that is $(\lambda x.relabel\ x)\ \varepsilon_L(t)$, and then beta-reduces on the orange path to $relabel\ \varepsilon_L(t)$. On the cyan path term $(\lambda x.relabel\ x)\ t$ beta-reduces to $relabel\ t$ and then is *context-sensitively* erased to $relabel\ \varepsilon'_L(t)$. Observe that $relabel\ \varepsilon_L(t) \not\equiv relabel\ \varepsilon'_L(t)$ in general because ε'_L captures a different behavior than that exposed by ε_L, specifically for public labeled values, e.g., when $t = Labeled\ 42 :: Labeled\ L\ Int$. To the best of our knowledge, this work is the first to point out this issue. Furthermore, we identify problematic cases in the formalization of previous work on **LIO** [51,54] which lead to breaking the one-step simulation—see details in Appendix A of [58]. By using the *two-step erasure* technique, we can craft a sound erasure function that is *homomorphic over substitution* and is *context-aware*. The erasure function replaces $relabel$ with $relabel_\bullet$, rule [RELABEL$_{\bullet 1}$] simulates rule [RELABEL$_1$] and rule [RELABEL$_{\bullet 2}$] performs *context-sensitive* erasure by producing $Labeled\ \bullet$, see Figure 43. Even though the actual erasure is done by rule [RELABEL$_{\bullet 2}$], we still have to erase the *arugment* of $relabel$, or else the erasure function would not

be *homomorphic over substitution*. Simulation of the context rule [RELABEL$_\bullet$] follows then by inductive hypothesis.

Primitive $\langle * \rangle$ raises a similar problem. To illustrate this point, consider the term $(Labeled\ t_1)\ \langle * \rangle\ (Labeled\ t_2)$ of type $Labeled\ H\ Int$, which reduces to $Labeled\ (t_1\ t_2)$ according to rule $[\langle * \rangle_3]$.[14] Following the orange path we get $\varepsilon_L(Labeled\ t_1)\ \langle * \rangle\ \varepsilon_L(Labeled\ t_2)$, by applying the erasure function *homomorphically*, i.e., $(Labeled\ \bullet)\ \langle * \rangle\ (Labeled\ \bullet)$ which reduces to $Labeled\ (\bullet\ \bullet) \not\equiv Labeled\ \bullet$, obtained instead by first reducing the term and then erasing following the cyan path. Observe that rule $[\langle * \rangle_3]$ produces a function application within a $Labeled$ constructor, therefore it cannot possibly commute for sensitive labeled values, which always rewrite the content of a labeled value to $\bullet$. We then prove simulation using *two-steps erasure* again. Specifically, the erasure function replaces $\langle * \rangle$ with $\langle * \rangle_\bullet$, see Figure 42, and erasure is then performed by means of its semantics rules $[\langle * \rangle_{\bullet 1}, \langle * \rangle_{\bullet 2}, \langle * \rangle_{\bullet 3}]$, listed in Figure 43, which simulate rules $[\langle * \rangle_1, \langle * \rangle_2, \langle * \rangle_3]$ respectively. Observe that $[\langle * \rangle_{\bullet 3}]$ ignores the content of the labeled values and simply yields $Labeled\ \bullet$ to enforce the simulation property. Since $fmap$ is defined in terms of $\langle * \rangle$, we likewise replace it with new node $fmap_\bullet$ and give its semantics in terms of $\langle * \rangle_\bullet$, see rule [FMAP$_\bullet$]. We remark that $fmap_\bullet$, $\langle * \rangle_\bullet$ and $relabel_\bullet$ and their semantics rules are introduced in the calculus as a device to prove *simulation* (they only occur in *erased* programs), they are not part of the surface syntax nor **MAC**.

10.2 Scheduler Requirements

We take advantage of the level of abstraction of our concurrent semantics and make our proof parametric in the scheduler state and its semantics. For this reason, we study what are the sufficient requirements of a scheduler to guarantee PSNI in our calculus. We evaluate our characterization of schedulers by formalizing a round-robin scheduler, similar to that used by GHC's run-time system [31], and show that it satisfies the requirements listed in this section.

Our proof is valid for schedulers which are (i) deterministic, (ii) fulfill a restricted variant of single-step simulation from Figure 22, i.e., schedulers may not leverage on sensitive information to determine what observable thread should be scheduled next, (iii) do not leak secret information when scheduling a sensitive threads and (iv) guarantee progress of observable threads, i.e., execution of observable threads cannot be indefinitely deferred by sensitive ones. In the following, we use labels ℓ_L and ℓ_H to denote a security level that is visible resp. invisible to the attacker, i.e., $\ell_\mathrm{L} \sqsubseteq \ell_A$ and $\ell_\mathrm{H} \not\sqsubseteq \ell_A$. Furthermore, we call a

[14] In our conference version [57], rule $[\langle * \rangle_3]$ raises a problem also for public labeled values, because the erasure function is not homomorphic over function application, in particular $\varepsilon_L(t_1\ \ t_2\ ::\ MAC\ \ H\ \ \tau) = \bullet \not\equiv \varepsilon_L(t_1)\ \ \varepsilon_L(t_2)$. To avoid this problem, we replace function application with substitution, i.e. $(Labeled\ (\lambda x.t_1))\ \langle * \rangle\ (Labeled\ t_2) \rightsquigarrow Labeled\ (t_1\ [x\ /\ t_2])$, at the price of having a non-standard stricter semantics for $\langle * \rangle$. The erasure function presented here is homomorphic over function application and the semantics of $\langle * \rangle$ is standard.

scheduler step that runs a *non-sensitive* thread, e.g., $\omega_1 \xrightarrow{(\ell_L,n,e)} \omega_2$, *public* or *low* step. Similarly we refer to a run of a *sensitive* thread, e.g., $\omega_1 \xrightarrow{(\ell_H,n,e)} \omega_2$, as *secret* or *high* step. We formally characterize schedulers for which our security guarantees apply.

Requirement 1

i) Determinancy*: if* $\omega_1 \xrightarrow{(\ell,n,e)} \omega_2$ *and* $\omega_1 \xrightarrow{(\ell',n',e)} \omega_2'$*, then* $\ell \equiv \ell'$*,* $n \equiv n'$ *and* $\omega_2 \equiv \omega_2'$*.*

ii) Restricted Simulation*: if* $\omega_1 \xrightarrow{(\ell_L,n,e)} \omega_2$ *then* $\varepsilon_{\ell_A}(\omega_1) \xrightarrow{(\ell_L,n,\varepsilon_{\ell_A}(e))} \varepsilon_{\ell_A}(\omega_2)$*.*

iii) No Observable Effect*: if* $\omega_1 \xrightarrow{(\ell_H,n,e)} \omega_2$ *then* $\omega_1 \approx_{\ell_A} \omega_2$*.*

iv) Progress*: If* $\omega_1 \xrightarrow{(\ell_L,n,e)} \omega_1'$ *and* $\omega_1 \approx_{\ell_A} \omega_2$ *then* ω_2 *will schedule thread* (ℓ_L, n) *eventually.*

Observe that determinancy of the scheduler is essential for determinancy of the concurrent semantics—after all, the scheduler state is part of the concurrent configuration. As it is expected from the concurrent calculus, we assume that the abstract scheduler satisfies a variant of the single-step simulation restricted to low steps[15]. "No observable effect", i.e., Requirement (iii), ensures that high steps do not leak sensitive information in the scheduler state—we extend ℓ_A-equivalence to scheduler states, that is $\omega_1 \approx_{\ell_A} \omega_2$ if and only if $\varepsilon_{\ell_A}(\omega_1) \equiv \varepsilon_{\ell_A}(\omega_2)$. Observe that the erasure function of the scheduler state is scheduler specific, and thus we leave it unspecified. Requirement (iv) avoids revealing sensitive data by observing progress of non-sensitive threads via public events. Intuitively, a concurrent program might reveal sensitive information by forcing a sensitive thread to induce starvation of a non-sensitive thread, thus potentially suppressing subsequent public events. The formal definition of *eventually* is technically interesting. Since we aim to a modular proof, the scheduler is considered in isolation from the pool thread, therefore the scheduler cannot predict how long high threads are going to run. We overcome this technicality by indexing the ℓ_A-equivalence relation between scheduler states. We then use the indexes to encode a single-step progress principle, i.e., Requirement 2 (explained below), and to exclude starvation, by making the progress principle a well-founded induction principle, i.e., Requirement 3 (explained below).

Definition 2 (Annotated Scheduler ℓ_A-equivalence). *Two states are* (i,j)*-*ℓ_A*-equivalent, written* $\omega_1 \approx_{\ell_A}^{(i,j)} \omega_2$ *if and only if* $\omega_1 \approx_{\ell_A} \omega_2$ *and* i *and* j *are upper bounds over the number of sensitive threads scheduled before the next common non-sensitive thread in* ω_1 *and* ω_2*, respectively.*

[15] Different to our conference version [58], we do not require lock-step simulation for high scheduler steps, i.e., when $\ell_H \not\sqsubseteq \ell_A$, for which is instead sufficient to show indistinguishability. This choice gives the same security guarantees and simplifies the formalization of a non-interfering scheduler.

The relation $\omega_1 \approx^{(i,j)}_{\ell_A} \omega_2$ captures an alignment measure of two ℓ_A-equivalent states and how close they are to schedule the next common non-sensitive thread. Informally, our non-interference proof excludes *starvation* of observable threads, that can leak information to the attacker, by ensuring that two ℓ_A-equivalent schedulers will eventually align and schedule the same non-sensitive thread, regardless of how the global configuration evolves. Specifically, our calculus requires that the indexes in $\omega_1 \approx^{(i,j)}_{\ell_A} \omega_2$ strictly decreases after every reduction. We capture the interplay between the (i,j)-ℓ_A-equivalent relationship and the evolution of schedulers by establishing unwinding-like conditions [15].

Requirement 2 (Progress) *Given* $\omega_1 \xrightarrow{(\ell_L,n,e)} \omega_1'$, *and* $\omega_1 \approx^{(i,j)}_{\ell_A} \omega_2$ *then:*

- *If* $j = 0$, *then there exists state* ω_2' *such that* $\omega_2 \xrightarrow{(\ell_L,n,e')} \omega_2'$, *for any event* e' *such that* $e \approx_{\ell_A} e'$.
- *If* $j > 0$, *then there exists* ℓ_H, n' *such that* $\forall\ e' \Rightarrow \exists\ \omega_2' : \omega_2 \xrightarrow{(\ell_H,n',e')} \omega_2'$

If a scheduler runs a public thread, then a (i,j)-ℓ_A-equivalent scheduler runs *at most* j secret threads before the same public thread. In particular, if $j = 0$ then the two schedules align and the threads generate ℓ_A-equivalent events[16], otherwise a secret thread is run ($j > 0$). In the second case the scheduler cannot predict what event will be triggered by thread (ℓ_H, m), therefore, as a conservative approximation, the step may involve *any* possible event e', which in addition determines the final state ω_2'. Conceptually, by repeatedly applying Requirement 2, Requirement (iii) and by transitivity of $\approx_{\ell_A}$, we could build the chain of high steps that precedes the common low-step. However, this recursion scheme is not well founded in general, because it does not exclude starvation, e.g., for *non-preemptive* schedulers [20]. The following requirement guarantees instead that such chain is finite, i.e., that public threads cannot starve indefinitely due to secret threads.

Requirement 3 (No Starvation) *Given* $\omega_1 \xrightarrow{(\ell_L,n,e)} \omega_1'$, $\omega_2 \xrightarrow{(\ell_H,n',e')} \omega_2'$, *such that* $\omega_1 \approx^{(i,j)}_{\ell_A} \omega_2$, *then there exist* j' *such that* $j' < j$ *and* $\omega_1' \approx^{(i,j')}_{\ell_A} \omega_2'$.

Intuitively, the combination of Requirement 2 and 3 ensures that the two schedules will align *eventually*[17]. Figure 44 highlights this intuition. The colored scheduler steps denote running either a secret (red for ℓ_H) or a public (blue for

[16] In our conference version [58], the requirement expects the same event e in the other step, which is too strict. Intuitively an event $Fork\ \ell_H\ n$ contains a bit of secret information, namely the number n of secret threads, which could differ in the other run. The relation $e \approx_{\ell_A} e'$, defined as $\varepsilon_{\ell_A}(e) \equiv \varepsilon_{\ell_A}(e')$ captures this scenario: $Fork\ \ell_H\ n \approx_{\ell_A} Fork\ \ell_H\ n'$, because $\varepsilon_{\ell_A}(Fork\ \ell_H\ n) \equiv \varepsilon_{\ell_A}(Fork\ \ell_H\ n') \equiv Step$.

[17] In our conference version [58], Requirements 2 and 3 are combined, but technically we need to split these two requirements. Progress of the concurrent configuration requires the former, while the latter ensures a well-founded inductive principle.

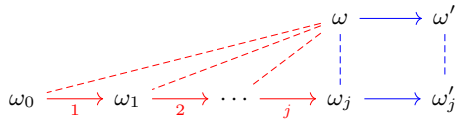

Fig. 44: Two (i, j)-ℓ_A-equivalent schedulers align in at most j steps.

$$[\,] \approx_{\ell_A}^{(0,0)} [\,] \qquad \frac{\omega_1 \approx_{\ell_A}^{(i,j)} \omega_2}{(\ell_{\mathrm{L}}, n) : \omega_1 \approx_{\ell_A}^{(0,0)} (\ell_{\mathrm{L}}, n) : \omega_2} \qquad \frac{\omega_1 \approx_{\ell_A}^{(i,j)} \omega_2}{(\ell_{\mathrm{H}}, n) : \omega_1 \approx_{\ell_A}^{(i+1,j)} \omega_2}$$

$$\frac{\omega_1 \approx_{\ell_A}^{(i,j)} \omega_2}{\omega_1 \approx_{\ell_A}^{(i,j+1)} (\ell_{\mathrm{H}}, n) : \omega_2}$$

Fig. 45: Annotated ℓ_A-equivalence (Round-robin).

ℓ_{L}) thread respectively and the dashed line links ℓ_A-equivalent states. Given two initial scheduler states such that $\omega_0 \approx_{\ell_A}^{(i,j)} \omega$, where ω runs a public thread, *progress*, i.e., Requirement 2, guarantees that ω_0 steps to ω_1, running a secret thread. By Requirement 3, it follows that $\omega_1 \approx_{\ell_A}^{(i,j')} \omega$, where $j' < j$. After repeating this mechanism at most j times (j is strictly smaller after each step), we obtain $\omega_j \approx_{\ell_A}^{(i,0)} \omega$, from which it follows that ω_j runs the same thread, stepping to ω'_j. We conclude that $\omega'_j \approx_{\ell_A} \omega'$ by low-simulation and determinism, i.e., Requirements (i) and (ii).

Definition 3 (Non-interfering Scheduler). *A scheduler is non-interfering if it is satisfies Requirements 1,2, and 3.*

Round Robin We show that round-robin fulfills all the requirements and hence is an eligible candidate scheduler for our calculus. Firstly, it is immediately evident from the reductions that round-robin is *deterministic*, i.e., it fulfills scheduler requirement (i). We define the erasure function to filter out the identifiers of threads non observable to the attacker, i.e., $\varepsilon_{\ell_A}(s) = \mathit{filter}\ (\lambda(\ell, n) \to \ell \sqsubseteq \ell_A)\ s$. By induction on the scheduler reduction, it follows that round-robin satisfies *restricted simulation*, *no observable effect*, i.e., scheduler requirements (ii) and (iii). Before proving *progress* Figure 45 defines annotated ℓ_A-equivalence. In particular, if $\omega_1 \approx_{\ell_A}^{(0,0)} \omega_2$ for non-empty states ω_1 and ω_2, then round-robin will schedule the same *low* thread in the next reduction. Lastly round-robin is *starvation-free* because it has a finite time-slot and is preemptive.

Proposition 4 (RR non-interfering) *Round-robin is non-interfering.*

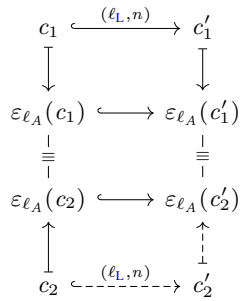

Fig. 46: 1-Step Progress.

10.3 Progress-sensitive Non-Interference

The proof of progress-sensitive non-interference relies on lemmas similar to those listed in Requirement 1. In the following, we write $c_1 \hookrightarrow_{(\ell,n)} c_2$ to denote that configuration c_1 steps to c_2 executing thread (ℓ, n) and we use ℓ_L and ℓ_H to denote $\ell_L \sqsubseteq \ell_A$ and $\ell_H \not\sqsubseteq \ell_A$ respectively. As usual, we write $\hookrightarrow^\star$ for the reflexive transitive closure of $\hookrightarrow$. We write $c_1 \approx_{\ell_A} c_2$ if and only if $\varepsilon_{\ell_A}(c_1) \equiv \varepsilon_{\ell_A}(c_2)$, to denote ℓ_A-equivalence between configurations and we lift scheduler annotations, i.e., $c_1 \approx^{(i,j)}_{\ell_A} c_2$ if and only if $c_1 \approx_{\ell_A} c_2$ and $\omega_1 \approx^{(i,j)}_{\ell_A} \omega_2$.

Proposition 5

i) Determinancy*: if* $c_1 \hookrightarrow c_2$ *and* $c_1 \hookrightarrow c_3$ *then* $c_2 \equiv c_3$.
ii) Restricted Simulation*: if* $c_1 \hookrightarrow_{(\ell_L,n)} c_2$ *then* $\varepsilon_{\ell_A}(c_1) \hookrightarrow_{(\ell_L,n)} \varepsilon_{\ell_A}(c_2)$.
iii) No Observable Effect*: if* $c_1 \hookrightarrow_{(\ell_H,n)} c_2$ *then* $c_1 \approx_{\ell_A} c_2$.

Using Proposition 5, we show that the concurrent semantics preserves ℓ_A-equivalence.

Proposition 6 ($\approx_{\ell_A}$ Preservation) *If* $c_1 \approx_{\ell_A} c_2$ *and* $c_1 \hookrightarrow_{(\ell,n)} c_1'$*, then*

- *If* $\ell \not\sqsubseteq \ell_A$*, then* $c_1' \approx_{\ell_A} c_2$.
- *If* $\ell \sqsubseteq \ell_A$ *and* $c_2 \hookrightarrow_{(\ell,n)} c_2'$*, then* $c_1' \approx_{\ell_A} c_2'$.

Progress sensitive non-interference requires to prove that ℓ_A-equivalence is preserved between two ℓ_A-equivalent configurations, even if only one steps. When a secret thread steps, the theorem follows easily by Proposition 6 and transitivity. The interesting case of the proof consists in showing *progress* of a public thread, which is simulated by the execution of multiple high threads followed by the same public thread, which corresponds to the diagram in Figure 44. Intuitively we prove *progress* by firstly simulating the secret threads that precede the public thread in the schedule (*scheduler progress*), then by simulating the common public thread under erasure (*restricted simulation*) and lastly

reconstructing from the erased step the original step in the other public thread. Before proving this proposition, we have to restrict configurations c_1 and c_2 to be *valid*—we explain why we need this assumption later on.

Definition 4 (Valid Configuration). *A concurrent configuration c is valid if and only if it does not contain any invalid memory reference, node $\bullet$ and terms $new_\bullet$, $write_\bullet$, $fork_\bullet$, $fmap_\bullet$, $\langle * \rangle_\bullet$, $relabel_\bullet$.*

Assuming valid configurations, we can prove *1-Step simulation*, i.e., the reconstruction of the other public step.

Proposition 7 (1-Step Progress) *If $c_1 \approx^{(i,0)}_{\ell_A} c_2$, $c_1 \hookrightarrow_{(\ell_L,n)} c_1'$ and c_2 is valid, then there exists c_2' such that $c_2 \hookrightarrow_{(\ell_L,n)} c_2'$.*

The diagram in Figure 46 shows our proof technique. Since the initial configurations are ℓ_A-equivalent, i.e., $c_1 \approx^{(i,0)}_{\ell_A} c_2$, then the erased initial configurations are equivalent, i.e., $\varepsilon_{\ell_A}(c_1) \equiv \varepsilon_{\ell_A}(c_2)$. Furthermore, since the schedulers in c_1 and c_2 are *aligned* (the second index in the annotated ℓ_A-equivalence is 0), the fact that the first scheduler runs thread (ℓ_L, n), implies that the second runs it as well (Proposition 2). Given $c_1 \hookrightarrow_{(\ell_L,n)} c_1'$ we obtain the erased reduction step $\varepsilon_{\ell_A}(c_1) \hookrightarrow_{(\ell_L,m)} \varepsilon_{\ell_A}(c_2)$, by *restricted simulation* and we then *reconstruct* c_2' and the other step $c_2 \hookrightarrow_{(\ell_L,n)} c_2'$ from the step $\varepsilon_{\ell_A}(c_1) \hookrightarrow_{(\ell_L,m)} \varepsilon_{\ell_A}(c_2)$, $\varepsilon_{\ell_A}(c_2) \equiv \varepsilon_{\ell_A}(c_1)$ and the assumption that c_1 and c_2 are valid.

Validity We explain by means of an example why we need to assume that the configurations c_1 and c_2 are *valid*. The fact that non-sensitive threads can write to sensitive resources, such as memories, complicates the reconstruction of a non-erased reduction step from an erased one, because, intuitively, too much information has been erased. For instance, since the erasure function rewrites secret memories and addresses to $\bullet$, we need to assume that the other program is in a "consistent state" in order to replay sensitive write memory-operations. Concretely, consider a public thread performing a secret write, i.e., $\langle \Sigma, write\ (Ref\ n)\ t \rangle \longrightarrow \langle \Sigma(\ell_H)[n] := t, return\ () \rangle$. A low-equivalent program will be $\langle \Sigma', write\ (Ref\ n')\ t' \rangle$, for some store Σ', address n' and term t' such that $\Sigma \approx_{\ell_A} \Sigma'$, $Ref\ n \approx_{\ell_A} Ref\ n'$ and $t \approx_{\ell_A} t'$. Unfortunately, there is no guarantee that n' is a valid address in memory $\Sigma'(\ell_H)$. Observe that the erasure function maps is non-injective: it maps *both* valid and invalid references to $Ref\ \bullet$, therefore knowing that n is defined in $\Sigma(\ell_H)$ does not guarantee that n' is valid in $\Sigma'(\ell_H)$. Before proving *progress*, we show that our semantics preserves *validity*.

Proposition 8 (Valid Preservation) *If c_1 is valid and $c_1 \hookrightarrow c_2$ then c_2 is valid.*

Proposition 9 (Progress) *If $c_1 \approx^{(i,j)}_{\ell_A} c_2$, $c_1 \hookrightarrow_{(\ell_L,n)} c_1'$, and c_1, c_2 are valid configurations, then there exists c_2' and c_2'' such that $c_2 \hookrightarrow^\star c_2' \hookrightarrow_{(\ell_L,n)} c_2''$.*

Proof (Sketch) The proof is driven by scheduler *progress*, i.e. Requirement 2, which determines what thread is scheduled next.

- ($j > 0$) The scheduler runs a secret thread, which is executed leading to the next intermediate configuration c_2', i.e. $c_2 \hookrightarrow_{(\ell_H, n')} c_2'$. By *no starvation*, i.e., Requirement 3, and *no observable effect*, i.e. Proposition 5.iii, it follows that $c_1 \approx_{\ell_A}^{(i,j')} c_2'$ for some $j' < j$ and we then apply induction.
- ($j = 0$) The scheduler runs public thread (ℓ_L, n) and the proposition follows from Proposition 7.

By combining *progress*, i.e., Proposition 9 and ℓ_A-equivalence *preservation*, i.e., Proposition 6, we prove PSNI.

Theorem 2 (Progress-sensitive non-interference) *Given valid global configurations c_1, c_1', c_2, and a non-interfering scheduler, if $c_1 \approx_{\ell_A} c_2$ and $c_1 \hookrightarrow c_1'$, then there exists c_2' such that $c_2 \hookrightarrow^\star c_2'$ and $c_2 \approx_{\ell_A} c_2'$.*

We conclude with a corollary that asserts that **MAC** satisfies PSNI.

Corollary 1 ***MAC** satisfies PSNI.*

Proof By applying Theorem 2 and Proposition 4.

11 Related Work

Mechanized Proofs Russo presents the library **MAC** as a functional pearl and relies on its simplicity to convince readers about its correctness [45]. This work bridges the gap on **MAC**'s lack of formal guarantees and exhibits interesting insights on the proofs of its soundness. **LIO** is a library structural similar to **MAC** but dynamically enforcing IFC [54]. The core calculus of **LIO**, i.e., side-effect free computations together with exception handling, has been modeled in the Coq proof assistant [53]. Different from our work, these mechanized proofs do not simplify the treatment of sensitive exceptions by masking them in erased programs. In parallel to [53], Breeze [23] is a pure programming language that explores the design space of IFC and exceptions, which is accompanied with mechanized proofs in Coq. Bichhawat et al. develop an intra-procedural analysis for Javascript bytecode, which prevents implict leaks in presence of exceptions and unstructured control flow constructs [7].

Concurrency Considering IFC for a general scheduler could lead to refinements attacks. In this light, Russo and Sabelfeld provide termination-insensitive non-interference for a wide-class of deterministic schedulers [44]. Barthe et al. adopt this idea for Java-like bytecode [3]. Although we also consider deterministic schedulers, our security guarantees are stronger by considering leaks of information via abnormal termination. Heule et al. describe how to retrofit IFC in a programming language with sandboxes [20]. Similar to our work, their soundness

proofs are parametric on deterministic schedulers and provide progress-sensitive non-interference with informal arguments regarding thread progress—in this work, we spell out formal requirements on schedulers capable to guarantee thread progress. A series of work for π-calculus consider non-deterministic schedulers while providing progress-sensitive non-interference [21, 22, 27, 40].

Security Libraries Li and Zdancewic's seminal work [29] shows how the structure *arrows* can provide IFC as a library in Haskell. Tsai et al. extend that work to support concurrency and data with heterogeneous labels [56]. Russo et al. implement the security library **SecLib** using a simpler structure than arrows [43], i.e. monads—rather than labeled values, this work introduces a monad which statically label side-effect free values. The security library **LIO** [51, 52] dynamically enforces IFC for both sequential and concurrent settings. **LIO** presents operations similar to $fmap$ and $\langle * \rangle$ for labeled values with differences in the returning type due to **LIO**'s checks for clearence—this work provides a foundation to analyze the security implications of such primitives. Mechanized proofs for **LIO** are given only for its core sequential calculus [52]. Inspired by **SecLib** and **LIO**'s designs, **MAC** leverages Haskell's type system to enforce IFC [45]. **HLIO** uses advanced Haskell's type-system features to provide a hybrid approach: IFC is statically enforced while allowing the programmers to defer selected security checks until run-time [10]. Our work studies the security implications of extending **LIO**, **MAC**, and **HLIO** with a rich structure for labeled values. Devriese and Piessens provide a monad transformer to extend imperative-like APIs with support for IFC in Haskell [13]. Jaskelioff and Russo implements a library which dynamically enforces IFC using secure multi-execution (SME) [25]—a technique that runs programs multiple times (once per security level) and varies the semantics of inputs and outputs to protect confidentiality. Rather than running multiple copies of a program, Schmitz et al. provide a library with *faceted values* [48], where values present different behavior according to the privilege of the observer. Different from the work above, we present a fully-fledged mechanized proof for our sequential and concurrent calculus which includes references, synchronization variables, and exceptions.

IFC Tools IFC research has produced compilers capable of preserving confidentiality of data: Jif [37] and Paragon [8] (based on Java), and FlowCaml [49] (based on Caml). The SPARK language presents a IFC analysis which has been extended to guarantee progress-sensitive non-inference [41]. JSFlow [17] is one of the state-of-the-art IFC system for the web (based on JavaScript). These tools preserve confidentiality in a fine-grained fashion where every piece of data is explicitly label. Specifically, there is no abstract data type to label data, so our results cannot directly apply to them.

Operating Systems **MAC** borrows ideas from Mandatory Access Control (MAC) [5, 6] and phrases them into a programming language setting. Although originated in the 70s, there are modern manifestations of this idea [28, 35, 60], applied

to diverse scenarios, like the web [4, 55] and mobile devices [9, 26]. Due to its complexity, it is not surprising that OS-based MAC systems lack accompanying soundness guarantees or mechanized proofs—**seL4** being the exception [35]. The level of abstractions handled by **MAC** and OSes are quite different, thus making uncertain how our insights could help to formalize OS-based MAC systems. MAC systems [5] assign a label with an entire OS process—settling a single policy for all the data handled by it. In principle, it would be possible to extend such MAC-like systems to include a notion of labeled values with the functor structure as well as the relabeling primitive proposed by this work. For instance, COWL [55] presents the notion of *labeled blob* and *labeled XHR* which is isomorphic to the notion of labeled values, thus making possible to apply our results. Furthermore, because many MAC-like system often ignore termination leaks [14, 60], there is no need to use call-by-name evaluation to obtain security guarantees.

12 Conclusion

We present a *full-fledged* formalization of **MAC** in Agda, where non-interference is proven by term erasure. To the best of our knowledge, this is the first work of its kind for IFC libraries in Haskell, both for completeness and number of features included in the model. Thanks to our mechanized proofs, we identify challenges arising from erasing terms depending on the context where they appear and propose *two-steps erasure*—an effective technique to systematically deal with such cases. We present an extension of **MAC** that provides labeled values with an applicative functor-like structure and a relabeling operation, enabling convenient and expressive manipulation of labeled values using side effect-free code and saving programmers from introducing unnecessary sub-computations, e.g., forking a thread. We have proved this extension secure both in sequential and concurrent settings, using *two-steps erasure*. This work bridges the gap between existing IFC libraries (which focus on side-effecting code) and the usual Haskell programming model (which favors pure code), with a view to making IFC in Haskell more practical. Our mechanized proofs also make explicit sufficient scheduler requirements to guarantee PSNI—something that has been only treated informally before [20, 51]. As a result, our security proofs for the concurrent calculus are valid for a wide-range of deterministic schedulers. It is our hope that the insights gained by this work will help to formally verify other IFC programming languages.

References

1. A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proc. of the European Symposium on Research in Computer Security (ESORICS '08)*. Springer-Verlag, 2008.
2. Anindya Banerjee and David A. Naumann. Stack-based access control and secure information flow. *Journal Functional Programming*, 15:131–177, March 2005.
3. G. Barthe, T. Rezk, A. Russo, and A. Sabelfeld. Security of multithreaded programs by compilation. *Special issue of ACM Transactions on Information and System Security (TISSEC)*, August 2009.
4. Lujo Bauer, Shaoying Cai, Limin Jia, Timothy Passaro, Michael Stroucken, and Yuan Tian. Run-time monitoring and formal analysis of information flows in Chromium. In *Proc. of the Annual Network & Distributed System Security Symposium*. Internet Society, 2015.
5. David E. Bell and L. La Padula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997, Rev. 1, MITRE Corporation, Bedford, MA, 1976.
6. K. J. Biba. Integrity considerations for secure computer systems. ESD-TR-76-372, 1977.
7. Abhishek Bichhawat, Vineet Rajani, Deepak Garg, and Christian Hammer. *Information Flow Control in WebKit's JavaScript Bytecode*. Springer Berlin Heidelberg, 2014.
8. Niklas Broberg, Bart van Delft, and David Sands. Paragon for practical programming with information-flow control. In *APLAS*, volume 8301 of *Lecture Notes in Computer Science*, pages 217–232. Springer, 2013.
9. Sven Bugiel, Stephan Heuser, and Ahmad-Reza Sadeghi. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *Proc. of the 22Nd USENIX Conference on Security*, SEC'13. USENIX Association, 2013.
10. P. Buiras, D. Vytiniotis, and A. Russo. HLIO: Mixing static and dynamic typing for information-flow control in Haskell. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP '15)*. ACM, 2015.
11. Pablo Buiras, Deian Stefan, and Alejandro Russo. On dynamic flow-sensitive floating-label systems. In *Proceedings of the 2014 IEEE 27th Computer Security Foundations Symposium*, CSF '14, pages 65–79, Washington, DC, USA, 2014. IEEE Computer Society.
12. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.
13. D. Devriese and F. Piessens. Information flow enforcement in monadic libraries. In *Proc. of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI '11)*. ACM, 2011.
14. Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. In *Proc. of the twentieth ACM symp. on Operating systems principles*, SOSP '05. ACM, 2005.
15. J. A. Goguen and J. Meseguer. Unwinding and inference control. In *Security and Privacy, 1984 IEEE Symposium on*, April 1984.
16. J.A. Goguen and J. Meseguer. Security policies and security models. In *Proc of IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1982.

17. D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. In *Proc. of the ACM Symposium on Applied Computing (SAC '14)*. ACM, March 2014.
18. D. Hedin and David Sands. Noninterference in the presence of non-opaque pointers. In *Proc. of the 19th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 2006.
19. M. Hellman. A cryptanalytic time-memory trade-off. *IEEE Transactions on Information Theory*, 26(4):401–406, July 1980.
20. Stefan Heule, Deian Stefan, Edward Z. Yang, John C. Mitchell, and Alejandro Russo. IFC inside: Retrofitting languages with dynamic information flow control. In *Conference on Principles of Security and Trust (POST)*. Springer, April 2015.
21. Kohei Honda, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. Secure information flow as typed process behaviour. In *Proc. of the 9th European Symposium on Programming Languages and Systems*. Springer-Verlag, 2000.
22. Kohei Honda and Nobuko Yoshida. A uniform type structure for secure information flow. *ACM Trans. Program. Lang. Syst.*, October 2007.
23. C. Hritcu, M. Greenberg, B. Karel, B. C. Peirce, and G. Morrisett. All your IFCexception are belong to us. In *Proc. of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2013.
24. John Hughes. Why functional programming matters. *The Computer Journal*, 32, 1984.
25. M. Jaskelioff and A. Russo. Secure multi-execution in Haskell. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, LNCS. Springer-Verlag, June 2011.
26. Limin Jia, Jassim Aljuraidan, Elli Fragkaki, Lujo Bauer, Michael Stroucken, Kazuhide Fukushima, Shinsaku Kiyomoto, and Yutaka Miyake. Run-time enforcement of information-flow properties on Android (extended abstract). In *Computer Security—ESORICS 2013: 18th European Symposium on Research in Computer Security*. Springer, September 2013.
27. Naoki Kobayashi. Type-based information flow analysis for the π-calculus. *Acta Inf.*, 42(4), December 2005.
28. Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *Proceedings of ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07. ACM, 2007.
29. P. Li and S. Zdancewic. Encoding information flow in Haskell. In *Proc. of the IEEE Workshop on Computer Security Foundations*. IEEE Computer Society, 2006.
30. P. Li and S. Zdancewic. Arrows for secure information flow. *Theoretical Computer Science*, 411(19):1974–1994, 2010.
31. Simon Marlow. *Parallel and concurrent programming in Haskell*. O'Reilly, July 2013.
32. Conor Mcbride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 2008.
33. Simon Meurer and Roland Wismüller. Apefs: An infrastructure for permission-based filtering of android apps. In AndreasU. Schmidt, Giovanni Russello, Ioannis Krontiris, and Shiguo Lian, editors, *Security and Privacy in Mobile Information and Communication Systems*, volume 107. Springer Berlin Heidelberg, 2012.
34. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

35. Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. seL4: From general purpose to a proof of information flow enforcement. *2012 IEEE Symposium on Security and Privacy*, 0, 2013.
36. A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 228–241, January 1999.
37. A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java Information Flow. `http://www.cs.cornell.edu/jif`, 2001.
38. Philippe Oechslin. *Making a Faster Cryptanalytic Time-Memory Trade-Off*, pages 617–630. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.
39. F. Pottier and V. Simonet. Information Flow Inference for ML. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 319–330, January 2002.
40. François Pottier. A simple view of type-secure information flow in the π-calculus. In *In Proc. of the 15th IEEE Computer Security Foundations Workshop*, pages 320–330, 2002.
41. Willard Rafnsson, Deepak Garg, and Andrei Sabelfeld. Progress-sensitive security for SPARK. In *Engineering Secure Software and Systems - 8th International Symposium, ESSoS 2016, London, UK, April 6-8, 2016. Proceedings*, 2016.
42. Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: Practical fine-grained decentralized information flow control. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09. ACM, 2009.
43. A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell. In *Proc. ACM SIGPLAN symposium on Haskell (HASKELL '08)*. ACM, September 2008.
44. A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *Proc. IEEE Computer Sec. Foundations Workshop*, pages 177–189, July 2006.
45. Alejandro Russo. Functional pearl: Two can keep a secret, if one of them uses Haskell. In *Proc. of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015. ACM, 2015.
46. A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.
47. Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.
48. Thomas Schmitz, Dustin Rhodes, Thomas H. Austin, Kenneth Knowles, and Cormac Flanagan. Faceted dynamic information flow via control and data monads. In Frank Piessens and Luca Viganò, editors, *POST*, volume 9635 of *Lecture Notes in Computer Science*. Springer, 2016.
49. V. Simonet. The Flow Caml system. Software release at `http://cristal.inria.fr/~simonet/soft/flowcaml/`, 2003.
50. G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM symposium on Principles of Programming Languages (POPL '98)*, 1998.
51. D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Maziéres. Addressing covert termination and timing channels in concurrent information flow systems. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP '12)*. ACM, 2012.
52. D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Proc. of the ACM SIGPLAN Haskell symposium (HASKELL '11)*, 2011.

53. D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in the presence of exceptions. *Arxiv preprint arXiv:1207.1457, to appear in Journal of Functional Programming, Cambridge University Press*, 2012.
54. Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in Haskell. In *Proceedings of the 4th ACM symposium on Haskell*, pages 95–106, New York, NY, USA, 2011. ACM.
55. Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, Dave 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)*. USENIX Association, October 2014.
56. T. C. Tsai, A. Russo, and J. Hughes. A library for secure multi-threaded information flow in Haskell. In *Proc. IEEE Computer Security Foundations Symposium (CSF '07)*, July 2007.
57. Marco Vassena, Pablo Buiras, Lucas Waye, and Alejandro Russo. Flexible manipulation of labeled values for information-flow control libraries. In *Proceedings of the 12th European Symposium On Research In Computer Security*. Springer, September 2016.
58. 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 '16, pages 15–28, New York, NY, USA, 2016. ACM.
59. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 60–76, New York, NY, USA, 1989. ACM.
60. Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proc. of the 7th USENIX Symp. on Operating Systems Design and Implementation*. USENIX, 2006.

$$(\textsc{Relabel}_\chi)$$
$$relabel\ (Labeled_\chi\ t) \rightsquigarrow Labeled_\chi\ t$$

$$(\langle * \rangle_{\chi 1})$$
$$\frac{t_2 \rightsquigarrow t_2'}{(Labeled_\chi\ t_1)\ \langle * \rangle\ t_2 \rightsquigarrow (Labeled_\chi\ t_1)\ \langle * \rangle\ t_2'}$$

$$(\langle * \rangle_{\chi 2})$$
$$(Labeled_\chi\ t_1)\ \langle * \rangle\ (Labeled\ t_2) \rightsquigarrow Labeled_\chi\ t_1$$

$$(\langle * \rangle_{\chi 3})$$
$$(Labeled_\chi\ t_1)\ \langle * \rangle\ (Labeled_\chi\ t_2) \rightsquigarrow Labeled_\chi\ t_1$$

$$(\langle * \rangle_{\chi 4})$$
$$(Labeled\ t_1)\ \langle * \rangle\ (Labeled_\chi\ t_2) \rightsquigarrow Labeled_\chi\ t_2$$

Fig. 47: Semantics of flexible labeled values with exceptions.

Appendix

A Flexible Labeled Values in Sequential Calculus

In this section, we extend the semantics of flexible labeled values described in Section 9 for the sequential setting, where labeled values have an additional constructor, namely $Labeled_\chi$. This constructor is used to prevent *sensitive* exceptions from leaking into a *non-sensitive* context, when embedding a secret computation in a public one using $join$. The semantics of the primitives $relabel$ and $\langle * \rangle$ handle exceptional values, by propagating the exceptions, which is exactly what happens in rule [$\textsc{Relabel}_\chi$]—see Figure 47. Rules [$\langle * \rangle_{\chi 1}$, $\langle * \rangle_{\chi 2}$, $\langle * \rangle_{\chi 3}$, $\langle * \rangle_{\chi 4}$] yield (propagate) the first exception observed when $\langle * \rangle$ is applied to exceptional values. In particular, rule [$\langle * \rangle_{\chi 3}$] applies when both arguments are exceptions and returns the first one triggered during evaluation, i.e., the left one. Rules [$\langle * \rangle_{\chi 1}$, $\langle * \rangle_{\chi 2}$, $\langle * \rangle_{\chi 3}$] are somewhat unusual. In particular, even though our language is *non-strict*, the rules give a *strict* semantics to $\langle * \rangle$—note that they reduce unnecessarily the second term, even though it is not used in the final result. It would have been more natural, in this context, to replace them by a single rule $Labeled_\chi\ t_1\ \langle * \rangle\ t_2 \rightsquigarrow Labeled_\chi\ t_1$, that does not evaluate the second term. The two alternative semantics are equivalent, except for abnormal *non-terminating* terms, that we denote with $\bot$. With *strict* semantics the term $(Labeled_\chi\ t_1)\ \langle * \rangle\ \bot$ results in $\bot$, because it loops due to rule [$\langle * \rangle_{\chi 1}$], instead it terminates with a *non-strict* semantics, i.e., $(Labeled_\chi\ t_1)\ \langle * \rangle\ \bot \rightsquigarrow (Labeled_\chi\ t_1)$. We remark that the two semantics are equivalent for *terminating* programs and therefore security is not at stake: the sequential calculus is already *termination insensitive*. Technically, we give a strict definition of $\langle * \rangle$, because erasing sensitive exceptions are replaced by

$$
\begin{array}{l}
\textbf{data}\ \mathit{MVar}\ \ell\ \tau \\
\mathit{newMVar} :: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow \mathit{MAC}\ \ell_{\mathrm{L}}\ (\mathit{MVar}\ \ell_{\mathrm{H}}\ \tau) \\
\mathit{takeMVar} :: \mathit{MVar}\ \ell\ \tau \rightarrow \mathit{MAC}\ \ell\ \tau \\
\mathit{putMVar} :: \mathit{MVar}\ \ell\ \tau \rightarrow \tau \rightarrow \mathit{MAC}\ \ell\ ()
\end{array}
$$

Fig. 48: API of synchronization primitives.

non-exceptional values, i.e., $\varepsilon_L(\mathit{Labeled}\ H\ \tau)\ (\mathit{Labeled}_\chi\ t) = \mathit{Labeled}\ \bullet$. Therefore, we could not prove simulation for a non-strict applicative functor, since, crucially, it is sensitive to exceptions. While these behaviour could be simulated by an erasure function that preserves sensitive exceptions, i.e., $\varepsilon_L(\mathit{Labeled}\ H\ \tau)\ (\mathit{Labeled}_\chi\ \tau) = \mathit{Labeled}_\chi\ \bullet$, it is an open question how to prove *single-step simulation* for join, specifically for rules [JOIN,JOIN$_\chi$].

B Synchronization Primitives

In this section we extend our calculus with synchronization primitives, an essential feature for concurrent programs. Using synchronized mutable variables (MVar) users can implement simple inter-thread communication mechanisms such as binary semaphores and channels.

The type $\mathit{MVar}\ \ell\ \tau$ denotes a labeled mutable location that is either empty or full and contains a term of type τ of security level τ. Figure 48 shows the API of basic synchronization primitives, based on MVar. Specifically, function $\mathit{newMVar}$ creates an empty MVar. Function $\mathit{takeMVar}$ empty a full MVar and returns its content or *blocks* otherwise. Function $\mathit{putMVar}$ fills an empty MVar or blocks otherwise. Primitive $\mathit{newMVar}$ performs a *write* operation, therefore its type is restricted to comply with the *no write-down* policy, just like the type of new for memory. Interestingly, and unlike memory primitives read and write, the type of $\mathit{takeMVar}$ and $\mathit{putMVar}$ accepts only one security level. Intuitively, that is the case because MVar's primitives perform *both* read and write side-effects, therefore both *no read-up* and *no write-down* security policies apply. For instance, to execute $\mathit{putMVar}$, it is necessary to observe (read) if the MVar is empty. We show how those security policy guide our design and lead us to give the API shown in Figure 48 as the only secure option. Assume that primitive $\mathit{takeMVar}$ had a completely unrestricted type, i.e., $\forall\ \ell_1\ \ell_2\ .\ \mathit{MVar}\ \ell_1\ \tau \rightarrow \mathit{MAC}\ \ell_2\ \tau$. Since $\mathit{takeMVar}$ returns the content of the MVar—a *read* effect that is secure only if ℓ_2 is at least as sensitive as ℓ_1, i.e., $\ell_1 \sqsubseteq \ell_2$. Observe however that $\mathit{takeMVar}$ empties the MVar as well, after returning its content—a *write* effect that is secure only if ℓ_1 is at least as sensitive as ℓ_2, i.e., $\ell_2 \sqsubseteq \ell_1$. By the antisymmetry of the security lattice, it follows that the interaction between computations and synchronization variables is secure only when they have the same security level, i.e., $\ell_1 \equiv \ell_2$. The same principle applies for $\mathit{putMVar}$.

$$
\begin{array}{lll}
\text{Memory} & \ell\ t_s ::= [\,] \mid c : t_s & \\
\text{Cell} & c ::= \otimes \mid [\![t]\!] & \\
\text{Types:} & \tau ::= \cdots \mid \mathit{MVar}\ \ell\ \tau & \\
\text{Values:} & v ::= \cdots \mid \mathit{MVar}\ n & \\
\text{Terms:} & t ::= \cdots \mid \mathit{newMVar} \mid \mathit{takeMVar}\ t \mid \mathit{putMVar}\ t_1\ t_2 &
\end{array}
$$

$$\text{(NEWMVAR)}\quad \frac{|\Sigma(\ell)| = n}{\langle \Sigma, \mathit{newMVar}\rangle \longrightarrow \langle \Sigma(\ell)[n] := \otimes, \mathit{return}\ (\mathit{MVar}\ n)\rangle}$$

$$\text{(PUTMVAR}_1\text{)}\quad \frac{t_1 \rightsquigarrow t_1'}{\langle \Sigma, \mathit{putMVar}\ t_1\ t_2\rangle \longrightarrow \langle \Sigma, \mathit{putMVar}\ t_1'\ t_2\rangle}$$

$$\text{(PUTMVAR}_2\text{)}\quad \frac{\Sigma(\ell)[n] = \otimes}{\langle \Sigma, \mathit{putMVar}\ (\mathit{MVar}\ n)\ t\rangle \longrightarrow \langle \Sigma(\ell)[n] := [\![t]\!], \mathit{return}\ ()\rangle}$$

$$\text{(TAKEMVAR}_1\text{)}\quad \frac{t \rightsquigarrow t'}{\langle \Sigma, \mathit{takeMVar}\ t\rangle \longrightarrow \langle \Sigma, \mathit{takeMVar}\ t'\rangle}$$

$$\text{(TAKEMVAR}_2\text{)}\quad \frac{\Sigma(\ell)[n] = [\![t]\!]}{\langle \Sigma, \mathit{takeMVar}\ (\mathit{MVar}\ n)\rangle \longrightarrow \langle \Sigma(\ell)[n] := \otimes, \mathit{return}\ t\rangle}$$

Fig. 49: **MAC** with synchronization primitives.

B.1 Calculus

Figure 49 extends the concurrent calculus with synchronization primitives. A synchronization variable is represented as a value $\mathit{MVar}\ n :: \mathit{MVar}\ \ell\ \tau$ where n is an address[18], pointing to the n-th cell of the ℓ-memory, which contains a term of type τ. We adjust our memory model to work with synchronization variables [19]. We introduce a new syntactic category, memory cell c, which can be either *empty*, i.e., $\otimes$, or *full* with some term t, i.e., $[\![t]\!]$. Rule [NEWMVAR] evaluates term $\mathit{newMVar}$ by adding an empty memory cell to the ℓ-labeled memory, i.e., $\Sigma(\ell)[n] := \otimes$ and returning a reference to it, i.e., $\mathit{MVar}\ n$. Rule [PUTMVAR$_1$] evaluates the reference and rule [PUTMVAR$_2$] fills the empty cell it refers to with the term, i.e., $\Sigma(\ell)[n] := [\![t]\!]$ and returns unit. Rule [TAKEMVAR$_1$] evaluates the

[18] In **MAC** a MVar is just a wrapper around unlabeled synchronization variables from the standard library. Here we denote synchronization variables as an index, just like we did for memory references.

[19] We model mutable references as a special case of synchronization variables that are always full.

$$\varepsilon_{\ell_A}(\otimes) = \otimes \qquad \varepsilon_{\ell_A}([\![t]\!]) = [\![\varepsilon_{\ell_A}(t)]\!]$$

(a) Erasure for memory cells.

$$\varepsilon_{\ell_A}(newMVar :: MAC\ \ell_{\mathrm{L}}\ (MVar\ \ell_{\mathrm{H}}\ \tau)) = \begin{cases} newMVar_\bullet & \text{if } \ell_{\mathrm{H}} \not\sqsubseteq \ell_A \\ newMVar & \text{otherwise} \end{cases}$$

$$\varepsilon_{\ell_A}(MVar\ n :: MVar\ \ell_{\mathrm{H}}\ \tau) = \begin{cases} MVar\ \bullet & \text{if } \ell_{\mathrm{H}} \not\sqsubseteq \ell_A \\ MVar\ n & \text{otherwise} \end{cases}$$

(b) Erasure for $newMVar$ and $MVar$.

Fig. 50: Erasure function for memory cells and synchronization primitives.

reference and rule [TAKEMVAR$_2$] returns the content of the non-empty cell it refers to, i.e., $\Sigma(\ell)[n] = [\![t]\!]$ for *some* term t, and empties it, i.e., $\Sigma(\ell)[n] := \otimes$. Observe that the premise of rules [PUTMVAR$_2$] and [TAKEMVAR$_2$] accounts for the *blocking* behavior of the synchronization primitives by making the configuration *stuck*. In particular, primitive $putMVar$ *blocks* if the cell is non-empty, i.e., $\langle \Sigma, putMVar\ (MVar\ n)\ t\rangle \not\longrightarrow$ if $\Sigma(\ell)[n] \not\equiv \otimes$ and similarly $takeMVar$ *blocks* if the cell is empty, i.e., $\langle \Sigma, takeMVar\ (MVar\ n)\rangle \not\longrightarrow$ if $\Sigma(\ell)[n] \equiv \otimes$.

B.2 Erasure Function

Proving that synchronization primitives are secure is straightforward in our setting. The primitives are clearly *deterministic* and showing *single-step simulation* is simpler than for references because primitives $putMVar$ and $takeMVar$ work within the same security level. Memory cells are erased homomorphically (Figure 50a). Applying the *two-steps erasure* technique, the erasure function replaces term $newMVar$ with $newMVar_\bullet$, when it creates a sensitive synchronization variable—see Figure 50b. The erasure function rewrites the address of a synchronization reference to $\bullet$ if it points to a sensitive memory. Figure 51 shows rule [NEWMVAR$_\bullet$], which reduces term $newMVar_\bullet$, returns a dummy reference, i.e., $MVar\ \bullet$, and *skips* the write effect, leaving the store Σ unchanged. Observe that we do not need to replace $takeMVar$ with a special term $takeMVar_\bullet$, because the primitive can only write to a memory at the same security level as the computation, therefore either they are both sensitive and the computation rewritten to $\bullet$ or both non-sensitive and erased homomorphically.

Terms: $t ::= \cdots \mid newMVar_\bullet$

(NEWMVAR$_\bullet$)
$\langle \Sigma, newMVar_\bullet \rangle \longrightarrow \langle \Sigma, return\ (MVar\ \bullet) \rangle$

Fig. 51: Semantics of $newMVar_\bullet$.

PAPER III

Revised version of

Securing Concurrent Lazy Programs Against Information Leakage,

by Marco Vassena, Joachim Breitner and Alejandro Russo,

30th IEEE Computer Security Foundations Symposium.

CHAPTER

THREE

SECURING CONCURRENT LAZY PROGRAMS AGAINST INFORMATION LEAKAGE

Abstract. Many state-of-the-art information-flow control (IFC) tools are implemented as Haskell libraries. A distinctive feature of this language is lazy evaluation. In his influencal paper on why functional programming matters [19], John Hughes proclaims:

> *Lazy evaluation is perhaps the most powerful tool for modularization in the functional programmer's repertoire.*

Unfortunately, lazy evaluation makes IFC libraries vulnerable to leaks via the internal timing covert channel. The problem arises due to *sharing*, the distinguishing feature of lazy evaluation, which ensures that results of evaluated terms are stored for subsequent re-utilization. In this sense, the evaluation of a term in a high context represents a *side-effect* that eludes the security mechanisms of the libraries. A naïve approach to prevent that consists in forcing the evaluation of terms before entering a high context. However, this is not always possible in lazy languages, where terms often denote infinite data structures. Instead, we propose a new language primitive, $lazyDup$, which duplicates terms *lazily*. We make the security library **MAC** robust against internal timing leaks via lazy evaluation, by using $lazyDup$ to duplicate terms manipulated in high contexts, as they are evaluated. We show that well-typed programs satisfy progress-sensitive non-interference in our lazy calculus with non-strict references. Our security guarantees are supported by mechanized proofs in the Agda proof assistant.

1 Introduction

Information-Flow Control [41] (IFC) scrutinizes source code to track how data of different sensitivity levels (e.g., public or sensitive) flows within a program,

and raises alarms when confidentiality might be at stake. There are several special-purpose compilers and interpreters which apply this technology: *Jif* [29] (based on Java), *FlowCaml* [36] (based on Caml and not developed anymore), *Paragon* [9] (based on Java), and *JSFlow* [16] (based on JavaScript). Rather than writing compilers/interpreters, IFC can also be provided as a library in the functional programming language Haskell [22].

Haskell's type system enforces a disciplined separation of side-effect free from side-effectful code, which makes it possible to introduce input and output (I/O) to the language without compromising on its *purity*. Computations performing side-effects are encoded as values of abstract types which have the structure of monads [26]. This distinctive feature of Haskell is exploited by state-of-the-art IFC libraries (e.g., **LIO** [48] and **MAC** [40]) to identify and restrict "leaky" side-effects without requiring changes to the language or runtime.

Another distinctive feature of Haskell is its *lazy evaluation* strategy. This evaluation is non-strict, as function arguments are not evaluated until required by the function, and it performs *sharing*, as the values of such arguments are stored for subsequent uses. In contrast, eager evaluation, also known as *strict evaluation*, reduces function arguments to their denoted values before executing the function.

From a security point of view, it is unclear which evaluation strategy—lazy or strict—is more suitable to preserve secrets. To start addressing this subtlety, we need to consider the interaction between evaluation strategies and covert channels.

Sabelfeld and Sands [42] suggest that lazy evaluation might be intrinsically safer than eager evaluation for leaks produced by termination—as lazy evaluation could skip the execution of *unneeded* non-terminating computations that might involve secrets. In multi-threaded systems, where termination leaks are harmful [47], a lazy evaluation strategy seems to be the appropriate choice.

Unfortunately, although lazy evaluation could "save the day" when it comes to termination leaks, it is also vulnerable to leaks via another covert channel due to sharing. Buiras and Russo [11] described an attack against the **LIO** library [47] where lazy evaluation is exploited to leak information via the *internal timing covert channel* [45]. This covert channel manifests by the mere presence of concurrency and shared resources. It gets exploited by setting up threads to race for a public shared resource in such a way that the secret value affects their timing and hence the winner of the race. **LIO** removes such leaks for public shared-resources which can be identified by the library (e.g., references). Due to lazy evaluation, variables introduced by `let`-bindings and function applications—which are beyond **LIO**'s control[20]—become shared resources and their evaluation affects the threads' timing behavior. Note that the *internal* timing channel leverages the *order* with which threads access the shared resource, not their exe-

[20] As a shallow EDSL, **LIO** reuses much of the host language features to provide security (e.g., type-system and variable bindings). This design choice makes the code base small at the price of not fully controlling the features provided by the host language.

```
let ℓ = [1 .. 10000000]
    r = sum ℓ
in do fork^LIO    -- Secret thread
        (do s ← unlabel secret
            when (s ≡ 1 ∧ r ⩾ 10) return ())
      no_ops; no_ops
        -- Public threads
      fork^LIO (do sendPublicMsg (r − r))
      fork^LIO (do no_ops; sendPublicMsg 1)
```

Fig. 1: Lazy evaluation attack

cution time, which constitutes a different covert channel, known as the *external* timing covert channel [6, 14]. The attacker model for the external timing covert channel assumes that the attacker has access to an arbitrarily precise stopwatch to measure the wall-clock execution time of instructions and thereby deduce information about secrets. This paper does not address the external timing covert channel, which is a harder problem and for which mitigation techniques exist [2, 52, 53].

Figure 1 shows the lazy evaluation attack. In **LIO**, every thread has a current label which serves a role similar to the *program counter* in traditional IFC systems [51]. The first thread inspects a secret value ($s \leftarrow unlabel\ secret$), which sets the current label to secret. We refer to threads with such current label as *secret threads*. The other spawned threads have their current label set to public, therefore we call them *public threads*. Observe that the variable r hosts an expression that is somewhat expensive to calculate, as it first builds a list with ten million numbers ($\ell = [1\,..\,10000000]$) before summing up its elements ($r = sum\ \ell$). Importantly, the variable r is referenced by both the secret and the public threads. Observe that every thread is secure in isolation—the secret thread always returns $()$ and the public threads read no secret. Assume that the expression no_ops is some irrelevant computation that takes slightly longer than half the time it takes to sum up the ten million numbers. Then the public threads race to send a message on a shared-public channel using the function $sendPublicMsg$:

▷ If $s \equiv 1$, then the secret thread has by now evaluated the expression referenced by r, in order to check if $r \geqslant 10$ holds. Due to sharing, the first public thread will not have to re-calculate r and can output 0 almost imediately, while the other public thread is still occupied with no_ops.

▷ If $s \equiv 0$, then the secret thread did not touch r. While the first public thread now has to evaluate r the second public thread has enough time to perform no_ops and output 1 first.

As a result, the last message on the public channel reveals the secret s. This attack can be magnified to a point where whole secrets are leaked systemati-

cally and efficiently [47]. Similar to **LIO**, other state-of-the-art concurrent IFC Haskell libraries [10, 40] suffer from this attack.

A naïve fix is to force variable r to be fully evaluated before any public threads begin their execution. This works, but it defeats the main purpose of lazy evaluation, namely to avoid evaluating unneeded expressions. Furthermore, it is not always possible to evaluate expressions to their denoted value. Haskell programmers like to work with infinite structures, even though only finite approximation of them are actually used by programs. For example, if variable ℓ in Figure 1 were the list $[1/n \mid n \leftarrow [1\,..]]$ of reciprocals of all natural numbers and r the sum of those bigger than one millionth ($r = sum\ (takeWhile\ (\geqslant 1\mathrm{e}{-6})\ \ell)$). The evaluation of r uses only a finite portion of ℓ, so the modified program still terminates. But naïvely forcing ℓ to normal form would hang the program. This demonstrates that simply forcing evaluation as a security measure is unsatisfying, as it can introduce divergence and thus change the meaning of a program.

Instead, we present a novel approach to explicitly control sharing at the language level. We design a new primitive called $lazyDup$ which *lazily* duplicates unevaluated expressions. The attack in Figure 1 can then be neutralized by replacing r with $lazyDup\ r$ in the secret thread, which will then evaluate its own copy of r, without affecting the public threads.

To the best of our knowledge, this work is the first one to formally address the problem of internal timing leaks via lazy evaluation. In summary, our contributions are:

- We present $lazyDup$, a primitive to restrict sharing in lazy languages with mutuable references.
- By injecting $lazyDup$ when spawning threads, we demonstrate that internal timing leaks via lazy evaluation are closed. The primitive $lazyDup$ is not only capable to secure **MAC** against lazy leaks, but also a wide range of other security Haskell libraries (e.g., **LIO** and **HLIO**).
- We prove that well-typed programs satisfy progress-sensitive non-interference (PSNI) for a wide-range of deterministic schedulers. However, for ease of exposition in this article, we focus only on a round-robin scheduler—the same scheduler used in GHC's runtime system[21]. Our non-interference claims are supported by mechanized proofs in the Agda proof assistant [31] and are parametric on the chosen (deterministic) scheduler.
- As a by-prodcut of interest for the programming language community, we provide—to the best of our knowledge—the first operational semantics for lazy evaluation with mutable references.

This paper is organized as follows. Section 2 provides a brief overview on **MAC**. Section 3 describes our formalization for a concurrent non-strict calculus with sharing that also includes references. Primitive $lazyDup$ is described in Section 4. Section 5 shows how $lazyDup$ can remove leaks via lazy evaluation

[21] The Glasgow Haskell Compiler (GHC) is a state-of-the-art, industrial-strength, open source, Haskell compiler.

```
   -- Abstract types
data Labeled ℓ τ
data MAC ℓ τ

   -- Monadic structure for computations
instance Monad (MAC ℓ)

   -- Core operations
label    :: ℓL ⊑ ℓH ⇒ τ → MAC ℓL (Labeled ℓH τ)
unlabel  :: ℓL ⊑ ℓH ⇒ Labeled ℓL τ → MAC ℓH τ
fork^MAC :: ℓL ⊑ ℓH ⇒ MAC ℓH () → MAC ℓL ()
```

Fig. 2: Core API for **MAC**

and Section 6 provides the corresponding security guarantees. Related work is given in Section 7 and Section 8 concludes.

2 The MAC Library

To set the stage of the work at hand, we briefly introduce the relevant aspects of the **MAC** IFC library [40].

Security lattice The sensitivity of data is indicated by labels. These are partially ordered by $\sqsubseteq$ and form a security lattice [12]. Concretely, $\ell_1 \sqsubseteq \ell_2$ holds if data labeled with label ℓ_1 is allowed to flow to entities labeled with ℓ_2. Although **MAC** is parameterized on the security lattice, for simplicity we focus on the classic two-point lattice where the label H denotes secret (high) data, the label L denotes public (low) data, and $H \not\sqsubseteq L$ is the only disallowed flow. In **MAC**, each label is represented as an abstract data type. To improve readability, subscripts on label metavariables hint at their relationship, e.g., if ℓ_L and ℓ_H appear together, then $\ell_L \sqsubseteq \ell_H$ holds.

Security Types Figure 2 shows the core of **MAC**'s API. The abstract type $Labeled\ \ell\ \tau$ classifies data of type τ with a security label ℓ. For example, $creditCard :: Labeled\ H\ Int$ represents a sensitive integer, while $weather :: Labeled\ L\ String$ is a public string. The abstract type $MAC\ \ell\ \tau$ denotes a (possibly) side-effectful secure computation which handles information at sensitivity level ℓ and yields a value of type τ as a result. Importantly, a $MAC\ \ell\ \tau$ computation enjoys a monadic structure, i.e., it is built by the two fundamental operations $return :: \tau \to MAC\ \ell\ \tau$ and $(\gg\!=) :: MAC\ \ell\ \tau \to (\tau \to MAC\ \ell\ \tau') \to MAC\ \ell\ \tau'$ (called "bind"). The operation $return\ x$ produces a computation that returns the value denoted by x without causing side-effects. The function $(\gg\!=)$ is used to *sequence* computations and their corresponding side-effects. Specifically, $m \gg\!= f$ takes the *result* of running the computation m and passes it to the function f, which then returns a second computation to run.

```
impl :: Labeled H Bool → MAC H (Labeled L Bool)
impl secret = do
  bool ← unlabel secret
  if bool then label True
          else label False
```

Fig. 4: Implicit flows are ill-typed ($H \not\sqsubseteq L$).

Haskell provides syntax sugar for monadic computations known as **do**-notation. For instance, the program $m \gg\!= \lambda x \rightarrow return\ (x + 1)$, which adds 1 to the value produced by m, can be written as shown in Figure 3.

```
do x ← m
   return (x + 1)
```

Fig. 3: **do**-notation

Flows of information Abstractly, the side-effects of a $MAC\ \ell\ \tau$ computation involve either reading or writing data. We need to ensure that these actions respect the flows of information that are permitted by the security lattice. The functions *label* and *unlabel* allow $MAC\ \ell\ \tau$ computations to securely interact with labeled expressions, which are the simplest kind of resources available in **MAC**. If a $MAC\ \ell_{\mathrm{L}}$ computation writes data into a sink, the computation needs to have at most the sensitivity of the sink itself. This restriction, known as *no write-down* [5], preserves the sensitivity of data handled by the $MAC\ \ell_{\mathrm{L}}$-computation. The function *label* creates a fresh, labeled value. From the security point of view, this action corresponds to allocating a fresh location in memory and immediately writing a value into it—hence the no write-down principle applies. The type signature of *label* has a *type constraint* before the symbol $\Rightarrow$, which is a property that types must follow. The constraint $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$ ensures that, when calling $label\ x$, the level of the computation ℓ_{L} is no more confidential than the sensitivity ℓ_{H} of the labeled value that it creates. In contrast, a computation $MAC\ \ell_{\mathrm{H}}\ \tau$ is only allowed to read labeled values at most as sensitive as ℓ_{H}. This restriction is known as *no read-up* [5] and gets enforced by the constraint $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$ in the type signature of *unlabel*. This paper focuses on labeled expression, but **MAC** provides additional side-effecting primitives for exception handling, network communication, references, and synchronization primitives [40].

Implicit flows The interaction between the type of a $MAC\ \ell$-computation and the no write-down restriction makes an implicit flow ill-typed. Figure 4 shows a program that attempts to *implicitly* leak a Boolean secret, which is correctly rejected by the compiler. In order to branch on sensitive data, a program needs first to unlabel it, which forces the computation to be of type $MAC\ H\ \tau$, for some type τ. Regardless of which branch is taken, the computation is at level H and cannot therefore write into public data due to the *no write-down* restriction. Trying to do so, as shown in Figure 4, incurs in a violation of the security policy

Types: $\tau ::= () \mid \tau_1 \to \tau_2$
Values: $v ::= () \mid \lambda x.t$
Terms: $t ::= v \mid x \mid t_1\ t_2$
Stacks: $S ::= [\,] \mid C : S$
Continuations: $C ::= x \mid \#x$

$$(\textsc{App}_1)\quad \frac{\text{fresh}(x)}{(\Delta, t_1\ t_2, S) \rightsquigarrow (\Delta[x \mapsto t_2], t_1, x : S)}$$

$$(\textsc{App}_2)\quad (\Delta, \lambda y.t, x : S) \rightsquigarrow (\Delta, t\ [x\ /\ y], S)$$

$$(\textsc{Var}_1)\quad (\Delta[x \mapsto t], x, S) \rightsquigarrow (\Delta, t, \#x : S)$$

$$(\textsc{Var}_2)\quad (\Delta, v, \#x : S) \rightsquigarrow (\Delta[x \mapsto v], v, S)$$

Fig. 5: Syntax and semantics à la Sestoft

and a type error! Observe that the application of *label* is rejected since its type constraint cannot be satisfied, i.e., $H \not\sqsubseteq L$.

Concurrency The mere possibility to run (conceptually) simultaneous $MAC\ \ell$ computations provides attackers with new tools to bypass security checks. In particular, threads introduce the *internal timing covert channel* described in the introduction. Furthermore, it considerably magnifies the bandwidth of the termination covert channel, where secrets are learned by observing the terminating behavior of threads [47]. To securely support concurrency, **MAC** forces programmers to decouple computations which depend on sensitive data from those performing public side-effects. In this manner, non-terminating loops based on secrets cannot affect the outcome of public events. In this light, the type signature of $fork^{\mathsf{MAC}}$ in Figure 2 only allows spawning threads, i.e., a secure computation with type $MAC\ \ell_{\mathrm{H}}\ ()$, which are at least as sensitive as the current computation, i.e., $MAC\ \ell_{\mathrm{L}}\ ()$. It is secure to do so because that decision depends on less sensitive data ($\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$).

3 Lazy Calculus

In order to rigorously analyze the information leaks introduced by sharing, we need to build on top of a formal semantics that is operationally precise enough to make sharing observable. The default choice for such a semantics is Launchbury's "Natural Semantics for lazy evaluation" [21], where the structure of the heap is explicit and sharing, as well as cyclic data structures, are manifestly visible. The heap is a partial map from names to terms. This representation is still more abstract than other formalisations such as the Spineless Tagless G-machine (STG) [32], which concerns itself with pointers and memory representation, and is the basis of the Haskell implementation GHC [25]. That much operational detail would only clutter this work, and in terms of lazy evaluation, Launchbury's semantics is a suitable model of the actual implementation.

This work will have to address concurrency, for which a big-step semantics such as Launchbury's is unsuitable for. Therefore, we build on Sestoft's rendering of Launchbury's semantics as an abstract machine with small-step semantics [44]. Here, a judgement $(\Delta, t, S) \rightsquigarrow (\Delta', t', S')$ indicates that a configuration consisting of a current expression t, a heap Δ, and a stack S takes one step to the configuration on the right hand side of the arrow.

The rules in Figure 5 describe the transitions of the abstract machine for the standard syntactical constructs. Rule ($\textsc{App}_1$) initiates a function call. Since we work in a lazy setting, the function argument t_2 is not evaluated at this point. Instead, it is stored on the heap as a *thunk*, i.e., an unevaluated expression, under a fresh name x with regard to the whole configuration—which corresponds to allocating memory. The machine proceeds to evaluate the function expression t_1 to a lambda expression. Then, rule ($\textsc{App}_2$) takes over and substitutes the name of the argument x, which is found on the stack, into the body t of the lambda expression. If the evaluation of t required the value of x, the machine would get stuck.

Rule ($\textsc{Var}_1$) finds the corresponding thunk t on the heap and, after leaving an *update marker* $\#x$ on the stack, begins to evalute the thunk—intuitively, this marker indicates that when the evaluation of the current term finishes, the denoted value gets stored in x. During evaluation, x is removed from the heap. If the evaluation of t required the value of x, then the machine would get stuck. This effect is desired: if the binding for x were to remain on the heap, evaluation would simply start to run in circles. Removing the variable from the heap, a technique called *blackholing*, makes this error condition detectable. When the machine reduces the thunk to a value v, rule ($\textsc{Var}_2$) pops the update marker from the stack and puts x back on the heap, but now referencing to the value v. Every future use of x will use v directly instead of re-calculating it. This *updating* operation is the crucial step to implement sharing behavior.

We simplified Sestoft's presentation of the semantics in a few ways to remove aspects not relevant for the discussion at hand and to facilitate our machine-checked proofs in Agda: *i)* our syntax does not include mutual recursive **let** expressions; *ii)* in contrast to Sestoft and Launchbury, we allow non-trivial arguments in function application, i.e., our terms are not necessarily in Administrative Normal Form (ANF). In that manner, a non-recursive let expression such as $\mathbf{let}\ x = t_1\ \mathbf{in}\ t_2$ can be expressed as $(\lambda x.t_2)\ t_1$; *iii)* although omitted in this presentation, our formalism sports types with multiple values (e.g., Boolean expressions) and the corresponding case-analysis clause (e.g., $\mathbf{if\text{-}then\text{-}else}$) by using the rules found in [8].

3.1 Security Primitives

We now extend this standard calculus with the security primitives of **MAC** as shown in Figure 6. The new type $\mathit{Labeled}\ \ell\ \tau$ consists of pure values $t :: \tau$ wrapped in $\mathit{Labeled}$, and annotates them with the security level ℓ. We call $\mathit{Labeled}\ 42 :: \mathit{Labeled}\ \ell\ \mathit{Int}$ a pure, side-effect free labeled-ℓ resource with

$$
\begin{array}{l}
\ell \\
\tau ::= \cdots \mid \mathit{MAC}\ \ell\ \tau \mid \mathit{Labeled}\ \ell\ \tau \\
v ::= \cdots \mid \mathit{return}\ t \mid \mathit{Labeled}\ t \\
t ::= \cdots \mid t_1 \gg\!= t_2 \mid \mathit{label}\ t \mid \mathit{unlabel}\ t \\
C ::= \cdots\ x \mid \gg\!= t \mid \mathit{unlabel}
\end{array}
$$

$$
\frac{}{(\Delta, \mathit{label}\ t, S) \rightsquigarrow (\Delta, \mathit{return}\ (\mathit{Labeled}\ t), S)}\ (\text{LABEL})
$$

$$
\frac{}{(\Delta, \mathit{unlabel}\ t, S) \rightsquigarrow (\Delta, t, \mathit{unlabel} : S)}\ (\text{UNLABEL}_1)
$$

$$
\frac{}{(\Delta, \mathit{Labeled}\ t, \mathit{unlabel} : S) \rightsquigarrow (\Delta, \mathit{return}\ t, S)}\ (\text{UNLABEL}_2)
$$

$$
\frac{}{(\Delta, t_1 \gg\!= t_2, S) \rightsquigarrow (\Delta, t_1, \gg\!= t_2 : S)}\ (\text{BIND}_1)
\qquad
\frac{}{(\Delta, \mathit{return}\ t, \gg\!= t_2 : S) \rightsquigarrow (\Delta, t_2\ t, S)}\ (\text{BIND}_2)
$$

Fig. 6: Security primitives

content 42. We introduce a further form of labeled resource, namely references, in the next section. The semantics rules in Figure 6 are fairly straight-forward and follow the pattern seen in Figure 5. It is worth noting that thanks to the static nature of **MAC**, no run-time checks are needed to prevent insecure flows of information in these rules.

We remark that the constructor $\mathit{Labeled}$ is not available to the user, who can only use label ($\mathit{unlabel}$) to create (inspect) labeled resources. Besides these primitives, the user can create computations using the standard monad operations return and $\gg\!=$.

The actual **MAC** implementation knows even more labeled resources (e.g., network) [40]. **MAC** requires no modification to Haskell's type system in order to handle labels: each label is defined as an *empty type*, i.e., a type that has no value, and labeled resources (type $\mathit{Labeled}$) use labels as *phantom types*, i.e., a type parameter that only carries the sensitivity of data at the type-level.

3.2 References

We now extend the abstract machine with mutable references, a feature available in **MAC** to boost the performance of secure programs [40]. References live in the *memory* M, which is simply a list of variables, added as a component of the program configuration—see Figure 7. The address of a memory cell is its index in this list. The memory $M[n \mapsto x]$ is M with its n-th cell changed to refer to x. Observe that the memory M and the heap Δ are two distinct syntactic categories and that, while the latter contains arbitrary terms and enjoys sharing, the former merely contains pointers to the heap. A *labeled* reference is represented as a value $\mathit{Ref}\ n :: \mathit{Ref}\ \ell\ \tau$ where n is the address of the n-th memory cell, which contains a variable (a "pointer") to some term $t :: \tau$ on

$$
\begin{array}{lll}
\text{Configuration:} & c & ::= \langle M, \Delta, t, S\rangle \\
\text{Memory:} & M & ::= [\,] \mid x : M \\
\text{Addresses:} & n & ::= 0 \mid 1 \mid 2 \mid \cdots \\
\text{Types:} & \tau & ::= \cdots \mid \mathit{Ref}\ \ell\ \tau \\
\text{Values:} & v & ::= \cdots \mid \mathit{Ref}\ n \\
\text{Terms:} & t & ::= \cdots \mid \mathit{new}\ t \mid \mathit{read}\ t \\
 & & \quad \mid \mathit{write}\ t_1\ t_2 \\
\text{Continuations:} & C & ::= \cdots \mid \mathit{read} \mid \mathit{write}\ t
\end{array}
$$

$$\text{(LIFT)}\quad \frac{(\Delta, t, S) \rightsquigarrow (\Delta', t', S')}{\langle M, \Delta, t, S\rangle \longrightarrow \langle M, \Delta', t', S'\rangle}$$

$$\text{(NEW)}\quad \frac{n = |M| \qquad \text{fresh}(x)}{\langle M, \Delta, \mathit{new}\ t, S\rangle \longrightarrow \langle M[n \mapsto x], \Delta[x \mapsto t], \mathit{return}\ (\mathit{Ref}\ n), S\rangle}$$

$$\text{(WRITE}_1\text{)}\quad \langle M, \Delta, \mathit{write}\ t_1\ t_2, S\rangle \longrightarrow \langle M, \Delta, t_1, \mathit{write}\ t_2 : S\rangle$$

$$\text{(WRITE}_2\text{)}\quad \frac{\text{fresh}(x)}{\langle M, \Delta, \mathit{Ref}\ n, \mathit{write}\ t : S\rangle \longrightarrow \langle M[n \mapsto x], \Delta[x \mapsto t], \mathit{return}\ (), S\rangle}$$

$$\text{(READ}_1\text{)}\quad \langle M, \Delta, \mathit{read}\ t, S\rangle \longrightarrow \langle M, \Delta, t, \mathit{read} : S\rangle$$

$$\text{(READ}_2\text{)}\quad \langle M, \Delta, \mathit{Ref}\ n, \mathit{read} : S\rangle \longrightarrow \langle M, \Delta, \mathit{return}\ M[n], S\rangle$$

Fig. 7: Syntax and semantics for references

the heap[22]. Only secure computations can manipulate these *labeled references* using the secure primitives in Figure 8. Observe that the types are restricted according to the *no read-up* and *no write-down* restrictions—like those of label and $\mathit{unlabel}$.

The extended semantics is represented as the relation $c \longrightarrow c'$ which extends $\rightsquigarrow$ via [LIFT]—see Figure 7. Rule [NEW] allocates the second argument on the heap with a fresh name x, extends the memory with a new pointer to x and returns a reference to it. Rule [WRITE$_1$] evalutes its first argument to a reference and rule [WRITE$_2$] overrides the memory cell with a pointer to a newly allocated heap entry, just like new. The two [READ]-rules retrieve a pointer from memory. To the best of our knowledge, this is the first published operational semantics that models both lazy evaluation and mutable references, and

[22] **MAC**'s implementation of labeled reference is a simple wrapper around Haskell's type IORef. However, we denote references as a simple index into the labeled memory. This design choice does not affect our results.

$$
\begin{aligned}
&\mathit{new} &&:: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow \tau \to \mathit{MAC}\ \ell_{\mathrm{L}}\ (\mathit{Ref}\ \ell_{\mathrm{H}}\ \tau)\\
&\mathit{read} &&:: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow \mathit{Ref}\ \ell_{\mathrm{L}}\ \tau \to \mathit{MAC}\ \ell_{\mathrm{H}}\ \tau\\
&\mathit{write} &&:: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow \tau \to \mathit{Ref}\ \ell_{\mathrm{H}}\ \tau \to \mathit{MAC}\ \ell_{\mathrm{L}}\ ()
\end{aligned}
$$

Fig. 8: API of memory operations

although we constructed it using standard techniques, we would like to point out a crucial subtlety.

A naïve model might omit the extra memory M, let a reference simply contain a variable on the heap ($t ::= \cdots \mid \mathit{Ref}\ x$) and use the transition rule $\langle \Delta, \mathit{Ref}\ x, \mathit{write}\ t : S\rangle \longrightarrow \langle \Delta[x \mapsto t], \mathit{return}\ (), S\rangle$. This transition interacts badly with sharing, as shown by the program in Figure 9. Clearly, we expect the program to return "☺", but it returns "☹" with the naïve semantics! The new statement allocates a new variable x, binds it to $1+1$, and returns the reference $\mathit{Ref}\ x$. The next read statement brings variable x into scope, which is pure and we expect its denoted value to stay the same. However, under the naïve semantics, the following write statement changes x to 1 and therefore chaos ensues.

```
do r ← new (1 + 1)
   x ← read r
   write r 1
   if (x ≡ 2) then return "☺"
              else return "☹"
```

Fig. 9: Immutability

The solution is to add an extra layer of indirection, and distinguish between the mutable memory cells that make up a reference, and the heap locations that—although changed in [VAR$_2$]—are conceptually constant. We chose to keep track of them separately in the memory M and the heap Δ since we found that it makes formal reasoning easier. It is also viable to keep both on the heap, and just be disciplined as to which variables denote references and which denote values and thunks—this design choice would be closer to GHC's runtime, where both pure data and mutable references are addressed by pointers into a single heap.

3.3 Concurrency

Finally, we extend our language with concurrency in the form of threads whose execution interleave[23]. We consider *global configurations* of the form $\langle M, \Delta, T_s\rangle$, where thread pool T_s consists of a list of threads—see Figure 10. A thread (t, S) is an *interrupted* secure computation, consisting of a control term t and a stack S. Within

Thread pool:	T_s	$::= [\,] \mid (t, S) : T_s$
Configuration:	c	$::= \langle M, \Delta, T_s\rangle$
Terms:	t	$::= \cdots \mid \mathit{fork}\ t$

Fig. 10: Concurrent calculus

[23] **MAC** provides also synchronization variables [40], which we omit here.

$$\text{(SEQ)}\quad \frac{\langle M_1, \Delta_1, t_1, S_1\rangle \longrightarrow \langle M_2, \Delta_2, t_2, S_2\rangle}{\langle M_1, \Delta_1, T_s[n \mapsto (t_1, S_1)]\rangle \hookrightarrow \langle M_2, \Delta_2, T_s[n \mapsto (t_2, S_2)]\rangle}$$

$$\text{(FORK)}\quad \frac{m = |T_s|}{\langle M, \Delta, T_s[n \mapsto (\mathit{fork}\ t, S)]\rangle \hookrightarrow \langle M, \Delta, T_s[n \mapsto (\mathit{return}\ (), S)][m \mapsto (t, [\,])]\rangle}$$

Fig. 11: Semantics of the concurrent calculus

a global configuration, threads are identified by their position in the thread pool. For simplicity and brevity, the concurrent calculus features a Round Robin scheduler, the same kind of scheduler used by GHC's run-time system[24]—however, our results and semantics generalize to a wide range of deterministic schedulers. In the following, we omit the scheduler from the configuration and from the semantics rules for space reasons.

Figure 11 describes the two rules under which a global configuration c_1 steps to c_2 (written $c_1 \hookrightarrow c_2$). In both rules ([SEQ] and [FORK]), thread n is executed—the scheduler actually *deterministically* chooses which thread to run, which is retrieved from the thread pool T_s. In rule [SEQ], the selected thread, i.e., (t_1, S_1), takes a sequential step that is paired with the current memory and heap: $(\langle M_1, \Delta_1, t_1, S_1\rangle \longrightarrow \langle M_2, \Delta_2, t_2, S_2\rangle)$.

The global configuration is then updated accordingly to the final sequential configuration, in particular, the thread pool is updated with the reduced thread, i.e., $T_s[n \mapsto (t_2, S_2)]$. In rule [FORK], the selected thread spawns a thread—note that term $\mathit{fork}\ t$ is stuck in the sequential semantics and rule [SEQ] does not apply. The new thread is assigned the fresh identifier $m = |T_s|$—thread pool T_s contains threads $0 \ldots |T_s| - 1$. Lastly, the thread pool is updated with the parent thread, appropriately reduced to $(\mathit{return}\ (), S)$, and by inserting the new thread initialized with an empty stack, i.e., $(t, [\,])$, at position m. Note that a thread that tries to evaluate a variable x that is already under evaluation by another thread will not find this variable on the heap, due to the blackholing mechanism explained earlier. The thread is now blocked, guaranteeing that, even in the concurrent setting, every thunk will only be evaluated at most once. This mechanism is consistent with the operational semantics used by Finch et al. [3].

4 Duplicating Thunks

This section presents one of our main contributions: a primitive, called *lazyDup* that prevents sharing. Given a term t, evaluating $\mathit{lazyDup}\ t$ will *lazily* create a copy of t. The laziness is necessary in order to duplicate cyclic or infinite

[24] `https://ghc.haskell.org/trac/ghc/wiki/Commentary/Rts/Scheduler`

Terms: $t ::= \cdots \mid \mathit{lazyDup}\ t$

$$\text{(LAZYDUP}_1\text{)}\quad \frac{\neg\ \mathit{isVar}\ (t) \qquad \text{fresh}(x)}{\langle \Delta, \mathit{lazyDup}\ t, S\rangle \rightsquigarrow \langle \Delta[x \mapsto t], \mathit{lazyDup}\ x, S\rangle}$$

$$\text{(LAZYDUP}_2\text{)}\quad \frac{x \mapsto t\ \in\ \Delta \qquad \text{fresh}(x')}{\langle \Delta, \mathit{lazyDup}\ x, S\rangle \rightsquigarrow \langle \Delta[x' \mapsto [\![t]\!]^{\varnothing}], x', S\rangle}$$

$$\begin{aligned}
[\![t_1\ t_2]\!]^B &= [\![t_1]\!]^B\ [\![t_2]\!]^B \\
[\![\lambda x.t]\!]^B &= \lambda x.[\![t]\!]^{B\cup\{x\}} \\
[\![()]\!]^B &= () \\
[\![x]\!]^B &= \begin{cases} x & \text{if } x\ \in\ B \\ \mathit{lazyDup}\ x & \text{if } x\ \notin\ B \end{cases} \\
[\![\mathit{lazyDup}\ t]\!]^B &= \mathit{lazyDup}\ t
\end{aligned}$$

Fig. 12: Synatx and semantics of $\mathit{lazyDup}$

data structures without sending the program into a loop. In this paper, we lie the formal ground of $\mathit{lazyDup}$ and the security guarantees that it provides. We speculate that an implementation for GHC is feasible based on its internals and how $\mathit{lazyDup}$ has been conceived. We first present the basic semantics of $\mathit{lazyDup}$ and then describe how we handle references.

4.1 Semantics

Figure 12 extends the syntax and the semantics of the calculus with $\mathit{lazyDup}$. The rule [LAZYDUP$_1$] ensures that the argument of $\mathit{lazyDup}$ is a variable, if that is not already the case. The interesting rule is [LAZYDUP$_2$], which evaluates $\mathit{lazyDup}\ x$ and copies the expression t referenced by x. This closes the covert channel represented by x, but is insufficient, as t might mention further variables. Therefore, $\mathit{lazyDup}$ has to descent into t, and handle these as well. But instead of immediately duplicating the terms referenced by those variables, we simply wrap them in a call to $\mathit{lazyDup}$—this is the eponymous laziness.

Figure 12 shows (some of) the equations of the function $[\![t]\!]^B$ which implements this. It homomorphically traverses the tree t, while keeping track of the set of bound variables in its parameter B. Ground values and bound variables are left alone. When $\mathit{lazyDup}$ finds a free variable, i.e., one not in B, it wraps it with a call to $\mathit{lazyDup}$ as intended. In the following we omit the superscript B, when irrelevant. Finally, if $[\![\cdot]\!]$ comes across a call to $\mathit{lazyDup}$, it does not traverse further, as the existing $\mathit{lazyDup}$ already takes care of the duplication. In fact, without this case, evaluating expression $\mathit{lazyDup}\ (\mathit{lazyDup}\ t)$ would send the program into an infinite loop. We conjecture that introducting $\mathit{lazyDup}$ does

Values: $v ::= \cdots \mid \mathit{DRef}\ m$

$[\![\mathit{Ref}\ n]\!]^B = \mathit{DRef}\ n$
$[\![\mathit{DRef}\ n]\!]^B = \mathit{DRef}\ n$

(READ^D)
$\langle M, \Delta, \mathit{DRef}\ n, \mathit{read} : S\rangle \longrightarrow \langle M, \Delta, \mathit{return}\ (\mathit{lazyDup}\ M[n], S)\rangle$

(WRITE^D)
$$\frac{\text{fresh}(x)}{\langle M, \Delta, \mathit{DRef}\ n, \mathit{write}\ t : S\rangle \longrightarrow \langle M[n \mapsto x], \Delta[x \mapsto t], \mathit{return}\ (), S\rangle}$$

Fig. 13: Duplicate-on-read memory operations

(FORK)
$$\frac{|T_s| = m}{\langle M, \Delta, T_s[n \mapsto (\mathit{fork}\ t, S)]\rangle \hookrightarrow \langle M, \Delta, T_s[n \mapsto (\mathit{return}\ (), S)][m \mapsto (\ \mathit{lazyDup}\ \ t, [])]\rangle}$$

Fig. 14: Patch needed to secure **MAC**

not change the termination behavior. Note that the term $[\![t]\!]$ has at most one call to $\mathit{lazyDup}$ wrapped around each free variable.

4.2 References

Duplicating references requires particular care. To illustrate this, consider what does not work. We cannot leave references alone ($[\![\mathit{Ref}\ n]\!] = \mathit{Ref}\ n$) because thunks can be passed through the reference and open a new leaking channel. a We cannot either duplicate the reference and the term it currently references since this would change the semantics of mutable references. More concretely, consider a $\mathit{Ref}\ n$ with $M[n] = x$ and $\Delta(x) = t$. Assume we duplicate t to $\Delta(y) = [\![t]\!]$ for a fresh name y and let $[\![\mathit{Ref}\ n]\!] = \mathit{Ref}\ n'$ for a fresh memory cell n', such that $M[n' \mapsto y]$. If later $\mathit{Ref}\ n$ gets updated with the value 42, i.e., $M[n \mapsto z]$ with $\Delta(z) = 42$, then this change would be invisible to $\mathit{Ref}\ n'$, which would still refer to $[\![t]\!]$ through variable y. This is bad, as $\mathit{lazyDup}$ is not supposed to change the observable semantics of the program!

Crucially, we need to propagate any write operation on the original reference to the duplicated reference. One manner to achieve that is to have both references pointing to the same memory location but carefully preventing leaks from reading this shared resource. In this light, we introduce a new variant of reference, called *duplicate-on-read reference*[25], which is represented by $\mathit{DRef}\ n$. When reading from a $\mathit{DRef}\ n$, we wrap the read value in a call to $\mathit{lazyDup}$, as

[25] The same approach applies to synchronization variables.

shown in Figure 13, while write operations on a duplicate-on-read reference are executed as usual. Function $[\![.]\!]$ does not need to follow references and duplicate their content, but simply turns them into duplicate-on-read-references. In this sense, we apply $lazyDup$ lazily to reference: the duplication is suspended and continues when the reference is read.

5 Securing MAC

We now pinpoint the vulnerability leveraged by the attack sketched in the introduction and show how to modify **MAC** to close it using $lazyDup$. It turns out that one careful modification to the [FORK] rule suffices. This change, highlighted in greeen in Figure 14, ensures that when we create a new thread to evaluate t, it will work on a lazily duplicated copy of t, i.e., $lazyDup\ t$. As a result, each thunk shared between the parent and the child thread gets lazily duplicated: the parent thread works on the original thunk, while the child thread works on a copy.[26]

Let us trace how this fixes the leak shown in Figure 1. Let t be the code of the secret thread. When it is spawned, $lazyDup\ t$ is added to the thread pool. Note that the critical resource that causes the leak, namely variable r, is a free variable of t. As the secret thread executes $lazyDup\ t$, the occurrence of r in the code is replaced by $lazyDup\ r$ (rule [LAZYDUP$_2$]). Therefore, if $s \equiv 1$, the thread duplicates r before evaluating it, leaving r itself alone, just like when $s \equiv 0$ and the secret thread does not touch r at all. As a result, the timing behavior of public threads, i.e., the order with which they output a message on the public channel, is unaffected by the value of the secret s and the internal timing leak is closed.

Observe that $lazyDup$ *conservatively* avoids sharing between secret and public threads. In principle, it is acceptable for a secret thread to evaluate and update a thunk if that action does not depend on the secret—for example if that happens before any sensitive command such as $unlabel$. Assessing whether this is the case requires sophisticated program analysis techniques, which are beyond the scope of this paper. On the other hand, sharing from public to secret threads is always secure, and in fact $lazyDup$ allows for this "write-up" behavior: if, due to lucky scheduling, the public thread finishes evaluating r before the secret thread looks at it, then the latter will see the fully evaluated term and securily enjoy the benefits of sharing.

The primitive $lazyDup$ is capable of securing **LIO** as well even though it has to be used differently—see Appendix A for more details.

[26] It is secure to avoid duplication whenever the parent and the child thread share the same security level, which are both statically known in **MAC**, see Figure 2. Since the label of the child thread (ℓ_H) is at least as sensitive as that of the parent, i.e., $\ell_L \sqsubseteq \ell_H$, we only have to use $lazyDup$ if $\ell_L \sqsubset \ell_H$.

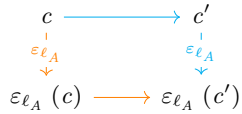

Fig. 15: Single-step simulation

6 Security Guarantees

In this section, we show that our calculus enforces *progress sensitive non-interference* (PSNI). We start by describing our proof technique, based on *term erasure*. To facilitate reasoning, we proceed to decorate our calculus with labels that keep track of the security level of terms stored in memory, heaps and configurations. We then prove PSNI for the decorated calculus and conclude that **MAC** is likewise secure by establishing a mutual simulation relation with the vanilla (undecorated) calculus.

6.1 Term Erasure

Term erasure is a technique to prove non-interference in functional programs [23] and IFC libraries (e.g., [10, 18, 47–50]). It relies on an erasure function, which we denote by ε_{ℓ_A}. This function rewrites data above the attacker's security level, denoted by label ℓ_A, to the special syntactic construct $\bullet$. At the core, this technique establishes a simulation between reductions of configurations and reductions of their erased counterparts. Figure 15 shows that erasing sensitive data from a configuration c and then taking a step (orange path) yields the same configuration as first taking a step and then erasing sensitive data (cyan path), i.e., the diagram *commutes*. If the configuration c were to leak sensitive data into a non-sensitive resource, then it will remain in $\varepsilon_{\ell_A}(c')$. The same data would be erased in $\varepsilon_{\ell_A}(c)$ and the diagram would not commute.

6.2 Decorated Calculus

The erasure proof technique was conceived to work on dynamic IFC approaches [23], where security labels are attached to terms. Applying term erasure to **MAC**, where labels are parts of the types instead of the terms, demands to extend our calculus with extra information about the sensitivity nature of terms. As in similar work [49, 50], we annotate terms with their type and make the erasure function *type-driven*. The annotated term $t :: \tau$ denotes that the term t has type τ. We likewise decorate configurations, heaps, memories, stacks, and continuations with labels.

Figure 16 summarizes the main changes required to decorate our calculus.

Pure conf. ℓ:	c	$::= (\Delta^\ell, t, S^\ell)$
Seq. conf. ℓ:	c	$::= \langle \Sigma, \Gamma, t, S^\ell \rangle$
Heap map:	Γ	$::= (\ell : Label) \to Heap\ \ell$
Store:	Σ	$::= (\ell : Label) \to Memory\ \ell$
Memory ℓ_H:	M	$::= \cdots \mid x^{\ell_L} : M$
Terms τ:	t	$::= \cdots \mid x^\ell$
Cont. ℓ:	C	$::= \cdots \mid x^\ell \mid \#x^\ell$
Conc. conf.:	c	$::= \langle \Sigma, \Gamma, \Phi \rangle$
Pool map:	Φ	$::= (\ell : Label) \to Pool\ \ell$

Fig. 16: Decorated Calculus

A pure configuration labeled with ℓ, i.e., $\langle \Delta^\ell, t, S^\ell \rangle$, consists of a labeled heap Δ^ℓ, and a labeled stack S^ℓ. An ℓ-labeled heap Δ^ℓ can be accessed by ℓ-labeled variables, e.g., x^ℓ. An ℓ-labeled stack contains exclusively ℓ-labeled continuations, which involve only ℓ-labeled variables, i.e., continuations x^ℓ and $\#x^\ell$. Furthermore an ℓ-labeled heap contains terms that can be evaluated only by threads at level ℓ. A sequential configuration $\langle \Sigma, \Gamma, t, S^\ell \rangle$ labeled with ℓ, consists of a store Σ, a current term t, an heap map Γ, and a labeled stack S^ℓ. An ℓ-labeled configuration denotes a computation of type $MAC\ \ell\ \tau$, for some type τ. Note that this does not necessarily mean that term t is a MAC computation—when the configuration steps the current term is changed with the next redex, which might have a completely different type. Instead, the combination of current term *and* stack guarantees that the whole configuration represents a $MAC\ \ell$ computation.

It is known that dealing with dynamic allocation of memory makes it challenging to prove non-interference (e.g., [4,17]). One manner to tackle this technicality is by establishing a bijection between public memory addresses of the two executions we want to relate and considering equality of public terms up to such notion [4]. Instead, and similar to other work [18,48–50], we compartmentalize the memory into isolated labeled segments, one for each label of the lattice. This way, allocation in one segment does not affect the others. A similar argument holds for the heap and the thread pool, which we therefore also organize in partitions, accessed through the heap map Γ respectively the pool map Φ. Since we now have multiple heaps in one configuration, we need to annotate the *free* variables with the label of the heap in which they are bound. So a variable x^ℓ denotes that x is bound in the heap $\Gamma(\ell)$. Variables bound inside a term remain unlabeled, e.g., $\lambda x.x$. A variable x^{ℓ_L} in a ℓ_H-labeled memory will have a label of at most the memory's sensitivity, $\ell_L \sqsubseteq \ell_H$. Unlike variables, we do not need to annotate memory cells n, as they only occur in a $Ref\ n$ expression, which carries a label in its type. So a reference $Ref\ n :: Ref\ \ell\ \tau$ points to the n-th entry in the ℓ-labeled memory. In the following, we write $\mathrm{fresh}(x^\ell)$ to denote that variable x is fresh with respect to heap $\Gamma(\ell) = \Delta^\ell$ and stack S^ℓ. We write $\Gamma[\ell][x^\ell] := t$ for the heap map obtained by performing the update $\Gamma(\ell)[x^\ell \mapsto t]$, and likewise for stores and pool maps.

6.3 Decorated Semantics

The interesting rules of the annotated semantics are shown in Figure 17. The rules for the pure fragment of the calculus are adapted to work with labeled

$$\text{(App}_1\text{)}\quad \frac{\text{fresh}(x^\ell)}{(\Delta^\ell, t_1\ t_2, S^\ell) \rightsquigarrow (\Delta^\ell[x^\ell \mapsto t_2], t_1, x^\ell : S^\ell)}$$

$$\text{(App}_2\text{)}\quad (\Delta^\ell, \lambda y.t, x^\ell : S^\ell) \rightsquigarrow (\Delta^\ell, t\ [x^\ell\ /\ y], S^\ell)$$

$$\text{(Var}_1\text{)}\quad (\Delta^\ell[x^\ell \mapsto t], x^\ell, S^\ell) \rightsquigarrow (\Delta^\ell, t, \#x^\ell : S^\ell)$$

$$\text{(Var}_2\text{)}\quad (\Delta^\ell, v, \#x^\ell : S) \rightsquigarrow (\Delta^\ell[x^\ell \mapsto v], v, S^\ell)$$

$$\text{(Lift)}\quad \frac{(\Gamma(\ell), t_1, S_1^\ell) \rightsquigarrow (\Delta^\ell, t_2, S_2^\ell)}{\langle \Sigma, \Gamma, t_1, S_1^\ell \rangle \longrightarrow \langle \Sigma, \Gamma[\ell \mapsto \Delta^\ell], t_2, S_2^\ell \rangle}$$

$$\text{(LazyDup}_1\text{)}\quad \frac{\neg\ isVar\ (t) \qquad \text{fresh}(x^\ell)}{\langle \Sigma, \Gamma, lazyDup\ t, S^\ell \rangle \longrightarrow \langle \Sigma, \Gamma[\ell][x^\ell] := t, lazyDup\ x^\ell, S^\ell \rangle}$$

$$\text{(LazyDup}_2\text{)}\quad \frac{x^{\ell_L} \mapsto t\ \in\ \Sigma(\ell_L) \qquad \text{fresh}(y^{\ell_H})}{\langle \Sigma, \Gamma, lazyDup\ x^{\ell_L}, S^{\ell_H} \rangle \longrightarrow \langle \Sigma, \Gamma[\ell_H][y^{\ell_H}] := [\![t]\!]^{\varnothing}, y^{\ell_H}, S^{\ell_H} \rangle}$$

$$\text{(New)}\quad \frac{|\Sigma(\ell_H)| = n \qquad \text{fresh}(x^{\ell_L})}{\langle \Sigma, \Gamma, new\ t, S^{\ell_L} \rangle \longrightarrow \langle \Sigma[\ell_H][n] := x^{\ell_L}, \Delta[\ell_L][x^{\ell_L}] := t, return\ (Ref\ n), S^{\ell_L} \rangle}$$

$$\text{(Write}_2\text{)}\quad \frac{\text{fresh}(x^{\ell_L})}{\langle \Sigma, \Gamma, Ref\ n, write\ t : S^{\ell_L} \rangle \longrightarrow \langle \Sigma[\ell_H][n] := x^{\ell_L}, \Gamma[\ell_L][x^{\ell_L}] := t, return\ (), S^{\ell_L} \rangle}$$

$$\text{(Read}_2\text{)}\quad \frac{\Sigma(\ell)[n] = x^\ell}{\langle \Sigma, \Gamma, Ref\ n, read : S^\ell \rangle \longrightarrow \langle \Sigma, \Gamma, return\ x^\ell, S^\ell \rangle}$$

$$\text{(Read}^D\text{)}\quad \langle \Sigma, \Gamma, DRef\ n, read : S^{\ell_H} \rangle \longrightarrow \langle \Sigma, \Gamma, return\ (lazyDup\ \Sigma(\ell_L)[n]), S^{\ell_H} \rangle$$

Fig. 17: Decorated Semantics

variables. Note that rule [APP$_2$] replaces the bound, hence unlabeled, variable y with the labeled variable x^{ℓ} and thus maintains the invariant that *free* variables are labeled.

Why do we get away with giving the pure fragment of the annotated calculus only access to the heap $\Gamma(\ell)$ in a configuration at level ℓ? What if the program accesses a variable at a different level ℓ'? Because that cannot happen in a safe program, as the following example shows. Consider the following reduction sequence:

$$
\begin{array}{rlll}
 & ([x^{\ell'} \mapsto t],\ x^{\ell'},\ [\,] &) & \\
\rightsquigarrow & ([\,],\ t,\ \# x^{\ell'} : [\,]) & & \text{-- rule [VAR}_1\text{]} \\
\rightsquigarrow^* & ([\,],\ v,\ \# x^{\ell'} : [\,]) & & \\
\rightsquigarrow & ([x^{\ell'} \mapsto v], v,\ [\,] &) & \text{-- rule [VAR}_2\text{]}
\end{array}
$$

In the first step, the ℓ-labeled configuration *reads* the variable $x^{\ell'}$. According to the *no read-up* security policy, this is only safe if $\ell' \sqsubseteq \ell$. In the last step, the ℓ'-labeled heap entry is updated with the value v. This constitutes a write operation, so accoring to the *no write-down* policy, this requires $\ell \sqsubseteq \ell'$. By the antisymmetry of the security lattice, it follows that $\ell \equiv \ell'$ must hold. So in the presence of *sharing*, a configuration complies with the *no write-down* and *no read-up* security policies only if it interacts soley with the ℓ-labeled heap.

Rule [LIFT] executes a pure reduction step, giving it access to the appropriate heap $\Gamma(\ell)$ and updating the heap map afterwards. Rules [LAZYDUP$_1$] and [LAZYDUP$_2$] adapt the semantics of *lazyDup* to label-partitioned heaps. The first rule takes care of allocating a non-trivial argument on the heap labeled as the current configuration. The second rule is the heart of our security leak fix: it handles the case where a high thread reads a thunk from a lower context. The rule fetches the thunk t from the lower heap, i.e., $t \mapsto \Sigma(\ell_{\mathrm{L}})$, and extends the heap labeled as the configuration with a copy of the thunk, i.e., $\Sigma[\ell_{\mathrm{H}}][y^{\ell_{\mathrm{H}}}] := [\![t]\!]^{\varnothing}$. Observe that this operation relabels the original thunk t from ℓ_{L} to ℓ_{H} securely because t is duplicated, ensuring that the free variables in t will, by the time they are about to be evaluated, be wrapped in *lazyDup*, so that [LAZYDUP$_2$] kicks in again. In rule [NEW], a computation at level ℓ_{L} creates a reference labeled with ℓ_{H}. The thunk t is allocated on the ℓ_{L} heap under the name $x^{\ell_{\mathrm{L}}}$, which is written to the fresh cell in memory $\Sigma(\ell_{\mathrm{H}})$. This ensures the invariant that in *well-typed* configurations a memory holds references to lower heaps. The same applies to rule [WRITE$_2$]. Rule [READ$_2$] enforces that a computation at level ℓ can only read from a non-duplicated reference if the referenced variable is at the same level ℓ. Relaxing this would allow a high thread to read a thunk from a low level and thus open another leaky channel. But the interplay of rule [FORK] (in its annotated variant in Figure 21 in Appendix D), rule [LAZYDUP$_2$] and *lazyDup* rewriting of references to duplicate-on-read references precludes this scenario. Rule [READ^D] then allows a ℓ_{H} high computation to read a low variable from a duplicate-on-read reference, by duplicating it to ensure security.

6.4 Erasure Function

The term $\varepsilon_{\ell_A}(t :: \tau)$ is obtained from a term t with type τ by erasing data not observable by an attacker at level ℓ_A. For clarity, we omit the type annotation when irrelevant or obvious. Ground values (e.g., $()$, $True$) are unaffected by the erasure function. For most syntactic forms, the function recurses homomorphically as in $\varepsilon_{\ell_A}(lazyDup\ t :: \tau) = lazyDup\ (\varepsilon_{\ell_A}(t :: \tau))$. The interesting cases are terms of type $Labeled\ \ell\ \tau$ and $Ref\ \ell\ \tau$. For such cases, the erasure function recurses as usual if $\ell \sqsubseteq \ell_A$. If, however, $\ell \not\sqsubseteq \ell_A$, and the resource is above the attacker's level, then it is erased and replaced by $\bullet$, e.g., $\varepsilon_{\ell_A}(Labeled\ t :: Labeled\ \ell\ \tau) = Labeled\ (\varepsilon_{\ell_A}(t :: \tau))$ if $\ell_A \sqsubseteq \ell$ or $Labeled\ \bullet$ otherwise. The erasure function is described with more detail in Appendix D.

6.5 Decorated Progress-Sensitive Non-Interference

The non-interference proof relies on the two main properties *determinancy* and *simulation*. Determinancy simply states that transitions are deterministic:

Proposition 10 (Determinancy) *If* $c_1 \hookrightarrow c_2$ *and* $c_1 \hookrightarrow c_3$ *then* $c_2 \equiv c_3$.

The equality in this statement is alpha-equality, i.e., up to the choice of variables. In the machine-checked proofs all variables are De Bruijn indexes, and we indeed obtain structural equality.

The choice of determinism makes the concurrent model robust against scheduler refinement attacks. The second property, i.e., *simulation*, says that if a thread steps in a global configuration, then, either the same thread steps in the erased configuration, when the thread's level is visible to the attacker, i.e., $\ell \sqsubseteq \ell_A$, or otherwise, the initial and resulting configuration are indistinguishable to the attacker. We call such indistinguishability relation ℓ_A-equivalence, written $c_1 \approx_{\ell_A} c_2$ and defined as $\varepsilon_{\ell_A}(c_1) \equiv \varepsilon_{\ell_A}(c_2)$. Observe that two ℓ_A-equivalent configurations contain exactly the same number of ℓ_A-equivalent public threads, but possibly a different number of secret threads. The notation $c_1 \hookrightarrow_{(\ell,n)} c_2$ expresses that the configuration c_1 runs the n-th thread at security level ℓ—threads are identified by label and number in the decorated semantics.

Proposition 11 (Simulation) *Given a global reduction step* $c_1 \hookrightarrow_{(\ell,n)} c_1'$ *then*

- $\varepsilon_{\ell_A}(c_1) \hookrightarrow_{(\ell,n)} \varepsilon_{\ell_A}(c_1')$, *if* $\ell \sqsubseteq \ell_A$, *or*
- $c_1 \approx_{\ell_A} c_1'$, *if* $\ell \not\sqsubseteq \ell_A$

From Propositions 10 and 11, we prove progress-sensitive non-interference. Note that, unlike our previous work [49], Proposition 11 does not simulate sensitive threads, because ℓ_A-equivalence suffices for PSNI. For more details, please refer to our Agda formalization[27].

[27] Available at `https://github.com/marco-vassena/lazy-mac`

Theorem 1 (PSNI) *Given two configurations $c_1 \approx_{\ell_A} c_2$ and a reduction $c_1 \hookrightarrow_{(\ell,n)} c_1'$, then there exists a configuration c_2' such that $c_1' \approx_{\ell_A} c_2'$ and $c_2 \hookrightarrow^* c_2'$.*

As usual, $\hookrightarrow^*$ denotes the transitive reflexive closure of $\hookrightarrow$.

Proof 1 *If $\ell \sqsubseteq \ell_A$, by ℓ_A-equivalence, configuration c_2 contains a thread identified by (ℓ, n), that is ℓ_A-equivalent to that run by c_1. However, c_2 might contain a* finite *number of high threads, which are scheduled before that. After running those high threads, i.e., $c_2 \hookrightarrow^* c_2''$, for some configuration c_2'', the same low thread is scheduled, i.e., $c_2'' \hookrightarrow_{(\ell,n)} c_2'$, for some other configuration c_2'. Applying the simulation proposition to the first set of steps yields $c_2 \approx_{\ell_A} c_2''$, as they are all above the attackers level, and by transitivity it follows that $c_1 \approx_{\ell_A} c_2''$, i.e., $\varepsilon_{\ell_A}(c_1) \equiv \varepsilon_{\ell_A}(c_2'')$. Applying simulation again we learn that $\varepsilon_{\ell_A}(c_2'') \hookrightarrow_{(\ell,n)} \varepsilon_{\ell_A}(c_2')$, since $\ell \sqsubseteq \ell_A$ as well as $\varepsilon_{\ell_A}(c_1) \hookrightarrow_{(\ell,n)} \varepsilon_{\ell_A}(c_1')$. The determinancy proposition shows $\varepsilon_{\ell_A}(c_1') \equiv \varepsilon_{\ell_A}(c_2')$ or, in other words, $c_1' \approx_{\ell_A} c_2'$. If $\ell \not\sqsubseteq \ell_A$, then simulation tells us $c_1 \approx_{\ell_A} c_1'$ and $c_2'' \approx_{\ell_A} c_2'$, so we obtain $c_1' \approx_{\ell_A} c_2'$ by transitivity.*

6.6 Simulation between Vanilla and Decorated semantics

To conclude the proof of the security guarantees, we have to relate the decorated semantics with the vanilla semantics. On the one hand, we show that we can strip off the annotations from a decorated program, run it under the vanilla semantics, and get the same behavior as running the decorated program in the decorated semantics. On the other hand, we show that we can annotate a well-typed vanilla program, based on the type derivations, and obtain a decorated program that executes correspondingly.

The main challenge is to map the partitioned heap, memory, and stack in the annotated calculus into a single heap, memory, and stack and vice versa. We apply techniques inspired by other IFC works on dynamic allocation [4] and partitioned heaps [18] and show that configurations in the annotated calculus are equal to those in the vanilla calculus *up to bijection on variables names and memory addresses*. These bijections describe how to flatten the partitioned memories and heaps into single entities without changing the results produced by programs—of course, modulo variable names and memory addresses. Note that our references are opaque to programs, i.e., there is no pointer arithmetic, equality, etc., which makes the proof easier.

We work with two bijections, Ψ_1 for heap variables and Ψ_2 for memory addresses:

$$\Psi_1 :: (Label \times Var) \to Var$$
$$\Psi_2 :: (Label \times \mathbb{N}) \to \mathbb{N}$$

Var is the set of variables and $\mathbb{N}$ the set of memory addresses. When we refer to both bijections, we simply write Ψ.

As one expects, we consider an annotated configuration *equivalent up to bjections* to a vanilla configuration, written $\langle \Sigma, \Gamma, t, S^\ell \rangle \cong_\Psi \langle M, \Delta, t', S \rangle$, if and only if their components are related, i.e., $\Sigma \cong_\Psi M$, $\Gamma \cong_\Psi \Delta$, $S^\ell \cong_\Psi S$, and $t \cong_\Psi t'$. The equivalences on memories ($\Sigma \cong_\Psi M$), heap ($\Gamma \cong_\Psi \Delta$), and stack ($S^\ell \cong_\Psi S$) are defined point-wise. Equivalence of terms is a congruence relation with $x^\ell \cong_\Psi y$ if and only if $\Psi_1\ (\ell, x) = y$ and $Ref\ n :: Ref\ \ell\ \tau \cong_\Psi Ref\ m$ and $DRef\ n :: Ref\ \ell\ \tau \cong_\Psi DRef\ m$ if and only if $\Psi_2\ (\ell, n) = m$. Using this notion of equivalence modulo Ψ, we can state the simulation results:

Proposition 12 (Decorated to Vanilla) *Given configurations $\langle \Sigma, \Gamma, t_1, S^\ell \rangle$ and $\langle M, \Delta, t_2, S \rangle$ which are well-typed and denote a computation of type $MAC\ \ell\ \tau$, if we have that:*

- $\langle \Sigma, \Gamma, t_1, S^\ell \rangle \longrightarrow \langle \Sigma', \Gamma', t_1', S'^\ell \rangle$ *and*
- $\langle \Sigma, \Gamma, t_1, S^\ell \rangle \cong_\Psi \langle M, \Delta, t_2, S \rangle$

then there exist M', Δ', t_2', S' and Ψ' such that:

- $\langle M, \Delta, t_2, S \rangle \longrightarrow \langle M', \Delta', t_2', S' \rangle$
- $\langle \Sigma', \Gamma', t_1', S'^\ell \rangle \cong_{\Psi'} \langle M', \Delta', t_2', S' \rangle$

Note that the resulting configurations are in relation according to some new bijection Ψ', rather than Ψ, as the bijection has to be extended with new memory or heap allocations. Dually, we show that configurations in the vanilla calculus can be simulated in the annotated one.

Proposition 13 (Vanilla to Decorated) *Given configurations $\langle \Sigma, \Gamma, t_1, S^\ell \rangle$ and $\langle M, \Delta, t_2, S \rangle$ which are well-typed and denote a computations of type $MAC\ \ell\ \tau$, if we have that:*

- $\langle M, \Delta, t_2, S \rangle \longrightarrow \langle M', \Delta', t_2', S' \rangle$
- $\langle \Sigma, \Gamma, t_1, S^\ell \rangle \cong_\Psi \langle M, \Delta, t_2, S \rangle$

then there exist Σ', Γ', t_1', S'^ℓ and Ψ' such that:

- $\langle \Sigma, \Gamma, t_1, S^\ell \rangle \longrightarrow \langle \Sigma', \Gamma', t_1', S'^\ell \rangle$ *and*
- $\langle \Sigma', \Gamma', t_1', S'^\ell \rangle \cong_{\Psi'} \langle M', \Delta', t_2', S' \rangle$

In both propositions we assume well-typed configurations. We omit the typing rules, which are rather standard. The important bits are present in the type signatures given in Figures 2 and 8. For the decorated calculus, the typing rules correspond to those of the vanilla calculus, but in addition ensure that the security labels appearing in the type conincide with those in the decorations. The proof is rather standard including references and variables allocation, where we keep some invariant regarding the lengths of heap and memories to connect the notion of "freshness" of variables on both calculi. The details of the proofs of these simulations can be found in Appendix B.

6.7 Vanilla Progress-Sensitive Non-Interference

We prove that *well-typed* programs in the vanilla lazy calculus satisfy progress-sensitive non-interference. This result relies on the PSNI proof for the decorated calculus and the simulations described above. We first define that two global configurations are ℓ_A-equivalence *up to a bijection* Ω, written $\langle M_1, \Delta_1, T_{s1}\rangle \approx^{\Omega}_{\ell_A} \langle M_2, \Delta_2, T_{s2}\rangle$, if and only if they are *well-typed* and their components are ℓ_A-equivalent up to bijection Ω, where ℓ_A-equivalence between terms is also type-driven and follows a structure similar to the one for the decorated calculus—the main difference being that it inspects the type-derivation of term and use the bijection Ω to relate memory addresses and heap variables. In the vanilla calculus we need to consider low-equivalence up to a bijection as in [4] to relate executions which might allocate a different amount of high entities, thus affecting the addresses and names of public references and variables respectively. Observe that bijection Ω connects heap variables and memory addresses of the vanilla calculus, that is Ω is a pair of bijections of type:

$$\Omega_1 :: Var \to Var$$
$$\Omega_2 :: \mathbb{N} \to \mathbb{N}$$

Configurations of the form $\langle [\,], \varnothing, [(t, [\,])]\rangle$ are initial configurations in the vanilla calculus, where the memory and thread's stacks are empty ($[\,]$), and the heap consists of an empty mapping ($\varnothing$).

Theorem 2 (Vanilla PSNI) *Given closed terms $t_1 :: MAC\ \ell\ \tau$ and $t_2 :: MAC\ \ell\ \tau$ written with the surface syntax (i.e., they do not contain constructors Labeled and Ref), we have that if:*

- $t_1 \approx^{\varnothing}_{\ell_A} t_2$, *and*
- $\langle [\,], \varnothing, [(t_1, [\,])]\rangle \hookrightarrow^* c_1$, *then:*

there exists c_2 and bijection Ω such that:

- $\langle [\,], \varnothing, [(t_2, [\,])]\rangle \hookrightarrow^* c_2$, *and*
- $c_1 \approx^{\Omega}_{\ell_A} c_2$

Proof 2 *(Sketch) Define $i_1 = \langle [\,], \varnothing, [(t_1, [\,])]\rangle$ and $i_2 = \langle [\,], \varnothing, [(t_2, [\,])]\rangle$. Since t_1 and t_2 are closed and well-typed terms in the surface syntax, we can lift them in the decorated calculus, as decorated terms t_1^D, t_2^D, and their corresponding initial annotated configurations i_1^D and i_2^D. Configurations i_1 and i_2 are equivalent up to the empty bijection $\varnothing$, i.e., $i_1^D \cong_{\emptyset} i_1$ and $i_2^D \cong_{\emptyset} i_2$ and $i_1^D \approx_{\ell_A} i_2^D$. By lifting Proposition 13 to thread pools and repetitively applying it, there exists a bijection Ψ_a and a configuration c_1^D, such that $i_1^D \hookrightarrow^* c_1^D$ and $c_1^D \cong_{\Psi_a} c_1$. By Theorem 1, there exists a decorated configuration c_2^D such that $i_2^D \hookrightarrow^* c_2^D$ and $c_1^D \approx_{\ell_A} c_2^D$. By lifting Proposition 12 to thread pools and repetitively applying it, we have that there exists a bijection Ψ_b and configuration c_2 such that $i_2 \hookrightarrow^* c_2$ where $c_2^D \cong_{\Psi_b} c_2$. We then conclude that c_1 and c_2 are ℓ_A-equivalent up to bijection Ω, obtained composing Ψ_a (from vanilla to decorated), and Ψ_b^{-1} (from decorated to vanilla), i.e., $c_1 \approx^{\Omega}_{\ell_A} c_2$, where $\Omega = \Psi_a \circ \Psi_b^{-1}$.*

7 Related Work

Mutable references and lazyness In Section 3.2 we present an operational semantics that features both mutable references and lazyness. It is a straight-forward combination of Sestoft's semantics with the standard approach to model references using a store, as described by Pierce et al. in the context of call-by-value [34, 35]. To the best of our knowledge, this is the first work that presents this combination. The "Awkward Squad" paper [33], which describes the implementation of I/O in Haskell, and addresses both references and concurrency, remarkably avoids dealing with sharing in its operational semantics.

deepDup Our primitive $lazyDup$ was inspired by the related primitive $deepDup$ proposed by the second author [7], with the aim to limit sharing in cases where it is actually detrimental to program performance. Because the terms in that work are in Administrative Normal Form (ANF), the rules for $deepDup$ look different from our [LAZYDUP$_2$], but this difference is inconsequential. We significantly improve over that work with the support to handle references, via the duplicate-on-read references introduced in Section 4.2. The Haskell library `ghc-dup` implements $deepDup$ without changes to the compiler or runtime, therefore we are optimistic that an implementation of $lazyDup$ is feasible.

Evaluation strategies and IFC Sabelfeld and Sands suggest that lazy evaluation might be safer than eager evaluation for termination leaks [42]. Buiras and Russo identify the risk imposed by internal timing leaks via lazy evaluation [11]. Vassena et al. enrich **MAC**'s API for labeled expressions by considering them as (applicative-like) functors [49] and show that their extension is vulnerable to termination leaks under eager evaluation, but secure under lazy evaluation. In a imperative sequential setting, Rafnsson et al. describe how Java's on-demand (lazy) class initialization process can be exploit to reveal secrets [38]. Strictness analysis detects functions that always evaluate their arguments, which can then be eagerly evaluated to boost performance of lazy evaluation [28]. In this context, this technique could be used to safely force the evaluation of shared thunks upfront. However, the analysis must necessarily be conservative, especially when it comes to infinite data structures and advanced features such as references and concurrency, therefore it is unlikely that all leaks could be closed by the analysis alone. Nevertheless, strictness analysis could avoid unnecessary duplication: the thunks, which are guaranteed by the analysis to be evaluated anyway, could be eagerly forced, and lazy duplication could be applied otherwise.

IFC libraries **LIO** dynamically enforces IFC applying similar concepts to **MAC** (i.e., labeled expressions, secure computations, etc.). We argue that **LIO** can be secure against the attack presented in this work by applying $lazyDup$ to the "rest of the computation" every time that the current label gets raised. For that, **LIO** needs to be reimplemented to work in a continuation passing style (CPS)—we leave this direction as future work. **HLIO** (hybrid-**LIO**) works as **LIO** except it

enforces IFC by combining type-level enforcement with dynamic checks [10]. To secure **HLIO**, $lazyDup$ needs to be inserted when forking threads if IFC gets enforced statically and when raising the current label if dynamic checks are involved. **HLIO** also needs to be reimplemented using CPS. In **MAC**, the type signature for the bind operator restricts computations to maintain the same security level. Its type could be relaxed to involve different increasing labels, along the lines of the "is protected" relation used in the typing rule of bind in the Dependency Core Calculus (DCC) [1]. However, in that case, a secure computation would not enjoy a standard monadic structure, but it would rather incorporate multiple monads.

Devriese and Piessens provide a monad transformer to extend imperative-like APIs with support for IFC [13]. Jaskelioff and Russo implement a library which dynamically enforces IFC using secure multi-execution (SME) [20]. Schmitz et al. [43] provide a library with *faceted values*, where values present different behavior according to the privilege of the observer. While these libraries do not support concurrency yet, we believe that, this work could secure them against lazy evaluation attacks, if they were extended with concurrency.

Programming languages Besides the already mentioned tools Jif, Paragon, FlowCaml, and JSFlow, we can remark the SPARK language and its IFC analysis, which has been extended to guarantee progress-sensitive non-inference [37] and JOANA [46], which stretches the scalability of static analyzes, in this case of Java programs. Some tools apply dependent-types to protect confidentiality (e.g., [24, 27, 30]). In such languages, type-checking triggers evaluation, potentially opening up possibilities to leak sensitive data via covert channels (e.g., lazy evaluation). In this light, it would be possible to learn something about a static secret when type-checking the program—an interesting direction for future work. Laminar combines programming languages and operating systems techniques to provide decentralized information flow control [39]. While supporting concurrency, Laminar does not handle covert channels like termination or internal timing leaks.

8 Conclusions

We present a solution to internal timing leaks via lazy evaluation, an open problem for security libraries written in Haskell. We believe that repairing existing libraries with $lazyDup$ would be reasonably a painless experience. The utilization of $lazyDup$ would make past and future systems built with security libraries more secure (e.g., Hails [15]). Even though it is still not clear which evaluation strategy is more beneficial for security, this work shows that the risks of lazy evaluation in concurrent settings can be successfully avoided.

Generally speaking, functional languages (and Haskell in particular) rely on their runtime (e.g., lazy evaluation, garbage collector, etc.) to provide essential features. Unfortunately, besides providing their functionality, they could also be

misused to jeopardize security. This work shows that a program can control parts of the complex runtime system (e.g., sharing) via a safe interface ($lazyDup$). Then, the obvious question is which other features of the runtime system could jeopardize security and how to safely control them—an intriguing thought to drive our future work.

References

1. M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A Core Calculus of Dependency. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 147–160, January 1999.
2. Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. Predictive black-box mitigation of timing channels. In *Proc. of the 17th ACM conference on Computer and Communications Security*. ACM, 2010.
3. Clem Baker-Finch, David J. King, and Phil Trinder. An operational semantics for parallel lazy evaluation. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 162–173, New York, NY, USA, 2000. ACM.
4. Anindya Banerjee and David A. Naumann. Stack-based access control and secure information flow. *Journal Functional Programming*, 15:131–177, March 2005.
5. David E. Bell and L. La Padula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997, Rev. 1, MITRE Corporation, Bedford, MA, 1976.
6. Andrew Bortz and Dan Boneh. Exposing private information by timing web applications. In *Proc. of the 16th World Wide Web*. ACM, 2007.
7. Joachim Breitner. dup – explicit un-sharing in Haskell. *CoRR*, abs/1207.2017, 2012.
8. Joachim Breitner. Formally proving a compiler transformation safe. In *Haskell Symposium*. ACM, 2015.
9. Niklas Broberg, Bart van Delft, and David Sands. Paragon for practical programming with information-flow control. In *APLAS*, volume 8301 of *Lecture Notes in Computer Science*, pages 217–232. Springer, 2013.
10. P. Buiras, D. Vytiniotis, and A. Russo. HLIO: Mixing static and dynamic typing for information-flow control in Haskell. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP '15)*. ACM, 2015.
11. Pablo Buiras and Alejandro Russo. Lazy programs leak secrets. In *Proc. Nordic Conference in Secure IT Systems (NORDSEC)*. Springer-Verlag, 2013.
12. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.
13. D. Devriese and F. Piessens. Information flow enforcement in monadic libraries. In *Proc. of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI '11)*. ACM, 2011.
14. Edward W. Felten and Michael A. Schneider. Timing attacks on Web privacy. In *Proc. of the 7th ACM conference on Computer and communications security*, CCS '00. ACM, 2000.
15. Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *10th Symposium on Operating Systems Design and Implementation*, October 2012.

16. D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: Tracking information flow in JavaScript and its APIs. In *Proc. of the ACM Symposium on Applied Computing (SAC '14)*. ACM, March 2014.
17. D. Hedin and David Sands. Noninterference in the presence of non-opaque pointers. In *Proc. of the 19th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 2006.
18. Stefan Heule, Deian Stefan, Edward Z. Yang, John C. Mitchell, and Alejandro Russo. IFC inside: Retrofitting languages with dynamic information flow control. In *Conference on Principles of Security and Trust (POST)*. Springer, April 2015.
19. John Hughes. Why functional programming matters. *The Computer Journal*, 32, 1984.
20. M. Jaskelioff and A. Russo. Secure multi-execution in Haskell. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, LNCS. Springer-Verlag, June 2011.
21. John Launchbury. A natural semantics for lazy evaluation. In *Principles of Programming Languages (POPL)*. ACM, 1993.
22. P. Li and S. Zdancewic. Encoding information flow in Haskell. In *Proc. of the IEEE Workshop on Computer Security Foundations*. IEEE Computer Society, 2006.
23. P. Li and S. Zdancewic. Arrows for secure information flow. *Theoretical Computer Science*, 411(19):1974–1994, 2010.
24. Luísa Lourenço and Luís Caires. Dependent information flow types. *SIGPLAN Not.*, 50(1), January 2015.
25. Simon Marlow and Simon Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. *Journal of Functional Programming*, 16(4-5):415–449, 2006.
26. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.
27. Jamie Morgenstern and Daniel R. Licata. Security-typed programming within dependently typed programming. In *Proc. of the 15th ACM SIGPLAN International Conference on Functional Programming*. ACM, 2010.
28. Alan Mycroft. *The theory and practice of transforming call-by-need into call-by-value*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1980.
29. A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 228–241, January 1999.
30. Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. Verification of information flow and access control policies with dependent types. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11. IEEE Computer Society, 2011.
31. Ulf Norell. Dependently typed programming in agda. In Andrew Kennedy and Amal Ahmed, editors, *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*, pages 1–2. ACM, 2009.
32. Simon Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.
33. Simon Peyton Jones. *Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell*, pages 47–96. IOS Press, January 2001.
34. Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

35. Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Software Foundations*. Electronic textbook, 2016. Version 4.0. `http://www.cis.upenn.edu/~bcpierce/sf`.
36. F. Pottier and V. Simonet. Information Flow Inference for ML. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 319–330, January 2002.
37. Willard Rafnsson, Deepak Garg, and Andrei Sabelfeld. Progress-sensitive security for SPARK. In *Engineering Secure Software and Systems - 8th International Symposium, ESSoS 2016, London, UK, April 6-8, 2016. Proceedings*, 2016.
38. Willard Rafnsson, Keiko Nakata, and Andrei Sabelfeld. Securing class initialization in Java-like languages. *IEEE Transactions on Dependable and Secure Computing*, 10(1), January 2013.
39. Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: Practical fine-grained decentralized information flow control. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09. ACM, 2009.
40. Alejandro Russo. Functional pearl: Two can keep a secret, if one of them uses Haskell. In *Proc. of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015. ACM, 2015.
41. A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.
42. Andrei Sabelfeld and David Sands. A per model of secure information flow in sequential programs. *Higher Order Symbol. Comput.*, 14(1), March 2001.
43. Thomas Schmitz, Dustin Rhodes, Thomas H. Austin, Kenneth Knowles, and Cormac Flanagan. Faceted dynamic information flow via control and data monads. In Frank Piessens and Luca Viganò, editors, *POST*, volume 9635 of *Lecture Notes in Computer Science*. Springer, 2016.
44. Peter Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(03), 1997.
45. G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM symposium on Principles of Programming Languages (POPL '98)*, 1998.
46. Gregor Snelting, Dennis Giffhorn, Jürgen Graf, Christian Hammer, Martin Hecker, Martin Mohr, and Daniel Wasserrab. Checking probabilistic noninterference using JOANA. *it - Information Technology*, 56(6):280–287, 2014.
47. D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Maziéres. Addressing covert termination and timing channels in concurrent information flow systems. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP '12)*. ACM, 2012.
48. D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Proc. of the ACM SIGPLAN Haskell symposium (HASKELL '11)*, 2011.
49. Marco Vassena, Pablo Buiras, Lucas Waye, and Alejandro Russo. Flexible manipulation of labeled values for information-flow control libraries. In *Proceedings of the 12th European Symposium On Research In Computer Security*. Springer, September 2016.
50. 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 '16, pages 15–28, New York, NY, USA, 2016. ACM.

51. D. Volpano, G. Smith, and C. Irvine. A Sound Type System for Secure Flow Analysis. *J. Computer Security*, 4(3):167–187, 1996.
52. Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Predictive mitigation of timing channels in interactive systems. In *Proc. of the 18th ACM conference on Computer and Communications Security*. ACM, 2011.
53. Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based Control and Mitigation of Timing Channels. In *Proc. ACM Conference on Programming Language Design and Implementation*. ACM, 2012.

$$\textsc{Var}\quad \frac{x \in Dom(\Gamma(\ell)) \qquad y \in Dom(\Delta) \qquad \Psi_{\Gamma,\Delta}(\ell, x) = y \qquad \Psi^{-1}_{\Gamma,\Delta}(y) = (\ell, x)}{x^{\ell} \cong_{\Psi_{\Gamma,\Delta}} y}$$

$$\textsc{Heap}\quad \frac{\forall\ \ell\ x\ y \qquad x^{\ell} \cong_{\Psi_{\Gamma,\Delta}} y \qquad \Gamma(\ell)(x) \cong_{\Psi_{\Gamma,\Delta}} \Delta(y)}{\Gamma \cong_{\Psi_{\Gamma,\Delta}} \Delta}$$

$$\textsc{Ref}\quad \frac{n < |\Sigma(\ell)| \qquad m < |\Delta| \qquad \Psi_{\Sigma,M}(\ell, n) = m \qquad \Psi^{-1}_{\Sigma,M}(m) = (\ell, n)}{Ref_{\ell}\ n \cong_{\Psi_{\Sigma,M}} Ref\ m}$$

$$\textsc{Memory}\quad \frac{\forall\ \ell\ n\ m \qquad Ref_{\ell}\ n \cong_{\Psi_{\Sigma,M}} Ref\ m \qquad \Sigma(\ell)[n] \cong_{\Psi_{\Sigma,M}} M[m]}{\Sigma \cong_{\Psi_{\Sigma,M}} M}$$

$$\textsc{Stack}_1\quad [\,]^{\ell} \cong_{\Psi} [\,]$$

$$\textsc{Stack}_2\quad \frac{C^{\ell} \cong_{\Psi} C \qquad S^{\ell} \cong_{\Psi} S}{C^{\ell} : S^{\ell} \cong_{\Psi} C : S}$$

$$\textsc{Var}_{\#}\quad \frac{x \notin Dom(\Gamma(\ell)) \qquad y \notin Dom(\Delta)}{\#x^{\ell} \cong_{\Psi_{\Gamma,\Delta}} \#y}$$

Fig. 18: Definition of $\cong_{\Psi_{\Gamma,\Delta}}$ and $\cong_{\Psi_{\Sigma,M}}$

Appendix

A Securing LIO

In **LIO**, it is not possible to know, at the time of forking, if the parent or the spawned thread will become sensitive, because threads get dynamically "tainted" when they observe a piece of sensitive information, e.g., by means of $unlabel$—an approach known as *floating-label system*. One could follow the same idea used in **MAC** and conservatively apply $lazyDup$ to all spawned threads. However, such approach would overly restrict sharing, e.g., if the thread never observes secrets. Instead, $lazyDup$ should be applied to the "rest of the computation" whenever the thread gets tainted—only then the evaluation of thunks can leak information! Implementing this idea requires to refactor the full implementation of **LIO** to work in a continuation-passing style, where the continuation represents the "rest of the computation". Then, when the thread gets tainted, $lazyDup$ can be applied to the continuation, thus disabling sharing with the parent thread from that point on.

B Simulation

In this section, we prove the *simulation* Propositions 12 and 13, that we used in Section 6 to prove Theorem 2. We give details for the interesting cases, i.e.,

rules [VAR$_1$,VAR$_2$,LAZYDUP$_1$, LAZYDUP$_2$,NEW]. Firstly, we refine the type for *bijection* on heap variables with a heap map Γ and a heap Δ. In particular we write Var^{Δ} to restrict the type of heap variables to those in the domain of Δ, i.e., $Var^{\Delta} = \{x \mid x \in Dom(\Delta)\}$. Similarly, we write $\mathbb{N}^M$, to restrict the type of memory addresses to those in the domain of memory M, i.e., $\mathbb{N}^M = \{n \mid n < |M|\}$. We then write $(\ell : Label \times P(\ell))$, for the dependent pair type (also known as Sigma-type) $\Sigma_{(\ell:Label)}P(\ell)$. We then give the following more precise type to heap variables and memory addresses bijections:

$$\Psi_{\Gamma,\Delta} :: (\ell : Label \times Var^{\Gamma(\ell)}) \rightarrow Var^{\Delta}$$
$$\Psi_{\Sigma,M} :: (\ell : Label \times \mathbb{N}^{\Sigma(\ell)}) \rightarrow \mathbb{N}^M$$

In particular heap-indexed and memory-indexed bijections relate *only* variables and addresses in their domains. In the following we abbreviate the pair of bijection $(\Psi_{\Gamma,\Delta}, \Psi_{\Sigma,M})$ with Ψ, and sometimes we specify only the relevant component of the pair to avoid clutter. Figure 18 shows the definition of equivalence up to heap-bijection ($\cong_{\Psi_{\Gamma,\Delta}}$) and equivalence up to memory-bijection ($\cong_{\Psi_{\Sigma,M}}$) for the interesting cases. Rule [VAR] relates the variables in the domain of $\Gamma(\ell)$ and Δ respectively, using the bijection $\Psi_{\Gamma,\Delta}$. Rule [HEAP] defines equivalence of heaps up to bijection pointwise for the variables in their domain, i.e., a store and a heap are equivalent up to bijection, if and only if they map related variables into related terms. Rules [REF,MEMORY] apply the same principles to memory addresses. In decorated calculus we write $Ref_{\ell}\ n$, to denote that the reference has type $Ref\ \ell\ \tau$, for some type τ—the vanilla reference $Ref\ m$ has the same type. Note that related stacks share the same structure, i.e., all their continuations are related, where the only interesting case involve the update marker continuation, i.e., $\#x$. Rule [VAR$_{\#}$] states that a decorated continuation $\#x^{\ell}$ and a vanilla continuation $\#y$ are related if and only if *both* variables are *free* in their respective heaps. Term-equivalence up to bijection is defined inductively on their structures, e.g., $lazyDup\ t^D \cong_{\Psi} lazyDup\ t$ if and only if $t^D \cong_{\Psi} t$. We remark that these relations are defined over well-typed terms of the *same* type, that is $t^D \cong_{\Psi} t$, assumes typing judgment $\pi \vdash t : \tau$, for some typing context π, and that t^D has type τ in the same typing context—we distinguish decorated terms (e.g., t^D), from vanilla terms (e.g., t), with a superscript. The typing rules for the vanilla calculus are standard and thus omitted—they corresponds to the type signatures given in Figures 2 and 8.

Weakening If two configurations are equivalent up to bijection Ψ, i.e., $c^D \cong_{\Psi} c$, then they are equivalent up to any bijection $\Psi \cup \{x^{\ell} \leftrightarrow y\}$, for any pair of variables x^{ℓ} and y, that are *fresh* in the respective configurations.

Strengthening If two configurations are equivalent via an extended bijection, e.g., $c^D \cong_{\Psi \cup \{x^{\ell} \leftrightarrow y\}} c$, then they are also equivalent in a reduced bijection, e.g., $c^D \cong_{\Psi} c$, if and only if they occur in their respective stack under an update marker, e.g., $\#x^{\ell}$ and $\#y$.

We now prove Propositions 12 and 13 for rules [VAR$_1$,VAR$_2$,LAZYDUP$_1$, LAZYDUP$_2$, NEW].

- Rule [VAR$_1$].
  - *Decorated to Vanilla*: Given a step $(\Delta^\ell[x^\ell \mapsto t^D], x^\ell, S^\ell) \rightsquigarrow (\Delta^\ell, t^D, \#x^\ell : S^\ell)$ and a vanilla configuration (Δ, t, S) and a bijection Ψ, such that $(\Delta^\ell[x^\ell \mapsto t^D], x^\ell, S^\ell) \cong_\Psi (\Delta, t, S)$, show that there exists a configuration (Δ', t', S') such that $(\Delta, t, S) \rightsquigarrow (\Delta', t', S')$ and a bijection Ψ' such that $(\Delta^\ell, t, \#x^\ell : S^\ell) \cong_{\Psi'} (\Delta', t', S')$. Since the initial configurations are in relation, then so are their heaps (recall $\Delta^\ell = \Sigma(\ell)$), current terms and stacks. Therefore the term t is a variable y, such that $x^\ell \cong_{\Psi_{\Gamma,\Delta} \cup \{x^\ell \leftrightarrow y\}} y$ and the heap Δ contains a binding for y, that is $\Delta^\ell[x^\ell \mapsto t^D] \cong_{\Psi_{\Gamma,\Delta} \cup \{x^\ell \leftrightarrow y\}} \Delta[y \mapsto t]$. The vanilla configuration then steps according to rule [VAR$_1$], i.e., $(\Delta[y \mapsto t], y, S) \rightsquigarrow (\Delta, t, \#y : S)$, which is equivalent to the decorated configuration up to the bijection $\Psi_{\Gamma,\Delta}$, i.e., the bijection obtained by removing mapping $x^\ell \leftrightarrow y$. Note that $t^D \cong_\Psi t$, since they are the image of related variables in related heaps and $\#x^\ell : S^\ell \cong_{\Psi_{\Gamma,\Delta}} \#y : S$, since the stacks are related and the variables are *both* free in Γ and Δ respectively—rule [VAR$_1$] removes them to achieve the *blackholing* effect.
  - *Vanilla to Decorated*: The proof is symmetric. In this case we have a vanilla [VAR$_1$] step, i.e., $(\Delta[y \mapsto t], y, S) \rightsquigarrow (\Delta, t, \#y : S)$. Since the vanilla configuration denotes a secure computation of type $MAC\ \ell\ \tau$, for some label ℓ and some type τ, then the equivalent decorated configuration is labeled with ℓ, i.e., $(\Delta^\ell[x^\ell \mapsto t^D], x^\ell, S^\ell)$. The proof then follows similarly, by making the same considerations about the vanilla configuration to draw the same conclusions about the decorated configuration.
- Rule [VAR$_2$].
  - *Vanilla to Decorated*: Consider a vanilla step [VAR$_2$], i.e., $(\Delta, v, \#x : S) \rightsquigarrow (\Delta[x \mapsto v], v, S)$. Since the configuration denotes a secure computation of type $MAC\ \ell\ \tau$, the Ψ-equivalent decorated configuration is labeled with ℓ and has the same shape, i.e., $(\Delta^\ell, v^D, \#x^\ell : S^\ell)$—if a vanilla term v is a value then v^D is also a value and if the top of a vanilla stack has an update marker, so does the equivalent decorated stack, by rules [STACK$_2$, VAR$_\#$]. The decorated configuration then steps according to rule [VAR$_2$], to the configuration $(\Delta^\ell[x^\ell \mapsto v^D], v^D, S^\ell)$, which is equivalent to the vanilla configuration up to bijection $\Psi \cup \{x^\ell \leftrightarrow x\}$—we extend related heaps with related terms, i.e., $v^D \cong_\Psi v$. Note that from $\#x^\ell \cong_{\Psi_{\Gamma,\Delta}} \#x$, we have that x^ℓ and x are *free* in $\Delta^\ell = \Gamma(\ell)$ and Δ respectively. By popping related continuations from related stacks we obtain related stacks, i.e., $S^\ell \cong_\Psi S$.
  - *Decorated to Vanilla*: The proof follows symmetrically. In this case the label ℓ is explicitly available in the decorated configuration.
- Rule [LAZYDUP$_1$].

- *Vanilla to Decorated*: Given step $\langle M, \Delta, lazyDup\ t, S\rangle \longrightarrow \langle M, \Delta[x \mapsto t], lazyDup\ x, S\rangle$, i.e., rule [LAZYDUP$_2$], lifted by rule [LIFT], where x is fresh in Δ and t is not a variable. Observe that, since the configuration denotes a secure computation of type $MAC\ \ell\ \tau$ for some label ℓ and some type τ, then the equivalent decorated initial configuration is labeled with ℓ, and it has $lazyDup\ t^D$ as the current term, where $t^D \cong_\Psi t$ and $\neg\ (isVar\ t^D)$. Then, the decorated configuration $\langle \Sigma, \Gamma, lazyDup\ t^D, S^\ell\rangle$ steps according to rule [LAZYDUP$_1$] to $\langle \Sigma, \Gamma[\ell][x^\ell] := t^D, lazyDup\ x^\ell, S^\ell\rangle$, for some fresh variable x^ℓ in $\Gamma(\ell)$. The final configurations are then equivalent up to the bijection $\Psi \cup \{x^\ell \leftrightarrow x\}$—note that this is a bijection because x^ℓ and x are fresh in $\Gamma(\ell)$ and Δ respectively, hence they are not mapped in Ψ. Specifically the heaps are extended with related terms and hence are related by the extended bijection $\Psi \cup \{x^\ell \leftrightarrow x\}$, and $lazyDup\ t^D \cong_{\Psi\cup\{x^\ell\leftrightarrow x\}} lazyDup\ t$.
- *Decorated to Vanilla*: The proof follows symmetrically. In this case the label ℓ is explicitly available in the decorated configuration and the step is simulated in the vanilla calculus by rule [LAZYDUP$_1$] lifted to vanilla sequential configuration by rule [LIFT].

– Rule [LAZYDUP$_2$].
- *Vanilla to Decorated* Given step $\langle M, \Delta, lazyDup\ x, S\rangle \longrightarrow \langle M, \Delta[y \mapsto [\![t]\!]^\varnothing], y, S\rangle$, i.e., rule [LAZYDUP$_2$], lifted by rule [LIFT], where variable y is fresh and variable x is bound to term t in the heap Δ, the Ψ-equivalent decorated configuration contains a Ψ-equivalent heap map Γ, i.e. $\Gamma \cong_\Psi \Delta$, a Ψ-equivalent stack S^{ℓ_H}, i.e. $S^{\ell_H} \cong_\Psi S$, and a Ψ-equivalent current term $lazyDup\ x^{\ell_L}$, i.e. $lazyDup\ x^{\ell_L} \cong_\Psi lazyDup\ x$, from which it follows that $x^{\ell_L} \cong_\Psi x$. Since variables and heaps are related, we have that the corresponding thunks are also related, i.e. there exists t^D, such that $\Gamma(\ell_L)(x^{\ell_L}) = t^D$ and $t^D \cong_\Psi t$. The decorated configuration then steps according to rule [LAZYDUP$_2$], giving heap map $\Gamma[\ell_H][y^{\ell_H}] := [\![t^D]\!]^\varnothing$ and current term y^{ℓ_H}, for some fresh variable y^{ℓ_H}. The resulting decorated configuration is then equivalent to the vanilla configuration up to the bijection $\Psi' = \Psi \cup \{y^{\ell_H} \leftrightarrow y\}$—it is a bijection because the variables are fresh. The heaps are related, i.e. $\Gamma[\ell_H][y^{\ell_H}] := [\![t^D]\!]^\varnothing \cong_{\Psi'} \Delta[y \mapsto [\![t]\!]^\varnothing]$, because we extend related heaps with related terms—function $[\![]\!]^\varnothing$ preserves equivalence up to bijection, i.e. if $t^D \cong_\Psi t$ then $[\![t^D]\!]^\varnothing \cong_\Psi [\![t]\!]^\varnothing$. The current terms are related by definition, i.e. $y^{\ell_H} \cong_{\Psi'} y$, because $(y^{\ell_H} \leftrightarrow y) \in \Psi'$.
- *Decorated to Vanilla*: The proof follows symmetrically. In this case the label ℓ is explicitly available in the decorated configuration and the step is simulated in the vanilla calculus by rule [LAZYDUP$_2$] lifted to vanilla sequential configuration by rule [LIFT].

– Rule [NEW]
- *Vanilla to Decorated*: Consider the step $\langle M, \Delta, new\ t, S\rangle \longrightarrow \langle M[n \mapsto x], \Delta[x \mapsto t], return\ (Ref\ n), S\rangle$, where $|M| = n$ and x is fresh.

The Ψ-equivalent configuration, consists of a Ψ-equivalent store Σ, i.e., $\Sigma \cong_\Psi M$, a Ψ-equivalent heap map Γ, i.e., $\Gamma \cong_\Psi \Delta$, a Ψ-equivalent current term $new\ t^D$, i.e., $new\ t^D \cong_\Psi new\ t$, from which we have $t^D \cong_\Psi t$. From the type derivation of the *well-typed* vanilla configuration, we know that term $new\ t$ has type $MAC\ \ell_{\mathrm{L}}\ (Ref\ \ell_{\mathrm{H}}\ \tau)$, for some type τ and some labels ℓ_{L} and ℓ_{H}, such that $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$. The decorated configuration steps according to rule [NEW], giving for a fresh variable $x^{\ell_{\mathrm{L}}}$ and $m = |\Sigma(\ell_{\mathrm{H}})|$, the store $\Sigma[\ell_{\mathrm{H}}][m] := x^{\ell_{\mathrm{L}}}$, heap map $\Delta[\ell_{\mathrm{L}}][x^{\ell_{\mathrm{L}}}] := t^D$ and current term $return\ (Ref_{\ell_{\mathrm{L}}}\ m)$. The resulting configurations are equivalent up to the bijection $\Psi' = (\Psi_{\Gamma,\Delta} \cup \{x^{\ell_{\mathrm{L}}} \leftrightarrow x\}, \Psi_{\Sigma,M} \cup \{(\ell_{\mathrm{H}}, m) \leftrightarrow n\})$, i.e., the bijection obtained by extending the heap variables and memory addresses bijections with the new mappings $x^{\ell_{\mathrm{L}}} \leftrightarrow x$ and $(\ell_{\mathrm{H}}, m) \leftrightarrow n$ respectively. The heaps are equivalent up to the bijection $\Psi_{\Gamma,\Delta} \cup \{x^{\ell_{\mathrm{L}}} \leftrightarrow x\}$ since we extend $\Psi_{\Gamma,\Delta}$-equivalent heaps variables with respectively *fresh* variables $x^{\ell_{\mathrm{L}}}$ and x, which are bound to Ψ-equivalent terms, i.e., $t^D \cong_\Psi t$. Similarly, the memories are equivalent up to the bijection $\Psi_{\Sigma,M} \cup \{(\ell_{\mathrm{H}}, m) \leftrightarrow n\}$, since we extend $\Psi_{\Sigma,M}$-equivalent memories, by assigning equivalent references i.e., $x^{\ell_{\mathrm{L}}} \cong_{\Psi_{\Gamma,\Delta} \cup \{x^{\ell_{\mathrm{L}}} \leftrightarrow x\}} x$ to *fresh* addresses. Observe that the current terms in the final configurations are related, i.e., $return\ (Ref_{\ell_{\mathrm{L}}}\ m) \cong_{\Psi'} return\ (Ref\ n)$, because the addresses are related by the bijection Ψ', i.e., $m \cong_{\Psi_{\Sigma,M} \cup \{(\ell_{\mathrm{H}}, m) \leftrightarrow n\}} n$.

- *Decorated to Vanilla*: The proof follows symmetrically. In this case the labels ℓ_{L} and ℓ_{H} are explicitly available in the decorated configurations, i.e., $\langle \Sigma, \Gamma, new\ t^D, S^{\ell_{\mathrm{L}}} \rangle \rightsquigarrow \langle \Sigma[\ell_{\mathrm{H}}][m] := x^{\ell_{\mathrm{L}}}, \Gamma[\ell_{\mathrm{L}}][x^{\ell_{\mathrm{L}}}] := t^D, return\ (Ref_{\ell_{\mathrm{H}}}\ m), S^{\ell_{\mathrm{L}}} \rangle$.

C Sharing and References

Our calculus captures sharing precisely, even in presence of references, and despite the extra-indirection between the memory and heap. We provide two examples showing the interaction among references, sharing, and thunks.

Example 1. Consider the following program, which creates a reference, immediately overwrites it with 1, and finally returns 0:

```
let x = 0 in
do r ← new x
   write r 1
   return x
```

If reference r pointed directly to x (no extra-indirection), the next write operation would actually rewrite x to 1 in the *immutable* heap and the program would return 1, instead of 0.

$$\varepsilon_{\ell_A}(\langle \Delta^{\ell}, t, S^{\ell} \rangle) = \begin{cases} \langle \varepsilon_{\ell_A}(\Delta^{\ell}), \varepsilon_{\ell_A}(t), \varepsilon_{\ell_A}(S^{\ell}) \rangle & \text{if } \ell \sqsubseteq \ell_A \\ \bullet & \text{otherwise} \end{cases}$$

$$\varepsilon_{\ell_A}(\langle \Sigma, \Gamma, t, S^{\ell} \rangle) = \begin{cases} \langle \varepsilon_{\ell_A}(\Sigma), \varepsilon_{\ell_A}(\Gamma), \varepsilon_{\ell_A}(t), \varepsilon_{\ell_A}(S^{\ell}) \rangle & \text{if } \ell \sqsubseteq \ell_A \\ \bullet & \text{otherwise} \end{cases}$$

$$\varepsilon_{\ell_A}(\mathit{Ref}\ n :: \mathit{Ref}\ \ell_{\mathrm{H}}\ \tau) = \begin{cases} \mathit{Ref}\ \bullet & \text{if } \ell_{\mathrm{H}} \not\sqsubseteq \ell_A \\ \mathit{Ref}\ n & \text{otherwise} \end{cases}$$

$$\varepsilon_{\ell_A}(\mathit{label}\ t :: \mathit{Mac}\ \ell_{\mathrm{L}}\ (\mathit{Labeled}\ \ell_{\mathrm{H}}\ \tau)) = \begin{cases} \mathit{label}\ \bullet & \text{if } \ell_{\mathrm{H}} \not\sqsubseteq \ell_A \\ \mathit{label}\ (\varepsilon_{\ell_A}(t :: \tau)) & \text{otherwise} \end{cases}$$

$$\varepsilon_{\ell_A}(\mathit{fork}\ t :: \mathit{Mac}\ \ell_{\mathrm{L}}\ ()) = \begin{cases} \mathit{fork}_{\bullet}\ (\varepsilon_{\ell_A}(t :: \mathit{Mac}\ \ell_{\mathrm{H}}\ ())) & \text{if } \ell_{\mathrm{H}} \not\sqsubseteq \ell_A \\ \mathit{fork}\ \varepsilon_{\ell_A}(t) & \text{otherwise} \end{cases}$$

$$\varepsilon_{\ell_A}(\bullet) = \bullet$$

Fig. 19: Erasure function

Example 2. Consider the following program, which writes a thunk in a reference, reads it and evaluates its content twice.

$$\begin{array}{l} \textbf{let}\ x = \mathit{id}\ 1\ \textbf{in} \\ \textbf{do}\ r \leftarrow \mathit{new}\ x \\ \quad y \leftarrow \mathit{read}\ r \\ \quad \textbf{when}\ (y \leqslant 0)\ \mathit{return}\ () \\ \quad z \leftarrow \mathit{read}\ r \\ \quad \textbf{when}\ (z \geqslant 0)\ \mathit{return}\ () \end{array}$$

This program demands the value of y to evaluate $y \leqslant 0$ and the value to z to evaluate $z \geqslant 0$, but, surprisingly enough, the value of z is already computed. This sounds counter-intuitive because we expect y and z to be bound to the same expression $\mathit{id}\ 1$, since the program does not overwrite reference r between the first and the second read. In fact, variables y and z are *aliases* of the same variable x, whose thunk $\mathit{id}\ 1$ is updated with 1 after checking $y \leqslant 0$, thanks to sharing, and used to check $z \geqslant 0$. Observe that, while this program does not contain an explicit *write* operation, it still does perform one subtly, in the heap, since it *indirectly* updates x.

D Erasure Function

Figure 19 shows the definition of the erasure functions for the interesting cases. Configurations, whose label is above that of the attacker, i.e., $\ell \not\sqsubseteq \ell_A$ are

$$\text{(HOLE}_1\text{)} \quad \bullet \rightsquigarrow \bullet \qquad\qquad \text{(HOLE}_2\text{)} \quad \bullet \longrightarrow \bullet$$

Fig. 20: Semantics rules for $\bullet$

$$\text{(FORK)} \quad \frac{\Phi(\ell_{\text{L}}) = T_{s1}[n \mapsto (\mathit{fork}\ t, S^{\ell_{\text{L}}})] \qquad \Phi(\ell_{\text{H}}) = T_{s2} \qquad |T_{s2}| = m \qquad T'_{s1} = T_{s1}[n \mapsto (\mathit{return}\ (), S^{\ell_{\text{L}}})] \qquad T'_{s2} = T_{s2}[m \mapsto (\mathit{lazyDup}\ t, [\,]^{\ell_{\text{H}}})]}{\langle \Sigma, \Gamma, \Phi \rangle \hookrightarrow \langle \Sigma, \Gamma, \Phi[\ell_{\text{L}} \mapsto T'_{s1}][\ell_{\text{H}} \mapsto T'_{s2}] \rangle}$$

$$\text{(FORK}_\bullet\text{)} \quad \frac{\Phi(\ell_{\text{L}}) = T_{s1}[n \mapsto (\mathit{fork}_\bullet\ t, S^{\ell_{\text{L}}})] \qquad T'_{s1} = T_{s1}[n \mapsto (\mathit{return}\ (), S^{\ell_{\text{L}}})]}{\langle \Sigma, \Gamma, \Phi \rangle \hookrightarrow \langle \Sigma, \Gamma, \Phi[\ell_{\text{L}} \mapsto T'_{s1}] \rangle}$$

Fig. 21: Semantics rules of *fork* and $\mathit{fork}_\bullet$

rewritten to $\bullet$, otherwise they are erased by erasing each component. Steps involving sensitive configurations are then simulated by rules [HOLE$_1$, HOLE$_2$], shown in Figure 20. Memories, heaps, stacks and thread pools labeled with ℓ are also collapsed to $\bullet$, if their label is not visible to the attacker, i.e., $\ell \not\sqsubseteq \ell_A$, otherwise they are erased homomorphically. Label partitioned data structures, i.e., heap maps, stores and pool maps, are erased pointwise, e.g. $\varepsilon_{\ell_A}(\Gamma) = \ell \mapsto \varepsilon_{\ell_A}(\Gamma(\ell))$. The term $\mathit{label}\ t :: \mathit{MAC}\ \ell_{\text{L}}\ (\mathit{Labeled}\ \ell_{\text{H}}\ \tau)$ is erased to $\mathit{label}\ \bullet$, if $\ell_{\text{H}} \not\sqsubseteq \ell_A$, so that rule [LABEL] commutes. The terms new, write, fork are interesting. Observe that all these terms perform a *write-effect*, to a non-lower security level, due to the no write-down policy, which allows a computation visible to the attacker ($\ell_{\text{L}} \sqsubseteq \ell_A$) to write to a non-visible resource ($\ell_{\text{H}} \not\sqsubseteq \ell_A$). Simulating such steps, i.e., the label-decorated version of rules [NEW, WRITE, FORK], is challenging and requires *two-steps erasure* [50], a technique that performs erasure in two-stages, by firstly rewriting the problematic constructs , such as new, write and fork to special constructs, i.e., $\mathit{new}_\bullet$, $\mathit{write}_\bullet$ and $\mathit{fork}_\bullet$, whose special semantics rule guarantees simulation. We remark that such special constructs are introduced due to mere technical reasons and they are not part of the plain calculus. We use $\mathit{fork}_\bullet$ as an example to illustrate this technique. Figure 21 shows rules [FORK] and [FORK$_\bullet$], that is the label annotated rules for fork and $\mathit{fork}_\bullet$ respectively. Rule [FORK] is similar to its annotated counterpart shown in Figure 14, save for the extra look-up and update through the thread pool map Φ. Rule [FORK$_\bullet$] mimics rule [FORK], for what concerns the parent thread, but it ignores thread t, which is not added to the thread pool. Observe that rule [FORK] does not correctly simulate fork operations that occur in high threads. In particular, the high thread pool T_{s2} is rewritten by the erasure function to $\bullet$, since $\ell_{\text{H}} \not\sqsubseteq \ell_A$, however $|\bullet| \not\equiv m$.

On the other hand by rewriting $fork$ to a new term, i.e., $fork_\bullet$, we are free to adjust its semantics, to correctly simulate a low thread forking a high one in erased configurations. Specifically, we can show that [FORK] commutes with [FORK$_\bullet$], by proving that for all thread pool maps Φ, Φ', such that $\Phi' = \Phi[\ell_{\mathrm{H}} \mapsto (t, S^{\ell_{\mathrm{H}}})]$ and $\ell_{\mathrm{H}} \not\sqsubseteq \ell_A$, then $\varepsilon_{\ell_A}(\Phi) \equiv \varepsilon_{\ell_A}(\Phi')$, i.e., the attacker is oblivious to writes in thread pools above its security level.