MAC A verified static information-flow control library

Downloaded from: https://research.chalmers.se, 2024-09-20 01:36 UTC

Citation for the original published paper (version of record):
Vassena, M., Russo, A., Buiras, P. et al (2018). MAC A verified static information-flow control library. Journal of Logical and Algebraic Methods in Programming, 95: 148-180.
http://dx.doi.org/10.1016/j.jlamp.2017.12.003

N.B. When citing this work, cite the original published paper.

research.chalmers.se offers the possibility of retrieving research publications produced at Chalmers University of Technology. It covers all kind of research output: articles, dissertations, conference papers, reports etc. since 2004. research.chalmers.se is administrated and maintained by Chalmers Library

(article starts on next page)

Journal of Logical and Algebraic Methods in Programming 95 (2018) 148–180

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

MAC
A verified static information-flow control library

Marco Vassena [a,**], Alejandro Russo [a,*], Pablo Buiras [b], Lucas Waye [b]

[a] Chalmers University of Technology, Sweden
[b] Harvard University, USA

ARTICLE INFO

Article history:
Received 1 April 2017
Received in revised form 22 September 2017
Accepted 5 December 2017
Available online 14 December 2017

Keywords:
Information Flow Control
Non Interference
Functional Programming
Haskell
Agda

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) responsible 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 noninterference for our sequential calculus and pinpoint sufficient requirements on the scheduler to prove progress-sensitive noninterference 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 greatly simplifies the noninterference proof and helps to prove many advanced features of **MAC**.

© 2017 Elsevier Inc. All rights reserved.

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 [40]. 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 flows only into appropriate places*.

* Corresponding author.
** Second corresponding author.
E-mail addresses: vassena@chalmers.se (M. Vassena), russo@chalmers.se (A. Russo), pbuiras@seas.harvard.edu (P. Buiras), lwaye@seas.harvard.edu (L. Waye).

https://doi.org/10.1016/j.jlamp.2017.12.003
2352-2208/© 2017 Elsevier Inc. All rights reserved.

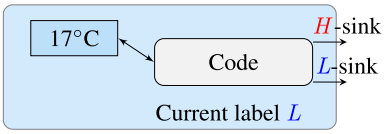


Fig. 1. Public computation.

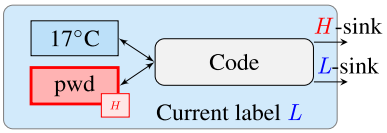


Fig. 2. Labeled values.

Language-based Information-Flow Control (IFC) [57] is a promising approach to enforcing information security in software. A traditional IFC enforcement scrutinizes how data of different sensitivity levels (e.g., public or private) flows within a program, detects when an unsafe flow of information occurs and take action to suppress the leakage. To do that, most IFC tools require the design of new languages, compilers, interpreters, or modifications to the runtime, e.g., [45,50,52,12]. Nonetheless, in the functional programming language Haskell, the strict separation between side-effect free and side-effectful code enables lightweight security enforcements. Specifically, it is possible to build a secure programming language atop Haskell, as an embedded domain-specific language that gets distributed and used as a Haskell library [34]. Many of the state-of-the-art Haskell security libraries, namely **LIO** [64], **HLIO** [16], and **MAC** [53], bring ideas from Mandatory Access Control [7] into a language-based setting.[1] These libraries promote a *secure-by-construction* programming model: any program written against their API does not leak secrets. This model is attractive, because it protects not only against benign code that leaks accidentally, e.g., due to a software bug, but also against a malicious program designed to do so. 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. IFC uses labels to model the sensitivity of data, which are then organized in a security lattice [17] 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. Fig. 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 [21]. While secure, programming in this model can be overly restrictive for users who want to manipulate differently-labeled values.

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. Fig. 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* [63].

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 [29]. Nevertheless, the programming model involving sub-computations that manipulate secrets forces an imperative style, whereby computations must be structured into separate compartments that

[1] From now on, we simply use the term libraries when referring to **LIO**, **HLIO**, and **MAC**.

must communicate explicitly. While side-effecting instructions have an underlying algebraic structure, called monad [41], research literature has neglected studying the algebraic structure of 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* [39]. 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 need 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 relabellings 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**[2] 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 noninterference (PINI) for the sequential calculus.
2. We enrich the calculus with scheduler-parametric concurrency and prove progress-sensitive noninterference (PSNI) [2] 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 [63,25]. 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.[3]

Highlights This work builds on our previous papers "Flexible Manipulation of Labeled Values for Information-Flow Control Libraries" [71] and "On Formalizing Information-Flow Control Libraries" [72], 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:

- 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 noninterference (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 noninterference (PSNI). Section 11 gives related work and Section 12 concludes.

2. Overview

In **MAC**, each label is represented as an abstract data type. Fig. 3 shows the core part of **MAC**'s API. Abstract data type *Labeled* ℓ *a* classifies data of type *a* with a security label ℓ. For instance, *pwd* :: *Labeled H String* is a sensitive string, while

[2] Available at https://bitbucket.org/MarcoVassena/mac-model/.
[3] https://hackage.haskell.org/package/mac.

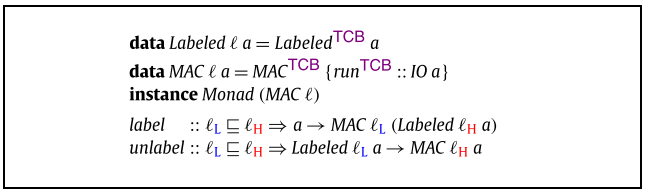


Fig. 3. Core API for **MAC**.

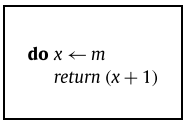


Fig. 4. do-notation.

rating :: *Labeled L Int* is a public integer. (Symbol :: is used to describe the type of terms in Haskell.) Abstract data type *MAC* ℓ *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 *MAC* ℓ *a* computation enjoys a monadic structure, i.e., it is built using the fundamental operations *return* :: $a \to$ *MAC* ℓ *a* and $(\gg\!=)$ :: *MAC* ℓ *a* $\to$ $(a \to$ *MAC* ℓ *b*$) \to$ *MAC* ℓ *b* (read as "bind"). The operation *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 $m \gg\!= \lambda x \to$ *return* $(x+1)$, which adds 1 to the value produced by *m*, can be written as shown in Fig. 4.

2.1. Secure flows of information

Generally speaking, side-effects in a *MAC* ℓ *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 *unlabel* allow *MAC* ℓ *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 $\ell_L \sqsubseteq \ell_H$. If a *MAC* ℓ_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* [7], 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 $\ell_L \sqsubseteq \ell_H$ ensures that when calling *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 *MAC* ℓ_H *a* is only allowed to read labeled values at most as sensitive as ℓ_H—observe the type constraint $\ell_L \sqsubseteq \ell_H$ in the type signature of *unlabel*. This restriction, known as *no read-up* [7], protects the confidentiality degree of the result produced by *MAC* ℓ_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.[4] Specifically side-effects, i.e., input–output operations, can only occur in monadic computations of type *IO a*. Crucially *pure* computations are inherently secure, while *IO* computations are potentially leaky. In **MAC**, a secure computation of type *MAC* ℓ *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*$^{\mathsf{TCB}}$ extracts the underlying *IO a* computation from a secure computation of type *MAC* ℓ *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*$^{\mathsf{TCB}}$ 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 Fig. 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* $\to$ *IO* () prints to standard output and *getLine* :: *IO String* reads from standard input.

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

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

Fig. 5. The password is exposed in *isWeak*.

```
isWeak :: Labeled H String → MAC L (Labeled H Bool)

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

Fig. 6. Label H protects the password in *isWeak*.

```
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.

Clearly the content of variable *pwd* should be handled with care by *isWeak* :: *String* → *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. Adjusting 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* → *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* → *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. Fig. 6 shows the modifications to the code needed to use a secure password strength checker. 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 (>>=), function run^{TCB} executes the whole computation, whose labeled result is then pattern matched with $Labeled^{TCB}$, exposing the boolean value, that is finally returned.[5]

2.2. Implicit flows

The interaction between the current label of a computation and the no read-up and no write-down security policies makes implicit flow ill-typed. Consider for instance, the ill-typed program in Fig. 7, that attempts to leak the value of the secret boolean in a public boolean. Unlike other IFC system, the code cannot branch on *secret* directly, because it is explicitly labeled, i.e., it has type *Labeled H Bool*, instead of *Bool*. In order to branch on sensitive data, the program needs first to unlabel it, thus incurring in the no read-up restriction that requires the computation to be sensitive as well, that is the program must have type *MAC H a* (for some type *a*). The only primitive that produces labeled data is *label*, which according to the no write-down restriction, prevents a sensitive computation from creating a public labeled value. Regardless of the branch taken, but for that reason, i.e., trying to label a piece of data with L in a computation labeled with H, the program in Fig. 7 is ill-typed.

[5] In Fig. 6, the code in the IO monad is trusted, hence the use of run^{TCB} and $Labeled^{TCB}$. The function *isWeak* is not trusted and the password is protected by **MAC** secure API.

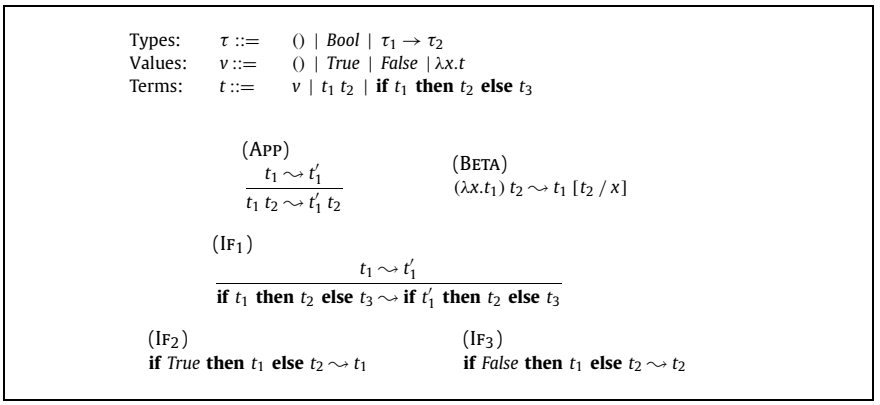


Fig. 8. Syntax and semantics of the pure calculus.

Features overview Modern programming languages provide many abstractions that simplify the development of complex software. In the following, we extend **MAC** with additional primitives that make software development within **MAC** practical, without sacrificing security. The list of programming features securely supported in **MAC** include exception handling (Section 5), references (Section 6), concurrency (Section 8), functors 9 and synchronization primitives (Appendix B).

3. The 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

Fig. 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 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-name* 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*.[6] Rule [If$_1$] evaluates the conditional of an if-then-else expression and rules [If$_2$, If$_3$] take the appropriate branch.

3.2. Core of *MAC*

We now extend this standard calculus with the security primitives of **MAC** as shown in Fig. 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 *Labeled* $\ell\ \tau$ represents a (possibly side-effect free) resource, which annotates with the security level ℓ a value $t :: \tau$ wrapped in *Labeled*. For example, *Labeled* 42 :: *Labeled L 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 Appendix B. The actual **MAC** implementation handles more labeled resources and provides a uniform implementation for them [53].[7] The constructor *Labeled* is not available to the user, who can only use *label* and *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 *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 *unlabel* live in the *MAC* monad and the typing rules in Fig. 10 ensure that the label of the resource is compatible with the security level of the current computation, as explained in the previous section. We explain the relation between those typing rules and their corresponding type signatures given as Haskell API in Fig. 3 as follows. The typing rules in Fig. 10 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

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

[7] In our conference version [72,71], we follow the original **MAC** paper [53] and represent all labeled resources using the same labeled data type *Res* t :: *Res* $\ell\ \tau$, where $t :: \tau$ determines the kind of resource. For example *Res* (*Id* 42) :: *Res* ℓ (*Id 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.

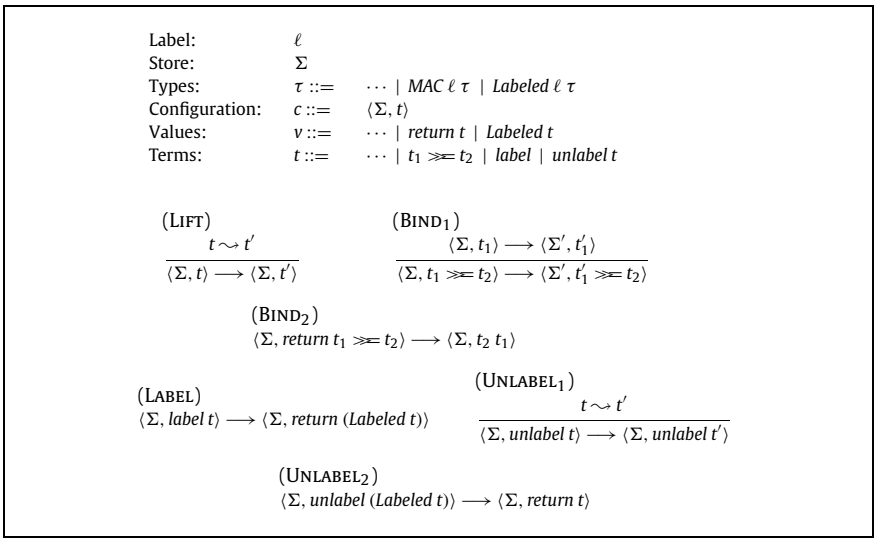


Fig. 9. Core of **MAC**.

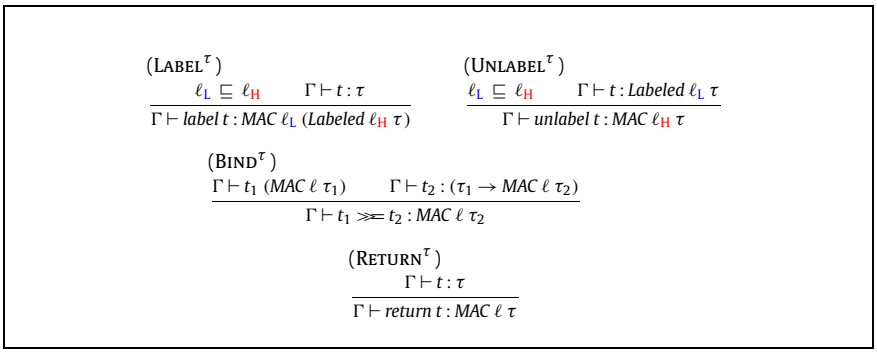


Fig. 10. Typing rules for the core of **MAC**.

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 [73]. Besides those primitives, computations are created using the standard monad operations *return* and $\gg\!=$. The primitive *return* lifts a term in the monad at security level ℓ by means of typing rule [Return$^\tau$]. Unlike the Dependency Core Calculus (DCC) [1], secure computations at different security levels do not mix in **MAC**: the typing rule [Bind$^\tau$] prevents that from happening—note the same label ℓ is expected both in the types of t_1 and t_2. Just like rules [Label$^\tau$, Unlabel$^\tau$], the typing rules [Return$^\tau$, Bind$^\tau$] are type scheme rules, i.e., there is such a rule for each label $\ell \in \mathscr{L}$. For easy exposition, in the following we give the type of **MAC**'s constructs as Haskell APIs.

We explicitly distinguish pure-term evaluation from top-level monadic-term evaluation, similarly to [66]. The extended semantics is represented as the relation $c_1 \longrightarrow c_2$, which extends $\rightsquigarrow$ via [Lift]. The semantics rules in Fig. 9 are fairly straight-forward and follow the pattern seen in the pure semantics, where some *context-rules*, e.g. [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 $\mathit{passwd}^{\mathsf{MAC}} :: \mathit{Labeled}\ H\ \mathit{String} \to \mathit{MAC}\ H\ ()$—note that the label of the computation is H, in order to unlabel the

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

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

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

Fig. 12. Primitive *join*.

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

Fig. 13. Ill-typed ($L \not\equiv H$).

password and hash it. (We treat password hashes as confidential data as well, because they could enable *offline* dictionary attacks otherwise.) 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., $putStrLn^{MAC} :: String \rightarrow MAC\ L\ ()$. Fig. 11 shows the code of the discussed routine. Observe that $putStrLn^{MAC}$ "Saving new password" $:: MAC\ L\ ()$ and $passwd^{MAC}\ pwd :: MAC\ H\ ()$ 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 Fig. 13 is rejected as ill-typed. Programs that handle data and channels with heterogeneous labels necessarily involve *nested MAC* $\ell\ a$ 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 complicates the programming model of **MAC** and hinders its applicability.[8] For example, *savePwd lpwd* requires to run the public computation to completion first, and then execute the resulting sensitive computation. We recognize this pattern of returning nested computations as a static version of a problem known in dynamic systems as *label creep* [57,15]—which occurs when the context gets tainted to the point that no useful operations are allowed anymore. In a *sequential setting*, **MAC** provides the primitive *join*,[9] which alleviates this problem by safely *integrating* more sensitive computations into less sensitive ones.

4.1. Primitive join

Fig. 12 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 7.5, programs written using the monadic API, *label*, *unlabel*, and *join* satisfy *progress-insensitive noninterference* (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., [46,60,23]), where the most effective manner to exploit termination takes exponential time in the size (of bits) of the secret [2].

In the semantics, Fig. 14 extends terms with the new primitive *join t*. Rule [Join] formalizes the semantics of *join* using big-step semantics—similar to other work [65,53], 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.[10]

Revisited example By replacing *return* with *join*, we can simplify the program *savePwd* from the previous section: compare the two versions of the program in Fig. 11 (using *return*) and in Fig. 15 (using *join*). In Fig. 15 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.

[8] 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.

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

[10] We refrain from using *label* t' because we will soon add exceptions to secure computations.

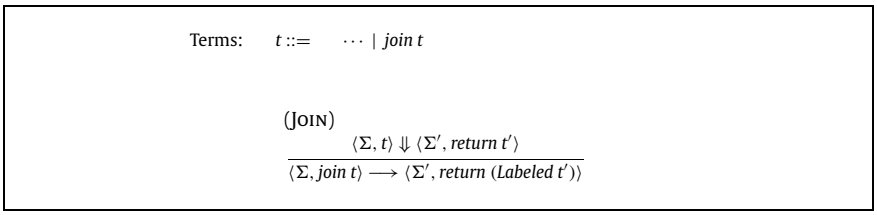


Fig. 14. Calculus with *join*.

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

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 Fig. 15. If primitive $passwd^{\mathsf{MAC}}$ 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 $passwd^{\mathsf{MAC}}$ 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., $passwd^{\mathsf{MAC}}$, 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 Fig. 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.

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, *throw* t, and *catch* $t_1\ t_2$—see Fig. 17. Term *throw* t aborts the current *MAC* computation with exception t, see rule [$\textsc{Bind}_\chi$]. Term *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 [66,28]. Observe that type signatures in Fig. 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 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.[11] In our calculus, *join* is the only primitive that combines computations with different labels and thus is potentially vulnerable to this attack.

[11] We refer interested readers to [53] for further details about this attack.

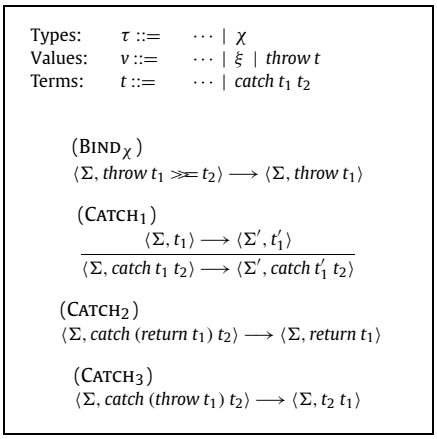


Fig. 17. Exception handling primitives.

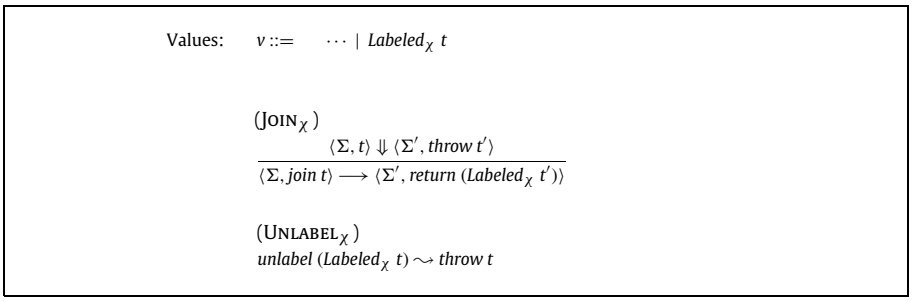


Fig. 18. Secure interaction between *join* and exceptions.

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 [66,28].

Fig. 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 program *savePwd* in the example from Fig. 15. The program prints `"Password Saved"` even though $passwd^{MAC}$ might have actually failed: it would be insecure to do otherwise! The only way to observe and recover from a failure of $passwd^{MAC}$, without compromising security, is to explicitly surround it with a *catch* block, i.e., *catch* ($passwd^{MAC}$ *pwd*) *handler*, and lift that computation with *join*.

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 passwords that are vulnerable to dictionary attacks. To do that, in Fig. 19, function *fetchDict* fetches a list of words from 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 *fetchDict* `"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., *Ref L* (*Map String* [*String*]). When the language *lang* dictionary is needed, the function reads the cached table (*dicts*) from the reference (*read r*) and checks if it has already been fetched (*lookup lang dicts*). If it is a hit (case *Just 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 (*write* (*insert dict dicts*) *r*) and returned.

```
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*.

```
data Ref ℓ τ
new :: ℓL ⊑ ℓH ⇒ τ → MAC ℓL (Ref ℓH τ)
read :: ℓL ⊑ ℓH ⇒ Ref ℓL τ → MAC ℓH τ
write :: ℓL ⊑ ℓH ⇒ τ → Ref ℓH τ → MAC ℓL ()
```

Fig. 20. API for references.

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

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

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

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

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

Fig. 21. MAC with references.

6.1. Calculus

Fig. 21 extends the calculus with mutable references, another feature available in **MAC**. Memory is compartmentalized into isolated labeled segments,[12] one for each label of the lattice, and accessed exclusively through the store Σ. A memory in the category *Memory* ℓ 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 $\Sigma(\ell)[n] := t$ 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 *unlabel*—see Fig. 20. A reference is represented as a value $\mathit{Ref}\ n :: \mathit{Ref}\ \ell\ \tau$ where n is an address,[13] 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

[12] 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.

[13] **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.

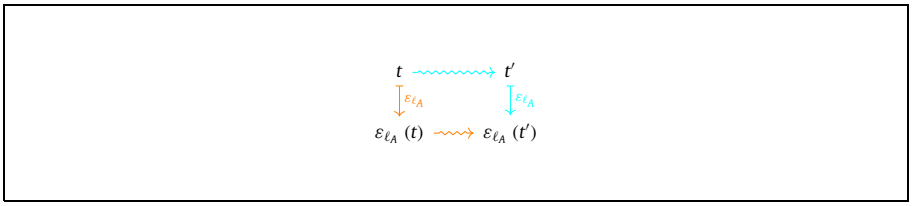


Fig. 22. Single-step simulation.

$|t_s|$ 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 gives an overview of the proof technique (*term erasure*), Section 7.2 describes *two-steps erasure*, a novel technique that overcomes some shortcomings of vanilla *term erasure*, Section 7.3 defines the erasure function and Section 7.5 concludes with the *progress-insensitive noninterference* theorem (PINI).

7.1. Term erasure

Term erasure is a proof technique to prove noninterference in functional programs. It was firstly introduced by Li and Zdancewic [35] and then used in a subsequent series of work on information-flow libraries [54,65,66,63,25]. 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 Fig. 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 sensitive data that has been leaked into 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. Two steps erasure

Unfortunately, the simulation property is often a too strong requirement: there are primitives of **MAC** that do not leak intuitively, and yet there is no definition of ε_{ℓ_A} that respects the commutativity of their simulation diagram. Typically, the erasure function erases either too much and breaks simulation of some other steps, or too little and some secret data remains after the erased term steps (see Section 10.1 for several concrete examples). To formally prove that those primitives are in fact secure, we have devised a technique called *two-steps erasure*, which performs erasure in two steps—a novel approach if compared with previous papers [64]. Rather than being a pure syntactic procedure, erasure is also performed by additional evaluation rules, triggered by special constructs introduced by the erasure function. Erasure occurs in two stages following the orange path in Fig. 22, firstly by rewriting a problematic primitive to an ad hoc construct (along the vertical solid arrow), secondly through the reduction step of that construct (along the horizontal curly arrow). In the following, we apply two-steps erasure systematically to gain the extra flexibility needed to prove that several advanced primitives, such as *new*, *write*, *join*, satisfy *single-step simulation*.

7.3. 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. Ground values (e.g., *True*) are unaffected by the erasure function and, for most terms, the function is homomorphically applied, e.g., $\varepsilon_{\ell_A}(t_1\ t_2 :: \tau) = \varepsilon_{\ell_A}(t_1 :: \tau' \to \tau)\ \varepsilon_{\ell_A}(t_2 :: \tau')$. Fig. 23 shows the definition of the erasure functions for the interesting cases. The *content* of a resource of type *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.[14] Similarly the erasure function rewrites the argument of *label* to $\bullet$, if it gets labeled with a

[14] The special term $\bullet$ can have any type τ. We give the typing rules for the extended calculus in Fig. C.52 in Appendix C.

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

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

Fig. 23. Term erasure for labeled values.

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

(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.

sensitive label or otherwise erased homomorphically. Observe that this definition respects the commutativity of the diagram in Fig. 22 for rule [Label].

Fig. 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 Fig. 24a. It is worth pointing out that 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*,[15] but aggressively erased to $\bullet$ as shown in Fig. 24a if paired with a store in a configuration. Intuitively the term alone is just the *description* of a secure computation,[16] which can be executed only if paired with a store in a configuration. 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 Fig. 24b. The erasure function collapses sensitive memories completely by rewriting them to $\bullet$ and erase non-sensitive ones homomorphically. Fig. 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 our *two-steps erasure* technique. Specifically the erasure function replaces constructs *new* and *write* with special constructs $\mathit{new}_\bullet$ and $\mathit{write}_\bullet$, whose semantics simulates that of the original terms with a *no-operation*—see Fig. 25. In particular rule [New$_\bullet$] leaves the store Σ unchanged (the argument to $\mathit{new}_\bullet$ is ignored), and returns a dummy reference with address $\bullet$. The same principle applies to $\mathit{write}_\bullet$. Rule [Write$_\bullet$1] evaluates the second argument to a reference, simulating rule [Write$_1$] and [Write$_\bullet$2] skips the write and just returns unit, simulating rule [Write$_2$]. The presence of these rules ensures that any sensitive write operation can be simulated in a lock-step fashion.

[15] This is different from the conference version of this work [72], where $\varepsilon_{\ell_A}(\mathit{MAC}\ \ell_H\ \tau :: t) = \bullet$ if $\ell_H \not\sqsubseteq \ell_A$. Erasing such terms homomorphically simplifies the formalization.

[16] Observe that in [72] 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.

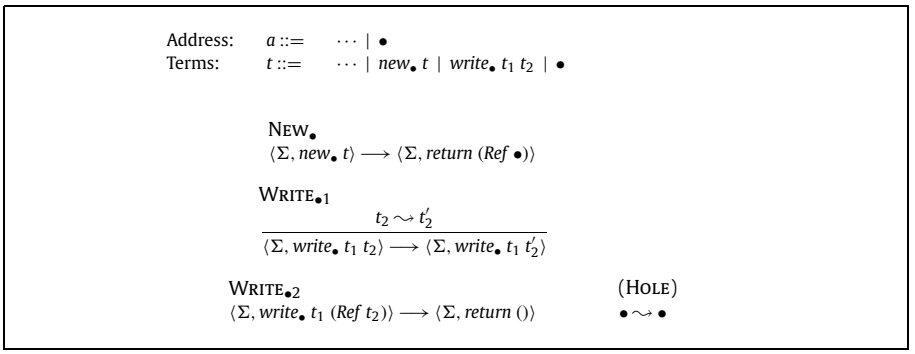


Fig. 25. Semantics of $\bullet$, $new_\bullet$ and $write_\bullet$.

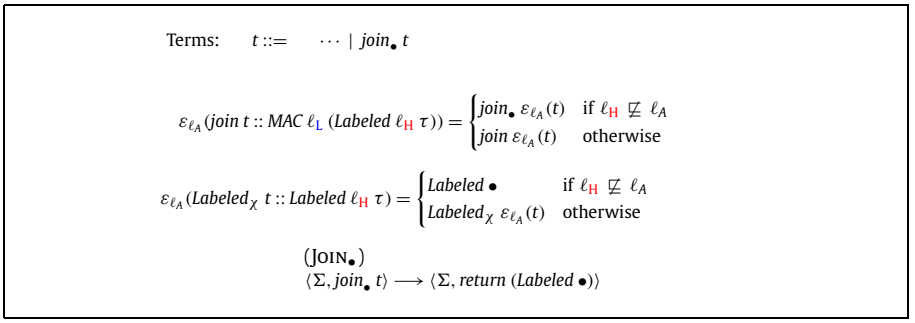


Fig. 26. Erasure of *join* and $Labeled_\chi$ and semantics of $join_\bullet$.

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

Fig. 26 shows the erasure function for the remaining terms of the sequential calculus, that is *join* and $Labeled_\chi$. Using the same technique that we have described previously, we replace *join* with special term $join_\bullet$, when it is used to run a sensitive computation ($\ell_H \not\sqsubseteq \ell_A$). Erasure is then performed by means of rule [JOIN$_\bullet$], which immediately returns a dummy labeled value ($Labeled\ \bullet$) 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 Fig. 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 $Labeled_\chi$ with *Labeled*, thus ensuring that rule [JOIN$_\bullet$] simulates rule [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 [UNLABEL$_\chi$] operates on labeled expressions containing exceptions. In this case, if the labeled exception is not visible to the attacker, then *unlabel* must be performed in a non-visible computation as well, due to the typing rules. Operation *unlabel* then gets rewritten to $\bullet$ and the step is then simulated by rule [HOLE] instead. As a result of that, and unlike the approach taken by Stefan et al. in [66], there are no sensitive labeled exceptions in erased terms.

7.4. Discussion

Term erasure We prove the single-step simulation directly over the small-step reduction relation. Instead, other works [35,54,65,66,63,25] prove the simulation by relating operational semantics step reductions (upper part in Fig. 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 [66]. 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 noninterference (e.g., [4,24]). 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 [25,64], 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. However, this memory model assumes infinite memory, since reference allocation never fails. This assumption is not realistic for actual systems, where physical resources such as memory are finite.[17] We conjecture that this gap between **MAC** and the model presented here, i.e., memory exhaustion, constitutes a covert channel that can be used to leak secrets with the same bandwidth as the termination covert channel [2]. In the conference version of this work [72], 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 $|\bullet| = \bullet$ 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. A non split-memory model requires some care when proving noninterference, and in fact, we have identified problems with the proofs in manuscripts and articles related to **LIO** [65,66]. We refer interested readers to Appendix B of our conference version [72] for details.

7.5. Progress-insensitive noninterference

The sequential calculus that we have presented satisfies *progress-insensitive noninterference*. 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 (Sketch) 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$_\chi$], which are simulated by rules [New$_\bullet$, Write$_{\bullet 1}$, Write$_{\bullet 2}$, Join$_\bullet$].

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

Proof By standard structural induction on the reductions.[18]

Before stating progress-insensitive noninterference, 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 noninterference.

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

[17] Unrestricted access to shared resources often constitutes a covert channel in concurrent systems. The resources can be either hardware (e.g., physical memory, caches [62] and multi-core processors) or software, such as components of the run-time system of a high-level language. These include the scheduler [55,56] and the garbage collector [48] or the language evaluation strategy such as lazy evaluation [70,14].

[18] Symbol $\equiv$ denotes equivalence up to alpha equivalence in the calculus with named variables. In our mechanized proofs we use Bruijn indexes and we obtain syntactic equality.

```
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.

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

Fig. 28. Attack magnification.

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

Fig. 29. API for concurrency.

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* ℓ computations provides attackers with new means to bypass security checks.

8.1. Termination attack

In Section 7, we have proved that the sequential calculus satisfies *progress-insensitive* noninterference, 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*.[19] 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 Fig. 27. We assume that there exists a function $print^{MAC}$ which prints an integer on a public channel. Observe how function *leak* may suppress subsequent public events with infinite loops.

Unfortunately concurrency magnifies the bandwidth of the termination covert channel to be linear in the size (of bits) of secrets [63],[20] 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 Fig. 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.).

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 [63]. 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 Fig. 29. Informally, it is secure to spawn sensitive computations (of type *MAC* ℓ_H ()) from non-sensitive ones (of type *MAC* ℓ_L ()) because that decision depends on data at level ℓ_L, which is no more sensitive ($\ell_L \sqsubseteq \ell_H$). From now on, we call *sensitive*

[19] If the physical execution time of a program depends on the value of the secret, then an attacker with an arbitrary precise stopwatch can deduce information about the secret by timing the program. This covert channel is known as the *external timing covert channel* [10,20]. This article does not address the external timing covert channel, which is a harder problem and for which mitigation techniques exist [3,75,76].

[20] Furthermore, the presence of threads introduce the *internal timing covert channel* [61], 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 :	$\Phi ::=$	$(\ell : \mathit{Label}) \to (\mathit{Pool}\ \ell)$
Thread Pool ℓ:	$t_s ::=$	$[\,] \mid t : t_s$
Configuration:	$c ::=$	$\langle \omega, \Sigma, \Phi \rangle$
Sequential Event ℓ:	$s ::=$	$\varnothing \mid \mathit{fork}(t)$
Concurrent Event ℓ:	$e ::=$	$\mathit{Step} \mid \mathit{Stuck} \mid \mathit{Done} \mid \mathit{Fork}\ \ell\ n$
Terms:	$t ::=$	$\cdots \mid \mathit{fork}\ t$

(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.

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

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

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

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

(*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

Fig. 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 Fig. 30a. Threads are secure computations of type *MAC* ℓ () and are organized in isolated thread pools according to their security label. A pool t_s in the category *Pool* ℓ 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 [53], we refer to Appendix B for details.

Relation $c_1 \hookrightarrow c_2$ denotes that concurrent configurations c_1 steps to c_2. Fig. 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* ℓ 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 Fig. 31 shows the interesting rules of the decorated semantics. Rule [SFork] is the only rule that explicitly generates event $\mathit{fork}(t)$ 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 Fig. 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

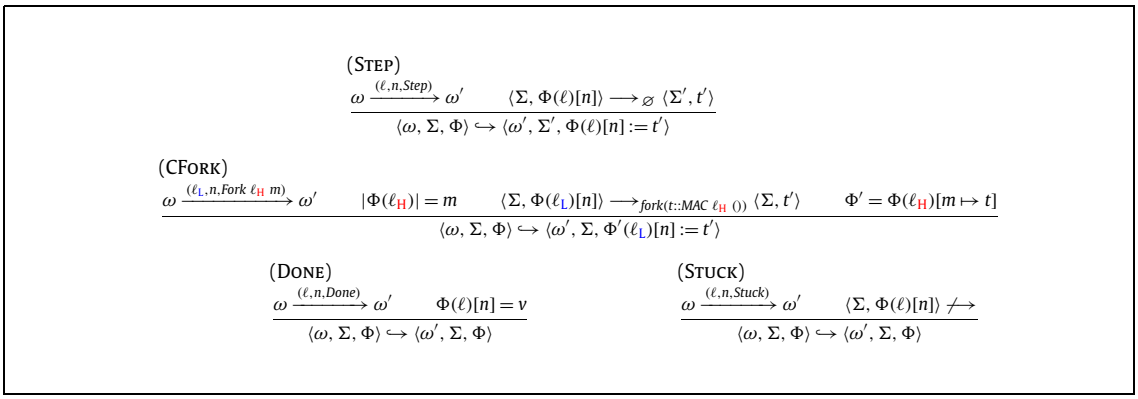


Fig. 32. Concurrent Semantics.

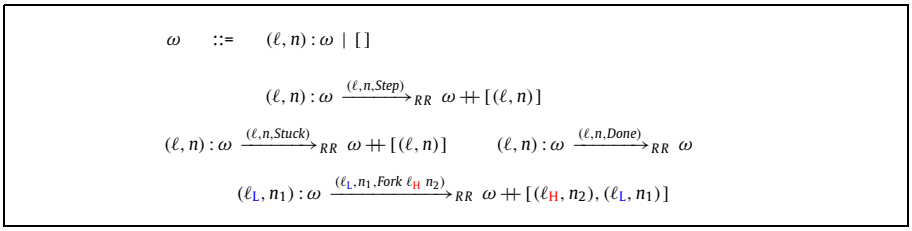


Fig. 33. Round-robin scheduler.

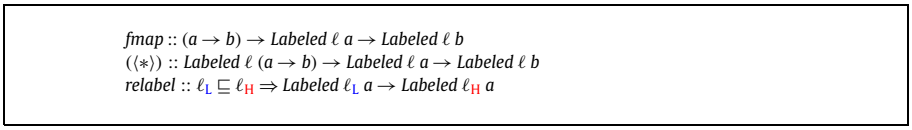


Fig. 34. API of Flexible labeled values.

[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* ℓ_H m, because the scheduled thread, identified by label ℓ_L and number n, spawns a child thread with type $t :: MAC\ \ell_H\ ()$, generating event $fork(t :: MAC\ \ell_H\ ())$. Observe that the spawned thread is uniquely identified by the label ℓ_H and number m and placed in pool $\Phi(\ell_H)$ in the *free* position $m = |\Phi(\ell_H)|$. 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

Fig. 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 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 *Stuck*), 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 $(\ell_L, n_1, Fork\ \ell_H\ n_2)$ 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 Fig. 34. Observe that these primitives operate on labeled data without using *label* and *unlabel*, thus avoiding incurring in the *no read-up* and *no write-down* restrictions and *irrespectively of their security level*.

```
isShort :: Labeled H String → Labeled H Bool
isShort = fmap (λpwd → |pwd| ≤ 5)
```

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

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. *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 Fig. 34. For instance, *fmap* $(+1)$ *d*, where *d* :: *Labeled H 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 *relabel* and $(\langle * \rangle)$. Primitive *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. Infix operator $(\langle * \rangle)$ supports function application within a labeled value, i.e. it allows to feed functions wrapped in a labeled value (*Labeled* ℓ $(a \to b)$ with arguments also wrapped (*Labeled* ℓ *a*), where aggregated results get wrapped as well (*Labeled* ℓ *b*).

Discussion In functional programming, operator $(\langle * \rangle)$ is part of the *applicative functors* [39] 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* ℓ *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 *clearance*, 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 [74] as well as poison-pill attacks [28]. While **MAC** does not yet currently support clearance, 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 Fig. 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 $|pwd| \leq 5$ 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 Fig. 35 with the homonym program in Fig. 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 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 Fig. 37a. The type system guarantees that the function is secure, because it has type *String* → *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 Fig. 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

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

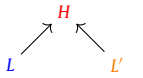
Fig. 36. Without *fmap pure* sensitive computations have an *impure* type.

```
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.

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

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

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

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

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

$$(\textsc{Relabel}_1)\quad \frac{t \rightsquigarrow t'}{\mathit{relabel}\ t \rightsquigarrow \mathit{relabel}\ t'}$$

$$(\textsc{Relabel}_2)\quad \mathit{relabel}\ (\mathit{Labeled}\ t) \rightsquigarrow \mathit{Labeled}\ t$$

Fig. 38. Calculus with flexible labeled values.

secret password, as shown in Fig. 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 Fig. 38, we extend our calculus with the primitives for flexible manipulation of labeled values, discussed in the previous section. Firstly we add terms $\mathit{fmap}\ t_1\ t_2$, $t_1\ \langle * \rangle\ t_2$ and $\mathit{relabel}\ t$, whose types correspond to those given in Fig. 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 are secure both in the concurrent and sequential calculus, where their semantics must be adjusted to handle exceptional values as well, i.e., constructor $\mathit{Labeled}_\chi$, which is not present in the concurrent calculus. We refer to Appendix A for more details.

Discussion The API of flexible labeled values shown in Fig. 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. Fig. 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

```
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*.

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

a Erasure for concurrent configuration.

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

(b) Erasure for thread pool.

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

(c) Erasure of fork.

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

(d) Erasure for *sequential* fork event.

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

(e) Erasure for *concurrent* fork event.

Fig. 40. Erasure function for concurrent calculus.

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. **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 noninterference*. Section 10.1 extends the erasure function for the concurrent calculus and for flexible labeled values. To obtain a parametric proof of noninterference, we assume certain properties about the scheduler. Specifically, our proof is valid for deterministic schedulers which fulfill progress and noninterference 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 noninterference 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

Fig. 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 (Fig. 40a). Similarly to store Σ, pool map

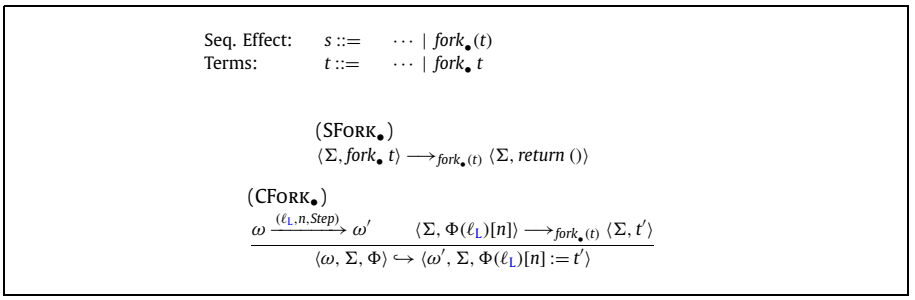


Fig. 41. Sequential and concurrent semantics of $fork_\bullet$.

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 Fig. 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 Fig. 40c. Sequential fork-events are erased similarly in order to ensure simulation, i.e., the erasure function rewrites $fork(t)$ to $fork_\bullet(\varepsilon_{\ell_A}(t))$ when t is sensitive—see Fig. 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 (Fig. 40e). In this case it rewrites event $Fork\ \ell_H\ n$ to *Step* whenever $\ell_H \not\sqsubseteq \ell_A$—the other events are not affected by the erasure function. In the sequential calculus $fork_\bullet$ is reduced by rule [SFork$_\bullet$], defined in Fig. 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 $MAC\ \ell_L\ ()$ forks a sensitive thread $MAC\ \ell_H\ ()$.

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*, which upgrades the security level of a labeled term. A public number, e.g., $Labeled\ 42 :: Labeled\ L\ Int$, should be treated as secret when in the context of relabeling, e.g., $relabel\ (Labeled\ 42) :: Labeled\ H\ Int$. Doing otherwise, i.e., erasing the term homomorphically, breaks simulation because sensitive data produced by *relabel* remains *after* erasure. For example $relabel\ (Labeled\ 42)$ is homomorphically erased to $relabel\ \varepsilon_L(Labeled\ 42 :: Labeled\ L\ Int)$ which reduces on the orange path in Fig. 22 to $Labeled\ 42 \not\equiv Labeled\ \bullet$, obtained on the cyan path by $\varepsilon_L(Labeled\ 42 :: Labeled\ H\ Int)$, 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 $relabel\ t :: Labeled\ H\ \tau$, where $t :: Labeled\ L\ \tau$. 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 $\varepsilon_L(relabel\ t :: Labeled\ H\ \tau) = relabel\ \varepsilon'_L(t :: Labeled\ L\ \tau)$, 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 [35,54,65,66,25], 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** [65,63] which lead to breaking the one-step simulation—see details in Appendix A of [72]. 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 Fig. 43. Even though the actual erasure is done by rule [Relabel$_\bullet$2], we still have to erase the *argument* of *relabel*, or else the erasure function would not be *homomorphic over substitution*. Simulation of the context rule [Relabel$_\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 [$\langle * \rangle_3$].[21] Following the orange path and applying the erasure

[21] In our conference version [71], 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.

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

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

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

Fig. 42. Erasure of flexible labeled values.

Terms: $t ::= \ldots \mid \mathit{fmap}_\bullet\ t_1\ t_2 \mid t_1\ \langle * \rangle_\bullet\ t_2 \mid \mathit{relabel}_\bullet\ t$

$$(\textsc{Fmap}_\bullet)\quad \mathit{fmap}_\bullet\ t_1\ t_2 \rightsquigarrow (\mathit{Labeled}\ \bullet)\ \langle * \rangle_\bullet\ t_2$$

$$(\langle * \rangle_\bullet 1)\quad \frac{t_1 \rightsquigarrow t_1'}{t_1\ \langle * \rangle_\bullet\ t_2 \rightsquigarrow t_1'\ \langle * \rangle_\bullet\ t_2}$$

$$(\langle * \rangle_\bullet 2)\quad \frac{t_2 \rightsquigarrow t_2'}{(\mathit{Labeled}\ t_1)\ \langle * \rangle_\bullet\ t_2 \rightsquigarrow (\mathit{Labeled}\ t_1)\ \langle * \rangle_\bullet\ t_2'}$$

$$(\langle * \rangle_\bullet 3)\quad (\mathit{Labeled}\ t_1)\ \langle * \rangle_\bullet\ (\mathit{Labeled}\ t_2) \rightsquigarrow \mathit{Labeled}\ \bullet$$

$$(\textsc{Relabel}_\bullet 1)\quad \frac{t \rightsquigarrow t'}{\mathit{relabel}_\bullet\ t \rightsquigarrow \mathit{relabel}_\bullet\ t'}$$

$$(\textsc{Relabel}_\bullet 2)\quad \mathit{relabel}_\bullet\ (\mathit{Labeled}\ t) \rightsquigarrow \mathit{Labeled}\ \bullet$$

Fig. 43. Semantics of $\mathit{fork}_\bullet$, $\langle * \rangle_\bullet$ and $\mathit{relabel}_\bullet$.

function *homomorphically*, we get that $\varepsilon_L(\mathit{Labeled}\ t_1)\ \langle * \rangle\ \varepsilon_L(\mathit{Labeled}\ t_2)$, that is $(\mathit{Labeled}\ \bullet)\ \langle * \rangle\ (\mathit{Labeled}\ \bullet)$ which reduces to $\mathit{Labeled}\ (\bullet\ \bullet) \not\equiv \mathit{Labeled}\ \bullet$, 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 Fig. 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 Fig. 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 $\mathit{Labeled}\ \bullet$ to enforce the simulation property. Since *fmap* is defined in terms of $\langle * \rangle$, we likewise replace it with new node $\mathit{fmap}_\bullet$ and give its semantics in terms of $\langle * \rangle_\bullet$, see rule $[\textsc{Fmap}_\bullet]$. We remark that $\mathit{fmap}_\bullet$, $\langle * \rangle_\bullet$ and $\mathit{relabel}_\bullet$ 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 [38], 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 Fig. 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., $\ell_L \sqsubseteq \ell_A$ and $\ell_H \not\sqsubseteq \ell_A$. Furthermore, we call a 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.

(*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.

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.[22] "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 wish to make our proof modular, our model is parametric in the scheduler, which is considered in isolation from the thread pool. In this situation, we cannot predict how long the high threads are going to run, because the scheduler is decoupled from the thread pool. 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.

The relation $\omega_1 \approx_{\ell_A}^{(i,j)} \omega_2$ 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 noninterference 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 $\omega_1 \approx_{\ell_A}^{(i,j)} \omega_2$ 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 [22].

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

- *If* $j = 0$, *then* $\forall\, e'\ \exists\, \omega_2'\colon\ \omega_2 \xrightarrow{(\ell_L,n,e')} \omega_2'$.
- *If* $j > 0$, *then there exists* ℓ_H, n' *such that* $\forall\, e'\ \exists\, \omega_2'\colon\ \omega_2 \xrightarrow{(\ell_H,n',e')} \omega_2'$.

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,[23] 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, n'), 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 [25]. 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* $\omega_1 \approx_{\ell_A}^{(i,j)} \omega_2$, *then there exist* j' *such that* $j' < j$ *and* $\omega_1' \approx_{\ell_A}^{(i,j')} \omega_2'$.

[22] Different to our conference version [72], 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.

[23] In our conference version [72], the requirement expects the same event e in the other step, which is too strict. Intuitively an event *Fork* ℓ_H n contains a bit of secret information, namely the number n of secret threads, which could differ in the other run. It follows from *restricted simulation*, i.e., Requirement 1.ii, that the two events are in fact ℓ_A-equivalent, i.e., $e \approx_{\ell_A} e'$, defined as $\varepsilon_{\ell_A}(e) \equiv \varepsilon_{\ell_A}(e')$. Note that ℓ_A-equivalence captures this scenario precisely: *Fork* ℓ_H $n \approx_{\ell_A}$ *Fork* ℓ_H n', because $\varepsilon_{\ell_A}(\textit{Fork}\ \ell_H\ n) \equiv \varepsilon_{\ell_A}(\textit{Fork}\ \ell_H\ n') \equiv \textit{Step}$.

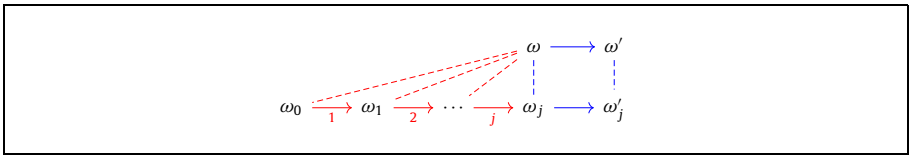


Fig. 44. Two (i,j)-ℓ_A-equivalent schedulers align in at most j steps.

$$[\,] \approx^{(0,0)}_{\ell_A} [\,] \qquad \frac{\omega_1 \approx^{(i,j)}_{\ell_A} \omega_2}{(\ell_L, n) : \omega_1 \approx^{(0,0)}_{\ell_A} (\ell_L, n) : \omega_2} \qquad \frac{\omega_1 \approx^{(i,j)}_{\ell_A} \omega_2}{(\ell_H, n) : \omega_1 \approx^{(i+1,j)}_{\ell_A} \omega_2}$$

$$\frac{\omega_1 \approx^{(i,j)}_{\ell_A} \omega_2}{\omega_1 \approx^{(i,j+1)}_{\ell_A} (\ell_H, n) : \omega_2}$$

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

Intuitively, the combination of Requirement 2 and 3 ensures that the two schedules will align *eventually*.[24] Fig. 44 highlights this intuition. The colored scheduler steps denote running either a secret (red for ℓ_H) or a public (blue for ℓ_L) thread respectively and the dashed line links ℓ_A-equivalent states. Given two initial scheduler states such that $\omega_0 \approx^{(i,j)}_{\ell_A} \omega$, 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 $\omega_1 \approx^{(i,j')}_{\ell_A} \omega$, where $j' < j$. After repeating this mechanism at most j times (j is strictly smaller after each step), we obtain $\omega_j \approx^{(i,0)}_{\ell_A} \omega$, 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* Fig. 45 defines annotated ℓ_A-equivalence. In particular, if $\omega_1 \approx^{(0,0)}_{\ell_A} \omega_2$ 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.*

10.3. Progress-sensitive Noninterference

The proof of progress-sensitive noninterference 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*

[24] In our conference version [72], 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.

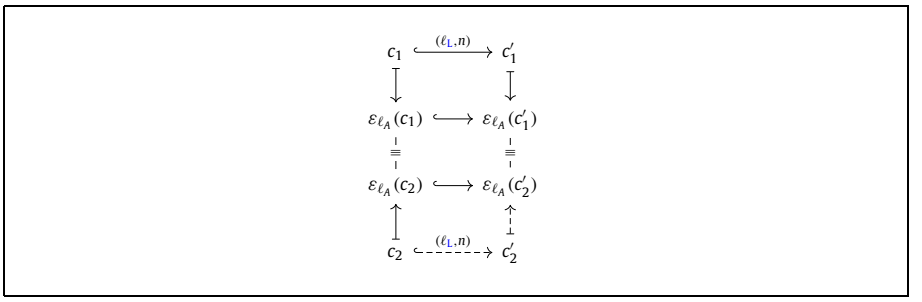


Fig. 46. 1-Step Progress.

- *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 noninterference 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 Fig. 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 Fig. 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 $\varepsilon_{\ell_A}(c_1) \hookrightarrow_{(\ell_L,m)} \varepsilon_{\ell_A}(c_1')$, by *restricted simulation* and we then *reconstruct* c_2' and the other step $c_2 \hookrightarrow_{(\ell_L,n)} c_2'$ from the step $\varepsilon_{\ell_A}(c_1) \hookrightarrow_{(\ell_L,m)} \varepsilon_{\ell_A}(c_1')$, $\varepsilon_{\ell_A}(c_1) \equiv \varepsilon_{\ell_A}(c_2)$ 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 noninterference). 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 [53]. 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 structurally similar to **MAC** but dynamically enforcing IFC [65]. The core calculus of **LIO**, i.e., side-effect free computations together with exception handling, has been modeled in the Coq proof assistant [66]. Different from our work, these mechanized proofs do not simplify the treatment of sensitive exceptions by masking them in erased programs. In parallel to [66], Breeze [28] 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 [9].

Parametricity Parametric polymorphism prevents a polymorphic function from inspecting its argument. In a similar manner, a non-interferent program cannot change its observable behavior depending on the secret. Researchers have explored further this deep and subtle connection by obtaining a translation from DCC [1] to System F in order to leverage on parametricity [69]. Shikuma and Igarashi [59] points out an error on such translation and gives a counterexample of a leaked translation. Recently, Bowman and Ahmed [11] provide a sound translation from DCC into System F.

Concurrency Considering IFC for a general scheduler could lead to refinements attacks. In this light, Russo and Sabelfeld provide termination-insensitive noninterference for a wide-class of deterministic schedulers [55]. Barthe et al. adopt this idea for Java-like bytecode [5]. 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 [25]. Similar to our work, their soundness proofs are parametric on deterministic schedulers and provide progress-sensitive noninterference 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 noninterference [26,32,27,49]. Mantel and Sudbrock propose a novel scheduler-independent trace-based information-flow control property for multi-threaded programs and identify the class of *robust scheduler*, which satisfy that condition [37]. While there are some similarities between the requirements of those robust schedulers and those discussed here in Section 10.2, that work assumes terminating threads, while our progress-sensitive noninterference theorem does not.

Security libraries Li and Zdancewic's seminal work [34] 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 [68]. Russo et al. implement the security library **SecLib** using a simpler structure than arrows [54], i.e. monads—rather than labeled values, this work introduces a monad which statically label side-effect free values. The security library **LIO** [64,63] enforces IFC for both sequential and concurrent settings dynamically. **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 [64]. Inspired by **SecLib** and **LIO**'s designs, **MAC** leverages Haskell's type system to enforce IFC [53] statically. Unlike **LIO**, data-dependent security policies cannot be expressed in **MAC**, due to its static nature. This limitation is addressed by **HLIO**, which provides a hybrid approach by means of some advanced Haskell's type-system features: IFC is statically enforced while allowing the programmers to defer selected security checks until run-time [16]. Several works have also investigated the use of dependent types to precisely capture the nature of data-dependent security policies [36,44,47,42].

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 [18].

Jaskelioff and Russo implements a library which dynamically enforces IFC using secure multi-execution (SME) [30]—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* [58], 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 [46] and Paragon [12] (based on Java), and FlowCaml [60] (based on Caml). The SPARK language presents a IFC analysis which has been extended to guarantee progress-sensitive non-inference [51]. JSFlow [23] 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) [7,8] and phrases them into a programming language setting. Although originated in the 70s, there are modern manifestations of this idea [74,33,43], applied to diverse scenarios, like the web [67,6] and mobile devices [31,13]. Due to its complexity, it is not surprising that OS-based MAC systems lack accompanying soundness guarantees or mechanized proofs—**seL4** being the exception [43]. 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 [7] 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 [67] 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 [19,74], 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 noninterference 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 [63,25]. 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.

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 [RELABEL$_\chi$]—see Fig. A.47. Rule [RELABEL$_{\bullet\chi}$] simulates rule [RELABEL$_\chi$] in the sequential setting, specifically when a public exceptional labeled value gets relabeled with a sensitive label (note that the resulting erased, i.e., *Labeled* $\bullet$, is sensitive and contains no exception). 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 non-exceptional values, i.e., $\varepsilon_L(Labeled_\chi\ t :: Labeled\ H\ \tau) = Labeled\ \bullet$. Therefore, we could not prove simulation for a non-strict applicative functor, since, crucially, it is sensitive to exceptions. While these behavior

$$(\textsc{Relabel}_\chi) \quad \textit{relabel}\ (\textit{Labeled}_\chi\ t) \rightsquigarrow \textit{Labeled}_\chi\ t \qquad (\textsc{Relabel}_{\bullet\chi}) \quad \textit{relabel}_\bullet\ (\textit{Labeled}_\chi\ t) \rightsquigarrow \textit{Labeled}\ \bullet$$

$$(\langle*\rangle_\chi 1) \quad \frac{t_2 \rightsquigarrow t_2'}{(\textit{Labeled}_\chi\ t_1)\ \langle*\rangle\ t_2 \rightsquigarrow (\textit{Labeled}_\chi\ t_1)\ \langle*\rangle\ t_2'}$$

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

$$(\langle*\rangle_\chi 3) \quad (\textit{Labeled}_\chi\ t_1)\ \langle*\rangle\ (\textit{Labeled}_\chi\ t_2) \rightsquigarrow \textit{Labeled}_\chi\ t_1$$

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

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

$$\begin{aligned}
&\textbf{data}\ \textit{MVar}\ \ell\ \tau\\
&\textit{newMVar} :: \ell_L \sqsubseteq \ell_H \Rightarrow \textit{MAC}\ \ell_L\ (\textit{MVar}\ \ell_H\ \tau)\\
&\textit{takeMVar} :: \textit{MVar}\ \ell\ \tau \rightarrow \textit{MAC}\ \ell\ \tau\\
&\textit{putMVar} :: \textit{MVar}\ \ell\ \tau \rightarrow \tau \rightarrow \textit{MAC}\ \ell\ ()
\end{aligned}$$

Fig. B.48. API of synchronization primitives.

could be simulated by an erasure function that preserves sensitive exceptions, i.e., $\varepsilon_L(\textit{Labeled}_\chi\ \tau :: \textit{Labeled}\ H\ \tau) = \textit{Labeled}_\chi\ \bullet$, it is an open question how to prove *single-step simulation* for *join*, specifically for rules [Join, Join$_\chi$].

Appendix 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 $\textit{MVar}\ \ell\ \tau$ denotes a labeled mutable location that is either empty or full and contains a term of type τ of security level τ. Fig. B.48 shows the API of basic synchronization primitives, based on *MVar*. Specifically, function *newMVar* creates an empty *MVar*. Function *takeMVar* empty a full *MVar* and returns its content or *blocks* otherwise. Function *putMVar* fills an empty *MVar* or blocks otherwise. Primitive *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 *takeMVar* and *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 *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 Fig. B.48 as the only secure option. Assume that primitive *takeMVar* had a completely unrestricted type, i.e., $\forall\ \ell_1\ \ell_2.\ \textit{MVar}\ \ell_1\ \tau \rightarrow \textit{MAC}\ \ell_2\ \tau$. Since *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 *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 *putMVar*.

B.1. Calculus

Fig. B.49 extends the concurrent calculus with synchronization primitives. A synchronization variable is represented as a value $\textit{MVar}\ n :: \textit{MVar}\ \ell\ \tau$ where n is an address,[25] 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.[26] 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 *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., $\textit{MVar}\ n$. Rule

[25] 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.

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

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

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

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

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

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

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

Fig. B.49. MAC with synchronization primitives.

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

(a) Erasure for memory cells.

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

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

b Erasure for *newMVar* and *MVar*.

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

[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 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., $\langle \Sigma, \mathit{putMVar}\ (\mathit{MVar}\ n)\ t\rangle \not\longrightarrow$ if $\Sigma(\ell)[n] \not\equiv \otimes$ and similarly *takeMVar blocks* if the cell is empty, i.e., $\langle \Sigma, \mathit{takeMVar}\ (\mathit{MVar}\ n)\rangle \not\longrightarrow$ 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 (Fig. B.50a). Applying the *two-steps erasure* technique, the erasure function replaces term *newMVar* with $\mathit{newMVar}_\bullet$, when it creates a sensitive synchronization variable—see Fig. B.50b. The erasure function rewrites the address of a synchronization reference to $\bullet$ if it points to a sensitive memory. Fig. B.51 shows rule [NewMVar$_\bullet$], which reduces term $\mathit{newMVar}_\bullet$, returns a dummy reference, i.e., $\mathit{MVar}\ \bullet$, and *skips* the write effect, leaving the store Σ unchanged. Observe that we do not need to replace *takeMVar* with a special term $\mathit{takeMVar}_\bullet$, because

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

$$(\textsc{NewMVar}_\bullet)\quad \langle \Sigma, newMVar_\bullet \rangle \longrightarrow \langle \Sigma, return\ (MVar\ \bullet) \rangle$$

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

$$(\textsc{Hole})\quad \Gamma \vdash \bullet : \tau$$

$$(\textsc{New}_\bullet)\quad \frac{\ell_L \sqsubseteq \ell_H \qquad \Gamma \vdash t : \tau}{\Gamma \vdash new_\bullet\ t : MAC\ \ell_L\ (Ref\ \ell_H\ \tau)}$$

$$(\textsc{Write}_\bullet)\quad \frac{\ell_L \sqsubseteq \ell_H \qquad \Gamma \vdash t_1 : \tau \qquad \Gamma \vdash t_2 : Ref\ \ell_H\ \tau}{\Gamma \vdash write_\bullet\ t_1\ t_2 : MAC\ \ell_L\ ()}$$

$$(\textsc{Join}_\bullet)\quad \frac{\ell_L \sqsubseteq \ell_H \qquad \Gamma \vdash t : MAC\ \ell_H\ \tau}{\Gamma \vdash join_\bullet\ t : MAC\ \ell_L\ (Labeled\ \ell_H\ \tau)}$$

$$(\textsc{Fork}_\bullet)\quad \frac{\ell_L \sqsubseteq \ell_H \qquad \Gamma \vdash t : MAC\ \ell_H\ ()}{\Gamma \vdash fork_\bullet\ t : MAC\ \ell_L\ ()}$$

$$(\textsc{Fmap}_\bullet)\quad \frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 : (Labeled\ \ell\ \tau_1)}{\Gamma \vdash fmap_\bullet\ t_1\ t_2 : Labeled\ \ell\ \tau_2}$$

$$(\langle * \rangle_\bullet)\quad \frac{\Gamma \vdash t_1 : Labeled\ \ell\ (\tau_1 \to \tau_2) \qquad \Gamma \vdash t_2 : (Labeled\ \ell\ \tau_1)}{\Gamma \vdash t_1\ \langle * \rangle_\bullet\ t_2 : Labeled\ \ell\ \tau_2}$$

$$(\textsc{Relabel}_\bullet)\quad \frac{\ell_L \sqsubseteq \ell_H \qquad \Gamma \vdash t : (Labeled\ \ell_L\ \tau)}{\Gamma \vdash relabel_\bullet\ t : Labeled\ \ell_H\ \tau}$$

$$(\textsc{NewMVar}_\bullet)\quad \frac{\ell_L \sqsubseteq \ell_H}{\Gamma \vdash newMVar_\bullet : MAC\ \ell_L\ (MVar\ \ell_H\ \tau)}$$

Fig. C.52. Typing rules for the extended calculus.

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 • or both non-sensitive and erased homomorphically.

Appendix C. Typing rules

Fig. C.52 gives the typing rules for the extended calculus, i.e., term • and other •-annotated terms used when applying two-steps erasure. The special term • can assume any type thanks to the typing rule [Hole]. The typing rule of each •-annotated term corresponds exactly to the typing rule of the unannotated terms. As a consequence of these typing rules, the erasure function is type-preserving, i.e., if $\Gamma \vdash t : \tau$ then $\varepsilon_{\ell_A}(\Gamma) \vdash \varepsilon_{\ell_A}(t) : \tau$.

References

[1] M. Abadi, A. Banerjee, N. Heintze, J. Riecke, A core calculus of dependency, in: Proc. ACM Symp. on Principles of Programming Languages, Jan. 1999, pp. 147–160.
[2] A. Askarov, S. Hunt, A. Sabelfeld, 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.
[3] A. Askarov, D. Zhang, A.C. Myers, Predictive black-box mitigation of timing channels, in: Proc. of the 17th ACM Conference on Computer and Communications Security, ACM, 2010.
[4] A. Banerjee, D.A. Naumann, Stack-based access control and secure information flow, J. Funct. Program. 15 (March 2005) 131–177.
[5] G. Barthe, T. Rezk, A. Russo, A. Sabelfeld, Security of multithreaded programs by compilation, ACM Trans. Inf. Syst. Secur. (Aug. 2009).
[6] L. Bauer, S. Cai, L. Jia, T. Passaro, M. Stroucken, Y. 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.
[7] D.E. Bell, L. La Padula, Secure Computer System: Unified Exposition and Multics Interpretation, Tech. Rep. MTR-2997, Rev. 1, MITRE Corporation, Bedford, MA, 1976.
[8] K.J. Biba, Integrity considerations for secure computer systems, 1977, ESD-TR-76-372.
[9] A. Bichhawat, V. Rajani, D. Garg, C. Hammer, Information Flow Control in WebKit's JavaScript Bytecode, Springer, Berlin Heidelberg, 2014.
[10] A. Bortz, D. Boneh, Exposing private information by timing web applications, in: Proc. of the 16th World Wide Web, ACM, 2007.

[11] W.J. Bowman, A. Ahmed, Noninterference for free, in: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1–3, 2015, 2015, pp. 101–113.
[12] N. Broberg, B. van Delft, D. Sands, Paragon for practical programming with information-flow control, in: APLAS, in: Lecture Notes in Computer Science, vol. 8301, Springer, 2013, pp. 217–232.
[13] S. Bugiel, S. Heuser, A.-R. 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.
[14] P. Buiras, A. Russo, Lazy programs leak secrets, in: Proc. Nordic Conference in Secure IT Systems, NORDSEC, Springer-Verlag, 2013.
[15] P. Buiras, D. Stefan, A. Russo, On dynamic flow-sensitive floating-label systems, in: Proceedings of the 2014 IEEE 27th Computer Security Foundations Symposium, CSF '14, IEEE Computer Society, Washington, DC, USA, 2014, pp. 65–79.
[16] P. Buiras, D. Vytiniotis, 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.
[17] D.E. Denning, P.J. Denning, Certification of programs for secure information flow, Commun. ACM 20 (7) (Jul. 1977) 504–513.
[18] D. Devriese, 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.
[19] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, R. 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.
[20] E.W. Felten, M.A. Schneider, Timing attacks on web privacy, in: Proc. of the 7th ACM Conference on Computer and Communications Security, CCS '00, ACM, 2000.
[21] J. Goguen, J. Meseguer, Security policies and security models, in: Proc of IEEE Symposium on Security and Privacy, IEEE Computer Society, 1982.
[22] J.A. Goguen, J. Meseguer, Unwinding and inference control, in: IEEE Symposium on Security and Privacy, April 1984, 1984.
[23] D. Hedin, A. Birgisson, L. Bello, A. Sabelfeld, JSFlow: tracking information flow in JavaScript and its APIs, in: Proc. of the ACM Symposium on Applied Computing, SAC '14, ACM, Mar. 2014.
[24] D. Hedin, D. Sands, Noninterference in the presence of non-opaque pointers, in: Proc. of the 19th IEEE Computer Security Foundations Workshop, IEEE Computer Society Press, 2006.
[25] S. Heule, D. Stefan, E.Z. Yang, J.C. Mitchell, A. Russo, IFC inside: retrofitting languages with dynamic information flow control, in: Conference on Principles of Security and Trust, POST, Springer, April 2015.
[26] K. Honda, V.T. Vasconcelos, N. Yoshida, Secure information flow as typed process behavior, in: Proc. of the 9th European Symposium on Programming Languages and Systems, Springer-Verlag, 2000.
[27] K. Honda, N. Yoshida, A uniform type structure for secure information flow, ACM Trans. Program. Lang. Syst. (Oct. 2007).
[28] C. Hritcu, M. Greenberg, B. Karel, B.C. Peirce, G. Morrisett, All your IFCexception are belong to us, in: Proc. of the IEEE Symposium on Security and Privacy, IEEE Computer Society, 2013.
[29] J. Hughes, Why functional programming matters, Comput. J. 32 (1984).
[30] M. Jaskelioff, A. Russo, Secure multi-execution in Haskell, in: Proc. Andrei Ershov International Conference on Perspectives of System Informatics, in: LNCS, Springer-Verlag, Jun. 2011.
[31] L. Jia, J. Aljuraidan, E. Fragkaki, L. Bauer, M. Stroucken, K. Fukushima, S. Kiyomoto, Y. 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, Sep. 2013.
[32] N. Kobayashi, Type-based information flow analysis for the π-calculus, Acta Inform. 42 (4) (Dec. 2005).
[33] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M.F. Kaashoek, E. Kohler, R. Morris, Information flow control for standard OS abstractions, in: Proceedings of ACM SIGOPS Symposium on Operating Systems Principles, SOSP '07, ACM, 2007.
[34] P. Li, S. Zdancewic, Encoding information flow in Haskell, in: Proc. of the IEEE Workshop on Computer Security Foundations, IEEE Computer Society, 2006.
[35] P. Li, S. Zdancewic, Arrows for secure information flow, Theor. Comput. Sci. 411 (19) (2010) 1974–1994.
[36] L. Lourenço, L. Caires, Dependent information flow types, SIGPLAN Not. 50 (1) (Jan. 2015).
[37] H. Mantel, H. Sudbrock, Flexible Scheduler-Independent Security, Springer, Berlin Heidelberg, 2010, pp. 116–133.
[38] S. Marlow, Parallel and Concurrent Programming in Haskell, O'Reilly, July 2013.
[39] C. Mcbride, R. Paterson, Applicative programming with effects, J. Funct. Program. (2008).
[40] S. Meurer, R. Wismüller, Apefs: an infrastructure for permission-based filtering of Android apps, in: A. Schmidt, G. Russello, I. Krontiris, S. Lian (Eds.), Security and Privacy in Mobile Information and Communication Systems, vol. 107, Springer, Berlin Heidelberg, 2012.
[41] E. Moggi, Notions of computation and monads, Inf. Comput. 93 (1) (1991) 55–92.
[42] J. Morgenstern, D.R. Licata, Security-typed programming within dependently typed programming, in: Proc. of the 15th ACM SIGPLAN International Conference on Functional Programming, ACM, 2010.
[43] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, G. Klein, seL4: from general purpose to a proof of information flow enforcement, in: 2012 IEEE Symposium on Security and Privacy 0, 2013.
[44] T. Murray, R. Sison, E. Pierzchalski, C. Rizkallah, Compositional verification and refinement of concurrent value-dependent noninterference, in: IEEE Computer Security Foundations Symposium, Lisbon, Portugal, Jun. 2016, pp. 417–431.
[45] A.C. Myers, JFlow: practical mostly-static information flow control, in: Proc. ACM Symp. on Principles of Programming Languages, Jan. 1999, pp. 228–241.
[46] A.C. Myers, L. Zheng, S. Zdancewic, S. Chong, N. Nystrom, Jif: Java information flow, http://www.cs.cornell.edu/jif, 2001.
[47] A. Nanevski, A. Banerjee, D. 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.
[48] M.V. Pedersen, A. Askarov, From trash to treasure: timing-sensitive garbage collection, in: 2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22–26, 2017, 2017, pp. 693–709.
[49] F. Pottier, A simple view of type-secure information flow in the π-calculus, in: Proc. of the 15th IEEE Computer Security Foundations Workshop, 2002, pp. 320–330.
[50] F. Pottier, V. Simonet, Information flow inference for ML, in: Proc. ACM Symp. on Principles of Programming Languages, Jan. 2002, pp. 319–330.
[51] W. Rafnsson, D. Garg, A. Sabelfeld, Progress-sensitive security for SPARK, in: Proceedings of the Engineering Secure Software and Systems – 8th International Symposium, ESSoS 2016, London, UK, April 6–8, 2016, 2016.
[52] I. Roy, D.E. Porter, M.D. Bond, K.S. McKinley, E. 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.
[53] A. 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.
[54] A. Russo, K. Claessen, J. Hughes, A library for light-weight information-flow security in Haskell, in: Proc. ACM SIGPLAN Symposium on Haskell, HASKELL '08, ACM, Sep. 2008.
[55] A. Russo, A. Sabelfeld, Securing interaction between threads and the scheduler, in: Proc. IEEE Computer Sec. Foundations Workshop, Jul. 2006, pp. 177–189.

[56] A. Russo, A. Sabelfeld, Security for multithreaded programs under cooperative scheduling, in: Proc. Andrei Ershov International Conference on Perspectives of System Informatics, in: LNCS, Springer-Verlag, Jun. 2006.
[57] A. Sabelfeld, A.C. Myers, Language-based information-flow security, IEEE J. Sel. Areas Commun. 21 (1) (Jan. 2003) 5–19.
[58] T. Schmitz, D. Rhodes, T.H. Austin, K. Knowles, C. Flanagan, Faceted dynamic information flow via control and data monads, in: F. Piessens, L. Viganò (Eds.), POST, in: Lecture Notes in Computer Science, vol. 9635, Springer, 2016.
[59] N. Shikuma, A. Igarashi, Proving noninterference by a fully complete translation to the simply typed lambda-calculus, in: Advances in Computer Science – ASIAN 2006. Secure Software and Related Issues, in: 11th Asian Computing Science Conference, Tokyo, Japan, December 6–8, 2006, 2006, pp. 301–315, Revised Selected Papers.
[60] V. Simonet, The flow Caml system, software release at http://cristal.inria.fr/~simonet/soft/flowcaml/, 2003.
[61] G. Smith, D. Volpano, Secure information flow in a multi-threaded imperative language, in: Proc. ACM Symposium on Principles of Programming Languages, POPL '98, 1998.
[62] D. Stefan, P. Buiras, E.Z. Yang, A. Levy, D. Terei, A. Russo, D. Mazières, Eliminating cache-based timing attacks with instruction-based scheduling, in: Proc. European Symp. on Research in Computer Security, in: LNCS, Springer-Verlag, 2013.
[63] D. Stefan, A. Russo, P. Buiras, A. Levy, J.C. Mitchell, 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.
[64] D. Stefan, A. Russo, J.C. Mitchell, D. Mazières, Flexible dynamic information flow control in Haskell, in: Proc. of the ACM SIGPLAN Haskell Symposium, HASKELL '11, 2011.
[65] D. Stefan, A. Russo, J.C. Mitchell, D. Mazières, Flexible dynamic information flow control in Haskell, in: Proceedings of the 4th ACM Symposium on Haskell, ACM, New York, NY, USA, 2011, pp. 95–106.
[66] D. Stefan, A. Russo, J.C. Mitchell, D. Mazières, Flexible dynamic information flow control in the presence of exceptions, Arxiv preprint arXiv:1207.1457, J. Funct. Prog., in press Cambridge University Press.
[67] D. Stefan, E.Z. Yang, P. Marchenko, A. Russo, D. Herman, B. Karp, D. Mazières, Protecting users by confining JavaScript with COWL, in: 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI 14, USENIX Association, Oct. 2014.
[68] T.C. Tsai, A. Russo, J. Hughes, A library for secure multi-threaded information flow in Haskell, in: Proc. IEEE Computer Security Foundations Symposium, CSF '07, Jul. 2007.
[69] S. Tse, S. Zdancewic, Translating dependency into parametricity, in: Proc. of the Ninth ACM SIGPLAN International Conference on Functional Programming, ACM, 2004.
[70] M. Vassena, J. Breitner, A. Russo, Securing concurrent lazy programs against information leakage, in: Proc. IEEE Computer Sec. Foundations Symposium, CSF '17, 2017.
[71] M. Vassena, P. Buiras, L. Waye, A. Russo, Flexible manipulation of labeled values for information-flow control libraries, in: Proceedings of the 12th European Symposium on Research in Computer Security, Springer, Sep. 2016.
[72] M. Vassena, A. Russo, On formalizing information-flow control libraries, in: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS '16, ACM, New York, NY, USA, 2016, pp. 15–28, http://doi.acm.org/10.1145/2993600.2993608.
[73] P. Wadler, 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, ACM, New York, NY, USA, 1989, pp. 60–76, http://doi.acm.org/10.1145/75277.75283.
[74] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, D. Mazières, Making information flow explicit in HiStar, in: Proc. of the 7th USENIX Symp. on Operating Systems Design and Implementation, USENIX, 2006.
[75] D. Zhang, A. Askarov, A.C. Myers, Predictive mitigation of timing channels in interactive systems, in: Proc. of the 18th ACM Conference on Computer and Communications Security, ACM, 2011.
[76] D. Zhang, A. Askarov, A.C. Myers, Language-based control and mitigation of timing channels, in: Proc. ACM Conference on Programming Language Design and Implementation, ACM, 2012.