On Formalizing Information-Flow Control Libraries

Marco Vassena
Chalmers University
41296, Gothenburg, Sweden
vassena@chalmers.se

Alejandro Russo
Chalmers University
41296, Gothenburg, Sweden
russo@chalmers.se

ABSTRACT

Many state-of-the-art IFC libraries support a variety of advanced features like mutuable data structures, exceptions, and concurrency, whose subtle interaction makes verification of security guarantees challenging. In this paper, we present a full-fledged, mechanically-verified model of **MAC**—a statically enforced IFC library. We describe three main insights gained during the formalization process. As previous libraries (e.g., **LIO** and **HLIO**), we utilize *term erasure* as the proof technique to show non-interference. This technique essentially states that the same public output should be produced if secrets are erased before or after program execution. Our first insight identifies challenges when the sensitivity of terms may depend on the context where they are used, thus affecting how they will be erased. This situation is not uncommon in **MAC** as well as other IFC libraries—in fact, we spot problems in the proofs of previous work. To deal with such complicated situations, we propose a novel erasure technique that performs erasure by additional evaluation rules, triggered by special-purpose constructs. Furthermore, we simplify reasoning about exception-aware primitives by removing sensitive exceptions from programs where secrets have been erased. We show progress insensitive non-interference for our sequential calculus and pinpoint sufficient requirements on the scheduler to prove progress-sensitive non-interference for our concurrent calculus. We prove that **MAC** is secure under a round-robin scheduler by simply instantiating our main scheduler-parametric theorem.

Keywords

Non-interference, Agda, Haskell

1. INTRODUCTION

Haskell is a pure functional language capable of providing information-flow control (IFC) via a library [Li and Zdancewic, 2006]. Different from other programming languages, Haskell type-system separates side-effect free from side-effectful computations—an essential feature to avoid harmful side-effects which may leak sensitive data. In recent years, researchers have increased their interest in such libraries, developing solutions that enforce IFC statically [Li and Zdancewic, 2010, Tsai et al., 2007, Russo et al., 2008, Russo, 2015], dynamically [Stefan et al., 2011, 2012a], and as a combination of both [Buiras et al., 2015]. Many of such libraries, namely **MAC** [Russo, 2015], **LIO** [Stefan et al., 2011], and **HLIO** [Buiras et al., 2015], bring ideas from Operating Systems research on Mandatory Access Control (MAC) [Bell and La Padula, 1976] into programming languages. These libraries secure programs even in presence of advance features like mutuable data structures (e.g., references), exceptions, concurrency, and synchronization primitives. This approach to IFC has proven competent, for instance, to protect sensitive data when third-party software is applied to Git repositories [Giffin et al., 2012]. From now on, we simply use the term libraries when referring to **MAC**, **LIO**, and **HLIO**.

The mentioned libraries structure computations using security monads [Abadi et al., 1999]—a special data type able to control the dissemination of sensitive data. As long as developers program against the libraries' API, the code is secure by construction. To provide non-interference [Goguen and Meseguer, 1982], these libraries enforce the *no read-up* and *no write-down* principles [Bell and La Padula, 1976]. The no read-up (no write-down) principle ensures that computations read (write) only from resources (to resources) *at most as sensitive* (*at least as sensitive*) as data found in scope.

Generally speaking, IFC libraries [Li and Zdancewic, 2010, Russo et al., 2008, Stefan et al., 2011] prove non-interference results by using the technique of *term erasure*: a program does not leak secrets if it produces the same observable outcome regardless of the fact that secrets are erased *before* or *after* execution. Such proofs frequently account for subtle interplay between programming languages features like, for instance, sub-computations and exceptions [Stefan et al., 2012b, Hritcu et al., 2013], security levels of variables and concurrency [Buiras et al., 2014], etc. It is precisely the complexity of features involved in IFC libraries, and their elusive interaction, which makes mechanized proofs, not only desirable, but needed to corroborate their security guarantees. To the best of our knowledge, there are no mechanized proofs of IFC libraries except for the core calculus of **LIO** [Stefan et al., 2012b], where no side-effectful operations are considered.

This paper presents a full formalization of **MAC** [Russo, 2015]—a security library which leverages Haskell type-

PLAS'16, October 24 2016, Vienna, Austria

ISBN 978-1-4503-4574-3/16/10. . . $15.00

DOI: http://dx.doi.org/10.1145/2993600.2993608

system to provide IFC—and some interesting insights gained from that. Many of them surpass **MAC** and pertain to **LIO** as well as **HLIO**[1]. In fact, our insights leads us to uncover some problems in **LIO**'s proofs and propose changes to repair its non-interference guarantees. This work aims not only to help identifying problems in existing IFC libraries, but also to assist library designers to correctly apply *term erasure* as a proof technique.

Our formalization respectively shows progress-insensitive and progress-sensitive non-interference for sequential and concurrent programs. For that, and similar to other IFC libraries, we define an *erasure* function on terms which maps sensitive data and computations to a special syntax node written as $\bullet$. Then, a simulation is established between the evaluation of the program and its *erased counterpart*—the simulation only captures programs which produce the same observable behavior. Our mechanized proofs for **MAC** provide us with the following insights:

▹ **Context-aware erasure** The decision of erasing terms might be context-dependent. For instance, erasing an argument in a multi-argument functions might depend on the value, or type, of some other arguments. Consider, for example, a function that respectively takes a number and a label, and stores the number in a fresh reference labeled with the given label. The decision to erase the first argument, i.e., the number, depends on the value of the second argument, i.e., the label. This dependency obstructs the definition of a sound *homomorphic* erasure function, complicating the analysis of security guarantees—we identify definitions which exhibit this problem in **LIO** [Stefan et al., 2011]. We propose a novel two-steps erasure technique to repair such cases.

▹ **Masking sensitive exceptions** In previous work, labeled exceptions are erased by erasing their content according to their label, but always preserving their exceptional state [Stefan et al., 2012b]. In contrast, we propose to 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! The simulation between terms and their erased counterparts guarantees that this rewriting is *sound*. Sensitive handling routines, the only routines which can distinguish exceptional from unexceptional sensitive values, are also erased and do not occur in erased programs either.

▹ **Scheduler requirements** When considering concurrent programs, we obtain a security proof which is valid for a wide-range of deterministic schedulers. We formally pin down sufficient requirements on the scheduler to guarantee progress-sensitive non-interference—a novel aspect if compared with previous work [Stefan et al., 2012a, Heule et al., 2015]. As an example, we instantiate our results with a round-robin scheduler (the scheduler used by Haskell runtime system).

We consider the insights above, together with our 4000 lines of mechanized proofs in Agda[2], the main contributions of this work. We furthermore describe some technical and novel aspects of our proofs, which we believe might come in handy to IFC researchers. In particular, our model (i) does not introduce any extra reduction relation, (ii) annotates concurrent transitions with threads' identifiers to obtain a scheduler-parametric non-interference proof, and (iii) partition memory and thread pools by security level to completely erased them when sensitive.

This paper is organized as follows. Section 2 formalizes the core of **MAC**, Section 3 presents the proof technique used to study the security guarantees, Sections 4 and 5 extend the calculus with exceptions and concurrency. Section 6 gives related work and Section 7 concludes.

[1]The formal guarantees of **HLIO** are simply reduced to those of **LIO**.

[2]Available at https://bitbucket.org/MarcoVassena/mac-agda

Label:	ℓ	
Types:	$\tau ::=$	$\mathit{Bool} \mid () \mid \tau_1 \to \tau_2$
		$\mathit{Id}\ \tau \mid \ \mid \mathit{MAC}\ \ell\ \tau \mid \mathit{Res}\ \ell\ \tau$
Values:	$v ::=$	$\mathit{True} \mid \mathit{False} \mid () \mid \lambda x.t$
		$\mid \mathit{MAC}\ t \mid \mathit{Res}\ t \mid \mathit{Id}\ t$
Terms:	$t ::=$	$v \mid t_1\ t_2 \mid \textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3$
		$\mid \mathit{return}\ t \mid t_1 \gg\!= t_2$
		$\mid \mathit{label} \mid \mathit{unlabel}\ t \mid \mathit{label}_\bullet$
		$\mid \mathit{join} \mid \mathit{join}_\bullet \mid \bullet$

Figure 1: Formal syntax for types, values, and terms.

2. THE CORE CALCULUS

This section formalizes the core of **MAC** as a simply typed call-by-name λ-calculus extended with booleans, unit values and monadic operations.

Calculus.

Figure 1 shows the formal syntax of the calculus, where meta variables ℓ, τ, v and t denote respectively labels, types, values, and terms. Most of these syntactic categories are self-explanatory with the exception of a few cases that we proceed to clarify. 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. We assume that labels form a lattice $(\mathscr{L}, \sqsubseteq)$. In examples we use the concrete classic two-point lattice with labels H and L denoting secret (high) and public (low) data respectively—where $H \not\sqsubseteq L$ is the only disallowed flow. Term MAC is the constructor of type $\mathit{MAC}\ \ell\ \tau$, which denotes a (possibly) side-effectful secure computation that handles information at sensitivity level ℓ and yields a result of type τ at the same security level. Constructor Res represents a labeled resource. Generally speaking, resources are sources and sinks of information: pure terms (e.g., number 42), a file, a reference, etc. The nature of the labeled resource is captured in its type. Data type $\mathit{Id}\ \tau$ is used to denote resources which do not trigger side-effects when manipulated, e.g., numbers or strings. For instance, $\mathit{Res}\ (\mathit{Id}\ 42) :: \mathit{Res}\ \ell\ (\mathit{Id}\ \mathit{Int})$ represents a resource labeled with ℓ, whose content is the number 42. We use Haskell notation $t :: \tau$ to denote that term t has type τ. By instantiating τ in $\mathit{Res}\ \ell\ \tau$ with different types (like we just did with Id), **MAC** is able to identify and securely provide operations on many kind of resources, e.g., $\mathit{Res}\ \ell\ (\mathit{IORef}\ \mathit{Int})$ for references to integers, $\mathit{Res}\ \ell\ (\mathit{Socket}\ \mathit{ByteStream})$ for network communication, and so on. Observe how **MAC** reuses the same data type for different kind of resources. Without loss of generality, next sections only consider *(pure) labeled expression*, i.e., labeled resources of type $\mathit{Res}\ \ell\ (\mathit{Id}\ \tau)$, which we abbreviate with the

$$
\begin{array}{l}
\mathit{return} :: \tau \rightarrow \mathit{MAC}\ \ell\ \tau \\
(\gg\!=)\ \ :: \mathit{MAC}\ \ell\ \tau_1 \rightarrow (\tau_1 \rightarrow \mathit{MAC}\ \ell\ \tau_2) \rightarrow \mathit{MAC}\ \ell\ \tau_2 \\
\mathbf{type}\ \mathit{Labeled}\ \ell\ \tau = \mathit{Res}\ \ell\ (\mathit{Id}\ \tau) \\
\mathit{label}\ \ \ :: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow \tau \rightarrow \mathit{MAC}\ \ell_{\mathrm{L}}\ (\mathit{Labeled}\ \ell_{\mathrm{H}}\ \tau) \\
\mathit{unlabel} :: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow \mathit{Labeled}\ \ell_{\mathrm{L}}\ \tau \rightarrow \mathit{MAC}\ \ell_{\mathrm{H}}\ \tau \\
\mathit{join} ::\ \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow \mathit{MAC}\ \ell_{\mathrm{H}}\ \tau \rightarrow \mathit{MAC}\ \ell_{\mathrm{L}}\ (\mathit{Labeled}\ \ell_{\mathrm{H}}\ \tau)
\end{array}
$$

Figure 2: API of core primitives.

(Return)

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash \mathit{return}\ t : \mathit{MAC}\ \ell\ \tau}$$

Figure 3: Type scheme rule for *return*.

(Label)

$$\mathit{label}\ t \rightsquigarrow \mathit{return}\ (\mathit{Res}\ (\mathit{Id}\ t))$$

(Unlabel)

$$\mathit{unlabel}\ (\mathit{Res}\ (\mathit{Id}\ t)) \rightsquigarrow \mathit{return}\ t$$

(Join)

$$\frac{t_1\ \Downarrow\ \mathit{MAC}\ t_2}{\mathit{join}\ t_1 \rightsquigarrow \mathit{return}\ (\mathit{Res}\ (\mathit{Id}\ t_2))}$$

Figure 4: Semantics for labeling operations.

type synonym $\mathit{Labeled}\ \ell\ \tau$. Constructors MAC and Res are part of **MAC**'s internals, therefore they are not available to users of the library and are not part of the surface syntax.

Terms.

Secure computations enjoy a monadic structure, i.e. they are built using the fundamental operations *return* and $\gg\!=$ (read as "bind"). The operation $\mathit{return}\ t$ produces a computation that returns term t 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. Operations *label* and *unlabel* create and read labeled expression within MAC computations, thus allowing terms of type $\mathit{MAC}\ \ell\ a$ to securely interact with such labeled resources. The primitive operator *join* securely embeds a sensitive computation into a less sensitive one. The special syntax nodes $\mathit{join}_\bullet$, $\mathit{label}_\bullet$ and $\bullet$ represent *erased terms* (explained in Section 3) and are used by our proof technique to examine the security guarantees of the calculus.

Types.

The typing judgment $\Gamma \vdash t : \tau$ denotes that term t has type τ assuming the typing environment Γ. All The typing rules are standard and thus omitted, except for $\bullet$ which can assume any type, i.e., $\Gamma \vdash \bullet : \tau$. For easy exposition, we describe the types of interesting constructs *return*, $(\gg\!=)$, *label*, *unlabel*, and *join* as Haskell APIs—see Figure 2. We explain their relation with traditional typing judgments by means of an example. The typing judgment of *return* is given in Figure 3. Note that rule [Return] is a *rule scheme*, i.e., there is such a judgment for every label $\ell \in \mathscr{L}$, where labels come from either type signatures or explicit type annotations in programs. In the API, what appears on the left-hand side of the symbol $\Rightarrow$ are *type constraints*, which are properties that must be statically fulfilled about the types that follow. To help readers, we indicate the relationship between labels in their subindexes, i.e., we use ℓ_{L} and ℓ_{H} to attest that $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$. Term *label* creates a new labeled expression, which is considered as a write operation from the security point of view. Consequently, the type constraint $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$ enforces the *no write-down* rule. It requires the label ℓ_{L} of the MAC computation to be no more confidential than ℓ_{H}, i.e., the label of the created labeled expression. Term *unlabel* performs a read operation, therefore, to comply with the *no read-up* rule, its type protects the confidentialy ℓ_{H} of the result produced by the MAC computation. To achieve that, type constraint $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$ ensures that the result of the computation may only involve unlabeled expressions ℓ_{L} which are, at most, as sensitive as ℓ_{H}. Lastly, *join* protects the result of a sensitive MAC computation inside a less sensitive one by the constraint $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$, avoiding the label creep problems in sequential programs [Russo, 2015]. We remark that these type constrains are built using *type classes*, a well-established feature of Haskell type system—thus, we omit the corresponding typing rules for **MAC**'s primitives. In what follows, we describe an example which illustrates **MAC**'s programming model, particularly the use of *label*, *unlabel*, and *join*.

Example.

The most common use of *label* is to classify data to be protected. As an example, consider a piece of Haskell code which simply asks for a password from the terminal.

```
putStrLn "Input your password?"
pwd ← getLine
...
```

At this point, the content of variable *pwd* should be handled with care in the rest of the program, which we symbolize with ellipsis (...). One way to protect *pwd* is by writing all password-related operations within **MAC**, where *pwd* is marked as sensitive data. For instance, the following code passes the password to a routine to check if the password is listed on dictionaries of commonly used passwords.

```
putStrLn "Input your password?"
pwd ← getLine
let lpwd = label pwd :: MAC L (Labeled H String)
run^MAC (lpwd >>= common)
```

Observe how *label* is used to mark *pwd* as sensitive by wrapping it inside a labeled expression of type $\mathit{Labeled}\ H\ \mathit{String}$. After that, the labeled password is passed to the function *common* by bind $((\gg\!=))$. (Function $\mathit{run}^{\mathsf{MAC}}$ simply runs the given MAC-computation.) Assuming that *common firstly* fetches online dictionaries, pre-processes them, and *then* inspects if the password appears in them, the type for such code could be $\mathit{common} :: \mathit{Labeled}\ H\ \mathit{String} \rightarrow \mathit{MAC}\ L\ (\mathit{MAC}\ H\ \mathit{Bool})$. The reason for having nested MAC-computations comes from the fact that *common* is handling information with both sensitivities, i.e., L and H. The outermost computation $(\mathit{MAC}\ L)$ is responsible to fetch the online dictionaries from the web, an action consid-

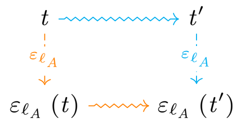

Figure 5: Single-step simulation.

ered observable. The inner computation (of type *MAC* H), on the other hand, arises from unlabeling *lpwd* (by using *unlabel*) in order to search the password in the fetched dictionaries. Clearly, while manageable for two security labels, if *common* were to handle information with many security labels, it could return a long sequence of nested *MAC*-computations. To mitigate this problem, *common* can apply *join* to "compress" its type to $common :: Labeled\ H\ String \rightarrow MAC\ L\ (Labeled\ H\ Bool)$, i.e., returning only one *MAC*-computation.

Semantics.

The small-step semantics of the the calculus is represented by the relation $t_1 \rightsquigarrow t_2$, which denotes that t_1 reduces to t_2. Most of the reduction rules are standard and thus omitted. Figure 4 shows the interesting rules for constructs *label*, *unlabel*, and *join*. Rule [Label] creates a labeled expression by wrapping it with constructs *Res* and *Id*. Dually, rule [Unlabel] returns the expression wrapped by constructs *Res* and *Id*. Rule [Join] formalizes the semantics of *join* using big-step semantics—similar to other work [Stefan et al., 2011, Russo, 2015], we restrict ourselves to terminating computations. The rule runs the sensitive computation and wraps the result into a labeled expression at the appropriate security level. Note that none of these rules involve any security check (labels are not even present in terms). Each flow of information have already been statically verified via type-checking. The rules of special nodes $label_\bullet$ and $join_\bullet$ will be given in Section 3.2. We only remark that node $\bullet$ reduces to itself according to rule [Hole], that is $\bullet \rightsquigarrow \bullet$.

3. TERM ERASURE

Term erasure is a proof technique to prove non-interference in functional programs. It was firstly introduced by Li and Zdancewic [Li and Zdancewic, 2010] and then used in a subsequent series of work on information-flow libraries [Russo et al., 2008, Stefan et al., 2011, 2012b,a, Heule et al., 2015]. The technique relies on an erasure function on terms, which we denote by ε_{ℓ_A}. This function essentially rewrites data above the attacker's security level, denoted by label ℓ_A, to the special syntax node $\bullet$. Once ε_{ℓ_A} is defined, the core of the proof technique consists of proving an essential relationship about the erasure function and reduction steps. The diagram in Figure 5 highlights this intuition. It shows that erasing sensitive data from a term t and then taking a step (orange path) is the same as firstly taking a step and then erasing sensitive data (cyan path), i.e., the diagram *commutes*. If term t leaks data whose sensitivity label is above ℓ_A, then erasing all sensitive data first and then taking a step might not be the same as taking a step and then erasing secret values—the leaked sensitive data in t' might remain in $\varepsilon_{\ell_A}(t')$ after all. From now on, we refer to this relationship as the *single-step simulation* between regular terms and erased ones.

$$\varepsilon_{\ell_A}(Res\ t :: Res\ \ell\ \tau) = \begin{cases} Res\ \varepsilon_{\ell_A}(t :: \tau) & \text{if } \ell \sqsubseteq \ell_A \\ Res\ \bullet & \text{otherwise} \end{cases}$$

$$\varepsilon_{\ell_A}(t :: MAC\ \ell\ \tau) = \bullet \text{ if } \ell \not\sqsubseteq \ell_A$$

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

Figure 6: Erasure function (interesting cases).

Discussion.

We prove the single-step simulation directly over the small-step reduction relation. Instead, other works [Li and Zdancewic, 2010, Russo et al., 2008, Stefan et al., 2011, 2012b,a, Heule et al., 2015] prove the simulation by relating small-step reductions (upper part in Figure 5) with reductions on a ℓ_A-indexed small-step relation of the form $t \rightsquigarrow_{\ell_A} \varepsilon_{\ell_A}(t')$, 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 runtime. 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 $\rightsquigarrow_{\ell_A}$.

3.1 Erasure function

We proceed to define the erasure function for our core 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., $\varepsilon_{\ell_A}(\textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3 :: ()) = \textbf{if}\ \varepsilon_{\ell_A}(t_1 :: Bool)\ \textbf{then}\ \varepsilon_{\ell_A}(t_2 :: ())\ \textbf{else}\ \varepsilon_{\ell_A}(t_3 :: ())$. Figure 6 shows the definition of the erasure functions for the interesting cases. The *content* of a resource of type $Res\ \ell\ \tau$ is erased homomorphically if ℓ is below the attacker's label ℓ_A, otherwise it is rewritten to $\bullet$. Secure computations of type $MAC\ \ell\ \tau$ are instead completely collapsed to $\bullet$ when ℓ is above the attacker's label and homomorphically erased otherwise. The erasure of *label* and *join* within observable computations, that is *MAC* computations with label ℓ_M such that $\ell_M \sqsubseteq \ell_A$, is non-standard. These cases deviate from the definitions seeing so far, i.e., either simply collapsing sensitive data to $\bullet$ or applying the erasure function homomorphically. Unfortunately, neither of these kinds of definitions guarantees single-step simulation in those cases.

To illustrate the challenge of erasing *label*, we consider an attacker at level L and the term $label\ t :: MAC\ L\ (Labeled\ H\ \tau)$. Observe that *label* t is executed by an observable *MAC* computation. In this case, the type

$$\varepsilon_{\ell_A}(label\ t :: MAC\ \ell_M\ (Labeled\ \ell\ \tau)) = \begin{cases} label\ \varepsilon_{\ell_A}(t) & \text{if } \ell \sqsubseteq \ell_A \\ label_\bullet\ \varepsilon_{\ell_A}(t) & \text{otherwise} \end{cases}$$

$$\varepsilon_{\ell_A}(join\ t :: MAC\ \ell_M\ (Labeled\ \ell\ \tau)) = \begin{cases} join\ \varepsilon_{\ell_A}(t) & \text{if } \ell \sqsubseteq \ell_A \\ join_\bullet\ \varepsilon_{\ell_A}(t) & \text{otherwise} \end{cases}$$

(LABEL•)
$label_\bullet\ t \rightsquigarrow return\ (Res\ \bullet)$

(JOIN•)
$join_\bullet\ t \rightsquigarrow return\ (Res\ \bullet)$

Figure 7: Two-steps erasure function.

$MAC\ L\ (Labeled\ H\ \tau)$ indicates that t is sensitive, therefore we should rewrite it to $\bullet$, i.e., $\varepsilon_L(label\ t) = label\ \bullet$. By doing so, however, the commutativity of the diagram in Figure 5 brakes. On the cyan path, we obtain that $label\ t \rightsquigarrow return\ (Res\ (Id\ t))$ (by rule [LABEL]) and that $\varepsilon_L(return\ (Res\ (Id\ t))) \equiv return\ \varepsilon_L(Res\ (Id\ t))$ (by applying erasure homomorphically to *return*), where $return\ \varepsilon_L(Res\ (Id\ t)) \equiv return\ (Res\ \bullet)$ (by erasure on sensitive labeled expressions—see Figure 6). In contrast, on the orange path, we have that $\varepsilon_L(label\ t) = label\ \bullet \rightsquigarrow return\ (Res\ (Id\ \bullet))$ (by [LABEL]). To adhere to commutativity, the terms at the end of both paths should be the same, which is not the case here, i.e., $return\ (Res\ \bullet) \not\equiv return\ (Res\ (Id\ \bullet))$. The problem arises from the fact that even after erasing *every* piece of sensitive information from *label* t, namely by rewriting t to $\bullet$, rule [LABEL] still produces the constructor *Id*, which instead gets erased on the cyan path. Observe that rule [LABEL] seems to intrinsically break simulation, regardless of the choice of the erasure function.

Term *join* t also raises a similar problem. Consider erasing a sensitive computation $t :: MAC\ H\ \tau$ embedded in a public one using *join*. On the orange path, we have that $\varepsilon_L(join\ t :: MAC\ L\ (Labeled\ H\ \tau)) \equiv join\ \varepsilon_L(t :: MAC\ H\ \tau)$ (by applying ε_{ℓ_A} homomorphically), which results in *join* $\bullet$ by erasure of sensitive computations (see Figure 6). Symbol $\bullet$ does not have a normal form by rule [HOLE], i.e., $\bullet \not\Downarrow MAC\ t'$, which prohibits the orange path from making a step since rule [JOIN] cannot be applied, thus breaking commutativity. The problem here is that the erasure function is erasing "too much".

The obstacles encountered when erasing *label* and *join* while guaranteeing *single-step simulation* rise from the fact that terms need to be erased differently depending on the context in which they are found. In the next section, we discuss and identify the limitations of a plain *term erasure* technique and propose a novel extension to overcome them.

3.2 Context-aware Erasure

Unfortunately, trying to stretch the definition of the erasure function to accommodate for the problematic cases shown above is futile. Firstly, note that simulation of [LABEL] is broken despite how we erase its arguments: construct *label* always yields construct *Id* on the orange path independently of its argument, which is instead always erased on the cyan path since it occurs inside constructor *Res*. Secondly, although the erasure definition could be adapted to restore commutativity of Figure 5 for *join*, it will necessary break commutativity for other cases. We support this statement by showing that this is the case for any arbitrary erasure function that is suitable for $join\ t :: MAC\ L\ (Labeled\ H\ \tau)$. Recall that rule [JOIN] evaluates a computation t embedded in *join* t to weak-head normal form, i.e., $t \Downarrow MAC\ t'$. As described in Section 3.1, the erasure function should necessarily preserve the constructor *MAC* when erasing $\varepsilon_L(t :: MAC\ H\ \tau)$ in order for the orange path to make an step. Consequently, we need a different behavior of our erasure function for sensitive computations when embedded in *join*, which we will capture in a different *auxiliary* erasure function ε'_L. Suppose we defined $\varepsilon_L(join\ t :: MAC\ L\ (Labeled\ H\ \tau)) = join\ \varepsilon'_L(t :: MAC\ H\ \tau)$, for some suitable ε'_L that exhibits the desired behavior. However, introducing a *different* erasure function in a *context-sensitive* way is fatal for commutativity of beta reductions. More precisely, the original erasure function is *no longer homomorphic over substitution*[3] , 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 for the erasure function to have [Li and Zdancewic, 2010, Russo et al., 2008, Stefan et al., 2011, 2012b, Heule et al., 2015]. As a result, function ε_{ℓ_A} is oblivious to the context in which the argument will be substituted. For example, term $(\lambda x.join\ x)\ t$ beta-reduces to *join* t and gets erased homomorphically, that is $(\lambda x.join\ x)\ \varepsilon_L(t)$, which then beta-reduces to $join\ \varepsilon_L(t) \not\equiv join\ \varepsilon'_L(t)$—recall that ε'_L captures a different behavior than that exposed by ε_L for sensitive computations embedded in *join*.

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** [Stefan et al., 2011, 2012a] which lead to breaking the one-step simulation—see details in Appendix. We propose a novel, and simple, *two-step erasure technique* in order to soundly obtain definitions of erasure functions capable to behave differently depending on the context where they are applied.

Two-steps erasure.

Our key observation is that we can soundly implement a *context-aware* erasure function by removing sensitive data in two stages. Rather than being a pure syntactic procedure, erasure can also be triggered by additional evaluation rules of special constructs. In that manner, the erasure function rewrites sensitive data as usual, i.e., in a syntactic manner, but synthesizes special constructs for those cases where it behaves differently according to the context where it gets applied. We remark, nevertheless, that such special constructs are introduced due to mere technical reasons and they are neither part of the surface syntax nor the of implementation of **MAC**—i.e., there is no performance degradation.

Figure 7 shows the definition of *label* and *join*. Special constructs $label_\bullet$ and $join_\bullet$ replace terms *label* and *join* respectively in the problematic cases, while the erasure function is applied homomorphically to their argument. Ad-

[3] In this specific case, it is possible to avoid the problem using a non-standard erasure function which eliminates variables as well, that is $\varepsilon'_L(x) = MAC\ \bullet$, since $MAC\ \bullet \Downarrow MAC\ \bullet$. However, constructs that do not use big-step semantics, such as those discussed in [Vassena et al., 2016], cannot be simulated in the same way because their context rules would require to reduce a value, i.e., $MAC\ \bullet \rightsquigarrow MAC\ \bullet$.

ditionally, rules [LABEL$_\bullet$] and [JOIN$_\bullet$] are responsible for synthesizing the right erased terms. The two-steps erasure now guarantees commutativity of cyan and orange paths. The former remains unchanged, reducing $label\ t \rightsquigarrow return\ (Res\ (Id\ t))$ (by rule [LABEL]), which is then erased to $return\ (Res\ \bullet)$. Following the latter instead, we go now from $label\ t$ to $label_\bullet\ \varepsilon_{\ell_A}(t)$ (by erasure), which then reduces according to rule [LABEL$_\bullet$] as $label_\bullet\ \varepsilon_{\ell_A}(t) \rightsquigarrow return\ (Res\ \bullet)$—thus, commuting precisely with the cyan path. Note that rule [LABEL$_\bullet$], contrary to [LABEL], yields an erased term that does not contain the constructor *Id*, hence guaranteeing *simulation*. Commutativity of rule [JOIN] follows in a similar way. While the cyan path also remains unchanged leading to $return\ (Res\ \bullet)$ after erasure, the orange path firstly erases $join\ t$ to $join_\bullet\ \varepsilon_{\ell_A}(t)$, and then reduces $join_\bullet\ \varepsilon_{\ell_A}(t) \rightsquigarrow return\ (Res\ \bullet)$ (by rule [JOIN$_\bullet$]), which, just like [LABEL$_\bullet$], guarantees *simulation* because it does not introduce constructor *Id*. Note that simulation holds also for rules [LABEL$_\bullet$] and [JOIN$_\bullet$], i.e., we are not simply moving the problem from constructs *label* and *join* to $label_\bullet$ and $join_\bullet$. We remark that *context-aware erasure* manifests frequently when extending the calculus with more advanced constructs, such as functor and relabeling operations [Vassena et al., 2016], and the proposed *two-steps erasure* can be systematically applied to restore commutativity of Figure 5.

3.3 Progress-Insensitive Non-Interference

The calculus that we have presented satisfies *progress-insensitive non-interference*. The proof of this result is based on two fundamental properties: *single-step simulation* and *determinancy* of the small step semantics. In the following, we assume well-typed terms.

PROPOSITION 1 (SINGLE-STEP SIMULATION). *If* $t_1 \rightsquigarrow t_2$ *then* $\varepsilon_{\ell_A}(t_1) \rightsquigarrow \varepsilon_{\ell_A}(t_2)$.

We proved Proposition 1 employing the *two-steps erasure* technique described in Section 3.2.

PROPOSITION 2 (DETERMINANCY). *If* $t_1 \rightsquigarrow t_2$ *and* $t_1 \rightsquigarrow t_3$ *then* $t_2 \equiv t_3$.

The proof of Proposition 2 is by standard structural induction on the two reductions. Before stating progress-insensitive non-interference, we define low-equivalence for terms.

DEFINITION 1 (ℓ_A-EQUIVALENCE). *Two terms* t_1 *and* t_2 *are indistinguishable from an attacker at security level* ℓ_A, *written* $t_1 \approx_{\ell_A} t_2$, *if and only if* $\varepsilon_{\ell_A}(t_1) \equiv \varepsilon_{\ell_A}(t_2)$.

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

PROPOSITION 3 (ℓ_A-EQUIVALENCE PRESERVATION). *If* $t_1 \approx_{\ell_A} t_2$, $t_1 \rightsquigarrow t_1'$, *and* $t_2 \rightsquigarrow t_2'$, *then* $t_1' \approx_{\ell_A} t_2'$.

Conventionally, Proposition 3 would use relation $\rightsquigarrow^*$, i.e., the reflexive transitive closure of $\rightsquigarrow$, because depending on the secret two low-equivalent terms may reduce in a different number of steps to low-equivalent terms. In our calculus, however, the only construct that can exhibit such behavior is *join*, which is defined using *big-step* semantics. Rule [JOIN] conceals the possibly different number of steps taken by sensitive computation on different executions, so that terms

Types:	$\tau ::=$	$\cdots$	$\mid\ \chi$
Values:	$v ::=$	$\cdots$	$\mid\ \xi$
Terms:	$t ::=$	$\cdots$	$\mid\ MAC_{\mathcal{X}}\ t \mid throw\ t \mid catch\ t\ t$

Figure 8: Extensions for exception handling.

$$throw :: \chi \to MAC\ \ell\ \tau$$
$$catch :: MAC\ \ell\ \tau \to (\chi \to MAC\ \ell\ \tau) \to MAC\ \ell\ \tau$$

Figure 9: API for exception handling.

seem to maintain low-equivalence in lock-step execution. We remark that the small step semantics is well-founded and that our results are mechanically verified. By repeatedly applying Proposition 3, we prove progress-insensitive non-interference, which informally states that if two low-equivalent terms reduce to values then also the values are low-equivalent.

THEOREM 1 (PINI). *If* $t_1 \approx_{\ell_A} t_2$, $t_1 \Downarrow v_1$ *and* $t_2 \Downarrow v_2$ *then* $v_1 \approx_{\ell_A} v_2$.

4. EXCEPTION HANDLING

In this section, we extend our core calculus with exceptions as described by the original **MAC** paper [Russo, 2015]. One interesting insights, gained by using a proof assistant to check our proofs, is a technique that simplifies security proofs by masking sensitive exceptions in erased terms. Although we do not provide further details, this technique could be used to simplify the soundness proofs of **LIO** [Stefan et al., 2012b].

Calculus, Terms, Types, and Semantics.

We extend the syntactic categories from our core calculus as described in Figure 8. We introduce a value ξ of exception type χ and a new constructor $MAC_{\mathcal{X}}\ t$, denoting a failing computation due to exception t. Terms $throw\ t$ and $catch\ t_1\ t_2$ aborts the current *MAC* computation with exception t and recover from an exception thrown in computation t_1 running exception handler t_2, respectively. Figure 9 gives the types for *throw* and *catch* in a form similar to Haskell APIs. The semantics of these two constructs is standard and thus omitted.

Join and exceptions.

The interplay between exceptions and *join* is delicate and security might be at stake if these two features were naively combined [Stefan et al., 2012b, Hritcu et al., 2013]. Observe that type signatures in Figure 9 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 (and affect) less sensitive ones. From now on, we refer to exceptions raised in a sensitive *MAC* computation as *sensitive exceptions*. Observe that sensitive exceptions might be responsible for suppressing observable events in less sensitive computations, which gives place to an implict flow! (We refer interested readers to [Russo, 2015] for further details about this attack.) In our calculus, the only primitive which combines computations with different labels is *join*. Therefore, to close leaks via exceptions, **MAC** modifies the semantics of *join* to catch exceptions, preventing them to propa-

$$\frac{t_1 \Downarrow MAC_X\ t_2}{join\ t_1 \rightsquigarrow return\ (Res_X\ t_2)}\ (\text{JOIN}_X) \qquad (\text{UNLABEL}_X)\ unlabel\ (Res_X\ t) \rightsquigarrow throw\ t$$

Figure 10: Secure exception handling.

$$\varepsilon_{\ell_A}(Res_X\ t :: Res\ \ell\ \tau) = \begin{cases} Res_X\ \varepsilon_{\ell_A}(t) & \text{if } \ell \sqsubseteq \ell_A \\ Res\ \bullet & \text{otherwise} \end{cases}$$

Figure 11: Erasure of Res_X.

gate to less sensitive computations—this solution is similar to previous work [Stefan et al., 2012b, Hritcu et al., 2013].

To implement such countermeasure, we firstly proceed to add a constructor denoting exceptions inside the *Labeled* $\ell\ \tau$ data type, i.e., the type of data produced by *join*. Specifically, we add internal constructor $Res_X\ t$, where $t :: \chi$. Figure 10 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. Note that the attacker cannot observe the exception anymore without first *unlabeling* the result. Such operation is then subject to the *no read-up* rule, which prevents observing a sensitive exception in a less sensitive computation. In Figure 10, we extend the semantics of *unlabel* with rule [UNLABEL$_X$], which handles constructor Res_X by rethrowing the exception.

4.1 Masking sensitive exceptions

Formally, we need to show that the new calculus preserves the *single-step simulation*. We ordinarily extend the erasure function to rewrite MAC_X, *throw* and *catch* to $\bullet$ if their computation label is sensitive; otherwise, erasure is applied homomorphically. The erasure of Res_X deserves more attention—see Figure 11. Note that the content of a sensitive exception is rewritten to $\bullet$ as expected, but also the constructor Res_X is replaced by *Res*. As a result of that, and different from [Stefan et al., 2012b], there exists no sensitive labeled exceptions in erased terms—thus *simplifying semantics*. Crucially, we have the freedom of choosing this definition without breaking the *one-step simulation*, because no other construct can detect, either explicitly or implicitly, the difference. For instance, rule [UNLABEL$_X$] 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.

5. CONCURRENCY

In this section, we extend our calculus with concurrency. The possibility to run simultaneous *MAC* ℓ computations provides attackers with new means to bypass security checks. In particular, concurrency magnifies the bandwidth of the termination covert channel to be linear in the size (of bits) of secrets [Stefan et al., 2012a][4]. The key observation is that a computation $t :: MAC\ \ell_H\ \tau$ embedded in $join\ t :: MAC\ \ell_L\ (Labeled\ \ell_H\ \tau)$ might not terminate depending on the value of the secret. If the computation t diverges, it might suppress public side-effects following *join* t, thus revealing a bit about the secret. To illustrate this point, consider the function $send :: Int \to MAC\ L\ ()$ which sends an integer to the attacker's server. By wrapping *join* t among two instances of *send*, i.e., $attack\ n = send\ n \ggg \lambda() \to join\ t \ggg \lambda() \to send\ n$, and assuming that t diverges if the secret is true, then the attacker knows that the secret is false when it receives *nn* in the server, otherwise the secret is true. An attacker might then leak the whole secret by spawning as many threads as bits in the 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.).

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

Figure 12: API for concurrency.

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

Figure 13: Syntax for concurrent calculus.

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 [Stefan et al., 2012a]. As a result, nonterminating computations based on secrets cannot affect the outcome of public events. To achieve this behavior, **MAC** replaces *join* by *fork*—see Figure 12. Informally, it is secure to spawn sensitive computations (of type $MAC\ \ell_H\ ()$) from non-sensitive ones (of type $MAC\ \ell_L\ ()$) 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* (*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. In Section 5.2, we prove that the concurrent calculus satisfies progress-sensitive noninterference (PSNI).

Calculus.

Figure 13 extends the calculus from Section 2 with concurrency. It introduces *global configurations* of the form $\langle s, \Phi \rangle$ composed by an abstract scheduler state s and a thread pool Φ. We remark that the whole calculus includes also shared memory, which we omit for brevity. Threads are secure computations of type $MAC\ \ell\ ()$ which get organized in isolated thread pools according to their security label. A

[4] Furthermore, the presence of threads introduce the *internal timing covert channel* [Smith and Volpano, 1998], 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.

$$\frac{\Phi[\ell][n] = t_1 \qquad t_1 \rightsquigarrow_e t_2 \qquad s_1 \xrightarrow{(\ell,n,e)} s_2}{\langle s_1, \Phi \rangle \hookrightarrow \langle s_2, \Phi[\ell][n] := t_2 \rangle}$$

Figure 14: Scheme rule for concurrent semantics.

pool t_s in the category *Pool* ℓ contains exclusively threads 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 $\Phi[\ell][n] = t$ to retrieve the nth-thread in the ℓ-thread pool—it is just syntax sugar for $\Phi(\ell) = t_s$ and $t_s[n] = t$. The notation $\Phi[\ell][n] := t$ denotes the thread pool obtained by performing the update $\Phi(\ell)[n \mapsto t]$. A thread pool can be fully erased, and reading from them results in an erased thread, i.e., $\bullet[n] = \bullet$ and updating has no effect, i.e., $\bullet[n \mapsto t] = \bullet$. As mentioned before, term *fork* t spawns thread t and replaces *join* in the calculus.

Semantics.

Figure 14 shows the scheme rule for evaluation steps of concurrent configurations, denoted by relation $\hookrightarrow$. The relation $s_1 \xrightarrow{(\ell,n,e)} s_2$ represents a transition in the scheduler, that depending on the initial state s_1, decides to run thread identified by (ℓ, n), which is retrieved from the configuration ($\Phi[\ell][n] = t_1$) and executed ($t_1 \rightsquigarrow_e t_2$). We decorate the sequential semantics ($\rightsquigarrow$) with event e, which is also present in the scheduler transition. Events inform the scheduler about the evolution of the global configuration, so that it can realize concrete scheduling policies and updating its state accordingly. Event *Step* denotes a single sequential step, event *Skip* denotes that a thread is *stuck*, e.g., on a synchronization variable, event *Done* is generated when a thread has terminated and event *Fork* ℓ n informs the scheduler that the current thread has forked a new thread identified by (ℓ, n). Event $\bullet$ is triggered by an erased thread $\bullet$, that is $\bullet \rightsquigarrow_\bullet \bullet$. Lastly, the thread pool is updated with the final state of the thread ($\Phi[\ell][n] := t_2$).

Parametric proof.

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 Haskell's run-time system, and show that it satisfies the requirements listed in this section. Furthermore, we constructively obtain a proof that **MAC** is secure with a round-robin scheduler by simply instantiating our main theorem.

5.1 Schedulers

Our proof is valid for schedulers which are (i) deterministic, (ii) fulfill the single-step simulation from Figure 5, i.e., schedulers may not leverage on sensitive information to determine what observable thread should be scheduled next, and (iii) guarantee progress of observable threads, i.e., execution of observable threads cannot be indefinitely deferred by sensitive ones. In the following, we formally characterize schedulers for which our security guarantees apply.

REQUIREMENT 1.

i) Determinancy*:*
if $s_1 \xrightarrow{(\ell,n,e)} s_2$ *and* $s_1 \xrightarrow{(\ell,n,e)} s_3$*, then* $s_2 = s_3$*.*

ii) Single-step simulation*:*
if $s_1 \xrightarrow{(\ell,n,e)} s_2$*, then* $\varepsilon_{\ell_A}(s_1) \xrightarrow{(\ell,n,\varepsilon_{\ell_A}(e))} \varepsilon_{\ell_A}(s_2)$*.*

iii) Progress*: sensitive threads cannot indefinitely defer execution of non-sensitive ones.*

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. Nevertheless, Requirement (i) is slightly weaker than what one might expect: it assumes that the scheduled thread (ℓ, n) is the same in both reductions. We discuss why we need to relax determinism in this way in Section 5.2. As it is expected from the concurrent calculus, we assume that the abstract scheduler satisfies the single-step simulation. Observe that the erasure function of the scheduler state is scheduler specific, and thus we leave it unspecified. Nevertheless, events inherit the security level of the threads that generated them, therefore they are erased accordingly, e.g., $\varepsilon_{\ell_A}(e :: \textit{Event}\ \ell) = \bullet$ if $\ell \not\sqsubseteq \ell_A$. Requirement (iii) 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. Unfortunately, the scheduler state and semantics are abstract, therefore we cannot define requirement (iii) more precisely. We overcome this technical limitation by indexing the low-equivalence relationship among scheduler states and using it to craft an additional requirement of the scheduler. We proceed to define ℓ_A-equivalence between scheduler states and its annotated version to guarantee progress as follows.

DEFINITION 2 (SCHEDULER ℓ_A-EQUIVALENCE).

i) Two states are ℓ_A*-equivalent, written* $s_1 \approx_{\ell_A} s_2$ *if and only if* $\varepsilon_{\ell_A}(s_1) \equiv \varepsilon_{\ell_A}(s_2)$

ii) Two states are (i,j)*-*ℓ_A*-equivalent, written* $s_1 \approx_{\ell_A}^{(i,j)} s_2$ *if and only if* $s_1 \approx_{\ell_A} s_2$ *and, according to* s_1 *and* s_2*,* i *and* j *are respectivelly upper bounds over the numbers of sensitive threads scheduled before the first, and the same, non-sensitive thread gets run.*

The relation $s_1 \approx_{\ell_A}^{(i,j)} s_2$ captures an alignment measure of two ℓ_A-equivalent states and how close they are to schedule the next common non-sensitive thread. Informally, our non-interference proof excludes *starvation* of threads, that can reveal *progress* 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 $s_1 \approx_{\ell_A}^{(i,j)} s_2$ 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 [Goguen and Meseguer, 1984]. More specifically, we describe what occurs with the (i,j) indexes when schedulers handle sensitive and observable events.

REQUIREMENT 2 (NON-INTERFERING SCHEDULER). *Given* $s_1 \xrightarrow{(\ell,n,e)} s_2$, $e \not\equiv \bullet$, *and* $s_1 \approx_{\ell_A}^{(i,j)} s_1'$, *then*

- *If* $\ell \sqsubseteq \ell_A$, *then one of the following holds:*
  - *i) If* $j = 0$, *then there exists state* s_2' *such that* $s_1' \xrightarrow{(\ell,n,e)} s_2' \ \wedge\ s_2 \approx_{\ell_A} s_2'$;
  - *ii) If* $j > 0$, *then there exists* h, m, j' *s.t.* $j' < j$ *and:* $\forall e'.e' \not\equiv \bullet \Rightarrow \exists s_2'.\ s_1' \xrightarrow{(h,m,e')} s_2' \ \wedge\ h \not\sqsubseteq \ell \ \wedge\ s_1 \approx_{\ell_A}^{(i,j')} s_2'$
- *If* $\ell \not\sqsubseteq \ell_A$, *then* $s_2 \approx_{\ell_A} s_1'$

Given two (i,j)-ℓ_A-equivalent scheduler states if one runs a non-sensitive thread ($\ell \sqsubseteq \ell_A$), then the other schedules either the same non-sensitive thread ($j = 0$) or a sensitive thread ($j > 0$), leading to an ℓ_A-equivalent state. As relevant technical detail, we remark that $e \not\equiv \bullet$ and $e' \not\equiv \bullet$ since we expect the *non-erased* scheduler to be *starvation-free* and *non-interfering*. Since we aim to a modular proof, the scheduler is considered in isolation from the pool thread, therefore case *ii)* cannot predict what event will be triggered by thread (h,m). As a conservative approximation then, the requirement must hold for *any* possible event e', which in addition determines the final state s_2'. Lastly, condition $j' < j$ guarantees that the common non-sensitive thread (ℓ,n) in s_1' will not starve indefinitely, i.e., it will *eventually* be scheduled withing the next j' reductions. If the first scheduler runs a sensitive thread ($\ell \not\sqsubseteq \ell_A$), then the resulting state is low-equivalent to the other ($s_2 \approx_{\ell_A} s_1'$). It might seem that scheduler requirements 1.ii (*single-step simulation*) and 2 (*non-interference*) are overlapping. However, they fulfill two different purposes. Specifically, the former is needed to prove *simulation* of the concurrent semantics—note that the scheduler state is part of the global configuration. We instead use the latter to prove progress-sensitive non-interference of the concurrent semantics. We outline the non-interference proof in Section 5.2.

Round-robin scheduler.

As an example of a secure scheduler that can be employed with our concurrent calculus, Figure 15 instantiates a round-robin scheduler with a time-slot of one step. The state of the scheduler is a queue that tracks the identifiers

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

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

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

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

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

$$s \xrightarrow{(\ell,n,\bullet)}_{RR} s$$

Figure 15: Round-robin scheduler.

$$[\,] \approx_L^{(0,0)} [\,] \qquad \frac{s_1 \approx_L^{(i,j)} s_2}{(L,n) : s_1 \approx_L^{(0,0)} (L,n) : s_2}$$

$$\frac{s_1 \approx_L^{(i,j)} s_2}{(H,n) : s_1 \approx_L^{(i+1,j)} s_2} \qquad \frac{s_1 \approx_L^{(i,j)} s_2}{s_1 \approx_L^{(i,j+1)} (H,n) : s_2}$$

Figure 16: Annotated L-equivalence (Round-robin).

of *alive* threads in the global configuration. The queue is concretely represented by a list of pairs, containing a label and a thread number, whose first element is the identifier of the next thread to be scheduled. After executing one step (event *Step*), the current thread has used up its time slot and is enqueued. If the scheduled thread cannot execute (event *Skip*), it is skipped and enqueued as well. When the current thread has terminated (event *Done*), the thread is not alive anymore and hence removed from the queue. Message $(\ell_{\mathrm{L}}, n_1, Fork\ \ell_{\mathrm{H}}\ n_2)$ informs the scheduler that thread (ℓ_{L}, n_1) has spawned thread (ℓ_{H}, n_2), which is then enqueued with the current thread. The last rule is not part of the actual scheduling algorithm and it is added exclusively to study the security guarantees of the scheduler.

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 1.i. We define the erasure function to filter out the identifiers of threads non observable to the attacker, i.e., $\varepsilon_{\ell_A}(s) = \mathit{filter}\ (\lambda(\ell,n) \rightarrow \ell \sqsubseteq \ell_A)\ s$. By induction on the scheduler reduction, it follows that round-robin satisfies the *single-step* simulation, i.e., scheduler requirement 1.ii. Note that round-robin is *starvation-free* because it has a finite time-slot and is preemptive. We remark that absence of *starvation* is a desirable property of schedulers, which is sufficient to guarantee *progress*, i.e., scheduler requirement 1.iii. Before proving that round-robin is non-interfering, i.e., scheduler requirement 2, Figure 16 instantiates an annotated L-equivalence assuming the two points lattice for simplicity. In particular, if $s_1 \approx_{\ell_A}^{(0,0)} s_2$ for non-empty states s_1 and s_2, then round-robin will schedule the same L-thread in the next reduction.

PROPOSITION 4 (ROUND-ROBIN IS SECURE). *Round-robin satisfies schedulers requirements 1 and 2.*

5.2 Progress-Sensitive Non-Interference

The non-interference proof for the concurrent semantics relies on *single-step simulation* and *determinancy*. Before discussing these properties, we formally define the erasure function for the rest of the concurrent calculus. Global configurations are erased by erasing each component separately, i.e., $\varepsilon_{\ell_A}(\langle s, \Phi \rangle) = \langle \varepsilon_{\ell_A}(s), \varepsilon_{\ell_A}(\Phi) \rangle$, thread pools are erased pointwise and pools are erased according to their label, i.e., $\varepsilon_{\ell_A}(t_s :: \mathit{Memory}\ \ell) = \bullet$ if $\ell_A \not\sqsubseteq \ell$ and homomorphically erased otherwise.

PROPOSITION 5 (SINGLE-STEP SIMULATION). *If* $c_1 \hookrightarrow c_2$ *then* $\varepsilon_{\ell_A}(c_1) \hookrightarrow \varepsilon_{\ell_A}(c_2)$.

Proposition 5 follows immediately by *single-step* simulation of the sequential calculus and scheduler requirement 1.ii.

We now discuss a subtle distinction between *predictability* and *determinancy*, two slightly different properties that come into play when erasing schedulers. To prove non-interference, we need to show *determinancy* of the concurrent semantics, i.e., if $c_1 \hookrightarrow c_2$ and $c_1 \hookrightarrow c_3$ then $c_2 \equiv c_3$. Note that the term erasure proof technique requires to construct a simulation between ℓ_A-equivalent configurations, therefore the semantics must be deterministic also in presence of $\bullet$. Intuition suggests that this should hold because a "reasonable" scheduler schedules a thread depending exclusively on its initial state, a property that we name *predictability* and we formally define as follows.

Definition 3 (Predictability). *Given* $s_1 \xrightarrow{(\ell,n,e)} s_2$ *and* $s_1 \xrightarrow{(\ell',n',e')} s_3$, *then* $\ell \equiv \ell'$, $n \equiv n'$ *and if* $e \equiv e'$ *then* $s_2 \equiv s_3$.

This property guarantees that given the same initial state, the schedulers will run the same thread ($\ell \equiv \ell'$ and $n \equiv n'$), and after receiving the same event, they will reduce to the same final state ($s_2 \equiv s_3$). Unfortunately, a rule of form $s_1 \xrightarrow{(\ell,n,\bullet)} s_2$ may break this property. For instance, consider rule $s \xrightarrow{(\ell,n,\bullet)}_{RR} s$ of the round-robin scheduler (see Figure 15). Such a rule schedules a thread (ℓ, n) non-deterministically, therefore the same initial state might not be sufficient alone to guarantee in general determinancy of the global configuration. Crucially, predictability is not preserved under erasure: we loose the ability to *predict* which sensitive thread is about to be scheduled because sensitive threads are erased by the erasure function—this is sensitive information because it may depend on secrets! Luckily, determinancy (scheduler requirement 1.i) is sufficient for our purposes—note that it is weaker than *predictability.* In fact, it is possible to control unpredictability by annotating the concurrent semantics as $\langle s_1, \Phi_1 \rangle \hookrightarrow_{(\ell,n)} \langle s_2, \Phi_2 \rangle$ if and only if $s_1 \xrightarrow{(\ell,n,e)} s_2$ (for some event e). The annotation has the purpose to witness what thread was originally scheduled, thus enabling scheduler determinancy.

Proposition 6 (Concurrent determinancy). *If* $c_1 \hookrightarrow_{(\ell,n)} c_2$ *and* $c_1 \hookrightarrow_{(\ell,n)} c_3$ *then* $c_2 \equiv c_3$.

We now proceed to prove non-interference. We ordinarily extend ℓ_A-equivalence to global configurations, written $c_1 \approx_{\ell_A} c_2$, if and only if $\varepsilon_{\ell_A}(c_1) = \varepsilon_{\ell_A}(c_2)$. Relation $\hookrightarrow^\star$ denotes the transitive reflexive closure of $\hookrightarrow$.

Theorem 2 (PSNI). *Given global configurations* c_1, c_1', *and* c_2 *which do not contain* $\bullet$ *and* $label_\bullet$, *and a scheduler fulfilling requirements 1.i, 1.ii, 1.iii and 2, if* $c_1 \approx_{\ell_A} c_2$ *and* $c_1 \hookrightarrow_{(\ell,n)} c_1'$, *then there exists* c_2' *such that* $c_2 \hookrightarrow^\star c_2'$ *and* $c_2 \approx_{\ell_A} c_2'$.

The proof of Theorem 2 is based on the non-interference of the scheduler (scheduler requirement 2) in addition to Propositions 5 and 6 of the concurrent semantics—see details in Appendix. Note that we exclude nodes $\bullet$ and $label_\bullet$ only from the non-erased configuration and uniquely to comply with Requirement 2. We conclude with a corollary which assets that **MAC** satisfies PSNI (which proofs is obtained by applying Theorem 2 and Proposition 4).

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

6. RELATED WORK

Mechanized proofs.

The library **MAC** is presented in [Russo, 2015] as a functional pearl and relies on its simplicity to convince readers about its correctness. This work bridges the gap on **MAC**'s lack of formal guarantees and exhibits interesting insights on the proofs of its soundness. **LIO** is a library structural similar to **MAC** but dynamically enforcing IFC [Stefan et al., 2011]. The core calculus of **LIO**, i.e., side-effect free computations together with exception handling, has been modeled in the Coq proof assistant [Stefan et al., 2012b]. Different from our work, these mechanized proofs do not simplify the treatment of sensitive exceptions by masking them in erased programs. In parallel to [Stefan et al., 2012b], Breeze [Hritcu et al., 2013] 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 [Bichhawat et al., 2014].

Concurrency.

Considering IFC for a general scheduler could lead to refinements attacks. In this light, Russo and Sabelfeld provide termination-insensitive non-interference for a wide-class of deterministic schedulers [Russo and Sabelfeld, 2006]. Barthe et al. [Barthe et al., 2009] adopt this idea for Java-like bytecode. Although we also consider deterministic schedulers, our security guarantees are stronger by considering leaks of information via abnormal termination. Heule et al. [Heule et al., 2015] describe how to retrofit IFC in a programming language with sandboxes. Similar to our work, their soundness proofs are parametric on deterministic schedulers and provide progress-sensitive non-interference with informal arguments regarding thread progress—in this work, we spell out formal requirements on schedulers capable to guarantee thread progress. A series of work for π-calculus consider non-deterministic schedulers while providing progress-sensitive non-interference [Honda et al., 2000, Kobayashi, 2005, Honda and Yoshida, 2007, Pottier, 2002].

Haskell.

Devriese and Piessens provide a monad transformer to extend imperative-like APIs [Devriese and Piessens, 2011]. Jaskelioff and Russo implements a library which dynamically enforces IFC using secure multi-execution (SME) [Jaskelioff and Russo, 2011]—a technique that runs programs multiple times (once per security level) and varies the semantics of inputs and outputs to protect confidentiality. Rather than running multiple copies of a program, Schmitz et al. [Schmitz et al., 2016] provide a library with *faceted values*, where values present different behavior according to the privilege of the observer. We hope that our insights will make easier to mechanize those proofs found in the work cited above.

Operating systems research.

MAC borrows ideas from Mandatory Access Control [Bell and La Padula, 1976, Biba, 1977] and phrases them into a programming language setting. Although originated in the 70s, there are modern manifestations of MAC applied to a wide range of scenarios [Efstathopoulos et al., 2005, Zeldovich et al., 2006, Krohn et al., 2007, Murray et al., 2013]. Due to its complexity, it is not surprising that OS-based MAC systems lack accompanying soundness guarantees or mechanized proofs—seL4 being the exception [Murray et al., 2013]. 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.

7. CONCLUSION

We present a *full-fledged* formalization of **MAC** in Agda, where non-interference is proven by term erasure. To the best of our knowledge, this is the first work of its kind for IFC libraries in Haskell, both for completeness and number of features included in the model. Thanks to our mechanized proofs, we identify challenges arising from erasing terms depending on the context where they appear and propose *two-steps erasure*—an effective technique to systematically deal with such cases. Additionally, we show exception *masking*, an alternative way to erase exceptions that simplifies proof of security guarantees in libraries equipped with exceptions and exceptions handling features. Our mechanized proofs also make explicit sufficient scheduler requirements to guarantee PSNI—something that has been only treated informally before [Stefan et al., 2012a, Heule et al., 2015]. 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.

Acknowledgments.

This work was supported by the Swedish research agency VR.

References

M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A Core Calculus of Dependency. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 147–160, January 1999.

Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a java-like language. In *Proceedings of the 15th IEEE Workshop on Computer Security Foundations*, CSFW '02. IEEE Computer Society, 2002.

G. Barthe, T. Rezk, A. Russo, and A. Sabelfeld. Security of multithreaded programs by compilation. *Special issue of ACM Transactions on Information and System Security (TISSEC)*, August 2009.

David E. Bell and L. La Padula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997, Rev. 1, MITRE Corporation, Bedford, MA, 1976.

K. J. Biba. Integrity considerations for secure computer systems. ESD-TR-76-372, 1977.

Abhishek Bichhawat, Vineet Rajani, Deepak Garg, and Christian Hammer. *Information Flow Control in WebKit's JavaScript Bytecode*, pages 159–178. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. ISBN 978-3-642-54792-8. . URL http://dx.doi.org/10.1007/978-3-642-54792-8_9.

P. Buiras, D. Vytiniotis, and A. Russo. HLIO: Mixing static and dynamic typing for information-flow control in Haskell. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP '15)*. ACM, 2015.

Pablo Buiras, Deian Stefan, and Alejandro Russo. On dynamic flow-sensitive floating-label systems. In *Proc. of the IEEE Computer Security Foundations Symposium*, CSF '14. IEEE Computer Society, 2014.

D. Devriese and F. Piessens. Information flow enforcement in monadic libraries. In *Proc. of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI '11)*. ACM, 2011.

Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. In *Proc. of the twentieth ACM symp. on Operating systems principles*, SOSP '05. ACM, 2005.

Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C. Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12. USENIX Association, 2012.

J. A. Goguen and J. Meseguer. Unwinding and inference control. In *Security and Privacy, 1984 IEEE Symposium on*, April 1984.

J.A. Goguen and J. Meseguer. Security policies and security models. In *Proc of IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1982.

Stefan Heule, Deian Stefan, Edward Z. Yang, John C. Mitchell, and Alejandro Russo. IFC inside: Retrofitting languages with dynamic information flow control. In *Conference on Principles of Security and Trust (POST)*. Springer, April 2015.

Kohei Honda and Nobuko Yoshida. A uniform type structure for secure information flow. *ACM Trans. Program. Lang. Syst.*, October 2007.

Kohei Honda, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. Secure Information Flow as Typed Process Behaviour. In *Proc. of the 9th European Symposium on Programming Languages and Systems*. Springer-Verlag, 2000.

C. Hritcu, M. Greenberg, B. Karel, B. C. Peirce, and G. Morrisett. All your IFCexception are belong to us. In *Proc. of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2013.

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

Naoki Kobayashi. Type-based information flow analysis for the π-calculus. *Acta Inf.*, 42(4), December 2005.

Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard os abstractions. In *Proceedings of ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07. ACM, 2007.

P. Li and S. Zdancewic. Encoding information flow in Haskell. In *Proc. of the IEEE Workshop on Computer Security Foundations (CSFW '06)*. IEEE Computer Society, 2006.

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

Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. sel4: From general purpose to a proof of information flow enforcement. *2012 IEEE Symposium on Security and Privacy*, 0, 2013.

François Pottier. A Simple View of Type-Secure Information Flow in the π-Calculus. In *In Proc. of the 15th IEEE Computer Security Foundations Workshop*, pages 320–330, 2002.

A. Russo and A. Sabelfeld. Securing Interaction between Threads and the Scheduler. In *Proc. IEEE Computer Sec. Foundations Workshop*, pages 177–189, July 2006.

A. Russo, K. Claessen, and J. Hughes. A library for lightweight information-flow security in Haskell. In *Proc. ACM SIGPLAN symposium on Haskell (HASKELL '08)*. ACM, September 2008.

Alejandro Russo. Functional Pearl: Two Can Keep a Secret, if One of Them Uses Haskell. In *Proc. of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015. ACM, 2015.

Thomas Schmitz, Dustin Rhodes, Thomas H. Austin, Kenneth Knowles, and Cormac Flanagan. Faceted dynamic information flow via control and data monads. In Frank Piessens and Luca ViganÃš, editors, *POST*, volume 9635 of *Lecture Notes in Computer Science*. Springer, 2016.

G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM symposium on Principles of Programming Languages (POPL '98)*, 1998.

D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Maziéres. Addressing covert termination and timing channels in concurrent information flow systems. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP '12)*. ACM, 2012a.

D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in the presence of exceptions. *Arxiv preprint arXiv:1207.1457, to appear in Journal of Functional Programming, Cambridge University Press*, 2012b.

Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible Dynamic Information Flow Control in Haskell. In *Proceedings of the 4th ACM symposium on Haskell*, pages 95–106, New York, NY, USA, 2011. ACM.

T. C. Tsai, A. Russo, and J. Hughes. A library for secure multi-threaded information flow in Haskell. In *Proc. IEEE Computer Security Foundations Symposium (CSF '07)*, July 2007.

Marco Vassena, Pablo Buiras, Lucas Waye, and Alejandro Russo. Flexible manipulation of labeled values for information-flow control libraries. In *Proceedings of the 12th European Symposium On Research In Computer Security*. Springer, September 2016.

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

APPENDIX

A. CONTEXT-SENSITIVITY IN LIO

LIO provides a labeling primitive *label* t_1 t_2 which labels term t_2 with label t_1. The definition of $\varepsilon_{\ell_A}(\mathit{label}\ t_1\ t_2)$ is *context-sensitive*: t_2 should be rewritten to $\bullet$ if $t_1 \not\sqsubseteq \ell_A$ or to $\varepsilon_{\ell_A}(t_2)$ otherwise. Note that due to the dynamic nature of **LIO**, labels are *terms* and t_1 is a concrete label only when fully evaluated [Stefan et al., 2012b]. As a result, the $\sqsubseteq$ relation is defined between terms, instead of plain labels, and it is partial when it involves unevaluated terms[5]. Unfortunately, that fact that $\sqsubseteq$ is partially defined affects the erasure function since it conservatively considers labels that are not fully evaluated as non-visible to the attacker. For example, $\varepsilon_{L}(\mathit{label}\ ((\lambda\ell.\ell)\ L)\ t_2)$ is erased to $\mathit{label}\ ((\lambda\ell.\ell)\ L)\ \bullet$, because the erasure considers that $((\lambda\ell.\ell)\ L) \not\sqsubseteq L$—observe that public information has been erased! While the erasure function defined in this way does not break homomorphic substitution, the price to pay is being conservative and having a non-standard security lattice. On the other hand, using two-steps erasure, we can delegate the actual erasure to a new rule [LABEL$_\bullet$], triggered by construct $\mathit{label}_\bullet$, inserted by the erasure function. The semantics rules of $\mathit{label}_\bullet$ determine whether t_2 should be erased only after fully evaluating label t_1—due to the dynamic nature of **LIO**, security labels are not statically available.

B. MEMORY IN LIO

Our mechanized formalization of **MAC** also includes references, which lead us to undercover some flaws in the model of **LIO** [Stefan et al., 2011, 2012b]. Our memory model provides deterministic allocation in a memory partitioned by security labels to avoid establishing a bijection between heaps in our proofs [Banerjee and Naumann, 2002]. Crucially, our formalization utilizes a memory model that guarantees these properties, while in previous works these characteristics are only *assumed*—memory is just a mapping from addresses to terms.

Problems.

In [Stefan et al., 2011], the erasure of construct *newRef* ℓ t and *writeRef* a t is incorrect, because it homomorphically erases t, irrespectively of the sensitivity of the expression written to memory—it should instead rewrite t

[5] Coq model at https://github.com/deian/lio-semantics/

Configuration	$c ::=$	$\langle \Sigma, t \rangle$
Store:	$\Sigma ::=$	$(\ell : Label) \rightarrow Memory\ \ell$
Memory ℓ	$t_s ::=$	$[\,] \mid t : t_s \mid \bullet$
Types	$\tau ::=$	$\cdots \mid Nat$
Terms	$t ::=$	$\cdots \mid newRef\ t$
		$\mid readRef\ t \mid writeRef\ t_1\ t_2$
Values	$v ::=$	$\cdots \mid n$

Figure 17: Formal syntax for extended calculus.

to $\bullet$ when ℓ is not visible to the attacker or when writing to a sensitive region in memory, just like for *label* ℓ t. Note that this definition breaks *single-step simulation*, a fundamental property to guarantee non-interference. Consider **LIO** semantics rule [NREF], slightly simplified and instantiated with the problematic case: $\langle \Sigma, newRef\ H\ t \rangle \longrightarrow \langle \Sigma', return\ a \rangle$, where $\Sigma' = \Sigma\ [a \mapsto Labeled\ H\ t]$ for some *fresh* address a (cyan path). Proving simulation requires to show that configuration $\langle \varepsilon_L(\Sigma), newRef\ H\ \varepsilon_L(t) \rangle$ reduces to $\langle \varepsilon_L(\Sigma'), return\ a \rangle$ (orange path). Unfortunately, in this case the diagram does not commute. By rule [NREF], configuration $\langle \varepsilon_L(\Sigma), newRef\ H\ \varepsilon_L(t) \rangle$ reduces to a configuration with memory $\varepsilon_L(\Sigma)\ [a \mapsto Labeled\ H\ \varepsilon_L(t)]$, which is different from $\varepsilon_L(\Sigma')$. Observe that $\varepsilon_L(\Sigma')$ erases the memory pointwise and thus resulting in mapping a to *Labeled* H $\bullet$ rather than *Labeled* H $\varepsilon_L(t)$ as done by memory $\varepsilon_L(\Sigma)\ [a \mapsto Labeled\ H\ \varepsilon_L(t)]$.

In a draft version of [Stefan et al., 2012b], which surpasses [Stefan et al., 2011], we have identified a second problem that concerns low-equivalence of memories, which is defined as $\varepsilon_{\ell_A}(\Sigma_1) \equiv \varepsilon_{\ell_A}(\Sigma_2)$—memories are erased pointwise. This definition is too restrictive because rules out memories with different number of sensitive locations as low-equivalent, even though the attacker cannot distinguish them. For instance, according to [Stefan et al., 2012b], memories $\Sigma_1 = [0 \mapsto Labeled\ H\ t_0, 1 \mapsto Labeled\ H\ t_1]$ and $\Sigma_2 = [0 \mapsto Labeled\ H\ t'_0]$ are not low-equivalent because $\varepsilon_L(\Sigma_1) \not\equiv \varepsilon_L(\Sigma_2)$, since $[0 \mapsto Labeled\ H\ \bullet, 1 \mapsto Labeled\ H\ \bullet] \not\equiv [0 \mapsto Labeled\ H\ \bullet]$. In particular, this definition of low-equivalence is not compatible with intrinsically secure programs in both the sequential or concurrent setting. For instance, executing a sensitive thread could result in breaking low-equivalent configurations as soon as it allocates sensitive references according to secret values.

C. MEMORY IN MAC

Split Memory.

We securely add memory to our calculus—see Figure 17. Memory is compartmentalized into isolated labeled segments, one for each label of the lattice, and accessed exclusively through the store Σ, similar to thread pool store Φ. A memory in the category *Memory* ℓ contains terms at security level ℓ. We write $\Sigma[\ell][n] = t$ to retrieve the nth-cell in the ℓ-memory—it is a syntax sugar for $\Sigma(l) = t_s$ and $t_s[n] = t$. The notation $\Sigma[\ell][n] := t$ denotes the store obtained by performing the update $\Sigma(l)[n \mapsto t]$. Lastly, we write $\mid t_s \mid = n$ to denote that memory t_s has length n.

We write $\langle \Sigma, t \rangle$ for a sequential configuration containing store Σ and term t. Figure 17 the sequential calculus with labeled references and *MAC* operations to allocate, read and write memory, all of which work with explicitly labeled references. Figure 18 shows the type of the new operations, which are restricted according to the *no read-up* and *no write-down* rules, like those of *label* and *unlabel*. While **MAC** leverages Haskell references—a labeled reference is a simple wrapper around *IORef*, in our calculus we implement references explicitly using *Nat* as the type of a memory address. More precisely, term $Res\ n :: Ref\ \ell\ \tau$ represents a labeled reference to the n-th cell of memory labeled with ℓ, which contains a term of type τ.

$$\begin{array}{l}
\textbf{type}\ Ref\ \ell\ \tau = Res\ \ell\ (IORef\ \tau) \\
newRef :: \ell_L \sqsubseteq \ell_H \Rightarrow \tau \rightarrow MAC\ \ell_L\ (Ref\ \ell_H\ \tau) \\
readRef :: \ell_L \sqsubseteq \ell_H \Rightarrow Ref\ \ell_L\ \tau \rightarrow MAC\ \ell_H\ \tau \\
writeRef :: \ell_L \sqsubseteq \ell_H \Rightarrow \tau \rightarrow Ref\ \ell_H\ \tau \rightarrow MAC\ \ell_L\ ()
\end{array}$$

Figure 18: API of memory operations

(NEW)
$$\frac{|\Sigma(l)| = n}{\langle \Sigma, newRef\ t \rangle \longrightarrow \langle \Sigma[\ell][n] := t, return\ (Res\ n) \rangle}$$

(WRITE)
$$\langle \Sigma, writeRef\ (Res\ n)\ t \rangle \longrightarrow \langle \Sigma[\ell][n] := t, return\ () \rangle$$

(READ)
$$\frac{\Sigma[\ell][n] = t}{\langle \Sigma, readRef\ (Res\ n) \rangle \longrightarrow \langle \Sigma, return\ t \rangle}$$

Figure 19: Semantics for memory operations.

Sequential configurations are reduced according to relation $c_1 \longrightarrow c_2$, where configuration c_1 steps to c_2. Every pure reduction $t_1 \rightsquigarrow t_2$ can be lifted to $\langle \Sigma, t_1 \rangle \longrightarrow \langle \Sigma, t_2 \rangle$, for some store Σ that remains unchanged. Figure 19 shows the interesting rules for *newRef*, *readRef* and *writeRef*, in which references are labeled with ℓ. Rule [NEW] extends the ℓ-labeled memory with the new term and returns a reference to it —memories are zero-indexed. Rule [WRITE] overwrites the content of the memory cell pointed by the reference and returns unit and [READ] retrieves the corresponding term from memory.

Simulation.

In order to prove that these operations do not break the security guarantees of **MAC**, we need to show that the *single-step simulation* property is preserved. We start by extending the erasure function for the new constructs. A configuration is erased by erasing respectively its store and term, i.e., $\varepsilon_{\ell_A}(\langle \Sigma, t \rangle) = \langle \varepsilon_{\ell_A}(\Sigma), \varepsilon_{\ell_A}(t) \rangle$. Stores are erased pointwise, i.e., by erasing the memory at each security level and memories are either fully erased when sensitive, i.e., $\varepsilon_{\ell_A}(t_s :: Memory\ \ell) = \bullet$ if $\ell \not\sqsubseteq \ell_A$, or erased pointwise otherwise. We remark that reading from an erased memory results in an erased term, i.e., $\bullet[n] = \bullet$ and updating it has no effect, i.e., $\bullet[n \mapsto t] = \bullet$, and furthermore its size is secret too, i.e., $\mid \bullet \mid = \bullet$. We can show by straightforward induction that the new rules can be simulated under erasure. The key property that guarantees simulation is to completely rewrite high memories to $\bullet$, which precisely capture the attacker's knowledge, and is particularly important when allocating and writing to high memories. Allocation does not leak information through the address of the new reference because $\mid \bullet \mid = \bullet$ and writing to a $\bullet$-memory does not make any change ($\bullet[\bullet \mapsto t] = \bullet$). Contrary to **LIO** memory model [Stefan et al., 2012b], allocation in a sensitive memory results in a low-equivalent memory, because the erased memory, before and after allocation, is $\bullet$. Although Haskell's memory is non-split, security guarantees are not compromised, because references are part of **MAC**'s internals and they cannot be inspected or deallocated explicitly.

D. PSNI

The statement of theorem 2 is standard, and therefore we only sketch its proof. We start by dividing it into two cases depending on whether the attacker can observe the scheduled thread. If the attacker cannot observe the scheduled thread, the changes made in the reduction can only affect high parts of the configuration, leading to a low-equivalence configuration. As a result, the second configuration is already transitively low-equivalent without taking any step. Instead, if the step involves an observable thread, we can show that a low-equivalent step can be made in the second configuration, possibly preceded by a finite number of steps involving exclusively high threads.

PROOF. By case analysis on $\ell \sqsubseteq \ell_A$.

- ($\ell \not\sqsubseteq \ell_A$). The execution of a thread in c_1 at security level ℓ cannot affect anything below ℓ in c_1', therefore $c_1 \approx_{\ell_A} c_1'$. Configuration c_2 steps to c_2 in 0 steps ($c_2 \hookrightarrow^{\star} c_2$), by transitivity of $\approx_{\ell_A}$ it follows that $c_1' \approx_{\ell_A} c_2$.
- ($\ell \sqsubseteq \ell_A$). By scheduler requirement 2, c_2 schedules either a high thread or thread (ℓ, n). In the first case, the proof follows by *well-founded* induction, otherwise the two ℓ_A-equivalent threads reduces triggering ℓ_A-equivalent events the thesis follow from Theorem 1 (PINI), appropriately generalized with events.

□