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 mutable 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, 2012, Stefan et al., 2011, 2012a, and as a combination of both [Buiras et al., 2015]. Many of such libraries, namely MAC [Russo, 2017, LIO [Stefan et al., 2011], and HLIO [Buiras et al., 2017], 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 mutable 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-
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 lead 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., 2012]. 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., 2012, Heule et al., 2013]. 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, 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.

## 2. THE CORE CALCULUS

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

### Calculus.

Figure 1 shows the formal syntax of the calculus, where meta variables \(\ell, \tau, v, t\) 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 \(\ell\)—this decision is made merely for clarity of exposition. We assume that labels form a lattice \((\mathcal{L}, \sqsubseteq)\) for network communication, and \((\mathcal{H}, \sqsubseteq)\) for references to integers, with \(\mathcal{L} \supseteq \mathcal{H}\). In examples we use the concrete classic two-point lattice with labels \(\mathcal{H}\) and \(\mathcal{L}\) denoting secret (high) and public (low) data respectively—where \(\mathcal{H} \supseteq \mathcal{L}\) is the only disallowed flow. Term MAC is the constructor of type MAC \(\ell \tau\), which denotes a (possibly) side-effectful secure computation that handles information at sensitivity level \(\ell\) and yields a result of type \(\tau\) at the same security level. Constructor \(\text{Res}\) 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 \(\text{Id} \tau\) is used to denote resources which do not trigger side-effects when manipulated, e.g., numbers or strings. For instance, \(\text{Res} (\text{Id} 42)\) \(\equiv\) \(\text{Res} \ell (\text{Id} \text{Int})\) represents a resource labeled with \(\ell\), whose content is the number 42. We use Haskell notation \(t \triangleright\triangleright t\) to denote that term \(t\) has type \(\tau\). By instantiating \(\tau\) in \(\text{Res} \ell \tau\) with different types (like we just did with \(\text{Id}\)), MAC is able to identify and securely provide operations on many kind of resources, e.g., \(\text{Res} \ell (\text{Int} \text{Ref})\) for references to integers, \(\text{Res} \ell (\text{Socket ByteStream})\) 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 \(\text{Res} \ell (\text{Id} \tau)\), which we abbreviate with the

---

\(^1\)The formal guarantees of HLIO are simply reduced to those of LIO.

\(^2\)Available at https://bitbucket.org/MarcoVassena/mac-agda
return :: τ → MAC ℓ τ
(≫=) :: MAC ℓ τ₁ → (τ₁ → MAC ℓ τ₂) → MAC ℓ τ₂

**Type.**

The typing judgment \( \Gamma \vdash t : \tau \) denotes that term \( t \) has type \( \tau \) assuming the typing environment \( \Gamma \). 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 \( \text{return} \), \( \ggseq \), \( \text{label} \), \( \text{unlabel} \), and \( \text{join} \) as Haskell APIs—see Figure 2. We explain their typing with traditional typing judgments by means of an example. The typing judgment of \( \text{return} \) is given in Figure 3. Note that rule [RETURN] is a rule scheme, i.e., there is such a judgment for every label \( \ell \in \mathcal{L} \), 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 \( \ell_{\ell} \) and \( \ell_{\bullet} \) to attest that \( \ell_{\ell} \subseteq \ell_{\bullet} \). Term \( \text{label} \) creates a new labeled expression, which is considered as a write operation from the security point of view. Consequently, the type constraint \( \ell_{\ell} \subseteq \ell_{\bullet} \) enforces the no write-down rule. It requires the label \( \ell_{\ell} \) of the MAC computation to be no more confidential than \( \ell_{\bullet} \), i.e., the label of the created labeled expression. Term \( \text{unlabel} \) performs a read operation, therefore, to comply with the no read-up rule, its type protects the confidentiality of \( \ell_{\bullet} \) of the result produced by the MAC computation. To achieve that, type constraint \( \ell_{\ell} \subseteq \ell_{\bullet} \) ensures that the result of the computation may only involve unlabeled expressions \( \ell_{\ell} \) which are, at most, as sensitive as \( \ell_{\bullet} \). Lastly, \( \text{join} \) protects the result of a sensitive MAC computation inside a less sensitive one by the constraint \( \ell_{\ell} \subseteq \ell_{\bullet} \), avoiding the label creep problems in sequential programs [Russinovich et al. 2013]. We remark that these type constrains are built using type classes, a well-established feature of Haskell type systems—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 \text{label}, \text{unlabel}, and \text{join}.

**Example.**

The most common use of \text{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 \( \text{pwd} \) should be handled with care in the rest of the program, which we symbolize with ellipsis (\( \ldots \)). One way to protect \( \text{pwd} \) is by writing all password-related operations within MAC, where \( \text{pwd} \) 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)
runMAC (lpwd ≫= common)
```

Observe how \text{label} is used to mark \( \text{pwd} \) as sensitive by wrapping it inside a labeled expression of type \( \text{Labeled H String} \). After that, the labeled password is passed to the function \( \text{common} \) by bind (\( \ggseq \)). (Function \( \text{runMAC} \) simply runs the given MAC-computation.) Assuming that \text{common} first fetches online dictionaries, pre-processes them, and then inspects if the password appears in them, the type for such code could be \( \text{common} :: \text{Labeled H String} \rightarrow \text{MAC L (MAC H Bool)} \). The reason for having nested MAC-computations comes from the fact that \text{common} is handling information with both sensitivities, i.e., \( \text{L} \) and \( \text{H} \). The outermost computation (\( \text{MAC L} \)) is responsible to fetch the online dictionaries from the web, an action consid-
Figure 5: Single-step simulation.

erased 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 → 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 \( t \sim t' \), which denotes that \( t \) reduces to \( t' \). Most of the reduction rules are standard and thus omitted. Figure 4 shows the interesting rules for constructs `unlabel`, `Res`, 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 et al., 2008, Stefan et al., 2011, 2012b, Heule et al., 2013] proves the simulation by relating small-step reductions (upper part in Figure 5) with reductions on a \( \ell_A \)-indexed small-step relation of the form \( t \sim_{\ell_A} t' \), 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 \( \sim_{\ell_A} \).

**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, Heule et al., 2013]. The technique relies on an erasure function on terms, which we denote by \( \varepsilon_{\ell_A} \). This function essentially erases data above the attacker’s security level, denoted by label \( \ell_A \), to the special syntax node \( \bullet \). Once \( \varepsilon_{\ell_A}(t) \) 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 \( \ell_A \), 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.

**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, Heule et al., 2013] prove the simulation by relating small-step reductions (upper part in Figure 5) with reductions on a \( \ell_A \)-indexed small-step relation of the form \( t \sim_{\ell_A} t' \), 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 \( \sim_{\ell_A} \).

**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 \( \tau \) 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., \( \varepsilon_{\ell_A}(t_1 \text{ if } t_2 \text{ else } t_3 :: ()) = \text{ if } \varepsilon_{\ell_A}(t_1 :: \text{Bool}) \text{ then } \varepsilon_{\ell_A}(t_2 :: ()) \text{ else } \varepsilon_{\ell_A}(t_3 :: ()) \). 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 \( \ell \) is below the attacker’s label \( \ell_A \), otherwise it is rewritten to \( \bullet \). Secure computations of type `MAC \ell \tau` are instead completely collapsed to \( \bullet \) when \( \ell \) 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 \( \ell_M \) such that \( \ell_M \not\subseteq \ell_A \), 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
\[
\begin{align*}
\varepsilon_{t_A}(\text{label } t :: MAC \ell_M (\text{Labeled } \ell \tau)) &= \\
&= \begin{cases} \\
\text{label } \varepsilon_{t_A}(t) & \text{if } \ell \subseteq \ell_A \\
\text{join } \varepsilon_{t_A}(t) & \text{otherwise}
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\varepsilon_{t_A}(\text{join } t :: MAC \ell_M (\text{Labeled } \ell \tau)) &= \\
&= \begin{cases} \\
\text{join } \varepsilon_{t_A}(t) & \text{if } \ell \subseteq \ell_A \\
\text{label } \varepsilon_{t_A}(t) & \text{otherwise}
\end{cases}
\end{align*}
\]

Figure 7: Two-steps erasure function.

MAC \ell_M (\text{Labeled } H \tau) indicates that \( t \) is sensitive, therefore we should rewrite it to \( \bullet \), i.e., \( \varepsilon_L(\text{label } t) = \text{label } \bullet \).

By doing so, however, the commutativity of the diagram in Figure 5 brakes. On the cyan path, we obtain that \( \text{label } t \rightsquigarrow \text{return } (\text{Res } (\text{Id } t)) \) (by rule \([\text{LABEL}]\)) and that \( \varepsilon_L(\text{return } (\text{Res } (\text{Id } t))) \equiv \text{return } \varepsilon_L(\text{Res } (\text{Id } t)) \) (by applying erasure homomorphically to return), where \( \text{return } \varepsilon_L(\text{Res } (\text{Id } t)) \equiv \text{return } (\text{Res } \bullet) \) (by erasure on sensitive labeled expressions—see Figure 6). In contrast, on the orange path, we have that \( \varepsilon_L(\text{label } t) = \text{label } \bullet \rightsquigarrow \text{return } (\text{Res } (\text{Id } \bullet)) \) (by \([\text{LABEL}]\)). To adhere to commutativity, the terms at the end of both paths should be the same, which is not the case here, i.e., \( \text{return } (\text{Res } \bullet) \not\equiv \text{return } (\text{Res } (\text{Id } \bullet)) \).

The problem arises from the fact that even after erasing every piece of sensitive information from \( t \), namely by rewriting \( t \) to \( \bullet \), rule \([\text{LABEL}]\) still produces the constructor \( \text{Id} \), which instead gets erased on the cyan path. Observe that rule \([\text{LABEL}]\) seems to intrinsically break simulation, regardless of the choice of the erasure function.

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

The obstacles encountered when erasing \( \text{label} \) and \( \text{join} \) while guaranteeing single-step simulation arise 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 \([\text{LABEL}]\) is broken despite how we erase its arguments: construct \( \text{label} \) always yields construct \( \text{Id} \) on the orange path independently of its argument, which is instead always erased on the cyan path since it occurs inside constructor \( \text{Res} \). Secondly, although the erasure definition could be adapted to restore commutativity of Figure 5 for \( \text{join} \), it will necessarily 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 \( \text{join } t :: MAC \ell_M (\text{Labeled } H \tau) \). Recall that rule \([\text{JOIN}]\) evaluates a computation \( t \) embedded in \( \text{join } t \) 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 \( \text{join} \), which we will capture in a different auxiliary erasure function \( \varepsilon'_L \). Suppose we defined \( \varepsilon_L(\text{join } t :: MAC \ell_M (\text{Labeled } H \tau)) = \text{join } \varepsilon'_L(t :: MAC \ H \tau) \), for some suitable \( \varepsilon'_L \) 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., \( \varepsilon_L(\lambda t_1 t_2 :: MAC \ell_M (\text{Labeled } H \tau)) = \lambda t_1 t_2 :: MAC t_1 t_2 \) —an essential property for the erasure function to have\(^4\) \([\text{Li} \text{ and } \text{Zdancewic}, 2010]\). Special constructs that do not use big-step semantics, such as those discussed in\(^5\) [\text{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 \( \text{label} \) and \( \text{join} \). Special constructs \( \text{label} \) and \( \text{join} \) replace terms \( \text{label} \) and \( \text{join} \) 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 \( MAC \bullet \not\equiv MAC \bullet \).

\(^4\)However, constructs that do not use big-step semantics, such as those discussed in [\text{Vassena et al., 2010}].

\(^5\)Cannot be simulated in the same way because their context rules would require to reduce a value, i.e., \( MAC \bullet \rightsquigarrow MAC \bullet \).
dionally, rules [LABEL] and [JOIN] 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 \( t \leadsto return \) (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 \( \varepsilon_{\ell A}(t) \) (by erasure), which then reduces according to rule [LABEL] as label \( \varepsilon_{\ell A}(t) \leadsto return \) (Res \( \bullet \))—thus, commuting precisely with the cyan path. Note that rule [LABEL], 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, \( \varepsilon_{\ell A}(t) \), and then reduces join, \( \varepsilon_{\ell A}(t) \leadsto return \) (Res \( \bullet \)) (by rule [JOIN]), which, just like [LABEL], guarantees simulation because it does not introduce constructor Id. Note that simulation holds also for rules [LABEL] and [JOIN], i.e., we are not simply moving the problem from constructs label and join to label and join. 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 determinacy of the small step semantics. In the following, we assume well-typed terms.

**Proposition 1 (Single-Step Simulation).** If \( t_1 \leadsto t_2 \) then \( \varepsilon_{\ell A}(t_1) \leadsto \varepsilon_{\ell A}(t_2) \).

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

**Proposition 2 (Determinancy).** If \( t_1 \leadsto t_2 \) and \( t_1 \leadsto t_3 \) 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 (\( \ell_A \)-equivalence).** Two terms \( t_1 \) and \( t_2 \) are indistinguishable from an attacker at security level \( \ell_A \), written \( t_1 \equiv_{\ell_A} t_2 \), 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 \( \ell_A \)-equivalence.

**Proposition 3 (\( \ell_A \)-equivalence Preservation).** If \( t_1 \equiv_{\ell_A} t_2 \), \( t_1 \leadsto t_1' \), and \( t_2 \leadsto t_2' \), then \( t_1' \equiv_{\ell_A} t_2' \).

Conventionally, Proposition 3 would use relation \( \leadsto^* \), i.e., the reflexive transitive closure of \( \leadsto \), 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 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 [Russell 2013]. 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. [2012a].

**Calculus, Terms, Types, and Semantics.**

We extend the syntactic categories from our core calculus as described in Figure 8. We introduce a value \( \ell \xi \) of exception type \( \chi \) and a new constructor MAC \( \tau \), denoting a failing computation due to exception \( \ell \). Terms throw \( t \) and catch \( t \) aborts the current MAC computation with exception \( \ell \) and recover from an exception thrown in computation \( t \) running exception handler \( t \), 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. 2012c, 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 implicit flow! (We refer interested readers to [Russell 2013] 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-
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 ℓ τ data type, i.e., the type of data produced by join. Specifically, we add internal constructor ResX t, where t :: χ. 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 11, we extend the semantics of unlabel with rule [UNLABELX], which handles constructor ResX 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 MACX, throw and catch to • if their computation label is sensitive; otherwise, erasure is applied homomorphically. The erasure of ResX deserves more attention—see Figure 11. Note that the content of a sensitive exception is rewritten to • as expected, but also the constructor ResX 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 [UNLABELX] 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 • 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]. The key observation is that a computation t :: MAC ℓH τ embedded in join t :: MAC ℓL τ (Labeled ℓH τ) may not terminate depending on the value of the secret. If the computation t diverges, it may suppress public side-effects following join t, thus revealing a bit about the secret. To illustrate this point, consider the function send::Int → MAC L () which sends an integer to the attacker’s server. By wrapping join t among two instances of send, i.e., attacker n = send n ≫= λ () → join t ≫= λ () → send n, and assuming that t diverges if the secret is true, then the attacker knows that the secret is false when it receives mn 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.).

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, non-terminating 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 ℓH ()) from non-sensitive ones (of type MAC ℓL ()) because the decision depends on data at level ℓL, which is more sensitive (ℓL ⊆ ℓH). 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 non-interference (PSNI).

Calculus.

Figure 13 extends the calculus from Section 2 with concurrency. It introduces global configurations of the form ⟨s, Φ⟩ 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 τ () which get organized in isolated thread pools according to their security label. A

4 Furthermore, the presence of threads introduce the interference (PSNI).
\[
\Phi[i][n] = t_1 \xrightarrow{t_1} t_2 \xrightarrow{\ell, n, e} s_1 \xrightarrow{(s_1, \Phi)} \langle s_2, \Phi[i][n] := t_2 \rangle
\]

**Figure 14:** Scheme rule for concurrent semantics.

The current thread has forked a new thread identified by \( \ell \). We use the standard list interface \([\_]\), \( t : ts \), and \( ts[n] \) for the empty list, the insertion of a term into an existing list, and accessing the n-th element, respectively. We write \( \Phi[i][n] = t \) to retrieve the n-thread in the \( \ell \)-thread pool—it is just syntax sugar for \( \Phi(\ell) = t_\ell \) and \( t_\ell[n] = t \). The notation \( \Phi[i][n] := t \) 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., \( \bullet[n \mapsto] = \bullet \). 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 \( \rightarrow \). 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 \( (\ell, n) \), which is retrieved from the configuration \( \Phi[i][n] = t_1 \) and executed \( (t_1 \xrightarrow{\ell} t_2) \). We decorate the sequential semantics \( \xrightarrow{\_} \) 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 \( \text{Step} \) denotes a single sequential step, event \( \text{Skip} \) denotes that a thread is stuck, e.g., on a synchronization variable, event \( \text{Done} \) is generated when a thread has terminated and event \( \text{Fork} \) \( \ell \) \( n \) informs the scheduler that the current thread has forked a new thread identified by \( (\ell, n) \). Event \( \bullet \) is triggered by an erased thread \( \bullet \), that is \( \bullet \xrightarrow{\_} \bullet \). Lastly, the thread pool is updated with the final state of the thread \( \Phi[i][n] := t_2 \).

**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 deterministic, 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 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:**

\[ \text{if } s_1 \xrightarrow{(\ell, n, e)} s_2 \text{ and } s_1 \xrightarrow{(\ell, n)} s_3, \text{ then } s_2 = s_3. \]

ii) **Single-step simulation:**

\[ \text{if } s_1 \xrightarrow{(\ell, n, e)} s_2, \text{ then } \varepsilon_A(s_1) \xrightarrow{(\ell, n, e, e)} \varepsilon_A(s_2). \]

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

Observe that determinancy of the scheduler is essential for determinacy of the concurrent semantics—after all, the scheduler state is part of the concurrent configuration. Nevertheless, Requirement 1 is slightly weaker than what one might expect: it assumes that the scheduled thread \( (\ell, n) \) is the same in both reductions. We discuss why we need to relax determinism in this way in Section 5.1. 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., \( \varepsilon_A(e :: \text{Event } \ell) = \bullet \) if \( \ell \notin A \). 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 \( \ell_A \)-equivalence between scheduler states and its annotated version to guarantee progress as follows.

**Definition 2 (Scheduler \( \ell_A \)-equivalence).**

i) Two states are \( \ell_A \)-equivalent, written \( s_1 \cong_{\ell_A} s_2 \) if and only if \( \varepsilon_A(s_1) \equiv_{\ell_A} \varepsilon_A(s_2) \).

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

The relation \( s_1 \cong^{(i, j)}_{\ell_A} s_2 \) captures an alignment measure of two \( \ell_A \)-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 \( \ell_A \)-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 \( s_1 \cong^{(i, j)}_{\ell_A} s_2 \) strictly decreases after every reduction. We capture the interplay between the \((i, j)\)-\( \ell_A \)-equivalence relationship and the evolution of schedulers by establishing unwinding-like conditions [Goguen and Meseguer, 198]. More specifically, we describe what occurs with the \((i, j)\) indexes when schedulers handle sensitive and observable events.
Requirement 2  (Non-Interfering Scheduler).
Given \( s_1 \xrightarrow{(l,n,e)} s_2, e \neq \bullet \), and \( s_1 \approx_{\ell_A} (i,j) s_1' \), then

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

- If \( \ell \not\subseteq \ell_A \), then \( s_2 \approx_{\ell_A} s_1' \)

Given two \((i,j)-\ell_A\)-equivalent scheduler states if one runs a non-sensitive thread (\( \ell \subseteq \ell_A \)), then the other scheduler state is part of 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 \( \text{Step} \)), the current thread has used up its time slot and is enqueued. If the scheduled thread cannot execute (event \( \text{Skip} \)), it is skipped and enqueued as well. When the current thread has terminated (event \( \text{Done} \)), the thread is not alive anymore and hence removed from the queue. Message \((t_L, n_1, \text{Fork} t_H n_2)\) informs the scheduler that thread \((t_L, n_1)\) has spawned thread \((t_H, n_2)\), which is then enqueued with the current thread. The last rule is not part of the actual scheduling algorithm and 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 \textit{deterministic}, i.e., it fulfills scheduler requirement 1\H. We define the erasure function to filter out the identifiers of threads non observable to the attacker, i.e., \( \varepsilon_{\ell_A} (s) = \lambda (\ell, n) \rightarrow \ell \not\subseteq \ell_A) s \). By induction on the scheduler reduction, it follows that round-robin satisfies the \textit{single-step} simulation, i.e., scheduler requirement 1\I. Note that round-robin is \textit{starvation-free} because it has a finite time slot and is preemptive. We remark that absence of \textit{starvation} is a desirable property of schedulers, which is sufficient to guarantee progress, i.e., scheduler requirement 1\H. 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 \( s_1 \approx_{L} s_2 \) 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 \textit{single-step simulation} and \textit{determinacy}. 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., \( \varepsilon_{\ell_A} (s, \Phi) = (\varepsilon_{\ell_A} (s), \varepsilon_{\ell_A} (\Phi)) \), thread pools are erased pointwise and pools are erased according to their label, i.e., \( \varepsilon_{\ell_A} (t_L :: \text{Memory} \ell) = \bullet \) if \( \ell_A \not\subseteq \ell \) and homomorphically erased otherwise.

**Proposition 5  (Single-step Simulation).** If \( c_1 \leftrightarrow c_2 \) then \( \varepsilon_{\ell_A} (c_1) \leftrightarrow \varepsilon_{\ell_A} (c_2) \).

Proposition 5 follows immediately by \textit{single-step} simulation of the sequential calculus and scheduler requirement 1\H.
We now discuss a subtle distinction between predictability and determinancy, two slightly different properties that come into play when using schedulers. To prove non-interference, we need to show determinancy of the concurrent semantics, i.e., if \( c_1 \rightarrow c_2 \) and \( c_1 \rightarrow c_3 \) then \( c_2 \equiv c_3 \). Note that the term erasure proof technique requires to construct a simulation between \( \ell_\alpha \)-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 \( s_1 \xrightarrow{\langle \ell, n, e \rangle} s_2 \) and \( s_1 \xrightarrow{\langle \ell', n', e' \rangle} s_3 \), 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 \( s_1 \xrightarrow{\langle \ell, n, e \rangle} s_2 \) may break this property. For instance, consider rule \( s \xrightarrow{\langle \ell, n, \bullet \rangle}^{RR} s \) of the round-robin scheduler (see Figure 15). Such a rule schedules a thread (\( \ell, n \)) nondeterministically, 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 lose 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 \( (s_1, \Phi_1) \rightarrow (s_2, \Phi_2) \) if and only if \( s_1 \xrightarrow{\langle \ell, n, e \rangle} s_2 \) (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 \( c_1 \xrightarrow{\langle \ell, n, e \rangle} c_2 \) and \( c_1 \xrightarrow{\langle \ell, n \rangle} c_3 \) then \( c_2 \equiv c_3 \).

We now proceed to prove non-interference. We ordinarily extend \( \ell_\alpha \)-equivalence to global configurations, written \( c_1 \equiv_\alpha c_2 \), and if only if \( \varepsilon_\alpha(c_1) = \varepsilon_\alpha(c_2) \). Relation \( \equiv_* \) denotes the transitive reflexive closure of \( \equiv \).

**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 \( 1 \), \( 1a \), \( 1ab \), and \( 2 \), if \( c_1 \equiv_\alpha c_2 \) and \( c_1 \xrightarrow{\langle \ell, n \rangle} c'_1 \), then there exists \( c'_2 \) such that \( c_2 \xrightarrow{\ell} c'_2 \) and \( c_2 \equiv_\alpha c'_2 \).

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 asserts 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, 2012] 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 [Hriciu 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 implicit 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, 2007]. 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 \( \pi \)-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., 2010] 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, 1973, 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. 

\cite{estathopoulos2005security, zeldovich2006information, krohn2007information, murray2013information}. Due to its complexity, it is not surprising that OS-based MAC systems lack accompanying soundness guarantees or mechanized proofs—\cite{seL4} being the exception. 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 \cite{stefan2012retrofitting, heule2015secure}. 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


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 \( \varepsilon_t (\text{label } t_1 t_2) \) is context-sensitive: \( t_2 \) should be rewritten to \( \bullet \) if \( t_1 \not
\preceq \ t_2 \) or to \( \varepsilon_t (t_2) \) otherwise. Note that due to the dynamic nature of LIO, labels are terms and \( t_1 \) is a concrete label only when fully evaluated \cite{Stefan2011, Stefan2012b}. As a result, the \( \preceq \) relation is defined between terms, instead of plain labels, and it is partial when it involves unevaluated terms. Unfortunately, that fact that \( \preceq \) 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, \( \varepsilon_t (\text{label } (\lambda \ell. \ell) \, L) \) is erased to \( \varepsilon_t (\text{label } (\lambda \ell. \ell) \, L) \bullet \), because the erasure considers that \( (\lambda \ell. \ell) \, L \not
\preceq \ L\) — 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 \[\text{labels}]\, triggered by construct \text{labels}, inserted by the erasure function. The semantics rules of \text{labels} 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 \cite{Stefan2011, Stefan2012b}. Our memory model provides deterministic allocation in a memory partitioned by security labels to avoid establishing a bijection between heaps in our proofs \cite{Banerjee2002}. 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 \cite{Stefan2011}, the erasure of construct \text{newRef } \ell t\ and \text{writeRef } a t\ is incorrect, because it homomorphically erases \( t \), irrespectively of the sensitivity of the expression written to memory — it should instead rewrite \( t \).

\footnote{Coq model at \url{https://github.com/deian/lio-semantics/}}
to • when ℓ is not visible to the attacker or when writing to a sensitive region in memory, just like for label τ 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: (Σ, newRef H t) → (Σ', return a), where Σ' = Σ [a → Labeled H t] for some fresh address a (cyan path). Proving simulation requires to show that configuration (εL(Σ), newRef H εL(t)) reduces to (εL(Σ'), return a) (orange path). Unfortunately, in this case the diagram does not commute. By rule [NRef], configuration (εL(Σ), newRef H εL(t)) reduces to a configuration with memory εL(Σ) [a → Labeled H εL(t)], which is different from εL(Σ'). Observe that this configuration erases the memory pointwise and thus resulting in mapping a to Labeled H • rather than Labeled H t as done by memory εL(Σ) [a → Labeled H εL(t)].

In a draft version of [Stefan et al. 2012], which surpasses [Stefan et al. 2011], we have identified a second problem that concerns low-equivalence of memories, which is defined as εL(Σ1) ≡ εL(Σ2) — memories are erased point-wise. This definition is too restrictive because rules out memories with different number of sensitive locations as equivalent, even though the attacker cannot distinguish them. For instance, according to [Stefan et al. 2012], memories Σ1 = [0 → Labeled H t0, 1 → Labeled H t1] and Σ2 = [0 → Labeled H t0'] are not low-equivalent because εL(Σ1) ≠ εL(Σ2), since [0 → Labeled H •, 1 → Labeled H •] ≠ [0 → Labeled H •]. 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 Σ[ℓ][n] := t to rename the nth-cell in the ℓ-memory —it is an infix sugar for Σ(ℓ) = t and τ t = t. The notation Σ[ℓ][n] := t denotes the store obtained by performing the update Σ(ℓ)[n] := t. Lastly, we write | t | = n to denote that memory tt has length n.

We write (Σ, t) 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 τ t represents a labeled reference to the n-th cell of memory labeled with ℓ, which contains a term of type τ.

Sequential configurations are reduced according to relation c1 → c2, where configuration c1 steps to c2. Every pure reduction t1 → t2 can be lifted to (Σ, t1) → (Σ, t2), 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., εL(⟨Σ, t⟩) = εL(Σ), εL(t)). Stores are erased pointwise, i.e., by erasing the memory at each security level and memories are either fully erased when sensitive, i.e., εL(t := Memory ℓ) = • if ℓ ⊈ ℓs, or erased pointwise otherwise. We remark that reading from an erased memory results in an erased term, i.e., •[n] = • and updating it has no effect, i.e., •[n] := t = •, and furthermore its size is secret too, i.e., •[•] = •. 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 •, which precisely captures 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 | • | = • and writing to a •-memory does
not make any change ($\bullet \bullet \rightarrow \ell = \bullet$). Contrary to LIO memory model [Stefan et al. 2012], 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 $\ell \subseteq \ell_A$.

- ($\ell \not\sqsubseteq \ell_A$). The execution of a thread in $c_1$ at security level $\ell$ cannot affect anything below $\ell$ in $c_1'$, therefore $c_1 \approx_{\ell_A} c_1'$. Configuration $c_2$ steps to $c_2$ in 0 steps ($c_2 \rightarrow^* c_2$), by transitivity of $\approx_{\ell_A}$ it follows that $c_1' \approx_{\ell_A} c_2$.

- ($\ell \sqsubseteq \ell_A$). By scheduler requirement, $c_2$ schedules either a high thread or thread ($\ell, n$). In the first case, the proof follows by well-founded induction, otherwise the two $\ell_A$-equivalent threads reduces triggering $\ell_A$-equivalent events the thesis follow from Theorem 1 (PINI), appropriately generalized with events.

\qed