Flexible Manipulation of Labeled Values for Information-Flow Control Libraries

Extended Version

Marco Vassena[1], Pablo Buiras[1], Lucas Waye[2], and Alejandro Russo[1]

[1] Chalmers University of Technology
{vassena,buiras,russo}@chalmers.se
[2] Harvard University
lwaye@seas.harvard.edu

Abstract. The programming language Haskell plays a unique, privileged role in Information-Flow Control (IFC) research: *it is able to enforce information security via libraries*. Many state-of-the-art libraries (e.g., **LIO**, **HLIO**, and **MAC**) allow computations to manipulate data with different security labels by introducing the notion of *labeled values*, which protect values with explicit labels by means of an abstract data type. While computations have an underlying algebraic structure in such libraries (i.e. monads), there is no research on structures for labeled values and their impact on the programming model. In this paper, we add the *functor* structure to labeled values, which allows programmers to conveniently and securely perform computations without side-effects on such values, and an *applicative* operator, which extends this feature to work on multiple labeled values combined by a multi-parameter function. This functionality simplifies code, as it does not force programmers to spawn threads to manipulate sensitive data with side-effect free operations. Additionally, we present a *relabel* primitive which securely modifies the label of labeled values. This operation also helps to simplify code when aggregating data with heterogeneous labels, as it does not require spawning threads to do so. We provide mechanized proofs of the soundness our contributions for the security library **MAC**, although we remark that our ideas apply to **LIO** and **HLIO** as well.

1 Introduction

Nowadays, many applications (apps) manipulate users' private data. Such apps *could have been written by anyone* and users who wish to benefit from their functionality are forced to grant them access to their data—something that most users will do without a second thought [21]. Once apps collect users' information, there are no guarantees about how they handle it, thus leaving room for data theft and data breach by malicious apps. The key to guaranteeing security without sacrificing functionality is not granting or denying access to sensitive data, but rather ensuring that *information only flows into appropriate places*.

Information-flow control (IFC) [32] is a promising programming language-based approach to enforcing security. IFC scrutinizes how data of different sensitivity levels (e.g., public or private) flows within a program, and raises alarms when there is an

unsafe flow of information. Most IFC tools require the design of new languages, compilers, interpreters, or modifications to the runtime, e.g., [4, 24, 26, 29]. In this scenario, the functional programming language Haskell plays a unique privileged role: *it is able to enforce security via libraries* [18] by using an embedded domain-specific language.

Many of the state-of-the-art Haskell security libraries, namely **LIO** [37], **HLIO** [6], and **MAC** [31], bring ideas from Mandatory Access Control [3] into a language-based setting. Every computation in such libraries has a *current label* which is used to (i) approximate the sensitivity level of all the data in scope and (ii) restrict subsequent side-effects which might compromise security. From now on, we simply use the term libraries when referring to **LIO**, **HLIO**, and **MAC**.

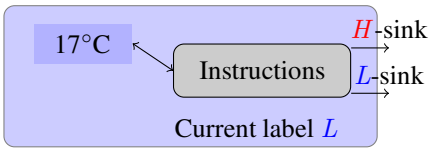

Fig. 1: Public computation

IFC uses labels to model the sensitivity of data, which are then organized in a security lattice [7] specifying the allowed flows of information, i.e., $\ell_1 \sqsubseteq \ell_2$ means that data with label ℓ_1 can flow into entities labeled with ℓ_2. Although these libraries are parameterized on the security lattice, for simplicity we focus on the classic two-point lattice with labels H and L to respectively denote secret (high) and public (low) data, and where $H \not\sqsubseteq L$ is the only disallowed flow. Figure 1 shows a graphical representation of a public computation in these libraries, i.e. a computation with current label L. The computation can read or write data in scope, which is considered public (e.g., average temperature of 17°C in the Swedish summer), and it can write to public (L-) or secret (H-) sinks. By contrast, a secret computation, i.e. a computation with current label H, can also read and write data in its scope, which is considered sensitive, but in order to prevent information leaks it can *only* write to sensitive/secret sinks. Structuring computations in this manner ensures that sensitive data does not flow into public entities, a policy known as noninterference [10]. While secure, programming in this model can be overly restrictive for users who want to manipulate differently-labeled values.

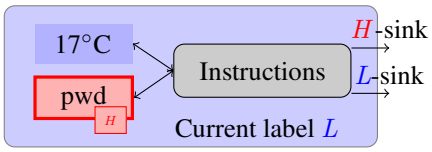

Fig. 2: Labeled values

To address this shortcoming, libraries introduce the notion of a *labeled value* as an abstract data type which protects values with explicit labels, in addition to the current label. Figure 2 shows a public computation with access to both public and sensitive pieces of information, such as a password (pwd). Public computations can freely manipulate sensitive labeled values provided that they are treated as black boxes, i.e. they can be stored, retrieved, and passed around as long as its content is not inspected. Libraries **LIO** and **HLIO** even allow public computations to inspect the contents of sensitive labeled values, raising the current label to H to keep track of the fact that a secret is in scope—this variant is known as a *floating-label* system.

Reading sensitive data usually amounts to "tainting" the entire context or ensuring the context is as sensitive as the data being observed. As a result, the system is susceptible to an issue known as *label creep*: reading too many secrets may cause the current

label to be so high in the lattice that the computation can no longer perform any useful side effects. To address this problem, libraries provide a primitive which enables public computations to spawn sub-computations that access sensitive labeled values without tainting the parent. In a sequential setting, such sub-computations are implemented by special function calls. In the presence of concurrency, however, they must be executed in a different thread to avoid compromising security through *internal timing* and *termination covert channels* [36].

Practical programs need to manipulate sensitive labeled values by transforming them. It is quite common for these operations to be naturally free of I/O or other side effects, e.g., arithmetical or algebraic operations, especially in applications like image processing, cryptography, or data aggregation for statistical purposes. Writing such functions, known as *pure* functions, is the bread and butter of functional programming style, and is known to improve programmer productivity, encourage code reuse, and reduce the likelihood of bugs [14]. Nevertheless, the programming model involving sub-computations that manipulate secrets forces an imperative style, whereby computations must be structured into separate compartments that must communicate explicitly. While side-effecting instructions have an underlying structure (called monad [22]), research literature has neglected studying structures for labeled values and their consequences for the programming model. To empower programmers with the simpler, functional style, we propose additional operations that allow pure functions to securely manipulate labeled values, specifically by means of a structure similar to *applicative functors* [20]. In particular, this structure is useful in concurrent settings where it is no longer necessary to spawn threads to manipulate sensitive data, thus making the code less imperative (i.e., side-effect free). Interestingly, the evaluation strategy of the host language (i.e. call-by-value or call-by-name) affects the validity of our security guarantees. Specifically, call-by-name turns out to naturally enforce progress-sensitive non-interference in a concurrent setting.

Additionally, practical programs often aggregate information from heterogeneous sources. For that, programs needs to upgrade labeled values to an upper bound of the labels being involved before data can be combined. In previous incarnations of the libraries, such relabelings require to spawn threads just for that purpose. As before, the reason for that is libraries decoupling every computation which manipulate sensitive data—even those for simply relabeling—so that the internal timing and termination covert channels imposed no threats. In this light, we introduce a primitive to securely relabel labeled values, which can be applied irrespective of the computation's current label and does not require spawning threads.

We provide a mechanized security proof for the security library **MAC** and claim our results also apply to **LIO** and **HLIO**. **MAC** has fewer lines of code and leverages types to enforce confidentiality, thus making it ideal to model its semantics in a dependently-typed language like Agda. The contributions of this paper are: (i) we introduce a *functor* structure equipped with an *applicative* operator that enables users to conveniently manipulate and combine labeled values using pure functions, encouraging a more functional (side-effect free) programming style; (ii) we introduce a relabeling primitive that securely modifies the label of labeled values, bypassing the need to spawn threads when aggregating heterogeneous data; (iii) we identify and discuss the impact

of the evaluation strategy of the host language on the security of the applicative operators in **MAC** with respect to the internal timing and termination covert channels; (iv) we implement a prototype of our ideas in the **MAC** library[3]; and (v) we formalize **MAC** with secure applicative operators as a λ-calculus, providing a mechanized proof in Agda of progress-insensitive (PINI) and progress-sensitive noninterference (PSNI) [1] for the sequential and (respectively) concurrent setting.

This paper is organized as follows. Section 2 describes the core aspects of **MAC**. Section 3 and 4 present functors, applicative, and relabeling operations. Section 5 gives formal guarantees. Section 6 gives related work and Section 7 concludes.

2 Background

In **MAC**, each label is represented as an abstract data type. Figure 3 shows the core part of **MAC**'s API. Abstract data type $\mathit{Labeled}\ \ell\ a$ classifies data of type a with a security label ℓ. For instance, $\mathit{creditCard} :: \mathit{Labeled}\ \ H\ \ \mathit{Int}$ is a sensitive integer, while $\mathit{weather} :: \mathit{Labeled}\ \ L\ \ \mathit{String}$ is a public string. (Symbol :: is used to describe the type of terms in Haskell.) Abstract data type $\mathit{MAC}\ \ \ell\ \ a$ denotes a (possibly) side-effectful secure computation which handles information at sensitivity level ℓ and yields a value of type a as a result. A $\mathit{MAC}\ \ \ell\ \ a$ computation enjoys a monadic structure, i.e. it is built using the fundamental operations $\mathit{return} :: a \rightarrow \mathit{MAC}\ \ \ell\ \ a$ and $(\gg\!=) :: \mathit{MAC}\ \ell\ a \rightarrow (a \rightarrow \mathit{MAC}\ \ell\ b) \rightarrow \mathit{MAC}\ \ell\ b$ (read as "bind"). The operation $\mathit{return}\ x$ produces a computation that returns the value denoted by x and produces no side-effects. The function $(\gg\!=)$ is used to *sequence* computations and their corresponding side-effects. Specifically, $m \gg\!= f$ takes a computation m and function f which will be applied to the *result* produced by running m and yields the resulting computation. We sometimes use Haskell's do-notation to write such monadic computations. For example, the program $m \gg\!= \lambda x \rightarrow \mathit{return}\ (x + 1)$, which adds 1 to the value produced by m, can be written as shown in Figure 4.

-- Abstract data types
data $\mathit{Labeled}\ \ell\ a$
data $\mathit{MAC}\ \ell\ a$

-- Monadic structure for computations
instance $\mathit{Monad}\ (\mathit{MAC}\ \ell)$

-- Core operations
$\mathit{label} :: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow a \rightarrow \mathit{MAC}\ \ell_{\mathrm{L}}\ (\mathit{Labeled}\ \ell_{\mathrm{H}}\ a)$
$\mathit{unlabel} :: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow \mathit{Labeled}\ \ell_{\mathrm{L}}\ a \rightarrow \mathit{MAC}\ \ell_{\mathrm{H}}\ a$

-- Only for sequential programs
$\mathit{join}^{\mathsf{MAC}} :: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow \mathit{MAC}\ \ell_{\mathrm{H}}\ a \rightarrow \mathit{MAC}\ \ell_{\mathrm{L}}\ (\mathit{Labeled}\ \ell_{\mathrm{H}}\ a)$

-- Only for concurrent programs
$\mathit{fork}^{\mathsf{MAC}} :: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow \mathit{MAC}\ \ell_{\mathrm{H}}\ () \rightarrow \mathit{MAC}\ \ell_{\mathrm{L}}\ ()$

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

[3] `https://hackage.haskell.org/package/mac`

```
do x ← m
   return (x + 1)
```

Fig. 4: **do**-notation

Secure flows of information Generally speaking, side-effects in a $MAC\ \ell\ a$ computation can be seen as actions which either read or write data. Such actions, however, need to be conceived in a manner that respects the sensitivity of the computations' results as well as the sensitivity of sources and sinks of information modeled as labeled values. The functions *label* and *unlabel* allow $MAC\ \ell\ a$ computations to securely interact with labeled values. To help readers, we indicate the relationship between type variables in their subindexes, i.e. we use ℓ_{L} and ℓ_{H} to attest that $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$. If a $MAC\ \ell_{\mathrm{L}}$ computation writes data into a sink, the computation should have at most the sensitivity of the sink itself. This restriction, known as *no write-down* [3], respects the sensitivity of the data sink, e.g., the sink never receives data more sensitive than its label. In the case of function *label*, it creates a fresh labeled value, which from the security point of view can be seen as allocating a fresh location in memory and immediately writing a value into it—thus, it applies the no write-down principle. In the type signature of *label*, what appears on the left-hand side of the symbol $\Rightarrow$ are *type constraints*. They represent properties that must be statically fulfilled about the types appearing on the right-hand side of $\Rightarrow$. Type constraint $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$ ensures that when calling *label* x (for some x in scope), the computation creates a labeled value only if ℓ_{L}, i.e. the current label of the computation, is no more confidential than ℓ_{H}, i.e. the sensitivity of the created labeled value. In contrast, a computation $MAC\ \ell_{\mathrm{H}}\ a$ is only allowed to read labeled values at most as sensitive as ℓ_{H}—observe the type constraint $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$ in the type signature of *unlabel*. This restriction, known as *no read-up* [3], protects the confidentiality degree of the result produced by $MAC\ \ell_{\mathrm{H}}\ a$, i.e. the result might only involve data ℓ_{L} which is, at most, as sensitive as ℓ_{H}.

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

Fig. 5: Implicit flows are ill-typed.

The interaction between the current label of a computation and the no write-down restriction makes implicit flow ill-typed, as shown in Figure 5. In order to branch on sensitive data, a program needs first to unlabel it, thus requiring the computation to be of type $MAC\ H\ a$ (for some type a). From that point, the computation cannot write to public data regardless of the taken branch. As **MAC** provides additional primitives responsible for producing useful side-effects like exception handling, network communication, references, and synchronization primitives—we refer the interested reader to [31] for further details.

Handling data with different sensitivity Programs handling data with heterogeneous labels necessarily involve *nested* $MAC\ \ell\ a$ computations in its return type. For instance, consider a piece of code m with type $MAC\ L\ (String, MAC\ H\ Int)$ which handles both public and secret information. Note that the type indicates that it returns a public string and a sensitive computation $MAC\ H\ Int$. While somehow manageable for a two-point lattice, it becomes intractable for general cases. In a *sequential setting*, **MAC** presents the primitive $join^{\mathsf{MAC}}$ to safely *integrate* more sensitive computations into less sensitive ones—see Figure 3. Operationally, function $join^{\mathsf{MAC}}$ runs the computation of

type $MAC\ \ell_{\mathrm{H}}\ a$ and wraps the result into a labeled expression to protect its sensitivity. As we will show in Section 5, Haskell programs written using the monadic API, *label*, *unlabel*, and $join^{\mathsf{MAC}}$ satisfy PINI, where leaks due to non-termination of programs are ignored. This design decision is similar to that taken by mainstream IFC compilers (e.g., [11, 25, 34]), where the most effective manner to exploit termination takes exponential time in the size (of bits) of the secret [1].

```
  -- Publish a number in a blog
publish :: Int → MAC L ()
  -- Attack
leak :: Int → Labeled H Secret → MAC L ()
leak n secret = do
  join^MAC (do bits ← unlabel secret
               when (bits !! n) loop
               return True)
  publish n
```

Fig. 6: Termination leak

Concurrency The mere possibility to run (conceptually) simultaneous $MAC\ \ \ell$ computations provides attackers with new tools to bypass security checks. In particular, the presence of threads introduce the internal timing covert channel, a channel that gets exploited when, depending on secrets, the timing behavior of threads affect the order of events performed on public-shared resources [35]. Furthermore, concurrency magnifies the bandwidth of the termination covert channel to be linear in the size (of bits) of secrets [36]. Since the same countermeasure closes both covert channels, we focus on the latter. What constitutes a termination leak is the fact that a non-terminating $MAC\ \ell_{\mathrm{H}}$-computation can suppress the execution of subsequently $MAC\ \ell_{\mathrm{L}}$-events. To illustrate this point, we present the attack in Figure 6. We assume that there exists a function *publish* which sends an integer to a public blog. Observe how function *leak* may suppress subsequent public events with infinite loops. If a thread runs $leak\ 0\ secret$, the code publishes 0 *only if* the first bit of *secret* is 0; otherwise it loops (see function *loop*) and it does not produce any public effect. Similarly, a thread running $leak\ 1\ secret$ will leak the second bit of *secret*, while a thread running $leak\ 2\ secret$ will leak the third bit of it and so on. To securely support concurrency, **MAC** forces programmers to decouple computations which depend on sensitive data from those performing public side-effects. As a result, non-terminating loops based on secrets cannot affect the outcome of public events. To achieve this behavior, **MAC** replaces $join^{\mathsf{MAC}}$ by $fork^{\mathsf{MAC}}$ as defined in Figure 3. It is secure to spawn sensitive computations ($MAC\ \ell_{\mathrm{H}}$) from non-sensitive ones ($MAC\ \ell_{\mathrm{L}}$) because that decision depends on data at level ℓ_{L}.

Example 1. To show how to program using **MAC**, we present a simple scenario where Alice writes an API that helps users prepare and file their taxes. Alice models a tax declaration as values of type *TaxDecl*, which is obtained based on users' personal information—modeled as values of type *Data*. She releases the first version of the API:

```
  -- API
declareTaxes :: Data → IO ()
declareTaxes user = send (fillTaxes user)
  -- Internal operations (not exported)
fillTaxes :: Data    → TaxDecl
send      :: TaxDecl → IO ()
```

We remark that, although we focus on this API for simplicity, Alice is using the concurrent version of **MAC**. Function $declareTaxes$ does two things: it fills out the tax forms (function $fillTaxes$) and sends them to the corresponding government agency (function $send$). Due to the use of $send$, function $declareTaxes$ returns a computation in the IO-monad—a special data type which permits arbitrary I/O effects in Haskell. Function $send$ generates a valid PDF for tax declarations and sends it to the corresponding authorities. However, there is nothing stopping this function from leaking tax information to unauthorized entities over the network. Alice's customers notice this problem and are concerned about how their sensitive data gets handled by the API.

Alice then decides to adapt the API to use **MAC**. For simplicity, we assume that **MAC** also includes a secure operation to send data over the network:

$$send^{\mathsf{MAC}} :: \ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}} \Rightarrow Labeled\ \ell_{\mathrm{H}}\ URL \rightarrow Labeled\ \ell_{\mathrm{H}}\ a \rightarrow MAC\ \ell_{\mathrm{L}}\ ()$$

This primitive sends a labeled value of type a to the URL given as an argument, e.g., via HTTP-request or other network protocol. Using **MAC**'s concurrent API and primitive $send^{\mathsf{MAC}}$, Alice rewrites her API to adhere to the following interface.

```
declareTaxes :: Labeled H URL → Labeled H Data → MAC L ()
declareTaxes url user = fork^MAC (do info ← unlabel user
                                     tax ← label (fillTaxes info)
                                     send^MAC url tax)
  -- Internal operations
fillTaxes :: Data → TaxDecl
```

Observe that Alice's API needs to spawn a secure computation of type $MAC\ H\ ()$ in order to unlabel and access user's data ($user$). Once user's data is accessible, a pure function gets applied to it ($fillTaxes\ \ info$), the result is relabeled (tax) again and a side-effectful action takes place ($send^{\mathsf{MAC}}$). In the next section we extend **MAC**'s API so that it is possible to manipulate labeled values with pure functions, like $fillTaxes$, and perform side-effectful actions, like $send^{\mathsf{MAC}}$, without the need to spawn threads.

3 Functors for Labeled Values

In this section, we show how labeled values can be manipulated using *functors*.

$$fmap :: (a \rightarrow b) \rightarrow Labeled\ \ell\ a \rightarrow Labeled\ \ell\ b$$

Fig. 7: Functor structure for labeled values

Intuitively, a functor is a container-like data structure which provides a method called $fmap$ that applies (maps) a function over its content, while preserving its structure. Lists are the most canonical example of a functor data-structure. In this case, $fmap$ corresponds to the function map,

which applies a function to each element of a list, e.g. $fmap\ (+1)\ [1,2,3] \equiv [2,3,4]$. A functor structure for labeled values allows to manipulate sensitive data without the need to explicitly extract it—see Figure 7. For instance, $fmap\ (+1)\ d$, where $d :: Labeled\ H\ Int$ stores the number 42, produces the number 43 as a sensitive labeled value. Observe that sensitive data gets manipulated without the need to use $label$ and $unlabel$, thus avoiding their overhead (no security checks are performed). Despite what intuition might suggest, it is possible to securely apply $fmap$ in any $MAC\ \ell$-computation to any labeled value *irrespectively of its security level*. A secure implementation of $fmap$ then allows manipulation of data without forking threads in a concurrent setting—thus, introducing flexibility when data is processed by pure (side-effect free) functions. However, obtaining a secure implementation of $fmap$ requires a careful analysis of its security implications.

Interestingly, the evaluation strategy of the programming language and the sequential or concurrent setting determine different security guarantees in the presence of $fmap$. Figure 9 shows our findings. In a sequential setting with *call-by-value* semantics, $fmap$ can be exploited to create a termination covert channel in a similar manner as it is done with $join^{\mathsf{MAC}}$. To illustrate this point, we rephrase the attack in Figure 6 to use $fmap$ rather than $join^{\mathsf{MAC}}$—see Figure 8. Under a *call-by-value* evaluation strategy, function $loopOn$ passed to $fmap$ is eagerly applied to the secret, which might introduce a loop depending on the value of the n-th bit of the secret—a termination leak. Under a *call-by-name* evaluation strategy, however, function $loopOn$ does not get immediately evaluated since $result$ is not needed for computing $publish\ n$. Therefore, $publish\ n$ gets executed independently of the value of the secret, i.e. no termination leaks are introduced. Instead, $loopOn$ gets evaluated when "unlabeling" $result$ and inspecting its value in a computation of type $MAC\ H\ a$ (for some a), which is secure to do so. Although *functors* can be used to exploit non-termination of programs, they impose no new risks for sequential programs (**MAC** already ignores termination leaks in such setting).

	Sequential	Concurrent
call-by-value	**PINI**	✗
call-by-name	**PINI**	**PSNI**

Fig. 9: Security guarantees

Unfortunately, a call-by-value concurrent semantics magnifies the bandwidth of the attack in Figure 8 to the point where confidentiality can be systematically and efficiently broken—see Figure 10. Assuming a secret of 100-bits, the magnification consists on leaking the whole secret by spawning a sufficient number of threads—each of them leaking a different bit. Since $leak$ cannot exploit the termination chan-

```
magnify :: Labeled H Secret → MAC L ()
magnify secret =
  for [0..99]
      (λn → fork^MAC (leak n secret))
```

Fig. 10: Attack magnification

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

Fig. 8: Termination leak under call-by-value evaluation

nel under a *call-by-name* evaluation strategy, the magnification attack becomes vacuous under such semantics. More precisely, the attack can only trigger the execution of $leak$ by first unlabeling $result$, an operation impossible to perform in a public computation—recall there is no $join^{\mathsf{MAC}}$ primitive for concurrent programs. As the table suggests, call-by-name gives the strongest security guarantees when extending **MAC** with functors. We remark that it is possible to close this termination channel under a call-by-value semantics by defining $Labeled$ with an explicit suspension, e.g. $\mathbf{data}\ Labeled\ \ell\ a = Labeled\ (() \to a)$, and corresponding forcing operation, so that $fmap$ behaves lazily as desired.

Example 2. Alice's realizes that she could spare her API from forking threads by exploiting the *functorial* structure of labeled values.

```
declareTaxes :: Labeled H URL → Labeled H Data → MAC L ()
declareTaxes url user = send^MAC url (fmap fillTaxes user)
  -- Internal operations
fillTaxes :: Data → TaxDecl
```

The construct $fmap$ applies the function $fillTaxes$ without requiring use of $unlabel$, while keeping the result securely encapsulated in a labeled value. Observe how the code is much less imperative, since there is no need to fork a thread to unlabel sensitive data just to apply a pure function to it.

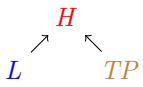

Fig. 11: Lattice.

While functors help to make the code more functional, there are still other programming patterns which draw developers to fork threads due to security reasons rather than the need for multi-threading. Specifically, when aggregating data from sources with incomparable labels, computations are forced to spawn a thread with a sufficiently high label. To illustrate this point, we present the following example.

Example 3. Alice knows that there is a third-party API which provides financial planning and she would gladly incorporate its functionality into her API. However, Alice wants to keep the third-party code isolated from hers, while still providing functionality to the user. To do so, she incorporates a new label into the system, namely TP and modifies the lattice as shown in Figure 11. The lattice reflects the mistrust that Alice has over the third-party code by making L and TP incomparable elements.

Alice's API is extended with the third-party code as follows.

```
declareTaxes :: Labeled H URL → Labeled H Data → MAC L ()
reportPlan   :: Labeled H URL → Labeled H Data → MAC L ()
  -- Internal operations
fillTaxes :: Data → TaxDecl
financialPlan :: Labeled TP (Data → FinancePlan)
```

Function $reportPlan$ needs to fork a thread in order to unlabel the third-party code ($financialPlan$).

```
reportPlan :: Labeled H URL → Labeled H Data → MAC L ()
reportPlan url user = do
  fork^MAC (do user          ← unlabel user
               financialPlan′ ← unlabel financialPlan
               plan          ← label (financialPlan′ user)
               send^MAC url plan)
```

In the next section, we show how to avoid forking threads for this kind of scenarios.

4 Applicative Operator and Relabeling

To aggregate sensitivity-heterogeneous data without forking, we further extend the API with the primitives shown in Figure 12. Primitive *relabel* copies, and possibly upgrades, a labeled value. This primitive is useful to "lift" data to an upper bound of all the data involved in a computation prior to combining them. Operator $(\langle * \rangle)$ supports function application within a labeled value, i.e. it allows to feed functions wrapped in a labeled value ($Labeled\ \ell\ (a \to b)$) with arguments also wrapped ($Labeled\ \ell\ a$), where aggregated results get wrapped as well ($Labeled\ \ell\ b$). We demonstrate the utility of *relabel* and $(\langle * \rangle)$ by rewriting Example 3.

$$
\begin{aligned}
&relabel :: \ell_L \sqsubseteq \ell_H \Rightarrow \\
&\qquad Labeled\ \ell_L\ a \to Labeled\ \ell_H\ a \\
&(\langle * \rangle) \quad :: Labeled\ \ell\ (a \to b) \to \\
&\qquad Labeled\ \ell\ a \to Labeled\ \ell\ b
\end{aligned}
$$

Fig. 12: Extended API for labeled values

Example 4. Alice easily modifies *reportPlan* as follows:

```
reportPlan url user = do
  let financialPlan′ = relabel financialPlan
  in  send^MAC url (financialPlan′ ⟨*⟩ user)
```

The third-party function (*financialPlan*) is relabeled to H, which is justified since $TP \sqsubseteq H$, and then applied to the user data ($financialPlan'\ \langle * \rangle\ user$) using the applicative (functor) operator. Note that the result is still labeled with H .

Discussion In function programming, operator $(\langle * \rangle)$ is part of the *applicative functors* [20] interface, which in combinations with *fmap*, is used to map functions over functors. Note that if labeled values fully enjoyed the applicative functor structure, our API would include also the primitive $pure :: a \to Labeled\ \ell\ a$. This primitive brings arbitrary values into labeled values, which might break the security principles enforced by **MAC**. Instead of *pure*, **MAC** centralizes the creation of labeled values in the primitive *label*. Observe that, by using *pure*, a programmer could write a computation $m :: MAC\ H\ (Labeled\ L\ a)$ where the *created* labeled information is sensitive rather than public. We argue that this situation ignores the no-write down principle, which might bring confusion among developers. More importantly, freely creating labeled values is not compatible with the security notion of *cleareance*, where secure computations have an upper bound on the kind of sensitive data the they can observe

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

Fig. 13: Formal syntax for types, values, and terms.

and generate. This notion becomes useful to address certain covert channels [40] as well as poison-pill attacks [13]. While **MAC** does not yet currently support cleareance, we state this research direction as future work.

5 Security guarantees

This section presents the core part of our formalization of **MAC** as a simply typed call-by-name λ-calculus extended with booleans, unit values, and monadic operations. Note that our mechanized proofs, available online[4], cover the full calculus which also includes references, synchronization variables, and exceptions. Given the number of advanced features in the calculus we remark that a proof assistant has proved to be an invaluable tool to verify the correctness of our proofs. Figure 13 shows the formal syntax. Meta variables τ, v and t denote types, values, and terms, respectively. Most of these syntactic categories are self-explanatory with the exception of a few cases that we proceed to clarify. We note that, even though labels are actual types in **MAC**, we use a separate syntactic category ℓ for clarity in this calculus. Furthermore, we assume that labels form a lattice $(\mathscr{L}, \sqsubseteq)$. Constructors MAC and Res represent a secure computation and a labeled resource, respectively. The latter is an established technique to lift arbitrary resources such as references and synchronization variables into **MAC** [31]. MAC and Res are **MAC**'s internals constructors, therefore they are not available to users of the library and are not part of the surface syntax. Data type $\mathit{Id}\ \tau$ denotes an expression of type τ and $\mathit{Res}\ (\mathit{Id}\ t)$ represents a labeled expression t, which we abbreviate as $\mathit{Labeled}\ t$. Similarly we write $\mathit{Labeled}\ \ell\ \tau$ for the type $\mathit{Res}\ \ell\ (\mathit{Id}\ \tau)$. Node $\langle * \rangle$ corresponds to the applicative (functor) operator and is overloaded for $\mathit{Labeled}\ \ell\ t$ and $\mathit{Id}\ \tau$. Every applicative functor is also a functor [20], hence $\mathit{fmap}\ f\ x$ is simply defined as $(\mathit{Labeled}\ f)\ \langle * \rangle\ x$. The special syntax nodes $\bullet$, $\langle * \rangle_\bullet$, and $\mathit{relabel}_\bullet$ represent *erased terms* 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. $\Gamma \vdash \bullet : \tau$.

Semantics The small-step semantics of the calculus is represented by the relation $t_1 \rightsquigarrow t_2$, which denotes that t_1 reduces to t_2 in one step. Most of the the rules are standard

[4] `https://bitbucket.org/MarcoVassena/mac-agda`

$$\text{(HOLE)} \quad \bullet \rightsquigarrow \bullet$$

$$\text{(LABELED}\langle * \rangle\text{)} \quad (Res\ t_1)\ \langle * \rangle\ (Res\ t_2) \rightsquigarrow Res\ (t_1\ \langle * \rangle\ t_2)$$

$$\text{(ID}\ \langle * \rangle\text{)} \quad Id\ (\lambda x.t_1)\ \langle * \rangle\ Id\ t_2 \rightsquigarrow Id\ ([x\ /\ t_2]\ t_1)$$

$$\text{(RELABEL)} \quad relabel\ (Res\ t) \rightsquigarrow Res\ t$$

$$\text{(UNLABEL)} \quad unlabel\ (Res\ (Id\ t)) \rightsquigarrow return\ t$$

Fig. 14: Semantics for non-standard constructs.

and hence omitted; the rules for interesting constructs are shown in Figure 14. Term $\bullet$ merely reduces to itself according to rule [HOLE]. Rule [LABELED$\langle * \rangle$] describes the semantics of operator $\langle * \rangle$, which applies a labeled function to a labeled value. Terms t_1 and t_2 are wrapped in Id so they cannot be combined by plain function application. As rule [ID$\langle * \rangle$] shows, Id is also an applicative operator and therefore $\langle * \rangle$ is used instead. Observe that symbol $\langle * \rangle$ is overloaded, where the type of its argument determines which rule to apply, i.e. either [LABELED$\langle * \rangle$] or [ID$\langle * \rangle$]. Rule [ID$\langle * \rangle$] requires a function to be in weak-head normal form ($(\lambda x.t_1)\ t_2$) where *beta reduction* occurs right away. (As usual, we write $[t_1\ /\ x]\ t_2$ for the capture-avoiding substitution of every occurrence of x with t_1 in t_2). This manner to write the rule is unusual since it would be expected that $Id\ f\ \langle * \rangle\ Id\ t \rightsquigarrow Id\ (f\ t)$. Nevertheless, the eagerness of $\langle * \rangle$ in its first argument is needed for technical reasons in order to guarantee non-interference. Rule [RELABEL] upgrades the label of a labeled value. Since relabeling occurs at the level of types, the reduction rules simply create another labeled term. Finally rule [UNLABEL] extracts the labeled value and returns it in a computation at the appropriate security level. We omit the two context rules that first reduce the labeled value to weak-head normal form and then the expression itself.

5.1 Sequential calculus

In this section, we prove progress-insensitive non-interference for our calculus. Similar to other work [19, 30, 37], we employ the *term erasure* proof technique. To that end, we introduce an erasure function which rewrites sensitive information, i.e. data above the security level of the attacker, to term $\bullet$. Since security levels are at the type-level, the erasure function is type-driven. We write $\varepsilon^{\tau}_{\ell_A}(t)$ for the erasure of term t with type τ of data above the security of the attacker ℓ_A. We omit the type superscript when it is either irrelevant or clear from the context. Figure 15 highlights the intuition behind the used proof technique: *showing that the drawn diagram commutes*. More precisely, we show that erasing sensitive data from a term t and then taking a step (lower part of the diagram) is the same as firstly taking a step (upper part of the diagram) and then erasing sensitive data. If term t leaks

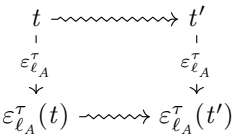

Fig. 15: Commutative diagram

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

$$\varepsilon_{\ell_A}^{Labeled\ \ell\ \tau}(t_1\ \langle * \rangle\ t_2) = \begin{cases} \varepsilon_{\ell_A}(t_1)\ \langle * \rangle\ \varepsilon_{\ell_A}(t_2) & \text{if } \ell \sqsubseteq \ell_A \\ \varepsilon_{\ell_A}(t_1)\ \langle * \rangle_\bullet\ \varepsilon_{\ell_A}(t_2) & \text{otherwise} \end{cases}$$

$$\varepsilon_{\ell_A}^{Labeled\ \ell\ \tau}(t_1\ \langle * \rangle_\bullet\ t_2) = \varepsilon_{\ell_A}(t_1)\ \langle * \rangle_\bullet\ \varepsilon_{\ell_A}(t_2)$$

$$\varepsilon_{\ell_A}^{Labeled\ \ell\ \tau}(relabel\ t) = \begin{cases} relabel\ \varepsilon_{\ell_A}(t) & \text{if } \ell \sqsubseteq \ell_A \\ relabel_\bullet\ \varepsilon_{\ell_A}(t) & \text{otherwise} \end{cases}$$

$$\varepsilon_{\ell_A}^{Labeled\ \ell\ \tau}(relabel_\bullet\ t) = relabel_\bullet\ \varepsilon_{\ell_A}(t) \qquad \varepsilon_{\ell_A}^{MAC\ \ell\ \tau}(t) = \bullet, \text{ if } \ell \not\sqsubseteq \ell_A$$

Fig. 16: Erasure function.

$$\text{(LABELED}\langle * \rangle_\bullet\text{)}\quad (Res\ t_1)\ \langle * \rangle_\bullet\ (Res\ t_2) \rightsquigarrow Res\ \bullet \qquad \text{(RELABEL}_\bullet\text{)}\quad relabel_\bullet\ (Res\ t) \rightsquigarrow Res\ \bullet$$

Fig. 17: Reduction rules for $\langle * \rangle_\bullet$ and $relabel_\bullet$.

data which sensitivity label is above ℓ_A, then erasing all sensitive data and 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 $\varepsilon^{\tau}_{\ell_A}(t')$.

Figure 16 shows the definition of the erasure functions for the interesting cases. Before explaining them, we remark that ground values (e.g., $True$) are unaffected by the erasure function and that, for most of the terms, the function is homomorphically applied, e.g., $\varepsilon^{()}_{\ell}(\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3) = \textbf{if } \varepsilon^{Bool}_{\ell}(t_1) \textbf{ then } \varepsilon^{()}_{\ell}(t_2) \textbf{ else } \varepsilon^{()}_{\ell}(t_3)$. Labeled resources are erased according to the label found in their type ($Res\ \ell\ \tau$). If the attacker can observe the term ($\ell \sqsubseteq \ell_A$), the erasure function is homomorphically applied; otherwise, it is replaced with $\bullet$. In principle, one might be tempted to apply the erasure function homomorphically for $\langle * \rangle$ and $relabel$, but such approach unfortunately breaks the commutativity of Figure 15. To illustrate this point, consider the term $(Res\ f)\ \langle * \rangle\ (Res\ x)$ of type $Labeled\ H\ Int$, which reduces to $Res\ (f\ \langle * \rangle\ x)$ according to rule [LABELED$\langle * \rangle$]. By applying the erasure function homomorphically, we get $\varepsilon_L(Res\ f)\ \langle * \rangle\ \varepsilon_L(Res\ x)$, that is $(Res\ \bullet)\ \langle * \rangle\ (Res\ \bullet)$ which reduces to $Res\ (\bullet\ \langle * \rangle\ \bullet) \not\equiv Res\ \bullet$. Operator $relabel$ raises a similar problem. Consider for example the term $relabel\ (Labeled\ 42) :: Labeled\ H\ Int$, where $Labeled\ 42 :: Labeled\ L\ Int$. If the erasure function were applied homomorphically, i.e. consider $relabel\ \varepsilon_L^{Labeled\ L\ Int}(Labeled\ 42)$, it means that *sensitive data produced by* $relabel$ *remains after erasure*—thus, breaking commutativity. Instead, we perform erasure in *two-steps*—a novel technique if compared with previous papers (e.g., [37]). Rather than being a pure syntactic procedure, erasure is also performed by additional evaluation rules, triggered by special constructs introduced by the erasure function. Specifically,

the erasure function replaces $\langle * \rangle$ with $\langle * \rangle_\bullet$ and erasure is then performed by means of rule [LABELED$\langle * \rangle_\bullet$]—see Figure 17. Following the same scheme, the erasure function replaces $relabel$ with $relabel_\bullet$ and rule [RELABEL$_\bullet$] performs the erasure. $\langle * \rangle_\bullet$ and $relabel_\bullet$ and their semantics rules are introduced due to mere technical reasons (as explained above) and they do not impact the performance of **MAC** since they are not part of its implementation. Finally, *terms* of type $MAC\ \ell\ \tau$ are replaced by $\bullet$ when the computation is more sensitive than the attacker level ($\ell\ \not\sqsubseteq\ \ell_A$); otherwise, the erasure function is homomorphically applied.

Progress-insensitive non-interference The non-interference proof relies on two fundamental properties of our calclulus: *determinism* and *distributivity*.

Proposition 1 (Sequential determinancy and distributivity)

- *If* $t_1 \rightsquigarrow t_2$ *and* $t_1 \rightsquigarrow t_3$ *then* $t_2 = t_3$.
- *If* $t_1 \rightsquigarrow t_2$ *then* $\varepsilon_{\ell_A}(t_1) \rightsquigarrow \varepsilon_{\ell_A}(t_2)$.

In Proposition 1, we show the auxiliary property that erasure distributes over substitution, i.e. $\varepsilon_{\ell_A}([x\ /\ t_1]\ t_2) = [x\ /\ \varepsilon_{\ell_A}(t_1)]\ \varepsilon_{\ell_A}(t_2)$. Note, however, that the erasure function does not always distribute over function application, i.e. $\varepsilon^{\tau}_{\ell_A}(t_1\ \ t_2) \not\equiv \varepsilon_{\ell_A}(t_1)\ \varepsilon_{\ell_A}(t_2)$ when $\tau = MAC\ h\ \tau'$ and $h\ \not\sqsubseteq\ \ell_A$. It is precisely for this reason that rule [ID$\langle * \rangle$] performs substitution rather than function application. Before stating non-interference, we formally define ℓ_A-equivalence.

Definition 1. *(ℓ_A-equivalence) Two terms are indistinguishable from an attacker at security level ℓ_A, written $x \approx_{\ell_A} y$, if and only if $\varepsilon_{\ell_A}(x) = \varepsilon_{\ell_A}(y)$.*

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

Proposition 2 (ℓ_A-equivalence preservation) *If $t_1 \approx_{\ell_A} t_2$, $t_1 \rightsquigarrow t_1'$, and $t_2 \rightsquigarrow t_2'$, then $t_1' \approx_{\ell_A} t_2'$.*

We finally prove progress-insensitive non-interference for the sequential calculus. We employ *big-step* semantics, denoted by $t\ \Downarrow\ v$, which reduces term t to value v in a finite number of steps.

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

5.2 Concurrent calculus

Figure 18 extends the calculus from Section 5 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. Threads are secure computations of type $MAC\ \ell\ ()$ which get organized in isolated thread pools according to their security label. A pool t_s in the category

Scheduler state:	s
Thread pool :	$\Phi ::= (\ell : Label) \rightarrow (Pool\ \ell)$
Pool ℓ:	$t_s ::= [\,] \mid t : t_s \mid \bullet$
Configuration:	$c ::= \langle s, \Phi \rangle$

Fig. 18: Syntax for concurrent calclulus.

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 a 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 $\Phi(\ell)[n \mapsto t]$. Reading from an erased thread pool results in an erased thread, i.e. $\bullet[n] = \bullet$ and updating it has no effect, i.e. $\bullet[n \mapsto t] = \bullet$.

Semantics The relation $\hookrightarrow_{(\ell,n)}$ represents an evaluation step for global configurations, where the thread identified by (ℓ, n) gets scheduled. Figure 19 shows the scheme rule for $\hookrightarrow_{(\ell,n)}$. The scheduled thread is retrieved from the configuration ($\Phi[\ell][n] = t_1$) and executed ($t_1 \rightsquigarrow_e t_2$). We decorate the sequential semantics with events e, which provides to the scheduler information about the effects produced by the scheduled instruction, for example $\bullet \rightsquigarrow_\bullet \bullet$. Events inform the scheduler about the evolution of the global configuration, so that it can realize concrete scheduling policies. 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) and updates its state according to the event e. Lastly, the thread pool is updated with the final state of the thread ($\Phi[\ell][n] := t_2$).

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

Fig. 19: Scheme rule for concurrent semantics.

Progress-sensitive non-interference Our concurrent calculus satisfies progress sensitive non-interference—a security condition often enforced by IFC techniques for π-calculus [12, 27]. A *global configurations* is erased by erasing its components, that is $\varepsilon_{\ell_A}(\langle s, \Phi\rangle) = \langle \varepsilon_{\ell_A}(s), \varepsilon_{\ell_A}(\Phi)\rangle$. The thread pool Φ is erased point-wise, pools are either completely collapsed if not visible from the attacker, i.e. $\varepsilon^{Pool\ \ell}_{\ell_A}(t_s) = \bullet$ if $\ell \not\sqsubseteq \ell_A$, or the erasure function is homomorphically applied to their content. The erasure of the scheduler state s is scheduler specific. To obtain a parametric proof of non-interference, we assume certain properties about the scheduler. Specifically, our proof is valid for deterministic schedulers which fulfill progress and non-interference themselves, i.e. schedulers cannot leverage sensitive information in threads to determine what to schedule next —Appendix A formalizes the requirements for such *suitable* schedulers. As for the sequential calculus, we rely on determinancy and distributivity of the concurrent semantics.

Proposition 3 (Concurrent determinancy and distributivity)

- *If* $c_1 \hookrightarrow_{(\ell,n)} c_2$ *and* $c_1 \hookrightarrow_{(\ell,n)} c_3$*, then* $c_2 = c_3$.
- *If* $c_1 \hookrightarrow_{(\ell,n,e)} c_2$*, then it holds that* $\varepsilon_{\ell_A}(c_1) \hookrightarrow_{(\ell,n,\varepsilon_{\ell_A}(e))} \varepsilon_{\ell_A}(c_2)$.

In the non-interference theorem, we write as usual $\hookrightarrow^\star$for the reflexive transitive closure of $\hookrightarrow$ and we generalize $\approx_{\ell_A}$ to denote ℓ_A-equivalence between configurations.

Theorem 2 (Progress-sensitive non-interference) *Given the global configurations* c_1*,* c_1'*,* c_2*, and assuming a deterministic and non-interfering scheduler that makes progress, 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* $c_2 \approx_{\ell_A} c_2'$.

6 Related work

Security Libraries Li and Zdancewic's seminal work [18] shows how the structure *arrows* can provide IFC as a library in Haskell. Tsai et al. [39] extend that work to support concurrency and data with heterogeneous labels. Russo et al. [30] implement the security library **SecLib** using a simpler structure than arrows, i.e. monads—rather than labeled values, this work introduces a monad which statically label side-effect free values. The security library **LIO** [36,37] dynamically enforces IFC for both sequential and concurrent settings. **LIO** presents operations similar to $fmap$ and $\langle * \rangle$ for labeled values with differences in the returning type due to **LIO**'s checks for clearence—this work provides a foundation to analyze the security implications of such primitives. Mechanized proofs for **LIO** are given only for its core sequential calculus [37]. Inspired by **SecLib** and **LIO**'s designs, **MAC** leverages Haskell's type system to enforce IFC [31]—this work does not contain formal guarantees and relies on its simplicity to convince the reader about its correctness. **HLIO** uses advanced Haskell's type-system features to provide a hybrid approach: IFC is statically enforce while allowing the programmers to defer selected security checks to be done at runtime [6]. Our work studies the security implications of extending **LIO**, **MAC**, and **HLIO** with a rich structure for labeled values. Devriese and Piessens provide a monad transformer to extend imperative-like APIs with support for IFC in Haskell [8]. Jaskelioff and Russo implements a library which dynamically enforces IFC using secure multi-execution (SME) [15]—a technique that runs programs multiple times. Rather than running multiple copies of a program, Schmitz et al. [33] provide a library with *faceted values*, where values present different behavior according to the privilege of the observer. Different from the work above, we present a fully-fledged mechanized proof for our sequential and concurrent calculus which includes references, synchronization variables, and exceptions.

IFC tools IFC research has produced compilers capable of preserving confidentiality of data: Jif [25] and Paragon [4] (based on Java), and FlowCaml [34] (based on Caml). The SPARK language presents a IFC analysis which has been extended to guarantee progress-sensitive non-inference [28]. JSFlow [11] is one of the state-of-the-art IFC system for the web (based on JavaScript). These tools preserve confidentiality in a fine-grained fashion where every piece of data is explicitly label. Specifically, there is no abstract data type to label data, so our results cannot directly apply to them.

Operating systems research MAC systems [3] assign a label with an entire OS process—settling a single policy for all the data handled by it. While proposed in the 70s, there are modern manifestations of this idea (e.g., [17, 23, 40]) applied to diverse scenarios like the web (e.g., [2, 38]) and mobile devices (e.g., [5, 16]). In principle, it would be possible to extend such MAC-like systems to include a notion of labeled values with the functor structure as well as the relabeling primitive proposed by this work. For instance, COWL [38] presents the notion of *labeled blob* and *labeled XHR* which is isomorphic to the notion of labeled values, thus making possible to apply our results. Furthermore, because many MAC-like system often ignore termination leaks (e.g., [9, 40]), there is no need to use call-by-name evaluation to obtain security guarantees.

7 Conclusions

We present an extension of **MAC** that provides labeled values with an applicative functor-like structure and a relabeling operation, enabling convenient and expressive manipulation of labeled values using side effect-free code and saving programmers from introducing unnecessary sub-computations (e.g., in the form of threads). We have proved this extension secure both in sequential and concurrent settings, exposing an interesting connection between evaluation strategy and progress-sensitive noninterference. This work bridges the gap between existing IFC libraries (which focus on side-effecting code) and the usual Haskell programming model (which favors pure code), with a view to making IFC in Haskell more practical.

Acknowledgement This work was supported in part by the Swedish research agencies VR and STINT, The Sloan Foundation, and by NSF grant 1421770.

References

1. Askarov, A., Hunt, S., Sabelfeld, A., Sands, D.: Termination-insensitive noninterference leaks more than just a bit. In: European Symposium on Research in Computer Security. Springer-Verlag (2008)
2. Bauer, L., Cai, S., Jia, L., Passaro, T., Stroucken, M., Tian, Y.: Run-time monitoring and formal analysis of information flows in Chromium. In: Annual Network & Distributed System Security Symposium. Internet Society (2015)
3. Bell, D.E., La Padula, L.: Secure computer system: Unified exposition and multics interpretation. Tech. Rep. MTR-2997, Rev. 1, MITRE Corporation, Bedford, MA (1976)
4. Broberg, N., van Delft, B., Sands, D.: Paragon for practical programming with information-flow control. In: APLAS. LNCS, vol. 8301, pp. 217–232. Springer (2013)
5. Bugiel, S., Heuser, S., Sadeghi, A.R.: Flexible and Fine-grained Mandatory Access Control on Android for Diverse Security and Privacy Policies. In: USENIX Conference on Security. SEC, USENIX Association (2013)
6. Buiras, P., Vytiniotis, D., Russo, A.: HLIO: Mixing static and dynamic typing for information-flow control in Haskell. In: ACM SIGPLAN International Conference on Functional Programming. ACM (2015)
7. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Communications of the ACM 20(7), 504–513 (1977)
8. Devriese, D., Piessens, F.: Information flow enforcement in monadic libraries. In: ACM SIGPLAN Workshop on Types in Language Design and Implementation. ACM (2011)
9. Efstathopoulos, P., Krohn, M., VanDeBogart, S., Frey, C., Ziegler, D., Kohler, E., Mazières, D., Kaashoek, F., Morris, R.: Labels and event processes in the asbestos operating system. In: ACM Symposium on Operating Systems Principles. SOSP, ACM (2005)
10. Goguen, J., Meseguer, J.: Security policies and security models. In: IEEE Symposium on Security and Privacy. IEEE Computer Society (1982)
11. Hedin, D., Birgisson, A., Bello, L., Sabelfeld, A.: JSFlow: Tracking information flow in JavaScript and its APIs. In: ACM Symposium on Applied Computing. ACM (2014)
12. Honda, K., Vasconcelos, V.T., Yoshida, N.: Secure Information Flow as Typed Process Behaviour. In: European Symposium on Programming Languages and Systems. Springer-Verlag (2000)

13. Hritcu, C., Greenberg, M., Karel, B., Peirce, B.C., Morrisett, G.: All your IFCexception are belong to us. In: IEEE Symposium on Security and Privacy. IEEE Computer Society (2013)
14. Hughes, J.: Why functional programming matters. The Computer Journal 32 (1984)
15. Jaskelioff, M., Russo, A.: Secure multi-execution in Haskell. In: PSI Ershov Informatics Conference. LNCS, Springer-Verlag (2011)
16. Jia, L., Aljuraidan, J., Fragkaki, E., Bauer, L., Stroucken, M., Fukushima, K., Kiyomoto, S., Miyake, Y.: Run-time enforcement of information-flow properties on Android (extended abstract). In: European Symposium on Research in Computer Security. Springer (2013)
17. Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M.F., Kohler, E., Morris, R.: Information flow control for standard OS abstractions. In: ACM SIGOPS Symposium on Operating Systems Principles. SOSP, ACM (2007)
18. Li, P., Zdancewic, S.: Encoding information flow in Haskell. In: IEEE Workshop on Computer Security Foundations. IEEE Computer Society (2006)
19. Li, P., Zdancewic, S.: Arrows for secure information flow. Theoretical Computer Science 411(19), 1974–1994 (2010)
20. Mcbride, C., Paterson, R.: Applicative programming with effects. Journal of Functional Programming (2008)
21. Meurer, S., Wismüller, R.: Apefs: An infrastructure for permission-based filtering of android apps. In: Schmidt, A., Russello, G., Krontiris, I., Lian, S. (eds.) Security and Privacy in Mobile Information and Communication Systems. vol. 107. Springer Berlin Heidelberg (2012)
22. Moggi, E.: Notions of computation and monads. Information and Computation 93(1), 55–92 (1991)
23. Murray, T., Matichuk, D., Brassil, M., Gammie, P., Bourke, T., Seefried, S., Lewis, C., Gao, X., Klein, G.: sel4: From general purpose to a proof of information flow enforcement. 2012 IEEE Symposium on Security and Privacy 0 (2013)
24. Myers, A.C.: JFlow: Practical mostly-static information flow control. In: ACM Symp. on Principles of Programming Languages. pp. 228–241 (1999)
25. Myers, A.C., Zheng, L., Zdancewic, S., Chong, S., Nystrom, N.: Jif: Java Information Flow (2001), `http://www.cs.cornell.edu/jif`
26. Pottier, F., Simonet, V.: Information Flow Inference for ML. In: ACM Symp. on Principles of Programming Languages. pp. 319–330 (2002)
27. Pottier, F.: A Simple View of Type-Secure Information Flow in the π-Calculus. In: IEEE Computer Security Foundations Workshop. pp. 320–330 (2002)
28. Rafnsson, W., Garg, D., Sabelfeld, A.: Progress-sensitive security for SPARK. In: Engineering Secure Software and Systems, London, UK (2016)
29. Roy, I., Porter, D.E., Bond, M.D., McKinley, K.S., Witchel, E.: Laminar: Practical fine-grained decentralized information flow control. In: ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI, ACM (2009)
30. Russo, A., Claessen, K., Hughes, J.: A library for light-weight information-flow security in Haskell. In: ACM SIGPLAN symposium on Haskell. ACM (2008)
31. Russo, A.: Functional Pearl: Two Can Keep a Secret, if One of Them Uses Haskell. In: ACM SIGPLAN International Conference on Functional Programming. ICFP, ACM (2015)
32. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications 21(1), 5–19 (2003)
33. Schmitz, T., Rhodes, D., Austin, T.H., Knowles, K., Flanagan, C.: Faceted dynamic information flow via control and data monads. In: Piessens, F., Viganò, L. (eds.) POST. LNCS, vol. 9635. Springer (2016)
34. Simonet, V.: The Flow Caml system (2003), software release at `http://cristal.inria.fr/~simonet/soft/flowcaml/`
35. Smith, G., Volpano, D.: Secure information flow in a multi-threaded imperative language. In: ACM symposium on Principles of Programming Languages (1998)

36. Stefan, D., Russo, A., Buiras, P., Levy, A., Mitchell, J.C., Maziéres, D.: Addressing covert termination and timing channels in concurrent information flow systems. In: ACM SIGPLAN International Conference on Functional Programming. ACM (2012)
37. Stefan, D., Russo, A., Mitchell, J.C., Mazières, D.: Flexible dynamic information flow control in Haskell. In: ACM SIGPLAN Haskell symposium (2011)
38. Stefan, D., Yang, E.Z., Marchenko, P., Russo, A., Herman, D., Karp, B., Mazières, D.: Protecting users by confining JavaScript with COWL. In: USENIX Symposium on Operating Systems Design and Implementation. USENIX Association (2014)
39. Tsai, T.C., Russo, A., Hughes, J.: A library for secure multi-threaded information flow in Haskell. In: IEEE Computer Security Foundations Symposium (2007)
40. Zeldovich, N., Boyd-Wickizer, S., Kohler, E., Mazières, D.: Making information flow explicit in HiStar. In: USENIX Symp. on Operating Systems Design and Implementation. USENIX (2006)

A Scheduler Requirements

In this section we formalize the requirements that a scheduler must fulfill in order to guarantee non-interference of the concurrent calculus. The proof of determinancy of the concurrent calculus relies on determinancy of the scheduler.

Requirement 1 (Scheduler Determinancy) *If* $s_1 \xrightarrow{(\ell,n,e)} s_2$ *and* $s_1 \xrightarrow{(\ell,n,e)} s_3$ *then* $s_2 = s_3$.

Similarly distributivity of the concurrent calculus requires distributivity of the scheduler.

Requirement 2 (Scheduler Distributivity) *If* $s_1 \xrightarrow{(\ell,n,e)} s_2$ *then* $\varepsilon_{\ell_A}(s_1) \xrightarrow{(\ell,n,\varepsilon_{\ell_A}(e))} \varepsilon_{\ell_A}(s_2)$.

In this case events share the same sensitivity of the thread that generated them, so that $\varepsilon_{\ell_A}(e) = \bullet$ if $\ell \not\sqsubseteq \ell_A$.

Requirement 3 (Scheduler Progress) *Every alive thread will eventually be scheduled.*

In order to prove non-interference of the concurrent calculus we expect the scheduler to be non-interfering. Our notion of non-interference for schedulers is non-standard due to the mutual dependency between scheduler and concurrent semantics. In fact in a step $s_1 \xrightarrow{(\ell,n,e)} s_2$, the scheduler decides which thread to run, i.e. (ℓ, n), but the concurrent semantics determines the event e. We then encode scheduler non-interference as follows:

Requirement 4 (Non-Interfering Scheduler) *If* $s_1 \xrightarrow{(\ell,n,e)} s_2$, $s_1 \approx_{\ell_A} s_1'$ *and* $\ell \sqsubseteq \ell_A$ *then one of the following holds:*

1. $\exists\, s_2' \,.\quad s_1' \xrightarrow{(\ell,n,e)} s_2' \quad\wedge\quad s_2 \approx_{\ell_A} s_2'$
2. $\exists\, h, m \,.\, \forall\, e' \,.\, \exists\, s_2' \,.\quad s_1' \xrightarrow{(h,m,e')} s_2' \quad\wedge\quad h \not\sqsubseteq \ell \quad\wedge\quad s_1' \approx_{\ell_A} s_2'$

A.1 Round-robin Scheduler

In this section we formalize a round-robin scheduler that satisfies our requirements. The state s of the scheduler is a queue containing the identifiers of alive threads. Figure 20 shows the semantics rule. In rule [STEP] the scheduled thread performs a reduction step (event $Step$), hence it is enqueued. When a thread has terminated, denoted by event $Done$, the corresponding identifier is dropped from the list ([DONE]). In rule [SKIP] the scheduled thread is stuck, i.e. it cannot perform any step, therefore it skips its turn and is enqueued. In rule [FORK] the scheduled thread spawns a new thread (identified by (h, m)). Note that both threads are enqueued in order to guarantee *progress*. Lastly rule [HOLE] accounts for events generated by erased threads, hence the event is $\bullet$. The erasure function of the scheduler state filters out the identifiers of threads non visible to the attacker, that is $\varepsilon_{\ell_A}(s) = \mathit{filter}\ (\lambda(\ell, n) \to \ell\ \sqsubseteq\ \ell_A)\ s$.

$$\text{(STEP)}\quad ((\ell, n) : s) \xrightarrow{(\ell, n, Step)} s +\!\!+ [(\ell, n)] \qquad \text{(DONE)}\quad ((\ell, n) : s) \xrightarrow{(\ell, n, Done)} s$$

$$\text{(SKIP)}\quad ((\ell, n) : s) \xrightarrow{(\ell, n, NoStep)} s +\!\!+ [(\ell, n)] \qquad \text{(FORK)}\quad ((\ell, n) : s) \xrightarrow{(\ell, n, Fork\ h\ m)} s +\!\!+ [(h, m), (\ell, n)]$$

$$\text{(HOLE)}\quad s \xrightarrow{(\ell, n, \bullet)} s$$

Fig. 20: Semantics of a round-robin scheduler