# Securing Concurrent Lazy Programs Against Information Leakage

Marco Vassena
*Chalmers University of Technology*
*vassena@chalmers.se*

Joachim Breitner
*University of Pennsylvania*
*joachim@cis.upenn.edu*

Alejandro Russo
*Chalmers University of Technology*
*russo@chalmers.se*

***Abstract*—Many state-of-the-art information-flow control (IFC) tools are implemented as Haskell libraries. A distinctive feature of this language is lazy evaluation. In his influencal paper on why functional programming matters [1], John Hughes proclaims:**

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

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

## I. Introduction

Information-Flow Control [2] (IFC) scrutinizes source code to track how data of different sensitivity levels (e.g., public or sensitive) flows within a program, and raises alarms when confidentiality might be at stake. There are several special-purpose compilers and interpreters which apply this technology: *Jif* [3] (based on Java), *FlowCaml* [4] (based on Caml and not developed anymore), *Paragon* [5] (based on Java), and *JSFlow* [6] (based on JavaScript). Rather than writing compilers/interpreters, IFC can also be provided as a library in the functional programming language Haskell [7].

Haskell's type system enforces a disciplined separation of side-effect free from side-effectful code, which makes it possible to introduce input and output (I/O) to the language without compromising on its *purity*. Computations performing side-effects are encoded as values of abstract types which have the structure of monads [8]. This distinctive feature of Haskell is exploited by state-of-the-art IFC libraries (e.g., **LIO** [9] and **MAC** [10]) to identify and restrict "leaky" side-effects without requiring changes to the language or runtime.

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

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

Sabelfeld and Sands [11] suggest that lazy evaluation might be intrinsically safer than eager evaluation for leaks produced by termination—as lazy evaluation could skip the execution of *unneeded* non-terminating computations that might involve secrets. In multi-threaded systems, where termination leaks are harmful [12], a lazy evaluation strategy seems to be the appropriate choice.

Unfortunately, although lazy evaluation could "save the day" when it comes to termination leaks, it is also vulnerable to leaks via another covert channel due to sharing. Buiras and Russo [13] described an attack against the **LIO** library [12] where lazy evaluation is exploited to leak information via the *internal timing covert channel* [14]. This covert channel manifests by the mere presence of concurrency and shared resources. It gets exploited by setting up threads to race for a public shared resource in such a way that the secret value affects their timing and hence the winner of the race. **LIO** removes such leaks for public shared-resources which can be identified by the library (e.g., references). Due to lazy evaluation, variables introduced by **let**-bindings and function applications—which are beyond **LIO**'s control[1]—become shared resources and their evaluation affects the threads' timing behavior. Note that the *internal* timing channel leverages the *order* with which threads access the shared resource, not their execution time, which constitutes a different covert channel, known as the *external* timing covert channel [15], [16]. The attacker model for the external

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

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

Figure 1: Lazy evaluation attack

timing covert channel assumes that the attacker has access to an arbitrarily precise stopwatch to measure the wall-clock execution time of instructions and thereby deduce information about secrets. This paper does not address the external timing covert channel, which is a harder problem and for which mitigation techniques exist [17]–[19].

Figure 1 shows the lazy evaluation attack. In **LIO**, every thread has a current label which serves a role similar to the *program counter* in traditional IFC systems [20]. The first thread inspects a secret value ($s \leftarrow \mathit{unlabel\ secret}$), which sets the current label to secret. We refer to threads with such current label as *secret threads*. The other spawned threads have their current label set to public, therefore we call them *public threads*. Observe that the variable $r$ hosts an expression that is somewhat expensive to calculate, as it first builds a list with ten million numbers ($\ell = [1\mathinner{..}10000000]$) before summing up its elements ($r = \mathit{sum}\ \ell$). Importantly, the variable $r$ is referenced by both the secret and the public threads. Observe that every thread is secure in isolation—the secret thread always returns $()$ and the public threads read no secret. Assume that the expression $\mathit{no\_ops}$ is some irrelevant computation that takes slightly longer than half the time it takes to sum up the ten million numbers. Then the public threads race to send a message on a shared-public channel using the function $\mathit{sendPublicMsg}$:

▷ If $s \equiv 1$, then the secret thread has by now evaluated the expression referenced by $r$, in order to check if $r \geqslant 10$ holds. Due to sharing, the first public thread will not have to re-calculate $r$ and can output 0 almost imediately, while the other public thread is still occupied with $\mathit{no\_ops}$.

▷ If $s \equiv 0$, then the secret thread did not touch $r$. While the first public thread now has to evaluate $r$, the second public thread has enough time to perform $\mathit{no\_ops}$ and output 1 first.

As a result, the last message on the public channel reveals the secret $s$. This attack can be magnified to a point where whole secrets are leaked systematically and efficiently [12]. Similar to **LIO**, other state-of-the-art concurrent IFC Haskell libraries [10], [21] suffer from this attack.

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

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

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

► We present $\mathit{lazyDup}$, a primitive to restrict sharing in lazy languages with mutuable references.

► By injecting $\mathit{lazyDup}$ when spawning threads, we demonstrate that internal timing leaks via lazy evaluation are closed. The primitive $\mathit{lazyDup}$ is not only capable to secure **MAC** against lazy leaks, but also a wide range of other security Haskell libraries (e.g., **LIO** and **HLIO**).

► We prove that well-typed programs satisfy progress-sensitive non-interference (PSNI) for a wide-range of deterministic schedulers. However, for ease of exposition in this article, we focus only on a round-robin scheduler—the same scheduler used in GHC's runtime system[2]. Our non-interference claims are supported by mechanized proofs in the Agda proof assistant [22] and are parametric on the chosen (deterministic) scheduler.

► As a by-product of interest for the programming language community, we provide—to the best of our knowledge—the first operational semantics for lazy evaluation with mutable references.

This paper is organized as follows. Section II provides a brief overview on **MAC**. Section III describes our formalization for a concurrent non-strict calculus with sharing that also includes references. Primitive $\mathit{lazyDup}$ is described in Section IV. Section V shows how $\mathit{lazyDup}$ can remove leaks via lazy evaluation and Section VI provides the corresponding security guarantees. Related work is given in Section VII and Section VIII concludes.

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

```
  -- Abstract types
data Labeled ℓ τ
data MAC ℓ τ
  -- Monadic structure for computations
instance Monad (MAC ℓ)
  -- Core operations
label    :: ℓ_L ⊑ ℓ_H ⇒ τ → MAC ℓ_L (Labeled ℓ_H τ)
unlabel  :: ℓ_L ⊑ ℓ_H ⇒ Labeled ℓ_L τ → MAC ℓ_H τ
fork^MAC :: ℓ_L ⊑ ℓ_H ⇒ MAC ℓ_H () → MAC ℓ_L ()
```

Figure 2: Core API for **MAC**

## II. The **MAC** IFC Library

To set the stage of the work at hand, we briefly introduce the relevant aspects of the **MAC** IFC library [10].

*Security lattice:* The sensitivity of data is indicated by labels. These are partially ordered by $\sqsubseteq$ and form a security lattice [23]. Concretely, $\ell_1 \sqsubseteq \ell_2$ holds if data labeled with label $\ell_1$ is allowed to flow to entities labeled with $\ell_2$. Although **MAC** is parameterized on the security lattice, for simplicity we focus on the classic two-point lattice where the label $H$ denotes secret (high) data, the label $L$ denotes public (low) data, and $H \not\sqsubseteq L$ is the only disallowed flow. In **MAC**, each label is represented as an abstract data type. To improve readability, subscripts on label metavariables hint at their relationship, e.g., if $\ell_{\mathrm{L}}$ and $\ell_{\mathrm{H}}$ appear together, then $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$ holds.

*Security Types:* Figure 2 shows the core of **MAC**'s API. The abstract type $\mathit{Labeled}\ \ell\ \tau$ classifies data of type $\tau$ with a security label $\ell$. For example $\mathit{creditCard} :: \mathit{Labeled}\ H\ \mathit{Int}$ is a sensitive integer, while $\mathit{weather} :: \mathit{Labeled}\ L\ \mathit{String}$ is a public string. The abstract type $\mathit{MAC}\ \ell\ \tau$ denotes a (possibly) side-effectful secure computation which handles information at sensitivity level $\ell$ and yields a value of type $\tau$ as a result. Importantly, a $\mathit{MAC}\ \ell\ \tau$ computation enjoys a monadic structure, i.e., it is built by the two fundamental operations $\mathit{return} :: \tau \rightarrow \mathit{MAC}\ \ell\ \tau$ and $(\gg\!=) :: \mathit{MAC}\ \ell\ \tau \rightarrow (\tau \rightarrow \mathit{MAC}\ \ell\ \tau') \rightarrow \mathit{MAC}\ \ell\ \tau'$ (called "bind"). The operation $\mathit{return}\ x$ produces a computation that returns the value denoted by $x$ without causing side-effects. The function $(\gg\!=)$ is used to *sequence* computations and their corresponding side-effects. Specifically, $m \gg\!= f$ takes the *result* of running the computation $m$ and passes it to the function $f$, which then returns a second computation to run. Haskell provides syntactic sugar for monadic computations known as **do**-notation. For instance, 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 3.

```
do x ← m
   return (x + 1)
```

Figure 3: **do**-notation

*Flows of information:* Abstractly, the side-effects of a $\mathit{MAC}\ \ell\ \tau$ computation involve either reading or writing data.

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

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

We need to ensure that these actions respect the flows of information that are permitted by the security lattice. The functions *label* and *unlabel* allow $\mathit{MAC}\ \ell\ \tau$ computations to securely interact with labeled expressions, which are the simplest kind of resources available in **MAC**. If a $\mathit{MAC}\ \ell_{\mathrm{L}}$ computation writes data into a sink, the computation needs to have at most the sensitivity of the sink itself. This restriction, known as *no write-down* [24], preserves the sensitivity of data handled by the $\mathit{MAC}\ \ell_{\mathrm{L}}$-computation. The function *label* creates a fresh, labeled value. From the security point of view, this action corresponds to allocating a fresh location in memory and immediately writing a value into it—hence the no write-down principle applies. The type signature of *label* has a *type constraint* before the symbol $\Rightarrow$, which is a property that types must follow. The constraint $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$ ensures that, when calling $\mathit{label}\ x$, the level of the computation $\ell_{\mathrm{L}}$ is no more confidential than the sensitivity $\ell_{\mathrm{H}}$ of the labeled value that it creates. In contrast, a computation $\mathit{MAC}\ \ell_{\mathrm{H}}\ \tau$ is only allowed to read labeled values at most as sensitive as $\ell_{\mathrm{H}}$. This restriction is known as *no read-up* [24] and gets enforced by the constraint $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$ in the type signature of *unlabel*. This paper focuses on labeled expression, but **MAC** provides additional side-effecting primitives for exception handling, network communication, references, and synchronization primitives [10].

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

*Concurrency:* The mere possibility to run (conceptually) simultaneous $\mathit{MAC}\ \ell$ computations provides attackers with new tools to bypass security checks. In particular, threads introduce the *internal timing covert channel* described in the introduction. Furthermore, it considerably magnifies the bandwidth of the termination covert channel, where secrets are learned by observing the terminating behavior of

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

(App$_1$)

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

(App$_2$)

$$(\Delta, \lambda y.t, x : S) \rightsquigarrow (\Delta, t\ [x\ /\ y], S)$$

(Var$_1$)

$$(\Delta[x \mapsto t], x, S) \rightsquigarrow (\Delta, t, \#x : S)$$

(Var$_2$)

$$(\Delta, v, \#x : S) \rightsquigarrow (\Delta[x \mapsto v], v, S)$$

Figure 5: Syntax and semantics à la Sestoft

threads [12]. To securely support concurrency, **MAC** forces programmers to decouple computations which depend on sensitive data from those performing public side-effects. In this manner, non-terminating loops based on secrets cannot affect the outcome of public events. In this light, the type signature of $\mathit{fork}^{\mathsf{MAC}}$ in Figure 2 only allows spawning threads, i.e., a secure computation with type $\mathit{MAC}\ \ell_{\mathrm{H}}\ ()$, which are at least as sensitive as the current computation, i.e., $\mathit{MAC}\ \ell_{\mathrm{L}}\ ()$. It is secure to do so because that decision depends on less sensitive data ($\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$).

## III. Lazy Calculus

In order to rigorously analyze the information leaks introduced by sharing, we need to build on top of a formal semantics that is operationally precise enough to make sharing observable. The default choice for such a semantics is Launchbury's "Natural Semantics for lazy evaluation" [25], where the structure of the heap is explicit and sharing, as well as cyclic data structures, are manifestly visible. The heap is a partial map from names to terms. This representation is still more abstract than other formalisations such as the Spineless Tagless G-machine (STG) [26], which concerns itself with pointers and memory representation, and is the basis of the Haskell implementation GHC [27]. That much operational detail would only clutter this work, and in terms of lazy evaluation, Launchbury's semantics is a suitable model of the actual implementation.

This work will have to address concurrency, for which a big-step semantics such as Launchbury's is unsuitable for. Therefore, we build on Sestoft's rendering of Launchbury's semantics as an abstract machine with small-step semantics [28]. Here, a judgement $(\Delta, t, S) \rightsquigarrow (\Delta', t', S')$ indicates that a configuration consisting of a current expression $t$, a heap $\Delta$, and a stack $S$ takes one step to the configuration on the right hand side of the arrow.

The rules in Figure 5 describe the transitions of the abstract machine for the standard syntactical constructs. Rule (App$_1$) initiates a function call. Since we work in a lazy setting, the function argument $t_2$ is not evaluated at this point. Instead, it is stored on the heap as a *thunk*, i.e., an unevaluated expression, under a fresh name $x$ with regard to the whole configuration—which corresponds to allocating memory. The machine proceeds to evaluate the function expression $t_1$ to a lambda expression. Then, rule (App$_2$) takes over and substitutes the name of the argument $x$, which is found on the stack, into the body $t$ of the lambda expression. The argument $x$ may, however, need to be evaluated at some point. Rule (Var$_1$) finds the corresponding thunk $t$ on the heap and, after leaving an *update marker* $\#x$ on the stack, begins to evalute the thunk—intuitively, this marker indicates that when the evaluation of the current term finishes, the denoted value gets stored in $x$. During evaluation, $x$ is removed from the heap. If the evaluation of $t$ required the value of $x$, the machine would get stuck. This effect is desired: if the binding for $x$ were to remain on the heap, evaluation would simply start to run in circles. Removing the variable from the heap, a technique called *blackholing*, makes this error condition detectable. When the machine reduces the thunk to a value $v$, rule (Var$_2$) pops the update marker from the stack and puts $x$ back on the heap, but now referencing to the value $v$. Every future use of $x$ will use $v$ directly instead of re-calculating it. This *updating* operation is the crucial step to implement sharing behavior.

We simplified Sestoft's presentation of the semantics in a few ways to remove aspects not relevant for the discussion at hand and to facilitate our machine-checked proofs in Agda: *i)* our syntax does not include mutual recursive **let** expressions; *ii)* in contrast to Sestoft and Launchbury, we allow non-trivial arguments in function application, i.e., our terms are not necessarily in Administrative Normal Form (ANF). In that manner, a non-recursive let expression such as **let** $x = t_1$ **in** $t_2$ can be expressed as $(\lambda x.t_2)\ t_1$; *iii)* although omitted in this presentation, our formalism sports types with multiple values (e.g., Boolean expressions) and the corresponding case-analysis clause (e.g., **if-then-else**) by using the rules found in [29].

### A. Security Primitives

We now extend this standard calculus with the security primitives of **MAC** as shown in Figure 6. The new type $\mathit{Labeled}\ \ell\ \tau$ consists of pure values $t :: \tau$ wrapped in *Labeled*, and annotates them with the security level $\ell$. We call $\mathit{Labeled}\ 42 :: \mathit{Labeled}\ \ell\ \mathit{Int}$ a pure, side-effect free labeled-$\ell$ resource with content 42. We introduce a further form of labeled resource, namely references, in the next section. The semantics rules in Figure 6 are fairly straight-forward and follow the pattern seen in Figure 5. It is worth noting that

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

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

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

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

$$\text{(BIND}_1\text{)}\quad (\Delta, t_1 \gg\!= t_2, S) \rightsquigarrow (\Delta, t_1, \gg\!= t_2 : S)$$

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

Figure 6: Security primitives

thanks to the static nature of **MAC**, no run-time checks are needed to prevent insecure flows of information in these rules.

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

The actual **MAC** implementation knows even more labeled resources (e.g., network) [10]. **MAC** requires no modification to Haskell's type system in order to handle labels: each label is defined as an *empty type*, i.e., a type that has no value, and labeled resources (type *Labeled*) use labels as *phantom types*, i.e., a type parameter that only carries the sensitivity of data at the type-level.

### B. References

We now extend the abstract machine with mutable references, a feature available in **MAC** to boost the performance of secure programs [10]. References live in the *memory* $M$, which is simply a list of variables, added as a component of the program configuration—see Figure 7. The address of a memory cell is its index in this list. The memory $M[n \mapsto x]$ is $M$ with its $n$-th cell changed to refer to $x$. Observe that the memory $M$ and the heap $\Delta$ are two distinct syntactic categories and that, while the latter contains arbitrary terms and enjoys sharing, the former merely contains pointers to the heap. A *labeled* reference is represented as a value $\mathit{Ref}\ n :: \mathit{Ref}\ \ell\ \tau$ where $n$ is the address of the $n$-th memory cell, which contains a variable (a "pointer") to some term $t :: \tau$ on the heap[3]. Only secure computations can manipulate these *labeled references* using the secure primitives in Figure 8. Observe that the types are restricted according to the *no read-up* and *no write-down* restrictions—like those of *label* and *unlabel*.

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

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

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

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

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

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

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

Figure 7: Syntax and semantics for references

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

Figure 8: API of memory operations

The extended semantics is represented as the relation $c \longrightarrow c'$ which extends $\rightsquigarrow$ via [LIFT]—see Figure 7. Rule [NEW] allocates the second argument on the heap with a fresh name $x$, extends the memory with a new pointer to $x$ and returns a reference to it. Rule [WRITE$_1$] evalutes its first argument to a reference and rule [WRITE$_2$] overrides the memory cell with a pointer to a newly allocated heap entry, just like

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

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

Figure 10: Syntax for the concurrent calculus

$new$. The two [READ]-rules retrieve a pointer from memory. To the best of our knowledge, this is the first published operational semantics that models both lazy evaluation and mutable references, and although we constructed it using standard techniques, we would like to point out a crucial subtlety.

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

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

Figure 9: Immutability.

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

### C. Concurrency

Finally, we extend our language with concurrency in the form of threads whose execution interleave[4].

We consider *global configurations* of the form $\langle M, \Delta, T_s \rangle$, where thread pool $T_s$ consists of a list of threads—see Figure 10. A thread $(t, S)$ is an *interrupted* secure computation, consisting of a control term $t$ and a stack S. Within a global configuration, threads are identified by their position in the thread pool. For simplicity and brevity, the concurrent calculus features a Round Robin scheduler, the same kind of scheduler used by GHC's run-time system[5]—however, our results and semantics generalize to a wide range of deterministic schedulers. In the following, we omit the scheduler from the configuration and from the semantics rules for space reasons.

Figure 11 describes the two rules under which a global configuration $c_1$ steps to $c_2$ (written $c_1 \hookrightarrow c_2$). In both rules ([SEQ] and [FORK]), thread $n$ is executed—the scheduler actually *deterministically* chooses which thread to run, which is retrieved from the thread pool $T_s$. In rule [SEQ], the selected thread, i.e., $(t_1, S_1)$, takes a sequential step that is paired with the current memory and heap: $(\langle M_1, \Delta_1, t_1, S_1 \rangle \longrightarrow \langle M_2, \Delta_2, t_2, S_2 \rangle)$. The global configuration is then updated accordingly to the final sequential configuration; in particular, the thread pool is updated with the reduced thread, i.e., $T_s[n \mapsto (t_2, S_2)]$. In rule [FORK], the selected thread spawns a thread—note that term $\mathit{fork}\ t$ is stuck in the sequential semantics and rule [SEQ] does not apply. The new thread is assigned the fresh identifier $m = |T_s|$—thread pool $T_s$ contains threads 0 ... $|T_s| - 1$. Lastly, the thread pool is updated with the parent thread, appropriately reduced to $(\mathit{return}\ (), S)$, and by inserting the new thread initialized with an empty stack, i.e., $(t, [\,])$, at position $m$.

Note that a thread that tries to evaluate a variable $x$ that is already under evaluation by another thread will not find this variable on the heap, due to the blackholing mechanism explained earlier. The thread is now blocked, guaranteeing that, even in the concurrent setting, every thunk will only be evaluated at most once. This mechanism is consistent with the operational semantics used by Finch et al. [30].

## IV. DUPLICATING THUNKS

This section presents one of our main contributions: a primitive, called *lazyDup*, to prevent sharing. Given a term $t$, evaluating *lazyDup* $t$ will *lazily* create a copy of $t$. The laziness is necessary in order to duplicate cyclic or infinite data structures without sending the program into a loop. We first present the basic semantics of *lazyDup* and then describe how we handle references.

### A. Semantics

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

[4] **MAC** provides also synchronization variables [10], which we omit here.

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

(Seq)

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

(Fork)

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

Figure 11: Semantics of the concurrent calculus

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

(LazyDup$_1$)

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

(LazyDup$_2$)

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

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

Figure 12: Synatx and semantics of *lazyDup*

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

$$\begin{aligned}
\llbracket \mathit{Ref}\ n \rrbracket^B &= \mathit{DRef}\ n \\
\llbracket \mathit{DRef}\ n \rrbracket^B &= \mathit{DRef}\ n
\end{aligned}$$

(Read$^D$)

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

(Write$^D$)

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

Figure 13: Duplicate-on-read memory operations

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

### B. References

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

Crucially, we need to propagate any write operation on the original reference to the duplicated reference. One manner to achieve that is to have both references pointing to the same memory location but carefully preventing leaks from reading this shared resource. In this light, we introduce a new variant of reference, called *duplicate-on-read reference*[6], which is represented by $\mathit{DRef}\ n$. When reading from a $\mathit{DRef}\ n$, we wrap the read value in a call to *lazyDup*, as shown in Figure 13, while write operations on a duplicate-on-read reference are executed as usual. Function $\llbracket . \rrbracket$ does not need to follow references and duplicate their content, but simply turns them into duplicate-on-read-references. In this sense, we apply *lazyDup* lazily to reference: the duplication is suspended and continues when the reference is read.

## V. Securing **MAC**

We now pinpoint the vulnerability leveraged by the attack sketched in the introduction and show how to modify **MAC**

[6] The same approach applies to synchronization variables.

(Fork)

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

Figure 14: Patch needed to secure **MAC**

to close it using *lazyDup*. It turns out that one careful modification to the [Fork] rule suffices. This change, highlighted in greeen in Figure 14, ensures that when we create a new thread to evaluate $t$, it will work on a lazily duplicated copy of $t$, i.e., *lazyDup* $t$. As a result, each thunk shared between the parent and the child thread gets lazily duplicated: the parent thread works on the original thunk, while the child thread works on a copy.[7]

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

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

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

## VI. Security Guarantees

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

$$\begin{tikzcd}
c \arrow[r] \arrow[d, dashed, "\varepsilon_{\ell_A}"'] & c' \arrow[d, dashed, "\varepsilon_{\ell_A}"] \\
\varepsilon_{\ell_A}(c) \arrow[r] & \varepsilon_{\ell_A}(c')
\end{tikzcd}$$

Figure 15: Single-step simulation

$$\begin{array}{lll}
\text{Pure conf. } \ell\text{:} & c & ::= (\Delta^{\ell}, t, S^{\ell}) \\
\text{Seq. conf. } \ell\text{:} & c & ::= \langle \Sigma, \Gamma, t, S^{\ell} \rangle \\
\text{Heap map:} & \Gamma & ::= (\ell : \mathit{Label}) \to \mathit{Heap}\ \ell \\
\text{Store:} & \Sigma & ::= (\ell : \mathit{Label}) \to \mathit{Memory}\ \ell \\
\text{Memory } \ell_{\mathrm{H}}\text{:} & M & ::= \cdots \mid x^{\ell_{\mathrm{L}}} : M \\
\text{Terms } \tau\text{:} & t & ::= \cdots \mid x^{\ell} \\
\text{Cont. } \ell\text{:} & C & ::= \cdots \mid x^{\ell} \mid \#x^{\ell} \\
\text{Conc. conf.:} & c & ::= \langle \Sigma, \Gamma, \Phi \rangle \\
\text{Pool map:} & \Phi & ::= (\ell : \mathit{Label}) \to \mathit{Pool}\ \ell
\end{array}$$

Figure 16: Decorated Calculus

### A. Term Erasure

*Term erasure* is a technique to prove non-interference in functional programs [31] and IFC libraries (e.g., [9], [12], [21], [32], [33]). It relies on an erasure function, which we denote by $\varepsilon_{\ell_A}$. This function rewrites data above the attacker's security level, denoted by label $\ell_A$, to the special syntactic construct $\bullet$. At the core, this technique establishes a simulation between reductions of configurations and reductions of their erased counterparts. Figure 15 shows that erasing sensitive data from a configuration $c$ and then taking a step (orange path) yields the same configuration as first taking a step and then erasing sensitive data (cyan path), i.e., the diagram *commutes*. If the configuration $c$ were to leak sensitive data into a non-sensitive resource, then it will remain in $\varepsilon_{\ell_A}(c')$. The same data would be erased in $\varepsilon_{\ell_A}(c)$ and the diagram would not commute.

### B. Decorated Calculus

The erasure proof technique was conceived to work on dynamic IFC approaches [31], where security labels are attached to terms. Applying term erasure to **MAC**, where labels are parts of the types instead of the terms, demands to extend our calculus with extra information about the

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

sensitivity nature of terms. As in similar work [33], [34], we annotate terms with their type and make the erasure function *type-driven*. The annotated term $t :: \tau$ denotes that the term $t$ has type $\tau$. We likewise decorate configurations, heaps, memories, stacks, and continuations with labels.

Figure 16 summarizes the main changes required to decorate our calculus. A pure configuration $\langle \Delta^\ell, t, S^\ell \rangle$, labeled with $\ell$, consists of a labeled heap $\Delta^\ell$, and a labeled stack $S^\ell$. An $\ell$-labeled heap $\Delta^\ell$ can be accessed by $\ell$-labeled variables, e.g., $x^\ell$. An $\ell$-labeled stack contains exclusively $\ell$-labeled continuations, which involve only $\ell$-labeled variables, i.e., continuations $x^\ell$ and $\#x^\ell$. Furthermore an $\ell$-labeled heap contains terms that can be evaluated only by threads at level $\ell$. A sequential configuration $\langle \Sigma, \Gamma, t, S^\ell \rangle$ labeled with $\ell$, consists of a store $\Sigma$, a current term $t$, an heap map $\Gamma$, and a labeled stack $S^\ell$. An $\ell$-labeled configuration denotes a computation of type $\mathit{MAC}\ \ell\ \tau$, for some type $\tau$. Note that this does not necessarily mean that term $t$ is a $\mathit{MAC}$ computation—when the configuration steps the current term is changed with the next redex, which might have a completely different type. Instead, the combination of current term *and* stack guarantees that the whole configuration represents a $\mathit{MAC}\ \ell$ computation.

It is known that dealing with dynamic allocation of memory makes it challenging to prove non-interference (e.g., [35], [36]). One manner to tackle this technicality is by establishing a bijection between public memory addresses of the two executions we want to relate and considering equality of public terms up to such notion [35]. Instead, and similar to other work [9], [32]–[34], we compartmentalize the memory into isolated labeled segments, one for each label of the lattice. This way, allocation in one segment does not affect the others. A similar argument holds for the heap and the thread pool, which we therefore also organize in partitions, accessed through the heap map $\Gamma$ respectively the pool map $\Phi$. Since we now have multiple heaps in one configuration, we need to annotate the *free* variables with the label of the heap in which they are bound. So a variable $x^\ell$ denotes that $x$ is bound in the heap $\Gamma(\ell)$. Variables bound inside a term remain unlabeled, e.g., $\lambda x.x$. A variable $x^{\ell_{\mathrm{L}}}$ in a $\ell_{\mathrm{H}}$-labeled memory will have a label of at most the memory's sensitivity, $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$. Unlike variables, we do not need to annotate memory cells $n$, as they only occur in a $\mathit{Ref}\ n$ expression, which carries a label in its type. So a reference $\mathit{Ref}\ n :: \mathit{Ref}\ \ell\ \tau$ points to the $n$-th entry in the $\ell$-labeled memory. In the following, we write $\mathrm{fresh}(x^\ell)$ to denote that variable $x$ is fresh with respect to heap $\Gamma(\ell) = \Delta^\ell$ and stack $S^\ell$. We write $\Gamma[\ell][x^\ell] := t$ for the heap map obtained by performing the update $\Gamma(\ell)[x^\ell \mapsto t]$, and likewise for stores and pool maps.

### C. Decorated Semantics

The interesting rules of the annotated semantics are shown in Figure 17. The rules for the pure fragment of the calculus

(App$_1$)

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

(App$_2$)

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

(Var$_1$)

$$(\Delta^\ell[x^\ell \mapsto t], x^\ell, S^\ell) \rightsquigarrow (\Delta^\ell, t, \#x^\ell : S^\ell)$$

(Var$_2$)

$$(\Delta^\ell, v, \#x^\ell : S) \rightsquigarrow (\Delta^\ell[x^\ell \mapsto v], v, S^\ell)$$

(Lift)

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

(LazyDup$_1$)

$$\frac{\neg\ \mathit{isVar}\ (t) \qquad \mathrm{fresh}(x^\ell)}{\begin{array}{l}\langle \Sigma, \Gamma, \mathit{lazyDup}\ t, S^\ell \rangle \longrightarrow \\ \qquad \langle \Sigma, \Gamma[\ell][x^\ell] := t, \mathit{lazyDup}\ x^\ell, S^\ell \rangle\end{array}}$$

(LazyDup$_2$)

$$\frac{x^{\ell_{\mathrm{L}}} \mapsto t\ \in\ \Sigma(\ell_{\mathrm{L}}) \qquad \mathrm{fresh}(y^{\ell_{\mathrm{H}}})}{\begin{array}{l}\langle \Sigma, \Gamma, \mathit{lazyDup}\ x^{\ell_{\mathrm{L}}}, S^{\ell_{\mathrm{H}}} \rangle \longrightarrow \\ \qquad \langle \Sigma, \Gamma[\ell_{\mathrm{H}}][y^{\ell_{\mathrm{H}}}] := [\![ t ]\!]^{\varnothing}, y^{\ell_{\mathrm{H}}}, S^{\ell_{\mathrm{H}}} \rangle\end{array}}$$

(New)

$$\frac{|\Sigma(\ell_{\mathrm{H}})| = n \qquad \mathrm{fresh}(x^{\ell_{\mathrm{L}}})}{\begin{array}{l}\langle \Sigma, \Gamma, \mathit{new}\ t, S^{\ell_{\mathrm{L}}} \rangle \longrightarrow \\ \langle \Sigma[\ell_{\mathrm{H}}][n] := x^{\ell_{\mathrm{L}}}, \Delta[\ell_{\mathrm{L}}][x^{\ell_{\mathrm{L}}}] := t, \mathit{return}\ (\mathit{Ref}\ n), S^{\ell_{\mathrm{L}}} \rangle\end{array}}$$

(Write$_2$)

$$\frac{\mathrm{fresh}(x^{\ell_{\mathrm{L}}})}{\begin{array}{l}\langle \Sigma, \Gamma, \mathit{Ref}\ n, \mathit{write}\ t : S^{\ell_{\mathrm{L}}} \rangle \longrightarrow \\ \qquad \langle \Sigma[\ell_{\mathrm{H}}][n] := x^{\ell_{\mathrm{L}}}, \Gamma[\ell_{\mathrm{L}}][x^{\ell_{\mathrm{L}}}] := t, \mathit{return}\ (), S^{\ell_{\mathrm{L}}} \rangle\end{array}}$$

(Read$_2$)

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

(Read$^D$)

$$\begin{array}{l}\langle \Sigma, \Gamma, \mathit{DRef}\ n, \mathit{read} : S^{\ell_{\mathrm{H}}} \rangle \longrightarrow \\ \qquad \langle \Sigma, \Gamma, \mathit{return}\ (\mathit{lazyDup}\ \Sigma(\ell_{\mathrm{L}})[n]), S^{\ell_{\mathrm{H}}} \rangle\end{array}$$

Figure 17: Decorated Semantics

are adapted to work with labeled variables. Note that rule [App$_2$] replaces the bound, hence unlabeled, variable $y$ with the labeled variable $x^{\ell}$ and thus maintains the invariant that *free* variables are labeled.

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

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

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

Rule [Lift] executes a pure reduction step, giving it access to the appropriate heap $\Gamma(\ell)$ and updating the heap map afterwards. Rules [LazyDup$_1$] and [LazyDup$_2$] adapt the semantics of *lazyDup* to label-partitioned heaps. The first rule takes care of allocating a non-trivial argument on the heap labeled as the current configuration. The second rule is the heart of our security leak fix: it handles the case where a high thread reads a thunk from a lower context. The rule fetches the thunk $t$ from the lower heap, i.e., $t \mapsto \Sigma(\ell_{\mathrm{L}})$, and extends the heap labeled as the configuration with a copy of the thunk, i.e., $\Sigma[\ell_{\mathrm{H}}][y^{\ell_{\mathrm{H}}}] := [\![t]\!]^{\varnothing}$. Observe that this operation relabels the original thunk $t$ from $\ell_{\mathrm{L}}$ to $\ell_{\mathrm{H}}$ securely because $t$ is duplicated, ensuring that the free variables in $t$ will, by the time they are about to be evaluated, be wrapped in *lazyDup*, so that [LazyDup$_2$] kicks in again. In rule [New], a computation at level $\ell_{\mathrm{L}}$ creates a reference labeled with $\ell_{\mathrm{H}}$. The thunk $t$ is allocated on the $\ell_{\mathrm{L}}$ heap under the name $x^{\ell_{\mathrm{L}}}$, which is written to the fresh cell in memory $\Sigma(\ell_{\mathrm{H}})$. This ensures the invariant that in *well-typed* configurations a memory holds references to lower heaps. The same applies to rule [Write$_2$]. Rule [Read$_2$] enforces that a computation at level $\ell$ can only read from a non-duplicated reference if the referenced variable is at the same level $\ell$. Relaxing this would allow a high thread to read a thunk from a low level and thus open another leaky channel. But the interplay of rule [Fork] (in its annotated variant in Figure 18 in Appendix C), rule [LazyDup$_2$] and *lazyDup* rewriting of references to duplicate-on-read references precludes this scenario. Rule [Read$^D$] then allows a $\ell_{\mathrm{H}}$ high computation to read a low variable from a duplicate-on-read reference, by duplicating it to ensure security.

### D. Erasure Function

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

### E. Decorated Progress-Sensitive Non-Interference

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

*Proposition 1 (Determinancy):* If $c_1 \hookrightarrow c_2$ and $c_1 \hookrightarrow c_3$ then $c_2 \equiv c_3$.

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

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

*Proposition 2 (Simulation):* Given a global reduction step $c_1 \hookrightarrow_{(\ell,n)} c_1'$ then

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

From Propositions 1 and 2, we prove progress-sensitive non-interference. Note that, unlike our previous work [33], Proposition 2 does not simulate sensitive threads, because $\ell_A$-equivalence suffices for PSNI. For more details, please refer to our Agda formalization[8].

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

*Theorem 1 (PSNI):* Given two configurations $c_1 \approx_{\ell_A} c_2$ and a reduction $c_1 \hookrightarrow_{(\ell,n)} c'_1$, then there exists a configuration $c'_2$ such that $c'_1 \approx_{\ell_A} c'_2$ and $c_2 \hookrightarrow^* c'_2$.
As usual, $\hookrightarrow^*$ denotes the transitive reflexive closure of $\hookrightarrow$.

*Proof:* If $\ell \sqsubseteq \ell_A$, by $\ell_A$-equivalence, configuration $c_2$ contains a thread identified by $(\ell, n)$, that is $\ell_A$-equivalent to that run by $c_1$. However, $c_2$ might contain a *finite* number of high threads, which are scheduled before that. After running those high threads, i.e., $c_2 \hookrightarrow^* c''_2$, for some configuration $c''_2$, the same low thread is scheduled, i.e., $c''_2 \hookrightarrow_{(\ell,n)} c'_2$, for some other configuration $c'_2$. Applying the simulation proposition to the first set of steps yields $c_2 \approx_{\ell_A} c''_2$, as they are all above the attackers level, and by transitivity it follows that $c_1 \approx_{\ell_A} c''_2$, i.e., $\varepsilon_{\ell_A}(c_1) \equiv \varepsilon_{\ell_A}(c''_2)$. Applying simulation again we learn that $\varepsilon_{\ell_A}(c''_2) \hookrightarrow_{(\ell,n)} \varepsilon_{\ell_A}(c'_2)$, since $\ell \sqsubseteq \ell_A$ as well as $\varepsilon_{\ell_A}(c_1) \hookrightarrow_{(\ell,n)} \varepsilon_{\ell_A}(c'_1)$. The determinancy proposition shows $\varepsilon_{\ell_A}(c'_1) \equiv \varepsilon_{\ell_A}(c'_2)$ or, in other words, $c'_1 \approx_{\ell_A} c'_2$. If $\ell \not\sqsubseteq \ell_A$, then simulation tells us $c_1 \approx_{\ell_A} c'_1$ and $c''_2 \approx_{\ell_A} c'_2$, so we obtain $c'_1 \approx_{\ell_A} c'_2$ by transitivity.

■

### F. Simulation between Vanilla and Decorated semantics

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

The main challenge is to map the partitioned heap, memory, and stack in the annotated calculus into a single heap, memory, and stack and vice versa. We apply techniques inspired by other IFC works on dynamic allocation [35] and partitioned heaps [32] and show that configurations in the annotated calculus are equal to those in the vanilla calculus *up to bijection on variables names and memory addresses*. These bijections describe how to flatten the partitioned memories and heaps into single entities without changing the results produced by programs—of course, modulo variable names and memory addresses. Note that our references are opaque to programs, i.e., there is no pointer arithmetic, equality, etc., which makes the proof easier.

We work with two bijections, $\Psi_1$ for heap variables and $\Psi_2$ for memory addresses:

$$\begin{aligned}&\Psi_1 :: (\mathit{Label}\ \times\ \mathit{Var}) \to \mathit{Var}\\ &\Psi_2 :: (\mathit{Label}\ \times\ \mathbb{N}) \to \mathbb{N}\end{aligned}$$

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

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

*Proposition 3 (Decorated to vanilla):* Given well-typed configurations $\langle \Sigma, \Gamma, t_1, S^\ell \rangle$ and $\langle M, \Delta, t_2, S \rangle$ that denote a computation of type $\mathit{MAC}\ \ell\ \tau$, if we have that:

- $\langle \Sigma, \Gamma, t_1, S^\ell \rangle \longrightarrow \langle \Sigma', \Gamma', t'_1, S'^\ell \rangle$ and
- $\langle \Sigma, \Gamma, t_1, S^\ell \rangle \cong_\Psi \langle M, \Delta, t_2, S \rangle$

then there exist $M'$, $\Delta'$, $t'_2$, $S'$ and $\Psi'$ such that:

- $\langle M, \Delta, t_2, S \rangle \longrightarrow \langle M', \Delta', t'_2, S' \rangle$
- $\langle \Sigma', \Gamma', t'_1, S'^\ell \rangle \cong_{\Psi'} \langle M', \Delta', t'_2, S' \rangle$

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

*Proposition 4 (Vanilla to decorated):* Given well-typed configurations $\langle \Sigma, \Gamma, t_1, S^\ell \rangle$ and $\langle M, \Delta, t_2, S \rangle$ that denote a computation of type $\mathit{MAC}\ \ell\ \tau$, if we have that:

- $\langle M, \Delta, t_2, S \rangle \longrightarrow \langle M', \Delta', t'_2, S' \rangle$
- $\langle \Sigma, \Gamma, t_1, S^\ell \rangle \cong_\Psi \langle M, \Delta, t_2, S \rangle$

then there exist $\Sigma'$, $\Gamma'$, $t'_1$, $S'^\ell$ and $\Psi'$ such that:

- $\langle \Sigma, \Gamma, t_1, S^\ell \rangle \longrightarrow \langle \Sigma', \Gamma', t'_1, S'^\ell \rangle$ and
- $\langle \Sigma', \Gamma', t'_1, S'^\ell \rangle \cong_{\Psi'} \langle M', \Delta', t'_2, S' \rangle$

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

### G. Vanilla Progress-Sensitive Non-Interference

We prove that *well-typed* programs in the vanilla lazy calculus satisfy progress-sensitive non-interference. This result relies on the PSNI proof for the decorated calculus and the simulations described above. We first define that two global configurations are $\ell_A$-equivalence *up to a bijection* $\Omega$, written $\langle M_1, \Delta_1, T_{s1} \rangle \approx^{\Omega}_{\ell_A} \langle M_2, \Delta_2, T_{s2} \rangle$, if and only if they are *well-typed* and their components are $\ell_A$-equivalent up to bijection $\Omega$, where $\ell_A$-equivalence between terms is also type-driven and follows a structure similar to the one for the decorated calculus—the main difference being that it inspects the type-derivation of term and use the bijection $\Omega$ to relate memory addresses and heap variables. In the vanilla calculus we need to consider low-equivalence up to a bijection as in [35] to relate executions which might allocate a different amount of high entities, thus affecting the addresses and names of public references and variables respectively. Observe that bijection $\Omega$ connects heap variables and memory addresses of the vanilla calculus, that is $\Omega$ is a pair of bijections of type:

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

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

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

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

there exists $c_2$ and bijection $\Omega$ such that:

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

*Proof:* (Sketch) Define $i_1 = \langle [\,], \varnothing, [(t_1, [\,])] \rangle$ and $i_2 = \langle [\,], \varnothing, [(t_2, [\,])] \rangle$. Since $t_1$ and $t_2$ are closed and well-typed terms in the surface syntax, we can lift them in the decorated calculus, as decorated terms $t_1^D$, $t_2^D$, and their corresponding initial annotated configurations $i_1^D$ and $i_2^D$. Configurations $i_1$ and $i_2$ are equivalent up to the empty bijection $\varnothing$, i.e., $i_1^D \cong_{\emptyset} i_1$ and $i_2^D \cong_{\emptyset} i_2$ and $i_1^D \approx_{\ell_A} i_2^D$. By lifting Proposition 4 to thread pools and repetitively applying it, there exists a bijection $\Psi_a$ and a configuration $c_1^D$, such that $i_1^D \hookrightarrow^* c_1^D$ and $c_1^D \cong_{\Psi_a} c_1$. By Theorem 1, there exists a decorated configuration $c_2^D$ such that $i_2^D \hookrightarrow^* c_2^D$ and $c_1^D \approx_{\ell_A} c_2^D$. By lifting Proposition 3 to thread pools and repetitively applying it, we have that there exists a bijection $\Psi_b$ and configuration $c_2$ such that $i_2 \hookrightarrow^* c_2$ where $c_2^D \cong_{\Psi_b} c_2$. We then conclude that $c_1$ and $c_2$ are $\ell_A$-equivalent up to bijection $\Omega$, obtained composing $\Psi_a$ (from vanilla to decorated), and $\Psi_b^{-1}$ (from decorated to vanilla), i.e., $c_1 \approx^{\Omega}_{\ell_A} c_2$, where $\Omega = \Psi_a \circ \Psi_b^{-1}$. ■

## VII. Related Work

*Mutable references and lazyness:* In Section III-B we present an operational semantics that sports both mutable references and lazyness. It is a straight-forward combination of Sestoft's semantics with the standard approach to model references using a store, as described by Pierce et al. in the context of call-by-value ([37], [38]). To the best of our knowledge, this is the first work that presents this combination. The "Awkward Squad" paper [39], which describes the implementation of I/O in Haskell, and addresses both references and concurrency, remarkably avoids dealing with sharing in its operational semantics.

*deepDup:* Our primitive *lazyDup* was inspired by the related primitive *deepDup* proposed by the second author [40], with the aim to limit sharing in cases where it is actually detrimental to program performance. Because the terms in that work are in Administrative Normal Form (ANF), the rules for *deepDup* look different from our [LAZYDUP$_2$], but this difference is inconsequential. We significantly improve over that work with the support to handle references, via the duplicate-on-read references introduced in Section IV-B. The Haskell library `ghc-dup` implements *deepDup* without changes to the compiler or runtime, therefore we are optimistic that an implementation of *lazyDup* is feasible.

*Evaluation strategies and IFC:* Sabelfeld and Sands suggest that lazy evaluation might be safer than eager evaluation for termination leaks [11]. Buiras and Russo identify the risk imposed by internal timing leaks via lazy evaluation [13]. Vassena et al. enrich **MAC**'s API for labeled expressions by considering them as (applicative-like) functors [34] and show that their extension is vulnerable to termination leaks under eager evaluation, but secure under lazy evaluation. In a imperative sequential setting, Rafnsson et al. describe how Java's on-demand (lazy) class initialization process can be exploit to reveal secrets [41]. Strictness analysis detects functions that always evaluate their arguments, which can then be eagerly evaluated to boost performance of lazy evaluation [42]. In this context, this technique could be used to safely force the evaluation of shared thunks upfront. However, the analysis must necessarily be conservative, especially when it comes to infinite data structures and advanced features such as references and concurrency, therefore it is unlikely that all leaks could be closed by the analysis alone. Nevertheless, strictness analysis could avoid unnecessary duplication: the thunks, which are guaranteed by the analysis to be evaluated anyway, could be eagerly forced, and lazy duplication could be applied otherwise.

*IFC libraries:* **LIO** dynamically enforces IFC applying similar concepts to **MAC** (i.e., labeled expressions, secure computations, etc.). We argue that **LIO** can be secure against the attack presented in this work by applying *lazyDup* to the "rest of the computation" every time that the current label gets raised. For that, **LIO** needs to be reimplemented to work in a continuation passing style (CPS)—we leave this direction as future work. **HLIO** (hybrid-**LIO**) works as **LIO**, except it enforces IFC by combining type-level enforcement with dynamic checks [21]. To secure **HLIO**, *lazyDup* needs to be inserted when forking threads if IFC gets enforced statically and when raising the current label if dynamic checks are involved. **HLIO** also needs to be reimplemented using CPS. In **MAC**, the type signature for the bind operator restricts computations to maintain the same security level. Its type could be relaxed to involve different increasing labels, along the lines of the "is protected" relation used in the typing rule of bind in the Dependency Core Calculus (DCC) [43]. However, in that case, a secure computation would not enjoy a standard monadic structure, but it would rather incorporate multiple monads.

Devriese and Piessens provide a monad transformer to extend imperative-like APIs with support for IFC [44]. Jaskelioff and Russo implement a library which dynamically enforces IFC using secure multi-execution (SME) [45]. Schmitz et al. [46] provide a library with *faceted values*, where values present different behavior according to the privilege of the observer. While these libraries do not support concurrency yet, we believe that, this work could secure them against lazy evaluation attacks, if they were extended with concurrency.

*Programming languages:* Besides the already mentioned tools Jif, Paragon, FlowCaml, and JSFlow, we can remark the SPARK language and its IFC analysis, which has been extended to guarantee progress-sensitive non-inference [47] and JOANA [48], which stretches the scalability of static analyzes, in this case of Java programs. Some tools apply dependent-types to protect confidentiality (e.g., [49]–[51]). In such languages, type-checking triggers evaluation, potentially opening up possibilities to leak sensitive data via covert channels (e.g., lazy evaluation). In this light, it would be possible to learn something about a static secret when type-checking the program—an interesting direction for future work. Laminar combines programming languages and operating systems techniques to provide decentralized information flow control [52]. While supporting concurrency, Laminar does not handle covert channels like termination or internal timing leaks.

## VIII. Conclusions

We present a solution to internal timing leaks via lazy evaluation, an open problem for security libraries written in Haskell. We believe that repairing existing libraries with *lazyDup* would be a reasonably painless experience. The utilization of *lazyDup* would make past and future systems built with security libraries more secure (e.g., Hails [53]). Even though it is still not clear which evaluation strategy is more beneficial for security, this work shows that the risks of lazy evaluation in concurrent settings can be successfully avoided.

Generally speaking, functional languages (and Haskell in particular) rely on their runtime (e.g., lazy evaluation, garbage collector, etc.) to provide essential features. Unfortunately, besides providing their functionality, they could also be misused to jeopardize security. This work shows that a program can control parts of the complex runtime system (e.g., sharing) via a safe interface (*lazyDup*). Then, the obvious question is which other features of the runtime system could jeopardize security and how to safely control them—an intriguing thought to drive our future work.

*Acknowledgments:* This work was supported by the Swedish research agency VR.

## References

[1] J. Hughes, "Why functional programming matters," *The Computer Journal*, vol. 32, 1984.

[2] A. Sabelfeld and A. C. Myers, "Language-Based Information-Flow Security," *IEEE J. Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, 2003.

[3] A. C. Myers, "JFlow: Practical mostly-static information flow control," in *ACM Symp. on Principles of Programming Languages*, 1999, pp. 228–241.

[4] F. Pottier and V. Simonet, "Information Flow Inference for ML," in *ACM Symp. on Principles of Programming Languages*, 2002, pp. 319–330.

[5] N. Broberg, B. van Delft, and D. Sands, "Paragon for practical programming with information-flow control." in *APLAS*, ser. LNCS, vol. 8301. Springer, 2013, pp. 217–232.

[6] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld, "JSFlow: Tracking information flow in JavaScript and its APIs," in *ACM Symposium on Applied Computing*. ACM, 2014.

[7] P. Li and S. Zdancewic, "Encoding information flow in Haskell," in *IEEE Workshop on Computer Security Foundations*. IEEE Computer Society, 2006.

[8] E. Moggi, "Notions of computation and monads," *Information and Computation*, vol. 93, no. 1, pp. 55–92, 1991.

[9] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières, "Flexible dynamic information flow control in Haskell," in *ACM SIGPLAN Haskell symposium*, 2011.

[10] A. Russo, "Functional Pearl: Two Can Keep a Secret, if One of Them Uses Haskell," in *ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP. ACM, 2015.

[11] A. Sabelfeld and D. Sands, "A per model of secure information flow in sequential programs," *Higher Order Symbol. Comput.*, vol. 14, no. 1, Mar. 2001.

[12] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Maziéres, "Addressing covert termination and timing channels in concurrent information flow systems," in *ACM SIGPLAN International Conference on Functional Programming*. ACM, 2012.

[13] P. Buiras and A. Russo, "Lazy programs leak secrets," in *Nordic Conference in Secure IT Systems*. Springer-Verlag, 2013.

[14] G. Smith and D. Volpano, "Secure information flow in a multi-threaded imperative language," in *ACM symposium on Principles of Programming Languages*, 1998.

[15] A. Bortz and D. Boneh, "Exposing private information by timing web applications," in *World Wide Web*. ACM, 2007.

[16] E. W. Felten and M. A. Schneider, "Timing attacks on Web privacy," in *ACM conference on Computer and communications security*, ser. CCS. ACM, 2000.

[17] A. Askarov, D. Zhang, and A. C. Myers, "Predictive black-box mitigation of timing channels," in *ACM conference on Computer and Communications Security*. ACM, 2010.

[18] D. Zhang, A. Askarov, and A. C. Myers, "Predictive mitigation of timing channels in interactive systems," in *ACM conference on Computer and Communications Security*. ACM, 2011.

[19] ——, "Language-based Control and Mitigation of Timing Channels," in *ACM Conference on Programming Language Design and Implementation*. ACM, 2012.

[20] D. Volpano, G. Smith, and C. Irvine, "A Sound Type System for Secure Flow Analysis," *J. Computer Security*, vol. 4, no. 3, pp. 167–187, 1996.

[21] P. Buiras, D. Vytiniotis, and A. Russo, "HLIO: Mixing static and dynamic typing for information-flow control in Haskell," in *ACM SIGPLAN International Conference on Functional Programming*. ACM, 2015.

[22] U. Norell, "Dependently typed programming in agda," in *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*, A. Kennedy and A. Ahmed, Eds. ACM, 2009, pp. 1–2. [Online]. Available: http://doi.acm.org/10.1145/1481861.1481862

[23] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Communications of the ACM*, vol. 20, no. 7, pp. 504–513, 1977.

[24] D. E. Bell and L. La Padula, "Secure computer system: Unified exposition and multics interpretation," MITRE Corporation, Bedford, MA, Tech. Rep. MTR-2997, Rev. 1, 1976.

[25] J. Launchbury, "A natural semantics for lazy evaluation," in *Principles of Programming Languages (POPL)*. ACM, 1993.

[26] S. Peyton Jones, "Implementing lazy functional languages on stock hardware: The spineless tagless G-machine," *Journal of Functional Programming*, vol. 2, no. 2, pp. 127–202, 1992.

[27] S. Marlow and S. Peyton Jones, "Making a fast curry: push/enter vs. eval/apply for higher-order languages," *Journal of Functional Programming*, vol. 16, no. 4-5, pp. 415–449, 2006.

[28] P. Sestoft, "Deriving a lazy abstract machine," *Journal of Functional Programming*, vol. 7, no. 03, 1997.

[29] J. Breitner, "Formally proving a compiler transformation safe," in *Haskell Symposium*. ACM, 2015.

[30] C. Baker-Finch, D. J. King, and P. Trinder, "An operational semantics for parallel lazy evaluation," in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '00. New York, NY, USA: ACM, 2000, pp. 162–173. [Online]. Available: http://doi.acm.org/10.1145/351240.351256

[31] P. Li and S. Zdancewic, "Arrows for secure information flow," *Theoretical Computer Science*, vol. 411, no. 19, pp. 1974–1994, 2010.

[32] S. Heule, D. Stefan, E. Z. Yang, J. C. Mitchell, and A. Russo, *IFC Inside: Retrofitting Languages with Dynamic Information Flow Control*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 11–31. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-46666-7_2

[33] M. Vassena and A. Russo, "On formalizing information-flow control libraries," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, ser. PLAS '16. New York, NY, USA: ACM, 2016, pp. 15–28. [Online]. Available: http://doi.acm.org/10.1145/2993600.2993608

[34] M. Vassena, P. Buiras, L. Waye, and A. Russo, "Flexible manipulation of labeled values for information-flow control libraries," in *Proceedings of the 12th European Symposium On Research In Computer Security*. Springer, Sep. 2016.

[35] A. Banerjee and D. A. Naumann, "Stack-based access control and secure information flow," *Journal Functional Programming*, vol. 15, pp. 131–177, 2005.

[36] D. Hedin and D. Sands, "Noninterference in the presence of non-opaque pointers," in *IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 2006.

[37] B. C. Pierce, A. A. de Amorim, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, and B. Yorgey, *Software Foundations*. Electronic textbook, 2016, version 4.0. http://www.cis.upenn.edu/~bcpierce/sf.

[38] B. C. Pierce, *Types and Programming Languages*. MIT Press, 2002.

[39] S. Peyton Jones, *Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell*. IOS Press, January 2001, pp. 47–96.

[40] J. Breitner, "dup – explicit un-sharing in Haskell," *CoRR*, vol. abs/1207.2017, 2012.

[41] W. Rafnsson, K. Nakata, and A. Sabelfeld, "Securing class initialization in Java-like languages," *IEEE Transactions on Dependable and Secure Computing*, vol. 10, no. 1, Jan. 2013.

[42] A. Mycroft, *The theory and practice of transforming call-by-need into call-by-value*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1980, pp. 269–281. [Online]. Available: http://dx.doi.org/10.1007/3-540-09981-6_19

[43] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke, "A Core Calculus of Dependency," in *ACM Symp. on Principles of Programming Languages*, 1999, pp. 147–160.

[44] D. Devriese and F. Piessens, "Information flow enforcement in monadic libraries," in *ACM SIGPLAN Workshop on Types in Language Design and Implementation*. ACM, 2011.

[45] M. Jaskelioff and A. Russo, "Secure multi-execution in Haskell," in *PSI Ershov Informatics Conference*, ser. LNCS. Springer-Verlag, 2011.

[46] T. Schmitz, D. Rhodes, T. H. Austin, K. Knowles, and C. Flanagan, "Faceted dynamic information flow via control and data monads." in *POST*, ser. LNCS, F. Piessens and L. Viganò, Eds., vol. 9635. Springer, 2016.

[47] W. Rafnsson, D. Garg, and A. Sabelfeld, "Progress-sensitive security for SPARK," in *Engineering Secure Software and Systems, London, UK*, 2016.

[48] G. Snelting, D. Giffhorn, J. Graf, C. Hammer, M. Hecker, M. Mohr, and D. Wasserrab, "Checking probabilistic noninterference using JOANA," *it - Information Technology*, vol. 56, no. 6, pp. 280–287, 2014. [Online]. Available: http://www.degruyter.com/view/j/itit.2014.56.issue-6/itit-2014-1051/itit-2014-1051.xml

[49] J. Morgenstern and D. R. Licata, "Security-typed programming within dependently typed programming," in *ACM SIGPLAN International Conference on Functional Programming*. ACM, 2010.

[50] A. Nanevski, A. Banerjee, and D. Garg, "Verification of information flow and access control policies with dependent types," in *IEEE Symposium on Security and Privacy*, ser. SP. IEEE Computer Society, 2011.

[51] L. Lourenço and L. Caires, "Dependent information flow types," *SIGPLAN Not.*, vol. 50, no. 1, Jan. 2015.

[52] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel, "Laminar: Practical fine-grained decentralized information flow control," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI. ACM, 2009.

[53] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. Mitchell, and A. Russo, "Hails: Protecting data privacy in untrusted web applications," in *10th Symposium on Operating Systems Design and Implementation*, October 2012.

## APPENDIX A. SECURING **LIO**

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

## APPENDIX B. SHARING IN PRESENCE OF REFERENCES

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

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

```
let $x = 0$ in
do $r \leftarrow \mathit{new}\ x$
   $\mathit{write}\ r\ 1$
   $\mathit{return}\ x$
```

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

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

```
let $x = \mathit{id}\ 1$ in
do $r \leftarrow \mathit{new}\ x$
   $y \leftarrow \mathit{read}\ r$
   when $(y \leqslant 0)$ $\mathit{return}\ ()$
   $z \leftarrow \mathit{read}\ r$
   when $(z \geqslant 0)$ $\mathit{return}\ ()$
```

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

## APPENDIX C. ERASURE FUNCTION

Figure 19 shows the definition of the erasure functions for the interesting cases. Configurations, whose label is above that of the attacker, i.e., $\ell \not\sqsubseteq \ell_A$ are rewritten to •, otherwise they are erased by erasing each component. Steps involving sensitive configurations are then simulated by rules [HOLE$_1$, HOLE$_2$], shown in Figure 20. Memories, heaps, stacks and thread pools labeled with $\ell$ are also collapsed to •, if their label is not visible to the attacker, i.e., $\ell \not\sqsubseteq \ell_A$, otherwise they are erased homomorphically. Label partitioned data structures, i.e., heap maps, stores and pool maps, are erased pointwise, e.g. $\varepsilon_{\ell_A}(\Gamma) = \ell \mapsto \varepsilon_{\ell_A}(\Gamma(\ell))$. The term *label* $t$ ::

(FORK)

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

(FORK$_\bullet$)

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

Figure 18: Semantics rules of *fork* and *fork*$_\bullet$

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

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

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

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

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

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

Figure 19: Erasure function

(HOLE$_1$) $\bullet \rightsquigarrow \bullet$ (HOLE$_2$) $\bullet \longrightarrow \bullet$

Figure 20: Semantics rules for $\bullet$

$\mathit{MAC}\ \ell_{\text{L}}\ (\mathit{Labeled}\ \ell_{\text{H}}\ \tau)$ is erased to $\mathit{label}\ \bullet$, if $\ell_{\text{H}} \not\sqsubseteq \ell_A$, so that rule [LABEL] commutes. The terms *new*, *write*, *fork* are interesting. Observe that all these terms perform a *write-effect*, to a non-lower security level, due to the no write-down policy, which allows a computation visible to the attacker ($\ell_{\text{L}} \sqsubseteq \ell_A$) to write to a non-visible resource ($\ell_{\text{H}} \not\sqsubseteq \ell_A$). Simulating such steps, i.e., the label-decorated version of rules [NEW, WRITE, FORK], is challenging and requires *two-steps erasure* [33], a technique that performs erasure in two-stages, by firstly rewriting the problematic constructs , such as *new*, *write* and *fork* to special constructs, i.e., *new*$_\bullet$, *write*$_\bullet$ and *fork*$_\bullet$, whose special semantics rule guarantees simulation. We remark that such special constructs are introduced due to mere technical reasons and they are not part of the plain calculus. We use *fork*$_\bullet$ as an example to illustrate this technique. Figure 18 shows rules [FORK] and [FORK$_\bullet$], that is the label annotated rules for *fork* and *fork*$_\bullet$ respectively. Rule [FORK] is similar to its annotated counterpart shown in Figure 14, save for the extra look-up and update through the thread pool map $\Phi$. Rule [FORK$_\bullet$] mimics rule [FORK] for what concerns the parent thread, but it ignores thread $t$, which is not added to the thread pool. Observe that rule [FORK] does not correctly simulate fork operations that occur in high threads. In particular, the high thread pool $T_{s2}$ is rewritten by the erasure function to $\bullet$, since $\ell_{\text{H}} \not\sqsubseteq \ell_A$, however $|\bullet| \not\equiv m$.

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