Addressing Covert Termination and Timing Channels in Concurrent Information Flow Systems

Deian Stefan[1] Alejandro Russo[2] Pablo Buiras[2] Amit Levy[1] John C. Mitchell[1] David Mazières[1]

(1) Stanford University, Stanford, CA, USA (2) Chalmers University of Technology, Gothenburg, Sweden

Abstract

When termination of a program is observable by an adversary, confidential information may be leaked by terminating accordingly. While this termination covert channel has limited bandwidth for sequential programs, it is a more dangerous source of information leakage in concurrent settings. We address concurrent termination and timing channels by presenting a dynamic information-flow control system that mitigates and eliminates these channels while allowing termination and timing to depend on secret values. Intuitively, we leverage concurrency by placing such potentially sensitive actions in separate threads. While termination and timing of these threads may expose secret values, our system requires any thread observing these properties to raise its information-flow label accordingly, preventing leaks to lower-labeled contexts. We implement this approach in a Haskell library and demonstrate its applicability by building a web server that uses information-flow control to restrict untrusted web applications.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.4.6 [*Security and Protection*]: Information flow controls

General Terms Security, Languages, Design

Keywords Monad, Library, Covert channels

1. Introduction

Covert channels arise when programming language features are misused to leak information [30]. For example, when termination of a program is observable to an adversary, a program may intentionally or accidentally communicate a confidential bit by terminating according to the value of that bit. While this termination covert channel has limited bandwidth for sequential programs, it is a significant source of information leakage in concurrent settings. Similar issues arise with covert timing channels, which are potentially widespread because so many programs involve loops or recursive functions. These channels, based on either internal observation by portions of the system or external observation, are also effective in concurrent settings.

We present an information-flow system, in a form of an execution monitor, that mitigates and eliminates termination and timing channels in concurrent systems, while allowing timing and termination of loops and recursion to depend on secret values. Because the significance of these covert channels depends on concurrency, we fight fire with fire by leveraging concurrency to mitigate these channels: we place potentially non-terminating actions, or actions whose timing may depend on secret values, in separate threads. In our system, each thread has an associated *current label* that keeps track of the sensitivity of the data it has observed and restricts the locations to which the thread can write. Hence, while termination and timing of these threads may expose secret values, our system requires any thread observing these properties to raise its information-flow label accordingly. This prevents lower security threads from observing confidential information. We implement this approach in a Haskell library and demonstrate its applicability by building a web server that applies information-flow control to untrusted web applications. One advantage of Haskell is that the Haskell type system prevents code from circumventing our dynamic information-flow tracking. Although we do not address underlying hardware issues such as cache timing, our language-level methods can be combined with hardware-level mechanisms as needed to provide comprehensive defenses against covert channels.

ICFP'12, September 9–15, 2012, Copenhagen, Denmark.

Reprinted from ICFP'12,, [Unknown Proceedings], September 9–15, 2012, Copenhagen, Denmark., pp. 201–213.

Termination covert channel Askarov et al. [2] show that, for sequential programs with outputs, leakage using the termination covert channel requires exponential time in the size of the secret. Moreover, if secrets are uniformly distributed, the attacker's advantage (after observing a polynomial amount of output) is negligible in comparison with the size of the secret. Because of this relatively low risk, accepted sequential information-flow tools, such as Jif [38] and FlowCaml [48], are only designed to address termination-insensitive noninterference. In a concurrent setting, however, the termination covert channel may be exploited more significantly [18]. We therefore focus on termination covert channels in concurrent programs and present an extension to our Haskell LIO library [52], which provides dynamic tracking of labeled values. By providing labeled explicit futures with `lFork` and `lWait`, our extension removes the termination covert channel from sequential and concurrent programs while allowing loops whose termination conditions depend on secret information.

Internal timing channel Multi-threaded programs can leak information through an *internal timing covert channel* [56] when the order of public events depends on secret data. Generally, an internal timing attack can be carried out whenever a race to acquire a shared public resource may be affected by secrets. We close this covert channel by decoupling the execution of computations that produce public events from computations that manipulate secret data. Using `lFork` and `lWait`, a computation depending on secret data proceeds in a new thread; LIO asserts that the number of instructions executed by threads producing public events does not depend on secrets. Therefore, a possible race to a shared public resource does not depend on the secret, eliminating internal timing leaks.

External timing channel External timing covert channels, which involve externally measuring the time used to complete operations that may depend on secret information, have been used in practice to leak information [6, 15] and break cryptosystems [17, 28, 57]. While several existing mechanisms mitigate external timing channels [1, 4, 19], these covert channels are not addressed by conventional information-flow tools and, in fact, most of the previous techniques for language-based information-flow control appear to have limited application. Our contribution to external timing channels is to bring the mitigation techniques from the OS community into the language-based security setting. Generalizing previous work [3], Zhang et al. [60] propose a black-box mitigation technique that we adapt to a language-based security setting. In this approach, the source of observable events is wrapped by a timing mitigator that delays output events so that they contain only a bounded amount of information. We take advantage of Haskell's ability to identify computations that produce outputs, and implement the mitigator as part of our information flow control library. Leveraging monad transformers [33], we show how to modularly extend LIO and other Haskell libraries that perform side-effects, to provide a suitable form of Zhang et al.'s mitigator.

In summary, the main contributions of this paper are:

- We present a dynamic information flow control (IFC) system that eliminates the termination and internal timing covert channels, while mitigating external timing. While these covert channels have been addressed differently in static IFC approaches, this is the first implementation of a language-based dynamic IFC system for concurrency that does not rely on cooperative-scheduling. Our system provides support for threads, lightweight synchronization primitives, and allows loops and branches to depend on sensitive values.
- We eliminate termination and internal-timing covert channels using concurrency, with potentially sensitive actions run in separate threads. This is implemented in a Haskell library that uses labeled concurrency primitives[1].
- We provide language-based support for resource-usage mitigation using monad transformers. We use this method to implement the black-box external timing mitigation approach of Zhang et al.; the method is also applicable to other covert channels, such as storage.
- We evaluate the language implementation by building a simple server-side web application framework. In this framework, untrusted applications have access to a persistent key-value store. Moreover, requests to apps may be from malicious clients colluding with the application in order to learn sensitive information. We show several potential leaks through timing and termination and show how our library is used to address them.

Section 2 provides background on information flow, Haskell, and the `LIO` monad. We describe how to address the termination covert channel in Section 3, the internal timing covert channel in Section 4, and the external timing channel in Section 5. Labeled communication primitives are detailed in Section 6. A formalization of the library is given in Section 7, with security guarantees detailed in Section 8. The implementation and experimental evaluation are presented in Section 9. Related work is described in Section 10. We conclude in Section 11.

2. Background

We build on the Haskell dynamic information flow control library called LIO [52]. This section describes LIO and some of its relevant background.

[1] The library implementations discussed in this paper can be found at `http://www.scs.stanford.edu/~deian/concurrent_lio`

2.1 Information flow control

The goal of IFC is to track and control the propagation of information. In an IFC system, every bit has an associated *label*. Labels form a lattice [9] governed by a partial order $\sqsubseteq$ pronounced "can flow to." The value of a bit labeled L_{out} can depend on a bit labeled L_{in} only if $L_{\text{in}} \sqsubseteq L_{\text{out}}$.

In a *floating-label* system, such as LIO, every execution context has a label that can rise to accommodate reading more sensitive data. For a computation P labeled L_P to observe an object labeled L_O, P's label must rise to the least upper bound or *join* of the two labels, written $L_P \sqcup L_O$. P's label effectively "floats above" the labels of all objects it observes. Furthermore, systems frequently associate a *clearance* with each execution context that bounds its label.

Specific label formats depend on the application and are not the focus of this work. Instead, we will focus on a very simple two-point lattice with labels `Low` and `High`, where `Low` $\sqsubseteq$ `High` and `High` $\not\sqsubseteq$ `Low`. We, however, note that our implementation is polymorphic in the label type and any label format that implements a few basic relations (e.g., $\sqsubseteq$, join $\sqcup$, and meet $\sqcap$) can be used when building applications. Additionally, the LIO library supports *privileges* which are used to implement decentralized information flow control as originally presented in [37]; though we do not discuss privileges in this paper, our implementation also provides privileged-aware versions of the combinators described in later sections.

2.2 Haskell

We chose Haskell because its abstractions allow IFC to be provided by a library [31]. Building a library is far simpler than developing a programming language from scratch (or heavily modifying a compiler). Moreover, a library offers backwards compatibility with a large body of existing Haskell code.

From a security point of view, Haskell's most distinctive feature is a clear separation of pure computations from those with side-effects. Any computation with side-effects must have a type encapsulated by the monad `IO`. The main idea behind the LIO library is that untrusted actions must be specified with a new Labeled I/O monad, `LIO`, instead of `IO`. Using `LIO` ensures that all computations obey information control flow.

2.3 The `LIO` monad

LIO dynamically enforces IFC, but without the features described in this paper, provides only *termination-insensitive* IFC [2] for sequential programs. At a high level, LIO provides the `LIO` monad intended to be used in place of `IO`. The library furthermore contains a collection of `LIO` actions, many of them similar to `IO` actions from standard Haskell libraries, except that the `LIO` versions contain label checks that enforce IFC. For instance, LIO provides file operations that look like those of the standard library, except that they confine the application to a dedicated portion of the file system where they store a label along with each file.

The `LIO` monad keeps a *current label*, L_{cur}, that is effectively a ceiling over the labels of all data that the current computation may depend on. `LIO` also maintains a *current clearance*, C_{cur}, which specifies an upper bound on permissible values of L_{cur}.

LIO does not individually label definitions and bindings. Rather, all symbols in scope are identically labeled with L_{cur}. The only way to observe or modify differently labeled data is to execute actions that internally access privileged symbols. Such actions are responsible for appropriately validating and adjusting the current label. As an example, the LIO file-reading function `readFile`, when executed on a file labeled L_F, first checks that $L_F \sqsubseteq C_{\text{cur}}$, throwing an exception if not. If the check succeeds, the function raises L_{cur}

to $L_{\text{cur}} \sqcup L_F$ before returning the file content. The LIO file-writing function, `writeFile`, throws an exception if $L_{\text{cur}} \not\sqsubseteq L_F$.

As previously mentioned, allowing experimentation with different label formats, LIO actions are parameterized by the label type. For instance, simplifying slightly:

```
readFile :: (Label l) ⇒ FilePath → LIO l String
```

To be more precise, it is really `(LIO l)` that is a replacement for the `IO` monad, where `l` can be any label type. The context `(Label l)⇒` in `readFile`'s type signature restricts `l` to types that are instances of the `Label` typeclass, which abstracts the label specifics behind the basic methods $\sqsubseteq$, $\sqcup$, and $\sqcap$.

2.4 Labeled values

Since LIO protects all nameable values with L_{cur}, we need a way to manipulate differently-labeled data without monotonically increasing L_{cur}. For this purpose, LIO provides explicit references to labeled, immutable data through a polymorphic data type called `Labeled`. A locally accessible symbol (at L_{cur}) can name, say, a `Labeled l Int` (for some label type `l`), which contains an `Int` protected by a different label.

Several functions allow creating and using `Labeled` values:

- `label :: (Label l)⇒l →a → LIO l (Labeled l a)`
  Given label $l : L_{\text{cur}} \sqsubseteq l \sqsubseteq C_{\text{cur}}$ and value v, action `label` l v returns a `Labeled` value guarding v with label l.
- `unlabel :: (Label l)⇒Labeled l a →LIO l a`
  If `lv` is a `Labeled` value v with label l, `unlabel lv` raises L_{cur} to $L_{\text{cur}} \sqcup l$ (provided $L_{\text{cur}} \sqcup l \sqsubseteq C_{\text{cur}}$ holds, otherwise it throws an exception) and returns v.
- `toLabeled :: (Label l)⇒l →LIO l a → LIO l (Labeled l a)`
  The dual of `unlabel`: given a label l, and an action `m` that would raise L_{cur} to L'_{cur} where $L'_{\text{cur}} \sqsubseteq l \sqsubseteq C_{\text{cur}}$, `toLabeled` l `m` executes `m` without raising L_{cur}, and instead encapsulates `m`'s result in a `Labeled` value protected by label l.
- `labelOf :: (Label l)⇒Labeled l a →l`
  Returns the label of a `Labeled` value.

As an example, we show an `LIO` action that adds two `Labeled Int`s:

```
addLIO lA lB = do a ← unlabel lA
                  b ← unlabel lB
                  return (a + b)
```

If the inputs' labels are L_A and L_B, this action raises L_{cur} to $L_A \sqcup L_B \sqcup L_{\text{cur}}$ and returns the sum of the values. To avoid raising the current label, and instead return a `Labeled Int`, `addLIO` can be wrapped by `toLabeled`:

```
add lA lB = toLabeled (labelOf lA ⊔ labelOf lB)
                      (addLIO lA lB)
```

Implicit flows in LIO We note that in an imperative language with labeled variables, dynamic labels can lead to implicit flows [10]. The canonical example is as follows:

```
public := 0;      // public has a Low label
if (secret)       // secret has a High label
  public := 1;    // public depends on secret
```

To avoid directly leaking the `secret` bit into `public`, one should track the label of the program counter and determine that execution of the assignment `public :=1` depends on `secret`, and raise `public`'s label when assigning `public :=1`. However, since the assignment executes conditionally depending on `secret`, now `public`'s *label* leaks the `secret` bit. LIO does not suffer from implicit flows. When branching on a secret, L_{cur} becomes `High` and therefore no public events are possible.

Listing 1 Exploiting the termination channel by brute-force

```
bruteForce:: String → Int → Labeled l Int → LIO l ()
bruteForce name n secret = forM_ [0..n] $ λi → do
  toLabeled High $ do
    s ← unlabel secret
    when (s == i) ⊥
  outputLow (name ++ " ≠ " ++ show i)
```

3. The termination covert channel

As mentioned in the introduction, information-flow control results and techniques for sequential settings do not naturally generalize to concurrent settings. In this section we highlight that the sequential LIO library is, like many IFC systems, susceptible to leaks due to termination and show that a naive, but typical, extension that adds concurrency drastically amplifies this leak. We present a modification to the LIO library that eliminates the termination covert channel from both sequential and concurrent programs; our solution allows for flexible programming patterns, even writing loops whose termination condition depends on secret data.

Sequential setting As described by [2], a brute-force attack, taking exponential time in the size (# of bits) of the secret, is the most effective way to exploit the termination channel in a sequential program. Listing 1 shows an implementation of such attack. Function `bruteForce` takes three arguments: a public string (helper) message, a public upper-bound on the secret, and the secret `Int` of type `Labeled Int`. Given the three arguments `bruteForce` returns an `LIO` action which when executed returns unit `()`, but produces intermediate side-effects. Specifically, `bruteForce` writes to a `Low` labeled channel using `outputLow` while L_{cur} is `Low`. We assume that the attack is executed with initial L_{cur} = `Low`, and secret `Int` labeled `High`.

The attack consists of iterating (variable `i`) over the domain of the secret (`forM_ [0..n]`), producing a publicly-observable output at every iteration until the secret is guessed. On every iteration L_{cur} is raised to the label of the `secret` within a `toLabeled` block. As described in Section 2.4, the current label outside the `toLabeled` block remains unaffected, and so the computation can continue producing publicly-observable outputs if the computation within the `toLabeled` block does not diverge. This is the case unless guess `i` is equal to the secret, at which point the computation diverges (`when (s ==i) ⊥`) and no additional publicly-observable outputs are produced. The leak due to termination is obvious: when the attacker, observing the `Low` labeled output channel, no longer receives any data, the value of the secret can be inferred given the previous outputs. For instance, to leak a 16-bit `secret`, we execute `bruteForce "secret" 65536 secret`. Assuming the value of the secret is 3, executing the action produces the outputs "`secret` $\neq 0$", "`secret` $\neq 1$", and "`secret` $\neq 2$" before diverging. The assumption here, and the rest of the paper, is that the code is untrusted (e.g., provided by the attacker) and therefore an observer that knows the implementation of `bruteForce` can directly infer that the value of the secret is 3. Observe that the code producing public outputs (`outputLow (msg ++" ≠" ++show i)`) does not inspect secret data, which makes it difficult to avoid termination leaks by simply tracking the flow of labeled data inside programs.

Concurrent setting Suppose that we (naively) add support for concurrency to LIO using a hypothetical primitive `fork`, which simply executes a given computation in a new thread. Although we can preserve termination-insensitive non-interference, we can extend the previous brute force attack to leak a secret value in linear, as opposed to exponential, time. In general, adding concurrency primitives in a straight-forward manner makes attacks that leverage the termination covert channel very effective [18]. To illustrate this point, consider the attack shown in Listing 2, which

Listing 2 A concurrent termination channel attack

```
concurrentAttack :: Int → Labeled l Int → LIO l ()
concurrentAttack k secret = forM_ [0..k] $ λi → do
  iBit ← toLabeled High $ do
           s ← unlabel secret
           return (extractBit i s) )
  fork $ bruteForce (show k ++ "-bit") 1 iBit
    where extractBit :: Int → Int → Int
          extractBit i n = (shiftR n i) .&. (bit 0)
```

leaks the bit-contents of a secret value in linear time. Given the bit-length `k` of the secret and the labeled `secret`, `concurrentAttack` returns an action which, when executed, extracts each bit of the secret (`extractBit i s`) and spawns a corresponding thread to recover the bit using the sequential brute-force attack of Listing 1 (`bruteForce (show k ++"-bit") 1 iBit`). By collecting the public outputs generated by the different threads (having the form "`0-bit` $\neq$ `0`", "`3-bit` $\neq$ `0`", "`1-bit` $\neq$ `0`", etc.), it is directly possible to recover the secret value. Observe that the divergence of one thread does not affect the termination of other threads and thus does not require observations external to the program, as in the sequential case.

3.1 Removing the termination covert channel in LIO

Since LIO has floating labels, a leak to a `Low` channel due to termination *cannot* occur after the current label is raised to `High`, unless the label raise is within an enclosed `toLabeled` computation. Hence, we can deduce that a piece of LIO code can exploit the termination covert channel only when using `toLabeled`. The key insight is that `toLabeled` is the single LIO combinator that effectively allows a piece of code to temporarily raise its current label, perform a computation, and then continue with the original current label. The attack in Listing 1 is a clear example that leverages this property of `toLabeled` to leak information.

Consider the necessary conditions for eliminating the termination channel of Listing 1: the execution of the publicly-observable `outputLow` action must not depend on the data or control flow of the secret computation executed within the `toLabeled` block. Hence, one approach to close the termination covert channel is by decoupling the execution of computations enclosed by `toLabeled`. To this end, we provide an alternative to `toLabeled` that executes computations that might raise the current label (as in `toLabeled`) in a newly-spawned thread. To observe the result (or non-termination) of such a spawned computation, the current label is firstly raised to the label of the (possibly) returned result. In doing so, after observing a secret result (or non-termination) of a spawned computation, actions that produce publicly-observable side-effects can no longer be executed. In this manner, the termination channel is closed.

In Listing 1, the execution of `outputLow` depends on the termination of the computation enclosed by `toLabeled`. However, using our proposed approach of spawning a new thread when "performing a `toLabeled`", if the code following the sensitive block wishes to observe whether or not the `High` computation has terminated, it would first need to raise the current label to `High`. Thereafter, an `outputLow` action cannot be executed regardless of the result (or termination) of the `toLabeled` computation.

Concretely, we close the termination channel by removing the insecure function `toLabeled` from `LIO` and, instead, provide the following (termination sensitive) primitives.

```
lFork :: Label l ⇒ l → LIO l a → LIO l (Result l a)
lWait :: Label l ⇒ Result l a → LIO l a
```

Intuitively, `lFork` can be considered as a concurrent version of `toLabeled`. `lFork l lio` spawns a new thread to perform the computation `lio`, whose current label may rise, and whose result is a value labeled with `l`. Rather than block, immediately after spawning a new thread, the primitive returns a value of type `Result l a`, which is simply a handler to access the labeled result produced by the spawned computation. Similar to `unlabel`, we provide `lWait`, which inspects values returned by spawned computations, i.e., values of type `Result l a`. The labeled wait, `lWait`, raises the current label to the label of its argument and then proceeds to inspect it.

Listing 3 Internal timing leak

```
doGuess secret guess cond = do
  toLabeled High $ do v ← unlabel secret
                      when (v ≠ guess) $ loopUntil cond
  outputLow (show guess)
  broadcastCondition cond

attack :: Labeled l Bool → LIO l ()
attack secret = do cond ← mkSharedCond
 {- thread 1: -}    fork $ doGuess secret True  cond
 {- thread 2: -}    fork $ doGuess secret False cond
```

In principle, rather than forking threads, it would be enough to prove that computations involving secrets terminate, e.g., by writing them in Coq or Agda. However, while this idea works in theory, it is still possible to crash an Agda or Coq program at runtime: for example, with a stack overflow. Generally, abnormal termination due to resource exhaustion exploits the termination channel just as effectively. Forking threads removes the termination channel by design. Although it might seem expensive, forking threads in Haskell is a light-weight operation [25].

We note that adding concurrency to LIO is a major modification which introduces security implications beyond that of handling the termination channel. In the following section, we describe the *internal timing covert channel*, a channel that is only present in programming languages that have support for concurrency and shared-resources.

4. The Internal timing covert channel

Multi-threaded programs, wherein threads share a common (public) resource, can leak sensitive information through the *internal timing covert channel* [56]. In an internal timing attack, sensitive data can be leaked by affecting the timing behavior of threads, which consequently alters the order of events on a shared public channel.

Listing 3 shows an example of the internal timing attack in LIO with the added `fork` primitive. Here, action `mkSharedCond` creates a shared resource that is used as a "condition variable", `loopUntil` waits until the condition is satisfied, and `broadcastCondition` sets the condition to signal all the waiting threads. The shared resource may be defined in terms of language level constructs, such as mutable references: `mkSharedCond` can create a public `Bool` reference set initially to `False`, `loopUntil` loops until the dereferenced value is `True` and `broadcastCondition` assigns `True` to it. Alternatively, it can be defined in terms of an implicit state, such as the scheduler: `mkSharedCond` and `broadcastCondition` can simply return `()`, while `loopUntil` delays the running thread for a reasonable amount of time (e.g., by using `threadDelay`).

Although in isolation both threads are secure (i.e., they satisfy non-interference), by executing them concurrently it is possible to leak information about `secret`. When executing `attack secret`, if `secret` is (labeled) `True`, thread 2 will output to the public (low) channel after thread 1 regardless of which thread is executed first. In other words, the produced output will be "`True`", "`False`". The converse holds when `secret` is `False` and the program prints out "`False`", "`True`". Notice that an attacker can infer the value of `secret` by simply observing the outputs on the public channel: the order of "`True`" and "`False`" is influenced by the secret data.

Unlike other timing channel attacks, internal timing attacks do not require a powerful attacker that must measure the execution time as to deduce secret information. The interleaving of threads can directly be used to produce leaks! Additionally, we note that although the example of Listing 3 only leaks a single bit, it is easy to construct an attack that uses a loop to leak the bit-contents of a secret value in linear time of its length. Tsai et al. [54] show such an amplified attack and demonstrate its effectiveness even in settings where little information about the run-time system (e.g., the scheduler) is available.

4.1 Removing the internal timing channel

As indicated by our example, the internal timing covert channel can be exploited when the time to produce public events (e.g., writing data to a public channel) depends on secrets. In other words, leaks due to internal timing occur when there is a race to acquire a public shared resource that may be affected by secret data. To close this channel, we apply the technique used to close the termination covert channel: we decouple the execution of computations that produce public events from computations that manipulate secret data. By using `lFork` and `lWait`, computations dealing with secrets are executed in a new thread. Consequently, any possible race to a shared public resource cannot depend on sensitive data, making leaks due to internal timing infeasible.

5. The external timing covert channel

In a real-world scenario, IFC applications interact with unlabeled, publicly observable, resources. For example, a server-side IFC web application interacts with a browser, which may itself be IFC-unaware, over a public network channel. Consequently, an adversary can take measurements *external* to the application (e.g., the application response time) from which they may infer information about confidential data computed by the web application. Although our results generalize (e.g., to the storage covert channel), in this section we address the *external timing covert channel*: an application can leak information over a channel to an observer that precisely measures message-arrival timings.

Most of the language-based IFC techniques that consider external timing channels are limited. Despite the successful use of external timing attacks to leak information in web [6, 15] and cryptographic [17, 28, 57] applications, they remain widely unaddressed by mainstream, practical IFC tools, including Jif [38]. Furthermore, most techniques that provide IFC in the presence of the external timing channel [1, 4, 19] are overly restrictive, e.g., they do not allow folding over secret data. In this work, we show a modular approach of mitigating the external timing covert channel for Haskell libraries, such as LIO.

5.1 Mitigating the external timing channel

Recently, a predictive black-box mitigation technique for external timing channels has been proposed [3, 60]. The predictive mitigation technique assumes that the attacker has control of the application (computing on secret data) and can measure the time a message is placed on a channel (e.g., when a response is sent to the browser). Treating the application as a black-box event source, a mitigator is interposed between the application and system output.

Internally, the mitigator keeps a *schedule* describing when outputs are to be produced. For example, the time mitigator might keep a schedule "predicting" that the application will produce an output every 1ms. If the application delivers events according to the schedule, or at a higher rate, the mitigator will be able to produce an output at every 1ms interval, according to the schedule, and thus leak no information.

The application may fail to deliver an event to the mitigator on time, and thus render the mitigator's schedule prediction false. At this point, the mitigator must handle the misprediction by selecting, or "predicting", a new schedule for the application. In most cases, this corresponds to doubling the application's *quantum*. For instance, following a misprediction where the quantum was 1 ms, the application will subsequently be expected to produce an output every 2 ms. It is at the point of switching schedules where an attacker learns information: rather than seeing events spaced at 1 ms intervals, the attacker now observes outputs at 2 ms intervals, indicating that the application violated the predicted behavior (a decision that can be affected by secret data). Askarov et al. [3] show that the amount of information leaked by this *slow-doubling* mitigator is polylogarithmic in the application runtime.

The aspects of the predictive mitigation technique of [3, 60] that make it particularly attractive to use in LIO are:

- The mitigator can adaptively reduce the quantum, as to increase the throughput of a well-behaved application in a manner that bounds the covert channel bandwidth (though with the leakage factor slightly larger than that of the slow-doubling mitigator);
- The mitigator can leverage public factors to decide a schedule. For example, in a web application setting where responses are mitigated, the arrival of an HTTP request can be used as a "reset" event. This is particularly useful as a quiescent application would otherwise be penalized for not producing an output according to the predicted schedule. Our web application of Section 9 implements this mitigation technique.
- The amount of information leaked is bounded by a combinatorial analysis on the number of attacker observations.

Monadic approach to black-box mitigation The functionality of different monads, such as I/O and error handling, can be combined in a modular fashion using *monad transformers* [33]. A monad transformer t, when applied to a monad m, generates a new, combined monad, $t\ m$, that shares the behavior of monad m as well as the behavior of the monad encoded in the monad transformer. The modularity of monad transformers comes from the fact that they consider the underlying monad m opaque, i.e., the behavior of the monad transformer t does not depend on the internal structure of m. In this light, we adopt Zhang et al.'s system-oriented predictive black-box mitigator to a language-based security setting in the form of a monad transformer.

5.2 Language-based mitigators

We envision the implementation of mitigators that address covert channels other than external timing. For example, we prototype a mitigator for the storage covert channel, which addresses attacks in which the message length is used to encode secret information. Hence, our mitigation monad transformer `MitM s q` is polymorphic in the mitigator-specific state `s` and quantum type `q` :

```
newtype MitM s q m a = MitM ...
```

We provide the function `evalMitM`, which takes an action of type `MitM s q m a` and returns an action of type `m a`, which when executed will mitigate the computation outputs. We note that the value constructor for the mitigation monad must not be exported to untrusted code, which can use it to circumvent the mitigation.

The time-mitigation monad transformer is a special case:

```
type TimeMitM = MitM TStamp TStampDiff
```

where the internal state `TStamp` is a time stamp, and the quantum `TStampDiff` is a time difference. Superficially, a value of type `TimeMitM m a` is a monadic computation that produces a value of type `a`. Internally, a time measurement is taken whenever an output is to be emitted in the underlying monad `m`, the internal state and quantum are adjusted to reflect the output event, and the output is delayed if it was produced ahead of the predicted schedule.

Consider, for instance, a version of `hPut` executing in the time mitigated `IO` monad, where every handle is mitigated:

```
type MIO = TimeMitM IO
...
hPut :: Handle → ByteString → MIO ()
```

If `hPut h` is invoked according to the specified schedule (e.g., at least every 1 ms), the actual IO function `IO.hPut` is used to write the provided byte-string every 1 ms. Conversely, if the function does not follow the predicted schedule, the quantum will be increased, and write-throughput to the file will decrease.

The use of a monad transformer leaves the possibility to use (almost) any underlying monad `m`, not just `IO` or `LIO`. However, this generality comes with a trade-off: either every computation `m` is mitigated, or trustworthy programmers must define *what* objects (e.g., file handles, sockets, references, etc.) they wish to mitigate and *how* to mitigate them (e.g., providing a definition for `hPut`, above). Given that the former design choice would not allow for distinguishing between inputs and outputs, we implemented the latter and more explicit mitigation approach.

To define *what* is to be mitigated, we provide the data type `data Mitigated s q a`, in terms of which a time-mitigated I/O file handle (as used in `hPut`) can simply be defined as:

```
type TimeMitigated = Mitigated TStamp TStampDiff
type Handle = TimeMitigated IO.Handle
```

`Mitigated` allows us to do mitigation at very fine granularity. Specifically, the monad transformer can be used to associate a mitigator with each `Mitigated` value (henceforth "handle"). This allows an application to write to multiple files, all of which are mitigated independently, and thus may be written to, at different rates[2]. It remains for us to address *how* the mitigators are defined.

Mitigators are implemented as instances of the type class `Mitigator`, which provides two functions:

```
class MonadConcur m ⇒ Mitigator m s q where
  -- | Create a Mitigated "handle".
  mkMitigated :: Maybe s    -- ^ Internal state
              → q           -- ^ Quantum
              → m a         -- ^ Handle constructor
              → MitM s q m (Mitigated s q a)

  -- | Mitigate an operation
  mitigate :: Mitigated s q a  -- ^ Mitigated "handle"
           → (a → m ())        -- ^ Output computation
           → MitM s q m ()
```

The context `MonadConcur m` is used to impose the requirement that the underlying monad be an `IO`-like monad which allows forking new `IO` threads (as to separate the mitigator from the computation being mitigated) and operations on synchronizing variables, MVars [25] (which are internal to the `MitM` transformer). The `mkMitigated` function is used to create a mitigated handle given an initial state, quantum, and underlying constructor. The default implementation of `mkMitigated` creates the mitigator state (internal to the transformer) corresponding to the handle. A simplified version of our `openFile` operation shows how `mkMitigated` is used:

```
openFile :: FilePath → IOMode → MIO Handle
openFile f mode = mkMitigated Nothing q $ do
  h ← IO.openFile f mode    -- Handle constructor
  return h                  -- Raw handle
    where q = mkQuant 1000 -- Initial quantum of 1ms
```

Here, the constructor `IO.openFile` creates a file handle to the file at path `f`. This constructor is supplied to `mkMitigated`, in addition to the "empty" state `Nothing`, and initial quantum `q` = 1 ms, which creates the corresponding mitigator and `Mitigated` handle (recall `Handle` is a type alias of `TimeMitigated IO.Handle`). We note that although the default definition of `mkMitigated` creates a mitigator per handle, instances may provide a definition that is more coarse-grained (e.g., associate mitigator with all handles of a thread).

Unlike for `mkMitigated`, each mitigator must define `mitigate`, which specifies how a computation should be mitigated. The function takes two arguments: the mitigated handle and a computation that produces an output on given the underlying, "raw" handle. Our time mitigator instance

```
instance ... ⇒ Mitigator m TStamp TStampDiff where
  mitigate mH act = ... -- Actual mitigation code
```

provides a definition for `mitigate`. Using `mitigate` we define our time mitigated `hPut` function as: `hPut hPut`

```
hPut :: Handle → ByteString → MIO ()
hPut mH bs = mitigate mH (λh → IO.hPut h bs)
```

The `mitigate` function first retrieves the internal state of the mitigator corresponding to the mitigated handle `mH` and forks a new thread (allowing other mitigated actions to be executed). In the new thread, a time measurement t_1 is taken. Then, if the time difference between t_1 and the internal mitigator time stamp t_0 exceeds the quantum q, the new mitigator quantum is set to $2q$; otherwise, the computation is delayed for $t_1 - t_0$ microseconds. Following, the `IO` action is executed, and the internal timestamp is replaced with the current time. We force operations on the same handle to be sequential and thus follow the latest schedule.

We finally remark that adapting an existing program to have mitigated outputs comes almost for free: a *trustworthy* programmer needs to define the constructor functions, such as `openFile`, and output functions, such as `hPut`, and simply *lift* all the remaining operations. We provide a definition for the function `lift :: Monad m ⇒m a →MitM s q m a`, which lifts a computation in the `m` monad into the mitigation monad, without performing any actual mitigation. A simple example illustrating this is the definition of `hGet` which reads a specified number of bytes from a handle:

```
hGet :: Handle → Int → TimeMitigated IO ByteString
hGet = lift ∘ IO.hGet ∘ mitVal
```

Here, `mitVal` simply returns the underlying "raw" handle.

6. Synchronization primitives in concurrent LIO

In the presence of concurrency, synchronization is vital. This section introduces an IFC-aware version of `MVars`, which are well-established synchronization Haskell primitives [25]. As with `MVars`, `LMVars` can be used in different manners: as synchronized mutable variables, as channels of depth one, or as building blocks for more complex communication and synchronization primitives.

A value of type `LMVar l a` is mutable location that is either empty or contains a value of type `a` labeled with `l`. `LMVars` are associated with the following operations:

```
newEmptyLMVar :: (Label l) ⇒ l → LIO l (LMVar l a)
putLMVar      :: (Label l) ⇒ LMVar l a → a → LIO l ()
takeLMVar     :: (Label l) ⇒ LMVar l a → LIO l a
```

Function `newEmptyLMVar` takes a label `l` and creates an empty `LMVar l a` for any desired type `a`. The creation succeeds only if the label `l` is between the current label and clearance of the `LIO` computation that creates it. Function `putLMVar` fills an `LMVar l a` with a value of type `a` if it is empty and blocks otherwise. Dually, `takeLMVar` empties an `LMVar l a` if it is full and blocks otherwise.

Note that both `takeLMVar` and `putLMVar` observe if the `LMVar` is empty in order to proceed to modify its content. Precisely,

[2] In cases where schedule mispredictions are common, it is important to implement the *l-grace* period policy of [60]. The policy states that when there are more than l mispredictions, the new scheduling should affect all mitigators.

Listing 4 Syntax for values, expressions, and types.

$$
\begin{array}{rrl}
\text{Label:} & & l \\
\text{LMVar:} & & m \\
\text{Value:} & v ::= & \texttt{true} \mid \texttt{false} \mid \texttt{()} \mid l \mid m \mid x \mid \lambda x.e \mid \texttt{fix}\ e \\
& & \mid \texttt{Lb}\ l\ e \mid (e)^{\texttt{LIO}} \mid \boxdot \mid \texttt{R}\ m \mid \bullet \\
\text{Expression:} & e ::= & v \mid e\ e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \\
& & \mid \texttt{return}\ e \mid e\ \texttt{>>=}\ e \mid \texttt{label}\ e\ e \\
& & \mid \texttt{unlabel}\ e \mid \texttt{lowerClr}\ e \mid \texttt{getLabel} \\
& & \mid \texttt{getClearance} \mid \texttt{labelOf}\ e \mid \texttt{out}\ e\ e \\
& & \mid \texttt{lFork}\ e\ e \mid \texttt{lWait}\ e \mid \texttt{newLMVar}\ e\ e \\
& & \mid \texttt{takeLMVar}\ e \mid \texttt{putLMVar}\ e\ e \\
& & \mid \texttt{labelOfLMVar}\ e \\
\text{Type:} & \tau ::= & \texttt{Bool} \mid \texttt{()} \mid \tau \to \tau \mid \ell \mid \texttt{Labeled}\ \ell\ \tau \\
& & \mid \texttt{Result}\ \ell\ \tau \mid \texttt{LMVar}\ \ell\ \tau \mid \texttt{LIO}\ \ell\ \tau
\end{array}
$$

`takeLMVar` and `putLMVar` perform a read and a write of the mutable location. Consequently, from a security point of view, operations on a given `LMVar l a` are executed only when the label `l` is below or equal to the clearance (i.e., $\texttt{l} \sqsubseteq C_{\text{cur}}$ due to the read) and above or equal to the current label (i.e., $L_{\text{cur}} \sqsubseteq \texttt{l}$ due to the write). Moreover, after either operation, L_{cur} is raised to `l`.

Many communication channels used in practice are similarly *bi-directional*, i.e., a read produces a write (and vice versa). For instance, reading a file may modify the access time in the inode; writing to a socket may produce an observable error if the connection is closed, etc. As described above, `LMVar` are bi-directional channels. If we treated them as uni-directional, observe that, a termination leak would be possible: a thread, whose current label is `Low` can use a `LMVar` labeled `Low` to send information to a computation whose current label is `High`; the `High` thread can then decide to empty the `LMVar` according to a secret value and thus leak information to the `Low` thread.

7. Formal semantics for LIO

In this section, we model our concurrent LIO implementation encompassing the concurrency primitives discussed in Sections 3, 4, and 6. We do not model the external timing mitigator since our monad transformer approach effectively treats computations as black-boxes. Thus, although our approach is more fine grained, the security guarantees of [3, 60] readily apply to our library.

We formalize our LIO library as a simply typed Curry-style call-by-name λ-calculus with some extensions. Listing 4 defines the formal syntax for the language. Syntactic categories v, e, and τ represent values, expressions, and types, respectively. Values are side-effect free while expressions denote (possible) side-effecting computations. Due to lack of space, we only show the reduction and typing rules for the core part of the library.

Values The syntax category v includes the symbol `true` and `false` representing Boolean values. Symbol `()` represents the unit value. Symbol ℓ denotes security labels. Symbol m represents `LMVars`. Values include variables (x), functions ($\lambda x.e$), and recursive functions ($\texttt{fix}\ e$). Special syntax nodes are added to this category: $\texttt{Lb}\ v\ e$, $(e)^{\texttt{LIO}}$, $\texttt{R}\ m$, $\boxdot$, and $\bullet$. Node $\texttt{Lb}\ v\ e$ denotes the run-time representation of a labeled value. Similarly, node $(e)^{\texttt{LIO}}$ denotes the run-time result of a monadic `LIO` computation. Node $\boxdot$ denotes the run-time representation of an empty `LMVar`. Node $\texttt{R}\ m$ is the run-time representation of a handle, implemented as a `LMVar`, that is used to access the result produced by spawned computations. Alternatively, $\texttt{R}\ m$ can be thought of as an explicit *future*.

Listing 5 Typing rules for special syntax nodes.

$$
\frac{}{\Gamma \vdash \bullet : \tau} \qquad \frac{}{\Gamma \vdash m : \texttt{LMVar}\ \ell\ \tau} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{Lb}\ l\ e : \texttt{Labeled}\ \ell\ \tau}
$$

$$
\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (e)^{\texttt{LIO}} : \texttt{LIO}\ \ell\ \tau} \qquad \frac{}{\Gamma \vdash \boxdot : \tau} \qquad \frac{\Gamma \vdash m : \texttt{LMVar}\ \ell\ \tau}{\Gamma \vdash \texttt{R}\ m : \texttt{Result}\ \ell\ \tau}
$$

Node $\bullet$ represents an erased term (explained in Section 8). None of these special nodes appear in programs written by users and they are merely introduced for technical reasons.

Expressions Expressions are composed of values (v), function applications ($e\ e$), conditional branches ($\texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$), and local definitions ($\texttt{let}\ x\ =\ e\ \texttt{in}\ e$). Additionally, expressions may involve operations related to monadic computations in the `LIO` monad. More precisely, `return` e and e `>>=` e represent the monadic return and bind operations. Monadic operations related to the manipulation of labeled values inside the `LIO` monad are given by `label` and `unlabel`. Expression `unlabel` e acquires the content of the labeled value e while in an `LIO` computation. Expression `label` $e_1\ e_2$ creates a labeled value, with label e_1, of the result obtained by evaluating the `LIO` computation e_2. Expression `lowerClr` e allows lowering of the current clearance to e. Expressions `getLabel` and `getClearance` return the current label and current clearance of an `LIO` computation, respectively. Expression `labelOf` e obtains the security label of labeled values. Expression `out` $e_1\ e_2$ denotes the output of e_2 to the output channel at security level e_1. For simplicity, we assume that there is only one output channel per security level. Expression `lFork` $e_1\ e_2$ spawns a thread that computes e_2 and returns a handle with label e_1. Expression `lWait` e inspects the value returned by the spawned computation whose result is accessed by the handle e. Non-proper morphisms related to creating, reading, and writing labeled `MVars` are respectively captured by expressions `newLMVar`, `takeLMVar`, and `putLMVar`.

Types We consider standard types for Booleans (`Bool`), unit (`()`), and function ($\tau \to \tau$) values. Type ℓ describes security labels. Type $\texttt{Result}\ \ell\ \tau$ denotes handles used to access labeled results produced by spawned computations, where the results are of type τ and labeled with labels of type ℓ. Type $\texttt{LMVar}\ \ell\ \tau$ describes labeled `MVars`, with labels of type ℓ and storing values of type τ. Type $\texttt{LIO}\ \ell\ \tau$ represents monadic `LIO` computations, with a result type τ and the security labels of type ℓ.

The typing judgments have the standard form $\Gamma \vdash e : \tau$, such that expression e has type τ assuming the typing environment Γ; we use Γ for both variable and store typings. Typing rules for the special syntax nodes are shown in Listing 5. These rules are liberal on purpose. Recall that special syntax nodes are run-time representations of certain values, e.g., labeled `MVars`. Thus, they are only considered in a context where it is possible to uniquely deduce their types. The typing for the remaining terms and expressions are standard and we therefore do not describe them any further. We do not require any of the commonly used extensions to Haskell's type-system, a direct consequence of the fact that security checks are performed at run-time. Since typing rules are straightforward, we assume that the type system is sound with respect to our semantics.

The `LIO` monad is essentially implemented as a State monad. To simplify the formalization and description of expressions, without loss of generality, we make the state of the monad part of the run-time environment. More precisely, each thread is accompanied by a local security run-time environment σ, which keeps track of the current label ($\sigma.\texttt{lbl}$) and clearance ($\sigma.\texttt{clr}$) of the running `LIO` computation. Common to every thread, the symbol Σ holds the global `LMVar` store ($\Sigma.\phi$) and the output channels ($\Sigma.\alpha_l$, one for

Listing 6 Semantics for non-standard expressions.

$$E \quad ::= \quad \ldots \mid \texttt{label}\ E\ e \mid \texttt{unlabel}\ E \mid \texttt{out}\ E\ e \mid \texttt{out}\ l\ E \mid \texttt{lFork}\ E\ e \mid \texttt{newLMVar}\ E\ e \mid \texttt{takeLMVar}\ E \mid \texttt{putLMVar}\ E\ e \mid \texttt{labelOfLMVar}\ E$$

(LAB)
$$\frac{\sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \sigma.\texttt{clr}}{\langle \Sigma, \langle \sigma, E[\texttt{label}\ l\ e] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma, E[\texttt{return}\ (\texttt{Lb}\ l\ e)] \rangle \rangle}$$

(UNLAB)
$$\frac{l' = \sigma.\texttt{lbl} \sqcup l \qquad l' \sqsubseteq \sigma.\texttt{clr} \qquad \sigma' = \sigma[\texttt{lbl} \mapsto l']}{\langle \Sigma, \langle \sigma, E[\texttt{unlabel}\ (\texttt{Lb}\ l\ e)] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma', E[\texttt{return}\ e] \rangle \rangle}$$

(OUTPUT)
$$\frac{\sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \sigma.\texttt{clr} \qquad \Sigma' = \Sigma[\alpha_l \mapsto \Sigma.\alpha_l \rhd \texttt{out}(v)]}{\langle \Sigma, \langle \sigma, E[\texttt{out}\ l\ v] \rangle \rangle \longrightarrow \langle \Sigma', \langle \sigma, E[\texttt{return}\ ()] \rangle \rangle}$$

(LFORK)
$$\frac{\sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \sigma.\texttt{clr} \qquad \Sigma' = \Sigma[\phi \mapsto \Sigma.\phi[m \mapsto \texttt{Lb}\ l\ \boxdot]] \qquad e' = e\ \texttt{>>=}\ \lambda x.\texttt{putLMVar}\ m\ x \qquad m \text{ fresh}}{\langle \Sigma, \langle \sigma, E[\texttt{lFork}\ l\ e] \rangle \rangle \overset{\texttt{fork}(e')}{\longrightarrow} \langle \Sigma', \langle \sigma, E[\texttt{return}\ (\texttt{R}\ m)] \rangle \rangle}$$

(LWAIT)
$$\langle \Sigma, \langle \sigma, E[\texttt{lWait}\ (\texttt{R}\ m)] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma, E[\texttt{takeLMVar}\ m] \rangle \rangle$$

(NLMVAR)
$$\frac{\sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \sigma.\texttt{clr} \qquad \Sigma' = \Sigma[\phi \mapsto \Sigma.\phi[m \mapsto \texttt{Lb}\ l\ e]] \qquad m \textit{ fresh}}{\langle \Sigma, \langle \sigma, E[\texttt{newLMVar}\ l\ e] \rangle \rangle \longrightarrow \langle \Sigma', \langle \sigma, E[\texttt{return}\ m] \rangle \rangle}$$

(TLMVAR)
$$\frac{\Sigma.\phi(m) = \texttt{Lb}\ l\ e \qquad e \neq \boxdot \qquad \sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \sigma.\texttt{clr} \qquad \sigma' = \sigma[\texttt{lbl} \mapsto \sigma.\texttt{lbl} \sqcup l] \qquad \Sigma' = \Sigma[\phi \mapsto \Sigma.\phi[m \mapsto \texttt{Lb}\ l\ \boxdot]]}{\langle \Sigma, \langle \sigma, E[\texttt{takeLMVar}\ m] \rangle \rangle \longrightarrow \langle \Sigma', \langle \sigma', E[\texttt{return}\ e] \rangle \rangle}$$

(PLMVAR)
$$\frac{\Sigma.\phi(m) = \texttt{Lb}\ l\ \boxdot \qquad \sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \sigma.\texttt{clr} \qquad \sigma' = \sigma[\texttt{lbl} \mapsto \sigma.\texttt{lbl} \sqcup l] \qquad \Sigma' = \Sigma[\phi \mapsto \Sigma.\phi[m \mapsto \texttt{Lb}\ l\ e]]}{\langle \Sigma, \langle \sigma, E[\texttt{putLMVar}\ m\ e] \rangle \rangle \longrightarrow \langle \Sigma', \langle \sigma', E[\texttt{return}\ ()] \rangle \rangle}$$

(GLABR)
$$\frac{e = \Sigma.\phi(m)}{\langle \Sigma, \langle \sigma, E[\texttt{labelOfLMVar}\ m] \rangle \rangle \longrightarrow \langle \Sigma, \langle \sigma, E[\texttt{labelOf}\ e] \rangle \rangle}$$

every security label l). A store ϕ is a mapping from LMVars to labeled values, while an output channel is a queue of events of the form out(v) (output) , for some value v. For simplicity, we assume that every store contains a mapping for every possible LMVar. The run-time environments Σ, σ, and a LIO computation form a *sequential configuration* $\langle \Sigma, \langle \sigma, e \rangle \rangle$.

The relation $\langle \Sigma, \langle \sigma, e \rangle \rangle \xrightarrow{\gamma} \langle \Sigma', \langle \sigma', e' \rangle \rangle$ represents a single evaluation step from expression e, under the run-time environments Σ and σ, to expression e' and run-time environments Σ' and σ'. We define this relation in terms of a structured operational semantics via evaluation contexts [14]. We say that e reduces to e' in one step. We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$. Symbol γ ranges over the *internal* events triggered by expressions. We utilize internal events to communicate between the threads and the scheduler. Listing 6 shows the reduction rules for the core contributions in our library. Rules (LAB) and (UNLAB) impose the same security constrains as for the sequential version of LIO [52].

Listing 7 Semantics for threadpools.

(STEP)
$$\frac{\langle \Sigma, t \rangle \longrightarrow \langle \Sigma', t' \rangle}{\langle \Sigma, t \lhd t_s \rangle \hookrightarrow \langle \Sigma', t_s \rhd t' \rangle}$$

(NO-STEP)
$$\frac{\langle \Sigma, t \rangle \not\longrightarrow \qquad t = \langle \sigma, e \rangle \qquad e \neq v}{\langle \Sigma, t \lhd t_s \rangle \hookrightarrow \langle \Sigma, t_s \rhd t \rangle}$$

(FORK)
$$\frac{\langle \Sigma, t \rangle \overset{\texttt{fork}(e)}{\longrightarrow} \langle \Sigma', \langle \sigma, e' \rangle \rangle \qquad t_{\text{new}} = \langle \sigma, e \rangle}{\langle \Sigma, t \lhd t_s \rangle \hookrightarrow \langle \Sigma', t_s \rhd \langle \sigma, e' \rangle \rhd t_{\text{new}} \rangle}$$

(EXIT)
$$\frac{l = \sigma.\texttt{lbl}}{\langle \Sigma, \langle \sigma, v \rangle \lhd t_s \rangle \hookrightarrow \langle \Sigma', t_s \rangle}$$

Rule (LAB) generates a labeled value if and only if the label is between the current label and clearance of the LIO computation. Rule (UNLAB) requires that, when the content of a labeled value is "retrieved" and used in a LIO computation, the current label is raised ($\sigma' = \sigma[\texttt{lbl} \mapsto l']$, where $l' = \sigma.\texttt{lbl} \sqcup l$), thus capturing the fact that the remaining computation might depend on e. Output channels are treated as deques of events. We use a standard deque-like interface with operations ($\lhd$) and ($\rhd$) for front and back insertion (respectively), and we also allow pattern-matching in the rules as a representation of deconstruction operations. Rule (OUTPUT) adds the event $\texttt{out}(v)$ to the end of the output channel at security level l ($\Sigma.\alpha_l \rhd \texttt{out}(v)$).

The main contributions of our language are related to the primitives for concurrency and synchronization. Rule (LFORK) allows for the creation of a thread and generates the internal event $\texttt{fork}(e')$, where e' is the computation to spawn. The rule allocates a new LMVar in order to store the result produced by the spawned thread ($e\ \texttt{>>=}\ \lambda x.\texttt{putLMVar}\ m\ x$). Using that LMVar, the rule provides a handle to access to the thread's result ($\texttt{return}\ (\texttt{R}\ m)$). Rule (LWAIT) simply uses the LMVar for the handle. Rule (TLMVAR) describes the creation of a new LMVar with a label bounded by the current label and clearance ($\sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \sigma.\texttt{clr}$). As mentioned in Section 4, operations on LMVar are *bi-directional* and consequently the rules (TLMVAR), and (PLMVAR) require not only that the label of the mentioned LMVar be between the current label and current clearance of the thread ($\sigma.\texttt{lbl} \sqsubseteq l \sqsubseteq \sigma.\texttt{clr}$), but that the current label be raised appropriately. As such, considering the security level l of a LMVar, rule (TLMVAR) raises the current label ($\sigma' = \sigma[\texttt{lbl} \mapsto \sigma.\texttt{lbl} \sqcup l]$) when emptying ($\Sigma.\phi[m \mapsto \texttt{Lb}\ l\ \boxdot]$) its content ($\Sigma.\phi(m) = \texttt{Lb}\ l\ e$). Similarly, considering the security level l of a LMVar, rule (PLMVAR) raises the current label ($\sigma' = \sigma[\texttt{lbl} \mapsto \sigma.\texttt{lbl} \sqcup l]$) when filling ($\Sigma.\phi[m \mapsto \texttt{Lb}\ l\ e]$) its content ($\Sigma.\phi(m) = \texttt{Lb}\ l\ \boxdot$). Finally, rule (GLABR) fetches a labeled LMVar from the LMVar store ($e = \Sigma.\phi(m)$, i.e., a value of the form $\texttt{Lb}\ l\ e'$), and returns its label. To simplify the formalism, insecure programs "get stuck" in their evaluation. In practice, however, an exception is raised to the most outer trusted code, which handles it in an application-specific manner, e.g., in the case of a web server, the trusted code handles such exceptions by not sending a reply to the client. In some other cases, where the user is trusted, it may be desirable to display a notification explaining the source of error.

Listing 7 shows the formal semantics for threadpools. The relation $\hookrightarrow$ represents a single evaluation step for the threadpool, in contrast with $\longrightarrow$ which is only for a single thread. We write $\hookrightarrow^*$ for the reflexive and transitive closure of $\hookrightarrow$. Configurations are of

Listing 8 Erasure function.

$$\varepsilon_L(\langle\!|\Sigma, t_s|\!\rangle) = \langle\!|\varepsilon_L(\Sigma), \text{filter } (\lambda\langle\sigma, e\rangle.e \not\equiv \bullet)\ (\text{map } \varepsilon_L\ t_s)|\!\rangle$$

$$\varepsilon_L(\langle\sigma, e\rangle) = \begin{cases} \langle\sigma, \bullet\rangle & \sigma.\texttt{lbl} \not\sqsubseteq L \\ \langle\sigma, \varepsilon_L(e)\rangle & \text{otherwise} \end{cases}$$

$$\varepsilon_L(\Sigma) = \Sigma[\phi \mapsto \varepsilon_L(\Sigma.\phi)][\alpha_l \mapsto \varepsilon_L(\alpha_l)]_{l \in \text{Labels}}$$

$$\varepsilon_L(\alpha_l) = \begin{cases} \epsilon & l \not\sqsubseteq L \\ \text{map } \varepsilon_L\ \alpha_l & \text{otherwise} \end{cases}$$

$$\varepsilon_L(\phi) = \{(x, \varepsilon_L(\phi(x))) : x \in \text{dom}(\phi)\}$$

$$\varepsilon_L(\texttt{Lb}\ l\ e) = \begin{cases} \texttt{Lb}\ l\ \bullet & l \not\sqsubseteq L \\ \texttt{Lb}\ l\ \varepsilon_L(e) & \text{otherwise} \end{cases}$$

In the rest of the cases, ε_L is homomorphic.

the form $\langle\!|\Sigma, t_s|\!\rangle$, where Σ is the global runtime environment and t_s is a queue of sequential configurations. The front of the queue is the thread that is currently executing. Threads are scheduled in a round-robin fashion. The thread at the front of the queue executes one step, and it is then moved to the back of the queue (rule (STEP)). If this step involves a fork (represented by $\xrightarrow{\text{fork}(e)}$), a new thread is created at the back of the queue (rule (FORK)). The identifier t_{new} is bound in rule (FORK), and it stands for the configuration of the newly-forked thread, i.e., $t_{\text{new}} = \langle\sigma, e\rangle$ if the parent thread transition had a label $\texttt{fork}(e)$. Threads are also moved to the back of the threadpool if they are blocked, e.g., waiting to read a value from an empty `LMVar` (rule (NO-STEP) defines $\not\longrightarrow$ as the impossibility to make any progress). When a thread finishes, i.e., it can no longer reduce, the thread is removed from the queue (rule (EXIT)).

Considering IFC for a general scheduler could lead to refinements attacks (e.g., [21, 22, 39, 50, 51, 56]) or the need to severely restrict programs (e.g., [47]). By considering a deterministic scheduler, our approach is more permissive — it rejects fewer programs — and robust against refinement attacks. We remark that it is possible to generalize our work by considering a range of deterministic schedulers (e.g., those of [40]) without drastically changing our proof technique.

8. Security guarantees

In this section, we show that LIO computations satisfy termination-sensitive non-interference. As in [32, 43, 52], we prove this property by using the *term erasure* technique. The erasure function ε_L rewrites data at security levels that the attacker cannot observe into the syntax node $\bullet$.

Listing 8 defines the erasure function ε_L. This function is defined in such a way that $\varepsilon_L(e)$ contains no information above[3] level L, i.e., the function ε_L replaces all the information more sensitive than L in e with a hole ($\bullet$). In most of the cases, the erasure function is simply applied homomorphically (e.g., $\varepsilon_L(e_1\ e_2) = \varepsilon_L(e_1)\ \varepsilon_L(e_2)$). For threadpools, the erasure function is mapped into all sequential configurations; all threads with a current label above L are removed from the pool (filter $(\lambda\langle\sigma, e\rangle.e \not\equiv \bullet)$ (map $\varepsilon_L\ t_s$), where $\equiv$ denotes syntactic equivalence). The computation performed in a certain sequential configuration is erased if the current label is above L. For runtime environments and stores, we map the erasure function into their components. An output channel is erased into the empty channel (ϵ) if it is above L, otherwise the individual output events are erased according to ε_L. A labeled value is erased if the label assigned to it is above L.

[3] We loosely use the word "above" to mean $\sqsubseteq$, since labels may not be comparable.

Following the definition of the erasure function, we introduce a new evaluation relation $\longrightarrow_L$ as follows:

$$\frac{\langle\!|\Sigma, t_s|\!\rangle \longrightarrow \langle\!|\Sigma', t'_s|\!\rangle}{\langle\!|\Sigma, t_s|\!\rangle \longrightarrow_L \varepsilon_L(\langle\!|\Sigma', t'_s|\!\rangle)}$$

The relation $\longrightarrow_L$ guarantees that confidential data, i.e., data not below level L, is erased as soon as it is created. We write $\longrightarrow_L^*$ for the reflexive and transitive closure of $\longrightarrow_L$. Similarly, we introduce a relation $\hookrightarrow_L$ as follows:

$$\frac{\langle\!|\Sigma, t_s|\!\rangle \hookrightarrow \langle\!|\Sigma', t'_s|\!\rangle}{\langle\!|\Sigma, t_s|\!\rangle \hookrightarrow_L \varepsilon_L(\langle\!|\Sigma', t'_s|\!\rangle)}$$

As usual, we write $\hookrightarrow_L^*$ for the reflexive and transitive closure of $\hookrightarrow_L$. In order to prove non-interference, we will establish a simulation relation between $\hookrightarrow^*$ and $\hookrightarrow_L^*$ through the erasure function: erasing all secret data and then taking evaluation steps in $\hookrightarrow_L$ is equivalent to taking steps in $\hookrightarrow$ first, and then erasing all secret values in the resulting configuration. Note that this relation would not hold if information from some level above L was being leaked by the program. In the rest of this section, we only consider well-typed terms to ensure there are no stuck configurations.

For simplicity, we assume that the address space of the memory store is split into different security levels and that allocation is deterministic. Therefore, the address returned when creating an LMVar with label l depends only on the LMVars with label l already in the store.

We start by showing that the evaluation relations $\longrightarrow_L$ and $\hookrightarrow_L$ are deterministic.

Proposition 1 (Determinacy of $\longrightarrow_L$)**.** *If* $\langle\!|\Sigma, t|\!\rangle \longrightarrow_L \langle\!|\Sigma', t'|\!\rangle$ *and* $\langle\!|\Sigma, t|\!\rangle \longrightarrow_L \langle\!|\Sigma'', t''|\!\rangle$*, then* $\langle\!|\Sigma', t'|\!\rangle = \langle\!|\Sigma'', t''|\!\rangle$*.*

Proof. By induction on expressions and evaluation contexts, showing there is always a unique redex in every step. □

Proposition 2 (Determinacy of $\hookrightarrow_L$)**.** *If* $\langle\!|\Sigma, t_s|\!\rangle \hookrightarrow_L \langle\!|\Sigma', t'_s|\!\rangle$ *and* $\langle\!|\Sigma, t_s|\!\rangle \hookrightarrow_L \langle\!|\Sigma'', t''_s|\!\rangle$*, then* $\langle\!|\Sigma', t'_s|\!\rangle = \langle\!|\Sigma'', t''_s|\!\rangle$*.*

Proof. By induction on expressions and evaluation contexts, showing there is a unique redex in every step and using Lemma 1. □

The next lemma establishes a simulation between $\hookrightarrow^*$ and $\hookrightarrow_L^*$.

Lemma 1 (Many-step simulation)**.** *If* $\langle\!|\Sigma, t_s|\!\rangle \hookrightarrow^* \langle\!|\Sigma', t'_s|\!\rangle$*, then* $\varepsilon_L(\langle\!|\Sigma, t_s|\!\rangle) \hookrightarrow_L^* \varepsilon_L(\langle\!|\Sigma', t'_s|\!\rangle)$*.*

Proof. In order to prove this result, we rely on properties of the erasure function, such as the fact that it is idempotent and homomorphic to the application of evaluation contexts and substitution. We show that the result holds by case analysis on the rule used to derive $\langle\!|\Sigma, t_s|\!\rangle \hookrightarrow^* \langle\!|\Sigma', t'_s|\!\rangle$, and considering different cases for threads whose current label is below (or not) level L. □

The L-equivalence relation $\approx_L$ is an equivalence relation between configurations (and their parts), defined as the equivalence kernel of the erasure function ε_L: $\langle\!|\Sigma, t_s|\!\rangle \approx_L \langle\!|\Sigma', r_s|\!\rangle$ iff $\varepsilon_L(\langle\!|\Sigma, t_s|\!\rangle) = \varepsilon_L(\langle\!|\Sigma', r_s|\!\rangle)$. If two configurations are L-equivalent, they agree on all data below or at level L, i.e., they cannot be distinguished by an attacker at level L. Note that two queues are L-equivalent iff the threads with current label that flows to L are pairwise L-equivalent in the order appearing in the queue.

The next theorem shows the non-interference property. It essentially states that if we take two executions of a program with two L-equivalent inputs, then for every intermediate step of the computation of the first run, there is a corresponding step in the computation of the second run which results in an L-equivalent configuration. Note that this also includes the termination channel, since

L-equivalence of configurations requires the same public threads to be terminated. We formulate the theorem in terms of a function since we only consider programs that receive input (represented by the argument to the function) at the beginning of their execution and then produce outputs (represented by the `out` primitive).

Theorem 1 (Termination-sensitive non-interference)**.** *Given a function e (with no `Lb`, `m`, $()^{LIO}$, $\boxdot$, `R`, and $\bullet$) where $\Gamma \vdash e :$ `Labeled` $\ell\ \tau \rightarrow$ `LIO` ℓ (`Labeled` $\ell\ \tau'$), an attacker at level L, an initial security context σ, and runtime environments Σ_1 and Σ_2 where $\Sigma_1.\phi = \Sigma_2.\phi = \emptyset$ and $\Sigma_1.\alpha_k = \Sigma_2.\alpha_k = \epsilon$ for all levels k, then*

$$\begin{aligned}&\forall e_1 e_2.(\Gamma \vdash e_i : \texttt{Labeled}\ \ell\ \tau)_{i=1,2} \wedge\ e_1 \approx_L e_2 \\ &\quad \wedge \langle \Sigma_1, \langle \sigma, e\ e_1 \rangle \rangle \hookrightarrow^* \langle \Sigma_1', t_s^1 \rangle \\ &\Rightarrow \exists \Sigma_2' t_s^2.\langle \Sigma_2, \langle \sigma, e\ e_2 \rangle \rangle \hookrightarrow^* \langle \Sigma_2', t_s^2 \rangle \wedge\ \langle \Sigma_1', t_s^1 \rangle \approx_L \langle \Sigma_2', t_s^2 \rangle\end{aligned}$$

Proof. Since e_1 and e_2 are L-equivalent and Σ_1 and Σ_2 are initially empty, the initial configurations $\langle \Sigma_1, \langle \sigma, e\ e_1 \rangle \rangle$ and $\langle \Sigma_2, \langle \sigma, e\ e_2 \rangle \rangle$ must be L-equivalent. This implies that the erased configurations $\varepsilon_L(\langle \Sigma_1, \langle \sigma, e\ e_1 \rangle \rangle)$ and $\varepsilon_L(\langle \Sigma_2, \langle \sigma, e\ e_2 \rangle \rangle)$ must be syntactically equivalent. Also, by Lemma 1 (Simulation) we have $\varepsilon_L(\langle \Sigma_1, \langle \sigma, e\ e_1 \rangle \rangle) \hookrightarrow_L^* \varepsilon_L(\langle \Sigma_1', t_s^1 \rangle)$, and by Proposition 2 (Determinacy), we can always find a reduction $\varepsilon_L(\langle \Sigma_2, \langle \sigma, e\ e_2 \rangle \rangle) \hookrightarrow_L^* \varepsilon_L(\langle \Sigma_2', t_s^2 \rangle)$ where $\varepsilon_L(\langle \Sigma_1', t_s^1 \rangle) = \varepsilon_L(\langle \Sigma_2', t_s^2 \rangle)$. By Lemma 1 again, we have $\langle \Sigma_2, \langle \sigma, e\ e_2 \rangle \rangle \hookrightarrow^* \langle \Sigma_2', t_s^2 \rangle$, and therefore $\langle \Sigma_1', t_s^1 \rangle$ and $\langle \Sigma_2', t_s^2 \rangle$ are L-equivalent. □

9. Example Application: Dating Website

We implemented the concurrency primitives discussed in Sections 3, 4, and 6 using Concurrent Haskell [25]. We rely on `forkIO` and `MVars` to implement the forking primitives, and types `Result` and `LMVar`. Similarly, we implement the time-based mitigator detailed in Section 5, and a small library that mitigates the standard I/O file handle functions. We refer the interested reader to the source code, available at `http://www.scs.stanford.edu/~deian/concurrent_lio`. In this section we evaluate the feasibility of leaking information through timing-based covert channels as well as the effectiveness of LIO in addressing these leaks.

To this end, we built a simple dating website that allows third-party developers to build applications (or apps) that interact with a common database. Our website exposes a shared key-value store to third-party apps encoding interested-in relationships. A key corresponds to a user ID and its associated value represent the users that he/she is interested in. For simplicity, we do not consider the list of users sensitive, but interested-in relationships should remain confidential. In particular, a user should be able to learn which other users are interested in them, but should not be able to learn the interested-in relationships of other users.

The website consists of two main components: 1) a trusted web server that executes apps written using LIO and 2) untrusted third-party apps that may interact with users and read and write to the database. The database is simply a list of tuples mapping keys (users) to `LMVars` storing lists of users. Apps are separated from each other by URL prefixes. For example, the URL `http://xycombinator.biz/App1` points to `App1`. Requests with a particular app's URL prefix are serviced by invoking the app's request handler in an IFC-constrained, and time-mitigated, environment. We assume a powerful, but realistic adversary. In particular, malicious app writers may themselves be users of the dating site. We stress that the considered examples discussed below were deliberately chosen to highlight a plausible attack scenario and not necessarily as a realistic example. (For a production-use system that relies on LIO, we refer the reader to `http://gitstar.com`.) We also remark that programming with the concurrent version of LIO does not impose major challenges since its interface is very similar to that of the original library [52].

Termination covert channel As detailed in Section 3, the implementation of LIO [52], with `toLabeled`, is susceptible to a termination channel attack. In the context of our dating-website, a malicious app $term$, running on behalf of an (authenticated) user a can be used to leak information on another (target) user t as follows:

- Adversary a issues a request that contains a guess that user t has an interest in g: `GET` $/term$`?target=`t`&guess=`g
- The trusted app container invokes the app $term$ and forwards the request to it.
- The app $term$ then executes the following LIO code:

```
toLabeled ⊤ $ do v ← lookupDB t
                 when (g == v) ⊥
return $ mkHtmlResp200 "Bad guess"
```

  Here, `lookupDB` t is used to perform a database lookup with key t. If g is present in the database entry, the app will not terminate, otherwise it will respond, denoting the guess was wrong.

We found the termination attack to be very effective. Specifically, we measured the time required to reconstruct a database of 10 users to be 73 seconds[4].

If `toLabeled` is prohibited and `lFork` is used instead, the termination attack cannot be mounted. This is because `lWait` first raises the label of the app request handler. An attempt to output a response to the client browser will not succeed since the current label of the handler cannot flow to the label of the client's browser. (The browser label is used to restrict apps from sending responses that the end-user, in this case a, cannot observe.) It is important to note that errors of this kind are made indistinguishable from non-terminating requests. To accomplish this, our dating site catches label violation errors and converts them to $\bot$.

Internal timing covert channel To carry out an internal timing attack, an app must execute two threads that share a common resource. Concretely, an app can use internal timing to leak information on a target user t as follows:

- Adversary a issues a request containing a guess that t is interested-in g: `GET` $/internal$`?target=`t`&guess=`g
- The trusted app container invokes the app $internal$.
- App $internal$ then executes the following LIO code:

```
varHigh ← fork $
  toLabeled ⊤ $ do
    v ← lookupDB t
    when (g == v) (sleep 5000)
  appendToAppStorage g
varLow ← fork $ do sleep 3000
                   appendToAppStore -1
wait varHigh
wait varLow
r ← readFromAppStore
return $ mkHtmlResp200 r
```

The code spawns two threads. The first reads the high value in a `toLabeled`, sleeps for 5 seconds if the guess is correct, and then write the guess to a `Low`-labeled persistent store[5] The second thread simply write a placeholder (`-1`) after waiting for 3 seconds. Here, the ordering of the data in the store reveals whether the guess is correct. If the guess is incorrect, the store will read `g,-1`; if the guess is correct, the store will read `-1,g`.

[4] All our measurements were conducted on a laptop with a Intel Core i7 2620M (2.7GHz) processor and 8GB of RAM, with GHC 7.4.1.

[5] Apps can write to the database on behalf of invoking user, we use the store notion for simplicity.

We implemented a magnified version of the attack above by sending several requests to the server. The adversary repeatedly sends requests to $internal$ for each user in the system as a guess g. As with the termination channel attack, we found that internal timing attack is feasible. For a database of 10 users we managed to recover all the database entries in 66.92 seconds.

Our modifications to LIO can be used to address the internal timing attacks described above; replacing `toLabeled` with `lFork` eliminates the internal timing leaks. We observe that by using `lFork`, the time when the app executes `appendToAppStore` cannot be influenced by sensitive data. Hence, replacing `fork` and `wait` by their LIO counterparts renders the attack futile.

External timing covert channel We consider a simple external timing attack to our dating website in which the adversary a has access to a high-precision timer. An app $external$ colluding with a can use external timing to leak a target user t's interested-in relationship as follows:

- Authenticated adversary a issues requests containing the target user t: `GET` $/external$`?target=`t`&guess=`g
- The trusted container invokes $external$ with the request.
- App $external$ then proceeds to execute the following LIO code:

```
toLabeled ⊤ $ do
  v ← lookupDB t
  when (g == v) (sleep 5000)
return $ mkHtmlResp200 "done"
```

Given a target t and guess g, if the g is correct the thread sleeps; otherwise it does nothing. In both cases the final response is public. The attacker thus simply measures the response time – recognizing a delay as a correct guess. Despite its simplicity, we also found this attack to be effective. In 33 seconds, we recovered a database of 10 users. To address the leak, we mitigated the app handler, as described in Section 5. Concretely, the response time of an app was mitigated, taking into account the arrival of a request. Although we manged to recover 3 of the 10 user entries in 64 seconds—we found that recovering the remaining user entries was infeasible. The performance of well-behaved apps is unaffected.

10. Related Work

IFC security libraries The seminal work by Li and Zdancewic [31] presents an implementation of information-flow security as a Haskell library using arrows. Russo et al. [43] show a similar IFC security library based solely on monads, that library leverages Haskell's type-system to statically enforce non-interference. Tsai et al. [54] extend [31] by considering side-effects and concurrency. Different from our approach, they provide termination-insensitive non-interference under a cooperative scheduler and no synchronization primitives. Jaskelioff and Russo [24] propose a library that enforces non-interference by executing the program as many times as security levels, known as secure multi-execution [11]. More recently, we propose the use of the `LIO` monad to track information-flow dynamically [52]. Morgenstern et al. [36] encoded an authorization- and IFC-aware programming language in Agda. Their encoding, however, does not consider computations with side-effects. Devriese and Piessens [12] used monad transformers and parametrized monads to enforce non-interference, both dynamically and statically. None of the above approaches handle the termination covert channel. Moreover, except for [54] they do not consider a concurrent language.

Internal timing covert channel In addressing the internal timing cover channel, compared to this work, other language-based approaches sacrifice standard semantics, practical enforcement, permissiveness, and language expressiveness. The works [49–51, 56] rely on an unrealistic primitive `protect(c)` which hides the timing behavior of a command `c`. However, assuming a scenario where it is possible to modify the scheduler, [5, 40] show how an interaction between threads and the scheduler can be used to implement a generalized version of `protect(c)`. To close internal timing leaks, the work in [4] makes the unlikely assumption that rolling back a transaction takes the same time as committing it. In contrast, our `forkLIO` and `waitLIO` are implemented using standard concurrency primitives available in Haskell.

Low-determinism [58] states that public outputs must be deterministic such that no race on public data is possible. This concept inherently makes enforcement mechanisms non-compositional (e.g., two parallel threads that only write to a public channel is considered insecure). A model-checking and type-system approach to enforcing low-determinism have been presented in [23], and [53], respectively. Mantel et al. [34] use synchronization barriers after branching on secret data and before producing public outputs. Different from these enforcement techniques, our library scales to a large number of threads.

With respect to permissiveness, some works do not allow publicly-observable events after branching on secrets. Specifically [7, 8] avoid internal timing leaks by disallowing public events after branching on secret data. They consider a fixed number of threads and no synchronization primitives. Conversely, we allow spawning arbitrary threads that branch (or loop) on secrets while the program continues producing public events. Several approaches consider a restrictive language where dynamic thread creation is not allowed [7, 8, 16, 50, 51, 56, 58].

Russo and Sabelfeld [41] remove internal timing leaks under a cooperative scheduling by manipulating `yield` commands. However, the termination channel is intrinsically present under cooperative scheduling. Closer to our approach, Russo et al. [42] transform sequential programs into concurrent programs that spawn new threads when executing branches and loops on secret values. Although the idea of spawning threads for sensitive computations is similar, we use the approach in a different context. Firstly, Russo et al. apply their technique for a simple sequential While-language, while we consider concurrent programs with synchronization primitives in the context of a practical language. Secondly, and different from our work, their approach does not consider leaks due to termination, i.e., their transformation only guarantees termination-insensitive non-interference. Finally, their transformation approach is conservative in preserving security and, as such, the termination behavior of a transformed program may change. Our proposal, on the other hand, guarantees that the semantics of the program is that which the programmer writes.

Termination and external covert channels There are several language-based mechanisms for addressing the termination and external timing channels. Smith and Volpano [51, 55] describe a type-system that removes the termination channel by forbidding loops whose conditional depend on secrets. This restriction is also used in [34, 47]. The work by Hedin and Sands [19] avoids the termination and external timing covert channels for sequential Java bytecode by disallowing outputs after branching on secrets. This is similar to our approach; however, we allow the spawning new threads for such sensitive tasks, while the rest of the program can still perform public events. Agat [1] describes a code transformation that removes external timing leaks by padding programs with dummy computations, and avoids the termination channel by disallowing loops on secrets. One drawback of Agat's transformation is that if there is a conditional on secret data, and only one of the branches is non-terminating the transformed program is non-terminating. Despite this, the approach has been adapted for languages with concurrency [44, 45, 47]. Moreover, the transformation has been rephrased as a unification problem [29] and implemented with transactions [4]. While targeting sequential programs, secure

multi-execution [11] removes both the termination and external timing channels. However, the latter is only closed in a special configuration, e.g., if there are as many CPUs (or cores) as security levels. We refer the reader to the systematization of knowledge paper [26] for a more detailed description of possible enforcements for timing- and termination-sensitive non-interference.

Recently, Zhang et al. [61] propose a language-based mitigation approach for a simple While-language extended with a `mitigate` primitive. Their work relies on static annotations to provide information about the underlying hardware. Compared to their work, our functional approach is more general and can be extended to address other covert channels (e.g., storage). However, their attack model is more powerful in considering the effects of hardware, including caches. Nevertheless, we find their work to be complimentary: our system can leverage static annotations and the Xeon "no-fill" mode to address attacks relying on underlying hardware.

Secure operating systems and the termination channel A number of operating systems have been developed that intentionally left termination channels out of a belief that closing them was intractable, e.g., IX [35]. Another is Asbestos, whose limited their security ambitions to ensuring "that at least two cooperating processes are required to communicate information in violation of a label policy" [13]. In their seminal paper on the decentralized label model [37], which revived the operating system community's interest in information flow control, Myers and Liskov expressed skepticism the problem could ever be overcome with purely dynamic checks. HiStar [59] avoided hard-coding termination channels into the operating system. In practice, however, privileged software had to implement them anyway explicitly using privileges through untainting gates, because operating-system-level resource management requires knowing when a process has exited. By contrast, we believe that we have found abstractions that are both practical on their own, and sound with respect to non-interference.

π-calculus and information-flow Honda et al. [22] present a sophisticated type-system that addresses internal and termination covert channels in π-calculus. They classify channels into two types: *truly linear*, used exactly once, and *non-linear (non-deterministic)*, used an arbitrary number of times. The type-system allows public outputs after reading from linear channels but prevents a process from sending public outputs after receiving secret values on a non-linear channel. Without this restriction, a termination leak might occur because data might never arrive on the (non-linear) channel. The typing judgements guarantee that for every sender on a linear channel there is a corresponding receiver. Since it is not possible to have two processes writing to a common public linear channel, leaks dues to internal timing are not possible. Our library relies on essentially the same mechanism Honda et al. use to prevent leaks associated with non-linear channels. However, our approach enforces IFC dynamically rather than statically. Our systems are incomparable: we are more permissive in taking the dynamic approach [46], while Honda et al. are more permissive by allowing, in certain situations, outputs on public channels after inspecting secret data. Subsequent work [21] describes a more advanced type-system that utilizes a different classification for channels but imposes restrictions similar to [22]. Focusing on simplicity, Pottier [39] describes a type-system that disallows public outputs after reading secrets from a channel, similar to the restriction imposed by non-linear channels described above. Our work can be understood as a dynamic analog to [39] and [20]. Kobayashi [27] addresses and eliminates the termination and internal timing covert channels in addition to improving the precision of the type-system described in [21] to allow synchronization locks (similar to MVars). The semantics formulation in [27] is different from this and other related work [20–22, 39].

11. Summary

Many information flow control systems allow applications to sequence code with publicly visible side-effects after code that computes over sensitive data. Unfortunately, such sequencing leaks sensitive data through termination channels (which affect whether the public side-effects ever happen), internal timing channels (which affect the order of publicly visible side-effects), and external timing channels (which affect the response time of visible side-effects). Such leaks are far worse in the presence of concurrency, particularly when untrusted code can spawn new threads.

We demonstrate that such sequencing can be avoided by introducing additional concurrency when public values must reference the results of computations over sensitive data. We implemented this idea in an existing Haskell information flow library, LIO. In addition, we show how our library is amenable to mitigating external timing attacks by quantizing the appearance of externally visible side-effects. To evaluate our ideas, we prototyped the core of a dating web site showing that our interfaces are practical and our implementation does indeed mitigate these covert channels.

Acknowledgments We thank the anonymous reviewers for insightful comments and bringing several references to our attention. This work was funded by DARPA CRASH under contract #N66001-10-2-4088, by multiple gifts from Google, and by the Swedish research agencies VR and STINT. D. Stefan is supported by the DoD through the NDSEG Fellowship Program.

References

[1] J. Agat. Transforming out timing leaks. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 40–53, Jan. 2000.

[2] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proc. of the 13th ESORICS*. Springer-Verlag, 2008.

[3] A. Askarov, D. Zhang, and A. C. Myers. Predictive black-box mitigation of timing channels. In *Proc. of the 17th ACM CCS*. ACM, 2010.

[4] G. Barthe, T. Rezk, and M. Warnier. Preventing timing leaks through transactional branching instructions. *Electron. Notes Theor. Comput. Sci.*, 153, May 2006.

[5] G. Barthe, T. Rezk, A. Russo, and A. Sabelfeld. Security of multithreaded programs by compilation. In *Proc. European Symp. on Research in Computer Security*, pages 2–18, Sept. 2007.

[6] A. Bortz and D. Boneh. Exposing private information by timing web applications. In *Proc. of the 16th World Wide Web*. ACM, 2007.

[7] Boudol and Castellani. Noninterference for concurrent programs. In *Proc. ICALP'01*, volume 2076 of *LNCS*. Springer-Verlag, July 2001.

[8] G. Boudol and I. Castellani. Non-interference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1), June 2002.

[9] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[10] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[11] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *Proc. of the 2010 IEEE Symposium on Security and Privacy*, SP '10. IEEE Computer Society, 2010.

[12] D. Devriese and F. Piessens. Information flow enforcement in monadic libraries. In *Proc. of the 7th ACM SIGPLAN Workshop on Types in Language Design and Implementation*. ACM, 2011.

[13] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the asbestos operating system. In *Proc. of the twentieth ACM symp. on Operating systems principles*, SOSP '05. ACM, 2005.

[14] M. Felleisen. The theory and practice of first-class prompts. In *Proc. of the 15th ACM SIGPLAN-SIGACT Symp. on Principles of programming languages*, pages 180–190. ACM, 1988.

[15] E. W. Felten and M. A. Schneider. Timing attacks on web privacy. In *Proc. of the 7th ACM conference on Computer and communications security*, CCS '00. ACM, 2000.

[16] G. L. Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *Proc. of the 20th IEEE Computer Security Foundations Symposium*, CSF '07. IEEE Computer Society, 2007.

[17] H. Handschuh and H. M. Heys. A timing attack on RC5. In *Proc. of the Selected Areas in Cryptography*. Springer-Verlag, 1999.

[18] D. Hedin and A. Sabelfeld. A perspective on information-flow control. In *Proc. of the 2011 Marktoberdorf Summer School*. IOS Press, 2011.

[19] D. Hedin and D. Sands. Timing aware information flow security for a javacard-like bytecode. *Elec. Notes Theor. Comput. Sci.*, 141, 2005.

[20] M. Hennessy and J. Riely. Information flow vs. resource access in the asynchronous pi-calculus. *ACM Trans. Program. Lang. Syst.*, 24(5), Sept. 2002.

[21] K. Honda and N. Yoshida. A uniform type structure for secure information flow. *ACM Trans. Program. Lang. Syst.*, Oct. 2007.

[22] K. Honda, V. T. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In *Proc. of the 9th European Symposium on Programming Languages and Systems*. Springer-Verlag, 2000.

[23] M. Huisman, P. Worah, and K. Sunesen. A temporal logic characterisation of observational determinism. In *Proc. IEEE Computer Sec. Foundations Workshop*, July 2006.

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

[25] S. P. Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proc. of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1996.

[26] V. Kashyap, B. Wiedermann, and B. Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *Proc. of IEEE Symposium on Sec. and Privacy*. IEEE, 2011.

[27] N. Kobayashi. Type-based information flow analysis for the π-calculus. *Acta Inf.*, 42(4), Dec. 2005.

[28] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proc. of the 16th CRYPTO*. Springer-Verlag, 1996.

[29] B. Köpf and H. Mantel. Eliminating implicit information leaks by transformational typing and unification. In *Formal Aspects in Security and Trust, Third International Workshop (FAST'05)*, volume 3866 of *LNCS*. Springer-Verlag, July 2006.

[30] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[31] P. Li and S. Zdancewic. Encoding Information Flow in Haskell. In *CSFW '06: Proc. of the 19th IEEE Workshop on Computer Security Foundations*. IEEE Computer Society, 2006.

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

[33] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *In Proc. of the 22nd ACM Symposium on Principles of Programming Languages. ACMPress*, 1995.

[34] H. Mantel, H. Sudbrock, and T. Krausser. Combining different proof techniques for verifying information flow security. In *Proc. of the 16th international conference on Logic-based program synthesis and transformation*, LOPSTR'06. Springer-Verlag, 2007.

[35] M. D. Mcilroy and J. A. Reeds. Multilevel security in the unix tradition. *SoftwarePractice and Experience*, 22:673–694, 1992.

[36] J. Morgenstern and D. R. Licata. Security-typed programming within dependently typed programming. In *Proc. of the 15th ACM SIGPLAN International Conference on Functional Programming*. ACM, 2010.

[37] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, pages 129–142, 1997.

[38] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java information flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.

[39] F. Pottier. A simple view of type-secure information flow in the π-calculus. In *In Proc. of the 15th IEEE Computer Security Foundations Workshop*, pages 320–330, 2002.

[40] A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *Proc. IEEE Computer Sec. Foundations Workshop*, pages 177–189, July 2006.

[41] A. Russo and A. Sabelfeld. Security for multithreaded programs under cooperative scheduling. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics (PSI)*, LNCS. Springer-Verlag, June 2006.

[42] A. Russo, J. Hughes, D. Naumann, and A. Sabelfeld. Closing internal timing channels by transformation. In *Proc. of Asian Computing Science Conference*, LNCS. Springer-Verlag, Dec. 2006.

[43] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell. In *Proc. ACM SIGPLAN Symposium on Haskell*, pages 13–24. ACM Press, Sept. 2008.

[44] A. Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, volume 2244 of *LNCS*, pages 225–239. Springer-Verlag, July 2001.

[45] A. Sabelfeld and H. Mantel. Static confidentiality enforcement for distributed programs. In *Proc. Symp. on Static Analysis*, volume 2477 of *LNCS*, pages 376–394. Springer-Verlag, Sept. 2002.

[46] A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*, LNCS. Springer-Verlag, June 2009.

[47] A. Sabelfeld and D. Sands. Probabilistic noninterference for multithreaded programs. In *Proc. IEEE Computer Sec. Foundations Workshop*, pages 200–214, July 2000.

[48] V. Simonet. The Flow Caml system. Software release at `http://cristal.inria.fr/~simonet/soft/flowcaml/`, July 2003.

[49] Smith. Probabilistic noninterference through weak probabilistic bisimulation. In *Proc. IEEE Computer Sec. Foundations Workshop*, pages 3–13, 2003.

[50] G. Smith. A new type system for secure information flow. In *Proc. IEEE Computer Sec. Foundations Workshop*, June 2001.

[51] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 355–364, Jan. 1998.

[52] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Haskell Symposium*. ACM SIGPLAN, September 2011.

[53] T. Terauchi. A type system for observational determinism. In *Proc. of the 2008 21st IEEE Computer Security Foundations Symposium*, pages 287–300. IEEE Computer Society, 2008.

[54] T. C. Tsai, A. Russo, and J. Hughes. A library for secure multithreaded information flow in Haskell. In *Proc. IEEE Computer Sec. Foundations Symposium*, July 2007.

[55] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *Proc. of the 10th IEEE workshop on Computer Security Foundations*, CSFW '97. IEEE Computer Society, 1997.

[56] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *J. Computer Security*, 7(2–3), Nov. 1999.

[57] W. H. Wong. Timing attacks on RSA: revealing your secrets through the fourth dimension. *Crossroads*, 11, May 2005.

[58] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *Proc. IEEE Computer Sec. Foundations Workshop*, pages 29–43, June 2003.

[59] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *Proc. of the 7th USENIX Symp. on Operating Systems Design and Implementation*. USENIX, 2006.

[60] D. Zhang, A. Askarov, and A. C. Myers. Predictive mitigation of timing channels in interactive systems. In *Proc. of the 18th ACM CCS*. ACM, 2011.

[61] D. Zhang, A. Askarov, and A. C. Myers. Language-based control and mitigation of timing channels. In *Proc. of PLDI*. ACM, 2012.