A Library for Removing Cache-based Attacks in Concurrent Information Flow Systems
Extended Version

Pablo Buiras[1], Amit Levy[2], Deian Stefan[2], Alejandro Russo[1], and David Mazières[2]

[1] Chalmers University of Technology
[2] Stanford University

Abstract. Information-flow control (IFC) is a security mechanism conceived to allow untrusted code to manipulate sensitive data without compromising confidentiality. Unfortunately, untrusted code might exploit some covert channels in order to reveal information. In this paper, we focus on the LIO concurrent IFC system. By leveraging the effects of hardware caches (e.g., the CPU cache), LIO is susceptible to attacks that leak information through the *internal timing covert channel*. We present a *resumption*-based approach to address such attacks. Resumptions provide fine-grained control over the interleaving of thread computations at the library level. Specifically, we remove cache-based attacks by enforcing that every thread yield after executing an "instruction," i.e., atomic action. Importantly, our library allows for porting the full LIO library—our resumption approach handles local state and exceptions, both features present in LIO. To amend for performance degradations due to the library-level thread scheduling, we provides two novel primitives. First, we supply a primitive for securely executing pure code in parallel. Second, we provide developers a primitive for controlling the granularity of "instructions"; this allows developers to adjust the frequency of context switching to suit application demands.

1 Introduction

Popular website platforms, such as Facebook, run third-party applications (apps) to enhance the user experience. Unfortunately, in most of today's platforms, once an app is installed it is usually granted full or partial access to the user's sensitive data—the users have no guarantees that their data is not arbitrarily ex-filtrated once apps are granted access to it [18]. As demonstrated by Hails [9], information-flow control (IFC) addresses many of these limitations by restricting how sensitive data is disseminated. While promising, IFC systems are not impervious to attacks; the presence of *covert channels* allows attackers to leak sensitive information.

Covert channels are mediums not intended for communication, which nevertheless can be used to carry and, thus, reveal information [19]. In this work, we focus on the *internal timing covert channel* [33]. This channel emanates from the mere presence of concurrency and shared resources. A system is said to have an internal timing covert channel when an attacker, as to reveal sensitive data, can alter *the order of public events* by affecting the timing behavior of threads. To avoid such attacks, several authors propose decoupling computations manipulating sensitive data from those writing into public resources (e.g., [4, 5, 27, 30, 35]).

Decoupling computations by security levels only works when all shared resources are modeled. Similar to most IFC systems, the concurrent IFC system LIO [35] only models shared resources at the programming language level and does not explicitly consider the effects of hardware. As shown in [37], LIO threads can exploit the underlying CPU cache to leak information through the internal timing covert channel.

We propose using *resumptions* to model interleaved computations. (We refer the interested reader to [10] for an excellent survey of resumptions.) A resumption is either a (computed) value or an atomic action which, when executed, returns a new resumption. By expressing thread computations as a series of resumptions, we can leverage resumptions for controlling concurrency. Specifically, we can interleave atomic actions, or "instructions," from different threads, effectively forcing each thread to yield at deterministic points. This ensures that scheduling is not influenced by underlying caches and thus cannot be used to leak secret data. We address the attacks on the recent version of LIO [35] by implementing a Haskell library which ports the LIO API to use resumptions. Since LIO threads possess local state and handle exceptions, we extend resumptions to account for these features.

In principle, it is possible to force deterministic interleaving by means other than resumptions; in [37] we show an instruction-based scheduler that achieves this goal. However, Haskell's monad abstraction allows us to to easily model resumptions as a library. This has two consequences. First, and different from [37], it allows us to deploy a version of LIO that does not rely on changes to the Haskell compiler. Importantly, LIO's concurrency primitives can be modularly redefined, with little effort, to operate on resumptions. Second, by effectively implementing "instruction based-scheduling" at the level of library primitives, we can address cache attacks not covered by the approach described in [37] (see Section 5).

In practice, a library-level interleaved model of computations imposes performance penalties. With this in mind, we provide primitives that allow developers to execute code in parallel, and means for securely controlling the granularity of atomic actions (which directly affects performance).

Although our approach addresses internal timing attacks in the presence of shared hardware, the library suffers from leaks that exploit the termination channel, i.e., programs can leak information by not terminating. However, this channel can only be exploited by brute-force attacks that leak data external to the program—an attacker cannot leak data within the program, as can be done with the internal timing covert channel.

2 Cache Attacks on Concurrent IFC Systems

Figure 1 shows an attack that leverages the timing effects of the underlying cache in order to leak information through the internal timing covert channel. In isolation, all three threads are secure. However, when executed concurrently, threads B and C race to write to a public, shared variable `l`. Importantly, the race outcome depends on the state of the secret variable `h`, by changing the contents of underlying CPU cache according to its value (e.g., by creating and traversing a large array as to fill the cache with new data).

The attack proceeds as follows. First, thread A fills the cache with the contents of a public array `lowArray`. Then, depending on the secret variable `h`, it evicts data from the cache (by filling it with arbitrary data) or leaves it intact. Concurrently, public threads B and C delay execution long enough for A to finish. Subsequently, thread B accesses elements of the public array `lowArray`, and writes 0 to public variable `l`; if the array has been evicted from the cache (`h==0`), the amount of time it takes to perform the read, and thus the write to `l`, will be much longer than if the array is still in the cache. Hence, to leak the value of `h`, thread C simply needs to delay writing 1 to `l` long enough so that it is above the case where the cache is full (with the public array), but shorter than it take to refill the cache with the (public) array. Observing the contents of `l`, the attacker directly learns the value of `h`.

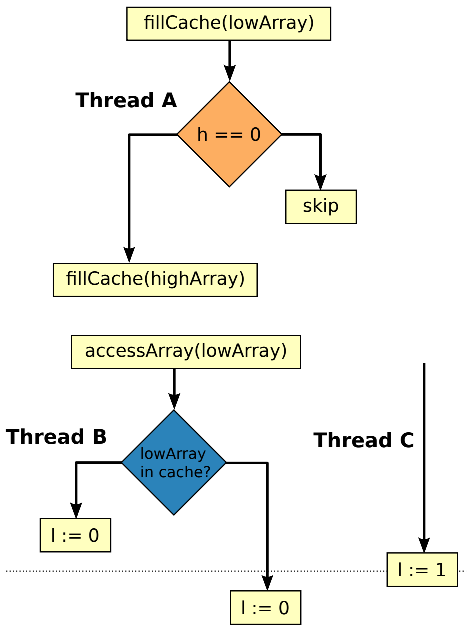

Fig. 1. Cache attack

Appendix A shows the concrete code of the attack for LIO. This simple attack has previously been demonstrated in [37], where confidential data from the GitStar system [9], build atop LIO, was leaked. Such attacks are not limited to LIO or IFC systems; cache-based attacks against many system, including cryptographic primitives (e.g., RSA and AES), are well known [1, 23, 26, 40].

The next section details the use of resumptions in modeling concurrency at the programming language level by defining atomic steps, which are used as the thread scheduling quantum unit. By scheduling threads according to the number of executed atoms, the attack in Figure 1 is eliminated. As in [37], this is the case because an atomic step runs till completion, regardless of the state of the cache. Hence, the timing behavior of thread B, which was previously leaked to thread C by the time of preemption, is no longer disclosed. Specifically, the scheduling of thread C's `l:=1` does not depend on the time it takes thread B to

```
data Thread m a where
  Done :: a → Thread m a
  Atom :: m (Thread m a) → Thread m a
  Fork  :: Thread m () → Thread m a
           → Thread m a
```

Fig. 2. Threads as Resumptions

```
sch :: [Thread m ()] → m ()
sch []                  = return ()
sch ((Done _) : thrds)  = sch thrds
sch ((Atom m) : thrds) =
  do res ← m; sch (thrds ++ [res])
sch ((Fork res res′) : thrds) =
  sch ((res : thrds) ++ [res′])
```

Fig. 3. Simple round-robin scheduler

read the public array from the cache; rather it depends on the atomic actions, which do not depend on the cache state. In addition, our use of resumptions also eliminates attacks that exploit other timing perturbations produced by the underlying hardware, e.g., TLB misses, CPU bus contention, etc.

3 Modeling Concurrency with Resumptions

In pure functional languages, computations with side-effects are encoded as values of abstract data types called *monads* [22]. We use the type $m\ a$ to denote computations that produce results of type a and may perform side-effects in monad m. Different side-effects are often handled by different monads. In Haskell, there are monads for performing inputs and outputs (monad *IO*), handling errors (monad *Error*), etc. The IFC system LIO simply exposes a monad, *LIO*, in which security checks are performed before any IO side-effecting action.

Resumptions are a simple approach to modeling interleaved computations of concurrent programs. A resumption, which has the form $res\ ::=\ x \mid \alpha\ \triangleright res$, is either a computed value x or an atomic action α followed by a new resumption res. Using this notion, we can break down a program that is composed of a series of instructions into a program that executes an atomic action and yields control to a scheduler by giving it its subsequent resumption. For example, program $P := i_1; i_2; i_3$, which performs three side-effecting instructions in sequence, can be written as $res_P := i_1; i_2 \triangleright i_3 \triangleright ()$, where $()$ is a value of a type with just one element, known as *unit*. Here, an atomic action α is any sequence of instructions. When executing res_P, instructions i_1 and i_2 execute atomically, after which it yields control back to the scheduler by supplying it the resumption $res'_P := i_3 \triangleright ()$. At this point, the scheduler may schedule atomic actions from other threads or execute res'_P to resume the execution of P. Suppose program $Q := j_1; j_2$, rewritten as $j_1 \triangleright j_2 \triangleright ()$, runs concurrently with P. Our concurrent execution of P and Q can be modeled with resumptions, under a round-robin scheduler, by writing it as $P||Q := i_1; i_2 \triangleright j_1 \triangleright i_3 \triangleright j_2 \triangleright () \triangleright ()$. In other words, resumptions allow us to implement a scheduler that executes $i_1; i_2$, postponing the execution of i_3, and executing atomic actions from Q in the interim.

Implementing threads as resumptions As previously done in [10, 11], Fig. 2 defines threads as resumptions at the programming language level. The thread type ($\mathit{Thread}\ m\ a$) is parametric in the resumption computation value type (a) and the monad in which atomic actions execute (m)[3]. (Symbol :: introduces type declarations and $\rightarrow$ denotes function types.) The definition has several value constructors for a thread. Constructor *Done* captures computed values; a value $\mathit{Done}\ a$ represents the computed value a. Constructor *Atom* captures a resumption of the form $\alpha \triangleright \mathit{res}$. Specifically, *Atom* takes a monadic action of type $m\ (\mathit{Thread}\ m\ a)$, which denotes an atomic computation in monad m that returns a new resumption as a result. In other words, *Atom* captures both the atomic action that is being executed (α) and the subsequent resumption (res). Finally, constructor *Fork* captures the action of spawning new threads; value $\mathit{Fork}\ \mathit{res}\ \mathit{res}'$ encodes a computation wherein a new thread runs resumption res and the original thread continues as res'.[4] As in the standard Haskell libraries, we assume that a fork does not return the new thread's final value and thus the type of the new thread/resumption is simply $\mathit{Thread}\ m\ ()$.

Programming with resumptions Users do not build programs based on resumptions by directly using the constructors of $\mathit{Thread}\ m\ a$. Instead, they use the interface provided by Haskell monads: $\mathit{return} :: a \rightarrow \mathit{Thread}\ m\ a$ and $(\ggg\!=) :: \mathit{Thread}\ m\ a \rightarrow (a \rightarrow \mathit{Thread}\ m\ b) \rightarrow \mathit{Thread}\ m\ b$. The expression $\mathit{return}\ a$ creates a resumption which consists of the computed value a, i.e., it corresponds to $\mathit{Done}\ a$. The operator $(\ggg\!=)$, called *bind*, is used to sequence atomic computations. Specifically, the expression $\mathit{res} \ggg\!= f$ returns a resumption that consists of the execution of the atomic actions in res followed by the atomic actions obtained from applying f to the result produced by res. (The precise definition of *return* and $\ggg\!=$ can be found in Appendix B.) We sometimes use Haskell's **do**-notation to write such monadic computations. For example, the expression $\mathit{res} \ggg\!= (\lambda a \rightarrow \mathit{return}\ (a+1))$, i.e., actions described by the resumption res followed by $\mathit{return}\ (a+1)$ where a is the result produced by res, is written as **do** $a \leftarrow \mathit{res}; \mathit{return}\ (a+1)$.

Scheduling computations We use round-robin to schedule atomic actions of different threads. Fig. 3 shows our scheduler implemented as a function from a list of threads into an interleaved computation in the monad m. The scheduler behaves as follows. If there is an empty list of resumptions, the scheduler, and thus the program, terminates. If the resumption at the head of the list is a computed value ($\mathit{Done}\ \_$), the scheduler removes it and continues scheduling the remaining threads (**sch** *thrds*). (Recall that we are primarily concerned with the side-effects produced by threads and not about their final values.) When the head of the list is an atomic step ($\mathit{Atom}\ m$), **sch** runs it ($\mathit{res} \leftarrow m$), takes the resulting resumption (res), and appends it to the end of the thread list (**sch** $(\mathit{thrds} +\!\!+ [\mathit{res}])$). Finally, when a thread is forked, i.e., the head of the list is a $\mathit{Fork}\ \mathit{res}\ \mathit{res}'$, the

[3] In our implementation, atomic actions α (as referred as in $\alpha \triangleright \mathit{res}$) are actions described by the monad m.

[4] Spawning threads could also be represented by a equivalent constructor $\mathit{Fork}' :: \mathit{Thread}\ m\ () \rightarrow \mathit{Thread}\ m\ a$, we choose *Fork* for pedagogical reasons.

spawned resumption is placed at the front of the list ($res : thrds$). Observe that in both of the latter cases the scheduler is invoked recursively—hence we keep evaluating the program until there are no more threads to schedule. We note that although we choose a particular, simple scheduling approach, our results naturally extend for a wide class of deterministic schedulers [28, 38].

4 Extending Resumptions with State and Exceptions

LIO provides general programming language abstrations (e.g., state and exceptions), which our library must preserve to retain expressiveness. To this end, we extend the notion of resumptions and modify the scheduler to handle thread local state and exceptions.

Thread local state As described in [34], the *LIO* monad keeps track of a *current label*, L_{cur}. This label is an upper bound on the labels of all data in lexical scope. When a computation C, with current label L_C, observes an object labeled L_O, C's label is raised to the least upper bound or *join* of the two labels, written $L_C \sqcup L_O$. Importantly, the current label governs where the current computation can write, what labels may be used when creating new channels or threads, etc. For example, after reading an object O, the computation should not be able to write to a channel K if L_O is more confidential than L_K—this would potentially leak sensitive information (about O) into a less sensitive channel. We write $L_C \sqsubseteq L_K$ when L_K at least as confidential as L_C and information is allowed to flow from the computation to the channel.

```
sch ((Atom m) : thrds) =
  do res ← m
     st  ← get
     sch (thrds ++ [put st ≻ res])
sch ((Fork res res′) : thrds) =
  do st ← get
     sch ((res : thrds) ++ [put st ≻ res′])
```

Fig. 4. Context-switch of local state

Using our resumption definition of Section 3, we can model concurrent LIO programs as values of type *Thread LIO*. Unfortunately, such programs are overly restrictive—since LIO threads would be sharing a single current label—and do not allow for the implementation of many important applications. Instead, and as done in the concurrent version of LIO [35], we track the state of each thread, independently, by modifying resumptions, and the scheduler, with the ability to context-switch threads with state.

Figure 4 shows these changes to **sch**. The context-switching mechanism relies on the fact that monad m is a state monad, i.e., provides operations to retrieve (*get*) and set (*put*) its state. *LIO* is a state monad,[5] where the state contains

[5] For simplicity of exposition, we use *get* and *set*. However, LIO only provides such functions to trusted code. In fact, the monad *LIO* is not an instance of *MonadState* since this would allow untrusted code to arbitrarily modify the current label—a clear security violation.

(among other things) L_{cur}. Operation $(\succ) :: m\ b \to Thread\ m\ a \to Thread\ m\ a$ modifies a resumption in such a way that its first atomic step (*Atom*) is extended with $m\ b$ as the first action. Here, *Atom* consists of executing the atomic step ($res \leftarrow m$), taking a snapshot of the state ($st \leftarrow get$), and restoring it when executing the thread again ($put\ st \succ res$). Similarly, the case for *Fork* saves the state before creating the child thread and restores it when the parent thread executes again ($put\ st \succ res'$).

Exception handling As described in [36], LIO provides a secure way to throw and catch exceptions—a feature crucial to many real-world applications. Unfortunately, simply using LIO's *throw* and *catch* as atomic actions, as in the case of local state, results in non-standard behavior. In particular, in the interleaved computation produced by **sch**, an atomic action from a thread may throw an exception that would propagate outside the thread group and crash the program. Since we do not consider leaks due to termination, this does not impact security; however, it would have non-standard and restricted semantics. Hence, we first extend our scheduler to introduce a top-level *catch* for every spawned thread.

Besides such an extension, our approach still remains quite limiting. Specifically, LIO's *catch* is defined at the level of the monad *LIO*, i.e., it can only be used inside atomic steps. Therefore, catch-blocks are prevented from being extended beyond atomic actions. To address this limitation, we lift exception handling to work at the level of resumptions.

To this end, we consider a monad m that handles exceptions, i.e., a monad for which $throw :: e \to m\ a$ and $catch :: m\ a \to (e \to m\ a) \to m\ a$, where e is a type denoting exceptions, are accordingly defined. Function *throw* throws the exception supplied as an argument. Function *catch* runs the action supplied as the first argument ($m\ a$), and if an exception is thrown, then executes the handler ($e \to m\ a$) with the value of the exception passed as an argument. If no exceptions are raised, the result of the computation (of type a) is simply returned.

```
throw e = Atom (LIO.throw e)
catch (Done a) _ = Done a
catch (Atom a) handler =
  Atom (LIO.catch
    (do res ← a
          return (catch res handler))
    (λe → return (handler e)))
catch (Fork res res') handler =
  Fork res (catch res' handler)
```

Fig. 5. Exception handling for resumptions

Figure 5 shows the definition of exception handling for resumptions. Since *LIO* defines *throw* and *catch* [36], we qualify these underlying functions with *LIO* to distinguish them from our resumption-level *throw* and *catch*. When throwing an exception, the resumption simply executes an atomic step that throws the exception in *LIO* (*LIO.throw e*).

The definitions of *catch* for *Done* and *Fork* are self explanatory. The most interesting case for *catch* is when the resumption is an *Atom*. Here, *catch* applies *LIO.catch* step by step to each atomic action in the sequence; this is necessary because exceptions can only be caught in the *LIO* monad. As shown in Fig. 5, if no exception is thrown, we simply return the resumption produced by m. Conversely, if an exception is raised, *LIO.catch* will trigger the exception handler which will return a resumption by applying the top-level *handler* to the exception e. To clarify, consider catching an exception in the resumption $\alpha_1 \triangleright \alpha_2 \triangleright x$. Here, *catch* executes α_1 as the first atomic step, and if no exception is raised, it executes α_2 as the next atomic step; on the other hand, if an exception is raised, the resumption $\alpha_2 \triangleright x$ is discarded and *catch*, instead, executes the resumption produced when applying the exception handler to the exception.

5 Performance Tuning

Unsurprisingly, interleaving computations at the library-level introduces performance degradation. To alleviate this, we provide primitives that allow developers to control the granularity of atomic steps—fine-grained atoms allow for more flexible programs, but also lead to more context switches and thus performance degradation (as we spend more time context switching). Additionally, we provide a primitive for the parallel execution of pure code. We describe these features—which do not affect our security guarantees—below.

Granularity of atomic steps To decrease the frequency of context switches, programmers can treat a complex set of atoms (which are composed using monadic bind) as a single atom using $singleAtom :: Thread\ m\ a \rightarrow Thread\ m\ a$. (See Appendix C.) This function takes a resumption and "compresses" all its atomic steps into one. Although *singleAtom* may seem unsafe, e.g., because we do not restrict threads from adjust the granularity of atomic steps according to secrets, in Section 6 we show that this is not the case—it is the atomic execution of atoms, regardless of their granularity, that ensures security.

Parallelism As in [37], we cannot run one scheduler **sch** per core to gain performance through parallelism. Threads running in parallel can still race to public resources, and thus vulnerable to internal timing attacks (that may, for example, rely on the L3 CPU cache). In principle, it is possible to securely parallelize arbitrary side-effecting computations if races (or their outcomes) to shared public resource are eliminated. Similar to *observational low-determinism* [41], our library could allow parallel computations to compute on disjoint portions of the memory. However, whenever side-effecting computations follow parallel code, we would need to impose synchronization barriers to enforce that all side-effects are performed in a pre-determined order. It is precisely this order, and LIO's safe side-effecting primitives for shared-resources, that hides the outcome of any potential dangerous parallel race. In this paper, we focus on executing pure code in parallel; we leave side-effecting code to future work.

Pure computations, by definition, cannot introduce races to shared resources since they do not produce side effects.[6] To consider such computations, we simply

[6] In the case of Haskell, lazy evaluation may pose a challenge since whether or not a thunk has been evaluate is indeed an effect on a cache [24]. Though our resumption-

extend the definition of *Thread* with a new constructor: $Parallel :: pure\ b \rightarrow (b \rightarrow Thread\ m\ a) \rightarrow Thread\ m\ a$. Here, *pure* is a monad that characterizes pure expressions, providing the primitive $runPure :: pure\ b \rightarrow b$ to obtain the value denoted by the code given as argument. The monad *pure* could be instantiated to *Par*, a monad that parallelizes pure computations in Haskell [21], with *runPure* set to *runPar*. In a resumption, *Parallel p f* specifies that *p* is to be executed in a separate Haskell thread—potentially running on a different core than the interleaved computation. Once *p* produces a value *x*, *f* is applied to *x* to produce the next resumption to execute.

```
sch (Parallel p f : thrds) =
  do res ← sync (λv → putMVar v (runPure p))
                (λv → takeMVar v)
                f
     sch (thrds ++ [res])
```

Fig. 6. Scheduler for parallel computations

Figure 6 defines **sch** for pure computations, where interaction between resumptions and Haskell-threads gets regulated. The scheduler relies on well-established synchronization primitives called MVars [13]. A value of type *MVar* is a mutable location that is either empty or contains a value. Function *putMVar* fills the MVar with a value if it is empty and blocks otherwise. Dually, *takeMVar* empties an MVar if it is full and returns the value; otherwise it blocks. Our scheduler implementation **sch** simply takes the resumption produced by the *sync* function and schedules it at the end of the thread pool. Function *sync*, internally creates a fresh MVar *v* and spawns a new Haskell-thread to execute *putMVar v* (*runPure p*). This action will store the result of the parallel computation in the provided MVar. Subsequently, *sync* returns the resumption *res*, whose first atomic action is to read the parallel computation's result from the MVar (*takeMVar v*). At the time of reading, if a value is not yet ready, the atomic action will block the whole interleaved computation. However, once a value *x* is produced (in the separate thread), *f* is applied to it and the execution proceeds with the produced resumption (*f x*).

6 Soundness

In this section, we extend the previous formalization of LIO [34] to model the semantics of our concurrency library. We present the syntax extensions that we require to model the behavior of the *Thread* monad:

Expression: $e ::= \ \dots \ \mid \ \mathbf{sch}\ e_s \ \mid \ Atom\ e \ \mid \ Done\ e \ \mid \ Fork\ e\ e \ \mid \ Parallel\ e\ e$

where e_s is a list of expressions. For brevity, we omit a full presentation of the syntax and semantics, since we rely on previous results in order to prove the security property of our approach. The interested reader is referred to [6].

Expressions are the usual λ-calculus expressions with special syntax for monadic effects and LIO operations. The syntax node **sch** e_s denotes the scheduler

based approach handles this for the single-core case, handling this in general is part of our ongoing work.

(Done)
$$\frac{}{\langle \Sigma, \mathbf{sch}\ (Done\ x : t_s)\rangle \longrightarrow \langle \Sigma, \mathbf{sch}\ t_s\rangle}$$

(Atom)
$$\frac{\langle \Sigma, m\rangle \longrightarrow^{*} \langle \Sigma', (e)^{\text{LIO}}\rangle}{\langle \Sigma, \mathbf{sch}\ (Atom\ (put\ \Sigma.\texttt{lbl} \gg m) : t_s)\rangle \longrightarrow \langle \Sigma', \mathbf{sch}\ (t_s +\!\!+ [put\ \Sigma.\texttt{lbl} \succ e])\rangle}$$

(Fork)
$$\frac{}{\langle \Sigma, \mathbf{sch}\ (Fork\ m_1\ m_2 : t_s)\rangle \longrightarrow \langle \Sigma, \mathbf{sch}\ ((m_1 : t_s) +\!\!+ [put\ \Sigma.\texttt{lbl} \succ m_2])\rangle}$$

Fig. 7. Semantics for **sch** expressions.

(Seq)
$$\frac{\langle \Sigma, e\rangle \longrightarrow \langle \Sigma', e'\rangle \qquad P \Rightarrow P'}{\langle \Sigma, e \parallel P\rangle \hookrightarrow \langle \Sigma', e' \parallel P'\rangle}$$

(Pure)
$$\frac{P \Rightarrow P' \qquad v_s \text{ fresh MVar} \qquad s = \Sigma.\texttt{lbl}}{\begin{array}{c}\langle \Sigma, \mathbf{sch}\ (Parallel\ p\ f : t_s) \parallel P\rangle \hookrightarrow \\ \langle \Sigma, \mathbf{sch}\ (t_s +\!\!+ [Atom\ (takeMVar\ v_s \ggg\!= f)]) \parallel P' \parallel (putMVar\ v_s\ (runPure\ p))_s\rangle\end{array}}$$

(Sync)
$$\frac{P \Rightarrow P'}{\begin{array}{c}\langle \Sigma, \mathbf{sch}\ (Atom\ (takeMVar\ v_s \ggg\!= f) : t_s) \parallel (putMVar\ v_s\ x)_s \parallel P\rangle \hookrightarrow \\ \langle \Sigma, \mathbf{sch}\ (f\ x : t_s) \parallel P'\rangle\end{array}}$$

Fig. 8. Semantics for **sch** expressions with parallel processes.

running with the list of threads e_s as its thread pool. The nodes $Atom\ e$, $Done\ e$, $Fork\ e\ e$ and $Parallel\ e\ e$ correspond to the constructors of the *Thread* data type. In what follows, we will use metavariables x, m, p, t, v and f for different kinds of expressions, namely values, monadic computations, pure computations, threads, MVars and functions, respectively.

We consider a global environment Σ which contains the current label of the computation ($\Sigma.\texttt{lbl}$), and also represents the resources shared among all threads, such as mutable references. We start from the one-step reduction relation[7] $\langle \Sigma, e\rangle \longrightarrow \langle \Sigma', e'\rangle$, which has already been defined for LIO [34]. This relation represents a single evaluation step from e to e', with Σ as the initial environment and Σ' as the final one. Presented as an extension to the $\longrightarrow$ relation, Figure 7 shows the reduction rules for concurrent execution using **sch**. The configurations for this relation are of the form $\langle \Sigma, \mathbf{sch}\ t_s\rangle$, where Σ is a runtime environment and t_s is a list of *Thread* computations. Note that the computation in an *Atom* always begins with either $put\ \Sigma.\texttt{lbl}$ for some label $\Sigma.\texttt{lbl}$, or with

[7] As in [35], we consider a version of $\longrightarrow$ which does not include the operation *toLabeled*, since it is susceptible to internal timing attacks.

takeMVar v for some MVar v. Rules (Done), (Atom), and (Fork) basically behave like the corresponding equations in the definition of **sch** (see Figures 3 and 4). In rule (Atom), the syntax node $(e)^{\text{LIO}}$ represents an LIO computation that has produced expression e as its result. Although **sch** applications should expand to their definitions, for brevity we show the unfolding of the resulting expressions into the next recursive call. This unfolding follows from repeated application of basic λ-calculus reductions.

Figure 8 extends relation $\longrightarrow$ into $\hookrightarrow$ to express pure parallel computations. The configurations for this relation are of the form $\langle \Sigma, \mathbf{sch}\ t_s \parallel P \rangle$, where P is an abstract process representing a pure computation that is performed in parallel. These abstract processes would be reified as native Haskell threads. The operator $(\parallel)$, representing parallel process composition, is commutative and associative.

As described in the previous section, when a *Thread* evaluates a *Parallel* computation, a new native Haskell thread should be spawned in order to run it. Rule (Pure) captures this intuition. A fresh MVar v_s (where s is the current label) is used for synchronization between the parent and the spawned thread. A process is denoted by *putMVar* v_s followed by a pure expression, and it is also tagged with the security level of the thread that spawned it.

Pure processes are evaluated in parallel with the main threads managed by **sch**. The relation $\Rightarrow$ nondeterministically evaluates one process in a parallel composition and is defined as follows.

$$\frac{\mathit{runPure}\ p \longrightarrow^{*} x}{(\mathit{putMVar}\ v_s\ (\mathit{runPure}\ p))_s \parallel P \Rightarrow (\mathit{putMVar}\ v_s\ x)_s \parallel P}$$

For simplicity, we consider the full evaluation of one process until it yields a value as just one step, since the computations involved are pure and therefore cannot leak data. Rule (Seq) in Figure 8 represents steps where no parallel forking or synchronization is performed, so it executes one $\longrightarrow$ step alongside a $\Rightarrow$ step.

Rule (Sync) models the synchronization barrier technique from Section 5. When an *Atom* of the form $(\mathit{takeMVar}\ v_s \gg\!\!= f)$ is evaluated, execution blocks until the pure process with the corresponding MVar v_s completes its computation. After that, the process is removed and the scheduler resumes execution.

Security guarantees We show that programs written using our library satisfy termination-insensitive non-interference, i.e., an attacker at level L cannot distinguish the results of programs that run with indistinguishable inputs (see Appendix D for more details) . This result has been previously established for the sequential version of LIO [34]. As in [20, 31, 34], we prove this property by using the *term erasure* technique.

In this proof technique, we define function ε_L in such a way that $\varepsilon_L(e)$ contains only information below or equal to level L, i.e., the function ε_L replaces all the information more sensitive than L or incomparable to L in e with a hole $(\bullet)$. We adapt the previous definition of ε_L to handle the new constructs in the library. 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 **sch** expressions, the erasure function is mapped into the list; all threads with a current label above L are removed from the pool (*filter* $(\not\equiv \bullet)$ $(map\ \varepsilon_L\ t_s)$), where $\equiv$ denotes syntactic equivalence). Analogously, erasure for a parallel composition consists of removing all processes using an MVar tagged with a level not strictly below or equal to L. The computation performed in a certain *Atom* is erased if the label is not strictly below or equal than L. This is given by

$$\varepsilon_L(Atom\ (put\ s \gg m)) = \begin{cases} \bullet & ,\ s \not\sqsubseteq L \\ put\ s \gg \varepsilon_L\ (m) & ,\ \text{otherwise} \end{cases}$$

A similar rule exists for expressions of the form $Atom\ (takeMVar\ v_s \ggg\!= f)$. Note that this relies on the fact that an atom must be of the form $Atom\ (put\ s \gg m)$ or $Atom\ (takeMVar\ v_s \ggg\!= f)$ by construction. For expressions of the form $Parallel\ p\ f$, erasure behaves homomorphically, i.e. $\varepsilon_L(Parallel\ p\ f) = Parallel\ \varepsilon_L(p)\ (\varepsilon_L \circ f)$.

Following the definition of the erasure function, we introduce the evaluation relation $\hookrightarrow_L$ as follows: $\langle \Sigma, \mathbf{sch}\ t_s \parallel P\rangle \hookrightarrow_L \varepsilon_L(\langle \Sigma', \mathbf{sch}\ t'_s \parallel P'\rangle)$ if $\langle \Sigma, \mathbf{sch}\ t_s \parallel P\rangle \hookrightarrow \langle \Sigma', \mathbf{sch}\ t'_s \parallel P'\rangle$. The relation $\hookrightarrow_L$ guarantees that confidential data, i.e., data not below or equal-to level L, is erased as soon as it is created. 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. In the rest of this section, we consider well-typed terms to avoid stuck configurations.

Proposition 1 (Many-step simulation). *If* $\langle \Sigma, \mathbf{sch}\ t_s \parallel P\rangle \hookrightarrow^* \langle \Sigma', \mathbf{sch}\ t'_s \parallel P'\rangle$, *then it holds that* $\varepsilon_L(\langle \Sigma, \mathbf{sch}\ t_s \parallel P\rangle) \hookrightarrow_L^* \varepsilon_L(\langle \Sigma', \mathbf{sch}\ t'_s \parallel P'\rangle)$.

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: $\langle \Sigma, \mathbf{sch}\ t_s \parallel P\rangle \approx_L \langle \Sigma', \mathbf{sch}\ r_s \parallel Q\rangle$ iff $\varepsilon_L(\langle \Sigma, \mathbf{sch}\ t_s \parallel P\rangle) = \varepsilon_L(\langle \Sigma', \mathbf{sch}\ r_s \parallel Q\rangle)$. If two configurations are L-equivalent, they agree on all data below or at level L, i.e., an attacker at level L is not able to distinguish them.

The next theorem shows the non-interference property. The configuration $\langle \Sigma, \mathbf{sch}\ []\rangle$ represents a final configuration, where the thread pool is empty and there are no more threads to run.

Theorem 1 (Termination-insensitive non-interference). *Given a computation e, inputs e_1 and e_2, an attacker at level L, runtime environments Σ_1 and Σ_2, then for all inputs e_1, e_2 such that $e_1 \approx_L e_2$, if* $\langle \Sigma_1, \mathbf{sch}\ [e\ e_1]\rangle \hookrightarrow^* \langle \Sigma'_1, \mathbf{sch}\ []\rangle$ *and* $\langle \Sigma_2, \mathbf{sch}\ [e\ e_2]\rangle \hookrightarrow^* \langle \Sigma'_2, \mathbf{sch}\ []\rangle$, *then* $\langle \Sigma'_1, \mathbf{sch}\ []\rangle \approx_L \langle \Sigma'_2, \mathbf{sch}\ []\rangle$.

This theorem essentially states that if we take two executions from configurations $\langle \Sigma_1, \mathbf{sch}\ [e\ e_1]\rangle$ and $\langle \Sigma_2, \mathbf{sch}\ [e\ e_2]\rangle$, which are indistinguishable to an attacker at level L ($e_1 \approx_L e_2$), then the final configurations for the executions $\langle \Sigma'_1, \mathbf{sch}\ []\rangle$ and $\langle \Sigma'_2, \mathbf{sch}\ []\rangle$ are also indistinguishable to the attacker ($\langle \Sigma'_1, \mathbf{sch}\ []\rangle \approx_L \langle \Sigma'_2, \mathbf{sch}\ []\rangle$). This result generalizes when constructors *Done*, *Atom*, and *Fork* involve exception handling (see Figure 5). The reason for this lies in the fact that *catch* and *throw* defer all exception handling to *LIO.throw* and *LIO.catch*, which have been proved secure in [36].

7 Case study: Classifying location data

We evaluated the trade-offs between performance, expressiveness and security through an LIO case study. We implemented an untrusted application that performs K-means clustering on sensitive user location data, in order to classify GPS-enabled cell phone into locations on a map, e.g., home, work, gym, etc. Importantly, this app is untrusted yet computes clusters for users without leaking their location (e.g., the fact that Alice frequents the local chapter of the Rebel Alliance). K-means is a particularly interesting application for evaluating our scheduler as the classification phase is highly parallelizable—each data point can be evaluated independently.

We implemented and benchmarked three versions of this app: (i) A baseline implementation that does not use our scheduler and parallelizes the computation using Haskell's *Par* Monad [21]. Since in this implementation, the scheduler is not modeled using resumptions, it leverages the parallelism features of *Par*. (ii) An implementation in the resumption based scheduler, but pinned to a single core (therefore not taking advantage of parallelizing pure computations). (iii) A parallel implementation using the resumption-based scheduler. This implementation expresses the exact same computation as the first one, but is not vulnerable to cache-based leaks, even in the face of parallel execution on multiple cores.

We ran each implementation against one month of randomly generated data, where data points are collected each minute (so, 43200 data points in total). All experiments were run ten times on a machine with two 4-core (with hyperthreading) 2.4Ghz Intel Xeon processors and 48GB of RAM. The secure, but non-parallel implementation using resumptions performed extremely poorly. With mean 204.55 seconds (standard deviation 7.19 seconds), it performed over eight times slower than the baseline at 17.17 seconds (standard deviation 1.16 seconds). This was expected since K-means is highly parallelizable. Conversely, the parallel implementation in the resumption based scheduler performed more comparably to the baseline, at 17.83 seconds (standard deviation 1.15 seconds).

To state any conclusive facts on the overhead introduce by our library, it is necessary to perform a more exhaustive analysis involving more than a single case study.

8 Related work

Cryptosystems Attacks exploiting the CPU cache have been considered by the cryptographic community [16]. Our attacker model is weaker than the one typically considered in cryptosystems, i.e., attackers with access to a stopwatch. As a countermeasure, several authors propose partitioning the cache (e.g., [25]), which often requires special hardware. Other countermeasures (e.g. [23]) are mainly implementation-specific and, while applicable to cryptographic primitives, they do not easily generalize to arbitrary code (as required in our scenario).

Resumptions While CPS can be used to model concurrency in a functional setting [7], resumptions are often simpler to reason about when considering security guarantees [10, 11]. The closest related work is that of Harrison and Hook [11]; inspired by a secure multi-level operating system, the authors utilize resumptions to model interleaving and layered state monads to represent threads. Every layer corresponds to an individual thread, thereby providing a notion of local state. Since we do not require such generality, we simply adapt the scheduler to context-switch the local state underlying the *LIO* monad. We believe that authors overlooked the power of resumptions to deal with timing perturbations produced by the underlying hardware. In [10], Harrison hints that resumptions could handle exceptions; in this work, we consummate his claim by describing precicely how to implement *throw* and *catch*.

Language-based IFC There is been considerable amount of literature on applying programming languages techniques to address the internal timing covert channel (e.g. [28, 33, 35, 39, 41]). Many of these works assume that the execution of a single step, i.e., a reduction step in some transition system, is performed in a single unit of time. This assumption is often made so that security guarantees can be easily shown using programming language semantics. Unfortunately, the presence of the CPU cache (or other hardware shared state) breaks this correspondence, making cache attacks viable. Our resumption approach establishes a correspondence between atomic steps at the implementation-level and reduction step in a transition system. Previous approaches can leverage this technique when implementing systems, as to avoid the reappearance of the internal timing channel.

Agat [2] presents a code transformation for sequential programs such that both code paths of a branch have the same memory access pattern. This transformation has been adapted in different works (e.g., [32]). Agat's approach, however, focuses on avoiding attacks relying on the data cache, while leaving the instruction cache unattended.

Russo and Sabelfeld [29] consider non-interference for concurrent while-like-programs under cooperative and deterministic scheduling. Similar to our work, this approach eliminates cache-attacks by restricting the use of yields. Differently, our library targets a richer programming languages, i.e., it supports parallelism, exceptions, and dynamically adjusting the granularity of atomic actions.

Secure multi-execution [8] preserves confidentiality of data by executing the same sequential program several times, one for each security level. In this scenario, cache-based attacks can only be removed in specific configurations [14] (e.g., when there are as many CPU cores as security levels).

Hedin and Sands [12] present a type-system for preventing external timing attacks for bytecode. Their semantics is augmented to incorporate history, which enables the modeling of cache effects. Zhang et al. [42] provide a method for mitigating external events when their timing behavior could be affected by the underlying hardware. Their semantics focusses on sequential programs, wherein attacks due to the cache arise in the form of externally visible events. Their solution is directly applicable to our system when considering external events.

System security In order to achieve strong isolation, Barthe et al. [3] present a model of virtualization which flushes the cache upon switching between guest operating systems. Flushing the cache in such scenarios is common and does not impact the already-costly context-switch. Although this technique addresses attacks that leverage the CPU cache, it does not address the case where a shared resource cannot be controlled (e.g., CPU bus).

Allowing some information leakage, Kopft et al. [17] combines abstract interpretation and quantitative information-flow to analyze leakage bounds for cache attacks. Kim et al. [15] propose StealthMem, a system level protection against cache attacks. StealthMem allows programs to allocate memory that does not get evicted from the cache. StealthMem is capable of enforcing confidentiality for a stronger attacker model than ours, i.e., they consider programs with access to a stopwatch and running on multiple cores. However, we suspect that StealthMem is not adequate for scenarios with arbitrarily complex security lattices, wherein not flushing the cache would be overly restricting.

9 Conclusion

We present a library for LIO that leverages resumptions to expose concurrency. Our resumption-based approach and "instruction"- or atom-based scheduling removes internal timing leaks induced by timing perturbations of the underlying hardware. We extend the notion of resumptions to support state and exceptions and provide a scheduler that context-switches programs with such features. Though our approach eliminates internal-timing attacks that leverage hardware caches, library-level threading imposes considerable performance penalties. Addressing this, we provide programmers with a safe mean for controlling the context-switching frequency, i.e., allowing for the adjustment of the "size" of atomic actions. Moreover, we provide a primitive for spawning computations in parallel, a novel feature not previously available in IFC tools. We prove soundness of our approach and implement a simple case study to demonstrate its use. Our techniques can be adapted to other Haskell-like IFC systems beyond LIO. The library, case study, and details of the proofs can be found at [6].

Acknowledgments We would like to thank Josef Svenningsson and our colleagues in the ProSec and Functional Programming group at Chalmers for useful comments. This work was supported by the Swedish research agency VR, STINT, the Barbro Osher foundation, DARPA CRASH under contract #N66001-10-2-4088, and multiple gifts from Google. Deian Stefan is supported by the DoD through the NDSEG Fellowship Program.

Bibliography

[1] O. Aciiçmez. Yet another microarchitectural attack:: exploiting I-cache. In *Proceedings of the 2007 ACM workshop on Computer security architecture*, CSAW '07. ACM, 2007.

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

[3] G. Barthe, G. Betarte, J. Campo, and C. Luna. Cache-leakage resilient OS isolation in an idealized model of virtualization. In *Proc. IEEE Computer Sec. Foundations Symposium*. IEEE Computer Society, june 2012.

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

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

[6] P. Buiras, A. Levy, D. Stefan, A. Russo, and D. Mazières. A library for removing cache-based attacks in concurrent information flow systems: Extended version. `http://www.cse.chalmers.se/~buiras/resLIO.html`, 2013.

[7] K. Claessen. A poor man's concurrency monad. *J. Funct. Program.*, May 1999.

[8] 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.

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

[10] B. Harrison. Cheap (but functional) threads. *J. of Functional Programming*, 2004.

[11] W. L. Harrison and J. Hook. Achieving information flow security through precise control of effects. In *Proc. IEEE Computer Sec. Foundations Workshop*. IEEE Computer Society, 2005.

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

[13] 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.

[14] 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.

[15] T. Kim, M. Peinado, and G. Mainar-Ruiz. STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud. In *Proc. of the USENIX Conference on Security Symposium*, Security'12. USENIX Association, 2012.

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

[17] B. Köpf, L. Mauborgne, and M. Ochoa. Automatic quantification of cache side-channels. In *Proceedings of the 24th international conference on Computer Aided Verification*, CAV'12. Springer-Verlag, 2012.

[18] M. Krohn, A. Yip, M. Brodsky, R. Morris, and M. Walfish. A World Wide Web Without Walls. In *6th ACM Workshop on Hot Topics in Networking (Hotnets)*, Atlanta, GA, November 2007.

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

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

[21] S. Marlow, R. Newton, and S. L. P. Jones. A monad for deterministic parallelism. In *Proc. ACM SIGPLAN Symposium on Haskell*, 2011.

[22] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[23] D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA conference on Topics in Cryptology*, CT-RSA'06. Springer-Verlag, 2006.

[24] B. Pablo and A. Russo. Lazy programs leak secrets. In *the Pre-proceedings of the 18th Nordic Conference on Secure IT Systems (NordSec)*, October 2013.

[25] D. Page. Partitioned cache architecture as a side-channel defence mechanism. *IACR Cryptology ePrint Archive*, 2005, 2005.

[26] C. Percival. Cache missing for fun and profit. In *Proc. of BSDCan 2005*, 2005.

[27] 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*, 2002.

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

[29] 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.

[30] 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.

[31] 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.

[32] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. IEEE Computer Sec. Foundations Workshop*, July 2000.

[33] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM Symp. on Principles of Prog. Languages*, Jan. 1998.

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

[35] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Mazières. Addressing covert termination and timing channels in concurrent information flow systems. In *The 17th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 201–213. ACM, September 2012.

[36] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in the presence of exceptions. *Arxiv preprint arXiv:1207.1457*, 2012.

[37] D. Stefan, P. Buiras, E. Z. Yang, A. Levy, D. Terei, A. Russo, and D. Mazières. Eliminating cache-based timing attacks with instruction-based scheduling. In *Proc. European Symp. on Research in Computer Security*, 2013.

[38] W. Swierstra. *A Functional Specification of Effects.* PhD thesis, University of Nottingham, November 2008.

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

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

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

```
attack :: LMVar LH Int → LIORef LH [Int] → LIORef LH [Int] →
          Labeled LH Int → LIO LH Int
attack lmv lref href h
  = do b ← traverse lref
       when b (do    -- Thread C
                 forkLIO (do x ← unlabel h
                             when (x ≡ 0) (do b ← traverse href
                                              when b (return ())))
                 threadDelay delay_C
                   -- Thread A
                 forkLIO (do b ← traverse lref
                             when b (putLMVar lmv 1))
                   -- Thread B
                 forkLIO (do threadDelay delay_B
                             putLMVar lmv 0)
                 return ()
       w ← takeLMVar lmv
       _ ← takeLMVar lmv
       return w
```

Fig. 9. Cache-attack that leaks one bit of a secret in *LIO*

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

A Cache-attack for *LIO*

```
guess hs = do let lL  = [1 .. constant] :: [Int]
                  lH  = [1 .. constant] :: [Int]
              lmv  ← newEmptyLMVar L
              lref ← newLIORef L lL
              href ← newLIORef H lH
              mapM (attack lmv lref href) hs
```

Fig. 10. Magnification of the attack in Figure 9

Fig. 9 shows the cache-attack described in Section 2 for *LIO*. We assume the classic two-point lattice (of type *LH*) where security levels L and H denote public and secret data, respectively. Function *attack* takes a public shared *LMVar* (*lmv*), two references to lists of public (*lref*) and secret data (*href*), and a secret integer h. The goal of *attack* is to return a public integer equal to h. For simplicity, we use *threadDelay* n, which places a thread to sleep for n micro seconds, to exploit the race to *lmv*—using a loop would work equally well. In Fig. 9, parameter *delay_C* is set to wait for thread C to finish running. Similarly, parameter *delay_B* imposes a delay on thread B before attempting to update *lmv* with 0. Variable w stores the first written value in *lmv*, which will coincide with the value of h.

Figure 10 shows the magnification of the attack for a list of secret integers (hs). Parameter *constant* determines the size of the lists with public and secret data, respectively. The magnification is simply to map function *attack* to the list of secrets. (The precise values of these parameters are machine-specific and experimentally determined.) Below we present the final component required for the attack:

$$\begin{aligned} &traverse\ ref = \mathbf{do}\ ls \leftarrow readLIORef\ ref \\ &\qquad\qquad\qquad return\ ((ls \equiv [x \mid x \leftarrow ls]) \wedge (reverse\ (reverse\ ls) \equiv ls)) \end{aligned}$$

B Monadic Operations for (*Thread m*)

$$\begin{aligned} &return\ x = Done\ x \\ &Done\ x\ \ggg\!= f = f\ x \\ &Atom\ m \ggg\!= f = Atom\ (\mathbf{do}\ res \leftarrow m \\ &\qquad\qquad\qquad\qquad\qquad return\ (res \ggg\!= f)) \\ &Fork\ res\ res' \ggg\!= f = Fork\ res\ (res' \ggg\!= f) \\ &Parallel\ p\ g \ggg\!= f = Parallel\ p\ (\lambda r \to g\ r \ggg\!= f) \end{aligned}$$

Fig. 11. Defintions *return* and $\ggg\!=$.

Figure 11 shows the precise definition for *return* and $\ggg\!=$. The interesting definitions are the ones related to bind. Computed values are represented by *Done*, so this is the only case when f is applied. The case for *Atom* constructs a resumption consisting in the first atomic step in m ($res \leftarrow m$) and returning a new resumption sequencing the subsequent atomic steps in m ($return\ (res \ggg\!= f)$). In this case, the *do-notation* describes operations in the monad m (not *Thread m*). The definition of *Fork* sequences the atomic actions found in the resumption res' ($res' \ggg\!= f$). Similarly, the case $Parallel\ p\ g$ sequences the atomic steps generated by $g\ r$ ($g\ r \ggg\!= f$), where r is the result of the spawned parallel computation.

C Granularity of Atomic Steps

Figure 12 shows the definition of function *singleAtom*. When applied, *singleAtom* collapses the atomic steps found between constructors *Fork* and *Parallel*. The cases for *Done*, *Fork*, and *Parallel* are self-explanatory. The case for *Atom* deserves some explanation. It only creates an *Atom* ($Atom\ (m \ggg\!= atomically)$), which first atomic step is performed by m, and the resulting resumption is given to the auxiliary function *atomically*. This function removes all the consecutive constructors *Atom* ($atomically\ (Atom\ m') = m' \ggg\!= atomically$).

D Soundness

We start by showing that the evaluation relations $\hookrightarrow$ and $\hookrightarrow_L$ are deterministic. Note that this is possible because we assume deterministic parallelism in our pure parallel computations. The following results rely on the previous determinacy results for sequential LIO.

```
$singleAtom :: Monad\ m \Rightarrow Thread\ m\ a \rightarrow Thread\ m\ a$
$singleAtom\ (Done\ x)\qquad = Done\ x$
$singleAtom\ (Atom\ m)\qquad = Atom\ (m \gg\!= atomically)$
  where
    $atomically\ (Done\ x)\qquad\quad = return\ (Done\ x)$
    $atomically\ (Atom\ m')\qquad\ = m' \gg\!= atomically$
    $atomically\ (Fork\ res\ res') = return\ (Fork\ res\ (singleAtom\ res'))$
    $atomically\ (Parallel\ p\ f)\ = return\ (Parallel\ p\ (\lambda r \rightarrow singleAtom\ (f\ r)))$
$singleAtom\ (Fork\ res\ res') = Fork\ res\ (singleAtom\ res')$
$singleAtom\ (Parallel\ p\ f) = Parallel\ p\ (\lambda r \rightarrow singleAtom\ (f\ r))$
```

Fig. 12. Collapsing atomic steps

Lemma 0 (Determinacy of $\hookrightarrow$). *If $\langle \Sigma, e\rangle \hookrightarrow \langle \Sigma', e'\rangle$ and $\langle \Sigma, e\rangle \hookrightarrow \langle \Sigma'', e''\rangle$, then $\langle \Sigma', e'\rangle = \langle \Sigma'', e''\rangle$.*

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

Lemma 1 (Determinacy of $\hookrightarrow_L$). *If $\langle \Sigma, e\rangle \hookrightarrow_L \langle \Sigma', e'\rangle$ and $\langle \Sigma, e\rangle \hookrightarrow_L \langle \Sigma'', e''\rangle$, then $\langle \Sigma', e'\rangle = \langle \Sigma'', e''\rangle$.*

Proof. By Lemma 0 and definition of ε_L.

The following lemma establishes a simulation between $\hookrightarrow$ and $\hookrightarrow_L$ when reducing the body of a thread whose current label is below or equal to level L. In this result, we use the fact that the reduction $\longrightarrow$ from the original LIO formalization has been proved to have this property.

Lemma 2 (Single-step simulation for public computations).
If $\langle \Sigma, \mathbf{sch}\ (t{:}t_s) \parallel P\rangle \hookrightarrow \langle \Sigma', \mathbf{sch}\ t'_s \parallel P'\rangle$ with $\Sigma.\mathtt{lbl} \sqsubseteq L$, then $\varepsilon_L(\langle \Sigma, \mathbf{sch}\ (t{:}t_s) \parallel P\rangle) \hookrightarrow_L \varepsilon_L(\langle \Sigma', \mathbf{sch}\ t'_s \parallel P'\rangle)$.

Proof. From previous results, we know that if m is a sequential LIO computation and $\langle \Sigma, m\rangle \longrightarrow \langle \Sigma', e\rangle$, then $\varepsilon_L(\langle \Sigma, m\rangle) \longrightarrow_L \varepsilon_L(\langle \Sigma', e\rangle)$.

- Case $t = Atom\ (put\ \Sigma.\mathtt{lbl} \gg m)$:
  $\varepsilon_L(\langle \Sigma, \mathbf{sch}\ (Atom\ (put\ \Sigma.\mathtt{lbl} \gg m) : t_s) \parallel P\rangle)$
  $= \langle \varepsilon_L(\Sigma), \mathbf{sch}\ (Atom\ (put\ \Sigma.\mathtt{lbl} \gg \varepsilon_L(m)) : \varepsilon_L(t_s)) \parallel \varepsilon_L(P)\rangle$
  $\hookrightarrow_L \varepsilon_L(\langle \varepsilon_L(\Sigma^1), \mathbf{sch}\ (\varepsilon_L(t_s) +\!\!+ [put\ \Sigma.\mathtt{lbl} \succ \varepsilon_L(e)]) \parallel \varepsilon_L(P')\rangle)$
  We know that $\varepsilon_L(\Sigma^1) = \varepsilon_L(\Sigma)$ from previous results, since LIO state transformations cannot introduce secrets observable by an attacker.
- Case $t = Parallel\ p\ f$:
  $\varepsilon_L(\langle \Sigma, \mathbf{sch}\ (Parallel\ p\ f : t_s) \parallel P\rangle)$
  $= \langle \varepsilon_L(\Sigma), \mathbf{sch}\ (Parallel\ \varepsilon_L(p)\ (\varepsilon_L \circ f) : \varepsilon_L(t_s)) \parallel \varepsilon_L(P)\rangle$
  $\hookrightarrow_L \varepsilon_L(\langle \varepsilon_L(\Sigma^1), \mathbf{sch}\ (\varepsilon_L(t_s) +\!\!+ [Atom\ (readLMVar\ v_s \gg\!= \varepsilon_L \circ f)]) \parallel \varepsilon_L(P)$
  $\parallel putLMVar\ v_s\ (runPure\ \varepsilon_L(p))\rangle)$

As before, we know that $\varepsilon_L(\Sigma^1) = \varepsilon_L(\Sigma)$, so the result follows directly from the properties of the erasure function.

– The other cases are similar.

We can also show that initial and final configurations for any reduction steps taken from a thread above L are equal when erased.

Lemma 3. *If $\langle \Sigma, \textbf{sch}\ (t : t_s) \parallel P\rangle \hookrightarrow \langle \Sigma', \textbf{sch}\ t'_s \parallel P'\rangle$ with $\Sigma.\texttt{lbl} \not\sqsubseteq L$, then $\varepsilon_L(\langle \Sigma, \textbf{sch}\ (t : t_s) \parallel P\rangle) = \varepsilon_L(\langle \Sigma', \textbf{sch}\ t'_s \parallel P'\rangle)$.*

Proof. Since $\varepsilon_L(\langle \Sigma, \textbf{sch}\ (t : t_s) \parallel P\rangle) = \langle \varepsilon_L(\Sigma^1), \bullet\rangle$, we only have to show that $\varepsilon_L(\Sigma) = \varepsilon_L(\Sigma^1)$, where Σ^1 is the modified environment after performing the reduction step. The proof is similar to the corresponding lemma in the original version of LIO, since the possible environment modifications are the same.

We can now prove the many-step simulation lemma.

Proposition 1 (Many-step simulation). *If $\langle \Sigma, \textbf{sch}\ t_s \parallel P\rangle \hookrightarrow^* \langle \Sigma', \textbf{sch}\ t'_s \parallel P'\rangle$, then it holds that $\varepsilon_L(\langle \Sigma, \textbf{sch}\ t_s \parallel P\rangle) \hookrightarrow^*_L \varepsilon_L(\langle \Sigma', \textbf{sch}\ t'_s \parallel P'\rangle)$.*

Proof. The proof is by induction on the derivation of $\langle \Sigma, \textbf{sch}\ t_s \parallel P\rangle \hookrightarrow^* \langle \Sigma', \textbf{sch}\ t'_s \parallel P'\rangle$. We consider a thread queue of the form $r : r_s$, and suppose that $\langle \Sigma, \textbf{sch}\ (e : r_s) \parallel P\rangle \hookrightarrow \langle \Sigma^1, r'_s\rangle$ and $\langle \Sigma^1, r'_s\rangle \hookrightarrow^* \langle \Sigma', \textbf{sch}\ t'_s \parallel P'\rangle$ (otherwise the reduction is not making any progress, and the result is trivial).

– If $\Sigma.\texttt{lbl} \sqsubseteq L$, the result follows by Lemma 2 and the induction hypothesis.
– If $\Sigma.\texttt{lbl} \not\sqsubseteq L$, the result follows by Lemma 3 and the induction hypothesis.

Finally, we prove the non-interference result, showing that two terminating runs that start with L-equivalent configurations must end in L-equivalent configurations.

Theorem 1 (Termination-insensitive non-interference). *Given a computation e, inputs e_1 and e_2, an attacker at level L, runtime environments Σ_1 and Σ_2, then for all inputs e_1, e_2 such that $e_1 \approx_L e_2$, if $\langle \Sigma_1, \textbf{sch}\ [e\ e_1]\rangle \hookrightarrow^* \langle \Sigma'_1, \textbf{sch}\ []\rangle$ and $\langle \Sigma_2, \textbf{sch}\ [e\ e_2]\rangle \hookrightarrow^* \langle \Sigma'_2, \textbf{sch}\ []\rangle$, then $\langle \Sigma'_1, \textbf{sch}\ []\rangle \approx_L \langle \Sigma'_2, \textbf{sch}\ []\rangle$.*

Proof. By definition of $\approx_L$, we have $\varepsilon_L(\langle \Sigma_1, \textbf{sch}\ [e\ e_1]\rangle) = \varepsilon_L(\langle \Sigma_2, \textbf{sch}\ [e\ e_2]\rangle)$. Then, by the simulation lemma (Proposition 1), we have

$$\varepsilon_L(\langle \Sigma_1, \textbf{sch}\ [e\ e_1]\rangle) \hookrightarrow^*_L \varepsilon_L(\langle \Sigma'_1, \textbf{sch}\ []\rangle)$$
$$\varepsilon_L(\langle \Sigma_2, \textbf{sch}\ [e\ e_2]\rangle) \hookrightarrow^*_L \varepsilon_L(\langle \Sigma'_2, \textbf{sch}\ []\rangle)$$

Moreover, from the determinacy of $\hookrightarrow^*_L$ given in Lemma 1, the end configurations must be the same, *i.e.* $\varepsilon_L(\langle \Sigma'_1, \textbf{sch}\ []\rangle) = \varepsilon_L(\langle \Sigma'_2, \textbf{sch}\ []\rangle)$. Finally, by definition of $\approx_L$, we conclude $\langle \Sigma'_1, \textbf{sch}\ []\rangle \approx_L \langle \Sigma'_2, \textbf{sch}\ []\rangle$.