Foundations for Parallel Information Flow Control Runtime Systems

Marco Vassena[1], Gary Soeller[2], Peter Amidon[2], Matthew Chan[3], and Deian Stefan[2]

[1] Chalmers University of Technology, Sweden
[2] University of California San Diego, USA
[3] Awake Security, USA

Abstract. We present the foundations for a new dynamic information flow control (IFC) parallel runtime system, LIO$_{\text{PAR}}$. To our knowledge, LIO$_{\text{PAR}}$ is the first dynamic language-level IFC system to (1) support deterministic parallel thread execution and (2) eliminate both internal- and external-timing covert channels that exploit the runtime system. Most existing IFC systems are vulnerable to external timing attacks because they are built atop vanilla runtime systems that do not account for security—these runtime systems allocate and reclaim shared resources, e.g., CPU-time and memory, *fairly* between threads at different security levels. While such attacks have largely been ignored—or, at best, mitigated—we demonstrate that extending IFC systems with parallelism leads to the *internalization* of these attacks. Our IFC runtime system design addresses these concerns by hierarchically managing resources—both CPU-time and memory—and making resource allocation and reclamation explicit at the language-level. We prove that LIO$_{\text{PAR}}$ is secure, i.e., it satisfies timing-sensitive non-interference, even when exposing clock and heap-statistics APIs.

1 Introduction

Language-level dynamic information flow control (IFC) is a promising approach to building secure software systems. With IFC, developers specify application-specific, data-dependent security policies. The language-level IFC system—often implemented as a library or as part of a language runtime system—then enforces these policies automatically, by tracking and restricting the flow of information throughout the application. In doing so, IFC can ensure that different application components—even when buggy or malicious—cannot violate data confidentiality or integrity.

The key to making language-level IFC practical lies in designing real-world programming language features and abstractions without giving up on security. Unfortunately, many practical language features are at odds with security. For example, even exposing language features as simple as `if`-statements can expose users to timing attacks [43, 67]. Researchers have made significant strides towards addressing these challenges—many IFC systems now support real-world features and abstractions safely [10, 15, 21, 35, 44, 51, 52, 55, 58, 62, 63, 65, 71, 72]. To the best of our knowledge, though, no existing language-level dynamic IFC supports parallelism. Yet, many applications rely on parallel thread execution. For example, modern Web applications typically handle user requests in parallel, on multiple CPU cores, taking advantage of modern hardware. Web applications built atop state-of-the-art dynamic IFC Web frameworks (e.g., Jacqueline [71], Hails [12, 13], and LMonad [46]), unfortunately, do not handle user requests in parallel—the language-level IFC systems that underlie them (e.g., Jeeves [72] and LIO [55]) do not support parallel thread execution.

In this paper we show that extending most existing IFC systems—even concurrent IFC systems such as LIO—with parallelism is unsafe. The key insight is that most IFC systems *do not* prevent sensitive computations from affecting public computations; they simply prevent public computations from *observing* such sensitive effects. In the sequential and concurrent setting, such effects are only observable to attackers *external* to the program and thus outside the scope of most IFC systems. However, when computations execute in parallel, they are essentially external to one another and thus do not require an observer external to the system—they can observe such effects internally.

Consider a program consisting of three concurrent threads: two public threads—p_0 and p_1—and a secret thread—s_0. On a single core, language-level IFC can ensure that p_0 and p_1 do not learn anything secret by, for example, disallowing them from observing the return values (or lack thereof) of the secret thread. Systems such as LIO are careful to ensure that public threads cannot learn secrets even indirectly, e.g., via covert channels that abuse the runtime system scheduler. In contrast, secret threads can leak information to an external observer that monitors public events (e.g., messages from public threads) by influencing the behavior of the public threads. For example, s_0 can terminate (or not) based on a secret and thus affect the amount of time p_0 and p_1 spend executing on the CPU—if s_0 terminated, the runtime allots the whole CPU to public threads, otherwise it only allots, say, two thirds of the CPU to the public threads; this allows an external attacker to trivially infer the secret (e.g., by measuring the rate of messages written to a public channel). Unfortunately, such *external timing attacks* manifest *internally* to the program when threads execute in parallel, on multiple cores. Suppose, for example, that p_0 and s_0 are co-located on a core and run in parallel to p_1. By terminating early (or not) based on a secret, s_0 affects the CPU time allotted to p_0, which can be measured by p_1. For example, p_1 can count the number of messages sent from p_0 on a public channel—the number of p_0 writes indirectly leaks whether or not s_0 terminated.

We demonstrate that such attacks are feasible by building several proof-of-concept programs that exploit the way the runtime system allocate and reclaim *shared* resources to violate LIO's security guarantees. Then, we design a new dynamic parallel language-level IFC runtime system called LIO_{PAR}, which extends LIO to the parallel setting by changing how *shared* runtime system resources—namely CPU-time and memory—are managed. Ordinary runtime systems (e.g., GHC for LIO) *fairly* balance resources between threads; this means that allocations or reclamations for secret LIO threads directly affect resources available for public LIO threads. In contrast, LIO_{PAR} makes resource management *explicit* and *hierarchical*. When allocating new resources on behalf of a thread, the LIO_{PAR} runtime does not "fairly" steal resources from all threads. Instead, LIO_{PAR} demands that the thread requesting the allocation explicitly gives up a portion of its own resources. Similarly, the runtime does not automatically relinquish the resources of a terminated thread—it requires the parent thread to explicitly reclaim them.

Nevertheless, automatic memory management is an integral component of modern language runtimes—high-level languages (e.g., Haskell and thus LIO) are typically garbage collected, relieving developers from manually reclaiming unused memory. Unfortunately, even if memory is hierarchically partitioned, some garbage collection (GC) algorithms, such as GHC's stop-the-world, may introduce timing covert channels [47]. Inspired by previous work on real-time GCs (e.g., [3, 5, 6, 16, 45, 49]), we equip LIO$_{\mathrm{PAR}}$ with a per-thread, interruptible garbage collector. This strategy is agnostic to the particular GC algorithm used: our hierarchical runtime system only demands that the GC runs within the memory confines of individual threads and their time budget.

In sum, this paper makes three contributions:

- ▶ We observe that several external timing attacks *manifest internally* in the presence of parallelism and demonstrate that LIO, when compiled to run on multiple cores, is vulnerable to such attacks (§2).
- ▶ In response to these attacks, we propose a novel parallel runtime system design that safely manages shared resources by enforcing explicit and *hierarchical* resource allocation and reclamation (§3). To our knowledge, LIO$_{\mathrm{PAR}}$ is the first parallel language-level dynamic IFC runtime system to address both internal and external timing attacks that abuse the runtime system scheduler, memory allocator, and GC.
- ▶ We formalize the LIO$_{\mathrm{PAR}}$ hierarchical runtime system (§4) and prove that it satisfies *timing-sensitive non-interference* (§5); we believe that this is the first general purpose dynamic IFC runtime system to provide such strong guarantees in the parallel setting [67].

We remark that neither our attack nor our defense is tied to LIO or GHC—we focus on LIO because it already supports concurrency. We believe that extending any existing language-level IFC system with parallelism will pose the same set of challenges—challenges that can be addressed using explicit and hierarchical resource management. Supplemental materials (detailed formal definitions and proofs) can be found in Appendixes A and B while the source code for our attacks can be found in Appendix C.

2 Internal manifestation of external attacks

In this section we give a brief overview of LIO and discuss the implications of shared, finite runtime system resources on security. We demonstrate several external timing attacks against LIO that abuse two such resources—the thread scheduler and garbage collector—and show how running LIO threads in parallel internalizes these attacks.

2.1 Overview of concurrent LIO information flow control system

At a high level, the goal of an IFC system is to track and restrict the flow of information according to a security policy—almost always a form of *non-interference* [14]. Informally, this policy ensures *confidentiality*, i.e., secret data should not leak to public entities, and *integrity*, i.e., untrusted data should not affect trusted entities.

To this end, LIO tracks the flow of information at a coarse-granularity, by associating *labels* with threads. Implicitly, the thread label classifies all the values in its scope and reflects the sensitivity of the data that it has inspected. Indeed, LIO "raises" the label of a thread to accommodate for reading yet more sensitive data. For example, when a $public$ thread reads $secret$ data, its label is raised to $secret$—this reflects the fact that the rest of the thread computation may depend on sensitive data. Accordingly, LIO uses the thread's *current label* or *program counter label* to restrict its communication. For example, a $secret$ thread can only communicate with other $secret$ threads.

In LIO, developers can express programs that manipulate data of varying sensitivity—for example programs that handle both $public$ and $secret$ data—by forking multiple threads, at run-time, as necessary. However, naively implementing concurrency in an IFC setting is dangerous: concurrency can amplify and internalize the *termination covert channel* [1, 61], for example, by allowing public threads to observe whether or not secret threads terminated. Moreover, concurrency often introduces *internal timing covert channels* wherein secret threads leak information by influencing the scheduling behavior of public threads. Both classes of covert channels are high-bandwidth and easy to exploit.

Stefan et al. [55] were careful to ensure that LIO does not expose these termination and timing covert channels *internally*. LIO ensures that even if secret threads terminate early, loop forever, or otherwise influence the runtime system scheduler, they cannot leak information to public threads. But, secret threads *do* affect public threads with those actions and thus expose timing covert channels *externally*—public threads just cannot detect it. In particular, LIO disallows public threads from (1) directly inspecting the return values (and thus timing and termination behavior) of secret threads, without first raising their program counter label, and (2) observing runtime system resource usage (e.g., elapsed time or memory availability) that would indirectly leak secrets.

LIO prevents public threads from measuring CPU-time usage directly—LIO does not expose a clock API—and indirectly—threads are scheduled fairly in a round-robin fashion [55]. Similarly, LIO prevents threads from measuring memory usage directly—LIO does not expose APIs for querying heap statistics—and indirectly, through garbage collection cycles (e.g., induced by secret threads) [47]—GHC's stop-the-world GC stops all threads. Like other IFC systems, the security guarantees of LIO are weaker in practice because its formal model does not account for the GC and assumes memory to be infinite [55, 58].

2.2 External timing attacks to runtime systems

Since secret threads can still influence public threads by abusing the scheduler and GC, LIO is vulnerable to *external timing and termination attacks*, i.e., attacks that leak information to external observers.To illustrate this, we craft several LIO programs consisting of two threads: a public thread p that writes to the external channel observed by the attacker and a secret thread s, which abuses the runtime to influence the throughput of the public thread. The secret thread can leak in many ways, for example, thread s can:

1. *fork bomb*, i.e., fork thousands of secret threads that will be interleaved with p and thus decrease its write throughput;
2. terminate early to relinquish the CPU to p and thus double its write throughput;
3. exhaust all memory to crash the program, and thus stop p from further writing to the channel;
4. force a garbage collection which, because of GHC's stop-the-world GC, will intermittently stop p from writing to the channel.

These attacks abuse the runtime's automatic *allocation* and *reclamation* of shared resources, i.e., CPU time and memory. In particular, attack 1 hinges on the runtime *allocating* CPU time for the new secret threads, thus reducing the CPU time allotted to the public thread. Dually, attack 2 relies on it *reclaiming* the CPU time of terminated threads—it reassigns it to public threads. Similarly, attacks 3 and 4 force the runtime

Core c_0		Core c_1
Secret Thread (s_0)	Public Thread (p_0)	Public Thread (p_1)

Secret Thread (s_0):

```
if secret
 then terminate
 else forever skip
```

Public Thread (p_0):

```
forever
   (write chan $p_0$)
```

Public Thread (p_1):

```
for [1..n] (write chan $p_1$)
ms <- read chan
$w_0$ <- count $p_0$ ms
$w_1$ <- count $p_1$ ms
return ($w_0$ < $w_1$)
```

Fig. 1: In this attack three threads run in parallel, colluding to leak secret `secret`. The two public threads write to a *public* output channel; the relative number of messages written on the channel by each thread directly leaks the secret (as inferred by p_1). To affect the rate that p_0 can write, s_0 conditionally terminates—which will free up time on core c_0 for p_0 to execute.

to allocate all the available memory and preemptively reassign CPU time to the GC, respectively.

These attacks are not surprising, but, with the exception of the GC-based attack [47], they are novel in the IFC context. Moreover these attacks are not exhaustive—there are other ways to exploit the runtime system—nor optimized—our implementation leaks sensitive data at a rate of roughly 2bits/second[4]. Nevertheless, they are feasible and—because they abuse the runtime—they are effective against language-level external-timing mitigation techniques, including [55, 75]. The attacks are also feasible on other systems—similar attacks that abuse the GC have been demonstrated for both the V8 and JVM runtimes [47].

2.3 Internalizing external timing attacks

LIO, like almost all IFC systems, considers external timing out of scope for its attacker model. Unfortunately, when we run LIO threads on multiple cores, in parallel, the allocation and reclamation of resources on behalf of secret threads is indirectly observable by public threads. Unsurprisingly, some of the above external timing attacks manifest internally—a thread running on a parallel core acts as an "external" attacker. To demonstrate the feasibility of such attacks, we describe two variants of the aforementioned scheduler-based attacks which leak sensitive information internally to public threads.

Secret threads can leak information by relinquishing CPU time, which the runtime reclaims and *unsafely* redistributes to public threads running on the same core. Our attack program consists of three threads: two public threads—p_0 and p_1—and a secret thread—s_0. Fig. 1 shows the pseudo-code for this attack. Note that the threads are secure in isolation, but leak the value of $secret$ when executed in parallel, with a round robin scheduler. In particular, threads p_0 and s_0 run concurrently on core c_0 using half of the CPU time each, while p_1 runs in parallel alone on core c_1 using all the CPU time. Both public threads repeatedly write their respective thread IDs to a *public channel*. The secret thread, on the other hand, loops forever or terminates depending on $secret$. Intuitively, when the secret thread terminates, the runtime system redirects its CPU time to p_0, thus both p_1 and p_0 write at the same rate. In converse, when the secret thread does not terminate early, p_0 is scheduled in a round-robin fashion with s_0 on the same core and can thus only write half as fast as p_1. More specifically:

[4] A more assiduous attacker could craft similar attacks that leak at higher bit-rates.

- If `secret = true`, thread s_0 terminates and the runtime system assigns all the CPU time of core c_0 to public thread p_0, which then writes at the same rate as thread p_1 on core c_1. Then, p_0 writes as many times as p_1, which then returns `true`.
- If `secret = false`, secret thread s_0 loops and public thread p_0 shares the CPU time on core c_0 with it. Then, p_0 writes messages at roughly half the rate of thread p_1, which writes more often—it has all the CPU time on c_1—and thus returns `false`.[5]

Secret LIO threads can also leak information by allocating many secret threads on a core with public threads—this reduces the CPU-time available to the public threads. For example, using the same setting with three threads from before, the secret thread forks a spinning thread on core c_1 by replacing command `terminate` with command `fork (forever skip)` c_1 in the code of thread s_0 in Fig. 1. Intuitively, if `secret` is `false`, then p_1 writes more often than p_0 before, otherwise the write rate of p_1 decreases—it shares core c_1 with the child thread of s_0—and p_0 writes as often as p_1.

Not all external timing attacks can be internalized, however. In particular, GHC's approach to reclaiming memory via a stop-the-world GC simultaneously stops all threads on *all* cores, thus the relative write rate of public threads remain constant. Interestingly, though, implementing LIO on runtimes (e.g., Node.js as proposed by Heule et al. [17]) with modern parallel garbage collectors that do not always stop the world would internalize the GC-based external timing attacks. Similarly, abusing GHC's memory allocation to exhaust all memory crashes all the program threads and, even though it cannot be internalized, it still results in information leakage.

3 Secure, parallel runtime system

To address the external and internal timing attacks, we propose a new dynamic IFC runtime system design. Fundamentally, today's runtime systems are vulnerable because they automatically allocate and reclaim resources that are shared across threads of varying sensitivity. However, the automatic allocation and reclamation is not in itself a problem—it is only a problem because the runtime steals (and grants) resources from (and to) differently-labeled threads.

Our runtime system, LIO_{PAR}, explicitly partitions CPU-time and memory among threads—each thread has a fixed CPU-time and memory *budget* or *quota*. This allows resource management decisions to be made locally, for each thread, independent of the other threads in the system. For example, the runtime scheduler of LIO_{PAR} relies on CPU-time partitioning to ensure that threads always run for a fixed amount of time, irrespective of the other threads running on the same core. Similarly, in LIO_{PAR}, the memory allocator and garbage collector rely on memory partitioning to be able to allocate and collect memory on behalf of a thread without being influenced or otherwise influencing other threads in the system. Furthermore, partitioning resources among threads enables fine-grained control of resources: LIO_{PAR} exposes secure primitives to (i) measure resource usage (e.g., time and memory) and (ii) elicit garbage collection cycles.

The LIO_{PAR} runtime does not automatically balance resources between threads. Instead, LIO_{PAR} makes resource management explicit at the language level. When

[5] The attacker needs to empirically find parameter n, so that p_1 writes roughly twice as much as thread p_0 with half CPU time on core c_0.

forking a new thread, for example, LIO_{PAR} demands that the parent thread give up part of its CPU-time and memory budgets to the children. Indeed, LIO_{PAR} even manages core ownership or *capabilities* that allow threads to fork threads across cores. This approach ensures that allocating new threads does not indirectly leak any information externally or to other threads. Dually, the LIO_{PAR} runtime does not re-purpose unused memory or CPU-time, even when a thread terminates or "dies" abruptly—parent threads must explicitly kill their children when they wish to reclaim their resources.

To ensure that CPU-time and memory can always be reclaimed, LIO_{PAR} allows threads to kill their children anytime. Unsurprisingly, this feature requires restricting the LIO_{PAR} floating-label approach more than that of LIO—LIO_{PAR} threads cannot raise their current label if they have already forked other threads. As a result, in LIO_{PAR} threads form a *hierarchy*—children threads are always at least as sensitive as their parent—and thus it is secure to expose an API to *allocate* and *reclaim* resources.

Attacks Revisited. LIO_{PAR} enforces security against *reclamation-based attacks* because secret threads cannot automatically relinquish their resources. For example, our hierarchical runtime system stops the attack in Fig. 1: even if secret thread s_0 terminates (`secret = true`), the throughput of public thread p_0 remains constant—LIO_{PAR} does not reassign the CPU time of s_0 to p_0, but keeps s_0 spinning until it gets killed. Similarly, LIO_{PAR} protects against *allocation-based attacks* because secret threads cannot steal resources owned by other public threads. For example, the *fork-bomb* variant of the previous attack fails because LIO_{PAR} aborts command `fork (forever skip)` c_1—thread s_0 does not own the core capability c_1—and thus the throughput of p_1 remains the same. In order to substantiate these claims, we first formalize the design of the *hierarchical* runtime system (§4) and establish its security guarantees (§5).

Trust model. This work addresses attacks that exploit runtime system resource management — in particular memory and CPU-time. We do not address attacks that exploit other shared runtime system state (e.g., event loops [66], lazy evaluation [8, 62]), shared operating system state (e.g., file system locks [25], events and I/O [23, 33]), or shared hardware (e.g., caches, buses, pipelines and hardware threads [11, 48]) Though these are valid concerns, they are orthogonal and outside the scope of this paper.

4 Hierarchical Calculus

In this section we present the formal semantics of LIO_{PAR}. We model LIO_{PAR} as a security monitor that executes simply typed λ-calculus terms extended with LIO security primitives on an abstract machine in the style of Sestoft [54]. The security monitor reduces secure programs and aborts the execution of leaky programs.

Semantics. The state of the monitor, written $(\Delta, pc, N \mid t, S)$, stores the state of a thread under execution and consists of a heap Δ that maps variables to terms, the thread's program counter label pc, the set N containing the identifiers of the thread's children, the term currently under reduction t and a stack of continuations S. Fig. 2 shows the interesting rules of the sequential small-step operational semantics of the security monitor. The notation $s \rightsquigarrow_{\mu} s'$ denotes a transition of the machine in state s that reduces to state s' in one step with thread parameters $\mu = (h, cl)$.[6] Since we are interested in modeling a system with *finite* resources, we parameterize the transition

[6] We use record notation, i.e., $\mu.h$ and $\mu.cl$, to access the components of μ.

Label	$\ell, pc, cl \in \mathscr{L}$	Params.	$\mu ::= (h, cl)$
Cores	$k \in \{1 \mathinner{..} \kappa\}, K \in \mathcal{P}(\{1 \mathinner{..} \kappa\})$	Heap	$\Delta \in \mathit{Var} \rightharpoonup \mathit{Term}$
Thread Id	$n \in \mathbb{N}, N \in \mathcal{P}(\mathbb{N})$	Budgets	$h, b \in \mathbb{N}$

$$\begin{array}{lrcl}
\text{Type} & \tau & ::= & () \mid \tau_1 \to \tau_2 \mid \mathit{Bool} \mid \mathscr{L} \mid \mathit{LIO}\ \tau \mid \mathit{Labeled}\ \tau \\
 & & & \mid \mathit{TId} \mid \mathit{Core} \mid \mathcal{P}(\{1 \mathinner{..} \kappa\}) \mid \mathbb{N} \\
\text{Value} & v & ::= & () \mid \lambda x.t \mid \mathit{True} \mid \mathit{False} \mid \ell \mid \mathit{return}\ t \mid \mathit{Labeled}\ \ell\ t^{\circ} \mid n \mid k \mid K \\
\text{Term} & t & ::= & v \mid x \mid t_1\ t_2 \mid \textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3 \mid t_1 \gg\!= t_2 \mid \mathit{label}\ t_1\ t_2 \\
 & & & \mid \mathit{unlabel}\ t \mid \mathit{fork}\ t_1\ t_2\ t_3\ t_4\ t_5 \mid \mathit{spawn}\ t_1\ t_2\ t_3\ t_4\ t_5 \mid \mathit{kill}\ t \\
 & & & \mid \mathit{size} \mid \mathit{time} \mid \mathit{wait}\ t \mid \mathit{send}\ t_1\ t_2 \mid \mathit{receive} \\
\text{CTerm} & t^{\circ} & ::= & t \text{ such that } \mathit{fv}(t) = \varnothing \\
\text{Cont.} & C & ::= & x \mid \textbf{then}\ t_2\ \textbf{else}\ t_3 \mid \gg\!= t_2 \mid \mathit{label}\ t \mid \mathit{unlabel} \mid \mathit{fork}\ t_1\ t_2\ t_3\ t_4 \\
 & & & \mid \mathit{spawn}\ t_1\ t_2\ t_3\ t_4 \mid \mathit{kill} \mid \mathit{send}\ t \\
\text{Stack} & S & ::= & [\,] \mid C : S \\
\text{State} & s & ::= & (\Delta, pc, N \mid t, S)
\end{array}$$

(App$_1$)
$$\frac{|\Delta| < \mu.h \qquad \text{fresh}(x)}{(\Delta, pc, N \mid t_1\ t_2, S) \rightsquigarrow_{\mu} (\Delta[x \mapsto t_2], pc, N \mid t_1, x : S)}$$

(App$_2$)
$$(\Delta, pc, N \mid \lambda y.t, x : S) \rightsquigarrow_{\mu} (\Delta, pc, N \mid t\ [x\ /\ y], S)$$

(Var)
$$\frac{x \mapsto t \in \Delta}{(\Delta, pc, N \mid x, S) \rightsquigarrow_{\mu} (\Delta, pc, N \mid t, S)}$$

(Bind$_1$)
$$(\Delta, pc, N \mid t_1 \gg\!= t_2, S) \rightsquigarrow_{\mu} (\Delta, pc, N \mid t_1, \gg\!= t_2 : S)$$

(Bind$_2$)
$$(\Delta, pc, N \mid \mathit{return}\ t_1, \gg\!= t_2 : S) \rightsquigarrow_{\mu} (\Delta, pc, N \mid t_2\ t_1, S)$$

(Label$_1$)
$$(\Delta, pc, N \mid \mathit{label}\ t_1\ t_2, S) \rightsquigarrow_{\mu} (\Delta, pc, N \mid t_1, \mathit{label}\ t_2 : S)$$

(Label$_2$)
$$\frac{pc \sqsubseteq \ell \sqsubseteq \mu.cl \qquad t^{\circ} = \Delta^{*}(t)}{(\Delta, pc, N \mid \ell, \mathit{label}\ t : S) \rightsquigarrow_{\mu} (\Delta, pc, N \mid \mathit{return}\ (\mathit{Labeled}\ \ell\ t^{\circ}), S)}$$

(Unlabel$_1$)
$$(\Delta, pc, N \mid \mathit{unlabel}\ t, S) \rightsquigarrow_{\mu} (\Delta, pc, N \mid t, \mathit{unlabel} : S)$$

(Unlabel$_2$)
$$\frac{pc \sqcup \ell \sqsubseteq \mu.cl}{(\Delta, pc, N \mid \mathit{Labeled}\ \ell\ t, \mathit{unlabel} : S) \rightsquigarrow_{\mu} (\Delta, pc \sqcup \ell, N \mid \mathit{return}\ t, S)}$$

Fig. 2: Sequential LIO$_{\text{PAR}}$.

with the maximum heap size $h \in \mathbb{N}$. Additionally, the clearance label cl represents an upper bound over the sensitivity of the thread's floating counter label pc. Rule [APP_1] begins a function application. Since our calculus is call-by-name, the function argument is saved as a *thunk* (i.e., an unevaluated expression) on the heap at fresh location x and the indirection is pushed on the stack for future lookups.[7] Note that the rule allocates memory on the heap, thus the premise $|\Delta| < h$ forbids a heap overflow, where the notation $|\Delta|$ denotes the size of the heap Δ, i.e., the number of bindings that it contains.[8] To avoid overflows, a thread can measure the size of its own heap via primitive $size$ (§4.2). If t_1 evaluates to a function, e.g., $\lambda y.t$, rule [APP_2] starts evaluating the body, in which the bound variable y is substituted with the heap-allocated argument x, i.e., $t\ [x\ /\ y]$. When the evaluation of the function body requires the value of the argument, variable x is looked up in the heap (rule [VAR]). In the next paragraph we present the rules of the basic security primitives. The other sequential rules are available in Appendix A.

Security Primitives. A labeled value $Labeled\ \ell\ t^{\circ}$ of type $Labeled\ \tau$ consists of term t of type τ and a label ℓ, which reflects the sensitivity of the content. The annotation t° denotes that term t is *closed* and does not contain any free variable, i.e., $fv(t) = \varnothing$. We restrict the syntax of labeled values with closed terms for security reasons. Intuitively, LIO_{PAR} allocates free variables inside a secret labeled values on the heap, which then leaks information to public threads with its size. For example, a public thread could distinguish between two secret values , e.g., $Labeled\ H\ x$ with heap $\Delta = [x \mapsto 42]$, and $Labeled\ H\ 0$ with heap $\Delta = \varnothing$, by measuring the size of the heap. To avoid that, labeled values are closed and the size of the heap of a thread at a certain security level, is not affected by data labeled at different security levels. A term of type $LIO\ \tau$ is a secure computation that performs side effects and returns a result of type τ. Secure computations are structured using standard monadic constructs $return\ t$, which embeds term t in the monad, and *bind*, written $t_1 \gg\!= t_2$, which sequentially composes two monadic actions, the second of which takes the result of the first as an argument. Rule [BIND_1] deconstructs a computation $t_1 \gg\!= t_2$ into term t_1 to be reduced first and pushes on the stack the continuation $\gg\!= t_2$ to be invoked after term t_1.[9] Then, the second rule [BIND_2] pops the topmost continuation placed on the stack (i.e., $\gg\!= t_2$) and evaluates it with the result of the first computation (i.e., $t_2\ t_1$), which is considered complete when it evaluates to a monadic value, i.e., to syntactic form $return\ t_1$. The runtime monitor secures the interaction between computations and labeled values. In particular, secure computations can construct and inspect labeled values exclusively with monadic primitives $label$ and $unlabel$ respectively. Rules [LABEL_1] and [UNLABEL_1] are straightforward and follow the pattern seen in the other rules. Rule [LABEL_2] generates a labeled value at security

[7] The calculus does not feature lazy evaluation. *Sharing* introduces the lazy covert channel, which has already been considered in previous work [62].

[8] To simplify reasoning, our generic memory model is basic and just counts the number of bindings in the heap. It would be possible to replicate our results with more accurate memory models, e.g., GHC's tagless G-machine (STG)[24] (the basis for GHC's runtime [40]), but that would complicate the formalism.

[9] Even though the stack size is unbounded in this model, we could account for its memory usage by explicitly allocating it on the heap, in the style of Yang et al. [69].

level ℓ, subject to the constraint $pc \sqsubseteq \ell \sqsubseteq cl$, which prevents a computation from labeling values below the program counter label pc or above the clearance label cl.[10] The rule computes the closure of the content, i.e., closed term t°, by recursively substituting every free variable in term t with its value in the heap, written $\Delta^{*}(t)$. Rule [UNLABEL_2] extracts the content of a labeled value and taints the program counter label with its label, i.e., it rises it to $pc \sqcup \ell$, to reflect the sensitivity of the data that is now in scope. The premise $pc \sqcup \ell \sqsubseteq cl$ ensures that the program counter label does not float over the clearance cl. Thus, the run-time monitor prevents the program counter label from floating above the clearance label (i.e., $pc \sqsubseteq cl$ always holds).

The calculus also includes concurrent primitives to allocate resources when forking threads ($fork$ and $spawn$ in §4.1), reclaim resources and measure resource usage ($kill$, $size$, and $time$ in §4.2), threads synchronization and communication ($wait$, $send$ and $receive$ in Appendix A).

4.1 Core Scheduler

In this section, we extend LIO_{PAR} with concurrency, which enables (i) *interleaved* execution of threads on a single core and (ii) *simultaneous* execution on κ cores. To protect against attacks that exploit the automatic management of shared *finite* resource (e.g., those in §2.3), LIO_{PAR} maintains a resource budget for each running thread and updates it as threads allocate and reclaim resources. Since κ threads execute at the same time, those changes must be coordinated in order to preserve the consistency of the resource budgets and guarantee *deterministic parallelism*. For this reason, the hierarchical runtime system is split in two components: (i) the *core scheduler*, which executes threads on a single core, ensures that they respect their resource budgets and performs security checks, and (ii) the top-level *parallel scheduler*, which synchronizes the execution on multiple cores and reassigns resources by updating the resource budgets according to the instructions of the core schedulers. We now introduce the core scheduler and describe the top-level parallel scheduler in §4.3.

Syntax. Fig. 3 presents the core scheduler, which has access to the global state $\Sigma = (T, B, H, \theta, \omega)$, consisting of a thread pool map T, which maps a thread id to the corresponding thread's current state, the time budget map B, a memory budget map H, core capabilities map θ, and the global clock ω. Using these maps, the core scheduler ensures that thread n: (i) performs $B(n)$ uninterrupted steps until the next thread takes over, (ii) does not grow its heap above its maximum heap size $H(n)$, and (iii) has exclusive access to the *free* core capabilities $\theta(n)$. Furthermore, each thread id n records the *initial* current label when the thread was created ($n.pc$), its clearance ($n.cl$), and the core where it runs ($n.k$), so that the runtime system can enforce security. Notice that thread ids are *opaque* to threads—they cannot forge them nor access their fields.

Hierarchical Scheduling. The core scheduler performs *deterministic* and *hierarchical* scheduling—threads lower in the hierarchy are scheduled first, i.e., parent threads are scheduled before their children. The scheduler manages a core run queue Q, which is structured as a binary tree with leaves storing thread ids and residual time budgets. The notation n^{b} indicates that thread n can run for b more steps before the next thread runs. When a new thread is spawned, the scheduler creates a subtree with the parent

[10] The labels form a security lattice $(\mathscr{L}, \sqcup, \sqsubseteq)$.

$$\begin{array}{llll} \text{Thread Map} & T & \in & \mathit{TId} \rightharpoonup \mathit{State} \\ \text{Time Map} & B & \in & \mathit{TId} \rightharpoonup \mathbb{N} \\ \text{Size Map} & H & \in & \mathit{TId} \rightharpoonup \mathbb{N} \\ \text{Core Map} & \theta & \in & \mathit{TId} \rightharpoonup \mathcal{P}(\{1 \mathinner{..} \kappa\}) \\ \text{Clock} & \omega & \in & \mathbb{N} \end{array} \qquad \begin{array}{llll} \text{Global State} & \Sigma & ::= & (T, B, H, \theta, \omega) \\ \text{Core Queue} & Q & ::= & \langle n^b \rangle \mid \langle Q_1 \mid Q_2 \rangle \\ \text{Event} & e & ::= & \epsilon \mid \mathbf{fork}(\Delta, n, t, b, h) \\ & & & \mid \mathbf{spawn}(\Delta, n, t, K) \\ & & & \mid \mathbf{kill}(n) \mid \mathbf{send}(n, t) \end{array}$$

$$\textsc{Step}\quad \frac{\Sigma.T(n) = s \qquad \mu = (\Sigma.H(n), n.cl) \qquad s \leadsto_\mu s'}{Q[\langle n^{1+b} \rangle] \xrightarrow{(n, s', \epsilon)}_\Sigma Q[\langle n^b \rangle]}$$

$$\textsc{Fork}\quad \frac{\begin{array}{c} \Sigma.T(n) = (\Delta, pc, N \mid b_2, \mathit{fork}\ \ell_{\mathrm{L}}\ \ell_{\mathrm{H}}\ h_2\ t : S) \\ pc \sqsubseteq \ell_{\mathrm{L}} \qquad n' \leftarrow \mathrm{fresh}^{\mathit{TId}}(\ell_{\mathrm{L}}, \ell_{\mathrm{H}}, n.k) \\ s = (\Delta, pc, \{n'\} \cup N \mid \mathit{return}\ n', S) \qquad \Delta' = \{x \mapsto \Delta(x) \mid x \in \mathit{fv}^*(t, \Delta)\} \\ \Sigma.H(n) = h_1 + h_2 \qquad |\Delta| \leqslant h_1 \quad |\Delta'| \leqslant h_2 \end{array}}{Q[\langle n^{1+b_1+b_2} \rangle] \xrightarrow{(n, s, \mathbf{fork}(\Delta', n', t, b_2, h_2))}_\Sigma Q[\langle \langle n^{b_1} \rangle | \langle n'^{b_2} \rangle \rangle]}$$

$$\textsc{Spawn}\quad \frac{\begin{array}{c} \Sigma.T(n) = (\Delta, pc, N \mid k, \mathit{spawn}\ \ell_{\mathrm{L}}\ \ell_{\mathrm{H}}\ K_1\ t : S) \\ \Sigma.\theta(n) = \{k\} \cup K_1 \cup K_2 \qquad pc \sqsubseteq \ell_{\mathrm{L}} \qquad n' \leftarrow \mathrm{fresh}^{\mathit{TId}}(\ell_{\mathrm{L}}, \ell_{\mathrm{H}}, k) \\ s = (\Delta, pc, \{n'\} \cup N \mid \mathit{return}\ n', S) \qquad \Delta' = \{x \mapsto \Delta(x) \mid x \in \mathit{fv}^*(t, \Delta)\} \end{array}}{Q[\langle n^{1+b} \rangle] \xrightarrow{(n, s, \mathbf{spawn}(\Delta', n', t, K_1))}_\Sigma Q[\langle n^b \rangle]}$$

$$\textsc{Stuck}\quad \frac{\begin{array}{c} \Sigma.T(n) = s \qquad \mathit{MaxHeapSize}(s, \Sigma.H(n)) \vee \mathit{UnlabelStuck}(n, \Sigma.T) \vee \\ \mathit{ForkStuck}(n, \Sigma.H, \Sigma.T) \vee \mathit{SpawnStuck}(s, \theta(n)) \vee \mathit{ValueStuck}(s) \vee \\ \mathit{WaitStuck}(n, T) \vee \mathit{ReceiveStuck}(s) \vee \mathit{KillStuck}(s) \end{array}}{Q[\langle n^{1+b} \rangle] \xrightarrow{(n, s, \epsilon)}_\Sigma Q[\langle n^b \rangle]}$$

$$\textsc{ContextSwitch}\quad \frac{s_\circ = ([\,], \bot, \varnothing \mid \mathit{return}\ (), [\,])}{Q[\langle n^0 \rangle] \xrightarrow{(\circ, s_\circ, \epsilon)}_\Sigma Q[\langle n_1^{\Sigma.B(n_1)} \rangle, \ldots, \langle n_{|Q|}^{\Sigma.B(n_{|Q|})} \rangle]}$$

Fig. 3: Concurrent LIO_{PAR}.

thread on the left and the child on the right. The scheduler can therefore find the thread with the highest priority by following the left spine of the tree and backtracking to the right if a thread has no residual budget.[11] We write $Q[\langle n^b \rangle]$ to mean the first thread encountered via this traversal is n with budget b. As a result, given the slice $Q[\langle n^{1+b} \rangle]$, thread n is the next thread to run, and $Q[\langle n^0 \rangle]$ occurs only if *all* threads in the queue

[11] This procedure might reintroduce a timing channel that leaks the number of threads running on the core. In practice, techniques from real time schedulers could be used to protect against such timing channels. The model of LIO_{PAR} does not capture the execution time of the runtime system itself and thus this issue does not arise in the security proofs.

have zero residual budget. We overload this notation to represent tree updates: a rule $Q[\langle n^{1+b} \rangle] \rightarrow Q[\langle n^{b} \rangle]$ finds the next thread to run in queue Q and decreases its budget by one.

Semantics. Fig. 3 formally defines the transition $Q \xrightarrow{(n,s,e)}_{\Sigma} Q'$, which represents an execution step of the *core scheduler* that schedules thread n in core queue Q, executes it with global state $\Sigma = (T, B, H, \theta, \omega)$ and updates the queue to Q'. Additionally, the core scheduler informs the parallel scheduler of the final state s of the thread and requests on its behalf to update the global state by means of event message e. In rule [STEP], the scheduler retrieves the next thread in the schedule, i.e., $Q[\langle n^{1+b} \rangle]$ and its state in the thread pool from the global state, i.e., $\Sigma.T(n) = s$. Then, it executes the thread for one sequential step with its memory budget and clearance, i.e., $s \rightsquigarrow_{\mu} s'$ with $\mu = (\Sigma.H(n), n.cl)$, sends the empty event ϵ to the parallel scheduler, and decrements the thread's residual budget in the final queue, i.e., $Q[\langle n^{b} \rangle]$. In rule [FORK], thread n creates a new thread t with initial label ℓ_{L} and clearance ℓ_{H}, such that $\ell_{\text{L}} \sqsubseteq \ell_{\text{H}}$ and $pc \sqsubseteq \ell_{\text{L}}$. The child thread runs on the same core of the parent thread, i.e., $n.k$, with fresh id n', which is then added to the set of children, i.e., $\{ n' \} \cup N$. Since parent and child threads do not share memory, the core scheduler must copy the portion of the parent's private heap reachable by the child's thread, i.e., Δ'; we do this by copying the bindings of the variables that are transitively reachable from t, i.e., $fv^{*}(t, \Delta)$, from the parent's heap Δ. The parent thread gives h_2 of its memory budget $\Sigma.H(n)$ to its child. The conditions $|\Delta| \leqslant h_1$ and $|\Delta'| \leqslant h_2$, ensure that the heaps do not overflow their new budgets. Similarly, the core scheduler splits the residual time budget of the parent into b_1 and b_2 and informs the parallel scheduler about the new thread and its resources with event $\mathbf{fork}(\Delta', n', t, b_2, h_2)$, and lastly updates the tree Q by replacing the leaf $\langle n^{1+b_1+b_2} \rangle$ with the two-leaves tree $\langle \langle n^{b_1} \rangle | \langle n'^{b_2} \rangle \rangle$, so that the child thread will be scheduled immediately after the parent has consumed its remaining budget b_1, as explained above. Rule [SPAWN] is similar to [FORK], but consumes core capability resources instead of time and memory. In this case, the core scheduler checks that the parent thread owns the core where the child is scheduled and the core capabilities assigned to the child, i.e., $\theta(n) = \{ k \} \cup K_1 \cup K_2$ for some set K_2, and informs the parallel scheduler with event $\mathbf{spawn}(\Delta', n', t, K_1)$. Rule [STUCK] performs busy waiting by consuming the time budget of the scheduled thread, when it is *stuck* and cannot make any progress—the premises of the rule enumerate the conditions under which this can occur (see Fig. 7 in Appendix A for details). Lastly, in rule [CONTEXTSWITCH] all the threads scheduled in the core queue have consumed their time budget, i.e., $Q[\langle n^{0} \rangle]$ and the core scheduler resets their residual budget using the budget map $\Sigma.B$. In the rule, the notation $Q[\langle n_i^{b} \rangle]$ selects the i-th leaf, where $i \in \{1 \mathinner{..} |Q|\}$ and $|Q|$ denotes the number of leaves of tree Q and symbol $\circ$ denotes the thread identifier of the core scheduler, which updates a dummy thread that simply spins during a context-switch or whenever the core is unused.

4.2 Resource Reclamation and Observations

The calculus presented so far enables threads to manage their time, memory and core capabilities hierarchically, but does not provide any primitive to reclaim their resources. This section rectifies this by introducing (i) a primitive to kill a thread and return its

KILL$_2$

$$\frac{\Sigma.T(n) = (\Delta, pc, \{\, n' \,\} \cup N \mid n', kill : S) \qquad s = (\Delta, pc, N \mid return\ (), S)}{Q[\langle n^{1+b} \rangle] \xrightarrow{(n, s, \mathbf{kill}(n'))}_{\Sigma} Q[\langle n^{b} \rangle]}$$

UNLABEL$_2$

$$\frac{pc \sqcup \ell \sqsubseteq \mu.cl \qquad \forall\, n \in N \;.\; pc \sqcup \ell \sqsubseteq n.pc}{(\Delta, pc, N \mid Labeled\ \ell\ t, unlabel : S) \rightsquigarrow_{\mu} (\Delta, pc \sqcup \ell, N \mid return\ t, S)}$$

GC

$$\frac{R = fv^{*}(t, \Delta) \cup fv^{*}(S, \Delta) \qquad \Delta' = \{\, x \mapsto \Delta(x) \mid x \in R \,\}}{\langle \Delta, pc, N \mid gc\ t, S \rangle \rightsquigarrow_{\mu} \langle \Delta', pc, N \mid t, S \rangle}$$

APP-GC

$$\frac{|\Delta| \equiv \mu.h}{\langle \Delta, pc, N \mid t_1\ t_2, S \rangle \rightsquigarrow_{\mu} \langle \Delta, pc, N \mid gc\ (t_1\ t_2), S \rangle}$$

SIZE

$$\langle \Delta, pc, N \mid size, S \rangle \rightsquigarrow_{\mu} \langle \Delta, pc, N \mid return\ |\Delta|, S \rangle$$

TIME

$$\frac{\Sigma.T(n) = (\Delta, pc, N \mid time, S) \qquad s = (\Delta, pc, N \mid return\ \Sigma.\omega, S)}{Q[\langle n^{1+b} \rangle] \xrightarrow{(n, s, \epsilon)}_{\Sigma} Q[\langle n^{b} \rangle]}$$

Fig. 4: LIO$_{\mathrm{PAR}}$ with resource reclamation and observation primitives.

resources back to the owner and (ii) a primitive to elicit a garbage collection cycle and reclaim unused memory. Furthermore, we demonstrate that the runtime system presented in this paper is robust against timing attacks by exposing a timer API allowing threads to access a global clock.[12] Intuitively, it is secure to expose this feature because LIO$_{\mathrm{PAR}}$ ensures that the time spent executing high threads is fixed in advanced, so timing measurements of low threads remain unaffected. Lastly, since memory is hierarchically partitioned, each thread can securely query the current size of its *private heap*, enabling fine-grained control over the garbage collector.

Kill. A parent thread can reclaim the resources given to its child thread n', by executing $kill\ n'$. If the child thread has itself forked or spawned other threads, they are transitively killed and their resources returned to the parent thread. The concurrent rule [KILL$_2$] in Fig. 4 initiates this process, which is completed by the parallel scheduler via event $\mathbf{kill}(n')$. Note that the rule applies only when the thread killed is a *direct* child of the parent thread—that is when the parent's children set has shape $\{\, n' \,\} \cup N$ for some set N. Now that threads can unrestrictedly reclaim resources by killing their children, we must revise the primitive $unlabel$, since the naive combination of $kill$ and $unlabel$ can result in information leakage. This will happen if a public thread forks another public thread,

[12] An *external* attacker can take timing measurements using network communications. An attacker equipped with an *internal* clock is equally powerful but simpler to formalize [47].

then reads a secret value (raising its label to secret), and based on that decides to kill the child. To close the leak, we modify the rule [UNLABEL$_2$] by adding the highlighted premise, causing the primitive $unlabel$ to fail whenever the parent thread's label would float above the *initial* current label of one of its children.

Garbage Collection. Rule [GC] extends LIO$_{\text{PAR}}$ with a *time-sensitive hierarchical* garbage collector via the primitive $gc\ t$. The rule elicits a garbage collection cycle which drops entries that are no longer needed from the heap, and then evaluates t. The sub-heap Δ' includes the portion of the current heap that is (transitively) *reachable* from the free variables in scope (i.e. those present in the term, $fv^*(t, \Delta)$ or on the stack $fv^*(S, \Delta)$). After collection, the thread resumes and evaluates term t under compacted private heap Δ'.[13] In rule [APP-GC], a collection is *automatically* triggered when the thread's next memory allocation would overflow the heap.

Resource Observations. All threads in the system share a global fine-grained clock ω, which is incremented by the parallel scheduler at each cycle (see below). Rule [TIME] gives all threads unrestricted access to the clock via monadic primitive $time$.

4.3 Parallel Scheduler

This section extends LIO$_{\text{PAR}}$ with *deterministic parallelism*, which allows to execute κ threads simultaneously on as many cores. To this end, we introduce the top-level parallel scheduler, which coordinates simultaneous changes to the global state by updating the resource budgets of the threads in response core events (e.g., fork, spawn, and kill) and ticks the global clock.

Semantics. Fig. 5 formalizes the operational semantics of the parallel scheduler, which reduces a configuration $c = \langle \Sigma, \Phi \rangle$ consisting of global state Σ and core map Φ mapping each core to its run queue, to configuration c' in one step, written $c \hookrightarrow c'$, through rule [PARALLEL] only. The rule executes the threads scheduled on each of the κ cores, which all step at once according to the concurrent semantics presented in §4.1–4.2, with the same current global state Σ. Since the execution of each thread can change Σ *concurrently*, the top-level parallel scheduler reconciles those actions by updating Σ *sequentially* and *deterministically*.[14] First, the scheduler updates the thread pool map T and core map Φ with the final state obtained by running each thread in isolation, i.e., $T' = \Sigma.T[n_i \mapsto s_i]$ and $\Phi' = \Phi[i \mapsto Q_i]$ for $i \in \{1 \mathinner{..} \kappa\}$. Then, it collects all concurrent events generated by the κ threads together with their thread id, sorts the events according to type, i.e., $sort\ [(n_1, e_1), ..., (n_\kappa, e_\kappa)]$, and computes the updated configuration by processing the events in sequence.[15] In particular, new threads are created first (event $\mathbf{spawn}(\cdot)$ and $\mathbf{fork}(\cdot)$), and then killed (event $\mathbf{kill}(\cdot)$)—the ordering between events of the same type is arbitrary and assumed to be fixed. Trivial events (ϵ) do not affect the configuration and thus their ordering is irrelevant. The function $\langle\!\langle es \rangle\!\rangle^c$ computes a final configuration by processing a list of events in order,

[13] In practice a garbage collection cycle takes time that is proportional to the size of the memory used by the thread. That does not hinder security as long as the garbage collector runs on the thread's time budget.

[14] Non-deterministic updates would make the model vulnerable to refinement attacks [41].

[15] Since the clock only needs to be incremented, we could have left it out from the configuration $c = \langle T', B, H, \theta, \Sigma.\omega + 1, \Phi' \rangle$; function $\langle\!\langle es \rangle\!\rangle^c$ does not use nor change its value.

Queue Map $\Phi \in \{1\,..\,\kappa\} \to \mathit{Queue}$ Configuration $c ::= \langle T, B, H, \theta, \omega, \Phi \rangle$

PARALLEL

$$\frac{\forall\, i \in \{1\,..\,\kappa\}.\Phi(i) \xrightarrow{(n_i, s_i, e_i)}_{\Sigma} Q_i \qquad T' = \Sigma.T[n_i \mapsto s_i] \qquad \Phi' = \Phi[i \mapsto Q_i] \\ c = \langle T', B, H, \theta, \Sigma.\omega + 1, \Phi' \rangle \qquad \langle \Sigma', \Phi'' \rangle = \langle\!\langle \mathit{sort}\ [(n_1, e_1), ..., (n_\kappa, e_\kappa)] \rangle\!\rangle^{c}}{\langle \Sigma, \Phi \rangle \hookrightarrow \langle \Sigma', \Phi'' \rangle}$$

$$\begin{aligned}
&\mathit{next}(\_, \epsilon, c) = c\\
&\mathit{next}(n_1, \mathbf{fork}(\Delta, n_2, t, b, h), c)\\
&\quad = \langle T', B'[n_2 \mapsto b], H'[n_2 \mapsto h], \theta', \omega, \Phi \rangle\\
&\quad \textbf{where } \langle T, B, H, \theta, \omega, \Phi \rangle = c\\
&\qquad\qquad s = (\Delta, n_2.pc, \varnothing \mid t, [\,])\\
&\qquad\qquad T' = T[n_2 \mapsto s]\\
&\qquad\qquad B' = B[n_1 \mapsto B(n_1) - b]\\
&\qquad\qquad H' = H[n_1 \mapsto H(n_1) - h]\\
&\qquad\qquad \theta' = \theta[n_2 \mapsto \varnothing]
\end{aligned}$$

$$\begin{aligned}
&\mathit{next}(n_1, \mathbf{spawn}(\Delta, n_2, t, K), c)\\
&\quad = \langle T', B', H', \theta'[n_2 \mapsto K], \omega, \Phi' \rangle\\
&\quad \textbf{where} \langle T, B, H, \theta, \omega, \Phi \rangle = c\\
&\qquad\qquad s = (\Delta, n_2.pc, \varnothing \mid t, [\,])\\
&\qquad\qquad T' = T[n_2 \mapsto s]\\
&\qquad\qquad B' = B[n_2 \mapsto B_0]\\
&\qquad\qquad H' = H[n_2 \mapsto H_0]\\
&\qquad\qquad \theta' = \theta[n_1 \mapsto \theta(n_1) \setminus \{n_2.k\} \cup K]\\
&\qquad\qquad \Phi' = \Phi[n_2.k \mapsto \langle n_2^{B_0} \rangle]
\end{aligned}$$

$$\begin{aligned}
&\mathit{next}(n, \mathbf{kill}(n'), \langle T, B, H, \theta, \omega, \Phi \rangle)\\
&\quad \mid\ n \notin \mathit{Dom}(T) = \langle T, B, H, \theta, \omega, \Phi \rangle\\
&\quad \mid\ n \in \mathit{Dom}(T) = \langle T \setminus N, B' \setminus N, H' \setminus N, \theta' \setminus N, \omega, \Phi' \rangle\\
&\quad \textbf{where } N = [\![\{n'\}]\!]^{T}\\
&\qquad\qquad B' = B[n \mapsto B(n) + \textstyle\sum_{i \in N, i.k = n.k} B(i)]\\
&\qquad\qquad H' = H[n \mapsto H(n) + \textstyle\sum_{i \in N, i.k = n.k} H(i)]\\
&\qquad\qquad \theta' = \theta[n \mapsto \theta(n) \cup \textstyle\bigcup_{i \in N} \theta(i) \cup \{i.k \mid i \in N, i.k \neq n.k\}]\\
&\qquad\qquad \Phi' = \lambda k.\Phi[k \mapsto \Phi(k) \setminus N]
\end{aligned}$$

Fig. 5: Top-level parallel scheduler.

accumulating configuration updates ($\mathit{next}(\cdot)$ updates the current configuration by one event-step): $\langle\!\langle (n, e) : es \rangle\!\rangle^{c} = \langle\!\langle es \rangle\!\rangle^{\mathit{next}(n,e,c)}$. When no more events need processing, the configuration is returned $\langle\!\langle [\,] \rangle\!\rangle^{c} = c$.

Event Processing. Fig. 5 defines function $\mathit{next}(n, e, c)$, which takes a thread identifier n, the event e that thread n generated, the current configuration and outputs the configuration obtained by performing the thread's action. The empty event ϵ is trivial and leaves the state unchanged. Event $(n_1, \mathbf{fork}(\Delta, n_2, t, b, h))$ indicates that thread n_1 forks thread t with identifier n_2, sub-heap Δ, time budget b and maximum heap size h. The scheduler deducts these resources from the parent's budgets, i.e., $B' = B[n_1 \mapsto B(n_1) - b]$ and $H' = H[n_1 \mapsto H(n_1) - h]$ and assigns them to the child, i.e., $B'[n_2 \mapsto b]$ and $H'[n_2 \mapsto h]$.[16] The new child shares the core with the parent—it has no core capabilities i.e., $\theta' = \theta[n_2 \mapsto \varnothing]$—and so the core map is left unchanged. Lastly, the scheduler adds the child to the thread pool and initializes its state, i.e., $T[n_2 \mapsto (\Delta, n_2.\ell_{\mathrm{L}}, \varnothing \mid t, [\,])]$. The scheduler handles event $(n_1, \mathbf{spawn}(\Delta, n_2, t, K))$ similarly. The new thread t gets scheduled on core $n_2.k$, i.e., $\Phi[n_2.k \mapsto \langle n_2^{B_0} \rangle]$, where the thread takes all the time

[16] Notice that $|\Delta| < h$ by rule [FORK].

and memory resources of the core, i.e., $B[n_2 \mapsto B_0]$ and $H[n_2 \mapsto H_0]$, and extra core capabilities K, i.e., $\theta'[n_2 \mapsto K]$. For simplicity, we assume that all cores execute B_0 steps per-cycle and feature a memory of size H_0. Event $(n, \mathbf{kill}(n'))$ informs the scheduler that thread n wishes to kill thread n'. The scheduler leaves the global state unchanged if the parent thread has already been killed by the time this event is handled, i.e., when the guard $n \notin Dom(T)$ is true—the resources of the child n' will have been reclaimed by another ancestor. Otherwise, the scheduler collects the identifiers of the descendants of n' that are *alive* ($N = [\![\{ n' \}]\!]^T$)—they must be killed (and reclaimed) *transitively*. The set N is computed recursively by $[\![N]\!]^T$, using the thread pool T, i.e., $[\![\varnothing]\!]^T = \varnothing$, $[\![\{ n \}]\!]^T = \{ n \} \cup [\![T(n).N]\!]^T$ and $[\![N_1 \cup N_2]\!]^T = [\![N_1]\!]^T \cup [\![N_2]\!]^T$. The scheduler then increases the time and memory budget of the parent with the sum of the budget of all its descendants scheduled on the *same* core, i.e., $\sum_{i \in N, i.k = n.k} B(i)$ (resp. $\sum_{i \in N, i.k = n.k} H(i)$)—descendants running on other cores do not share those resources. The scheduler reassigns to the parent thread their core capabilities, which are split between capabilities explicitly assigned but not in use, i.e., $\bigcup_{i \in N} \theta(i)$ and core capabilities assigned and in use by running threads, i.e., $\{ i.k \mid i \in N, i.k \neq n.k \}$. Lastly, the scheduler removes the killed threads from each core, written $\Phi(i) \setminus N$, by pruning the leaves containing killed threads and reassigning their leftover time budget to their parent, see Appendix A.2 for details.

5 Security Guarantees

In this section we show that LIO$_{\text{PAR}}$ satisfies a strong security condition that ensures timing-agreement of threads and rules out timing covert channels. In §5.1, we describe our proof technique based on *term erasure*, which has been used to verify security guarantees of functional programming languages [31], IFC libraries [7, 17, 59, 64**?**]), and an IFC runtime system [62]. In §5.2, we formally prove security, i.e., *timing-sensitive non-interference*, a strong form of non-interference [14], inspired by Volpano and Smith [67]—to our knowledge, it is considered here for the first time in the context of parallel runtime systems. Works that do not address external timing channels [62, 65] normally prove *progress-sensitive* non-interference, wherein the number of execution steps of a program may differ in two runs based on a secret. This condition is insufficient in the parallel setting: both public and secret threads may step simultaneously on different cores and any difference in the number of execution steps would introduce *external* and *internal* timing attacks. Similar to previous works on secure multi-threaded systems [38, 53], we establish a *strong* low-bisimulation property of the parallel scheduler, which guarantees that configurations that are indistinguishable to the attacker remain such and execute in lock-step. Theorem 1 and Corollary 1 use this property to ensure that any two related parallel programs execute in exactly the same number of steps.

5.1 Erasure Function

The term erasure technique relies on an *erasure function*, written $\varepsilon_L(\cdot)$, which rewrites secret data above the attacker's level L to special term $\bullet$, in all the syntactic categories: values, terms, heaps, stacks, global states and configurations.[17] Once the erasure function is defined, the core of the proof technique consists of proving an essential *commutativity*

[17] For ease of exposition, we use the two-point lattices $\{ L, H \}$, where $H \not\sqsubseteq L$ is the only disallowed flow. Neither our proofs nor our model rely on this particular lattice.

relationship between the erasure function and reduction steps: given a step $c \hookrightarrow c'$, there must exist a reduction that *simulates* the original reduction between the erased configurations, i.e., $\varepsilon_L(c) \hookrightarrow \varepsilon_L(c')$. Intuitively, if the configuration c leaked secret data while stepping to c', that data would be classified as public in c' and thus would remain in $\varepsilon_L(c')$— but such secret data would be erased by $\varepsilon_L(c)$ and the property would not hold. The erasure function leaves ground values, e.g., $()$, unchanged and on most terms it acts homomorphically, e.g., $\varepsilon_L(t_1\ t_2) = \varepsilon_L(t_1)\ \varepsilon_L(t_2)$. The interesting cases are for labeled values, thread configurations, and resource maps. The erasure function removes the content of secret labeled values, i.e., $\varepsilon_L(\mathit{Labeled}\ H\ t^\circ) = \mathit{Labeled}\ H\ \bullet$, and erases the content recursively otherwise, i.e., $\varepsilon_L(\mathit{Labeled}\ L\ t^\circ) = \mathit{Labeled}\ L\ \varepsilon_L(t)^\circ$. The state of a thread is erased per-component, homomorphically if the program counter label is public, i.e., $\varepsilon_L(\Delta, L, N, \mid t, S) = (\varepsilon_L(\Delta), L, N \mid \varepsilon_L(t), \varepsilon_L(S))$, and in full otherwise, i.e., $\varepsilon_L(\Delta, H, N, \mid t, S) = (\bullet, \bullet, \bullet \mid \bullet, \bullet)$. We give the full definition in Appendix B.

Resource Erasure. Resources must also be appropriately erased in order to satisfy the simulation property, as LIO$_{\text{PAR}}$ manages resources explicitly. The erasure function should *preserve* information about the resources (e.g., time, memory, and core capabilities) of *public threads*, since the attacker can explicitly assign resources (e.g., with fork and swap) and measure them (e.g., with size). But what about the resources of secret threads? One might think that such information is secret and thus it should be erased—intuitively, a thread might decide to assign, say, half of its time budget to its secret child depending on secret information. However, public threads can also assign (public) resources to a secret thread when forking: even though these resources currently belong to the secret child, they are *temporary*—the public parent might reclaim them later. Thus, we cannot associate the sensitivity of the resources of a thread with its program counter label when resources are managed *hierarchically*, as in LIO$_{\text{PAR}}$. Instead, we associate the security level of the resources of a secret thread with the sensitivity of its parent: the resources of a secret thread are *public* information whenever the program counter label of the parent is public and *secret* information otherwise. Furthermore, since resource reclamation is transitive, the erasure function cannot discard secret resources, but must rather redistribute them to the hierarchically closest set of public resources, as when *kill*ing them.

Time Budget. First, we project the identifiers of *public* threads from the thread pool T: $\mathit{Dom}_L(T) = \{ n_L \mid n \in \mathit{Dom}(T) \wedge T(n).pc \equiv L \}$, where notation n_L indicates that the program counter label of thread n is public. Then, the set $P = \bigcup_{n \in \mathit{Dom}_L(T)} \{ n \} \cup T(n).N$ contains the identifiers of all the public threads and their immediate children.[18] The resources of threads $n \in P$ are public information. However, the program counter label of a thread $n \in P$ is not necessarily public, as explained previously. Hence P can be disjointly partitioned by program counter label: $P = P_L \cup P_H$, where $P_L = \{ n_L \mid n \in P \}$ and $P_H = \{ n_H \mid n \in P \}$. Erasure of the budget map then proceeds on this partition, leaving the budget of the public threads untouched, and summing the budget of their secret children threads to the budgets of their descendants, which are instead omitted. In symbols, $\varepsilon_L(B) = B_L \cup B_H$, where $B_L = \{ n_L \mapsto B(n_L) \mid n_L \in P_L \}$ and $B_H = \{ n_H \mapsto B(n_H) + \sum_{i \in [\![\{ n_H \}]\!]^T} B(i) \mid n_H \in P_H \}$.

[18] The id of the spinning thread on each free core is also public, i.e., $\circ_k \in P$ for $k \in \{1 \mathinner{..} \kappa\}$.

Queue Erasure. The erasure of core queues follows the same intuition, preserving public and secret threads $n \in P$ and trimming all other secret threads $n_H \notin P$. Since queues annotate thread ids with their residual time budgets, the erasure function must reassign the budgets of all *secret* threads $n'_H \notin P$ to their closest ancestor $n \in P$ on the same core. The ancestor $n \in P$ could be either (i) another *secret* thread on the same core, i.e., $n_H \in P$, or, (ii) the spinning thread of that core, $\circ \in P$ if there is no other thread $n \in P$ on that core—the difference between these two cases lies on whether the original thread n' was *forked* or *spawned* on that core. More formally, if the queue contains no thread $n \in P$, then the function replaces the queue altogether with the spinning thread and returns the residual budgets of the threads to it, i.e., $\varepsilon_L(Q) = \langle \circ^B \rangle$ if $n_i \notin P$ and $B = \sum b_i$, for each leaf $Q[\langle n_i^{b_i} \rangle]$ where $i \in \{1 \,..\, |Q|\}$. Otherwise, the core contains at least a thread $n_H \in P$ and the erasure function returns the residual time budget of its secret descendants, i.e., $\varepsilon_L(Q) = Q{\downarrow_L}$ by combining the effects of the following mutually recursive functions:

$$
\begin{aligned}
&\langle n^b \rangle{\downarrow_L} = \langle n^b \rangle && \langle n_{1H}^{b_1} \rangle \curlyvee \langle n_{2H}^{b_2} \rangle = \langle n_{1H}^{b_1+b_2} \rangle \\
&\langle Q_1, Q_2 \rangle{\downarrow_L} = (Q_1{\downarrow_L}) \curlyvee (Q_2{\downarrow_L}) && Q_1 \curlyvee Q_2 = \langle Q_1, Q_2 \rangle
\end{aligned}
$$

The interesting case is $\langle n_{1H}^{b_1} \rangle \curlyvee \langle n_{2H}^{b_2} \rangle$, which reassigns the budget of the child (the right leaf $\langle n_{2H}^{b_2} \rangle$) to the parent (the left leaf $\langle n_{1H}^{b_1} \rangle$), by rewriting the subtree into $\langle n_{1H}^{b_1+b_2} \rangle$.

5.2 Timing-Sensitive Non-Interference

The proof of timing-sensitive non-interference relies on two fundamental properties, i.e., *determinancy* and *simulation* of parallel reductions. Determinancy requires that the reduction relation is deterministic.

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

The equivalence in the statement denotes alpha-equivalence, i.e., up to the choice of variable names. We now show that the parallel scheduler preserves L-equivalence of parallel configurations.

Definition 1 (L-equivalence). *Two configurations c_1 and c_2 are indistinguishable from an attacker at security level L, written $c_1 \approx_L c_2$, if and only if $\varepsilon_L(c_1) \equiv \varepsilon_L(c_2)$.*

Proposition 2 (Parallel Simulation). *Given a parallel reduction step $c \hookrightarrow c'$, then $\varepsilon_L(c) \hookrightarrow \varepsilon_L(c')$.*

By combining *determinism* (Proposition 1) and *parallel simulation* (Proposition 2), we prove *progress-insensitive non-interference*, which assumes progress of both configurations.

Proposition 3 (Progress-Insensitive Non-Interference). *If $c_1 \hookrightarrow c'_1$, $c_2 \hookrightarrow c'_2$ and $c_1 \approx_L c_2$, then $c'_1 \approx_L c'_2$.*

In order to lift this result to be timing-sensitive, we first prove *time sensitive progress*. Intuitively, if a valid [19] configuration steps then any low equivalent parallel configuration also steps.

[19] A configuration is valid if satisfies several basic properties, e.g., it does not contain special term $\bullet$. See Appendix B for details

Proposition 4 (Time-Sensitive Progress). *Given a valid configuration c_1 and a parallel reduction step $c_1 \hookrightarrow c_1'$ and $c_1 \approx_L c_2$, then there exists c_2', such that $c_2 \hookrightarrow c_2'$.*

Using progress-insensitive non-interference, i.e., Proposition 3 and time-sensitive progress, i.e., Proposition 4 in combination, we obtain a *strong* L-bisimulation property between configurations and prove *timing-sensitive non-interference*.

Theorem 1 (Timing-Sensitive Non-Interference). *For all valid configurations c_1 and c_2, if $c_1 \hookrightarrow c_1'$ and $c_1 \approx_L c_2$, then there exists a configuration c_2', such that $c_2 \hookrightarrow c_2'$ and $c_1' \approx_L c_2'$.*

The following corollary instantiates the timing-sensitive non-interference security theorem for a given LIO$_{\text{PAR}}$ parallel program, that explicitly rules out leaks via timing channels. In the following, the notation $\hookrightarrow_u$, denotes u parallel reduction steps, as usual.

Corollary 1. *Given a well-typed LIO$_{\text{PAR}}$ program t of type $\mathit{Labeled}\ \tau_1 \to \mathit{LIO}\ \tau_2$ and two closed secrets $t_1{}^{\circ}, t_2{}^{\circ} :: \tau_1$, let $s_i = ([\,], L, \varnothing, |\ t\ (\mathit{Labeled}\ H\ t_i{}^{\circ}), [\,])$, $c_i = (T_i, B, H, \theta, 0, \Phi_i)$, where $T_i = [n_L \mapsto s_i, \circ_j \mapsto s_{\circ}]$, $B = [n_L \mapsto B_0, \circ_j \mapsto 0]$, $H = [n_L \mapsto H_0, \circ_j \mapsto H_0]$, $\theta = [n_L \mapsto \{2\,..\,\kappa\}, \circ_j \mapsto \varnothing]$, $\Phi_i = [1 \mapsto \langle s_i \rangle, 2 \mapsto \langle \circ_2 \rangle, ..., \kappa \mapsto \langle \circ_\kappa \rangle]$, for $i \in \{1, 2\}$, $j \in \{1\,..\,\kappa\}$ and thread identifier n_L such that $n.k = 1$ and $n.cl = H$. If $c_1 \hookrightarrow_u c_1'$, then there exists configuration c_2', such that $c_2 \hookrightarrow_u c_2'$ and $c_1' \approx_L c_2'$.*

To conclude, we show that the *timing-sensitive* security guarantees of LIO$_{\text{PAR}}$ extend to concurrent *single-core* programs by instantiating Corollary 1 with $\kappa = 1$.

6 Limitations

Implementation. Implementing LIO$_{\text{PAR}}$ is a serious undertaking that requires a major redesign of GHC's runtime system. Conventional runtime systems freely share resources among threads to boost performance and guarantee fairness. For instance, in GHC, threads share heap objects to save memory space and execution time (when evaluating expressions). In contrast, LIO$_{\text{PAR}}$ strictly partitions resources to enforce security—threads at different security labels cannot share heap objects. As a result, the GHC memory allocator must be adapted to isolate threads' private heap, so that allocation and collection can occur independently and in parallel. Similarly, the GHC "fair" round robin scheduler must be heavily modified to keep track of and manage threads' time budget, to preemptively perform a context switch when their time slice is up.

Programming model. Since resource management is explicit, building applications atop LIO$_{\text{PAR}}$ introduces new challenges—the programmer must explicitly choose resource bounds for each thread. If done poorly, threads can spend excessive amounts of time sitting idle when given too much CPU time, or garbage collecting when not given enough heap space. The problem of tuning resource allocation parameters is not unique to LIO$_{\text{PAR}}$—Yang and Mazières' [70] propose to use GHC profiling mechanisms to determine heap size while the real-time garbage collector by Henriksson [16] required the programmer to specify the worst case execution time, period, and worst-case allocation of each high-priority thread. Das and Hoffmann [9] demonstrate a more automatic approach—they apply machine learning techniques to statically determine upper bounds on execution time and heap usage of OCaml programs. Similar techniques could be applied to LIO_{PAR} in order to determine the most efficient resource partitions. We further remark that this challenge is not unique to real-time systems or LIO_{PAR}; choosing privacy parameters in differential privacy shares many similarities [22, 30]. Even though LIO_{PAR} programming model might seem overly restrictive, we consider it appropriate for certain classes of applications (e.g., web applications and certain embedded systems). To further simplify programming with LIO_{PAR}, we intend to introduce privileges (and thus declassification) similar to LIO [12, 59] or COWL [60]. Floating-label systems such as LIO and LIO_{PAR} often suffer from *label creep* issues, wherein the current label gets tainted to a point where the computation cannot perform any useful side-effects [58]. Similar to concurrent LIO [57], LIO_{PAR} relies on primitive —fork— to address label creep[20], but, at the cost of a restricted floating-label mechanisms, LIO_{PAR} provides also parallel execution, garbage collection, and APIs for heap statistics, elapsed time, and kill.

7 Related work

There is substantial work on language-level IFC systems [10, 15, 21, 35, 44, 51, 52, 55, 58, 71, 72]. Our work builds on these efforts in several ways. Firstly, LIO_{PAR} extends the concurrent LIO IFC system [55] with parallelism—to our knowledge, this is the first *dynamic* IFC system to support parallelism and address the internalization of external timing channels. Previous static IFC systems implicitly allow for parallelism, e.g., Muller and Chong's [42], several works on IFC π-calculi [19, 20, 26], and Rafnsson et al. [50] recent foundations for composable timing-sensitive interactive systems. These efforts, howerver, do not model runtime system resource management. Volpano and Smith [67] enforce a timing agreement condition, similar to ours, but for a static concurrent IFC system. Mantel et al. [37] and Li et al. [32] prove non-interference for static, concurrent systems, using rely-guarantee reasoning.

Unlike most of these previous efforts, our hierarchical runtime system also eliminates classes of resource-based external timing channels, such as memory exhaustion and garbage collection. Pedersen and Askarov [47], however, were the first to identify automatic memory management to be a source of covert channels for IFC systems and demonstrate the feasibility of attacks against both V8 and the JVM. They propose a sequential static IFC language with labeled-partitioned memory and a label-aware timing-sensitive garbage collector, which is vulnerable to *external timing* attacks and satisfies only *termination-insensitive* non-interference.

Previous work on language-based systems—namely [36, 70]—identify memory retention and memory exhaustion as a source of denial-of-service (DOS) attacks. Memory retention and exhaustion can also be used as covert channels. In addressing those covert channels, LIO_{PAR} also addresses the DOS attacks outlined by these efforts. Indeed, our work generalizes Yang and Mazières' [70] region-based allocation framework with region-based garbage collection and hierarchical scheduling.

[20] Sequential LIO addresses label creep through primitive —toLabeled—, which executes a computation (that may raise the current label) in a separate context and restores the current label upon its termination. LIO_{PAR} does not feature —toLabeled—, because the primitive opens the termination covert-channel and thus is not timing-sensitive [**?**].

Our LIO$_{\mathrm{PAR}}$ design also borrows ideas from the secure operating system community. Our explicit hierarchical memory management is conceptually similar to HiStar's container abstraction [73]. In HiStar, containers—subject to quotas, i.e., space limits—are used to hierarchically allocate and deallocate objects. LIO$_{\mathrm{PAR}}$ adopts this idea at the language-level and automates the allocation and reclamation. Moreover, we hierarchically partition CPU-time; Zeldovich et al. [73], however, did observe that their container abstraction can be repurposed to enforce CPU quotas.

Deterland [68] splits time into ticks to address internal timing channels and mitigate external timing ones. Deterland builds on Determinator [4], an OS that executes parallel applications deterministically and efficiently. LIO$_{\mathrm{PAR}}$ adopts many ideas from these systems—both the deterministic parallelism and ticks (semantic steps)—to the language-level. Deterministic parallelism at the language-level has also been explored previous to this work [28, 29, 39], but, different from these efforts, LIO$_{\mathrm{PAR}}$ also hierarchically manages resources to eliminate classes of external timing channels.

Fabric [34, 35] and DStar [74] are distributed IFC systems. Though we believe that our techniques would scale beyond multi-core systems (e.g., to data centers), LIO$_{\mathrm{PAR}}$ will likely not easily scale to large distributed systems like Fabric and DStar. Different from Fabric and DStar, however, LIO$_{\mathrm{PAR}}$ addresses both internal and external timing channels that result from running code in parallel.

Our hierarchical resource management approach is not unique—other countermeasures to external timing channels have been studied. Hu [23], for example, mitigates both timing channels in the VAX/VMM system [33] using "fuzzy time"—an idea recently adopted to browsers [27]. Askarov et al.'s [2] mitigate external timing channels using predicative black-box mitigation, which delays events and thus bound information leakage. Rather than using noise as in the fuzzy time technique, however, they predict the schedule of future events. Some of these approaches have also been adopted at the language-level [47, 55, 75]. We find these techniques largely orthogonal: they can be used alongside our techniques to mitigate timing channels we do not eliminate.

Real-time systems—when developed with garbage collected languages [3, 5, 6, 16]—face similar challenges as this work. Blelloch and Cheng [6] describe a real-time garbage collector (RTGC) for multi-core programs with *provable* resource bounds—LIO$_{\mathrm{PAR}}$ *enforces* resource bounds instead. A more recent RTGC created by Auerbach et al. [3] describes a technique to "tax" threads into contributing to garbage collection as they utilize more resources. Henricksson [16] describes a RTGC capable of enforcing hard and soft deadlines, once given upper bounds on space and time resources used by threads. Most similarly to LIO$_{\mathrm{PAR}}$, Pizlo et al. [49] implement a hierarchical RTGC algorithm that independently collects partitioned heaps.

8 Conclusion

Language-based IFC systems built atop off-the-shelf runtime systems are vulnerable to resource-based external-timing attacks. When these systems are extended with thread parallelism the attacks become yet more vicious—they can be carried out internally. We presented LIO$_{\mathrm{PAR}}$, the design of the first dynamic IFC hierarchical runtime system that support deterministic parallelism and eliminates both resource-based internal- and external-timing covert channels. To our knowledge, LIO$_{\mathrm{PAR}}$ is the first parallel system to satisfy progress- and time-sensitive non-interference.

Bibliography

[1] Askarov, A., Hunt, S., Sabelfeld, A., Sands, D.: Termination-insensitive noninterference leaks more than just a bit. In: European symposium on research in computer security. pp. 333–348. Springer (2008)

[2] Askarov, A., Zhang, D., Myers, A.C.: Predictive black-box mitigation of timing channels. In: Proceedings of the 17th ACM conference on Computer and communications security. pp. 297–307. ACM (2010)

[3] Auerbach, J., Bacon, D.F., Cheng, P., Grove, D., Biron, B., Gracie, C., McCloskey, B., Micic, A., Sciampacone, R.: Tax-and-spend: democratic scheduling for real-time garbage collection. In: Proceedings of the 8th ACM international conference on Embedded software. pp. 245–254. ACM (2008)

[4] Aviram, A., Weng, S.C., Hu, S., Ford, B.: Efficient system-enforced deterministic parallelism. Communications of the ACM **55**(5), 111–119 (2012)

[5] Baker Jr, H.G.: List processing in real time on a serial computer. Communications of the ACM **21**(4), 280–294 (1978)

[6] Blelloch, G.E., Cheng, P.: On bounding time and space for multiprocessor garbage collection. In: ACM SIGPLAN Notices. vol. 34, pp. 104–117. ACM (1999)

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

[8] Buiras, P., Russo, A.: Lazy programs leak secrets. In: Nordic Conference on Secure IT Systems. pp. 116–122. Springer (2013)

[9] Das, A., Hoffmann, J.: Ml for ml: Learning cost semantics by experiment. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 190–207. Springer (2017)

[10] Fernandes, E., Paupore, J., Rahmati, A., Simionato, D., Conti, M., Prakash, A.: FlowFence: Practical data protection for emerging IoT application frameworks. In: USENIX Security Symposium. pp. 531–548 (2016)

[11] Ge, Q., Yarom, Y., Cock, D., Heiser, G.: A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. Journal of Cryptographic Engineering pp. 1–27 (2016)

[12] Giffin, D.B., Levy, A., Stefan, D., Terei, D., Mazières, D., Mitchell, J., Russo, A.: Hails: Protecting data privacy in untrusted web applications. Journal of Computer Security **25**

[13] Giffin, D.B., Levy, A., Stefan, D., Terei, D., Mazières, D., Mitchell, J., Russo, A.: Hails: Protecting data privacy in untrusted web applications. In: Proceedings of the Symposium on Operating Systems Design and Implementation. USENIX (2012)

[14] Goguen, J.A., Meseguer, J.: Unwinding and inference control. pp. 75–86 (Apr 1984)

[15] Hedin, D., Birgisson, A., Bello, L., Sabelfeld, A.: JSFlow: Tracking information flow in JavaScript and its APIs. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing. pp. 1663–1671. ACM (2014)

[16] Henriksson, R.: Scheduling Garbage Collection in Embedded Systems. Ph.D. thesis, Department of Computer Science (1998)

[17] Heule, S., Stefan, D., Yang, E.Z., Mitchell, J.C., Russo, A.: IFC inside: Retrofitting languages with dynamic information flow control. In: Proceedings of the Conference on Principles of Security and Trust. Springer (2015)

[18] Hewitt, C., Bishop, P., Steiger, R.: A universal modular actor formalism for artificial intelligence. In: Proceedings of the 3rd International Joint Conference on Artificial Intelligence. pp. 235–245. IJCAI'73, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1973), `http://dl.acm.org/citation.cfm?id=1624775.1624804`

[19] Honda, K., Vasconcelos, V., Yoshida, N.: Secure information flow as typed process behaviour. In: European Symposium on Programming. pp. 180–199. Springer (2000)

[20] Honda, K., Yoshida, N.: A uniform type structure for secure information flow. ACM Transactions on Programming Languages and Systems (TOPLAS) **29**(6), 31 (2007)

[21] Hritcu, C., Greenberg, M., Karel, B., Pierce, B.C., Morrisett, G.: All your ifcexception are belong to us. In: Security and Privacy (SP), 2013 IEEE Symposium on. pp. 3–17. IEEE (2013)

[22] Hsu, J., Gaboardi, M., Haeberlen, A., Khanna, S., Narayan, A., Pierce, B.C., Roth, A.: Differential privacy: An economic method for choosing epsilon. In: Proceedings of the 2014 IEEE 27th Computer Security Foundations Symposium. pp. 398–410. CSF '14, IEEE Computer Society, Washington, DC, USA (2014). https://doi.org/10.1109/CSF.2014.35, `https://doi.org/10.1109/CSF.2014.35`

[23] Hu, W.M.: Reducing timing channels with fuzzy time. Journal of computer security **1**(3-4), 233–254 (1992)

[24] Jones, P., L, S.: Implementing lazy functional languages on stock hardware: The spineless tagless g-machine. Journal of Functional Programming **2**, 127–202 (July 1992)

[25] Kemmerer, R.A.: Shared resource matrix methodology: An approach to identifying storage and timing channels. ACM Transactions on Computer Systems (TOCS) **1**(3), 256–277 (1983)

[26] Kobayashi, N.: Type-based information flow analysis for the π-calculus. Acta Informatica **42**(4-5), 291–347 (2005)

[27] Kohlbrenner, D., Shacham, H.: Trusted browsers for uncertain times. In: USENIX Security Symposium. pp. 463–480 (2016)

[28] Kuper, L., Newton, R.R.: Lvars: lattice-based data structures for deterministic parallelism. In: Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing. pp. 71–84. ACM (2013)

[29] Kuper, L., Todd, A., Tobin-Hochstadt, S., Newton, R.R.: Taming the parallel effect zoo: Extensible deterministic parallelism with lvish. ACM SIGPLAN Notices **49**(6), 2–14 (2014)

[30] Lee, J., Clifton, C.: How much is enough? choosing ϵ for differential privacy. In: Proceedings of the 14th International Conference on Information Security. pp. 325–340. ISC'11, Springer-Verlag, Berlin, Heidelberg (2011), `http://dl.acm.org/citation.cfm?id=2051002.2051032`

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

[32] Li, X., Mantel, H., Tasch, M.: Taming message-passing communication in compositional reasoning about confidentiality. In: Programming Languages and Systems - 15th Asian Symposium, APLAS 2017, Suzhou, China, November 27-29, 2017, Proceedings. pp. 45–66 (2017). https://doi.org/10.1007/978-3-319-71237-6_3, `https://doi.org/10.1007/978-3-319-71237-6_3`

[33] Lipner, S., Jaeger, T., Zurko, M.E.: Lessons from VAX/SVS for high-assurance VM systems. IEEE Security & Privacy **10**(6), 26–35 (2012)

[34] Liu, J., Arden, O., George, M.D., Myers, A.C.: Fabric: Building open distributed systems securely by construction. Journal of Computer Security **25**(4-5), 367–426 (2017)

[35] Liu, J., George, M.D., Vikram, K., Qi, X., Waye, L., Myers, A.C.: Fabric: A platform for secure distributed computation and storage. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. ACM (2009)

[36] Liu, J., Myers, A.C.: Defining and enforcing referential security. In: International Conference on Principles of Security and Trust. pp. 199–219. Springer (2014)

[37] Mantel, H., Sands, D., Sudbrock, H.: Assumptions and guarantees for compositional noninterference. In: 2011 IEEE 24th Computer Security Foundations Symposium. pp. 218–232 (June 2011). https://doi.org/10.1109/CSF.2011.22

[38] Mantel, H., Sabelfeld, A.: A unifying approach to the security of distributed and multi-threaded programs. J. Comput. Secur. **11**(4), 615–676 (Jul 2003), `http://dl.acm.org/citation.cfm?id=959088.959094`

[39] Marlow, S., Newton, R., Peyton Jones, S.: A monad for deterministic parallelism. ACM SIGPLAN Notices **46**(12), 71–82 (2012)

[40] Marlow, S., Peyton Jones, S.: Making a fast curry: push/enter vs. eval/apply for higher-order languages. Journal of Functional Programming **16**(4-5), 415–449 (2006). https://doi.org/10.1017/S0956796806005995

[41] McCullough, D.: Specifications for multi-level security and a hook-up. In: 1987 IEEE Symposium on Security and Privacy(SP). vol. 00, p. 161 (April 1987). https://doi.org/10.1109/SP.1987.10009, `doi.ieeecomputersociety.org/10.1109/SP.1987.10009`

[42] Muller, S., Chong, S.: Towards a practical secure concurrent language. In: Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming Languages, Systems, Languages, and Applications. pp. 57–74. ACM Press, New York, NY, USA (Oct 2012)

[43] Myers, A.C., Zheng, L., Zdancewic, S., Chong, S., Nystrom, N.: Jif 3.0: Java information flow (July 2006), `http://www.cs.cornell.edu/jif`

[44] Nadkarni, A., Andow, B., Enck, W., Jha, S.: Practical DIFC enforcement on android. In: USENIX Security Symposium. pp. 1119–1136 (2016)

[45] North, S.C., Reppy, J.H.: Concurrent garbage collection on stock hardware. In: Conference on Functional Programming Languages and Computer Architecture. pp. 113–133. Springer (1987)

[46] Parker, J.L.: LMonad: Information flow control for haskell web applications. Ph.D. thesis, University of Maryland, College Park (2014)

[47] Pedersen, M.V., Askarov, A.: From trash to treasure: timing-sensitive garbage collection. In: Proceedings of the 38th IEEE Symposium on Security and Privacy. IEEE (2017)

[48] Percival, C.: Cache missing for fun and profit (2005)

[49] Pizlo, F., Hosking, A.L., Vitek, J.: Hierarchical real-time garbage collection. In: Proceedings of the 2007 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems. pp. 123–133. LCTES '07, ACM, New York, NY, USA (2007). https://doi.org/10.1145/1254766.1254784, `http://doi.acm.org/10.1145/1254766.1254784`

[50] Rafnsson, W., Jia, L., Bauer, L.: Timing-sensitive noninterference through composition. In: Maffei, M., Ryan, M. (eds.) Principles of Security and Trust. pp. 3–25. Springer Berlin Heidelberg, Berlin, Heidelberg (2017)

[51] Roy, I., Porter, D.E., Bond, M.D., McKinley, K.S., Witchel, E.: Laminar: Practical fine-grained decentralized information flow control, vol. 44. ACM (2009)

[52] Russo, A.: Functional pearl: Two can keep a secret, if one of them uses haskell. In: ACM SIGPLAN Notices. vol. 50, pp. 280–288. ACM (2015)

[53] Sabelfeld, A., Sands, D.: Probabilistic noninterference for multi-threaded programs. In: Proceedings of the 13th IEEE Workshop on Computer Security Foundations. pp. 200–. CSFW '00, IEEE Computer Society, Washington, DC, USA (2000), `http://dl.acm.org/citation.cfm?id=794200.795151`

[54] Sestoft, P.: Deriving a lazy abstract machine. J. Funct. Program. **7**(3), 231–264 (May 1997). https://doi.org/10.1017/S0956796897002712, `http://dx.doi.org/10.1017/S0956796897002712`

[55] Stefan, D., Russo, A., Buiras, P., Levy, A., Mitchell, J.C., Mazières, D.: Addressing covert termination and timing channels in concurrent information flow systems. In: International Conference on Functional Programming (ICFP). ACM SIGPLAN (September 2012)

[56] Stefan, D., Russo, A., Mazières, D., Mitchell, J.C.: Disjunction category labels. In: Nordic Conference on Security IT Systems (NordSec). Springer (October 2011)

[57] Stefan, D., Russo, A., Mazières, D., Mitchell, J.C.: Flexible dynamic information flow control in the presence of exceptions. Journal of Functional Programming (2016), original submisison in 2012

[58] Stefan, D., Russo, A., Mazières, D., Mitchell, J.C.: Flexible dynamic information flow control in the presence of exceptions. Journal of Functional Programming **27**

(2017)

[59] Stefan, D., Russo, A., Mitchell, J.C., Mazières, D.: Flexible dynamic information flow control in Haskell. In: Haskell Symposium. ACM SIGPLAN (September 2011)

[60] Stefan, D., Yang, E.Z., Marchenko, P., Russo, A., Herman, D., Karp, B., Mazières, D.: Protecting users by confining JavaScript with COWL. In: USENIX Symposium on Operating Systems Design and Implementation. USENIX Association (2014)

[61] Tsai, T.c., Russo, A., Hughes, J.: A library for secure multi-threaded information flow in haskell. In: Computer Security Foundations Symposium, 2007. CSF'07. 20th IEEE. pp. 187–202. IEEE (2007)

[62] Vassena, M., Breitner, J., Russo, A.: Securing concurrent lazy programs against information leakage. In: 30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017. pp. 37–52 (2017). https://doi.org/10.1109/CSF.2017.39, `https://doi.org/10.1109/CSF.2017.39`

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

[64] Vassena, M., Russo, A.: On formalizing information-flow control libraries. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security. pp. 15–28. PLAS '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2993600.2993608, `http://doi.acm.org/10.1145/2993600.2993608`

[65] Vassena, M., Russo, A., Buiras, P., Waye, L.: Mac a verified static information-flow control library. Journal of Logical and Algebraic Methods in Programming (2017). https://doi.org/https://doi.org/10.1016/j.jlamp.2017.12.003, `http://www.sciencedirect.com/science/article/pii/S235222081730069X`

[66] Vila, P., Köpf, B.: Loophole: Timing attacks on shared event loops in chrome. In: USENIX Security Symposium (2017)

[67] Volpano, D., Smith, G.: Eliminating covert flows with minimum typings. In: Proceedings of the 10th IEEE Workshop on Computer Security Foundations. pp. 156–. CSFW '97, IEEE Computer Society, Washington, DC, USA (1997), `http://dl.acm.org/citation.cfm?id=794197.795081`

[68] Wu, W., Zhai, E., Wolinsky, D.I., Ford, B., Gu, L., Jackowitz, D.: Warding off timing attacks in deterland. In: Conference on Timely Results in Operating Systems, Monterey, CS, US (2015)

[69] Yang, E.Z., Mazières, D.: Dynamic space limits for haskell. SIGPLAN Not. **49**(6), 588–598 (Jun 2014). https://doi.org/10.1145/2666356.2594341, `http://doi.acm.org/10.1145/2666356.2594341`

[70] Yang, E.Z., Mazières, D.: Dynamic space limits for haskell. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM (2014)

[71] Yang, J., Hance, T., Austin, T.H., Solar-Lezama, A., Flanagan, C., Chong, S.: Precise, dynamic information flow for database-backed applications. In: ACM SIGPLAN Notices. vol. 51, pp. 631–647. ACM (2016)

[72] Yang, J., Yessenov, K., Solar-Lezama, A.: A language for automatically enforcing privacy policies. In: ACM SIGPLAN Notices. vol. 47, pp. 85–96. ACM (2012)

[73] Zeldovich, N., Boyd-Wickizer, S., Kohler, E., Mazières, D.: Making information flow explicit in histar. In: Proceedings of the 7th symposium on Operating systems design and implementation. pp. 263–278. USENIX Association (2006)

[74] Zeldovich, N., Boyd-Wickizer, S., Mazieres, D.: Securing distributed systems with information flow control. In: NSDI. vol. 8, pp. 293–308 (2008)

[75] Zhang, D., Askarov, A., Myers, A.C.: Predictive mitigation of timing channels in interactive systems. In: Proceedings of the 18th ACM conference on Computer and communications security. pp. 563–574. ACM (2011)

A Full Calculus

Context Rules. In Fig. 6, for completeness, we report the remaining context rules of sequential LIO$_{\mathrm{PAR}}$—they rules simply evaluate their arguments by pushing (popping) the appropriate continuation on the stack.

IF$_1$
$$(\Delta, pc, N \mid \textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3, S) \rightsquigarrow_\mu (\Delta, pc, N \mid t_1, \textbf{then } t_2 \textbf{ else } t_3 : S)$$

IF$_2$
$$(\Delta, pc, N \mid \mathit{True}, \textbf{then } t_2 \textbf{ else } t_3 : S) \rightsquigarrow_\mu (\Delta, pc, N \mid t_2, S)$$

IF$_3$
$$(\Delta, pc, N \mid \mathit{False}, \textbf{then } t_2 \textbf{ else } t_3 : S) \rightsquigarrow_\mu (\Delta, pc, N \mid t_3, S)$$

FORK$_1$
$$(\Delta, pc, N \mid \mathit{fork}\ t_1\ t_2\ t_3\ t_4\ t_5, S) \rightsquigarrow_\mu (\Delta, pc, N \mid t_1, \mathit{fork}\ t_2\ t_3\ t_4\ t_5 : S)$$

FORK$_2$
$$(\Delta, pc, N \mid \ell, \mathit{fork}\ t_2\ t_3\ t_4\ t_5 : S) \rightsquigarrow_\mu (\Delta, pc, N \mid t_2, \mathit{fork}\ \ell\ t_3\ t_4\ t_5 : S)$$

FORK$_3$
$$(\Delta, pc, N \mid \ell_2, \mathit{fork}\ \ell_1\ t_3\ t_4\ t_5 : S) \rightsquigarrow_\mu (\Delta, pc, N \mid t_3, \mathit{fork}\ \ell_1\ \ell_2\ t_4\ t_5 : S)$$

FORK$_4$
$$(\Delta, pc, N \mid h, \mathit{fork}\ \ell_1\ \ell_2\ t_4\ t_5 : S) \rightsquigarrow_\mu (\Delta, pc, N \mid t_4, \mathit{fork}\ \ell_1\ \ell_2\ h\ t_5 : S)$$

SPAWN$_1$
$$(\Delta, pc, N \mid \mathit{spawn}\ t_1\ t_2\ t_3\ t_4\ t_5, S) \rightsquigarrow_\mu (\Delta, pc, N \mid t_1, \mathit{spawn}\ t_2\ t_3\ t_4\ t_5 : S)$$

SPAWN$_2$
$$(\Delta, pc, N \mid \ell, \mathit{spawn}\ t_2\ t_3\ t_4\ t_5 : S) \rightsquigarrow_\mu (\Delta, pc, N \mid t_2, \mathit{spawn}\ \ell\ t_3\ t_4\ t_5 : S)$$

SPAWN$_3$
$$(\Delta, pc, N \mid \ell_2, \mathit{spawn}\ \ell_1\ t_3\ t_4\ t_5 : S) \rightsquigarrow_\mu (\Delta, pc, N \mid t_3, \mathit{spawn}\ \ell_1\ \ell_2\ t_4\ t_5 : S)$$

SPAWN$_4$
$$(\Delta, pc, N \mid K, \mathit{spawn}\ \ell_1\ \ell_2\ t_4\ t_5 : S) \rightsquigarrow_\mu (\Delta, pc, N \mid t_4, \mathit{spawn}\ \ell_1\ \ell_2\ K\ t_5 : S)$$

WAIT$_1$
$$(\Delta, pc, N \mid \mathit{wait}\ t, S) \rightsquigarrow_\mu (\Delta, pc, N \mid t, \mathit{wait} : S)$$

KILL$_1$
$$(\Delta, pc, N \mid \mathit{kill}\ t, S) \rightsquigarrow_\mu (\Delta, pc, N \mid t, \mathit{kill} : S)$$

Fig. 6: Context rules of sequential LIO$_{\mathrm{PAR}}$.

Stuck Conditions. Fig. 7 formally defines the conditions used in rule [STUCK], which identify a thread as *stuck*. When adding garbage collection, i.e., rule [APP-GC], we also remove the condition $\mathit{MaxHeapStuck}(s)$ from rule [STUCK]—such condition triggers an automatic garbage collection cycle that reduces via rule [GC].

FORKSTUCK

$$\frac{\begin{array}{c} T(n) = (\_, pc, \_ \mid \ell_2, \mathit{fork}\ \ell_1\ t : S) \\ \ell_1 \not\sqsubseteq \ell_2 \lor pc \not\sqsubseteq \ell_1 \lor H(n) = h_1 + h_2 \qquad |\Delta| > h_1 \lor |\Delta'| > h_2 \end{array}}{\mathit{ForkStuck}(n, H, T)}$$

UNLABELSTUCK

$$\frac{\begin{array}{c} T(n) = (\_, pc, N \mid \mathit{Labeled}\ \ell\ t, \mathit{unlabel} : S) \\ pc \sqcup \ell \not\sqsubseteq n.cl \lor \exists\, n \in N \,.\, pc \sqcup \ell \not\sqsubseteq n.pc \end{array}}{\mathit{UnlabelStuck}(n, T)}$$

WAITSTUCK

$$\frac{\begin{array}{c} T(n) = (\_, pc, \_ \mid n', \mathit{wait} : S) \\ n.cl \not\sqsubseteq pc \lor \mathit{eval}(T)(n') = n' \not\twoheadrightarrow (\_, \_, \_ \mid v, [\,]) \end{array}}{\mathit{WaitStuck}(n, T)}$$

VALUESTUCK

$$\frac{s = (\_, \_, \_ \mid v, [\,])}{\mathit{ValueStuck}(s)}$$

MAXHEAPSIZE

$$\frac{\begin{array}{c} s = (\Delta, \_, \_ \mid t_1\ t_2, \_) \\ |\Delta| = h \end{array}}{\mathit{MaxHeapSize}(s, h)}$$

SPAWNSTUCK

$$\frac{\begin{array}{c} s = (\Delta, pc, N \mid k, \mathit{spawn}_{K_1}\ \ell_1\ \ell_2\ t : S) \\ pc \not\sqsubseteq \ell_1 \lor \ell_1 \not\sqsubseteq \ell_2 \lor k \cup K_1 \not\subseteq K \end{array}}{\mathit{SpawnStuck}(s, K)}$$

KILLSTUCK

$$\frac{s = (\Delta, pc, N \mid n, \mathit{kill} : S) \qquad n \notin N}{\mathit{KillStuck}(s)}$$

RECEIVESTUCK

$$\frac{s = (\Delta, pc, N, [\,] \mid \mathit{receive}, S)}{\mathit{ReceiveStuck}(s)}$$

Fig. 7: Stuck conditions.

Reachable Variables and Term Closure. Fig. 8 presents two operations that compute and substitute free variables. Fig. 8a formally defines the set of transitively reachable free variables for all syntactic categories, i.e., stacks, continuations and message queues. The function simply computes the free variables by induction on the structure of terms, stacks, continuation and message queues recursively. Notice that labeled values are closed and thus the function returns the empty set for them, i.e., $fv^*(\mathit{Labeled}\ \ell\ t^\circ, \Delta) = \varnothing$. Fig. 8b defines a closure function, i.e., $\Delta^*_B(t)$, which, given an open term t with bound variables B and a heap Δ, computes the corresponding closed term t° by recursively substituting free variables to close the term. When the function traverses a lambda expression, i.e., $\lambda x.t$, it recurs in the body and adds variable x to the set of bound variables, i.e., $\Delta^*_{B\cup\{x\}}(t)$. When the function finds a free variable x, it looks it up in the heap and repeats the process by closing the corresponding thunk, i.e., $\Delta^*(\Delta(x))$. When omitted, the set of bound variables is empty, i.e., $\Delta^*(t) = \Delta^*_\varnothing(t)$. Notice that the recursion is well-funded because our heap does not contain cyclic definitions (the syntax of the calculus does not feature recursive let bindings and therefore every term can be closed).

$$
\begin{aligned}
fv^*(x, \Delta) &= \{x\} \cup fv^*(\Delta(x), \Delta)\\
fv^*(\lambda x.t, \Delta) &= fv^*(t, \Delta) \setminus \{x\}\\
fv^*(t_1\ t_2, \Delta) &= fv^*(t_1, \Delta) \cup fv^*(t_2, \Delta)\\
fv^*(\mathit{Labeled}\ \ell\ t^\circ, \Delta) &= \varnothing\\[1em]
fv^*([\,], \Delta) &= \varnothing\\
fv^*(C : S, \Delta) &= fv^*(C, \Delta) \cup fv^*(S, \Delta)\\[1em]
fv^*(x, \Delta) &= \{x\} \cup fv^*(\Delta(x), \Delta)\\
fv^*(\gg\!\!= t, \Delta) &= fv^*(t, \Delta)\\
fv^*(\mathit{label}\ t, \Delta) &= fv^*(t, \Delta)\\[1em]
fv^*([\,], \Delta) &= \varnothing\\
fv^*(t \lhd ts, \Delta) &= fv^*(t, \Delta) \cup fv^*(ts, \Delta)
\end{aligned}
$$

(a) Transitively-reachable free variables (excerpt).

$$
\Delta^*_B(x) = \begin{cases} x & \text{if } x \in B\\ \Delta^*(\Delta(x)) & \text{if } x \notin B \end{cases}
$$

$$
\Delta^*_B(t_1\ t_2) = \Delta^*_B(t_1)\ \Delta^*_B(t_2)
$$

$$
\Delta^*_B(\lambda x.t) = \Delta^*_{B\cup\{x\}}(t)
$$

(b) Term closure (excerpt).

Fig. 8: Free variables manipulation.

A.1 Thread Synchronization and Communication

LIO$_{\mathrm{PAR}}$ features primitive $wait\ n'$, which allows a thread to wait on the result of some thread n', see Rule [WAIT] in Fig. 9. If the thread has terminated, i.e., its term is a value and the stack of continuations is empty, its value is returned to the waiting thread. The condition $n'.cl \sqsubseteq pc$ ensures that the waited-upon thread has an appropriate clearance (indicating that its label must be low enough to observe its result) for security reasons. In all other cases, the [STUCK] rule applies via the [WAITSTUCK] condition.

Furthermore, LIO$_{\mathrm{PAR}}$ features also thread communication primitives, which enable a programming model akin to the actor model [18]. In particular, primitive $send\ n\ t$ enables *asynchronous* best-effort delivery of message t to thread n—there are no guarantees that the message will be delivered. Note that rule [SEND$_2$] simply generates an event that instructs the top-level parallel scheduler to deliver a message, which then might get dropped for security reasons or otherwise. Conversely, rule [RECEIVE] executes *synchronous* primitive $receive$, which inspects the thread's messages queue ts and extracts the next message—if the queue is empty, the thread gets stuck. [21] The rule generates event $\mathbf{send}(n_2, t)$, so that the parallel scheduler delivers message t to thread n_2, by means of function $next(\cdot)$. The scheduler drops the message if the receiver is dead, i.e., $n_2 \notin Dom(T)$, or if the sender is dead, i.e., $n_2 \notin Dom(T)$. If the receiver has sufficient memory budget, i.e., $|\Delta'_2| < H(n_2)$ and is at least as sensitive as the sender, i.e., $pc_1 \sqsubseteq pc_2$, then the scheduler delivers and enqueues the message in the message queue, i.e., $ts_2 \triangleright t$, or drops it otherwise.[22]

[21] The receiver thread will also get stuck if the type of the message does not match the expected type. We are not concerned with the type-safety of primitive $receive$, which could be recovered with dynamic typing.

[22] This feature makes threads vulnerable to Denial of Service (DOS) attacks, in which the attacker floods a thread with messages until it runs out of memory. We could restore security by adding message integrity or by pre-allocating a memory budget for queue messages.

$$\begin{array}{lrl} \text{Message Queue} & ts & ::= [\,] \mid t \triangleleft ts \\ \text{State} & s & ::= (\Delta, pc, N, ts \mid t, S) \end{array}$$

WAIT

$$\frac{T(n) = (\Delta, pc, N, ts \mid n', \mathit{wait} : S) \qquad n'.cl \sqsubseteq pc \qquad T(n') = (\_, \_, \_, \_ \mid \mathit{return}\ v, [\,]) \qquad s = (\Delta, pc, N, ts \mid v, S)}{Q[\langle n^{1+b} \rangle] \xrightarrow{(n,s,\epsilon)}_{\Sigma} Q[\langle n^{b} \rangle]}$$

SEND$_1$

$$(\Delta, pc, N, ts \mid \mathit{send}\ t_1\ t_2, S) \rightsquigarrow (\Delta, pc, N, ts, \mid t_1, \mathit{send}\ t_2 : S)$$

SEND$_2$

$$\frac{n \mapsto (\Delta, pc, N, ts \mid n', \mathit{send}\ t : S) \in \Sigma.T \qquad s = (\Delta, pc, N, ts \mid \mathit{return}\ (), S)}{Q[\langle n^{1+b} \rangle] \xrightarrow{(n,s,\mathbf{send}(n',t))}_{\Sigma} Q[\langle n^{b} \rangle]}$$

RECEIVE

$$\frac{n \mapsto (\Delta, pc, N, t \triangleleft ts \mid \mathit{receive}, S) \in \Sigma.T \qquad s = (\Delta, pc, N, ts \mid \mathit{return}\ t, S)}{Q[\langle n^{1+b} \rangle] \xrightarrow{(n,s,\epsilon)}_{\Sigma} Q[\langle n^{b} \rangle]}$$

$$\begin{array}{l} \mathit{next}(n_1, \mathbf{send}(n_2, t), \langle T, B, H, \theta, \Phi \rangle) \\ \quad \mid\ n_1 \notin \mathit{Dom}(T) \vee n_2 \notin \mathit{Dom}(T) = \langle T, B, H, \theta, \Phi \rangle \\ \quad \mid\ |\Delta_2'| < H(n_2) \wedge pc_1 \sqsubseteq pc_2 = \\ \qquad \langle T[n_2 \mapsto (\Delta_2', pc_2, N_2, ts_2 \triangleright t \mid t_2, S_2)], B, H, \theta, \Phi \rangle \\ \quad \mid\ \mathit{otherwise} = \langle T, B, H, \theta, \Phi \rangle \\ \quad \mathbf{where}\ (\Delta_1, pc_1, \_, \_ \mid \_, \_) = T(n_1) \\ \qquad\qquad (\Delta_2, pc_2, N_2, ts_2 \mid t_2, S_2) = T(n_2) \\ \qquad\qquad \Delta_2' = \Delta_2 \cup \{ x \mapsto \Delta_1(x) \mid x \in \mathit{fv}^*(t, \Delta_1) \} \end{array}$$

Fig. 9: Thread synchronization and communication primitives.

$$Q \setminus \varnothing = Q$$
$$Q \setminus (\{\, n \,\} \cup N) = \mathbf{delete}(Q, n) \setminus N$$

(a) $Q \setminus N$ removes nodes N from the tree Q.

$$\mathbf{delete}(\langle n^{b} \rangle, n) = \langle \circ^{0} \rangle$$

$$\mathbf{delete}(\langle \langle n^{b} \rangle \mid Q \rangle, n') = \begin{cases} \mathbf{add}(Q, b) & \text{if } n \equiv n' \\ \langle \langle n^{b} \rangle \mid \mathbf{delete}(Q, n') \rangle & \text{otherwise} \end{cases}$$

$$\mathbf{delete}(\langle Q \mid \langle n^{b} \rangle \rangle, n') = \begin{cases} \mathbf{add}(Q, b) & \text{if } n \equiv n' \\ \langle \mathbf{delete}(Q, n') \mid \langle n^{b} \rangle \rangle & \text{otherwise} \end{cases}$$

$$\mathbf{delete}(\langle Q_1 \mid Q_2 \rangle, n) = \begin{cases} \langle \mathbf{delete}(Q_1, n) \mid Q_2 \rangle & \text{if } n \in Q_1 \\ \langle Q_1, \mathbf{delete}(Q_2, n) \rangle & \text{if } n \in Q_2 \end{cases}$$

(b) $\mathbf{delete}(Q, n)$ removes node $n \in Q$.

$$\mathbf{add}(\langle n^{b_1} \rangle, b_2) = \langle n^{b_1 + b_2} \rangle \qquad \mathbf{add}(\langle Q_1 \mid Q_2 \rangle, b) = \langle \mathbf{add}(Q_1, b) \mid Q_2 \rangle$$

(c) $\mathbf{add}(Q, b)$ adds b to the budget of the leftmost thread.

Fig. 10: Functions that trim and readjust the budget in a queue tree.

A.2 Queue Pruning

When threads get killed, the parallel scheduler removes them from the core queue where they are scheduled. We write $Q \setminus N$, for the queue obtained by removing threads N from queue Q, by repeated calls to function $\mathbf{delete}(\cdot)$, see Fig. 10. Function $\mathbf{delete}(Q, n)$ locates thread n in the queue and trims its leaf—if the thread is alone in the queue, it is replaced with the dummy busy-waiting thread id, i.e., $\circ$, and the core becomes free. Note that the current budget of a killed thread is not discarded, but it is reassigned to its closest relative (either to the parent, the immediate child or the spinning thread), by means of function $\mathbf{add}(\cdot)$, for security reasons. Otherwise, a high thread could influence the schedule on its own core by forking a thread and giving up, say, half of its budget, and then killing it immediately afterwards. If the parent does not regain the child's current time budget, then nothing would run for the time the child was originally given, which could lead to information leakage, when executed in parallel with other cores.

$$\begin{array}{lll}
\text{Term} & t & ::= \cdots \mid \bullet \mid \mathit{label}_{L}\ t_1\ t_2 \mid \mathit{unlabel}_{L}\ t \\
 & & \mid \mathit{fork}_{L}\ t_1\ t_2\ t_3\ t_4\ t_5 \mid \mathit{spawnL}\ t_1\ t_2\ t_3\ t_4\ t_5 \\
\text{Cont.} & C & ::= \cdots \mid \mathit{label}_{L}\ t \mid \mathit{unlabel}_{L} \\
 & & \mid \mathit{fork}_{L}\ t_1\ t_2\ t_3\ t_4 \mid \mathit{spawnL}\ t_1\ t_2\ t_3\ t_4 \\
\text{Stack} & S & ::= \cdots \mid \bullet \\
\text{Queue} & ts & ::= \cdots \mid \bullet \\
\text{Event} & e & ::= \cdots \mid \mathbf{fork}_{L}(n, b, h) \mid \mathbf{spawn}_{L}(\Delta, n, K)
\end{array}$$

Fig. 11: Calculus with erased terms.

B Security Proofs

B.1 Two-Steps Erasure

During execution, a public thread might create secret data, often by elevating low data. For example, primitive $\mathit{label}\ \ell\ t$ labels a piece of data t with label ℓ in rule [LABEL$_2$] from Figure 2. Depending on the sensitivity of the label, the erasure function needs to rewrite t to either $\bullet$, if $\ell \equiv H$ or $\varepsilon_L(t)$ if $\ell \equiv L$, in order to respect the simulation property for rule [LABEL$_2$]. Unfortunately, the label is a runtime value, thus it might not be known when we apply the erasure function, e.g., in rule [LABEL$_1$]. *Two-steps erasure* is a technique that extends term erasure and simplify reasoning about such operations, especially when the decision of erasing data depends on the context or on runtime values [64]. In particular, this technique ensures that those operations commute under the erasure function by rewriting problematic primitives with new ad-hoc constructs, that erase terms at runtime, when sufficient information is available. Figure 11 extends the calculus with erasure-aware primitives. Fig. 12 and 16 define the erasure function for all the syntactic categories of LIO$_{\text{PAR}}$, which rewrites the problematic terms with the those extra primitives, which are reduced according to the rules in Fig. 13 and 15. Equipped with those extra primitives, we use term erasure with two-steps erasure to prove security of LIO$_{\text{PAR}}$ in the next section.

Example. The erasure function rewrites label to new construct label_{L}, which firstly evaluates the label via rule [LABEL$_{L1}$] and then erase the second argument as needed, when the value of the label is known in rules [LABEL$_{L2}$] and [LABEL$_{L3}$].

$$\varepsilon_L(()) = () \qquad \varepsilon_L(\ell) = \ell \qquad \varepsilon_L(\lambda x.e) = \lambda x.\varepsilon_L(e)$$

$$\varepsilon_L(\mathit{return}\ e) = \mathit{return}\ \varepsilon_L(e) \qquad \varepsilon_L(n) = n \qquad \varepsilon_L(k) = k$$

$$\varepsilon_L(\mathit{Labeled}\ \ell\ t^{\circ}) = \begin{cases} \mathit{Labeled}\ \ell\ \bullet & \text{if } \ell \not\sqsubseteq L \\ \mathit{Labeled}\ \ell\ \varepsilon_L(t)^{\circ} & \text{otherwise} \end{cases}$$

(a) Values.

$$\varepsilon_L(\Delta) = \{x \mapsto \varepsilon_L(\Delta(x)) \mid x \in \mathit{Dom}(\Delta)\} \qquad \varepsilon_L(x) = x$$

$$\varepsilon_L(t_1\ t_2) = \varepsilon_L(t_1)\ \varepsilon_L(t_2) \qquad \varepsilon_L(t_1 \gg\!= t_2) = \varepsilon_L(t_1) \gg\!= \varepsilon_L(t_2)$$

$$\varepsilon_L(\mathit{label}\ t_1\ t_2) = \mathit{label}_L\ \varepsilon_L(t_1)\ \varepsilon_L(t_2)$$

$$\varepsilon_L(\mathit{unlabel}\ t) = \mathit{unlabel}_L\ \varepsilon_L(t)$$

$$\varepsilon_L(\mathit{fork}\ t_1\ t_2\ t_3\ t_4\ t_5) = \mathit{fork}_L\ \varepsilon_L(t_1)\ \varepsilon_L(t_2)\ \varepsilon_L(t_3)\ \varepsilon_L(t_4)\ \varepsilon_L(t_5)$$

$$\varepsilon_L(\mathit{spawn}\ t_1\ t_2\ t_3\ t_4\ t_5) =$$
$$\mathit{spawn}_L\ \varepsilon_L(t_1)\ \varepsilon_L(t_2)\ \varepsilon_L(t_3)\ \varepsilon_L(t_4)\ \varepsilon_L(t_5)$$

$$\varepsilon_L(\mathit{wait}\ t) = \mathit{wait}\ \varepsilon_L(t)$$

(b) Heaps and terms.

$$\varepsilon_L([\,]) = [\,] \qquad \varepsilon_L(C : S) = \varepsilon_L(C) : \varepsilon_L(S) \qquad \varepsilon_L(x) = x$$

$$\varepsilon_L(\gg\!= t) = \gg\!= \varepsilon_L(t) \qquad \varepsilon_L(\mathit{label}\ t) = \mathit{label}_L\ \varepsilon_L(t)$$

$$\varepsilon_L(\mathit{unlabel}) = \mathit{unlabel}_L$$

$$\varepsilon_L(\mathit{fork}\ t_1\ t_2\ t_3\ t_4) = \mathit{fork}_L\ \varepsilon_L(t_1)\ \varepsilon_L(t_2)\ \varepsilon_L(t_3)\ \varepsilon_L(t_4)$$

$$\varepsilon_L(\mathit{spawn}\ t_1\ t_2\ t_3\ t_4) = \mathit{spawn}_L\ \varepsilon_L(t_1)\ \varepsilon_L(t_2)\ \varepsilon_L(t_3)\ \varepsilon_L(t_4)$$

$$\varepsilon_L(\mathit{wait}) = \mathit{wait}$$

(c) Stacks and continuations.

Fig. 12: Erasure for sequential LIO$_{\text{PAR}}$.

LABEL_{L1}

$(\Delta, pc, N, ts \mid label_L\ t_1\ t_2, S) \rightsquigarrow_\mu (\Delta, pc, N, ts \mid t_1, label_L\ t_2 : S)$

LABEL_{L2}

$$\frac{pc \sqsubseteq H \sqsubseteq \mu.cl}{(\Delta, pc, N, ts \mid H, label_L\ t : S) \rightsquigarrow_\mu (\Delta, pc, N, ts \mid return\ (Labeled\ H\ \bullet), S)}$$

LABEL_{L3}

$$\frac{pc \sqsubseteq L \sqsubseteq \mu.cl \qquad t^\circ = \Delta^*(t)}{(\Delta, pc, N, ts \mid L, label_L\ t : S) \rightsquigarrow_\mu (\Delta, pc, N, ts \mid return\ (Labeled\ L\ t^\circ), S)}$$

(UNLABEL_{L1})

$(\Delta, pc, N, ts \mid unlabel_L\ t, S) \rightsquigarrow_\mu (\Delta, pc, N, ts \mid t, unlabel_L : S)$

UNLABEL_{L2}

$$\frac{pc \sqcup L \sqsubseteq \mu.cl \qquad \forall\, n \in N\,.\, pc \sqcup L \sqsubseteq n.pc}{(\Delta, pc, N, ts \mid Labeled\ L\ t, unlabel_L : S) \rightsquigarrow_\mu (\Delta, pc \sqcup \ell, N, ts \mid return\ t, S)}$$

UNLABEL_{L3}

$$\frac{pc \sqcup H \sqsubseteq \mu.cl \qquad \forall\, n \in N\,.\, pc \sqcup L \sqsubseteq n.pc}{(\Delta, pc, N, ts \mid Labeled\ H\ t, unlabel_L : S) \rightsquigarrow_\mu (\bullet, \bullet, \bullet, \bullet \mid \bullet, \bullet)}$$

FORK_{L1}

$(\Delta, pc, N, ts \mid fork_L\ t_1\ t_2\ t_3\ t_4\ t_5, S) \rightsquigarrow_\mu (\Delta, pc, N, ts \mid t_1, fork_L\ t_2\ t_3\ t_4\ t_5 : S)$

FORK_{L2}

$(\Delta, pc, N, ts \mid \ell, fork_L\ t_2\ t_3\ t_4\ t_5 : S) \rightsquigarrow_\mu (\Delta, pc, N, ts \mid t_2, fork_L\ \ell\ t_3\ t_4\ t_5 : S)$

FORK_{L3}

$(\Delta, pc, N, ts \mid \ell_2, fork_L\ \ell_1\ t_3\ t_4\ t_5 : S) \rightsquigarrow_\mu (\Delta, pc, N, ts \mid t_3, fork_L\ \ell_1\ \ell_2\ t_4\ t_5 : S)$

FORK_{L4}

$(\Delta, pc, N, ts \mid h, fork_L\ \ell_1\ \ell_2\ t_4\ t_5 : S) \rightsquigarrow_\mu (\Delta, pc, N, ts \mid t_4, fork_L\ \ell_1\ \ell_2\ h\ t_5 : S)$

SPAWN_{L1}

$(\Delta, pc, N, ts \mid spawn_L\ t_1\ t_2\ t_3\ t_4\ t_5, S) \rightsquigarrow_\mu (\Delta, pc, N, ts \mid t_1, spawn_L\ t_2\ t_3\ t_4\ t_5 : S)$

SPAWN_{L2}

$(\Delta, pc, N, ts \mid \ell, spawn_L\ t_2\ t_3\ t_4\ t_5 : S) \rightsquigarrow_\mu (\Delta, pc, N, ts \mid t_2, spawn_L\ \ell\ t_3\ t_4\ t_5 : S)$

SPAWN_{L3}

$(\Delta, pc, N, ts \mid \ell_2, spawn_L\ \ell_1\ t_3\ t_4\ t_5 : S) \rightsquigarrow_\mu (\Delta, pc, N, ts \mid t_3, spawn_L\ \ell_1\ \ell_2\ t_4\ t_5 : S)$

SPAWN_{L4}

$(\Delta, pc, N, ts \mid K, spawn_L\ \ell_1\ \ell_2\ t_4\ t_5 : S) \rightsquigarrow_\mu (\Delta, pc, N, ts \mid t_4, spawn_L\ \ell_1\ \ell_2\ K\ t_5 : S)$

HOLE

$(\bullet, \bullet, \bullet, \bullet \mid \bullet, \bullet) \rightsquigarrow_\mu (\bullet, \bullet, \bullet, \bullet \mid \bullet, \bullet)$

Fig. 13: Sequential reduction rules for erased terms.

$$\varepsilon_L(n_L, s, e) = (n_L, \varepsilon_L(s), \varepsilon_L(e)) \qquad \varepsilon_L(n_H, s, e) = (n_H, \varepsilon_L(s), \epsilon) \qquad \varepsilon_L(\epsilon) = \epsilon$$

$$\varepsilon_L(\mathbf{kill}(n)) = \mathbf{kill}(n) \qquad \varepsilon_L(\mathbf{send}(n, t)) = \mathbf{send}(n, \varepsilon_L(t))$$

$$\varepsilon_L(\mathbf{spawn}(\Delta, n, t, K)) = \begin{cases} \mathbf{spawn}(\varepsilon_L(\Delta), n, \varepsilon_L(t), K) & \text{if } n.pc \sqsubseteq L \\ \mathbf{spawn}_L(\varepsilon_L(\Delta), n, K) & \text{otherwise} \end{cases}$$

$$\varepsilon_L(\mathbf{fork}(\Delta, n, t, b, h)) = \begin{cases} \mathbf{fork}(\varepsilon_L(\Delta), n, \varepsilon_L(t), b, h) & \text{if } n.pc \sqsubseteq L \\ \mathbf{fork}_L(b, h) & \text{otherwise} \end{cases}$$

$$\varepsilon_L(\Delta, pc, N, ts \mid t, S) = \begin{cases} (\bullet, \bullet, \bullet, \bullet \mid \bullet, \bullet) & \text{if } pc \not\sqsubseteq L \\ (\varepsilon_L(\Delta), pc, N, \varepsilon_L(ts) \mid \varepsilon_L(t), \varepsilon_L(S)) & \text{otherwise} \end{cases}$$

Fig. 14: Erasure for concurrent LIO$_{\text{PAR}}$.

FORK$_{L5}$

$$\frac{\begin{array}{c} \Sigma.T(n) = (\Delta, pc, N, ts \mid b_2, \mathit{fork}_{L}\ L\ \ell_{\mathrm{H}}\ h_2\ t : S) \\ pc\ \sqsubseteq\ L \qquad n' \leftarrow \mathrm{fresh}^{\mathit{TId}}(L, \ell_{\mathrm{H}}, n.k) \\ s = (\Delta, pc, \{n'\} \cup N, ts \mid \mathit{return}\ n', S) \qquad \Delta' = \{x \mapsto \Delta(x) \mid x \in \mathit{fv}^*(t, \Delta)\} \\ \Sigma.H(n) = h_1 + h_2 \qquad |\Delta| \leqslant h_1 \quad |\Delta'| \leqslant h_2 \end{array}}{Q[\langle n^{1+b_1+b_2}\rangle] \xrightarrow{(n,s,\mathbf{fork}(\Delta',n',t,b_2,h_2))}_{\Sigma} Q[\langle\langle n^{b_1}\rangle | \langle n'^{b_2}\rangle\rangle]}$$

FORK$_{L6}$

$$\frac{\begin{array}{c} \Sigma.T(n) = (\Delta, pc, N, ts \mid b_2\ \mathit{fork}_{L}\ H\ \ell_{\mathrm{H}}\ h_2\ t : S) \\ pc\ \sqsubseteq\ H \qquad n' \leftarrow \mathrm{fresh}^{\mathit{TId}}(H, \ell_{\mathrm{H}}, n.k) \\ s = (\Delta, pc, \{n'\} \cup N, \mid \mathit{return}\ n', S) \qquad \Delta' = \{x \mapsto \Delta(x) \mid x \in \mathit{fv}^*(t, \Delta)\} \\ \Sigma.H(n) = h_1 + h_2 \qquad |\Delta| \leqslant h_1 \quad |\Delta'| \leqslant h_2 \end{array}}{Q[\langle n^{1+b_1+b_2}\rangle] \xrightarrow{(n,s,\mathbf{fork}_{L}(n',b_2,h_2))}_{\Sigma} Q[\langle\langle n^{b_1}\rangle | \langle n'^{b_2}\rangle\rangle]}$$

SPAWN$_{L4}$

$$\frac{\begin{array}{c} \Sigma.T(n) = (\Delta, pc, N, ts \mid k, \mathit{spawn}_{L}\ L\ \ell_{\mathrm{H}}\ K_1\ t : S) \\ \Sigma.\theta(n) = \{k\} \cup K_1 \cup K_2 \qquad pc\ \sqsubseteq\ L \qquad n' \leftarrow \mathrm{fresh}^{\mathit{TId}}(L, \ell_{\mathrm{H}}, k) \\ s = (\Delta, pc, \{n'\} \cup N \mid \mathit{return}\ n', S) \qquad \Delta' = \{x \mapsto \Delta(x) \mid x \in \mathit{fv}^*(t, \Delta)\} \end{array}}{Q[\langle n^{1+b}\rangle] \xrightarrow{(n,s,\mathbf{spawn}(\Delta',n',t,K_1))}_{\Sigma} Q[\langle n^{b}\rangle]}$$

SPAWN$_{L5}$

$$\frac{\begin{array}{c} \Sigma.T(n) = (\Delta, pc, N, ts \mid k\ \mathit{spawn}_{L}\ H\ \ell_{\mathrm{H}}\ K_1\ t : S) \\ \Sigma.\theta(n) = \{k\} \cup K_1 \cup K_2 \qquad pc\ \sqsubseteq\ H \qquad n' \leftarrow \mathrm{fresh}^{\mathit{TId}}(H, \ell_{\mathrm{H}}, k) \\ s = (\Delta, pc, \{n'\} \cup N \mid \mathit{return}\ n', S) \qquad \Delta' = \{x \mapsto \Delta(x) \mid x \in \mathit{fv}^*(t, \Delta)\} \end{array}}{Q[\langle n^{1+b}\rangle] \xrightarrow{(n,s,\mathbf{spawn}_{L}(\Delta',n',K_1))}_{\Sigma} Q[\langle n^{b}\rangle]}$$

Fig. 15: Concurrent reduction rules for erased terms.

$$Dom_{L}(T) = \{ n_{L} \mid n \in Dom(T) \wedge T(n).pc \equiv L \}$$

$$P = \bigcup_{n \in Dom_{L}(T)} \{ n \} \cup T(n).N$$

$$\varepsilon_{L}(T) = \{ n \mapsto \varepsilon_{L}(T(n)) \mid n \in P \} \qquad \varepsilon_{L}(\Phi) = \lambda k.\varepsilon_{L}(\Phi(k))$$

$$\varepsilon_{L}(\langle T, B, H, \theta, \Phi, \omega \rangle) = \langle \varepsilon_{L}(T), \varepsilon_{L}(B), \varepsilon_{L}(H), \varepsilon_{L}(\theta), \varepsilon_{L}(\Phi), \omega \rangle$$

(a) Thread maps, core maps and parallel configuration.

$$P_{L} = \{ n_{L} \mid n \in P \} \qquad P_{H} = \{ n_{H} \mid n \in P \}$$

$$\begin{aligned}
&\varepsilon_{L}(B) = B_{L} \cup B_{H}\\
&\quad \textbf{where } B_{L} = \{ n_{L} \mapsto B(n_{L}) \mid n_{L} \in P_{L} \}\\
&\qquad\qquad B_{H} = \{ n_{H} \mapsto B(n_{H}) + \textstyle\sum_{i \in [\![\{ n_{H} \}]\!]^{T}} B(i) \mid n_{H} \in P_{H} \}\\
&\varepsilon_{L}(H) = H_{L} \cup H_{H}\\
&\quad \textbf{where } H_{L} = \{ n_{L} \mapsto H(n_{L}) \mid n_{L} \in P_{L} \}\\
&\qquad\qquad H_{H} = \{ n_{H} \mapsto \textstyle\sum_{i \in [\![\{ n_{H} \}]\!]^{T}, n_{H}.k = i.k} H(i) \mid n_{H} \in P_{H} \}\\
&\varepsilon_{L}(\theta) = \theta_{L} \cup \theta_{H}\\
&\quad \textbf{where } \theta_{L} = \{ n_{L} \mapsto \theta(n_{L}) \mid n_{L} \in P_{L} \}\\
&\qquad\qquad \theta_{H} = \{ n_{H} \mapsto \theta(n_{H}) \cup K_1 \cup K_2 \mid n_{H} \in P_{H} \}\\
&\qquad\qquad K_1 = \textstyle\bigcup_{i \in [\![\{ n_{H} \}]\!]^{T}} \theta(i)\\
&\qquad\qquad K_2 = \{ i.k \mid i \in [\![\{ n_{H} \}]\!]^{T}, i.k \neq n_{H}.k \}
\end{aligned}$$

(b) Time, memory and core budget maps.

$$\begin{aligned}
&next(n_{1L}, \textbf{spawn}_{L}(\Delta, n_{2H}, K), \langle T, B, H, \theta, \omega, \Phi \rangle) = \langle T', B', H', \theta', \omega, \Phi' \rangle\\
&\quad \textbf{where } T' = T[n_{2H} \mapsto (\bullet, \bullet, \bullet, \bullet \mid \bullet, \bullet)]\\
&\qquad\qquad B' = B[n_{2H} \mapsto B_0]\\
&\qquad\qquad H' = H[n_{2H} \mapsto H_0]\\
&\qquad\qquad \theta' = \theta[n_{1L} \mapsto \theta(n_{1L}) \setminus \{ n_{2H}.k \} \cup K][n_{2H} \mapsto K]\\
&\qquad\qquad \Phi' = \Phi[n_{2H}.k \mapsto \langle n_{2H}^{B_0} \rangle]\\
&next(n_{1L}, \textbf{fork}_{L}(n_{2H}, b, h), \langle T, B, H, \theta, \omega, \Phi \rangle) = \langle T', B', H', \theta', \omega, \Phi \rangle\\
&\quad \textbf{where } T' = T[n_{2H} \mapsto (\bullet, \bullet, \bullet, \bullet \mid \bullet, \bullet)]\\
&\qquad\qquad B' = B[n_{1L} \mapsto B(n_{1L}) - b][n_{2H} \mapsto b]\\
&\qquad\qquad H' = H[n_{1L} \mapsto H(n_{1L}) - h][n_{2H} \mapsto h]\\
&\qquad\qquad \theta' = \theta[n_{2H} \mapsto \varnothing]
\end{aligned}$$

(c) Processing of erased events.

Fig. 16: Erasure for parallel LIO$_{\text{PAR}}$.

$$\textsc{Next}_1 \quad \langle n^b \rangle[\langle n^b \rangle] \qquad \textsc{Next}_2 \quad \frac{Q_1[\langle n_1^b \rangle] \quad b > 0}{\langle Q_1 \mid Q_2 \rangle[\langle n_1^b \rangle]} \qquad \textsc{Next}_3 \quad \frac{Q_1[\langle n_1^0 \rangle] \quad Q_2[\langle n_2^b \rangle]}{\langle Q_1 \mid Q_2 \rangle[\langle n_2^b \rangle]}$$

(a) Hierarchical Scheduling.

$$\varepsilon_L(Q) = \begin{cases} \langle \circ^0 \rangle & \text{if } Q \cap P = \varnothing \\ Q \downarrow_L & \text{otherwise} \end{cases}$$

$$\langle n^b \rangle \downarrow_L = \langle n^b \rangle \qquad \langle n_{1H}^{b_1} \rangle \curlyvee \langle n_{2H}^{b_2} \rangle = \langle n_{1H}^{b_1+b_2} \rangle$$
$$\langle Q_1, Q_2 \rangle \downarrow_L = (Q_1 \downarrow_L) \curlyvee (Q_2 \downarrow_L) \qquad Q_1 \curlyvee Q_2 = \langle Q_1, Q_2 \rangle$$

(b) Core erasure.

Fig. 17: Core queues.

B.2 Lemmas

We now prove a number of lemmas auxiliary to our security proofs. Fig. 17a formalizes the hierarchical scheduling policy and Fig. 17b repeats the erasure function for core queues, where $Q \cap P = \varnothing$ abbreviates the predicate $\forall\, i \in \{1\,..\,|Q|\}. Q[\langle n_i^{b_i} \rangle] \wedge n_i \notin P$, i.e., the queue contains only secret threads, whose resources are not observable.

Lemma 1. *If $Q[\langle n^0 \rangle]$, then one of the following holds:*

1. *If $n \in P$, then $\varepsilon_L(Q)[\langle n^0 \rangle]$*
2. *If $n \notin P$, then there exists $n' \in P$ such that $\varepsilon_L(Q)[\langle n'^0 \rangle]$*

Proof. First, $Q[\langle n^0 \rangle]$ implies that all threads in queue Q have budget 0. Then, since $\varepsilon_L(Q)$ simply redistribute the residual budgets of the secret threads on the queue, any thread scheduled by $\varepsilon_L(Q)$ will have budget 0 intuitively. Formally, we have two cases depending on whether $n \in P$ or not.

1. By induction on the scheduling policy. Case [Next$_1$] is trivial, case [Next$_2$] is impossible and case [Next$_3$] follows by induction. If $n \equiv \circ$ the lemma is trivial, i.e., $\varepsilon_L(\langle \circ^0 \rangle) = \langle \circ^0 \rangle \downarrow_L = \langle \circ^0 \rangle$.
2. If $Q \cap P = \varnothing$, then $\varepsilon_L(Q) = \langle \circ^0 \rangle$. If $Q \cap P \neq \varnothing$, we perform induction. Case [Next$_1$] and [Next$_2$] are absurd, in case [Next$_3$], we inspect the result of $(Q_1 \downarrow_L) \curlyvee (Q_2 \downarrow_L)$. If that is a leaf, e.g., $\langle n_H^0 \rangle$, then $n_H \in P_H$ (erasure reassigns the budgets of the threads in Q_1 and Q_2 to the closest ancestor in $Q \cap P$) and the lemma follows by rule [Next$_1$]. Otherwise, we inductively apply Lemma 1 to $Q_1[\langle n_1^0 \rangle]$ and Lemma 1.2 to $Q_2[\langle n^0 \rangle]$.

Lemma 2. *If $Q[\langle n_L^b \rangle]$, then $\varepsilon_L(Q)[\langle n_L^b \rangle]$.*

Proof. The core scheduler runs *public* thread n_L, thus $n_L \in P$ and $\varepsilon_L(Q) = Q\downarrow_L$. Then the proof follows by induction on the scheduling policy $Q[\langle n_L^b\rangle]$, observing in case [NEXT$_2$] and [NEXT$_3$] that $\langle Q_1, Q_2\rangle\downarrow_L = \langle Q_1\downarrow_L, Q_2\downarrow_L\rangle$, because either Q_1 or Q_2 contains public thread n_L and using Lemma 1 in case [NEXT$_3$].

Lemma 3. Proof. *If $Q[\langle n_H^{1+b}\rangle]$ and $n_H \in P$, then, there exists b', such that $b \leqslant b'$, $\varepsilon_L(Q)[\langle n_H^{1+b'}\rangle]$.*

Since $n_H \in P$, then $Q \cap P \neq \varnothing$ and $\varepsilon_L(Q) = Q\downarrow_L$ and we proceed by induction. Case [NEXT$_1$] is trivial. In case [NEXT$_2$], we perform case analysis on the result of $(Q_1\downarrow_L) \curlyvee (Q_2\downarrow_L)$, if that is a branch, i.e., $\langle Q_1\downarrow_L, Q_2\downarrow_L\rangle$, then the lemma follows by induction on $Q_1[\langle n_H^{1+b}\rangle]$, otherwise, $Q_1\downarrow_L = \langle n_H^{1+b}\rangle$ and $Q_2\downarrow_L = \langle {n'}_H^{b'}\rangle$, for some other secret thread n', with leftover budget b', which then gets collapsed into the budget of its ancestor n, i.e., $\langle n_H^{1+b}\rangle \curlyvee \langle {n'}_H^{b'}\rangle = \langle n_H^{1+b+b'}\rangle$. The lemma then follows by applying rule [NEXT$_2$] to rule [NEXT$_1$], i.e., scheduling $\langle n_H^{1+b+b'}\rangle$. The same line of reasoning applies to case [NEXT$_3$], where the public ancestor, say $n'_L \in P$, of secret thread $n_H \in P$ is in the left branch, i.e., $Q_1[\langle {n'}_L^0]\rangle$, thus $(Q_1\downarrow_L) \curlyvee (Q_2\downarrow_L)$ equals to $\langle Q_1\downarrow_L, Q_2\downarrow_L\rangle$. Then, the lemma follows by applying Lemma 1 to $Q_1[\langle {n'}_L^0]\rangle$ and induction on $(Q\downarrow_L)[\langle n_H^{1+n}]$.

Lemma 4. *If $Q[\langle n_H^{1+b}\rangle]$, $n_H \notin P$ and $Q \cap P \neq \varnothing$, then there exists $n'_H \in Q \cap P$ and $b' \geqslant b$, such that $\varepsilon_L(Q)[\langle {n'}_H^{1+b'}\rangle]$.*

Proof. Intuitively, we need to find in core Q, the closest ancestor $n'_H \in Q \cap P$ of secret thread n_H— there exists one threads form a hierarchy and $Q \cap P \neq \varnothing$—and show that the ancestor gets scheduled in the erased queue and that erasure function redistributes the residual budget to it. We do that by induction on $Q[\langle n_H^{1+b}\rangle]$. Case [NEXT$_1$] contradicts the hypothesis $Q \cap P \neq \varnothing$, hence the lemma is vacuously true. Case [NEXT$_2$] follows by induction in the left sub-queue, i.e., $Q_1[\langle n_H^{1+b}\rangle]$. Intuitively, the ancestor of n_H can only be in the left subtree, thus $Q_1 \cap P \neq \varnothing$ and $\langle Q_1, Q_2\rangle\downarrow_L = \langle Q_1\downarrow_L, Q_2\downarrow_L\rangle$. [23] In case [NEXT$_3$], we inspect the result of $(Q_1\downarrow_L) \curlyvee (Q_2\downarrow_L)$. If that is a leaf, by Lemma 1 there exists a *secret* thread n_{1H} with zero residual budget, such that $Q_1\downarrow_L = \langle n_{1H}^0\rangle$ and $Q_2\downarrow_L = \langle n_{2H}^{1+b_2}\rangle$ for some other thread $n_{2H} \notin Q \cap P$ with left-over budget $b_2 \geqslant b$ (since core erasure does *not* discard left-over budgets). [24] Then, the erased queue is $\langle n_{1H}^{1+b_1+b_2}\rangle$, and the lemma follows by rule [NEXT$_1$]. If $(Q_1\downarrow_L) \curlyvee (Q_2\downarrow_L)$ is a branch, then either we apply Lemma 1 and induction, if $Q_2 \cap P \neq \varnothing$, or derive a contradiction, otherwise. Intuitively, if $Q_2 \cap P = \varnothing$, then $Q_1 \cap P \neq \varnothing$ and either (i) all the threads in the left branch are secret, i.e., $Q_1 \subseteq P_H$, which contradicts the fact that $(Q_1\downarrow_L) \curlyvee (Q_2\downarrow_L)$ is a branch, or (ii) there exists a public thread $n'_L \in Q_1$, and we can show that $n_H \in P_H$, contradicting the second hypothesis of the lemma.

Lemma 5. *Properties of the erasure function:*

1. $|\varepsilon_L(\Delta)| \equiv |\Delta|$

[23] The public parent thread cannot be in the right branch, i.e., Q_2, because parents are moved to the left branch when forking.

[24] Either $n_{2H} \equiv n_H$, or it is the oldest ancestor of n_H on the core.

2. $fv^*(\varepsilon_L(\Delta), \varepsilon_L(t)) \equiv fv^*(\Delta, t)$
3. $\varepsilon_L(t_1 \; [x \; / \; t_2]) = \varepsilon_L(t_1)$

Proof. By straightforward induction on the arguments. The second lemma relies on secret labeled values being *closed*, that is $fv^*(\Delta, \mathit{Labeled}\ H\ t^\circ)) = \varnothing$ because t is closed, and $fv^*(\varepsilon_L(\Delta), \varepsilon_L(\mathit{Labeled}\ H\ t^\circ)) = fv^*(\varepsilon_L(\Delta), \mathit{Labeled}\ H\ \bullet) = \varnothing$. If we allow open labeled terms, then the lemma does *not* hold, e.g., $fv^*(\Delta, \mathit{Labeled}\ H\ x) = \{x\} \not\equiv \varnothing = fv^*(\varepsilon_L(\Delta), \mathit{Labeled}\ H\ \bullet)$.

Lemma 6. *For all time budget maps B, memory budget maps H, core capabilities maps θ, core queues Q, thread pool T and public threads $n \in P$, let $N = \{n\}$, then the following hold:*

- $\sum_{i \,\in\, [\![N]\!]^T, i.k = n.k} B(i) = \sum_{j \,\in\, [\![N]\!]^{\varepsilon_L(T)}, j.k = n.k} \varepsilon_L(B)(j)$
- $\sum_{i \,\in\, [\![N]\!]^T, i.k = n.k} H(i) = \sum_{j \,\in\, [\![N]\!]^{\varepsilon_L(T)}, j.k = n.k} \varepsilon_L(H)(j)$
- $\bigcup_{i \,\in\, [\![N]\!]^T} \theta(i) = \bigcup_{i \,\in\, [\![N]\!]^{\varepsilon_L(T)}} \varepsilon_L(\theta)(i)$
- $\{i.k \mid i \in [\![N]\!]^T, i.k \neq n.k\} = \{i.k \mid i \in [\![N]\!]^{\varepsilon_L(T)}, i.k \neq n.k\}$.
- $\varepsilon_L(Q \setminus [\![N]\!]^T) = \varepsilon_L(Q) \setminus [\![N]\!]^{\varepsilon_L(T)}$.

Proof. This lemma relates the observable parts of the budget maps and the erasure function, and ensures that budget erasure returns *all* the secret resources, i.e., the resources allocated by the secret descendants of thread $n \in P$, regardless of their number. Intuitively, the left-hand side of the equation involves an arbitrary number of those secret threads and the right-hand none, because the erasure function removes them from the thread pool map. More precisely, we partition the descendants of thread $n \in P$ in two groups, based on whether the attacker can observe their resources, formally: $[\![N]\!]^T = X \cup Y$, such that $X \subseteq P$ and and $Y \not\subseteq P$. We observe that the erased thread map only contains only the threads with visible resources, i.e., $[\![N]\!]^{\varepsilon_L(T)} = X$, since $Dom(\varepsilon_L(T)) = P$, that is erasure removes the secret threads contained in Y. Then, we further partition set X in two sets depending on the security level of the threads, i.e., $X = X_L \cup X_H$, where $X_L \subseteq P_L$ and $X_H \subseteq P_H$. First, we observe that the erasure function leaves the budgets of public threads *unchanged*, i.e., $\sum_{(i_L \,\in\, X_L)} B(i_L) = \sum_{(i_L \,\in\, X_L)} B_L(i_L)$, where $\varepsilon_L(B) = B_L \cup B_H$ from Figure 16b. Then, we only need to show that $\sum_{(i_H \,\in\, X_H \cup Y)} B(i_H) = \sum_{(i_H \,\in\, X_H)} B_H(i_H)$. The equality follows by two observations. Firstly, both sides sum the budget of the secret threads in X_H, i.e., $B(i_H)$ for $i_H \in X_H$. Secondly, the remaining secret threads $i_H \in Y \not\subseteq P$ are descendants of some secret thread $n_H \in X_H \subseteq P$ ($X \cup Y = [\![N]\!]^T$), and therefore their budget is accounted for on the right-hand side in the summation $\sum_{i \,\in\, [\![\{n_H\}]\!]^T} B(i)$. The same line of reasoning applies for memory, core capabilities budgets and core queues.

The next lemma ensures that processing any event e generated by a *secret* thread changes only the secret parts of the global state. L-equivalence of configurations (and all the other syntactic categories) is defined as the kernel of the erasure function, i.e., $c \approx_L c'$ iff $\varepsilon_L(c) \equiv \varepsilon_L(c')$, see Definition 1.

Lemma 7. *If* $next(n_H, e, c) = c'$ *then* $c \approx_L c'$.

Proof. By case analysis on event e. Case ϵ is trivial. Rule [SPAWN] ensures that the child thread is at least as sensitive as the parent n_H, hence if $e = \mathbf{spawn}(\Delta, n_2, t, K)$, then n_2 is secret and furthermore $n_2 \notin P$, hence $c \approx_L c'$, because the global state c' changes only in parts that are not observable by the attacker, i.e., $T \approx_L T[n_2 \mapsto (\Delta, n_2.pc, \varnothing, [\,] \mid t, [\,])]$, $T \approx_L T[n_2 \mapsto s]$, $B \approx_L B[n_2 \mapsto B_0]$ and $H \approx_L H[n_2 \mapsto H_0]$, because erasure filters out the new secret binding $n_2 \notin P$. Furthermore, if the parent thread has observable resources, i.e., $n_H \in P$, then erasure reassigns the core capabilities of the child to the parent, i.e., $\theta[n_H \mapsto \theta(n_H) \setminus \{n_2.k\} \cup K][n_2 \mapsto K] \approx_L \theta$ (Figure 16b). Lastly, the core maps are L-equivalent, i.e., $\Phi \approx_L \Phi[n_2.k \mapsto \langle n_2^{B_0} \rangle]$, because $\Phi(n_2.k) \approx_L \langle n_2^{B_0} \rangle$ and $\varepsilon_L(\langle n_2^{B_0} \rangle) = \langle \circ^0 \rangle = \varepsilon_L(\Phi)(n_2.k)$, since $n_2 \notin P$ and $n_2.k$ is *free* in Φ (thread n_1 could have not spawned on that core otherwise). Case $\mathbf{fork}(\Delta, n_2, t, b, h)$ is similar. Case $\mathbf{kill}(n')$ is trivial, if $n' \notin Dom(T)$. Otherwise, we observe that $n' \notin P$ and so are its descendants, i.e., $i \notin P$ where $i \in N$ and $N = [\![\{ n' \}]\!]^T$, and the changes to the budget maps involve only *secret* threads. If the parent thread has observable resources, i.e., $n_H \in P$, then the new budget maps are also indistinguishable to the attacker, e.g., $B[n \mapsto B(n) + \sum_{i \in N, i.k = n.k} B(i)] \approx_L B$. The final core map is L-equivalent to the initial map, i.e., $\Phi(k) \approx_L \Phi(k) \setminus N$ for $k \in \{1 \,..\, \kappa\}$, because N contains only *secret* threads and their left-over budget is reassigned to their closest ancestor in P. Lastly, we prove case $\mathbf{send}(n_2, t)$ by case analysis on the guards of function $next(\cdot)$. The lemma follows trivially in the first and in the third case—the global state does not change because the message is dropped. In the second case, the message gets delivered in the message queue of thread n_2, under the condition that $pc_1 \sqsubseteq pc_2$, i.e., $pc_2 \equiv H$. If $n_2 \notin P$, erasure drops its configuration from the thread pool and we obtain L-equivalence. If $n_2 \in P_H$, then $\varepsilon_L(T)(n_2) = (\bullet, \bullet, \bullet, \bullet \mid \bullet, \bullet)$ and the update does not change the thread pool, because $\bullet \triangleright t \equiv \bullet$ and $\bullet \cup \Delta \equiv \bullet$.

Lemma 8. *If* $next(n_L, e, c) = c'$, *then* $next(n_L, \varepsilon_L(e), \varepsilon_L(c)) = \varepsilon_L(c')$.

Proof. Intuitively, the lemma ensures that processing any event e generated by *public* thread $n_H \in P$ commutes with the erasure function. We prove that by case analysis on e. Case ϵ is trivial. We distinguish two cases for event $\mathbf{spawn}(\Delta, n_2, t, K)$ depending on the security level of the child thread. If *public*, i.e., $T(n_2).pc \equiv L$, then erasure rewrites the event to $\mathbf{spawn}(\varepsilon_L(\Delta), n_2, \varepsilon_L(t), K)$ and the updates to the budget and core maps commute with erasure, e.g., $\varepsilon_L(T[n_2 \mapsto (\Delta, L, \varnothing, [\,] \mid t, [\,])]) = \varepsilon_L(T)[n_2 \mapsto (\varepsilon_L(\Delta), L, \varnothing, [\,] \mid \varepsilon_L(t), [\,])]$. If the child thread is *secret*, i.e., $T(n_2).pc \equiv H$, we apply *2-steps erasure* to simulate the sensitive write operation [64]. In particular, erasure rewrites the event to special event $\mathbf{spawn}_L(\varepsilon_L(\Delta), n_2, K)$, and function $next(\cdot)$ handles it so that it gives global state $\varepsilon_L(c')$ (Fig. 16c). Notice that the child thread has a secret current label and its parent is public, thus $n_2 \in P_H$ and $\varepsilon_L(T[n_2 \mapsto (\Delta, L, \varnothing, [\,] \mid t, [\,])]) = \varepsilon_L(T)[n_2 \mapsto (\bullet, \bullet, \bullet, \bullet \mid \bullet, \bullet)]$. Case $\mathbf{fork}(\Delta, n_2, t, b, h)$ is similar. In case $\mathbf{kill}(n_2)$, we apply Lemma 6 to show that the resources assigned to the parent thread n_L remain the same under erasure. In case $\mathbf{send}(n_2, t)$, we distinguish two cases depending on the sensitivity of n_2. If *public*, i.e., $T(n_2).pc \equiv L$, then the lemma is trivial. In function $next(\cdot)$ the same guard holds true under erasure, i.e., $n_L \in Dom(T)$

iff $n_L \in Dom(\varepsilon_L(T))$, $\varepsilon_L(T)(n_L) = \varepsilon_L(T(n_L))$ and $|\Delta| = |\varepsilon_L(\Delta)|$ (Lemma 5.1), and the public updates to the thread pool commutes under erasure. If the recipient is *secret*, i.e., $T(n_2).pc \equiv H$, then we further distinguish between $n_2 \in P_H$, which follows as before, and $n_2 \notin P_H$, which implies that $n_2 \notin Dom(\varepsilon_L(T))$, the message is dropped and the thread map remains unchanged, i.e., $\varepsilon_L(T) \equiv \varepsilon_L(T[n_2 \mapsto s])$.

By means of this lemma, we show that erasure commutes with processing parallel events *commute*.

Lemma 9. $\varepsilon_L(\langle\!\langle sort\ es \rangle\!\rangle^c) \equiv \langle\!\langle sort\ (map\ \varepsilon_L(\cdot)\ es) \rangle\!\rangle^{\varepsilon_L(c)}$

Proof. Intuitively, the proof relies on the fact that *public* events, i.e., events generated by public threads, are erased homomorphically, i.e., $\varepsilon_L(n_L, e) = (n_L, \varepsilon_L(e))$ and *secret* events are rewritten to ϵ, i.e., $\varepsilon_L(n_H, e) = (\_, \epsilon)$ otherwise.[25] The proof follows by induction on the event list es. The base case is trivial. In the inductive case, we need to consider how event erasure affects sorting. Intuitively, the erasure function does not affect the relative order of public events, i.e., (n_L, e), because $\varepsilon_L(e)$ has the same priority as e. Instead, secret events end up at the end of the list because their event is erased to ϵ, which has the least priority. If the next event is *secret*, we use Lemma 7 and we apply induction, after removing the corresponding event in the erased list—the erased event is trivial (ϵ) and it does not change the state, i.e., $\langle\!\langle sort\ ((n_H, \epsilon) : map\ \varepsilon_L(\cdot)\ es) \rangle\!\rangle^{\varepsilon_L(c)} \equiv \langle\!\langle sort\ (map\ \varepsilon_L(\cdot)\ es) \rangle\!\rangle^{\varepsilon_L(c)}$. If the next event is *public*, then the scheduler processes the corresponding erased event in the erased configuration. The proof then follows by Lemma 8 and straightforward induction.

We conclude this section by showing that hierarchical scheduling (Figure 17a) is deterministic.

Lemma 10. *If* $Q[\langle n_1^{b_1} \rangle]$ *and* $Q[\langle n_2^{b_2} \rangle]$ *then* $n_1 \equiv n_2$ *and* $b_1 \equiv b_2$.

Proof. Induction on the scheduling relation.

B.3 Progress-Insensitive Non-Interference

Using the auxiliary lemmas listed above, we prove *progress-inensitive* non-interference, which guarantees that L-equivalence is preserved under assumptions that both configurations make progress. As explained in Section 5.2, the property relies on*determinism* of the stepping relation and *simulation*.

Proposition 5 (Determinism). *For all states* s_1, s_2, s_3, *core queues* Q_1, Q_2, Q_3, *thread ids* n_1, n_2, *events* e_1, e_2, *global states* Σ, *configurations* c_1, c_2, c_3, *the following hold:*

1. *If* $s_1 \rightsquigarrow_\mu s_2$ *and* $s_1 \rightsquigarrow_\mu s_3$ *then* $s_2 \equiv s_3$
2. *If* $Q_1 \xrightarrow{(n_1, s_1, e_1)}_\Sigma Q_2$ *and* $Q_1 \xrightarrow{(n_2, s_2, e_2)}_\Sigma Q_3$ *then* $n_1 \equiv n_2$, $s_1 \equiv s_2$, $e_1 \equiv e_2$ *and* $Q_2 \equiv Q_3$.
3. *If* $c_1 \hookrightarrow c_2$ *and* $c_1 \hookrightarrow c_3$ *then* $c_2 \equiv c_3$

[25] The thread that generates ϵ could be different from n_H, see Proposition 7.2 and 7.3. Since the event ϵ has no effects on the state, the thread identifier is irrelevant for the rest of the proof.

Proof.

1. Case analysis on the sequential step relation.
2. By Lemma 10, we derive $n_1 \equiv n_2$, thus the same thread is scheduled on both cores. Then, we do case analysis and, since Σ is the same in both reductions, the threads step likewise, i.e., $s_1 \equiv s_2$ (using Lemma 5.1 in case [STEP]), and generate the same event, $e_1 \equiv e_2$. The resulting cores are equal, i.e., $Q_1 \equiv Q_2$, since both threads have the same residual budget (Lemma 10).
3. The lemma follows immediately by applying determinism, i.e., Lemma 5.2, on all the core reductions $\Phi(i) \xrightarrow{(n_i,s_i,e_i)}_{\Sigma} Q_i$, for $i \in \{1 \mathinner{..} \kappa\}$.

Proposition 6 (Sequential Simulation). *If $s \rightsquigarrow s'$ then $\varepsilon_L(s) \rightsquigarrow \varepsilon_L(s')$.*

Proof. If the current label is H, then simulation follows by rule [HOLE], which simply ticks, i.e., $(\bullet, \bullet, \bullet, \bullet \mid \bullet, \bullet) \rightsquigarrow (\bullet, \bullet, \bullet, \bullet \mid \bullet, \bullet)$. If the current label is L, then simulation follows by applying the same step rule under reduction. For those, we discuss only the interesting cases, where we apply 2-steps erasure [64]—all the others cases are trivial and follow by applying Lemma 5 when needed, e.g., 5.1 for rule [APP$_2$] and 5.3 for rule [GC]. The erasure function rewrites term *label* to special term $label_L$, then rule [LABEL$_{L1}$] simulates [LABEL$_1$] and rule [LABEL$_{L2}$] ([LABEL$_{L3}$]) simulates [LABEL$_2$] when the assigned label is *public*, i.e., L, (resp. *secret*, i.e., H). Similarly, erasure rewrites term *unlabel* to special term $unlabel_L$, then rule [UNLABEL$_{L1}$] simulates [UNLABEL$_1$] and rule [UNLABEL$_{L2}$] ([UNLABEL$_{L3}$]) simulates [UNLABEL$_2$], when the labeled value is *public*, i.e., $Labeled\ L\ \varepsilon_L(t)^{\circ}$ for some term t (resp. *secret*, i.e., $Labeled\ H\ \bullet$).

We now lift *simulation* to core reductions. For brevity, we write thread's state $\bullet$ for erased state $(\bullet, \bullet, \bullet, \bullet \mid \bullet, \bullet)$, state $s_{\circ}$ for the spinning thread's state, i.e., $s_{\circ} = ([\,], \bot, \varnothing, [\,] \mid return\ (), [\,])$.

Proposition 7 (Core Simulation). *Given a core reduction step $Q \xrightarrow{(n,s,e)}_{\Sigma} Q'$, then one of the following holds:*

1. *If $n \in P$, then $\varepsilon_L(Q) \xrightarrow{\varepsilon_L(n,s,e)}_{\varepsilon_L(\Sigma)} \varepsilon_L(Q')$*
2. *If $n \notin P$ and $Q \cap P = \varnothing$, then $\varepsilon_L(Q) \xrightarrow{(\circ,s_{\circ},\epsilon)}_{\varepsilon_L(\Sigma)} \varepsilon_L(Q')$*
3. *If $n \notin P$ and $Q \cap P \neq \varnothing$, then $\exists\ n' \in P_H$, such that $\varepsilon_L(Q) \xrightarrow{(n',\bullet,\epsilon)}_{\varepsilon_L(\Sigma)} \varepsilon_L(Q')$*

Proof.

1. The first case simulates the execution of a thread with visible resources, i.e., $n \in P$, which executes similarly under erasure. We distinguish two cases depending on the thread's current label.
   - ▶ If $n_L \in P_L$, we show that the same thread is scheduled using Lemma 2 and we proceed by case analysis on the core step. Since the thread's current label is public, erasure preserves the thread's structure and its resources. As a result the erased thread steps likewise, i.e., it performs exactly the same reduction step. In particular, we apply Proposition 6 in case [STEP] and Lemma 1 in case [CONTEXTSWITCH].

- ▶ If $n_H \in P_H$, we apply Lemma 3, i.e., the core scheduler executes the same secret thread, which then *ticks*, i.e., it reduces with rule [STEP] applied to rule [HOLE], since its configuration gets completely erased, i.e., $\varepsilon_L(T(n_H)) = (\bullet, \bullet, \bullet, \bullet \mid \bullet, \bullet)$.

2. The second case simulates the execution of a core allocated to threads with no visible resources, i.e., $n \notin P$ and $Q \cap P = \varnothing$. This case occurs if some secret thread with public resources *spawns* another secret thread on a core that it owns. Under erasure, the core is still *free*, i.e., $\varepsilon_L(Q) = \langle \circ^0 \rangle$ and the spinning thread $\circ$ takes over in the erased core queue via rule [CONTEXTSWITCH].
3. The third case simulates the execution of a secret thread with no visible resources, i.e., $n \notin P$, which shares the core with threads gets scheduled on the queue Q, which contains some other thread with visible resources, i.e., $Q \cap P \neq \varnothing$. For example, this happens when a secret thread with visible resources *forks* another secret thread on the same core. Then, we apply Lemma 4 and conclude that the core scheduler executes its closest ancestor $n'_H \in P_H$, which remains in the erased core. [26] Thread n'_H simulates the execution of thread n_H by rule [STEP] and [HOLE] (see above). We remark that core erasure cancels out the effects of fork from queue Q' (rule [FORK]), as it collapses both thread n_H and its new child to the ancestor thread n'_H, which regains their resources.

Proposition 2 (Parallel Simulation). Given a parallel reduction step $c \hookrightarrow c'$, then $\varepsilon_L(c) \hookrightarrow \varepsilon_L(c')$.

Proof. The proof requires to show that running the parallel scheduler on the erased initial state, i.e., $\varepsilon_L(c)$, by means of rule [PARALLEL] from Fig. 5, gives a final state that corresponds to the erased state obtained in original step i.e., $\varepsilon_L(c')$. The fact that the core scheduler could schedule different threads on erased cores makes the proof interesting. Intuitively, public cores, i.e., cores that execute public threads, proceed in lock-step with the original configuration (Proposition 7.1) and the parallel scheduler processes their public events in the same relative order, so that erasure and changes to the state *commute*. Instead, secret cores execute either the *public* spinning thread (Proposition 7.2) or a another public thread $n'_H \in P_H$ (Proposition 7.3). In either case, they generate the trivial event, i.e., ϵ, which leaves the global state unchanged, hence the different order with which they get processed is irrelevant under erasure.

Firstly, we apply *core simulation*, i.e., Proposition 7, to the core scheduler step on each core $i \in \{1..\kappa\}$. Then, the parallel scheduler changes the global state accordingly: it updates the erased thread pool $\varepsilon_L(T)$, the core map $\varepsilon_L(\Phi)$ and then processes the events generated by the core scheduler. Those operations either *commute* under erasure, whenever they affect *public* parts of the state, or have *no effect*, otherwise. We show that, by case analysis on the security level of the threads scheduled in the erased configuration. In particular, for each thread n_i that runs in the original configuration, one of the clauses of Proposition 7 applies.

[26] The current label of the ancestor cannot be public, i.e., $n'_L \in P_L$), otherwise $n_H \in P_H$ and case 1 applies instead.

Commutativity holds in the first case (Proposition 7.1), since $\varepsilon_L(T[n \mapsto s]) \equiv \varepsilon_L(T)[n \mapsto \varepsilon_L(s)]$, when $n \in P$. In the second case (Proposition 7.2), we show $\varepsilon_L(T[n \mapsto s]) \equiv \varepsilon_L(T)[\circ \mapsto \varepsilon_L(s_\circ)]$ by rewriting both sides of the equation to $\varepsilon_L(T)$. On the left-hand side, erasure filters out the secret thread from T, since $n \notin P$, while the update does not change the thread pool on the right-hand side, because $\varepsilon_L(T)(\circ) \equiv \varepsilon_L(s_\circ) \equiv ([\,], \bot, \varnothing, [\,] \mid \mathit{return}\ (), [\,])$ In the third case (Proposition 7.3), the ancestor $n' \in P$ takes over n in the erased core and the update to the thread map has no effect, i.e., $\varepsilon_L(T) = \varepsilon_L(T[n' \mapsto \bullet])$, since $\varepsilon_L(T)(n'_H) = (\bullet, \bullet, \bullet, \bullet \mid \bullet, \bullet)$. In all cases, core map erasure is homomorphic, i.e., $\varepsilon_L(\Phi) = \lambda k.\varepsilon_L(\Phi(k))$, then core map updates always commute with erasure, i.e., $\varepsilon_L(\Phi[k \mapsto Q_i]) \equiv \varepsilon_L(\Phi)[k \mapsto \varepsilon_L(Q_i)]$. To conclude the proof, we apply Lemma 9 and show that erasing and processing events *commute*, i.e., $\varepsilon_L(\langle\!\langle \mathit{sort}\ es \rangle\!\rangle^c) \equiv \langle\!\langle \mathit{sort}\ (\mathit{map}\ \varepsilon_L(\cdot)\ es) \rangle\!\rangle^{\varepsilon_L(c)}$.

We conclude with the proof of *progress-insensitive* non-interference.

Proposition 3 (Progress-Insensitive Non-Interference). *If $c_1 \hookrightarrow c_1'$, $c_2 \hookrightarrow c_2'$ and $c_1 \approx_L c_2$, then $c_1' \approx_L c_2'$.*

Proof. We apply *parallel simulation*, i.e., Proposition 2 and derive the erased reductions $\varepsilon_L(c_1) \hookrightarrow \varepsilon_L(c_1')$ and $\varepsilon_L(c_2) \hookrightarrow \varepsilon_L(c_2')$. From $c_1 \approx_L c_2$, we obtain $\varepsilon_L(c_1) \equiv \varepsilon_L(c_2)$ (Definition 1), and Proposition 5.3 (*parallel determinism*) gives $\varepsilon_L(c_1') \equiv \varepsilon_L(c_2')$, i.e., $c_1' \approx_L c_2'$.

B.4 Timing-Sensitive Non-Interference

We now lift the security guarantees of LIO$_{\mathrm{PAR}}$ to be *timing-sensitive*. Intuitively, timing-insensitive non-interference ensures that the timing of any parallel program does not depend on secret information, when executed with LIO$_{\mathrm{PAR}}$ runtime system. More precisely, the theorem ensures that if a parallel program steps with some secret information, then it also steps with different secret information. The key property to proving this stronger form of non-interference is *time-sensitive* progress, i.e., Proposition 4, which reconstructs the *single step* taken by program in the other execution. This property relies on some side conditions that guarantee that the program satisfies some basic correctness properties, that we formally specify in the definition of *valid configuration*.

Definition 2 (Valid Configuration). *A configuration $\langle T, B, H, \theta, \Phi, \omega \rangle$ is valid if and only if it satisfies the following properties:*

- ▶ *If $T(n) = s$, then state s is well-formed, i.e., it is the state of a well-typed thread and $x \in \mathit{Dom}(s.\Delta)$ for all variables $x \in \mathit{fv}(s)$;*
- ▶ *For all cores $k \in \{1\,..\,\kappa\}$, $\circ_k \mapsto s_\circ \in T$, $\circ_k \mapsto 0 \in B$, $\circ_k \mapsto H_0 \in H$, $\circ_k \mapsto \varnothing \in \theta$;*
- ▶ *$\mathit{Dom}(T) = \mathit{Dom}(B) = \mathit{Dom}(H) = \mathit{Dom}(\theta)$;*
- ▶ *If $n \in Q$, then $n \in \mathit{Dom}(T)$;*
- ▶ *If $Q[\langle n^b \rangle]$, then $b \leqslant B(n)$, $|T(n).\Delta| \leqslant H(n)$ and $n.k \notin \theta(n)$;*
- ▶ *For all threads $n_1\ n_2 \in \mathit{Dom}(T)$, such that $n_1 \neq n_2$, $\theta(n_1) \cap \theta(n_2) = \varnothing$;*

It is easy to see that the initial parallel configuration (Corollary 1) is *valid*. The next lemma shows that valid configurations that execute with the operational semantics of LIO$_{\mathrm{PAR}}$ remain valid.

Lemma 11 (Valid Invariant). *If configuration c is valid and $c \hookrightarrow c'$, then c' is valid.*

Proof. By case analysis on all the reduction relations.

Proposition 4 (Time-Sensitive Progress). *For all valid configurations c_1, c_1' and c_2 and parallel reduction steps $c_1 \hookrightarrow c_1'$, if $c_1 \approx_L c_2$, then there exists a configuration c_2', such that $c_2 \hookrightarrow c_2'$.*

Proof. The parallel scheduler has only one reduction rule, i.e., [PARALLEL], thus the proof mainly relies on *core progress*, i.e., showing that for each core that steps in the first configuration, then the corresponding core in the second configuration also steps. Formally, for all *valid* cores $Q_1 \approx_L Q_2$ and global states $\Sigma_1 \approx_L \Sigma_2$, if $Q_1 \xrightarrow{m_1}_{\Sigma_1} Q_1'$ and $Q_1 \approx_L Q_2$, then there exists Q_2' and m_2, such that $Q_2 \xrightarrow{m_2}_{\Sigma_2} Q_2'$. Since $c_1 \approx_L c_2$, the core maps in the configurations are L-equivalent, i.e., $\Phi_1 \approx_L \Phi_2$, hence $\Phi_1(i) \approx_L \Phi_2(i)$, for all $i \in \{1 \, .. \, \kappa\}$. In particular, we apply *core simulation*, i.e., Proposition 7, which gives us $\varepsilon_L(Q_1) \xrightarrow{m_3}_{\varepsilon_L(\Sigma_1)} \varepsilon_L(Q_1')$, for some message m_3, and use that together with $\varepsilon_L(Q_1) \equiv \varepsilon_L(Q_2)$ and the assumption that Q_1 and Q_2 are valid to reconstruct Q_2', m_2 and the other step $Q_2 \xrightarrow{m_2}_{\varepsilon_L(\Sigma_2)} Q_2'$.

The task of reconstructing a core step is facilitated by the fact that the core semantics always steps, regardless of the number of threads on the core, their resources and state. Specifically, either threads execute *sequentially* ([STEP]), or they perform a *concurrent* operation ([FORK,SPAWN,WAIT]), or they are *stuck* [STUCK] otherwise. Note that, even if no thread has sufficient time budget to step, then rule [CONTEXTSWITCH] takes over and that even *free* cores step thanks to the core's *spinning* thread, i.e., $\circ$.

Theorem 1 (Timing-Sensitive Non-Interference). *For all valid configurations c_1 and c_2, if $c_1 \hookrightarrow c_1'$ and $c_1 \approx_L c_2$, then there exists a configuration c_2', such that $c_2 \hookrightarrow c_2'$ and $c_1' \approx_L c_2'$*

Proof. We apply time-sensitive progress, i.e., Proposition 4 to valid configurations $c_1 \approx_L c_2$ and obtain the second reduction $c_2 \hookrightarrow c_2'$ for some configuration c_2'. We then derive L-equivalence of the final configurations, i.e., $c_1' \approx_L c_2'$, from progress-insensitive non-interference, i.e., Proposition 3, applied to $c_1 \hookrightarrow c_1'$, $c_2 \hookrightarrow c_2'$ and $c_1 \approx_L c_2$.

Corollary 1 generalizes timing-sensitive non-interference to many steps by applying Theorem 1 and Proposition 11 as many times.

C Attack Code

Below we list the code for the parallel scheduler-based attacks. The code depends on the following external packages:

- ▶ `lio-0.11.6.0`
- ▶ `hashable-1.2.7.0`
- ▶ `text-1.2.3.1`
- ▶ `array-0.5.1.1`
- ▶ `bytestring-0.10.8.1`

Haskell package manager Cabal can be used to configure and install these dependencies with the command `cabal install packages`. Both attack modules rely on some helper functions found in Appendix C.3. Additionally, both attacks rely on thresholds that must be determined empirically as they are machine dependent.

The attacks can be compiled and executed using GHC version 8.0.2 with the following commands, where `CODE` is the file containing the attack code and `SECRET` represents the secret value to leak (*0* or *1*),

```
$ ghc -threaded -rtsopts CODE lib.hs -o attack
$ ./attack SECRET +RTS -N2 -RTS
```

Section C.1 and C.2 list the code of the reclamation and allocation attacks (`reclamation.hs` and `allocation.hs`), sketched in Section 2.3. In the attacks, a secret thread affects the CPU-time available to other public threads by terminating early (the scheduler *reclaims* its quota of CPU-time), or forking another secret thread (the scheduler *allocates* a new CPU-time quota). The attacks are written using the LIO library and disjunction category (`DC`) labels to specify the security lattice [56].

C.1 Reclamation Attack

```haskell
-- reclamation.hs

module Main where

import System.Environment
import Data.List

import LIO
import LIO.LIORef
import LIO.DCLabel
import LIO.Concurrent
import LIO.Run

import Lib

-- Thresholds.
-- These parameters are machine specific and estimated empirically.
len         = 100000
threshold   = 3000

-- The code run by the secret thread.
-- If secret == 1, the thread loops,
-- otherwise it terminates right away.
highThread :: DC (DCLabeled Int) -> DC Int
highThread secret = do
  s <- unlabel secret
  case s of
    1 -> do
      t1 <- busyWait 100000
      return 0
    _ -> return 0

-- Simple heuristic to determine the value of the secret
analyze :: (String, Int) -> Int
analyze (_,a) = if a > threshold then 1 else 0

-- Count the number of messages from each thread
-- in the public channel and infer the secret.
count :: LIORef DCLabel [String] -> DC Int
count channel= do
  msgs <- readLIORef channel
  let acc = map (\x -> (head x, length x)) (group msgs)
  return $ analyze $ head acc

-- Fork the two public threads that write to
-- the same public channel
runLowThreads :: LIORef DCLabel [String] -> DC Int
runLowThreads channel = do
```

```haskell
  t1 <- lFork low (writeA len channel)
  t2 <- lFork low (writeB len channel)
  () <- lWait t1
  () <- lWait t2
  -- analyze the public channel to infer the secret
  count channel

-- Start the the secret thread and the two public threads,
-- which return the secret.
leak :: DC (DCLabeled Int) -> DC Int
leak secret = do
  channel <- newLIORef low []
  secretThread <- lFork secretL (highThread secret)
  secretValue <- runLowThreads channel
  return secretValue

main :: IO ()
main = do
  l <- getArgs
  let secret = getArg l
  secret <- evalLIO (leak (label secretL secret)) init
  print $ "The secret is " ++ (show secret)
  return ()
```

C.2 Allocation Attack

```haskell
-- allocation.hs

module Main where

import System.Environment
import Data.List

import LIO
import LIO.LIORef
import LIO.DCLabel
import LIO.Concurrent
import LIO.Run

import Lib

-- Thresholds.
-- These parameters are machine specific and estimated empirically.
len           = 100000
lowThreshold  = 4000
highThreshold = 15000

-- The code run by the secret thread.
-- If secret == 1, then the thread forks a child,
-- otherwise it loops.
highThread :: DC (DCLabeled Int) -> DC Int
highThread secret = do
  s <- unlabel secret
  case s of
    1 -> do
      s1 <- lFork high (busyWait len)
      t1 <- busyWait len
      res <- lWait s1
      return 0
    _ -> do
      t1 <- busyWait len
      return 0

-- Simple heuristic to determine the value of the secret.
analyze :: (String, Int) -> Int
analyze (_,a) =
  if (a < lowThreshold) || (a > highThreshold)
    then 1
    else 0

-- Count the number of messages from each thread
-- in the public channel and infer the secret.
count :: LIORef DCLabel [String] -> DC Int
count channel= do
```

```haskell
  msgs <- readLIORef channel
  let acc = map (\x -> (head x, length x)) (group msgs)
  return $ analyze $ head acc

-- Fork the two public threads that write
-- to the same public channel
runLowThreads :: LIORef DCLabel [String] -> DC Int
runLowThreads channel = do
  t1 <- lFork low (writeA len channel)
  t2 <- lFork low (writeB len channel)
  () <- lWait t1
  () <- lWait t2
  -- analyze the public channel to infer the secret
  count channel

leak :: DC (DCLabeled Int) -> DC Int
leak secret = do
  channel <- newLIORef low []
  secretThread <- lFork high (highThread secret)
  secretValue <- runLowThreads channel
  return secretValue

main :: IO ()
main = do
  l <- getArgs
  let secret = getArg l
  secret <- evalLIO (leak (label high secret)) init
  print $ "The secret is " ++ (show secret)
  return ()
```

C.3 Helper Functions

```haskell
-- lib.hs

module Lib where

import LIO
import LIO.LIORef
import LIO.DCLabel

import Control.Monad

-- Command line parsing
getArg []     = -1
getArg (a:r) = read a

-- Labels
low = "Public" %% "Public"
secretL = "Secret" %% "Secret"
high = low `lub` secretL

-- Initial LIO state (current label and clearance)
init = LIOState { lioLabel      = low
                , lioClearance = high }

-- Write a message to the channel for a given number of times.
write :: String -> Int -> LIORef DCLabel [String] -> DC ()
write msg n ref = replicateM_ n trace
  where trace = do
          () <- modifyLIORef ref (\l -> msg:l)
          return ()

-- Write "A" to the channel
writeA :: Int -> LIORef DCLabel [String] -> DC ()
writeA len ref = write "A" len ref

-- Write ``B" to the channel
writeB :: Int -> LIORef DCLabel [String] -> DC ()
writeB len ref = write "B" len ref

-- Busy waiting.
busyWait :: Int -> DC Int
busyWait 0 = return 1
busyWait n = do
  acc <- busyWait (n - 1)
  return $ acc + n
```