Foundations for Parallel Information Flow Control Runtime Systems

Marco Vassena[1(✉)], Gary Soeller[2], Peter Amidon[2], Matthew Chan[3], John Renner[2], and Deian Stefan[2(✉)]

[1] Chalmers University, Gothenburg, Sweden
vassena@chalmers.se
[2] UC San Diego, San Diego, USA
deian@cs.ucsd.edu
[3] Awake Security, Sunnyvale, USA

Abstract. We present the foundations for a new dynamic information flow control (IFC) parallel runtime system, LIO_{PAR}. To our knowledge, LIO_{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_{PAR} is secure, i.e., it satisfies progress- and 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.

This work was supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation program sponsored by DARPA and by gifts from Cisco and Fujitsu. This work was partly done while Marco Vassena and Matthew Chan were at UCSD.

F. Nielson and D. Sands (Eds.): POST 2019, LNCS 11426, pp. 1–28, 2019.
https://doi.org/10.1007/978-3-030-17138-4_1

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 [42,64]. Researchers have made significant strides towards addressing these challenges—many IFC systems now support real-world features and abstractions safely [10,15,20,34,43,50,51,54,55,59,60,62,67,68]. 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 [67], Hails [12,13], and LMonad [45]), unfortunately, do not handle user requests in parallel—the language-level IFC systems that underlie them (e.g., Jeeves [68] and LIO [54]) 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). But, 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 allocates and reclaims *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 GC, may introduce timing covert channels [46]. Inspired by previous work on real-time GCs (e.g., [3,5,6,16,44,48]), we equip LIO_{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 (Sect. 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 (Sect. 3). To our knowledge, LIO_{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_{PAR} hierarchical runtime system (Sect. 4) and prove that it satisfies *progress- and timing-sensitive non-interference* (Sect. 5); we believe that this is the first general purpose dynamic IFC runtime system to provide such strong guarantees in the parallel setting [64].

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.

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 the 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* [1,58], 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. [54] 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 [54]. 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) [46]—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 [54,55].

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 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 [46], 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[1]. Nevertheless, they are feasible and—because they abuse the runtime—they are effective against language-level external-timing mitigation techniques, including [54,71]. The attacks are also feasible on other systems—similar attacks that abuse the GC have been demonstrated for both the V8 and JVM runtimes [46].

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

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

`if secret`
`  then terminate`
`  else forever skip`
	`forever`
	`  (write chan` p_0`)`
		`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.

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. Figure 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, causing both p_1 and p_0 to 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:

- 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`.[2]

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.

[2] 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.

The LIO_{PAR} runtime does not automatically balance resources between threads. Instead, LIO_{PAR} makes resource management explicit at the language level. When 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 at any time. 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 (Sect. 4) and establish its security guarantees (Sect. 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 [63], lazy evaluation [7,59]), shared operating system state (e.g., file system locks [24], events and I/O [22,32]), or shared hardware (e.g., caches, buses, pipelines and hardware threads [11,47]) 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 [53]. The security monitor reduces secure programs and aborts the execution of leaky programs.

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

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

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

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

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

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

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

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

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

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

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

Fig. 2. Sequential LIO_{PAR}.

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. Figure 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 $\mu = (h, cl)$.[3] Since we are interested in modeling a system with *finite* resources, we parameterize the transition 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.[4] 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.[5] To avoid overflows, a thread can measure the size of its own heap via primitive *size* (Sect. 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 the extended version of this paper.

Security Primitives. A labeled value *Labeled* $\ell\ t^{\circ}$ of type *Labeled* τ 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* τ 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

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

[4] The calculus does not feature lazy evaluation. Laziness, because of *sharing*, introduces a covert channel, which has already been considered in previous work [59].

[5] To simplify reasoning, our generic memory model is basic and assumes a uniform size for all the objects stored in the heap. We believe that it is possible to refine our generic model with more accurate memory models (e.g., GHC's tagless G-machine (STG) [23], the basis for GHC's runtime [39]), but leave this to future work.

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.[6] 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 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.[7] 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 Sect. 4.1), reclaim resources and measure resource usage (*kill*, *size*, and *time* in Sect. 4.2), threads synchronization and communication (*wait*, *send* and *receive* in the extended version of this paper).

4.1 Core Scheduler

In this section, we extend LIO$_{\text{PAR}}$ 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 Sect. 2.3), LIO$_{\text{PAR}}$ 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 Sect. 4.3.

[6] 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. [66].

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

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

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

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

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

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

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

Fig. 3. Concurrent $\mathrm{LIO}_{\mathrm{PAR}}$.

Syntax. Figure 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 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.[8] 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 have zero residual budget. We overload this notation to represent tree updates: a rule $Q[\langle n^{1+b} \rangle] \to Q[\langle n^b \rangle]$ finds the next thread to run in queue Q and decreases its budget by one.

Semantics. Figure 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 $\ell_{\mathrm{L}} \sqsubseteq \ell_{\mathrm{H}}$ and $pc \sqsubseteq \ell_{\mathrm{L}}$. 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

[8] When implemented, this procedure might introduce a timing channel that leaks the number of threads running on the core. In practice, techniques from real time schedulers can be used to protect against such timing channels. The model of $\mathrm{LIO}_{\mathrm{PAR}}$ does not capture the execution time of the runtime system itself and thus this issue does not arise in the security proofs.

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 the extended version of this paper 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 $i \in \{1 \,..\, |Q|\}$ 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 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.[9] Intuitively, it is secure to expose this feature because LIO_{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

[9] An *external* attacker can take timing measurements using network communications. An attacker equipped with an *internal* clock is equally powerful but simpler to formalize [46].

$$\textsc{Kill}_2\quad \frac{\Sigma.T(n) = (\Delta, pc, \{n'\} \cup N \mid n', kill : S) \qquad s = (\Delta, pc, N \mid return\ (), S)}{Q[\langle n^{1+b} \rangle] \xrightarrow{(n,s,\mathbf{kill}(n'))}_{\Sigma} Q[\langle n^{b} \rangle]}$$

$$\textsc{Unlabel}_2\quad \frac{pc \sqcup \ell \sqsubseteq \mu.cl \qquad \forall\, n \in N\ .\ pc \sqcup \ell \sqsubseteq n.pc}{(\Delta, pc, N \mid Labeled\ \ell\ t, unlabel : S) \rightsquigarrow_{\mu} (\Delta, pc \sqcup \ell, N \mid return\ t, S)}$$

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

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

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

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

Fig. 4. LIO_{PAR} with resource reclamation and observation primitives.

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, 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 [$\textsc{Unlabel}_2$] 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_{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'.[10] In rule [App-GC], a collection is *automatically* triggered when the thread's next memory allocation would overflow the heap.

[10] 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.

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_{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.

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

Parallel

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

$$\mathit{next}(\_, \epsilon, c) = c$$

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

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

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

Fig. 5. Top-level parallel scheduler.

Semantics. Figure 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 Sects. 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*.[11] 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 $i \in \{1..\kappa\}$. 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.[12] In particular, new threads are created first (event **spawn**$(\cdot)$ and **fork**$(\cdot)$), and then killed (event **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, accumulating configuration updates ($next(\cdot)$ updates the current configuration by one event-step): $\langle\!\langle (n, e) : es \rangle\!\rangle^c = \langle\!\langle es \rangle\!\rangle^{next(n,e,c)}$. When no more events need processing, the configuration is returned $\langle\!\langle [] \rangle\!\rangle^c = c$.

Event Processing. Figure 5 defines function $next(n, e, c)$, 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 $H'[n_2 \mapsto h]$.[13] 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 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.

[11] Non-deterministic updates would make the model vulnerable to refinement attacks [40].

[12] 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.

[13] Notice that $|\Delta| < h$ by rule [Fork].

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 the extended version of this paper for details.

5 Security Guarantees

In this section we show that LIO_{PAR} satisfies a strong security condition that ensures timing-agreement of threads and rules out timing covert channels. In Sect. 5.1, we describe our proof technique based on *term erasure*, which has been used to verify security guarantees of functional programming languages [30], IFC libraries [8,17,54,56,61], and an IFC runtime system [59]. In Sect. 5.2, we formally prove security, i.e., *progress- and timing-sensitive non-interference*, a strong form of non-interference [14], inspired by Volpano and Smith [64]—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 [59,62] 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 [36,52], we establish a *strong* low-bisimulation property of the parallel scheduler, which guarantees that attacker-indistinguishable configurations execute in lock-step and remain indistinguishable. 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.[14] Once the erasure function is defined, the core of the proof technique

[14] 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.

consists of proving an essential *commutativity* 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., $\varepsilon_L(\Delta, L, N, |\ t, S) = (\varepsilon_L(\Delta), L, N\ |\ \varepsilon_L(t), \varepsilon_L(S))$, and in full otherwise, i.e., $\varepsilon_L(\Delta, H, N, |\ t, S) = (\bullet, \bullet, \bullet\ |\ \bullet, \bullet)$.

Resource Erasure. Since LIO_{PAR} manages resources explicitly, the simulation property above requires to define the erasure function for resources as well. 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_{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 *killing* them.

Time Budget. First, we project the identifiers of *public* threads from the thread pool $T : \mathit{Dom}_L(T) = \{\, n_L \mid n \in \mathit{Dom}(T) \wedge T(n).\mathit{pc} \equiv L \,\}$, 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.[15] 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

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

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 $B_H = \{ n_H \mapsto B(n_H) + \sum_{i \in [\![\{ n_H \}]\!]^T} B(i) \mid n_H \in P_H \}$.

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:

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

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 progress- and timing-sensitive non-interference relies on two fundamental properties, i.e., *determinacy* and *simulation* of parallel reductions. Determinacy 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). *If $c \hookrightarrow c'$, 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 progress-sensitive, we first prove *timing-sensitive progress*. Intuitively, if a *valid* configuration steps then any low equivalent parallel configuration also steps.[16]

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

Theorem 1 (Progress- and 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 non-interference security theorem from above for a given LIO_{PAR} parallel program, that explicitly rules out leaks via timing channels. In the following, the notation $\hookrightarrow_u$ denotes u reduction steps of the parallel scheduler.

Corollary 1. *Given a well-typed* LIO_{PAR} *program* t *of type Labeled* $\tau_1 \rightarrow$ *LIO* τ_2 *and two closed secrets* $t_1^\circ, t_2^\circ :: \tau_1$, *let* $s_i = ([\,], L, \varnothing, \mid t\ (\textit{Labeled}\ H\ t_i^\circ), [\,])$, $c_i = (T_i, B, H, \theta, 0, \Phi_i)$, *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_{PAR} extend to concurrent *single-core* programs by instantiating Corollary 1 with $\kappa = 1$.

6 Limitations

Implementation. Implementing LIO_{PAR} is a serious undertaking that requires a major redesign of GHC's runtime system. Conventional runtime systems freely

[16] A configuration is valid if satisfies several basic properties, e.g., it does not contain special term •. See the extended version of this paper for details.

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_{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_{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_{PAR}—Yang and Mazières' [66] 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. Moreover, this challenge is not unique to real-time systems or LIO_{PAR}; choosing privacy parameters in differential privacy, for example, shares many similarities [21,29].

The LIO_{PAR} programming model is also likely easier to use in certain application domains—e.g., web applications where the tail latency of a route can inform the thread bounds, or embedded systems where similar latency requirements are the norm. Nevertheless, in order to simplify programming with LIO_{PAR}, we intend to introduce privileges (and thus declassification) similar to LIO [12,56] or COWL [57].

Coarse-grained, floating-label systems such as LIO and LIO_{PAR} can suffer *label creep*, wherein the current computation gets tainted to a point where it cannot perform any useful writes [55]. Sequential LIO [56] addresses label creep through a primitive, `toLabeled`, which executes a computation (that may raise the current label) in a separate context and restores the current label upon its termination. Similar to concurrent LIO [54], LIO_{PAR} relies on fork to address label creep and not `toLabeled`—the latter exposes the termination covert-channel [54]. Even though LIO_{PAR} has a more restricted floating-label semantics than concurrent LIO, LIO_{PAR} also supports parallel execution, garbage collection, and new APIs for getting heap statistics, counting elapsed time, and killing threads.

7 Related Work

There is substantial work on language-level IFC systems [10,15,20,34,43,50,51,54,55,67,68]. Our work builds on these efforts in several ways. Firstly, LIO_{PAR} extends the concurrent LIO IFC system [54] 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 [41], several works on IFC π-calculi [18,19,25], and Rafnsson et al. [49] recent foundations for composable timing-sensitive interactive systems. These efforts, however, do not model runtime system resource management. Volpano and Smith [64] enforce a timing agreement condition, similar to ours, but for a static concurrent IFC system. Mantel et al. [37] and Li et al. [31] 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 [46], 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 [35,66]—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' [66] region-based allocation framework with region-based garbage collection and hierarchical scheduling.

Our LIO_{PAR} design also borrows ideas from the secure operating system community. Our explicit hierarchical memory management is conceptually similar to HiStar's container abstraction [69]. In HiStar, containers—subject to quotas, i.e., space limits—are used to hierarchically allocate and deallocate objects. LIO_{PAR} adopts this idea at the language-level and automates the allocation and reclamation. Moreover, we hierarchically partition CPU-time; Zeldovich et al. [69], however, did observe that their container abstraction can be repurposed to enforce CPU quotas. Deterland [65] 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_{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 [27,28,38], but, different from these efforts, LIO_{PAR} also hierarchically manages resources to eliminate classes of external timing channels.

Fabric [33,34] and DStar [70] are distributed IFC systems. Though we believe that our techniques would scale beyond multi-core systems (e.g., to data centers), LIO_{PAR} will likely not easily scale to large distributed systems like Fabric and DStar. Different from Fabric and DStar, however, LIO_{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 [22], for example, mitigates both timing channels in the VAX/VMM system [32] using "fuzzy time"—an idea recently adopted to browsers [26]. 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 [46,54,71]. 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_{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_{PAR}, Pizlo et al. [48] 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 these attacks become yet more vicious—they can be carried out internally. We presented LIO_{PAR}, the design of the first dynamic IFC hierarchical runtime system that supports deterministic parallelism and eliminate s both resource-based internal- and external-timing covert channels. To our knowledge, LIO_{PAR} is the first parallel system to satisfy progress- and timing-sensitive non-interference.

References

1. Askarov, A., Hunt, S., Sabelfeld, A., Sands, D.: Termination-insensitive noninterference leaks more than just a bit. In: Jajodia, S., Lopez, J. (eds.) ESORICS 2008. LNCS, vol. 5283, pp. 333–348. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88313-5_22
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., et al.: 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. Commun. ACM **55**(5), 111–119 (2012)
5. Baker Jr., H.G.: List processing in real time on a serial computer. Commun. ACM **21**(4), 280–294 (1978)
6. Blelloch, G.E., Cheng, P.: On bounding time and space for multiprocessor garbage collection. ACM SIGPLAN Not. **34**, 104–117 (1999)
7. Buiras, P., Russo, A.: Lazy programs leak secrets. In: Riis Nielson, H., Gollmann, D. (eds.) NordSec 2013. LNCS, vol. 8208, pp. 116–122. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41488-6_8
8. 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)
9. Das, A., Hoffmann, J.: ML for ML: learning cost semantics by experiment. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10205, pp. 190–207. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54577-5_11
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. J. Cryptographic Eng. **8**, 1–27 (2016)
12. Giffin, D.B., et al.: Hails: protecting data privacy in untrusted web applications. J. Comput. Secur. **25**(4–5), 427–461 (2017)
13. Giffin, D.B., et al.: 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, April 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: Focardi, R., Myers, A. (eds.) POST 2015. LNCS, vol. 9036, pp. 11–31. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46666-7_2
18. Honda, K., Vasconcelos, V., Yoshida, N.: Secure information flow as typed process behaviour. In: Smolka, G. (ed.) ESOP 2000. LNCS, vol. 1782, pp. 180–199. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-46425-5_12
19. Honda, K., Yoshida, N.: A uniform type structure for secure information flow. ACM Trans. Program. Lang. Syst. (TOPLAS) **29**(6), 31 (2007)
20. Hritcu, C., Greenberg, M., Karel, B., Pierce, B.C., Morrisett, G.: All your IFCException are belong to us. In: 2013 IEEE Symposium on Security and Privacy (SP), pp. 3–17. IEEE (2013)
21. Hsu, J., et al.: Differential privacy: an economic method for choosing epsilon. In: Proceedings of the 2014 IEEE 27th Computer Security Foundations Symposium, CSF 2014, pp. 398–410. IEEE Computer Society, Washington, DC (2014)

22. Hu, W.-M.: Reducing timing channels with fuzzy time. J. Comput. Secur. **1**(3–4), 233–254 (1992)
23. Jones, S.L.P.: Implementing lazy functional languages on stock hardware: the spineless tagless G-machine. J. Funct. Program. **2**, 127–202 (1992)
24. Kemmerer, R.A.: Shared resource matrix methodology: an approach to identifying storage and timing channels. ACM Trans. Comput. Syst. (TOCS) **1**(3), 256–277 (1983)
25. Kobayashi, N.: Type-based information flow analysis for the π-calculus. Acta Informatica **42**(4–5), 291–347 (2005)
26. Kohlbrenner, D., Shacham, H.: Trusted browsers for uncertain times. In: USENIX Security Symposium, pp. 463–480 (2016)
27. 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)
28. Kuper, L., Todd, A., Tobin-Hochstadt, S., Newton, R.R.: Taming the parallel effect zoo: extensible deterministic parallelism with LVish. ACM SIGPLAN Not. **49**(6), 2–14 (2014)
29. Lee, J., Clifton, C.: How much is enough? Choosing ϵ for differential privacy. In: Lai, X., Zhou, J., Li, H. (eds.) ISC 2011. LNCS, vol. 7001, pp. 325–340. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24861-0_22
30. Li, P., Zdancewic, S.: Arrows for secure information flow. Theoret. Comput. Sci. **411**(19), 1974–1994 (2010)
31. Li, X., Mantel, H., Tasch, M.: Taming message-passing communication in compositional reasoning about confidentiality. In: Chang, B.-Y.E. (ed.) APLAS 2017. LNCS, vol. 10695, pp. 45–66. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-71237-6_3
32. Lipner, S., Jaeger, T., Zurko, M.E.: Lessons from VAX/SVS for high-assurance VM systems. IEEE Secur. Priv. **10**(6), 26–35 (2012)
33. Liu, J., Arden, O., George, M.D., Myers, A.C.: Fabric: building open distributed systems securely by construction. J. Comput. Secur. **25**(4–5), 367–426 (2017)
34. 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)
35. Liu, J., Myers, A.C.: Defining and enforcing referential security. In: Abadi, M., Kremer, S. (eds.) POST 2014. LNCS, vol. 8414, pp. 199–219. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54792-8_11
36. Mantel, H., Sabelfeld, A.: A unifying approach to the security of distributed and multi-threaded programs. J. Comput. Secur. **11**(4), 615–676 (2003)
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
38. Marlow, S., Newton, R., Peyton Jones, S.: A monad for deterministic parallelism. ACM SIGPLAN Not. **46**(12), 71–82 (2012)
39. Marlow, S., Peyton Jones, S.: Making a fast curry: push/enter vs. eval/apply for higher-order languages. J. Funct. Program. **16**(4–5), 415–449 (2006)
40. McCullough, D.: Specifications for multi-level security and a hook-up. In: 1987 IEEE Symposium on Security and Privacy (SP), p. 161, April 1987
41. 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, October 2012

42. Myers, A.C., Zheng, L., Zdancewic, S., Chong, S., Nystrom, N.: Jif 3.0: Java information flow, July 2006
43. Nadkarni, A., Andow, B., Enck, W., Jha, S.: Practical DIFC enforcement on android. In: USENIX Security Symposium, pp. 1119–1136 (2016)
44. North, S.C., Reppy, J.H.: Concurrent garbage collection on stock hardware. In: Kahn, G. (ed.) FPCA 1987. LNCS, vol. 274, pp. 113–133. Springer, Heidelberg (1987). https://doi.org/10.1007/3-540-18317-5_8
45. Parker, J.L.: LMonad: information flow control for Haskell web applications. Ph.D. thesis, University of Maryland, College Park (2014)
46. 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)
47. Percival, C.: Cache missing for fun and profit (2005)
48. 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, LCTES 2007, pp. 123–133. ACM, New York (2007)
49. Rafnsson, W., Jia, L., Bauer, L.: Timing-sensitive noninterference through composition. In: Maffei, M., Ryan, M. (eds.) POST 2017. LNCS, vol. 10204, pp. 3–25. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54455-6_1
50. 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)
51. Russo, A.: Functional pearl: two can keep a secret, if one of them uses Haskell. ACM SIGPLAN Not. **50**, 280–288 (2015)
52. Sabelfeld, A., Sands, D.: Probabilistic noninterference for multi-threaded programs. In: Proceedings of the 13th IEEE Workshop on Computer Security Foundations, CSFW 2000, p. 200. IEEE Computer Society, Washington, DC (2000)
53. Sestoft, P.: Deriving a lazy abstract machine. J. Funct. Program. **7**(3), 231–264 (1997)
54. 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
55. Stefan, D., Russo, A., Mazières, D., Mitchell, J.C.: Flexible dynamic information flow control in the presence of exceptions. J. Funct. Program. **27** (2017)
56. Stefan, D., Russo, A., Mitchell, J.C., Mazières, D.: Flexible dynamic information flow control in Haskell. In: Haskell Symposium. ACM SIGPLAN, September 2011
57. Stefan, D., et al.: Protecting users by confining JavaScript with COWL. In: USENIX Symposium on Operating Systems Design and Implementation. USENIX Association (2014)
58. Tsai, T.-C., Russo, A., Hughes, J.: A library for secure multi-threaded information flow in Haskell. In: 20th IEEE Computer Security Foundations Symposium, CSF 2007, pp. 187–202. IEEE (2007)
59. 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, 21–25 August 2017, pp. 37–52 (2017)
60. Vassena, M., Buiras, P., Waye, L., Russo, A.: Flexible manipulation of labeled values for information-flow control libraries. In: Askoxylakis, I., Ioannidis, S., Katsikas, S., Meadows, C. (eds.) ESORICS 2016. LNCS, vol. 9878, pp. 538–557. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45744-4_27

61. Vassena, M., Russo, A.: On formalizing information-flow control libraries. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS 2016, pp. 15–28. ACM, New York (2016)
62. Vassena, M., Russo, A., Buiras, P., Waye, L.: MAC a verified static information-flow control library. J. Log. Algebraic Methods Program. (2017)
63. Vila, P., Köpf, B.: Loophole: timing attacks on shared event loops in chrome. In: USENIX Security Symposium (2017)
64. Volpano, D., Smith, G.: Eliminating covert flows with minimum typings. In: Proceedings of the 10th IEEE Workshop on Computer Security Foundations, CSFW 1997, p. 156. IEEE Computer Society, Washington, DC (1997)
65. 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)
66. Yang, E.Z., Mazières, D.: Dynamic space limits for Haskell. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014, pp. 588–598. ACM, New York (2014)
67. Yang, J., Hance, T., Austin, T.H., Solar-Lezama, A., Flanagan, C., Chong, S.: Precise, dynamic information flow for database-backed applications. ACM SIGPLAN Not. **51**, 631–647 (2016)
68. Yang, J., Yessenov, K., Solar-Lezama, A.: A language for automatically enforcing privacy policies. ACM SIGPLAN Not. **47**, 85–96 (2012)
69. 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)
70. Zeldovich, N., Boyd-Wickizer, S., Mazieres, D.: Securing distributed systems with information flow control. NSDI **8**, 293–308 (2008)
71. 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)