Eliminating Cache-Based Timing Attacks with Instruction-Based Scheduling

Deian Stefan[1], Pablo Buiras[2], Edward Z. Yang[1], Amit Levy[1], David Terei[1],
Alejandro Russo[2], and David Mazières[1]

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

Abstract. Information flow control allows untrusted code to access sensitive and trustworthy information without leaking this information. However, the presence of covert channels subverts this security mechanism, allowing processes to communicate information in violation of IFC policies. In this paper, we show that concurrent deterministic IFC systems that use time-based scheduling are vulnerable to a cache-based internal timing channel. We demonstrate this vulnerability with a concrete attack on Hails, one particular IFC web framework. To eliminate this internal timing channel, we implement instruction-based scheduling, a new kind of scheduler that is indifferent to timing perturbations from underlying hardware components, such as the cache, TLB, and CPU buses. We show this scheduler is secure against cache-based internal timing attacks for applications using a single CPU. To show the feasibility of instruction-based scheduling, we have implemented a version of Hails that uses the CPU retired-instruction counters available on commodity Intel and AMD hardware. We show that instruction-based scheduling does not impose significant performance penalties. Additionally, we formally prove that our modifications to Hails' underlying IFC system preserve non-interference in the presence of caches.

1 Introduction

The rise of extensible web applications, like the Facebook Platform, is spurring interest in information flow control (IFC) [27, 35]. Popular platforms like Facebook give approved apps full access to users' sensitive data, including the ability to violate security policies set by users. In contrast, IFC allows websites to run untrusted, third-party apps that operate on sensitive user data [11, 21], ensuring they abide by security policies in a mandatory fashion.

Recently, Hails [11], a web-platform framework built atop the LIO IFC system [39, 40], has been used to implement websites that integrate third-party untrusted apps. For example, the code-hosting website `GitStar.com` built with Hails uses untrusted apps to deliver core features, including a code viewer and wiki. GitStar relies on LIO's IFC mechanism to enforce robust privacy policies on user data and code.

LIO, like other IFC systems, ensures that untrusted code does not write data that may have been influenced by sensitive sources to public sinks. For example, an untrusted address-book app is allowed to compute over Alice's friends list and display a stylized version of the list to Alice, but it cannot leak any information about her friends to arbitrary end-points. The flexibility of IFC makes it particularly suitable for the web, where access control lists often prove either too permissive or too restrictive.

J. Crampton, S. Jajodia, and K. Mayes (Eds.): ESORICS 2013, LNCS 8134, pp. 718–735, 2013.
© Springer-Verlag Berlin Heidelberg 2013

However, a key limitation of IFC is the presence of *covert channels*, i.e., "channels" not intended for communication that nevertheless allow code to subvert security policies and share information [22]. A great deal of research has identified and analyzed covert channels [25]. In this work, we focus on the *internal timing covert channel*, which occurs when sensitive data is used to manipulate the timing behavior of threads so that other threads can observe the order in which shared public resources are used [38, 44]. Though we do not believe our solution to the internal timing covert channel affects (either positively or negatively) other timing channels, such as the external timing covert channel, which is derived from measuring external events [1, 5, 12] (e.g., wall-clock), addressing these channels is beyond our present scope.

LIO eliminates the internal timing covert channel by restricting how programmers write code. Programmers are required to explicitly decouple computations that manipulate sensitive data from those that can write to public resources, eliminating covert channels *by construction*. However, decoupling only works when *all* shared resources are modeled. LIO only considers shared resources that are expressible by the programming language, e.g., shared-variables, file descriptors, semaphores, channels, etc. Implicit operating system and hardware state can still be exploited to alter the timing behavior of threads, and thus leak information. Reexamining LIO, we found that the underlying CPU cache can be used to introduce an internal timing covert channel that leaks sensitive data. A trivial attack can leak data at 0.75 bits/s and, despite the low bandwidth, we were able to leak all the collaborators on a private GitStar.com project in less than a minute.

Several countermeasures to cache-based attacks have previously been considered, primarily in the context of cryptosystems following the work of Kocher [18] (see Section 8). Unfortunately, many of the techniques are not designed for IFC scenarios. For example, modifying an algorithm implementation, as in the case of AES [7], does not naturally generalize to arbitrary untrusted code. Similarly, flushing or disabling the cache when switching protection domains, as suggested in [6, 49], is prohibitively expensive in systems like Hails, where context switches occur hundreds of times per second. Finally, relying on specialized hardware, such as partitioned caches [29], which isolate the effects of one partition from code using a different partition, restricts the deployability and scalability of the solution; partitioned caches are not readily available and often cannot be partitioned to an arbitrary security lattice.

This paper describes a countermeasure for cache-based attacks when execution is confined to a single CPU. Our method generalizes to arbitrary code, imposes minimal performance overhead, scales to an arbitrary security lattice, and leverages hardware features already present in modern CPUs. Specifically, we present an instruction-based scheduler that eliminates internal timing channels in concurrent programs that time-slice a single CPU and contend for the same cache, TLB, bus, and other hardware facilities. We implement the scheduler for the LIO IFC system and demonstrate that, under realistic restrictions, our scheduler eliminates such attacks in Hails web applications.

Our contributions are as follows.

- We implement a cache-based internal timing attack for LIO.
- We close the cache-based covert channel by scheduling user-level threads on a single CPU core based on the number of instructions they execute (as opposed to the

amount of time they execute). Our scheduler can be used to implement other concurrent IFC systems which implicitly assume instruction-level scheduling (e.g., [13, 14, 32, 38, 46]).

- ▶ We implement our instruction-based scheduler as part of the Glasgow Haskell Compiler (GHC) runtime system, atop which LIO and Hails are built. We use CPU performance counters, prevalent on most modern CPUs, to pre-empt threads according to the number of retired instructions. The measured impact on performance, when compared to time-based scheduling, is negligible.

  We believe these techniques to be applicable to operating systems that enforce IFC, including [20, 26, 47], though at a higher cost in performance for application code that is highly optimized for locality (see Section 5).

- ▶ We augment the LIO [40] semantics to model the cache and formally prove that instruction-based scheduling removes leaks due to caches.

The paper is organized as follows. Section 2 discusses cache-based attacks and existing countermeasures. In Section 3 presents our instruction-based scheduling solution. Section 4 describes our modifications to GHC's runtime, while Section 5 analyses their performance impact. Formal guarantees and discussions of our approach are detailed in Sections 6 and 7. We describe related work in Section 8 and conclude in Section 9.

2 Cache Attacks and Countermeasures

The severity of information leakage attacks through the CPU hardware cache has been widely considered by the cryptographic community (e.g. [28, 31]). Unlike crypto work, where attackers extract sensitive information through the execution of a fixed crypto algorithm, we consider a scenario in which the attacker provides arbitrary code in a concurrent IFC system. In our scenario, the adversary is a developer that implements a Hails app that interfaces with user-sensitive data using LIO libraries.

We found that, knowing only the cache size of the underlying CPU, we can easily build an app that exploits the shared cache to carry out an internal timing attack that leaks sensitive data at 0.75 bits/s. Several IFC systems, including [13, 14, 32, 38, 40, 46], model internal timing attacks and address them by ensuring that the outcome of a race to a public resource does not depend on secret data. Unfortunately, these systems only account for resources explicitly modeled at the programming language level and not underlying OS or hardware state, such as the CPU cache or TLB. Hence, even though the semantics of these systems rely on instruction-based scheduling (usually to simplify expressing reduction rules), real-world implementations use time-based scheduling for which the formal guarantees do not hold. The instruction-based scheduler proposed in this work can be used to make the assumptions of such concurrent IFC systems match the situation in practice. In the remainder of this section, we show the internal timing attack that leverages the hardware cache. We also discuss several existing countermeasures that could be employed by Hails.

2.1 Example Cache Attack

We mount an internal timing attack by influencing the scheduling behavior of threads through the cache. Consider the code shown in Figure 1. The attack leaks the secret

```
1.                                lowArray := new Array[M];
2.                                fillArray(lowArray)
---------------------------------------------------------------------------------
1. if secret                        |1. for i in [1..n]      |1. for i in [1..n+m]
2.   then highArray := new Array[M] |2.   skip               |2.   skip
3.        fillArray(highArray)      |3. readArray(lowArray)  |3. outputLow(0)
4.   else skip                      |4. outputLow(1)         |
---------------------------------------------------------------------------------
             thread 1               |        thread 2        |      thread 3
```

Fig. 1. A simple cache attack

boolean value secret in thread 1 by affecting when thread 2 writes to the public channel relative to thread 3.

The program starts (lines 1–2) by creating and initializing a public array lowArray whose size M corresponds to the cache size; fillArray simply sets every element of the array to 0 (this will place the array in the cache). The program then spawns three threads that run concurrently. Assuming a round-robin time-based scheduler, the execution of the attack proceeds as illustrated in Figure 2, where secret is set to true (top) and false (bottom), respectively.

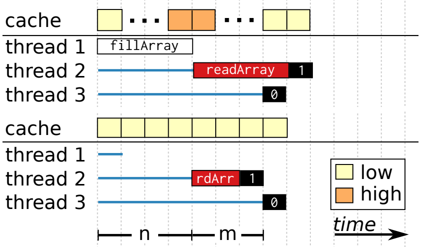

Fig. 2. Execution of the cache attack with secret true (top) and false (bottom)

- Depending on the secret value secret, thread 1 either performs a no-operation (skip on line 4), leaving the cache intact, or evicts lowArray from the cache (lines 2–3) by creating and initializing a new (non-public) array highArray.
- We assume that thread 1 takes less than n steps to complete its execution—a number that can be determined experimentally; in Figure 2, n is four. Hence, to allow all the effects on the cache due to thread 1 to settle, thread 2 delays its computation by n steps (lines 1–2). Subsequently, the thread reads every element of the public array lowArray (line 3), and finally writes 1 to a public output channel (line 4). Crucial to carrying out the attack, the duration of thread 2's reads (line 3) depends on the state of the cache: if the cache was modified by thread 1, i.e., secret is true, thread 2 needs to wait for all the public data to be retrieved from memory (as opposed to the cache) before producing an output. This requires evicting highArray from the cache and fetching lowArray, a process that takes a non-negligible amount of time. However, if the cache was not touched by thread 1, i.e., secret is false, thread 2 will get few cache misses and thus produce its output with no delay.
- We assume that thread 2 takes less than m, where m<n, steps to complete reading lowArray (line 3) when the reads hit the cache, i.e., lowArray was not replaced by highArray. Like n, this metric can be determined experimentally; in Figure 2, m is three. Using this, thread 3 simply delays its computation by n+m steps (lines 1–2) and then writes 0 to a public output channel (line 3). The role of thread 3 is solely to

serve as a baseline for thread 2's output: producing its output before thread 2 when the latter is filling the cache, i.e., secret is true; conversely, it produces an output after thread 2 if thread 1 did not touch the cache, i.e., secret is false.

We remark that the race between thread 2 and thread 3 to write to a shared public channel, influenced by the cache state, is precisely what facilitates the attack. We described how to leak a single bit, but the attack can easily be magnified by wrapping it in a loop. Note also that we have assumed the attacker has complete control of the cache—i.e., the cache is not affected by other code running in parallel. However, the attack is still plausible under weaker assumptions so long as the attacker deals with the additional noise, as exemplified by the timing attacks on AES [28].

2.2 Existing Countermeasures

The internal timing attack arises as a result of cache effects influencing thread-scheduling behavior. Hence, one series of countermeasures addresses the problem through low-level CPU features that provide better control of the cache.

Flushing the cache. Naively, we can flush the cache on every context switch. In the context of Figure 1, this guarantees that, when thread 2 executes the readArray instruction, its duration is not affected by thread 1 evicting lowArray from the cache—the cache will *always* be flushed on a context switch, hence thread 3 will always write to the output channel first.

No-fill cache mode. Several architectures, including Intel's Xeon and Pentium 4, support a cache *no-fill* mode [15]. In this mode, read/write hits access the cache; misses, however, read from and write to memory directly, leaving the cache unchanged. As considered by Zhang et al. [49], we can execute all threads that operate on non-public data in this mode. This approach guarantees that sensitive data cannot affect the cache. Unfortunately, threads operating on non-public data and relying on the cache will suffer from performance degradation.

Partitioned cache. Another approach is to partition the cache according to the number of security levels, as suggested in [49]. Using this architecture, a thread computing on secret data only accesses the secret partition, while a thread computing on public data only access the public one. This approach effectively corresponds to giving each differently-labeled thread access to its own cache and, as a result, the scheduling behavior of public threads cannot be affected by evicting data from the cache.

Unfortunately, none of the aforementioned solutions can be used in systems built with Hails (e.g., GitStar). Flushing the cache is prohibitively expensive for preemptive systems that perform a context switch hundreds of times per second—the impact on performance would gravely reduce usability. The no-fill mode solution is well suited for systems wherein the majority of the threads operate on public data. In such cases, only threads operating on sensitive data will incur a performance penalty. However, in the context of Hails, the solution is only slightly less expensive than flushing the cache. Hails threads handle HTTP requests that operate on individual (non-public) user data, hence most threads will not be using the cache. Another consequence of threads handling differently-labeled data is that partitioned caches can only be used in a limited way (see Section 8). Specifically, to address internal timing attacks, it is required that we

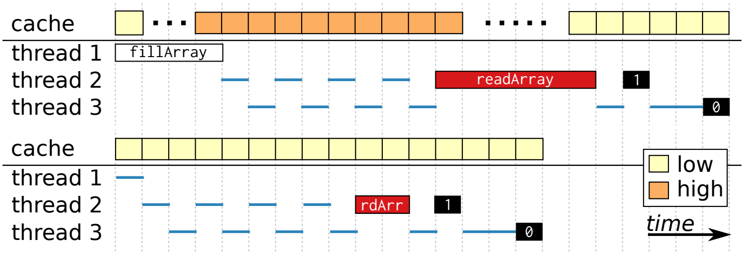

Fig. 3. Execution of cache attack program of Figure 1 with `secret` set to true (top) and false (bottom). In both executions, we highlight that the threads execute one "instruction" at a time in a round-robin fashion. The concurrent threads take the same amount of time to complete execution as in Figure 2. However, since we use instructions to context switch threads, the interleaving between `thread 2` or 3 is not influenced by the actions in `thread 1`, and thus the internal timing attack does not arise—the threads' output order cannot encode sensitive data.

partition the cache according to the number of security levels in the lattice. Given that most existing approaches can only partition caches up to 16-ways at the OS level [24], and fewer at the hardware level, an alternative scalable approach is necessary. Moreover, neither flushing nor partitioning the cache can handle timing perturbations arising from other pieces of hardware such as the TLB, buses, etc.

3 Instruction-Based Scheduling

As the example in Figure 2 shows, races to acquire public resources are affected by the cache state, which in turn might be affected by secret values. It is important to highlight that the number of instructions executed in a given quantum of time might vary depending on the state of the cache. It is precisely this variability that reintroduces dangerous races into systems. However, the actual set of instructions executed is not affected by the cache. Hence, we propose scheduling threads according to the number of instructions they execute, rather than the amount of time they consume. The point at which a thread produces an output (or any other visible operation) is determined according to the number of instructions it has executed, a measurement unaffected by the amount of time it takes to perform a read/write from memory.

Consider the code in Figure 1 executing atop an instruction-based scheduler. An illustration of this is shown in Figure 3. For simplicity of exposition, the instruction granularity is at the level of commands (`skip`, `readArray`, etc.) and therefore context switches are triggered after one command gets executed. (In Section 4, we describe a more practical and realistic instruction-based scheduler.) Observe that the amount of time it takes to execute an instruction has not changed from the time-based scheduler of Figure 2. For example, `readArray` still takes 6 units of time when `secret` is true, and 2 when it is false. Unlike Figure 2, however, the interleaving between `thread 2` and `thread 3` did not change depending on the state of the cache (which did change according to `secret`). Therefore, a race to write to the public channel between `thread 2` and `thread 3` cannot be caused by the secret, through the cache. The second thread always executes `n`+1 = 5 instructions before writing `1` to the public channel, while the third thread always executes `n`+`m`+1 = 8 instructions before writing `0`.

Our proposed countermeasure, the implementation of which is detailed in Section 4, eliminates the cache-based internal timing attacks without sacrificing scalability and with a minor performance impact. With instruction-based scheduling, we do not require flushing of the cache. In this manner, applications can safely utilize the cache to retain most of their performance without giving up system security, and unlike current partitioned caches, we can scale up to consider arbitrarily complex lattices.

4 Implementation

We implemented an instruction-based scheduler for LIO. In this section, we describe this implementation and detail some key design features we believe to be useful when modifying concurrent IFC systems to address cache-based timing attacks.

4.1 LIO and Haskell

LIO is a Haskell library that exposes concurrency to programmers in the form of "green," lightweight threads. Each LIO thread is a *native* Haskell thread that has an associated security level (label) which is used to track and control the flow of information to/from the thread. LIO relies on Haskell libraries for creating new threads and the runtime system for managing them.

In general, M lightweight Haskell threads may concurrently execute on N OS threads. (It is common, however, for multiple Haskell threads to execute on a single OS thread, i.e., $M : 1$ mapping.) The Haskell runtime, as implemented by the GHC system, uses a round-robin scheduler to context switch between concurrently executing threads. Specifically, the scheduler is invoked whenever a thread blocks/terminates or a timer signal alarm is received. The timer is used to guarantee that the scheduler is periodically executed, allowing the runtime to implement preemptive scheduling.

4.2 Instruction-Based Scheduler

As previously mentioned, timing-based schedulers render systems, such as LIO, vulnerable to cache-based internal timing attacks. We implement our instruction-based scheduler as a drop-in replacement for the existing GHC scheduler, using the number of retired instructions to trigger a context switch.

Specifically, we use performance monitoring units (PMUs) present in almost all recent Intel [15] and AMD [3] CPUs. PMUs expose hardware performance counters that are typically used by developers to optimize code—they provide metrics such as the number of cache misses, instructions executed per cycle, branch mispredictions, etc. Importantly, PMUs also provide a means for counting the number of retired instructions.

Using the perfmon2 [9] Linux monitoring interface and helper user-level library `libpfm4`, we modified the GHC runtime to configure the underlying PMU to count the number of retired instructions the Haskell process is executing. Specifically, with perfmon2 we set a data performance counter register to $2^{64} - n$, which the CPU increments upon retiring an instruction.[1] Once the counter overflows, i.e., n instructions have been retired, perfmon2 is sent a hardware interrupt. In our implementation, we configured perfmon2 to handle the interrupt by delivering a signal to the GHC runtime.

If threads share no resources, upon receiving a signal, the executing Haskell thread can immediately save its state and jump to the scheduler. However, preempting a thread which is operating on a shared memory space can be dangerous, as the thread may have left memory in an inconsistent state. (This is the case for many language runtimes, not solely GHC's.) To avoid this, GHC produces code that contains *safe points* where threads may yield. Hence, a signal does not cause an immediate preemption. Instead, the signal handler simply sets a flag indicating the arrival of a signal; at the next safe point, the thread "cooperatively" yields to the scheduler.

To ensure liveness, we must guarantee that given any point in execution, a safe point is reached in n instructions. Though GHC already inserts many safe points as a means of invoking the garbage collector (via the scheduler), tight loops that do not perform any allocation are known to hang execution [10]. Addressing this eight-year old bug, which would otherwise be a security concern in LIO, we modified the compiler to insert safe points on function entry points. This modification, integrated in the mainline GHC, has almost no effect on performance and only a 7% bloat in average binary size.

4.3 Handling IO

Threads yield at safe points in their execution paths as a result of a retired instruction signal. However, there are circumstances in which threads would like to explicitly yield prior to the reception of a retired instruction signal. In particular, when a thread performs a blocking operation, it immediately yields to the scheduler, registering itself to wake up when the operation completes. Thus, any IO action is a yield which allows the thread to give up the rest of its scheduling quantum.

While yields are not intrinsically unsafe, it is not safe to allow the leftover scheduling quantum to be passed on to the next thread. Thus, after running any asynchronous IO action, the runtime must reset the retired instruction counter. Hence, whenever a thread enters the scheduler loop due to being blocked, we reset the retired instruction counter.

5 Performance Evaluation

We evaluated the performance of instruction-based scheduling against existing time-based approaches using the nofib benchmark suite [30]. nofib is the standard benchmarking suite used for measuring the performance of Haskell implementations.

In our experimental setup, we used the latest development version of GHC (the Git master branch as of November 6, 2012). The measurements were taken on the same hardware as Hails [11]: a machine with two dual-core Intel Xeon E5620 (2.4GHz) processors, and 48GB of RAM.

[1] Though the bit-width of the hardware counters vary (they are typically 40-bits wide) perfmon2 internally manages a 64-bit counter.

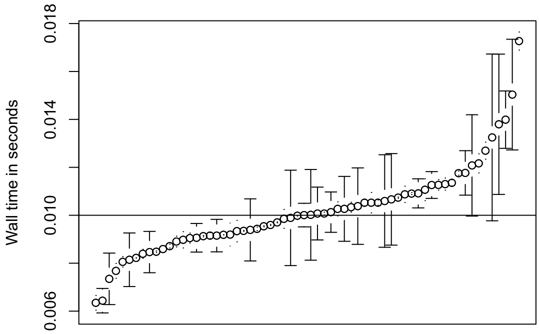

Fig. 4. Mean time between timer signal and retired-instruction signal. Each point represents a program from nofib, which have been sorted on the x-axis by their mean time.

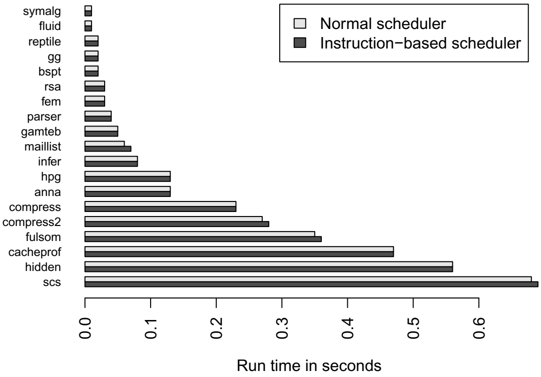

Fig. 5. Change to run time from instruction-based scheduling

We first needed to find an instruction budget—number of instructions to retire before triggering the scheduler. We found a poorly chosen instruction budget could increase runtime by 100%. To determine a good parameter, we measured the mean time between retired-instruction signals with an initially guessed instruction budget parameter. We then adjusted the parameter so the median test program had a 10 millisecond mean time-slice (the default quantum size in vanilla GHC with time-based scheduling) and verified our final choice by re-running the measurements. For our specific setup, an instruction budget of approximately 37,100,000 retired-instructions corresponded to a 10 millisecond time quantum. We plot the mean and standard deviation across all nofib applications with the final tuning parameter in Figure 4. We found that most programs receive a signal within 2 milliseconds of when they would have normally received the signal using the standard time-based scheduler. While the instruction budget parameter will vary across machines, it is relatively simple to bootstrap this parameter by performing these measurements at startup and tuning the budget accordingly.

Next, we compared the performance of Haskell's timer-based scheduler with our instruction-based scheduler. We used a subset of the nofib benchmark suite called the real benchmark, which consists of "real world programs", as opposed to synthetic benchmarks (however, results for the whole nofib suite are comparable). Figure 5 shows the run time of these programs with both scheduling approaches. With an optimized instruction budget parameter, instruction-based scheduling has no impact to the runtime of the majority of nofib applications and results in only a very slight increase in runtime for others (about 1%).

This result may seem surprising: instruction-based scheduling purposely punishes threads with good data locality, so one might expect a more substantial performance impact. We hypothesize that this is the case due to two reasons. First, with preemptive scheduling, we are already inducing cache misses when we switch from running one thread to another—instruction-based scheduling only perturbs when these preempts occur, and as seen in Figure 4, these perturbations are very minor. Second, modern L2 caches are quite large, meaning that hardware is more forgiving of poor data locality—an effect that has been measured in the behavior of stock lazy functional programs [2].

6 Cache-Aware Semantics

In this section we recall relevant design aspects of LIO [40] and extend the original formalization to consider how caches affect the timing behavior of programs. Importantly, we formalize instruction-based scheduling and show how it removes cache-based internal timing covert channels.

6.1 LIO Overview

At a high level, LIO provides the `LIO` monad, which is used in place of `IO`. Wrapping standard Haskell libraries, LIO exports a collection of functions that untrusted code may use to access the filesystem, network, shared variables, etc. Unlike the standard libraries, which usually return `IO` actions, these functions return actions in the `LIO` monad, thus allowing LIO to perform label checks before executing a potentially unsafe action.

Internally, the `LIO` monad keeps track of a *current label*, L_{cur}. The current label is effectively a ceiling over the labels of all data that the current computation may depend on. This label eliminates the need to label individual definitions and bindings: symbols in scope are (conceptually) labeled with L_{cur}.[2] Hence, when a computation C, with current label L_C, observes an object labeled L_O, C's label is raised to the least upper bound or *join* of the two labels, written $L_C \sqcup L_O$. Importantly the current label governs where the current computation can write, what labels may be used when creating new channels or threads, etc. For example, after reading O, the computation should not be able to write to a channel K if L_C is more restricting than L_K—this would potentially leak sensitive information (about O) into a less sensitive channel.

Note that an LIO computation can only execute a sub-computation on sensitive data by either raising its current label or forking a new thread in which to execute this sub-computation. In the former case, raising the current label prevents writing to less sensitive endpoints. In the latter case, to observe the result (or timing and termination behavior) of the sub-computation the thread must wait for the forked thread to finish, which

[2] As described in [39], LIO does, however, allow programmers to heterogeneously label data they consider sensitive.

$$\frac{\lBrace \Sigma, \langle \sigma, e \rangle \rBrace_{\zeta} \longrightarrow_k \lBrace \Sigma', \langle \sigma', e' \rangle \rBrace_{\zeta'} \qquad q > 0}{\lBrace \Sigma, \zeta, q, \langle \sigma, e \rangle \lhd t_s \rBrace \hookrightarrow \lBrace \Sigma', \zeta', q - k, \langle \sigma', e' \rangle \lhd t_s \rBrace}\ (\text{STEP}) \qquad \frac{q \leq 0}{\lBrace \Sigma, \zeta, q, t \lhd t_s \rBrace \hookrightarrow \lBrace \Sigma', \zeta, q_i, t_s \rhd t \rBrace}\ (\text{PREEMPT})$$

Fig. 6. Semantics for threadpools under round-robin time-based scheduling

first raises the current label. A consequence of this design is that differently-labeled computations are decoupled, which, as mentioned in Section 1, is key to eliminating the internal timing covert channel.

In the next subsection, we will outline the semantics for a cache-aware, time-based scheduler where the cache attack described in Section 2 is possible. Moreover, we show that we can easily adapt this semantics to model the new LIO instruction-based scheduler. Interested readers may refer to the extended version of the paper, which can be found online at [41].

6.2 Cache-Aware Semantics

We model the underlying CPU cache as an abstract memory shared among all running threads, which we will denote with the symbol ζ. Every step of the sequential execution relation will affect ζ according to the current instruction being executed, the runtime environment, and the existing state of the cache. As in [40], each LIO thread has a thread-local runtime environment σ, which contains the current label $\sigma.\texttt{lbl}$. The global environment Σ, common to all threads, holds references to shared resources.

In addition, we explicitly model the number of machine cycles taken by a single execution step as a result of the cache. Specifically, the transition $\zeta \rightharpoonup_k^{(\Sigma,\sigma,e)} \zeta'$ captures the parameters that influence the cache (Σ, σ, and e) as well as the number of cycles k it takes for the cache to be updated.

A *cache-aware* evaluation step is obtained by merging the reduction rule of LIO with our formalization of CPU cache as given below:

$$\frac{\lBrace \Sigma, \langle \sigma, e \rangle \rBrace \overset{\gamma}{\rightharpoonup} \lBrace \Sigma', \langle \sigma', e' \rangle \rBrace \qquad \zeta \rightharpoonup_k^{(\Sigma,\sigma,e)} \zeta' \qquad k \geq 1}{\lBrace \Sigma, \langle \sigma, e \rangle \rBrace_{\zeta} \xrightarrow{\gamma}_k \lBrace \Sigma', \langle \sigma', e' \rangle \rBrace_{\zeta'}}$$

We read $\lBrace \Sigma, \langle \sigma, e \rangle \rBrace_{\zeta} \xrightarrow{\gamma}_k \lBrace \Sigma', \langle \sigma', e' \rangle \rBrace_{\zeta'}$ as "the configuration $\lBrace \Sigma, \langle \sigma, e \rangle \rBrace$ reduces to $\lBrace \Sigma', \langle \sigma', e' \rangle \rBrace$ in one step, but k machine cycles, producing event γ and modifying the cache from ζ to ζ'." As in LIO [40], the relation $\lBrace \Sigma, \langle \sigma, e \rangle \rBrace \overset{\gamma}{\rightharpoonup} \lBrace \Sigma', \langle \sigma', e' \rangle \rBrace$ represents a single execution step from thread expression e, under the run-time environments Σ and σ, to thread expression e' and run-time environments Σ' and σ'. Events are used to communicate information between the threads and the scheduler, e.g., when spawning new threads.

Figure 6 shows the most important rules of our time-based scheduler in the presence of cache effects. We elide the rest of the rules for brevity. The relation $\hookrightarrow$ represents a single evaluation step for the program threadpool, in contrast with $\longrightarrow$ which is only for a single thread. Configurations are of the form $\lBrace \Sigma, \zeta, q, t_s \rBrace$, where q is the number of cycles available in the current time slice and t_s is a queue of thread configurations of the form $\langle\sigma,e\rangle$. We use a standard deque-like interface with operations $\lhd$ and $\rhd$ for front and back insertion, respectively, i.e., $\langle\sigma,e\rangle \lhd t_s$ denotes a threadpool in which the first thread is $\langle\sigma,e\rangle$ while $t_s \rhd \langle\sigma,e\rangle$ indicates that $\langle\sigma,e\rangle$ is the last one.

As in LIO, threads are scheduled in a round-robin fashion. Our scheduler relies on the number of cycles that each step takes; we respectively write q_i and q as the initial and remaining number of cycles assigned to a thread in each quantum. In rule (STEP), the number of cycles k that the current instruction takes is reflected in the scheduling quantum. Consequently, threads that compute on data that is not present in the cache will take more cycles, i.e., have a higher k, so they will run "slower" because they are allowed to perform fewer reduction steps in the remaining time slice. In practice, this permits attacks, such as that in Figure 1, where the interleaving of the threads can be affected by sensitive data. Rule (PREEMPT) is used when the thread has exhausted its cycle budget, triggering a context switch by moving the current thread to the end of the queue.

We can adapt this semantics to reflect the behavior of the new instruction-based scheduler. To this end, we replace the number of cycles q with an instruction budget; we write b_i for the initial instruction budget and b for the current budget. Crucially, we change rule (STEP) into rule (STEP-CA), given by

$$\text{(STEP-CA)}\quad \frac{\{\!|\Sigma,\langle\sigma,e\rangle|\!\}_{\zeta} \longrightarrow_k \{\!|\Sigma',\langle\sigma',e'\rangle|\!\}_{\zeta'} \qquad b>0}{\{\!|\Sigma,\zeta,b,\langle\sigma,e\rangle \lhd t_s|\!\} \hookrightarrow \{\!|\Sigma',\zeta',b-1,\langle\sigma',e'\rangle \lhd t_s|\!\}}\,.$$

Rule (STEP-CA) executes a sequential instruction in the current thread, provided the instruction budget is not empty ($b>0$), and updates the cache accordingly ($\{\!|\Sigma,\langle\sigma,e\rangle|\!\}_{\zeta} \longrightarrow_k \{\!|\Sigma',\langle\sigma',e'\rangle|\!\}_{\zeta'}$). It is important to remark that the effects of the underlying cache ζ, as indicated by k, are intentionally ignored by the scheduler. This subtle detail captures the essence of removing the cache-based internal timing channel. (Our formalization of a time-based scheduler does not ignore k and thus is vulnerable.) Similarly, rule (PREEMPT) turns into rule (PREEMPT-CA), where q and q_i are respectively replaced with b and b_i to reflect the fact that there is an instruction budget instead of a cycle count. The rest of the rules can be adapted in a straightforward manner. Our rules have the invariant that the instruction budget gets decremented by one when a thread executes one instruction.

By changing the cache-aware semantics in this way, we obtain a generalized semantics for LIO. In fact, the previous semantics for LIO [40], is a special case, with $b_i = 1$, i.e., the threads perform only one reduction step before a context-switch happens. In addition, it is easy to extend our previous termination-sensitive non-interference result to the instruction-based semantics. The security guarantees of our approach are stated below.

Theorem 1 (Termination-sensitive non-interference). *Given a program function f, an attacker that observes data at level L, and a pair of inputs e_1 and e_2 indistinguishable to the attacker, then for every reduction sequence starting from $f(e_1)$ there is a corresponding reduction sequence starting from $f(e_2)$ such that both sequences reach indistinguishable configurations.*

Proof Sketch: Our proof relies on the *term erasure technique* as used in [23, 34, 39], and follows in a similar fashion to that of [40]. We refer the interested reader to the extended version of the paper for details [41].

7 Limitations

This section discusses some limitations of our current implementation, the significance of these limitations, and how the limitations can be addressed.

Nondeterminism in the hardware counters. While the retired-instruction counter should be deterministic, in most hardware implementations there is some degree of nondeterminism. For example, on most x86 processors the instruction counter adds an extra instruction every time a hardware interrupt occurs [45]. This anomaly could be exploited to affect the behavior of an instruction-based scheduler, causing it to trigger a signal early. However, this is only a problem if a high thread is able to cause a large number of hardware interrupts in the underlying operating system. In the Hails framework, attackers can trigger interrupts by forcing a server to frequently receive HTTP responses, i.e., trigger a hardware interrupt from the network interface card. Hails, however, provides mechanisms to mitigate the effects of external events, using the techniques of [4, 48], that can reduce the frequency of such operations. Nevertheless, the feasibility of such attacks is not directly clear and left as future work.

Scheduler and garbage collector instruction counts. For performance reasons, we do not reset the retired-instruction counter prior to re-entering user code. This means that instruction counts include the instructions executed from when the previous thread received the signal, to when the previous thread yields, to when the next thread is scheduled. While this suggests that thread are not completely isolated, we think that this interaction is extremely difficult to exploit. This is because the number of instructions it takes for the scheduler to schedule a new thread is essentially fixed, and the "time to yield" for any code is highly dependent on the compiler, which we assume is not under the control of an adversary.

Parallelism. Unfortunately, we cannot simply run instruction-based scheduling on multiple cores. Threads running in parallel will be able to race to public resources. Under normal conditions, such races can be still influenced by the state of the (L3) cache. Some parallelism is, however, possible. For instance, we can extend the instruction-based scheduler to parallelize regions of code that do not share state or have side effects (e.g., synchronization operations or writes to channels). To this end, when a thread wishes to perform a side effect, it is required that all the other threads lagging behind (as per retired-instruction count) first complete the execution of their side effects. Hence, an implementation would rely on a synchronization barrier whenever a side-effecting computation is executed; at the barrier, the execution of all the side effects is done in a pre-determined order. Although we believe that this "optimization" is viable, we have not implemented it, since it requires major modifications to the GHC runtime system and the performance gains due to parallelism requiring such strict synchronization barriers are not clear. We leave this investigation to future work.

Even without built-in parallelism, we believe that instruction-based scheduling represents a viable and deployable solution when considering modern web applications and data-centers. In particular, when an application is distributed over multiple machines, these machines do not share a processor cache and thus can safely run the application concurrently. Attacks which involve making these two machines access shared external resources can be mitigated in the same fashion as external timing attacks [4, 40, 48, 49]. Load-balancing an application in this manner is already a well-established technique for deploying applications.

8 Related Work

Impact of cache on cryptosystems. Kocher [18] was one of the first to consider the security implications of memory access-time in implementations of cryptographic primitives and systems. Since then, several attacks (e.g., [28, 31]) against popular systems have successfully extracted secret keys by using the cache as a covert channel. As a countermeasure, several authors propose partitioning the cache (e.g., [29]). Until recently, partitioned caches have been of limited application in dynamic information flow control systems due to the small number of partitions available. The recent Vantage cache partition scheme of Sanchez and Kozyrakis [37], however, offers tens to hundreds of configurable partitions and high performance. As hardware is not yet available with Vantage, it is hard to evaluate its effectiveness for our problem domain. However, we expect it to be mostly complimentary to our instruction-based scheduler. Specifically, a partitioned cache can be used to safely run threads in parallel, each group of threads using instruction-based schedulers. Other countermeasures (e.g., [28]) are primarily implementation-specific, and, while applicable to cryptographic primitives, they do not easily generalize to arbitrary code.

Language-based information-flow security. Several works (e.g., [13]) consider systems that satisfy *possibilistic non-interference* [38], which states that a concurrent program is secure iff the possible observable events do not depend on sensitive data. An alternative notion, *probabilistic non-interference*, considers a concurrent program secure iff the probability distribution over observable events is not affected by sensitive data [44]. Zdancewic and Myers introduce *observational low-determinism* [46], which intuitively states that the observable behavior of concurrent systems must be deterministic. After this seminal work, several authors improve on each other's definitions on low-determinism (e.g., [14]). Other IFC systems rely on deterministic semantics and a determined class of runtime schedulers (e.g., [32]).

The lines of work mentioned above assume that the execution of a single step is performed in a single unit of time, corresponding to an instruction, and show that races to publicly-observable events cannot be influenced by secret data. Unfortunately, the presence of the cache breaks the correspondence between an instruction and a single unit of time, making cache attacks viable. Instruction-based scheduling could be seen as a necessary component in making the previous concurrent IFC approaches practical.

Agat [1] presents a code transformation for sequential programs such that both code paths of a branch have the same memory access pattern. This eliminates timing covert channels, even those relying on the cache. This transformation has been adapted by several authors (e.g., [36]). This approach, however, focuses on avoiding attacks relying on the data cache, while leaving the instruction cache unattended.

Russo and Sabelfeld [33] consider non-interference for concurrent systems under cooperative and deterministic scheduling. An implementation of such a system was presented by Tsai et al. in [42]. This approach eliminates internal timing leaks, including those relying on the cache, by restricting the use of yields. Cooperative schedulers are intrinsically vulnerable to attacks that use termination as a covert channel. In contrast, our solution is able to safely preempt non-terminating computations while guaranteeing termination-sensitive non-interference.

Secure multi-execution [8] preserves confidentiality of data by executing the same sequential program several times, one for each security level. In this scenario, the cache-based covert channel can only be removed in specific configurations [16]. Zhang et al. [49] provide a method to mitigate external events when their timing behavior could be affected by the underlying hardware. This solution is directly applicable to our system when considering external events. Similar to our work, they consider an abstract model of the hardware machine state which includes a description of time. However, their semantics focus on sequential programs, wherein attacks due to the cache arise in the form of externally visible events.

Hedin and Sands [12] present a type-system for preventing external timing attacks for bytecode. Their semantics is augmented to incorporate history, which enables the modeling of cache effects. We proceed in a similar manner when extending the original LIO semantics [40] to consider caches.

System security. In order to achieve strong isolation, Barthe et al. [6] present a model of virtualization which flushes the cache upon switching between guest operating systems. Different from our scenario, flushing the cache in such scenarios is common and does not impact the already-costly context-switch.

Allowing some information leakage, Köpft et al. [19] combines abstract interpretation and quantitative information-flow to analyze leakage bounds for cache attacks. Kim et al. [17] propose StealthMem, a system level protection against cache attacks. StealthMem allows programs to allocate memory which does not get evicted from the cache. In fact, this approach could be seen as a software-level partition of the cache. StealthMem is capable of enforcing confidentiality for a stronger attacker model than ours, i.e., they consider programs with access to wall-clock and perhaps running on multi-cores. As other works on partition caches, StealthMem does not scale to scenarios with arbitrarily complex security lattices.

Performance monitoring counters. The use of PMUs for tasks other than performance monitoring is a relatively recent one. Vogl and Ekert [43] also use PMUs, but for monitoring applications running within a virtual machine, allowing instruction level monitoring of all or specific instructions. While the mechanism is the same, our goals are different: we merely seek to replace interrupts generated by a clock-based timer with interrupts generated by hardware counters; their work introduces new interrupts that trigger vmexits. This causes a considerable slowdown, while we achieve no major performance impact.

9 Conclusion

Cache-based internal timing attacks constitute a practical set of attacks. We present instruction-based scheduling as a solution to remove such attacks. Different from simply flushing the cache on a context switch or partitioning the cache, this new class of schedulers also removes timing perturbations introduced by other components of the underlying hardware (e.g., the TLB, CPU buses, etc.). To demonstrate the applicability of our solution, we implemented a scheduler using the CPU retired-instruction counters available on commodity Intel and AMD hardware. We integrated the scheduler into the Hails IFC web framework, replacing the timing-based scheduler. This integration was, in part, possible because of the scheduler's negligible performance impact and, in part, due to our formal guarantees. Specifically, by generalizing previous results, we proved that instruction-based scheduling for LIO preserves confidentiality and integrity of data, i.e., termination-sensitive non-interference. Finally, we remark that our design, implementation, and proof are not limited to LIO; we believe that instruction-based scheduling is applicable to other concurrent deterministic IFC systems where cache-based timing attacks could be a concern.

Acknowledgments. We would like to thank David Sands for useful comments. This work was funded by DARPA CRASH under contract #N66001-10-2-4088, by multiple gifts from Google, and by the Swedish research agency VR, STINT, and the Barbro Osher foundation. Deian Stefan is supported by the DoD through the NDSEG Fellowship Program.

References

[1] Agat, J.: Transforming out timing leaks. In: Proc. ACM Symp. on Principles of Programming Languages, pp. 40–53 (January 2000)

[2] Ahmad, A., DeYoung, H.: Cache performance of lazy functional programs on current hardware. Technical report, CMU (December 2009)

[3] AMD. BIOS and kernel developer's guide for AMD family 11h processors (July 2008)

[4] Askarov, A., Zhang, D., Myers, A.C.: Predictive black-box mitigation of timing channels. In: Proc. of the 17th ACM CCS, ACM (2010)

[5] Barthe, G., Rezk, T., Warnier, M.: Preventing timing leaks through transactional branching instructions. Electron. Notes Theor. Comput. Sci. 153 (May 2006)

[6] Barthe, G., Betarte, G., Campo, J., Luna, C.: Cache-leakage resilient OS isolation in an idealized model of virtualization. In: 2012 IEEE 25th Computer Security Foundations Symposium (CSF). IEEE Computer Society (June 2012)

[7] Bonneau, J., Mironov, I.: Cache-collision timing attacks against AES. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 201–215. Springer, Heidelberg (2006)

[8] Devriese, D., Piessens, F.: Noninterference through secure multi-execution. In: Proc. of the 2010 IEEE Symposium on Security and Privacy, SP 2010. IEEE Computer Society (2010)

[9] Eranian, S.: Perfmon2: a flexible performance monitoring interface for Linux. In: Proc. of the 2006 Ottawa Linux Symposium, pp. 269–288. Citeseer (2006)

[10] GHC. Infinite loops can hang Concurrent Haskell (2005), `http://hackage.haskell.org/trac/ghc/ticket/367`

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

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

[13] Honda, K., Vasconcelos, V.T., Yoshida, N.: Secure information flow as typed process behaviour. In: Smolka, G. (ed.) ESOP 2000. LNCS, vol. 1782, pp. 180–199. Springer, Heidelberg (2000)

[14] Huisman, M., Worah, P., Sunesen, K.: A temporal logic characterisation of observational determinism. In: Proc. IEEE Computer Sec. Foundations Workshop (July 2006)

[15] Intel. Intel 64 and IA-32 architectures software developer's manual (August 2012)

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

[17] Kim, T., Peinado, M., Mainar-Ruiz, G.: STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud. In: Proceedings of the 21st USENIX Conference on Security Symposium, Security 2012. USENIX Association (2012)

[18] Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996)

[19] Köpf, B., Mauborgne, L., Ochoa, M.: Automatic quantification of cache side-channels. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 564–580. Springer, Heidelberg (2012)

[20] Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M.F., Kohler, E., Morris, R.: Information flow control for standard OS abstractions. In: Proc. of the 21st Symp. on Operating Systems Principles (October 2007a)

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

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

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

[24] Lin, J., Lu, Q., Ding, X., Zhang, Z., Zhang, X., Sadayappan, P.: Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In: Proc. of the Intl. Symposium on High Performance Computer Architecture. IEEE (2008)

[25] Millen, J.: 20 years of covert channel modeling and analysis. In: IEEE Symp. on Security and Privacy (1999)

[26] Murray, T., Matichuk, D., Brassil, M., Gammie, P., Bourke, T., Seefried, S., Lewis, C., Gao, X., Klein, G.: sel4: from general purpose to a proof of information flow enforcement. In: Proceedings of the 34th IEEE Symp. on Security and Privacy (2013)

[27] Myers, A.C., Liskov, B.: A decentralized model for information flow control. In: Proc. of the 16th ACM Symp. on Operating Systems Principles, pp. 129–142 (1997)

[28] Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: the case of AES. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 1–20. Springer, Heidelberg (2006)

[29] Page, D.: Partitioned cache architecture as a side-channel defence mechanism. IACR Cryptology ePrint Archive 2005 (2005)

[30] Partain, W.: The nofib benchmark suite of Haskell programs. In: Proceedings of the 1992 Glasgow Workshop on Functional Programming (1992)

[31] Percival, C.: Cache missing for fun and profit. In: Proc. of BSDCan 2005 (2005)

[32] Russo, A., Sabelfeld, A.: Securing interaction between threads and the scheduler. In: Proc. IEEE Computer Sec. Foundations Workshop, pp. 177–189 (July 2006a)

[33] Russo, A., Sabelfeld, A.: Security for multithreaded programs under cooperative scheduling. In: Virbitskaite, I., Voronkov, A. (eds.) PSI 2006. LNCS, vol. 4378, pp. 474–480. Springer, Heidelberg (2007)

[34] Russo, A., Claessen, K., Hughes, J.: A library for light-weight information-flow security in Haskell. In: Proc. ACM SIGPLAN Symposium on Haskell, pp. 13–24. ACM Press (September 2008)

[35] Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications 21(1) (January 2003)

[36] Sabelfeld, A., Sands, D.: Probabilistic noninterference for multi-threaded programs. In: Proc. IEEE Computer Sec. Foundations Workshop, pp. 200–214 (July 2000)

[37] Sanchez, D., Kozyrakis, C.: Vantage: Scalable and efficient fine-grain cache partitioning. In: International Symposium on Computer Architecture. ACM IEEE (2011)

[38] Smith, G., Volpano, D.: Secure information flow in a multi-threaded imperative language. In: Proc. ACM Symp. on Principles of Programming Languages, pp. 355–364 (January 1998)

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

[40] 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: Proc. of the 17th ACM SIGPLAN International Conference on Functional Programming (September 2012)

[41] Stefan, D., Buiras, P., Yang, E., Levy, A., Terei, D., Russo, A., Mazières, D.: Eliminating cache-based timing attacks with instruction-based scheduling: Extended version (2013), `http://www.cse.chalmers.se/~buiras/esorics2013_extended.pdf`

[42] Tsai, T.C., Russo, A., Hughes, J.: A library for secure multi-threaded information flow in Haskell. In: Proc. IEEE Computer Sec. Foundations Symposium (July 2007)

[43] Vogl, S., Eckert, C.: Using Hardware Performance Events for Instruction-Level Monitoring on the x86 Architecture. In: Proceedings of the 2012 European Workshop on System Security EuroSec 2012 (2012)

[44] Volpano, D., Smith, G.: Probabilistic noninterference in a concurrent language. J. Computer Security 7(2-3) (November 1999)

[45] Weaver, V.M., McKee, S.A.: Can hardware performance counters be trusted? Workload Characterization 08 (2008), `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4636099`

[46] Zdancewic, S., Myers, A.C.: Observational determinism for concurrent program security. In: Proc. IEEE Computer Sec. Foundations Workshop, pp. 29–43 (June 2003)

[47] Zeldovich, N., Boyd-Wickizer, S., Kohler, E., Mazières, D.: Making information flow explicit in HiStar. In: Proc. of the 7th Symp. on Operating Systems Design and Implementation, Seattle, WA, pp. 263–278 (November 2006)

[48] Zhang, D., Askarov, A., Myers, A.C.: Predictive mitigation of timing channels in interactive systems. In: Proc. of the 18th ACM CCS. ACM (2011)

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