Detector$^+$: An Approach for Detecting, Isolating, and Preventing Timing Attacks

Arsalan Javeed, Cemal Yilmaz*, Erkay Savas

Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul 34956, Turkey

Abstract

In this work, we present a novel approach, called *Detector*$^+$, to detect, isolate, and prevent timing-based side channel attacks (i.e., timing attacks) at runtime. The proposed approach is based on a simple observation that the time measurements required by the timing attacks differ from those required by the benign applications as these attacks need to measure the execution times of typically quite short-running operations. Detector$^+$, therefore, monitors the time readings made by processes and mark consecutive pairs of readings that are close to each other in time as suspicious. In the presence of suspicious time measurements, Detector$^+$ introduces noise into the measurements to prevent the attacker from extracting information by using these measurements. The sequence of suspicious time measurements are then analyzed by using a sliding window based approach to pinpoint the malicious processes at runtime. We have empirically evaluated the proposed approach by using five well known timing attacks, including Meltdown, together with their variations, representing some of the mechanisms that an attacker can employ to become stealthier. In one evaluation setup, each type of attack was carried out concurrently by multiple processes. In the other setup, multiple types of attacks were carried out concurrently. In all the experiments, Detector$^+$ detected all the malicious time measurements with almost a perfect accuracy, prevented all the attacks, and correctly pinpointed all the malicious processes involved in the attacks without any false positives after they have made a few time measurements with an average runtime overhead of 1.56%.

Keywords: side channel attacks, timing attacks, runtime attack detection, isolation, and prevention

1. Introduction

Side channels enable an attacker to infer information about a secret by measuring and analyzing the information unintentionally leaked by a computing system, such as execution times and power consumption. Over the recent years, research on the *side channel attacks* has revealed numerous novel ways to exfiltrate secret information, which is otherwise proved to be challenging [1–3].

An important category of side channel attacks, which is the focus of this paper, is *timing attacks* [4–6], such as Meltdown [7], Evict+Reload [8], Flush+Flush [9], Flush+Reload [10], and Prime+Probe [11]. At a very high level, timing attacks exploit the differences between the execution times of certain operations. For example, the Meltdown attack [7] leverages the observable time differences between fetching data from the cache and from the RAM memory to exfiltrate private data belonging to other processes. Similarly, cache-based timing attacks, including Evict+Reload, Flush+Flush, Flush+Reload, and Prime+Probe, analyze the time differences between cache and memory fetches to infer the secret keys processed by cryptographic applications [8–11].

In a series of previous works, we demonstrated that side channel attacks, which leverage information unintentionally leaked by the systems under attack, ironically leak information by themselves through the same or related channels, which can be analyzed to detect, isolate, and prevent ongoing attacks at runtime [12–14].

More specifically, in [12], we have developed a number of approaches for detecting cache-based timing attacks, which operate by monitoring the contentions in L1 cache memory and emitting warnings about possible ongoing attacks when the contentions reach a "suspicious" level. In [14], we have monitored the contentions in L3 cache memory to detect the same or similar types of attacks by identifying similarities in cache access patterns between processes and known attacks. And, in [13], we have monitored segmentation faults occurring at memory addresses that are close to each other to detect, isolate, and prevent the Meltdown attacks.

In all of these aforementioned works, we obtained quite promising results [12–14]. One issue, however, was that we had to develop a different approach for each type of attack. In particular, we had to analyze each attack in isolation and figure out what needs to be monitored (e.g., L1/L3 cache memory accesses or segmentation faults), what type of data to be collected (e.g., L1/L3 miss ratio or addresses at which segmentation faults occur), how to analyze the collected data, and when and how to take the countermeasures to prevent the ongoing attacks.

We, therefore, believe that developing a specialized approach for detecting each different type of timing attacks may not be sustainable in the long run. One issue is that so-

*Corresponding author

Email addresses: ajaveed@sabanciuniv.edu (Arsalan Javeed), cemal.yilmaz@sabanciuniv.edu (Cemal Yilmaz), erkay.savas@sabanciuniv.edu (Erkay Savas)

phisticated strategies may need to be developed to ensure the interference-free deployment of multiple detection approaches, so that the systems can reliably be protected from different types of attacks. Similarly, the collective accuracy of the existing attack detection approaches [12–18] in the presence of multiple different types of attacks carried out concurrently, is (to the best of our knowledge) yet to be evaluated. Another issue is that as the aforementioned strategy requires to analyze each timing attack in detail, it may not necessarily be suitable for zero-day attacks. For example, although the approaches proposed in [12–14] has a chance of detecting previously unknown attacks that leverage the same shared resources (e.g., L1 and L3 cache memory) monitored by these approaches, an attack utilizing a completely different shared resource would render them useless.

In this work, we, therefore, develop a generic approach, called *Detector*$^+$ (named after Kleene plus), to detect, isolate, and prevent timing attacks. The proposed approach is based on a simple observation: All timing attacks need to measure time, but their timing behaviors differ from those of the benign applications, especially the ones running in the production environments. In this context, we define the *timing behavior* of a process as the "typical" durations, which are needed to be timed by the process. More specifically, we observe that the malicious processes, compared to benign processes, often need to measure the execution times of quite shorter-running operations, such as accessing a single memory location. Note that making a *time measurement*, i.e., measuring the execution time of an operation, requires two time readings: one before the operation and the other after the operation, such that the execution time is computed as the difference between these readings. And, when the duration to be timed, is short, these two time readings occur close to each other in time.

Detector$^+$, therefore, monitors the time readings at runtime on a per process basis. When the time difference between two consecutive readings is "suspiciously" low, Detector$^+$ marks the pair as a suspicious measurement and introduces noise into the actual measurement to prevent possible ongoing attacks. The sequence of suspicious time measurements are then analyzed by using a sliding window-based approach to pinpoint the malicious processes, so that appropriate actions can be taken in time. These countermeasures are, however, beyond the scope of this work.

To evaluate the proposed approach, we have conducted a series of experiments by using five well-known timing attacks (Meltdown [7], Evict+Reload [8], Flush+Flush [9], Flush+Reload [10], and Prime+Probe [11]) together with a well-known suite of benign applications [19], representing the applications that are commonly encountered in production environments. To further evaluate Detector$^+$, we have also tested it on different variations of the aforementioned attacks, each of which represents a mechanism that an attacker can employ to become stealthier. In one type of variation, each attack was carried out concurrently by multiple processes. In another variation, multiple types of attacks were carried out concurrently. In all the experiments, Detector$^+$ detected all the malicious time measurements with almost a perfect accuracy, prevented all the attacks, and correctly pinpointed all the malicious processes involved in the attacks without any false positives after they have made a few time measurements with an average runtime overhead of 1.56%.

The remainder of paper is organized as follows: Section 2 provides background information on the timing attacks used in the paper; Section 3 presents the attacker model; Section 4 introduces the proposed approach; Section 5 presents the experiments; Section 6 evaluates the potential threats to validity; Section 7 discusses the countermeasures that can be taken against Detector$^+$; Section 8 presents related work; and Section 9 concludes with possible directions for future work.

2. Background

In this section, we present background information about the timing attacks used in the paper without any intention of discussing all the details. The interested reader can get more information about these attacks by following the citations provided.

2.1. Meltdown

In a Meltdown attack [7], the malicious process aims to access memory locations that belong to other processes. To this end, the malicious process attempts to use the value of a byte, which it does not have any rights to access, as an index into an adversarial array, which is purposefully created by the malicious process. The attacker is aware of the fact the request will eventually fail with a runtime error, such as with a SIGSEGV signal representing a segmentation fault. Due to out-of-order execution, however, the error occurs after the indexed item in the adversarial array was brought to the cache memory, which was intentionally flushed by the malicious process before the access. Although, the value of the target byte is never returned to the malicious process, the microarchitectural state of the CPU has now changed, leaking information; the item, the index of which is the value of the target byte, is now in the cache memory. To figure out the index that was accessed, thus the value of the target byte, the malicious process then probes the cache by accessing the indices in its adversarial array and each access is timed. Since the cache was flushed by the malicious process before the access, the index that takes the shortest amount time to access would indicate that the respective item is actually fetched from the cache memory (rather than the RAM memory), which, in turn, reveals the value of the target byte. Other bytes can then be targeted (as needed) by using the same mechanism.

Note that there are different variations of the Meltdown attack [7]. All of these variations, however, make the same type of time measurements. In particular, they all measure the time it takes to access a single memory location. Therefore, from the perspective of Detector$^+$, there is no difference between these variations. Consequently, we opted to use the original version of the attack (as described above) in this work without losing the generality.

2.2. Prime+Probe

The Prime+Probe attack [5, 11, 20, 21] has two steps; prime and probe. The malicious process carried out these steps one after another in a loop to exfiltrate information from a victim process. In the prime step, the malicious process fills the whole cache memory with its own data. It then spends an ample amount of idle time waiting for the victim process to utilize the cache. Then, in the probe step, the malicious process probes the data that it brought to the cache memory in the prime step to figure out the data (thus, the cache lines/sets) that was evicted from the cache. To this end, each memory access is timed and the ones that take longer than the others, indicate the ones that were evicted from the cache, presumably by the victim process. The malicious process then uses this information to infer a secret about the victim process (such as, the secret key processed by the victim [5, 11]).

2.3. Evict+Reload

The Evict+Reload attack [8, 22] exfiltrates information from a victim process by figuring out how frequently the victim process uses different code segments in shared libraries. In the evict step, the malicious process evicts a portion of the shared library from the cache. As the victim process accesses the shared library the respective parts are brought to the cache. In the reload step, the malicious process accesses the portions that it evicted in the first step. Each access is timed. The accesses that take shorter time than others, indicate the parts of the shared library that was (presumably) accessed by the victim process.

2.4. Flush+Reload

The Flush+Reload attack [10, 23] uses a special machine instruction, called `clflush`, to operate. In the flush step, the malicious process evicts the entries of interest from all levels of the cache hierarchy by using the aforementioned instruction. Then, in the reload step, the malicious process measures the time it takes to access the entries evicted in the first step. Shorter access times indicate the entries that were (presumably) brought to the cache by the victim process.

2.5. Flush+Flush

The Flush+Flush attack [9], as was the case with the Flush+Reload attack, leverages the `clflush` instruction. Unlike the Flush+Reload attack, however, this attack measures the execution time of the `clflush` instruction, rather than the execution time of a memory access. The rationale is that if the entry to be evicted is already in the cache, then `clflush` takes longer to execute as the entry needs to be removed from all levels of the cache hierarchy. Otherwise, `clflush` takes shorter time as there is nothing to be evicted.

3. Attacker Model

In this section, we present a number of definitions to model the timing attacks.

Definition 1. *An* **attacker** *is a party, which controls either a single or group of user space processes on a target platform, with malicious intentions to exfiltrate sensitive information by snooping the private data manipulated by other processes.*

Definition 2. *A* **timing attacker** *is a type of attacker, who operates by utilizing the differences between the execution times of certain operations as an essential component of its attack mechanism.*

Remark 1. *The operations that need to be timed are short living operations. To measure the execution time of an operation, the timing attacker requires two time readings; one before the operation and another after the operation. Both readings are carried out by the same process executing on a machine where Detector$^+$ is operational, such as on the same machine with the system under attack. Note that this does not prevent the attacker from using multiple processes in an attack. Furthermore, time measurements may need to be repeated both to factor out the noise and/or to exfiltrate more data.*

Remark 2. *The timing attacker has neither special privileges nor direct access to other processes' data. All the mechanisms for reading and/or measuring time is under the abstraction of a software layer, such as an operating system. The attacker can neither bypass these mechanisms nor tamper with the data collected by them.*

Proposition 1. *Measuring the execution times of short-living operations requires quick consecutive time readings, resulting in a side channel per se, which can be monitored and used for detecting ongoing attacks.*

Note that Proposition 1 is generic in the sense that regardless of the shared resources utilized in the attacks or the operations timed, as long as the attacks rely on quick consecutive time readings, it causes malicious processes exhibit a certain characteristic behavior, which can be used to distinguish them from benign processes. Furthermore, introducing noise into the suspicious time measurements can prevent the attacks or make it difficult for the attackers to correlate the observed behavior with the private information under attack. This, therefore, removes the need for developing specialized detection, isolation, and prevention mechanisms for different types of timing attacks, which requires the manual analysis of the shared resources leveraged, the operations used, and the specific mechanisms employed by the attacks. By the same token, Proposition 1 can also be used against zero-day timing attacks.

Proposition 2. *By Remark 2, there exists a detection methodology since the side-channel, which is unintentionally created by the timing attacker, cannot be eliminated.*

This proposition is, indeed, strongly supported by our previous works [12–14], where we developed specialized approaches for detecting the Meltdown and cache-based timing attacks, demonstrating that the side-channel attacks, in the process of leveraging the information leaked by victim systems, unintentionally leak information by themselves, which can be used for detecting, isolating, and preventing them.

Algorithm 1: Detector$^+$

```
1  Input:
2    pid: PID of the process requesting to a time reading
3
4  curr_time ← readtime
5  pid.read_cnt++
6  if curr_time - pid.last_reading ≤delta_threshold then
7      pid.suspicious_reads++
8      if pid.read_cnt == window_size then
9          score ← pid.suspicious_reads/pid.read_cnt
10         if score ≥ warning_threshold then
11             emit a warning
12             pid.warning_cnt++
13             if pid.warning_cnt ≥ alarm_threshold then
14                 raise an alarm
15                 pid.warning_cnt ← 0
16             end
17         end
18         else
19             pid.read_cnt ← 0
20             pid.suspicious_reads ← 0
21             pid.warning_cnt ← 0
22         end
23     end
24     introduce noise
25     curr_time ← readtime
26 end
27 pid.last_reading ← curr_time
28 return curr_time
```

4. Detector$^+$

Detector$^+$ monitors the time readings made by processes on a per process basis with the goal of identifying consecutive readings that are suspiciously too close to each other.

4.1. Approach

Algorithm 1 presents the method employed by Detector$^+$. Note that this algorithm is carried out every time a process attempts to read the time. Furthermore, as there may be different ways for the processes to read the time depending on the underlying hardware and software platforms, we opt to provide the algorithm in a platform agnostic manner.

Every time a process requests a time reading, the time of the request is obtained (line 4) and compared to the last time the process requested a time reading (line 6). If the time difference between these two consecutive readings, which is, from now on, referred to as time delta (in short, *delta*), is lower than or equal to a predetermined threshold value, called *delta threshold* (*delta_threshold* in Algorithm 1), then the time delta (thus the pair of readings) is marked as suspicious (line 7).

The delta threshold parameter, being a hyper-parameter of Detector$^+$, needs to be set, such that the false positive rate as well as the false negative rate is minimized as much as possible. Note that any approach, which determines the delta threshold, such that the false positive rate is minimized, requires the presence of benign processes only for analysis. Whereas, the approaches based on minimizing the false negative rate, assume that the attacks are known a priori, such that time deltas can be collected from the malicious processes in a controlled manner for analysis. Therefore, in the presence of both the benign and malicious processes, the delta threshold can be chosen in a way that balances the false positive and false negative rates according to the needs. For this work, however, we followed the former approach by using a well known benchmarking suite of benign processes [19] to determine the delta threshold, without requiring any prior knowledge of the attacks. More specifically, we picked a threshold value, which results in a false positive rate of smaller than 0.01% (Section 5.2).

Once a suspicious time delta is detected, to thwart a possible ongoing attack, Detector$^+$ introduces noise into the time measurement (line 24), such that malicious process is prevented from extracting reliable information by analyzing the differences between the time measurements. Note that determining the best approach for introducing the noise is out of the scope of this paper. Consequently, we opted to use a simple, yet quite effective approach. In particular, we introduce a random amount of idle clock cycles (between 512 and 4352 clock cycles) by executing a random number of NOP (no-operations) instructions, causing slightly delayed time readings in the presence of suspicious time deltas. The results of our experiments strongly suggest that this approach can prevent the attacks from being successful or significantly reduce their success rates (Section 5.4).

Detector$^+$, not only detects and prevents the attacks, but also pinpoints the malicious processes carrying out the attacks. This is important as once the malicious processes are pinpointed, appropriate countermeasures can be taken to prevent the ongoing attacks or to reduce their harmful consequences.

To determine the malicious processes, we use a non-overlapping sliding window based approach, which can be configured with the help of 3 hyperparameters: *window_size*, *warning_threshold*, and *alarm_threshold*. The *window_size* parameter defines the size of the windows to be used for analysis, i.e., the number of consecutive time deltas in a window. Note that Detector$^+$ forms and analyzes the windows on a per process basis. If the ratio of the number of suspicious time deltas in a window exceeds a predetermined value (i.e., *warning_threshold*), a *warning* is emitted about the offending process (lines 10-12). And, if the warnings persist for a number of consecutive windows indicated by *alarm_threshold*, then an *alarm* is raised and the offending process is marked as suspicious (lines 13-15).

In the presence of an alarm, various countermeasures can be taken against the suspicious processes, such as terminating them, migrating them to different machines, or sandboxing them for further analysis. Such remediation strategies are, however, beyond the scope of this work.

4.2. Implementation

We have implemented Detector$^+$ in a Linux distribution, namely CentOS 7 with kernel v.3.10.0-957.5.1.el7.x86_64 and

glibc v.2.19. In Linux (as is the case with the other modern operating systems), there are two ways of reading the time using the services provided by the operating system: by making a *system call* (in short, *syscall*) and by making a *virtual system call* (in short, *vsyscall*). Virtual system calls are an alternative mechanism provided by an operating system for a small number of frequently used system calls (such as the timing-related calls) to reduce the runtime overheads by avoiding context switches.

The operating system implements the vsyscalls by mapping a fixed-size (1024-byte) *virtual dynamic shared object* (*vdso*) into the address space of each process [24]. When a process requests a timing-related service of the operating system, the request is captured by the glibc library, which implements the core functionalities for the user-level processes. If vdso is enabled and a vsyscall is available for the requested service, glibc reroutes the request to the respective vsyscall, avoiding the context switch. Otherwise, the request is rerouted to a regular syscall, causing a context switch. Note that vdso can be enabled/disabled at will during boot time. And, regardless of whether the vdso is on or off, the user processes remain oblivious of the underlying dynamics.

Detector$^+$, therefore, instruments all the timing-related syscalls and vsyscalls with the probe given in Algorithm 1. For this study, we have modified the operating system kernel for the former and the wrappers provided by the glibc library for the latter.

Besides the syscalls and vsyscalls, the only mechanism (to the best of our knowledge) that one can employ to read the time, is to use the native `rdtsc` instruction (or its variations), which collectively will be referred to as *rdtsc* in the remainder of the paper.

One issue with rdtsc is that a process can use it to surreptitiously read the time without letting the operating system know. As this paper strongly suggests that being able to measure the time is an integral part of the timing attacks. For improved security, we are, therefore, a strong advocate of allowing accesses to the timing-related services only through privileged software/hardware entities, so that suspicious time measurements can be monitored and appropriate countermeasures can be taken. Indeed, the security threat exhibited by rdtsc (in particular, whether it should be banned from the user space or not) has been a topic of debate for a while [6, 25, 26]. It is exactly for this reason that operating systems (with the help of the CPUs) provide facilities, which make rdtsc available only in ring 0, such that non-ring 0 accesses result in runtime failures. For example, in Linux, this is achieved by using the `prctl(PR_SET_TSC,...)` instruction [26, 27].

Note that, for Detector$^+$, it is not about disabling the user-level accesses to rdtsc, but about getting this instruction wrapped up by a privileged entity, so that all the accesses can be monitored and controlled. One way to achieve this could be to move the rdtsc accesses to a different protection ring, which has also been a topic of a debate [26]. For this study, we, therefore, instrumented all the binaries and source codes, such that Detector$^+$ is notified about every access to rdtsc by running the logic in Algorithm 1.

5. Experiments

To evaluate Detector$^+$, we have conducted a series of experiments. These experiments were specifically designed to address the following research questions: 1) Do timing attacks exhibit distinguishing timing behaviors? (Section 5.2); 2) Can the attackers be pinpointed? (Section 5.3); 3) Can the attacks be prevented? (Section 5.4); 4) Can the runtime overhead be kept at an acceptable level? (Section 5.5); and 5) Can the attack variations be detected? (Section 5.6).

5.1. Setup

Attack types. In the experiments, we have used 5 different timing attacks, namely Meltdown [7], Evict+Reload[8], Flush+Flush[9], Flush+Reload[10], and Prime+Probe[11] (Section 2). We have chosen these attacks since they are well-known representatives of all the existing timing attacks and they have been used in many related works for evaluation purposes [7, 12, 13, 22, 28–34].

Attack variations. We have also evaluated the proposed approach on two different attack variations (Section 5.6), both of which mimic some of the strategies that an attacker can use to become stealthier [13]. In one set of variations, each type of timing attack was carried out by multiple processes running concurrently (Section 5.6). In the other set of variations, multiple types of different timing attacks were carried out concurrently.

Benign processes. As the suite of benign processes, which we needed to evaluate whether the timing behaviors of the attacks differ from those of the benign processes as well as to measure the runtime overhead of the proposed approach, we used the Phoronix benchmarking suite [19]. We chose this suite because it represents a wide spectrum of scenarios, which are commonplace in today's production environments, including cryptographic and numerical computations, audio and video encoding, various CPU-, memory-, network- and disk-bound computations, and database operations. Furthermore, the Phoronix benchmarks have also been used in related works to evaluate preventive countermeasures against timing attacks [13, 34].

Table 1 presents information about the Phoronix benchmarks that we were able to run on our computing platforms. The first five columns in this table, depict the name, ID, and the version of the benchmark as well as the the number of different applications and sub-benchmarks involved, respectively. The remaining columns of this table will be discussed later in Sections 5.2 and 5.5.

Operational framework. All the experiments were carried out on an E5630 Intel Xeon system equipped with 32 GB of RAM, 32 KB of L1, 256 KB of L2 and 12288 KB of L3 cache memory running CentOS v7 operating system with kernel v3.10.0-957.5.1.el7.x86_64.

5.2. Study 1: Do timing attacks exhibit distinguishing timing behaviors?

Our first research question was whether the timing behaviors of the attacks differ from those of the benign processes. If

benchmark	ID	version	application count	sub benchmarks	time delta count	false positives (%)	runtime overhead (%)
Audio Encoding	1	1.0.1	2	2	1.07 e04	0	0.02
Cryptography	2	1.1.0	5	11	3.58 e04	0	1.55
Compression	3	1.0.1	5	5	6.94 e04	0	1.43
Compilation	4	1.2.1	6	6	1.86 e05	0	1.09
Video Encoding	5	1.1.0	3	3	4.10 e05	0	0.86
Molecular Biology	6	1.0.2	1	1	1.26 e06	0	0.004
Memory	7	1.1.0	5	12	2.57 e06	0	2.00
Workstation	8	1.1.0	9	11	2.71 e06	0	1.73
Multicore	9	1.2.0	16	17	3.36 e06	0	0.72
Gaming	10	1.0.1	2	5	1.65 e07	0	0.73
Machine Learning	11	1.2.0	2	2	2.07 e07	0	1.37
Network	12	1.1.0	2	6	2.15 e07	0	6.75
Database	13	1.1.0	2	5	2.33 e07	0	≈ 0
Scientific Computing	14	1.0.0	2	9	3.09 e07	0	1.42
Desktop Graphics	15	1.2.0	2	5	4.03 e07	0	0.29
Disk	16	1.3.0	5	17	7.75 e07	0	3.42
Server-Memory	17	1.1.3	30	56	1.60 e08	0	1.01
Kernel	18	1.1.0	13	25	1.76 e08	0	2.28
Server-CPU	19	1.0.0	24	36	2.28 e08	1.86 e-09	1.38
Server	20	1.2.1	8	25	2.65 e08	0	1.72
Overall	-	-	144	259	1.07 e09	≈ 0	1.56

Table 1: Phoronix benchmarks used in the study.

this is not the case, then Detector$^+$ will certainly fail to fulfill our claims. Note that, in this context, the *timing behavior* of a process indicates the "typical" durations, which are needed to be timed by the process.

To address this research question, we carried out a series of experiments to determine the delta thresholds for each timing mechanism, i.e., by using the system calls, virtual system calls, or the rdtsc instructions. We do this because different timing mechanisms can impose different amount of measurement noise. Note that, since Detector$^+$ is aware of the actual timing mechanism being used for each time measurement (as the instrumentation code specified in Algorithm 1 is inserted on a per timing mechanism basis), a different delta threshold can be used for each mechanism to determine the suspicious time deltas.

For this study, we opted to determine the delta thresholds based on the distributions of the time deltas obtained from the benign processes (Section 5.1), avoiding the need for knowing the attacks a priori. To this end, we had to run the Phoronix benchmarks, which took about 2 weeks of computation time, resulting in more than one billion (1.07 e09) time deltas (Table 1).

For this study, we needed to log all the deltas. Note that the aforementioned logging step is something, which is required only for carrying out the study, so that we can report our findings by performing a detailed analysis in an offline manner. That is, Detector$^+$ does not log the time deltas for later processing as it simply computes the time deltas and determines whether they are suspicious or not by making simple comparisons at runtime (Algorithm 1), requiring a constant amount of memory per process to operate.

It turned out that, for the computing platforms we used in the study, the most reliable and scalable way of collecting the data, was to carry the logging task in the OS kernel. We, therefore, disabled vdso and forwarded all the timing calls to the system calls. This, indeed, enabled us to compute a lower bound on the detection accuracy of the malicious timing deltas since, among all the different timing mechanisms, using the system calls was the one that introduced the highest level of noise in the measurements due to the context switches required. And, the more the measurement noise, the more the relative differences tend to diminish between the malicious and benign time measurements, thus making it more difficult to differentiate malicious deltas from the benign ones.

Figure 1 visualizes the time deltas we obtained from the benign processes (in clock cycles). Each box in the figure represents the distribution of time deltas obtained from a particular benchmark. The top and bottom bars of a box depict the first and third quartile of the distribution, respectively, i.e., half of the time deltas fall into the box. Furthermore, the median and mean values are represented by the middle bars and the triangle symbols associated with the boxes. The information about the actual benchmarks used in this study as well as the total number of time deltas observed in each benchmark can be found in Table 1.

Using the maximum false positive rate of 0.01% (see Section 4 for more discussion), we determined 4635 clock cycles as the delta threshold to be used (depicted by the dashed line in Figure 1). With this threshold, among all of the 1.07 e09 time deltas obtained from benign processes, only two of them (both of which occurred in benchmark ID=19) were, indeed, incor-

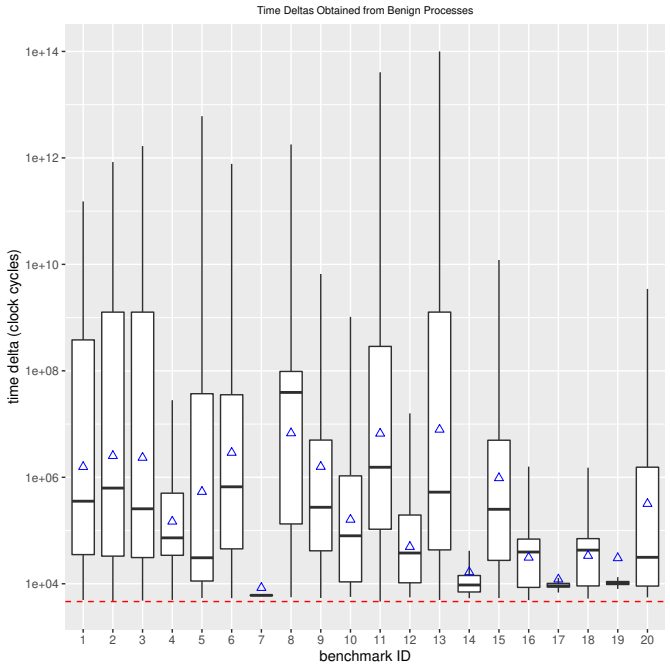

Figure 1: Distributions of the time deltas obtained from benign processes. The triangles depict the average time deltas and the line indicates the delta threshold of 4635.

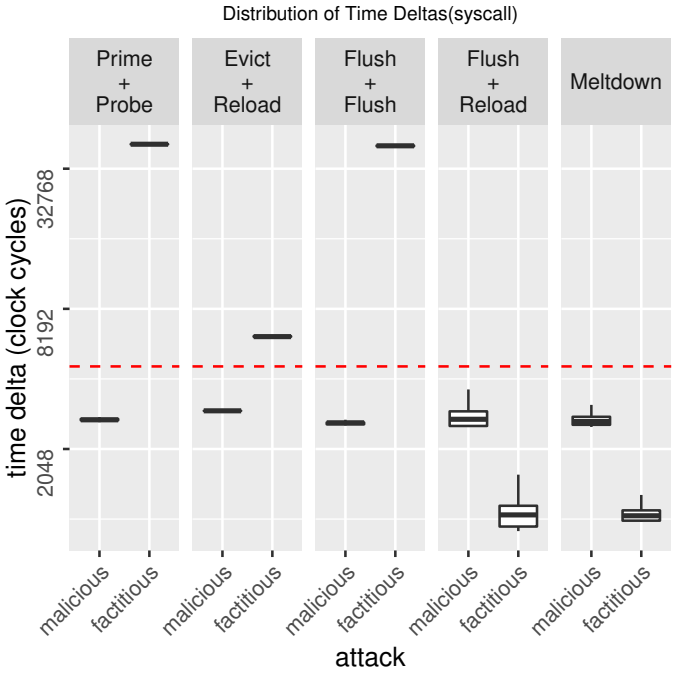

Figure 2: The distributions of the malicious and factitious time deltas obtained from different attacks when the system calls are used as the timing mechanism. The dashed line indicates the delta threshold of 4635 clock cycles.

Algorithm 2: Pseudo-Attack Loop

```
1 attack loop
2 |   ...
3 |   start_time ← readtime
4 |   // malicious intent goes here
5 |   end_time ← readtime
6 |   ...
7 end
```

rectly marked as suspicious, resulting in an actual false positive rate of ≈ 0% (Table 1).

After determining the delta threshold using the benign processes, we used it on the time deltas obtained from the timing attacks to determine the false negative rate. To this end, we ran each attack about 30 seconds, which turns out to be sufficient amount of time for these attacks to succeed, and repeated the experiments 3 times. In total, we have collected about 188 million (18.84 e07) time deltas from the attacks.

We first observed that the time deltas obtained from malicious processes tend to come from multiple (typically 2) distributions. An in-depth analysis revealed that this is because the malicious processes typically make their measurements in a loop, such that after every time measurement, some computations are performed, for example, to store the measurements and/or to analyze them.

Algorithm 2 illustrates this phenomenon. At every iteration of the attack loop, a time measurement is made by using two time readings; one at line 3 and the other at line 5. Assuming, for example, that the loop iterates two times, four time readings would be intercepted by Detector$^+$: t_1, t_2, t_3, and t_4, where t_1 and t_2; and t_3 and t_4 correspond to the time readings at lines 3 and 5 in the first and second iteration of the loop, respectively. Thus, three time deltas are computed: $\Delta_1 = t_2 - t_1$, $\Delta_2 = t_3 - t_2$, and $\Delta_3 = t_4 - t_3$. Note, however, that only two of these deltas correspond to the actual time measurements that the attacker carries out: Δ_1 and Δ_3, each of which represents a measurement from line 3 to line 5. The other time delta, namely Δ_2, is an artifact of the monitoring process as the time deltas are computed without the knowledge of the contexts the processes may be in. More specifically, Δ_2 represents a time measurement from line 5 to line 3, which does not correspond to an actual time measurement made by the attacker.

Consequently, the time deltas obtained from such an attack loop would tend to come from two distributions; one representing the malicious measurements from line 3 to line 5 and the other representing the factitious measurements from line 5 to line 3. In the remainder of the paper, the time deltas that correspond to the actual time measurements that the attacker makes with malicious intentions, such as Δ_1 and Δ_3, are referred to as *malicious time deltas* (*malicious deltas*, for short). And, all the other time deltas obtained from an attack will be referred to as *factitious time deltas* (*factitious deltas*, for short).

Figure 2 demonstrates an advent of this phenomenon in our experiments by visualizing the distributions of the malicious and factitious time deltas obtained from different attacks. For a given attack, the two distributions were significantly different from each other. Note that, for this work, we are primarily concerned with the malicious deltas as the factitious deltas are considered to be benign.

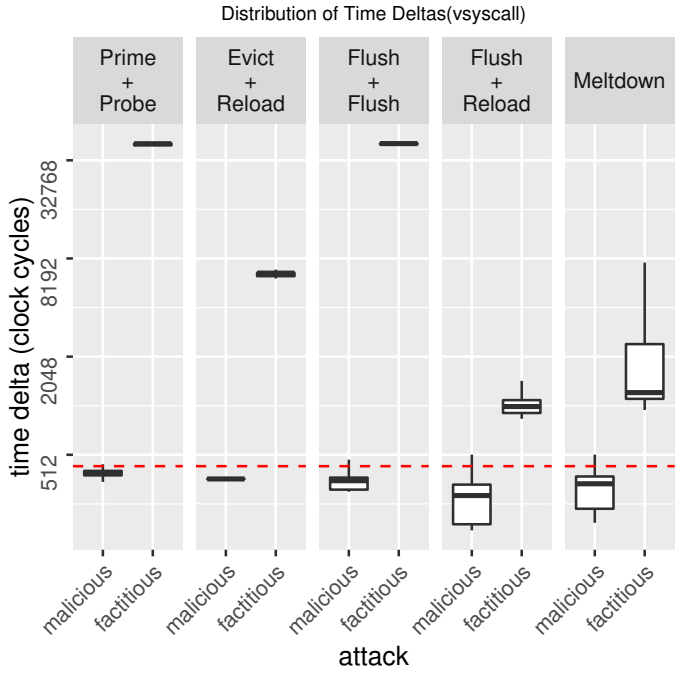

Figure 3: The distributions of the malicious and factitious time deltas obtained from different attacks when the virtual system calls are used as the timing mechanism. The dashed line indicates the delta threshold of 436 clock cycles.

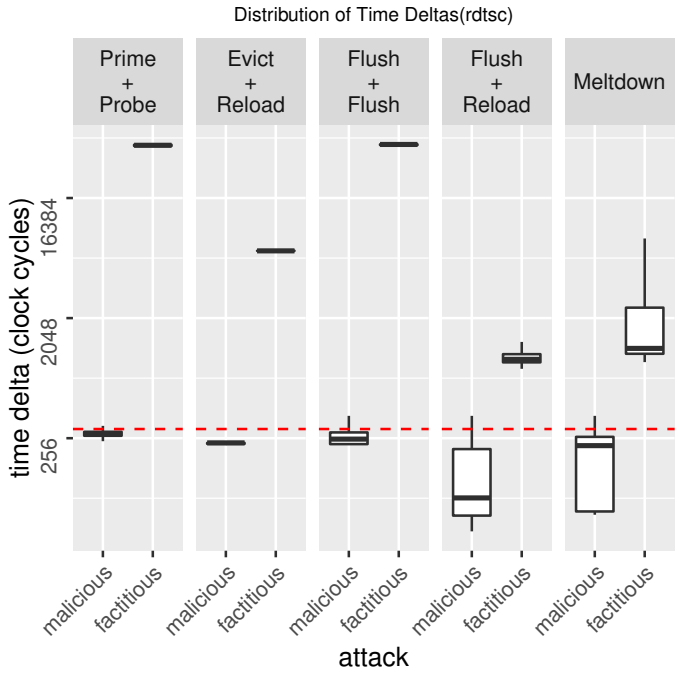

Figure 4: The distributions of the malicious and factitious time deltas obtained from different attacks when the rdtsc instructions are used as the timing mechanism. The dashed line indicates the delta threshold of 300 clock cycles.

Applying the selected delta threshold of 4635 to the malicious deltas obtained from the attacks (depicted by the dashed line in Figure 2), which roughly aligned with the 99.99th quantile, we obtained a false negative rate of 0.0009%. That is, only 0.0009% of the malicious deltas were above the threshold.

All told, we have obtained an almost perfect accuracy in detecting malicious deltas (i.e., an accuracy of $\approx$ 100% with a false positive rate of $\approx$ 0% and a false negative rate of 0.0009%), strongly supporting our hypothesis that timing attacks exhibit distinguishable timing behaviors.

Repeating the experiments with the remaining types of timing mechanisms (e.g., vsyscalls and rdtsc) and focusing on the 99.99th quantile, resulted in the delta thresholds of 436 and 300 for vsyscall and rdtsc, respectively. The distributions of the malicious and factitious deltas obtained with these timing mechanisms can further be found in Figures 3 and 4. Expectedly, as the noise introduced by measurement mechanism decreased, the delta threshold decreased. Note that to carry out this study, we made the attacks to use the timing mechanism of choice (i.e., syscall, vsyscall, or rdtsc). We did this because our ultimate goal was to reason about the delta thresholds, had the attacks used different timing mechanisms. Therefore, whether the attacks succeeded with the alternative timing mechanisms was irrelevant for us in this regard.

5.3. Study 2: Can the attackers be pinpointed?

After determining the delta thresholds, we focused on our second research question: Can suspicious time deltas be used to pinpoint the malicious processes?

Note that the sooner the malicious processes are determined, the better it is in terms of carrying out the appropriate countermeasures on time to prevent the ongoing attacks or to reduce their harmful consequences. To this end, we aimed to reduce the values of our hyper-parameters, i.e., *window_size*, *warning_threshold*, and *alarm_threshold*, as much as possible (Section 4).

We observed that, with *window_size* = 4, *warning_threshold* = 0.5, and *alarm_threshold* = 1, which will be referred to as the *default configuration* in the remainder of the paper, the first warning as well as the first alarm for each malicious process was emitted after the process made a total of 5 time readings (i.e., 4 time deltas). That is, all the malicious processes were correctly pinpointed after the first window of time deltas. And, this was done without emitting neither a false warning nor a false alarm for any of the benign processes.

A related configuration, which produced slightly earlier warnings for the malicious processes at the expense of two false warnings, but no false alarms for the benign processes, was *window_size* = 2, *warning_threshold* = 0.5, and *alarm_threshold* = 2. With these hyper-parameters, for each malicious process, the first warning was emitted after the process made 3 time readings (i.e., after the first window of 2 time deltas) and the first alarm was emitted after a total of 5 time readings (i.e., after the second window of 2 time deltas).

We have also observed that some care must be taken when setting the warning threshold above 0.5. This is because (as discussed in Section 5.2) the time deltas obtained from malicious

processes tend to come from at least two different distributions, one of which represents the malicious measurements. Therefore, setting the warning threshold above 0.5 may prevent the detection of the malicious processes.

On the other hand, setting the warning threshold lower than 0.5 may increase the number of false warnings and false alarms. This issue can, however, be addressed by increasing the window size and/or the alarm threshold. For example, using a warning threshold of lower than 0.01 (i.e., less than one suspicious delta per 100 deltas) still perfectly pinpoints all the malicious processes in our experiments without having any false warnings or alarms, when the window size is set above 100. Or, setting the warning threshold to 0.01 with a window size of 100 requires an alarm threshold of at least 2 to avoid any false alarms.

5.4. Study 3: Can the attacks be prevented?

Our next question was then whether the attacks can be prevented. To this end, in the presence of a suspicious time delta, we have introduced noise in the respective measurement by executing a random number of NOP instructions (each of which takes one clock cycle to execute), hampering the usability of the measurements made by the attacker (Section 4).

In particular, we experimented with the following noise levels: 512, 768, 1024, 1280, and 4352. For a given noise level n, we determined the actual noise to be introduced, i.e., the number of NOPs to be executed, by randomly picking a number between $[256, n]$. Note that the lower bound in the aforementioned range ensures that at least a minimum number of NOPs are guaranteed to be executed.

For each attack, starting from the lowest noise level, we ran the attack 20 times and computed the *prevention effectiveness* obtained by the selected noise level as the percentage of the experiments, in which the attack failed to operate. To this end, we used the oracles that came with the source code distributions of the attacks (Section 2). More specifically, each attack had a mechanism, indicating if the attack was successfully carried out. We kept on increasing the noise level until we had a perfect prevention effectiveness, i.e., until all the attacks were prevented under the given noise level.

Figure 5 presents the results we obtained. We first observed that the proposed approach prevented all the attacks with perfect effectiveness (i.e., with 100% prevention effectiveness). Although the noise level required varied from one attack to another, using a sufficiently large noise level (in our case, 4352) prevented all the attacks.

Note that an important feature of Detector$^+$ is that since the noise is introduced for each suspicious delta, Detector$^+$ does not wait until a warning or an alarm is emitted for a process before acting on it. That is, attacks can still be prevented even if no warnings or no alarms are raised.

5.5. Study 4: Can the runtime overhead be kept at an acceptable level?

After observing that Detector$^+$ detected, isolated, and prevented all the attacks, we evaluated its runtime overhead. Since the security-related approaches, such as Detector$^+$, target the

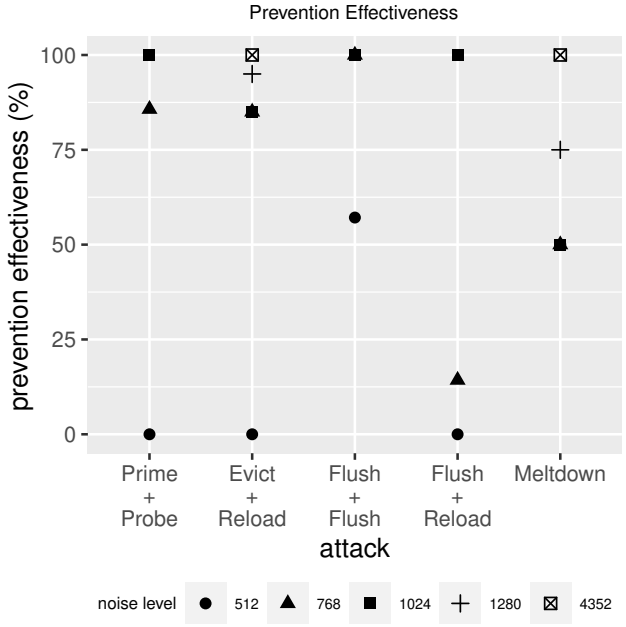

Figure 5: Prevention effectiveness obtained under different noise levels.

fielded instances of software systems, excessive runtime overheads are generally not acceptable.

To carry out the study, we have opted to measure the runtime overhead of the proposed approach by enabling vdso and forwarding all the timing calls to vsyscalls, which also allowed us to mimic the scenarios where rdtsc is wrapped with a fast mechanism (Section 4.2). This, therefore, enabled us to compute an upper bound on the runtime overhead of the proposed approach since the vsyscalls introduce significantly less overhead compared to the syscalls (Section 5.2), making the overhead introduced by Detector$^+$ more apparent.

We have then executed the Phoronix benchmarks between 9 to 15 times (depending on the amount of time required for each benchmark) on both the original operating system and the operating system instrumented with Detector$^+$. This took us to run a total of 259 sub-benchmarks involving 144 applications for about 5 months nonstop.

Each benchmark was designed to report its own performance measurements. We used these measurements to compute the runtime overheads as follows: $((P' - P)/P) * 100$, where P and P' are the performance measurements obtained from the original and the instrumented system, respectively. The lower the overhead, the better the proposed approach is.

Table 1 presents the overheads we obtained. We observed that the overall runtime overhead of the proposed approach was 1.56%, on average. Indeed, for 52% (137 out of 259) of the sub-benchmarks, Detector$^+$ introduced virtually no overheads. And, for most of the remaining sub-benchmarks the overhead was close to the average overhead of 1.56% (Table 1). One exception was the network benchmarks (benchmark ID=12), where we observed an average overhead of 6.75%. Interestingly enough, no suspicious deltas were detected for this benchmark. That is, no action, except for comparing the observed

time deltas with the threshold value, was needed. Furthermore, in terms of the duration of this benchmark as well as the total number of time deltas observed in it, we could not identified any singularities either. That is, there were other benchmarks with similar durations and similar number of time deltas, but resulting in virtually no overheads. An in-depth analysis then revealed that the performance of this benchmark is greatly affected by the network traffic present in the underlying platforms. We indeed observed that even the performance measurements obtained from the different repetitions of the same sub-benchmarks running on the same platform varied between 2.2% and 5.5%, possibly explaining the singularity in the overheads.

5.6. Study 5: Can the attack variations be detected?

We have then evaluated Detector$^+$ under different attack variations. Each attack variation mimicked a mechanism that can be used by an attacker to stay stealthier.

Carrying out an attack by using multiple processes. We first focused on the variations, in which the same attack is carried out concurrently by multiple processes. Note that this can potentially reduce the suspiciousness of individual processes by distributing the malicious activities over multiple processes, each of which can, for example, target a different part of the secret information.

To carry out the study, we executed each attack by using $p = 2$, 5, and 10 concurrent processes and monitored the time deltas using the default configuration of Detector$^+$ (i.e., *window_size* = 4, *warning_threshold* = 0.5, and *alarm_threshold* = 1). We observed that, except for 2 processes, all the malicious processes were pinpointed after the first window; the first warning and the first alarm for these processes were emitted after they had made 5 time readings (i.e., after 4 time deltas). And, the two aforementioned malicious processes were pinpointed after the second window; the first alarm as well as the first warning for these processes were emitted after they had made 9 time readings. One of these processes was a Flush+Flush process when $p = 2$ and the other was a Prime+Probe process when $p = 5$. It turned out this happened because the first few malicious time deltas obtained from these processes were unexpectedly high. We believe that this happened due to the noise introduced by spawning multiple processes at around the same times. Note that such noise also makes it difficult (if not impossible) for the attacker to extract information from the measurements (see Section 7 for more information).

Carrying out different attacks concurrently. Last but not least, we have evaluated Detector$^+$ on attack scenarios, in which different types of attacks were carried out concurrently. Note that this type of variation is different than the previous type of variation because in the former a single type of attack is carried out at a time by using multiple concurrent processes.

To carry out the study, we executed all of the 5 attacks concurrently and used the default Detector$^+$ configuration (i.e., *window_size* = 4, *warning_threshold* = 0.5, and *alarm_threshold* = 1) for analysis. Furthermore, we have repeated the experiments 6 times. All of the malicious processes used in these experiments were pinpointed after the first window, i.e., after they have made 5 time readings (i.e., 4 time deltas).

6. Threats to validity

One external threat concerns the representativeness of the timing attacks used in the study, namely Meltdown [7], Evict+Reload [8], Flush+Flush [9], Flush+Reload [10], and Prime+Probe [11]. However, all of these attacks are well-known attacks and they have all been used in related works [7, 13, 22, 28, 31, 32]. Furthermore, as the base attacks, we have used the publicly available implementations of these attacks together with the facilities provided by these implementations to determine whether the attacks were successful or not. We have also evaluated Detector$^+$ on different variations of these attacks by carrying out the attacks using multiple processes and by carrying out multiple different types of attacks concurrently, mimicking some of the mechanisms that attackers may employ to stay stealthier.

Another threat concerns the representativeness of the benign processes used in the study. In the experiments, we have used the Phoronix benchmarks as the suite of benign processes [19]. Phoronix, indeed, provides a variety of software systems, which are commonly encountered in production environments, including network, database, and machine learning systems; cryptographic, scientific computing, audio/video processing, and graphics applications; and a wide spectrum of IO-/CPU-bounded applications for workstations and servers(Table 1). Phoronix has also been used in related works [13, 35–37].

7. Countermeasures against Detector$^+$

Detector$^+$, as is the case with other detection frameworks, can (and will) naturally become the subject of attack techniques that aim to find innovative ways of avoiding detection by Detector$^+$. In this section, we discuss some of the potential countermeasures that can be taken against Detector$^+$ as well as possible mitigation strategies that Detector$^+$ can employ against them.

Detector$^+$ operates by monitoring and analyzing how processes perform time measurements and their patterns thereof. An attacker may, therefore, attempt to surreptitiously measure the time and/or temper with the measurements made by Detector$^+$. Except rdtsc, this, however, requires elevated privileges (which is against the premises of the timing attacks) as the aforementioned time measurement mechanisms use either system calls or virtual system calls.

Regarding rdtsc, as the results of the studies we carried out in this paper strongly suggest that monitoring and analyzing timing behaviors can be used as a countermeasure against timing attacks, we are a strong advocate of allowing accesses to rdtsc (and, for the same reason, to all timing related services as well as to hardware performance counters/events [38, 39]) only through privileged software/hardware entities, so that suspicious time measurements can be monitored and appropriate

countermeasures can be taken. Further discussion on this can be found in Section 4.2.

An attacker may also increase the time gap between the consecutive pairs of time readings in an attempt to stay stealthy. If the resulting time deltas stay above the delta threshold, then they will not be marked as suspicious. One way this could be done is to measure the execution time of a series of operations, rather than a single operation. For example, in the cache-based timing attacks, rather than measuring the time it takes to access a single memory location to determine whether the respected data is in the cache, one could measure the time required for accessing multiple memory locations. This, however, makes the attacks more complicated to implement as more involved analyses are required to factor out the ambiguity in the measurements.

Another way could be to intentionally introduce deterministic amount of noise into the measurements, which can then be factored out by the attacker after the measurements. For example, during a time measurement, in addition to accessing a memory location of interest, the attacker can carry out a fixed number of NOP instructions to stay above the delta threshold. However, the more *intentional* noise introduced by the attacker, the more *unintentional* noise is generated by the underlying platform (e.g., operating system), which makes the analysis complicated (if not impossible at all). This is because there is no guarantee as to how these benign instructions will be executed. For example, during the execution of these instructions, thus in the middle of a time measurement, the operating system may take the control of the CPU from the malicious process and give it to another process, which would inadvertently introduce a great deal of system noise.

To evaluate this conjecture, we carried out a study where we introduced a fixed number of NOPs into the time measurements made by the Meltdown attack. The noise was then subtracted from the actual time readings before they are analyzed by Meltdown. We experimented with different levels of intentional noise. For each noise level (i.e., the number of NOPs introduced into each time measurement), we used the reliability oracle that comes with the source code distribution of Meltdown to measure the accuracy of the attack. The accuracy in this context is simply computed by the percentage of the attempts, in which the target memory location is read successfully. Thus, the higher the accuracy, the better the attack is.

Figure 6 presents the results we obtained. As the level of intentional noise is increased, which the attacker needs in order to stay above the delta threshold, the accuracy of the attack is diminished and eventually reached 0% where not even a single byte of information was exfiltrated, supporting our hypothesis.

Note that to account for this phenomenon the attacker would need more measurements (which increases the amount of time required by the attack) and/or more sophisticated analyses, thus reducing the chance of success. Nevertheless, the hyperparameters offered by Detector$^+$ can be used as a defense mechanism for these countermeasures. For example, one can increase the delta threshold, which makes the longer time deltas also look suspicious. This, however, may reduce the accuracy in isolating the malicious processes. To cope with this, the window

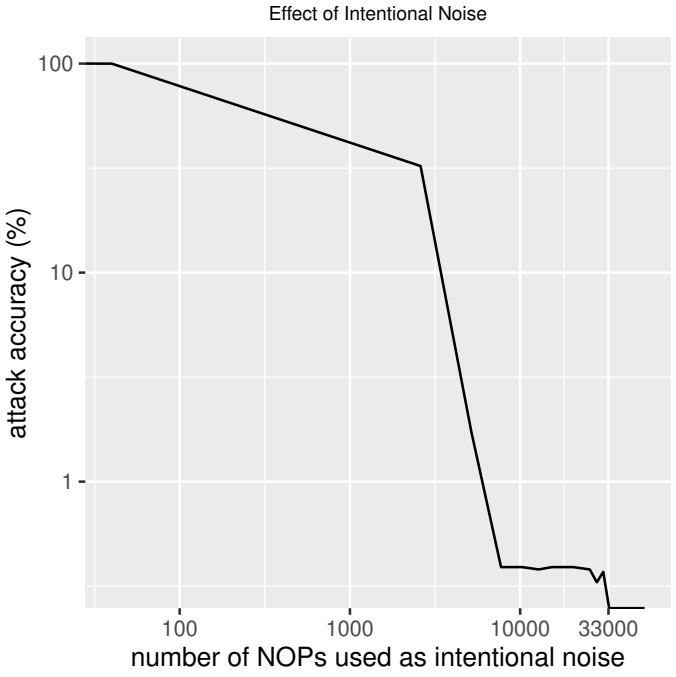

Figure 6: The effect of the intentional noise introduced into the time measurements on the accuracy of the Meltdown attack.

size as well as the warning and alarm thresholds can be increased. For example, in the presence of intentionally introduced noise in the Meltdown attack (Figure 6), we increased the delta threshold to 7555, such that the accuracy of the attack is kept under 1%. With this new threshold, we observed that having window size = 30000, warning threshold= 0.5, and alarm threshold = 50, pinpointed all the malicious processes without emitting any false alarms for the benign processes. Note that although increasing the values of these hyper-parameters may increase the detection times of the malicious processes, since Detector$^+$ does not wait until a process is marked as malicious before acting on it (i.e., in the presence of suspicious time deltas, the time measurements are thwarted regardless of whether the respective processes are marked as malicious or not), the chance of success for the attacks will still be reduced (if not prevented at all). After all, a complementary way of reducing false alarms is to explicitly mark the trusted processes (such as, the system processes), so that they can be excluded from the analysis, which, in turn, can also help reduce the values of the hyper-parameters.

In summary, Detector$^+$ utilizes a strong feature set, that proves to be highly generic and extremely effective against detecting and eliminating a wide range of cache-based timing attacks by itself and/or in conjunction with other detection techniques.

8. Related work

Information exfiltration from cryptographic algorithms employing timing-channels had been extensively studied [4–6, 40, 41] in the past. These attacks often leverage a timing-channel

being established as a side effect to cache-contentions. Typically, the attacker constantly monitors a set of cache lines to determine if they have been accessed by another process; either through continual eviction of a cache line or continuously filling a small portion of cache accessible to attacker. In both of the mentioned schemes, statistically notable change in the access latency enables discovery of accessed cache lines by a victim process.

Timing attacks have been demonstrated to circumvent sandboxes and even kernel space hardening (such as ASLR) [42–46]. One mitigation strategy extensively highlighted in the literature is to ensure that critical parts of the systems, which process secret information, should remain agnostic of observable sequences of cache accesses [5]. This, for example, can prevent the influence of plain and cipher text parameters in cryptographic applications from being deterministically observed. Bernstein [4] suggested that, one way of mitigation must involve constant time implementation of the cipher. However, this remain a very difficult problem [4].

Cock et al. [47] claim that although storage based side-channels can be discovered and prevented through formal analysis, as is the case with seL4 [48] micro-kernel, nonetheless such an approach does not extend to timing-channels. Moreover, they [47] advocate, although in-theory a constant-time implementation approach can guarantee absence of a timing-channel; in reality, it remains prone to the CPU architecture type, as was the case with an OpenSSL vulnerability which had been rectified on x86 by such a fix, while remained ineffective on ARM. Furthermore, Coppens et al. [49] uncovered that the timing behavior of cryptographic software is directly dependent upon its utilization of variable-latency instructions in the code. To remediate, a compiler based solution is presented, which removes control- and data-flow dependence on the secret key. Although the presented approach had been effective for addressing the main problem nonetheless, it incurred significant performance overhead. Along the similar lines, Rodrigues et al. [50] developed a more effective compile-time tool, discovering timing-channel vulnerabilities manifested in the implicit control flow of the actual implementation of cryptographic code. Their presented approach successfully revealed a known vulnerability in OpenSSL (v1.0.1e), use casing the effectiveness of their tool. Approaches for developing security-aware and attack-resistant cache architectures have also been studied in literature [51–57].

Mitigating timing-channels for adversaries by coarsification of high resolution clock sources has also been explored in the literature. Hu et al. [58] present timestamp fuzzing technique for VAX systems in classical literature. They present that reducing the resolution of high resolution clock sources reduces the bandwidth of a timing-channel. To this end, they present a set of techniques which involve adding random amount of noise to all system wide clock sources, essentially limiting the resolution of timestamps. However, since such a technique would introduce noise to all time measurements, it would also deny the legitimate applications of high-resolution measurements. This would also inadvertently penalize the system throughput, as coordination of the hardware level operations depend upon high resolution timers. Martin et al. [59] present a similar technique, that selectively limits the resolution of the rdtsc instructions. Our work is different from this work that we are concerned with not only the hardware timekeeping instructions (such as, rdtsc), but also the software timekeeping mechanisms (such as syscalls and vsyscalls). Furthermore, the aforementioned work requires some hardware modifications (as it proposed a new machine instruction) and does not automatically determine the malicious processes. It simply changes the granularity of the rdtsc instructions for some pre-determined (i.e., given) processes. Our work, however, is implemented at the software level and pinpoints the malicious processes, so that they cannot further harm the system by, for example, intentionally making time measurements to hurt the performance of the system.

Language based predictive mitigation for timing-channels remain another topic of research, quantizing bounds over security of information flow. Zhang et al. [60] propose well-typed programs can only leak a bounded amount of information through a timing-channel. To this end, they propose a mitigation language employing predictive timing to ensure strict bounds on the execution time of programs. In contrast, Li et al. [61, 62] present a hardware focused approach by developing statically verifiable hardware description language to ensure information-flow security. Porter et al. [63] present an augmentative solution to retrofit an existing OS, based on decentralized information flow control, which remains promising in providing end-to-end security guarantees. In the aforementioned approach, the security policies need to be specified by labeling data prior to its processing in context-sensitive security methods. Nonetheless, such an approach remains partially effective for the prevention of timing-channels and can result in, considerable performance overhead. These aforementioned preventive approaches may lack generality to cater for wider spectrum of timing-channel attacks, yet strengthen the defensive posture of a system for their respective category of attacks, albeit at the cost of performance. Such approaches, however, may lack prevention against zero-day attacks. We, therefore, believe that for sensitive and security first application scenarios, the aforementioned approaches can complement Detector^{+} and provide stronger defense mechanisms.

Reactive, dynamic, and attack specific mitigative approaches have also been presented in literature. Kulah et al [12] present an anomaly based detection approach to prevent Prime+Probe, Flush+Reload, and Flush+Flush attacks. Their presented approach monitors contentions occurring in shared resources and associates them with processes, such that any suspicious level of contentions lead to the issue of a warning and subsequently a preventive action to stop an ongoing attack. Akyildiz et al [13] present an approach tailored to detect and prevent the Meltdown attack by monitoring segmentation faults occurring at memory addresses close to each other. In the presence of a segmentation fault, the aforementioned approach flushes the cache hierarchy as a reactive measure to prevent possible information leakage. The sequence of segmentation faults are then analyzed to pinpoint the malicious processes.

Nomani et al. [64] suggest leveraging information available from hardware performance counters to learn and predict up-

coming execution phases of programs. A program scheduler equipped with this information can adaptively schedule programs such that an on-going attack can either be thwarted to a reasonable extent. Zhang et al. [65] present an approach to mitigate Prime+Probe attack by relying on frequent intentional cache-cleanings which in-effect obfuscates usable information in the timing data to render it useless for an attacker. The aforementioned approaches lack a general mechanism to prevent all timing-channel attacks, as they're tailored to the specifics of the certain attacks. In contrast, Detector$^+$ is a generic approach, agnostic to the specifics used in the timing attacks and can detect an attack as long as the attack demonstrates fine-grained time reading behavior.

9. Concluding Remarks

In this work, we have presented a novel approach, called Detector$^+$, for detecting, isolating, and preventing timing-based side channel attacks at runtime.

The proposed approach is based on a simple observation that the way, timing attacks perform their time measurements and their patterns thereof differ from those of the benign processes. In particular, timing attacks need to measure the execution times of typically quite short-running operations. Detector$^+$, therefore, monitors time readings made by processes and mark consecutive pairs of readings that are close to each other in time as suspicious. In the presence of a suspicious time measurement, a random amount of noise is introduced in the measurement to prevent the attacker from extracting information by using the measurement. The sequence of suspicious measurements is then analyzed at runtime by using a sliding window-based approach to determine the malicious processes.

We evaluated the proposed approach by conducting a series of experiments. In these experiments, we used five well-known timing attacks together with a well-known suite of benign applications, representing the applications that are commonly encountered in production environments. To further evaluate the proposed approach in the presence of stealthier attacks, we have also tested Detector$^+$ with different variations of the timing attacks. In all the experiments, Detector$^+$ detected all the malicious time measurements with almost a perfect accuracy, prevented all the attacks, and correctly pinpointed all the malicious processes involved in the attacks without any false positives after they have made a few time measurements with an average runtime overhead of 1.56%.

We believe that this line of research is novel and interesting. Therefore, we continue working in this field. In particular, we are interested in understanding the fundamental mechanisms involved in side channel attacks, identifying the commonalities between these mechanisms, and developing low-overhead approaches to detect, isolate, and prevent not only the known attacks, but also the zero-day attacks at runtime.

Acknowledgment

This research was supported by Amazon Web Services, Inc. ("AWS") Research Gift for developing generic approaches for detecting, isolating, and preventing timing-based side channel attacks at runtime.

References

[1] S. Zander, G. Armitage, P. Branch, A survey of covert channels and countermeasures in computer network protocols, IEEE Communications Surveys & Tutorials 9 (3) (2007) 44–57.

[2] J. Szefer, Survey of microarchitectural side and covert channels, attacks, and defenses, Journal of Hardware and Systems Security 3 (3) (2019) 219–234.

[3] J. Betz, D. Westhoff, G. Müller, Survey on covert channels in virtual machines and cloud computing, Transactions on Emerging Telecommunications Technologies 28 (6) (2017) e3134.

[4] D. J. Bernstein, Cache-timing attacks on aes.

[5] D. A. Osvik, A. Shamir, E. Tromer, Cache attacks and countermeasures: the case of aes, in: Cryptographers' track at the RSA conference, Springer, 2006, pp. 1–20.

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

[7] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, M. Hamburg, Meltdown, arXiv preprint arXiv:1801.01207.

[8] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, S. Mangard, Armageddon: Cache attacks on mobile devices, in: 25th USENIX Security Symposium (USENIX Security 16), 2016, pp. 549–564.

[9] D. Gruss, C. Maurice, K. Wagner, S. Mangard, Flush+ flush: a fast and stealthy cache attack, in: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Springer, 2016, pp. 279–299.

[10] Y. Yarom, K. Falkner, Flush+ reload: A high resolution, low noise, l3 cache side-channel attack, in: 23rd USENIX Security Symposium (USENIX Security 14), 2014, pp. 719–732.

[11] F. Liu, Y. Yarom, Q. Ge, G. Heiser, R. B. Lee, Last-level cache side-channel attacks are practical, in: 2015 IEEE symposium on security and privacy, IEEE, 2015, pp. 605–622.

[12] Y. Kulah, B. Dincer, C. Yilmaz, E. Savas, Spydetector: An approach for detecting side-channel attacks at runtime, International Journal of Information Security 18 (4) (2019) 393–422.

[13] T. A. Akyildiz, C. B. Guzgeren, C. Yilmaz, E. Savas, Meltdowndetector: A runtime approach for detecting meltdown attacks, Future Generation Computer Systems 112 (2020) 136–147.

[14] M. Chiappetta, E. Savas, C. Yilmaz, Real time detection of cache-based side-channel attacks using hardware performance counters, Applied Soft Computing 49 (2016) 1162–1174.

[15] Y. Wang, A. Ferraiuolo, G. E. Suh, Timing channel protection for a shared memory controller, in: 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), IEEE, 2014, pp. 225–236.

[16] N. Wistoff, M. Schneider, F. K. Gürkaynak, L. Benini, G. Heiser, Prevention of microarchitectural covert channels on an open-source 64-bit risc-v core, arXiv preprint arXiv:2005.02193.

[17] Q. Ge, Y. Yarom, T. Chothia, G. Heiser, Time protection: the missing os abstraction, in: Proceedings of the Fourteenth EuroSys Conference 2019, 2019, pp. 1–17.

[18] Y. Wang, G. E. Suh, Efficient timing channel protection for on-chip networks, in: 2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip, IEEE, 2012, pp. 142–151.

[19] Phoronix test suite, `https://www.phoronix-test-suite.com/`, accessed: (2021-01-16) (2019).

[20] D. Wang, Z. Qian, N. Abu-Ghazaleh, S. V. Krishnamurthy, Papp: Prefetcher-aware prime and probe side-channel attack, in: Proceedings of the 56th Annual Design Automation Conference 2019, 2019, pp. 1–6.

[21] M. Mushtaq, A. Akram, M. K. Bhatti, R. N. B. Rais, V. Lapotre, G. Gogniat, Run-time detection of prime+ probe side-channel attack on aes encryption algorithm, in: 2018 Global Information Infrastructure and Networking Symposium (GIIS), IEEE, 2018, pp. 1–5.

[22] D. Gruss, R. Spreitzer, S. Mangard, Cache template attacks: Automating attacks on inclusive last-level caches, in: 24th USENIX Security Symposium (USENIX Security 15), 2015, pp. 897–912.

[23] Y. Yarom, N. Benger, Recovering openssl ecdsa nonces using the flush+ reload cache side-channel attack., IACR Cryptol. ePrint Arch. 2014 (2014) 140.

[24] On vsyscalls and the vdso, https://lwn.net/Articles/446528/, accessed: (2021-01-16) (2011).
[25] Y. Oyama, How does malware use rdtsc? a study on operations executed by malware with cpu cycle measurement, in: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Springer, 2019, pp. 197–218.
[26] Time-stamp counter disabling oddities in the linux kernel, https://blog.cr0.org/2009/05/time-stamp-counter-disabling-oddities.html, accessed: (2021-01-16) (2009).
[27] Y. Ji, S. Lee, E. Downing, W. Wang, M. Fazzini, T. Kim, A. Orso, W. Lee, Rain: Refinable attack investigation with on-demand inter-process information flow tracking, in: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, 2017, pp. 377–390.
[28] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, J. Torrellas, Attack directories, not caches: Side channel attacks in a non-inclusive world, in: 2019 IEEE Symposium on Security and Privacy (SP), IEEE, 2019, pp. 888–904.
[29] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, D. Gruss, A systematic evaluation of transient execution attacks and defenses, in: 28th USENIX Security Symposium (USENIX Security 19), 2019, pp. 249–266.
[30] L. G. Bruinderink, A. Hülsing, T. Lange, Y. Yarom, Flush, gauss, and reload–a cache attack on the bliss lattice-based signature scheme, in: International Conference on Cryptographic Hardware and Embedded Systems, Springer, 2016, pp. 323–345.
[31] Y. Yarom, Mastik: A micro-architectural side-channel toolkit, Retrieved from School of Computer Science Adelaide: http://cs. adelaide. edu. au/yval/Mastik 16.
[32] D. Wang, A. Neupane, Z. Qian, N. B. Abu-Ghazaleh, S. V. Krishnamurthy, E. J. Colbert, P. Yu, Unveiling your keystrokes: A cache-based side-channel attack on graphics libraries., in: NDSS, 2019.
[33] X. Zhang, Y. Xiao, Y. Zhang, Return-oriented flush-reload side channels on arm and their implications for android devices, in: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, 2016, pp. 858–870.
[34] B. Gulmezoglu, T. Eisenbarth, B. Sunar, Cache-based application detection in the cloud using machine learning, in: Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, 2017, pp. 288–300.
[35] S. Sharath, A. Basu, Performance of eucalyptus and openstack clouds on futuregrid, International Journal of Computer Applications 80 (13).
[36] T. Deshane, Z. Shepherd, J. Matthews, M. Ben-Yehuda, A. Shah, B. Rao, Quantitative comparison of xen and kvm, Xen Summit, Boston, MA, USA (2008) 1–2.
[37] M. Loukeris, Efficient computing in a safe environment, in: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2019, pp. 1208–1210.
[38] P. Guide, Intel® 64 and ia-32 architectures software developer's manual, Volume 3B: System programming Guide, Part 2 (11).
[39] P. J. Mucci, S. Browne, C. Deane, G. Ho, Papi: A portable interface to hardware performance counters, in: Proceedings of the department of defense HPCMP users group conference, Vol. 710, Citeseer, 1999.
[40] O. Acıiçmez, W. Schindler, A vulnerability in rsa implementations due to instruction cache analysis and its demonstration on openssl, in: Cryptographers' Track at the RSA Conference, Springer, 2008, pp. 256–273.
[41] O. Acıiçmez, B. B. Brumley, P. Grabher, New results on instruction cache attacks, in: International Workshop on Cryptographic Hardware and Embedded Systems, Springer, 2010, pp. 110–124.
[42] R. Hund, C. Willems, T. Holz, Practical timing side channel attacks against kernel space aslr, in: 2013 IEEE Symposium on Security and Privacy, IEEE, 2013, pp. 191–205.
[43] Y. Zhang, A. Juels, M. K. Reiter, T. Ristenpart, Cross-vm side channels and their use to extract private keys, in: Proceedings of the 2012 ACM conference on Computer and communications security, 2012, pp. 305–316.
[44] Z. Wu, Z. Xu, H. Wang, Whispers in the hyper-space: high-bandwidth and reliable covert channel attacks inside the cloud, IEEE/ACM Transactions on Networking 23 (2) (2014) 603–615.
[45] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, A. D. Keromytis, The spy in the sandbox: Practical cache attacks in javascript and their implications, in: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, 2015, pp. 1406–1418.
[46] A. Moghimi, G. Irazoqui, T. Eisenbarth, Cachezoom: How sgx amplifies the power of cache attacks, in: International Conference on Cryptographic Hardware and Embedded Systems, Springer, 2017, pp. 69–90.
[47] D. Cock, Q. Ge, T. Murray, G. Heiser, The last mile: An empirical study of timing channels on sel4, in: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, 2014, pp. 570–581.
[48] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al., sel4: Formal verification of an os kernel, in: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, 2009, pp. 207–220.
[49] B. Coppens, I. Verbauwhede, K. De Bosschere, B. De Sutter, Practical mitigations for timing-based side-channel attacks on modern x86 processors, in: 2009 30th IEEE Symposium on Security and Privacy, IEEE, 2009, pp. 45–60.
[50] B. Rodrigues, F. M. Quintão Pereira, D. F. Aranha, Sparse representation of implicit flows with applications to side-channel detection, in: Proceedings of the 25th International Conference on Compiler Construction, 2016, pp. 110–120.
[51] F. Liu, R. B. Lee, Random fill cache architecture, in: 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, IEEE, 2014, pp. 203–215.
[52] D. Page, Partitioned cache architecture as a ėide-channel defence mechanism.
[53] G. Dessouky, T. Frassetto, A.-R. Sadeghi, Hybcache: Hybrid side-channel-resilient caches for trusted execution environments, in: 29th USENIX Security Symposium (USENIX Security 20), 2020, pp. 451–468.
[54] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, D. Ponomarev, Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks, ACM Transactions on Architecture and Code Optimization (TACO) 8 (4) (2012) 1–21.
[55] Z. Wang, R. B. Lee, New cache designs for thwarting software cache-based side channel attacks, in: Proceedings of the 34th annual international symposium on Computer architecture, 2007, pp. 494–505.
[56] F. Liu, H. Wu, K. Mai, R. B. Lee, Newcache: Secure cache architecture thwarting cache side-channel attacks, IEEE Micro 36 (5) (2016) 8–16.
[57] A. Harris, S. Wei, P. Sahu, P. Kumar, T. Austin, M. Tiwari, Cyclone: Detecting contention-based cache information leaks through cyclic interference, in: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, 2019, pp. 57–72.
[58] W.-M. Hu, Reducing timing channels with fuzzy time, Journal of computer security 1 (3-4) (1992) 233–254.
[59] R. Martin, J. Demme, S. Sethumadhavan, Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks, in: 2012 39th Annual International Symposium on Computer Architecture (ISCA), IEEE, 2012, pp. 118–129.
[60] D. Zhang, A. Askarov, A. C. Myers, Language-based control and mitigation of timing channels, in: Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation, 2012, pp. 99–110.
[61] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, F. T. Chong, Sapper: A language for hardware-level security policy enforcement, in: Proceedings of the 19th international conference on Architectural support for programming languages and operating systems, 2014, pp. 97–112.
[62] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, B. Hardekopf, Caisson: a hardware description language for secure information flow, ACM Sigplan Notices 46 (6) (2011) 109–120.
[63] D. E. Porter, M. D. Bond, I. Roy, K. S. McKinley, E. Witchel, Practical fine-grained information flow control using laminar, ACM Transactions on Programming Languages and Systems (TOPLAS) 37 (1) (2014) 1–51.
[64] J. Nomani, J. Szefer, Predicting program phases and defending against side-channel attacks using hardware performance counters, in: Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy, 2015, pp. 1–4.
[65] Y. Zhang, M. K. Reiter, Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud, in: Proceedings of the

2013 ACM SIGSAC conference on Computer & communications security, 2013, pp. 827–838.