MeltdownDetector: A Runtime Approach for Detecting Meltdown Attacks

Taha Atahan Akyildiz, Can Berk Guzgeren, Cemal Yilmaz, and Erkay Savas

Faculty of Engineering and Natural Sciences
Sabanci University
Istanbul 34956
Turkey

Abstract

In this work, we present a runtime approach, called MeltdownDetector, for detecting, isolating, and preventing ongoing Meltdown attacks that operate by causing segmentation faults. Meltdown exploits a hardware vulnerability that allows a malicious process to access memory locations, which do not belong to the process, including the physical and kernel memory. The proposed approach is based on a simple observation that in order for a Meltdown attack to be worthwhile, either a single byte of data located at a particular memory address or a sequence of consecutive memory addresses (i.e., sequence of bytes) need to be read, so that a meaningful piece of information can be extracted from the data leaked. MeltdownDetector, therefore, monitors segmentation faults occurring at memory addresses that are close to each other and issues a warning at runtime when these faults become "suspicious." Furthermore, MeltdownDetector flushes the cache hierarchy after every suspicious segmentation fault, which, in turn, prevents any information leakage. In the experiments we carried out to evaluate the proposed approach, MeltdownDetector successfully detected all the attacks in every subject workload under every combination of attack detection configuration and attack variation used in the experiments and correctly pinpointed all the malicious processes involved in these attacks without issuing

Email address: {aakyildiz, cguzgeren, cyilmaz, erkays}@sabanciuniv.edu (Taha Atahan Akyildiz, Can Berk Guzgeren, Cemal Yilmaz, and Erkay Savas)

any false alarms and without leaking even a single byte of data. Furthermore, the runtime overhead of MeltdownDetector was 0.55%, on average.

Keywords:
Meltdown, side-channel attacks, countermeasures, runtime detection, prevention, and isolation

---

1. Introduction

Memory isolation is an integral security mechanism provided by today's modern computing systems. Being a part of both instruction set architectures and operating systems, this mechanism prevents processes from accessing each other's memory or kernel memory, thus enabling processes with different owners run safely and concurrently on the same physical computing platforms, such as on the same workstation/server platforms or in the same cloud.

A recently discovered attack, called *Meltdown* (officially logged as CVE-2017-5754 under the Common Vulnerabilities and Exposures database [1]), can, however, overcome memory isolation, allowing malicious processes to access the memory that is not allocated to them, including the physical and kernel memory [2].

Meltdown exploits a hardware vulnerability, which is present in commonplace processors today, such as in modern Intel microarchitectures [2]. In a nutshell, a malicious process attempts to access a memory location that does not belong to the process, knowing that the request will eventually fail (with a segmentation fault, for example). However, due to out-of-order execution – a hardware mechanism for speeding up computations by reordering the instructions to be executed, such that clock cycles which would otherwise be wasted are utilized to the extent possible – the access violation is caught after the content of the requested memory location is brought to the cache. Once the violation is determined, the requesting process is notified by the operating system (OS), e.g., by raising a `SIGSEGV` signal. From this point on, although the content of the memory location cannot directly be read in the user space since

the access has already been forbidden, as the content is now in the cache, the malicious process can use microarchitectural cache-based covert channels, such as Flush+Reload [3], to infer the content.

According to [4], almost all the Intel processors, which implement out-of-order execution, since 1995, are potentially susceptible to Meltdown. The attack can also be carried out on processors with transactional memory support, which were debuted in late 2013 by Intel. In this work, we are solely concerned with Meltdown attacks on conventional, non-transactional memory systems. Note that such systems have been widely deployed in the field as laptop, workstation, server, and cloud computers since 1995. Therefore, even if the hardware vulnerabilities causing Meltdown were fixed in the new generations of processors, it would certainly take a long while to replace all the susceptible processors in the field [5]. Therefore, detecting and preventing Meltdown attacks are still relevant.

Part of the reason as to why Meltdown can be quite hazardous is that due to some solid performance reasons, modern operating systems, such as Linux, map both the physical and kernel memory into to the virtual address space of every running process. This, however, enables any malicious process employing Meltdown to access any part of the physical and kernel memory during any phase of its execution. Consequently, one countermeasure against Meltdown could be to avoid having such memory mappings or to make it difficult for malicious processes to make sense of these mappings. This can, indeed, be achieved by using recent OS patches, such as KPTI (formerly known as KAISER) [6] and EPTI [7]. However, not all of these patches are general and they can significantly affect the performance of real-world environments. For example, EPTI supports virtual machines only, requiring a hardware-support feature called "EPT switching" within guest virtual machines without hypervisor involvement, and it has a runtime overhead of up to 13% [7]. Similarly, KPTI, although a general solution, has an overhead in the range of 13% and 17% [5, 8].

In this work, we present a low-overhead approach, called *MeltdownDetector*, to detect, isolate, and prevent ongoing Meltdown attacks at runtime. The

proposed approach is based on a simple observation we make: In order for a Meltdown attack to be worthwhile, either a single byte of data located at a particular memory address or a sequence of consecutive memory addresses (i.e., sequence of consecutive bytes) need to be read, so that a meaningful piece of information can be extracted from the data leaked. MeltdownDetector, therefore, monitors segmentation faults occurring at memory addresses that are close to each other and issues a warning at runtime when these faults become "suspicious." Furthermore, to prevent even a single byte of data from being leaked, MeltdownDetector flushes the cache hierarchy after every suspicious segmentation fault, regardless of when or whether an ongoing attack is detected.

To evaluate the proposed approach, we carried out a series of experiments with well-known benchmarks (i.e., workloads) where we varied both the MeltdownDetector configuration and the attack itself (e.g., by having the attack extended over a period of time and carried out with multiple processes in coordination). In these experiments, MeltdownDetector successfully detected all the attacks in every subject workload under every combination of MeltdownDetector configuration and attack variation and correctly pinpointed all the malicious processes involved in these attacks without issuing any false alarms and without leaking even a single byte of data. Furthermore, the runtime overhead of MeltdownDetector was 0.55%, on average.

The remainder of the paper is organized as follows: Section 2 discusses the Meltdown attack; Section 3 presents the attacker model; Section 4 introduces the proposed approach; Section 5 presents the experiments we carried out and the results we obtained; Section 6 discusses potential threats to validity; Section 7 presents related work; and Section 8 presents concluding remarks and possible directions for future work.

2. Meltdown Attack

This section provides a brief overview of the Meltdown attack. The interested reader can obtain further details in [2].

Algorithm 1: A code segment (taken from [2]) illustrating the integral part of Meltdown.

```
1 xor rax, rax
2 retry:
3 mov al, byte [rcx]
4 shl rax, 0xc
5 jz retry
6 mov rbx, qword [rbx + rax]
```

Algorithm 1 (taken from [2]) illustrates the integral part of Meltdown. The byte located at address `rcx`, which does not belong to the process issuing the instruction, is attempted to be read (line 3). The malicious process knows that the operation will eventually fail with a segmentation fault (e.g., with a `SIGSEGV` signal). However, due to out-of-order execution the subsequent instructions (lines 4-6) are executed before the signal is actually raised. More specifically, the value of the byte at address `rcx` is first read into register `rax` (line 4) and then used as an index to access an element in a strategically created array, which starts at address `rbx` (line 6). In the remainder of the paper, this array will be referred to as the "`rbx` array".

As expected, the segmentation fault finally occurs at address `rcx`. Although the value stored in `rax`, i.e., the value of the byte stored at address `rcx`, is never released to the user space, the element at index `rax` in the `rbx` array (which is, indeed, created by the malicious process) is brought to the cache. This, in turn, creates a microarchitectural side-channel because the malicious process can now discover the value of the byte at address `rcx`, by figuring out the element, thus the index, accessed in the `rbx` array.

To this end, a microarchitectural cache-based covert channel, such as Flush+Reload [3], can be used. First, all the elements of the `rbx` array are evicted from the cache. To this end, the `clflush` machine instruction, which evicts the cache line that contains a given memory address from the entire cache

hierarchy, can be used. Then, the malicious process executes the code segment given in Algorithm 1 to read the value of the byte located at address `rcx`, which belongs to a different process. After the segmentation fault has occurred, the malicious process probes the cache to figure out which element was accessed in the `rbx` array, thus the value of the byte.

To this end, the malicious process iterates over the elements in the `rbx` array, each of which maps to a different cache line, and search for an element, which takes shorter time to access, compared to the others. Note that accessing an element in cache is profoundly much faster than accessing an element in memory. Therefore, the index of the element, which takes shorter time to access, will be the value of the byte located at address `rcx`.

To deal with cache prefetching – a mechanism for improving the runtime performance of systems by fetching memory to cache before the respective content is actually needed – the malicious process, rather than accessing consecutive elements in the `rbx` array, accesses elements that are far apart from each other based on the value of the byte to be leaked. For example, instead of accessing `[rbx + rax]`, one can access `[rbx + 4096 * rax]` to avoid the noise caused by cache prefetching.

3. Attacker Model

We consider that the attacker uses Meltdown that operates by causing segmentation faults. The attacker's ultimate expectation is to discover the values stored at some memory locations, for which the attacker does not have any appropriate permissions to access. There is locality in the attacks in the sense that the attacker attempts to read chunks of memory, each of which consists of either a single byte or a sequence of consecutive bytes, so that a meaningful piece of information can be extracted from the data discovered.

In order for the proposed approach to work, the attacker can neither bypass the segmentation fault handling mechanism provided in cooperation with both the hardware and operating system (e.g., the attacker cannot hide a segmenta-

tion fault from the operating system kernel) nor temper with the data collected by them for each segmentation fault (e.g., the addresses at which the faults have occurred) or prevent the proposed approach from accessing this data.

4. Meltdown Detector

MeltdownDetector monitors the segmentation faults occurring at memory addresses that are close to each other and emits a warning when a collection of such faults becomes suspicious. We distinguish between three types of segmentation faults: *type 0*, *type 1*, and *type 2*.

The type 0 category represents the segmentation faults that are caused by dereferencing null pointers – a common cause of segmentation faults. For example, given $u32 * addr = NULL$, an operation of the form $*addr{=}\ldots$ would cause a segmentation fault at address 0x00 and an operation of the form $*(addr{+}{+})$ would cause a fault at address 0x04. Consequently, we mark all the segmentation faults that occur at addresses, which are smaller than or equal to a predetermined *cutoff* value (in our case, *cutoff* = 1024), as type 0 faults. Since these segmentation faults are considered benign, MeltdownDetector filters out all type 0 faults without taking any actions.

Unlike the type 0 category, the type 1 and type 2 categories are determined by the operating system. For example, in Linux-like systems, the type of a segmentation fault is communicated to the user space using the `si_code` or `si_errno` field (depending on the platform) of the `siginfo_t` structure.

Type 1 segmentation faults occur when a process attempts to access a memory address that does not belong to the process, i.e., a memory address, which is not in the same address space with the process. Type 2 segmentation faults, on the other hand, occur when the address is in the same address space with the process, but the process does not have the appropriate rights to access it.

We opt to distinguish between type 1 and type 2 segmentation faults, because it turns out that some software systems may occasionally use type 2 faults in order to speed up certain types of operations. For example, modern

Java virtual machines (JVMs) leverage type 2 faults to perform safepoint polls and/or thread-local handshakes [9]. Occasionally, JVM needs to stop all the Java threads (i.e., safepoint polling) or a group of selected threads (i.e., thread-local handshaking). However, this needs to be done at a point in time, which is "safe" for the threads. Furthermore, as these polls are occasional (if not rare at all), the mechanism to be used should be quite efficient in the absence of a poll. One solution approach is to make the threads call into JVM when they are at a safe point. However, for various technical reasons, such as the presence of threads running in a busy-loop, this approach is not desirable. Instead, JVM allocates a physical page and picks an address in this page, which is typically aligned with the page (i.e., for 4 KB physical pages, an address that ends with 0x000). When a thread is at a safe point, it accesses the selected memory location. In the absence of a poll, the access is quite fast and the execution resumes with the next instruction (i.e., nothing happens). However, when JVM needs to call in the threads, it prevents all the accesses to the selected physical page (e.g., by calling `mprotect` with `PROT_NONE`), so that the threads attempting to access the predetermined memory location cause type 2 segmentation faults. The segmentation faults are then handled by JVM, effectively enabling JVM to call in the threads. For each JVM instance (i.e., JVM process), the physical page(s) and the memory address(es) used for polling, are fixed.

In MeltdownDetector, we handle type 1 and type 2 segmentation faults similarly but separately. We do this to further improve the accuracy of the proposed approach by utilizing the differing characteristics of both types. More specifically, when dealing with type 2 faults, we use virtual memory addresses. That is, for type 2 faults, we check the closeness between virtual addresses. This is because, to have a type 2 segmentation fault at an address, the address should be in the same address space with the offending process. And, in a given address space, the same virtual addresses map to the same physical addresses. Furthermore, the probability of multiple processes with different address spaces causing segmentation faults at virtual addresses that are close to each other by chance is low, especially when the size of the virtual address space is considered,

which can be as large as 2^{48}. Note that Meltdown can cause type 2 segmentation faults, if the attacker targets some protected memory locations in its own address space.

When dealing with type 1 segmentation faults, on the other hand, we use physical page offsets. That is, for type 1 faults, we check the closeness between physical page offsets. Note that this type of faults occur when the attacker targets memory addresses that belong to other processes. As the address spaces of the offending and the victim processes are different, different virtual addresses can map to the same physical addresses. We, therefore, use physical page offsets because consecutive bytes within a page are guaranteed to have consecutive page offsets regardless of the address space they belong to. This, in turn, enables the proposed approach to detect and prevent attacks that are carried out in coordination with multiple processes (possibly with separate address spaces), where each process reads a different portion of the memory, such that a sequence of consecutive bytes are read in effect. Note that although consecutive page offsets do not necessarily indicate consecutive addresses, we conjecture that the probability of the same or multiple processes causing type 1 segmentation faults on page offsets that are close to each other by chance is low, especially in production environments.

Last but not least, for either type of segmentation faults, we opt to carry out the analysis in a system-wide manner, where the segmentation faults, regardless of the processes causing them, are analyzed together, rather than on a per process basis, where the segmentation faults caused by each process are analyzed in isolation. We do this to detect and prevent coordinated attacks carried out by multiple processes.

4.1. Approach

Algorithm 2 illustrates the core steps in MeltdownDetector, which are carried out in the kernel space after a segmentation fault has been detected, but right before the control is passed to the user space (i.e., right before a `SIGSEGV` signal is raised). In this algorithm, *addr* is the virtual memory address, at

Algorithm 2: MeltdownDetector

Input: *addr*: memory address
type: segmentation fault type
pid: process ID

```
1  if addr ≤ cutoff then
2      ▷ type 0
3      Do nothing
4  else
5      if type is type 1 then
6          ▷ type 1
7          addr ← addr & PAGE_OFFSET_MASK
8          hist ← type1.hist
9      else
10         ▷ type 2
11         hist ← type2.hist
12     counter ← 0
13     hist ← hist ∪ {.addr ← addr, .pid ← pid}
14     for each address a in [addr - (diameter/2), addr + (diameter/2)] do
15         if a in hist then
16             counter++
17     if counter ≥ threshold then
18         Issue a warning & report the pids involved
19     Flush cache hierarchy
```

which the segmentation fault has occurred, *type* is the type of the segmentation fault as determined by the operating system, *pid* is the ID of the process causing the fault, and *type*1.*hist* and *type*2.*hist* are the hash tables (indexed by memory addresses) for previously seen type 1 and type 2 segmentation faults, respectively.

If *addr* is smaller than or equal to a predetermined *cutoff* value (in our case, *cutoff* = 1024), we mark the segmentation fault as type 0 and filter it out without taking any further actions (lines 1-4). Otherwise (i.e., *addr* > *cutoff*), we check to see if the segmentation fault is a type 1 or type 2 fault (line 5).

For a type 1 segmentation fault, we use the physical page offset of *addr* (line 7 – in our case, $PAGE\_OFFSET\_MASK = 0xFFF$ for 4 KB physical pages). For a type 2 segmentation fault, we use the virtual address *addr* as it is (lines 9-11). As from this point on (lines 12-20), the handling of the segmentation fault does not depend on its type, we make the *hist* pointer to point to the respective hash table; to *type*1.*hist* for type 1 faults (line 8) and to *type*2.*hist* for type 2 faults (line 11).

We then populate the *hist* hash table with *addr* and *pid* (line 13), indicating that the respective segmentation fault is a suspicious one (as it is not type 0). To determine whether a warning needs to be issued or not, we check to see whether other segmentation faults have occurred at addresses around *addr* (lines 14-17).

To this end, MeltdownDetector offers two parameters: *diameter* and *threshold*. If the total number of segmentation faults that have occurred in the range $[addr - (diameter/2), addr + (diameter/2)]$ is larger than or equal to *threshold* (line 18), a warning indicating the presence of a potential ongoing Meltdown attack is issued together with the IDs of the processes responsible for the warning (line 19). Note that when physical page offsets are in use, the range of addresses to be checked is computed in a circular manner; offset 0 comes after the last page offset and the last page offset comes before offset 0.

Furthermore, regardless of whether a warning has been issued or not, when the segmentation fault is of type 1 or type 2, we flush the cache hierarchy (line 20). We do this to prevent even a single byte of information from being leaked.

Once a warning has been issued, various countermeasures, such as terminating the suspicious processes or migrating them to different machines, can be taken. However, these countermeasures are beyond the scope of this work.

4.2. Implementation

We have developed two implementations of MeltdownDetector for Linux operating system (kernel v3.10.0-957.5.1.el7.x86_64). In one implementation, we directly modify the OS kernel, whereas in the other implementation we develop a dynamic instrumentation script using SystemTap – a framework for dynamically instrumenting Linux kernel at runtime [10]. Note that although both implementations are semantically equivalent (and, indeed, share exactly the same MeltdownDetector code), they differ in the way they are deployed. More specifically, the former implementation requires a rebuild of the kernel followed by a reboot of the system, whereas the latter can be installed or uninstalled at runtime without requiring any reboots. Although the latter implementation is obviously more flexible, it is not clear how much additional runtime overhead (if any) the dynamic instrumentation framework may incur. We, therefore, opted to have both implementations, so that we can compare their overheads.

In both implementations, we instrument the `__bad_area_nosemaphore(...)` kernel function defined in `arch/x86/mm/fault.c` with an implementation of the algorithm given in Algorithm 2. Note that the aforementioned kernel function is invoked after a segmentation fault has been detected, but before a `SIGSEGV` signal is raised and the control is passed to the user space.

The memory address, at which the segmentation fault has occurred, as well as the kernel-identified type of the fault, are passed as arguments to this kernel function. Furthermore, the ID of the process responsible for the segmentation fault is obtained by using the `current` pointer – a global kernel pointer pointing to the `task_struct` of the currently executing (thus, the offending) process.

Last but not least, we flush the cache hierarchy by using the `wbinvd()` kernel function, which uses a machine instruction with the same name (i.e., the Write

Back and Invalidate Cache instruction) to invalidate (i.e., flush) all the cache lines in the hierarchy.

5. Experiments

We carried out a series of experiments to evaluate MeltdownDetector.

5.1. Experimental Design

In these experiments, we manipulated 4 independent variables: attack variation, workload, attack detection configuration, and implementation.

Attack variations. As the base Meltdown attack, we used the implementation [11] provided by the original paper [2]. More specifically, we used `memdump.c`, which attempts to dump the entire physical memory.

To further evaluate the proposed approach on different attack variations, we have also modified the base attack, such that an attack can be extended over a period of time and carried out by multiple processes. Note that both mechanisms can be leveraged by an attacker to become stealthier. For example, an attacker, instead of attacking an address space one memory address after another in a rapid manner, can attack an address and then go silent for a while before attacking the subsequent address. The attacker can even utilize multiple malicious processes, each of which attacks different memory addresses, such that the data collectively gathered by these processes forms the basis of the attack. In both cases, the intention of the attacker is to avoid being detected by replacing short periods of intense suspicious activities with long periods of indistinct activities, which are generally harder to detect.

To carry out the aforementioned evaluations, we introduced two parameters to the base attack, namely N and T. N is the number of processes that are concurrently carrying out the attack. T is the maximum amount of time a malicious process waits before attacking the subsequent byte (i.e., address). Given N and T, we divide the address space, against which the attack will be carried out (i.e., the physical memory space), into non-overlapping sliding

windows of N addresses each and spawn N malicious processes, each of which has its own distinct memory space. Each process uses the original attack code (i.e., `memdump.c`) to attack a distinct address within a window and waits for a random amount of time, not exceeding T, before attacking a distinct address in the subsequent window.

We, in particular, experiment with all the setups represented by the cross product of $N = \{1, 2, 5, 10\}$ and $T = \{30, 60, 180, 300\}$. The durations in T are given in seconds. Furthermore, in the experimental setups where $N = 1$, consecutive memory addresses are attacked one after another by a single process (as is the case in the original attack), but the attack is extended over a period of time depending on the value of T.

Workloads. We use two well-known benchmark suites as workloads, namely Phoronix [12] and SPECjvm2008 [13]. We chose these workloads because they leverage real life applications to mimic a wide spectrum of common computations, including the ones that are typically carried out on servers and workstations, such as cryptographic and numerical computations, audio and video encoding, various CPU-, memory-, network-, and disk-bound computations, and database operations.

Phoronix is an open-source benchmark suite, which fully automates the installation, execution, and result aggregation for a wide variety of benchmarks. We, in particular, chose Phoronix, as it is the benchmark, on which the performance of the well-known preventive countermeasure against Meltdown, namely KPTI, has been evaluated for numerous CPU architectures, so that we can compare the runtime overhead of MeltdownDetector to that of the only alternative we know of. In the experiments, we used all the Phoronix benchmarks that we could install on our experimental platform (see Table 1 for more information).

SPECjvm2008 is also a well-known benchmark [13], which aims to measure the performance of Java runtime environments. We opted to use this additional benchmark, because, compared to Phoronix, it caused significantly more number of segmentation faults due to the use of type 2 segmentation faults for safepoint polls and thread-local handshakes (Section 4). Furthermore, as also suggested by

the developers of this benchmark [13], we ran the benchmark with the `--lagom` option, which guarantees that a fixed set of workloads were executed every time the benchmark is run.

Attack detection configurations. Throughout the experiments, we use $cutoff = 1024$ (Section 4.1). We chose this value because it is large enough to filter out the segmentation faults that occur due to null-pointer dereferences or due to pointer arithmetic operations involving null pointers.

Furthermore, to study the effect of the *diameter* and *threshold* parameters on attack detection, we experiment with different MeltdownDetector configurations. More specifically, we use $threshold = \{2, 4\}$ with $diameter = 8$, $threshold = \{2, 4, 8\}$ with $diameter = 16$, $threshold = \{2, 4, 8, 16\}$ with $diameter = 32$, and $threshold = \{2, 4, 8, 16, 32\}$ with $diameter = 64$.

Implementation. We experiment with two implementations of MeltdownDetector: a kernel implementation and a SystemTap implementation (Section 4.2). We do this solely to evaluate the runtime overheads of these otherwise semantically equivalent implementations.

All the experiments were carried out on an E5630 Intel Xeon platform with 32 GB of RAM and 32 KB of L1, 256 KB of L2, and 12288 KB of L3 cache memory, running CentOS v6.1810.2 operating system with kernel v3.10.0-957.5.1.el7.x86_64.

5.2. Evaluation Framework

To evaluate the accuracy of the proposed approach, we keep track of whether the attacks are detected or not. To evaluate the extent to which the malicious processes are pinpointed, we compare the IDs of the suspicious processes identified by the proposed approach with those of the actual malicious processes. To evaluate how early the attacks are detected, we measure the number of distinct memory addresses, at which the attacker caused segmentation faults, e.g., the number of bytes attempted to be stolen by the attacker, before the attack is detected. The faster the attacks are detected, the better the proposed approach is.

To evaluate the extent to which flushing the cache hierarchy after every suspicious segmentation fault (i.e., type 1 or type 2 faults), prevents Meltdown, we use the reliability tool (i.e., `reliability.c`) that comes with the Meltdown implementation [11]. This tool simply measures the accuracy of the attack by computing the percentage of the bytes that are successfully read. The smaller the accuracy, the better the proposed approach is. For example, an attack accuracy of 0 indicates that not even a single byte of data is leaked.

Last but not least, we measure the runtime overhead of the proposed approach. To this end, we execute the workloads both on the original kernel and on the modified kernel; and compute the overhead as $((P' - P)/P) * 100$, where P and P' are the execution times of the workloads on the original and modified kernel, respectively. The smaller the runtime overhead, the better the proposed approach is. For each experimental setup, we repeat the experiments 5 times.

Note that between the configuration parameters of MeltdownDetector, namely *diameter* and *threshold*, the runtime overhead of the proposed approach solely depends on the former as the loop between lines 14-17 in Algorithm 2 iterates *diameter* times. We, therefore, computed the runtime overhead for the smallest and largest values of *diameter* used in the experiments (i.e., $diameter = 8$ and $diameter = 64$, respectively) by arbitrarily setting the *threshold* parameter to its largest value for a given *diameter*. Furthermore, as the runtime overheads obtained from different *diameter* values in the experiments were close to each other, we report the average overheads (together with the minimum and maximum overheads).

5.3. Data and Analysis

We first observed that non-stop 165-hour (about 7 days) and 1.5-hour executions of the Phoronix and SPECjvm2008 workloads, caused a total of 42 and 41627 segmentation faults, respectively (Table 1). In the Phoronix workloads, all the segmentation faults were type 0 faults, i.e., they all occurred at memory addresses below the $cutoff = 1024$ value. In the SPECjvm2008 workload, while 278 of the segmentation faults were type 0, the remaining 41349 were type 2.

When we ran the same workload by disabling the use of type 2 segmentation faults for polling (Section 4) with the `-XX:-ThreadLocalHandshakes` option of Java, no segmentation faults were observed, further assuring that all of these segmentation faults were indeed caused by JVMs for safepoint polls and thread-local handshakes. To sum up, during the 7-day non-stop executions of our workloads, we observed no benign type 1 segmentation faults, i.e., none of the processes attempted to access a memory location that did not belong to them.

We then observed that, for the Phoronix workloads, MeltdownDetector incurred no runtime overheads; as all the segmentation faults occurred in these workloads were type 0, they were all filtered out by MeltdownDetector without performing any actions, such as flushing the cache hierarchy (lines 1-4 in Algorithm 2). This result is especially important considering that KPTI, which is a well-known approach for mitigating Meltdown, incurs runtime overheads of between 13% and 17% for the same workloads [5, 8].

For the SPECjvm2008 workload, the kernel implementation of MeltdownDetector (Section 5.2) incurred an average overhead of 0.72% ($min = 0.46\%$ and $max = 0.97\%$). We, furthermore, observed that the SystemTap-based implementation of MeltdownDetector, i.e., the more flexible implementation (compared to the statically modified kernel implementation), which does not require any reboots of the systems for deployment, imposed similar overheads: $min = 0.37\%$, $avg = 0.37\%$, and $max = 0.38\%$.

suite	workload	all segfaults	type 0	type 1	type 2	processes causing segfaults	execution time (secs.)
Phoronix	Audio Encoding	0	0	0	0	0	352.707
	Compilation	0	0	0	0	0	6255.819
	Compression	0	0	0	0	0	886.798
	Computational Biology	0	0	0	0	0	278.195
	Computational	0	0	0	0	0	4668.748
	CPU	0	0	0	0	0	17351.362
	Cryptography	0	0	0	0	0	1047.164
	Database	3	3	0	0	1	404435.648
	Desktop Graphics	3	3	0	0	3	52643.977
	Disk	0	0	0	0	0	16022.908
	GUI Toolkits	0	0	0	0	0	471.657
	Linux System	0	0	0	0	0	36514.507
	Machine Learning	36	36	0	0	12	10481.544
	Memory	0	0	0	0	0	3218.241
	Multicore	0	0	0	0	0	16828.762
	Network	0	0	0	0	0	1660.745
	Server	0	0	0	0	0	4566.657
	Video Encoding	0	0	0	0	0	489.485
	Workstation	0	0	0	0	0	16825.223
SPEC	SPECjvm2008	41627	278	0	41349	52	5342.268

Table 1: Statistics about the benign segmentation faults occurred (i.e., the ones caused by benign processes) during the executions of the subject workloads. The columns, respectively, depict the workloads, the numbers of total, type 0, type 1, and type 2 segmentation faults, the numbers of distinct processes causing these segmentation faults, and the execution times of the workloads.

We next observed that no false alarms were raised by MeltdownDetector. That is, none of the unfiltered segmentation faults (i.e., the benign type 2 segmentation faults, as we did not have any benign type 1 faults) occurred during the executions of our subject workloads, were close enough to each other, such that a false warning could be issued under any of the MeltdownDetector configurations used in the experiments (Section 5.1). Indeed, there was not even a pair of segmentation faults that occurred at consecutive memory addresses.

We then observed that MeltdownDetector successfully detected all the attacks in every subject workload under every combination of MeltdownDetector configuration and attack variation used in the experiments and correctly pinpointed all the malicious processes involved in these attacks. And, it did so without leaking even a single byte of data. We validated this by running the reliability tool (Section 5.2) that came with the original attack distribution, both on the original and modified kernels. While the accuracies of the attacks on the former were typically above 99%, those on the latter were 0% all the time, indicating that no data at all was leaked.

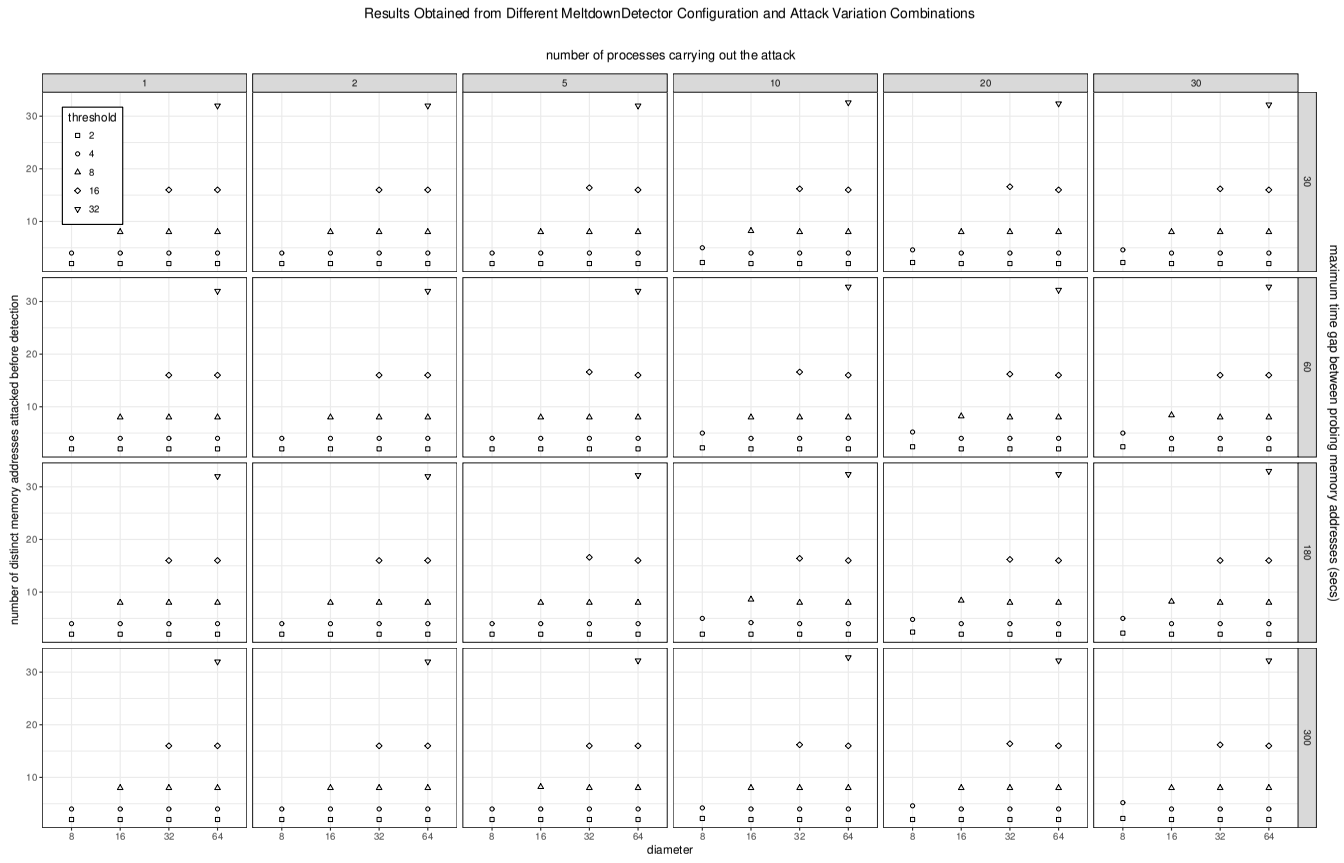

Figure 1: Number of distinct memory addresses probed by an attacker, before the attack was detected under various MeltdownDetector configurations and attack variations. The horizontal and vertical axes denote the *diameter* parameter and the number of distinct memory addresses probed by the attacker, respectively, whereas the columns and rows depict the N parameter (number of processes concurrently carrying out the attack) and the T parameter (maximum time lag in seconds, for which a malicious process waits in between probing two addresses), respectively.

Note further that regardless of where the attacks might be started during the executions of the workloads, no false alarms would have been issued or no benign processes would have been incorrectly marked as suspicious. This is because all the benign unfiltered segmentation faults were of type 2, whereas all the malicious ones were of type 1. Had the Meltdown attacks generated type 2 segmentation faults, then some of the benign processes could have been incorrectly marked as suspicious. An in-depth analysis, however, revealed that all of the 41349 benign type 2 segmentation faults caused by the SPECjvm2008 workload, occurred at only 107 distinct virtual memory addresses. Considering the size of a virtual address space for a process, which was as large as 2^{48}, the probability of other processes causing segmentation faults around these 107 addresses by chance was low. Note further that Meltdown attacks can cause type 2 segmentation faults only when a malicious process carries out the attack against some protected memory locations, which are in the same address space with the process (Section 4), thus limiting the scope of an attack. All other attempts, such as accessing memory locations that do not belong to the malicious process, result in type 1 segmentation faults.

Last but not least, we evaluated how early the attacks were detected under different MeltdownDetector configurations and attack variations (Section 5.1). Figure 1 presents the results we obtained.

We first observed that, as expected, in the original Meltdown attack as well as in all of its variations where the attack was carried out by probing only the consecutive memory addresses using a single process (i.e., $N = 1$), regardless of the values of *diameter* and T (time lag between attacks), the attacks were detected after *threshold* number of addresses had been probed by the attacker, which, given a *threshold* value, the earliest possible time at which an attack can be detected. For example, when $threshold = 2$, the original attacks were detected as soon as the second distinct memory address was probed by the attacker.

We then observed that MeltdownDetector did not get affected by the attack variations used in the experiments in terms of the number of distinct memory

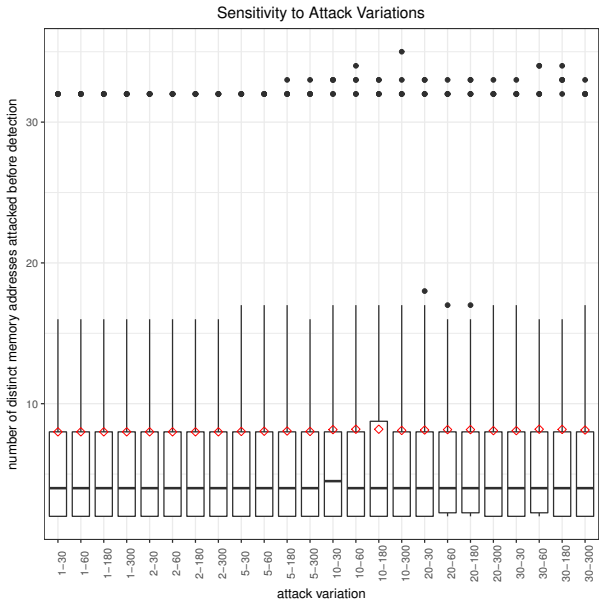

Figure 2: Sensitivity of MeltdownDetector to attack variations. The numbers in the horizontal tick labels, respectively, represent the number of malicious processes concurrently carrying out the attack (i.e., N) and the maximum amount of time a malicious process waits before probing the subsequent byte (i.e., T). For each attack variation, the distribution of the data obtained from all of the MeltdownDetector configurations used in the experiments, is visualized.

addresses probed before the attack could be detected. We believe that this was because all the memory addresses attacked during the experiments were locally close to each other – a decision we made to mimic realistic attacks (Section 3). For every attack variation, Figure 2 visualizes the distribution of the results we obtained from all of the MeltdownDetector configurations used in the experiments in the form of a box and whisker plot. Each tick on the horizontal axis depicts an attack variation, where the numbers in the tick label, respectively, represent the number of malicious processes concurrently carrying out the attack (i.e., N) and the maximum amount of time a malicious process waits before probing the subsequent byte (i.e., T). For each box, the bottom, middle, and top bars depict the first, second (i.e., median), and third quartile of the data, respectively, whereas the diamond shape depicts the average value. The

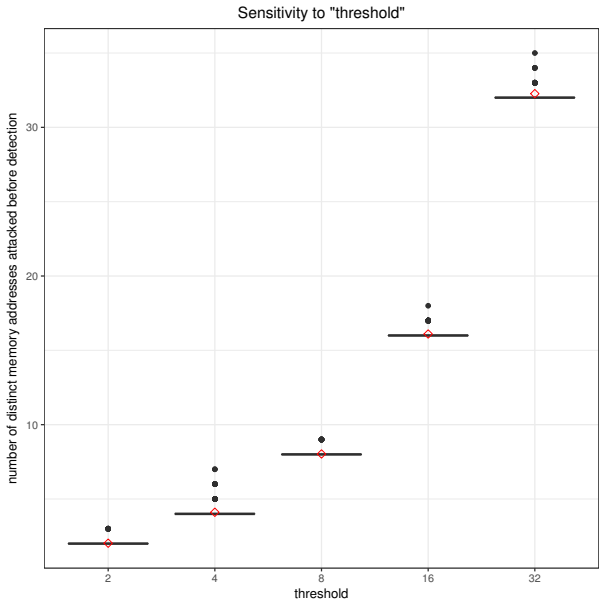

Figure 3: Sensitivity of MeltdownDetector to the *threshold* parameter. For each *threshold* value, the distribution of the data obtained from all the experiments with the same *threshold* value is visualized.

average values were within 0.186 of each other with $min = 8$, $max = 8.186$, and $stddev = 0.07$.

Analyzing the sensitivity of MeltdownDetector to the *threshold* parameter, we observed, as expected, that as the *threshold* value increased, the number of memory addresses probed before the attack can be detected increased (Figure 3). The average values were 2.03, 4.11, 8.03, 16.10, and $32,27$ for $threshold = 2$, 4, 8, 16, and 32, respectively.

Analyzing the sensitivity of MeltdownDetector to the *diameter* parameter, we observed that this parameter generally did not have any profound effects in the experiments. We believe that this was also due to the locality of the memory addresses attacked during the experiments (Section 3). Figure 4 summarizes the results we obtained. As the *threshold* parameter had an effect on the results (Figure 3) and not all the *diameter* values were used with every *threshold* value (as $threshold \leq diameter$), we, in this figure, visualize only the distributions of

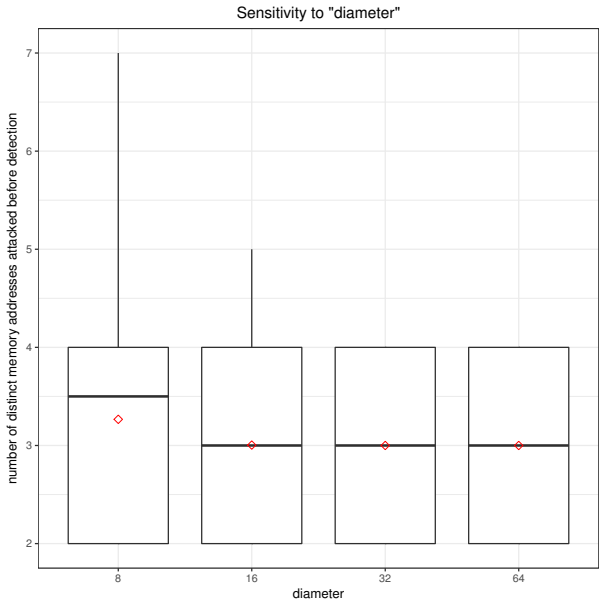

Figure 4: Sensitivity of MeltdownDetector to the *diameter* parameter. For each *diameter* value, only the results obtained from the experiments where $threshold = 2$ or $threshold = 4$ are visualized as not all *threshold* values are used with every *diameter* value.

the results obtained from the experiments where $threshold = 2$ or $threshold = 4$ (i.e., the ones that were used with all the *diameter* values). The average values were within 0.2 of each other with $min = 3.0$, $max = 3.2$, and $stddev = 0.076$.

Note that regardless of whether or when the attacks are detected, MeltdownDetector does not leak any information. This is because we flush the cache hierarchy after every type 1 an type 2 segmentation faults whether or not an attack has been detected (line 20 in Algorithm 2).

5.4. Discussion

In this study, we flush the cache hierarchy in its entirety by using the `wbinvd()` kernel function, affecting all the processes sharing the same cache memories. One way to further reduce the runtime overhead of MeltdownDetector, could be to flush only the cache lines belonging to the suspicious process(es). We, indeed, evaluated a similar approach in our empirical studies, where we only

flushed the cache line that contained the suspicious address, at which a segmentation fault had occurred. To this end, we used the `clflush(addr)` kernel function, which, in turn, uses a machine instruction with the same name to do the job. We, however, observed that this strategy did not fully prevent the leak. More specifically, although the accuracies of the attacks generally dropped to less than 3% (obtained by using the reliability tool that came with the original attack distribution – see section Section 5.2 for more information), they were not 0% as was the case with `wbinvd()`. We could not exactly figure out the reason behind this phenomenon and left it as a future work.

6. Threats to Validity

In this work we are mainly concerned with external threats to study. One possible threat concerns the representativeness of the Meltdown attacks used in the study. To the best of our knowledge, there are no reported incidents involving Meltdown attacks in the literature [4]. Therefore, we used the implementation of the attack provided by the original paper [2]. We, furthermore, systematically varied it by having multiple processes carrying out the attack, which is extended over a period of time where each malicious process waits for a random amount of time before targeting the subsequent byte.

Another threat is the representativeness of the workloads used in the study. To alleviate this threat as much as possible we used two well-known benchmarks, namely Phoronix [12] and SPECjvm2008 [13], both of which leverage real-life applications to mimic a wide spectrum of common computations, including the ones that are typically carried out on servers and workstations, such as cryptographic and numerical computations, audio and video encoding, various CPU-, memory-, network-, and disk-bound computations, and database operations. We, furthermore, evaluated the proposed approach by considering all possible scenarios, in which the workloads and attack variations can be interleaved with each other.

A similar threat concerns the representativeness of the hardware and software (e.g., operating system) platforms used in the experiments. We, on the other hand, used a quite commonplace CPU architecture and a well-known operating system. Note further that the proposed approach can be implemented in any operating system that handles segmentation faults.

7. Related Work

Side-channels are unintended manifestations about the secret information-dependent aspects of system operations, e.g., the execution time, power consumption, electromagnetic emanation, micro-architectural artifacts, etc [14, 15, 16, 17, 18]. Among all different types of side-channels, the ones that are most relevant to this work are the cache-based side-channels [19, 20, 21, 22, 3, 15, 23, 24, 25, 26, 27, 28, 29], as Meltdown also leverages a cache-based side-channel to discover the content stored at a specific memory address.

A number of approaches have been developed to detect the presence of ongoing side-channel attacks [30, 31, 32, 33]. However, all of these existing approaches address the side-channel attacks that are carried out against the software implementation of cryptographic applications with the goal of discovering sensitive information processed by the application, such as a secret key. From this perspective, Meltdown is quite different, because in this attack, there is no target process to protect from the attackers, in the sense that an attacker can target any parts of the physical and/or kernel memory, thus any process, during an attack. In side-channel attacks addressed by the existing approaches, however, the cryptographic process, code or sensitive data, which is to be protected, such as a process encrypting/decrypting messages using AES and performing ECDSA signature generation [19, 3, 30, 31, 34, 35], is typically known beforehand. Therefore, detection mechanisms generally make use of this information. For example, SpyDetector [30], instruments the parts where cryptographic operations are carried in an application, which can be exposed to attacks, such that when the extent to which the application suffers from cache misses becomes

"suspicious," a warning about a possible ongoing attack is issued. In the Meltdown attack, however, any part of any process, including the kernel processes, can be targeted. Therefore, it is generally not clear how to adopt the existing approaches (if at all possible) to detect Meltdown attacks.

Furthermore, the aforementioned detection approaches require either an online or an offline training phase, where data about benign and/or malicious processes is collected, so that supervised or unsupervised detection models can be trained. MeltdownDetector, on the other hand, requires no training at all.

8. Concluding Remarks

In this paper, we have presented MeltdownDetector, which is a runtime approach for detecting, isolating, and preventing ongoing Meltdown attacks that operate by causing segmentation faults. The proposed approach does not need any training and can be implemented in any operating system that handles segmentation faults. Furthermore, operational systems can even be patched with MeltdownDetector at runtime without requiring any reboots, by using dynamic kernel instrumentation frameworks, such as SystemTap [10] in Linux-like operating systems.

We believe that this line of research is novel. One possible avenue for future research is to develop approaches for performing selective flushing of the cache hierarchy (i.e., flushing only the cache lines that belong to the suspicious processes), which can, in turn, further reduce the runtime overheads. Another avenue is to develop similar approaches, which are possibly integrated with hardware performance counters-based system traces [30, 31], to detect Meltdown attacks that leverage transactional memory instructions.

References

[1] Common vulnerabilities and exposures, `https://cve.mitre.org`, accessed: 2019-05-24.

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, M. Hamburg, Meltdown: Reading kernel memory from user space, USENIX Security Symposium 2 (5) (2018) 43–51.

[3] Y. Yarom, K. Falkner, Flush+reload: A high resolution, low noise, l3 cache side-channel attack, in: 23rd USENIX Security Symposium (USENIX Security 14), USENIX Association, San Diego, CA, 2014, pp. 719–732.
URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[4] Meltdown and spectre: Vulnerabilities in modern computers leak passwords and sensitive data, https://meltdownattack.com, accessed: 2019-05-10.

[5] R. Bennett, C. Callahan, S. Jones, M. Levine, M. Miller, A. Ozment, How to live in a post-meltdown and -spectre world, Queue 16 (4) (2018) 30:18–30:30. doi:10.1145/3277539.3281471.
URL http://doi.acm.org/10.1145/3277539.3281471

[6] Kaiser: Hiding the kernel from user space, https://lwn.net/Articles/738975/, accessed: 2019-05-10.

[7] Z. Hua, D. Du, Y. Xia, H. Chen, B. Zang, EPTI: Efficient defence against meltdown attack for unpatched vms, in: 2018 USENIX Annual Technical Conference (USENIX ATC 18), USENIX Association, Boston, MA, 2018, pp. 255–266.
URL https://www.usenix.org/conference/atc18/presentation/hua

[8] The current spectre / meltdown mitigation overhead benchmarks on linux 5.0., https://www.phoronix.com/scan.php?page=article&item=linux50-spectre-meltdown&num=1, accessed: 2019-07-08.

[9] JEP 312: Thread-local handshakes., http://openjdk.java.net/jeps/312, accessed: 2019-07-08.

[10] Systemtap, https://sourceware.org/systemtap, accessed: 2019-05-14.

[11] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, M. Hamburg, Meltdown Proof-of-Concept (2018).
URL https://github.com/IAIK/meltdown

[12] Phoronix test suite., http://www.phoronix-test-suite.com, accessed: 2019-06-21.

[13] Specjvm2008 benchmarking suite., https://www.spec.org/jvm2008, accessed: 2019-05-16.

[14] P. C. Kocher, Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems, in: N. Koblitz (Ed.), Advances in Cryptology – CRYPTO'96, Vol. 1109 of Lecture Notes in Computer Science, Springer-Verlag Berlin Heidelberg, 1996, pp. 104–113, http://www.cryptography.com/public/pdf/TimingAttacks.pdf.

[15] O. Acıiçmez, Ç. K. Koç, Trace-driven cache attacks on aes (short paper), in: P. Ning, S. Qing, N. Li (Eds.), Information and Communications Security, Vol. 4307 of Lecture Notes in Computer Science, Springer-Verlag Berlin Heidelberg, 2006, pp. 112–121, full paper available at eprint.iacr.org/2006/138/.

[16] O. Aciicmez, C. K. Koc, J.-P. Seifert, On the power of simple branch prediction analysis, in: Proceedings of the 2Nd ACM Symposium on Information, Computer and Communications Security, ASIACCS '07, ACM, New York, NY, USA, 2007, pp. 312–320. doi:10.1145/1229285.1266999.
URL http://doi.acm.org/10.1145/1229285.1266999

[17] O. Aciicmez, C. K. Koc, J.-P. Seifert, Predicting secret keys via branch prediction, in: Proceedings of the 7th Cryptographers' Track at the RSA Conference on Topics in Cryptology, CT-RSA'07, Springer-Verlag, Berlin,

Heidelberg, 2006, pp. 225–242. doi:10.1007/11967668_15.
URL http://dx.doi.org/10.1007/11967668_15

[18] Q. Ge, Y. Yarom, D. Cock, G. Heiser, A survey of microarchitectural timing attacks and countermeasures on contemporary hardware, Journal of Cryptographic Engineering 8 (1) (2018) 1–27. doi:10.1007/s13389-016-0141-6.
URL https://doi.org/10.1007/s13389-016-0141-6

[19] D. J. Bernstein, Cache-timing attacks on aes (2005).

[20] N. Benger, J. Pol, N. P. Smart, Y. Yarom, "ooh aah... just a little bit": A small amount of side channel can go a long way, in: Proceedings of the 16th International Workshop on Cryptographic Hardware and Embedded Systems — CHES 2014 - Volume 8731, Springer-Verlag, Berlin, Heidelberg, 2014, pp. 75–92. doi:10.1007/978-3-662-44709-3_5.
URL https://doi.org/10.1007/978-3-662-44709-3_5

[21] G. Irazoqui, M. S. Inci, T. Eisenbarth, B. Sunar, Wait a minute! a fast, cross-vm attack on aes, in: A. Stavrou, H. Bos, G. Portokalidis (Eds.), Research in Attacks, Intrusions and Defenses, Springer International Publishing, Cham, 2014, pp. 299–319.

[22] Y. Yarom, N. Benger, Recovering openssl ecdsa nonces using the flush+reload cache side-channel attack, IACR Cryptology ePrint Archive 2014 (2014) 140.
URL https://eprint.iacr.org/2014/140

[23] G. Bertoni, V. Zaccaria, L. Breveglieri, M. Monchiero, G. Palermo, AES power attack based on induced cache miss and countermeasure, in: International Symposium on Information Technology: Coding and Computing (ITCC 2005), Volume 1, 4-6 April 2005, Las Vegas, Nevada, USA, 2005, pp. 586–591. doi:10.1109/ITCC.2005.62.
URL http://dx.doi.org/10.1109/ITCC.2005.62

[24] M. Neve, J.-P. Seifert, Z. Wang, A refined look at bernstein's aes side-channel analysis, in: ASIACCS, 2006, p. 369.

[25] C. Rebeiro, M. Mondal, D. Mukhopadhyay, Pinpointing cache timing attacks on AES, in: VLSI Design, 2010, pp. 306–311.

[26] E. Tromer, D. A. Osvik, A. Shamir, Efficient cache attacks on aes, and countermeasures, J. Cryptology 23 (1) (2010) 37–71. doi:10.1007/s00145-009-9049-y.
URL http://dx.doi.org/10.1007/s00145-009-9049-y

[27] C. Percival, Cache missing for fun and profit, in: Proc. of BSDCan 2005, 2005.

[28] J. Blömer, V. Krummel, Analysis of countermeasures against access driven cache attacks on AES, in: Selected Areas in Cryptography, 2007, pp. 96–109.

[29] O. Aciiçmez, W. Schindler, A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL, in: CT-RSA, 2008, pp. 256–273.

[30] Y. Kulah, B. Dincer, C. Yilmaz, E. Savas, Spydetector: An approach for detecting side-channel attacks at runtime, International Journal of Information Securitydoi:10.1007/s10207-018-0411-7.
URL https://doi.org/10.1007/s10207-018-0411-7

[31] M. Chiappetta, E. Savas, C. Yilmaz, Real time detection of cache-based side-channel attacks using hardware performance counters, Appl. Soft Comput. 49 (C) (2016) 1162–1174. doi:10.1016/j.asoc.2016.09.014.
URL https://doi.org/10.1016/j.asoc.2016.09.014

[32] M. Payer, Hexpads: A platform to detect "stealth" attacks, in: J. Caballero, E. Bodden, E. Athanasopoulos (Eds.), Engineering Secure Software and Systems, Springer International Publishing, Cham, 2016, pp. 138–154.

[33] T. Zhang, Y. Zhang, R. B. Lee, Cloudradar: a real-time side-channel detection system in clouds, in: Intrusions and Defenses (RAID), 2016.

[34] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, M. Costa, Strong and efficient cache side-channel protection using hardware transactional memory, in: 26th USENIX Security Symposium (USENIX Security 17), USENIX Association, Vancouver, BC, 2017, pp. 217–233.
URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/gruss

[35] S. Briongos, G. Irazoqui, P. Malagón, T. Eisenbarth, Cacheshield: Detecting cache attacks through self-observation, in: Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy, CODASPY '18, ACM, New York, NY, USA, 2018, pp. 224–235. doi:10.1145/3176258.3176320.
URL http://doi.acm.org/10.1145/3176258.3176320