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
Email: {aakyildiz, cguzgeren, cyilmaz, erkays}@sabanciuniv.edu

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 successful, 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 caches after every suspicious segmentation fault, preventing even a single byte of data from being leaked. In the experiments we carried out to evaluate the proposed approach, MeltdownDetector successfully detected all the attacks, correctly isolated all the malicious processes, and did so at the earliest possible time after the attacks have started with an average runtime overhead of 0.34% and without even leaking a single byte of information.

Index Terms—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 us to concurrently run processes with different owners on the same physical computing platforms, such as on the same desktop/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 [11]), can, however, overcome memory isolation, allowing malicious processes to access the memory that is not allocated to them, including the physical and kernel memory [17].

Meltdown exploits a hardware vulnerability, which is present in commonplace processors today, such as in modern Intel microarchitectures [17]. 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 memory 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 as the access has already been forbidden, since the content is now in the cache, the malicious process uses microarchitectural cache-based covert channels, such as Flush+Reload [27], to infer the content.

According to [18], 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, desktop, server, and Cloud computers since 1995. Therefore, even if the hardware vulnerabilities causing Meltdown are fixed in the new generations of processors, it will certainly take a long while to replace all the susceptible processors in the field [6]. Therefore, detecting and preventing Meltdown attacks will be relevant at least until then.

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, which 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) [13]. However, these patches can significantly affect the performance of real-world environments by a factor in the range of 5% to 30% [6].

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 successful, 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, to prevent even a single byte of data from being leaked, MeltdownDetector flushes the cache hierarchy after every suspicious segmentation fault, regardless of whether an ongoing attack is detected or not.

Note that MeltdownDetector incurs runtime overhead only when a segmentation fault occurs. Therefore, due to the factors that the benign processes, especially the ones running in production environments, are not likely to cause frequent segmentation faults, especially when the total number of memory accesses that they make is considered, and that the malicious processes, after being detected, can be dealt with by, for example, terminating them, we conjecture that the runtime overhead of MeltdownDetector can be kept under control.

We implemented MeltdownDetector in a Linux kernel and carried out a set of experiments. In these experiments, MeltdownDetector successfully detected all the attacks, correctly isolated all the malicious processes, and did so at the earliest possible time after the attacks have started with an average runtime overhead of 0.34% and without even leaking a single byte of information.

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

Listing 1: A code segment (taken from [17]) 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]
```

2 Meltdown Attack

This section provides a brief overview of the Meltdown attack. The interested reader can obtain further details in [17].

Listing 1 (taken from [17]) 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 `SIGSEGV` signal (i.e., segmentation fault). 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 `SIGSEGV` signal is finally raised 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 [27], is used. First, all the elements of the `rbx` array are evicted from the cache by, for example, using machine instructions, such as `clflush`, which evicts the cache line that contains a given memory address from the entire cache hierarchy. Then, the malicious process executes the code segment given in Listing 1 to read the value of a byte that belongs to a different process. After the segmentation fault has occurred, the malicious process finally probes the cache to figure out which element was accessed in the `rbx` array, thus the value of the byte.

More specifically, the elements in the `rbx` array are probed to detect an element (or elements) that can be accessed faster than the others; fetching an element from cache is much faster than fetching it from memory. 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 MELTDOWN DETECTOR

MeltdownDetector monitors the segmentation faults occurring at memory addresses that are close to each other and emits a warning when a set of such faults becomes suspicious. Note that this approach can be carried out on per process basis, where the segmentation faults caused by each process are analyzed in isolation, or in a system-wide manner, where all the segmentation faults regardless of the processes causing them are analyzed together. Furthermore, as addresses to be monitored, virtual addresses or physical addresses can be used.

Although the proposed approach is general enough to be used with any combination of the design decisions discussed above, we, in this work, opt to analyze the segmentation faults in a system-wide manner by using physical memory page offsets. We made the former decision because it enables one to detect and prevent attacks that are carried out by multiple processes in coordination, where each process reads a different portion of the memory, such that a sequence of bytes are read in effect.

We made the latter decision because extracting page offsets does not require a conversion between virtual and physical addresses and consecutive addresses (virtual or physical) have consecutive page offsets, given that offsets are processed in a circular manner. Although the reverse is not necessarily true, i.e., consecutive page offsets do not necessarily indicate consecutive addresses, the odds of having a single process or multiple processes cause segmentation faults on consecutive page offsets by chance, are low, thus tolerable.

3.1 Approach

Listing 2 illustrates the core steps in MeltdownDetector, which is carried out in the kernel space after a segmentation fault is detected, but right before the control is passed to the user space (i.e., right before a `SIGSEGV` signal is raised). In the algorithm, $addr$ is the physical page offset of the memory address, at which a segmentation fault has just occurred (in our case, the least significant 12 bits of an address), pid is the ID of the process responsible for the segmentation fault, and $hist$ is a hash table of previously seen `SIGSEGV` signals, for each of which both $addr$ (as the key) and pid (as the value) are stored.

Listing 2: MeltdownDetector

```
1  if addr > cutoff then
2  |   counter ← 0
3  |   hist ← hist ∪ {.addr ← addr, .pid ← pid}
4  |   for each a in [addr - range, addr + range] do
5  |   |   if a in hist then
6  |   |   |   counter++
7  |   |   end
8  |   end
9  |   if counter ≥ threshold then
10 |   |   issue a warning
11 |   end
12 |   flush cache hierarchy
13 end
```

One common cause of segmentation faults in processes is null pointer dereferencing (i.e., dereferencing memory address 0x00). Since these segmentation faults as well as the ones caused by pointer arithmetic operations involving null pointers, are benign, we filter out (line 1) all the `SIGSEGV` signals that are raised at addresses, which are smaller than or equal to a predetermined $cutoff$ value (in our case, $cutoff = 1024$). For example, given $u32 * addr = NULL$, while an operation of the form $*addr = \ldots$ would raise a `SIGSEGV` signal at address 0x00, an operation of the form $*(addr{+}{+})$ would raise the same signal at address 0x04.

If $addr$ is not filtered, we populate the $hist$ hash table with $addr$ and pid (line 3), indicating that the respective segmentation fault is a suspicious one. To determine the level of suspiciousness, we check to see whether other segmentation faults have occurred at addresses around $addr$ (lines 4-8).

To this end, MeltdownDetector offers two parameters: $range$ and $threshold$. If the total number of segmentation faults that have occurred in the range $[addr - range, addr + range]$ is larger than or equal to $threshold$ (line 9), a warning indicating the presence of a potential ongoing Meltdown attack is issued together with the respective process IDs as the potential malicious processes (line 10). Note that as we use physical page offsets in the analysis, the range of addresses to be checked is computed in a circular manner; offset 0 comes after the last offset and last offset comes before offset 0.

Furthermore, regardless of whether a warning has been raised or not, when $addr$ is not a filtered address, we flush the cache hierarchy (line 12) to prevent even a single byte of information from being leaked. Once a warning has been issued, on the other hand, various countermeasures, such as terminating the suspicious processes or migrating them to different machines, can be taken to further prevent these processes from harming the system by, for example, causing denial of

service (DoS) [28]. However, these countermeasures of the latter form are beyond the scope of this work.

3.2 Implementation

We have implemented MeltdownDetector in Linux kernel v3.10.0-957.5.1.el7.x86_64 (Section 4). Note, however, that the same implementation can readily be converted to a SystemTap script – a tool for dynamically instrumenting Linux kernel at runtime [24], such that the patch can be deployed at runtime without even requiring to reboot the underlying machine.

In the kernel, we implemented the algorithm given in Listing 2 as a part of the `__bad_area_nosemaphore(...)` kernel function defined in `arch/x86/mm/fault.c`, which is one of the functions invoked 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 is passed as an argument to this kernel function and the ID of the process responsible for the fault is obtained by using the `current` pointer – a pointer to the `task_struct` of the currently executing process. Furthermore, we flush the cache hierarchy by using the `wbinvd()` kernel function, which uses a machine instruction with the same name to invalidate (i.e., flush) all the cache lines in the hierarchy.

4 Experiments

We carried out a series of experiments to evaluate MeltdownDetector.

4.1 Experimental Setup

In these experiments, as the Meltdown attack code we used the implementation [16] provided by the original paper [17]. More specifically, we used the one in `memdump.c`, which attempts to dump the entire physical memory.

As the workload present during the attack, we used SPECjvm2008 benchmark suite [23], which leverages real life applications to mimic a wide spectrum of common computations, including the the ones that are typically carried out on servers, such as cryptographic and numerical computations. This benchmark suite is composed of a number of sub-benchmarks, each which can be executed separately. To create a realistic setup, we randomly divided this suite into 10 non-overlapping groups of 4 sub-benchmarks each. In the remainder of the paper, we refer to these groups as workloads.

For each workload, we first started all the sub-benchmarks within the workload. We then executed the Meltdown attack after a warm-up period and let everything to run in parallel until the benchmarks are completed.

MeltdownDetector was configured, such that the $cutoff$ value used for filtering out addresses was 1024, and the $range$ and $threshold$ values used for detection were 8 and 4, respectively (Listing 2). Note that all of these values are parameters, with which MeltdownDetector can be configured. We opted to use this configuration in the experiments, because the former parameter is large enough to filter out the segmentation faults that occur due to dereferencing null pointers or due to pointer arithmetic operations involving null pointers. Similarly, we used the latter set of parameters as they, in a sense, mimic a scenario where a 128-bit (16-byte) secret key is the target of the attack. Note further that although the threshold value we used was 4, no information was actually leaked to the attacker as we flush the cache hierarchy after every segmentation fault of interest (line 12 in Listing 2).

We also configured MeltdownDetector, such that after detecting the presence of an ongoing attack, no further predictions and flushes are performed, mimicking the situation that the malicious processes are dealt with by, for example, terminating them.

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.

4.2 Evaluation Framework

To evaluate the accuracy of the proposed approach, we kept track of whether the attack under each workload was detected or not. To evaluate the extent to which the malicious process(es) can be isolated (i.e., pinpointed), we compared the ID(s) of the suspicious process (i.e., the ones that caused a warning) with that (those) of the actual malicious process(es). To evaluate the precision, we measured the number of unique 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, prevents Meltdown, we used the reliability tool (i.e., `reliability.c`) that comes with the Meltdown implementation [16]. This tool simply measures the accuracy of the attack by computing the percentage of the bytes that are read successfully. The smaller the accuracy, the better the proposed approach is. Furthermore, an accuracy of 0, indicates that not even a single byte of data is leaked.

Last but not least, we have also measured the runtime overheads. To this end, we executed the workloads both on the original kernel and on the kernel instrumented with MeltdownDetector and computed the overhead as $((P'-P)/P)*100$, where P and P' are

workload	total segfaults	filtered segfaults	unfiltered segfaults	segfaults by benign processes	segfaults by Meltdown	benign processes causing segfaults	# of bytes attempted before detection
1	18743	20	18723	1	18722	7	4
2	40335	13	40322	10	40312	9	4
3	26499	12	26487	5	26482	9	4
4	215101	31	215070	30	215040	8	4
5	813621	36	813585	1416	812169	5	4
6	5400887	42	5400845	7405	5393440	5	4
7	4966860	12	4966848	1617	4965231	5	4
8	4659575	12	4659563	5157	4654406	5	4
9	1600207	14	1600193	3432	1596761	5	4
10	993984	53	993931	1412	992519	3	4

TABLE 1: Results obtained in the experiments. The columns, respectively, depict the workload, the number of total, filtered, and unfiltered segmentation faults, the number of segmentation faults caused by benign processes and Meltdown, the number of benign processes causing segmentation faults, and the number of memory addresses attempted to be read by Meltdown before the attack was detected.

the execution times of the workloads on the original and instrumented kernel, respectively. The smaller the runtime overhead, the better the proposed approach is. For each workload, the experiments were repeated 5 times.

4.3 Data and Analysis

Table 1 presents the results we obtained. Overall, MeltdownDetector successfully detected all the attacks, correctly isolated all the malicious processes, and did so at the earliest possible time after the attacks have started (which depends on the $threshold$ value) with an average runtime overhead of 0.34% and without even leaking a single byte of information.

More specifically, all the attacks were detected right after the malicious processes had caused segmentation faults on 4 unique memory addresses, which given $threshold = 4$, is, indeed, the earliest possible time at which an attack can be detected. An in-depth analysis also revealed that had we used $threshold = 2$, all the attacks would have been detected as soon as the attackers attempted to read the second byte.

Regardless of where the attacks have been detected, no information was leaked. To further validate this, we ran the reliability tool (Section 4.2) both on the original kernel and on the kernel instrumented with MeltdownDetector. While the accuracy of the attack on the former was typically above 99%, that on the latter was 0%.

We, furthermore, observed that, benign processes rarely caused segmentation faults especially when the number of memory accesses that they make are considered. Note that this, in turn, helps reduce the runtime overhead of MeltdownDetector as the overhead, such as flushing cache hierarchy, only occurs when a `SIGSEGV` signal is raised. Overall, only 0.001% (20530 out of 18735612) of all the segmentation faults were caused by benign processes. Furthermore, 0.01% (245 out of 20530) of all these benign segmentation faults were filtered out by using $cutoff = 1024$ (line 1 in Listing 2). The remainder (99.99%) of the segmentation faults were considered to be suspicious by MeltdownDetector. However, none of these segmentation faults have occurred at consecutive memory addresses or addresses that were close to each other, such that a warning was issued. Therefore, no false alarms were generated. That is, when the benchmarks were run without the presence of a Meltdown attack, MeltdownDetector issued no false warnings although the cache hierarchy was flushed after every unfiltered segmentation fault.

4.4 Discussion

Note that by using the `wbinvd()` kernel function, we flush the cache hierarchy in its entirety, affecting all the processes sharing the same caches. Therefore, 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 studies, where we only flushed the cache line that contains the suspicious address (`addr`), at which a segmentation fault has 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 (Section 2). We, however, observed that this strategy could not fully prevented the leak. More specifically, although the accuracy of the attacks generally dropped to less than 3%, it was 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.

5 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, however, there are no reported incidents involving Meltdown attacks in the literature [18]. Therefore, we used the proof-of-concept implementation of the attack provided by the original paper [17], [18]. Another threat is the workloads used in the study. To alleviate this threat as much as possible we used workloads, which leverage

real life applications to mimic a wide spectrum of common computations, including the the ones that are typically carried out on servers, such as cryptographic and numerical computations. A similar threat is the representativeness of the hardware and software platforms (e.g., operating system) 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 (i.e., pretty much in all the modern operating systems), regardless of the underlying hardware platform.

6 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 [2], [3], [4], [14]. Among all different types of side-channels, the ones that are most relevant to this work are the cache-based side-channels [1], [2], [5], [7], [8], [9], [12], [19], [21], [22], [25], [26], [27], 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 [10], [15], [20], [29]. 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, which is to be protected, such as a process encrypting/decrypting messages using AES [7], [10], [15], [27], is typically known beforehand. Therefore, detection mechanisms generally make use of this information. For example, SpyDetector [15], 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.

7 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, regardless of the underlying hardware platform. Operational systems can even be patched with MeltdownDetector at runtime without requiring any reboots by using dynamic kernel instrumentation frameworks, such as SystemTap [24], which is available 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) to further reduce the runtime overheads. Another avenue is to develop and evaluate various countermeasures that can be taken after an attack has been detected. Yet another avenue is to develop additional mechanisms, such that further trade-offs between accuracy, precision, and runtime overhead can be made. Last but not least, we are also working on developing similar approaches, which are possibly integrated with hardware performance counters-based system traces, to detect Meltdown attacks that leverage transactional memory instructions.

References

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

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

[3] O. Aciicmez, C. K. Koc, and 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, pages 225–242, Berlin, Heidelberg, 2006. Springer-Verlag.

[4] O. Aciicmez, C. K. Koc, and 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, pages 312–320, New York, NY, USA, 2007. ACM.

[5] N. Benger, J. Pol, N. P. Smart, and 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*, pages 75–92, Berlin, Heidelberg, 2014. Springer-Verlag.

[6] R. Bennett, C. Callahan, S. Jones, M. Levine, M. Miller, and A. Ozment. How to live in a post-meltdown and -spectre world. *Queue*, 16(4):30:18–30:30, Aug. 2018.

[7] D. J. Bernstein. Cache-timing attacks on aes, 2005.

[8] G. Bertoni, V. Zaccaria, L. Breveglieri, M. Monchiero, and 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*, pages 586–591, 2005.

[9] J. Blömer and V. Krummel. Analysis of countermeasures against access driven cache attacks on AES. In *Selected Areas in Cryptography*, pages 96–109, 2007.

[10] M. Chiappetta, E. Savas, and C. Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. *Appl. Soft Comput.*, 49(C):1162–1174, Dec. 2016.

[11] Common vulnerabilities and exposures. https://cve.mitre.org. Accessed: 2019-05-24.

[12] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar. Wait a minute! a fast, cross-vm attack on aes. In A. Stavrou, H. Bos, and G. Portokalidis, editors, *Research in Attacks, Intrusions and Defenses*, pages 299–319, Cham, 2014. Springer International Publishing.

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

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

[15] Y. Kulah, B. Dincer, C. Yilmaz, and E. Savas. Spydetector: An approach for detecting side-channel attacks at runtime. *International Journal of Information Security*, Jun 2018.

[16] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown Proof-of-Concept, 2018.

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

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

[19] M. Neve, J.-P. Seifert, and Z. Wang. A refined look at bernstein's aes side-channel analysis. In *ASIACCS*, page 369, 2006.

[20] M. Payer. Hexpads: A platform to detect "stealth" attacks. In J. Caballero, E. Bodden, and E. Athanasopoulos, editors, *Engineering Secure Software and Systems*, pages 138–154, Cham, 2016. Springer International Publishing.

[21] C. Percival. Cache missing for fun and profit. In *Proc. of BSDCan 2005*, 2005.

[22] C. Rebeiro, M. Mondal, and D. Mukhopadhyay. Pinpointing cache timing attacks on AES. In *VLSI Design*, pages 306–311, 2010.

[23] Specjvm2008 benchmarking suite. https://www.spec.org/jvm2008. Accessed: 2019-05-16.

[24] Systemtap. https://sourceware.org/systemtap. Accessed: 2019-05-14.

[25] E. Tromer, D. A. Osvik, and A. Shamir. Efficient cache attacks on aes, and countermeasures. *J. Cryptology*, 23(1):37–71, 2010.

[26] Y. Yarom and N. Benger. Recovering openssl ecdsa nonces using the flush+reload cache side-channel attack. *IACR Cryptology ePrint Archive*, 2014:140, 2014.

[27] Y. Yarom and K. Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, 2014. USENIX Association.

[28] S. Yu. *Distributed Denial of Service Attack and Defense*. Springer, 2014.

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