**Security Analysis of Meltdown on x86 Based Virtual Machines**

Prof. Azra Nasreen, Kshitij Tiwari, Mallappa Bagi

**Abstract**

The discovery of the Spectre and Meltdown microarchitectural vulnerabilities has generated a lot of interest in the security guarantees enforced by the CPU. By exploiting a race condition between speculative memory access and access protection checking, meltdown is able to bring privileged data into low level CPU caches. Thereafter, using the Flush and Reload attack, a cache side channel is exploited to leak the privileged data. This paper explores the techniques for implementing both, Flush and Reload, and Meltdown in Linux using C. For evaluating the attacks, a success rate metric is defined. On a 32-bit Linux virtual machine, Meltdown is found to leak data very reliably, with a success rate of 60%. The various mitigations for meltdown are applied and the success rate is found to be extremely low, with a maximum success rate of 0.1%. Finally, it is noted that the mitigations tested are stop-gap measures to fix the symptoms of meltdown. The core microarchitectural flaw cannot be fixed, which has resulted in a plethora of meltdown-like attacks such as Foreshadow, the MDS family, plundervolt, etc. being uncovered.

**Keywords:** Meltdown, Flush and Reload, Flush+Reload, virtual memory, memory protection, kernel modules, procfs, virtual machine, Linux, TSC, speculative execution, KPTI, microcode, Haswell

1. **Introduction**

The most fundamental security guarantee in a modern Operating System is memory isolation, which is enforced as part of virtual memory systems. A user-space process is allowed to legally access only a set of virtual addresses which are determined by the kernel’s memory management modules. Any attempt to access other addresses results in an exception.

Meltdown is a security vulnerability enabling an unprivileged process to access any memory location in its address space, which often contains kernel memory, hardware memory, and other user processes memory as a consequence of virtual memory mapping. Hence, Meltdown “melts down” the security barrier provided by virtual memory semantics, and renders the security guarantees made by the OS and the CPU null and void.

In this paper, a practical approach to implementing Meltdown on a 32-bit Linux virtual machine (VM) is presented. The various techniques for implementing the building blocks of the attack, i.e., timing, memory allocation, enforcing data locality, and exception handling are analyzed. The attack is then demonstrated, showing how a user-space process can leak data from the virtualized kernel memory. The results are then presented, followed by an analysis of the mitigations.

1. **Literature Survey**

Side channel attacks utilize information produced as a by-product of the operation of physical devices to steal information. Traditional attacks involve measuring electromagnetic radiation, power consumption, or using acoustics to record data such as key strokes. With the discovery of the Flush and Reload attack [1], interest in timing analysis as a side channel attack has increased substantially. It was clearly demonstrated that given a sufficiently accurate timing mechanism, one could leak the recently used data with the processor cache serving as a side channel. The approaches in [2] and [3] outline exploitations of such attacks.

Extending the idea of Flush and Reload, Meltdown [4] and Spectre [5] are aimed at breaking down the security guarantees of the CPU by allowing unprivileged processes to steal data from privileged memory locations. This grants a user-space process access to another user-space process’s memory, or access to the kernel-space memory. Access to kernel space memory is particularly dangerous as it contains the page table and the TLB, which contain the per-process memory-mappings. Both of these attacks exploit the side effects of speculative execution semantics in the processor.

Speculative execution [6] is a technique used by most modern processors to enhance the performance. This involves premature execution of independent instructions by the CPU to hide latencies. For example, a load operation stalls till the required data is loaded from the memory. This latency can be hidden by saturating the ALUs with instructions that do not depend on the loaded value and keeping their results ready for commit.

Critical flaws in the implementation of speculation on most Intel and some ARM CPUs gives rise to the Meltdown attack. The Spectre attack combines speculative execution and control flow prediction. All modern-day CPUs are vulnerable to Spectre.

Several similar vulnerabilities such as Foreshadow, the MDS family, Zombieload 2, and Plundervolt, have been uncovered. All of them follow the same basic principle of spectre and meltdown, using the cache as a covert side channel to leak data.

1. **Proposed system**

Values local to the process are first stolen using cache side channel attack. The possible methods for timing memory accesses are explored and optimal timing technique is arrived at. The Flush and Reload is then implemented and its efficacy tested. The ideas of kernel modules, makefiles and kernel memory allocation are then discussed. Finally, the meltdown attack is demonstrated and techniques to improve its efficacy are explored.

**3.1 Measuring Memory Access Time**

A very high level of accuracy is required when measuring time taken for memory access. The general paradigm for this involves reading CPU wall times before and after the access and measuring the difference between them. There is a large variety of APIs for getting the CPU wall time. For the system, the API is required to satisfy the following criteria.

1. Measurement is sufficiently precise, i.e., captures adequate granularity of time
2. The measurement is highly accurate. This implies that there should be minimum delay between the issue of the call and the return of wall time by the CPU. This involves taking care of various overheads, such as context switching to the kernel space, delay between CPU issue of clock read instruction and the return of clock value.

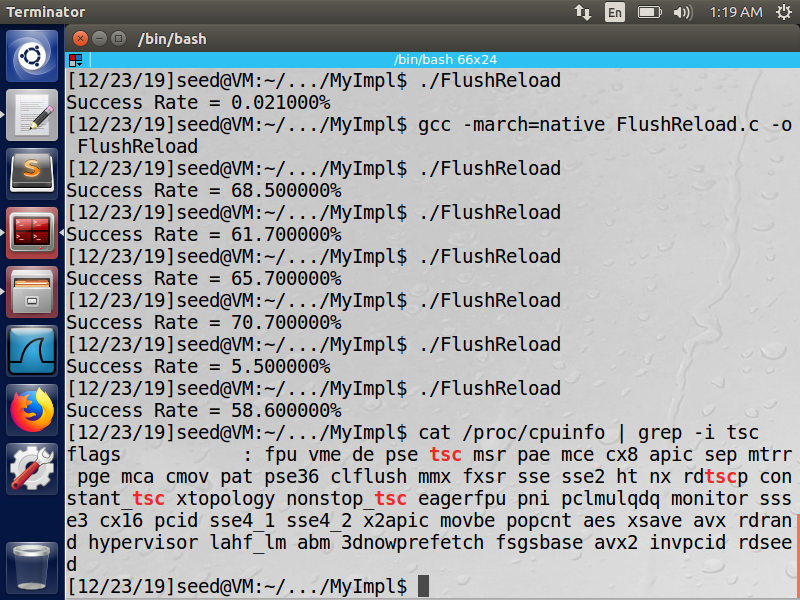
Calculation of wall clock time, i.e., the actual time is very expensive, as it requires conversion of counter values into wall clock format and synchronization of the CPU with known accurate times. A key observation here is that the wall clock time is not required, as the requirement is finding the time difference before and after the access.

The clock counters available in Linux are:

* **HPET: High Precision Event Timer** [7]. Jointly developed by Microsoft and Intel and incorporated in chipsets since 2005. It gives a precision of around 100ns.
* **Acpi\_pm clock** [8]**:** This clock source is obtained from the ACPI Power Management Timer. The advantage of this clock is the frequency is not dependent on the power, hence is unaffected by power changes. It tuns at 3.85Mhz, hence gives a precision of 280ns. It is more expensive to query and less accurate than the HPET.
* **TSC: Timestamp counter.** This counter is started during bootup and is driven by the CPU crystal oscillator. Hence it operates at the same frequency as the CPU. This enables it to run at extremely high update frequency. For CPU speed of 3Ghz, it is updated 3 times in a nanosecond. This is the most accurate clock source. It is easy to access as it just involves reading a register value.

As noted in [9], in older CPUs it is important to fix the frequency of the timer as the frequency of the CPU is not constant. This limitation is removed in modern CPUs by having invariant TSC, i.e., the TSC is run at a fixed rate.

Hence, the TSC is chosen as the clock source due to it having maximum accuracy and high speed of response.



**Figure 3.1 Checking for TSC support in the ‘cpuinfo’ file**

The processor information file “cpuinfo” located in the procfs [10] is first probed to check for TSC support as shown in Figure 3.1.

The RDTSCP C API is used for reading the contents of the TSC. RDTSCP is used to prevent out of order execution from giving incorrect time values. It performs any necessary serialization itself and is more efficient than manually enforcing serialization in the code.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**3.2 Flush and Reload**

An oracle buffer of 256 entries, each of size 4KB is created. This is done because the page size is 4KB and may cause multiple entries to be cached as a result of cache locality.

Figure 3.2 outlines the working of the Flush and Reload attack.

* The first step is the flush operation. All the entries of the oracle are initialized and then flushed from the cache. This ensures that none of the oracle entries are cached beforehand, and prevents incorrect results.
* Second step involves the execution of victim code. The victim uses a secret character (local to the victim function) to modify an entry in the oracle. When this happens the oracle entry gets cached. No further operations are performed by the victim to avoid overwriting the cache.
* The third step is the reload operation. The time taken to access each entry in the oracle is measured. The entry that has minimum access time is the one cached by the victim. Hence, the index of the cached entry is obtained, which can then be used to get the secret value.

Minimum access time on Oracle[k]

Victim caches oracle[k]

0 1 2 3 ……………..... **k** ……… 256

**Oracle Buffer**

**Cache**

Time access to each element of oracle

**Attacker**

**Figure 3.2 Working principle of Flush and Reload**

Another method of performing flush+reload involves determining a threshold value for cache access time. In this method, the first access whose time is less than the threshold is taken as the cached value. This offers some reduction in computation to be performed.

It was found that both techniques yielded similar success rates, and the minimum time technique was chosen since the amount of computation is not high, and it gives a better guarantee of correctness.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**3.3 Meltdown**

The attack targets kernel space memory from a user-space process. The two popularly used APIs for kernel memory allocation are vmalloc and kmalloc. Kmalloc guarantees that memory allocated is physically contiguous and offers higher performance. Vmalloc guarantees that memory allocated is virtually contiguous, and can distribute blocks if insufficient contiguous physical memory is available. Kmalloc is selected for the system since the attack is time sensitive.

Kernel modules are utilized to allocate kernel space memory. A kernel module is program that can be loaded and unloaded dynamically from kernel, without which kernel would need to be recompiled for adding features.

For exception handling, the C signal library is used to manipulate an error buffer. SIGSEGV is the error code thrown by the CPU on encountering invalid memory access. An error handler, which resets the buffer to the initial checkpoint is defined and registered with SIGSEGV. Once the attack has occurred, the handler is triggered, resetting the buffer and preventing program from crashing.

**3.4 Encouraging Data to remain in Cache**

Keeping the data hot in the cache is essential to the success of the attack. The conditions where it gets overwritten and flushed out of the cache need to be avoided. The cache state itself is private to the CPU and is generally treated as inaccessible to the programmer. Various techniques which increase the likelihood of data remaining in the cache are utilized prior to executing the attack.

1. Prefetching data  
   The non-faulting prefetch functions provided by x86 are used to prefetch the secret data. Prefetch is done from both the kernel module as well as the attacking user-space program.
2. Procfs  
   This is a specialized file system used by the kernel to manage data about the processes in a hierarchical manner. It can also be used to store per-process data which is made available to all the other processes. A procfs entry of the secret key from the kernel module is created. This is then opened by the attacker to cache the data.

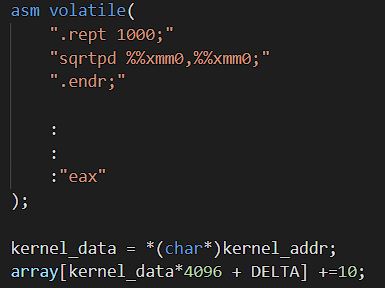
Further, the time between caching the data and running the attack is minimized. This avoids instances of data getting flushed out as a result of context switches.

**3.5 Delaying Commit – Keeping the ALU busy**

Most modern speculative processors follow a pipelined architecture. The execution of an instruction is divided into 4 stages, Issue, Execute, Write-Result and commit.

The commit phase occurs in order while execution phase occurs out of order.

The exception generated by illegal memory access is thrown by the CPU only after the instruction has been committed. However, the attacking instructions can be executed out of order and the data can be stolen from the cache. To this end, the commit of the segmentation fault needs to be delayed as much as possible in order to improve the chances of the attack.



**Figure 3.3 Arithmetically intensive assembly code preceding illegal memory access instruction**

As shown in Figure 3.3, computationally intensive arithmetic instructions, such as square root, log calculation, etc. are inserted prior to the illegal memory access. This introduces a delay equal to the execution time of the ALU.

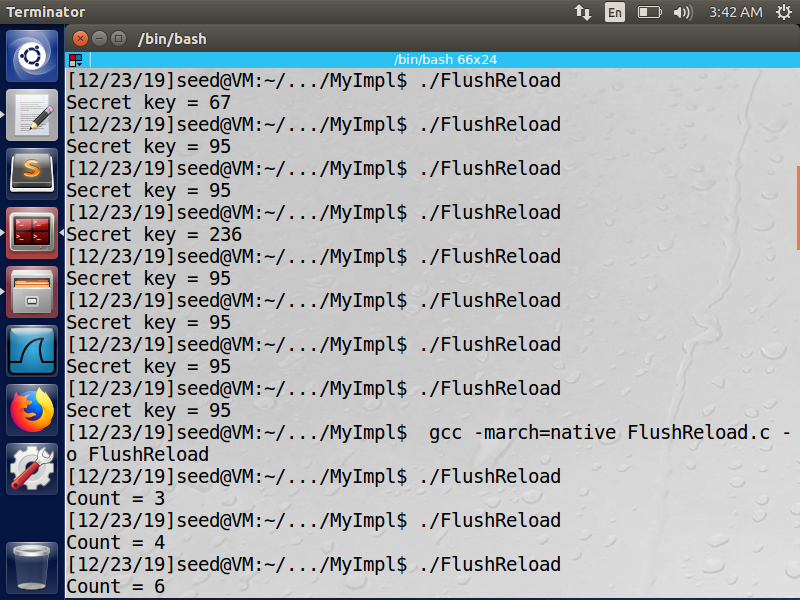
**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

1. **Results**A Success Rate metric is used to measure the performance of the system. For testing the system, a trial with 1000 runs of the attack is defined. For each trial, the success rate is defined in equation 4.1.

**Equation 4. 1 Success rate metric**

Where a successful run occurs when the attack is successful, i.e., the system leaks the data at the protected memory location. The total number of runs is set to 1000 for each trial.  
The metric is used to evaluate both Flush and Reload, and Meltdown.

* 1. **Flush and Reload**For proving the correctness, the Flush and Reload attack is run once with a secret key of 95. From Figure 4.1, it can be seen that the attack reliably leaks the data.   
     To quantify the performance gain, 4 trials of the attack are run and the success rates are tabulated in Table 4.1 and visualized in Figure 4.2.



**Figure 4. 1 Snapshot of Flush and Reload**

|  |  |
| --- | --- |
| Trial Number | Success Rate (%) |
| 1 | 68.5 |
| 2 | 61.7 |
| 3 | 65.7 |
| 4 | 70.7 |

**Table 4. 1 Performance of Flush and Reload**

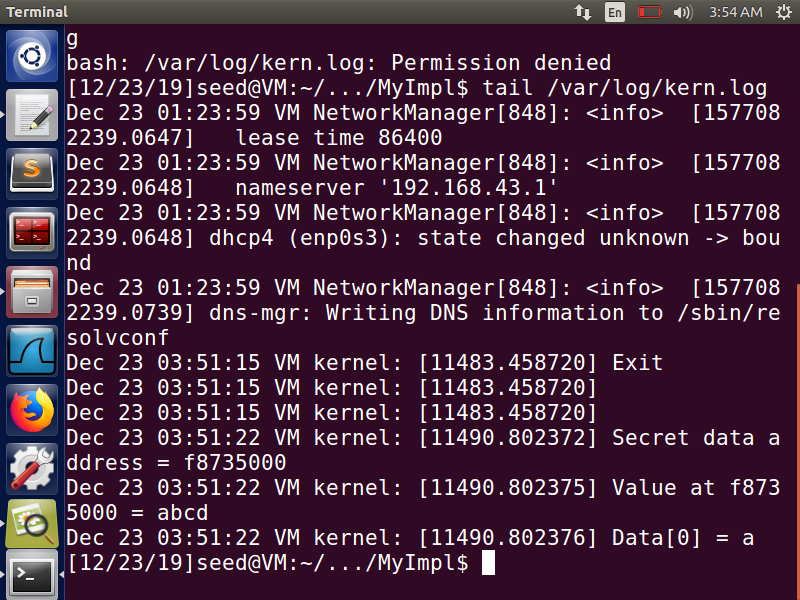
**Figure 4. 2 Visualization of Flush and Reload performance**

From the figures, it is observed that the attack is highly effective, with success rates well above 60%, averaging at around 67%

* 1. **Meltdown**

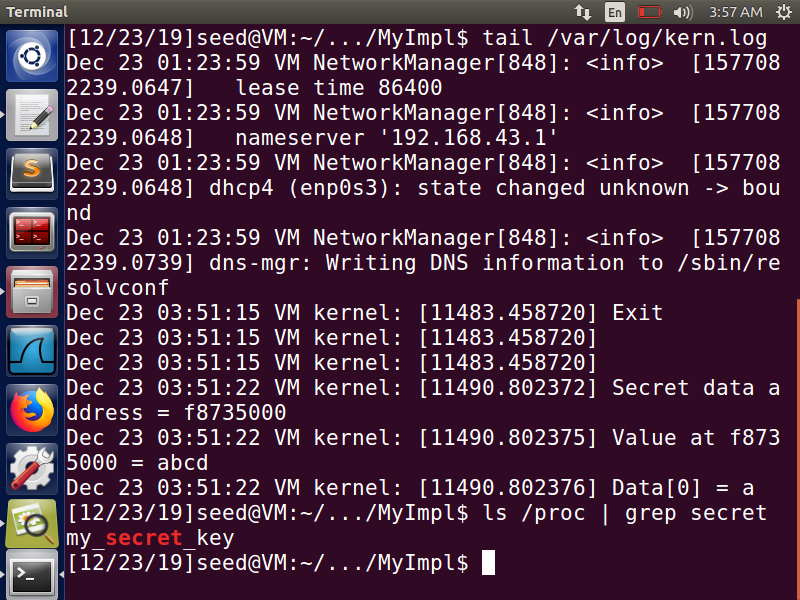
The attack is run on an Intel Haswell based 32-bit Virtual Machine running Ubuntu.

The kernel module is inserted and the address and data at the secret location are logged. The string “abcd” is stored at a random location. To verify if kernel memory has been allocated, the module logs the base data address, the base data, and the string. The log can then be checked for the data as shown in Figure 4.3.



**Figure 4. 3 Log generated by the kernel module**

Further, to ensure that the data has been created and stored in the memory, the procfs is checked for the secret key file created by the kernel module as shown in Figure 4.4.



**Figure 4. 4 Verifying the creation of procfs entry**

The attack is then run targeting the base address of the secret string. The success rates for 1000 attempts in 4 experiments are tabulated in Table 4.2 and visualized in Figure 4.5.

|  |  |
| --- | --- |
| Trial Number | Success Rate (%) |
| 1 | |  | | --- | | 60.1 | |
| 2 | |  | | --- | | 57.8 | |
| 3 | |  | | --- | | 60.7 | |
| 4 | |  | | --- | | 76.7 | |

**Table 4. 2 Performance of Meltdown**

**Figure 4. 5 Visualization of Meltdown performance**

The attack successfully leaks data 64% of the cases on average, hence completely breaking down the security guarantees of the CPU.

1. **Mitigations**

It should be noted that only a redesign of the CPU architecture and new firmware is a permanent fix to the root cause of the problem, i.e., processor leaking out the value of protected memory locations via the cache. Apart from this, several mitigations have been proposed as a stop-gap measures to prevent the exploitation of meltdown on vulnerable CPUs.

* **KPTI: Kernel Page Table Isolation**  
  A major feature of Linux that enabled the meltdown attack was the way virtual address spaces were allocated by the kernel. Each user-space process had the memory of the entire system mapped out in its virtual address space.   
  With KPTI [11], the kernel maintains two separate page tables, one for kernel space processes that have full access to the memory, and another for user space processes, which have access only to their own memory, and limited kernel space addresses for easy servicing of interrupts, exception handling, etc.
* **Vendor Supplied Microcode Updates**  
  Microcode is very low-level code that controls the operation of the CPU and is permanently embedded in the hardware.  
  Intel [16] and ARM [17], have been rolling out microcode updates to patch the vulnerability via the Windows Update, and the Linux Kernel update distribution mechanisms.   
  However, in some cases, these updates have involved a reduction in performance of upto 30%.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

1. **Mitigation Analysis**

As before, the procedure of creating an entry at a known kernel memory location with a kernel module and attacking it from a user space process is repeated.

The success rates for four trials of the attack are tabulated in Table 6.1, and visualized in Figure 6.1. For comparison, a comparison of success rates before and after the mitigation are given in Figure 6.2.

|  |  |
| --- | --- |
| Trial Number | Success Rate (%) |
| 1 | 0 |
| 2 | 0.1 |
| 3 | 0 |
| 4 | 0.1 |

**Table 6. 1 Performance after mitigation**

**Figure 6. 1 Visualization of performance after mitigation**

**Figure 6. 2 Comparison of performance before and after mitigation**

It is observed that the success rate becomes negligible, and doesn’t exceed 0.1%.

Further, this analysis assumes that the attacker knows the location of the secret data which is highly unlikely in real world scenarios. While it is possible to dump the data of the kernel page table as shown in [1], it also exponentially increases the effort required for searching for the required application mappings.

1. **Conclusion**

A practical approach to exploiting meltdown on x86 based systems, specifically within a virtual machine environment has been presented. It is observed that the CPU leaks the privileged data through the cache which exposes a timing channel that is exploited to steal the data. The success rate of the attack is significantly high, with average success rate of 64%. The proposed mitigations are applied and are found to be extremely effective, with the attack failing almost 100% of the times, with success rate almost zero.

It is noted that the system has included a 4th generation Intel Haswell i5 CPU, which is relatively outdated. From the 8th generation coffee lake chips, Intel has altered the hardware design which makes it trickier to exploit this flaw, with lower success rates reported.

However, for a complete in-principle mitigation of the flaw, the hardware architecture of the CPU, including the wiring of cache and the boundary check control logic will need to be revisited by the manufacturers.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**REFERENCES**

[1] Yarom, Falkner; Flush+Reload: a High Resolution, Low Noise, L3 cache side channel attack. SEC'14 Proceedings of the 23rd USENIX conference on Security Symposium, Pages 719-732

[2] HUND, R., WILLEMS, C.,ANDHOLZ, T. Practical Timing Side Channel Attacks against Kernel Space ASLR. In S&P (2013)

[3] Taylor Hornby, *Side-channel attacks on everyday applications: distinguishing inputs with FLUSH+RELOAD*, <https://www.blackhat.com/docs/us-16/materials/us-16-Homby-Side-Channel-Attacks-On-Everyday-Applications-wp.pdf>.

[4] Lipp et. al; Meltdown: Reading Kernel Memory from User Space. Proceedings of the 27th USENIX conference on Security Symposium, 2018

[5] Kocher et. al, Spectre Attacks: Exploiting Speculative Execution, proceedings of the 40th IEEE Symposium on Security and Privacy, 2019

[6] TOMASULO, R. M. An efficient algorithm for exploiting multiple arithmetic units. IBM Journal of research and Development, 1967

[7] INTEL. IA-PC HPET Specification Rev 1.0a 1 IA-PC HPET (High Precision Event Timers) Specification,  
<https://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/software-developers-hpet-spec-1-0a.pdf>

[8] UEFI, Advanced Configuration and Power Interface Specification,  
<https://uefi.org/sites/default/files/resources/ACPI_6_2.pdf>

[9] AMD, Game Timing and Multicore Processors,  
<http://developer.amd.com/wordpress/media/2013/03/Game_Timing_Multicore_Processors.pdf>

[10] FAULKNER, GOMES, The Process File System and Process Model in UNIX System V, Proceedings of the USENIX conference, 1991.

[11] HANSEN, D. [PATCH 00/23] KAISER: unmap most of the kernel from user space page tables, <https://lkml.org/lkml/2017/10/31/884>, Oct 2017.

[12] HANSEN, D. [v2] KAISER: unmap most of the kernel from user space page tables, <https://lkml.org/lkml/2017/11/8/752>, Nov 2017.

[13] HANSEN, D. [v3] KAISER: unmap most of the kernel from user space page tables, <https://lkml.org/lkml/2017/11/10/433>, Nov 2017.

[14] HANSEN, D. [v4] KAISER: unmap most of the kernel from user space page tables, https://lkml.org/lkml/2017/11/22/956, Nov 2017.

[15] LWN. The current state of kernel page-table isolation, https://lwn.net/SubscriberLink/741878/eb6c9d3913d7cb2b/, Dec. 2017.

[16] INTEL. Intel analysis of speculative execution side channels,  
[https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf](https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf%20)  , Jan. 2018.

[17] ARM. Software implications for Spectre/ Meltdown on Arm cores, Version 1.3,  
<https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Security%20update%2018%20June%2018/Software_Overview_for_Arm_Cores_v1.3.pdf?revision=1ddb05ba-5b68-468a-b837-a480a94e1c8e>,  
Jun. 2018.

[18] JOHNSON, K. KVA Shadow: Mitigating Meltdown on Windows, <https://blogs.technet.microsoft.com/srd/2018/03/23/kva-shadow-mitigating-meltdown-on-windows/>, March 2018.

[19] Zhichao Hua , Dong Du , Yubin Xia , Haibo Chen , Binyu Zang, EPTI: efficient defence against meltdown attack for unpatched VMs, Proceedings of the 2018 USENIX Conference, July 11-13, 2018, Boston, MA, USA

[20] A. Prout et al., “Measuring the impact of Spectre and Meltdown,” in High Perf. Extreme Computing Conference. IEEE, 2018, pp. 1–5.

[21] PHORONIX. Linux 4.12 To Enable KASLR By Default, <https://www.phoronix.com/scan.php?page=news_item&px=KASLR-Default-Linux-4.122017>

[22] HENNESSY, J. L., AND PATTERSON, D. A. Computer Architecture: A Quantitative Approach, 6 ed. Morgan Kaufmann, 2017.

[23] Greg Kroah-Hartman, Linux 4.15.1  
 https://cdn.kernel.org/pub/linux/kernel/v4.x/ChangeLog-4.15.1

[24] Linux Kernel Foundation. Memory Allocation Guide  
<https://www.kernel.org/doc/html/latest/core-api/memory-allocation.html>