Practical Implementation and Security Analysis of Meltdown on x86 Based Virtual Machines

**This color means that task is complete**

1. **Intro – flush + reload, spectre, meltdown**
2. **Motivation – security implications, practical implementation**
3. Methodology – **measuring cache time (struct timepsec, timeval, rdtscp)**, **flush + reload – thresholding vs min.time approach**, **Kernel Modules, vmalloc and kmalloc, encouraging data to remain in cache (procfs, prefetch, etc.), trapping exceptions in C, delaying commit – keeping ALU busy.**
4. **Results**
5. **Mitigation – KPTI, microcode patches, linux kernel versions**
6. **Mitigation Analysis**
7. **Conclusion**

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

**Introduction**

Side channel attacks utilize information produced as a by-product of the operation of physical devices. Traditional attacks involve measuring electromagnetic radiation, power consumption, or using acoustics to record data such as key strokes.

With the discovery of the Flush+Reload attack [2], interest in timing analysis as a side channel attack has increased substantially. It was clearly demonstrated that given a sufficiently accurate timing mechanism, it was possible to leak the recently used data with the processor cache serving as a side channel. The approaches in [5] and [6] outline practical approaches to exploiting this attack.

Extending the idea of Flush+Reload, Meltdown [1] and Spectre [23] 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 [3] 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.

In this paper, we present a practical approach to implementing Meltdown on a 32-bit Linux virtual machine (VM). We analyze the various techniques for implementing the building blocks of the attack, i.e., timing, memory allocation, enforcing data locality, and exception handling. We demonstrate how a user-space process can leak data from the virtualized kernel memory. We then present our results, following which mitigation measures are discussed. We finally analyze the efficacy of the mitigation and present our results.

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

**Motivation**

The most fundamental security guarantee in a modern OS 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. By allowing a user-space process full blown access to the kernel memory, 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.

While a lot of work has been done to demonstrate the potential of meltdown, implementation strategies for meltdown have not seen much discussion. Our goal is to study the strategies for practical implementation and test them on a commonly used 4th generation Intel Haswell CPU.

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

**Methodology**

We start by stealing values local to the process using cache side channel attack. The possible methods for timing memory accesses are explored and optimal timing technique is arrived at. We then look at implementing flush reload and check its efficacy. The ideas of kernel modules, makefiles and kernel memory allocation are then discussed. Finally, we demonstrate the meltdown attack and look at improving its efficacy. We conclude by analyzing the effectiveness of the security measures introduced to mitigate the vulnerability.

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

**Measuring Memory Access Time**

We need very high level of accuracy 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 our purposes, we require an API that satisfies 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.

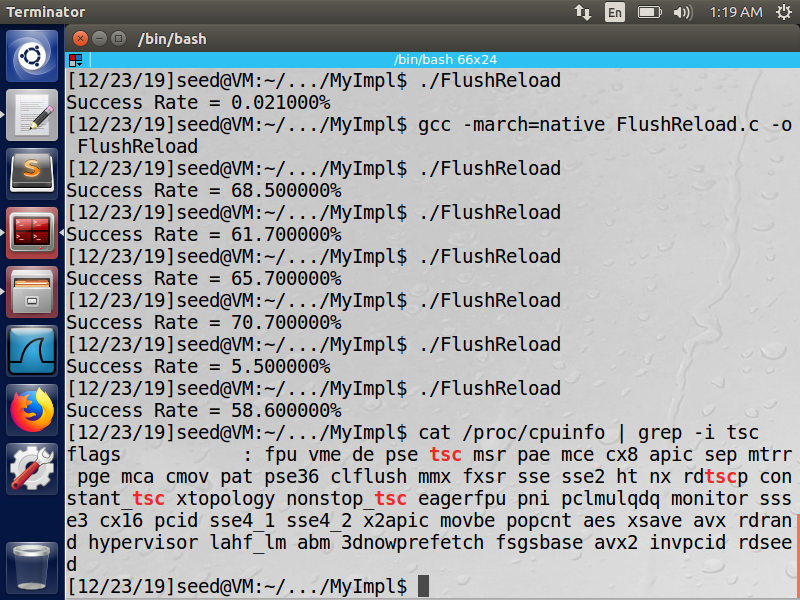
Calculation of wall clock time, i.e., the actual time is very expensive, as it requires conversion of counter values into wall clock format and synchronization of the CPU with known accurate times. A key observation here is that the wall clock time is not required, as we are only interested in finding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

The clock counters available in Linux are:

1. **HPET: High Precision Event Timer**. Jointly developed by Microsoft and Intel and incorporated in chipsets since 2005. It gives a precision of around 100ns.
2. **Acpi\_pm clock:** 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.
3. **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.   
   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, we choose the TSC as the clock source for maximum accuracy and high speed of response.

**!!IMAGE : $cat /proc/cpuinfo | grep -i tsc**



We first check for TSC support by probing the processor information file “cpuinfo” located in the procfs [17].

We make use of the RDTSCP C API 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.

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

**Flush + Reload**

We create an oracle buffer of 256 entries, each entry is of size 4KiB. This is done because the page size is 4KiB and may cause multiple entries to be cached as a result of cache locality.

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, we have obtained the index of the cached entry which is 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**

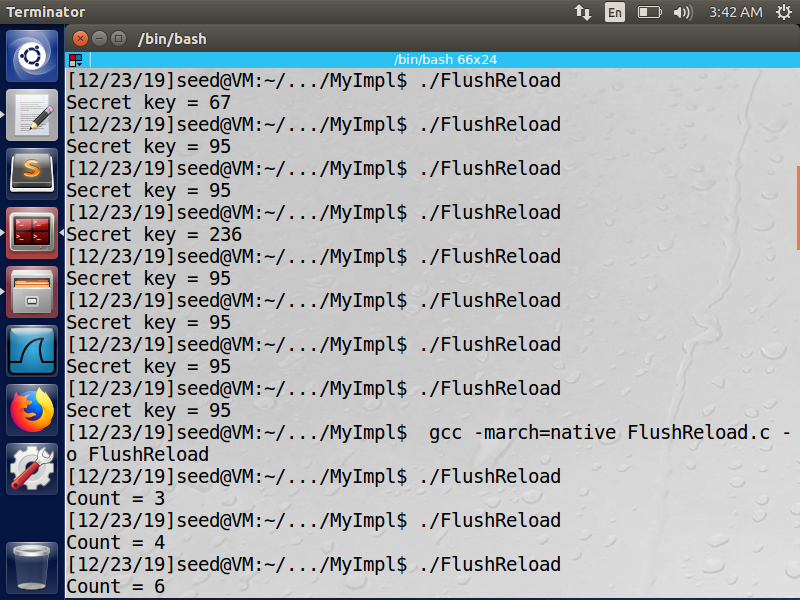
Time access to each element of oracle

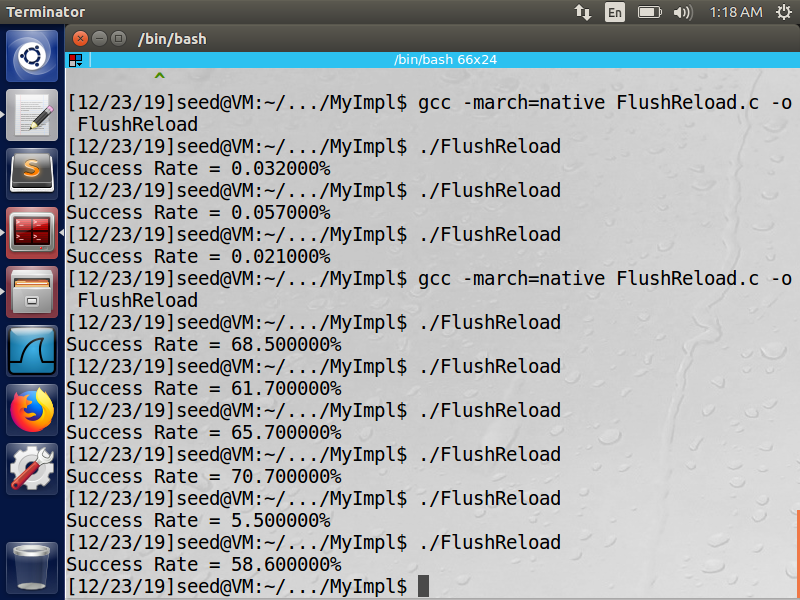
**Attacker**

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

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

**!!IMAGE: FLUSH + RELOAD statistics with 1000 iterations**



(!!!!BETTER TO TABULATE)

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

**Makefiles and Kernel Modules**

Make file is special file that contains objects and its target dependencies. Make is Unix utility that used for execution of makefile.It keeps track of the files that are changed (or updated).

If we do not use makefiles we will need to recompile entire project for changes made to even a single file. This is extremely inefficient as it involves recompiling hundreds of files.

In contrast, a makefile based approach only relies on the information from make. Only the changed file is recompiled and integrated into the project, which saves a great deal of time.

Kernel module is program that can be loaded or unloaded dynamically from kernel. Module can be loaded on the fly without recompiling it.

Without kernel module, adding new features to the kernel requires recompilation.

Further, it also takes a long time to compile as kernel size is large. Since kernel modules can be dynamically loaded and unloaded based on the requirement, memory can be used efficiently by the kernel.

We are using kernel module to load the secret into kernel space memory, which we then attack from user space programs.

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

**Allocation of Memory from the Kernel**

In our approach, memory allocated and owned by the kernel in kernel space is attacked. Two popularly used APIs for allocation of contiguous memory are **kmalloc()** and **vmalloc()**.

Kmalloc() is used when we require the data to be physically contiguous as well as virtually contiguous. It is generally used for latency sensitive applications such as interrupt handlers. However, it fails in case of insufficient contiguous physical memory.

Vmalloc() allocates virtually contiguous memory, which is not physically contiguous. The memory blocks are allocated from locations spread across the disk. This enables non-faulting allocation even when physically contiguous space is unavailable. However, it increases the access latency, and is slower to return compared to kmalloc.

We make use of kmalloc() since the attack is time sensitive.

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

**Encouraging Data to remain in Cache**

Keeping the data hot in the cache is essential to the success of the attack. We need to avoid the conditions where it gets overwritten and flushed out of the cache. The cache state itself is private to the CPU and is generally treated as inaccessible to the programmer. We make use of various techniques which increase the likelihood of data remaining in the cache prior to executing the attack.

1. Prefetching data  
   We make use of the non-faulting prefetch functions provided by x86 to prefetch the secret data. We prefetch 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. We create a procfs entry of the secret key from the kernel module. This is then opened by the attacker to cache the data.

Apart from this, we also try to minimize the time between caching the data and running the attack. This avoids instances of data getting flushed out as a result of context switches.

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

**Trapping Exceptions in C**

In our Program we need to trap memory access violation error. C does not have any inbuilt exception handling mechanism. Hence, we write our own error handling utility.

Error handling utility is based on Signal concept. Interrupt is raised by the kernel on receipt of error signal from the CPU. We make use of the signal library to manipulate our error buffer.

We initialize a checkpoint buffer to initial state, i.e., no errors generated. We also define error handler function, where we reset the checkpoint buffer to initial state.

SIGSEGV is the error thrown by the CPU on encountering invalid memory access. The handler function is registered with SIGSEGV.

We check for exception by sampling the check point buffer. If there is no error, we proceed with the attack. Once the attack has occurred, the error is detected, and our error handler is called, which resets the checkpoint state, to prevent fatal error.

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

**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.

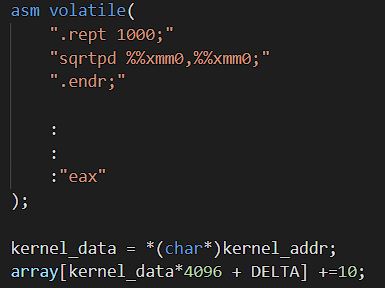
1. Issue: The instruction is issued and brought into the CPU’s re-order buffer.
2. Execute: Checks if it is safe to execute the instruction, and runs the required functional units to execute the task.
3. Write-Result: The result is written onto the data bus and sent to the commit buffer
4. Commit: The result is written into memory

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, we can execute the attacking instructions out of order and steal the data from the cache. To this end, we need to delay the commit of the segmentation fault as much as possible in order to improve the chances of the attack.

We insert computationally intensive arithmetic instructions, such as square root, log calculation, etc. prior to the illegal memory access. This introduces a delay equal to the execution time of the ALU.

**!!!PIC OF ASSEMBLY CODE**



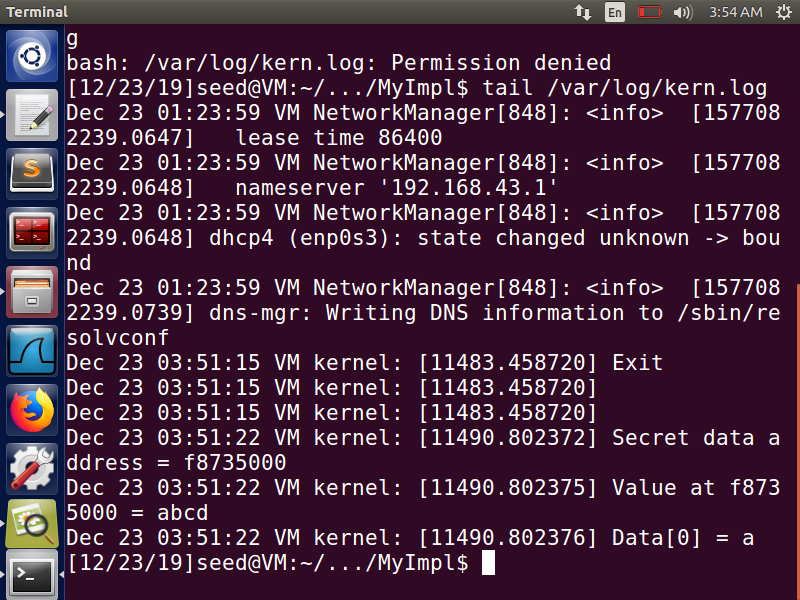
**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Results**

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

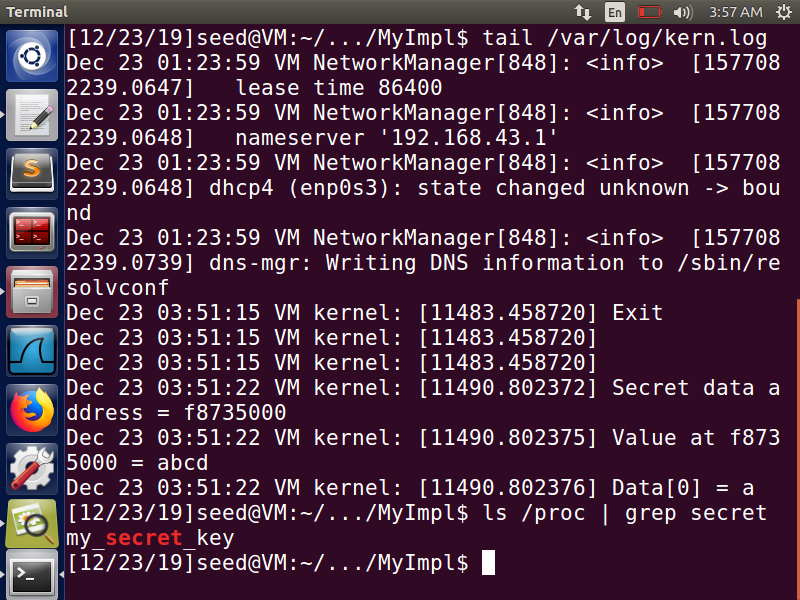
The kernel module is inserted and the address and data at the secret location are logged. The string “abcd” is stored at a random location.

**!!! Kernel-Module figure**



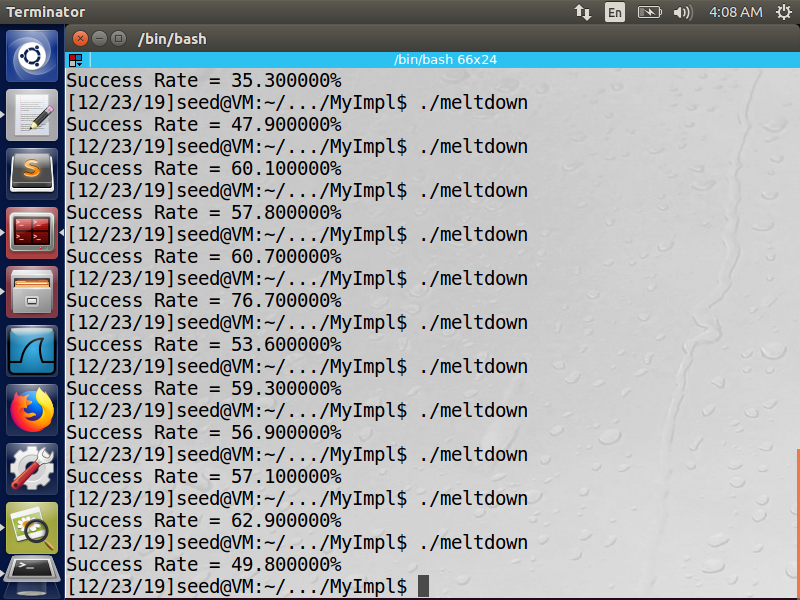
We further ensure that the data has been created and stored in the memory by checking the procfs for the secret key file.

**!!!Procfs Entry**



We then run the attack targeting the base address of the secret string. The success rates for 1000 attempts in 4 experiments are shown in the figure.

**!!! RESULTS**



The attack successfully leaks data in more than 50% of the cases on average, hence completely breaking down the security guarantees of the CPU.

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

**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.

1. **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, 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.
2. **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 and ARM 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%.

We next look at the efficacy of these patches by retrying our attack.

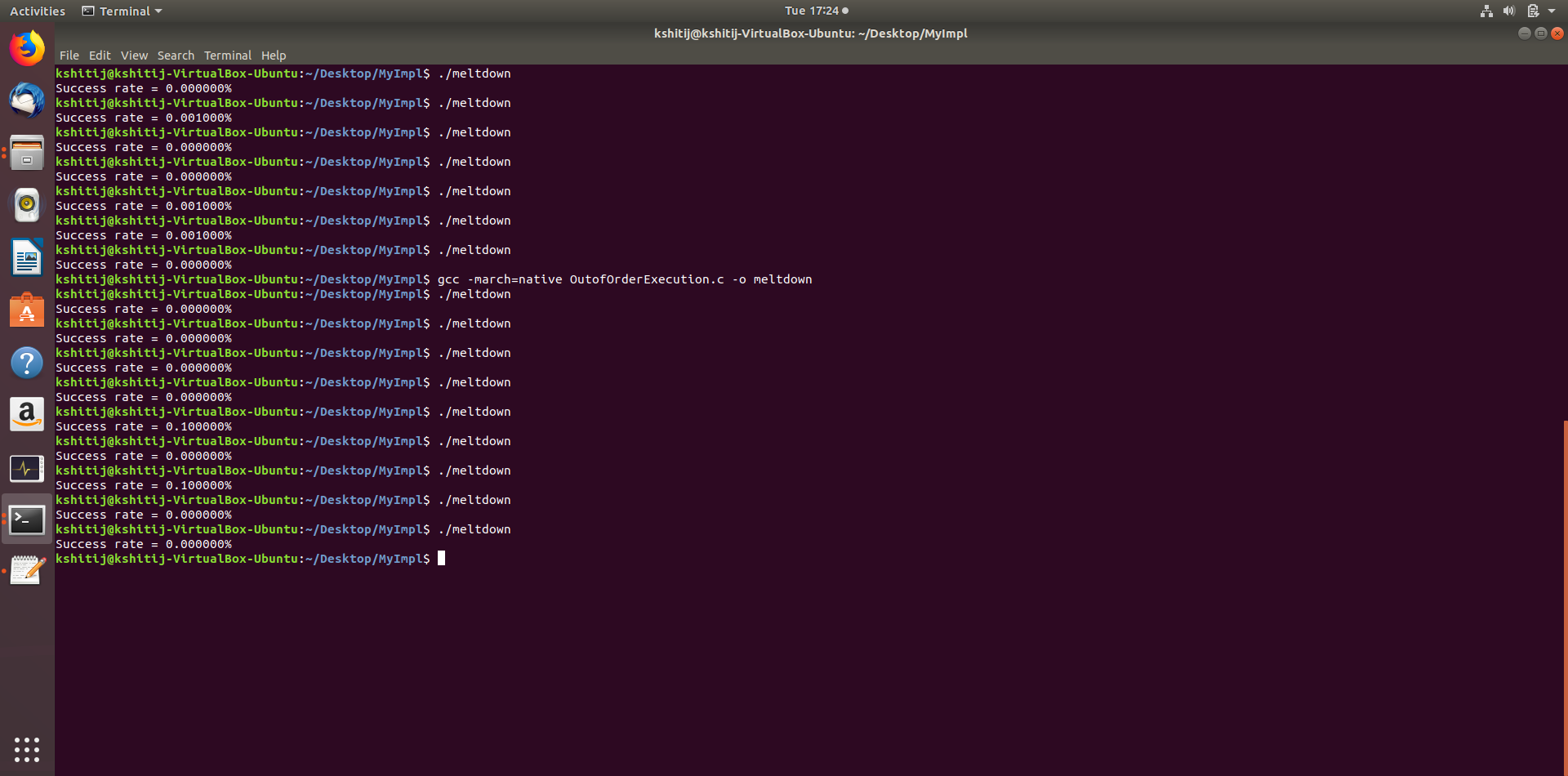
**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Mitigation Analysis**

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

As can be seen in the table, we get very low success rates, ranging from 0 to 0.1%.

**!!!MITIGATION IMAGE**



Further, this analysis assumes that the attacker knows the location of the secret data which is higly 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.

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

**Conclusion**

We have presented a practical approach to exploiting meltdown on x86 based systems, specifically within a virtual machine environment. We observe that the CPU leaks the privileged data through the cache which exposes a timing channel that is exploited to steal the data.

We note that our analysis 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] Lipp et. al; Meltdown: Reading Kernel Memory from User Space. Proceedings of the 27th USENIX conference on Security Symposium, 2018

[2] 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

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

[4] 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.

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

[6] 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>.

[7] 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 Jan 2018

[8] 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

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

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

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

[12] 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.

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

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

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

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

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

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

[19] 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>

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

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

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

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