Spectre and Meltdown Analysis

Jared Dyreson
Engineering & Computer Science
California State University Fullerton
Fullerton, United States
jareddyreson@csu.fullerton.edu

***Abstract*—This document is an overview of the Spectre and Meltdown hardware vulnerabilities found in modern CPUs through the advent of branch prediction. Instructions can be loaded and executed ahead of the current instruction based on previous instructions. If these predictions are incorrect, the CPU will roll back to a save state it set before attempting this branch execution. Speculative analysis however, holds a fatal flaw as it does not abide by the same regulations as normal instruction execution does. This leads to potential memory leaks during the execution of specially crafted payloads. Spectre attacks specifically attempt to train the CPU allow certain variables to not be checked during runtime, allowing for bypassing of bound checks. Meltdown exploits on the other hand, will try to execute instructions out of order and try to bypass memory protections on speculatively executed instructions. Other avenues of exploitation can reside in return oriented programming, which attempts to mark portions of the stack as executable, leading to arbitrary code execution.**

I. Introduction

Methods of exploiting systems using "many physical effects such as power consumption, electromagnetic radiation, or acoustic noise" [1, p. 1] have been around for a long time. These attacks generally prey on oversights in architecture design of the hardware itself. In recent years, there has been a shift from exploiting hardware to using software to leverage attacks. Common techniques used before include buffer overflows, which would override portions of the stack to achieve arbitrary code execution. These avenues have been hardened by stack canaries and compiler warnings when using unsafe system calls such as strcpy and strcat. More sophisticated attacks have been exploiting components in the Linux Kernel, such as the JIT (just-in-time) compiler and the eBPF (extended Berkeley Packet Filter). Other microarchitectural components on the system can also be used in attacks, such as the CPU cache and cache lines. The aforementioned vectors are utilized in Spectre and Meltdown exploits, in conjunction with other side-channel buffers.

II. CPU Cache

In performing calculations, it is important for the CPU to have an external memory bank to retrieve instructions and data that it wishes to reuse. Data that is no longer needed is removed using several different algorithms, such as LRU (least recently used) and FIFO (First In First Out). Implementation of these replacement algorithms are done directly in the hardware and vary in their effectiveness. The caches reside close to or inside the CPU itself, leading to quick and easy access. Since data is stored here, this is one possible place where data can be leaked, as it does not need to be fetched from main memory. Attackers can "measure the time it takes to perform a memory read", and in doing so can pinpoint where data is stored [1, p. 4]. Then, with this knowledge, they can craft a payload that can influence the state of the cache and allow for the reading of a specific piece of data. Speculative code execution can allow for unprivileged processes to read these segments of data, as these restrictions are not enforced during this time.

III. Exploitation Techniques

There are two main instances where Spectre can be used; conditional and unconditional branches. Both methodologies differ in their initial and intermediate steps but still achieve the same result.

A. Conditional Branches

Conditional branches are exploited by mis-training the CPU into thinking some series of instructions is safe. In the white paper "Spectre Attacks: Exploiting Speculative Execution", the approach used was having a variable be cleared for use in indexing an array.

```
if(x < array1_size)
    y = array2[array1[x] * 4096]
```

Here, the attacker is setting up a vulnerable state for the cache and in speculative execution. It is important to note that "changes made to the cache state are not reverted" [1, p. 2]. The CPU first does a conditional check if the variable is less than the size of the array. Since the size is not in the cache, the CPU needs to fetch the address from main memory. While waiting for the variable 'x' to be retrieved, the CPU will now speculate what the address should be for the indexing of array2. As noted above, array2 has not been loaded into the cache and needs to be read into the cache. In the process of retrieving the second bit of information, the original condition may be deemed as false, telling the CPU that this branch is incorrect. However, since the CPU fetched for the block of memory, it now resides in the cache. A side-channel attack like Flush+Reload, which "detects whether a specific memory location is cached" can be deployed to lead this piece of data [2, p. 5]. The authors of this paper call these instructions *transient*, as they revert the CPU back to its original state but leave trace elements [1].

B. Unconditional Branches

Variants that utilize unconditional branches attempt to lead the CPU down a path that will expose certain regions of memory. Most of these paths would not "occur during legitimate program execution" [1, p. 8] and do not require the leveraging of branch instructions. ROP gadgets littered throughout the victims memory can be used as springboards for luring the CPU. Before continuing, it is important to define what exactly a ROP gadget is. Gadgets can be classified as any sequence of instructions that have a following return instruction. Since most programs are dynamically linked, there is also plenty of space for functions and gadgets to reside that would allow for the offloading of sensitive data. Attacks generally use two registers to capture this leaked data, one containing an address to the data targeted and another to offload that has already been loaded in. These are particulary of use if the attacker knows which registers will be used.

```
; Code written by Jared Dyreson
mov rdi,printf_format
mov rsi,0x404000
xor rax,rax
call printf
```

In this example, we know that rsi will be loaded with the hex value of 0x404000. Theoretically, an attacker can use misdirection to eventually reveal the value of rsi and export it via side channels.

IV. JIT and eBPF

The JIT compiler bridges the gap for interpreted languages such as Java and C#, which can theoretically be run on any CPU architecture. Machine instructions are generated during runtime and there is no assembling/linking stage. More traditional compilers, such as AOT (ahead of time), do all of this during compilation for the architecture of the host machine. Generally, code produced by the JIT compiler can come in a variety of "temperatures" and each level is associated with how optimized the code is. These are; cold, warm, hot, very hot and scorching. The compiler can gauge how often a set of instructions is being called by sampling the stack frame. Also, functions can be deescalated to "cold, to further improve startup time" [3]. The eBPF can be seen as an extension of JIT compilers, as they allow dynamic loading of sandbox environments in kernel space. These environments can be used "without changing kernel source code or loading kernel modules" [4]. In this intermediate layer resides both a verifier and JIT compiler. To ensure the process has the correct permissions to run bytecode, the eBPF checks the privileges of the process. Also, the verifier attempts to spot programs that hang in an infinite loop and will block them from executing.

V. JIT Mitigations

Attacks using Javascript have been used as PoC (proof of concept), which can lead to concerns of code execution in the browser utilizing Spectre and Meltdown. Users on Reddit had made patches for the jailbroken iDevices in late 2017 before Mozilla and Apple had made strides to mitigate the issue. SharedArrayBuffer objects had previously been used to shared data amongst threads however, in the advent of Spectre, was disabled entirely. These rudimentary patches only allowed the Javascript interpreter to use ArrayBuffers, which mitigated the risk of exposing sensitive data. In 2020, it was decided to re-enable shared memory and the changelog included that "postMessage() will no longer throw for SharedArrayBuffer objects and shared memory across threads will be available" [5]. However, this does not entirely fix the issue, as cross thread memory sharing could still result in data leaks. One avenue that could possibly patch this issue is by interleaving the data across multiple cores. It was noted that "branch prediction is not influenced by operations on other cores" [1, p. 9] and thus spreading the data out could result in obfuscation. Performance would take a hit, however, as the CPU would need to spend extra cycles to pinpoint where in each cache per each CPU core.

VI. Meltdown

Akin to Spectre, Meltdown exploits a microarchitectural vulnerability that executes instructions out-of-order. When the CPU is idling after finishing an instruction that requires monitoring, it uses that time to schedule future operations that will need to be executed. Instructions here can be run "in parallel as long as their results follow the architectural definition" [2, p. 2]. Unfortunately, instructions that are loaded ahead of time do not have to abide by the same heavy restrictions of memory retrieval. This will allow unprivileged process to read memory regions that were not previously granted, such as kernel space memory. Operations can be conducted even "before the processor has committed the results of all prior instructions" [2, p. 3]. These types of bugs are called Read after Write (RAW) and Write after Read (WAR), as data changes do not have enough time to propagate throughout code execution. Similar to how Spectre leaves the cache in an impartial state, data can be leaked through the means of side channels.

A. Address Space Mapping

To separate each process, the CPU will assign a unique address space to it, similar to how namespaces in programming languages work. This helps relive the processor from managing a monolithic structure by partitioning it into even blocks called pages. Each of these pages are subdivided into a user and kernel space. The latter space is needed for when "the CPU is running in privileged mode", such as accessing restricted memory regions [2, p. 4]. Most operating systems will also map the entire contents of the kernel into each user process. In needing this space for data manipulation and other various operations, the entire physical memory is mapped into kernel space and typically are placed fixed addresses. To help mitigate this, kernel address space layout randomization (KASLR) was introduced, which changed where kernel data structures were loaded upon boot.

B. Possible Attack Ventures

To conduct an attack, first the adversary would lead the CPU into loading in desired memory that it does not have access to. Before the CPU can deny such request, it is speculating what the outcome will be if it is granted. In doing this, the address for the data is loaded into a register and will access a certain cache line based on the content retrieved. Using side-channel attacks, one could find data associated with the register and can effectively map all of kernel memory by repeating this process. Since the kernel maps all of physical memory in it's own address space, this can allow rouge processes to access sensitive data in other processes. This can be seen as a form of privilege escalation and works without assuming control of the kernel itself. In this state, the attacker still needs to be wary of KASLR, as the memory being siphoned is still randomized. However, "the randomization is limited to 40 bit(s)" which means one can "spray" the KASLR space and attempt to read a value from a tested address. If successful, all of physical memory can be read as usual. It should be noted that KASLR makes the Meltdown vulnerability less reliable. Theoretically, one could increase the rate of success if a kernel bug is discovered that compromises KASLR This could come in the form of leaking key addresses to kernel data structures or disabling it entirely. One iOS/MacOS bug resided in XNU's set_dp_control_port() function, which left a dangling pointer to a Mach port. With this information, one "guess the page on which the kernel task port lives" [6], defeating KASLR.

VII. Operating System Mitigations

Most operating systems now have attempted to implement safety mechanisms in order for user and kernel spaces to act in isolation. As described above, each process is split down the middle to have unprivileged and privileged code execution. Before, both of these address spaces used the same lookup tables and could coexist peacefully. However, since shared memory between the kernel and user space is now dangerous, new methods of combating this have been created. Kernel address isolation to have side-channels efficiently removed (KAISER), is one such implementation. It designates two separate tables, where one is "for the user program that contains only locations for user memory" and the other is for the operating system [7, p. 3]. In this, we attempt to limit the surface area of exposed kernel memory, lowering the chances of data being leaked. However, the cost for doing such is that the operating system needs to switch "between user and kernel mode becomes more expensive" [7, p. 3] Other means of mitigations have been proposed, such as adding return trampolines. This is when "dummy branch that has a return statement before the jump and call instruction to prevent branch target injection" [8, p. 3]. This can help prevent conditional branch exploitation, as it prevents the CPU from easily being lead down a wrong path.

A. Performance after Mitigations

The new patches for both Spectre and Meltdown have seen performance decreases due to how they are implemented. IO (Input/Output) operations particularly have seen a significant hit. One mitigation introduced was to constantly clear the "memory lookup table cache" when accessing files on disk, which is an expensive operation. This attempts to not allow any information to reside in the cache, therefore minimizing the amount of data that can be leaked. While in theory it could be seen as a positive, the trade-offs have shown a "10-30% (decrease)" in database applications [8, p. 4]

B. IO Patterns Worst Affected

Some IO operations will suffer more than others with these new patches unfortunately. Programs that utilize "meta-data operations" are most affected, as these include employing file open/close operations, permissions and file existence. One possible way to alleviate this issue is by simply being more mindful of how many IO operations are needed. Any unnecessary file access should be completely avoided, however this is easier said than done. Companies like Ellexus have created software (Mistral) that can help locate where "bottlenecks and applications with bad I/O" [7, p. 5]. These tools are helpful for enterprises that deal with millions of IO operations per second in their workflow, such as gas exploration or genome mapping. Other pieces of software in their suite can help identify "common issues such an application that has a high ratio of file opens to writes" [7, p. 5]. Patterns derived from these tools can help software engineers to write better code to help alleviate this issue.

VIII. Summary

To conclude, Spectre and Meltdown are powerful hardware vulnerabilities present in most consumer and enterprise CPUs. Attacks are generally difficult to formulate and are highly specific in what data they wish to extract. These leaks are also not easily traceable, as they do not leave behind any evidence. However, the severity of these holes should not be understated and we are still feeling the effects years later. To properly and fully handle this issue, major chip manufactures need to add more security measures in speculative execution. This could be in the form of creating new instructions to "provide a barrier against speculative execution" [8, p. 3]. There is no such instruction present in Intel's ISA and speculative execution will occur regardless. Other types of proposals such as "return trampolines" have been proposed that can be retroactively applied to all affected chipsets. Patches in operating systems such as KAISER provide a temporary fix in separating user and kernel processes but add a performance penalty. User applications that involve several IO operations are the most affected due to frequent context switching and cache clearing. Some methodologies of working around this bottleneck have been proven to help mitigate the performance hit.

References

[1] P. Kocher, J. Horn, A. Fogh, *et al.*, "Spectre attacks: Exploiting speculative execution," pp. 1–19, 2019. DOI: 10.1109/SP.2019.00002.

[2] M. Lipp, M. Schwarz, D. Gruss, *et al.*, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD: USENIX Association, Aug. 2018, pp. 973–990, ISBN: 978-1-939133-04-5. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

[3] A. S. Gillis, "What is a just-in-time (jit) compiler and how does it work?" *TheServerSide.com*, Dec. 2019. [Online]. Available: https://www.theserverside.com/definition/just-in-time-compiler-JIT.

[4] "What is ebpf? an introduction and deep dive into the ebpf technology," [Online]. Available: https://ebpf.io/what-is-ebpf.

[5] Mozilla, "Sharedarraybuffer - javascript: Mdn," *JavaScript — MDN*, May 2021. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArrayBuffer.

[6] Tim, *A survey of recent iOS kernel exploits*, Jan. 1970. [Online]. Available: https://googleprojectzero.blogspot.com/2020/06/a-survey-of-recent-ios-kernel-exploits.html.

[7] R. Francis, *How the Meltdown and Spectre bugs work and what you can do to prevent a performance plummet*, 2018.

[8] A. Efe and M. O. Güngör, *The impact of meltdown and spectre attacks*, *International Journal of Multidisciplinary Studies and Innovative Technologies*, vol. 3, pp. 38–43, 2019, ISSN: 2602-4888.