Speculating on Speculative Execution

Assessing the Risk of Simultaneous Hyperthreading

Jefferson Dinerman
Secondary School Student and Independent Researcher
Alexandria, VA | United States of America

***Abstract*—Threat actors continue to design exploits that specifically target physical weaknesses in processor hardware rather than more traditional software vulnerabilities. The now infamous attacks, Spector and Meltdown, ushered in a new era of hardware-based security vulnerabilities that have caused some experts to question whether the potential cybersecurity risks associated with simultaneous multithreading (SMT), also known as hyperthreading (HT), are potent enough to outweigh its computational advantages. A small pool of researchers now touts the need to disable SMT completely. However, this appears to be an extreme reaction; while a more security focused environment might be inclined to disable SMT, environments with a greater level of risk tolerance that may need the performance advantages offered by SMT to facilitate business operations, should not disable it by default and instead evaluate software application-based patch mitigations. This paper provides insights that can help make informed decisions when determining the suitability of SMT by exploring key processes related to multithreading, reviewing the most common exploits, and describing why Spectre and Meltdown do not necessarily warrant disabling HT.**

Keywords*—*Speculative execution; hyperthreading; Spectre; meltdown; simultaneous multithreading

I. Introduction

Although the news fervor regarding hardware-based vulnerabilities has begun to subside, the potential risk to unprotected systems remains just as relevant. Threat actors continue to design exploits that specifically target physical weaknesses in processor's hardware. The now infamous attacks, Spector and Meltdown ushered a new era of hardware-based security exploits by attacking vulnerabilities in computer processor hardware instead of attacking software vulnerabilities. Immediately following their disclosure, Intel and AMD scrambled to quickly release firmware patches to their processors in order to mitigate the potential risk of exploitation.

However, not everyone felt that the mitigations were sufficient. Google later announced that it was disabling Hyper-threading on all Chromebooks running Chrome OS 74. Additionally, Theo de Raddit, founder and owner of OpenBSD, railed against Hyper-threading stating that, "SMT is fundamentally broken because it shares resources between the two CPU instances and those shared resources lack security differentiators." [1] He went on to explain that the risk of side-channel attacks like Spectre and Meltdown are dangerous enough to warrant disabling Hyper-threading on all computers running OpenBSD OS [2].

Despite the previously mentioned security concerns, system architects should not automatically follow Google and de Raddit's lead. Instead, architects ought to conduct a thorough analysis of their own environment to determine the risk level of leveraging multithreading. The following provides insights that can help architects make informed decisions when determining the suitability of SMT by exploring key processes related to SMT, reviewing prominent SMT exploits, and describing why Spectre and Meltdown does not necessarily warrant disabling SMT. Building this case, the remainder of the paper will first provide a high-level description of the key processes and components involved during speculative execution. Second, briefly describe the most renown Spectre and Meltdown variants. Third, leveraging the Common Vulnerability Scoring System, provide a methodology to characterize these exploits' severity.

II. Key Processes and Components

A. Simultaneous Multithreading

Processors were originally constructed from a single core with a single thread of execution. However advantages in a number of areas to include energy efficiency, true concurrency, performance, isolation, and reliability led to CPUs designs with multiple physical cores (each possessing their own ALU, registers, and other necessary components) on a single die [3]. The die is still commonly referred to as a single CPU despite it being composed of numerous individual cores. Fig. 1 below depicts how multiple cores can operate within a single die.

However, additional transitors packed onto a more compact form led to greater heat dispertion and power consumption - something that is not desirable in a processor. In order to combant this, SMT was created. SMT allowed a single core to to appear as two (or more) logical cores to an OS with each logical core running a single thread of execution. This meant that although a CPU may only have four physical cores (quad core), with SMT, the OS would percieve it as having eight logical cores. If a program required that the processor fetch a section of data from main memory, instead of the rest of the core sitting idly by waiting for the data transfer, the ALU or FPU could begin computation on another section of instructions.

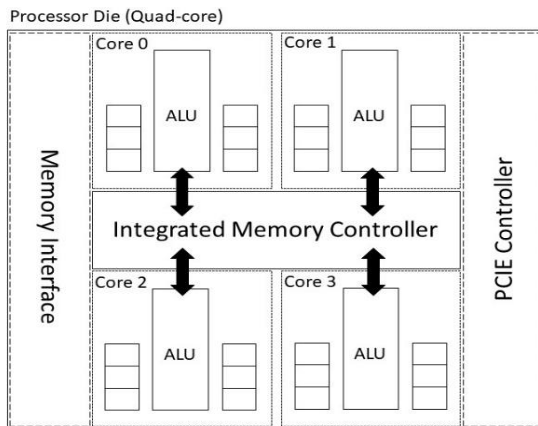

Fig. 1. Typical Architecture and Interactions for Multiple Cores on a Single Die.

B. Caching

Caching is a technique used by processors to compensate for the dichotomy between the processing speeds of the CPU and the slower speeds of main memory (RAM). Modern CPUs use a hierarchy of successively slower but larger caches built directly into the chip in order to decrease the latency between execution of instruction sets. The cache is divided into fixed-sized chunks of memory called lines. Each line is typically 64 or 128 bytes long, with larger cache sizes resulting in greater speed performance. Depicted in Fig. 2, when the processor attempts to fetch data from memory, it will first check the L1 cache at the top of the hierarchy for a copy of the data [4]. If the data is found, it is referred to as a cache hit, if not it is referred to as a cache miss. This process is then repeated moving down the chain from L1 to L2 to L3 cache until a hit is found. If all three caches result in a miss, then the processor will check system memory for the necessary data.

C. Speculative Execution

Speculative execution, also known as branch prediction, is the process by which a microprocessor closely tied into the fetch stage of the CPU instruction cycle makes a prediction as to what is most likely the next sequence of instructions in a program.

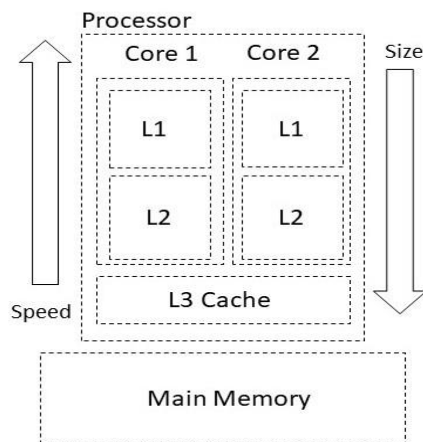

Fig. 2. Typical Cache Hierarchy.

Modern Intel processors have multiple forms of speculative execution for direct and indirect branches. Direct branch instructions relocate the stack pointer to a predefined memory address prior to the execution of the program [5]. In contrast, indirect branch instructions can jump to random memory addresses in the program computed at run time. In the Intel x86 architecture, this can be accomplished via the `jmp` instruction to jump to an address in a register, memory location, or on the stack. `jmp eax`, `jmp [eax]`, and `ret` respectively. Similar indirect and direct branch calls are also supported on ARM, MIPS, and RISK V architectures.

There are several functions that optimize indirect branches in order to compensate for the additional clock cycles necessary to process the instruction. The Branch Target Buffer (BTB) is a simple cache managed by the control unit that stores a mapping from addresses that recently executed branch instructions to destination addresses [6]. The BTB is leveraged by the processor to predict future code addresses before the decoding stage of a branch instruction. The Return Stack Buffer (RSB) stores a clone of the most recently processed section of the call stack [7]. Both BTB and RSB can greatly improve the speed of a running program by reducing the amount of computational work necessary for the processor.

D. Protected Memory

Protected memory is a method of controlling memory access rights on embedded systems. Its main purpose is to prevent apps from accessing regions of memory that they do not have proper security rights. For instance, one tab open in a browser should be restricted from accessing working memory from another tab. Without proper memory protection, data sections are vulnerable to memory related exploits or code injections [8]. This can be prevented by either software or hardware solutions. Software solutions assign a key value to a program during runtime that it must provide in order to gain access to the protected memory [9]. Hardware solutions work in a similar manner but separate threads of execution with logical barriers.

III. The Exploits

A. Spectre

Spectre is a general term used to describe a group of microarchitecture attacks that 'trick' the processor into speculatively executing malicious instruction sequences [10]. Because these instructions are eventually reverted by the CPU, they are referred to as transient instructions. By influencing which transient instructions are speculatively executed, information about memory addresses can be leaked from protected memory. Two prominent variants of the Spectre attacks exist - CVE-2017-5753 and CVE-2017-5715.

The first variant maliciously trains the branch prediction algorithm into erroneously executing a section of code that would not normally have been executed in the normal Von Neumann sequence. Listing 1 provides a short code example provided by the original Google Project Zero paper that better illustrates how this attack occurs [11].

In listing 1, the untrusted_offset_from_caller variable contains attack-controlled data. During the first phase of the

attack, the code is repeatedly invoked with a valid value (i.e. a value between 0 and arr1->length-1) for untrusted_offset_from_caller and thusly trains the branch predictor to predict the conditional on line 11 to evaluate as true. Next, during the second (or exploit) phase, the attacker invokes the same code but this time with the untrusted_offset_from_caller variable set to a value out of bounds for arr1. Because the first phase of the attack trained the branch predictor to expect the conditional to return true, if arr1->length is not cached, the processor will begin speculatively executing.

```
1.  struct array {
2.      unsigned long length;
3.      unsigned char data[];
4.  };
5.
6.  struct array *arr1 = ...; /* small array */
7.  struct array *arr2 = ...; /* array of size 0x400 */
8.  /* >0x400 (OUT OF BOUNDS!) */
9.
10. unsigned long untrusted_offset_from_caller = ...;
11. if (untrusted_offset_from_caller < arr1->length) {
12.     unsigned char value = arr1-
    >data[untrusted_offset_from_caller];
13.     unsigned long index2 =
    ((value&1)*0x100)+0x200;
14.     if (index2 < arr2->length) {
15.         unsigned char value2 = arr2->data[index2];
16.     }
17. }
```

Listing. 1. Spectre Exploit Pseudocode.

arr1->data[untrusted_offset_from_caller] even before the conditional has been fully evaluated.

Eventually the processor will finish evaluating the conditional and return to its normal non-speculative path. The out-of-bounds index call will be rolled back along with value2. However, the speculative execution will still temporarily store arr2->data[index2] in the L1 cache. By measuring the difference between the time to load arr2->data[0x200] and.

arr2->data[0x300], it can be determined whether value of index2 was 0x200 or 0x300; this determines if arr1->data[untrusted_offset_from_caller]&1 is 0 or 1. By selecting an appropriate value for untrusted_offset_from_caller, this process can leak address information stored in protected memory that usually would not be released but due to speculative evaluation is temporarily stored in the L1 cache [12].

The second variant of Spector relies on selecting a gadget in the victim's address space and influencing the target to speculatively execute that gadget. A gadget is a series of predefined machine instructions from the program being exploited. In order to execute the gadget, the attacker trains the BTB to erroneously predict an indirect branch to the address of the gadget. The training consists of the attacker repeatedly indirectly branching to the address of the gadget. After the BTB is trained to speculatively execute the address of the gadget, a similar approach as in variant one can be applied to leak protected memory.

B. *Meltdown*

Unlike Spector, Meltdown (CVE-2017-5754) does not rely on vulnerabilities in the victim's source code, but instead leverages out of order execution to run malicious code instructions in user-space. Note, an important part of OS memory management is how the OS links virtual memory to physical memory. When a process requests system memory, the OS will allocate a section of physical memory for use by the program. The OS then provides the running process with a virtual memory address that links to the actual physical memory address. A mapping between all virtual memory addresses and their physical counterparts is stored and referenced by the memory management unit (MMU).

To better illustrate how Meltdown leaks kernel physical memory addresses, consider the following short code section written in Listing 2 [13].

```
1.  ; rcx = kernel address, rbx = probe array
2.  xor rax, rax
3.  retry:
4.  mov al, byte [rcx]
5.  shl rax, 0xc
6.  jz retry
7.  mov rbx, qword [rbx + rax]
```

Listing. 2. Meltdown Exploit Pseudocode.

Three main steps are involved in the execution of the Meltdown vulnerability. Step one, an attacker selects the kernel restricted memory location they are attempting to gain access to and then loads that address into a specified register. This is accomplished in line 4 where the values of the kernel address (virtual memory) are stored in the least significant byte of the *RAX* register. When line 4 is executed, the *mov* instruction is fetched by the core before being decoded into microcode (μOps), allocated, and sent to the record buffer. By leveraging a process called out-of-order execution, the decoded and allocated μOps of lines 5-7 will be executed before line 4 is completed.

Step two consists of 'tricking' the processor into storing the restricted memory addresses in the L1 cache. Line 5 multiplies the secret value from Step 1 (the linked physical address to the virtual memory location) by the page size. This negates the ability for the hardware pre-fetcher to load adjacent memory locations into cache. In the example above, a single byte is read at a time resulting in the dimensions of the probe being 256 x 4096. Finally, Line 7 adds the multiplied secret to the base address of the probe array. Similar, to the Spectre attack, once the processor finishes the *mov* instruction it will throw an interrupt error and roll-back the following instructions, but the probe array will have already been stored in the L1 cache.

Step three, the exploit recovers the secret value from Step 1 via a microarchitecture side-channel attack on the L1 cache. When the transient instructions defined in step two are

executed, only one line of the probe array is cached. The position of that line in the array will depend on the value of the original secret. Therefore, by iterating over the full 256-page array and measuring the quickest load time, (*i.e. the cached one)* the original secret value can be determined. If this same process is repeated for each of the kernel restricted memory addresses, a complete memory dumb can be constructed of the kernel protect memory.

IV. Understanding the CVSS

An understanding of vulnerability severity is an important guidepost for architects deciding if they want to disable HT. CVSS scores are a useful tool to measure vulnerability severity. CVSS scores are developed by the multi-stakeholder organization Forum of Incident Response and Security Teams (FIRST) and serve as an important vulnerability severity assessment engine for many prominent organizations that include the National Institute of Standards and Technology (NIST) NVD database. CVSS assigns a vulnerability severity score that ranges from 0 – 10, with 10 being the most severe.

The heart of CVSS scoring is a combination of an exploitability assessment and a vulnerability impact assessment. Exploitability assessments characterize how easily a malicious actor may be able to exploit the vulnerability. Exploitability is comprised of possible attack vectors (AV), attack complexity (AC), the level of privilege (PR) required on a computer needed to execute an exploit, and whether the exploit can be executed without user interaction (UI). Impact assessments characterize the magnitude of the vulnerability in terms of confidentiality loss, integrity loss, and availability loss to the data on the computer.

Fig. 3-5 from the NIST National Vulnerability Database demonstrates that neither Spectre variants nor Meltdown exhibit an overall CVSS score above a 'MEDIUM'. (4.0 – 6.9 Overall Score).

For a more detailed explanation of CVSS scoring, refer to the CVSS user guide [14].

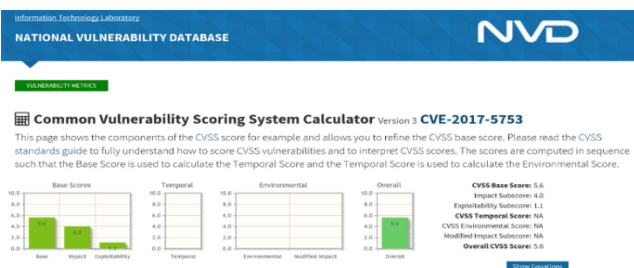

Fig. 3. Spectre CVE-2017-5753–5.6 Score.

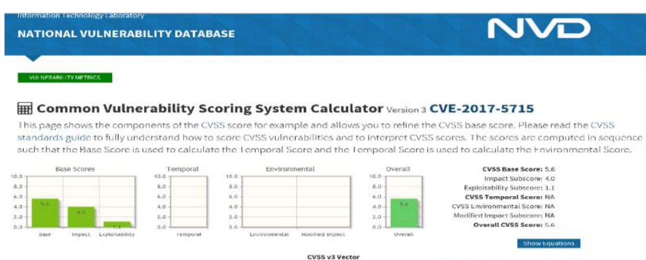

Fig. 4. Spectre CVE-2017-5715–5.6 Score.

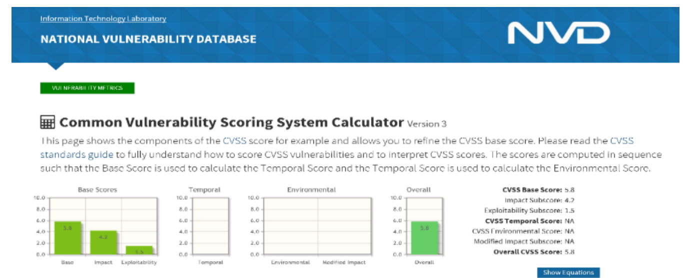

Fig. 5. Meltdown CVE-2017-5754–5.8 Score.

V. To Disable or Not to Disable SMT

Prominent subject matter experts such as Theo de Raddit argue that SMT's security vulnerabilities outweigh its performance advantages. However, this paper contends that while disabling hyperthreading may be appropriate in some environments, it should not be a defacto standard for enterprise architects. The CVSS assessment does not warrant such an action.

Although Spectre and Meltdown do present serious security concerns, their overall vulnerability severity remains a relatively unremarkable 'MEDIUM'. They both have a 'LOW' severity for exploitability due to the fact that neither attack can leverage a network connection for lateral movement. Additionally, these exploits are extremely difficult to operationalize and the ability for a malicious actor to successfully launch an attack is far from certain. Resultantly, both attacks have a higher attack complexity and a lower exploitability severity.

Spectre and Meltdown's impact severity do slightly elevate to the 'MEDIUM' range but remains well below the 'HIGH' severity threshold. Data confidentiality risks are more concerning because of memory dumping from kernel protected addresses, or information leaking from protected memory, but the risk to data availability or data integrity remains 'LOW'. Potential data leaks do not result in an alteration of stored data or the ability to access saved data.

VI. Conclusion

There is no 'one size fits all' approach to risk tolerance. Unique to each environment, is a careful balance between security and performance. Business and operational requirements should drive cybersecurity risk mitigation decisions. This balance aptly applies when determining whether to disable SMT. Even though both Spectre and Meltdown both rank 'Medium' in CVSS severity, a more security focused environment might be inclined to disable SMT and accept the performance loss. However, environments with a less cautious risk tolerance that needs the performance advantages from SMT to facilitate business operations should not disable SMT by default and instead evaluate software application-based patch mitigations.

References

[1] Theo Raditt, Disable SMT/Hyperthreading in all Intel BIOSes, e-mail message, OpenBSD Journal, August 24, 2018. https://undeadly.org /cgi?action=article;sid=20180824024934

[2] Ibid.

[3] Donald Firesmith, "Multicore Processing", Carnegie Mellon Software Institute Blog, August 21, 2017, https://insights.sei.cmu.edu/sei_blog/multicore-processing-and-virtualization/ttps://insights.sei.cmu.edu/sei_blog/2017/08/multicore-processing.html

[4] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom, "Spectre Attacks: Exploiting Speculative Execution", ArXiv e-prints, January 2018, p 4 - 7.

[5] David Gregg, M. Ertl, (2003). "Optimizing indirect branch prediction accuracy in virtual machine interpreters", Sigplan Notices - SIGPLAN 38, 2003, p 278-288.

[6] Po-Yung Chang, Eric Hao, and Yale Patt, "Target Prediction for Indirect Jumps," ACM SIGARCH Computer Architecture News, March 1998, pg 274 – 282.

[7] Giogo Maisuradze and Christian Rossow, "Speculative Execution Using Return Stack Buffers", Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, October 15 – 19, 2018, pg 2109 – 2111.

[8] Ibid, pg 2111 – 2114.

[9] Ibid, pg 2110.

[10] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom, "Spectre Attacks: Exploiting Speculative Execution", ArXiv e-prints, January 2018, pg 2.

[11] Jann Horn,"Reading Privileged Memory with a Side-Channel Attack", Google Project Zero Blog, January 3, 2018. https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html

[12] John Criswell, Nicholas Geoffray, and Vikram Adve, "Memory Safety for Low-Level Software/Hardware Interactions", USENIX Security Symposium, August 10 – 14, 2009, pg 10.

[13] Moritz Lipp et all. 2018, "Meltdown: Reading Kernel Memory from User Space", USENIX Security Symposium, August 15 – 17, 2018, pg. 980.

[14] CVSS User Guide. 2019. FIRST.org. https://www.first.org/cvss/user-guide.