Transient Execution and Side Channel Analysis: a Vulnerability or a Science Experiment?

Michael Shepherd, Scott Brookes, and Robert Denz
Riverside Research, Lexington MA, USA
shepherdm1@wit.edu
sbrookes@riversideresearch.org
rdenz@riversideresearch.org

Keywords: Side Channel, Transient Execution, Spectre, Meltdown, Prime and Probe, Flush and Reload

Abstract: In the world of computer security, attackers are constantly looking for new exploits to gain data from or control over a computer system. One category of exploit that can prove quite effective at accessing privileged data is side channel exploits. These exploits attempt to take advantage of vulnerabilities that are inherent in the design of a system rather than vulnerabilities in the code that has been written for and is running on said system. In other words, they exploit side effects of computation. Examples of this include measuring the power consumption of a system's processor over time and analysing that power usage to leak system secrets or reading secrets from a system by analysing the electromagnetic radiation the system leaks as it processes data. Another type of side channel attack is a cache-based side channel attack, which exploits the timings of cache and memory accesses to determine data from the target system. We discuss some of the more common types of side channel attacks used to interpret data values from the microarchitectural changes created by transient executions. In particular, we will focus on attacks that are capable of recovering data that is processed through transient execution in some way and then wrongly accessed using a side channel, such as the Spectre and Meltdown classes of attack. We also discuss other attacks of a similar type and survey some popular mitigations for these attacks. We provide a survey of all available Spectre proof-of-concept repositories on GitHub, evaluating whether they work on different platforms. Finally, we review our experiences with these types of attacks on modern systems and comment on the attacks' practicality, reliability, and portability. We conclude that these types of attacks are interesting, but there are some practicality and reliability concerns that make other attacks easier much of the time.

1. Introduction

Side channel attacks utilize by-products of computation such as power, electromagnetic (EM) radiation, or timing to leak information about the computation. Most recently, the use of timing side channels alongside transient execution has led to new side channel vulnerabilities.

The transient domain refers to a type of instruction executed by the processor before it is known whether the instruction should be executed. One example is speculative execution, where the processor might use its Branch Target Buffer (BTB) to guess which direction a branch will take and transiently execute instructions on that path. Instructions executed in this transient domain do not make changes to the visible processor state unless they are committed. If the processor determines that the transient instructions are not valid, it discards the results of the instructions.

Although transient execution does not change architectural processor state until the execution is validated, it does leave traces in the microarchitectural state. For example, transient instructions often change the state of the cache by evicting or caching certain data. Cache-based side channels can then recover this microarchitectural state. As a result, program secrets that are operated on in the transient domain can be recovered from the microarchitectural state of the processor by attackers using a side channel.

In this paper, we examine different types of attacks that expose secrets in the transient domain and recover those secrets using cache-based timing side channels. As this class of attacks does not require physical access to the machine, they are especially potent.

The main contributions of this paper include:

- Comparing and contrasting different cache-based timing side channel attack methodologies.
- Explaining how various transient execution attacks work.
- Providing a thorough analysis of the Spectre proofs-of-concept available in the public domain.
- Demonstrating a novel attack, SpectreXP, which uses a network connection to exploit Spectre across the process isolation boundary.

In Section 2, we provide a brief discussion of background and related work. In Section 3, we explain cache-based timing side channels and study methods for recovering secrets from the cache using timing information. Then, in Section 4, we examine different types of attacks that reveal secrets in the transient domain. After these explanations, we will discuss our practical experience during our experimentation with these types of attacks on different types of real systems in Section 5. Section 6 offers concluding thoughts and future work.

2. Background & Related Work

Transient execution attacks turned the computer security world upside down in 2018 with the publication of the original Spectre (Kocher et al, 2019) and Meltdown (Lipp et al, 2018) attacks. The gravity of the situation became clear when it was shown that Spectre could be exploited via Javascript to break out of a browser sandbox (Novack and Reichert, 2018).

Patches for the original vulnerabilities came out relatively quickly. Browsers reduced the precision of their timers to interrupt the cache-based timing side channels used to recover secrets in the Spectre attack (Wagner, 2018) while operating system vendors implemented patches such as KPTI that mitigated Meltdown (kernel.org; Gruss, 2017).

Unfortunately, patching this class of vulnerability turned out not to be so easy. Not only was it shown that reducing timer precision was not sufficient to stop Spectre (McIlroy et al, 2019) but a variety of additional instances of this class of attack began to surface including Foreshadow (Van Bulck et at, 2018; Weisse et al, 2018), ZombieLoad (Schwarz et al, 2019), RIDL (Van Schaik et al, 2019), Fallout (Canella et at, 2019), and others. Indeed, Canella et al. (2019) provide an analysis showing that there are tens of instances of this class of attack between the Spectre and Meltdown approaches alone.

This leads us to the current state of the art. Defenders are playing a game of whack-a-mole as researchers are still bringing new attacks that exploit the leakage from the microarchitecture to the architectural level of the processor. Works that survey this landscape in greater detail than allowed in the scope of this paper include those by Xiong and Szefer (2020), Canella et al. (2019) and Canella et al. (2020). While these works deeply evaluate the technical aspects and theory of transient execution attacks, this paper will focus more heavily on our experience with researching currently available exploit examples and testing them on real world hardware and software. We will also discuss our analysis of the limitations and practicality issues that exist that affect the usefulness and danger of transient execution attacks in the wild.

3. Cache-Based Timing Side Channel Methods: Extracting Secrets from the Cache

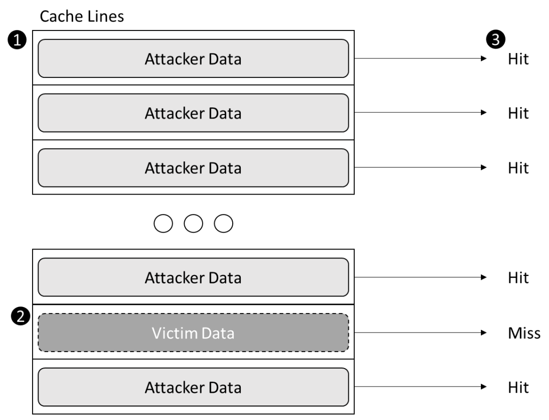
Two side channel methods are used in a majority of cases to retrieve data from a cache altered by transient execution. These methods are known as prime and probe and flush and reload.

3.1 Prime and Probe

A popular side channel for leaking data from the cache is known as "prime and probe" (Osvik et al, 2006; Tromer et al, 2010) which uses a 3-step process, illustrated in Figure 1, to pull data from the cache without assumptions about shared memory.

1. First, the attacker "primes" the cache, filling it with random attacker-controlled data. This ensures that if the attacker attempts to access any of this data, it will result in a cache "hit", or a very fast access of the requested data as it is retrieved from cache instead of the significantly slower RAM.
2. The attacker then waits for the victim to execute code that accesses the target data. This process of pulling data into the cache causes some of the attacker-controlled data to be evicted from the cache.
3. Finally, the attacker "probes" the cache by attempting to access its attacker-controlled data, looking for a cache "miss" where data was evicted in step 2. In other words, it looks for a slow data access taking longer to access than it would to retrieve that data from the cache memory.

Figure 1: Prime and Probe uses 3 steps: 1 - the attacker fills the cache with known data, 2 - the victim accesses the secret, evicting some attacker-controlled data, 3 - the attacker looks for a miss to determine which data



was accessed by the victim

By mapping certain cache lines to certain values, the value of the accessed data can be determined based on which cache line was evicted in the second step. This allows for the extraction of data, including data that is accessed only transiently, from a system without the attacker directly accessing that data using only changes in the cache. This can even be deployed against the Last Level Cache (LLC) as shown by Liu et al. (2015).

3.2 Flush and Reload

A newer type of cache timing side channel called "flush and reload" (Yarom and Falkner, 2014) seeks to improve upon the speed and accuracy of prime and probe with the trade-off of requiring that the victim and attacker processes share memory. One way to achieve this shared memory would be running the attacker and victim process hyperthreaded on the same core, sharing their L1 and L2 cache. However, you can also leverage the last level cache to read data from another process, as it is shared across all cores as shown in Figure 2.

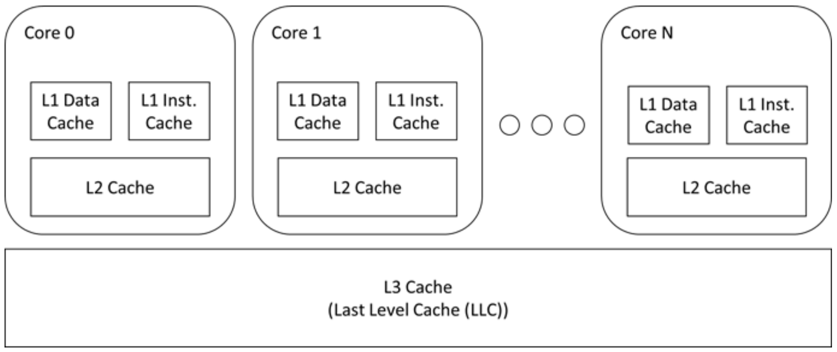


Figure 2: While each core has its own L1 and L2 cache, the L3 cache or "last level cache" (LLC) is shared among all cores, allowing for attacks using flush and reload to work across cores using the LLC as shared memory

Flush and reload is based on the same principles as prime and probe, except it is looking for a cache hit instead of a cache miss, as illustrated in Figure 3.

1. The first step of a flush and reload attack is to simply flush the entire cache.
2. Once the cache has been emptied, flush and reload, like prime and probe, waits for the victim to access target data, again either transiently or correctly. Instead of evicting attacker-controlled data from the cache, this causes some cache lines in the shared memory to "light up" or become filled with data as the system pulls the requested data into the previously empty cache.

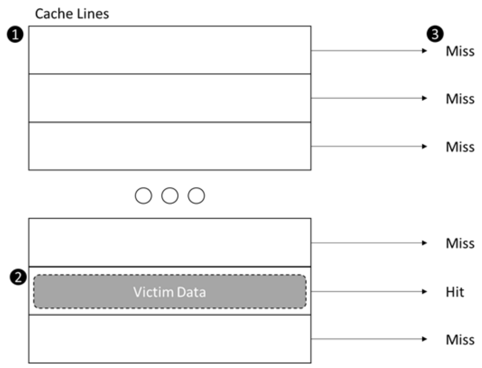


Figure 3: Flush and Reload uses three steps: 1 - the attacker empties the cache, 2 - the victim accesses the secret, filling a cache line with data, 3 - the attacker looks for a cache hit to determine which data was accessed by the victim

3. This attack then "reloads" the cache by going line by line, looking for a cache "hit," as the lines that have data in them after step 2 will respond to a read much faster than empty lines that must retrieve the requested data from RAM.

4. Transient Execution Attacks: Exposing Secrets in the Transient Domain

While side channel attacks can be used to detect changes made in the cache at the microarchitectural level, that alone is not enough to leak data from a process. Transient attacks make use of transient execution, or execution that is performed ahead of time and thrown out if it is wrong, to leave microarchitectural changes on the cache that can then be picked up by a side channel attack.

4.1 Meltdown

Meltdown (Lipp et al, 2018) leverages a feature known as "out of order execution" in order to overcome memory isolation and read privileged memory, including arbitrary kernel memory. Modern processors use out of order execution to increase performance, as this feature enables the processor to look ahead in the instruction sequence and schedule future operations to idle execution units on the CPU.

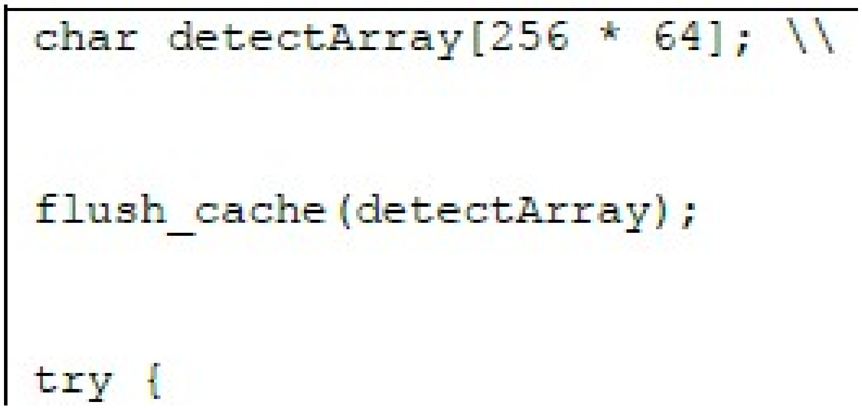
Meltdown leverages the fact that while a CPU will throw out data if it was accessed with insufficient permissions, it may transiently load privileged data with an unprivileged program. The CPU throwing out the accessed data ensures normal program execution, but Meltdown leverages the fact that this out of order memory access makes microarchitectural changes to the cache. These changes can be detected and deciphered using a side channel such as flush and reload or prime and probe, bypassing security.

4.1.1 Pseudocode

To illustrate the Meltdown attack, consider the pseudocode shown in Figure 4.

First, we declare a memory region of the same size as the entire cache. Then, we flush the cache. Next, we dereference a kernel address, effectively assigning secret data to the variable `a`. Although this will cause a segmentation fault in the program, it will not crash before the transient domain looks ahead and continues execution to include the operation on the `a`th 64 byte region of `detectArray`. At this point we know that one cache line has been filled with data: the `a`th one corresponding to the secret value. By detecting which cache line results in a hit, one can determine the value of the secret.

Note that the key here is using the secret as an offset within known data. This allows inspection of the known data's footprint in cache to reveal the value of the secret.



```
char detectArray[256 * 64]; \\

flush_cache(detectArray);

try {
```

Figure 4: Code snippet for Meltdown attack from Godbolt (2018)

4.1.2 Meltdown Mitigations

After Meltdown's discovery, manufacturers and developers rushed to find mitigations to this exploit, as it gives an attacker complete access to a system's privileged memory. One solution would be to disable out of order execution, but that would result in an unacceptable loss of performance. Other mitigations have high performance overhead as well. One example of this is kernel page table isolation (KPTI) (kernel.org; Gruss, 2017) for the Linux kernel, which creates a shadow copy of the system's page tables for use when operating in user space mode that do not have kernel space memory mapped in them aside from pointers required for system calls and interrupts. This means that while Meltdown can still leak memory, it cannot leak kernel or physical memory as there are no mappings in the user space page tables for those addresses. KPTI can have a performance impact as high as 30% on certain workloads (Gregg, 2018).

4.2 Spectre

Spectre (Kocher et al, 2019) is another transient execution attack which leverages a feature of modern CPUs called "speculative execution." Speculative execution improves performance by executing instructions/calculations before they're needed by the program so they can be ready once they are needed. However, while out of order execution simply looks ahead in the instruction sequence, speculative execution attempts to "guess" what the result of a branch (e.g., an 'if' statement) will be and performs operations that may not be required. In this way, if they are required, the execution has already been completed, and if not, the system simply discards the transient execution.

The issue with Spectre is that instructions executed speculatively can bypass security such as bounds checking and wrongly access data within the victim process's memory address space. To perform a Spectre attack, the attacker trains the CPU to expect a branch to go a particular way by invoking victim code with valid inputs, then invokes said code with a malicious input to make it wrongly speculate and access secret data in the transient domain. Figure 5's code snippet shows a vulnerable function usable by a Spectre attacker, feeding it valid 'x' values to train the branch predictor to expect the if statement to be true, then giving it a malicious x value that results in an out of bounds transient memory access that can be recovered via a side channel attack.

```
if (x < array1 size)
```

Figure 5: Spectre gadget

4.2.1 Spectre Mitigations

Spectre is very difficult to mitigate. With Meltdown, simply making the attacker unable to access page tables for sensitive kernel and physical memory means that it cannot leak secrets. The fact that Spectre leaks any data in the victim's memory address space means that Spectre can leak secrets from within the process itself. Simply disabling speculative execution would defeat Spectre, but this would also result in significant and unacceptable performance loss as modern systems rely heavily on speculative execution for their high performance.

There are some proposed mitigations for Spectre class attacks. One example that Mozilla chose for its Firefox browser was to lower its timer resolution (Wagner, 2018). The lower resolution should make it difficult or impossible to measure the timing of memory reads to an accurate enough degree for a side channel attack to function. In other words, this does not stop Spectre itself but makes it impossible to retrieve the value of the leaked secret from the microarchitecture.

Another solution that has been developed to mitigate Spectre is known as a “return trampoline” or “retpoline” (Intel, June 2018). This is a method of steering the CPU in an infinite loop in the transient domain to stop it from speculating on the target of an indirect jump, achieving the same effect as turning off speculation for specific portions of a victim program. This means that speculation operates normally, but vulnerable pieces of code can be surrounded by a retpoline to prevent them from executing in the transient domain and leaking data.

The other main mitigation recommended by CPU manufacturers for Spectre is to add an “LFENCE” instruction before critical pieces of code (AMD, 2020; Intel, Jan. 2018). This instruction does not execute until all previous instructions have finished executing, stopping the critical code from executing speculatively and keeping Spectre from exploiting it.

4.3 ZombieLoad

Zombieload (Schwarz et al, 2019) is another type of transient execution attack that works by leaking recently loaded or stale values across logical cores using the CPU’s fill buffer. This attack uses “zombie loads” or more specifically, loads of data that fault at the architectural or microarchitectural level and need to be re-done at a later time. These loads can result in the “dead” data being loaded transiently, aka a zombie load. Once the data has been transiently loaded, it is retrieved using a side channel attack. This attack can currently bypass mitigations for Spectre and Meltdown because it uses the CPU fill buffer instead of the cache. Currently, there are no low performance impact protections that work to fully mitigate ZombieLoad. The high-performance impact protection is to completely disable hyperthreading and flushing the microarchitectural states during context switches. Note that variance in CPU microarchitecture design and features results in ZombieLoad attacks working intermittently across varying CPU architectures.

5. Experience

Throughout our experience evaluating transient execution side channel attacks, we encountered many oddities that make researching and troubleshooting them difficult. The first of these difficulties was the fact that much of the information that exists on these topics is outdated, contradictory, or otherwise difficult to parse and understand. The second and most prevalent of these difficulties was instantiating proof-of-concept codebases we found for these attacks. Many of the codebases we came across either did not work or worked only partially, and it was common for them to have little accompanying documentation with the code base. Compounding the issue, the accuracy and functionality of much of this code is inconsistent across devices. What works flawlessly on most devices may not work at all on another. This, combined with the limited knowledge available about in-depth CPU architecture due largely to the proprietary nature of the information, makes troubleshooting these proofs-of-concept non-trivial.

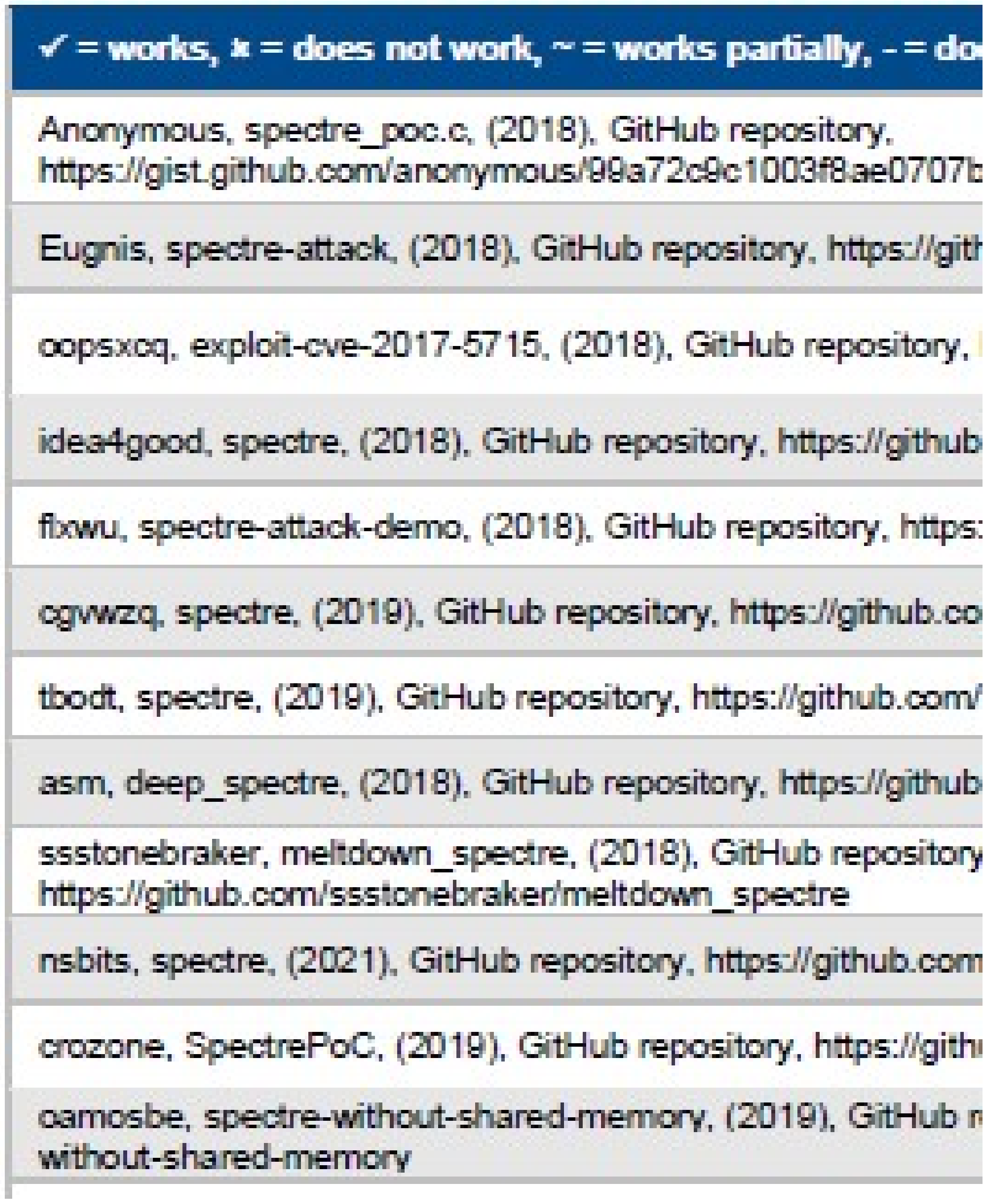
5.1 Transient Execution Practicality Issues

These transient execution attacks each seemed to have their own issues when it comes to their practical use in a real-world attack in our experience. Of the different attacks we researched and experimented with, Meltdown seemed to have the fewest limitations and the highest potential for use in an actual attack before the patches that mitigated it’s usefulness, whereas Spectre and ZombieLoad both have their own set of limitations.

5.1.1 Spectre Limitations

We came across several limitations that make Spectre impractical or inefficient. One of the main issues we had with Spectre was with consistency, namely being able to get the same code to work on different systems in a consistent manner. The main issues with this appeared to revolve around using either the flush and reload or prime and probe side channel. Spectre implementations using flush and reload worked with 100% accuracy on 2 of the 3 systems we had access to for testing, but not at all on the third. Conversely, prime and probe Spectre implementations we tried worked only partially on the third system (with very low and inconsistent accuracy), but not at all on the first 2. This inconsistent accuracy is illustrated in Table 1, where we have tested all currently available GitHub repositories implementing Spectre and recorded the results per machine. The first and second of the 3 machines tested gave identical results, so they have been grouped together. The third machine uses a pre-release processor from Intel with new, largely undisclosed security features built in, likely contributing heavily to the differences in attack performance between it and machines 1 and 2. Interestingly, this machine did not defeat all of the proofs-of-concept despite its higher security configuration.

Table 1: Spectre repositories were tested on 3 different machines running Linux, the first of which was virtualized on an Intel® Core™ i7-6700HQ, the second of which was running natively on an Intel® Core™i5-8259U, and the third of which was running natively on an unidentified Intel® Xeon™Ice Lake processor



✓ = works, ✗ = does not work, ~ = works partially, - = do
Anonymous, spectre_poc.c, (2018), GitHub repository, https://gist.github.com/anonymous/99a72c9c1003f8ae0707b
Eugnis, spectre-attack, (2018), GitHub repository, https://git
oopsxoq, exploit-cve-2017-5715, (2018), GitHub repository,
idea4good, spectre, (2018), GitHub repository, https://github
flxwu, spectre-attack-demo, (2018), GitHub repository, https:
cgvwzq, spectre, (2019), GitHub repository, https://github.co
tbodt, spectre, (2019), GitHub repository, https://github.com/
asm, deep_spectre, (2018), GitHub repository, https://github
ssstonebraker, meltdown_spectre, (2018), GitHub repository https://github.com/ssstonebraker/meltdown_spectre
nsbits, spectre, (2021), GitHub repository, https://github.com
crozone, SpectrePoC, (2019), GitHub repository, https://gith
oamosbe, spectre-without-shared-memory, (2019), GitHub r without-shared-memory

Furthermore, we found that not only did the two different side channels used not work across different systems consistently, but they also did not seem to hold up to their theoretical abilities. The best example of this is with flush and reload, which should be able to work across physical cores by operating in the last level cache, but we found that this was not the case. Over the course of our testing, flush and reload based attacks always needed the attacker and victim to be pinned to the same core as the side channel attack would only succeed in the L1 or L2 cache.

The second issue we found is that even if the attack does work on the hardware it's being run on, the attacker must have a significant amount of control over the victim program in order to steal data from it. It needs to be able to call functions within the victim with specific, attacker-controlled inputs. In most cases the amount of access, permissions, and knowledge required would make it more efficient to simply use a different privilege escalation attack to achieve the same result.

5.1.2 ZombieLoad Limitations

ZombieLoad, while an interesting concept and even while being reasonably fast and accurate in the values it is reading, is again not practical for most attacks. This stems largely from the fact that the attacker does not get a choice of what data ZombieLoad recovers as it simply grabs whatever was in the last load. This could be useful as an attack in exactly the right conditions, but in most scenarios would require the same data to be accessed many times in a row to steal any relevant victim secrets. An attack such as leaking a single AES might take years to accomplish. This means that a different type of attack will usually be a better option.

5.2 SpectreXP: A Cross-Process Spectre Proof-of-Concept

One issue we found when researching Spectre and attempting to design a practical attack that could simulate a real-world data leak was that most of the proof of concepts that currently exist are only a single process, which steals data from inside itself by reading out of bounds from an array. We were unable to get the ones that do implement Spectre across multiple processes working for various reasons, mostly relating to issues with the side channels being used.

To prove that Spectre can in fact work across multiple processes, we created a novel Spectre proof-of-concept, which we have dubbed as "SpectreXP". Our proof-of-concept consists of a simple victim program that is intentionally vulnerable to Spectre by containing code like that shown in Figure 5. The attacker program sends inputs to the victim via a socket. As with other Spectre variants, the attacker sends many valid values as inputs to the victim program to mistrain the CPU branch predicter. Once the branch predicter has been sufficiently trained, the attacker sends the victim an input that is invalid. This will cause the victim to speculatively access secret memory, leaving changes in the cache that the attacker can recover, as shown in Figure 6.

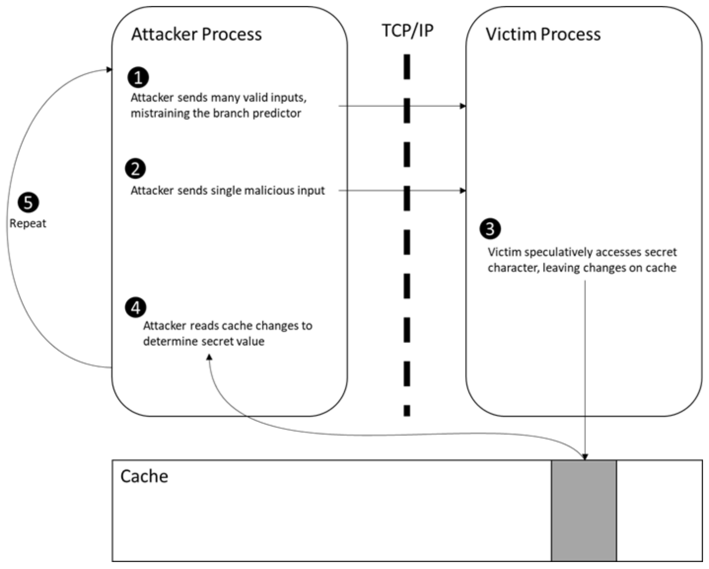

Attacker Process
TCP/IP
Victim Process
❶
Attacker sends many valid inputs, mistraining the branch predictor
❷
Attacker sends single malicious input
❺
Repeat
❸
Victim speculatively accesses secret character, leaving changes on cache
❹
Attacker reads cache changes to determine secret value
Cache

Figure 6: SpectreXP: 1 – the attacker trains the branch predictor with valid data. 2 - the attacker sends a malicious input to force a speculative read. 3 – the victim uses that input to read arbitrary data speculatively. 4 – the attacker reads microarchitectural changes using flush and reload to determine part of secret value. 5 – repeat until entire secret has been read.

This proves that it is possible to get Spectre to work across multiple processes as long as they share memory in the form of the cache, specifically the L1 or L2 caches due to the found limitations of flush and reload. This means that the 2 processes also need to be pinned to adjacent logical cores via hyperthreading. The code for this new proof of concept can be found here: https://github.com/riversideresearch/spectreXP

6. Conclusion

In this paper, we presented our research into timing-based side channel attacks and the Meltdown, Spectre, and ZombieLoad attacks, as well as going over our experiences with experimenting with these types of attacks on real hardware. We found that the theoretical properties of side channel attacks did not always hold up on real hardware and were not consistent across different hardware, meaning that the attacks that rely on them also often fail to work partially or entirely. Side channels are also difficult to research due to the outdated, contradictory, or complex nature of much of the existing research. We also learned that debugging these experiments can often prove to be non-trivial due to the proprietary, poorly documented, or otherwise non-disclosed nature of the hardware or software being worked with. Due to these issues, we have come to the conclusion that there are few, if any, situations in which transient execution attacks are truly effective or efficient against current technology in the real world.

6.1 Future Work

Future work would revolve around gathering up all the information available on this topic and verifying it piece by piece. The goal of this would be to build a "unified theory" of side channel and transient execution attacks, finding and correcting as many inaccuracies and contradictions as possible while compiling and standardizing the ideas, vocabulary, and practices of existing work. This would allow us to dig deeper into the intricacies of CPU architecture and learn why, for instance, certain attacks do not work consistently across different hardware. This goal would benefit greatly from the testing of these attacks on more machines and architectures than we had access to for experimentation purposes, such as those running AMD or ARM processors. Other metrics that would be useful to record about these exploits in the course of this research would be the accuracy in retrieving known data over many trials, how quickly they receive that data, and how easy they are to detect and mitigate. Learning about these topics and collecting data from additional experiments would then allow us to create faster, more accurate, and more resilient attacks to push the boundaries of these technologies.

One "final" goal to come out of all this would be that of building a proof-of-concept for an attack that could be dropped onto nearly any combination of hardware and software and run with nearly 100% accuracy, finally allowing us to circumvent the need for heavy customization per each system being attacked. Another possible goal is using transient execution and side channels to break the barrier between host and guest in a virtualized environment and wrongly pass data back and forth from guest to host or vice versa.

References

AMD. (2020, September) *Software Techniques for Managing Speculation on AMD Processors*.
Canella, C., Genkin, D., Giner, L., Gruss, D., Lipp, M., Minkin, M., Moghimi, D., Piessens, F., Schwarz, M., Sunar, B. and Van Bulck, J. (2019, November) "Fallout: Leaking data on meltdown-resistant cpus", *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (pp. 769-784).
Canella, C., Khasawneh, K.N. and Gruss, D. (2020, September) "The evolution of transient-execution attacks", *Proceedings of the 2020 on Great Lakes Symposium on VLSI* (pp. 163-168).
Canella, C., Van Bulck, J., Schwarz, M., Lipp, M., Von Berg, B., Ortner, P., Piessens, F., Evtyushkin, D. and Gruss, D. (2019) "A systematic evaluation of transient execution attacks and defenses", *28th USENIX Security Symposium (USENIX Security 19)* (pp. 249-266).
Godbolt, Matt. (2018, 13 January) "Meltdown and Spectre." YouTube, uploaded by Matt Godbolt, [online], https://youtu.be/IPhvL3A-e6E

Gregg, Brendan. (2018, 9 Februry). “KPTI/KAISER Meltdown Initial Performance Regressions.” [online] https://www.brendangregg.com/blog/2018-02-09/kpti-kaiser-meltdown-performance.html
Gruss, D., Lipp, M., Schwarz, M., Fellner, R., Maurice, C. and Mangard, S. (2017, July) “Kaslr is dead: long live kaslr”, *International Symposium on Engineering Secure Software and Systems* (pp. 161-176). Springer, Cham.
Intel. (2018, January) *Intel Analysis of Speculative Execution Side Channels*.
Intel. (2018, June) “Retpoline: A Branch Target Injection Mitigation”, [online], https://software.intel.com/content/dam/develop/external/us/en/documents/retpoline-a-branch-target-injection-mitigation.pdf
Kernel.org. “Page Table Isolation”, [online] https://www.kernel.org/doc/html/latest/x86/pti.html
Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T. and Schwarz, M. (2019, May) “Spectre attacks: Exploiting speculative execution”, *2019 IEEE Symposium on Security and Privacy (SP)* (pp. 1-19). IEEE.
Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D. and Yarom, Y. (2018) “Meltdown: Reading kernel memory from user space”, *27th USENIX Security Symposium (USENIX Security 18)* (pp. 973-990).
Liu, F., Yarom, Y., Ge, Q., Heiser, G. and Lee, R.B. (2015, May), “Last-level cache side-channel attacks are practical”, *2015 IEEE symposium on security and privacy* (pp. 605-622). IEEE.
Mcilroy, R., Sevcik, J., Tebbi, T., Titzer, B.L. and Verwaest, T. (2019) “Spectre is here to stay: An analysis of side-channels and speculative execution”, *arXiv preprint arXiv:1902.05178*.
Noack, L. and Reichert, T. (2018) “Exploiting Speculative Execution (Spectre) via JavaScript”, *Advanced Microkernel Operating Systems*, p.11.
Osvik, D.A., Shamir, A. and Tromer, E. (2006, February) “Cache attacks and countermeasures: the case of AES”, *Cryptographers’ track at the RSA conference* (pp. 1-20). Springer, Berlin, Heidelberg.
Schwarz, M., Lipp, M., Moghimi, D., Van Bulck, J., Stecklina, J., Prescher, T. and Gruss, D. (2019, November) “ZombieLoad: Cross-privilege-boundary data sampling”, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (pp. 753-768).
Tromer, E., Osvik, D.A. and Shamir, A. (2010) “Efficient cache attacks on AES, and countermeasures”, *Journal of Cryptology*, *23*(1), pp.37-71.
Van Bulck, J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T.F., Yarom, Y. and Strackx, R. (2018) “Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution”, *27th USENIX Security Symposium (USENIX Security 18)* (pp. 991-1008).
Van Schaik, S., Milburn, A., Österlund, S., Frigo, P., Maisuradze, G., Razavi, K., Bos, H. and Giuffrida, C. (2019, May) “RIDL: Rogue in-flight data load”, *2019 IEEE Symposium on Security and Privacy (SP)* (pp. 88-105). IEEE.
Wagner, L. (2018, 3 January) “Mitigations landing for new class of timing attack”, [online], https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/
Weisse, O., Van Bulck, J., Minkin, M., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Strackx, R., Wenisch, T.F. and Yarom, Y. (2018) “Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution”.
Xiong, W. and Szefer, J. (2020) "Survey of transient execution attacks." *arXiv preprint arXiv:2005.13435*.
Yarom, Y. and Falkner, K. (2014) “FLUSH+ RELOAD: A high resolution, low noise, L3 cache side-channel attack”, *23rd USENIX Security Symposium (USENIX Security 14)* (pp. 719-732).