NetSpectre: Evaluating Speculative Execution with Peripheral Observations on Network Latency

Harry Liu
hl3717@columbia.edu
Columbia University
New York, New York, USA

Rui Gao
rg3533@columbia.edu
Columbia University
New York, New York, USA

ABSTRACT

This paper aims to serve as an introduction to the inner workings of Spectre with an explanation of various CPU features utilized such as cache and speculative execution, and we assume the reader to have some basic knowledge in Spectre. Coupled with an analysis of NetSpectre, we are ready to show that Spectre is hard to do but can be extremely powerful once done right.

KEYWORDS

Spectre, NetSpectre

1 INTRODUCTION

Programmers today are trained to view the computer as an abstract machine of wonder: the CPU will carry out its instructions flawlessly just as prescribed in the manual. On the other hand, things are quite different under the hood, and a sequence of obscure yet legitimate operations can turn the CPU upside down to produce side effects beyond the user's wildest dreams.

Let us start with a brief review of modern CPU's. When the CPU fetches a conditional branch, it takes more cycles from that point (in the hundreds if the CPU needs to load data from RAM) to compute its direction. Speculative execution keeps the pipelines occupied by choosing the most likely outcome based on history, and simply throws away the incorrect work in case of a failure in prediction. In the most unexpected way, attackers have found ways to exploit this feature to infer private data from side-channels.

Spectre (retrospectively labeled Variant 1) was discovered independently by Jann Horn from Google's Project Zero and other researchers across industry and academia [4]. Quietly haunting the computer, Spectre passes through the bounds check for a conditional branch and reads the side effects of speculative execution in the CPU's data cache hierarchy. Spectre, along with Meltdown, marked a significant moment in computer security [6].

Since its inception in 2017 and disclosure in 2018, Spectre has spawned numerous variants and related vulnerabilities, collectively known as the "Spectre-class" vulnerabilities [5]. These attacks share the same underlying structure of the original Spectre, but each cleverly exploits a unique part of the CPU's micro-architecture and is named by its principal method. Key variants besides the original Variant 1 (Bounds Check Bypass) include Variant 2 (Branch Target Injection) and Meltdown reclassified as Variant 3 (Rogue Data Cache Load). Figure 1–3 include the workflows of Variant 1–3 for comparison purposes [3].

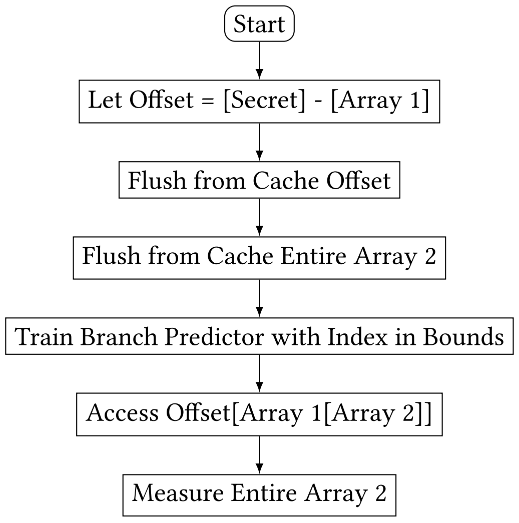
Start
Let Offset = [Secret] - [Array 1]
Flush from Cache Offset
Flush from Cache Entire Array 2
Train Branch Predictor with Index in Bounds
Access Offset[Array 1[Array 2]]
Measure Entire Array 2

Figure 1: Variant 1

Ongoing research continues to reveal further nuances of the Spectre family, pushing continual updates in both software mitigation and hardware design to secure systems against these pervasive threats with extensive efforts from software developers, hardware manufacturers, and security professionals.

NetSpectre enters the scene in 2019 with a novel idea to expose data cache as a side-channel over a network with its own flavor of information leaking gadgets [9]. In this paper, we are going to first reproduce the original Spectre to read a secret array from the side-channel. Based on the lessons learned, we will then explore the constituent parts of NetSpectre complete with a network analysis and develop our own idea of achieving the goal of NetSpectre through an amplified side-channel.

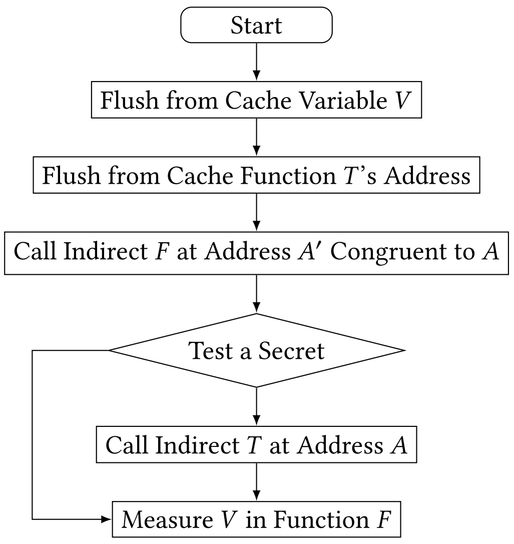
Start
Flush from Cache Variable V
Flush from Cache Function T's Address
Call Indirect F at Address A' Congruent to A
Test a Secret
Call Indirect T at Address A
Measure V in Function F

Figure 2: Variant 2

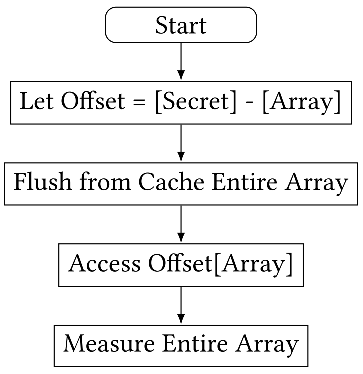
Start
Let Offset = [Secret] - [Array]
Flush from Cache Entire Array
Access Offset[Array]
Measure Entire Array

Figure 3: Variant 3

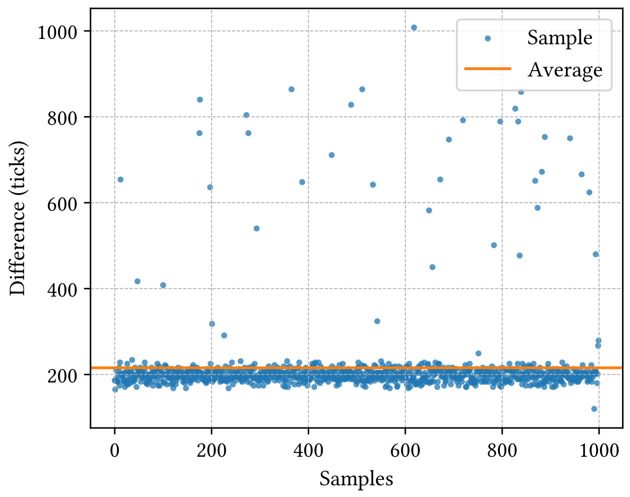

Figure 4: Difference in access time.

2 SPECTRE VARIANT 1

In order to gain the experience and confidence necessary to examine and explain NetSpectre, we decided to reproduce the original Spectre first. This proved a much more difficult task than at first imagined, but the wide-spread fame of Spectre convinced us from time to time that any mistake was likely the mistake of our own, and a working implementation came only after a long way of thinking and understanding the architecture of CPU's.

2.1 Setup

Since Spectre was discovered on Intel, we were compelled to use Intel CPU's. In preparation for the eventual evaluation of NetSpectre, we procured a tiny EC2 instance of virtual machines on Amazon's cloud free of charge. Surprisingly, this configuration proved useful in later experiments as it has only one CPU core of Intel Xeon CPU E5-2676 v3 (code-name Haswell) and has a small amount of usable memory of 1 gigabyte. Although most experiments are repeated on a local machine with Intel Core i9-10900K (code-name Comet Lake) for verification, only the results from the tiny EC2 instance running Amazon's own Linux (based on Red Hat Enterprise Linux) will be reported because it is the only complete set.

2.2 First Experiment

First of all, we needed to verify the reliability of data cache as a side-channel, and the most obvious way is to access a memory location with the `mov` instruction. To flush a cache line (from the entire hierarchy), Intel provides the `clflush` instruction. There is also the `RDTSC` instruction as well as the POSIX `CLOCK_MONOTONIC` with its corresponding API `clock_gettime` at our disposal to measure time. Although the experiment have been performed in both flavors, we are including only the results from the `RDTSC` instruction because they turn out to be much more stable.[1]

Figure 4 shows the additional time measured in reference cycles (at its nominal frequency) the CPU takes to access a memory location not in the cache hierarchy (L1–L3). Here, numbers over 10,000 were filtered as they were considered artifacts of context switches and were rare to begin with (in the thousandth). Note that the average difference is more or less uniform at a bit over 200 cycles, and it translates to slightly less than 100 nanoseconds for our tiny EC2 CPU with base frequency 2.40 GHz. This is an important number as its relative quantity in regard to the network's latency plays a critical role in assessing NetSpectre.

[1]https://github.com/coms6424-s24/Spectre/blob/main/tsc/mov.c

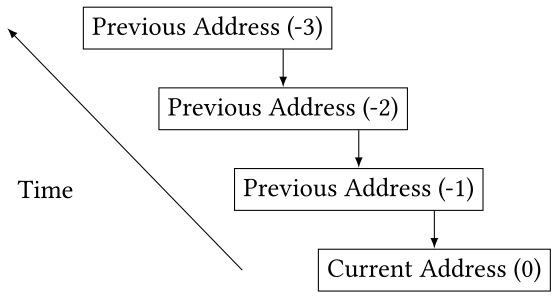

Figure 5: Branch history.

During this stage, we observed the out-of-order nature of the Intel CPU first-hand and learned to use memory fences to our advantage. Some weird effects also popped up that made us wonder the vast complexity of the underlying architecture. Why is the timestamp counter always a multiple of 3 from RDTSC, for example?

2.3 Main Experiment

Next, we were about to tackle the actual Spectre but were prudent enough to build it up from a version that accessed an element in a known array and inferred the value of the element through the side-channel. Here, we observed that the 64 bytes in a cache line are bundled in eviction. Hence, the useful elements in the array must stay 64 bytes away from each other. In the same manner, if we do not want two memory locations to be evicted together, we have to manipulate their locations by inserting extraneous elements between them or keep them in different segments entirely, for example, in rodata versus data.

Later, we made the array larger but were puzzled to see that certain elements striding a constant number away were either cached at the same time. Upon careful examination of the code, we noticed that the stride came from our own parameters. Given that the content of the global array was not specified in C and would be placed in the bss segment to take the value of all 0's upon execution, the entire array would then be repeatedly mapped to the same 0 page. As a workaround, we manually set the array elements to 0 but any value would do.

Soon, we discovered another trick to stabilize the flow of our program: we had to load some variables into cache prior to the main experiment (just like we had to keep some others out) and to execute any critical function once before to leave its footprint in cache so that the future runs will take a consistent amount of time. Meanwhile, we found out that the addition of printf statements in C had the nice property of improving the success rate of our program by making its execution less out-of-order and hence closer to our imagination.

2.4 Branch Prediction

As soon as we added the conditional statement to the subroutine we affectionately dubbed sub and attempted to read beyond the length of the array, we ran into a real obstacle. As a matter of fact, we spent a disproportionate amount of time on this small piece of code.

Previously, we have learned that branch prediction is based on history, but its actual mechanism is buried under the hood. At first, we assumed a naïve implementation such that, from a jump instruction, it would go to the address last jumped to. That is, if we had called sub with a proper index in bounds, the next call to sub would slip into the body of the conditional statement regardless of the actual index; but our assumption failed. Consequently, we thought that we were not trying hard enough and issued multiple such calls in sequence and then in a loop; but without success.

Eventually, we realized that our imagination for the branch predictor did not match reality. A not-so-quick search online found us the details of various Spectre variants in a concise format [2], where the branch predictor is pictured to use more than one entry in history. Immediately, it occurred to me that the branch predictor could look at the previous address the instruction had jumped from. Figure 5 illustrates the history information used to index a single branch on an Intel CPU.

Further search online landed us on the original paper of Spectre [4], where the depth of history for the so-called branch history buffer is revealed to be 5. Our experiment showed that 1 less time led to no success, but training the branch predictor 5 times from the same address was enough to ensure success.

2.5 Final Assembly

Finally, we had something working, even though the printout supposed to belong to a single pass of the loop seemed to contain effects of later passes, indicating that the depth of out-of-order execution is longer than we imagined. In the process of tuning the parameters, we made the observation that the initial condition of the program highly influenced its execution such that if the first call to sub produced garbage, later calls were most likely going to throw similar garbage, hinting at the CPU's nature as a deterministic machine.

Last, we turned our attention to those extraneous printf statements, and conceived of ways to replace most of our wait statements with data dependency or C language features to produce highly-readable code of a simpler structure. It was through this exercise we observed that the memory fences could not stand in for busy-waiting at the end of a loop; it appeared that they did not wait for instructions before themselves in the linear order of the program as in its text segment.

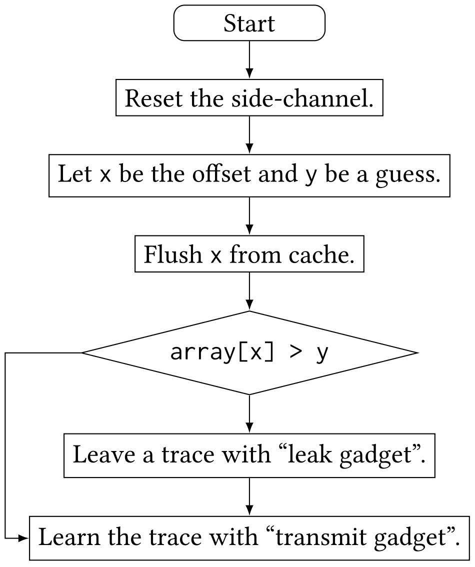
Start
Reset the side-channel.
Let x be the offset and y be a guess.
Flush x from cache.
array[x] > y
Leave a trace with "leak gadget".
Learn the trace with "transmit gadget".

Figure 6: NetSpectre

No way are we claiming perfection of our implementation, as a trivial change in the sense of a high-level language could break it, and it needs more busy-waiting between training the branch predictor and reading the results on Intel Core i9-10900K. Nevertheless, it worked reasonably well on the tiny EC2 instance and printed the entire secret array almost every time.[2]

3 SPECTRE VARIANT NET

Now, we are ready to look further into NetSpectre. Overall, it follows the same structure as the original Spectre but has its own "leak gadget" in place of the cascaded array access with different options for the body of the conditional statement. Listing 1 and 2 quote from the paper the pseudo-code for the implements [9]. Figure 6 shows the workflow.

Listing 1: Leak Gadget 1

```
if (x < length)
    if (array[x] > y)
        flag &= true
```

Listing 2: Leak Gadget 2

```
if (x < length)
    if (array[x] > y)
        _mm256_instruction();
```

3.1 First Experiment

Since AVX instructions are proposed as a new side-channel, we need again to verify the reliability. Fortunately, the paper gives the VPAND instruction as an example. Using the same code in our previous experiment as a baseline, we were not able to reproduce a difference in access time on any of our computers despite substantial effort.

Following this experience, we traced the reference in the paper to a technical blog with actual steps [1]. In summary, we need to call a floating-point AVX instruction such as VADDPS on (the upper half of) the 256-bit registers in a tight loop, and the throughput is only half as usual during an initial period. Although we got it to work on the tiny EC2 instance (on Haswell), we were unable to reproduce the same on Intel Core i9-10900K (on Comet Lake).[3]

Furthermore, we wonder if floating-point AVX is used often enough in network stack, which is the target assumed by the paper. However, the difference manifested can be more than 10,000 cycles, depending on the depth of the loop, and sleeping in the middle does reset the behavior for another try as suggested by the paper [9].

3.2 Main Experiment

Note that the workflow of NetSpectre lends itself well to binary search with a different guess each time. Once again, we took baby steps and wrote a binary search that directly accessed the content of the array. Once we made sure that the structure worked, we threw in the leak gadget and were able to infer the array up to its length.

At this time, we were still confident that we only needed a few tweaks for success, but the more we looked at it, the more our attention was drawn to the second `if` statement in the "leak gadget". Knowing that the CPU always runs speculatively, we deduced that the second `if` statement is also under the influence of speculative execution, and the leak gadget would do its job based on the act of the branch predictor over the values of x and y from previous iterations.

Furthermore, we need an in-bounds x in order to train the branch predictor. Given that our code uses 0 for x on the array that literally reads "Secret Password", indeed it is 'S' that is printed for each remaining character beyond the length of the array, coinciding with our theory.[4]

Logically speaking, we need the first `if` statement to close its eyes and the second `if` statement to open its eyes, but the second `if` statement may not even have a change to resolve its true value if the first re-steers first. Therefore, the first `if` condition must have a longer chain of dependency in order for the whole thing to work.

3.3 Memory Experiment

Last but not least, it is worth mentioning that NetSpectre tries not to use the `clflush` instruction and instead emulates

[2]https://github.com/coms6424-s24/Spectre/blob/main/spectre/v1.c

[3]https://github.com/coms6424-s24/Spectre/blob/main/tsc/avx.c

[4]https://github.com/coms6424-s24/Spectre/blob/main/spectre/v0.c

the download of a 2–3 megabytes file to flush x from cache. In fact, it worked quite well and probably evicted entries of the page table in the process. In our experiment, where an 8 megabyte array is sequentially accessed 8 bytes at a time, it consistently reproduces a difference in access time over 500 nanoseconds. Enlarging the array can theoretically amplify the difference but can then take too long to exclude interference in the middle.[5]

4 NETWORK ANALYSIS

The Net part of NetSpectre attempts to use the "leak" and "transmit" gadgets remotely. Since a small signal can be easily buried in a large amount of noise, the attack depends heavily on network latency: a latency too high or too variable violates the precise timing requirement for successful exploitation.

4.1 Setup

In hope of reproducing the effect of NetSpectre, we followed the approach in the paper to send 1,000,000 pings over the wide area network (WAN) from our local machine in New York to our tiny EC2 instance in Northern Virginia. This setup reflects a typical interstate data path, providing real-world context for our analysis. In order to have something to compare with, we sent another set of 1,000,000 pings. To cover the case for local area network (LAN), we followed the paper to send 100,000 pings each time.

Standard `ping` tools on Windows, Mac OS X, and Linux offer precision in milliseconds, but NetSpectre demands the detection of differences as small as 500 nanoseconds (per our previous experiment). As a result, we developed our own `ping` tool to send echo requests, whose results coincide with but are more precise than standard tools.[6]

Note that echo requests are part of the Internet Control Message Protocol [7] and operate on the Network layer. Therefore, Echo Requests constitute the most direct method of measuring network latency, and the time variation of the actual implementation of NetSpectre can hardly be less.

During initial testing period, we found that latency was more stable at night. Consequently, we chose to conduct our data collection at night time to increase our chance.

4.2 Latency Analysis

Figure 7 and 8 each gives a zoomed-out view of the latency distribution. There are some similarities between the batches, showing an apparent consistency across the measurements. However, upon closer scrutiny of the means and standard deviations of the WAN cases, as shown in Figure 9 and 10, discrepancies started to appear. Specifically, the difference between the means of 2 data sets is over 0.25 milliseconds

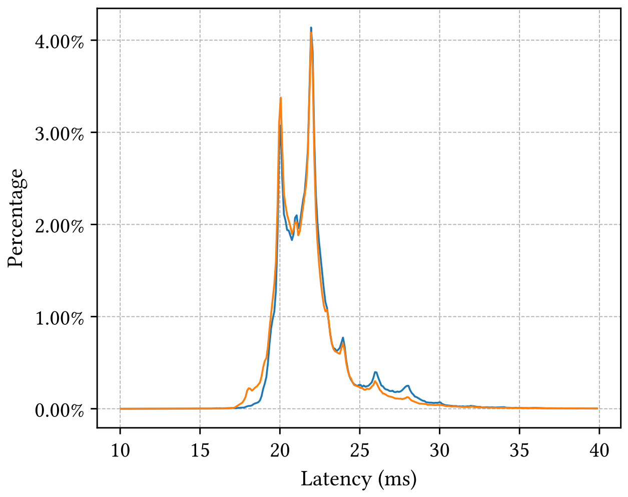

Figure 7: Frequency distribution of WAN latency.

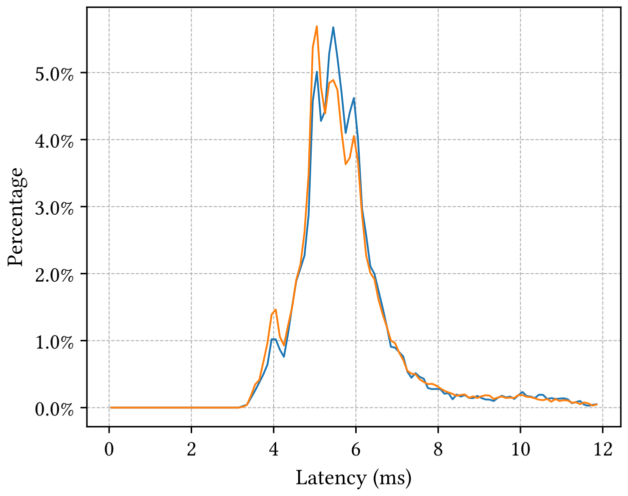

Figure 8: Frequency distribution of LAN latency.

(250 microseconds), orders of magnitude higher than our measured resolution of the side-channel (500 nanoseconds) or the 500–1000 cycles (200–400 nanoseconds) indicated in the paper [9].

On the other hand, the standard deviations, once grouped in time order, vary between 3 and 6 milliseconds, indicating substantial variability that complicates our analysis. In fact, the standard deviation in Figure 10 does not tend to stability as we zoom out from the 100,000 groups to the 1,000,000 collection. Following this fact, we wonder if larger sample sizes are inherently more stable for our purpose. In other words, if we were given 8 more sets of 1,000,000 pings, would

[5]https://github.com/coms6424-s24/Spectre/blob/main/thrash/v0.c
[6]https://github.com/coms6424-s24/Spectre/blob/main/net/ping.c

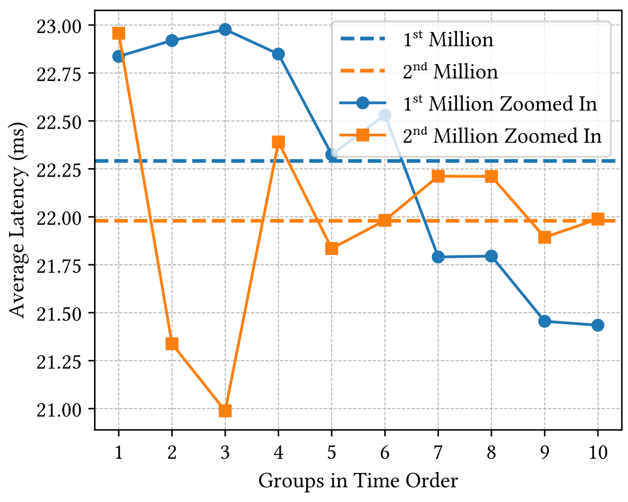

Figure 9: Average latency grouped over time.

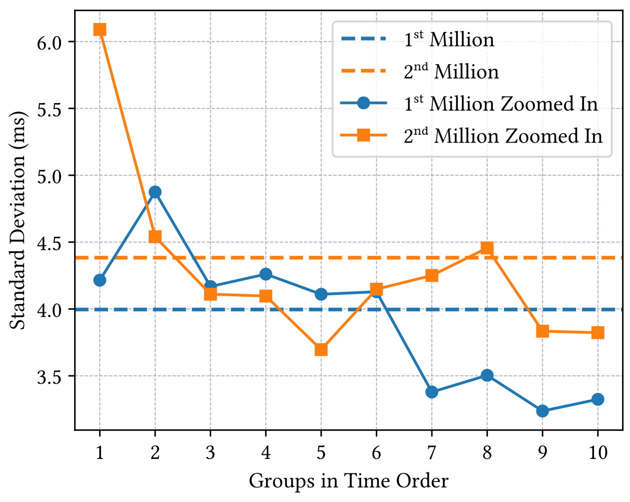

Figure 10: Standard deviation grouped over time.

we see a collective deviation smaller than our existing 2 sets at all?

Since the authors of the paper have employed a naïve method of averaging over a large number of measurements to analyze the data, it seems that they are assuming the independence of individual measurements (to invoke the law of large numbers). However, measuring network latency is hardly an independent event due to the presence of queues in each node in the middle. Moreover, the multi-modality of the observed distributions does not lend itself to such a simple method, either.

Although the paper mentions the use of Bayes classifiers as a potential way of analyzing the same data, it provides no substantive details on its implementation. What are the model and the features, for example? To explore this avenue, a rigorous statistical analysis might be necessary.

5 AMPLIFICATION

Given the circumstances, it seems best to amplify the side-channel so that it can be easily seen over a noisy base channel. While such a method is mentioned in the original Spectre paper [4], we have taken inspiration from Google's security team to exploit the pseudo-LRU policy of CPU cache [8].

5.1 First Experiment

First, we wanted to cross-check our understanding of cache with reality and to verify the information from the Linux utility `cpuid` (that the tiny EC2 instance has an 8-way 64-set L1 data cache of 32 kilobytes). Therefore, we made an experiment to iterate through 8 elements, each 64 cache lines apart, to evict an existing element in cache.

When the experiment initially showed no difference, we were puzzled for a bit. Soon, we realized that the timestamp counter returned by `RDTSC` had some minimum resolution, and the smallest number we have seen is 18. In fact, `RDTSC` returns 18 for a lone memory fence. As soon as we changed the parameters to work for the L3 cache, we saw the expected difference.[7]

As a side note, the same code did not work on Intel Core i9-10900K, because its cache is no longer inclusive: getting some element out of the L3 cache does not get it out of L1.

5.2 Second Experiment

Next, we verified the access sequence on [8] used to exploit an 8-way associative cache. This step turned out easier than expected due to our previous effort in understanding its mechanism. Table 1 shows the conversion of the indices from the source Web page to our code.[8]

Here, we have used the L1 cache as we want to see a small effect amplified. With 8 iterations (of going through the same sequence) only, we are able to see on average 30 cycles of difference. When the number of iterations is too large, however, context switch often kicks in to interfere with the results. Therefore, with the L1 cache, it seems difficult to arbitrarily amplify the effect of a hit versus a miss.

Note that it sometimes fails and returns results contrary to our expectation, but such cases are minor. For now, we are blaming it on the only core of the tiny EC2 instance being shared by all processes on the system.

[7]https://github.com/coms6424-s24/Spectre/blob/main/cache/test.c
[8]https://github.com/coms6424-s24/Spectre/blob/main/cache/plru.c

Table 1: Index look-up.

A	B	C	D	K	F	G	H
0	1	2	3	4	5	6	7
X	A	B	C	K	D	F	G
8	0	1	2	4	3	5	6
K	H	A	B	K	C	D	F
4	7	0	1	4	2	3	5
K	G	H	A	K	B	C	D
4	6	7	0	4	1	2	3

5.3 Remaining Experiment

By now, we have run out of time. Ideally, we would then make a “leak gadget” that combines some version of Spectre with the above as well as a “transmit gadget” for the Net variant of Spectre. Furthermore, we would like to think over the sequence in Table 1 and come up with a pattern.

6 CONCLUSION

This paper has offered an introduction to Spectre by first measuring the reliability of data cache as a side-channel and then reproducing the secret stealing capability of the original Spectre through the side-channel. During the process, Intel’s branch predictor was studied and successfully exploited, and various other details of the CPU architecture were revealed.

Attention was then turned to NetSpectre with attempts made to reproduce the results in the paper [9]. Although we made the “leak gadget” to work with binary search, we were unable to read past the length of the array and gave an explanation of the phenomenon instead. Next, we went ahead to perform a network analysis only to find that, in our setup, the latency would be so high and variable that a small signal could hardly be visible with a simple comparison between random samples.

Following this event, we offered a way forward to achieve the stated goal of NetSpectre by amplifying the side-channel. Specifically, we performed the pseudo-LRU exploit on the data cache with the access pattern found by Google [8] and observed a difference that could be multiplied arbitrarily in a loop. Naturally, the next step would be to understand the sequence and combine it with the overall exploit.

Through reproduction of published results (or failure to do so in certain cases), we have demonstrated the principles of Spectre and shown that any serious attempt at such attacks requires a deep understanding of the underlying hardware.

Last but not least, by constantly thinking about computer architecture in this project, we were able to develop intuition that would otherwise take a long time to form. May it be the goal of this project or not, by deepening our understanding of the principles of such attacks, we are better prepared for the future to design secure systems regardless of whether we work in the specific area of computer security or not.

REFERENCES

[1] Agner Fog and John D. McCalpin. 2015. *Test results for Intel’s Sandy Bridge processor.* Agner’s CPU blog. https://www.agner.org/optimize/blog/read.php?i=378

[2] Gavin Guo. 2017. *Spectre (v1/v2/v4) vs. Meltdown (v3).* Linux Foundation. https://events19.linuxfoundation.cn/wp-content/uploads/2017/11/Understanding-Spectre-v2-and-How-the-Vulnerability-Impact-the-Cloud-Security_Gavin-Guo.pdf

[3] Jann Horn. 2018. *Reading privileged memory with a side-channel.* Google Project Zero. https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html

[4] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2020. Spectre attacks: exploiting speculative execution. *Communications of the ACM* 63, 7 (June 2020), 93–101. https://doi.org/10.1145/3399742

[5] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre returns! speculation attacks using the return stack buffer. In *Proceedings of the 12th USENIX Conference on Offensive Technologies* (Baltimore, MD, USA) *(WOOT’18).* USENIX Association, USA, 3.

[6] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg, and Raoul Strackx. 2020. Meltdown: reading kernel memory from user space. *Communications of the ACM* 63, 6 (May 2020), 46–56. https://doi.org/10.1145/3357033

[7] Jonathan B. Postel. 1981. *Internet Control Message Protocol.* Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/rfc792

[8] Stephen Röttger and Artur Janc. 2021. *A Spectre proof-of-concept for a Spectre-proof web.* Google Security Team. https://security.googleblog.com/2021/03/a-spectre-proof-of-concept-for-spectre.html

[9] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. 2019. NetSpectre: Read Arbitrary Memory over Network. In *Computer Security – ESORICS 2019: 24th European Symposium on Research in Computer Security, Luxembourg, September 23–27, 2019, Proceedings, Part I* (Luxembourg, Luxembourg). Springer-Verlag, Berlin, Heidelberg, 279–299. https://doi.org/10.1007/978-3-030-29959-0_14