Spectre Side Channel Attack

Hrishikesh Jedhe Deshmukh
210050073

Khushang Singla
210050085

Sankalan Baidya
210050141

***Abstract*—In the ever-evolving landscape of cybersecurity, the emergence of sophisticated threats has compelled us to reassess and fortify the modern computing systems. Spectre attacks have garnered significant attention due to their ability to exploit vulnerabilities arising from speculative execution. Almost every contemporary microprocessor architecture is vulnerable to spectre attacks. Spectre attacks, a class of side-channel attacks, exploit the speculative execution feature present in almost all modern processors. By manipulating the processor's speculative execution pathways, malicious actors can leak sensitive information from the victim's memory. This paper delves into Spectre attacks on without use of shared memory. Not sharing memory is typically considered secure against side-channel attacks. However, we demonstrate Spectre attack without using shared memory to leak the victim's private information. We aim to shed light on the intricacies of the Spectre attack without any shared memory and contribute to the ongoing effort of making systems secure against these speculative attacks.**

1. Introduction

Computations performed by devices often leave observable side effects beyond the normal output. Side channel attacks focus on exploiting these side effects to extract sensitive information.

Speculative execution is one of the design techniques that has facilitated the increase in processor speed over the last decade. This technique is used in almost every modern processor and it guesses the likely future execution path and prematurely executes instructions on this path. Consider an example where program's control flow depends on uncached value present in the memory. Instead of idling, processor guesses the path and then executes the instructions on this path speculatively. Whenever the value is fetched from memory, processors checks if its guess was correct. If it was indeed correct, it continues on the same path, otherwise it discards the results of the speculative execution and resumes execution from the correct path.

Kocher et al.[2] demonstrated the viability of speculative attacks on modern processors using a proof of concept code which contains both attacker and victim in a single process. Attacker and victim also accessed a shared array to make this speculative attack possible. We extended this code to make possible spectre attack in a multi-process environment where attacker and victim are in different processes and access shared memory. Then we also demonstrated this attack on non-shared memory by using prime and probe to retrieve the data brought in cache hierarchy by *transient instructions*. Transient instructions are those instructions which are speculatively executed but then the effect of these instructions on the CPU is reverted back.

We have extended the original spectre attack code to exploit the vulnerability when attacker and victim are in two different processes while sharing some memory. Also we showed viability of this attack in unshared memory setting, by keeping both attacker and victim in same process without accessing each other's memory.

2. Related Work

2.1. Spectre attack

Original proof of concept code has a victim function in which transient instructions are exploited by attacker to retrieve secret data. The function is as follows:

```
void victim_function(size_t x) {
   if (x < array1_size) {
      temp &= array2[array1[x] * 512];
}
```

Branch predictor is initally mistrained to always predict that the `if` statement is true by providing valid inputs of `x`. Then attacker provides an invalid input of `x` to the victim function. This causes the branch predictor to mispredict and speculatively execute the `if` statement. The transient instruction `temp &= array2[array1[x] * 512]` brings the data at `array2[array1[x] * 512]` in the cache hierarchy. Attacker then accesses `array2` indices in a loop and measures the time taken to access each index. Note that `array2` is shared while `array1` is not. If the time taken to access an index is less than a threshold, it means that the data at that index was brought in the cache hierarchy by the transient instruction. This way attacker can retrieve the secret data.

Important thing to consider while carrying out spectre attack is to not have speculative instructions before we call victim functions. This is because the processor will speculate even these instructions before it has chance to specukate the instructions in victim function. This leads to lesser accuracy in the attack.

2.2. Prime and Probe attack

Cache sets accessed by victim can be detected using prime and probe [1]. In this attack, attacker first primes the cache by accessing a set of cache lines. Then attacker waits for the victim to access the same set of cache lines. Attacker then probes the cache to see which cache lines are still in the cache. The cache lines which are not present in the cache are the ones accessed by the victim.

A linked list is used to prime the cache. After victim accesses a cache set, the cache is probed using the same linked list. Linked list is used to do prime and probe to avoid the use of memory fences.

3. Methodology

3.1. Spectre attack as covert channel

Original code had both attacker and victim in the same process. We separated them both into different processes while retaining shared memory. Instead of `array2`, which acted as shared memory in the original proof of concept code, we mapped a file to both processes. This file now acts as shared memory, having the exact same functionality of `array2`.

Both attacker and victim are forked and execed by a parent process. Both processes need to be mapped to the same core of the CPU for this attack to work as we are detecting the secret data by measuring its latency of access. In the victim process, `array2` points to the mapped file. In the attacker process, `array2` points to the same mapped file. Thus we can use the memory of the file as shared memory to probe for lines which are brought into cache speculatively.

3.2. Spectre without shared memory

We modelled spectre attack without shared memory by keeping attacker and victim in same process. Attacker and victim do not access each other's data structures. Attacker first primes the cache by accessing a certain number of linked list nodes. Then attacker calls the victim function with valid input. Then attacker probes the cache by accessing the same linked list nodes. This is done repeatedly to mistrain the branch predictor. Then on the last iteration, attacker calls the victim function with invalid input. This causes the branch predictor to mispredict and speculatively execute the `if` statement. Attacker then probes the cache to see which cache lines have been accesses by the victim.

Prime and probe is done on L1D cache here. This is possible due to the fact that attacker and victim reside in the same process.

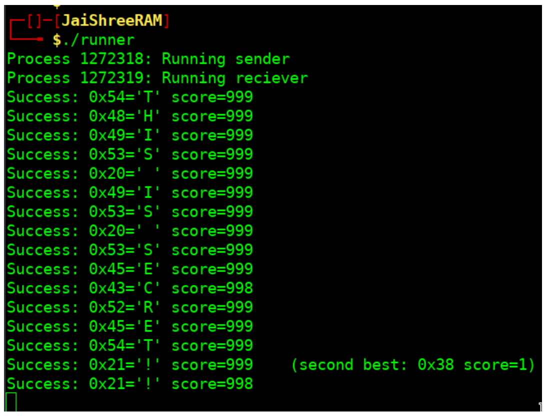
3.3. Spectre without shared memory - multiple processes

We tried separating attacker and victim into two different process while not having any shared memory. Method is same as described before. Attacker primes the cache, then calls the victim function and then probes the cache to find the lines accessed by victim. We also mapped both the processes to the same core. But this attack was not successful.

4. Results

4.1. Covert channel

We were able to achieve very high accuracy on AMD Zen 3 and Haswell for the case when attacker and victim are in two different processes but have shared memory. For each byte, we had 1000 iterations to detect it. Score in the below figure denotes the number of times out of 1000 that the byte was detected. Score of 999 in almost all cases shows that the attack is very accurate.



```
┌─[]─[JaiShreeRAM]
└──╼ $./runner
Process 1272318: Running sender
Process 1272319: Running reciever
Success: 0x54='T' score=999
Success: 0x48='H' score=999
Success: 0x49='I' score=999
Success: 0x53='S' score=999
Success: 0x20=' ' score=999
Success: 0x49='I' score=999
Success: 0x53='S' score=999
Success: 0x20=' ' score=999
Success: 0x53='S' score=999
Success: 0x45='E' score=999
Success: 0x43='C' score=998
Success: 0x52='R' score=999
Success: 0x45='E' score=999
Success: 0x54='T' score=999
Success: 0x21='!' score=999    (second best: 0x38 score=1)
Success: 0x21='!' score=998
```

Figure 1. Demonstration of spectre as covert channel

4.2. Prime and Probe

We used the existing code to carry out prime and probe attack on L1D cache. Even in this case attacker and victim reside in the same process. Prime and probe is highly accurate on Haswell architecture but very inaccurate on Alder Lake and Zen 3. We were able to detect the cache lines accessed by the victim with almost 100% accuracy.

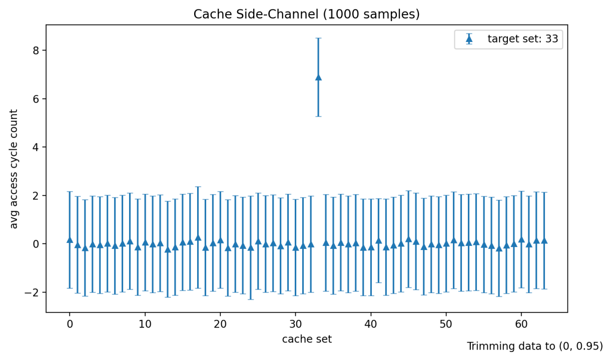


Figure 2. Normalized access times of cache sets

Above figure 2 shows *normalized* access time of cache sets on Y-axis and set numbering on X-axis. To normalize the times, we first find access times of all cache sets without victim accessing any cache set. Then we subtract these times from the access times of cache sets when victim accesses them. This gives us the normalized access times. We can see the cache set accessed by victim has considerably higher access time than other cache sets. This shows that prime and probe is highly accurate on Haswell architecture.

4.3. Spectre on unshared memory - single process

We achieved very high accuracy on Haswell by combining spectre and prime and probe for the case where memory is not shared. Owing to the fact that prime and probe is inaccurate on Alder Lake and Zen 3, we were not able to achieve high accuracy on these architectures. We were able to detect the secret byte with 100% accuracy on Haswell.

As we can observe in the Figure 3, the secret byte is always detected with 100% accuracy. But with it, prime and probe also detects few more bytes. But these are constant over every iteration and thus can be filtered out easily. This noise is present owing to the fact that victim also accesses few more memory lines apart from the secret byte, so prime and probe also detects these unwanted lines.

4.4. Spectre on unshared memory - multiple processes

We were not able to achieve high accuracy on any of the architectures for this case. Prime and probe is still able to detect the line accessed by victim, but with it many more lines are also being detected. The noise is very high in this case. This is due to the fact that a context switch is needed before the attacker can probe the cache. This context switch brings in many more lines in the cache, thus creating this noise.

5. Observations

5.1. Spectre on AMD and Intel

Spectre attack works with high accuracy on AMD processors (we tried on Zen 3), but is extremely noisy on Alder Lake. While mistraining the branch predictor, pseudo-random access pattern is used to stop prefetcher from detecting it. But prefetcher present on Alder Lake is still able to detect this access pattern and prefetches the secret bytes even before they are speculatively brought into cache. This leads to very high noise in the attack.

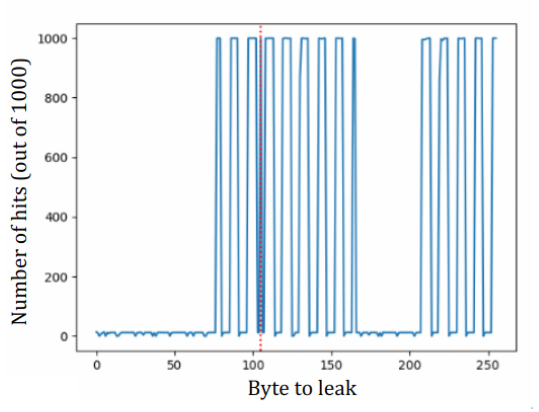


Figure 4. Detection of secret bytes on Alder Lake

In the above figure 4, Y-axis denotes the number of times the byte was detected out of 1000 iterations. X-axis denotes the byte number. Red dotted line denoted the secret byte that we want to leak. We can see that a peak is present on the red line, denoting that the secret byte is being detected. But at the same time lot of other bytes are also being detected. This is the noise present in the attack. This noise is very high on Alder Lake because of its prefetcher.

5.2. RDTSC binning

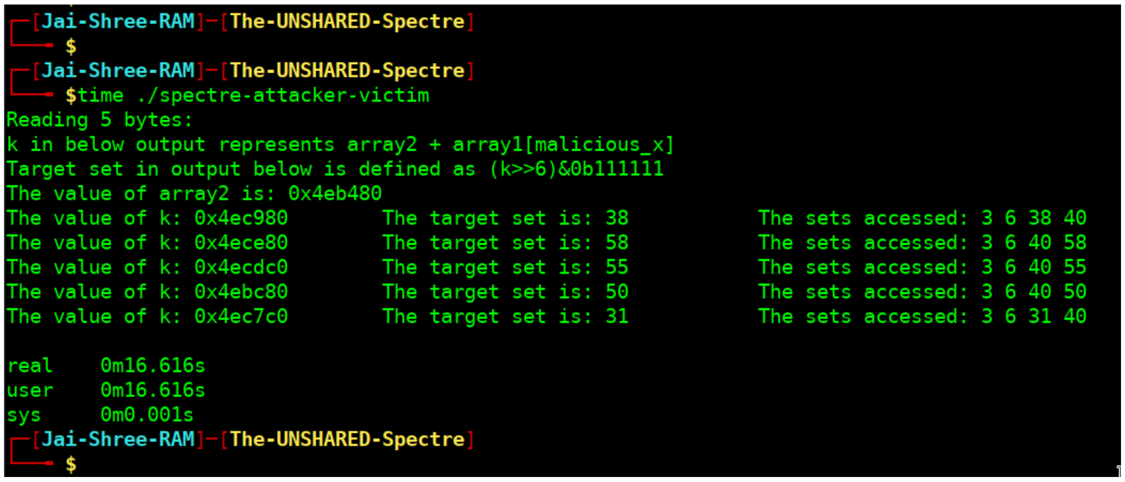
We also found out that RDTSC on Zen 3 processors always returns the value which is a multiple of 32 or 33. AMD might have implemented binning of RDTSC to avoid timing attacks on its processors. Intel processors show no suggestion of such a binning being implemented.

5.3. Flush variables involved in speculation

While carrying out spectre attack, in the speculative condition of `if(x < array_size)`, it is necessary to ensure that the line corresponding to `array_size` is not present in cache hierarchy. This ensures that the transient instructions have enought time to load the secret memory into cache hierarchy before they are squashed. Although hard to verify, this observation also hints towards the fact that the transient instructions are squashed as soon as the branch condition result is obtained. Processor does not wait for these transient instructions to reach the head of Reorder Buffer before squashing them.

5.4. Branch Predictor

While mistraining the branch predictor on Haswell, we unrolled the for loop so that the victim function gets called by different instruction pointers. Spectre attack accuracy dropped massively after doing this. Branch predictor was not getting mistrained due to victim function being called by different instruction pointers. This shows us that branch predictor likely maintains a history of instruction pointers from which the victim function was called. Having different



```
┌─[Jai-Shree-RAM]─[The-UNSHARED-Spectre]
└──╼ $
┌─[Jai-Shree-RAM]─[The-UNSHARED-Spectre]
└──╼ $time ./spectre-attacker-victim
Reading 5 bytes:
k in below output represents array2 + array1[malicious_x]
Target set in output below is defined as (k>>6)&0b111111
The value of array2 is: 0x4eb480
The value of k: 0x4ec980        The target set is: 38          The sets accessed: 3 6 38 40
The value of k: 0x4ece80        The target set is: 58          The sets accessed: 3 6 40 58
The value of k: 0x4ecdc0        The target set is: 55          The sets accessed: 3 6 40 55
The value of k: 0x4ebc80        The target set is: 50          The sets accessed: 3 6 40 50
The value of k: 0x4ec7c0        The target set is: 31          The sets accessed: 3 6 31 40

real    0m16.616s
user    0m16.616s
sys     0m0.001s
┌─[Jai-Shree-RAM]─[The-UNSHARED-Spectre]
└──╼ $
```

Figure 3. Demonstration of spectre without sharing of memory

instruction pointers likely causes this history to get fuzzy leading to branch predictor not getting mistrained.

5.5. Prime and Probe using linked list

We also tried prime and probe attack without using linked list. Instead we used memory fences to ensure that the next memory access happens after the previous access in completed. But this approach does not work, likely due to memory fences adding heavy overloads of their own leading to wrong timing measurements. Linked lists naturally ensure that the next memory access happens after the previous access is completed due to their structure. Thus, using linked lists for prime and probe is superior to using memory fences.

References

[1] Miro Haller. Cachesc. https://github.com/Miro-H/CacheSC, 2020.

[2] 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. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2019.