Spectre Attack

Romain de BEAUCORPS, Jules TIMMERMAN

1 Exploiting a modern CPU

Memory on a system is compartmentalized, meaning applications have their own address space that usually cannot be accessed from another process. The spectre attack [4] exploits critical vulnerabilities in how processors execute instructions, allowing the attacker to leak memory which is normally unavailable from their perspective. Even worse, it can manipulate error-free programs into disclosing secret data, even though the programmer followed best practices.

1.1 Speculative execution

Modern processors have been getting faster and faster through techniques such as speculative execution, which allows the processor to put its idle time to use in order to squeeze out as much performance as possible. When the program's control flow depends on a branching instruction relying on an uncached value, a naive implementation would simply be to wait for the value to be fetched from memory and then resuming the program's execution. A modern processor tries to use this idle time by basically guessing the result of the branching instruction, and continuing the execution on the path it thinks is the most likely. In the event of a correct guess, it simply keeps executing and we have gained a lot of otherwise wasted time. If it didn't guess correctly, it returns to its previously saved state and goes on.

1.2 A design flaw

Speculative execution therefore allows us to execute code that shouldn't be executed, however any negative side effects resulting from this malicious execution are supposedly overwritten by the CPU. So, is there a way to exploit this mechanism ? The answer is yes, and it is all possible thanks to memory caching, which we will see next.

1.3 Reading from Cache versus from Memory

Since reading from memory is slower than a CPU instruction by orders of magnitude, modern CPUs use little, very fast memory units called caches, physically located near the CPU and thus much faster than regular memory. When reading at a memory address, this address and the surrounding ones are stored in the cache, to try and anticipate future reads or writes at that address. The interesting thing is, when wrongfully executed instructions get rolled back, everything is restored except the cache. The exploit is then to read a normally unavailable value, and using it to load an address in the cache that we control.

2 Building it from the ground up

We used the following code to measure the number of CPU cycles required to access a value.

```
        int junk = 0;
            register uint64_t time1, time2;
            for(int i = 0; i < 10; i++){
                    volatile uint8_t * addr = &array[i*4096];
                    register uint64_t time1 = __rdtscp(&junk);
                    junk = *addr;
                    register uint64_t time2 = __rdtscp(&junk) -
↪   time1;
                    printf("Access time for array[%d*4096]: %d
↪   CPU cycles\n",i, (int) time2);
            }
```

Listing 1: Measurement of time taken to access data

The instruction `clflush`, that we use with the C function `_mm_clflush`, allows us to remove a given address from the cache, and the instruction `rdtscp`, called with the C function `__rdtscp`, is used to measure the number of CPU cycles that has elapsed.

We use an array made of blocks that are of size 4096, effectively spreading our data. As such, accessing a value in a block will only load said block in the cache. It won't load the other blocks in the cache.

We first initialize the array to a non-zero value to prevent any optimisation from the compiler or the OS. We then flush it from the cache and then access specific values from the array, effectively loading them in the cache. We then measure the access time for each values.

Here is an example of the results obtained from the execution in Table 1 :

Cell	Access time (in CPU cycles)
0	656
1	416
2	404
3	108
4	428
5	388
6	392
7	108
8	392
9	404

Table 1: Measure of access time for data in the cache and out of the cache

We can now see that the values we accessed, that is 3 and 7, are taking significantly less CPU cycles to be fetched while other values are taking more cycles due to the CPU having to access the memory instead of the cache.

3 Flush and Reload

The Flush and Reload paper [5] explains the properties of the attack. It is an upgrade from another attack that has a high success rate and is more precise, as it exploits memory lines. Because the Last-Level-Cache (LLC) is shared, it makes the Flush and Reload attack cross-core, and even cross-VM. The article then gives background information about page sharing, cache architecure and how RSA works. It describes afterward how the attack precisely works and uses it on a concrete example. It then gives some potential solutions to mitigate this attack (that we will present later).

We are trying to exploit the following code :

```
uint8_t array[256 * 4096];
const unsigned char secret; // Encrypted secret value

void victim(void){
    unsigned char tmp = _decrypt(secret);
    array[tmp * 4096];
    // Do stuff with array[tmp * 4096]
}
```

Listing 2: Victim's code

We use the code present in the `flushandreload.c` file [2]. Here, we use the same trick as in the first part. We prevent any potential optimisation from the compiler or the OS by initializing the array to non-zero values, then flush it to clean the cache and execute the `victim` function which will then try to access some values from the array, caching it at the same time. We then use the same measurement routine, using an empirically determined threshold (thanks to the first part) to print only the values that are most likely to be the one accessed.

When executing our program, we get the following output :

```
Access time for array[94*4096]:  92 CPU Cycles.
```

The secret is 94 !

As the Flush and Reload suggests [5], there are multiple solutions that could be used to mitigate this attack. Because hardware-based solutions would take too much time to be effective, software-based solutions are preferred. One solution would be to limit the access to the `clflush` instruction, limiting the possibilities to evict cache lines. You could also try to limit page sharing between processes, but it would increase the memory requirements, making that solution unfeasible. You could also try to use a technique known as software diversification to generate unique copies of programs, limiting sharing.

4 Out-of-order execution

We use the code in `outoforder.c`. We have refactored the code to have clear functions instead of having everything in the `main` function. We used the code from spectre paper as a base [3] [1].

The idea of the exploit is using the fact that addresses cached by a wrong speculative execution aren't deleted from the cache during the rollback. In the victim function, we have a conditional branch. Ideally, the variable used in the condition is a cache miss, meaning the CPU has to fetch the data from outside the cache and it will take some time. Meanwhile, it will start speculatively executing the instructions it "thinks" are the most likely to be the correct control flow. By training the CPU in taking the *true* branch, it will speculatively execute the code inside, even when the input shouldn't pass the *if* condition. `array2` will then be accessed and put in the cache. The CPU will then rollback instructions that were executed but it will not remove the data accessed from the cache, meaning the `x` block of `array2` will be in the cache.

We can now combine this idea with the Flush and Reload attack : we first train the CPU to take the *true* branch, then flush the array from memory to not get false positives. Afterward we try executing with a malicious `x` that doesn't verify the condition. While fetching `x` from memory, the CPU speculatively loads the block in the cache. We then measures to see that only the `x` block should have a lower access time.

Because this attack doesn't always work due to many different variables like CPU parameters (such as Inter Turbo Boost) or the frequency varying, we use a soft statistical approach. We repeat the attack multiple times and note the number of times the access time was a hit, that is to say, under a certain threshold.

Values measured are found in Table 2 :

Index	Number of hits (out of 1000)
91	0
92	3
93	7
94	9
95	11
96	0
97	998
98	9
99	11
100	12
101	11
102	12

Table 2: Out-of-order execution : the 97th block is accessed when it shouldn't

We use the `-O0` flag when compiling with GCC. We also use `cpupower frequency-set -g performance`

5 Spectre attack

The Spectre Attack paper [3] first gives background information about out-of-order and speculative execution, as well as how memory (and caches) works. It also gives a brief explanation about micro-architectural side-channel attacks and return-oriented programming (ROP), a technique used by redirecting code execution to use specific bits of code found in memory. It then gives an overview of the attack and gives a few variations of it. The one we are going to use is variant 1 where we try and exploit the misprediction of conditional branches. The second variant uses similar ideas as ROP by mistraining the branch predictor in going to certain addresses. Other variants are given that do not focuses on exploiting the cache. Finally, some potential solutions are given to mitigate this attack.

In the Spectre Attack, we exploit speculative execution from the processor to have it access sensitive data. We tried to exploit the code given to us in Listing 4 where we would trick the branch predictor into executing the *true* branch with a malicious `x` so that we would access the `keys.priv` field,

```
struct key_pair{
        uint8_t pub[16];
        uint8_t priv[16];
};
```

Listing 3: Structure containing keys for cryptographic purposes

For example, since the two fields are contiguous in memory, accessing index

16 of the `keys.pub` field would instead result in accessing the first value of the `keys.priv` field. This can be done by exploiting out-of-order execution in the `get_pub_element` function.

We can then try retrieving the value that was read using a Flush and Reload attack. The code used is present in the `spectre.c` file [2]. We use the same file structure as the out-of-order proof of concept from before. The `readIndex` is used to execute the attack with the offset passed as a parameter. We return the value that had the most cache hits.

We also load the `keys.priv` field in the cache before executing the attack. In our code example, we just force load it by accessing it but in a real world scenario, we could imagine that a function would use that value, effectively loading it in the cache (for example, an encryption function). This allows us to have a larger window for the attack as accessing the value during the speculative execution of `get_pub_element` will result in a cache hit. This hypothesis is plausible as we can imagine that there is a function from the library that uses the `keys.priv` field, effectively loading it into the cache.

Unfortunately, we weren't able to get any results from this attack, whereas we were having results with the proof of concept from the spectre paper [3]. We also tried changing our code so that it would resemble the proof of concept, that is, accessing an array directly from the `get_pub_element` function, effectively removing the need for the speculative execution to return from the function. This still didn't yield any results.

A potential source of problem might be the fact that the code resides in another library, although we didn't have time to do another experiment to check if it was a problem by moving the victim's code in the same file as the attacker.

Other reasons might have to do with things like prefetchers that could be having an effect.

As the paper explains [3], there are multiple ways we could mitigate the spectre attack. The obvious one would be to prevent speculative execution. While it would completely invalidate the spectre attack, it would have a significant impact on performance. You could also try to isolate the secret data. Indeed, the spectre attack uses the same privilege as the users so hiding sensitive data from the user would block the spectre attack. You could also try to limit the access to accessing the leaked data by either preventing access to the data if it was speculatively accessed, or by patching other ways to read the leaked data e.g. the Flush and Reload attack here. Finally, you could prevent the second variant by having more control over indirect branches.

6 Conclusion

While we did not manage to execute the spectre attack on a real example, we did succeed in wrongly executing code thanks to out-of-order execution. We still learned how cache works and how to use a flush and reload attack to exploit caching. By managing the out-of-order execution, we understood the global idea behind the spectre attack. Succeeding in a spectre attack should simply require more tinkering instead of understanding new concepts.

References

[1] *Code Example.* URL: `https://gist.github.com/anonymous/99a72c9c1003f8ae0707b4927ec1bd8a` (visited on 12/05/2022).

[2] *Gitlab.* URL: `https://gitlab.istic.univ-rennes1.fr/` (visited on 12/05/2022).

[3] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 1–19. DOI: `10.1109/SP.2019.00002`.

[4] *Spectre Attack.* URL: `https://spectreattack.com/` (visited on 12/05/2022).

[5] Y. Yarom and K. Falkner. "Flush+reload: A high resolution, low noise, l3 cache side-channel Attack". In: 2014.

Appendix

```c
#include "victim.h"

const struct key_pair keys = {.pub = "aaaabbbbccccdddd", .priv =
↪  "zzzzyyyyxxxxwwww"};

uint8_t get_pub_element(size_t x){
        if (x < key_size){
                return keys.pub[x];
        }
        else{
                return 0;
        }
}
```

Listing 4: Victim code to be exploited