

Spectre Attack Lab

Copyright © 2018 by Wenliang Du.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. If you remix, transform, or build upon the material, this copyright notice must be left intact, or reproduced in a way that is reasonable to the medium in which the work is being re-published.

**This lab has been modified for usage in COMP 435 Security Concepts @ UNC.
Copyright © 2026**

Introduction

Discovered in 2017 and publicly disclosed in January 2018, the Spectre attack exploits critical vulnerabilities existing in many modern processors, including those from Intel, AMD, and ARM [1]. The vulnerabilities allow a program to break inter-process and intra-process isolation, so a malicious program can read the data from the area that is not accessible to it. Such an access is not allowed by the hardware protection mechanism (for inter-process isolation) or software protection mechanism (for intra-process isolation), but a vulnerability exists in the design of CPUs that makes it possible to defeat the protections. Because the flaw exists in the hardware, it is very difficult to fundamentally fix the problem, unless we change the CPUs in our computers. The Spectre vulnerability represents a special genre of vulnerabilities in the design of CPUs. Along with the Meltdown vulnerability, they provide an invaluable lesson for security education.

The learning objective of this lab is for students to gain first-hand experiences on the Spectre attack. The attack itself is quite sophisticated, so we break it down into several small steps, each of which is easy to understand and perform. Once students understand each step, it should not be difficult for them to put everything together to perform the actual attack. This lab covers a number of topics described in the following:

- Spectre attack
- Side channel attack
- CPU caching
- Out-of-order execution and branch prediction inside CPU microarchitecture

Lab Environment. You should use your assigned class VM for this lab. While this is true for **any** lab, it is especially important here as certain vendors may have advanced Spectre mitigations, making them impossible to use for this lab anyways.

Turn-in: Make note of TODOs for each Task. You will complete these in Gradescope as you go along!

Acknowledgment. This lab was developed with the help of Kuber Kohli and Hao Zhang, graduate students in the Department of Electrical Engineering and Computer Science at Syracuse University.

Code Compilation

If you do not have much experience with C code, you must first compile it into an executable program. To do so, you can simply run the command below on the terminal, replacing “myprog.c” with the .c file you want to compile, and “myprog” with the name of the executable you wish to make. For example, the command

```
$ gcc -o myprog myprog.c
```

creates an executable you can run from the command line:

```
$ ./myprog
```

Tasks 1 and 2: Side Channel Attacks via CPU Caches

The Spectre attack uses the CPU cache as a side channel to steal a protected secret. The technique used in this side-channel attack is called FLUSH+RELOAD [2]. We will study this technique first and it will be used as a building block in later tasks.

A CPU cache is a hardware cache used by the CPU of a computer to reduce the average cost (time or energy) to access data from the main memory. Accessing data from CPU cache is much faster than accessing from the main memory. When data are fetched from the main memory, they are usually cached by the CPU, so if the same data are used again, the access time will be much faster. Therefore, when a CPU needs to access some data, it first looks at its caches. If the data is there (this is called cache hit), it will be fetched directly from there. If the data is not there (this is called miss), the CPU will go to the main memory to get the data. The time spent in the latter case is significant longer. Most modern CPUs have CPU caches.

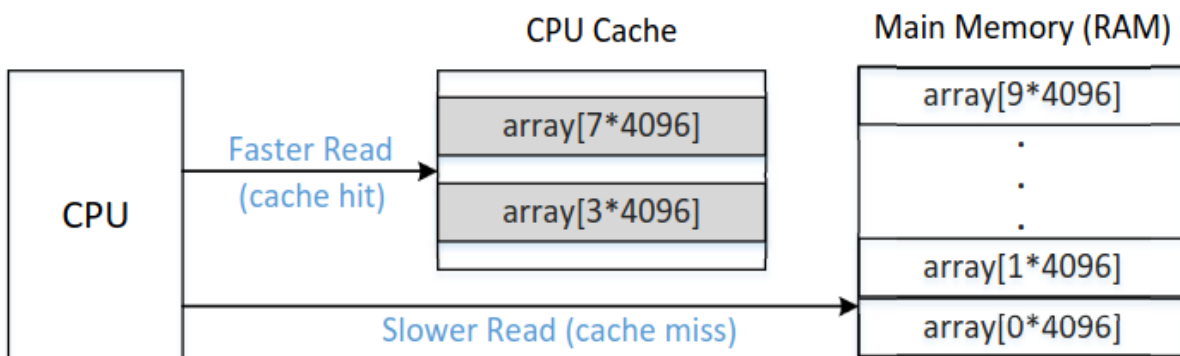


Figure 1: Cache hit and miss

Task 1: Reading from Cache versus from Memory

The cache memory is used to provide data to the high speed processors at a faster speed. The cache memories are very fast compared to the main memory. Let us see the time difference. In the following code (CacheTime.c), we have an array of size 10*4096. We first access two of its elements, array[3*4096] and array[7*4096]. Therefore, the pages containing these two elements will be cached. We then read the elements from array[0*4096] to array[9*4096] and measure the time spent in the memory reading.

Listing 1 illustrates the difference. In the code, Line 24 reads the CPU's timestamp (TSC) counter before the memory read, while Line 28 reads the counter after the memory read. Their difference is the time (in terms of number of CPU cycles) spent in the memory read. It should be noted that caching is done at the cache block level, not at the byte level. A typical cache block size is 64 bytes. We use array[k*4096], so no two elements used in the program fall into the same cache block.

```
8
9  int main(int argc, const char **argv) {
10     int junk=0;
11     register uint64_t time1, time2;
12     volatile uint8_t *addr;
13     int i;
14     // Initialize the array
15     for(i=0; i<10; i++) array[i*4096]=1;
16     // FLUSH the array from the CPU cache
17     for(i=0; i<10; i++) _mm_clflush(&array[i*4096]);
18     // Access some of the array items
19     array[3*4096] = 100;
20     array[7*4096] = 200;
21     for(i=0; i<10; i++) {
22         addr = &array[i*4096];
23         _mm_lfence();
24         time1 = _rdtsc();
25         _mm_lfence();
26         junk = *addr;
27         _mm_lfence();
28         time2 = _rdtsc() - time1;
29         _mm_lfence();
30         printf("Access time for array[%d*4096]: %d CPU cycles\n",i, (int)time2);
31     }
32     return 0;
33 }
34
```

Listing 1.

TODOS

1. Please compile the following code using `gcc -o cachetime CacheTime.c`, and run it.
2. Is the access of `array[3*4096]` and `array[7*4096]` faster than that of the other elements? You should run the program at least 10 times and describe your observations.
3. From the experiment, you need to find a threshold that can be used to distinguish these two types of memory access: accessing data from the cache versus accessing data from the main memory. **Make a note of this threshold**, as it is important for the rest of the tasks in this lab.

A documentation page that may be useful:

<https://learn.microsoft.com/en-us/cpp/intrinsics/rdtscp?view=msvc-170>

Task 2: Using Cache as a Side Channel

The objective of this task is to use the side channel to extract a secret value used by the victim function. We studied this in lecture 10 and 11. Assume there is a victim function that uses a secret value as index to load some values from an array. Also assume that the secret value cannot be accessed from the outside. Our goal is to use side channels to get this secret value. The technique that we will be using is called FLUSH+RELOAD [2]. Figure 2 illustrates the technique, which consists of three steps:

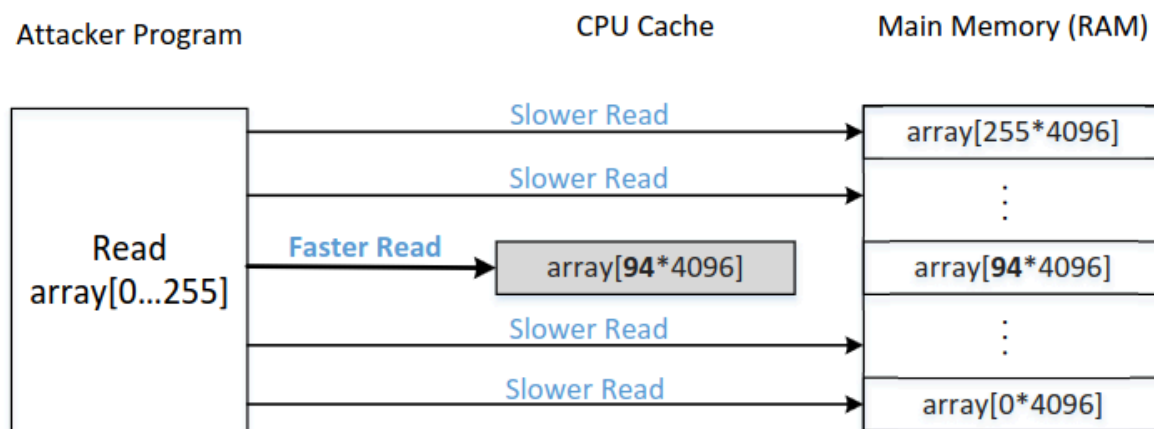


Figure 2: Diagram depicting the Side Channel Attack

1. FLUSH the entire array from the cache memory. FLUSH is a CPU instruction that evicts data from all CPU caches.
2. Invoke the victim function, which accesses one of the array elements based on the value of the secret. (Here, the secret is 94, so the element at `array[94*4096]`.) This action causes the corresponding array element to be cached.
3. RELOAD the entire array, and measure the time it takes to reload each element. If one specific element's loading time is fast, it is very likely that element is already in the cache. This element must be the one accessed by the victim function. Therefore, we can figure out what the secret value is.

The FlushReload.c program in your starter code uses the FLUSH+RELOAD technique to find out a one-byte secret value contained in the variable secret. Since there are 256 possible values for a one-byte secret, we need to map each value to an array element. The naive way is to define an array of 256 elements (i.e., `array[256]`). However, this does not work. Caching is done at a block level, not at a byte level. If `array[k]` is accessed, a block of memory containing this element will be cached. Therefore, the adjacent elements of `array[k]` will also be cached, making it difficult to infer what the secret is. To solve this problem, we create an array of 256×4096 bytes. Each element used in our RELOAD step is `array[k*4096]`. Because 4096 is larger than a typical cache block size (64 bytes), no two different elements `array[i*4096]` and `array[j*4096]` will be in the same cache block.

Since `array` may be allocated starting at the same cache block as the variables in the adjacent memory, `array[0*4096]` may be accidentally cached due to the caching of those variables. Therefore, we should avoid using `array[0*4096]` in the FLUSH+RELOAD method (for other index k , `array[k*4096]` does not have a problem). To make it consistent in the program, we use `array[k*4096 + DELTA]` for all k values, where `DELTA` is defined as a constant 1024.

The `_mm_lfence()` mnemonic is necessary to ensure lines 35-40 are executed in order. Without it, the CPU may leverage out-of-order execution, which would affect the timing results significantly. Adding the `_mm_lfence()` guarantees that the first `_rdtsc()` call happens, then the address is accessed, then the second `_rdtsc()` call happens.

TODOS

1. Please compile the program using gcc and run it (see Section 2 for compilation instructions). It should be noted that the technique is not 100 percent accurate, and you may not be able to observe the expected output all the time.
2. Run the program for at least 20 times, and count how many times you will get the secret correctly.
3. **You should also adjust the threshold `CACHE_HIT_THRESHOLD` to the one derived from Task 1 (80 is used in this code).**

Task 3: Out-of-Order Execution and Branch Prediction

The objective of this task is to understand the out-of-order execution in CPUs. We walk through an experiment to observe such an execution.

Background: Out-Of-Order Execution

The Spectre attack relies on an important feature implemented in most CPUs. To understand this feature, let us see the following code. This code checks whether *x* is less than *size*, if so, the variable *data* will be updated. Assume that the value of *size* is 10, so if *x* equals 15, the code in Line 3 will not be executed.

```
1  data = 0;
2  if (x < size) {
3      data = data + 5;
4  }
```

The above statement about the code example is true when looking from outside of the CPU. However, it is not completely true if we get into the CPU, and look at the execution sequence at the microarchitectural level. If we do that, we will find out that Line 3 may be successfully executed even though the value of *x* is larger than *size*. This is due to an important optimization technique adopted by modern CPUs. It is called **out-of-order execution**.

Out-of-order execution is an optimization technique that allows CPU to maximize the utilization of all its execution units. Instead of processing instructions strictly in a sequential order, a CPU executes them in parallel as soon as all required resources are available. While the execution unit of the current operation is occupied, other execution units can run ahead.

In the code example above, at the microarchitectural level, Line 2 involves two operations: load the value of *size* from the memory, and compare the value with *x*. If *size* is not in the CPU caches, it may take hundreds of CPU clock cycles before that value is read. Instead of sitting idle, modern CPUs try to predict the outcome of the comparison, and speculatively execute the branches based on the estimation. Since such execution starts before the comparison even finishes, the execution is called out-of-order execution. Before doing the out-of-order execution, the CPU stores its current state and value of registers. When the value of *size* finally arrives, the CPU will check the actual outcome. If the prediction is true, the speculatively performed execution is committed and there is a significant performance gain. If the prediction is wrong, the CPU will revert back to its saved state, so all the results produced by the out-of-order execution will be discarded like it has never happened. That is why from outside we see that Line 3 was never executed. Figure 3 illustrates the out-of-order execution caused by Line 2 of the sample code.

Intel and several CPU makers made a severe mistake in the design of the out-of-order execution. They wipe out the effects of the out-of-order execution on registers and memory if such an execution is not supposed to happen, so the execution does not lead to any visible effect. However, they forgot one thing, **the effect on CPU caches**. During the out-of-order execution, the referenced memory is fetched into a register and is also stored in the cache. If the results of the out-of-order execution have to be discarded, the caching caused by the execution should also be discarded. Unfortunately, this is not the case in most CPUs. Therefore, it creates an observable

effect. Using the side-channel technique described in Tasks 1 and 2, we can observe such an effect. The Spectre attack cleverly uses this observable effect to find out protected secret values.

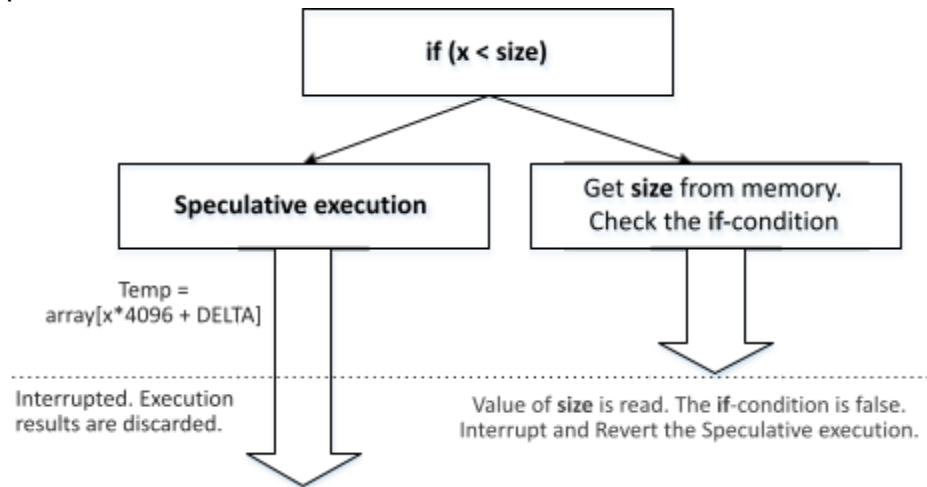


Figure 3: Speculative execution (out-of-order execution)

The Experiment

In this task, we use an experiment to observe the effect caused by an out-of-order execution. The code used in this is in `SpectreExperiment.c`. Some of the functions used in the code are the same as that in the previous tasks, so they will not be repeated.

For CPUs to perform a speculative execution, they should be able to predict the outcome of the if condition. CPUs keep a record of the branches taken in the past, and then use these past results to predict what branch should be taken in a speculative execution. Therefore, if we would like a particular branch to be taken in a speculative execution, we should train the CPU, so our selected branch can become the prediction result. The training is done in the for loop starting from Line 61. Inside the loop, we invoke `victim()` with a small argument (from 0 to 9). These values are less than the value `size`, so the true-branch of the if-condition in Line 49 is always taken. This is the training phase, which essentially trains the CPU to expect the if-condition to come out to be true.

Once the CPU is trained, we pass a larger value (97) to the `victim()` function (Line 69). This value is larger than `size`, so the false-branch of the if-condition inside `victim()` will be taken in the actual execution, not the true-branch. However, we have flushed the variable `size` from the memory, so getting its value from the memory may take a while. This is when the CPU will make a prediction, and start speculative execution.

TODOS:

1. Please compile the SpectreExperiment.c program shown in Listing 3 (see Section 2 for the compilation instruction); run the program.
2. There may be some noise in the side channel due to extra things cached by the CPU, we will reduce the noise later, but for now you can execute the task multiple times to observe the effects. Please observe whether Line 69 is executed or not when 97 is fed into victim().
3. Please also do the following: Replace Line 62 with victim(i + 20); run the code again and think about what changed, and why.

Task 4: The Spectre Attack

As we have seen from the previous task, we can get CPUs to execute a true-branch of an if statement, even though the condition is false. If such an out-of-order execution does not cause any visible effect, it is not a problem. However, most CPUs with this feature do not clean the cache, so some traces of the out-of-order execution are left behind. The Spectre attack uses these traces to steal protected secrets.

These secrets can be data in another process or data in the same process. If the secret data is in another process, the process isolation at the hardware level prevents a process from stealing data from another process. If the data is in the same process, the protection is usually done via software, such as sandbox mechanisms. The Spectre attack can be launched against both types of secrets. However, stealing data from another process is much harder than stealing data from the same process. For the sake of simplicity, this lab only focuses on stealing data from the same process.

When web pages from different servers are opened inside a browser, they are often opened in the same process. The sandbox implemented inside the browser will provide an isolated environment for these pages, so one page will not be able to access another page's data. Most software protections rely on condition checks to decide whether an access should be granted or not. With the Spectre attack, we can get CPUs to execute (out-of-order) a protected code branch even if the condition checks fails, essentially defeating the access check.

Setup for the Experiment

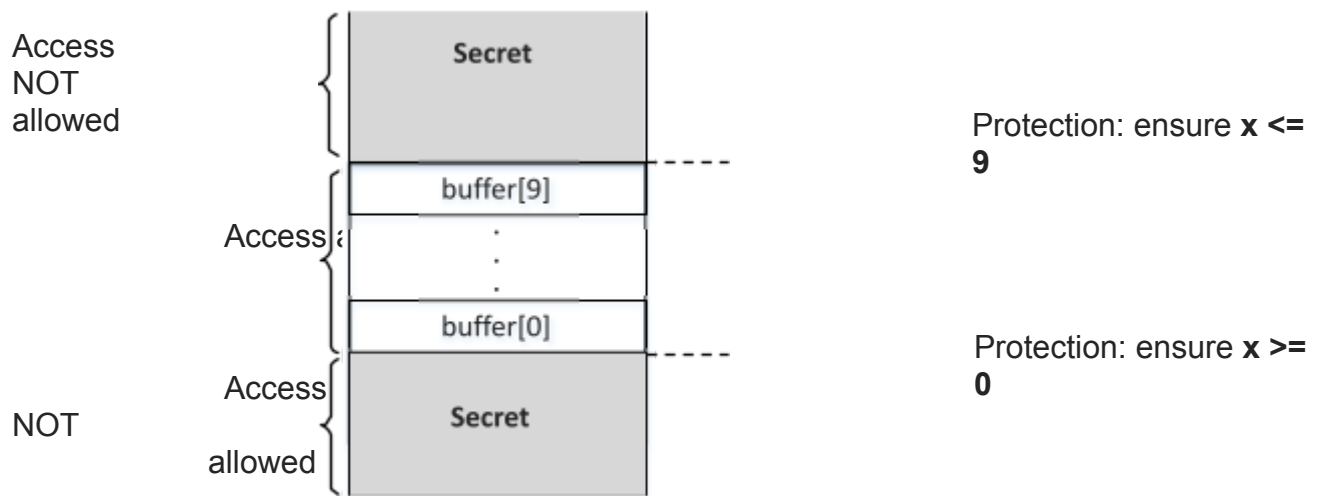


Figure 4: Experiment setup: the buffer and the protected secret

Figure 4 illustrates the setup for the experiment. In this setup, there are two types of regions: restricted region and non-restricted region. The restriction is achieved via an if-condition implemented in a sandbox function described below. The sandbox function returns the value of `buffer[x]` for an `x` value provided by users, only if `x` is between the buffer's lower and upper bounds. (A buffer is a region of memory used to temporarily hold data while it is stored, transferred, read, or processed.) Therefore, this sandbox function will never return anything in the restricted area to users.

```
unsigned int bound_lower = 0;
unsigned int bound_upper = 9;
uint8_t buffer[10] = {0,1,2,3,4,5,6,7,8,9};

// Sandbox Function
uint8_t restrictedAccess(size_t x)
{
    if (x <= bound_upper && x >= bound_lower) {
        return buffer[x];
    } else {
        return 0;
    }
}
```

There is a secret value in the restricted area (either above the buffer or below it), and the secret's address is known to the attacker, but the attacker cannot directly access the memory holding the secret value. The only way to access the secret is through the above sandbox function. From the previous section, we have

learned that although the true-branch will never be executed if x is larger than the buffer size, at microarchitectural level, it can be executed and some traces can be left behind when the execution is reverted.

The Program Used in the Experiment

The code for the basic Spectre attack is in `SpectreAttack.c`. In this code, there is a secret defined in Line 10. Assume that we cannot directly access the secret, `bound_lower`, or `bound_upper` variables (we do assume that we can flush the two bound variables from the cache). Our goal is to print out the secret using the Spectre attack. The code below only steals the first byte of the secret. You can extend it to print out more bytes.

Most of the code is the same as that in `SpectreExperiment.c`, so we will not repeat their explanation here. The most important part is in Lines 67, 68, and 73. Line 73 calculates the offset of the secret from the beginning of the buffer (we assume that the address of the secret is known to the attacker; in real attacks, there are many ways for attackers to figure out the address, including guessing). The offset is definitely beyond the scope of the buffer, so it is larger than the upper bound of the buffer or smaller than the lower bound (i.e., a negative number). The offset is fed into the `restrictedAccess()` function. Since we have trained the CPU to take the true-branch inside `restrictedAccess()`, the CPU will return `buffer[index beyond]`, which contains the value of the secret, in the out-of-order execution. The secret value then causes its corresponding element in `array[]` to be loaded into cache. All these steps will eventually be reverted, so from the outside, only zero is returned from `restrictedAccess()`, not the value of the secret. However, the cache is not cleaned, and `array[s*4096 + DELTA]` is still kept in the cache. Now, we just need to use the side-channel technique to figure out which element of the `array[]` is in the cache.

TODOS:

1. Please compile and execute `SpectreAttack.c`.
2. Note whether you are able to steal the secret value. If there is a lot of noise in the side channel, you may not get consistent results every time. To overcome this, you should execute the program multiple times and see whether you can get the secret value. Record any observations

Task 5: Improve the Attack Accuracy

In the previous tasks, it may be observed that the results do have some noise and the results are not always accurate. This is because CPU sometimes load extra values in cache expecting that it might be used at some later point, or the threshold is not very accurate. This noise in cache can affect the results of our attack. We need to perform the attack multiple times; instead of doing it manually, we can use the following code to perform the task automatically.

We basically use a statistical technique. The idea is to create a score array of size 256, one element for each possible secret value. We then run our attack multiple times. Each time, if our attack program says that k is the secret (this result may be false), we add 1 to `scores[k]`. After running the attack for many times, we use the value k with the highest score as our final estimation of the secret. This will produce a much more reliable estimation than the one based on a single run. The revised code is found in `SpectreAttackImproved`.

NOTE: To compile this program, you'll also need to include `Secret.c`, which we've already provided. `Secret.c` contains the secret string, which you can change for debugging purposes. To compile multiple files:

```
$ gcc -o spectre SpectreAttackImproved.c Secret.c
```

TODOS:

1. Please compile and run the code.
2. You may observe that when running the code above, the one with the highest score is very likely to be `scores[0]`. Please figure out why, and fix the code above, so the actual secret value (which is not zero) will be printed out.
3. Line 88 seems useless, but from our experience on Ubuntu 20.04, without this line, the attack will not work. On the Ubuntu 16.04 VM, there is no need for this line. Can you explain why we do or do not need this line?
4. Line 90 causes the program to sleep for 10 microseconds. How long the program sleeps does affect the success rate of the attack. You should try several other values, and pick the one that yields the best results. Record your result.

Task 6: Steal the Entire Secret String

In the previous task, we just read the first character of the secret string. In this task, we need to print out the entire string using the Spectre attack. Please write your own code or extend the code in Task 5; include your execution results in the report. Note: you can assume that secret is at most twenty characters long

Submission

You should clone the Lab1 repo in Classroom50 and make your changes there. When you have finished the lab, you will turn in your finished code for task 6 to the lab 1 "Code Submission" Gradescope assignment. Submit your written answers for tasks 1-5 to the Lab 1 "Written Answers" assignment

References

- [1] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, January 2018.
- [2] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium, SEC'14*, pages 719–732, Berkeley, CA, USA, 2014. USENIX Association.