**Spectre Attack Lab**

**1. Goal and Deliverables**

The learning objective of this lab is for students to gain first-hand experiences on the Spectre attack.

Please submit the lab report to D2L.

**2. 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.

**3. Code Compilation**

For most of our tasks, you need to add –march=native flag when compiling the code with gcc. The march flag tells the compiler to enable all instruction subsets supported by the local machine. For example, we compile myprog.c using the following command:

$gcc –march=native –o myprog myprog.c

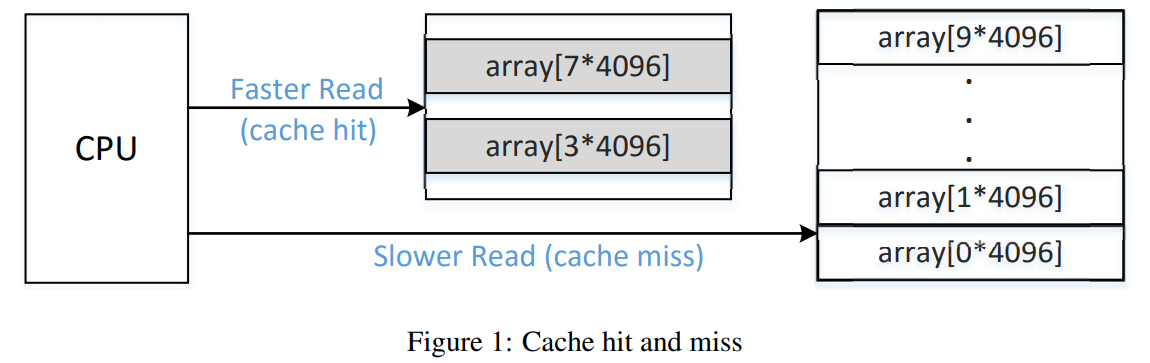
**4. Tasks 1 and 2: Side channel attacks via CPU Caches**

Both the Meltdown and Spectre attacks use 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. The code developed in these two tasks 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.

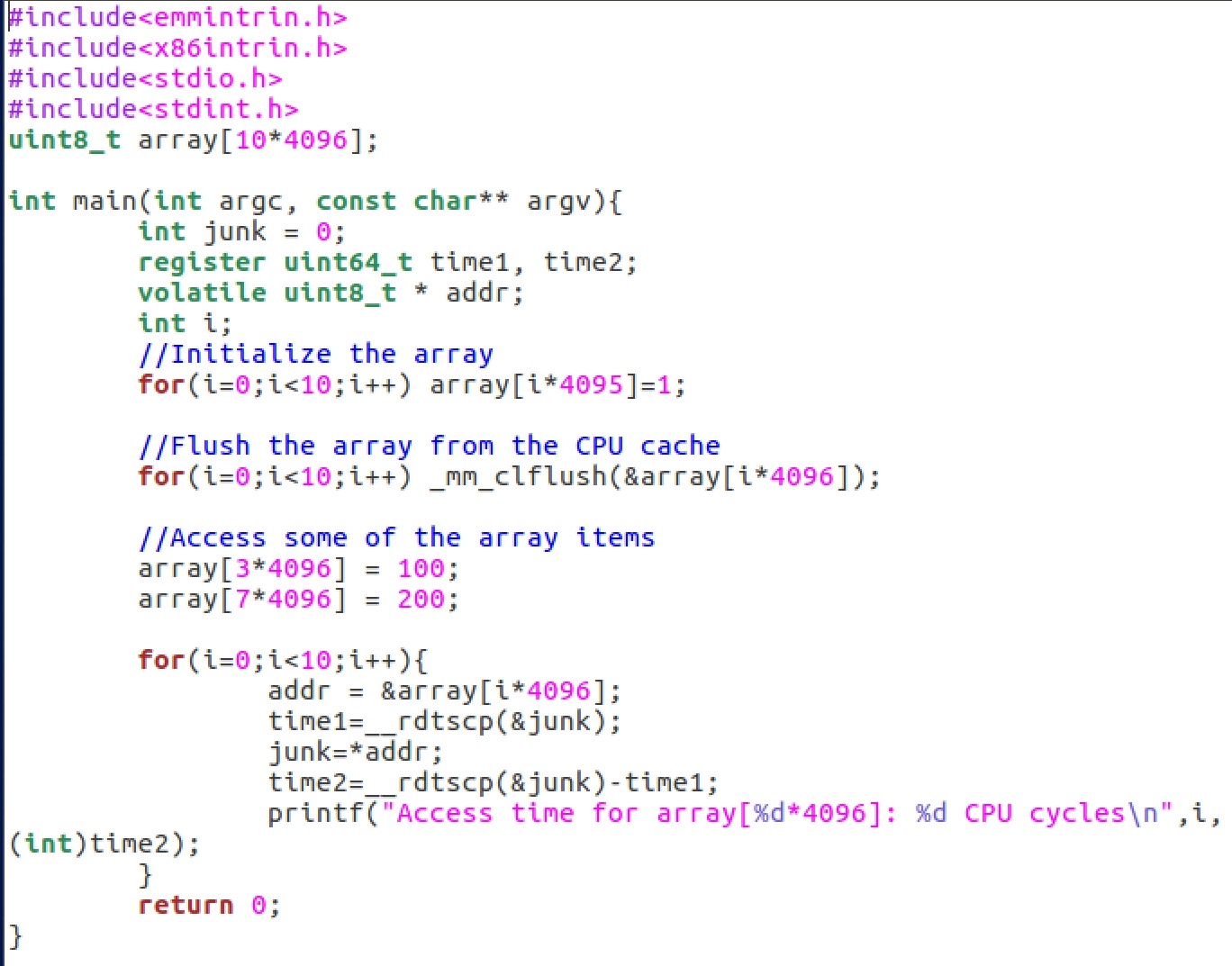
**4.1 Task 1: Reading from Cache versus from Memory**

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. Figure 1 illustrates the difference.



Step 1: Type the command *nano CacheTime.c* and enter the C source code blow

Listing 1 CacheTime.c



In the code, statement “time1=\_\_rdtscp(&junk);” reads the CPU’s timestamp (TSC) counter before the memory read, while statement “time2=\_\_rdtscp(&junk)-time1” 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.

Step 2: Compile the code by using command *gcc –march=native –o CacheTime CacheTime.c*

Step 3: Run the code by using command: *./CacheTime*

Is the access of array [3\*4096] and array [7\*4096] faster than that of other elements? You should run the program at least 10 times. 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.

Please attach a screenshot for the output of CacheTime.c and provide the threshold to distinguish accessing data from the cache versus accessing data from the main memory.

Text

Description automatically generated

The best threshold after many runs that I can come up with is approximately 460 cycles. Very rarely do the 3 and 7 indexes of the array get accessed slower than 460 cycles, and very rarely do any of the others get accessed faster than 460 cycles (the best being 450).

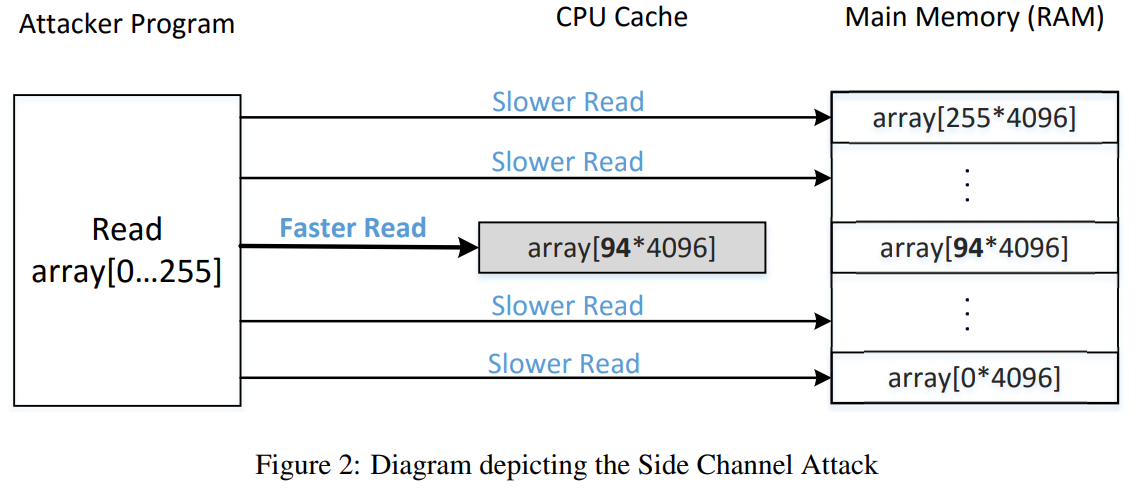
**4.2 Task 2: Using Cache as a Side Channel**

The objective of this task is to use the side channel to extract a secret value used by the victim function. 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:

1. FLUSH the entire array from the cache memory to make sure the array is not cached

2. Invoke the victim function, which accesses one of the array elements based on the value of the secret. 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.

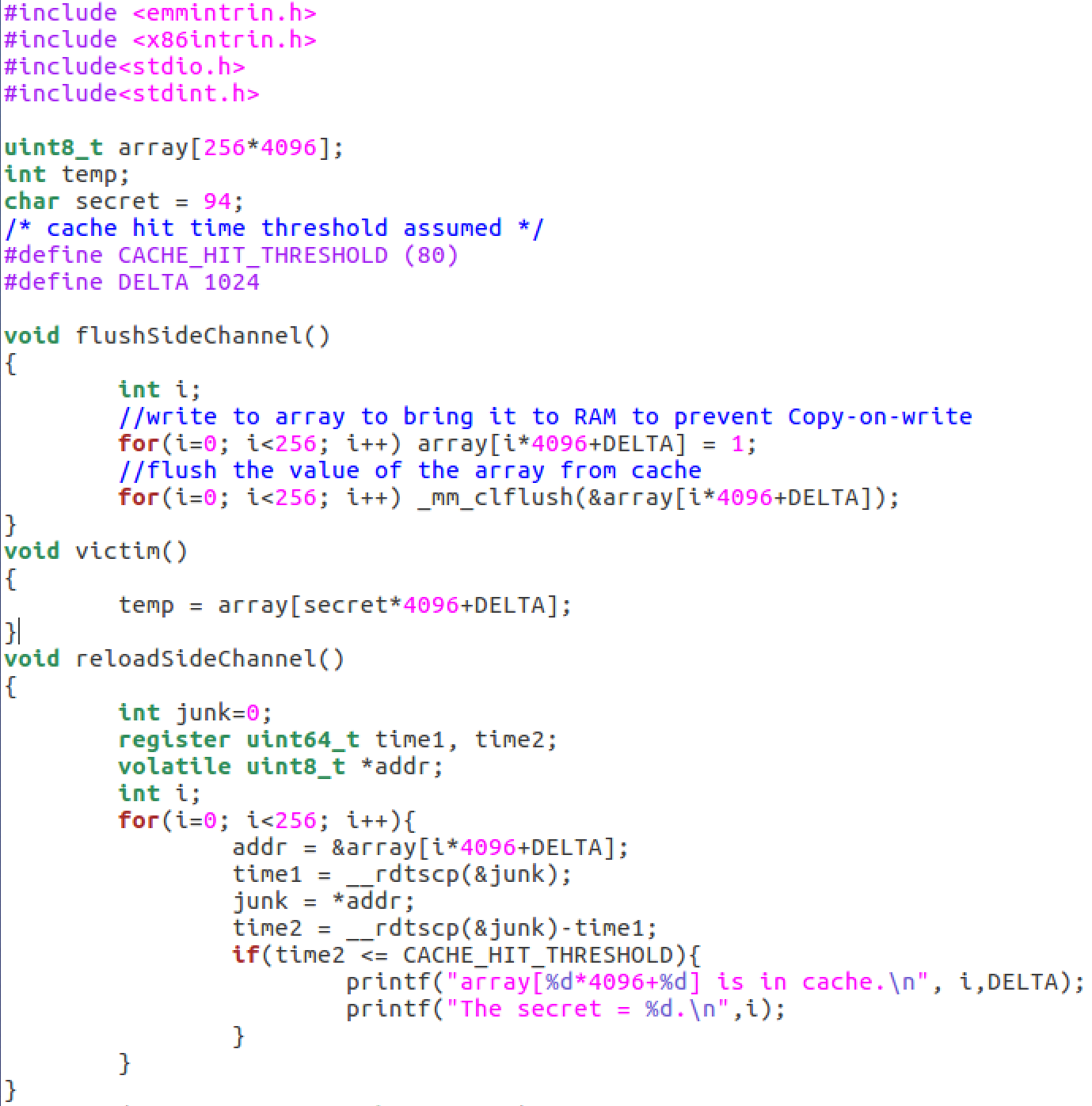


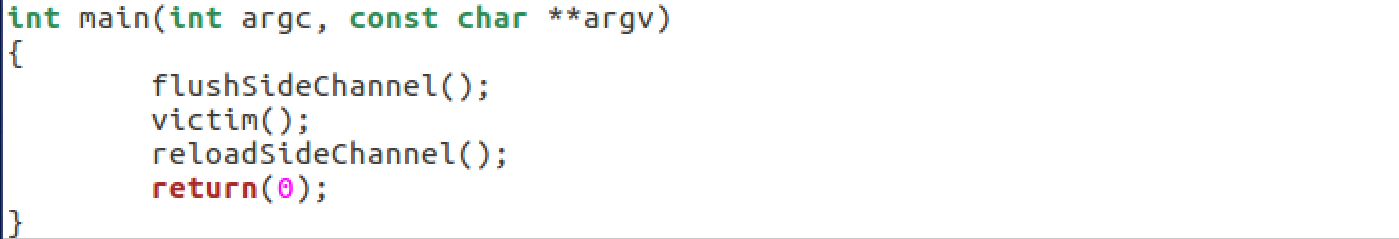
The following program uses the FLUSH+RELOAD technique to find out a one-byte secret value contained in the variable secret. Since there are 256 possible values for a one-byte secret, we need to map each value to an array element. The naïve way is to define an array of 256 elements (i.e., array[256]). 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 it. 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[0\*4096] may fall into the same cache block as the variables in the adjacent memory, it 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.

Step 4: Please enter, compile, the run the code in Listing 2 (see step 1-3 for compilation instruction). Note that, the CACHE\_HIT\_THRESHOLD value (80 in this program) is determined by the previous run of the program. If the fastest speed you got for block 3 and 7 is around 170, and the rest of them is above 400. You could use 180, 190, 200, etc as a threshold value. In other words, if the number is smaller than the threshold, it could be a “secret”. Please feel free to change the threshold value and see the impact on the result.

Listing 2: FlushReload.c





Step 5: Please run the program for at least 20 times, and count how many times you will get the secret correctly. 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. You can also adjust the threshold CACHE\_HIT\_THRESHOLD to the one derived from task 1 (80 is used in this code).

Please show a screenshot of a successful case where you get the secret. Also, please record the number of times you get the secret correctly out of 20.

Graphical user interface, application

Description automatically generated

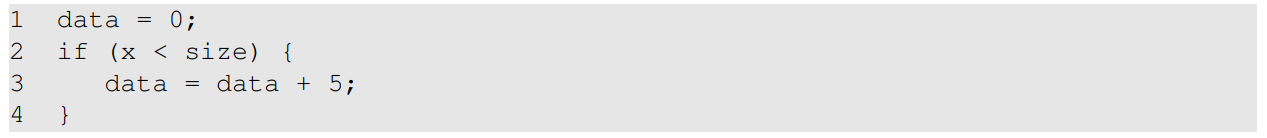
My threshold of 460 was actually a bit high, so I got the secret 94 in every output, but I also got other numbers such as 2, 4, 0, 59, and other random numbers that happened to be accessed just as fast. Because 94 was consistently appearing, it can be assumed that it is the actual cashed value. I changed my threshold to 430 for the next section of the lab though.

**5. 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 will use an experiment to help students observe such kind of execution.

**5.1 Out-Of-Order Execution**

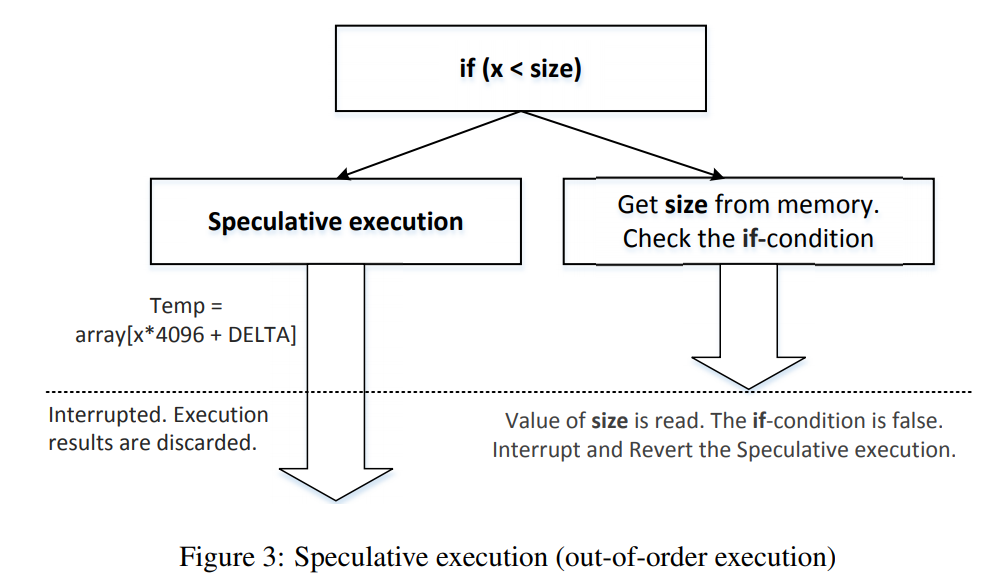
The Spectre attack relies on an important feature implemented in most CPUs. To understand this feature, let us see the following code. This code checks whether x is less than size, if so, the variable data will be updated. Assume that the value of size is 10, so if x equals 15, the code in Line 3 will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



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 micro architectural 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. This 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.

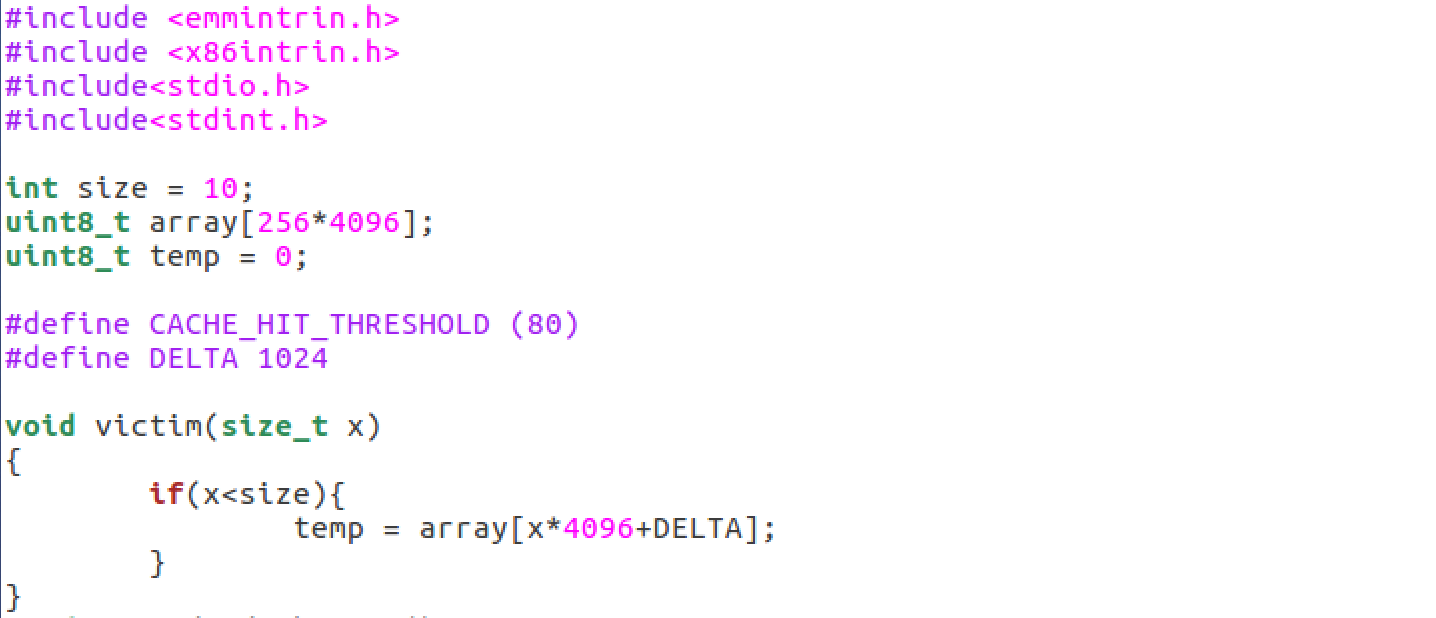


Intel and several CPU makers made a severe mistake in the design of the out-of-order execution. They wipe out the effects of the out-of-order execution on registers and memory if such an execution is not supposed to happen, so the execution does not lead to any visible effect. However, they forgot one 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 Task 1 and 2, we can observe such an effect. The Spectre attack cleverly uses this observable effect to find out protected secret values.

**5.2 The Experiment**

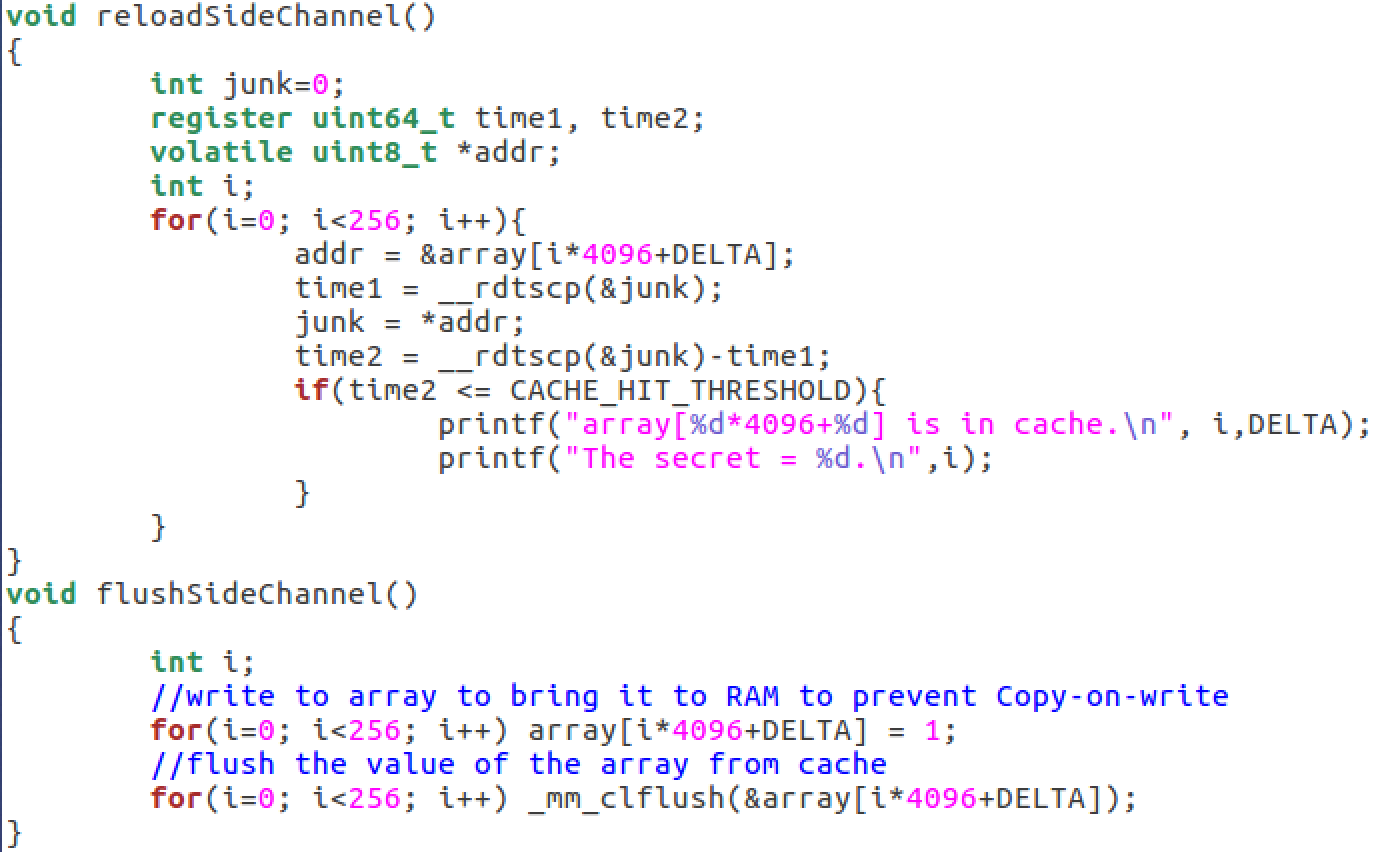
In this task, we use an experiment to observe the effect caused by an out-of-order execution. The code used in this experiment is shown below. Some of the functions used in the code is the same as that in the previous tasks, so they will not be repeated.

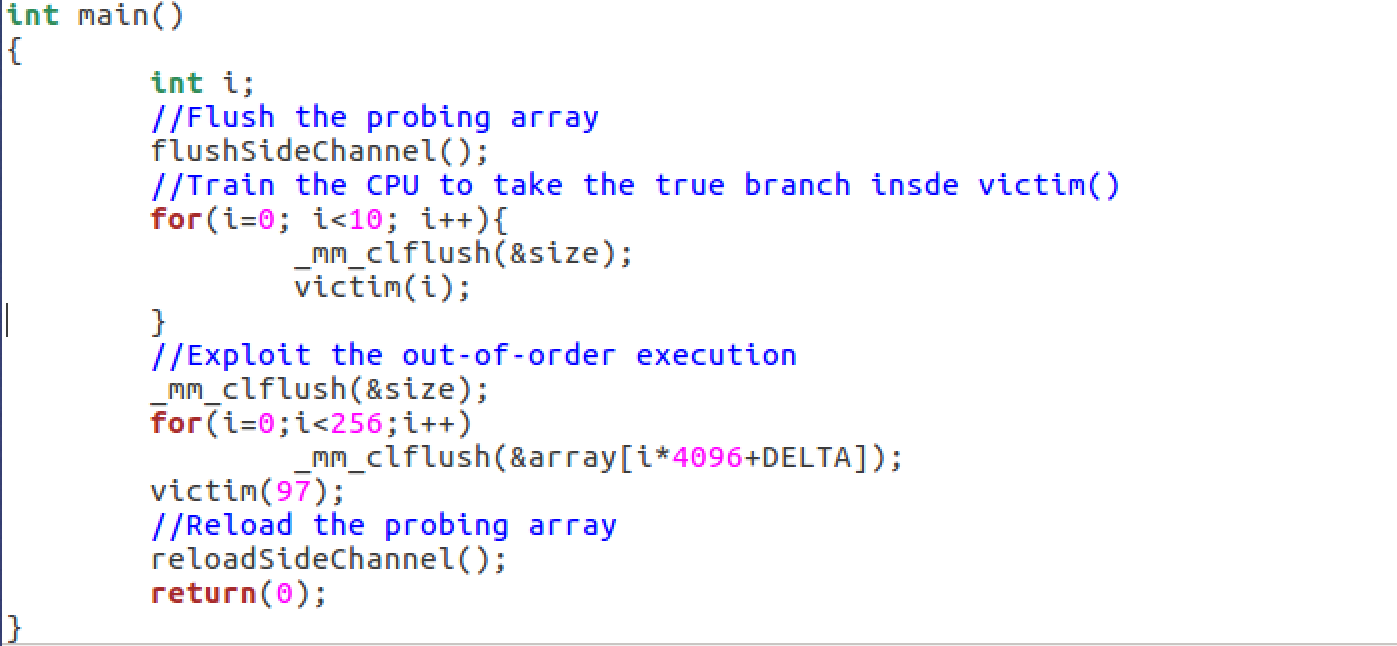
Listing 3: SpectreExperiment.c



②

①











⑤

④

③

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 ③. 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 ① 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 ⑤). 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.

Step 6: Please compile the SpectreExperiment.c program shown in Listing 3 and run the program. 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 and show your screenshot whether Line ② is executed or not when 97 is fed into victim().

A picture containing text

Description automatically generated

To observe whether or not Line 2 is executed or not, I created an “else” for the if-statement to prove that clearly, when 97 (the only number >= 10 that we use) is used as input, the if-statement returns false and executes the “else.”

Step 7: Comment out the lines marked with and execute again.

Please provide a screenshot and explain your observation. After you are done with this experiment, uncomment them, so the subsequent tasks are not affected.



After recompiling with the commented-out lines, we can no longer find 97 in the cache. This is because without making sure that the variable size is flushed from the cache, the CPU might decide that it’s easier to just compute the if-statement first and then decide the branch to take instead of using its past experiences with the if-statement to guess that the statement will resolve to true.

Step 8: Replace Line ④ with victim(i+20); run the code again.

Please provide a screenshot and explain your observation.

Text

Description automatically generated

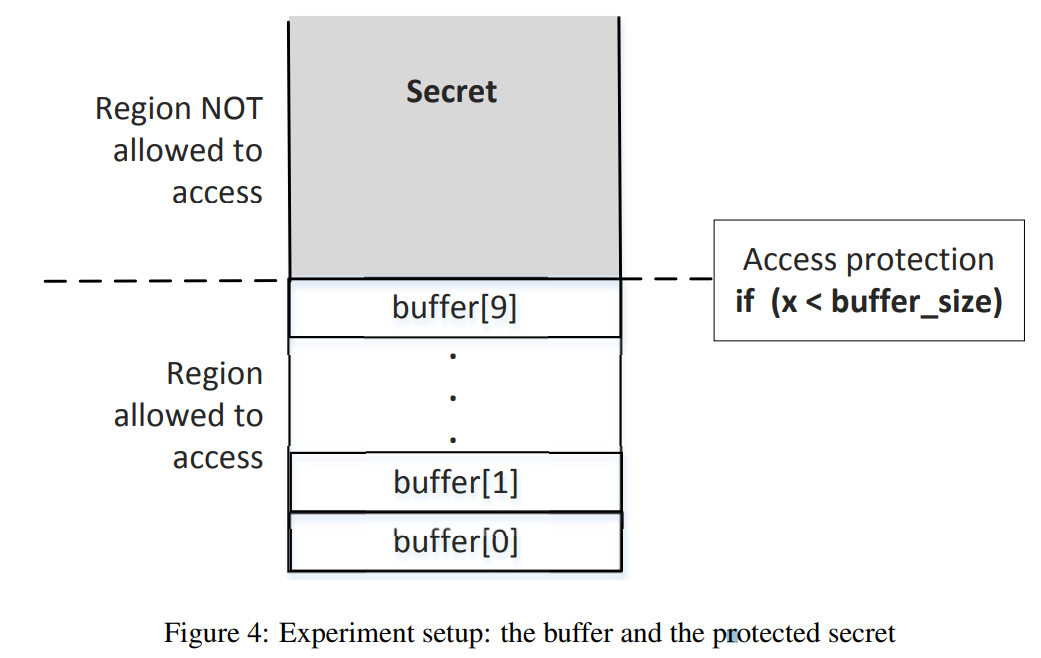
In this scenario, you can see that instead of training the CPU 10 times to execute the true statement, it executes the false statement (“Tyler is so cool”) instead since our input values are all greater than 10. Because of this, the CPU is not trained to predict the “true” outcome, and so the input of 97 didn’t run distinctively faster than the others, preventing us to see it pass under the threshold and identify it as having a value in the cache. In fact, other random values appear to be stored in the cache based on their CPU cycles.

**6. 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 is left behind. The Spectre attack uses these traces to steal protected secrets.

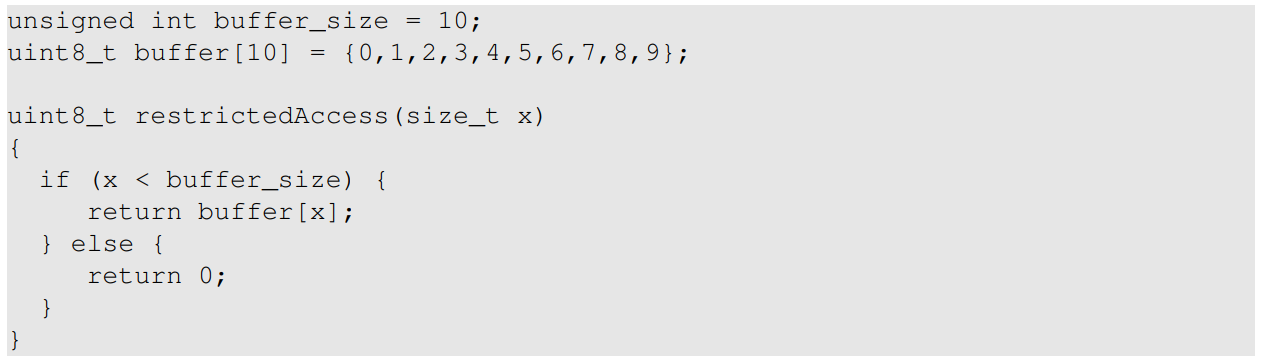
These secrets can be data in another process or data in the same process. If the secret data is in another process, the process isolation at the hardware level prevents a process from stealing data from another process. If the data is in the same process, the protection is usually done via software, such as sandbox mechanisms. The Spectre attack can be launched against both types of secret. 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) protected code branch even if the condition checks fails, essentially defeating the access check.



**6.1 The Setup for the Experiment**

Figure 4 illustrates the setup for the experiment. In this setup, there are two regions: a restricted region and a 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 less than the size of the buffer; otherwise, nothing is returned. Therefore, this sandbox function will never return anything in the restricted area to users.



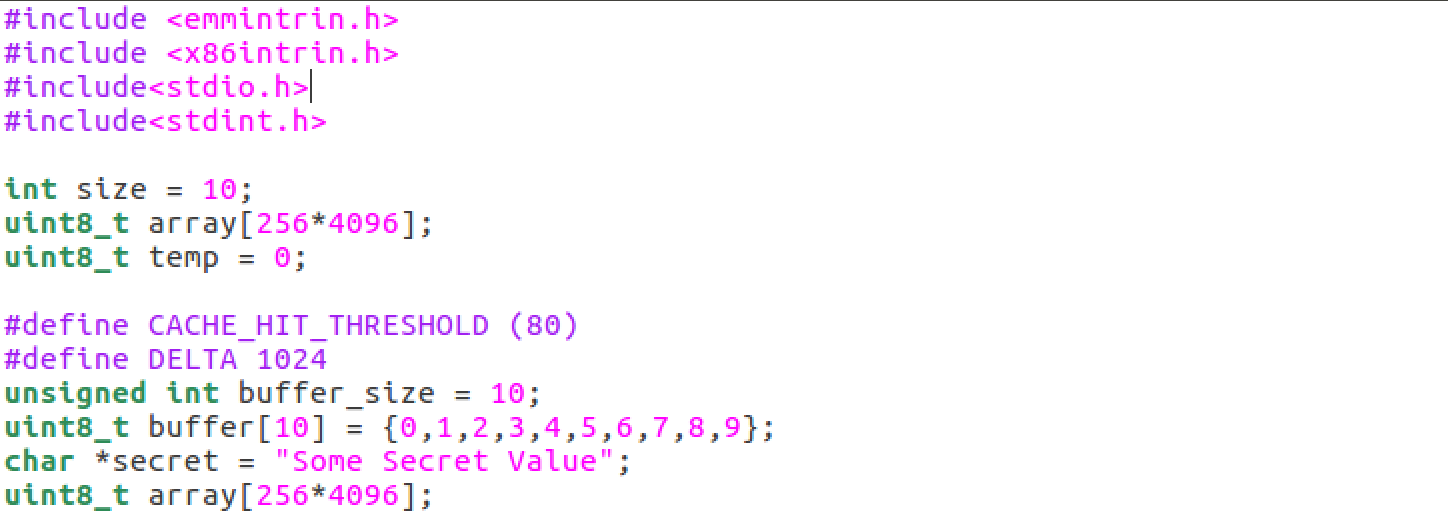
There is a secret value in the restricted area, the address of which is known to the attacker. However, the attacker cannot directly access the memory holding the secret value; the only way to access the secret is through the above sandbox function. From the previous task, we have learned that although the true-branch will never be executed if x is larger than the buffer size, at micro architectural level, it can be executed and some traces can be left behind when the execution is reverted.

**6.2 The Program Used in the Experiment**

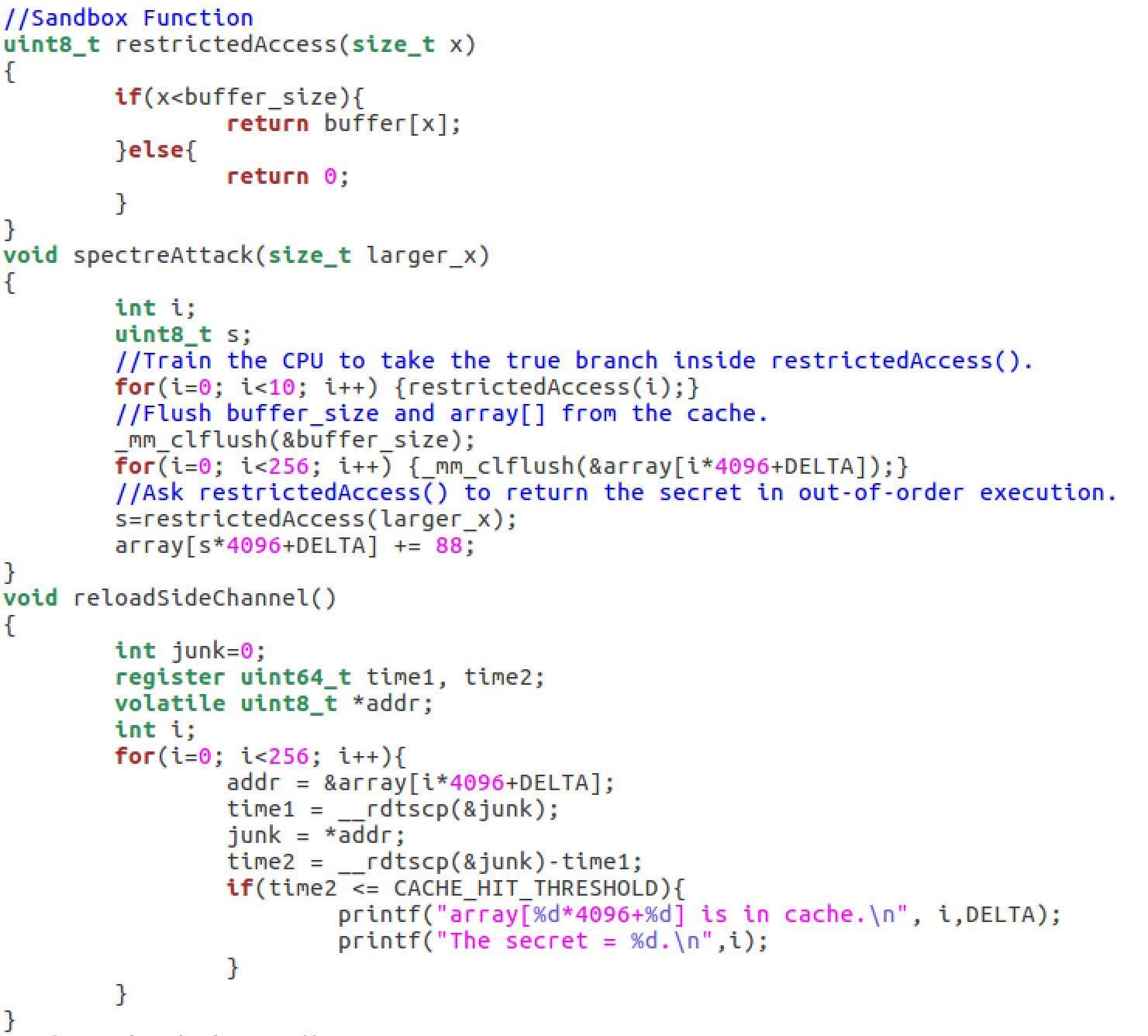
The code for the basic Spectre attack is shown below. In this code, there is a secret defined in the Line ①. Assume that we cannot directly access the secret variable or the buffer\_size variable (we do assume that we can flush buffer\_size from the cache). Our goal is to print out the secret using the Spectre attack.

The code below only steals the first byte of the secret. You can extend it to print out more bytes.

Listing 4: SpectreAttack.c

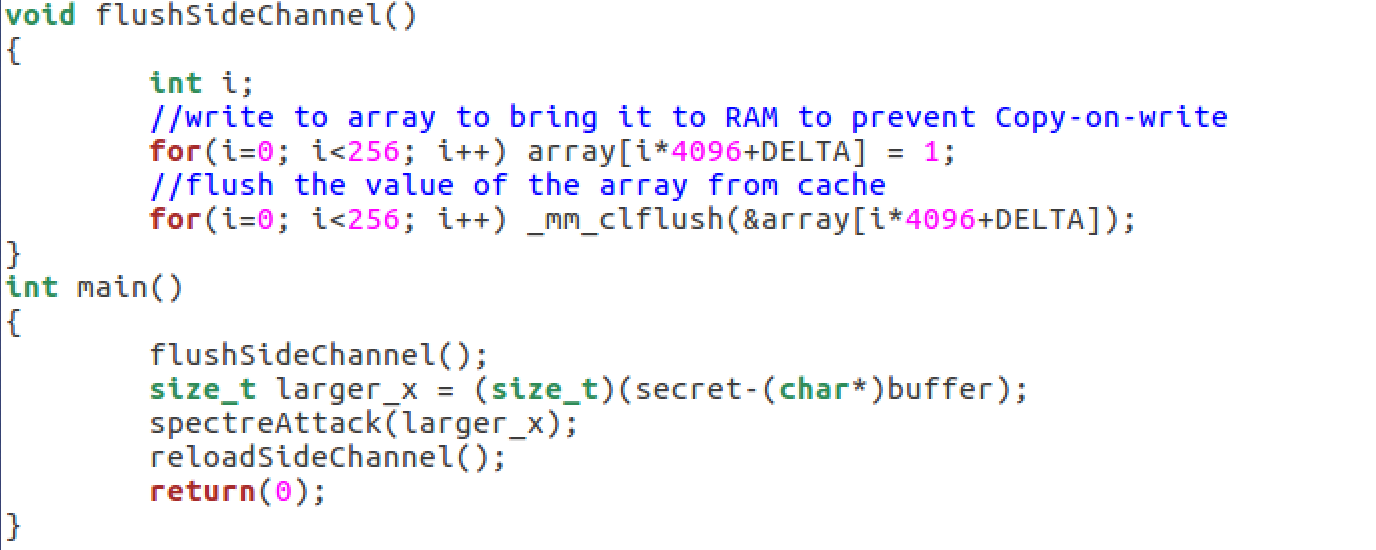


①



③

②



④

Most of the code is the same as that in Listing 3, so we will not repeat their explanation here. The most important part is in Lines ②, ③, and ④. Line ④ 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, which is definitely larger than 10, is fed into the restrictedAccess() function. Because we have trained the CPU to take the true-branch inside restrictedAccess(), the CPU will return buffer[larger\_x], 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 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.

Step 9: Please compile and execute SpectreAttack.c. 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.

Please provide a screenshot and describe your observation and note whether you are able to steal the secret value.

Text

Description automatically generated

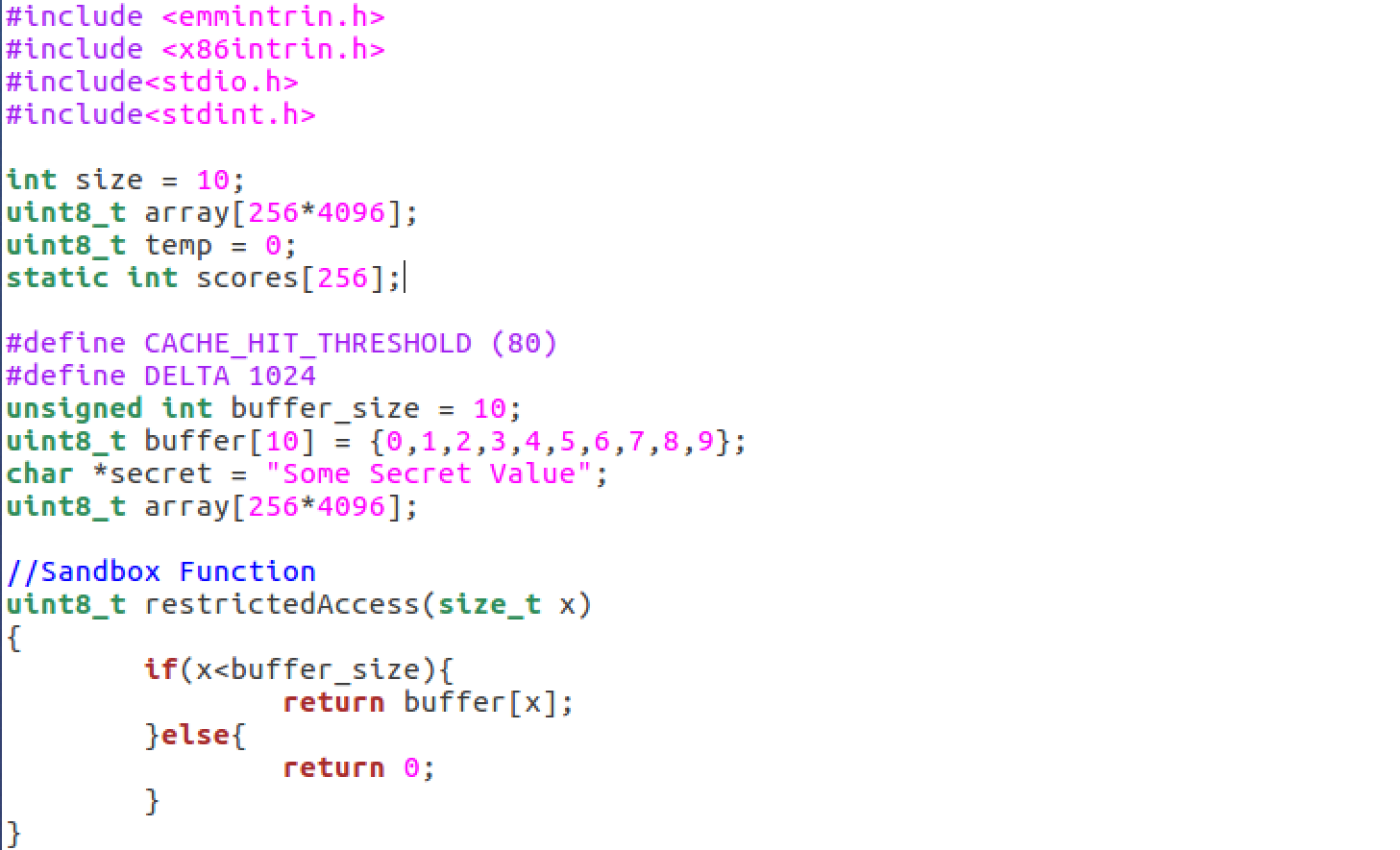
Supposedly, the secret value I’m getting is 0 (it’s the most consistent out of anything else). However, I’m not entirely sure that this makes sense. I’m wondering if something is wrong because 0 is the returned value of the failed if-statement. I think there is a need to improve the accuracy.

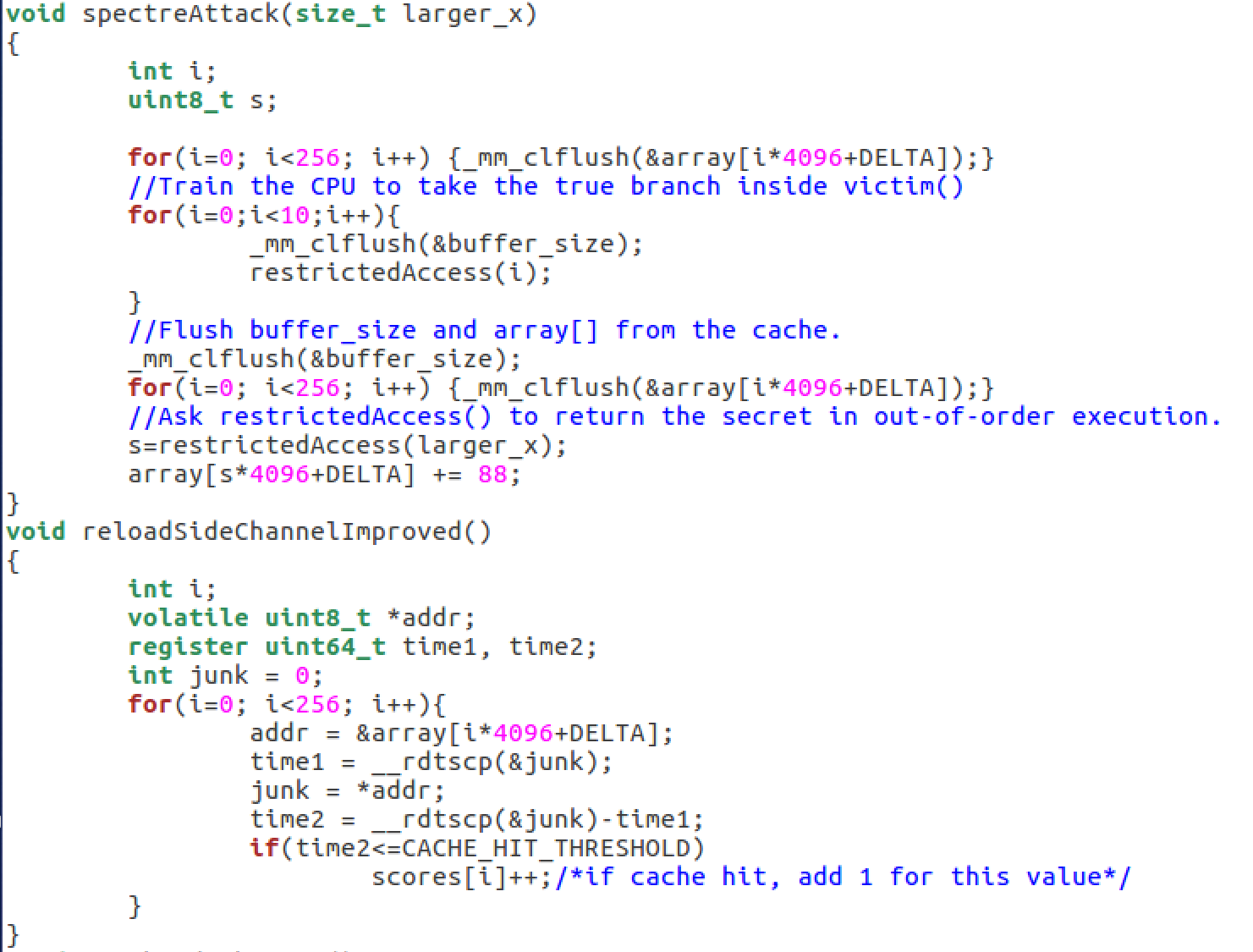
**7. Task 5: Improve the Attack Accuracy**

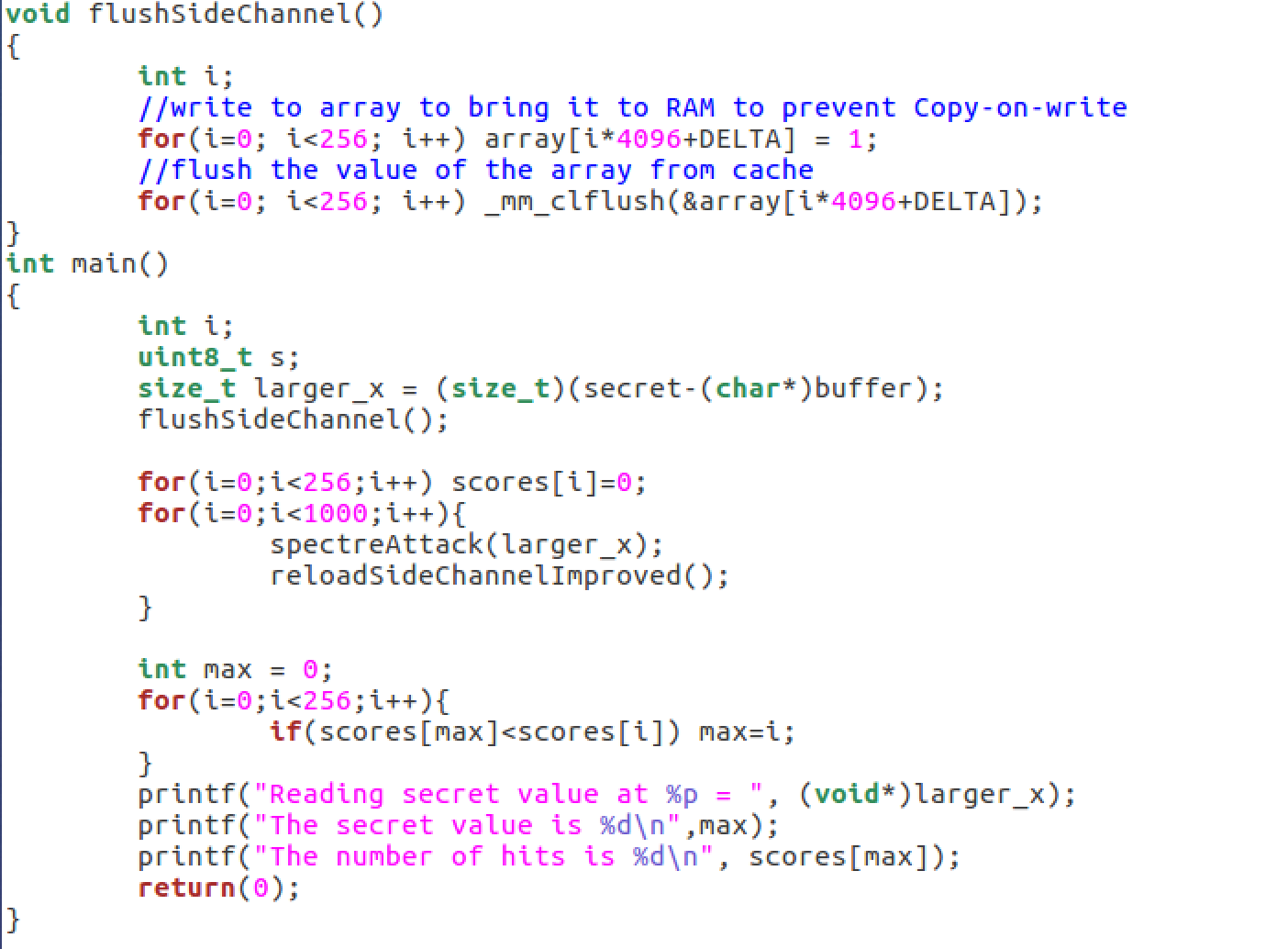
In the previous tasks, it may be observed that the results do have some noise and the results are not always accurate. This is because CPU sometimes load extra values in cache expecting that it might be used at some later point, or the threshold is not very accurate. This noise in cache can affect the 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 for 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 mush reliable estimation than the one based on a single run. The revised code is shown in the following.

Listing 5: SpectreAttackImproved.c







Step 10: Please compile and run the code.

You may observe that when running the code above, the one with the highest score is always scores[0]. Please figure out the reason, and fix the code above, so the actual secret value (which is not zero) will be printed out. Please provide a screen shot on your **results** and the **edited code**.

The reason the score is always zero is because the function actually returns 0, so it’s always stored in the cache, resulting in very quick accesses every time. To avoid this, since we know none of our desired characters line up with the ASCII character for 0, we can cut it out of the comparison by initializing max with 1 and running the comparison for loop from 1. (My final code is shown after the bonus – you can see the alteration of the max variable there)

Note – I commented out the other print statements since I found the value I was looking for and no longer needed to reference them.

Text

Description automatically generated

**8. Task 6: Steal the Entire Secret String (Bonus 5 points)**

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. please provide a screen shot on your **results** and the **edited code**.

To successfully steal the entire string, all you (theoretically) have to do is loop the “get the secret character” section of the main function 17 times, adding an addition 1 each time to the larger\_x value to shift where you’re looking in memory. However, despite several attempts to craft a for loop that works, my for loop does not successfully find the next characters. Somehow, the returned values from the attack end up being incorrect. I can prove that the *idea* itself of increasing larger\_x by increments of 1 works with the following screenshots (I went in and changed larger\_x in the spectreAttack call to be +1, +2, and +3 for these screenshots, getting the letters “ome” of “Some”), but I cannot quite get it to work as a loop (shown below). I also have my code below for you to see if you can find what’s incorrect. My assumption would be something is getting added to the cache without me knowing, but the cache clears should take care of any such issue, so I’m confused as to why this isn’t working.

Text

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

**Reference**

[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, 13 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.

**Document may help**

1. How to compile and run C code in Ubuntu: <http://akira.ruc.dk/~keld/teaching/CAN_e14/Readings/How%20to%20Compile%20and%20Run%20a%20C%20Program%20on%20Ubuntu%20Linux.pdf>