Chair of IT Security
Brandenburg University of Technology

Asya Mitseva, M.Sc.
Fabian Mildner, B.Sc.
Prof. Dr.-Ing. Andriy Panchenko

Summer Term 2025

Part VI: System Security **Deadline: 17th July 2025**

Cyber Security Lab (Ethical Hacking)
The Spectre Vulnerability

Introduction

In the year 2017, a pair of serious system vulnerabilities were discovered and publically disclosed: *Meltdown* [2] and *Spectre* [1]. These two vulnerabilities quickly made headlines all around the world, since they could not only be exploited to provide dangerous capabilities to an attacker, but they were also present in a majority of real-world computing systems. In particular, they allow an attacker to circumvene the intra- and inter-process isolation implemented by both the CPU and the Operating System, enabling them to read data from memory locations that they should not have access to. In principle, CPUs do support privilege checks and are able to refuse the execution of instructions that would lead to an unprivileged memory access. However, the vulnerabilities hinge on common optimizations implemented in nearly all modern processors, which may lead to certain abusable side-effects. This can allow an attacker to read supposedly inaccessible data through a timing-based side-channel attack. As a result, if an attacker could execute code on your system, they could, in theory, abuse *Meltdown* or *Spectre* to steal secret keys and passwords, or even map the entire system memory. This fact made these vulnerabilities a dreaded influence in the months following their disclosure.

While “Meltdown” is based on a flaw primarily present in a wide range of Intel processors, *Spectre* is a more general issue that applies to practically all processors that implement *Speculative Execution* – including Intel, AMD and ARM. As these vulnerabilities are found in the hardware of the system, a fundamental fix to the problem would require the installation of a newly designed CPU that does not have these flaws. Although software mitigations to these issues have also been released, they can not provide perfect security and are also known to have a major impact on the performance of the system. Hence, *Spectre* in particular may remain an important vulnerability even for the foreseeable future.

The goal of this laboratory exercise is to get familiar with the *Spectre* attack. We will be utilizing the CPU cache as a side channel to steal a protected secret. The technique used in this side-channel attack is called *FLUSH+RELOAD* [3]. Since the Spectre attack is sophisticated and abuses complex underlying mechanisms, we will break it down into several small steps that are easy to understand and perform. In the first two tasks, we will study the *FLUSH+RELOAD* technique. Then, the source code developed in these tasks will be used as a building block in the remaining tasks.

Preparation of the Experimental Setup

The topics of this lab are highly dependent on the underlying hardware and software. In particular, the behavior of your CPU and your CPU cache may lead to very specific issues that could cause trouble in solving the tasks. For this reason, we recommend that you *avoid solving the tasks on your own computer*. Instead, we are providing you with a lightweight remote machine in which you can run your experiments. Precise instructions on how you can access your machine will be provided to you through moodle.

Furthermore, the way you compile your code might cause a difference in its behavior. Therefore, it is recommended that you disable all code optimizations using the `-O0` flag. Furthermore, you should add the `-march=native` flag to your compilation using `gcc`. This flag forces the compiler to enable all instruction subsets supported by the local machine. Hence, your usual compilation procedure should look something like this:

```
$ gcc myproc.c -O0 -march=native -o myprog
```

Tasks

Task 1: Reading from Cache versus from Memory

When running a program on a computer, the loading and management of data tends to play a crucial role. The next instructions of the program must be looked up, variable values must be loaded, required files must be opened up and read. Naturally, all this data can not be “loaded” within the small data registers of the CPU at all times. Instead, it is saved in bigger and cheaper data stores, such as the hard disk drive, and loaded as-needed. Nowadays, most computing systems employ a hierarchy of memory banks to save and work with data – ranging from abundant, yet slow space on the hard disk, over the faster yet smaller main memory (RAM), up to very fast yet small caches within the CPU. In the briefest possible explanation, the CPU cache works by keeping recently used pieces of data in a fast-to-access data store, thus speeding up memory accesses if the same piece of data is needed again. Together, these differing technologies allow reasonably fast memory accesses in most practical use cases.

In this task, we begin by examining the effects that the cache has on the time it takes to access data at a given memory location. Examining these results will be a crucial step in developing our Spectre-exploit. To this end, you should write a piece of code that demonstrates how long it takes to read a one-byte variable (`uint8_t`) when it is loaded in the cache (“cache hit”), and when it is *not* loaded in the cache (“cache miss”). Based on this, you shall determine a threshold that allows you to decide whether you encountered a cache hit or a cache miss based on the access times you observe. Since the exact timing will differ slightly (or sometimes greatly) on each run, you should run your measurement code at least 100 times and analyze the results in a suitable manner (e.g. averages, medians, boxplots, ...) to make an informed choice on the threshold you will use for the future tasks.

Hints:

1. Since we are directly interacting with low-level system resources, a lower-level language such as C++ is recommended to be able to access them.
2. In order to ensure that your variable is not inside the cache, you may explicitly *flush* the cache line associated with its memory address using the `_mm_clflush` function from the `emmintrin.h` library. To steer clear of confusing issues, you may have to load the variable of interest once by assigning/reading it before performing the flushing operation.
3. It is recommended that you measure your timing in CPU Cycles. This can be done using the `__rdtscp` function from the `x86intrin.h` library.
4. When organizing the access times in a text file, it is easy to make a boxplot using *gnuplot* or *Python’s matplotlib*. If you plot cache-hits and cache-misses side-by-side, also considering outliers, it will be easy to compare the times and choose an ideal threshold.

Task 2: Using the Cache as a Side Channel

Now that we can recognize the difference between a *cache hit* and a *cache miss*, we are able to use the cache as a side channel to determine whether some memory address was accessed previously or not. In this task, we will utilize this fact to leak our first secret. Namely, let us assume that within a process, a victim function outside our control uses a secret one-byte-value to index an array. I.e. if s is the secret value, then at some point, the victim function will access the array element at index s. The attacker wants to find out this secret value, but has no access to the internals of the victim function. However, he is able to manipulate the cache and call the victim function whenever he wishes.

In the following, we will employ a side channel technique called *FLUSH+RELOAD* [3]. Figure 1 illustrates this technique, which consists of three steps:

1. *FLUSH* the entire array from the cache memory to ensure that no part of the array is 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. When we encounter an array element with a very fast access time (below our threshold), it is likely that the element was retrieved through a *cache hit*. This implies the element was previously accessed by the victim function. Thus, we can find out what the secret value is by examining which of the array elements was accessed before.

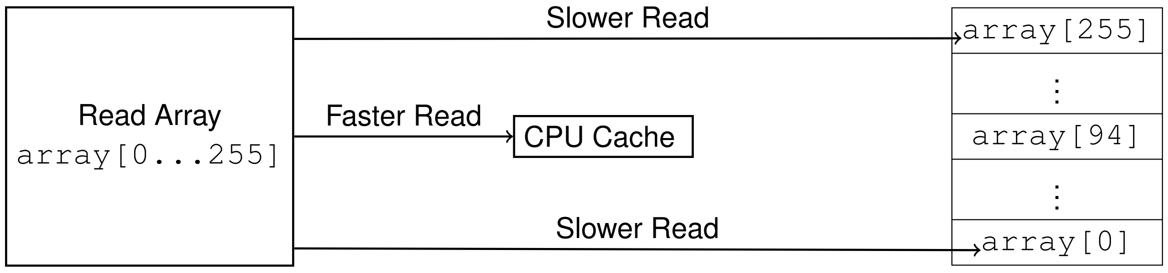

Figure 1: An overview of the side channel attack.

Your task is to write a small program that uses the *FLUSH+RELOAD* technique to find out a one-byte-secret as described above. To this end, we provide you with a program skeleton `FlushReload.c` that encompasses this experiment setup, allowing the victim function to access one of 256 possible array elements. Your program should be able to succeed regularly in extracting the secret, even if your program might not succeed *all* the time to discover the (correct) secret.

In addition, you will notice that the code uses a custom `DataBlock` datatype, of which only a small piece of data is actually accessed. Discuss the purpose of this datatype and why it was introduced for this experiment. Would you also be able to find out the secret if we were using a simple array, such as `uint8_t array[256]`? What implications does this have for the attack outlined above in a real-world setting?

Hints:

1. You should think about the size that a cache line typically has in modern processors - it usually does not match exactly the size of the variables we want to retrieve. Why can this be problematic?
2. If you use a very modern computer, you may struggle to get the FLUSH+RELOAD technique to work. This has to do with prefetching, in which the CPU may load data it expects to be needed in the future into the cache, even if said data was not accessed recently. Therefore, you may discover a surprising amount of cache hits when iterating over the array, or when examining neighbors of the secret element. Fortunately, this effect is not an issue on the virtual system we provide to you.
3. When you want to test whether your program works, you should avoid spamming your compiled program several times in quick succession. This can lead to confusing side effects, which are best avoided by allowing a little bit of time between each program launch.
4. If you get a lot of wrong results, you may have to review the threshold you chose.

Task 3: The Idea of Spectre — Side-Effects of Speculative Execution

The Spectre attack relies on an important feature that is implemented in practically all modern CPUs: Speculative Execution. To understand what this feature entails, let us examine the following piece of code:

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

On a high-level view, this code checks whether `x` is less than `size`. If this is the case, the variable `data` is updated. Otherwise, no changes are made, i.e. the code in line 3 will not be executed.

While this explanation of the code example holds true from a high-level view, things get more complicated when looking inside the CPU. In the piece of code above, line 2 involves two operations at a microarchitectural level: load the value of `size` from memory and compare the value with `x`. If `size` is not in the CPU cache, it may take hundreds of CPU clock cycles before that value is read and the comparison can be executed.

So, instead of sitting idle, modern CPUs try to think ahead by predicting the outcome of the comparison and speculatively executing the branches based on the estimation. Since such execution starts even before the comparison is finished, this is an example of *out of order execution*. Before doing the out of order execution, the CPU stores its current state and value of registers. When the actual value of `size` is known, the CPU checks the actual outcome. If the prediction is true, the speculatively performed execution is committed and there is a significant performance gain. If the prediction was wrong, the CPU reverts back to its saved state and all results produced by the out of order execution are discarded. Hence, it seems like the affected line of code was never executed... except for one little detail.

Can you see the side effect that may occur because of this optimization done at a microarchitecural level? How can this be of use in building a more complicated attack?

Task 4: Experimenting with the Spectre Principle

Having understood the side-effects of Speculative Execution and the out-of-order execution it may result in, we are able to examine how to leverage these effects in practice. Once again, you are provided a code skeleton `SpectreExperiment.c`, which you should fill with the implementations you made in the previous tasks. Examine the code and work to understand what is being done here. Then compile and test it. How does the code end up extracting the secret value successfully (in a fair amount of cases)? In case you are having issues, be sure to keep the previous hints and compilation instructions in mind. Note that it is to be expected that the experiment will fail at times - we will be dealing with this in the following tasks.

In particular, you should elaborate on the following questions:

1. What is the role of the for-loop marked as *Exhibit A.1* in the code? Why is the victim function invoked with small numbers first, before later invoking it with a larger value (see *Exhibit A.2*)? How does this relate to the predictions the CPU makes as part of the speculative execution?
2. See what happens when you comment out the lines marked as *Exhibit B* and the lines marked as *Exhibit C* individually. What do you observe, and what role do these lines play exactly for the success of the attack?
3. Examine what happens when you replace the line marked as *Exhibit D* with the following: `victim(i+20)`. Does the attack succeed? Why?

Be sure to revert all the changes you make when answering these three questions, as you want to avoid the side-effects they cause in the coming tasks.

Task 5: The Spectre Attack

In the previous tasks, we have gradually worked to understand how the principle of Spectre works. Due to the fact that modern CPUs generally do not flush the cache after having performed a wrong guess for the speculative execution, we are able to detect array accesses even from parts of the code that should have never been executed. This allows us to leak what location of an array would have been accessed during execution of the code. In this task, we will examine how this principle can be abused to leak not just array accesses, but *actual data at arbitrary locations in memory*. With this, it should become clear how dangerous Spectre really is, as it may allow an attacker to dump interesting parts of the memory, including but not limited to cryptographic keys or important access tokens. It should be noted that Spectre is even able to do this across different processes, since even though process isolation protections are aided by hardware, they are still fundamentally reliant on branches that can fall victim to speculative execution.

To avoid further complications during this task sheet, we will focus on the case of stealing a secret from within a single process, just as we did in the previous tasks. A common example where this may be of interest is the Browser, which employs a sandbox mechanism to ensure that individual web pages run in an isolated environment. Although each web page is contained by the same main process, the sandbox ensures that web pages are not able to access the data of another webpage, which would have dangerous ramifications. Most software protections fundamentally rely on simple condition checks to decide whether an access should be granted or not. As we saw previously, with the Spectre attack, we can get the CPU to execute (out of order) a protected code branch even if the condition checks should fail, thus allowing us to entice certain implicit accesses that enable us to recover secret data.

In our experiment setup, we assume that an attacker is permitted to access certain areas in memory (here, realized by a `buffer`). On each memory access that he performs, a function checks whether his request lies within the bounds of his accessible memory. If it does, the data in memory is returned – if it does not, the data at the location is *not* returned. There may be other data in memory that the attacker may be interested in, such as a `secret` string. However, even if he knows the location of this data, an attacker is unable to retrieve it by normal means. This is one of the most simple examples of a sandbox. The following code realizes this basic functionality:

```c
unsigned int bound_lower = 0;
unsigned int bound_upper = 9;
uint8_t buffer[10] = {0,1,2,3,4,5,6,7,8,9};
char    *secret    = "Some Secret Value";

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

Listing 1: Sandbox Mechanism

To practice the Spectre attack, we provide you with another code skeleton `SpectreAttack.c`. The above mentioned sandbox is implemented here, which allows an attacker to access a location in memory, namely his dedicated `buffer`. The `restrictedAccess` function is meant to represent the underlying access controls in a software-based sandbox – thus, any attempt to access parts of the secret string should also be done through the `restrictedAccess` function, by providing the right offset. Of course, such out-of-bounds accesses will fail for the attacker. To keep the implementations simple, you shall also assume that our `array` is still (directly) accessible to the attacker, without having to go through the `restrictedAccess` function. Furthermore, you shall assume that the attacker knows the memory location of the secret string. Finally, the attacker is also able to flush the cache however he likes, since these functions are generally not protected within a process.

Your goal in this task is to develop an attack that allows you to read out the first byte / character of the secret string using the FLUSH+RELOAD technique we used in the previous tasks. To achieve this, you will want to setup the cache in such a way that the RELOAD step is able to tell you the secret value. This means you must entice a certain array access by abusing the techniques we examined in the previous tasks. Do note that out-of-order execution may go through several instructions before the result of the condition check becomes available. In the end, you should have filled-in the `SpectreAttack.c` skeleton in such a way that you are able to steal the first character, “S” (83), through the side-channel in a semi-reliable manner. We will work to improve the reliability of the attack in the following tasks.

Task 6: Improving the Attack Accuracy

In the previous tasks, you have likely struggled with some more or less unreliable results. Sometimes, you may not find any value in the array that appears to be within the cache. If you are unlucky, you might even extract the wrong secret value in some cases, when multiple values are present in the cache. These uncertainties (“noise”) in the side-channel are caused by the complicated cache-management strategies employed by the computer. Their effects differ from system-to-system, yet their general goal is to load and retain data in the cache that has a high likelihood of being useful in the future. Amongst other things, this can lead to the CPU loading values into the cache preemptively, even if they have never been accessed before.

One major way to maximize the accuracy of your results is by choosing an ideal threshold. Further tricks already employed in the code skeletons help in making the attacks more accurate. Nonetheless, remaining uncertainties may require you to run your code multiple times to achieve your desired results. Fortunately, a sufficiently patient attacker is usually able to do this in a real-world setting. Therefore, your goal in this task is to work on the reliability of your attack. In particular, your task is to implement an automation within your C-code that runs your attack multiple times (when no result is found) and, based on the results, is able to provide a reliable final result to the user. In case you struggle with the discovery of wrong secrets, you should also implement an improvement to reliably deal with this issue.

Hints:

1. If you decide to automate your attack in a sequential manner, you may notice bad results when you run your attack in quick succession. To remedy this, it is necessary to add a small delay in between.
2. You may face weird side-effects when realizing a delay using the `usleep` function. Instead, you may have more success with a technique called “busy waiting”. Take some time to learn about the difference between these two approaches. Once again, keep the compilation recommendations in mind to ensure your code runs as intended.

Task 7: Stealing the Secret String

After finishing the previous tasks, you are now able to reliably steal secret bytes at arbitrary memory locations. Extend your script to steal the entire secret string using the Spectre attack and print the whole string to the console. Once again, you may assume that you are aware of the position and even the length of the field containing the secret data.

Preparation of Your Submission

Your submission for this lab should contain the final versions of any and all source code you have written and files you have generated, along with a detailed writeup about the tasks you solved. Your writeup should be given in an appropriate format, containing detailed descriptions of how you solved each task, what theoretical insights you gained, what observations you have made as well as images / screenshots / listings to aid the understanding of your approach. Your writeup should be submitted in PDF format.

This time around, the code you submit is of particular importance. Be sure to submit the code samples for all tasks in which you fill the provided code skeletons and / or extend on your existing code. Additionally, your Writeup should contain detailed answers to the theoretical questions posed throughout the task sheet, and reasoning for the design decisions that you performed in the development of your code snippets.

Please submit your solutions in a packed zip file named after your full name, with the following structure and minimum required contents:

```
firstname_lastname.zip
├── Writeup.pdf
├── CacheMeasure.c
├── FlushReload.c
├── SpectreExperiment.c
├── SpectreAttack.c
├── SpectreAttackImproved.c
└── SpectreAttackFull.c
```

Preparation for the Q&A Session

Prepare yourself for a Q&A session of up to 40 minutes. During the Q&A session, you should be prepared to *explain and demonstrate* how you solved the tasks. Take special care to explain how your solution works and why you took the approach you took. Note that simply presenting a piece of code without any explanation will not yield any points! Last but not least, you should be ready to answer spontaneous (theoretical) questions asked by the reviewer in relation to the topics of this lab.

References

[1] P. Kocher, D. Genkin, et al. “Spectre Attacks: Exploiting Speculative Execution”. In: *arXiv e-prints*, arXiv:1801.01203 (01/2018), arXiv:1801.01203. arXiv: 1801.01203 [cs.CR] (cit. on p. 1).

[2] M. Lipp, M. Schwarz, et al. *Meltdown*. 2018. arXiv: 1801.01207 [cs.CR]. URL: https://arxiv.org/abs/1801.01207 (cit. on p. 1).

[3] Y. Yarom and K. Falkner. “FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack”. In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, 2014, pp. 719–732. ISBN: 978-1-931971-15-7 (cit. on pp. 2, 5).