Internship report

System-level design for the assessment of shared resources effects on multiprocessor systems security

Khazaal Farah & Couturier Ulysse

SUPERVISORS :
Le Nours Sebastien, Mendez Real Maria

Contents

1 Introduction **2**
1.1 Context . . . 2
1.2 Problematic . . . 2

2 Background and Discovering gem5 **3**
2.1 What is gem5 ? . . . 3
2.2 Architecture of gem5 . . . 3
2.3 Targeted architecture . . . 5

3 Basics of attacks **6**
3.1 Cache memory . . . 6
3.1.1 Importance of having a cache hierarchy . . . 6
3.1.2 Cache functioning . . . 6
3.2 Flush and Reload . . . 10
3.3 Prime and Probe . . . 11

4 Modelization and experiments **13**
4.1 Model 1 : One Core . . . 13
4.1.1 System Behavior . . . 13
4.1.2 Flush and Reload attack . . . 19
4.1.3 Conclusion . . . 25
4.2 Model 2 : Two Cores . . . 25
4.2.1 System Behavior . . . 25
4.2.2 Flush and Reload Attack . . . 31
4.2.3 Prime and Probe attack . . . 36
4.3 Limitations of Syscall Emulation Mode . . . 44
4.3.1 List of Limitations . . . 45

5 Conclusion **47**

1 Introduction

1.1 Context

In industry and research, the race for progress motivates all players in the electronics industry to always push their limits in terms of optimization. Whether in terms of execution speed, energy consumption, or portability, design hours are allocated to these themes. However, the hardware security aspect struggles to impose itself, despite being an essential aspect, knowing that according to the ANSSI (French National Information Systems Security Agency), between 2019 and 2020, the number of victims of cyber attacks has quadrupled.

It is in this context that this three-month internship takes place, at the IETR laboratory of Polytech Nantes (France). As a study support, it is the gem5 virtual prototyping software that has been used. The project's aim is to study the possibilities given by gem5 software to analyze processor architecture's vulnerabilities. This project focuses on exploring the capabilities of gem5 software for analyzing vulnerabilities in processor architectures. The main emphasis lies in understanding how the security of multiprocessor systems is influenced by the use of shared resources.

The work is based on the github sources [2] provided by the gem5 community, and on the work of Pierre Ayoub and Clémentine Maurice: "Reproducing Specter Attack with gem5: How To Do It Right?" [7]

1.2 Problematic

According to this introduction, the question that will be examined throughout this report is as follows: What level of detail is required for modeling a cybersecurity system using the Syscall Emulation mode of gem5?

2 Background and Discovering gem5

2.1 What is gem5 ?

Gem5 is a merge between two simulators from 2011 : M5 and GEM5 [4]. The role of gem5 is to provide a cycle-accurate simulation platform for computer architecture research. It enables developers to analyze, model, and experiment with various designs and configurations. In today's rapidly evolving landscape of processors and systems, the need for a reliable and trustworthy configuration becomes crucial. Moreover these systems are becoming increasingly complex and difficult to comprehend. gem5 serves as an excellent solution to address these challenges by allowing developers to deeply understand and test their solutions on simulated devices, avoiding issues related to cost or delivery delays.

Researchers can leverage gem5 for malware analysis, creating safe environments to study malware behavior. It aids vulnerability assessment by analyzing software behavior under potential threats, contributing to the identification of security flaws and impact assessment of attacks. Additionally, Gem5 assists in exploit development, enabling the study of exploits across different architectures and aiding in patch development. It proves valuable in hardware security, simulating hardware components to study hardware-based threats.

2.2 Architecture of gem5

The architecture of gem5 encompasses various key components, including SimObjects that represent both hardware and software elements (DRAM , CPU , Memory systems , Cache memories etc...). The system component plays a crucial role in connecting and managing interactions among SimObjects. Additionally, gem5 incorporates CPU models for different architectures, a memory system that simulates cache hierarchies and memory components, and simulation control mechanisms for parameter configuration and data collection (including multiple types of debugging). These components collectively contribute to the comprehensive functionality and adaptability of the gem5 simulator, enabling its users to perform diverse and efficient simulations. To work, gem5 combines different programming languages to create an easy manipulating software.

Initially, all simulated entities, referred to as SimObjects, are coded in C++ based on an open-source coding style principles [1]. These highly detailed objects are entirely modifiable, allowing alterations to the source code.
Subsequently, by incorporating m5 libraries into Python code, these C++ de-

scribed SimObjects can be accessed within a Python script. Python is selected for its user-friendly class implementation.

The Python script's role is to construct the simulated system and initiate the simulation, serving as the script invoked during gem5 construction. Ultimately, upon creating and storing the script, the gem5 build through scons can be initiated, specifying the intended simulated system. The build encompasses distinct options: debug, opt, and fast, with the latter omitted. The debug mode facilitates code debugging using tools like GDB (GNU debugger), while opt provides an optimal balance between simulation speed and internal visibility. Subsequent to launching the simulation, gem5 generates three debug files containing statistics, configurations, and other customizable flags indicated in the shell. These files are comprehensively explained in the github [3]. The behavior of a general simulation is succinctly summarized in Figure 1.

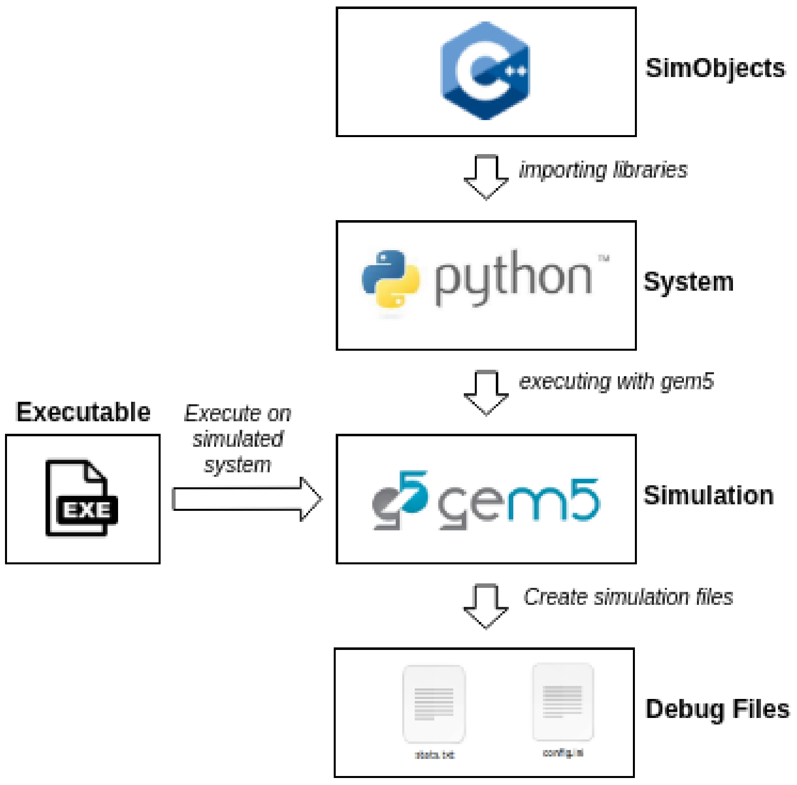


Figure 1 – Simulation files link in gem5

2.3 Targeted architecture

Since gem5 is designed to facilitate complete customization of electronic devices, it offers a lot of distinct features, necessitating selections to be made for each component. Basically, main used components are : CPU's, level 1 Instruction and data Caches, shared level 2 cache memory, data buses, and RAM.

The selected RAM is a DDR3 1600 MHZ 8x8. This one is recommended by the gem5 tutorial as it is the simplest memory. All other RAM memories inherit from it. The data buses are implemented using an Xbar architecture, also given in the tutorial. In terms of cache memory, gem5 provides two options : classic caches, and Ruby caches. Classic cache have been chosen due to the complexity of the Ruby Cache and its advanced capabilities which are not our priority. Classic caches are adequately configurable for our study, offering a fundamental configuration of replacement policy, indexing policy, and other modifiable attributes. The selected indexing policy is "Set Associative Mapping," while the chosen replacement policy is "Least Recently Used". Finally, all CPU used are "ArmO3CPU". First because they are out of order, with a behavior closer to modern processors, and they implement branch predictor. Secondly, they are based on Arm architecture. This architecture is often implemented in embedded systems like phones or connected devices, the instruction set is also easier to understand. In gem5, O3 CPU's also allow to use a software called Konata, which gives a graphical overview of execution behavior of the simulation. The table 1 summarizes all components of targeted architecture.

CPU	ArmO3CPU
Cache	Classic cache
Data bus	XBar
Memory	DDR3 1600 MHZ 8x8

Table 1 – Table of targeted architecture's characteristics.

3 Basics of attacks

Now that gem5 and the architecture of simulated device have been presented, a quick explanation of the attacks has to be done to understand why are modern processors vulnerable.

3.1 Cache memory

3.1.1 Importance of having a cache hierarchy

The latency of execution is a problem that occurs in a lot of electronic applications. When an instruction is executed, all steps of its completion are studied to be optimized.

Searching in the memory system for a data or an instruction can take a large amount of clock cycles. So here comes the importance of cache memory.

Cache memories are placed between main memory and cores. Very recent data are stored there in case they will be reused soon to avoid going through the entire main memory. This is why cache memories are so important in modern architectures, and are in almost all devices. As cache memories are expensive, they are way smaller than main memories. To optimize their use, they are often divided into a cache hierarchy. Here a maximum of two level of cache will be studied but there can be more levels.

3.1.2 Cache functioning

As cache memories are used in all modern architecture, understanding their vulnerabilities plays an important role in enhancing cyber-security. Before digging into security, the behavioral aspect of cache memory has to be fully understood.

In simple terms, when a process requests data, the core checks if that data is already in the cache. If it is, the data is quickly accessed from there, saving time (this is called a cache hit). If the data isn't in the cache, the core retrieves it from the main memory (this is a cache miss). If the data is expected to be used again, it's stored in the cache for faster access next time.

Caches have specific rules for how they organize and replace data, which includes indexing and replacement policies

3.1.2.1 Indexing policy

To determine the location to which a memory block is mapped based on its address, our system utilizes the Set Associative mapping technique by default. For a clearer understanding of Set Associative mapping, it's helpful to first grasp the concepts of Direct Mapping and Associative Mapping. Let's start with Direct Mapping:

Direct Mapping is a mapping technique used to determine the physical address corresponding to a given memory location in a cache or memory hierarchy. It divides the memory address into three components: the tag, the index, and the offset. The tag uniquely identifies a memory block, the index selects the cache or memory location, and the offset specifies the position within the memory block.

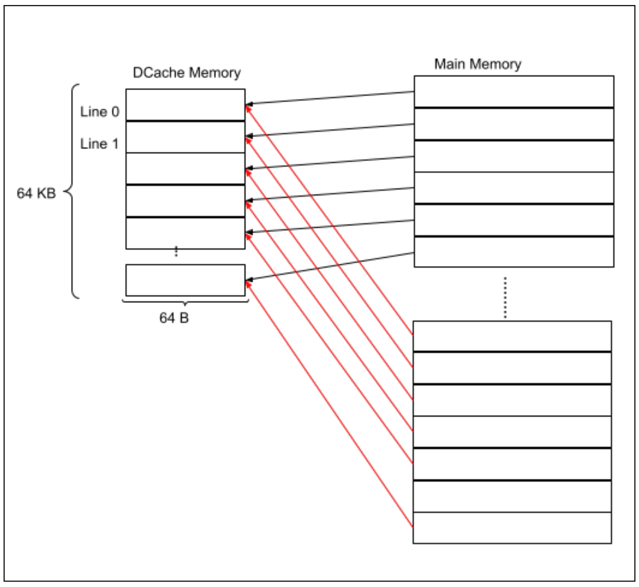


Figure 2 – Direct Mapping

On the other hand, Associative Mapping is a technique that allows a memory block to be stored in any available cache or memory location without restrictions. Each memory block is associated with a tag that uniquely identifies it.

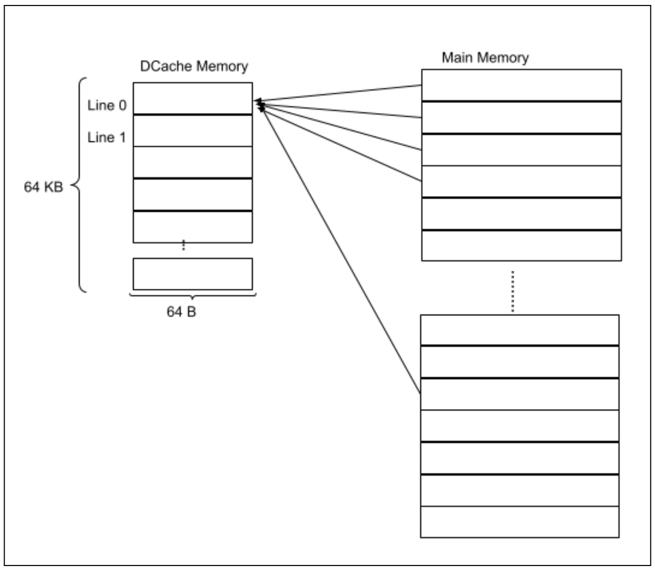


Figure 3 – Associative Mapping

Set Associative Mapping combines elements of both Direct Mapping and Associative Mapping to strike a balance. It aims to reduce conflicts or collisions that occur in Direct Mapping while minimizing the hardware complexity associated with fully Associative Mapping.

In Set Associative Mapping, the cache is divided into sets, and each set contains multiple cache lines or memory locations. Each memory block is mapped to a specific set based on its memory address. Within each set, the memory blocks are differentiated using tags. The index determines the set number, and each set contains multiple cache lines or memory locations. The index represents the block number within a set rather than the entire cache.

By employing Set Associative Mapping, our system achieves a balance between reducing conflicts and minimizing hardware complexity, allowing for efficient and effective memory block mapping.
Here an example of 2-Set-Associative mapping (2 means that we have 2 lines per Set).

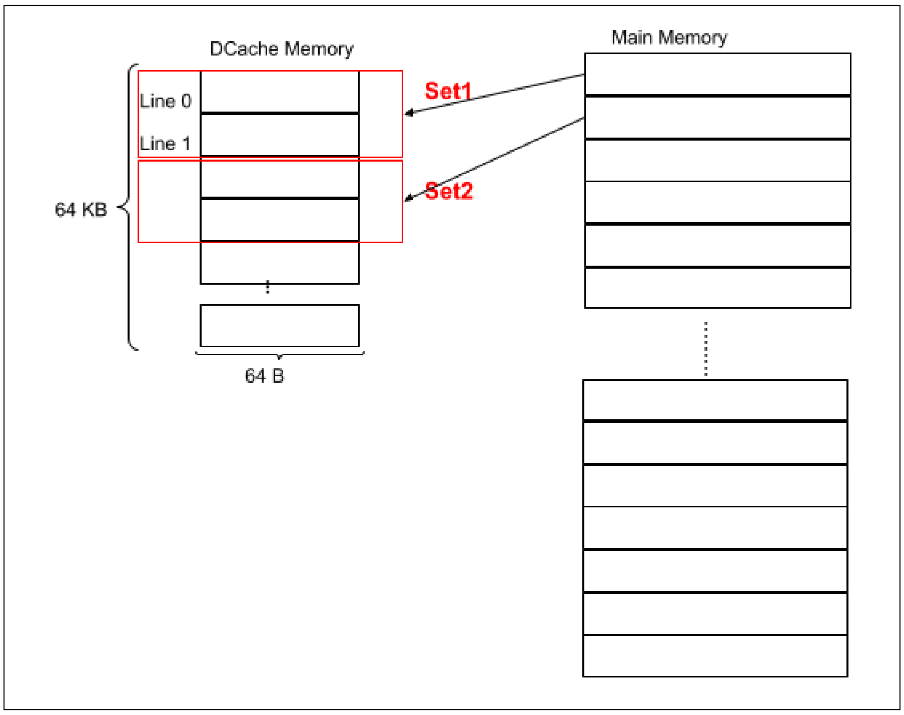


Figure 4 – 2-Set-Assocative Mapping

The cache write policy used is the "Write Back" Method . More information are available on the website researchgate.com [8].

3.1.2.2 Timing vulnerabilities

The main characteristic of cache leading to backdoor for malicious processes is all about timing. Because cache memories reduces a lot read and write timings, a difference in execution time is observed if a process uses more cached data. The figure 5 below explains this timing difference for two cases.

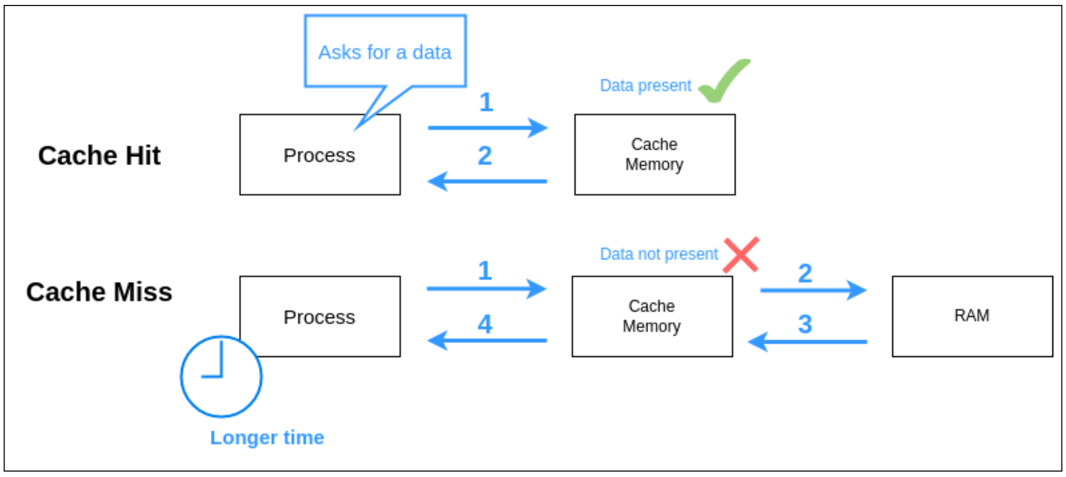


Figure 5 – Timing difference in execution

Here the initial data request is processed in just two steps : from process to cache, and cache to process. For the second request, the amount of steps has doubled. Request has to go to cache memory, but because the data is not stored there it goes to the RAM, before going back to the process with a longer execution time than the first case. By using instructions to measure execution time, a process can, by using this aspect of cache memory, know if its data was in the cache or not. Starting from this point, a lot of attacks use this aspect to steal information.

3.2 Flush and Reload

First attack is called Flush and Reload. It uses a particular assembly instruction called "flush" which evicts a data from cache memory. In modern processors, some libraries and their address can be shared between two process. The behavior of a flush and reload is explained in the figure 6 below.

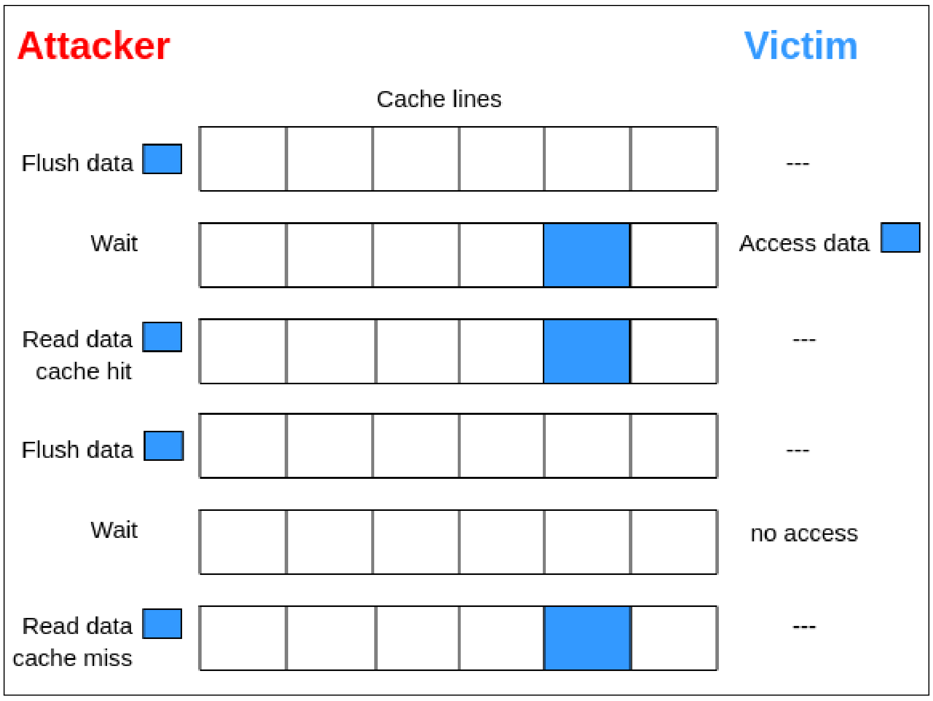


Figure 6 – Execution of a flush and reload

Cache memory is represented in the center. White boxes represents empty cache lines, and blue ones are cache lines where a data is mapped. Victim and attacker are sharing two things : a data and a cache. First attacker flushes the blue data, leading to an eviction from the cache and an empty cache line corresponding to this address. Then, after waiting for the victim to execute he can know if this shared data has been accessed or not by looking at the reading duration. If the read (or reload) time is shorter than a threshold, then data is cached and has been accessed by the victim. If it is longer than this threshold, then this data has not been accessed. If he knows what is this data, he can deduce what did the victim. The threshold varies according to the architecture and must be established to ensure proper execution. A paper from Yuval Yarom and Katrina Falkner [9] describes then how an attacker can crack a key or any password using this strategy.

3.3 Prime and Probe

In a Prime and Probe attack, the attacker exploits the ability to write a big volume of data, sufficient to displace the victim's data from the cache. If the attacker fills the cache with its data and then wait for the victim to execute it's code, he can know which lines of the cache have been accessed by measuring execution time and comparing it with a threshold. In fact data accessed by the victim are supposed to evict attacker's ones, so by reloading them, attacker knows which data have been evicted because of cache misses.

To visualize what is supposed to happen in the cache, the state of the cache of three main states of these behaviors is summarized in figure 7.

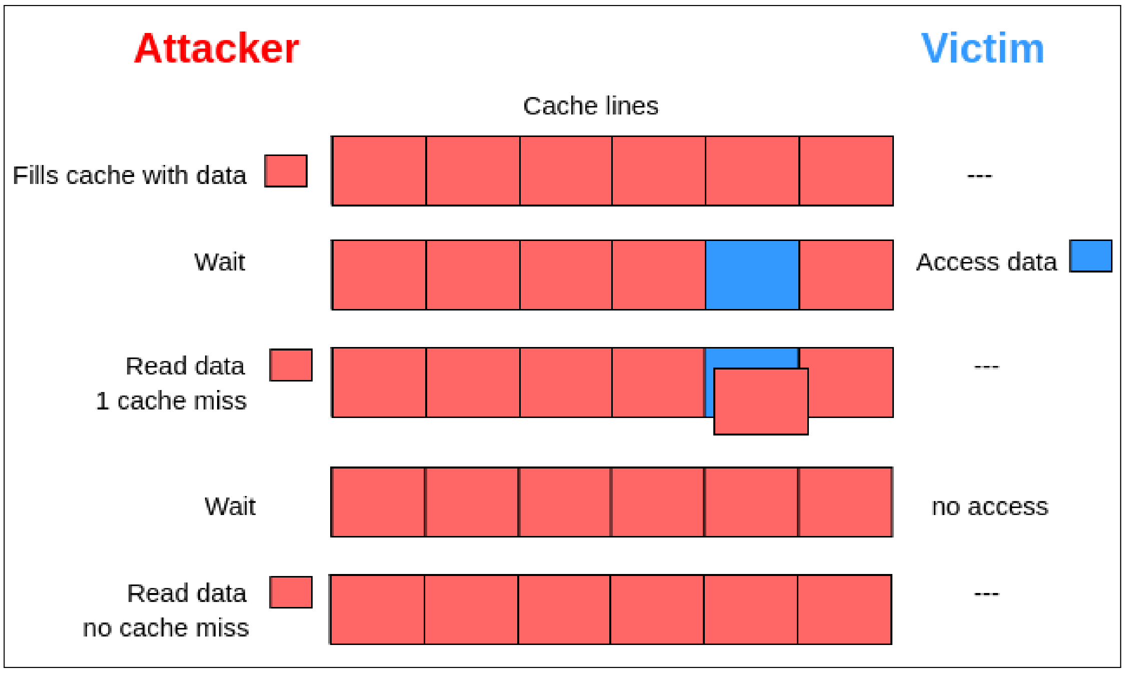


Figure 7 – Execution of a prime and probe

Here, only the last level of cache is represented in the center because it is shared one. Because of this, it is likely to store information coming from different process, like data from attacker, and victim, at the same spot. The attacker first fills cache with its data, and waits for victim to access (or not) to some specific locations. At the end it reads again all data, and if one of them takes more time to reload, it means that it has been evicted, so victim read something there.

3.3.0.1 Cache requirements for Prime and Probe

To work, prime and probe has to be executed on cache that are fully inclusive. This means that if a data is present in a level 1 of cache, it has also to be present in all others. Cached data has to be in all the hierarchy. This attack also requires to have a cache shared between different processes.

4 Modelization and experiments

4.1 Model 1 : One Core

Once that the supporting software and cache vulnerabilities used by attackers have been presented, the aim is to set up a system. This system will be stimulated by some external entities : victim and attacker, and it will be described using in and out descriptions, automatas and characteristics.

4.1.1 System Behavior

The system implemented in this experiment is presented below .

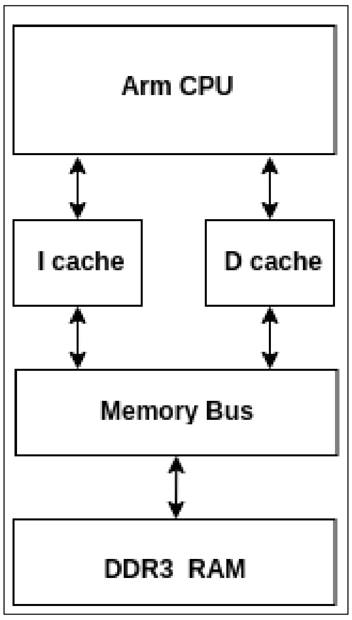


Figure 8 – One level System

4.1.1.1 Entities definition

In this experiment, a primary core assumes the dual roles of both the victim and the attacker. However, these roles are associated with distinct functions.

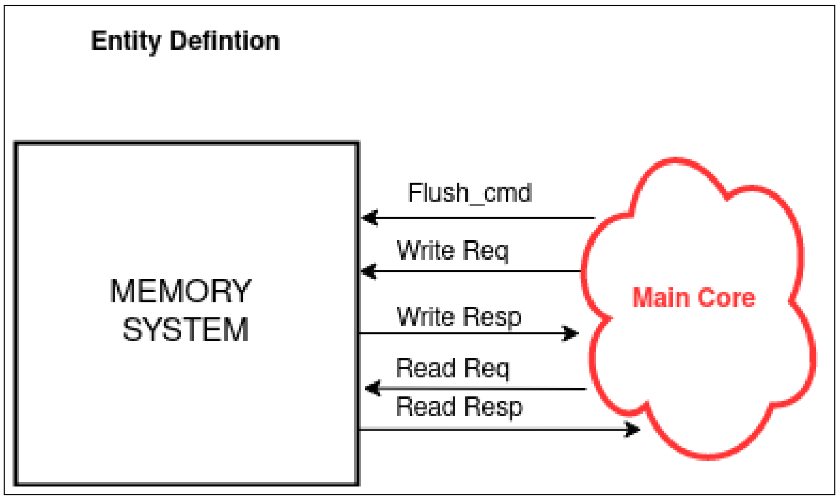
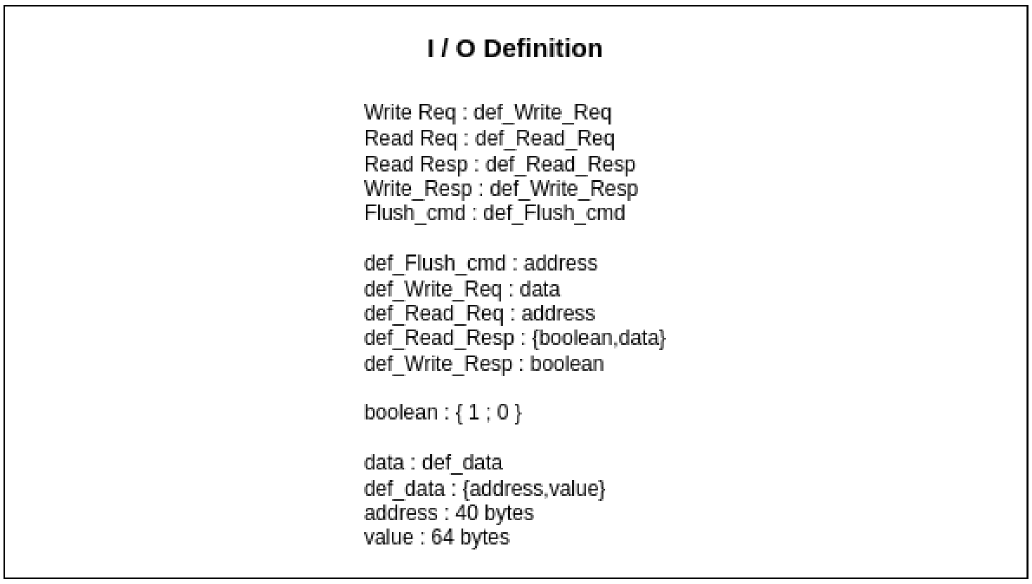


Figure 9 – Entities Definition

4.1.1.2 Definition of input and outputs

In this section , we will discuss each entity's input and output in more detail. The table below represents our system's inputs and outputs.



I / O Definition

Write Req : def_Write_Req
Read Req : def_Read_Req
Read Resp : def_Read_Resp
Write_Resp : def_Write_Resp
Flush_cmd : def_Flush_cmd

def_Flush_cmd : address
def_Write_Req : data
def_Read_Req : address
def_Read_Resp : {boolean,data}
def_Write_Resp : boolean

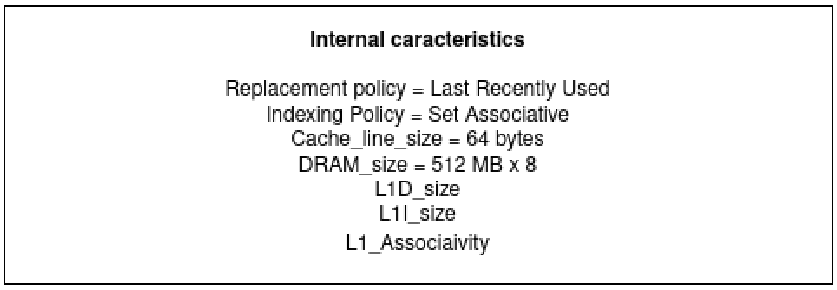
boolean : { 1 ; 0 }

data : def_data
def_data : {address,value}
address : 40 bytes
value : 64 bytes

Figure 10 – Definition of inputs outputs

Commands can be divided into two types: requests and responses. Requests tell the memory system where and what to read, write, or clear. The system then handles the request and provides either the read value (in the case of a read request) or a simple confirmation (a boolean) to show that the operation is done.

Apart from inputs and outputs, there are some internal features of the system that can be seen in the figure below Figure 11. Some of these features, like frequency, might not affect the model we've chosen, so not all of them are listed.



Internal caracteristics

Replacement policy = Last Recently Used
Indexing Policy = Set Associative
Cache_line_size = 64 bytes
DRAM_size = 512 MB x 8
L1D_size
L1I_size
L1_Associaivity

Figure 11 – Other important internal characteristics

4.1.1.3 Behavioral Description

Now we need to establish the way the system acts. This behavior is guided by various requests received from and sent to the attacker and the victim. The depiction in Figure 12 showcases a universal behavior, describing the system's actions regardless of the external entities' actions. The illustration is split into three parts for clearer viewing. The initial part summarizes the three possible requests to activate the system. Subsequent figures detail individual behaviors following each of these requests.

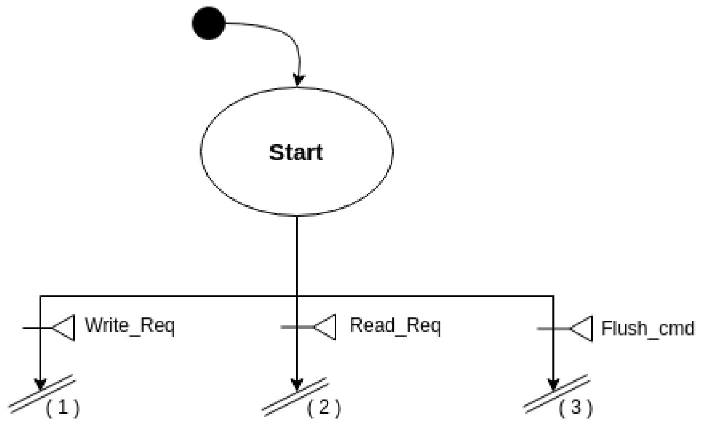


Figure 12 – First part of System behavior

The first Request described is present in the figure below.

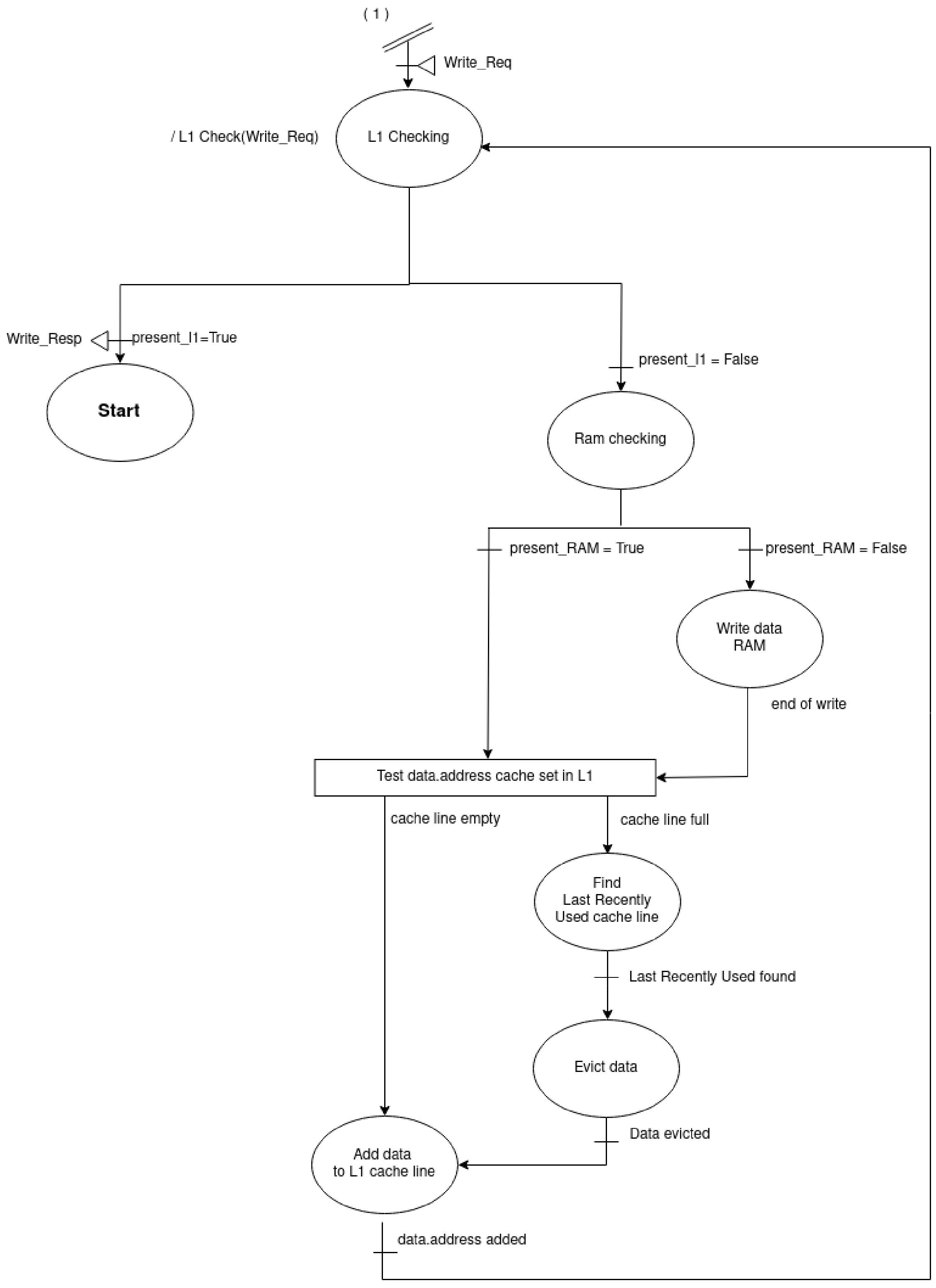


Figure 13 – First part of System behavior

Following a Write Request, the system performs a check to determine whether the requested data's address is stored in L1 cache. If the data's address is found in L1 cache, the data is written to the correct address, and the system transitions to its initial state. However, if the address isn't present in L1 cache, the system proceeds to check RAM. In a similar scenario, if the address is absent from RAM as well, the system writes the data into RAM. After this, a further examination takes place. The system evaluates whether the allocated cache line for the data's address is at its maximum capacity. If this condition is true, eviction process is triggered, necessitating substitution of the least recently utilized data with the new address.

After this procedure, the system reverts back to its initial state.

Now going to the read request, the figure below describes in detail how the system react in case of a read operation.

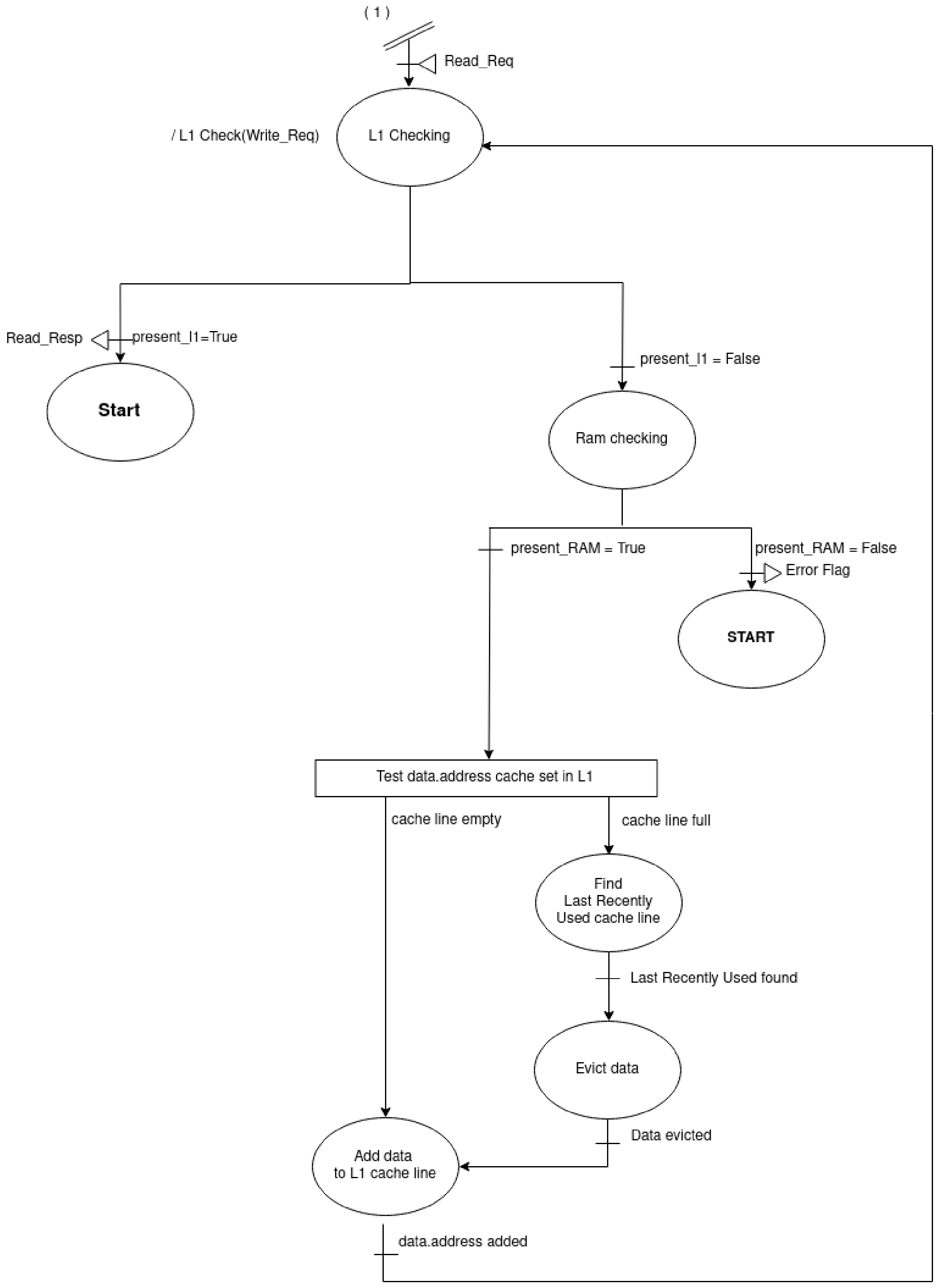


Figure 14 – Second part of System behavior

As discussed before for the write request, a read request follows the same process. The only difference lies in how RAM memory responds. In this case, if the data isn't found in the RAM, an error flag will be raised. This is because a read cannot be done if the data doesn't exist.

And finally the reaction of the system when a flush_cmd takes place will be detailed in the figure below.

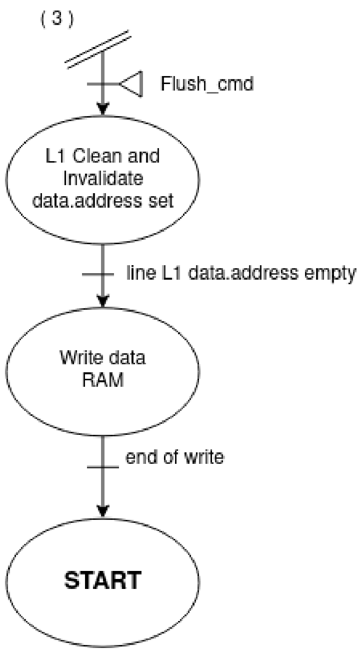

(3)
Flush_cmd
L1 Clean and Invalidate data.address set
line L1 data.address empty
Write data RAM
end of write
START

Figure 15 – Third part of System behavior

4.1.2 Flush and Reload attack

4.1.2.1 Definition of entities behavior

Let's begin by examining a simplified system that operates with a single level and a single CPU. The aim is to observe how it responds to a flush and reload attack.
The behavior of the entities can be represent by the automata below.

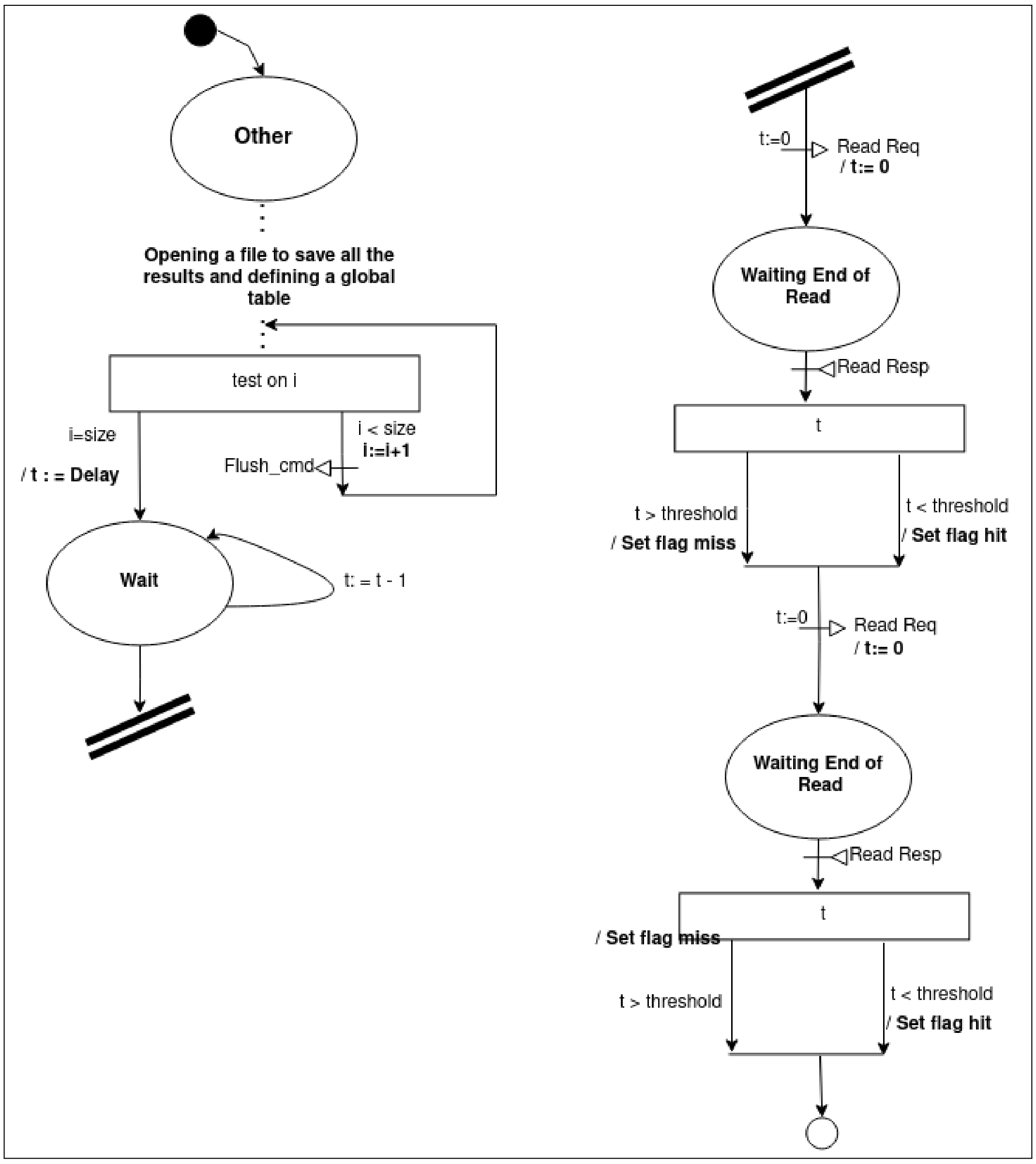


Figure 16 – Entities Behavior

4.1.2.2 Experiment

The experiment's aim is to showcase a real-world implementation of "Flush and Reload" attack within a controlled environment. By doing this, it helps show how certain systems could be vulnerable due to leaks of information based on timing. The realised code is present in the github [3] **Expectation**

The expected order is :

1) Clear the cache and prepare memory addresses to ensure clean measurement conditions.

2) From the victim's perspective:
- Simulate victim's actions using "access_time_victim_side" function.
- Access all memory locations pointed by the shared table.
- Measure access times.
- Ensure that no other instruction is pre-fetched after each access for accurate timing.
- Expect access time larger than a threshold (cache misses) due to data absence in the cache from previous flushing.

3) From the attacker's perspective:
-Simulate attacker's behavior with "access_time_attacker_side" function.
-Observe memory access times similar to victim's steps.
-Perform cache operations to maintain timing data integrity.
- Anticipate access time smaller than a threshold (cache hits) due to victim's prior access to shared data.

Simulation
Now, it is time to examine actual results obtained from the previous test and experiment described earlier. These results will provide us with a comprehensive understanding of behavior and performance of cache memory system throughout experiment.

The results are present in the figure below.

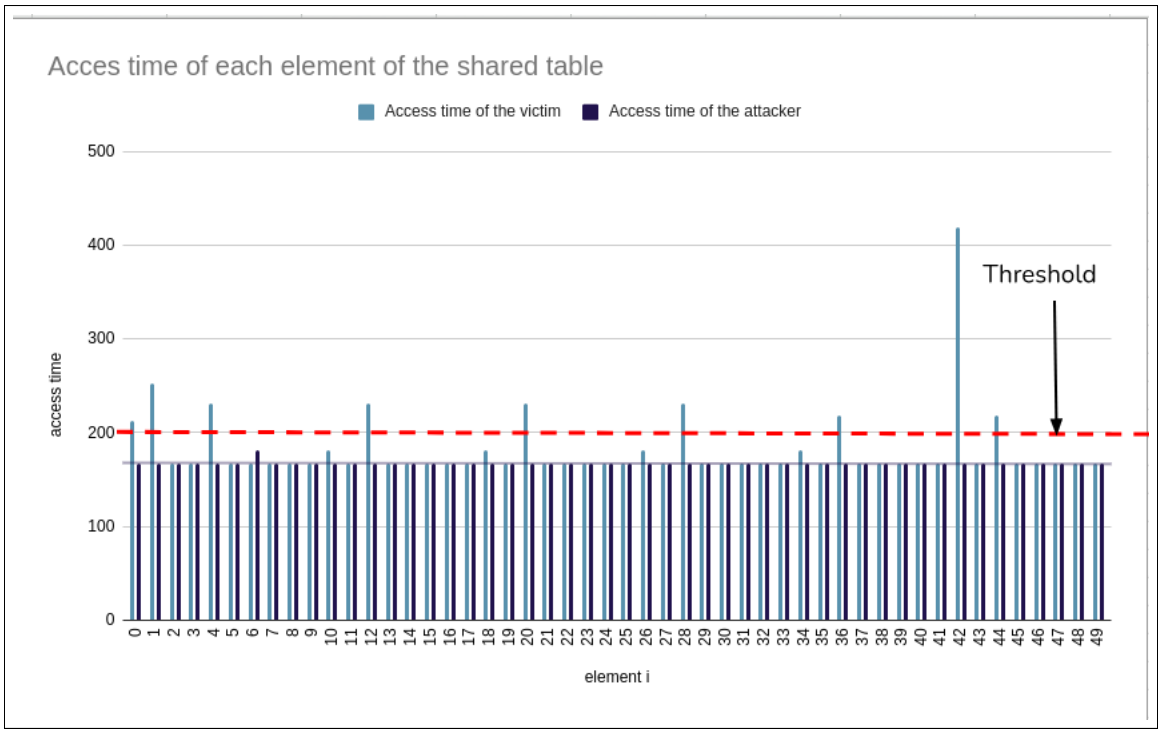


Figure 17 – Access time to the element of the table vs the elements

As expected, time it takes for the victim to access the shared table sometimes goes beyond 200, which is the threshold for cache misses. However, after each miss, noticing an instance of cache hits can be normal. This is because of pre-fetching mechanism that tries to anticipate all data needed. Put differently, in our scenario, the line size is 64B, and a table has been created with elements of long data type. As observed in the diagram, after a specific point, we encounter 8 hits following each miss. The system fetches not just the initial element of a line, but the entire line. In this context, a line consists of 8 elements, given the division of 64 by 8.

Looking at the attacker's perspective, consistently observing, the time taken to access the shared table remains consistently below the threshold.This indicates consistent cache hits for the attacker. So, in line with the previous predictions, the results align with what was anticipated. So the attacker can conclude that the victim had an access to the shared table.

4.1.2.3 Example of Cache Timing Attack for Key Recovery in Encryption Operations on 1CPU

In the realm of cybersecurity, attackers constantly seek innovative ways to exploit vulnerabilities and extract sensitive information from system, so for this reason , we tried to write a code of a real case that uses the cache timing attack. A cache timing attack is a type of side-channel attack that takes advantage of variations in the time it takes to access data in the cache memory. The "Flush and Reload" technique is one of the methods used in cache timing attacks.

Experiment
The code used for this experiment demonstrates a scenario where two participants work together to securely exchange sensitive information. In this process, one participant sends encrypted data to the other, who has the decryption key. To understand the encryption method used and unveil the original data, a "flush and reload" approach is employed. This method takes advantage of differences in memory access times, using the cache to leak unintentional information. By accurately timing memory accesses, we can gain insights into the encryption method used and ultimately decipher the original data. This example highlights the vulnerability of encryption to cache timing attacks and emphasizes the need to ensure exclusive access to sensitive data during communication.

Here's a brief overview of how the code achieves this:

- User Input: User provides sensitive data for the encryption algorithm.

- Encryption Process: The encryption function involve basic multiplication and addition operations carried out on a clear message with a certain secret key. It's important to recognize that the operation applied to the plaintext is determined by the value of each bit within the secret key.

- Cache Management: The "flush_operation()" function clears the cache, ensuring that the attacker's memory accesses remain unaffected by cache hits.

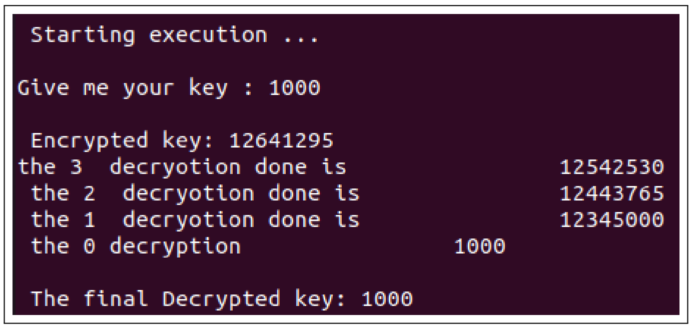
- Measurement and Analysis: The "second_participant_read" function measures access times for the multiplication and addition functions. By comparing these access times, it determines which operation the victim performed.

- first_participant Operations: The first_participant_access function simulates an encryption operations. It performs a sequence of encryption operations using the addition and multiplication functions.

- Main function: In the main function, the following steps are executed:
  1-Flushes the cache.
  2-Calls first_participant_access to simulate victim's encryption operations.
  3-Reads the detected operations using second_participant_read.
  4-Decrypts the key by reversing detected operations. Outputs the decrypted key.

Expectation
The key idea is that the attacker can detect which operations were performed by the victim (addition or multiplication) based on the measured cache access times.

Simulation
The results of this code can be seen in the documents below :



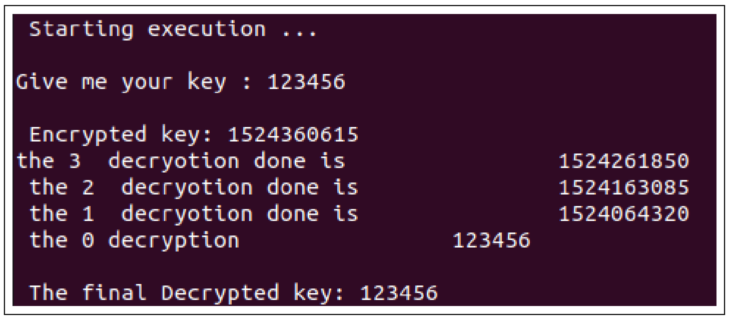
```
 Starting execution ...

Give me your key : 1000

 Encrypted key: 12641295
the 3  decryotion done is                   12542530
 the 2  decryotion done is                  12443765
 the 1  decryotion done is                  12345000
 the 0 decryption                   1000

 The final Decrypted key: 1000
```

Figure 18 – proof 1



```
 Starting execution ...

Give me your key : 123456

 Encrypted key: 1524360615
the 3  decryotion done is                   1524261850
 the 2  decryotion done is                  1524163085
 the 1  decryotion done is                  1524064320
 the 0 decryption                   123456

 The final Decrypted key: 123456
```

Figure 19 – proof 2

As evident from the figures above, the code enables us to deduce the message of another individual through the utilization of the flush and reload technique.

4.1.3 Conclusion

To sum up, our thorough analysis of the established system has allowed us to carry out the flush and reload attack. This in-depth understanding of how the system works has been crucial for successfully implementing and observing the flush and reload attack in action.

Additionally, it's worth mentioning that the Syscall Emulation mode "SE" enables us to perform a flush and reload attack using a single C code with just one CPU and application. However, it's important to note that this setup might not fully represent real-world scenarios since shared data are located in the same code.

Regarding limitations, it's worth pointing out that the text debug files generated during all simulations are quite large. Although there was an intention to visualize memory or cache mapping for different data, this was not feasible due to the files' size limitations .

4.2 Model 2 : Two Cores

Now to go further and closer to real cases, a second system is modeled with two cores. This time two attacks will stimulate this system to get conclusions and analyzes.

4.2.1 System Behavior

The system we implement in gem5 is presented in the figure 20 below.

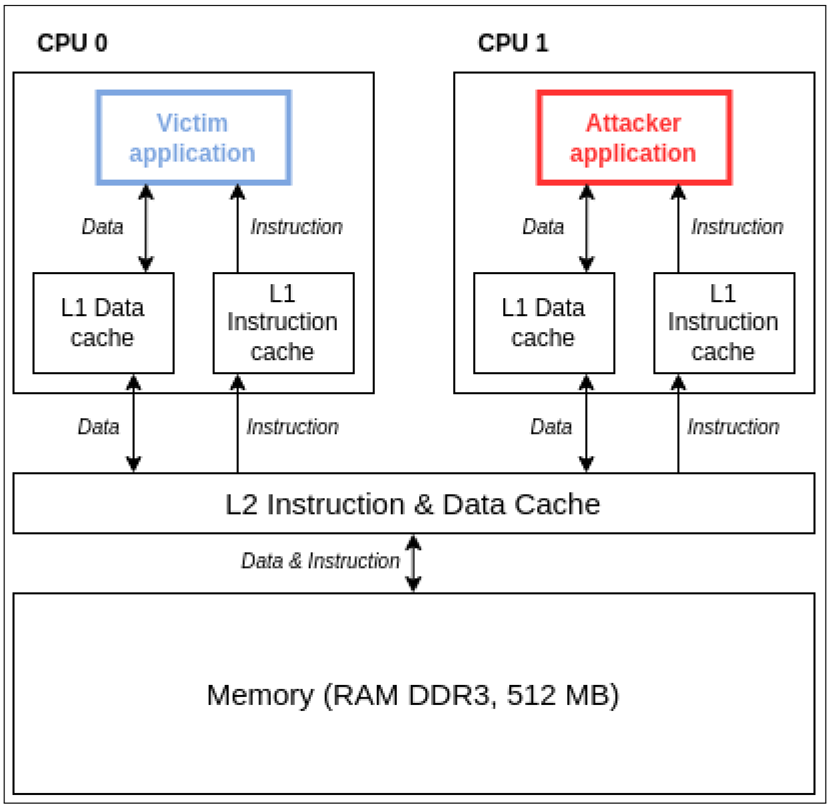


Figure 20 – gem5 simulated system with attacker and victim

Here as two cores are implemented, a distinction has to be done between private and shared caches. Each core will have instruction and data private caches, and they will share one l2 cache. As explained in section 3.1.2.2, this will be the backdoor for a spy to observe what victim's application is doing.

4.2.1.1 Model definition

This model will have one memory system in the center, with two external entities. The victim and the attacker are the only entities that will be taken into account. To simplify the system approach for cyber-security, inputs and outputs or external peripherals that affect memory system are not taken into account. The figure 21 shows system's overall.

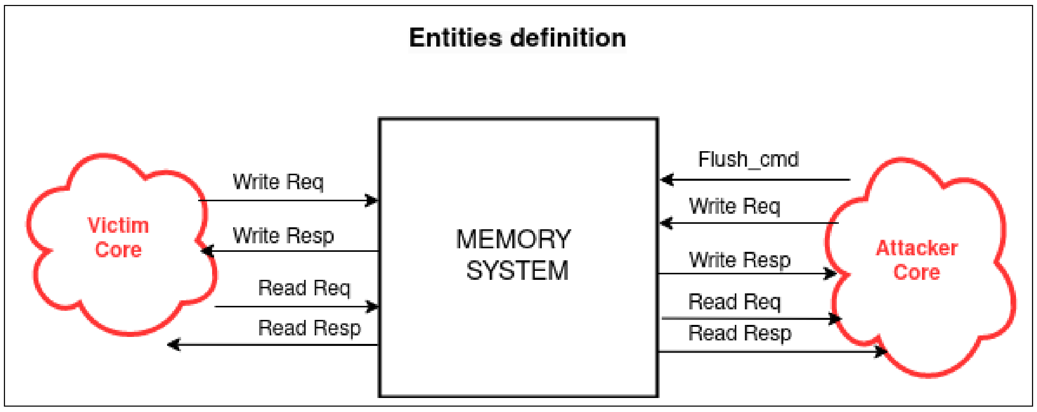


Figure 21 – Definition of entities

Attacker and victim just send read or write request and receive read or write response, with a possibility for attacker to use the flush. All of these interactions have now to be described.

4.2.1.2 Definition of input and outputs

The input and output definitions remain consistent with those depicted in Figure 10.
Similarly, the internal features remain unchanged from before in the figure 11. However, the only addition is the consideration of L2 associativity.

4.2.1.3 Behavioral description

Entities' behavior is done according to all different requests coming from and sent to attacker and victim cores. It is a general purpose behavior on figure 22 allowing to describe what will happen in the system no matter what the two external entities are doing. The capture is divided in four for a better visibility. The first one resumes the three requests that can be done to activate the system. After that each screenshot describes one behavior after one of these request. It is quite similar as One CPU system's behavior, but with new states associated with l2.

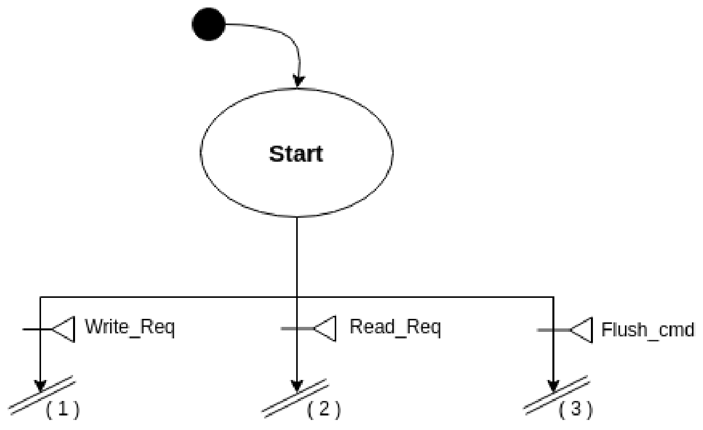


Figure 22 – First part of System behavior

Here, from a start state, system can only go in three other states depending on the incoming request of Read Write or Flush. First described one is Write (1) in the figure 23 below.

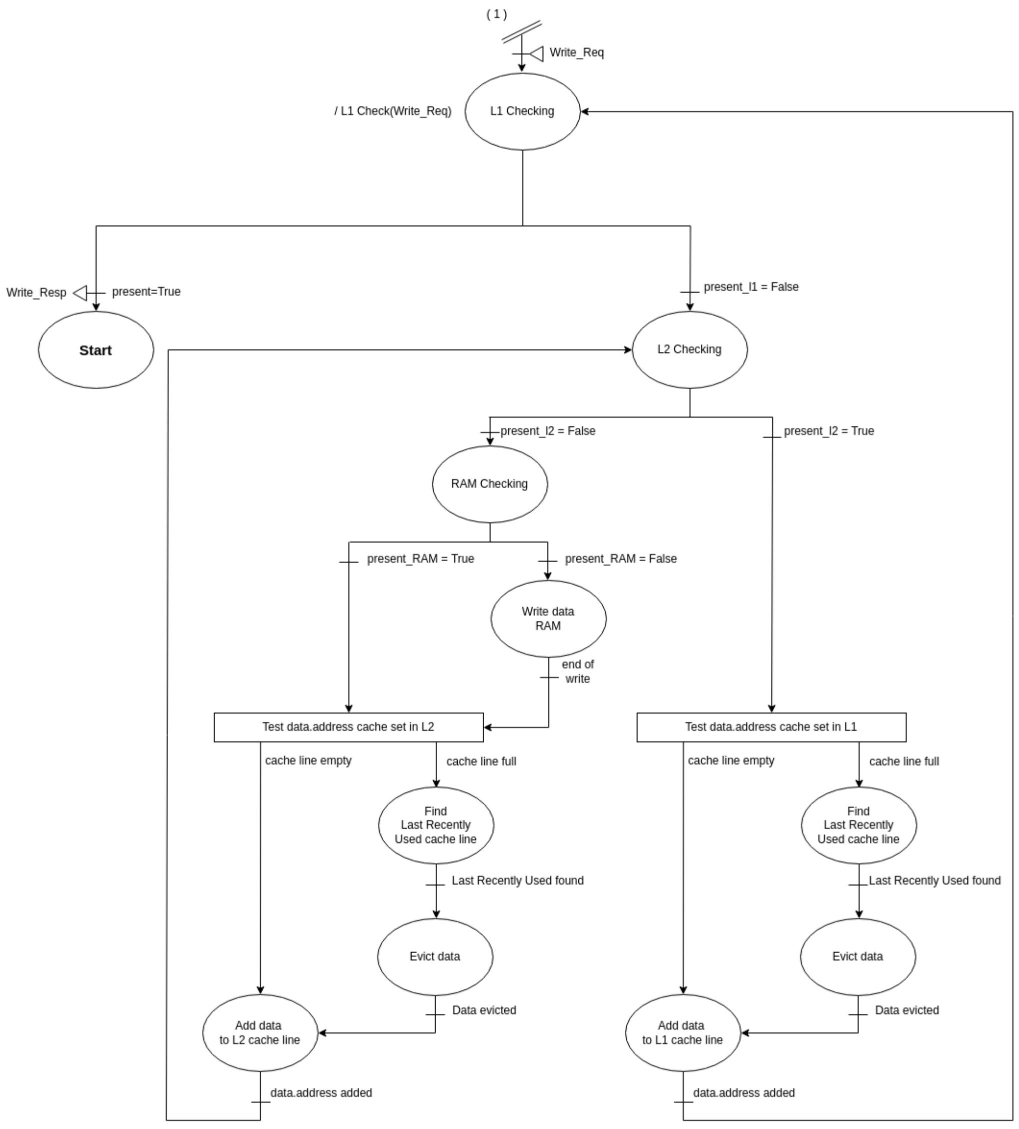


Figure 23 – System behavior for a Write Request

Here is a list of the different steps of this behavior:

1. After a write request, the system checks if the requested data's address is present in L1 cache.
2. If the address is found in L1 cache, the data is written to the correct location, and the system transitions to the Start state.
3. If the address is not present in L1 cache, the system checks L2 cache and repeats the operation.

4. In case the address is located in RAM but not in L2, the system brings the address back to its corresponding cache set (due to set associative mapping).
5. Within this cache set, the system checks if lines are empty or occupied.
6. If a line is empty, the address is written on that line.
7. If no lines are empty, the system identifies the Least Recently Used (LRU) data, clears it, sends it back to memory, and writes the new data in its place.

Given that both the manipulated data and a cache line are 64 bytes long, cache ways are not considered in this context. For a read operation, the behavior (illustrated in Figure 24) follows a similar pattern.

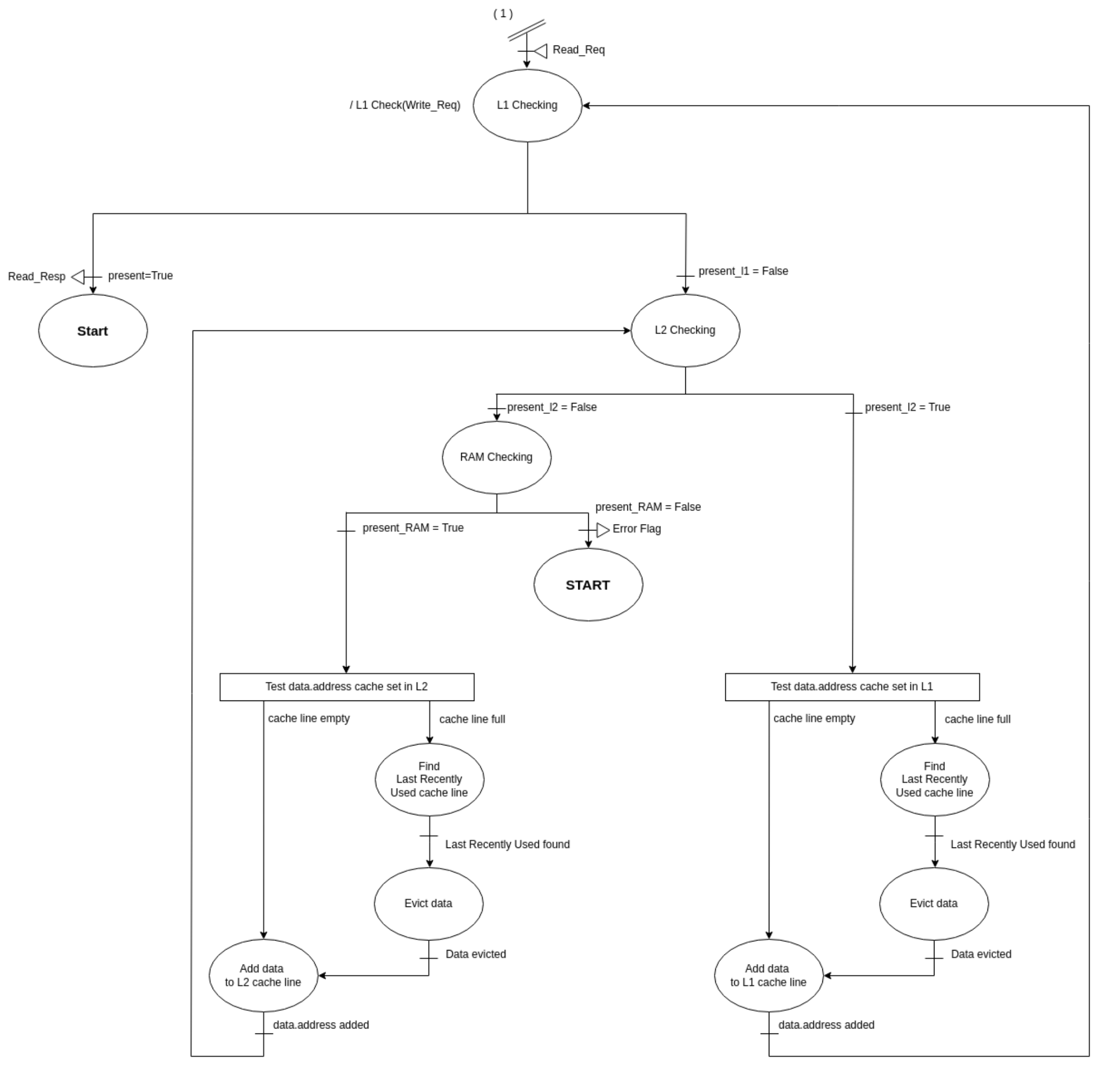


Figure 24 – System behavior for a Read Request

Here the only difference is that if asked data is not present it raises a flag, instead of writing it like previous operation. The last process to describe is the flush process in figure 25 below.

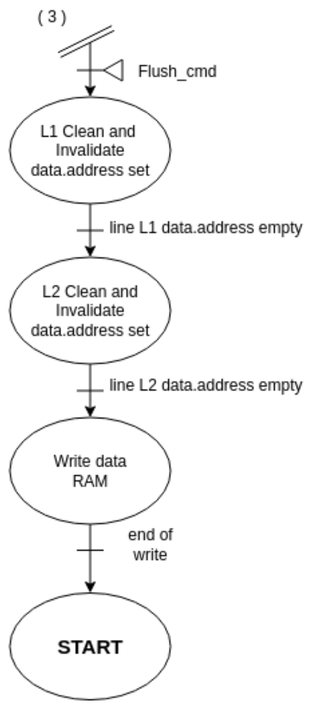

(3)
Flush_cmd
L1 Clean and Invalidate data.address set
line L1 data.address empty
L2 Clean and Invalidate data.address set
line L2 data.address empty
Write data RAM
end of write
START

Figure 25 – System behavior for a Flush Request

Here the behavior has been simplified a lot because flush operation as shown for example in its assembly instruction does different operations. Basically it checks the address in l1, evicts and invalidate it, and repeats the operation for all other cache levels. Finally it writes the actual value of the data in RAM.

Next step is now to apply cyber attacks to this system, to get simulation results and validate or not the model.

4.2.2 Flush and Reload Attack

4.2.2.1 Definition of entities behavior

In this section, we focus on a system with two CPUs : the attacker's CPU and the victim's CPU. To analyze their behavior, we have created two separated processes in C codes, which are provided in [3].The type of CPU used is ArmO3CPU . The behavior of the attack and the victim can be represented by the automata below.

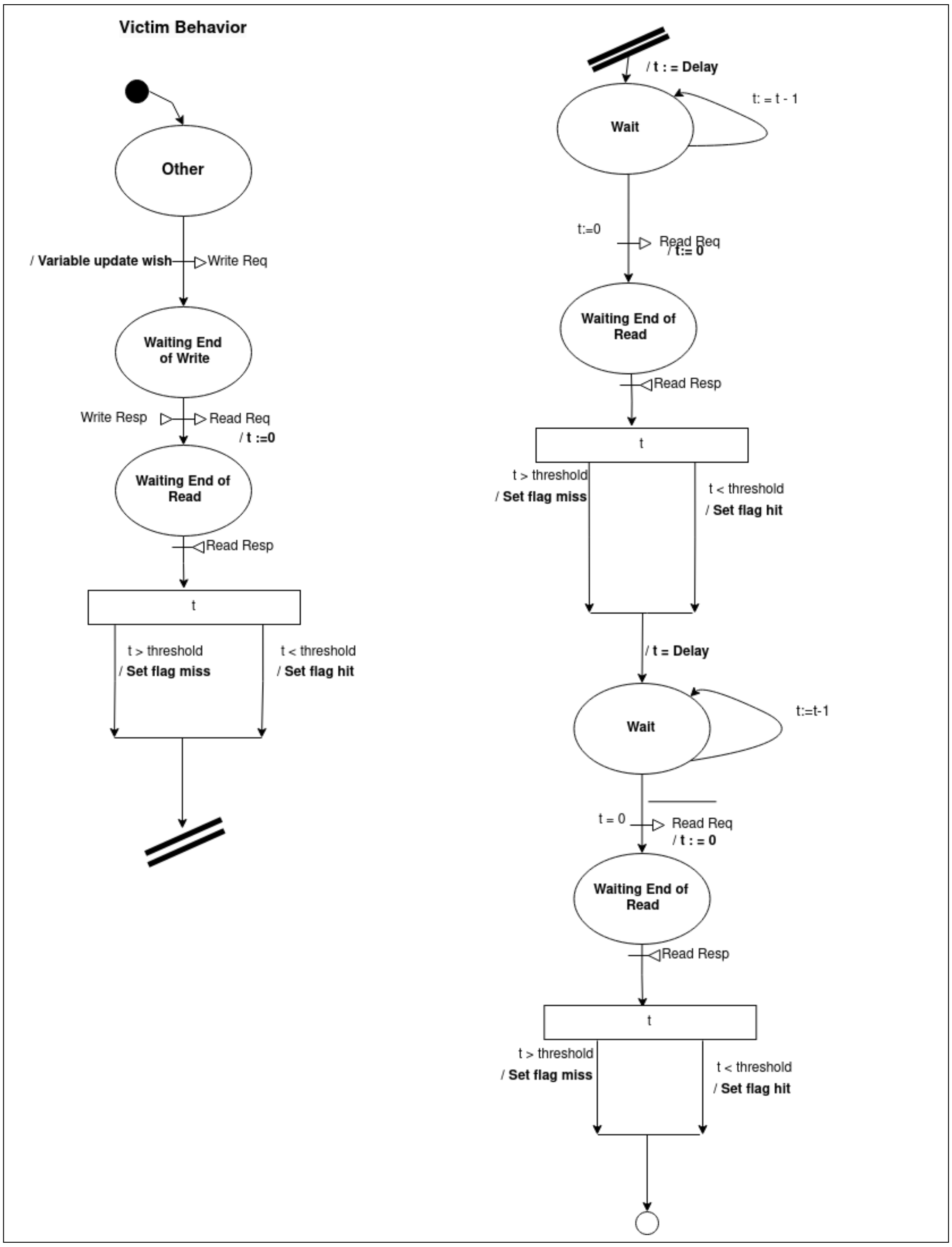


Figure 26 – Victim Behavior

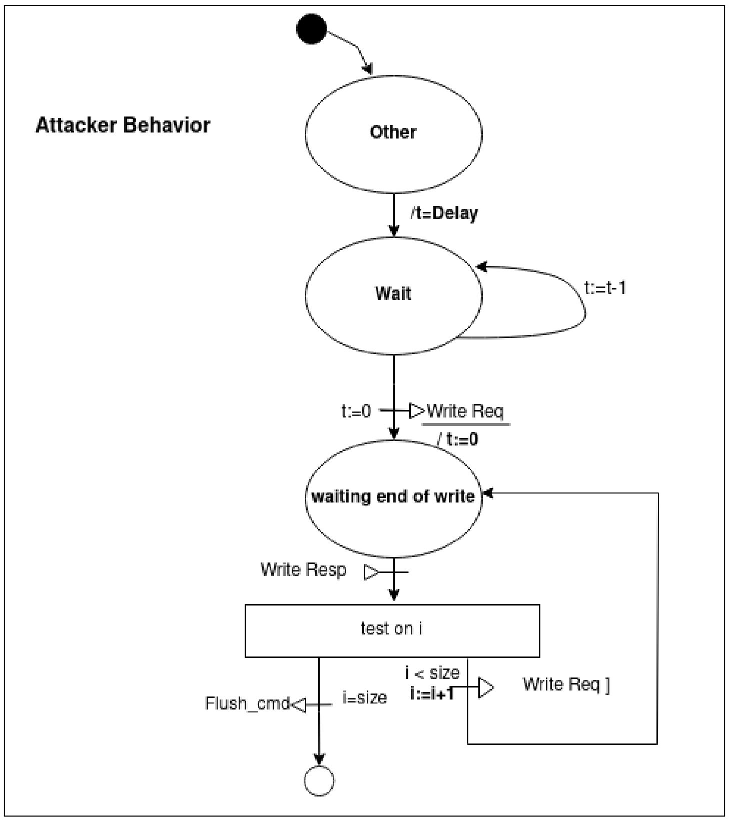


Figure 27 – Attacker Behavior

4.2.2.2 Experiment

We will conduct an investigation centered on calculating access times from the victim's perspective, with a specific focus on victim's cache side. While this experiment might not directly replicate a realistic attack scenario, its value lies in its potential to reveal any irregularities or anomalies occurring within victim's environment. The primary objective of this experiment is to enable the target to access specific data. Following this, the attacker will proceed to clear all cache memory on their end and subsequently attempt to retrieve particular data from target's side.

Before accessing any data, we ensure a complete flushing of the L2 cache memory. To expedite the flushing process and minimize the required time, we have configured the L2 cache size to be 16KB. Furthermore, both the L1 Instruction cache and the L1 Data cache sizes have been set to 8KB.

Expectation

The expected order of execution is :

1 - The victim initially tries to access a mem_block, and we measure the time required for this access.

2 - Next, the attacker flush the L1 and L2 cache memory to ensure that the victim's data is no longer present in the cache.

3 - To verify that the cache memory has been successfully flushed and the victim's element is not present, we allow the victim to access their data again after the flush. We expect the number of cycles required to be greater than in the initial access, indicating a cache miss.

4 - Furthermore, to reinforce our results, we let the victim access their data once more. This time, we expect a cache hit, which would result in a lower number of cycles.

This experiment aims to demonstrate the behavior of cache memory in relation to accessing specific data, the effects of flushing the cache, and the subsequent cache hits or misses.

Simulation
First of all, to ensure that these steps are performed in the correct order, a "for loop" construct is used in source code. To maintain control over execution time and measure the duration of the process, rdtsc() function is employed. This function retrieves the current number of clock ticks at a precise moment. By capturing the start time using start = rdtsc(), it is possible to track the elapsed time accurately. The figure below just explain how the code is executed with the number of tick at the beginning of each step .

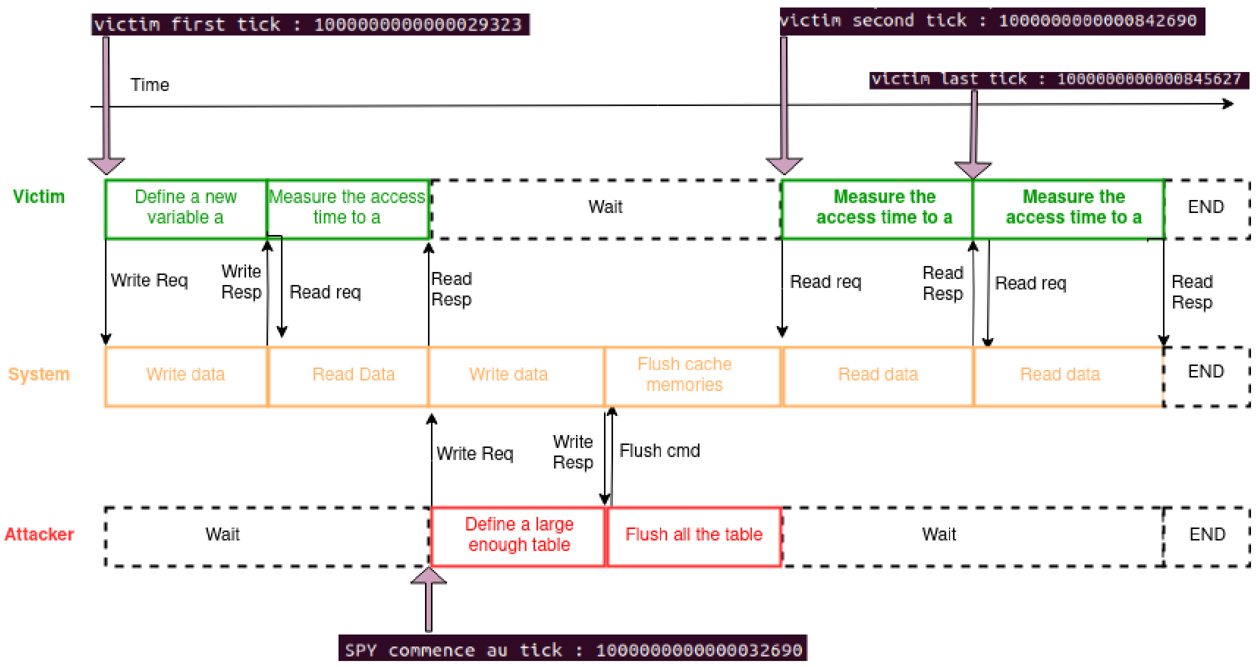
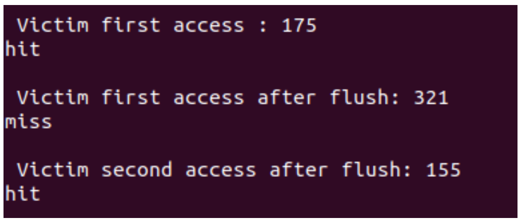

victim first tick : 1000000000000029323
victim second tick : 1000000000000842690
victim last tick : 1000000000000845627
Time
Victim
Define a new variable a
Measure the access time to a
Wait
Measure the access time to a
Measure the access time to a
END
Write Req
Write Resp
Read req
Read Resp
Read req
Read Resp
Read req
Read Resp
System
Write data
Read Data
Write data
Flush cache memories
Read data
Read data
END
Write Req
Write Resp
Flush cmd
Attacker
Wait
Define a large enough table
Flush all the table
Wait
END
SPY commence au tick : 1000000000000032690

Figure 28 – Correct order

Now, it is time to examine the actual results obtained from all the previous tests and experiments described earlier. These results will provide a comprehensive understanding of the behavior and performance of the cache memory system throughout the experiment.
The figure presented below illustrates the results obtained from system setup.



```
 Victim first access : 175
hit

 Victim first access after flush: 321
miss

 Victim second access after flush: 155
hit
```

Figure 29 – Results of the experiment

As anticipated, victim's access time after cache flush is greater than the access time before flush. This can be attributed to the flushing process conducted on the attacker's side, which removed desired data from the cache. Consequently, when the victim attempts to access the variable, it is not readily available in the cache, resulting in a cache miss and an increased access time as the data needs to be fetched from the main memory. Additionally, it is worth noting that the first cache hit observed during the experiment is typically larger than the subsequent cache hits. This discrepancy arises from an icache miss.

4.2.2.3 Conclusion

To sum up, the set up system at the beginning of this experiment helps understanding final results. We also showed how the flush and reload attack could be done using that system.

However, it's important to note that executing the flush and reload attack is not as straightforward as mere observation suggests. By looking from the attacker's side, trying this attack with two CPUs doesn't work well. This is because of the limits in the software that were talked about in paragraph 5 using the SE model. Since CPUs can't share memory, they can't use shared libraries or variables. So, the attacker can't know for sure if the victim accessed a certain variable. This variable is stored in different places in the stack of each process, so the attacker can't really watch what the victim is doing.

4.2.3 Prime and Probe attack

4.2.3.1 Definition of entities behavior

Another experiment is to use the fact that the attacker can write a large amount of data that can be enough to erase victim's data from the cache, as a Prime and Probe attack acts. To implement this attack using our methodology, new behaviors of victim and attacker are done. Victim's behavior is shown below 30. It is again represented as an automata.

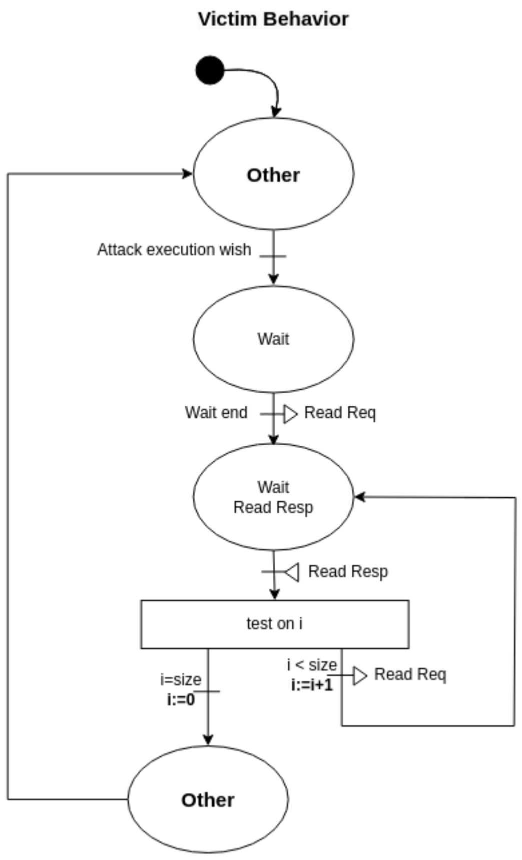


Figure 30 – Victim Behavior

Here, as the aim is to show how the system acts when a application evicts attacker's data in cache hierarchy, victim needs to manipulate a big amount of data here implemented as a table. Victim's first wait is set up to ensure that attacker has time to fills the cache. Then it Sends a lot of read request for a lot of data in a table. Number of requests is determined by variable size. Once all accesses are done, it goes back to a state named "Other", meaning that it can do other things which are not taken into account here.

After that, attacker's behavior is described in figure 31 below with an automata.

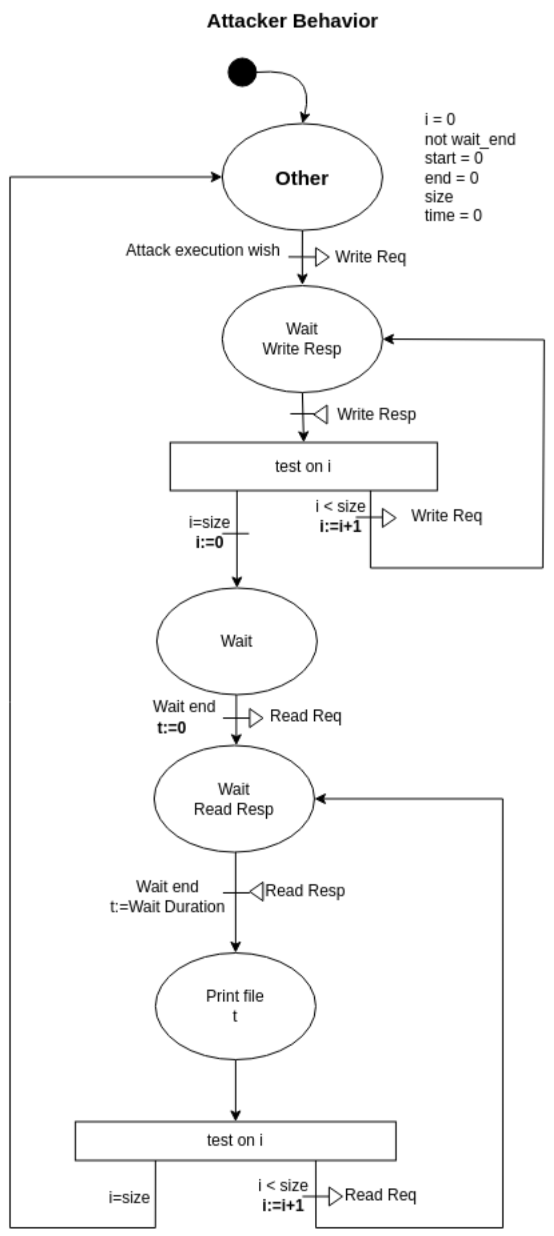


Figure 31 – Attacker Behavior

This automata can be cut in three part. First two states are for filling cache with data (Prime phase). Then a wait state allows victim to manipulate its tab. Then last states are for reading again its table and measure reading time, to deduce information by printing these times in a file (Probe phase).

These two automatas are implemented as C programs executed by python system in gem5. They are available in the Git repository in : Model2_TwoCores/ . However, these C implementations would not work in a real modern processors because some aspects have been avoided to simplify. A more real implementation has been done in Miro Haller's paper [5] paper about revisiting Microarchitectural Side-Channels attacks.

4.2.3.2 Experiment

Here, the chosen approach to assess the relevance of this model with gem5 is as follows: evaluating using printed file on attacker's side how much manipulated data are detected by using a detecting rate. By varying l1 and l2 size, conclude on how these variable may influence system's vulnerability regarding prime and probe attack. Finally by comparing simulation and expectation, also conclude on the reliability of the system and its level of complexity. For experiments, as attacker's simulation gives reading times, there are different ways of exploiting them. Chosen one are : plotting in a graph reading time for each index of the table, plotting a histogram of the amount of each reading time, using this, trying to map every data in our system hierarchy (RAM, l2, then l1).

Expectations

As everything is known for entities and system, a mapping of every attacker's data can be done and is the expected result of simulation. Cache sizes need first to be fixed and are : 64 kB for l1 Dcaches, 16 kB for l1 ICaches and 128 kB for l2 shared cache. As manipulated data are 64 bytes long, a quick calculation gives that 1024 data can be manipulated in l1 dcache, and 2048 in l2 cache. Once these sizes are fixed, it is possible to imagine how attackers fills cache hierarchy with its 2500 indexes table. This size has been chosen to fill both caches enough. This filling is shown in figure 32. Values on the left have been observed after a first simulation, to have an idea of how much data are stored by default in caches, and conclude on how much data from a table can use the attacker. In fact by not manipulating any data, caches are filled with some information by default like registers addresses or libraries for example.

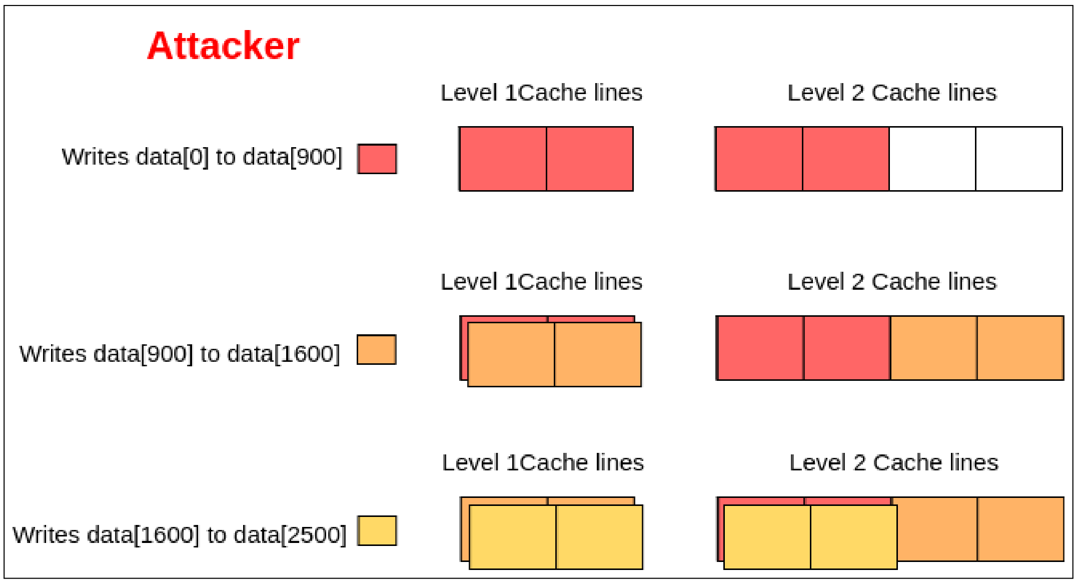


Figure 32 – Expectation on how is filled the system on first attacker's writing

Replacement policy is here set as Last Recently Used, meaning that it is first read data that will be first evicted. This is the order or all steps :

1- About 900 data are written in l1 cache and l2 as almost anything has been written there.

2- For about 700 following ones, as l1 is full, system will evict cache lines to put new data. As l2 is bigger, no data are evicted and everything is stored there.

3- For last ones, l1 needs to evict lines, and now l2 also, because it is full.

At the end, first written data are stored in RAM, last ones are only in l2, and intermediate ones are in l1 cache. Described behavior takes here into account that gem5 classic caches only implement a mostly inclusive policy. This is why orange data, even if they are not in l1, are still located in l2.

In fact as to deduce information attacker will measure reading time for all indexes before and after victim, only ones with a new location will provide information. Data from 1600 to 2500 will always be in l1 cache. Data from 0 to 900 will always be in RAM. Only data between 900 and 1600 will first be in l2, but then evicted. So there is a maximum of about 700-800 data to detect for this implementation of a prime and probe. As implementing an attack can take months and even years, this amount of detected data is supposed enough to conclude on our two objectives : evaluate influence of cache size on vulnerability, and evaluate the relevance of gem5 in terms of hardware cyber-security.

According to previous expected cache filling by attacker, the second graph to imagine is the representation of reading time for all indexes of the table, before and after victim's actions. Normally, only times between 900 and 1600 should move, and be longer because of their new location in RAM. Two graphs of expectation are plot in the figure 33 below.

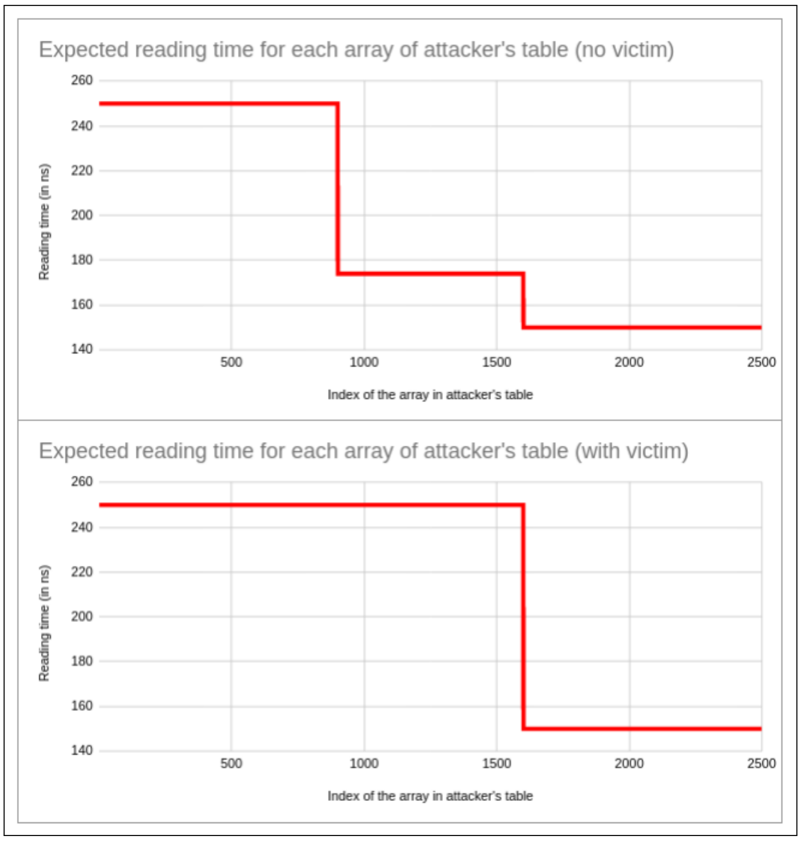


Figure 33 – Graphs of expected attacker's reading times with and without victim

By comparing these two measures, attacker should be able to deduce that victim evicted some of its data and to know their index in the table. Next step is not implemented but it is then usual to know for which data on which cache set they are mapped, and then try to guess more information about the victim. Here in this graph it is possible to link with the mapping. Data between 0 and 900 take a long time to read, showing their location in RAM, next ones are read in l2 giving a reading time around 170, and then last ones are still in l1 cache, giving the shortest reading time. All of these times have been measured before and depends on the simulated type of CPU.

Simulation

This system is then implemented and simulated. The two previous graphs are then plot to compare expectation and simulation. These two graphs are shown in figure 34 below.

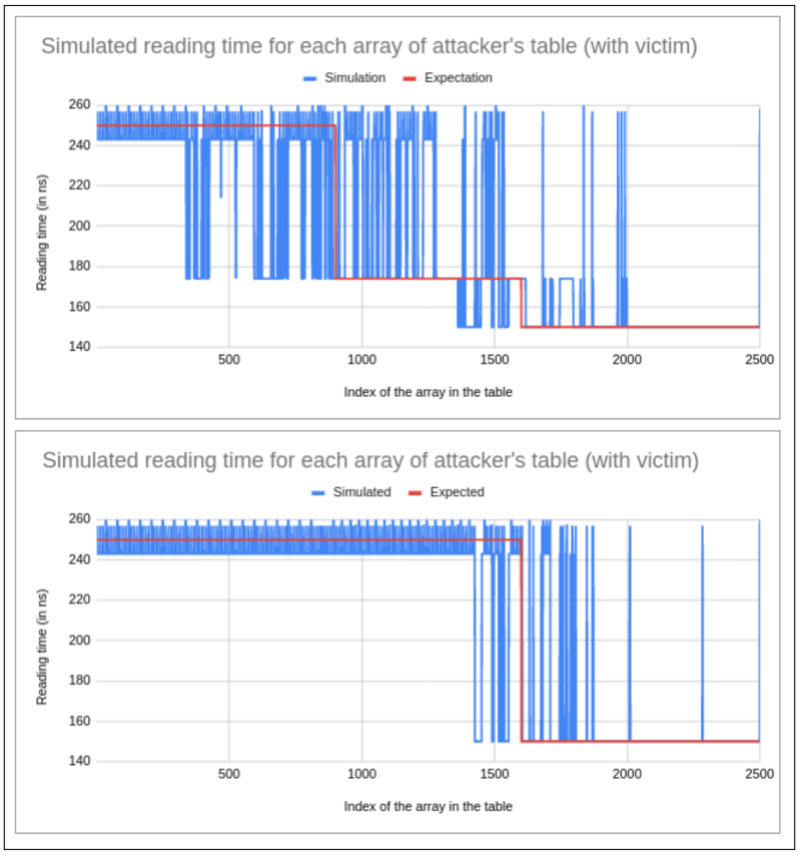


Figure 34 – Graphs of simulated attacker's reading times with and without victim

Here simulation is plot in blue and expectation in red. First a trend can be observed on the two curves that at some point simulation sticks with expectation. For the case where victim reads a table, simulation is closer to expectation with evicted data that have been moved to RAM. However, it seems that mapping is not as expected at all, leading to data that are not stored at their expected location. This lack of accuracy in the mapping can be due to different missing points in the model, but as it is still possible to detect a variable amount of victim's data, it is still possible to conclude on global trends. The amount of moved data is here counted and this is repeated for different cache size.

Other simulation with variable cache sizes

L2 size (in kB)	L1 Size (in kB)	Rate of detected eviction by attacker in l2
32	8	21%
32	16	20%
32	32	22%
32	64	20%
32	128	14%
64	8	34%
64	16	30%
64	32	27%
64	64	22%
64	128	14%
128	8	64%
128	16	60%
128	32	54%
128	64	32%
128	128	16%

Table 2 – Table of Detection rate for variable l1 and l2

To conclude on if gem5 and our model are trustworthy enough to observe behavior in hardware cyber-security, the rate of detected victim's data is measured for different cache size. All simulation results are summarized in the table 2.

Here, it is possible to conclude on the importance of cache size. Globally for a victim manipulating 2500 data, the eviction rate is higher when the l2 shared cache is big. On the other hand the more l1 is small the more attacker deduces information. This is quite logical and can be linked to expectation. As data that attacker can use to deduce things are in only l2 cache, the more l2 is big the more he will have data there. And the more l1 is small, the less data he will store and read back there. His objective is to have as less data in l1 as possible, and as much in l2 as possible. In a real case with a fully inclusive cache (or using gem5 Ruby caches), attacker should have a higher detecting rate.

4.2.3.3 Conclusion

To conclude for this experiments, this model used in gem5 allows to deduce global trends. By not taking into account indexes of evicted data or rates amount, but only the global trend, we conclude that systems with big l2 and very small l1 are more vulnerable.

However, a real prime and probe attack would then need to know the precise mapping of all data, to know how cache is mapped in term of sets, lines and ways (in a set-associative mapping). Here, things are missing in our model, which does not take fully into account this mapping of addresses in sets, this can explain differences in simulation. Figure 35 tries to compare expectation and simulation. In simulation, it is possible to observe the real mapping and the importance of taking it into account. Data are not mapped in block, but are split in all cache memory, making it harder for an attacker to deduce where his evicted information are located.

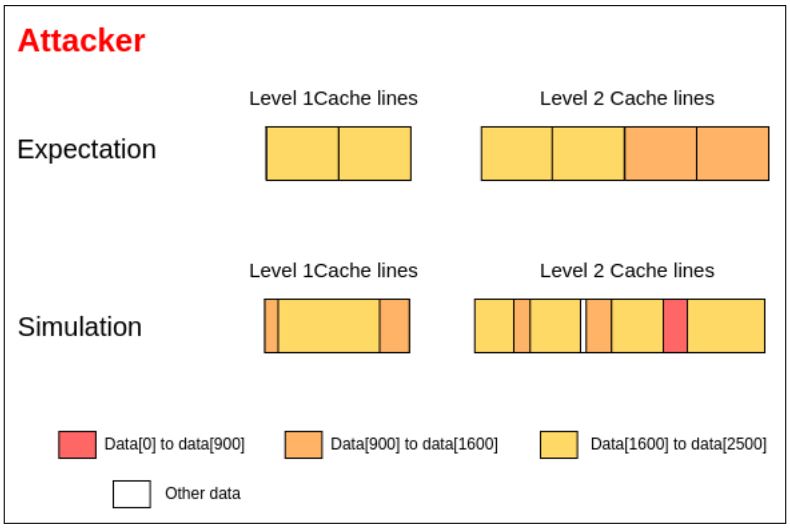


Figure 35 – Comparison between expectation and simulation on mapping

One last thing is that gem5 simple cache models don't give a fully inclusive indexing policy but a mostly inclusive one. This "mostly" adds unknown variation in measurement. In Syscall Emulation mode and without Ruby caches, we are not exactly sure on how our cores transform addresses into set number and lines number. We don't even know if classic caches use virtual or physical mapping.
A solution could be to use Ruby caches, and Full System mode. First because Ruby are fully configurable and can be fully inclusive, with a mapping close to reality. Secondly because Full system would implement a Kernel, adding more details in memory and cache mapping policies.

4.3 Limitations of Syscall Emulation Mode

When delving into the exploration of gem5, it's essential to recognize both the strengths and weaknesses of its operational modes in order to develop a thorough grasp of its capabilities. While gem5 serves as a potent and adaptable instrument for architectural research and performance analysis, it's important to acknowledge that its Syscall emulation (SE) mode, like any software, is not exempt from limitations. These limitations, though not diminishing its value, provide contextual understanding of the operational boundaries inherent to gem5. In the following section, we will delve into some of the notable limitations that researchers and simulation practitioners should consider when employing the SE mode for various research and simulation objectives.

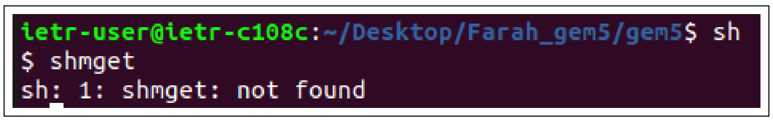
4.3.1 List of Limitations

1- Shared Memories : The limitation related to shared memory configuration and inter-processor communication is particularly evident when utilizing gem5's Syscall Emulation (SE) mode. In this mode, gem5 ensures itself to strike a balance between simulation accuracy and performance by emulating system calls, but it doesn't fully emulate the operating system. Consequently, while SE mode offers a lightweight approach to simulation, it presents challenges in explicitly defining shared memory regions between multiple CPUs. The absence of comprehensive operating system support can prevent the accurate replication of scenarios that heavily rely on shared memory synchronization mechanisms. Therefore, researchers and users opting for the SE mode in gem5 should recognize this limitation, especially when investigating complex multiprocessing scenarios where direct shared memory interactions between CPUs are a main aspect of the analysis.

Additionally, it's important to note that certain functions like "shmget" and its derivatives, which are utilized for creating shared memory, are not implemented in gem5's Syscall Emulation (SE) mode.

Demonstration

To illustrate the absence of shared memory capability in the SE mode, we conducted a straightforward test within the terminal. The objective was to attempt the creation of shared memory using the shmget command.



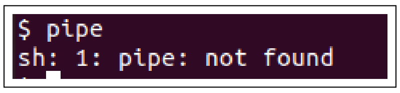
```
ietr-user@ietr-c108c:~/Desktop/Farah_gem5/gem5$ sh
$ shmget
sh: 1: shmget: not found
```

Figure 36 – shmget not found

2- Pipe Communication : In the realm of computer architecture, a pipe serves as an inter-process communication mechanism that allows two processes to exchange data. They enable efficient communication between processes while ensuring data integrity and synchronization. One process can generate data and pass it to another process through the pipe, even if the two processes are not directly connected.

However, in Gem5's SE mode, while the implementation of pipes is not available, researchers and developers can still explore their functionalities using other modes of Gem5. Pipes, being a fundamental mechanism for data exchange between processes, require intricate synchronization and management that might not align with SE mode's goals of performance optimization.

Demonstration



```
$ pipe
sh: 1: pipe: not found
```

Figure 37 – pipe not found

3- Debug flags : A significant drawback of SE Mode is its lack of organized debug flags and a graphical interface to visualize cache memory activities, especially in complex system setups. Few options are available and have been studied by some researchers like Jason Lowepower [6] but are not conducive to precise conclusions. This absence makes it challenging to easily understand what's happening within cache memories and the MMU (Memory Management Unit). Without user-friendly visual tools, interpreting the interactions and performance implications of different components becomes a complex and time-consuming task, requiring manual code analysis for deeper insights.
Moreover, we've tried to implement this graphical interface by converting debug flags using python scripts but the executable was not working well due to large files size.

5 Conclusion

To conclude, addressing the details of modeling a cyber security system using the Syscall Emulation mode of gem5 necessitates a nuanced consideration of the required level of detail. The setup we used made it possible to study how the whole system behaves. This helped us understand things like how cache size affects the system and how likely it is for an attack to work on the setup. With this objective in mind, the utilization of Ruby caches combined with a kernel operating in Full System mode becomes mandatory. Finally, considering potential updates that could have proved beneficial in our context, the incorporation of additional graphical interfaces could have served as valuable aids in comprehending the simulated devices, as previously elucidated. This could be a very interesting development project for any further work.

References

[1] gem5 community. *C/C++ Coding Style*. eng-ENG. URL: https://www.gem5.org/documentation/general_docs/development/coding_style/ (visited on 06/09/2023).

[2] The gem5 dev community and contributors. *The official repository for the gem5 computer-system architecture simulator*. eng-ENG. URL: https://github.com/gem5/gem5.

[3] Farah KHAZAAL Ulysse COUTURIER. *internshipproject*. eng-ENG. URL: https://github.com/FarahKhazaal/gem5_Multiprocessor_security.git (visited on 06/10/2020).

[4] *gem5: The gem5 simulator*. eng-ENG. URL: https://www.gem5.org/.

[5] Miro Haller. *Revisiting Microarchitectural Side-Channels*. eng-ENG. URL: https://github.com/Miro-H/CacheSC/blob/master/docs/revisiting-microarchitectural-side-channels-Miro-Haller.pdf (visited on 06/10/2020).

[6] Jason Lowe-Power. *Visualizing Spectre with gem5*. eng-ENG. URL: http://www.lowepower.com/jason/visualizing-spectre-with-gem5.html (visited on 06/01/2018).

[7] Clémentine Maurice Pierre Ayoub. *Reproducing Specter Attack with gem5 : How To Do It Right ?* eng-ENG. URL: https://inria.hal.science/hal-03215326/document (visited on 05/03/2021).

[8] *Writeback method*. eng-ENG. URL: https://www.researchgate.net/figure/A-A-Write-Through-cache-with-No-Write-Allocation-B-A-Write-Back-cache-with-Write_fig4_318860805.

[9] Yuval Yarom and Katrina Falkner. *FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack*. eng-ENG. URL: https://eprint.iacr.org/2013/448.pdf.