Reproducing Spectre Attack with gem5

How To Do It Right?

Pierre Ayoub
pierre.ayoub@eurecom.fr
EURECOM
Sophia Antipolis, France

Clémentine Maurice
clementine.maurice@inria.fr
Univ Lille, CNRS, Inria
Lille, France

ABSTRACT

As processors become more and more complex due to performance optimization and energy saving, new attack surfaces raise. We know that the micro-architecture of a processor leaks some information into the architectural domain. Moreover, some mechanisms like speculative execution can be exploited to execute malicious instructions. As a consequence, it allows a process to spy another process or to steal data. These attacks are consequences of fundamental design issues, thus they are complicated to fix and reproduce. Simulation would be of a great help in term of scientific research for micro-architectural security, but it also leads to new challenges. We try to address the first challenges to demonstrate that simulation could be useful in research and an interesting technique to develop in the future.

CCS CONCEPTS

• **Security and privacy → Hardware attacks and countermeasures**; **Hardware security implementation**.

KEYWORDS

Reproducibility, Micro-architecture, Branch Predictor, Cache Side-Channel, Flush+Reload, Transient Execution Attack, Spectre, Simulation, gem5

ACM Reference Format:
Pierre Ayoub and Clémentine Maurice. 2021. Reproducing Spectre Attack with gem5: How To Do It Right?. In *14th European Workshop on Systems Security (EuroSec '21), April 26, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3447852.3458715

1 INTRODUCTION

Modern processors use various micro-architectural mechanisms to increase their performance. Attacks that exploit these mechanisms are called micro-architectural attacks. Caches, which are used to store recently accessed data for quick accesses, have been extensively studied for their timing leakage like in the Flush+Reload attack [18]. By differentiating cached and uncached data through timing measurements, an attacker could spy another process. Speculative execution is used to speculate on the next instruction to execute when the instruction flow is uncertain, particularly on branch instruction for branch prediction [13] [15]. In 2018, a new class of attacks has been released, namely Spectre [7], which is able to steal protected data by exploiting the speculative execution mechanism of a processor.

Since its disclosure, many variants of the Spectre attack have been discovered and various countermeasures have been developed, both software or hardware-based [3]. Unfortunately, currently none of them have been widely adopted, mainly due to their performance overhead or the implementation difficulty.

In this paper, we investigate a novel approach in the field of micro-architectural attack development, which is to use the state-of-the-art simulator gem5 to reproduce and study the Spectre attack. These attacks are closely tied to the hardware implementation, thus, their reproducibility on a simulator is not trivial. We are also interested in the accuracy of the simulation. Our goal is to explore the benefits of the simulation for security researchers. The main contributions of our paper are:

- From our experiments, we provide several guidelines that we deem important for micro-architectural security research, in both attack development and reproduction (Section 4).
- We demonstrate the usage of gem5 to develop new techniques for helping attack development and understanding, like visualizing the micro-architectural state of a processor under attack (Section 5).
- We demonstrate the reproducibility of Spectre on a cycle-accurate simulator and evaluate its accuracy by comparing it to experiments on real hardware (Section 7).
- We discuss the requirements of gem5 to simulate those attacks and the parameters that affect their efficiency (Section 8).

2 BACKGROUND

2.1 Caches

The latency when loading a word from the main memory is too high compared to the computation speed of the processor, thus the memory is organized into a hierarchy, with smaller but faster memories close to the processor. A cache memory is a small, fast, and expensive memory which aims at keeping more often used data close and quickly accessible. A *cache hit* happens when the processor finds the requested data in the cache, otherwise it is called a *cache miss*. The hierarchy is divided into several levels, often between 2 and 4 – denoted *L1* to *L4*. The *Last-Level-Cache (LLC)* is the closest level to the main memory. Caches are organized into multiple *lines*, which are the smallest unit to load or flush data from the cache, often equal to 64 consecutive bytes. A fixed number of lines are grouped into cache *sets*, which corresponds to the cache

EuroSec '21, April 26, 2021, Online, United Kingdom

ACM ISBN 978-1-4503-8337-0/21/04…$15.00
https://doi.org/10.1145/3447852.3458715

associativity. A cache can be accessible from only one core, called a *private* cache, or from multiple cores of the processor, called a *shared* cache. In multi-core processors, the LLC is often shared between cores. Caches can be *inclusive*, which means that a level n of cache contains all the data and instructions of level $m < n$, in addition to its own exclusive content since it is larger.

2.2 Branch Prediction

When the processor does not know which instructions it needs to execute next, it may use an optimization called *speculative execution*, which consists in *speculatively* executing one of the possible instruction streams. If the speculation was wrong, then the pipeline is *flushed*, and we say that these instructions have been *squashed*.

Branch prediction is one form of speculative execution, where the processor speculates on the result of encountered branches [13] [15]. The branch predictor, which is an internal component of the processor, guesses the outcome of a branch instruction, *e.g.* if the *outcome* of a *conditional* branch is *taken* or *not taken*, even before the evaluation of the condition. Different branch types exist (unconditional and indirect branches) as well as different prediction types (instruction prediction, outcome prediction and target prediction).

The branch predictor uses some structures, the important one in this paper being the *Pattern History Table (PHT)*. First, a *history pattern* is a sequence of n bits indicating, for each bit, the taken or not taken direction for the last n executed branches in fetch order [19]. A *saturating counter* is a bit sequence – between 2 and 4 bits – incremented (resp. decremented) when a branch is taken (resp. not taken). The counter's uppermost bit is used as the prediction – set to 1 means that the branch was previously taken at least $\frac{1}{2}$ time. The PHT is an array indexed by a bit sequence identifying a scenario – a history pattern –, where each entry stores a saturating counter used as the prediction for the outcome of the branch.

There exist many algorithms and architectures for branch predictors. We briefly present the two architectures used in the remainder of this paper. The *two-level global history-based* predictor performs two steps for each branch prediction: (1) Indexing the PHT according to the current history pattern, which gives the outcome of the branch. (2) Predicting the address based on the predicted outcome. "Global history-based" means that the PHT is indexed by a unique history pattern stored in a shift register, which includes all encountered branches, being the simplest form of two-level predictors. The *Bi-Mode* branch predictor [9] is identical to the previous one except that it separates *mostly-taken* and *mostly-not-taken* branches into two different PHTs to increase prediction's efficiency.

2.3 Spectre

A *transient instruction* refers to an instruction partially or completely executed by the processor. It can affect the processor's micro-architectural state, but leaves its architectural state without any trace of its execution. The Spectre variant we studied (Spectre-v1, *a.k.a.* Spectre-PHT-SA-IP) [7] exploits speculative execution, more precisely, it tricks the branch predictor on a conditional branch's outcome. It uses the PHT of the branch predictor to induce a victim executing a conditional branch with a malicious and unauthorized parameter. The unauthorized instructions modify the micro-architectural state in a way that is dependent of the targeted value – the secret to recover. After the transient execution, the attacker probes the micro-architectural domain using a covert channel and infers the targeted value from it. The covert channel can exploit any primitive, however, Spectre is frequently exploited with cache covert channels for their efficiency, particularly Flush+Reload [18].

```
if (x < array1_size)
    y = array2[array1[x]];
```

Listing 1: Conditional branch targeted by Spectre.

Consider the branch in Listing 1, where the attacker is able to make the victim execute the snippet, control the `x` parameter and have access to `array2`. Several steps are needed to exploit the PHT: (1) **Setup**. Setup the covert channel by flushing all elements of `array2` from the cache for Flush+Reload. (2) **Training**. Execute the branch a lot of time with a valid `x`, thus increasing the PHT's saturating counters at their maximum for this branch. (3) **Encoding**. Send a malicious `x`, such as `x = &secret - array1`, so the processor will execute transiently the `array2[array1[x]]` memory access. (4) **Recovering**. Recover the value from the covert channel. For Flush+Reload, load every value of `array2` and measure the induced latency with a timer. When a cache hit is detected for index `i`, `i = secret` because it has been cached during the transient execution.

2.4 gem5

gem5 is a state-of-the-art cycle-accurate computer simulator [1], meaning that it simulates hardware components on a cycle-by-cycle basis, instead of simulating the instruction-set of an architecture. It is used both in the academic and industry world for research and development of new hardware technologies, and ARM is a well-supported architecture. It is made of two parts: (1) The C++ core where the logic of the components is programmed; (2) The Python interface linked to the C++ objects in order to interconnect and easily configure these elements. Each object has "ports" to send or receive packets from other object.

To the best of our knowledge, there is no systematic review that compares simulators applied to security research. However, gem5 meets our requirements to simulate our attack – cycle-accurate, modular, ARM architecture, caches, out-of-order execution, branch prediction – and is widely used.

3 RELATED WORK

To the best of our knowledge, there is little to no existing literature that highlights the best practices in micro-architectural security research in terms of reproducibility, neither in native environments or with simulators like gem5. The closest to our simulation work is a Low-Power's blog post [12] that reproduces the Spectre attack on gem5. It demonstrates that Spectre works on the simulator, with a quick discussion on branch predictors and compilers effects. In this paper, we try to go deeper by proposing new research techniques and evaluating the accuracy of the simulation. While the attack side is not well-studied on gem5, the countermeasure side already benefits from simulators, as many papers use gem5 to develop and validate their countermeasures [11]. To choose our gem5 processor in our simulation, we follow Endo et al. [4] who simulated an ARM

Cortex-A8 and an ARM Cortex-A9 processors with gem5 and an out-of-order processor, as well as Gutierrez et al. [6] who simulated an ARM Cortex-A15 processor with gem5 and the same processor in a survey of errors categorization using gem5.

4 GUIDELINES ON MICRO-ARCHITECTURAL ATTACK DEVELOPMENT AND REPRODUCTION

During our study, we were confronted to some proof-of-concept implementations with reproducibility problems, that might have been avoided with more "best practices" in security research. We encourage any security researcher to follow our generic guidelines, both for helping other researchers to reproduce their work and to reproduce the work of others.

4.1 Development

These general guidelines apply during the development of an attack, to improve reproducibility in the end. At the end, we also provide two additional guidelines specific to the development of timing covert channels and transient execution attacks.

Compiler version. Two versions of the same compiler can produce very different machine code from the same source code. Generally, the reason lies into all the optimization mechanisms that are implemented. As a good research practice, we suggest to always specify the compiler version, and to use the same version as the author(s) of the original work if a problem arises.

Compiler optimizations. Even more than compiler versions, compilation flags could completely change the outcome of an attack, both increasing or decreasing efficiency. After having experimented with attacks working with a flag and not with another, working on a system and not another because of an optimization, we consider compilers optimizations harmful for security research and reproducibility. We suggest to always disable most of the optimizations mechanisms of the compiler in a research perspective, except when the goal is to maximize the throughput on a specific implementation for a specific architecture. If a researcher observes that a particular optimization of *GCC* is required to make an attack work, we strongly suggest using the `-fopt-info` compilation flag to understand and isolate this optimization.

Manual optimizations. Manual optimizations could be important for some attacks. For instance, for a Spectre attack which targets the *Return Stack Buffer* (RSB) of the branch predictor – a structure used to hold and predict the address for a return instruction based on the jump for the function call –, inlining some functions allows to unload the RSB and hence the branch predictor, leading to a more efficient manipulation of the RSB. In those cases, we prefer to manually optimize the code instead of having the compiler doing it. Needed manual optimizations have to be found *via* experimentation.

Prefetcher. The prefetcher is a processor's component used to load memory elements into the cache even before they are used by a program. Prefetching un-accessed elements in the cache can skew the result of the cache covert channel. Disabling the prefetcher is generally difficult as it requires special privileges during boot time. Instead, the best method is to use a mixed-up access pattern to prevent the prefetcher to detect the next memory element that will be accessed, *e.g.*, for `i` from 0 to 255, index a table with the index `i_mix = ((i * 167) + 13) & 255` [7].

Re-ordering. An instruction, if no precaution is taken, can be delayed or moved into the instruction flow both by the compiler and the processor – statically at compilation time or dynamically at runtime, respectively. Therefore, it is important to ensure that instructions are executed at the right moment, especially for `flush` instructions that are critical for the covert channel. In order to verify this at runtime, gem5 can be a great help (Section 5). For the compiler step, the developer should take precautions by using *barriers* if one instruction is absolutely dependent of a previous one inside the micro-architectural domain (like cache state): (1) The `dsb` instruction on ARM, equivalent of the Intel `mfence` instruction, which is a memory barrier; (2) The `isb` instruction on ARM, which is an instruction barrier.

Timer for covert channel. The *timing method* corresponds to the technique used to measure the latency between two memory accesses for the cache side-channel part. We independently found the same results as Moritz Lipp *et al.* [10] and recommend approaches (1) and (2): (1) The POSIX `clock_gettime()` function provides a nanosecond resolution timer and is the easiest method to use. (2) A *counter thread* [10], where we create a dedicated parallel thread incrementing a counter indefinitely, provides an approximation of a cycle-accurate timer while being more complex to use. (3) The `PMCCNTR_EL0` register, an *ARMv8 PMC*, provides a cycle-accurate timer, but its usage is unrealistic as it needs a special kernel module, thus, `root` permissions. (4) The `perf_event` interface, a *Linux API* [17], should provide a cycle-accurate timer, but it was the worst method experimentally – we suppose that the overhead was too heavy for our cache side-channel.

Transient execution window. It corresponds to the time slice – or more precisely, the number of cycles – where (potentially malicious) instructions are executed in the transient domain, *i.e.*, executed speculatively then discarded when the speculation was wrong. For instance, in Spectre-PHT, this time depends on the time taken to compute the condition of the targeted branch. To make the computation of the attacked branch condition as long as possible, we can use slow operations like a division. It will increase the reproducibility of the attack across different systems. Note that in a real attack scenario, we do not choose explicitly the instructions of the targeted branch.

4.2 Reproducibility by Environment Control

These guidelines apply to maximize the chances of reproducing an attack by controlling the attack's environment.

Pinning. This technique prevents the scheduler of the operating system to change the execution core of one process (using the `taskset` command on *Unix* systems). By default, executed processes switch from one core to another during their life-time. For instance, pinning the Spectre process can improve the efficiency of the covert channel part, because if one process performing a cache attack is switched to another core, depending on the processor

micro-architecture and the attacked level of cache, the targeted cache can be changed without the process noticing.

Page size. The cache side-channel implementation (Flush+Reload in our case) tries to spread its probe array over multiples pages in order to defeat the cache prefetcher, since many do not look further than the page size in memory. Page size depends on the kernel and the processor architecture, but 4 kB is very common, thus the size is often hard-coded in the implementation – but it could be different. However, a functionality called *Huge Page* allows the usage of pages larger than 1 MB or 1 GB, and could be enabled transparently in the system [1]. If our cache side-channel rely on a specific page size to work, to avoid impacting its efficiency we should disable this functionally by modifying the kernel's parameters [2].

Frequency. All modern processors adapt their frequencies to the current workload in real time, which corresponds to *frequency scaling* – often referenced as DVFS. If the frequency changes between two latency measurements, the latter could become irrelevant if the frequency value is correlated to the latency. This feature should be disabled using `cpupower` [3] in Linux.

Mitigations. Software or hardware mitigations can be deployed on a system. In our case, since there is currently no hardware protection to defeat Spectre, Linux kernel developers have implemented software mitigations to defeat a Spectre attack against the kernel. These mitigations should not interfere with an attack against a user process. Verifying enabled mitigations can be achieved by a lookup into some files exposed by `sysfs` [4]. In our system, a mitigation called `__user pointer sanitization` was enabled, which consists in disabling speculative execution for some sensitive pointers that are passed to the kernel by the user during a system call, which could be an attempt of a speculative bound check bypass [16].

5 BENEFITS OF (PIPELINE) VISUALIZATION

Some simulation techniques can enhance micro-architectural security research. Section 3 presents papers that successfully used gem5 to develop and validate countermeasures. For offensive security, simulation offers full control and monitoring over each element of the system. For example, the pipeline visualization is a simulator feature which aims to record all instructions that flowed into the pipeline of the simulated processor. Using an external program named Konata [14], we can graphically visualize all the instructions. For each of them, we can identify the cycle of each processing step and see if an instruction has been fetched and committed, flushed or speculatively executed. Using this, we have been able to observe multiple Spectre attack scenarios:

(1) The attack works as expected, we can visually see the malicious transient memory read happening, allowing to have a better understanding of how the attack works.
(2) The training of the branch predictor is not successful, which leads to no transient instruction executed. Being able to observe that is a strong clue for a researcher, since this fact is normally completely hidden from the programmer. This is useful to understand why the attack could fail and identify one reason from the multiple possible ones.
(3) Konata also allows to visualize the order in which instructions are executed. It can be used to identify then solve the re-ordering problem mentioned in Section 4.1.

This pipeline visualization is an easy-to-use tool to obtain advice about the micro-architectural behavior of the attack. gem5 is able to record everything and output the state of any component in the system, *e.g.*, the structures of the branch predictor or the cache.

6 SETUP

The first phase of our work is the implementation of Spectre on a native system. The goal is to have an attack that works correctly and meets special requirements for gem5. We had to do this because the reference implementation [2] was not working properly on our system. The second phase is the replication of the attack on a simulated system using gem5, which involved the development of our own system. We call a gem5 model *faithful* when the number of simulated component and mechanisms matches the real hardware, and call *accurate* a simulation run when some metrics on the simulation matches the same metrics on the real hardware.

For the first phase with the native system, we work on a Raspberry Pi powered by an ARM processor. We chose this board because it is inexpensive and representative of what we can find in the wild. We chose an ARM processor because this architecture is well-supported on our simulator (better supported than x86). More precisely, we used: (1) Raspberry Pi v4 Model B Rev 1.1 as our main board; (2) ARM Cortex-A72 processor, implemented by the Broadcom BCM2711 SoC, as an ARMv8-A processor, with an inclusive LLC; (3) Kali Linux (stable v2020.2a) in the 64-bit flavor as operating system; (4) Linux v4.19.118-Re4son-v8l+ as a kernel; (5) GCC v9.3 as a compiler.

The second phase was under our own simulated system, using gem5 v20.0 with the 64-bit Ubuntu v14.04 LTS and the ARM64 Linux kernel v4.18.0 images, both provided by gem5's developers.

7 IMPLEMENTATION

We provide an explanation of our implementation design and choices during our work in this section. The implementation details, screenshots of visualization, code and experiments are available [5].

7.1 Implementing the Spectre Attack on ARM

The reference implementation – which do not follows our guidelines – of Spectre for ARM [2] was not working properly on our hardware. Indeed, it was relying on compiler optimizations to work correctly. Moreover, towards the goal of reproducibility, we need a specific implementation to fit our requirements, which are: (1) Stable results, meaning that the number of retrieved bytes should be comprised between a reasonable interval; (2) An implementation that follows of our guidelines, in particular *no* advanced compiler flags as we do not want to rely on a particular compiler mechanism for our attack to work, unlike the other implementations we tried;

[1] Huge page check on a Raspberry Pi: `sudo modprobe configs && zcat /proc/config.gz | grep -i k_page`
[2] Huge page disabling on a Linux kernel: add `transparent_hugepage=never` in `/boot/cmdline.txt`
[3] With `root` permissions, DVFS can be disabled issuing the following command: `cpupower frequency-set -g performance`
[4] The path of the file being `/sys/devices/system/cpu/vulnerabilities/spectre_v{1,2}`
[5] https://pierreay.github.io/reproduce-spectre-gem5/

(3) Customizable attack parameters at runtime, allowing parameter exploration for efficiency; (4) Usable in the same way both on native hardware and on gem5 to compare them; (5) *Metrics* output, which are numbers used to compare simulation accuracy (Section 8.)

Conceptually, we used the same algorithm implemented in the original Spectre-PHT proof-of-concept [7], which is an x86 implementation, and we ported it to ARM. Then, we applied all our guidelines of Section 4.1, which was a critical point of our implementation to make it work. To ensure the *cache miss threshold* calibration for Flush+Reload, *i.e.*, the number of cycles where we distinguish the latency of a cache hit from a cache miss, we used the code developed for the *Cache Template Attacks* [5] which allows to obtain the distribution of latencies for cache hits and misses. We used the `clock_gettime()` function as a timer for its efficiency and availability. To measure our metrics, we used counters incremented during our attack as well as the `perf_event` Linux kernel interface for micro-architectural counters. This interface provides access to the Performance Monitoring Unit (PMU) of processor from the user land. This implementation works properly both under the native ARM system and gem5.

7.2 Implementing an ARM gem5 System

In order to reproduce the attack, we need a processor with at least: (1) A speculative execution mechanism with a branch predictor to execute transient instruction; (2) A cache to observe a timing difference when accessing the data.

We derive the `DerivO3CPU` class for our simulated processor. This is an out-of-order processor with a 7-stage pipeline (fetch, decode, rename, issue/execute/writeback, commit). It allows using several caches and a branch predictor. We disqualified the `HPI` class which is a high-performance in-order processor, since transient execution attacks target modern out-of-order processors.

gem5 supports multiple simulation modes. We use the full-system simulation mode, which is the more faithful and demanding in computational power. We define all the simulated hardware objects to simulate a complete and working operating system *via* Python scripts. System-call emulation, where system calls are implemented by gem5 itself and not by a native kernel, is not suitable for a faithful experiment. We derive the `Cache` class for our caches and `ArmDTB` / `ArmITB` for the TLB. We derive the `ArmSystem` class as a system class to handle the ARMv8 architecture and extensions. We use the `VExpress_GEM5_V1` class (simulate the *RealView VExpress* platform) to handle memory mapping, interrupts and GIC, and two `PciVirtIO` to handle operating system and workload images as *VirtIO PCI* block device.

We use `perf_event` to collect metrics. To reproduce it on the gem5 system, we had to patch the gem5 source code to handle this Linux interface. This patch has been included in gem5 such that the latest release handles `perf_event` correctly. We use the `ArmPMU` class configured with an `ArmPPI` interrupt to access the PMU's counters.

We analyzed gem5's implementation of 5 branch predictors, but in the remainder we only describe our configuration. To match as closely as possible our native processor's branch predictor, we choose to use the gem5's *two-level global history-based bi-mode* predictor. Despite the fact that the Bi-Mode predictor separates *mostly-taken* and *mostly-not-taken* branches into two different PHTs, the attack trains the attacked branch as a mostly-taken branch. Consequently, this branch should only be predicted from the mostly-taken PHT, hence, our Bi-Mode predictor acts mostly like the *two-level global history-based* branch predictor of our native processor. We verify this experimentally in Section 8.

Table 1: Ratio between gem5 and Raspberry Pi runs for each metric. A value below 1 means that gem5's metric is lower than the Raspberry Pi's metric.

Metric	Mean	Standard Deviation
Retrieved Bytes	1.05	NaN
Iterations	0.57	3.81
Cycles	0.31	2.12
Cache Misses	584.08	4581.02
Mispredicted Branches	0.99	2.41

After configuring several parameters – caches, branch predictor, TLB, pipeline –, we obtain a system that reproduces broadly our Raspberry Pi with a 4-core processor and two levels of caches – private L1D and L1I and shared L2. Moreover, our simulated processor should perform similarly to our native one for our attack.

Workflow. We prepare 3 images: (1) Kernel image (Linux); (2) Operating system image (Ubuntu); (3) Workload image, dynamically generated with a script, containing the binary of the attack and experiment scripts. We boot-up the system once with a fast and unfaithful processor (`AtomicSimpleCPU`). Then, we take a snapshot of the system state and stop the simulation. This snapshot can be then used at any time. We restore the simulation at the previous point with a detailed and slow processor (`DerivO3CPU`), this is called a *fast-forwarding*. We connect to the simulated system using `telnet`, mount the workload image, and launch the experiment script. Results are then displayed on the terminal.

8 EVALUATION

Reproducing an attack is a first step, but the main question is: "How much is the reproduced attack accurate to the original one?". Indeed, to be useful in security research, the simulation has to be accurate.

Metrics are numbers used to compare runs between the native hardware and the gem5 simulation. They are also useful to understand the attack behavior, *e.g.* allowing to understand if an under-efficiency of the attack is caused by the cache or the branch predictor. Note that the choice is limited, both in terms of number and nature, mainly due to micro-architectural constraints. We choose to monitor: (1) Number of correctly *retrieved bytes* by the attack; (2) Number of *iterations* to retrieve the given number of bytes; (3) Number of *cycles* taken (quantifying the time taken); (4) Number of *cache misses* occurred during the attack; (5) Number of *mispredicted branches*. The last two are micro-architectural counters monitored during the core of the attack – flushing, speculative execution, covert-channel –, while the others are user-land counters. For each run of an attack, 5 metrics are monitored and collected in a table at the end of the run. We have `497` runs on the Raspberry Pi and `10` runs on the gem5 system. This difference can be explained by the time it takes for one run on each system: a few minutes on the Raspberry Pi, between 1 and 5 hours on gem5, depending on the configuration.

Table 1 describes our results. The ratio of each metric is computed by dividing the mean or the standard deviation observed on the gem5 system by the one observed on the Raspberry Pi system. The distribution of our metrics is more disparate on the gem5 system (higher standard deviation), surely due to the low number of runs.

Retrieved Bytes Since the attack retrieved every bytes for each run on the gem5 system (40 bytes per run), the standard deviation was equal to 0, hence the `NaN` notation. The efficiency of the attack on gem5 is close to the reality, with less noise in the system and more predictable results.

Iterations and Cycles The attack on gem5 performs nearly 2 times fewer iterations and approximately 3 times fewer cycles than the native system.

Cache Misses We observe many more cache misses on gem5 than on our native system. We believe that this is an internal problem of communication between the real number of cache misses in gem5 and the number exposed via `perf_event`, but this needs more investigation to determine if the observation is correct or if the tool is faulty.

Mispredicted Branches Mispredicted branches are the most important metric. We see that gem5 has nearly exactly (0.99) the same number of mispredicted branches than our native system, in average. It means that our configuration for the gem5 branch predictor successfully represents the branch predictor of our ARM processor – at least, for this attack.

9 DISCUSSION

Simulation is successfully applied in numerous scientific fields, after addressing the challenges of usability and accuracy. If simulation becomes widely used, thus it would be easy to reproduce older attacks and to study them, *e.g.* for countermeasures development. Indeed, when hardware countermeasures and newer processor architectures will be released, it could become impossible to reproduce an attack. Moreover, not only the replication side will benefit of it but also the offensive side. With faithful models, a researcher could explore and discover new vulnerabilities with a simulated model, then apply them to the real world. We imagine several useful application, for instance it could be powerful to apply parameters enumeration: supposing that we develop a new attack idea that could work only under specific hardware conditions, then we can try different hardware parameters to discover the vulnerable ones.

There are still some limitations, as our work remains to be generalized on other kind of hardware. Moreover, the runs on gem5 for an attack are very slow, and there is still less noise on the gem5 system than the native one, which impacts the practicability and the fidelity of the system.

10 CONCLUSION

In this paper, we first described how to increase reproducibility in security research, further we believe that fully reproducible builds are a promising way of doing it [8]. We have shown how to start micro-architectural security research with a cycle-accurate simulator, namely gem5. We saw that it is possible to simulate micro-architectural attacks, moreover, that it could be accurate comparing it to a native machine.

We can imagine several ways to use gem5 in order to find new attacks and countermeasures, but also to help in attack and countermeasure implementation and understanding. We can identify three future directions on gem5 for security research: (1) Visualization is a very powerful technique to understand the micro-architectural behavior, and this knowledge can be directly applied to security. Currently, we can conveniently observe the pipeline, but being able to visualize the state and history of the branch predictor or the caches would also be very effective. (2) We can improve model's faithfulness by implementing missing components to make a specific attack work. These components are still to identify by future research. (3) We can improve the possibilities given by a gem5 simulated system by fixing – just like we did with `perf_event` – or implementing missing security features. For instance, TrustZone and SGX are still unsupported, but studying the micro-architecture impact on these features under gem5 could be a great advantage.

This work benefited from the support of the DGA and the project ANR-19-CE39-0008 ARCHI-SEC.

REFERENCES

[1] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* (2011).

[2] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. [n.d.]. *TransientFail Repository*. IAIK. https://github.com/IAIK/transientfail Consulted on 2020-08-30.

[3] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security*.

[4] Fernando A Endo, Damien Couroussé, and Henri-Pierre Charles. 2014. Micro-Architectural Simulation of In-Order and Out-Of-Order ARM Microprocessors with Gem5 *(SAMOS XIV)*. CEA.

[5] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security*.

[6] Anthony Gutierrez, Joseph Pusdesris, Ronald G. Dreslinski, Trevor Mudge, Chander Sudanthi, Christopher Emmons, Mitch Hayenga, and Nigel Charles Paver. 2014. Sources of Error in Full-System Simulation *(ISPASS)*.

[7] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution *(S&P)*.

[8] Chris Lamb, Holger Levsen, Mattia Rizzolo, and Vagrant Cascadian. [n.d.]. *Reproducible Builds*. https://reproducible-builds.org/

[9] Chih-Chieh Lee, I-Cheng K. Chen, and Trevor N. Mudge. 1997. The Bi-Mode Branch Predictor. In *MICRO*.

[10] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security)*.

[11] Kevin Loughlin, Ian Neal, Jiacheng Ma Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. 2021. DOLMA: Securing Speculation with the Principle of Transient Non-Observability. In *USENIX Security*.

[12] Jason Lowe-Power. [n.d.]. *Visualizing Spectre with gem5*. http://www.lowepower.com/jason/visualizing-spectre-with-gem5.html

[13] Sparsh Mittal. 2016. A Survey of Techniques for Dynamic Branch Prediction. *CCPE* (2016).

[14] Ryota Shioya. [n.d.]. *Visualizing the out-of-order CPU model*. http://learning.gem5.org/tutorial/presentations/vis-o3-gem5.pdf

[15] James E. Smith. 1998. A Study of Branch Prediction Strategies *(ISCA)*.

[16] Linus Torvalds. [n.d.]. *Spectre Side Channels*. https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/spectre.html Consulted on 2020-09-06.

[17] Vincent M. Weaver. [n.d.]. *Linux perf event Features and Overhead*. http://web.eece.maine.edu/~vweaver/projects/perf_events/ Consulted on 2021-02-05.

[18] Yuval Yarom and Katrina Falkner. 2014. Flush+Reload: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security*.

[19] Tse-Yu Yeh and Yale N. Patt. 1991. Two-Level Adaptive Training Branch Prediction *(MICRO)*.