SpecDefender: Transient Execution Attack Defender using Performance Counters

Amit Choudhari[1], Sylvain Guilley[2] and Khaled Karray[3].

[1] ORCID: 0000-0002-5578-7061 ; Institut Polytechnique de Paris, France
[2] ORCID: 0000-0002-5044-3534 ; Secure-IC S.A.S., France
[3] ORCID: 0000-0001-9400-2848 ; Secure-IC S.A.S., France

Abstract

Side-channel attacks based on speculative execution have gained enough traction for researchers. This has resulted in the development of more creative variants of Spectre and its defences. However, many of these defence strategies end up making speculative execution or branch prediction ineffective. While these techniques protect the system, they cut down performance by more than 50%. Hence, these solutions cannot be deployed.

In this paper, we present a framework that not only protects against different variants of Spectre but also maintains the performance. We prototyped this framework using a novel tool SpecDefender. It leverages Hardware Performance Counter (HPC) registers to dynamically detect active Spectre attacks and performs dynamic instrumentation to defend against them. This makes the tool widely applicable without any need for static analysis. Overall, the tool brings back the balance between performance and security.

The tool was evaluated based on its accuracy and precision to detect an attack in different scenarios. It exhibit ¿90% precision when five out of ten processes were simultaneously attacked. The response time for the tool to detect is 2 sec. Furthermore, the throughput of the process under attack was comparable to normal execution in presence of SpecDefender.

Keywords: Transient execution attack, speculative execution, Spectre, runtime attack detection, efficient mitigation.

1 Introduction

Performance and power are two primary deciding factors in the CPU supremacy race. Hence CPU architecture designers are forced to consider the trade-off between power and performance. The security aspects are overlooked in this tug of war. One such performance-enhancing feature is speculative execution. It avoids stalling the pipeline by predicting the outcome of conditional branching and speculatively executing corresponding instructions. If the prediction is correct, the results are committed. Whereas, if the prediction turns out to be incorrect, it aborts speculative execution and discards the results by flushing the pipeline. However, the speculative execution has already affected the micro-architectural state, which comprises internal registers, flags and cache, cannot be rolled back. The modified micro-architectural state can be captured using covert channel cache attacks, such as Flush+Reload [25], Flush+Flush [9], that take advantage of the faster load time of recently accessed data. Spectre is an attack that is based on the combination of three factors, CPU optimization that allows non-linear code flow using branch predictor, execution of illegal instructions using speculative execution and retrieval of leaked information using cache attack. The meaningful information leakage is amplified by selectively targeting the vulnerable code. This leaked information consist of metadata of a library in use, or even a private key recovery.

The feasibility of executing a successful attack is

non-trivial, citing its likelihood as per CWE-1037 as low [1]. Nonetheless, the attack relies on fundamental performance-optimizing blocks on all modern general-purpose processors. Therefore, the impact of spectre on security cannot be ignored. There are numbers of detection and mitigation techniques that have been proposed [24][21]. Detection techniques can be *static*, e..g., leveraging symbolic execution [10], program verification frameworks [4], of *dynamic*, e.g., leveraging performance counters [15], or detecting Spectre gadgets off-the-fly [19]. For static analysis tools, given a set of inputs and secure memory objects, these tools identify patterns that are vulnerable to Spectre attack. They are excellent at identifying the correctness of a program but lack soundness and completeness of the analysis when the size is large. On the contrary, dynamic analysis is quite efficient at identifying known patterns of attack.

Disabling speculative execution or branch prediction will cause a significant drop in performance and hence mitigation of Spectre at the hardware level is difficult. Introducing load barriers before conditional jumps [11], linearizing code by eliminating branches [21], protecting linear basic blocks by flushing BTB (Branch target buffer) at the entrance [16]. A study on performance evaluation of deployed mitigations in compiler and operating systems for Spectre have already observed an overhead of 20% [5]. Some mitigation proposes a brute force approach of force stopping the program under attack [15]. This approach seems applicable only to a non-crucial application but cannot be applied to crucial applications such as firewalls and antivirus. It will allow an attacker to perform a multistage attack where Spectre will be the first stage used to kill the firewall service. All these mitigations ensure security at the cost of performance or denial of service. Both these consequences are undesirable to a user and hence cannot be productized. This raises two important questions.

Is it possible to maintain availability while Spectre is active? Is it possible to ensure security without impacting user experience?

In this paper, we answer both of these questions positively. We propose a novel approach to maintain a balance between security and performance. We present a tool that dynamically identifies Spectre attacks on a process and protects the system with temporary mitigation while the attack is still active. This dynamic defence mechanism allows users to efficiently run the Spectre-vulnerable process without the need for hardware/software upgrades, OS/compiler mitigations or hardware countermeasures. There are two ways to perform dynamic protection:

1. Temporary run-time instrumentation of process with mitigation.

2. Maintain two pre-compiled binaries, one efficient without Spectre mitigation and another inefficient with Spectre mitigation.

Run-time instrumentation and loading of inefficient Spectre-safe binary are performed only when an attack is detected. It not only ensures the availability of the service but also provides better performance overall. Spectre is a transient execution attack. It implies it is temporary and the attack needs to perform a cache attack along with continuous patterned execution to leak information. Cache attacks are known to be suppressed under noise, hence it takes a longer time to leak secrets[9]. Our tool SpecDefender has a very short response time which helps in detecting Spectre before it is successful.

For our evaluation, we analyzed the precision and accuracy of the tool to identify and defend against Spectre while multiple attacks are active on different processes. We evaluate the time taken from the start of analysis to defending the attack. And compare the performance gain of our solution with the existing approach of statically disabling branch prediction.

1.1 Contributions

Our primary contributions are:

- To emphasise the importance of the availability of a process. Explore the possibility of Spectre performing denial of service attacks and more complicated attacks.

- Our solution uses the multi-class model to detect different variants of Spectre and efficiently defend against it with only a temporary drop in performance.

- We evaluate our solution on its precision, accuracy and performance. Moreover, we also show the efficacy of the solution on Spectre v1 and v2.

- Our results give confidence in a dynamic approach that could be plugged into a system security service that monitors the health of running programs.

1.2 Outline

The paper is organized as follows. Section 2 provides an understanding of the Spectre attack, the core modules leveraged by the attack, different variants of the attack and simulation of the attack on GEM5. Moreover, we also take a deeper look at the timing traces left while the attack is active. Section 3 describes the state-of-the-art solutions available for the detection and mitigation of Spectre attacks. In Section 4 we present SpecDefender, our tool to defend against Spectre. In Section 5 we evaluate the performance, accuracy and evaluation. Section 6 summarizes and emphasises the impact of this tool. Finally, further works are envisioned in Section 7.

2 Background

This section provides background information on modules involved in speculative execution attacks and two variants of Spectre. We reproduce the attack on GEM5 with both variants and observe the pattern of the microarchitectural state as the attack progresses.

2.1 Branch prediction

In a CPU, conditional branches cause control hazards. Executing an incorrect branch path forces the pipeline to flush. It eventually results in the degradation of throughput. A branch predictor makes a calculated guess of a likely outcome of the conditional branch. Enqueuing most likely instruction ensures that the pipeline is full and throughput is optimum. There are three types of branches:

1. Direct jumps and calls: Always follow a static path.
2. Indirect jumps and calls: Jump address varies.
3. Conditional branches: The decision for a jump is conditional.

Out of these three, only 2 and 3 are responsible for Spectre. To handle these types, the processor uses different components to predict the outcome of a branch. The Branch Target Buffer (BTB) maintains a mapping of the branch instruction and its recently taken next instruction. It enables the processor to fetch the next instruction before even decoding the branch instruction. Branch Prediction Buffer (BPB) is a two or more-bit buffer that maintains a state used to guess the outcome of a conditional branch. These states are strongly-taken, weakly-taken, weakly-not-taken and strongly-not-taken. A branch history register (BHR) keeps the branch history of the recent conditional branches.

2.2 Speculative execution

In an instruction pipeline, there are often instances where the next instruction cannot be executed due to data or control hazards. For instance, in a `for` loop with n iterations where the loop invariant is not available, the loop code is still executed speculatively. That is the current register state is stored and the next instruction is speculatively executed. The result is only committed after the loop invariant is fetched and validated. If the speculation was correct, a stall was avoided and hence resulting in a performance boost. But in case of incorrect speculation, the results are discarded and the old register state is restored to execute the right instruction.

It would seem that processor adheres to the correctness of the program without any side effects. But, this erroneous execution of instruction leaves microarchitectural traces and is hence called transient instructions. Spectre uses these microarchitectural traces to fetch information about the result of the executed instruction. For example, the result of a conditional branch can be delayed by flushing the cache causing the processor to speculatively execute the code of the attacker's choice.

2.3 Microarchitectural side-channel attack

Micro-architectural analysis technique such as cache attack is used to gather information about a program containing a side-channel. At any given time multiple programs are running on the CPU with shared resources. These shared resources may result in unintended information leakage. These resources could be L1 data cache, instruction cache, branch history, and even some cross-core and cross-CPU shared IO.

For demonstrating Spectre, we focus on cache attack techniques such as Prime+Probe, Flush+Reload, Flush+Flush, etc. These are timing attacks, where a chosen cache line is probed to know whether it was used by another program. For instance, an attacker would fill the cache lines with its data and allow the victim program to run. After some time the attacker will read its data. While doing so, the attacker will know whether the corresponding cache line has also been used by the victim program from its fetch delay caused by a cache miss.

2.4 Spectre

A Spectre attack lets an attacker speculatively execute a sensitive block of the program which will not be possible in sequential in-order execution. This speculative execution causes the microarchitectural state to change and the secret information can be made to leak using covert channel [11]. This attack is carried out in three stages and we demonstrate it with a Spectre-prone code snippet in listing 1.

In the first stage of initialization, the attacker mistrains the branch predictor and prepares for the covert channel to extract secret information. That is it forces to speculatively execute the `if` block in `victim_function()` and perform flush part of the Flush+Reload attack. In the second stage of an attack, a delay is introduced to get the result of `if` condition that results in speculatively executing sensitive code. The delay can be caused by flushing the variable from the cache. In listing 1, the variable `array1_size` is flushed and the value for `x` is chosen such that the data in `char *secret` is fetched in cache. In a sequential execution, the data outside the `array1_size` bound can never be accessed due to `if` condition check. In the third stage of recovery, cache attacks are performed to retrieve the leaked information. Flush+Flush or Flush+Reload techniques are used to determine the time delay to fetch the data. For example, a cache line that was replaced by `victim_function` to fetch `array2[array1[x]]` will take longer to load the data. With this hint, the attacker can guess the secret data as the cache line corresponds to `array1[x]`. And as `x` is controlled by an attacker, any data mapped in the virtual address space of the victim program can be leaked. To achieve higher success rate to leak data and overcome noise, these three steps are tried several times.

The core of a Spectre attack is an attempt to perform speculative execution of sensitive code in order to leak secret information. There are two variants of this attack that we shall discuss in this paper.

Listing 1: Spectre prone code snippet [3].

```
uint8_t unused1[CACHELINE];
uint8_t array1[160] = {1, 2, 3, 4, 5,
    6, 7, 8, 9, 10, 11, 12, 13, 14,
                          15, 16};
uint8_t unused2[CACHELINE];
uint8_t array2[256 * PAGESIZE];
char *secret = "All we have to decide
    is what to do with the time!";
/* Function that will be tricked by
    Spectre. */
static void victim_function(size_t x)
    {
     /* Flush the variables used in the
          condition to add
     a higher delay. */
     mfence();
     flush(&array1_size);
     flush(&x);
     /* Ensure data is flushed at this
        point. */
     mfence();
     ifence();
     /* Perform a legitimate array
        access, with bound checking.
     This branch will be tricked by
```

```
    Spectre during the attack
  phase. We use a division instead
    of an int-comparison
  since it takes more time, thus
    increase the
  transient execution window. */
  if ((float) x / (float)
    array1_size < 1)
    temp &= array2[array1[x] *
      PAGESIZE];
}
```

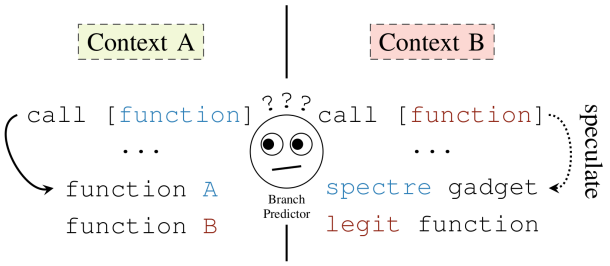

Figure 1: Mistraining branch address in Spectre V2 [11].

2.4.1 Spectre v1: Conditional branch misprediction

This variant exploits conditional branches by speculatively executing sensitive code and leaking arbitrary memory. Listing 1 is a standard example of a Spectre v1 attack. The attacker uses `x` to read arbitrary value, in this case `secret`. And `array1_size` is flushed from the cache to introduce delay while resolving the conditional branch. This forces the CPU to speculatively load `array2[array1[x]]` with unchecked malicious `x` chosen by the attacker. Using Flush+Reload, the attack is finally completed by measuring the load location of `array2[array1[x]]` i.e value of `array1[x]`.

There are several reasons for not knowing the result of the conditional branch immediately. For example, a cache miss while checking the branch, complex arithmetic dependencies or nested speculative execution. Additionally, speculative executions are observed even when there is no delay but only by applying branch prediction results, that turn out to be incorrect. Simple ways to avoid such array out of bounds is by always applying modulus to index. For example, `array2[array1[x%array1_size]]`. This quick fix doesn't solve Spectre entirely but will have an upper bound on possible leakage.

2.4.2 Spectre v2: Indirect branch gadget

Indirect branches are commonly used in the code to add dynamic capabilities. However, these indirect branches can be poisoned to create a gadget code. That can further be used to read arbitrary memory from another context to leak data. The idea is that cache miss can cause a delay in the determination of the target address. And to optimize, speculative execution uses the last used address for branching. So, an address valid in the previous context, that is invalid in the current context can be invoked speculatively. To perform an attack, the adversary mistrains the branch predictor with a gadget address in one context and forces the CPU to reuse the mistrained address in another context (see Figure 1).

2.4.3 Attack simulation

Even though the attack has exact steps to reproduce, it is not easy to leak the data. Both mistraining the branch predictor and performing a cache attack require several attempts. And the difficulty is increased further in real scenarios with multi-core, multi-process systems adding more noise. Moreover, Spectre does not leave a trace, hence there is no evidence of real Spectre attacks.

To examine the micro-architectural state during Spectre attack we simulated Spectre v1 on Gem5 [3]. Gem5 is an open-source processor simulator that encompasses system-level architecture details. It simulates hardware components on a cycle-by-cycle basis, instead of simulating the instruction set of architecture. This fits the cycle-accurate requirement needed to simulate a Spectre attack. In addition, we used Konata [22], a powerful tool used to investigate the pipeline state of the system. This helped us visualize patterns in branch hit/miss speculatively executed committed/retired instructions during the attack.

Figure 2 shows the pipeline traces collected while running a Spectre attack. For completion, every instruction goes through all pipeline stages, instruction fetch (purple), decode (blue), rename (green), issue (yellow) and writeback (orange). Spectre is a three-stage attack, where the first stage mistrains the branch predictor. This behaviour can be observed in the macroscopic view on the right side (green highlighted trace). A snippet of code was executed 5 times without any misprediction or speculative failure. After mistraining the branch predictor second stage tries to speculatively execute an illegal instruction. This results in flushing the pipeline line and discarding the computed results. These two stages are run for several iterations to ensure the victim code execution leaves a micro-architectural trace in the cache. This experiment confirmed Spectre behavior through the patterns and we make key observations:

1. Periodic branch miss might lead to a higher branch hit rate.
2. Speculative execution is forced to fail, this might result into lower performance.
3. Variables are flushed from cache by the attacker to introduce delay, this would lead to a higher cache miss rate.

From these observations we perform feature selection to design SpecDefender in Section 4.1.1.

3 State-of-the-art methodologies

In this section, we discuss the related work on detection and mitigation techniques for a Spectre.

3.1 Detection

As Spectre could potentially abuse any conditional branch, it is inefficient to add load barriers before every branch instruction. Hence, it is important to know the hardware architecture and the dependency of the condition on inputs to efficiently detect Spectre vulnerable code. RH scanner, is a static analysis tool for scanning Spectre v1 [6]. Oo7 performs static taint analysis to detect information flow from inputs to potential branches [23]. However, static taint analysis suffers from over-tainting and undertainting issues due to inaccurate control flow graphs. SPECTECTOR mathematically defines a new notion to prevent speculative execution and uses it for symbolic execution to prove the absence of Spectre gadget [10]. However, it lacks soundness and completeness, a common drawback of symbolic execution while analysing large programs. Dynamic analysis tools such as SpecFuzz [18] and OSIRIS [24], perform random mutation of inputs in runtime to detect speculative execution errors. It limits due to the probabilistic and hence misses out on errors.

3.2 Mitigation

Disabling speculative execution or branch prediction is the most intuitive mitigation, but unfortunately, this will hit the performance. SPECFUSCATOR [21], proposes an interesting solution of unwrapping the loops and eliminating conditional branches. This technique eventually ends up deactivating the branch predictor, hence it is deployable. Swivel [16] is a compiler framework for WebAssembly that converts basic blocks into linear blocks and instruments guards on entry and exit of the block. These guards prevent against mistraining of conditional branches.

In this paper, we look at the problem holistically and propose a more deployable solution.

4 SpecDefender: dynamic defence tool against Spectre

Spectre abuses the performance of optimizing modules in a CPU. Hence, the mitigation and fixes for Spectre either end up reducing the performance drastically or stopping the service. Both these outcomes are undesirable to the user. For critical applications such as firewalls and antivirus, stopping is not a solution. As it could lead to a multi-staged attack. We introduce SpecDefender, a tool that dynamically detects Spectre attacks, and defends against them while maintaining the overall performance. It is a novel and

Figure 2: Macroscopic view of the pipeline stages during active Spectre attack. [right] Green highlight shows mistrain and [left] red shows retired instructions due to incorrect speculative execution.

practical approach that tackles Spectre from a software availability standpoint. That is, the program should always be operational even under attack.

In this section, we introduce the design of SpecDefender in the first part. Then discuss the implementation.

4.1 Design of Specdefender

SpecDefender is a tool based on the same principle as an IDS (Intrusion detection system). It dynamically detects a threat from an anomaly in the system behaviour and prevents it by blocking the attack while the threat is still active. The dynamic analysis makes the tool portable without any need to parse the source code. It makes the tool applicable for a large set of old pre-compiled applications.

As shown in figure 3, SpecDefender operates in two phases, a detector phase and an instrumentation phase. In the detection phase, SpecDefender inspects each process with its run-time counters. If anomalous behaviour due to Spectre is detected, The process is instrumented with mitigation at run-time. The process continues to be under observation until the attack is active. After detecting the absence of the attack, the process is again instrumented by removing the mitigation to perform normally. This adaptive behaviour of the tool keeps a balance between security and performance. In the following section, we will discuss these stages and the rationale behind them in detail.

4.1.1 Detector

In a typical malware attack, the malicious application creates sockets, and files or changes permissions to leak data. Spectre is a transient execution side-channel attack, that does not leave any trace of a successful attack [11]. However, transient execution attacks do change the microarchitectural state of the CPU [14]. Hence we chose to use Hardware performance counters to detect active Spectre attacks on a process.

Hardware Performance Counters (HPC) HPC are special hardware registers primarily used for monitoring performance and debugging program execution. It shows global and per-process run-time event counters for branch hit/miss, cache hit/miss, CPU cycles etc. These counters fit perfectly for our use case as they are generic, can be read at run-time and do not need any pre-configuration. Based on our attack simulation on GEM5 in Section 2.4.3 we identified six HPC registers such as branch-miss, branch-predict, cache-miss, cache-reference, instructions, speculative_load. The branch-miss and branch-predict values indicate a constant mistraining of a branch. To measure the time delay, the attacker has to perform several cache flushes. The cache-miss and cache-reference counters reflect these

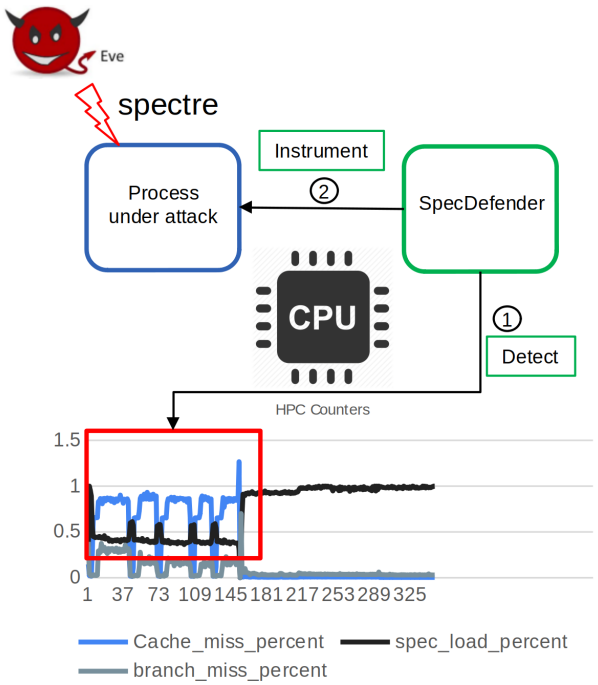

Figure 3: Design of SpecDefender.

flushes. Moreover, due to the continuous flushing of caches, the number of instructions executed in a preset duration could be seen from Instructions HPC. Forcefully retiring instructions for leaving micro-architectural trace will be observed from variation in speculative_load counter values.

Feature selection We collected continuous HPC data while running the process in four different modes. Execution without attack (`normal`), execution under attack (`attack`), load barrier instrumented execution under attack (`safe_attack`) and load barrier instrumented execution not under attack (`safe`). In this section we only focus on `attack` and `normal` mode, other modes will be discussed in detail in section 4.1.3. From Figure 4, it can be observed that the cache miss rate is high when the attack is active. On the other hand, branch miss rate seems to be indistinguishable for `attack` and `normal` modes [Figure 5]. The reason for this observation is primarily due to the total number of branches in a process is relatively high causing high noise. Hence, it is difficult to observe the forced mistraining of a single branch. The same observation was concluded for instruction HPC. Speculative load HPC values show observable distinguishable between instrumented (`safe, safe_attack`) and non-instrumented modes (`normal, attack`)[Figure 6]. Hence, we dropped branch rates and instructions from the feature and only kept speculative load and cache miss rates. This eventually increased the accuracy of the trained model [Figure 7].

4.1.2 Temporary runtime instrumentation

Having detected the Spectre dynamically, shutting down the service seems to be the most intuitive solution. It will impact the user experience and create room for a two-stage threat model, where the first stage is to shut down a service. To ensure the availability of software, we decided to instrument the service with a protection code at runtime. Speculative execution is the primary cause for Spectre attack and adding load barriers before conditional jumps prevent [11][10]. Inspired by MOVFUSCATOR, a compiler that generates binary with only MOV instruction,

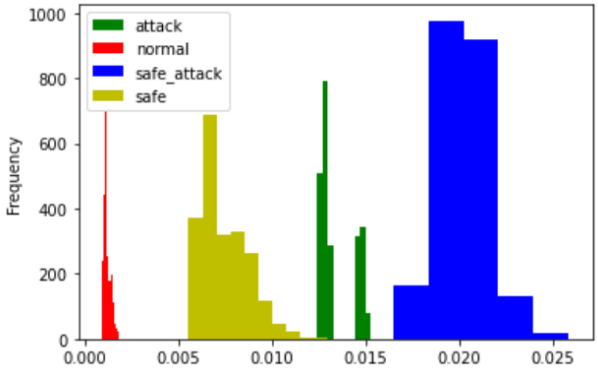

Figure 4: Histogram plot for cache miss rate in different modes.

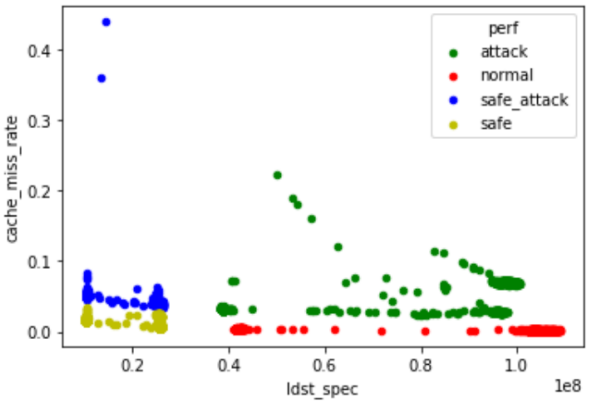

Figure 6: Scatter plot of cache miss rate versus speculative load for different modes.

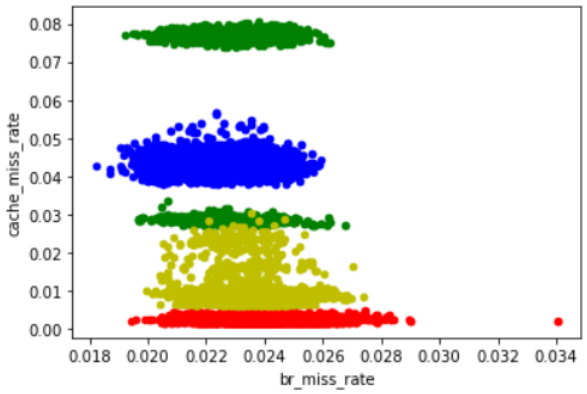

Figure 5: Scatter plot for branch miss rate in different modes.

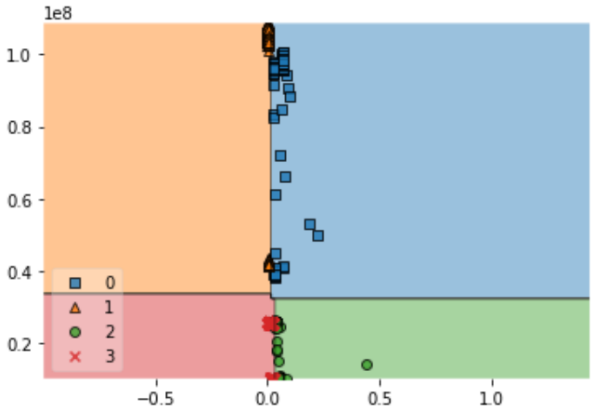

Figure 7: Decision diagram plotted on test set. Legend [0]: `attack`, [1]: `normal`, [2]: `safe_attack`, [3]: `safe`.

some techniques linearize the code to eliminate conditional jumps [21].

Dynamic binary instrumentation (DBM) tools run an application in a virtual environment and perform JIT for instrumentation [8]. There are other more developer-friendly but architecture-specific tools such as PIN, that allow dynamic instrumentation [20]. Another creative way would require adding NOP instructions during compilation and instrumenting these instructions with a load barrier at runtime. In the open source community, there already exist several generic and platform-specific DBI tools [7][2][17]. Hence, we used pre-instrumented programs for demonstrating our proof of concept.

4.1.3 Multi state classification

After finding a differentiable pattern between a process under Spectre attack and the same process not under attack, we need a defence mechanism against the attack. This can be performed in two ways.

1. Pre-compile two versions of executable. First version is efficient but prone to spectre attack. Second version has static mitigations against spectre, hence secure but inefficient. This executable is run when attack is active.That is in state `safe_code_under_attack` and `safe_code` [Figure 8]

2. Dynamically instrument the program with spectre mitigation. This method is uninterrupted and more future proof, without need for recompiling app.

The HPC counters for this instrumented/secure program is again classified as `attack`. To solve this issue seamlessly, we trained the model to detect four states through which any program under attack would transition. The first state is `normal`, where the program is run without any mitigation and is not under attack. When a Spectre attack is active in the running process, it is classified as a `attack` and immediately instrumented with protection (load barriers and retpoline). After instrumentation, the process is still under attack but classified as `safe_attack`. It stays in this state till the attack is active. Once the attack stops, it is classified as `safe` and immediately instrumented to remove protective code. The program is again classified as `normal` and continues to run at best performance. This cycle allows the program to efficiently transition from `normal` execution mode to `attack` and back to `normal`. Moreover, it also improves the performance of the process without compromising security.

4.2 Working and implementation

As discussed in section 4.1, SpecDefender operates in two phases, the detection phase and the instrumentation phase. In the detection phase, the tool periodically reads the HPC registers for cache-miss, cache-references, and ldst-spec. For every running process, it collects samples of this data every 100 msec for 2 sec. This data is fed to a pre-trained model to classify the state of the program [Figure 8]. On classification, the process is instrumented if it is detected in state `attack` and `safe_attack`. After this, the tool continues to inspect another running process. As it continuously inspects all the running processes sequentially, it might take longer to inspect the same process again. A simple solution is to spawn multiple threads to collect data or prioritize based on past activity. The overall execution time for a single process is less than 2.5 sec. The majority of inspection time is for data collection.

In the detection phase, the multi-class model is trained with XGBoost one vs one classifier. The accuracy of this model for the random test set was above 98%. The source code for the tool, jupyter notebook, pre-trained model, sample Spectre attack and README file is available at `https://github.com/amitsirius/Specdefender`

5 Evaluation

We evaluate the SpecDefender tool based on its capability to detect different variants of Spectre attack, its accuracy and precision to detect all known vulnerabilities and performance based on verification time.

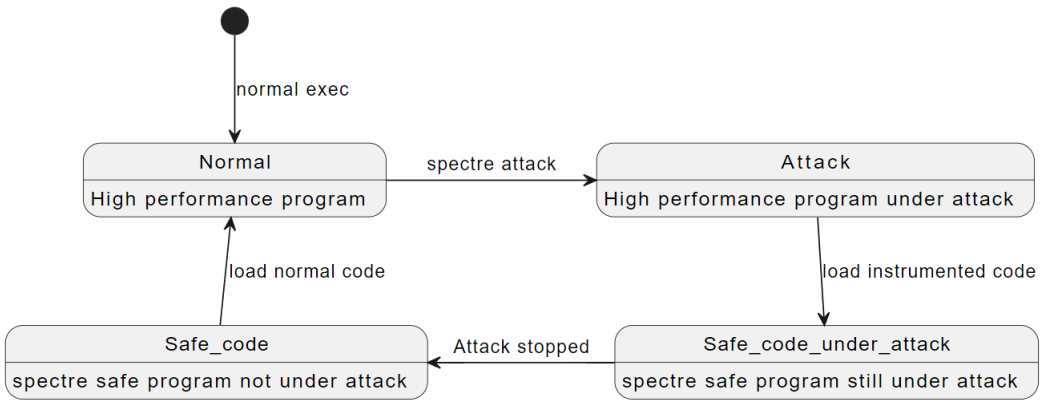

Figure 8: State transition of a program under Spectre attack.

5.1 Accuracy and precision

5.1.1 Setup

We noted the true positive(TP), true negative(TN), false positive (FP) and false negative (FN) for computing the accuracy and precision of the tool [Table. 1]. We designed six test cases for analyzing the factors affecting accuracy and precision. `TP` are the instances where the attack was rightly detected by the tool i.e `attack` and `safe_attack`. Whereas, `TN` instances describe the instance when the attack was absent and the rightly detected the absence i.e `normal` and `safe`. Each test `xA_yN` will contain `x` process under attack and `y` process that are not under attack. For example, in `3A_1N` SpecDefender will run in an environment with three processes simultaneously under active Spectre attack and one process that is not under attack. Each test case is run only for one cycle, i.e from `attack` state to `normal`. This test is performed for multiple fresh iterations to eliminate any causal effects of previous wrong detection. One additional test `5A_5N_L` is performed for a longer duration to understand the behaviour when the tool runs for a longer duration. In other words, it allows Specdefender to probe the same process for multiple iterations.

5.1.2 Observation

Figure 9 shows that accuracy of the tool declines as the number of processes under attack increases. Similar behaviour is followed by the precision graph as well. The primary reason for this is inaccuracy in calculating HPC counter values per process. Simultaneously running multiple attacks adds more noise in the calculation and hence results in misclassification by the detector. On the other hand, test `5A_5N_L` shows an improvement in detection. It infers that multiple rounds on the same process cause improvement in both accuracy and precision. After the attack has stopped, the process is identified in a normal state on every subsequent iteration. This results in more samples identified as `TN`.

5.2 Effectiveness on different variants

As discussed in section 2.4, Spectre attack has different variants. And these variants uses speculative execution to perform illegal execution of code (v1) or gadget code (v2) or ROP attack (v3)[13]. SpecDefender tool detected Spectre-v1 and v2 with high accuracy. Although the vulnerable code was different, the pattern observed with HPC counters is similar. Spectre v3 is platform specific and left for future work.

Table 1: Accuracy and precision table for six test cases.

Test	TP	TN	FP	FN	Total	Accuracy	Precision	FP rate	FN rate
1A_3N	7	41	0	2	50	0.96	1.000	0.000	0.222
2A_2N	9	38	0	3	50	0.94	1.000	0.000	0.250
3A_1N	17	29	1	3	50	0.92	0.944	0.033	0.150
3A_3N	12	44	2	3	61	0.918	0.857	0.043	0.200
5A_5N	12	43	3	4	62	0.887	0.800	0.065	0.250
5A_5N_L	13	44	1	2	60	0.95	0.929	0.022	0.133

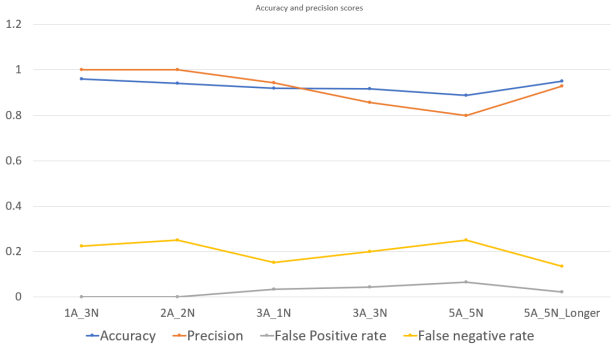

Figure 9: Accuracy and precision for six test cases.

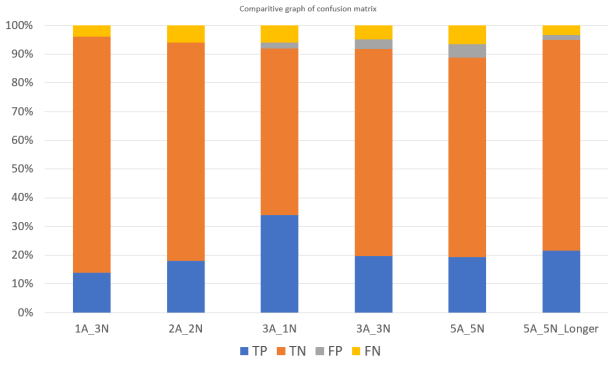

Figure 10: Bargraph showing proportion of TP, TN, FP, FN for six test cases.

5.3 Performance

We evaluate the performance on two fronts. Firstly, the responsiveness of the tool to examine the time taken by each module and find room for improvement. Secondly, the comparative throughput of the process undergoing dynamic instrumentation and static mitigation.

5.3.1 Responsiveness of tool

We profiled individual modules of the tool as per their functionality to evaluate its responsiveness for detecting Spectre [Figure 11]. HPC data collection is the most time-consuming module. Finding an optimal time to detect the state without loss of precision will reduce the response time. Further optimizations are possible by adapting the design to collect HPC data offline in parallel instead of in sequential collection. Compared to time spent on HPC data collection, all other modules take less than 10% time. The response time observed over multiple iterations for detecting Spectre is less than 2.3 sec. Whereas, a successful spectre attack takes at least a few minutes to leak data [3].

5.3.2 Throughput of process under attack

One of the goals of SpecDefender was to maintain a balance between performance and security. To evaluate this, we ran the process in three different scenarios [Figure 12]. First, normal execution without any mitigation against Spectre (Normal_exec). Second, a program with mitigation against Spectre. Third, dynamically protected process under Spectre attack using SpecDefender (Optimized_exec). It was observed

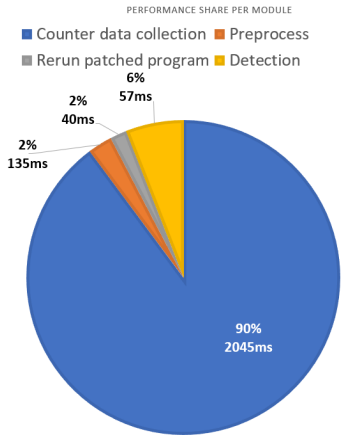

Figure 11: Execution time for each block compared to total time.

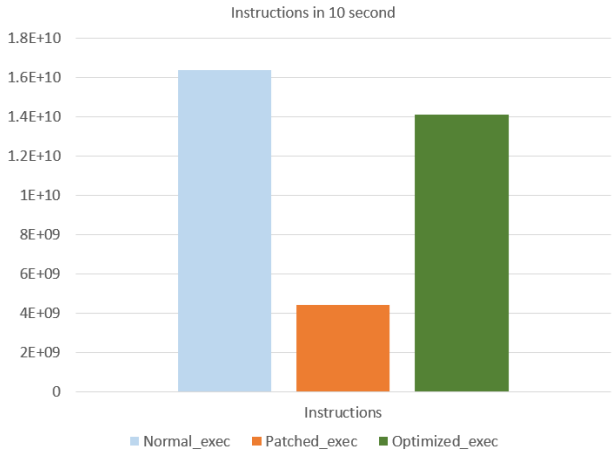

Figure 12: Throughput comparison of program run in normal, statically mitigated and SpecDefender optimized scenarios.

that process with SpecDefender had a throughput of 87% as that of normal execution. Whereas the statically mitigated process had a throughput of less than 40%.

6 Conclusion & Perspectives

We presented a generic framework that dynamically defends against Spectre attack while ensuring that the availability and performance of the process are not compromised. To demonstrate this idea, we developed a tool, SpecDefender that dynamically detects Spectre attacks from patterns in HPC and temporarily instruments the program. Using this tool we successfully demonstrated security by effectively detecting two variants of Spectre with a short response time. We also show the throughput of the process under attack has an insignificant drop. Often it is difficult to perform static analysis on already deployed old programs to find Spectre-prone code. SpecDefender overcomes this by detecting the attack on micro-architectural traces and remains agnostic of source code. We envision this framework to be part of a security kernel service that actively monitors the running process.

7 Further works

Using SpecDefender, we described a proof of concept to resist Spectre dynamically. To the best of our knowledge, we are the first to tackle this attack while maintaining the availability and performance of the software. Many different adaptations and experimentation have been left for future work due to a lack of time. From the initial feedback from the reviewers, we understand there is a need for improving the response time of the tool. We aim to achieve this by interleaving detection and escalating the protection. One such adaptation is by spliting the functionality of the tool into three independent stages. Where stage 1 collects data, stage 2 detects the attack using a pre-trained model, and stage 3 instruments the program. Such pipelining techniques should improve the response time of the tool. We also plan to perform stress testing on standardized testbeds and improve the tool's robustness. Responsiveness is key to detect the so-called "evasive Spectre" attacks, studied recently [12], hence developing our pipelined architecture is a strategy in this respect.

Acknowledgments

The work presented in this paper was realized in the framework of the ARCHI-SEC project number ANR-19-CE39-0008-03 (`https://archi-sec.telecom-paristech.fr/`) supported by the French "Agence Nationale de la Recherche".

References

[1] Cwe-1037. Available from MITRE, CWE-ID CWE-1037., 2018.

[2] Paul-Antoine Arras, Anastasios Andronidis, Luís Pina, Karolis Mituzas, Qianyi Shu, Daniel Grumberg, and Cristian Cadar. Sabre: Load-time selective binary rewriting. *Int. J. Softw. Tools Technol. Transf.*, 24(2):205–223, 2022.

[3] Pierre Ayoub and Clémentine Maurice. Reproducing spectre attack with gem5: How to do it right? In *Proceedings of the 14th European Workshop on Systems Security*, EuroSec '21, page 15–20, New York, NY, USA, 2021. Association for Computing Machinery.

[4] Gilles Barthe, Sunjay Cauligi, Benjamin Grégoire, Adrien Koutsos, Kevin Liao, Tiago Oliveira, Swarn Priya, Tamara Rezk, and Peter Schwabe. High-assurance cryptography in the spectre era. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 1884–1901. IEEE, 2021.

[5] Jonathan Behrens, Adam Belay, and M. Frans Kaashoek. Performance evolution of mitigating transient execution attacks. In Yérom-David Bromberg, Anne-Marie Kermarrec, and Christos Kozyrakis, editors, *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*, pages 251–265. ACM, 2022.

[6] Nick Clifton. Spectre variant 1 scanning tool, Mar 2019.

[7] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting COTS binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1497–1511. IEEE, 2020.

[8] Cosmin Gorgovan, Amanieu D'Antras, and Mikel Luján. MAMBO: A low-overhead dynamic binary modification tool for ARM. *ACM Trans. Archit. Code Optim.*, 13(1):14:1–14:26, 2016.

[9] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: A fast and stealthy cache attack. DIMVA 2016, page 279–299, Berlin, Heidelberg, 2016. Springer-Verlag.

[10] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. Spectector: Principled detection of speculative information flows. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1–19. IEEE, 2020.

[11] 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. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1–19. IEEE, 2019.

[12] Congmiao Li and Jean-Luc Gaudiot. Challenges in detecting an "evasive spectre". *IEEE Computer Architecture Letters*, 19(1):18–21, 2020.

[13] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 2109–2122. ACM, 2018.

[14] Maria Mushtaq, Ayaz Akram, Muhammad Khurram Bhatti, Maham Chaudhry,

Vianney Lapotre, and Guy Gogniat. Nightswatch: a cache-based side-channel intrusion detector using hardware performance counters. In Jakub Szefer, Weidong Shi, and Ruby B. Lee, editors, *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy, HASP@ISCA 2018, Los Angeles, CA, USA, June 02-02, 2018*, pages 1:1–1:8. ACM, 2018.

[15] Maria Mushtaq, David Novo, Florent Bruguier, Pascal Benoit, and Muhammad Khurram Bhatti. Transit-guard: An os-based defense mechanism against transient execution attacks. In *26th IEEE European Test Symposium, ETS 2021, Bruges, Belgium, May 24-28, 2021*, pages 1–2. IEEE, 2021.

[16] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean M. Tullsen, and Deian Stefan. Swivel: Hardening webassembly against spectre. In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 1433–1450. USENIX Association, 2021.

[17] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In Jeanne Ferrante and Kathryn S. McKinley, editors, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 89–100. ACM, 2007.

[18] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. Specfuzz: Bringing spectre-type vulnerabilities to the surface, 2020.

[19] Zhenxiao Qi, Qian Feng, Yueqiang Cheng, Mengjia Yan, Peng Li, Heng Yin, and Tao Wei. Spectaint: Speculative taint analysis for discovering spectre gadgets. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021.

[20] Vijay Janapa Reddi, Alex Settle, Daniel A. Connors, and Robert S. Cohn. PIN: a binary instrumentation tool for computer architecture research and education. In Edward F. Gehringer, editor, *Proceedings of the 2004 workshop on Computer architecture education - Held in conjunction with the 31st International Symposium on Computer Architecture, WCAE@ISCA 2004, Munich, Germany, June 19, 2004*, page 22. ACM, 2004.

[21] Martin Schwarzl, Claudio Canella, Daniel Gruss, and Michael Schwarz. *Specfuscator: Evaluating Branch Removal as a Spectre Mitigation*, volume 12674 of *Lecture Notes in Computer Science*, pages 293–310. Springer, 2021.

[22] Ryota Shioya. Konata: instruction pipeline visualizer for onikiri2-kanata/gem5-o3pipeview formats. `https://github.com/shioyadan/Konata`, 2021.

[23] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead defense against spectre attacks via program analysis. *IEEE Trans. Software Eng.*, 47(11):2504–2519, 2021.

[24] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. Osiris: Automated Discovery of Microarchitectural Side Channels. In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 1415–1432. USENIX Association, 2021.

[25] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, l3 cache Side-Channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, August 2014. USENIX Association.