SpecTaint: Speculative Taint Analysis for Discovering Spectre Gadgets

Zhenxiao Qi
UC Riverside
zqi020@ucr.edu

Qian Feng
Baidu USA
fengqian@baidu.com

Yueqiang Cheng*
NIO Security Research
yueqiang.cheng@nio.io

Mengjia Yan
MIT
mengjiay@mit.edu

Peng Li
ByteDance
peng.li@bytedance.com

Heng Yin
UC Riverside
heng@cs.ucr.edu

Tao Wei
Ant Group
lenx.wei@antgroup.com

***Abstract*—Software patching is a crucial mitigation approach against Spectre-type attacks. It utilizes serialization instructions to disable speculative execution of potential Spectre gadgets in a program. Unfortunately, there are no effective solutions to detect gadgets for Spectre-type attacks. In this paper, we propose a novel Spectre gadget detection technique by enabling dynamic taint analysis on speculative execution paths. To this end, we simulate and explore speculative execution at system level (within a CPU emulator). We have implemented a prototype called SpecTaint to demonstrate the efficacy of our proposed approach. We evaluated SpecTaint on our Spectre Samples Dataset, and compared SpecTaint with existing state-of-the-art Spectre gadget detection approaches on real-world applications. Our experimental results demonstrate that SpecTaint outperforms existing methods with respect to detection precision and recall by large margins, and it also detects new Spectre gadgets in real-world applications such as Caffe and Brotli. Besides, SpecTaint significantly reduces the performance overhead after patching the detected gadgets, compared with other approaches.**

I. Introduction

Spectre has now become an emerging attack vector [29]–[31] that breaks down the isolation between processes. Spectre attacks have prompted widespread security concerns, since they can exploit critical vulnerabilities across the spectrum of different processor architectures, including those from Intel, AMD, and ARM. Any devices with these vulnerable processors could be leveraged to defeat the security protection in operating systems (OSs) [29], browsers [21] and infrastructures [45] built on these devices.

Spectre mitigation at the hardware level is challenging. To ensure the security of processors, SafeSpec [27] proposes shadow hardware structures designed for speculative execution, such that the microarchitectural state can also be discarded to avoid leakage through side channels. InvisiSpec [46] provides a speculative buffer that stores speculative loads. Thereby, speculative loads are invisible to the cache line state. While these techniques can mitigate Spectre attacks by disabling cache side channels, they are still at the design stage.

Software patching is another way to mitigate Spectre attacks. Software vendors cannot always assume their services are running on patched processors, especially for these services running on third-party cloud platforms. Fortunately, software patching provides vendors an opportunity to take control over the security of their products. When there is no effective remedy applied to the hardware, software vendors can utilize serialization instructions [13] to patch Spectre gadgets in software and protect their products.

However, it is nontrivial to find a Spectre gadget (a sequence of instructions that leaks information via speculative execution). It is impossible to blindly serialize all conditional branching instructions in a program, which will introduce a very high performance penalty. Many researchers propose various solutions to strike a balance between security and performance. Generally speaking, they try to find a set of potential Spectre gadgets in a program and only patch these gadgets to avoid high runtime overhead. Spectre V1 Scanner from RedHat [5] and MSCV Spectre 1 pass [41] search in binary for gadget patterns, and only patch gadgets that match predefined patterns. Tools like SPECTECTOR [23] and oo7 [43] conduct more advanced static analysis such as symbolic execution and taint analysis to detect Spectre gadgets. They are more accurate and generic than simple pattern matching. However, static analysis approaches are known to be imprecise, bringing high false positives and sometimes false negatives when detecting Spectre gadgets. Having realized the limitations in the static analysis, SpecFuzz [39] takes a dynamic analysis approach. It extends fuzzing to not only monitor the normal execution of a program but also simulate its speculative execution paths. It simulates speculative execution by inserting speculative execution logic into the original program at compile time, and detects Spectre gadgets when out-of-bound memory accesses are observed during fuzzing. However, to ensure high fuzzing throughput during simulation of speculative execution, its simulation logic is oversimplified, causing both high false positives and false negatives (see Section II-B for more detailed discussions).

In this paper, we would like to enable dynamic taint analysis to discover Spectre gadgets. We propose to conduct

*The main work was done when Yueqiang Cheng worked at Baidu Security.

Network and Distributed Systems Security (NDSS) Symposium 2021
21-25 February 2021, Virtual
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.24466
www.ndss-symposium.org

dynamic taint analysis on speculative execution paths and discover Spectre gadgets based on data flow patterns. Compared with syntax-based [5] and sanitizer-based approaches [39], our gadget pattern can capture the semantics of Spectre gadgets. As substantiated in the evaluation (Section VI), our semantic-based gadget pattern produces fewer false positives, compared with the existing works. Furthermore, static taint analysis suffers from the severe over-tainting and under-tainting issues due to imprecise memory alias analysis, incomplete control flow graph extraction, etc. As shown in our evaluation (Section VI), oo7 has poor precision and recall rates when analyzing real-world programs.

However, existing dynamic binary taint analysis platforms cannot detect Spectre gadgets because speculative execution is invisible to normal program execution. Therefore, we extend a dynamic taint analysis platform and instrument speculative execution logic on-the-fly. As a result, we can simulate speculative execution and capture data flow patterns on speculative paths for Spectre gadget discovery. To the best of our knowledge, we are the first to enable dynamic taint analysis on speculative paths for Spectre gadget detection.

We have implemented a prototype SpecTaint to demonstrate the efficacy of speculative taint analysis in detecting Spectre gadgets. We evaluated the performance of SpecTaint on our Spectre Samples Dataset and six real-world programs. The experimental results demonstrate that SpecTaint outperforms the baseline approaches in terms of the precision and recall by large margins. It also has reasonable runtime efficiency and reduces the performance overhead after patching detected gadgets by 73%, compared with the conservative hardening strategy. Besides, SpecTaint can detect new Spectre gadgets in real-world applications like Brotli [7] and Caffe framework [1].

We summarize our contributions as follows:

- We propose a dynamic speculative execution simulation platform that enables dynamic taint analysis on speculative execution paths. With the support of dynamic taint analysis, we deploy a semantic-based gadget detector that detects exploitable Spectre gadgets during the program execution.
- We build a synthetic dataset by inserting known Spectre gadgets into selected real-world programs. The dataset can be considered as a benchmark to help researchers evaluate their Spectre gadget detection tools. It will benefit the security community.
- We implement a prototype SpecTaint to demonstrate the efficacy of our approach and compare SpecTaint with state-of-the-art tools on the Spectre Samples Dataset and real-world programs. Our evaluation indicates that SpecTaint outperforms existing methods with respect to precision and recall with reasonable runtime efficiency. Moreover, SpecTaint discovered new Spectre gadgets that were not detected by the other tools. Besides, it significantly reduces the execution overhead after patching detected gadgets, compared with the other approaches.

II. Overview

In this section, we first walk through a motivating example to explain how a Spectre V1 gadget is exploited by attackers. Then we introduce the background and limitations of state-of-the-art tools that discover Spectre gadgets from binaries. Furthermore, we propose our approach and show the capabilities of our approach to address these limitations. Then we give a brief introduction about the overview and mechanism of SpecTaint. At last, we discuss the scope and assumptions of this work.

A. Motivating Example

Listing 1 shows a code snippet that can be exploited to launch a Spectre V1 attack [34].

```
1  void victim_function(size_t user_input)
2  {
3      ...
4      if(user_input < get_len(array1)){
5          secret = array1[user_input];  //RS: Read Secret
6          temp &= array2[secret * 256]; //LS: Leak Secret
7      }
8  }
9  int main()
10 {
11     victim_function(user_input);
12     return 0;
13 }
```

Listing 1: Code snippet containing Spectre gadgets.

In this example, the `if` statement is a sanity check that ensures the following array access is within a valid range. When evaluating the sanity check, the outcome of the branch at line 4 may take many CPU cycles to be determined (e.g., due to the delay caused by a load from the main memory). To avoid the performance penalty caused by this delay, the branch prediction unit (BPU) will predict the branch outcome, from where the instructions will be executed speculatively. An attacker can poison the branch predictor by feeding crafted inputs to the program to intentionally trick the BPU into making an expected prediction on that branch. Then the attacker can launch the Spectre attack by running the code with an out-of-bounds value as input. In this case, the BPU predicts the branch outcome to be true and the processor speculatively executes instructions at line 5 and line 6. Consequently, an arbitrary value can be read using an out-of-bounds index to access `array1` at line 5. Then another index related to the loaded secret is used to access `array2` and results in cache line state changes for `array2`. After the processor finds out the prediction to be wrong, it discards all architectural effects made by speculative instructions. However, side effects (e.g., cache line state changes) still remain at the micro-architectural level, and the attacker can launch a cache side-channel attack (e.g., `Flush+reload` [48]) to retrieve the secret value.

B. Background and Rationale

Speculative execution is a hardware feature that is invisible to the program execution. Therefore, to apply software-based program analysis techniques on detecting Spectre gadgets, the first step is to simulate speculative execution at the software level, which is also the principle for all related works. To this end, existing approaches utilize different methods to simulate speculative execution, either statically or dynamically. RH scanner [5] (also known as Spectre V1 scanner) is a static analysis tool. It simulates speculative execution by scanning

both targets of a conditional branch. By doing so, it at least covers one path that is not taken during real execution, which is considered as a speculative path. During scanning, it searches a certain pattern in the binary to detect Spectre gadgets. However, the syntax-based code pattern used by RH scanner could produce a large number of unexploitable candidates, since not all detected gadgets can be controlled by attackers.

Oo7 [43] also conducts a static binary analysis. The difference lies in that it conducts a semantic-based gadget detection. That is, it leverages the data flow analysis to construct a semantic code pattern and identifies the code snippet that not only satisfies predefined patterns but also can be controlled via user inputs. The intuition behind it is that if the gadgets can be influenced by user inputs, attackers can leverage carefully-constructed inputs to exploit these gadgets. To this end, oo7 utilizes static taint analysis to trace information flow from inputs. However, it is well-accepted that static taint analysis suffers from severe over-tainting and under-tainting issues due to imprecise memory alias analysis, inaccurate control flow graphs and call graphs, etc.

The aforementioned works examine speculative execution statically and deploy different program analysis techniques for Spectre gadget detection. However, the detection capabilities of these approaches inherit the limitations of static analysis techniques. Therefore, these techniques by design suffer from high false positives and false negatives.

SPECTECTOR [23] mathematically defines a semantic notion of security against speculative execution and develops an algorithm based on symbolic execution to prove the absence of speculative leaks. However, this approach inherits the bottlenecks of symbolic execution and has to sacrifice the soundness and completeness of analysis when analyzing large programs.

SpecFuzz [39] takes a fuzzing approach to dynamically detect Spectre gadgets. It exposes speculative execution to fuzzing by inserting the speculative execution logic into the program at compile time, and relies on random mutation of program inputs to detect speculative execution errors during program execution. To ensure high fuzzing throughput, its gadget detection logic is oversimplified. More specifically, it has the following limitations:

- **Simplistic Gadget Modeling.** To avoid high runtime overhead during fuzzing, SpecFuzz simply leverages a memory safety checker (e.g., AddressSanitizer) to detect out-of-bounds memory accesses and considers all out-of-bounds memory accesses to be potential Spectre gadgets. This modeling is oversimplified and error-prone. An out-of-bounds memory access during speculative execution may not be controlled via user inputs and does not necessarily constitute a Spectre V1 gadget. Our experiments in Section VI substantiate this claim.
- **Probabilistic Detection.** SpecFuzz detects Spectre gadgets by capturing out-of-bounds memory access errors during speculative execution. However, even if a true Spectre gadget is indeed exercised during fuzzing, it may or may not trigger any out-of-bounds memory access errors. SpecFuzz relies on random mutation of program inputs to trigger these errors. As a result, the Spectre gadget detection of SpecFuzz is probabilistic.
- **Flawed Exception Handling.** Exceptions are likely to occur during simulated speculative execution, because it executes a path which might not be expected. To deal with exceptions during speculative execution, again for simplicity, SpecFuzz stops the simulation immediately and restores the execution to a previously saved state. This exception handling is flawed, because it does not correctly simulate how the hardware actually behaves. In reality, when encountering an exception during speculative execution, the processor can continue the speculative execution until it is terminated [34]. Consequently, SpecFuzz might miss Spectre gadgets that are located after the exception-raising instruction, thereby causing false negatives.

In this work, we would also like to take a dynamic analysis approach to detect Spectre gadgets, to ensure high detection accuracy. We resort to independent test case generation systems (such as fuzzing and symbolic execution) to produce high-quality test cases to achieve high detection coverage. In order to achieve high detection accuracy in real-world software, we need to strike a balance between scalability and gadget detection fidelity. That is, our analysis must be able to cope with complex real-world software, and faithful enough to ensure high detection accuracy.

To this end, we propose to perform dynamic taint analysis on speculative execution paths, and conduct taint-based pattern checking to characterize Spectre gadgets at the semantic level. Essentially, by performing dynamic taint analysis on speculative execution paths, we can detect memory accesses that are dependent on the program input (which attackers can control) and may cause information leakage through cache side channels. This taint-based gadget pattern checking might not be as precise as SPECTECTOR [23], but our scheme is designed to be scalable and practical for detecting Spectre gadgets from real-world programs. It is more faithful than the one used in SpecFuzz [39], as it is deterministic (once the program inputs are determined) rather than probabilistic (relying on random mutation of program inputs to trigger errors). Our evaluation in Section VI shows that this taint-based approach is able to achieve much better detection accuracy than SpecFuzz, by paying more runtime overhead on speculative taint analysis. In other words, our trade-off between scalability and gadget detection fidelity is justified.

C. System Overview

Architecture. As illustrated in Fig. 1, we extend a system emulator to simulate speculative execution and enable dynamic taint analysis on top of it. Given a target program, we simulate speculative execution of the CPU by dynamically forcing the CPU emulator to execute code paths (speculative instructions) which will not be executed at normal execution. We also conduct tainting analysis on speculatively executed instructions to detect Spectre gadgets.

Workflow. Fig. 2 illustrates the workflow of SpecTaint. SpecTaint will go through two stages, the normal execution stage and speculative execution stage. At the normal execution stage, SpecTaint runs the target program with seeds generated by external fuzzers to explore as many execution paths as possible. SpecTaint will start to simulate speculative execution

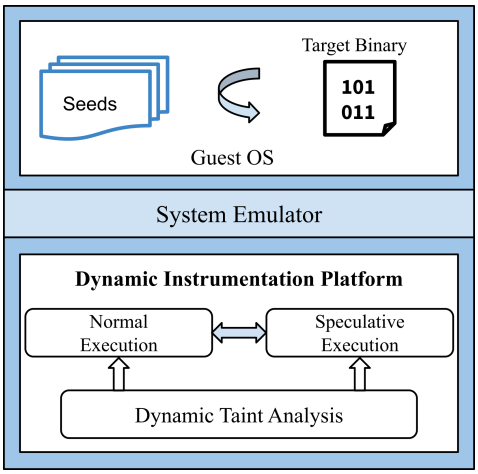

Fig. 1: Architecture of SpecTaint.

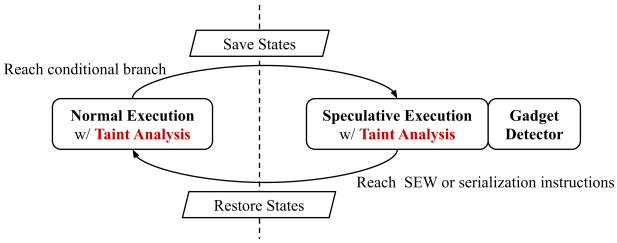

Fig. 2: Workflow of SpecTaint.

when encountering a conditional branch and backup the current execution state. When the speculative execution window (SEW) is reached, SpecTaint will roll back the execution state to the previously saved checkpoint. Like the rollback of speculative execution in the hardware, our simulated rollback also squashes all the side effects produced during speculative execution. At the speculative execution stage, a Spectre gadget detector conducts pattern checking on each speculative execution path to detect gadgets. To mitigate Spectre attacks, the reported gadgets are forwarded to automatic serialization tools (e.g., Speculative Load Hardening [18]).

Threat Model and Scope. We share the same threat model with other Spectre gadget detection tools [39], [43]. That is, our analyzed programs are benign but might be vulnerable. We do not deal with malicious code that might deliberately thwart or escape our analysis. Our approach can be used to analyze the exploit code for Spectre gadgets in a malware sample, but it is not our focus in this paper. In this work, we focus on detecting gadgets in victim programs that can be exploited by Spectre V1 attacks and leak sensitive data through cache side channels. Meltdown-type attacks do not require gadgets in victim programs and can be launched in malicious programs. Therefore, they fall out of the scope of this work. Our simulation focuses on Spectre V1 gadgets detection at the binary level. We do not simulate micro-operations, instruction execution including instruction fetching, decoding, etc., and irrelevant hardware structures such as the reorder buffer (ROB). We only use the size of ROB to calculate the speculative execution window (SEW).

III. Dynamic Speculative Execution Simulation

In this section, we introduce our speculative execution simulation platform for Spectre gadgets detection. Our platform extends the binary analysis platform DECAF [19], which is built on top of the system emulator QEMU [14]. Specifically, we will showcase how we utilize the system emulator to simulate speculative execution triggered by branch mispredictions and how we explore speculative paths in a depth-first manner.

A. Misprediction Simulation

Spectre V1 exploits out-of-bounds memory accesses in speculative execution triggered by one or more mispredictions of conditional branches. SpecTaint extends a system emulator to simulate this behavior. In the system emulator, `pc` stores the next instruction to be executed; in the case of a conditional jump, it stores the jump target address. To simulate the misprediction behavior of processors, SpecTaint inverts the direction of the conditional jump by replacing the `pc` with the untaken target address. As a result, it will execute the untaken path first at a conditional branch and enter the simulated speculative execution. Before it goes to the untaken path, a state checkpoint is saved, which contains the current state of the emulator, e.g., CPU registers. During the simulated speculative execution, any modification to the memory will be logged. Specifically, the original value will be saved before a speculatively executed instruction modifies it. When the speculative execution window is reached, the simulated speculative execution will be terminated and the control flow transfers back to the last saved checkpoint. To maintain the correctness of execution, the memory modifications will be restored by writing the original values back to memory and the CPU state will be restored with the saved state. As for other conditions to terminate speculative execution, we consider serialization instructions (e.g. `SYSCALL`, `LFENCE`, etc.) listed by Mambretti et al. [35] and terminate the simulated speculative execution when encountering these instructions.

Exception Handling. Exceptions can be very common during the simulation of speculative executions. This is because a CPU will execute a path which is not expected to execute. A straightforward way for simulation is to terminate the simulation and roll back whenever an exception is triggered, as done in SpecFuzz [39]. However, it could miss Spectre gadgets after the exception point along the speculative path, since speculative execution is not supposed to be terminated when hardware exceptions are raised in many CPU models [34]. The speculative execution is expected to continue even if an exception occurs. To guarantee a precise simulation, our approach does not roll back when an exception occurs, and instead forces it to silently simulate this exception instruction without raising any exception and continue speculative execution on the next instruction.

To simulate this behavior, we extend the exception handler of the system emulator. More specifically, we use a customized exception handler to capture errors in speculative execution. When an exception due to the permission violation (e.g., accessing kernel space from user-land) occurs during a simulated speculative execution, we will silently simulate this exception instruction without raising any exception and continue the user-land execution after the exception instruction. By doing

so, SpecTaint is able to continue the simulation even when exceptions occur. For other exceptions which SpecTaint is not able to handle (e.g., caused by invalid jump target or incomplete address translation) because the emulator does not know where the next instruction to be executed is, SpecTaint will make a state rollback and restore to the previously saved checkpoint.

B. Exploration of Speculative Execution Paths

The path exploration involves two path types, the normal execution (NE) path, and speculative execution (SE) path. To explore the NE paths, we feed seeds generated by fuzzing tools into the target program. When encountering a conditional branching instruction, SpecTaint redirects the execution to the untaken branch first to explore SE paths. Crucially, CPUs can perform nested speculative execution in a SE path. Since we center on Spectre V1 which exploits the misprediction of conditional branches, we take into account nested speculative executions triggered by conditional branches. To simulate this behavior, SpecTaint explores SE paths for each conditional branch in NE paths and SE paths within SEW. In other words, SpecTaint explores the speculative paths in a depth-first manner. During the exploration, SpecTaint monitors whether any termination condition is met, and restore to the last saved state if there is any.

Execution State Rollback. It is crucial to restore the execution state after the speculative execution is terminated, so as to maintain the correctness of program execution. An execution state includes the state of all the CPU registers and used memory regions. The execution state rollback has two operations, state backup and resume. For a conditional branch, SpecTaint first backups the current execution state before simulates speculative execution on that branch. When the SE simulation is terminated, SpecTaint resumes the backup state by resetting the current execution environment with the backup state. To backup an execution state, SpecTaint saves the emulator state, e.g., all registers. For memory, it is too heavyweight to save the entire memory, so we adopt a lightweight "copy-on-write" approach. More specifically, SpecTaint keeps track of memory regions that are modified during speculative execution. Before the memory regions are modified, SpecTaint saves the original values. When making a state rollback, SpecTaint writes the original values back to the memory in a reversed order as they are modified. Moreover, SpecTaint uses dynamic taint tracking to detect Spectre gadgets. Thus, it also restores taint information that is created during simulated speculative execution. More details are discussed in Section IV-A.

Path Exploration. The path exploration considers two types of path coverage, the NE and SE path coverage. The path coverage on normal execution is used to explore more paths during the normal execution. We utilize fuzzing techniques such as AFL [2] to improve path coverage on normal execution. The SE path coverage is to measure how many speculative paths are covered. The goal is to explore speculative execution comprehensively so as to avoid missing Spectre gadgets on uncovered speculative paths.

The switch point is the instruction that transfers the execution mode from NE to SE. SpecTaint treats a conditional branch as a switch point and conducts the SE simulation. Once entering the speculative execution mode, SpecTaint explores each conditional branch and its targets for the speculative exploration. The speculative path exploration algorithm is shown in Algorithm 1. We use pc to represent the current instruction, and $gadget\_set$ to store the locations of detected Spectre gadgets. The speculative execution path exploration is a depth-first traversal on the control flow graph of the program at the speculative execution mode. When exploring speculative paths, SpecTaint will first check whether the current execution path has reached the SEW limit w, or any other termination conditions, and if so, restore to the last saved state. If the current instruction at pc is a conditional branch, SpecTaint will simulate a misprediction by walking through both targets of this branch instruction. Before exploring each target, SpecTaint will first backup the state and push it into a stack $saved\_state$. After finishing the path exploration on a target (e.g., reaching the SEW limit), SpecTaint will resume the last saved state and continue for the next target. Along with the exploration, SpecTaint conducts the gadget pattern checking on the current instruction. If it matches, SpecTaint will save this gadget into $gadget\_set$ and continue exploration until termination conditions are met.

Algorithm 1: Speculative Path Exploration Algorithm.

```
Input  : Entry point: pc; SEW size: w; Backup state
         set: saved_state;
Output: gadgets: gadget_set
gadget_set ← ∅ ;
inst_count ← 0;
Function explorer(pc, w):
    while inst_count < w do
        if is_terminator(pc) then
            state = saved_state.pop();
            restore(state);
            pc = state.pc;
            explorer(pc, state.insn_count);
        end
        if is_branch(pc) then
            saved_state.push(checkpoint(pc));
            foreach t ∈ get_targets(pc) do
                explorer(t, w − insn_count);
            end
        end
        execute(pc);
        if gadget_checker(pc) then
            gadget_set ← pc;
        end
        pc ← next_pc;
        inst_count ++;
    end
    return;
```

Mitigating Path Explosion. As presented in our evaluation, the length of each speculative execution path is bounded by the SEW limit (see TableV). However, there can be a large number of paths within this SEW limit in the worst case. Therefore, we may still encounter a path explosion problem. According to the analysis of our evaluation dataset, we identify two kinds of cases where the exponential growth of paths may happen: *loops* and *recursive functions*. We propose our approach to

address the path explosion issue. That is, SpecTaint has a threshold on how many times simulation can happen on the same branch. In real hardware, it is unlikely that speculative execution can be triggered by the same branch repeatedly. After iterations, the same branch will not be mispredicted, since its memory value are very likely to be stored in registers or be cached after iterations [35]. As demonstrated by Kocher et al. [29], executing the same branch five times is enough to train the branch predictor. Therefore, we also set the threshold to five, and SpecTaint will not simulate speculative execution repeatedly on the same branch if it reaches the threshold. We admit that this approach might miss gadgets in paths we have not explored yet. More details about false negatives are discussed in Section VII.

IV. Spectre Gadget Detection

This section addresses how we detect Spectre gadgets during dynamic speculative execution simulation. More specifically, we first formalize the Spectre gadget definitions and then describe how we check the patterns to detect potential Spectre gadgets on speculative paths.

A. Dynamic Taint Tracking

If a variable is data-dependent on user inputs, we consider this variable is under an attacker's control. To find exploitable Spectre gadgets, it is essential to find gadgets that can be controlled through external inputs. Therefore, in order to capture this control relation, we utilize dynamic taint analysis to trace variables that are data-dependent on external inputs during execution. To this end, we label user inputs as taint sources and observe how the data flows from user inputs. Specifically, we utilize the whole-system dynamic taint analysis shipped with the platform we extend, DECAF [19], and perform taint propagation along with the execution of the program. DECAF's tainting rules have been formally verified to be sound (guarantee of no under-tainting at instruction level), and most of them have also been verified to be precise (guarantee of no over-tainting). The details are documented in this paper [24]. We conduct dynamic taint analysis on both NE and SE paths. Performing taint analysis on NE paths is to ensure the propagation of taint labels in normal program execution; performing taint analysis on SE paths is to facilitate the gadget checker to detect exploitable Spectre gadgets. When the SE path is terminated (e.g. when reaching SEW size), the CPU state and memory modifications will be restored, so as the taint information. When restoring to the previously saved state, we also clean the variables which are marked as tainted during the simulated speculative execution, in order to maintain the correctness of taint propagation.

B. Spectre Gadget Modeling

In this section, we formulate the gadget patterns for two types of Spectre V1 gadgets, Bounds Check Bypass (BCB) and Bounds Check Bypass Store (BCBS). To facilitate the discussion, we define the following notions before giving the Spectre gadget definitions.

- c is a conditional branch instruction.
- $\mathcal{T}(c)$ denotes a set of instructions in a speculative execution trace from c.
- $m(i)$ denotes i is a memory read instruction.
- $str(i)$ denotes i is a memory write instruction.
- $[i]$ denotes the memory value accessed by instruction i.
- $dep(i, j)$ denotes instruction i is data dependent on j.
- $t(i)$ denotes the operands of instruction i is tainted.
- δ denotes the size of speculative execution window.

BCB Gadget. In BCB attacks, the speculative load instruction is under an attacker's control, thus the attacker could read arbitrary values from memory. Then another load instruction is required to load the secret-indexed memory location with the intention of leaking the secret. To leak the secret through the cache side channel, the leak instruction has to use the loaded secret as the index to read from memory, thus the secret can be retrieved by monitoring the cache line state changes. In general, the BCB gadget involves a set of array operations and the index in the latter array operation is data-dependent on the value from the former array [34]. Essentially, the first array access is responsible for loading secrets, and the second one is responsible for leaking secrets. However, not all the code sequences matching such patterns are considered as a BCB gadget. The BCB gadget demands the index of the former array access should be under the attacker's control, such that the attacker could read arbitrary values from memory in the target program space by carefully manipulating the input values. To capture this control relation, we utilize dynamic taint analysis to track data flow from external inputs, which is discussed in IV-A. Suppose the speculative execution starts from a conditional branch c, and we formalize our BCB gadget pattern as $\Phi_{bcb}(c)$, and its definition is as follows:

$$\begin{aligned}\Phi_{bcb}(c) \coloneqq \exists i, j \in \mathcal{T}(c).m(i) \wedge m(j) \wedge dep(j, [i]) \\ \wedge t(i) \wedge |c, j| < \delta\end{aligned} \tag{1}$$

BCBS Gadget. Unlike BCB gadget, BCBS uses a speculative write (SW) to modify arbitrary memory locations. In BCBS attacks, attackers control the index of an array that would access out of boundary memory during speculative execution. As a result, attackers can modify an arbitrary memory location (e.g., return address) by manipulating the index of the array. We formalize our BCBS gadget pattern as $\Phi_{bcbs}(c)$, and its definition is as follows:

$$\Phi_{bcbs}(c) \coloneqq \exists i \in \mathcal{T}(c), str(i) \wedge t(i) \wedge |c, i| < \delta \tag{2}$$

In both patterns, we leverage dynamic taint analysis to determine whether the index of an array is under attackers' control or not. If an instruction i is tainted, we consider i is under attackers' control, and $t(i)$ is set to be true.

Gadget Classification. Our gadget patterns are similar to what is proposed in oo7 [43]. The primary difference lies in whether the branch instruction is tainted or not. Oo7 [43] considers that the branch should be tainted and controlled by the inputs. Thus attackers can poison the branch predictor to cause intentional misprediction of that branch by executing the victim program with carefully-constructed inputs. However, as discovered by Canella et al. [17], the branch prediction buffers

are shared and commonly indexed by the virtual address of the branch instruction, thus can be poisoned from another attack-controlled process by executing a congruent branch with the same virtual address. Based on this finding, Spectre gadgets can be exploited in the same address space (i.e., intra-process) or across address spaces (i.e., cross-process) [17]. Therefore, only considering tainted branches will exclude cross-process gadgets. Our gadget patterns cover both situations: if the speculative execution is triggered by a taint conditional branch, we will mark this gadget as an intra-process Spectre gadget; otherwise, this gadget will be marked as a cross-process Spectre gadget.

C. Gadget Detection

The gadget detection is to check whether the speculative instruction trace conforms to the gadget patterns described above. We deploy the gadget checker in simulated speculative execution. Although both our gadget patterns involve memory access operations, it is inefficient and unnecessary to check all the memory access instructions on SE paths. Instead, we only check memory access instructions which are tainted. To this end, we instrument memory read and write operations. To detect BCB gadgets, for each memory read, we check whether its source is tainted. To make sure the dependency $dep(j, [i])$ rule is satisfied between two instructions, we also track whether the tainted operand of one instruction j is propagated from instruction i. For BCBS gadgets, the pattern captures any tainted memory write whose destination address is marked as tainted within the SEW.

V. Implementation

We have implemented the prototype SpecTaint in C. More specifically, we wrote a C plugin of 1 KLOC in C code on top of DECAF [25] (a dynamic binary analysis platform built on top of QEMU 1.0). This plugin implements the state checkpoint management and Spectre gadget detection components. We reused the dynamic taint analysis plugin of DECAF for our taint analysis. Overall, the changes to develop our prototype do not exceed 2 KLOC. Besides, to increase the code coverage, we use AFL 2.52b [2] and honggfuzz [8] to generate seed inputs.

VI. Experimental Evaluation

In this section, we evaluate SpecTaint to answer the following research questions:

1) How effective is SpecTaint to find Spectre gadgets compared with other existing tools?
2) How efficient is SpecTaint to find Spectre gadgets in real-world applications?

This section is composed as follows: First, we briefly describe the experiment setup, the datasets, and the evaluation metrics used in our experiments (Sections VI-A). Second, we evaluate the efficacy of SpecTaint (Sections VI-B). Then, we evaluate the efficiency of SpecTaint on real-world applications (Sections VI-E). Finally, we conduct case studies of some Spectre gadgets detected by SpecTaint in real-world applications and the deep learning framework Caffe (Sections VI-F).

A. Experiment Setup

Baseline Methods. We compare SpecTaint with three baseline approaches: Spectre 1 Scanner from Red Hat (RH Scanner) [5], oo7 [43] and SpecFuzz [39]. RH Scanner is a static analysis tool that can be used to scan for Spectre gadgets. Oo7 [43] is another static analysis tool that utilizes static taint analysis to find Spectre gadgets. SpecFuzz [39] extends the fuzzing technique to detect errors in speculative execution and report Spectre gadgets. Another related work, SPECTECTOR, leverages symbolic execution to detect information-flow differences introduced by speculative execution. Since the manual settings to make it work on large programs are not open-source, it is hard to evaluate it on our real-world benchmarks. As discussed in SpecFuzz [39], the authors failed to run SPECTECTOR on the real-world benchmarks due to a large number of unsupported instructions. Therefore, we did not compare with SPECTECTOR on real-world benchmarks (same with SpecFuzz's).

Evaluation Dataset. The evaluation is conducted on two datasets, the Spectre Samples Dataset and Real-world Dataset. Aligned with other baseline works, we utilize the same Spectre Samples Dataset to demonstrate the detection capability of SpecTaint. For baseline comparison with related works, we collected six real-world applications and created two real-world datasets.

- **Spectre Samples Dataset.** This dataset is designed to demonstrate the efficacy of SpecTaint. We collected 15 Spectre V1 samples created by Paul Kocher [3] and compiled 15 samples with the same configuration (gcc-4.8.4 with O0) [39].
- **Real-world V1 Dataset.** For fair baseline comparison, we use the same dataset as SpecFuzz's [39]. This dataset contains six widely used applications: one cryptographic program from OpenSSL [12], a compression program (Brotli [7]), and four parsing programs (JSON [10], LibHTP [11], HTTP [9] and YAML [6]).
- **Real-world V2 Dataset.** Systematic baseline evaluation is difficult due to the shortage of ground truth in real-world programs. To solve this problem, we injected known Spectre gadgets into programs from Real-world V1 Dataset and created the Real-world V2 Dataset. We adopt the same injection approach proposed in LAVA [20] to build this dataset. More specifically, we utilize dynamic taint analysis to find attack points which can be controlled by input bytes that do not determine control flow and have not been modified much (see [20] for more details). Then we inject the Spectre gadgets from Spectre Sample Dataset into target programs and add code to make injected gadgets controllable via input bytes. As a result, we injected 15 Spectre V1 gadgets from Spectre Sample Dataset into 52 different locations in six programs. To make a fair comparison, the input seeds used in the evaluation are all generated by a fuzzing tool [8]. Therefore, we have no clue whether the injected locations are covered by input seeds in the evaluation. It is worth mentioning that this dataset is used to evaluate the detection coverage of SpecTaint and related works, instead of path/code coverage, and we choose the same seed corpus with SpecFuzz's to guarantee a fair comparison.

Evaluation Metrics. For the Real-world V1 dataset, which does not have ground truth, we manually verify the detection results and calculate the precision rate to quantify the performance of SpecTaint and baseline approaches. The precision is calculated as $precision(P) = \frac{TP}{TP+FP}$, where TP is the number of detected gadgets that are manually verified to be exploitable, and FP is the number of detected gadgets that are not exploitable based on manual analysis. For Real-world V2 Dataset, we only consider injected Spectre gadgets to be true positives and other reported results to be false positives. Then we calculate the precision and recall to quantify the effectiveness of the proposed approach and baseline methods. The recall is calculated as $recall(R) = \frac{TP}{TP+FN}$, where TP is the number of inserted gadgets which are correctly detected, and FN is the number of inserted gadgets that are missed. The precision is calculated as $precision(P) = \frac{TP}{TP+FP}$, where FP is the number of detected gadgets other than injected ones. To measure the efficiency, we reported runtime per speculative execution and number of paths explored per speculative execution along with other statistics (see Section VI-E for more details).

Configuration. The experiments were conducted on a desktop with 16 GB memory, Intel Core i7 12 cores at 3.70 GHz CPU, and running Linux 4.15. The Guest OS in QEMU is Ubuntu 14.04 with 1 GB memory. The speculative window is dependent on the space limit, i.e., ROB size and timing limits. We follow the configuration used by SpecFuzz [39] and also set the speculative window size to 250.

B. Baseline Evaluation on Spectre Samples Dataset

In this experiment, we compared SpecTaint with three baseline tools on the Spectre Sample Dataset. As presented in Table II, SpecTaint successfully detected all Spectre gadgets in the Spectre Samples Dataset, while RH Scanner relies on syntax-based pattern matching and missed three cases.

C. Baseline Comparison on Real-world V2 Dataset

We conducted the baseline comparison with three related works, RH Scanner, oo7, and SpecFuzz, on Real-world V2 Dataset. In this experiment, we focused on detecting the inserted gadgets, therefore we only consider inserted gadgets to be true positives and all other detection results to be false positives. For tools that utilize taint tracking to detect Spectre gadget (oo7 and SpecTaint), we mark input bytes as taint sources. Since the analysis of oo7 is very slow when performing whole input bytes tainting, we only mark input bytes that influence injected gadgets as taint sources. We adopt the same configuration for SpecTaint. To compare with RH Scanner and SpecFuzz, we have another configuration for SpecTaint, where we mark all input bytes as tainted. The results for the former configuration are labeled with "*" in Table I. For dynamic analysis tools (SpecTaint and SpecFuzz), we used an external fuzzing tool [8] to fuzz the six programs for 10 hours and fed the generated seeds as inputs to run the programs.

```
1     ...
2     if (parser->toknext >= num_tokens) {
3         // Inserted Spectre Gadget
4 #ifdef SPECTRE_VARIANT
5         if(global_idx < array1_size){
6             tmp &= array2[array1[global_idx] * 512];
7         }
8 #endif
9     ...
```

Listing 2: Missed Spectre gadget 1 by SpecFuzz.

```
1     ...
2     if (l->first + idx < l->max_size) {
3         return (void *) l->elements[l->first + idx];
4     } else {
5         // Inserted Spectre Gadget
6 #ifdef SPECTRE_VARIANT
7         int temp = 0;
8         int *addr = &global_idx;
9         if (*addr < array1_size) {
10            temp &= array2[array1[*addr] * 512];
11        }
12 #endif
13    ...
```

Listing 3: Missed Spectre gadget 2 by SpecFuzz.

Table I shows that, when tainting gadget-related input bytes, SpecTaint has no false positives and achieves a precision rate of 100%. Under the same configuration, however, oo7 still produced false positives. For instance, it reported 13 gadget candidates in LibHTP, but 12 of them are false positives. The results show that static taint analysis suffers from the over-tainting issue, thereby is hard to achieve high precision in detecting Spectre gadgets. As presented in Table I, oo7 has many false negatives. For example, it missed all inserted gadgets in YAML and Brotli. We examined these false negatives and found the reasons are as follows. Firstly, the detection results of oo7 [43] depend on the completeness of the control-flow graph (CFG) extraction. Some inserted gadgets are missed since it failed to extract a complete CFG due to the limitation of the static approach. Also, oo7 is limited by static inter-procedure taint tracking, and it missed many inserted gadgets because it failed to propagate the taint source to the injection points in the target programs. The evaluation results substantiate our claim that dynamic taint analysis is much more accurate and effective than static taint analysis in detecting Spectre gadgets.

As presented in Table I, SpecTaint outperforms RH Scanner and SpecFuzz under the whole input bytes tainting configuration. Since we only consider injected gadgets to be true positives in this dataset, other detection results are labeled as false positives. The analysis results of other gadgets reported by SpecTaint are presented in Table IV. Note that SpecTaint missed some injected gadgets in this experiment. We further investigated the results and found that missed gadgets are not covered by input seeds. However, SpecFuzz missed many injected gadgets that are covered by input seeds and detected by SpecTaint. According to our analysis, the reasons are first, SpecFuzz adopts a prioritized simulation of branch mispredictions; it selectively chooses whether to simulate the misprediction or not on a conditional branch. Therefore, it missed some injected gadgets. For example, as presented in Listing 2, the seeds are able to reach the branch at line 2, but SpecFuzz did not simulate branch misprediction over these

	RH Scanner						oo7					SpecFuzz					SpecTaint						
Program	GT	TP	FP	FN	Precision	Recall	TP	FP*	FN	Precision*	Recall	TP	FP	FN	Precision	Recall	TP	FP*	FP	FN	Precision*	Precision	Recall
JSMN	3	1	448	2	0.002	0.33	3	0	0	1.00	1.00	2	17	1	0.105	0.67	3	0	1	0	1.00	0.750	1.00
Brotli	13	2	811	11	0.003	0.15	0	0	13	N/A	0	7	43	6	0.140	0.54	12	0	17	1	1.00	0.141	0.92
HTTP	9	3	128	6	0.023	0.33	1	1	8	0.5	0.11	8	9	1	0.471	0.89	8	0	6	1	1.00	0.574	0.89
LibHTP	7	4	254	3	0.016	0.57	1	12	6	0.077	0.14	5	79	2	0.059	0.71	7	0	14	0	1.00	0.333	1.00
YAML	10	2	36	8	0.526	0.20	0	0	10	N/A	0	4	215	6	0.018	0.40	7	0	3	3	1.00	0.700	0.70
SSL	10	3	100	7	0.029	0.30	7	8	3	0.467	0.70	6	55	4	0.098	0.60	10	0	16	0	1.00	0.385	1.00

TABLE I: Evaluation Results on Real-world V2 dataset (GT: ground truth; TP: true positive; FP: false positive; FN: false negative). * means we only mark input bytes that influence injected gadgets as taint sources.

	Total #	RH Scanner	oo7	SpecFuzz	SpecTaint
Spectre	15	12	15	15	15

TABLE II: The number of detected Spectre gadgets from the Spectre Sample Dataset.

	JSMN	Brotli	HTTP	LibHTP	YAML	SSL
SpecFuzz	16	68	9	91	140	589
Reproduce	17	43	9	79	215	55
SpecTaint	1	17	6	14	3	16
Tainted branch	1	13	6	9	0	13
Unique	0	6	1	0	0	4

TABLE III: The number of detected gadgets in Real-world V1 Dataset. Reproduce shows the results that were reproduced using the open-sourced SpecFuzz implementation. Tainted branch means the gadgets are detected during speculative execution over tainted branches. Unique means the gadgets are detected only by SpecTaint.

two conditional branches, and thus missed the inserted gadget at line 5. Second, SpecFuzz stops the simulation and rolls back to a previously saved state once an invalid memory access or other exceptions are captured. Thus, it will miss gadgets located after the invalid memory access or exception. For example, as presented in Listing 3, SpecFuzz detected an out-of-bounds access at line 3, then it stopped the simulated speculative execution and restored. As a result, it failed to detect the inserted gadget in the "else" branch.

Table I also shows that RH Scanner missed many inserted gadgets. This is because it failed to explore the execution paths of target programs due to the limitation of the static path exploration it uses. It also produced a large number of detection results due to the simplistic syntax-based gadget modeling. In this experiment, since there are too many results reported by RH Scanner and we only focus on inserted gadgets, we did not inspect all detection results and only consider inserted gadgets to be true positives. In conclusion, the evaluation results show that SpecTaint outperforms state-of-the-art tools in terms of precision and recall when analyzing real-world programs.

D. Baseline Evaluation on Real-world V1 Dataset

We further conducted a baseline comparison with another dynamic analysis approach SpecFuzz [39]. In this experiment, we evaluate two aspects of dynamic Spectre gadget detection tools. The first is the effectiveness, which aims at comparing the number of detected gadgets. Since there is no ground truth in this dataset, we manually analyze detected gadgets and consider exploitable gadgets to be true positives based on human knowledge. The second is efficiency. This is to evaluate the performance of gadget detection tools and runtime performance of target programs after patching detected gadgets.

	SpecFuzz			SpecTaint		
Program	TP	FP	Precision	TP	FP	Precision
JSMN	1	16	0.059	1	0	1
Brotli	5	38	0.116	11	6	0.647
HTTP	2	7	0.222	2	4	0.333
LibHTP	3	76	0.038	3	11	0.214
YAML	0	215	0	0	3	0
SSL	2	53	0.036	6	10	0.375

TABLE IV: True positive, false positive and precision of gadgets detected by SpecFuzz and SpecTaint on Real-world V1 dataset.

```
1 ...
2 for (k = 1; k < width; k ++)
3 {
4     octet = parser->raw_buffer.pointer[k];
5     value = (value << 6) + (octet & 0x3F);
6 }
7 ...
```

Listing 4: One false positive in YAML reported by SpecFuzz.

Effectiveness. We evaluated the detection effectiveness on the Real-world V1 Dataset, six original programs without gadget instrumentation. We run SpecTaint and SpecFuzz on these programs and manually analyze how many Spectre gadgets are detected in these programs. To make a fair comparison, we first reproduced similar detection results as reported in the SpecFuzz paper, and used the same seed corpus to run SpecTaint. More specifically, we ran SpecFuzz on each program for 10 hours (single-thread mode), collected the detection results and generated seeds; then we used the same seeds to run SpecTaint on six applications. The number of detected gadgets is presented in Table III. Note that for OpenSSL, we were not able to reproduce similar results, so we listed it as a reference. For the rest, we were able to reproduce similar results as presented in their paper [39].

As presented in Table III, SpecTaint detected fewer gadgets than SpecFuzz did. Then we manually investigated each of the detected gadgets and determined true positives and false positives. As listed in Table IV, SpecTaint outperforms SpecFuzz in terms of precision rate. Although SpecFuzz reported a large number of gadget candidates, most of them are false positives according to our manual inspection. For example, some candidates reported by SpecFuzz only contain one out-of-bound access, because SpecFuzz considers a memory error to be a sign of a Spectre gadget. Furthermore, some false

positives are not data dependent on input bytes, as presented in Listing 4, so they can not be controlled to read and leak arbitrary values. We also found that some candidate gadgets are under loops or constant comparison statements, where speculation can be resolved shortly. Therefore, these gadgets are considered false positives. This is also why SpecTaint produces false positives (see more discussion in Section VII). As presented in Table IV, SpecTaint has significantly higher precision than SpecFuzz. For instance, SpecFuzz and SpecTaint detected three true positives for LibHTP, but SpecTaint only has 11 false positives, while SpecFuzz reported 76 false positives. Also, SpecFuzz missed some true positives that are reported by SpecTaint. For instance, SpecTaint reported 11 true positives for Brotli, and six of them are not detected by SpecFuzz (labeled as unique in Table III). One explanation is that the gadget detection of SpecFuzz is probabilistic, and generated seeds did not trigger memory errors when executing those gadgets, thus they were missed by SpecFuzz's sanitizer-based gadget detector. This also substantiates that the taint-based gadget pattern checking of SpecTaint is deterministic because it detects any executed gadgets as long as they satisfy the pre-defined gadget patterns.

```
1
2  static BROTLI_INLINE BrotliDecoderErrorCode
       ProcessCommandsInternal(
3      int safe, BrotliDecoderState* s) {
4      ...
5      // transform_idx is tainted
6      if (transform_idx < (int)transforms->num_transforms)
       {
7        const uint8_t* word = &words->data[offset];
8        int len = i;
9        if (transform_idx == transforms->cutOffTransforms
       [0]) {
10         memcpy(&s->ringbuffer[pos], word, (size_t)len);
11       } else {
12         len = BrotliTransformDictionaryWord(&s->
       ringbuffer[pos], word, len, transforms,
       transform_idx);
13       }
14       ...
15     }
16     ...
17 }
18 // read and leak secret using transform_idx as index.
19 #define BROTLI_TRANSFORM_PREFIX_ID(T, I) ((T)->transforms
       [((I) * 3) + 0])
20 #define BROTLI_TRANSFORM_PREFIX(T, I) (&(T)->
       prefix_suffix[(T)->prefix_suffix_map[
       BROTLI_TRANSFORM_PREFIX_ID(T, I)]])
21
22 int BrotliTransformDictionaryWord(uint8_t* dst, const
       uint8_t* word, int len,
23     const BrotliTransforms* transforms, int transform_idx)
         {
24     int idx = 0;
25     const uint8_t* prefix = BROTLI_TRANSFORM_PREFIX(
       transforms, transform_idx);
26     ...
27 }
```

Listing 5: Speculative BCB gadget found in Brotli.

Performance Overhead after Patching. Afterward, we applied a serialization tool to patch all reported gadget locations in six programs and then compared the performance of six programs after patching the reported gadgets. Specifically, we used a modified version of Speculative Load Hardening (SLH) [18] shipped with SpecFuzz and patched the programs. We only patched the gadgets reported by SpecFuzz and SpecTaint, and compared the runtime performance with fully hardened programs (patching all conditional branches). In this experiment, we used benchmarks shipped with the programs, if available, as well as test benchmarks provided by SpecFuzz. Figure 3 shows the comparison results. As we can see, patching the gadgets detected by SpecTaint introduces negligible overhead, compared with patching gadgets detected by SpecFuzz and full hardening. On average, the performance overhead was reduced by 55% compared with SpecFuzz's patching and 73% compared with full hardening. As presented in Table III, SpecTaint produced fewer gadget candidates than SpecFuzz. Therefore, the runtime overhead introduced by patching those gadgets were reduced greatly. We also found that in some cases SLH patching introduces large runtime overhead. For instance, SpecFuzz found three more gadget candidates in HTTP, but the performance slowdown caused by these three candidates is almost 60%. It is because these three candidates are located on hot paths that are exercised frequently. This also demonstrates the importance of high precision in gadget detection.

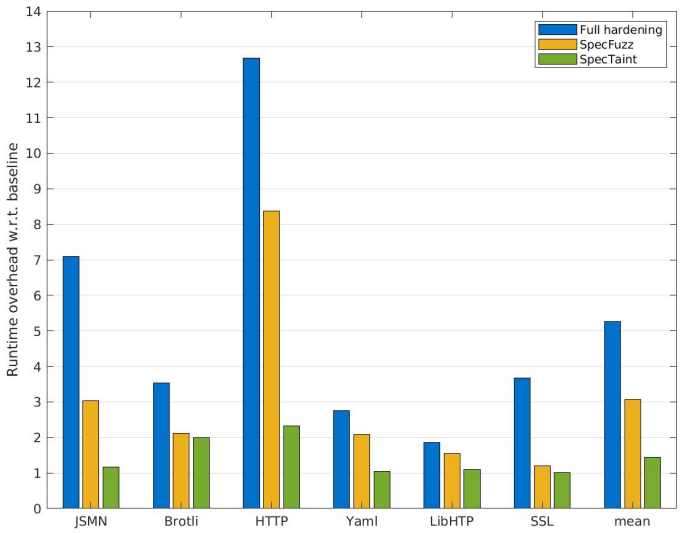

Fig. 3: The performance overhead after patching w.r.t native.

E. Efficiency Evaluation

In this experiment, we evaluated the runtime performance of SpecTaint on the six real-world applications. Since the workflow of SpecTaint is fundamentally different from SpecFuzz, it is not easy to have an end-to-end runtime comparison. SpecFuzz extends the fuzzing technique and its fuzzing procedure and gadget detector are closely coupled. While SpecTaint is a detection tool and can receive test cases from fuzzers. In this evaluation, we marked the user inputs as taint sources and simulate speculative execution over tainted branches. Note that SpecTaint is able to simulate speculative execution on every conditional branch. The runtime of simulating speculative over tainted branches is reasonable to reflect the efficiency because, as presented in Table III, around 80% of detected gadgets are enclosed by tainted branches (intra-process gadgets IV-B).

We collected several statistical results including the total number of executed branches at the normal mode, the

Lib Name	# of Seeds	Binary Size	Analysis Time(h)	# of Branch	# of Tainted Branch	# of Nested Branch	Path/Branch	Time/Path(ms)
JSMN	6204	17K	10.75	554017	178542	5498472	31	7.0
Brotli	100	440K	0.65	33912	8900	274948	31	8.6
HTTP	944	85K	5.24	184044	22531	94477	5	200.0
LibHTP	155	549K	0.4	108101	29836	423263	14	3.4
YAML	2525	251K	6.84	5311472	604917	30303111	50	0.8
SSL	46	11M	17.5	122694	56914	189332	4	432.0

TABLE V: Runtime Performance Results on real-world applications.

total number of tainted branches for speculative execution simulation, the total number of nested branches explored in speculative execution mode, and the number of speculative execution paths on average to be explored. We also collected the execution time for each simulated speculative execution path. The execution time includes the taint analysis, path exploration time, and gadget detection time.

Since SpecTaint extends a whole-system emulation platform and performs dynamic taint analysis, by design it pays more runtime overhead for speculative taint analysis. However, as presented in Table V, we can see that SpecTaint is able to analyze large programs within a reasonable amount of time. For example, for Brotli, SpecTaint finished 8,900 branches as switch points for speculative execution simulation within 40 minutes. That means that SpecTaint can finish the SE path exploration including state management, taint analysis, and pattern checking for one switch point within 0.04s. Besides, SpecTaint has reasonable analysis time. It can finish the analysis within a few hours for most of the programs; only JSMN and SSL exceed 10 hours. In fact, SSL is substantially more complex than the other programs, and running it in the emulator is already very slow. On average, SpecTaint takes around 6.8 hours to analyze one program. Compared with SpecFuzz, which takes 10 hours to reproduce the results presented in Table III, the analysis time in Table V suggests that SpecTaint can achieve precise simulation of speculative execution and perform dynamic speculative taint analysis without introducing much overhead.

F. Case Study

As presented in Table III, SpecTaint discovered 11 new Spectre gadgets that were not detected by SpecFuzz [39]. After manual inspection, we confirmed that ten of them are exploitable and one gadget is considered a false positive (not exploitable, see VII). In this section, we showcase one detected Spectre gadget from Real-world V1 Dataset due to the page limitation. To further demonstrate the capability of SpecTaint, we present one Spectre gadget detected by SpecTaint from a well-known machine learning framework, Caffe [1].

Speculative BCB in BROTLI. Brotli is a generic-purpose lossless compression program. SpecTaint found an exploitable Spectre gadget in function *ProcessCommandsInternal* and *BrotliTransformDictionaryWord*. Listing 5 presents the relevant code snippets. Before calling the function *BrotliTransformDictionaryWord* at line 11, it first checks whether `transform_idx` is less than `num_transforms` to avoid potential overflow. In function *BrotliTransformDictionaryWord*, it uses `transform_idx` as index to perform two memory accesses using a macro *BROTLI_TRANSFORM_PREFIX* (load a value from an array using `transform_idx`, then uses the loaded value as index to read another array). If the branch at line 5 was mispredicted during speculative execution, *BROTLI_TRANSFORM_PREFIX* would perform out-of-bound memory access and leak the loaded value via converted cache side channels.

In this example, three properties make this Spectre gadget exploitable. Firstly, `transform_idx` is marked as tainted, which is propagated from user inputs. This means the attacker can control its value by manipulating the input. Secondly, computing the branch outcome at line 5 may take hundreds of CPU cycles, when `transforms->num_transforms` is not in cache and needs to be fetched from memory. Thus it opens a large speculative execution window and allows the execution of the following gadget during speculative execution. Finally, the macro *BROTLI_TRANSFORM_PREFIX* loads the out-of-bound value using `transform_idx` as an index, then use the loaded value as an index to access another array; it first reads potential secret via an out-of-bound memory access and leaks the secret by using the secret as an index to access another array. Thus, the attacker can retrieve the secret via the cache side channel. As mentioned, this gadget is newly detected by SpecTaint. Although it is hard to find the reason why SpecFuzz missed this gadget, we speculate that SpecFuzz failed to detect it due to incomplete speculative execution simulation. As discussed earlier, SpecFuzz favors fuzzing throughput by adopting a lightweight speculative execution simulation strategy, which selectively overlooks some speculative paths.

```
1  template <typename Dtype>
2  void Solver<Dtype>::UpdateSmoothedLoss(Dtype loss, int
       start_iter,
3      int average_loss) {
4      if (losses_.size() < average_loss) {
5          losses_.push_back(loss);
6          int size = losses_.size();
7          smoothed_loss_ = (smoothed_loss_ * (size - 1) +
       loss) / size;
8      } else {
9          int idx = (iter_ - start_iter) % average_loss;
10         smoothed_loss_ += (loss - losses_[idx]) /
       average_loss;
11         losses_[idx] = loss;
12     }
13 }
```

Listing 6: Speculative Bounds Check Bypass (on Stores) gadget found in Caffe framework.

Caffe (v1.0). We also show a typical bounds check bypass on store gadget in Listing 6. This gadget is discovered in the Caffe framework using CIFAR-10 as the input model. We found the gadget during the model training process. In this experiment, we focus on the coverage of the internal code invoked by the test model, rather than the precision

of the model itself. Thus, we trained the CIFAR-10 model with the CIFAR-10 dataset consisting of 60000 images in 10 classes. The function `UpdateSmoothedLoss` is defined in `src/caffe/solver.cpp` and is frequently called during the training process. The branch outcome at *line* 4 depends on the value from `losses_.size()`, which may take hundreds of cycles to compute. In this case, the CPU will predict the branch target and continue to execute speculatively. The attackers can carefully poison the branch predictor into mispredicting the direction of the branch. This may result in the CPU making a wrong speculative prediction over this branch, and the value of `idx` will be greater than its value in normal execution. As a result, the `idx` would reach out of the array boundary, and a boundary check bypass on store would happen at *line* 11.

VII. Discussion

In this section, we discuss the limitations of SpecTaint and possible remedies.

```
1  static BrotliDecoderErrorCode ReadSymbolCodeLengths(
2      uint32_t alphabet_size, BrotliDecoderState* s) {
3      ...
4      code_len = BROTLI_HC_FAST_LOAD_VALUE(p);
5      if (code_len < BROTLI_REPEAT_PREVIOUS_CODE_LENGTH) {
6          ProcessSingleCodeLength(code_len, &symbol, &
       repeat, &space,
7              &prev_code_len, symbol_lists, code_length_histo
       , next_symbol);
8      }
9      ...
10 }
11 static BROTLI_INLINE void ProcessSingleCodeLength(
       uint32_t code_len,
12     uint32_t* symbol, uint32_t* repeat, uint32_t* space,
13     uint32_t* prev_code_len, uint16_t* symbol_lists,
14     uint16_t* code_length_histo, int* next_symbol) {
15     *repeat = 0;
16     if (code_len != 0) {
17         symbol_lists[next_symbol[code_len]] = (uint16_t)
       (*symbol);
18         next_symbol[code_len] = (int)(*symbol);
19         *prev_code_len = code_len;
20         *space -= 32768U >> code_len;
21         code_length_histo[code_len]++;
22     }
23     (*symbol)++;
24 }
```

Listing 7: A false positive in Brotli detected by SpecTaint.

False positives in detection. In this section, we discuss the cases where detected gadgets are not exploitable. First, when the triggering instruction can be resolved shortly, e.g., loops and constant comparison, the following gadgets cannot be executed before the speculation is resolved. However, in this case, SpecTaint would simulate speculative execution with the default SEW and detect the following gadgets that satisfy the pre-defined pattern.

Second, when the triggering instruction opens a large SEW (e.g., due to a cache miss), SpecTaint may produce false positives in some cases. For instance, SpecTaint detects a Spectre gadget in Brotli shown in Listing 7. In this example, the `code_len` is tainted, which is propagated from user inputs. If the branch at line 4 was mispredicted and the CPU speculatively executes the function `ProcessSingleCodeLength`, it would pass an out-of-bounds `code_len` to function `ProcessSingleCodeLength`. At line 17, the out-of-bounds `code_len` is used as an index to access the array `next_symbol`; the loaded value is further used as an index to access another array `symbol_list`. Thus, the attack can retrieve the out-of-bound value by monitoring the cache state. In practice, this gadget is hard to exploit to launch a Spectre V1 attack because the out-of-bound access at line 17 uses `code_len` as an index. By the time `code_len` is available, the conditional branch at line 5 has already been resolved, which means the speculative execution is terminated. According to our pattern checking policy, SpecTaint still treats it as a Spectre gadget.

To guarantee a low false negative rate, we conservatively assume the maximum values for CPU optimization parameters such as ROB limit. This means our detected gadgets may not be exploitable in some processor models.

Incomplete path coverage. Like other dynamic analysis tools, our approach is also bounded by the quality of test cases. In other words, our approach would fail to detect gadgets in uncovered paths. We can leverage state-of-the-art fuzzers [2], [8] to increase path coverage. Moreover, SpecTaint can be combined with other static Spectre gadget detection tools [5], [43] and test the uncovered paths to improve coverage. To ensure security, all the uncovered paths can be hardened conservatively for security-sensitive projects.

Control dependent attacks. Our Spectre gadget detection depends on dynamic taint analysis that keeps track of direct data flows. It will not detect Spectre gadgets that are control dependent on user inputs. Compared with the gadgets that are controlled via direct data-floe dependency, the control-dependent gadgets often are of limited attack capabilities and are currently beyond the scope of this work. We leave the investigation of this kind of gadgets as future work.

VIII. Related Work

We have discussed related works closely throughout the paper. In this section, we briefly survey additional related works. We focus on gadget detection techniques. Many other approaches aim at designing the hardware architecture to defeat the transient execution vulnerabilities [36], [47], [49]. Since they are orthogonal to our approach, we will only briefly discuss these approaches in this section.

Transient Execution Attack Variants. Transient execution attacks include Spectre-type attacks and Meltdown-type attacks in general. Spectre-type attacks can be categorized into Spectre-PHT [29], Spectre-BTB [29], [31], Spectre-RSB [30], [34] and Spectre-STL [17], [37]. They focus on exploiting different hardware caches. For example, Spectre-PHT attacks poison the Pattern History Table (PHT) to trigger speculative execution. Spectre-BTB exploits the Branch Target Buffer (BTB). Meltdown-type attacks usually exploit fault-handling exceptions such as virtual memory exception [16], [32], [45], or an exception reading a disabled or privileged register [26], [42]. Aligned with other tools [23], [39], we only focus on detecting gadgets in victim programs that can be exploited by Spectre V1 attacks and leak sensitive data through cache side channels.

Spectre Gadget Detection. Spectre gadget detection can be categorized as static and dynamic detection techniques. One direction of the static analysis technique is to model the Spectre gadget by using its syntax pattern, such as Spectre 1 Scanner from RedHat [5] and MSCV Spectre 1 pass [41], and conduct the pattern search on binaries for potential candidates. These tools produce a large number of false positives. Furthermore, their approaches are not generic and only designed for gadgets with special patterns. Another direction is to explore a more precise modeling by using symbolic execution or static taint analysis to detect Spectre gadgets. These approaches are more reliable and generic [23], [43], but they are still bounded by limitations of the static analysis. For example, oo7 utilizes a static tainting analysis to capture the data-dependency of Spectre gadget for detection. However, it inherits the limitation of static taint analysis, such as over-tainting and under-tainting issues. SPECTECTOR [23] proposes to use symbolic execution to automatically prove speculative non-interference, or to detect violations. However, it inherits the limitations of symbolic execution and has to sacrifice soundness and completeness of analysis when analyzing large programs. SpecFuzz [39] extends the fuzzing technique and detect memory errors during simulated speculative execution. However, to ensure high fuzzing throughput, the simulation logic is over-simplified, resulting in poor precision and recall. Compared with these approaches, SpecTaint is designed to provide a more precise detection approach that is also scalable for detecting Spectre gadgets from real-world programs.

Mitigation and Defense. Intel proposed hardware fixes [44] including improved process and privilege-level separation, but they are only designed for Spectre 2.0. ConTExT [36] provides a new architecture design using a temporary buffer to mitigate information leakage during speculative execution. It relies on users to identify the confidential information and perform dynamic taint analysis on hardware to keep track of the confidential information. Other approaches [28], [46] either proposed to isolate the cache side channel or provide a speculative buffer as a temporary buffer to mitigate the cache leakage, but they are still at the design stage. At the system level, the kernel page-table isolation is proposed to mitigate Meltdown attack [22]. Many approaches [4], [18] start to add mitigation instructions (serializing instructions or mitigation instructions) at the compiler time to mitigate Spectre attacks. Since SpecTaint can effectively provide more precise Spectre gadget candidates, it can greatly reduce the number of instructions for mitigation insertion. Therefore, SpecTaint can improve the runtime performance after patching, which has been substantiated in Section 3.

Dynamic Analysis. PIN [33], DynamoRIO [15], and Valgrind [38] are powerful dynamic instrumentation tools. In fact, our approach can also be implemented on these platforms. Xforce [40] is the first tool to propose the idea of force execution, but Xforce is used for code coverage based exploration, and it is not designed for the speculative execution simulation. Our approach is inspired by their approach and builds the unique features for the speculative execution simulation that enables dynamic taint analysis on speculative paths.

IX. Conclusion

In this paper, we enable dynamic taint analysis for Spectre V1 gadget detection. To this end, we present a system-level approach to simulate and explore the speculative execution and provide fine-grained gadget patterns for precise gadget detection. We have implemented a prototype SpecTaint to demonstrate the efficacy of our proposed approach. We evaluated the effectiveness of SpecTaint on our Spectre Samples Dataset and real-world programs. Our experimental results demonstrate that SpecTaint outperforms the existing methods with reasonable runtime efficiency, and it discloses new Spectre V1 gadgets from real-world applications.

Availability

The source code of SpecTaint and the dataset used in the evaluation can be found via https://github.com/bitsecurerlab/SpecTaint.git.

Acknowledgement

We would like to thank the anonymous reviewers for their valuable suggestions and comments. This work was supported by the Office of Naval Research under Award No. N00014-17-1-2893. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] Caffe. https://caffe.berkeleyvision.org/.

[2] American Fuzzy Lop (2.52b). http://lcamtuf.coredump.cx/afl/, 2011.

[3] Spectre Mitigations in Microsoft's C/C++ Compiler. https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html, 2018.

[4] Spectre mitigations in MSVC. https://blogs.msdn.microsoft.com/vcblog/2018/01/15/spectre-mitigations-in-msvc/, 2018.

[5] SPECTRE Variant 1 scanning tool. https://access.redhat.com/blogs/766093/posts/3510331, 2018.

[6] LibYAML. https://pyyaml.org/wiki/LibYAML, 2019.

[7] Brotli. https://brotli.org, Accessed: June 2020.

[8] Honggfuzz. http://honggfuzz.com/, Accessed: June 2020.

[9] HTTP. https://github.com/nodejs/http-parser, Accessed: June 2020.

[10] JSMN. https://github.com/zserge/jsmn, Accessed: June 2020.

[11] LibHTP. https://github.com/OISF/libhtp, Accessed: June 2020.

[12] OpenSSL. https://www.openssl.org/, Accessed: June 2020.

[13] Intel. https://xem.github.io/minix86/manual/intel-x86-and-64-manual-vol3/o_fe12b1e2a880e0ce-273.html, cited by 2019.

[14] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *In Proceedings of the USENIX Annual Technical Conference (ATC '05)*, ATC, 2005.

[15] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, 2003.

[16] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[17] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[18] C Carruth. Speculative load hardening. https://docs.google.com/document/d/1wwcfv3UV9ZnZVcGiGuoITT61eKo3TmoCS3uXLcJR0/edit#heading=h.phdehs44eom6., 2018.

[19] Ali Davanian, Zhenxiao Qi, Yu Qu, and Heng Yin. Decaf++: Elastic whole-system dynamic taint analysis. In *the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID), September 2019.*, RAID, 2019.

[20] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. Lava: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121, 2016.

[21] J. Fustos and H. Yun. Spectrerewind: A framework for leaking secrets to past instructions. *arXiv, 2003.12208*.

[22] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. Kaslr is dead: Long live kaslr. In *the 9th International Symposium on Engineering Secure Software and Systems (ESSoS'17)*, 2017.

[23] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. SPECTECTOR: principled detection of speculative information flows. *CoRR*, abs/1812.08639, 2018.

[24] A. Henderson, L. K. Yan, X. Hu, A. Prakash, H. Yin, and S. McCamant. Decaf: A platform-neutral whole-system dynamic binary analysis platform. *IEEE Transactions on Software Engineering*, 43(2):164–184, 2017.

[25] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. Make it work, make it right, make it fast: Building a platform-neutral whole-system dynamic binary analysis platform. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA, 2014.

[26] Intel. Q2 2018 speculative execution side channel update. 2018.

[27] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2019.

[28] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer. Dawg: A defense against cache timing attacks in speculative execution processors. 2018.

[29] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *CoRR*, abs/1801.01203, 2018.

[30] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *Proceedings of the 12th USENIX Conference on Offensive Technologies*, WOOT'18, 2018.

[31] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[32] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[33] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, 2005.

[34] Giorgi Maisuradze and Christian Rossow. Ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security(CCS'18)*, 2018.

[35] Andrea Mambretti, Matthias Neugschwandtner, Alessandro Sorniotti, Engin Kirda, William Robertson, and Anil Kurmus. Speculator: A tool to analyze speculative execution attacks and mitigations. In *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC '19, page 747–761, New York, NY, USA, 2019. Association for Computing Machinery.

[36] Claudio Canella Robert Schilling Florian Kargl Daniel Gruss Michael Schwarz, Moritz Lipp. Context: A generic approach for mitigating spectre. *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS'20)*, 2020.

[37] Marina Minkin, Daniel Moghimi, Moritz Lipp, Michael Schwarz, Jo Van Bulck, Daniel Genkin, Daniel Gruss, Frank Piessens, Berk Sunar, and Yuval Yarom. Fallout: Reading kernel writes from user space. *CoRR*, 2019.

[38] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, 2007.

[39] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. Specfuzz: Bringing spectre-type vulnerabilities to the surface. *CoRR*, abs/1905.10311, 2019.

[40] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. X-force: Force-executing binary programs for security applications. In *23rd USENIX Security Symposium (USENIX Security 14)*, 2014.

[41] Read Sprabery, Konstantin Evchenko, Abhilash Raj, Rakesh B. Bobba, Sibin Mohan, and R H Campbell. Scheduling, isolation, and cache allocation: A side-channel defense. 2018.

[42] Julian Stecklina and Thomas Prescher. Lazyfp: Leaking FPU register state using microarchitectural side-channels. *CoRR*, abs/1806.07480, 2018.

[43] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead defense against spectre attacks via binary analysis. *CoRR*, abs/1807.05843, 2018.

[44] T. Warren. Intel processors are being redesigned to protect against spectre. 2018.

[45] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. Foreshadow-ng: Breaking the virtual memory abstraction with transient out-of-order execution. Technical report, 2018.

[46] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy. *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[47] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel atacks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, 2017.

[48] 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.

[49] Si Yu, Xiaolin Gui, and Jiancai Lin. An approach with two-stage mode to detect cache-based side channel attacks. In *Proceedings of the 2013 International Conference on Information Networking (ICOIN)*, ICOIN '13, 2013.