oo7: Low-overhead Defense against Spectre Attacks via Binary Analysis

Guanhua Wang
National University of Singapore

Sudipta Chattopadhyay
Singapore Univ. of Tech. and Design

Ivan Gotovchits
Carnegie Mellon University

Tulika Mitra
National University of Singapore

Abhik Roychoudhury
National University of Singapore
abhik@comp.nus.edu.sg

ABSTRACT

The Spectre vulnerability in modern processors has been reported earlier this year (2018). The key insight in this vulnerability is that speculative execution in processors can be misused to access secrets speculatively. Subsequently even though the speculatively executed states are squashed, the secret may linger in micro-architectural data structures such as cache, and hence can be potentially accessed by an attacker via side channels. In this report, we propose *oo7*, a binary analysis framework to check and fix code snippets against potential vulnerability to Spectre attacks. Our solution employs control flow extraction, taint analysis and address analysis to detect tainted conditional branches and their ability to impact memory accesses. Fixing is achieved by selectively inserting a small number of fences, instead of inserting fences after every conditional branch. Due to the accuracy in our analysis, *oo7* suggests inserting less fences, and is shown experimentally to impose acceptably low performance overheads; less than 2% performance overhead is observed in our experiments on GNU Core utilities. Moreover, the accuracy of the analysis allows *oo7* to effectively detect fourteen (14) out of the fifteen (15) Spectre vulnerable code patterns proposed by Paul Kocher [6], a feat that could not be achieved by the Spectre mitigation in C/C++ compiler proposed by Microsoft. While *oo7* is both low-overhead and effective, for large scale deployment of our solution we need to investigate and optimize the time taken by our compile-time analysis. Finally, we show that similar binary analysis solutions are possible for detecting and fixing Meltdown.

1 INTRODUCTION

The Spectre [7] and Meltdown [9] vulnerabilities in modern processors were discovered in 2017 and made public in January 2018. The attacks that exploit these vulnerabilities can potentially affect almost all modern processors irrespective of the vendor (Intel, AMD, ARM) and the computer system (desktop, laptop, mobile) as long as the processor performs out-of-order and speculative execution. Thus these vulnerabilities have far reaching impact and received enormous attention from both hardware and software communities.

Out-of-order and speculative execution [4] are indispensable micro-architectural optimizations for performance enhancement, ubiquitous in almost all modern high-performance processors except for the simplest micro-controllers. The basic idea behind out-of-order execution is to enable the hardware to execute the instructions in an order that is different from the program order, i.e., the order in which the instructions appear in the program binary. While any long-latency instruction (say instruction $\mathcal{I}$) and its dependents are waiting to complete execution, out-of-order execution enables to get useful work done by executing in parallel the instructions that are not dependent on $\mathcal{I}$ even if they are further down in the program order. Speculative execution introduces further aggressive optimization where the conditional branches are predicted in hardware and the instructions along the predicted branch path are executed speculatively and out-of-order w.r.t. the conditional branch. Once the conditional branch direction is resolved and if the branch was branch was wrongly predicted, the instructions along the speculative execution path are squashed and the execution proceeds along the correct path. The key to these optimizations is that the transient instructions (the out-of-order instructions and the instructions along the speculative path) do not make any changes to the architectural states that are visible to the programmer, namely the registers and memory content, till all the prior instructions in program order have completed execution and the branch outcome is known. The temporary results during transient instructions execution are maintained in internal micro-architectural structures and caches that cannot be accessed by software through the instruction-set architecture — the formal and well-defined interface between the architecture and the software. Thus these micro-architectural optimizations are supposed to be completely transparent to the programmer.

Both Spectre and Meltdown exploit out-of-order and speculative execution to deliberately induce the execution of targeted transient instructions. The transient instructions are tricked to bring in secret data into the internal micro-architectural states, specifically the cache. Of course these transient instructions are subsequently squashed but the secret content in the cache remains. The attacker then carefully accesses the cache content (that is supposed to be hidden to the software) through cache side-channel attacks [12]. The mitigation of Meltdown vulnerability requires changes to the operating system kernel code that can fix the issue completely. Spectre, however, represents a whole class of attacks. The original website of Spectre states that "As [Spectre] is not easy to fix, it will haunt us for a long time." In this work, we focus our effort to identify program binaries that are vulnerable to Spectre attack and patch those binaries with minimal performance overhead as a mitigation technique. We present a comprehensive solution, called *oo7*, based on binary analysis that is both accurate and scalable. As Spectre attacks exploit speculative execution, a natural thought might be to prevent speculative execution for every branch or a set of sensitive branches identified by developers. We show that blindly preventing speculative execution leads to an unacceptable performance overhead, let alone the impracticality of relying on developers to identify “sensitive" branches. Indeed, the current Spectre mitigation, as introduced by Microsoft compiler [2], misses most (13 out of 15) litmust tests for Spectre vulnerabilities [6].

Identifying program binaries and the exact locations in those binaries that are susceptible to Spectre attack is challenging for multiple reasons. First, the analysis should work at binary level rather than at the source code level. This is because the binary represents the exact code being executed on processor micro-architecture whose very nature is exploited by Spectre attacks. Binary analysis remains a difficult endeavor for many reasons, including the difficulty of extracting control flow from program binaries. Due to the presence of control transfer instructions such as register indirect jumps in binaries, re-constructing basic program structures such as control flow graphs itself remains challenging. The recent interest in Spectre and meltdown attacks have raised the prospect of checking for Spectre vulnerabilities at the binary level. As is shown by our work, this can be accomplished via control flow extraction, taint analysis and address analysis at the binary level. In particular, taint analysis is needed to compute which instructions are attacker controlled and also for computing data dependencies across instructions. Address analysis is needed for precise computation of static data dependencies. Furthermore, as we show in our work, via the use of address analysis (which memory addresses are touched by an instruction), we can also estimate the amount of leakage risked by a Spectre code vulnerability.

Second, the analysis should detect all the different variants of Spectre. The number of possibilities is endless at high-level programming language level as evidenced by the vulnerable code patterns detected by *oo7*. This also eliminates the possibility of simple syntax checking to detect the code patterns. We distill down these different variants into a set of simple conditions (at binary level) that should hold good in vulnerable code fragment for Spectre attack to manifest itself. Third, we need inter-procedural analysis including library code as the different parts of the vulnerable code pattern can straddle across procedural boundaries. Last but not the least, the analysis needs to model the transient instructions execution along the speculative path that has never been required in traditional program analysis dealing with only programmer visible execution. We have extended the binary analysis in *oo7* to accurately model speculative execution, which is absolutely necessary to detect Spectre vulnerability.

In summary, we have designed and provided a technical machinery for checking the Spectre attacks, via data dependency analysis and taint propagation. We show that *oo7* is robust enough to detect 14 out of the 15 variants of Spectre attacks proposed as litmus tests by Paul Kocher [6], whereas Microsoft Visual C++ Compiler could detect only 2 of the 15 variants [6]. We continue to employ our checker in search for Spectre attacks in the wild. We tested over 150 binaries from several real world projects including `botan`, `coreutils`, `darknet`, `gdb`, `php` and `redis`. Our proposed *oo7* approach analyzes these binaries within 4968 seconds on average, with the minimum analysis time being 1 second. We note that so far, no Spectre attacks have been found in the wild. By making Spectre detection tools such as *oo7* available in the public domain, we hope that the search for zero day Spectre attacks in the wild, can be substantially accelerated via community participation.

We also evaluated the repair strategies of *oo7* on real world binaries selected from `coreutils` [3]. Our evaluation reveals that the runtime overhead introduced by our repair is only up to 1.78%. This is in stark contrast to a substantial 72% runtime overhead when speculative execution is prevented for all conditional branches via fences.

2 BACKGROUND AND MOTIVATION

We first give an overview of the Spectre vulnerability reported in [7].

Spectre vulnerability. The code fragment in the following exhibits the basic structure of Spectre vulnerability.

```
void victim_function_v01(size_t x) { //CB (branch)
     if (x < array1_size) {
          //IM1 (access to array1)
          //IM2 (access to array2)
          temp &= array2[array1[x] * 256];
     }
}
```

Typically a Spectre vulnerability involves three instructions – *(i) CB*: a branch instruction dependent on untrusted input, *(ii) IM1*: a memory access (`load`) pointing to a location of a secret, and *(iii) IM2*: a memory access (`load` or `store`) where the accessed address depends on the value of the secret pointed by *IM1*. The objective of the attacker is to determine the value of a secret in the victim function `victim_function_v01`.

The input `x` is controlled by an attacker. The key idea is to train the branch predictor in the execution platform via input `x` and to mispredict the branch when $x \geq array1\_size$. Most modern processors employ speculative execution as a crucial micro-architectural feature to boost runtime performance. This means, when the conditional check $x \geq array1\_size$ is mispredicted, the execution may still continue to execute *IM1* and *IM2* from the wrong path (i.e. under the true leg of the conditional $x < array1\_size$). Eventually, the false outcome of the conditional $x \geq array1\_size$ is known. As a result, instructions *IM1* and *IM2* are never completed to affect the functional state of the program, however, they affect the state of the cache. The cache state is not flushed due to performance reasons. This very phenomenon is exploited by an attacker to ex-filtrate the value of `array1[x]`, a potential secret.

IM1 and *IM2* execute speculatively when the check *CB* fails and $x \geq array1\_size$ holds. As a result, the address `&array1[x]` points to a location outside the bound of `array1` and potentially to a secret. We note that depending on the value of `array1[x]`, the address `array2 + (array1[x]*256)` accesses different sets of the cache. Hence, if `array2` is accessible by attacker, then she can launch a cache side-channel attack to determine the value of `array1[x]` (i.e. the secret). Specifically, assume that `array2` was not cached before the memory access *IM2*. The attacker writes a separate and fairly simple attack code to access `array2[y*256]` for all possible values of `y` from `0x00` to `0xFF`. The attacker observes that `array2[y*256]` will result in a cache hit if $y = array1[x]$. In other words, by timing the access of `array2` via the attack code, an attacker can determine the value of `array1[x]` (i.e. the secret) in the victim function `victim_function_v01`. Finally, by controlling the value of `x`, the attacker can point to different locations of the secret and ex-filtrate the entire secret by cache side-channel attacks.

In this section, we discuss the technical challenges to detect and fix Spectre vulnerabilities via simple examples. We will then provide the key insight behind our *oo7* approach.

2.1 Challenges in Detecting Spectre

From the viewpoint of software analysis and testing, the detection of Spectre vulnerability faces the following technical challenges:

1. **More than syntax checking:** Spectre vulnerability involves code patterns whose behaviour can be controlled by untrusted inputs. Syntactic checks over program code are unlikely to pinpoint which parts of the code are controlled by untrusted inputs, potentially leading to a significant amount of false positives.
2. **More than intra-procedural analysis:** As demonstrated by Paul Kocher [6], Spectre vulnerable code patterns may span across multiple procedure boundaries. Restricting analysis to a single procedure may either miss Spectre vulnerabilities (false negatives) or introduces significant number of false positives.
3. **More than state-of-the-art software testing:** Spectre vulnerability exploits low-level micro-architectural features, specifically, the speculative execution. State-of-the-art software test execution only follows execution in the program order. Hence, state-of-the-art software testing will skip Spectre vulnerabilities, leading to false negatives.
4. **Modeling more than program semantics:** As Spectre vulnerability exploits micro-architectural features (i.e. speculative execution), it is crucial for an analysis, which is designed to detect Spectre vulnerability, to accurately understand the way speculative execution happens. To this end, it is important to investigate the number of instructions that can be executed speculatively.

In the next section, we will introduce simple code fragments to discuss the aforementioned challenges in more detail. For the sake of simplicity, we use examples that reflect `C`-language like syntaxes. However *oo7* directly operates on the binary code to take into account all compiler optimizations and to accurately reflect the impact of speculative execution.

2.2 Examples of Spectre Vulnerabilities

Figure 1(a) captures a code fragment of function `f()` with attacker-controlled input `x`. Function `f()` calls two different functions `g()` and *h()*, as defined in Figure 1(c). For the sake of demonstration, we assume that neither `g()` nor `h()` were inlined by the compiler. Function `g()` contains two indirect memory accesses `array1[y]` (say *IM1*) and `array2[array1[y]]` (say *IM2*) where *IM2* is data-dependent on *IM1*. Hence, function `g()` satisfies two memory-access-related conditions for launching a Spectre attack, given that `y` can be manipulated via an attacker.

Function `f'()` in Figure 1(b) is similar to function `f()`, except that the call to function `g()` is preceded by a call to function `long()`. We assume that `long()` executes a substantial number of instructions at runtime. We skip the code for `long()` as it is not relevant for the rest of the discussion. In the following, we outline the technical challenges in detecting Spectre vulnerabilities in the context of the code fragments in Figure 1(a)-(b).

Will syntax checking work? A natural thought to detect Spectre vulnerability might be to exploit the syntactic structure of the code. This is to discover code patterns exhibiting a conditional branch (say *CB*) that encloses both *IM1* and *IM2*. As observed from Figure 1(a), function `f()` exhibits two such code patterns for both of its conditional branches. However, the return value of function `h()` is independent of attacker controlled input `x`. Hence, the first conditional check in `f()` and the enclosed call `g(y)`, despite capturing a Spectre-like pattern, they do not manifest any Spectre vulnerability at runtime. This shows that a naive syntactic check may fail to accurately detect Spectre-like vulnerabilities in arbitrary binary code.

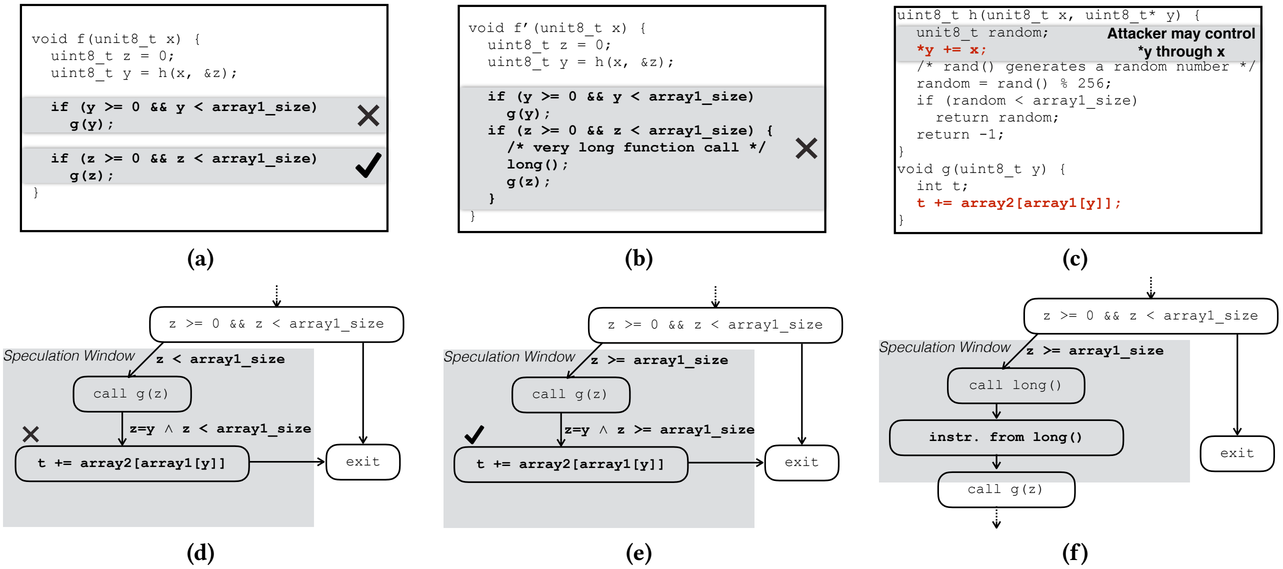
Will intra-procedural analysis work? Semantic analysis can be applied both intra-procedural and inter-procedural. For intra-procedural analysis, we might assume that any variable changed or returned via a procedure call is *untrusted*. For example, in function `f()`, we can assume that both variable `y` (returned by the procedure call `h(x,&z)`) and variable `z` (might be updated by the procedure call `h(x,&z)`) are untrusted. This leads the analysis to manifest two Spectre vulnerabilities, one each for the conditional checks in `f()`. However, as observed from the definition of function `h()`, its return value cannot be influenced by attacker. Hence, the aforementioned conservative take on intra-procedural analysis raises several *false alarms*.

From a different standpoint, we observe that the signature of Spectre may span across multiple procedure boundaries in function `f()`. For example, consider the second conditional check in function `f()` that indeed manifests a Spectre vulnerability during execution. However, to detect such a vulnerability, it is crucial that the necessary conditions for the Spectre vulnerability are checked across procedure boundaries. The vulnerability in our example is similar to the following Spectre litmus test, as proposed by Paul Kocher and missed by Microsoft compiler mitigation [6].

```
__declspec(noinline)
void leakByteNoinlineFunction(uint8_t k) {
     temp &= array2[(k)* 512];
}
void victim_function_v03(size_t x) {
     if (x < array1_size)
          leakByteNoinlineFunction(array1[x]);
}
```

It is likely that the Microsoft compiler mitigation failed to detect the presence of Spectre vulnerability across procedure boundaries. This litmus test is accurately detected and fixed by our *oo7* approach.

Will state-of-the-art testing work? A different approach to expose Spectre vulnerabilities will be to leverage the progress on software testing. To this end, let us consider the use of symbolic execution to expose Spectre vulnerabilities. Since `x` is controlled by attacker, the objective is to execute the code of function `f()` while `x` being a symbolic input. Figure 1(d) captures an excerpt of the control flow graph (CFG) where the Spectre vulnerability actually manifests. Along the control flow edges, we also show the



```
void f(unit8_t x) {
  uint8_t z = 0;
  uint8_t y = h(x, &z);

  if (y >= 0 && y < array1_size)
    g(y);

  if (z >= 0 && z < array1_size)
    g(z);
}
```

```
void f'(unit8_t x) {
  uint8_t z = 0;
  uint8_t y = h(x, &z);

  if (y >= 0 && y < array1_size)
    g(y);
  if (z >= 0 && z < array1_size) {
    /* very long function call */
    long();
    g(z);
  }
}
```

```
uint8_t h(uint8_t x, uint8_t* y) {
  uint8_t random;
  *y += x;
  /* rand() generates a random number */
  random = rand() % 256;
  if (random < array1_size)
    return random;
  return -1;
}
void g(uint8_t y) {
  int t;
  t += array2[array1[y]];
}
```

Figure 1: Analyzing Spectre vulnerabilities. ✓ captures a Spectre vulnerability whereas ✗ captures its absence. "Speculation Window" captures the instruction bandwidth executed speculatively, i.e., after a branch instruction is executed and before the outcome of the branch is resolved: (a) code fragment where one branch exhibits Spectre vulnerability. Functions `g()` and `h()` are shown in Figure 1(c); (b) code fragment where none of the branches exhibit Spectre vulnerabilities, (c) highlights the statement to show the taint propagation through attacker controlled input `x`, (d) since classic symbolic execution does not model speculative execution, it does not detect the Spectre vulnerabilities in Figure 1(a); (e) invariants passed through the control flow edges that exposes the Spectre vulnerability in Figure 1(a); (f) the number of instructions in `long()` goes past the speculative window. Hence, the Spectre-like code fragment in Figure 1(b) cannot be exploited.

constraints propagated during the symbolic execution. As observed before the memory accesses via `array1` and `array2`, the control flow satisfies the invariant $y < array1\_size$. This satisfactorily passes the array bound checks of `array1` for the respective memory access and does not expose the Spectre vulnerability.

The reason that testing overlooks Spectre vulnerabilities is that the test execution does not take into account micro-architectural features, such as speculation, which happens to be the central reason behind Spectre vulnerabilities. The phenomenon is shown in Figure 1(e) where memory accesses via `array1` and `array2` take place via speculative execution, i.e., while waiting for the *false* outcome of branch condition $z \geq 0 \wedge z < array1\_size$. As a result, these memory accesses are executed with the invariant $y \geq array1\_size$, as shown alongside the respective control flow edges in Figure 1(e). This leads to a violation of array bound checks for `array1` in function `g()`. Such a contrived test execution strategy is, however, missing in state-of-the-art testing tools. We note that the aforementioned argument is applicable to any testing strategy and not just in the context of symbolic execution.

Why model micro-architectural features? As explained in the preceding section, the memory accesses via `array1` and `array2` (in function `g()`) manifests Spectre vulnerabilities when they are executed speculatively. Each processor can only execute a limited number of instructions speculatively. In particular, the number of speculatively executed instructions cannot exceed the size of reorder buffer (ROB) of a processor. In Figure 1(d)-(f), we highlight such limited window of speculatively executed instructions as the "Speculation Window". This means if the memory accesses via `array1` and `array2` do not fit into the speculation window, then the Spectre vulnerability cannot occur at runtime.

Figure 1(b) captures the code fragment of function `f'()`. Similar to function `f()`, in `f'()` a syntactic check will detect two Spectre vulnerabilities, one each for the conditional checks. Without modeling the speculation window, yet using inter-procedural data flow analysis, `f'()` will exhibit one Spectre vulnerability, specifically, for the second conditional check. In reality, however, this Spectre vulnerability never occurs, as the memory accesses via `array1` and `array2` fall outside the speculation window (cf. Figure 1(f)). This example shows that it is critical to devise inter-procedural analysis together with the specific micro-architectural features such as the speculation window.

2.3 Challenges in Mitigating Spectre

A naive approach to fix Spectre vulnerability is to put fences (e.g. `lfence` in x86) after every branch instruction. Fences perform a serialization operation. In particular, a fence does not execute until all instructions before it have been completed. Similarly, any instruction after the fence can execute only after the fence completes. As a result, fences restrict the speculative execution (speculation indeed executes instructions after a branch before the branch completes) and hence, the exploitation of Spectre vulnerabilities. Such

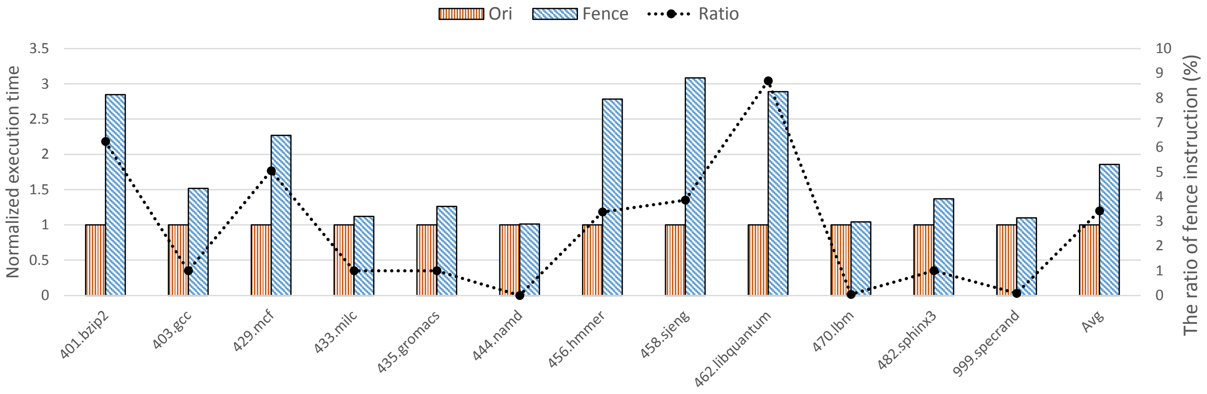


Figure 2: The performance overhead introduced by fence instructions.

an approach will potentially induce substantial overhead at the cost of security. This is because, speculative execution is an essential and fundamental technique to improve the runtime performance. As observed from Figure 1(a), only one conditional check manifests Spectre vulnerability, while in Figure 1(b), none of the conditional checks manifest the vulnerability. Blindly inserting fences after a branch will, therefore, lead to unnecessary performance overhead in functions `f()` and `f'()` without additional security guarantees. In essence, even though inserting fences stops the exploitation of Spectre vulnerabilities, such insertion should be carefully guided to avoid substantial performance overhead. In our *oo7* approach, we provide such guidance automatically via analysis results. This keeps the runtime overhead at check, yet providing strong protection against Spectre.

Figure 2 outlines the performance overhead incurs in SPEC2006 benchmarks [5] when fences are inserted for every branch instruction. On average, the execution time increases by a factor of 1.8, while the memory fence instructions are only 5.8% of total executed instructions. Moreover, `sjeng4` highlights the largest overhead, capturing an increased execution time by a factor of 3.25. Our *oo7* approach discovers that none of the branches in SPEC2006 benchmark is tainted and thus, they do not exhibit Spectre vulnerability. This result further motivates the requirement of systematic analysis methods to detect and fix Spectre vulnerabilities.

3 *OO7* APPROACH AT A GLANCE

Our proposed *oo7* approach broadly revolves around two stages: *(i)* Analysis stage, which checks the binary code and discover potential code regions susceptible to Spectre attacks. *(ii)* Fixing stage, which automatically synthesizes fences to patch the binary. This is accomplished via the guidance from the analysis stage.

The analysis stage involves the following steps to detect potential code regions with Spectre vulnerability:

(1) **Detecting** *CB***:** We perform an inter-procedural taint analysis where each input arriving from an untrusted source (e.g. network, file or console commands) is tainted. The propagation of all such taints are tracked as they go through the different program statements. For instance, in Figure 1(a), variable `x` is untrusted and its taint is propagated to parameter `y` in function `h()` defined in Figure 1(c). This taint is further propagated to variable `z` in function `f()`. The objective of this stage is to identify the set of branch instructions *CB* whose outcome depend on untrusted variable. To this end, we check whether a branch condition involves tainted variables. Both in Figure 1(a) and in Figure 1(b), only the second conditional check involves tainted variable `z`. Therefore, *CB* contains the second conditional check for function `f()` and `f'()`.

(2) **Detecting** *IM1***:** For a branch instruction $br \in CB$, we check the reachability of a memory instruction *IM1* that depends on tainted variable. To this end, we launch a depth-first search, up to the depth bounded by the speculation window, to locate *IM1* in a program path reachable from *br*. Since the search is bounded by speculation window, it substantially reduces the possibility to consider *IM1* that can never be executed speculatively. For example, in Figure 1(e), the search from the conditional branch will go past all instructions in function `g()`. This, in turn, locates `array1[y]` as *IM1*, as the taint from the untrusted source `x` was propagated to variable `z` in function `f()`. This taint, then was propagated to parameter `y` of function `g()`. In contrast, for Figure 1(f), the bounded search from the conditional check stops before the call to function `g()`. Hence, our analyzer does not identify any instruction as *IM1*.

(3) **Detecting** *IM2***:** In the last step, we search for a memory instruction *IM2*, which refers to an *address* depending on the *value* accessed by *IM1*. To this end, we perform a data dependency analysis during the depth-first search from a branch condition $br \in CB$, as explained in the preceding paragraph. Specifically, during the depth-first search, if *IM1* is identified, then we check any memory instruction *IM2* where the accessed address is data-dependent on *IM1*. Moreover, we check whether such an instruction *IM2* is reachable within the speculation window. For example, in Figure 1(e), the address accessed via `array2` depends on the value `array1[y]`. Since `array1[y]` was identified as *IM1*, we identify the memory access via `array2` as *IM2*. Thus, we locate both *IM1* and

IM2 reachable from the second conditional check in `f()`, as identified *CB*. This triggers a Spectre vulnerability.

In the fixing stage, we walk through the branch instructions identified in the analysis stage and automatically insert fences *only* before these branch instructions. For example, in function `f()` and function `f'()`, our analysis identifies only the second conditional check in `f()` to be amenable to Spectre vulnerability. Thus, our fixing stage inserts only one fence instruction. This is in contrast to four fences for a naive solution inserting fences after every branch instruction.

4 METHODOLOGY

In this section, we describe our *oo7* approach in detail.

4.1 Foundation

To describe our analysis process, we use the notations in Table 1 for the rest of the paper.

Table 1: Symbols used in describing *oo7*

Symbol	Interpretation
$br(inst)$	*true* if and only if *inst* is a branch instruction
$mem(inst)$	*true* if and only if *inst* is a memory access instruction
$load(inst)$	*true* if and only if *inst* is a load instruction
$\tau(inst)$	*true* if and only if instruction *inst* is tainted
$\Delta(inst1, inst2)$	minimum no. of instructions executed to reach $inst2$ from $inst1$. If $inst2$ is unreachable from $inst1$, then $\Delta(inst1, inst2) = \infty$.
$Dep(inst1, inst2)$	*true* if and only if instruction $inst2$ is data-dependent on $inst1$
$val(X)$	value located at memory address X
SEW	Speculation window for the targeted platform

We say that an instruction is *tainted*, i.e., $\tau(inst)$ is true, if and only if the instruction operates on some tainted operands. For example, in the operation "$inst : y := x + z$", if either x or z is tainted, then the instruction *inst* is tainted and $\tau(inst)$ holds.

4.2 Our Approach *oo7*

In the following, we first formally introduce the Spectre vulnerability checking condition. Then, we will describe the core program analysis embodied within the *oo7* approach to check the satisfiability of such condition.

4.2.1 ***Checking Condition.***

For a given target program, our *oo7* approach analyzes the program to detect Spectre vulnerabilities. To this end, *oo7* aims to locate instructions *CB*, *IM1* and *IM2* such that the following condition $\Phi_{spectre}$ is satisfied:

$$\boxed{\begin{aligned} \Phi_{spectre} \equiv br(CB) \wedge load(IM1) \wedge mem(IM2) \wedge \\ \tau(CB) \wedge \tau(IM1) \wedge \tau(IM2) \wedge \\ (\Delta(CB, IM1) \leq SEW) \wedge (\Delta(CB, IM2) \leq SEW) \wedge \\ Dep(CB, IM1) \wedge Dep(IM1, IM2) \end{aligned}} \tag{1}$$

Intuitively, the first two lines of $\Phi_{spectre}$ capture the presence of tainted branch instructions *CB* and tainted memory-access instructions *IM1* and *IM2*. The last two lines show that *IM1* and *IM2* are located within the speculation window of the branch instruction *CB*, and they are data-dependent.

4.2.2 ***Taint Analysis.***

We use taint analysis [1] to determine whether conditional branch instructions (e.g. *CB*) and the memory-access instructions (e.g. *IM1* and *IM2*) can be controlled via untrusted inputs. In the following, we outline the taint propagation policies used to detect Spectre vulnerabilities.

Default Taint Policies. The default taint propagation policy considers the *value-to-value* taint propagation. Concretely the following rules are followed while propagating value taints:

- Computation instruction e.g. `op(X, y, z)`: The operation "`op`" works on X, y and finally computes the output z. If X is tainted, then the taint is *directly* propagated to z.
- Load instruction e.g. `Load(X, y)`: The operation `load` fetches the value from memory address X to register y. If $val(X)$ is tainted, then the register y is tainted.
- Store instruction e.g. `Store(y, X)`: The operation `store` transfers the value in register X into memory address y. If X is tainted, then $val(y)$ is tainted.

Pointer to Value Taint Propagation. After discovering the tainted branch instruction i.e., *IB*, we temporarily enable pointer to value taint propagation. This is to locate *IM1* and *IM2*. For instance, consider the load operation `Load(X, y)`, which loads the value $val(X)$ to a register y. If the address X is tainted, then we temporarily enable the taint propagation from address X to register y. We terminate the address to value taint propagation if *IM1* and *IM2* are not located within the speculation window SEW. We note that it is sufficient to track value taints for detecting *CB*. This is because, the purpose of Spectre attack is to control the branch outcome of *CB* via untrusted input(s). Thus, the propagation of value taints from the untrusted inputs will detect the presence of *CB*.

4.2.3 ***Program Analysis to Check*** $\Phi_{spectre}$.

Algorithm 1 outlines our approach to *detect* Spectre vulnerabilities in arbitrary binary programs. In particular, we aim to find the set of triplets of the form $\langle CB, IM1, IM2 \rangle$ that satisfy the condition $\Phi_{spectre}$ for Spectre vulnerability (cf. Equation 1). While analyzing the program, the variable $step$ monitors the satisfiability status of $\Phi_{spectre}$. Once the checker finds instances of *CB*, *IM1* and *IM2*, it records the presence of Spectre vulnerability into Φ.

The checker maintains the program state into *GS* after interpreting an instruction *inst*. The program state is further used to propagate the taints according to the policy described in Section 4.2.2. For each tainted instruction *inst*, i.e., when $\tau(inst)$ holds, our analysis invokes *oo7.check*. This is to check whether $\Phi_{spectre}$ can be

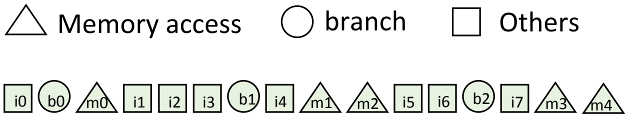
Algorithm 1 Spectre Detection Algorithm

```
Input: P: Program binary
Output: Φ: A set of triplets of the form ⟨CB, IM1, IM2⟩ capturing Spectre vulnerabilities
$\Phi \leftarrow \emptyset$;
TS.policy ← VtoV                                   ▷ Taint policy set value-to-value
step ← None                                        ▷ Initialize Spectre detection stage
Let inst be the first instruction of P
while inst ≠ exit do
    GS ← INTERPRETER.exe(inst)                     ▷ GS: Global State
    TAINTENGINE.taint(inst, GS)                    ▷ propagate taints
    if $\tau(inst)$ then                           ▷ oo7 is invoked only for tainted instruction
        DS ← oo7.check(inst)                       ▷ DS: Detector State
    end if
    inst ← P.next()                                ▷ fetch next instruction
end while
procedure oo7.CHECK(inst)
    step ← DS.step()                               ▷ Checks the stage of detection
    if br(inst) then                               ▷ check for CB
        DS ← DS.setCB(inst)                        ▷ recognize that inst might capture CB
        step ← STEP_CB                             ▷ progress the detection stage to CB
        TS.policy ← PtoV                           ▷ enable pointer-to-value taint
    end if
    if $(load(inst) \wedge step = STEP\_CB)$ then
        cb ← DS.CB()                               ▷ get CB from detection state
        if $(Dep(cb, inst) \wedge \Delta(cb, inst) \leq SEW)$ then      ▷ check for IM1
            DS ← DS.setIM1(inst)                   ▷ recognize that inst might capture IM1
            step ← STEP_IM1                        ▷ progress the detection stage IM1
        end if
    end if
    if $(mem(inst) \wedge step = STEP\_IM1)$ then
        DS ← DS.setCB(inst)                        ▷ get CB from detection state
        if $(Dep(cb, inst) \wedge \Delta(cb, inst) \leq SEW)$ then      ▷ check for IM2
            DS ← DS.setIM2(inst)                   ▷ recognize that inst might capture IM2
            Φ ∪ = ⟨DS.CB(), DS.IM1(), DS.IM2()⟩    ▷ catch Spectre
            step ← None                            ▷ reset checker
            TS.policy ← VtoV                       ▷ disable pointer-to-value taint
        end if
    end if
    if $(step = STEP\_CB \wedge \Delta(DS.CB(), inst) > SEW)$ then      ▷ Outside SEW
        step ← None                                ▷ Reset detection beyond speculation window
        TS.policy ← VtoV
    end if
    if $(step = STEP\_IM1 \wedge \Delta(DS.CB(), inst) > SEW)$ then     ▷ Outside SEW
        step ← None                                ▷ Reset detection beyond speculation window
        TS.policy ← VtoV
    end if
    return DS
end procedure
```

satisfied. As shown in Algorithm 1, the procedure *oo7.check* involves first three conditional blocks to check the presence of *CB*, *IM1* and *IM2*, respectively. Within the third conditional block, if the presence of *IM2* is detected, then $\Phi_{spectre}$ is satisfied. This, in turn, captures the presence of Spectre vulnerability. The checker is then reset (i.e. *step* is assigned to *None*) to continue hunting more Spectre vulnerabilities.

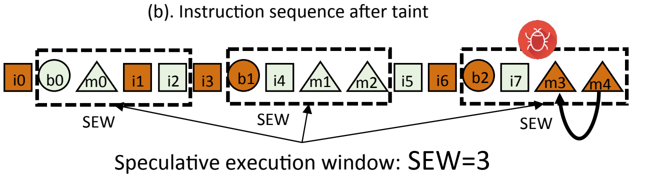
The last two conditional blocks in *oo7.check* reflect scenarios where *IM1* or *IM2* are not discovered within the speculation window of branch *CB*. Hence, in such scenarios, the checking stage is reset (i.e. *step* is assigned to *None*) and *oo7.check* progresses the checking stage only when another tainted branch is detected in its first conditional block.

Figure 3 highlights how Spectre vulnerability is detected via Algorithm 1. As observed from Figure 3(c), despite the presence of tainted branch b1, there is no spectre vulnerability surrounding b1. This is because of the absence of respective *IM1* and *IM2* that are necessary for $\Phi_{spectre}$ to be satisfiable. Finally, from the tainted branch b2, we can locate memory accesses m3 and m4 within the speculation window *SEW*. Moreover, b2, m3 and m4 satisfy the checks for *CB*, *IM1* and *IM2*, respectively, in the context of $\Phi_{spectre}$. Hence, the sequence $\langle b2, m3, m4 \rangle$ triggers a Spectre vulnerability.



(a). Normal instruction sequence

(b). Instruction sequence after taint



(c). Instruction sequence with speculative execution

Figure 3: Workflow of Spectre detection. (a) an arbitrary sequence of instructions, (b) tainted instructions shown via dark blocks, (c) Spectre vulnerability is detected with speculation window *SEW*=3

4.2.4 *Detecting variants of Spectre vulnerability*.

In the preceding section, we discussed the detection of the basic version of Spectre vulnerability. As discussed by the inventors of Spectre, the vulnerability can also be exploited via *indirect branch instructions*. For example, in indirect branches, the branch target address may reside in a register or memory. Hence, such branches may lead the program control to reach multiple destinations at runtime. If the branch predictor is trained with attacker preferred addresses, then during speculation, the control may reach to the attacker preferred address. This might force the program to execute an attacker code. This is a powerful attack, yet we can easily catch such Spectre vulnerability within our *oo7* framework. In particular, for each indirect branch in the binary code, we check whether it is *tainted*. A tainted indirect branch reflects that the attacker can control the target of the branch instruction, leading to the potential execution of malicious code during speculation.

Our implementation of *oo7* considers both variants of Spectre vulnerabilities, i.e., spectre vulnerabilities exhibiting the sequence $\langle CB, IM1, IM2 \rangle$ and spectre vulnerabilities via indirect branches. Hence, *oo7* is capable to detect both these variants during the binary analysis.

4.2.5 *Quantifying the Leakage*.

oo7 not only detects Spectre vulnerabilities, but also quantifies the potential leakage of information caused by each detected vulnerability. To this end, we observe that the index of memory access *IM1* (i.e. the index of `array1` in a typical Spectre vulnerability) is used to point to secret bytes. Hence, if we can compute the range of values such an index can hold over different program executions, then we can quantify the potential leakage of information.

As an example, consider the Listing 1 where the function `victim` exhibits a typical Spectre vulnerability. Let us assume that set of values possessed by the parameter `x` is $Value_x$. Thus, $|Value_x| - array1\_size$ captures the amount of memory that can be read by an attacker who exploits a Spectre vulnerability in function `victim`.

Let us assume that we abstract the set of values of a variable by a set of intervals. Considering Listing 1, we observe that variable `b` holds values $\{(100, 200], [20, 20]\}$ before the function call `victim(b)`. Similarly, variable `c` holds values $\{(50, 100]\}$ before the function call `victim(c)`. Since, function `victim` is called from different contexts, values of parameter `x` must take into account both these calling contexts. To this end, values of `x` can be computed as a *set union* of values arriving via all calling contexts. Therefore, we can abstract away the set of values of `x` via $\{(100, 200], [20, 20]\} \cup \{(50, 100]\} = \{(50, 200], [20, 20]\}$. This means an attacker can read at most 151 bytes via `x`.

Listing 1: An example for VSA.

```
/* a Spectre vulnerability function */
void victim(int x) {
  if (x < array1_size)
    temp = array2[array1[x]];
}
void foo(int a) {
  int b, c;
  if (a > 100) {
    b = a;
  } else {
    c = a; b = 20;
  }
  if (b <= 200) victim(b);
  if (c > 50)  victim(c);
}
```

We use value set analysis [11] on binary code to quantify the leakage as described in the preceding paragraph.

4.2.6 *Code repair.*

In the following, we present three different repair strategies to automatically fix Spectre vulnerabilities.

Memory fence. The first and simplest repair strategy is to inject serializing instructions (e.g memory fences) after the conditional branch *CB*. The original article describing Spectre attacks [7] suggests to insert memory fences following each conditional branch and its destination. Thanks to our analysis, we have the exact sequence $\langle CB, IM1, IM2\rangle$ vulnerable to Spectre attacks. As a result, we can accurately locate the program point where the memory fence should be inserted. In particular, we insert memory fences following *CB* instruction and immediately before the execution of *IM1*. This prevents execution from loading the secret value into the cache speculatively. Nevertheless, inserting memory fences may affect the overall program performance. However, we note that *oo7* inserts memory fences only for the branches identified as *CB* (via Algorithm 1), instead of inserting fences after each conditional branch. We show empirically that such a strategy has acceptable performance overhead.

Inserting `NOP` instructions. Our second repair strategy is to insert `NOP` instructions between *CB* and *IM1*. For instance, consider the example shown in Figure 4(a). As observed in Figure 4(b), we can insert four `NOP` instructions to make the distance between *CB* and *IM1* going beyond the speculation window (i.e. *SEW*). This results in a Spectre vulnerability that cannot be exploited for an architecture with speculation window *SEW*. In general, for each detected *CB*, we insert $SEW - \Delta(CB, IM1)$ number of `NOP` instructions to fix the respective Spectre vulnerability.

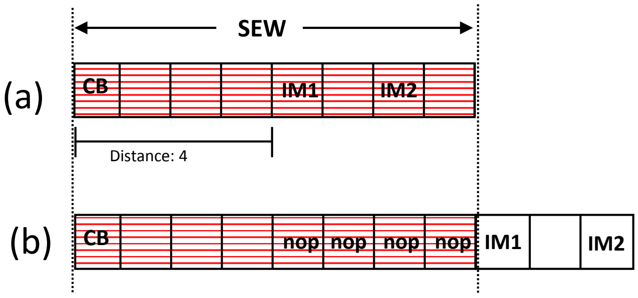


Figure 4: Use `NOP` instruction to fix Spectre vulnerabilities.

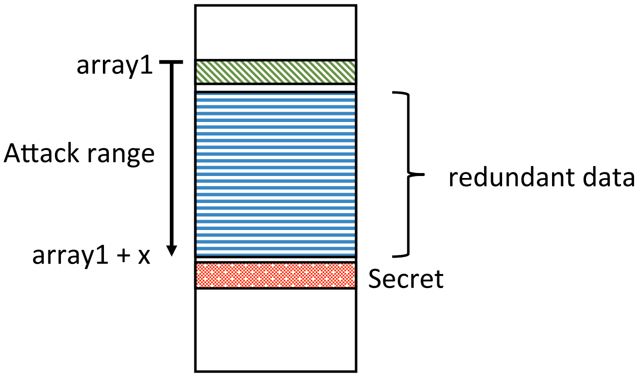


Figure 5: Padding redundant data to prevent an attacker accessing the secret

Padding redundant data. Both repair strategies, as mentioned in the preceding paragraphs, negatively affect the performance of the victim program. It is, however, possible to repair Spectre vulnerability without any performance overhead. In particular, *oo7* quantifies the leakage (cf. Section 4.2.5) to investigate the range of memory touched by an attacker. We leverage this information to pad some redundant data in such a fashion that the attacker is never able to access the secret value. This is outlined in Figure 5. The green area is the base array used to access the secret (i.e. `array1` in Spectre vulnerability) and the red area contains the secret data. The redundant data is padded between `array1` and the sensitive area holding the secret. We note that the attack range computed by *oo7* is an over-approximation. As a result, the padded data guarantees that the attacker is never able to access the secret. Nevertheless, the downside of this repair strategy is the associated memory overhead. Moreover, the location of the base array cannot be automatically identified in the source code. Hence, this repair strategy can only be performed manually.

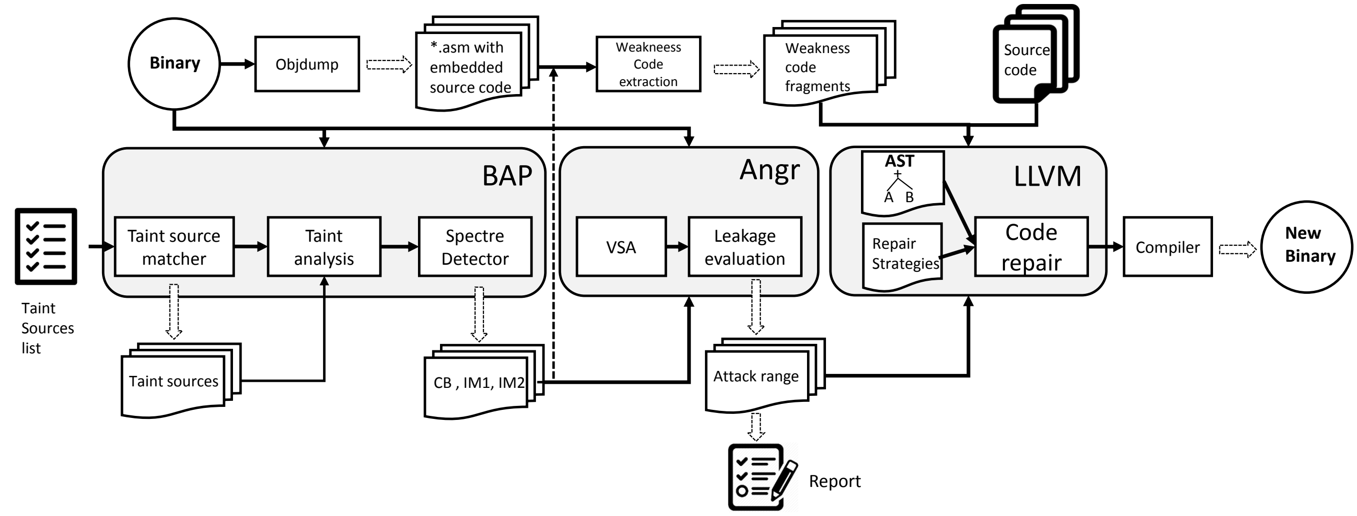


Figure 6: Overview of *oo7* framework.

5 IMPLEMENTATION

Figure 6 provides an overview of *oo7* framework. *oo7* contains three main modules: a Spectre detection module for detecting the Spectre vulnerability, a leakage evaluation module for quantifying the information leakage. This provides a metric for selecting the repair strategy in the code repair module. Finally, the code repair module fixes the Spectre vulnerabilities in the source code given the source code is available.

We adopt BAP [1] as our primary taint analysis platform (cf. Section 4.2.2). BAP provides a toolkit for implementing automated static binary analysis and it supports multiple architectures such as x86, x86-64, ARM and etc. In our *oo7* framework, BAP first takes a binary program and the taint sources as inputs. The taint sources are a subset of the taint source list. The list of taint sources is filtered by the taint source matcher according to the symbol table of the input binary. A taint source is an API that imports the data from an untrusted channel such as network, user input and the file reader interface.

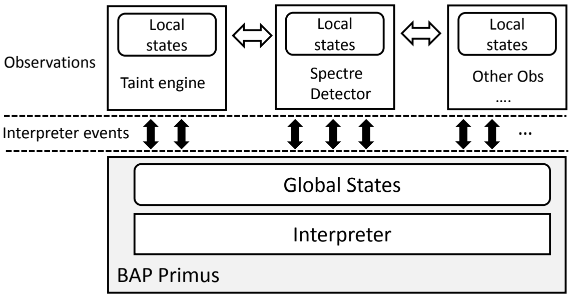


Figure 7: The architecture of the Spectre detection module.

The detailed architecture of the Spectre detection module (cf. Section 4.2.3) is outlined in Figure 7. Primus is a micro-execution framework in BAP that can be used to interpret a program. The core component of Primus is the **Interpreter**. It emulates the execution of a program and provides several interfaces to extract crucial information during the interpretation. Such interfaces use a publish/subscriber architecture to watch the interpreter events. The subscribers are allowed to listen to arbitrary changes in the interpreter state (i.e. Global states). The subscriber also has its own local states that can be shared with other subscribers.

BAP propagates the taints by considering all possible execution scenarios. Loops are unrolled up to a certain depth and might introduce a source of unsoundness in *oo7* if the unrolling depth is optimistic. However, with correctly provided loop bounds, this source of unsoundness can be easily eliminated. During the static analysis, BAP wakes up the specific subscriber when analyzing the events registered by the subscriber. For example, the taint engine is invoked by the analysis when it completes the interpretation of an instruction (post-execution event). When the subscriber of taint engine is invoked, it checks the taint data from the taint source and propagates it if the instruction satisfies the taint policy (cf. Section 4.2.2). The Spectre detector module is invoked by BAP interpreter after the post-execution branch and post-execution-memory events. Spectre detector checks the state of the interpreted instruction in the light of satisfying the condition $\Phi_{spectre}$ explained in Equation 1.

The leakage evaluation module (cf. Section 4.2.5) is implemented on the Angr [11]. Angr is a python-based framework combining both static and dynamic symbolic analysis for the multi-architecture binary.

Our Spectre detection directly works the binary code. Once a vulnerable code fragment is detected in the binary, we locate the corresponding source code fragment for repair. To this end, we compile the binary with debug option (e.g "-g" in gcc compiler). Hence, the disassembled binary code is embedded with the source code fragments. The source code fragments surrounding the Spectre vulnerable code is extracted. For matching these code fragments and automatically repair them in the source code, we implement a method on top of LLVM [8] compiler infrastructure. In particular, we construct an abstract syntax tree (AST) for both the victim program code and the extracted code fragments capturing the detected Spectre vulnerability (via Algorithm 1). We locate the branch instruction involving Spectre (i.e. *CB*) by matching the AST of the Spectre code with the AST of the victim program code. The AST of the victim code is then modified directly by inserting `lfence` or `nop` instructions to repair the Spectre vulnerability (cf. Section 4.2.6).

6 EVALUATION

This section presents detailed evaluation of *oo7*. We also discuss our experiences in finding Spectre vulnerabilities in the wild.

Experimental setup. In Section 4, we proposed an accurate checking condition $\Phi_{spectre}$ (cf. Equation 1) for detecting Spectre attacks that use the CPU cache as a covert channel. In particular, the memory access *IM2* uses the cache convert channel to ex-filtrate the secret. However, as hinted by Paul Kocher [6], there might exists covert channels other than caches. This means, it might be possible to ex-filtrate the secret via other covert channels even in the presence of only the tainted conditional branch *CB* and a tainted memory access *IM1* located within the speculation window of *CB*. Formally, this is captured as a weaker variant $\Phi^{weak}_{spectre}$ of our original condition $\Phi_{spectre}$ as follows:

$$\boxed{\begin{aligned} \Phi^{weak}_{spectre} \equiv br(CB) \wedge load(IM1) \wedge \\ \tau(CB) \wedge \tau(IM1) \wedge \\ (\Delta(CB, IM1) \leq SEW) \wedge \\ Dep(CB, IM1) \end{aligned}} \tag{2}$$

In our evaluation, we aim to detect $\Phi^{weak}_{spectre}$ to broaden the horizon of potential Spectre vulnerabilities. We note that the number of fixes generated with $\Phi^{weak}_{spectre}$ is certainly bounded by the fixes generated with $\Phi_{spectre}$. Consequently, the runtime overhead due to *oo7* fixes is an upper bound on the runtime overhead to prevent Spectre attacks launched using cache covert channels.

We evaluated *oo7* on an execution platform embedded with a Sandy bridge Macro-Architecture CPU E2620. This CPU has 168 reorder buffer (ROB) entries. The ROB entries are shared by all hardware threads except that eight entries are reserved for non-speculative execution. Hence, during the evaluation of *oo7*, we set the speculation window (i.e. *SEW*) to 160.

Research questions. Specifically, our evaluation of *oo7* investigates the following research questions:

(1) **RQ1:** How effective is *oo7* in detecting Spectre vulnerabilities in binary code?
(2) **RQ2:** How efficient is the repair strategy introduced by *oo7*?
(3) **RQ3:** What is the time taken by *oo7* to detect and fix Spectre vulnerabilities?

6.1 RQ1: Effectiveness

The latest Microsoft Visual C++ compiler [2] has integrated a `/Qspectre` switch for mitigating a limited set of potentially vulnerable coding patterns related to the Spectre vulnerabilities. Specifically, after compiling an application with `/Qspectre` enabled, the Visual C++ compiler attempts to insert an `lfence` upon detecting Spectre code patterns.

Paul Kocher [6] has evaluated the Visual C++ compiler by using Spectre example code from the original publication describing Spectre [7] and 14 other variants. The evaluation shows that only two of the Spectre vulnerable examples are identified and repaired by the Visual C++ compiler.

For testing the effectiveness of *oo7*, we also run the 15 Spectre vulnerable examples created by inventors of Spectre. *Our evaluation reveals that 14 out of 15 examples are identified by* oo7 *as Spectre vulnerabilities.* In the following, we will discuss a few such examples that are accurately detected by our *oo7* approach and missed by the Visual C++ compiler.

The following example (example `v03` [6]) involves Spectre code involving a procedure call:

```
__declspec(noinline)
void leakByteNoinlineFunction(uint8_t k) {
     temp &= array2[(k)* 512];
}
void victim_function_v03(size_t x) {
     if (x < array1_size)
          leakByteNoinlineFunction(array1[x]);
}
```

Our *oo7* approach identifies that both the conditional branch and the argument to `leakByteNoinlineFunction` are tainted. Consequently, the vulnerability is exposed by *oo7*.

Another example (example `v04` [6]) uses an `array1` index data-dependent on untrusted input `x`:

```
void victim_function_v04(size_t x) {
     if (x < array1_size)
          temp &= array2[array1[x << 1] * 512];
}
```

This example differs from the original Spectre code [7] that it uses an index `x << 1` instead of using `x` directly. However, due to our taint analysis, `x << 1` is accurately identified to be tainted. This makes both the conditional branch and `array1[x << 1]` to be tainted, leading to a Spectre vulnerability.

The undetected example (example `v13` [6]) contains code patterns that exhibit control dependency between tainted data and untainted code as follows:

```
__inline int is_x_safe(size_t x) {
     if (x < array1_size)
          return 1;
     return 0;
}
void victim_function_v13(size_t x) {
     if (is_x_safe(x))
          temp &= array2[array1[x] * 512];
}
```

In this example, the conditional branch in the victim function `victim_function_v13` should be tainted, as the return value of `is_x_safe(x)` is controlled via untrusted input `x`. However, this is a special case, as the return value is not data-dependent on `x`. Instead, the return value is manipulated via the control dependency

Table 2: Program characteristics

Program	Binary Size (Byte)	Analysis time (s)	Conditional Branches	Tainted $\langle CB, IM1 \rangle$	Max NOP inserted
cat	101200	82	66	2	156
cksum	93848	56	20	4	152
head	115040	198	107	5	154
touch	186264	336	37	3	136
tac	109240	162	62	13	142
factor	178632	272	147	15	155
ptx	169272	486	124	17	153

on `x`. Our current implementation does not support taint propagation via control dependency. As a result, we are unable to reveal the Spectre vulnerability in this example. This, however, does not significantly limit the application of *oo7*, as a precise control dependency tracking will automatically enhance the effectiveness of *oo7* for detecting Spectre vulnerabilities.

6.2 RQ2: Performance Overheads

To stress test the repair strategies of *oo7*, we evaluate it with real world applications. Specifically, we selected several binaries from project `coreutils` [3] to compare the runtime performance of our repair strategies with respect to the runtime performance of inserting fences after all branches. Table 2 outlines some salient features of the selected binaries for our evaluation. In particular, we list the number of conditional branches, as well as the number $\langle CB, IM1 \rangle$ pairs that are tainted. As observed, the number of tainted $\langle CB, IM1 \rangle$ pairs is substantially lower than the number of conditional branches. We also note that the tainted $\langle CB, IM1 \rangle$ pairs are not *true positives*. However, to check the runtime overhead of our mitigation, we employ our repair strategies for any potential Spectre vulnerability detected by *oo7*. Specifically, we employ the insertion of `lfence` and `NOP` instructions, as discussed in Section 4.2.6. Table 2 also outlines the maximum number of `NOP` instructions inserted over any tainted $\langle CB, IM1 \rangle$ pairs. Since we did not identify any secret data in the chosen subject binaries, we did not use our repair strategy that involves the padding of redundant data (cf. Figure 5).

Figure 8 demonstrates the normalized execution time for various repair strategies employed by *oo7*. As observed from Figure 8, the average runtime overhead is 72% when fences are inserted at all conditional branches. In contrast, the repair strategy of *oo7* only incurs 1.57% and 1.78% performance overhead on average for repair strategies inserting `lfence` and `NOP` instructions, respectively. We also count the number of executed `lfence` instructions. To this end, we use pin tool [10]. The marker lines in Figure 8 captures the number of executed `lfence` instructions with respect to the total number of instructions executed. When `lfence` is inserted after each conditional branch, we observe that the average ratio of executed `lfence` is 2.1% with respect to the total number of executed instructions. With our *oo7* approach, however, this ratio is a negligible 0.001%.

6.3 RQ3: Analysis and Fixing Time in *oo7*

The time taken by our analysis engine depends on the size and complexity of the binary. In the selected programs (cf. Table 2), `ptx` consumes the longest time (486s). This is due to its large binary size (169272 bytes). Moreover, `ptx` contains several complex loop structures that further introduces overhead to our analysis. Table 2 also outlines the time taken to generate the mitigation code using `lfence` and `NOP` instructions. We observed that the generation of repair code is efficient and it always finishes within 10 seconds.

To further evaluate the scalability of our analysis method, we have chosen 150 binaries in project `botan`, `darknet`, `gdb`, `php` and `redis` from the Google oss-fuzz project. We have not yet found any real Spectre vulnerabilities in these projects. To analyze these binaries for detecting $\Phi^{weak}_{spectre}$, our *oo7* approach took an average of 4968 seconds, with a maximum and minimum detection time of 30 hours and 1 seconds, respectively.

7 THREATS TO VALIDITY

The effectiveness of *oo7* depends on the following crucial factors:

1. The current implementation of *oo7* does not accurately track the control dependency. As a result, we might miss some Spectre vulnerabilities, as observed during our evaluation. We consider this threat to be attributed to the coverage and precision of taint analysis. In future, a more accurate taint analysis engine will automatically benefit our proposed *oo7* approach.
2. *oo7* relies on BAP, which, in turn incorporates a taint analysis engine. The taint analysis statically interprets the code by unrolling loops up to a certain depth. For optimistic loop unrolling, therefore, our *oo7* approach might introduce false negatives. However, with correct or pessimistic loop bound supplied to BAP, such a threat to our approach can be eliminated.
3. *oo7* needs to set the speculation window *SEW* for analysis. We note that *SEW* depends on the execution platform. Hence, incorrect configuration of *SEW* may result in both false positive and false negatives. In our experiments, we set *SEW* to be the maximum length of speculation window (bounded by the size of reorder buffer) in commodity `x86` processors.
4. Our repair strategy works on modifying the source code. To this end, we need to map the binary code with the respective source code fragments. In the absence of source code, *oo7* will not be able to fix the Spectre vulnerabilities. Nevertheless, it will still be able to detect the vulnerabilities for arbitrary binary code. This can be used to send reports to the developers when source code is unavailable.

8 COMBATTING MELTDOWN

Meltdown is a recently discovered vulnerability which can exploit the side effects of out-of-order-execution [9]. Meltdown does not rely on the software vulnerabilities and it can be launched directly from the attacker code. Although the primary objective of *oo7* is to detect Spectre vulnerability, we can easily adapt to detect meltdown signatures in potentially attack code.

For launching Meltdown, the attacker aims to load value from a kernel address ka. This, of course, would result in an exception. However, exploiting the out-of-order paradigm of execution, the value from the address ka might already be brought into the cache

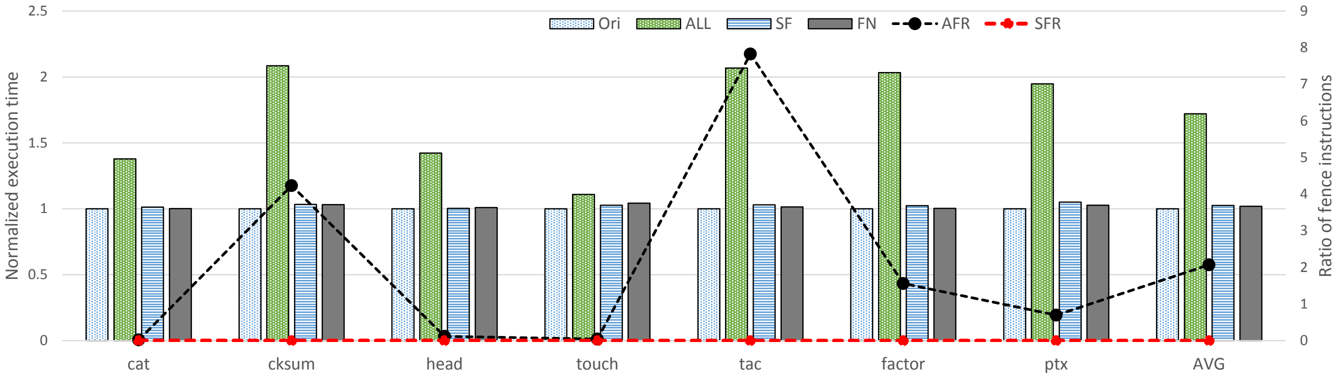


Figure 8: Comparing the runtime overhead due to mitigation introduced by *oo7*. Ori=Original program, ALL=inserting fences after all conditional branches, SF=inserting fences after Spectre branches, FN=inserting `NOP` instructions after Spectre branches, AFR= the ratio of fence instructions in execution of ALL, SFR=the ratio of fence instructions in execution of SF.

before the exception is raised. This cached value, then, can be exfiltrated via standard cache side-channel attacks (e.g. using a probe array `array2` in the Spectre code).

Let us assume that *KA* captures the set of sensitive addresses that the attacker does not have permission to access. For example, *KA* might capture the set of kernel memory addresses. A meltdown signature is detected if a load instruction *L1* points to an address in *KA* and a memory access *IM1* is data-dependent on *L1*. Specifically, the detection of Meltdown is captured via the condition $\Phi_{meltdown}$

$$\boxed{\begin{aligned}\Phi_{meltdown} \equiv load(L1) \wedge mem(IM1) \wedge \\ addresss(L1) \cap KA \neq \emptyset \wedge \\ Dep(IM1, L1)\end{aligned}} \tag{3}$$

where $addresss(L1)$ captures the set of addresses accessed by load instruction *L1*. We use value set analysis [11] to detect the set of values/addresses accessed by an instruction. This computes a sound over-approximation of $addresss(L1)$. If any kernel address belongs to the set $addresss(L1)$, then the program dependency graph is checked to discover *IM1* dependent on *L1*.

We note that the original meltdown code involves `flush` instruction (to flush the probe array from the cache) and `RETSC` instruction (to time the access of probe array during *IM1* access). Detecting the presence of these instructions is straightforward and we remove it from $\Phi_{meltdown}$ for brevity.

9 CONCLUSION

In this report, we have designed, developed and evaluated *oo7* for detecting Spectre vulnerabilities in arbitrary binary code. Our approach is employed post-compilation to take into account all compiler optimizations. We envision that a systematic analysis is crucial both for detecting Spectre vulnerabilities and to repair them with minimal performance overhead.

For detecting Spectre vulnerabilities in the wild and promote further research in the area, we have made our Spectre vulnerability detection code accessible via the following password protected web-site (password to access the website is available on request). http://www.comp.nus.edu.sg/~abhik/ftp/oo7/

ACKNOWLEDGMENTS

This research is supported in part by the National Research Foundation, Prime Minister's Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate. The research is also partially supported by SUTD research grant no. SRIS17123.

REFERENCES

[1] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. BAP: A binary analysis platform. In *International Conference on Computer Aided Verification*, pages 463–469. Springer, 2011.
[2] Microsoft community. C++ developer guidance for speculative execution side channels. https://docs.microsoft.com/en-us/cpp/security/developer-guidance-speculative-execution, 2018.
[3] GNU. Coreutils. https://github.com/coreutils/coreutils, 2018.
[4] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2017.
[5] John L Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.
[6] Paul Kocher. Spectre mitigations in microsoft's C/C++ compiler. https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html.
[7] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.
[8] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.
[9] 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.
[10] 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, pages 190–200, New York, NY, USA, 2005. ACM.
[11] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.
[12] Yuval Yarom and Katrina Falkner. Flush+ reload: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium*, volume 1, pages 22–25, 2014.