A Formal Approach to Secure Speculation

Kevin Cheang Cameron Rasmussen Sanjit A. Seshia
{*kcheang,crasmussen,sseshia*}*@berkeley.edu*
University of California, Berkeley

Pramod Subramanyan
spramod@cse.iitk.ac.in
Indian Institute of Technology, Kanpur

Abstract—**Transient execution attacks like Spectre, Meltdown and Foreshadow have shown that combinations of microarchitectural side-channels can be exploited to create side-channel leaks that are greater than the sum of their parts. While both hardware and software mitigations have been proposed against these attacks, provable security has remained elusive.**

This paper introduces a formal methodology for enabling secure speculative execution on modern processors. We propose a new class of information flow security properties called trace property-dependent observational determinism (TPOD). We use this class to formulate a secure speculation property. Our property formulation and associated adversary models help formalize the class of transient execution vulnerabilities. We demonstrate applicability of our methodology by verifying secure speculation for several illustrative programs.

I. INTRODUCTION

Recently discovered transient execution attacks like Spectre, Meltdown and Foreshadow [1–13] have shown that side channel vulnerabilities are more exploitable than was previously believed. While caches or branch predictors leaking information is not exactly news [14–20] in 2019, Spectre is interesting because it combines side channels to produce a leak that is "greater than the sum of its parts." A number of mitigations have been proposed to these vulnerabilities [9–12, 21–25] and many of these have been adopted into widely-used software like the Linux kernel [26, 27], Microsoft Windows [28, 29] and the Microsoft Visual Studio compilers and associated libraries [23]. However, these mitigations are not provably secure and in fact some, e.g. Spectre mitigations in Microsoft Visual Studio, are known to be incomplete [30].

Transient execution attacks exploit microarchitectural side-channels present in modern high-performance processors. High-performance processors contain several microarchitectural optimizations — e.g., branch prediction, data and instruction caching, out-of-order execution, and speculative memory address disambiguation, to name just a few — in order to execute programs more efficiently [31, 32]. Many of these optimizations rely on the technique of speculation [33–36]. The processor uses a prediction structure to guess whether a particular execution is likely to occur before its results are available and speculatively executes as per the prediction. If the prediction turns out to be wrong, architectural state – which consists of register and memory values – is restored to its value before speculative execution started and execution restarts along the correct path. In many cases, it is possible to build predictors that mostly guess correctly and speculation leads to huge performance and power benefits.

It is important to emphasize that when misspeculation is resolved, only architectural state is restored while *microarchitectural* state, such as cache and branch predictor state, is *not*. Transient execution attacks exploit this fact by mistraining a prediction structure to speculatively execute vulnerable wrong-path instructions and exfiltrate confidential information by examining the microarchitectural side-effects of misspeculation.

The above leads to two obvious templates for preventing these vulnerabilities: (i) do not speculate, or (ii) do not leak information through microarchitectural side channels. Many mitigations do indeed take the first approach by turning off speculation in a targeted manner [9, 10, 21, 24, 26–29]. While most of these mitigations were developed through careful manual analysis of known exploitable vulnerabilities, automated tools for Spectre mitigation also take this approach [23]. Unfortunately, the latter have been found to be incomplete [30] while the former do not come with provable security guarantees. The larger point here is that there is no formal methodology for reasoning about the security of mitigations to transient execution vulnerabilities.

Some research has also taken the second approach of attempting to close the exfiltration side-channel by ensuring it does not leak any information at all. For instance the Dynamically Allocated Way Guard (DAWG) closes the cache side-channel by partitioning between protection domains [37]. However, other side-channels (prefetchers, DRAM row buffers, load store queues, etc.) potentially remain exploitable with these solutions and partitioning comes with a significant performance penalty. Here too, it remains unclear whether partitioning a few exfiltration channels is sufficient to prevent all transient execution vulnerabilities.

Besides the lack of provable security, another problem with current approaches are their large performance penalties. In this context, it is noteworthy that recent versions of the Linux Kernel have turned off certain Spectre mitigations by default because performance slowdowns of up to 50% [38, 39] were observed for certain workloads. We believe these high overheads are a result of being unable to reason about security of the mitigations. If we could systematically reason about security, it will be possible to develop more aggressive mitigations that disable speculation in a very targeted manner and have a much lower performance overhead.

All of the above points to the need for verification techniques for secure speculation. This problem is most closely related to the secure information flow problem, which has been studied by a rich body of literature [40–46]. Unfortunately, existing work on secure information flow is not sufficient to precisely capture the class of transient execution vulnerabilities. Specifically, it is important to note that traditional notions of information flow security like non-interference [41] and observational determinism [42–44] are only satisfied when there is *no information flow* from confidential state to adversary observable state. In the context of Spectre, this would imply no information flow from confidential memory locations to microarchitectural side channels. For most programs of interest, e.g., the Linux kernel and Microsoft Windows operating system, all modern commercial processors *do* leak information about confidential operating system state through microarchitectural side channels like caches, prefetchers, DRAM row buffers, etc. Therefore, traditional formulations of secure information flow are always violated for such programs regardless of whether they are vulnerable to transient execution attacks.

The above points to one of the key challenges in the verification of secure speculation: *formulating the right property.* We need a way of precisely capturing only the *new* leaks introduced by the interaction of microarchitectural side channels with speculation. These new leaks stand in contrast to the previously known side-channel leaks which are already captured via traditional notions of secure information flow such as noninterference/observational determinism.

A second important challenge is coming up with a general system and adversary model that can be used to reason about the category of transient execution attacks, as opposed to pattern-matching known vulnerabilities. Thirdly, we need a verification methodology that can be used to prove that specific programs satisfy secure speculation.

In this paper, we address each of the above challenges. We introduce a formal methodology for reasoning about security against transient execution attacks. Our approach is based on the formulation of a new class of information flow security properties called *trace property-dependent observational determinism* (TPOD). These properties, an extension of observational determinism, are defined with respect to a trace property and intuitively TPOD captures the following notion of security: *does violation of the trace property introduce new counterexamples to observational determinism?*

We use TPOD to reason about the security of software-based Spectre mitigations. For this, we present an assembly intermediate representation (AIR) into which machine code can be lifted and introduce speculative operational semantics for this AIR. We introduce a general adversary that captures transient execution attacks, and define a secure speculation property against this adversary as an instance of TPOD. We verify secure speculation in an automated fashion using bounded model checking and induction in the UCLID5 verification tool [47, 48] on a suite of small but illustrative benchmarks, several of which are from the literature on Spectre mitigations [30].

A. Contributions

This paper's contributions are the following.

- We introduce a novel methodology for reasoning about the security of microarchitectural speculation mechanisms. Our methodology can prove that a program is secure against transient execution vulnerabilities.
- We introduce a new class of information-flow security properties called trace property-dependent observational determinism. This class of properties allows us to reason about information leaks that occur due to interactions between microarchitectural mechanisms.
- We introduce a speculative operational semantics for an assembly intermediate representation, an adversary model for transient execution attacks over this representation and a secure speculation property. Violations of the property correspond to transient execution vulnerabilities.
- We demonstrate our methodology by automatically proving secure speculation for an suite of illustrative programs.

The rest of this paper is organized as follows. Section II presents an overview of transient execution attacks. Section III reviews observational determinism and introduces trace property-dependent observational determinism. Section IV describes the assembly intermediate representation and speculative operational semantics for it. The adversary model and the secure speculation property are described in Section V. Sections VI and VII present our verification approach and case studies. Section VIII reviews related work and Section IX provides some concluding remarks.

II. OVERVIEW

In this section, we present an overview of transient execution vulnerabilities as exemplified by Spectre and review the verification challenges posed by these vulnerabilities.

A. Introduction to Transient Execution Attacks

Transient execution attacks involve two components: an untrusted component (the attacker) who interacts with a trusted component (the victim) over some communication interface. The attacker exfiltrates confidential information from the victim by exploiting microarchitectural artifacts of misspeculation in high-performance processors. As shown in Figure 1, a transient execution attack has four stages. We explain these four stages using the code snippet shown in Figure 3(a), which is vulnerable to Spectre variant 1.

(S1) Prepare: In the first stage, the attacker prepares the exfiltration side channel and mistrains the branch predictor in

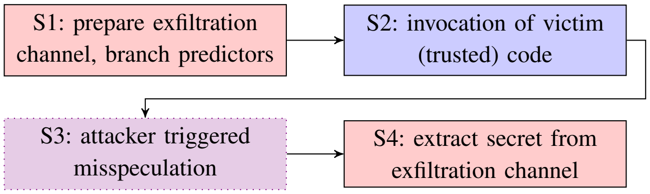


Figure 1: Four Stages of a Speculative Execution Attack. Execution of untrusted code is shown in red, while trusted code is in blue. We show the attacker-triggered misspeculation in the trusted code in the violet dotted box.

order to bring the system into a vulnerable state. A commonly used exfiltration side channel is the cache. In this context, preparation refers to priming the cache by executing load/store instructions with effective addresses that are stored in the same cache sets as the victim data. One way to mistrain the branch predictor is to repeatedly execute the victim code with carefully chosen input arguments so that predictor learns that a particular branch should always be taken (or not taken).

(S2) Invocation: In the second stage, the attacker invokes victim code with carefully chosen input arguments to trigger misspeculation. This invocation occurs over some communication interface between the untrusted and trusted code. One such interface is through system calls and returns; here the attacker is an untrusted user-mode process while the victim is the operating system kernel. Another example in the context of browser-based sandboxing, e.g. Native Client [49], would be function calls and returns. The attacker is untrusted code running within the sandbox while the victim is NaCl's trusted API. Many other vulnerable interfaces exist: hypercalls, enclave entry, software interrupts, etc. The victim may not even be explicitly invoked: implicit invocation is possible by mistraining the branch predictor or by causing a hardware interrupt to occur! For simplicity, this paper focuses on a function call/return interface but our techniques are easily generalized to other interfaces.

(S3) Exploitable Misspeculation: The victim code now executes. At some point it will misspeculate in an attacker-controlled manner resulting in the execution of "wrong path" instructions. These wrong path instructions update speculative architectural state – register and memory values – and microarchitectural state including caches, branch predictors and prefetchers. Eventually the wrong path is resolved and its instructions are flushed. Speculative updates to architectural state (registers and memory) are flushed, but microarchitectural state (e.g., cache updates) is not restored.

While many past attacks have exploited microarchitectural side channels to extract confidential data [14–20], the difference with transient execution vulnerabilities is that the latter only manifest due to misspeculation in the processor. *Even programs whose architectural (non-speculative) execution is carefully designed to not have any side-channel leaks could be vulnerable to transient execution attacks.*

(S4) Exfiltration: Finally, control returns to the attacker who examines microarchitectural side-channel state to exfiltrate confidential data from the victim. In cache-based attacks, this involves *probing* the cache in order to infer secrets.

B. Spectre Variants and Associated Verification Challenges

We now describe the Spectre variant 1 vulnerability and a few modifications to it as exemplars of transient execution attacks. We use this discussion to motivate the research challenges posed by transient execution attacks. While we focus on Spectre variant 1 for ease of exposition, the research questions raised here apply to all other transient execution attacks.

We discuss the four code snippets shown in Figure 3. In each of the snippets, the vulnerable victim function is `foo`. This function is trusted but is invoked by an untrusted attacker with an arbitrary attacker chosen argument `i`. `foo` has access to two arrays: `a1` and `a2`. Note that any architectural execution of these functions should never see accesses to `a1[i]` for `i` $\geq$ `N`. Therefore, one might expect that no information could possibly leak about these values in the array through any side-channel. As we will see, the Spectre attack shows how these values can be inferred by a clever attacker.

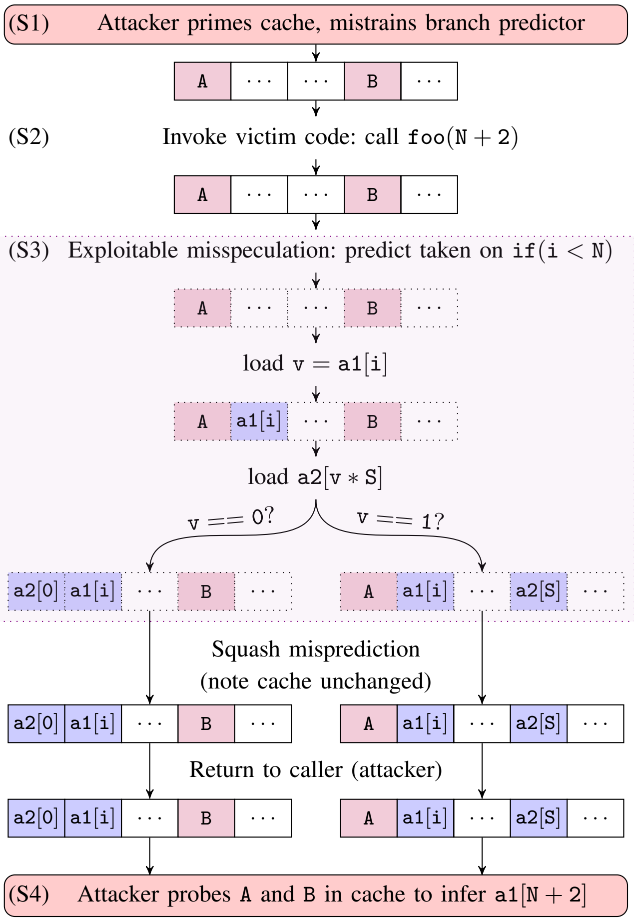


Figure 2: Cache state evolution in Spectre variant 1. The rectangular boxes show the addresses that are cached. Untrusted accesses are red while accesses by trusted code are blue. For simplicity, we show the attack on a direct-mapped cache.

1) Spectre Variant 1: Figure 3a shows a snippet of code that demonstrates vulnerability to the Spectre variant 1 attack [2, 3]. To help explain the vulnerability, we show how cache state evolves during each stage of the attack in Figure 2.

S1 First, the attacker sets up ("primes") the cache by bringing two addresses `A` and `B` into the cache. These addresses are carefully chosen so as to reside in the same cache set as the subsequently-accessed addresses `a2[0]` and `a2[S]` respectively. Next, the attack mistrains the predictor to speculate that the branch on line 4 will be taken. Now, the attack is ready to be launched.

S2 The attacker invokes `foo` with an argument `i` $=$ `N` $+ 2$.

```
1  uint8_t a1[M];
2  uint8_t a2[P];
3  uint8_t foo(unsigned i) {
4    if (i < N) {
5      uint8_t v = a1[i];
6      return a2[v*S];
7    }
8    return 0;
9  }
```

(a) Spectre v1 vulnerability.

```
1  uint8_t a1[M];
2  uint8_t a2[P];
3  uint8_t foo(unsigned i) {
4    if (i < N) {
5      _mm_lfence();
6      return a2[a1[i]*S];
7    }
8   return 0;
9  }
```

(b) Fix for Spectre v1.

```
1  uint8_t a1[M];
2  uint8_t a2[P];
3  uint8_t foo(unsigned i) {
4    if (i < N) {
5      uint8_t v = a1[0];
6      return a2[v*S]+i;
7    }
8    return 0;
9  }
```

(c) Conditionally vulnerable variant.

```
1  uint8_t a1[M], a2[P];
2  uint8_t foo(unsigned i) {
3    if (i < N) {
4      uint8_t v = a1[i];
5      _mm_lfence();
6      return a2[v*S];
7    }
8   return 0;
9  }
```

(d) Another fix for Spectre v1.

Figure 3: Illustrative examples for verification of secure speculation. In all code snippets assume that `M` $>$ `N` and that argument `i` is an untrusted (low-security) input to the trusted (high-security) function `foo`.

S3 The argument `i = N + 2` along with branch predictor mistraining in S1 triggers a misspeculation on line 4. This results in `a1[N+2]` and `a2[a1[N+2]*S]` being speculatively brought into the cache. Eventually, the processor realizes that the branch prediction was incorrect and "undoes" modifications to architectural state, but cache state is *not* restored.

S4 In the final stage, the attacker exploits the fact that the address brought into the cache on line 6 depends on the *value* (not address) of `a1[N+2]`. The attacker determines this address by loading `A` and `B`. One of these will miss in the cache and this timing channel allows the attacker to infer the value of `a1[N+2]`.

2) Fixes to Spectre Variant 1: As the leaks in Spectre are due to interactions between the branch predictor and the cache, a straightforward fix is to prevent speculation. We can make the code in Figure 3a secure by inserting a *load fence* [9, 22] as shown in Figures 3b and 3d. Figure 3b is easy to understand: the load fence on line 5 ensures that no memory accesses are made until the processor is sure that the branch will be taken.

Figure 3d is slightly more involved. The load fence executes after the first load and before the second load. At first glance, it may appear to be insecure, because `a1[i]` can still be brought into the cache speculatively. However `i` is attacker-chosen while the base address of `a1` can also be inferred by the attacker. Therefore, bringing `a1[i]` into the cache leaks no additional information. Figure 3d is secure.

3) Conditional Vulnerability: Figure 3c presents an interesting variation of Figure 3a. In this case, the first memory load always accesses `a1[0]`. Since this value is leaked through the cache (when `i` $<$ `N`) even without misspeculation, it would seem that this code is not vulnerable to transient execution attacks. However, if `N` $= 0$, then `a1[0]` should not be accessed. But the attacker can mistrain the branch predictor to predict that the branch on line 4 is taken[1] and then infer the value of `a1[0]`. This code exhibits transient execution vulnerabilities when $\mathtt{N} = 0$ but not when $\mathtt{N} > 0$!

4) Verification Challenges: In Figure 3a, information about `a1[i]` leaked when $\mathtt{i} = \mathtt{N} + 2$. For this value of `i`, `foo` should not have performed any memory/cache accesses. This points to one challenge in verifying secure speculation: the verification model needs to capture *interactions* between *microarchitectural side-channels* to detect leaks.

Another challenge is demonstrated by Figures 3b, 3c and 3d. Identifying the vulnerability requires precise semantic analysis of program behavior. Simply matching vulnerable code patterns (e.g., branches followed by dependent loads) results in both false positives and negatives.

Finally, it is important to note that the secure versions `foo` in Figures 3b and 3d do not satisfy traditional notions of information flow security [40]: noninterference [41] or observational determinism [42–44] because there *is* information flow from `a1` to the cache side-channel even if the function is executed on a processor without a branch predictor.

III. SPECIFICATION USING TRACE PROPERTY-DEPENDENT OBSERVATIONAL DETERMINISM

To address the challenges raised in § II-B4, this paper formulates a secure speculation property that precisely captures transient execution vulnerabilities. Toward this end, in this section we first review observational determinism [42–44], a class of security properties that can capture certain

---

[1] One way to do this might be to exploit aliasing in branch predictor indexing by training a different branch which maps to the same predictor index.

$$\begin{tikzcd}
\pi_1^0 \arrow[r, "L"] \arrow[d, phantom, "\approx_{\mathcal{L}}"] & \cdots & \pi_1^i \arrow[r, "H_1"] \arrow[d, phantom, "\approx_{\mathcal{L}}"] & \pi_1^{i+1} \arrow[r, "L"] \arrow[d, phantom, "\approx_{\mathcal{L}}"] & \pi_1^{i+2} \arrow[d, phantom, "\approx_{\mathcal{L}}"] & \cdots & \pi_1^j \arrow[r, "H_1"] \arrow[d, phantom, "\approx_{\mathcal{L}}"] & \pi_1^{j+1} \arrow[r, "L"] \arrow[d, phantom, "\approx_{\mathcal{L}}"] & \cdots \\
\pi_2^0 \arrow[r, "L"] & \cdots & \pi_2^i \arrow[r, "H_2"] & \pi_2^{i+1} \arrow[r, "L"] & \pi_2^{i+2} & \cdots & \pi_2^j \arrow[r, "H_2"] & \pi_2^{j+1} \arrow[r, "L"] & \cdots
\end{tikzcd}$$

Figure 4: Illustrating observational determinism: low instructions are labelled L, while high instructions are labelled H_1 and H_2, proof obligations are shown in green and assumptions are shown in blue.

notions of confidentiality. We then motivate and describe trace property-dependent observational determinism: a novel class of information flow properties that includes secure speculation.

A. Preliminaries

We model system behavior using traces which are a sequence of system states. The definition of system states is left abstract for now. We refer to traces using π, π_1, π_2, etc. and states by s, s_0, s_1, etc. The notation π^i refers the ith element of the trace; e.g., if $\pi = \langle s_0, s_1, s_2, s_3, s_4, \ldots \rangle$, then $\pi^3 = s_3$.

We consider concurrent systems consisting of two components: an untrusted low-security component and a trusted high-security component. These components interact via some interface (e.g., system calls and returns) which prompt transitions from the low component to the high component or vice versa. A typical confidentiality requirement is that the low component must not be able to distinguish between secret states of the high component.

1) Low-Equivalence of States: The above notion of indistinguishability is expressed via low-equivalence of states. We say that two states s and s' are low-equivalent if they are indistinguishable to the low component. This is denoted by $s \approx_{\mathcal{L}} s'$. Like system states, the definition of low-equivalence is left abstract for now. Low-equivalence is extended to traces in the obvious way. Two traces are low-equivalent if all their states are low-equivalent: $\pi_1 \approx_{\mathcal{L}} \pi_2$ if $\forall i.\ \pi_1^i \approx_{\mathcal{L}} \pi_2^i$.

2) Modeling Computation: The system computes by identifying an operation to execute and transitioning to the next state based on its transition relation $\rightsquigarrow$. If system state s_i can transition to state s_j, then $(s_i, s_j) \in \rightsquigarrow$, which we write as $s_i \rightsquigarrow s_j$. As with states, we leave $\rightsquigarrow$ abstract for now.

The operation executed by the low component in a particular state s is denoted by $op_{\mathcal{L}}(s)$; $op_{\mathcal{L}}(s)$ is $\perp$ if the low component is not being executed in state s. Similarly, the operation executed by the high component in the state s is denoted by $op_{\mathcal{H}}(s)$. We will overload notation and refer to $op_{\mathcal{L}}(\pi)$ and $op_{\mathcal{H}}(\pi)$ to denote the trace of operations executed by the low and high components respectively in π.

B. Observational Determinism

A system satisfies observational determinism if for every pair of traces of the system such that: (i) the two traces' initial states are low-equivalent, and (ii) the low operations executed at every step of the two traces are identical, then the two traces are also low-equivalent. Equation 1 shows this definition.

$$\forall \pi_1, \pi_2. \quad (1)$$
$$\left(\pi_1^0 \approx_{\mathcal{L}} \pi_2^0 \wedge op_{\mathcal{L}}(\pi_1) = op_{\mathcal{L}}(\pi_2)\right) \implies \left(\pi_1 \approx_{\mathcal{L}} \pi_2\right)$$

Observational determinism is shown pictorially in Figure 4. The figure shows a pair of traces with their initial states being low-equivalent. In subsequent steps, the low operations are identical and this is denoted by labelling low transitions as L. However, high operations may differ between the traces, so we label its transitions as H_1 and H_2. Observational determinism holds if every corresponding pair of states in these two traces are low-equivalent. A violation of observational determinism is some sequence of low operations that can distinguish between some two secret states of the high component.

1) Limitations of Observation Determinism for Secure Speculation: As a strawman proposal, consider an observational determinism property that attempts to capture secure speculation by requiring that the trace of memory accesses by function `foo` in Figure 3a be identical for all pairs of invocations where the untrusted argument i is equal.

Two such pairs of traces are shown in Figure 6. `N=4` in both pairs; in (a), `i=0` and the program does not misspeculate while in (b), `i=5` and the program misspeculates. The values v_1 and v_2 correspond to the confidential data stored at the location `a1[i]`. We see that the property is violated in (b) as the traces of memory addresses differ if $v_1 \neq v_2$. This violation is due to the transient execution vulnerability. It is also violated in (a) because the program leaks `a1[i]` even though there is no misspeculation. The larger point is that observational determinism can capture transient execution vulnerabilities only if the program satisfies the observational determinism property – has zero violations of the property – in the absence of misspeculation. Most programs of interest (e.g., the Linux kernel) do not satisfy such a property. Applying the strawman methodology to these programs results in a flood of counterexamples to observational determinism that are completely unrelated to speculation, rendering the methodology useless.

We wish isolate violations of observational determinism solely caused by the satisfaction/violation of a particular property of the trace (e.g., misspeculation). As noted above, observational determinism does not allow us to do this generally for different programs.[2] In the following, we capture this security requirement in the form of a 4-safety property to isolate these trace property-dependent violations.

---

[2] We explain this further in Section VI.

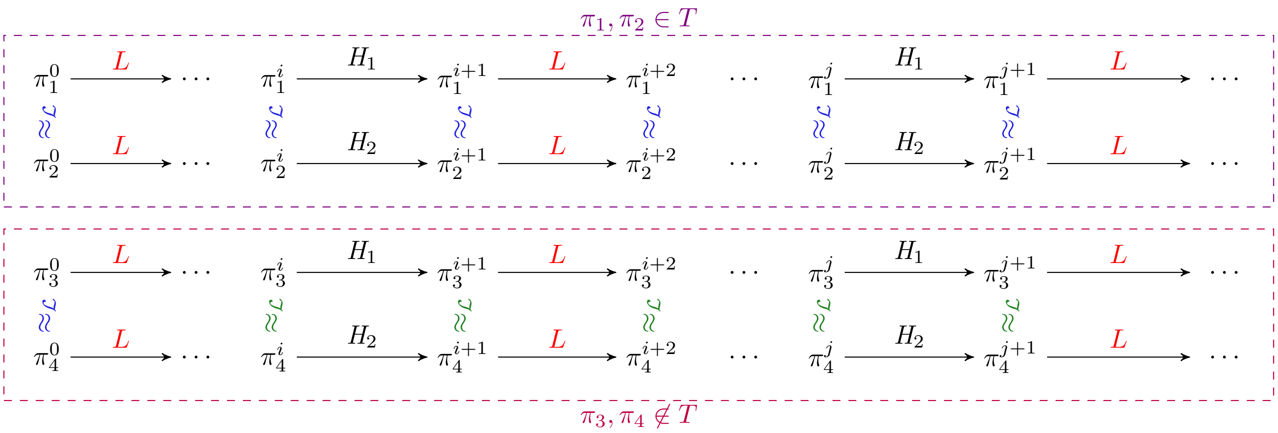


Figure 5: Illustrating trace property-dependent observational determinism. As in Figure 4 low instructions are labelled L, while high instructions are labelled H_1 and H_2, proof obligations are shown in green and assumptions are shown in blue.

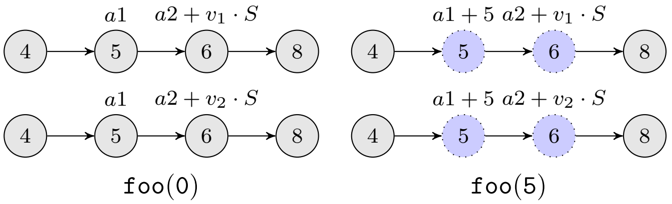


Figure 6: Illustrating the strawman observational determinism property for Figure 3a. Numbers within each state refer to program counter values (shown as line numbers). Labels above each state indicates the data memory address accessed (if any). States shown in dotted circles are specuative states.

C. Trace Property-Dependent Observational Determinism

In a processor that never misspeculates – either because it does not have a branch predictor or because the branch predictor is perfect – there is no information leakage due to transient execution. Therefore, finding transient execution vulnerabilities is equivalent to finding information leaks that would not have occurred in the absence of misspeculation.

1) Definition of TPOD: To formulate the above notion of information leakage, we introduce a class of information flow properties called trace property-dependent observational determinism (TPOD), a hyperproperty over four traces that is defined with respect to a trace property. Let the four traces be π_1, π_2, π_3, π_4, and the trace property T.

Suppose the following assumptions hold:

1) traces π_1 and π_2 satisfy the trace property T,
2) traces π_3 and π_4 do not satisfy the trace property T,
3) all four traces execute the same low operations,
4) traces π_3 and π_4 execute the same high operations as π_1 and π_2 respectively,
5) traces π_1 and π_2 are low-equivalent and the initial states of π_3 and π_4 are low-equivalent.

Then, TPOD is satisfied if π_3 and π_4 are low-equivalent. High operations in π_1, π_3 and π_2, π_4 respectively must be identical; they are not necessarily identical in π_1, π_2 or π_3, π_4.

$$
\begin{aligned}
&\forall \pi_1, \pi_2, \pi_3, \pi_4 . \\
&\quad \pi_1 \in T \wedge \pi_2 \in T \wedge \pi_3 \notin T \wedge \pi_4 \notin T && \Longrightarrow \\
&\quad op_{\mathcal{L}}(\pi_1) = op_{\mathcal{L}}(\pi_2) = op_{\mathcal{L}}(\pi_3) = op_{\mathcal{L}}(\pi_4) && \Longrightarrow \\
&\quad op_{\mathcal{H}}(\pi_1) = op_{\mathcal{H}}(\pi_3) \wedge op_{\mathcal{H}}(\pi_2) = op_{\mathcal{H}}(\pi_4) && \Longrightarrow \\
&\quad \pi_1 \approx_{\mathcal{L}} \pi_2 \wedge \pi_3^0 \approx_{\mathcal{L}} \pi_4^0 && \Longrightarrow \\
&\quad \pi_3 \approx_{\mathcal{L}} \pi_4 && (2)
\end{aligned}
$$

TPOD is shown in Equation 2 and depicted in Figure 5.[3] A violation of TPOD corresponds to a sequence of low operations that were unable to distinguish between high states when the trace property T was satisfied, but *are* able to distinguish between high states when T is not satisfied. In other words, violation of the trace property T introduced a new counterexample to observational determinism.

2) Refinement and TPOD: In general, hyperproperties may not be preserved by refinement [43]. However, as we show below TPOD is subset-closed: if any set of traces satisfies TPOD, then every subset of this set also satisfies TPOD.

Lemma 1: Trace property-dependent observational determinism is a subset-closed hyperproperty.

Subset-closed hyperproperties are important because they are preserved by refinement [40]. This means that one can prove TPOD on an abstract system, and through iterative refinement show that TPOD holds on a concrete system that is a refinement of the abstract system. Therefore, TPOD can potentially be scalably verified on complex systems.

Corollary 1: Trace property-dependent observational determinism is preserved by refinement.

A minor extension to template shown in Equation 2 is to consider an antecedent trace property U that must be satisfied

---

[3] We follow the convention that the implication operator is right-associative.

by all traces. The trace property U may be used to model constraints on valid executions.

$$
\begin{aligned}
&\forall \pi_1, \pi_2, \pi_3, \pi_4. \\
&\quad \pi_1 \in U \wedge \pi_2 \in U \wedge \pi_3 \in U \wedge \pi_4 \in U && \implies \\
&\quad \pi_1 \in T \wedge \pi_2 \in T \wedge \pi_3 \notin T \wedge \pi_4 \notin T && \implies \\
&\quad op_{\mathcal{L}}(\pi_1) = op_{\mathcal{L}}(\pi_2) = op_{\mathcal{L}}(\pi_3) = op_{\mathcal{L}}(\pi_4) && \implies \\
&\quad op_{\mathcal{H}}(\pi_1) = op_{\mathcal{H}}(\pi_3) \wedge op_{\mathcal{H}}(\pi_2) = op_{\mathcal{H}}(\pi_4) && \implies \\
&\quad \pi_1 \approx_{\mathcal{L}} \pi_2 \wedge \pi_3^0 \approx_{\mathcal{L}} \pi_4^0 && \implies \\
&\quad \pi_3 \approx_{\mathcal{L}} \pi_4 && (3)
\end{aligned}
$$

This version of TPOD is shown in Equation 3. This extension is also subset-closed and preserved by refinement

IV. FORMAL MODELING OF SPECULATION

We now turn to the problem of formulating and reasoning about secure speculation. This requires the construction of a system model that captures speculative execution. Toward this end, this section describes an assembly intermediate representation and introduces speculative operational semantics for it.

A. System Model

Reasoning about secure speculation must be done using assembly language instructions, not in a high-level language because compiler optimizations may introduce branches where none exist in the source program, or may eliminate branches in the source program by turning them into conditional moves. That said, reasoning about a specific instruction set architecture (ISA) is cumbersome and gives little additional insight into the fundamental causes of transient execution vulnerabilities. Therefore, we present an assembly intermediate representation (AIR) that ISAs can be *lifted* into. We model speculation over the AIR by introducing a speculative operational semantics for it.[4]

$$
\begin{array}{lcl}
\langle \mathit{program} \rangle & ::= & \langle \mathit{instr} \rangle * \\
\langle \mathit{instr} \rangle & ::= & \langle \mathit{reg} \rangle := \langle \mathit{exp} \rangle \\
& | & \langle \mathit{reg} \rangle := \texttt{mem}[\langle \mathit{exp} \rangle] \\
& | & \texttt{mem} := \texttt{mem}[\langle \mathit{exp} \rangle \rightarrow \langle \mathit{exp} \rangle] \\
& | & \texttt{if}\ \langle \mathit{exp} \rangle\ \texttt{goto}\ \langle \mathit{const} \rangle \\
& | & \texttt{goto}\ \langle \mathit{const} \rangle \\
& | & \texttt{specfence} \\
\langle \mathit{exp} \rangle & ::= & \langle \mathit{const} \rangle \mid \langle \mathit{reg} \rangle \mid \Diamond_u \langle \mathit{exp} \rangle \mid \langle \mathit{exp} \rangle\ \Diamond_b \langle \mathit{exp} \rangle
\end{array}
$$

Figure 7: The Assembly Intermediate Representation (AIR). $\Diamond_u$ and $\Diamond_b$ are typical unary and binary operators respectively.

1) Assembly Intermediate Representation (AIR): The AIR shown in Figure 7. A program is a list of instructions. Instructions are one of the following types:

- updates to registers,
- loads from memory,
- stores to memory,
- conditional and unconditional jumps,
- speculation fences.

The first five types of instructions are standard. We introduce a *speculation fence* instruction which causes the processor to not fetch any more instructions until all outstanding branches are resolved. The load fence instructions in Figures 3b and 3d are modelled as speculation fences because the relevant aspect of these fences for this paper is that they stop speculation.

Note that jump targets must be constants in AIR. This is intentional and precludes the verification of programs using indirect jumps and returns in the current version of our verification tool. We do this to simplify the operational semantics for speculative execution. Modeling speculative execution of indirect jumps and returns requires modeling indirect branch predictors, branch target buffers and the return address stack. Introducing these structures into our operation semantics is conceptually straightforward but runs into scalability limitations during verification. We plan to extend the operational semantics to include these instructions while addressing scalability in future work.[5]

$$
\textsc{Const}\ \frac{}{\Delta, n \vdash c \Downarrow c} \qquad \textsc{Reg}\ \frac{\Delta[n, r] = v}{\Delta, n \vdash r \Downarrow v}
$$

$$
\textsc{UnOp}\ \frac{\Delta, n \vdash e \Downarrow v' \qquad \Diamond_u v' = v}{\Delta, n \vdash \Diamond_u e \Downarrow v}
$$

$$
\textsc{BinOp}\ \frac{\Delta, n \vdash e_1 \Downarrow v_1 \qquad \Delta, n \vdash e_2 \Downarrow v_2 \qquad v_1 \Diamond_b v_2 = v}{\Delta, n \vdash e_1 \Diamond_b e_2 \Downarrow v}
$$

Figure 8: Semantics of expression evaluation

"Flattening" indirect jumps and returns into a sequence of direct jumps is similar in principle to control-flow integrity (CFI) checks [52, 53]. Since secure programs will likely be implementing CFI anyway, we assert compilers can be modified in straightforward ways to produce code without indirect jumps and returns (with some performance cost).

2) Operational Semantics for AIR: In Figures 8 and 9, we introduce operational semantics for **speculative in-order** processors. We model speculation in the branch predictor for direct conditional branches. Other sources of misspeculation such as value prediction and memory address disambiguation are not considered in this model. Extending the semantics to include these is conceptually straightforward. However, this could result in models that are difficult to analyze using auto-

---

[4] The AIR itself is based on the binary analysis platform (BAP) intermediate language (IL) [50]; speculative operational semantics for it are novel. "Lifters" from x86 and ARM binaries to BAP can be found at [51].

[5] It is important to note that our exclusion of indirect jumps does not mean our verifier leaves programs vulnerable to Spectre variant 2. In programs without indirect branches, all indirect branch mispredictions will be redirected at decode, long before execution or memory access.

$$\textsc{RegisterUpdate}\ \frac{\begin{array}{c}\Delta, n \vdash e \Downarrow v \quad \Delta' = \Delta[(n,r) \to v] \quad \rho = pc[n] \quad \iota = \Pi[\rho] \quad pc' = pc[n \to \rho + 1] \\ \rho' = pc'[n] \quad \iota' = \Pi[\rho'] \quad \omega' = \omega.\langle \rho, \bot \rangle \quad \neg \mathit{resolve}(n, \beta, pc)\end{array}}{\Pi, \Delta, \mu, pc, \omega, \beta, n, r := e \quad \rightsquigarrow \quad \Pi, \Delta', \mu, pc', \omega', \beta, n, \iota'}$$

$$\textsc{Load}\ \frac{\begin{array}{c}\Delta, n \vdash e \Downarrow a \quad \mu[n, a] = v \quad \Delta' = \Delta[(n,r) \to v] \quad \rho = pc[n] \quad \iota = \Pi[\rho] \quad pc' = pc[n \to \rho + 1] \\ \rho' = pc'[n] \quad \iota' = \Pi[\rho'] \quad \omega' = \omega.\langle \rho, a \rangle \quad \neg \mathit{resolve}(n, \beta, pc)\end{array}}{\Pi, \Delta, \mu, pc, \omega, \beta, n, r := \texttt{mem}[e] \quad \rightsquigarrow \quad \Pi, \Delta', \mu, pc', \omega', \beta, n, \iota'}$$

$$\textsc{Store}\ \frac{\begin{array}{c}\Delta, n \vdash e_1 \Downarrow a \quad \Delta, n \vdash e_2 \Downarrow v \quad \mu' = \mu[(n, a) \to v] \quad \rho = pc[n] \quad \iota = \Pi[\rho] \quad pc' = pc[n \to \rho + 1] \\ \rho' = pc'[n] \quad \iota' = \Pi[\rho'] \quad \omega' = \omega.\langle \rho, a \rangle \quad \neg \mathit{resolve}(n, \beta, pc)\end{array}}{\Pi, \Delta, \mu, pc, \omega, \beta, n, \texttt{mem} := \texttt{mem}[e_1 \to e_2] \quad \rightsquigarrow \quad \Pi, \Delta, \mu', pc', \omega', \beta, n, \iota'}$$

$$\textsc{T-Pred}\ \frac{\begin{array}{c}\Delta, n \vdash e \Downarrow \texttt{true} \quad \mathit{mispred}(n, \beta, pc) = \texttt{false} \quad \rho = pc[n] \quad \iota = \Pi[\rho] \quad pc' = pc[n \to c] \\ \rho' = pc'[n] \quad \iota' = \Pi[\rho'] \quad \omega' = \omega.\langle \rho, \bot \rangle \quad \beta' = \mathit{update}(n, \rho, \iota, \beta) \quad \neg \mathit{resolve}(n, \beta, pc)\end{array}}{\Pi, \Delta, \mu, pc, \omega, \beta, n, \texttt{if}\ e\ \texttt{goto}\ c \quad \rightsquigarrow \quad \Pi, \Delta, \mu, pc', \omega', \beta', n, \iota'}$$

$$\textsc{T-Mispred}\ \frac{\begin{array}{c}\Delta, n \vdash e \Downarrow \texttt{true} \quad \mathit{mispred}(n, \beta, pc) = \texttt{true} \quad \rho = pc[n] \quad \iota = \Pi[\rho] \quad n' = n + 1 \\ \forall m, r.\ \Delta'[m, r] = \textsc{ite}(m = n', \Delta(n, r), \Delta(m, r)) \quad \forall m, a.\ \mu'[m, r] = \textsc{ite}(m = n', \mu(n, r), \Delta(m, r)) \\ pc' = pc[n' \to \rho + 1, n \to c] \quad \rho' = pc'[n'] \quad \iota' = \Pi[\rho'] \quad \beta' = \mathit{update}(n, \rho, \iota, \beta) \quad \neg \mathit{resolve}(n, \beta, pc)\end{array}}{\Pi, \Delta, \mu, pc, \omega, \beta, n, \texttt{if}\ e\ \texttt{goto}\ c \quad \rightsquigarrow \quad \Pi, \Delta', \mu', pc', \omega', \beta', n', \iota'}$$

$$\textsc{NT-Pred}\ \frac{\begin{array}{c}\Delta, n \vdash e \Downarrow \texttt{false} \quad \mathit{mispred}(n, \beta, pc) = \texttt{false} \quad \rho = pc[n] \quad \iota = \Pi[\rho] \quad pc' = pc[n \to \rho + 1] \\ \rho' = pc'[n] \quad \iota' = \Pi[\rho'] \quad \omega' = \omega.\langle \rho, \bot \rangle \quad \beta' = \mathit{update}(n, \rho, \iota, \beta) \quad \neg \mathit{resolve}(n, \beta, pc)\end{array}}{\Pi, \Delta, \mu, pc, \omega, \beta, n, \texttt{if}\ e\ \texttt{goto}\ c \quad \rightsquigarrow \quad \Pi, \Delta, \mu, pc', \omega', \beta', n, \iota'}$$

$$\textsc{NT-Mispred}\ \frac{\begin{array}{c}\Delta, n \vdash e \Downarrow \texttt{false} \quad \mathit{mispred}(n, \beta, pc) = \texttt{true} \quad \rho = pc[n] \quad \iota = \Pi[\rho] \quad n' = n + 1 \\ \forall m, r.\ \Delta'[m, r] = \textsc{ite}(m = n', \Delta(n, r), \Delta(m, r)) \quad \forall m, a.\ \mu'[m, r] = \textsc{ite}(m = n', \mu(n, r), \Delta(m, r)) \\ pc' = pc[n' \to c, n \to \rho + 1] \quad \rho' = pc'[n'] \quad \iota' = \Pi[\rho'] \quad \beta' = \mathit{update}(n, \rho, \iota, \beta) \quad \neg \mathit{resolve}(n, \beta, pc)\end{array}}{\Pi, \Delta, \mu, pc, \omega, \beta, n, \texttt{if}\ e\ \texttt{goto}\ c \quad \rightsquigarrow \quad \Pi, \Delta', \mu', pc', \omega', \beta', n', \iota'}$$

$$\textsc{Goto}\ \frac{\begin{array}{c}\rho = pc[n] \quad \iota = \Pi[\rho] \quad pc' = pc[n \to c] \quad \rho' = pc'[n] \\ \iota' = \Pi[\rho'] \quad \omega' = \omega.\langle \rho, \bot \rangle \quad \beta' = \mathit{update}(n, \rho, \iota, \beta) \quad \neg \mathit{resolve}(n, \beta, pc)\end{array}}{\Pi, \Delta, \mu, pc, \omega, \beta, n, \texttt{goto}\ c \quad \rightsquigarrow \quad \Pi, \Delta, \mu, pc', \omega', \beta', n, \iota'}$$

$$\textsc{SpecFence}\ \frac{n' = 0 \quad \rho' = pc[n'] \quad \iota' = \Pi[\rho'] \quad \neg \mathit{resolve}(n, \beta, pc)}{\Pi, \Delta, \mu, pc, \omega, \beta, n, \iota \quad \rightsquigarrow \quad \Pi, \Delta, \mu, pc, \omega, \beta, n', \iota'}$$

$$\textsc{Resolve}\ \frac{n' = n - 1 \quad \rho' = pc[n'] \quad \iota' = \Pi[\rho'] \quad \beta' = \mathit{update}(n, \rho, \iota, \beta) \quad \mathit{resolve}(n, \beta, pc)}{\Pi, \Delta, \mu, pc, \omega, \beta, n, \iota \quad \rightsquigarrow \quad \Pi, \Delta, \mu, pc, \omega, \beta', n', \iota'}$$

$$\textsc{havoc}\ \frac{n = 0 \quad \rho \notin \mathcal{T}_\rho \quad pc'[n] = \rho \quad \iota = \Pi[\rho] \quad \forall a.\ a \notin \mathcal{U}_\mu^{wr} \implies \mu'[0, a] = \mu[0, a]}{\Pi, \Delta, \mu, pc, \omega, \beta, n, \texttt{havoc}\ (\Delta,\ \texttt{mem}[\mathcal{U}_\mu^{wr}],\ \beta) \quad \rightsquigarrow \quad \Pi, \Delta', \mu', pc', \omega, \beta', n, \iota}$$

Figure 9: Operational Semantics for Statements in AIR.

mated verification tools because the verification engine would need to explore exponentially more instruction orderings.

Machine state s is the tuple $\langle \Pi, \Delta, \mu, pc, \omega, \beta, n, \iota \rangle$. Π is the program memory: a map from program counter values to instructions. Δ and μ are the state of the registers and data memory respectively while pc contains the program counter. ω is the trace of program and data addresses accessed so far. β is the branch predictor state, which we leave abstract in this paper and ι is the instruction that will be executed next.

The main novelty in these semantics is modeling misspeculation. n is an integer that represents *speculation level*: it is incremented each time we misspeculate on a branch and decremented when a branch is resolved. Speculation level 0 corresponds to architectural (non-speculative) execution. Δ, μ and pc – registers, memory and program counter respectively – are also indexed by the speculation level. $\Delta[n, r]$ refers to the value of the register r at speculation level n. $\Delta[(n, r) \to v]$ refers to a register state which is identical to Δ except that register r at speculation level n has been assigned value v. We adopt similar notation for μ and pc.

Expression Semantics are shown in Figure 8. Expressions are defined over the register state Δ. Notation $\Delta, n \vdash e \Downarrow v$

means that the expression e evaluates to value v given register state Δ at speculation level n. These are standard except for the additional wrinkle of the speculation level.

Statement Semantics are shown in Figure 9. A transition from the machine state $s = \langle \Pi, \Delta, \mu, pc, \omega, \beta, n, \iota \rangle$ to the machine state $s' = \langle \Pi', \Delta', \mu', pc', \omega', \beta', n', \iota' \rangle$ is written as $\langle \Pi, \Delta, \mu, pc, \omega, \beta, n, \iota \rangle \rightsquigarrow \langle \Pi', \Delta', \mu', pc', \omega', \beta', n', \iota' \rangle$. We now briefly describe the rules shown in Figure 9.

The REGISTERUPDATE rule models the execution of statements of the form $r := e$, expression e is written to the register r. This involves: (i) updating the value of register r at speculation level n to have the value of the expression e: $\Delta' = \Delta[(n, r) \to v]$, (ii) incrementing the pc at speculation level n: $pc' = pc[n \to \rho + 1]$ and (iii) appending $\langle \rho, \perp \rangle$ to the trace of memory addresses accessed by the program: $\omega' = \omega.\langle pc, \perp \rangle$. The $\perp$ in the second element of the tuple indicates that no data memory access is performed by this instruction. This rule is only executed when a branch is not being resolved: $\neg resolve(n, \beta, pc)$ and the next instruction to be executed is $\iota' = \Pi[\rho']$ where $\rho' = pc[n]$.

The LOAD and STORE rules are similar. LOAD updates the register state with value stored at memory location a at speculation level n: $v = \mu[n, a]$ while STORE leaves register state Δ unchanged and updates memory address a at speculation level n: $\mu' = \mu[(n, a) \to v]$. Both LOAD and STORE append $\langle \rho, a \rangle$ to the trace of memory addresses signifying accesses to program address ρ and data address a. As with REGISTERUPDATE, these rules only apply when a branch is not being resolved in this step: $\neg resolve(n, \beta, pc)$.

The T-PRED rule applies when a conditional jump `if` e `goto` c should be taken and is also predicted taken. In the semantics, we model misspeculation through an uninterpreted function $mispred(n, \beta, pc)$ where β is the branch predictor state (left abstract in our model), n is the speculation level and pc is a map from speculation levels to program counter values. This rule only applies when $mispred$ evaluates `false`. The rule sets the program counter at speculation level n to c: $pc' = pc[n \to c]$ and updates the branch predictor state β' using the uninterpreted function $update$. Just like the other rules discussed so far, this applies only when the predicate $resolve$ does not hold.

The T-MISPRED rule applies when a conditional jump `if` e `goto` c should be taken but is predicted not taken ($mispred$ evaluates to `true`). This rule changes system state in the following ways. First, the speculation level is incremented: $n' = n + 1$. Second, the state of the registers at level n in Δ is now copied over to level n' in Δ' while all other levels are identical between Δ and Δ'. The memory state μ is also modified in a similar way. The program counter at level n gets the correct target c, while the program counter at level n' gets the mispredicted fall-through target $\rho + 1$. Execution continues at speculation level n'.

NT-PRED, NT-MISPRED handle the case when the conditional branch should *not* be taken. These are similar to T-PRED and T-MISPRED. GOTO applies to direct jumps. Note we do not consider misprediction of direct jumps as they have constant targets and will be redirected at decode.

The rule SPECFENCE resolves all outstanding speculative branches by setting the speculation level back to zero. Note that pc, Δ and μ at level zero already have the "correct" values, so nothing further needs to be done.

The rule RESOLVE applies when a mispredicted branch is resolved. Resolution occurs when the uninterpreted predicate $resolve(n, \beta, pc)$ holds. At the time of resolution, branch predictor state β' is updated using the uninterpreted function $update$ and the speculation level n' is decremented. As in SPECFENCE, nothing else need be done as the other state variables have the correct values at the decremented level.

Rule HAVOC will be described in § V-D.

V. FORMULATING SECURE SPECULATION

This section formulates the secure speculation property. First, we formalize an adversary model that captures arbitrary transient execution attacks. Next, we present the secure speculation property. Violations of this property correspond to transient execution vulnerabilities.

A. Adversary Model

Recall system state s is the tuple $\langle \Pi, \Delta, \mu, pc, \omega, \beta, n, \iota \rangle$[6] and evolves according to the transition relation $\rightsquigarrow$ from Figure 9. As discussed in § III, the system has an untrusted low-security component and a trusted high-security component that execute concurrently. Our verification objective is to prove that confidential states of a specified trusted program are indistinguishable to an arbitrary untrusted program. This verification task requires the definition of: (i) the trusted program to be verified and the family of untrusted adversary programs, (ii) confidential states of the trusted program, (iii) how the adversary *tampers* with system state, and (iv) what parts of state are adversary *observable*.

1) The Trusted and Untrusted Programs: We assume that the trusted program resides in the set of instruction memory addresses denoted by $\mathcal{T}_\rho$. The trusted program itself is defined by $\Pi[\rho]$ for each $\rho \in \mathcal{T}_\rho$. Every address $\rho \notin \mathcal{T}_\rho$ is part of the untrusted component and $\Pi[\rho]$ is unconstrained for these addresses to model all possible adversarial programs.

We assume that untrusted code can invoke trusted code only by jumping to a specific entrypoint address $\mathcal{EP} \in \mathcal{T}_\rho$. $\mathcal{T}_\rho$, $\mathcal{EP}$ and instructions $\Pi[\rho]$ for all $\rho \in \mathcal{T}_\rho$ are known to the adversary. Note that the adversary may speculatively attempt to invoke addresses other than the entrypoint, only the non-speculative invocations are restricted. Without such an assumption, the adversary may be able to jump past defensively placed instructions.

In the example shown in Figure 3b, $\mathcal{T}_\rho$ contains all instruction addresses that are part of the function `foo`. The entrypoint $\mathcal{EP}$ is the address of the first instruction in `foo`. Note that if the adversary can directly jump to line 6 (i.e., skip the fence on line 5), the program is vulnerable – this is why the restriction on invocation of only the entrypoint is required.

---

[6] Note: We will use the notation $s.\mathit{field}$ to refer to elements of the tuple.

Such restrictions are implemented in all typical scenarios: system calls, software fault isolation, etc. To consider another example, if we are verifying secure speculation for system calls in an operating system kernel, $\mathcal{T}_\rho$ contains all kernel text addresses and the entrypoint $\mathcal{EP}$ is the syscall trap address.

Given the above definitions, the low operation executed in a state $s = \langle \Pi, \Delta, \mu, pc, \omega, \beta, n, \iota \rangle$ is $op_{\mathcal{L}}(s) \doteq \Pi[pc[0]]$ if $pc[0] \notin \mathcal{T}_\rho$ and $\bot$ otherwise. This definition refers to the non-speculative state – we are looking at $pc[0]$, not higher speculation levels. The instruction being speculatively executed may be different, and may in fact be from the trusted component. This is important because we use $op_{\mathcal{L}}$ to constrain adversary actions to be identical across traces, and these constraints can only refer to non-speculative state.

Finally, the trusted program must start off in some well-defined initial state. For instance, global variables may need to be initialized to specific values. We use the predicate $init_{\mathcal{T}}(s)$ to refer to a valid initial state of the trusted program.

2) Confidential States: The secret states that need to be protected from an adversary are the values stored in memory addresses a that belong to the set $\mathcal{S}_{\mathcal{T}}$. For Figure 3, $\mathcal{S}_{\mathcal{T}}$ contains all addresses that are part of the arrays `a1` and `a2`.

All other addresses are public state. We will use $\mathcal{P}_{\mathcal{T}}$ to denote the projection of the values stored at these public addresses: $\mathcal{P}_{\mathcal{T}}(\mu) \doteq \lambda a.\ \text{ITE}(a \notin \mathcal{S}_{\mathcal{T}}, \mu[0,a], \bot)$.

The high instruction executed in a state is denoted $inst_{\mathcal{T}}(s)$ and has the value $s.\Pi[s.pc[0]]$ when $pc[0] \in \mathcal{T}_\rho$ and $\bot$ otherwise. The high operation executed in state s is defined as a tuple of the high instruction and the public memory: $op_{\mathcal{H}}(s) \doteq \langle inst_{\mathcal{T}}(s), \mathcal{P}_{\mathcal{T}}(s.\mu) \rangle$. We include the values of public memory in this tuple because the high-program may be non-deterministic and we need to constrain the non-determinism to be identical across certain traces.

3) General Adversary Tampering ($\mathcal{G}$): The adversary $\mathcal{G}$ tampers with system state by executing an unbounded number of instructions to modify architectural and microarchitectural state. Adversary tampering is constrained in only two ways.

1) (**Conformant Store Addresses**) For every non-speculative state in which an untrusted store is executed, the target address of the store must belong to the set of adversary-writeable addresses: $\mathcal{U}_\mu^{wr}$. We denote a trace π where every state satisfies this condition by the predicate $conformantStoreAddrs(\pi)$, defined as follows.

$$\begin{aligned} &\forall i.\ \pi^i.n = 0 \land \pi^i.pc[0] \notin \mathcal{T}_\rho && \implies \\ &\quad \pi^i.\iota = \texttt{mem} := \texttt{mem}[e_1 \to e_2] \land \pi^i.\Delta[0, e_1] \Downarrow a && \implies \\ &\quad a \in \mathcal{U}_\mu^{wr} \end{aligned}$$

Constraining adversary stores is necessary in order to prevent the adversary from changing the trusted program's architectural (non-speculative) state arbitrarily.

2) (**Conformant Entrypoints**) Non-speculative adversary jumps to trusted code must target the entrypoint $\mathcal{EP}$. A trace π where every transition from untrusted to trusted code satisfies this condition is denoted by the predicate $conformantEntrypoints(\pi)$. This is defined as follows:

$$\begin{aligned} &\forall i, j.\ i < j \land \pi^i.n = \pi^j.n = 0 && \implies \\ &\quad (\forall k.\ i < k < j \implies \pi^k.n \neq 0) && \implies \\ &\quad \pi^i.pc[0] \notin \mathcal{T}_\rho \land \pi^j.pc[0] \in \mathcal{T}_\rho && \implies \\ &\quad \pi^j.pc[0] = \mathcal{EP} \end{aligned}$$

The above constraints says that if π^i and π^j are non-speculative states, all states between π^i and π^j are speculative, and π^i is part of the untrusted component while π^j is part the trusted component, then π^j must necessarily be at the entrypoint. Note this *does not* preclude speculative execution of "gadgets" in the trusted code that do not begin at the entrypoint.

The condition conformant store addresses captures the fact that the adversary cannot write to arbitrary memory locations. Conformant entrypoints ensures that execution of the trusted code starts at the entrypoint.

4) Conformant Traces: A trace π where: (i) π^0 is a non-speculative state and the trusted component has been initialized: $\pi^0.n = 0 \land init_{\mathcal{T}}(\pi^0)$, (ii) every state π^i satisfies the conformant stores condition and (iii) every pair of states π^i and π^j, where $i < j$, satisfy the conformant entrypoints condition is called a conformant trace, denoted by $conformant(\pi)$.

$$\begin{aligned} conformant(\pi) \doteq\ & \pi^0.n = 0 \land init_{\mathcal{T}}(\pi^0) && \land \\ & conformantStoreAddrs(\pi) && \land \\ & conformantEntrypoints(\pi) && (4) \end{aligned}$$

5) Adversary Observations: We model an adversary who can observe all architectural state and most microarchitectural state when executing; i.e. when $n = 0$ and $pc[0] \notin \mathcal{T}_\rho$. Specifically, the adversary can observe the following:

1) non-speculative register values: $\Delta[0, r]$ for all r.
2) non-speculative values stored at all memory addresses in the set $\mathcal{U}_\mu^{rd}$: $\mu[0, a]$ for all $a \in \mathcal{U}_\mu^{rd}$.
3) the trace of instruction and data memory accesses: ω.
4) the branch predictor state β.

The above implies that two states $s = \langle \Pi, \Delta, \mu, pc, \omega, \beta, n, \iota \rangle$ and $s' = \langle \Pi, \Delta', \mu', pc', \omega', \beta', n', \iota' \rangle$ are low-equivalent, denoted $s \approx_{\mathcal{L}} s'$, iff $(n = 0 \land pc[0] \notin \mathcal{T}_\rho) \implies (\forall r.\ \Delta[0, r] = \Delta'[0, r]) \land (\forall a.\ a \in \mathcal{U}_\mu^{rd} \implies \mu[0, a] = \mu'[0, a]) \land \omega = \omega' \land \beta = \beta'$.

We do not allow the adversary to observe $\Delta[n, r]$ and $\mu[n, a]$ for $n > 0$ because there is no way to "output" speculative state except through a microarchitectural side-channel. These side-channels are captured by the trace of memory accesses ω which models leaks via caches, prefetches, DRAM and well as other structures in the memory subsystem. The branch predictor state β captures all leaks caused by the branch predictor side-channel. Note that the adversary *can* observe the non-speculative values stored in memory for the addresses in the range $\mathcal{U}_\mu^{rd}$, and non-speculative values of the registers when adversary code is being executed.

B. Formalization of the Security Property

Using the above definitions, we are now ready to formalize the secure speculation property, shown in Equation 5.

$$
\begin{aligned}
&\forall \pi_1, \pi_2, \pi_3, \pi_4. \\
&\quad \mathit{conformant}(\pi_1) \land \mathit{conformant}(\pi_2) && \implies \\
&\quad \mathit{conformant}(\pi_3) \land \mathit{conformant}(\pi_4) && \implies \\
&\quad \forall i.\ \neg \mathit{mispred}(\pi_1^i.n, \pi_1^i.\beta, \pi_1^i.pc) && \implies \\
&\quad \forall i.\ \neg \mathit{mispred}(\pi_2^i.n, \pi_2^i.\beta, \pi_2^i.pc) && \implies \\
&\quad \exists i.\ \mathit{mispred}(\pi_3^i.n, \pi_3^i.\beta, \pi_3^i.pc) && \implies \\
&\quad \exists i.\ \mathit{mispred}(\pi_4^i.n, \pi_4^i.\beta, \pi_4^i.pc) && \implies \\
&\quad op_{\mathcal{L}}(\pi_1) = op_{\mathcal{L}}(\pi_2) = op_{\mathcal{L}}(\pi_3) = op_{\mathcal{L}}(\pi_4) && \implies \\
&\quad op_{\mathcal{H}}(\pi_1) = op_{\mathcal{H}}(\pi_3) \land op_{\mathcal{H}}(\pi_2) = op_{\mathcal{H}}(\pi_4) && \implies \\
&\quad \pi_1 \approx_{\mathcal{L}} \pi_2 \land \pi_3^0 \approx_{\mathcal{L}} \pi_4^0 && \implies \\
&\quad \pi_3 \approx_{\mathcal{L}} \pi_4 && (5)
\end{aligned}
$$

This an instantiation of the TPOD property shown in Equation 3. The trace property T is satisfied when no misspeculation occurs: $\pi \in T \iff \forall i.\ \neg \mathit{mispred}(\pi^i.n, \pi^i.\beta, \pi^i.pc)$.[7] The trace property U requires that all traces be conformant as defined in Equation 4. This ensures we only search for violations among traces representing valid executions of our system/adversary model.

A violation of Equation 5 occurs when there exists a sequence of adversary instructions such that traces π_1 and π_2 are low-equivalent, but π_3 and π_4 are not low-equivalent. In other words, we have an information leak that only occurs on a speculative processor; i.e. a transient execution vulnerability.

C. Illustrating Violation/Satisfaction of Secure Speculation

Let us consider the conditionally vulnerable Spectre variant shown in Figure 3c. A quadruple of traces for this program is shown in Figure 10a. Two calls to `foo` are made with arguments `i` $=$ `0` and `i` $=$ `1`. The two non-speculative traces π_1 and π_2 do not execute the if statement and so they have the same adversary observations (i.e., are low-equivalent). However, traces π_3 and π_4 speculatively execute the if statement and the adversary can observe differences in the memory addresses corresponding to the second array access: $a2[v_1 * S]$ and $a2[v_2 * S]$. All traces have the same adversary operations with one pair low-equivalent and non-speculative while the other pair is not low-equivalent and speculative. This is a violation of secure speculation.

Now consider the scenario when $N > 0$, say $N = 1$. This is shown in Figure 10b. The key difference here is that the non-speculative traces also make the second array access when `i` $= 0$. The second memory access reads from the addresses $a2 + v_1 * S$ and $a2 + v_2 * S$ in traces π_1 and π_2 respectively. There are two scenarios possible. Either $v_1 = v_2$ or $v_1 \neq v_2$. Suppose $v_1 = v_2$, then traces π_1 and π_2 are low-equivalent, but so are traces π_3 and π_4! Conversely, if $v_1 \neq v_2$, then the π_1 and π_2 are not low-equivalent and the secure speculation property holds vacuously.

D. Adversary Reduction Lemma

The general adversary's tampering described in § V-A3 allows the adversary to execute an unbounded number of arbitrary instructions. While this is fully general, it makes automated reasoning unscalable. To address this problem, we introduce a simpler "havocing adversary" $\mathcal{H}$ and prove that this adversary is as powerful as the general adversary $\mathcal{G}$.

$\mathcal{H}$ executes only one instruction that modifies non-speculative state: `havoc` $(\Delta, \mathtt{mem}[\mathcal{U}_\mu^{wr}], \beta)$. The semantics of this instruction are shown in Figure 9; it sets the registers, program counter, adversary writeable memory addresses and branch predictor to unconstrained values (i.e. "havocs" them).

Lemma 2: Every sequence of $s_i, \ldots, s_j$ with $op_{\mathcal{L}}(s_j) \neq \bot$ and $s_j.n = 0$ for every $i \leq j \leq k$ can be simulated by a single `havoc` $(\Delta, \mathtt{mem}[\mathcal{U}_\mu^{wr}], \beta)$ instruction.

The adversary reduction lemma lets us replace all sequences of non-speculative instructions executed by the adversary with `havoc`'s and helps scale verification. It is important to note that we cannot replace instruction sequences which contain speculative instructions because these may contain exploitable transient execution gadgets.

E. Discussion and Limitations

An important implication of the secure speculation property is that if a program satisfies Equation 5, then all observational determinism properties where low-equivalence is defined over ω, μ and β that hold for non-speculative execution of the program also hold for speculative executions. For instance, a tool like CacheAudit [54, 55] can be used to verify that the cache accesses of a program are independent of some secret. Note that even though a program's non-speculative execution may not leak information through cache (this is what CacheAudit verifies) that *does not mean* that its speculation execution will have the same properties. This is because CacheAudit does not model speculative execution. However, if we do prove Equation 5 for a program, then all properties proven by tools like CacheAudit also apply to the program's speculative execution.

Our operational semantics are for in-order processors only. Nevertheless, the secure speculation property can be used to analyze out-of-order execution and other speculation (e.g., memory address disambiguation) in a conceptually straightforward way by extending the semantics to model these features.

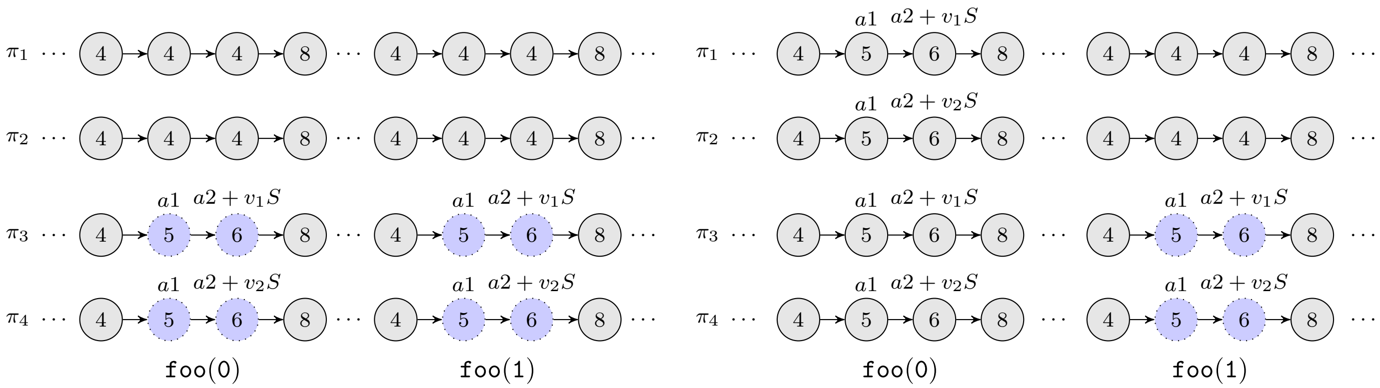
Specific programs may need additional constraints on the traces to avoid spurious counterexamples, especially if the set of secrets $\mathcal{S}_{\mathcal{T}}$ is over-specified. For example, in Figure 3(b), a tuple of traces where the `i` $<$ `N` never occurs would cause a violation of Equation 5 if $\mathcal{S}_{\mathcal{T}}$ also contained the addresses that point to `a1` and `a2`.

VI. VERIFICATION APPROACH

We have implemented an automated verifier to answer the following question: Given a program (e.g., C code) as input, does it satisfy the secure speculation property in Equation 5?

---

[7] Or equivalently in linear temporal logic: $\pi \models \Box \neg \mathit{mispred}$.



(a) Violation of secure speculation when N=0 in Figure 3c.

(b) Secure speculation satisfied when N=1 in Figure 3c.

Figure 10: Illustrating the secure speculation property for the code in Figure 3c. The numbers within each state refer to program counter values (shown as line numbers from the figure). A label above each state indicates the data memory address accessed by that instruction (if any). States shown in dotted circles are specuative states. Note the non-speculative traces "stutter" when the other traces are speculating. The values v_1 and v_2 refer to the contents of memory address a_1 in their respective traces. Note that traces π_1 and π_2 do not speculate while traces π_3 and π_4 do.

Our approach is fairly standard, based on the method of *self-composition* (see, e.g., [56]). For lack of space, we present only the essential aspects. Given the input program, we translate it into a transition system based on the adversary model and operational semantics presented in the previous section. The secure speculation property is a 4-safety property, meaning that we can turn it into a safety property to be checked on a 4-way self-composition of the transition system. We use term-level model checking [57] based on satisfiability modulo theories (SMT) solving to check whether the safety property holds for this 4-way self-composition. The model checker uses either bounded model checking (to find violations of the property) or k-induction (to prove the property).

The main new aspect of our verifier is the implementation of the transformation of the program into a transition system. We rely on two tools: the Binary Analysis Platform (BAP) [50] to translate x86 binaries into an intermediate format called BIL, and UCLID5 [47], an SMT-based model checking tool supporting both bounded model checking (BMC) and k-induction. BIL is an assembly-like intermediate language similar to AIR (described in Sec. IV). Overall, our workflow for each input program is as follows:

1) Compile C source code containing the victim function into an x86 binary file.
2) Translate the x86 binary file using BAP into the BIL intermediate language.
3) Translate the BIL into UCLID5 models and check the secure speculation property via self-composition. For each program, we first obtain a counterexample via BMC demonstrating the vulnerability; then, we insert an lfence at an appropriate point and prove the secure speculation property via k-induction.

We note that this workflow may be abstracted to a more general TPOD property.

The translation from BIL to UCLID5 implements the operational semantics given earlier, with the following key steps:

1) Datatypes in the BIL program such as addresses, memories, and words are converted to uninterpreted types for more scalable analysis and to obtain a more portable model that is not specific to 32-bit/64-bit architectures.
2) Each basic block of the BIL program is considered an atomic step of the transition system in the UCLID5 model after which the safety property is checked on the 4-way self-composition. This suffices as the deviations in behavior between the 4 traces happen at branch points.
3) At any speculative transition step, the program can resolve a misspeculation as per the RESOLVE rule.
4) All state variables are initialized to symbolic constants with the exception of the memory, where it is initialized to have the same value at every address except the program-specific secret address that stores the secret.

Given the model, the implication chain of the secure speculation property is translated into a number of assumptions and invariants. The invariants which we wish to check are whether the speculative program traces diverge in control flow, branch prediction or memory access observations, but only in the cases that they do not for the non-speculative traces. Proofs by induction require a few additional auxiliary invariants, whereas bounded model checking does not.

For a particular proof of the secure speculation property on one of the examples from Section VII in UCLID5, we first instantiate four programs as instances. In UCLID5, this is a composition of four transition systems within the main proof script. Next, we define a speculation flag that determines if a program is allowed to speculate or not and instantiate the program pairs `t1`, `t2` and `t3`, `t4` with their speculation flags turned off and on respectively. For initialization, we set the program counters, registers, and rollback states of the programs to be the same. We also set the memories of the program pairs `t1`,`t3` and `t2`, `t4` to be the same except for a confidential memory address represented by a symbolic constant, which corresponds to the conformant conditions. Note that only allowing one (symbolic) memory address to differ is sufficient because any counterexample with a larger difference can be extended to one in the single address case. Additionally, we make the assumption that the program counter and observational states, such as prior memory read addresses and the branch predictor state, are initially equal across the non-speculating programs `t1` and `t2`. During a transition step of the main proof, we step both of the non-speculative traces `t1` and `t2` only if `t3` and `t4` are not currently speculating and they "stutter" otherwise. This is to prevent any spurious counter-examples from divergent observational states caused by mismatching steps in the non-speculating and speculating programs. At each of these program transitions, a basic block of the program is executed. This abstraction is sound in the sense that the traces considered are a strict subset of the permissable traces of an out of order processor and if out of order execution is the cause of the observational determinism violation, then it should not be captured in our property. The secure speculation property is then encoded as equality across the program counter and observational states in the speculating programs `t3` and `t4`. This property was proven inductively and through bounded model checking in all of our examples. For inductive invariant checking, various auxiliary invariants were required to constrain the attacker input, rollback states, entry points, memories, and speculating states of the programs.

As a remark on the formulation of the 4-safety property, traces `t1` and `t2` are used to constrain which addresses are public and private. This allows us to generalize over the various examples for variant 1 of the spectre attack instead of encoding this specifically for each program example. More concretely, in the $N > 0$ case of Figure 3c, this constrains the value at the confidential memory address to be the same.

VII. CASE STUDIES

We used our verifier for a proof-of-concept demonstration to detect whether or not a snippet of C code is vulnerable to the Spectre class of attacks. As benchmarks, we rely on Paul Kocher's list of 15 victim functions vulnerable to the Spectre attack [30] in addition to the examples we presented earlier.

In particular, we show here results on Examples 1, 5, 7, 8, 10, 11, and 15 from Paul Kocher's list, along with the example from Figure 3 (c), and an example with nested if statements. We chose these based on what we believe are illustrative of a wide range of victim functions that are not easily detectable using the current static analysis tools such as Qspectre [58], which was only able to detect the first two examples in Kocher's list. We begin with a brief explanation of some of the examples and then discuss the results from applying bounded model checking and induction with our secure speculation property on our UCLID5 models. Fig. 11 lists all benchmarks we discuss here.

Example 5 (Figure 11b): This example is similar to the first variant but implemented within a for loop. The untrusted argument `x` may be larger than the array size, which causes the vulnerability, but if `x` is within bounds of the array, note that condition `i > 0` is also potentially vulnerable to the attack.[8]

Example 7 (Figure 11c): This example is interesting because it depends on the value of a static variable updated from a previous call of the function. Every call to the function should not make the second array access unless `x == last_x`.

Example 8 (Figure 11d): The ternary operator is interesting because the program counter is allowed to jump to two different basic blocks for the computation of the second array memory access as opposed to one block as in Example 1.

Example 10 (Figure 11e): This is the first example where a second load dependent on a secret is not required for a leak. Knowing whether or not `array2[0]` was accessed is enough to leak the secret at `array1[x]`.

Example 11 (Figure 11f): This example uses a call to memcpy to leak the secret, but because of the single byte access, it gets optimized to a single load and store.

Example 15 (Figure 11g): This example is interesting because it passes a pointer instead of an integer as the attacker controlled input. We assume the value stored in the pointer is constant across traces to ignore cases where the attacker forces a secret dependent branch during non-speculative execution.

	ex1	ex5	ex7	ex8	ex10	ex11	ex15	Fig. 3c	NI
BMC	6.6	9.0	10.2	5.7	9.6	6.4	5.8	6.6	12.9
Ind	5.0	5.0	5.7	4.6	5.8	5.9	4.8	4.8	5.4

Table I: Runtime (sec.) of each example using 5 steps for bounded model checking to find vulnerabilities and 1 step induction to prove correctness after inserting a memory fence. These experiments were run on a machine with an 2.20GHz Intel(R) Core(TM) i7-2670QM CPU with 5737MiB of RAM.

Example NI (Figure 11h) In this example, nested if statements cause the attack to occur without a second address load dependent on a secret. If the programs speculatively choose not to execute the second if statement, but only one program eventually executes the second if as a result of a resolution, then a leak can occur.

Table I lists the run-time (in seconds) required for each verification task with the memory fences implemented. As can be seen, the verifier is able to prove the correctness of these programs within a few seconds. Although these programs are small, this exercise gives us confidence that the method could be useful on larger programs. We assert that with the use of a stronger software model checking engine and the development of TPOD-specific abstractions, it will be possible to prove secure speculation for larger programs.

VIII. RELATED WORK

The most closely related past work to ours is CheckMate [59] which uses happens-before graphs to analyze

---

[8] Kocher's code has the condition `x >= 0` which causes an infinite loop.

```
void victim_function_v01(unsigned x) {
    if (x < array1_size) {
        __mm_lfence();
        temp &= array2[array1[x] * 512];
    }
}
```

(a) Example 1: Original Spectre BCB (bounds check bypass) example.

```
void victim_function_v05(unsigned x) {
    size_t i;
    if (x < array1_size) {
        for (i = x - 1; i > 0; i--)
            _mm_lfence();
            temp &= array2[array1[i] * 512];
    }
}
```

(b) Example 5: BCB with a for loop.

```
void victim_function_v07(unsigned x) {
    static unsigned last_x = 0;
    if (x == last_x) {
        _mm_lfence();
        temp &= array2[array1[x] * 512];
    }
    if (x < array1_size)
        last_x = x;
}
```

(c) Example 7: BCB with unsafe static variable check.

```
void victim_function_v08(unsigned x) {
    result = (x < array1_size);
    _mm_lfence();
    temp &= array2[array1[result ? (x + 1) : 0] *
         512];
}
```

(d) Example 8: BCB with the ternary conditional operator.

```
void victim_function_v10(unsigned x, unsigned k) {
    if (x < array1_size) {
        __mm_lfence();
        if (array1[x] == k)
            temp &= array2[0];
    }
}
```

(e) Example 10: BCB using an additional attacker controlled input.

```
void victim_function_v11(unsigned x) {
    if (x < array1_size) {
        _mm_lfence();
        temp = memcmp(&temp, array2 + (array1[x]
             * 512), 1);
    }
}
```

(f) Example 11: BCB using the memory comparison function.

```
void victim_function_v15(unsigned *x) {
    if (*x < array1_size) {
        _mm_lfence();
        temp &= array2[array1[*x] * 512];
    }
}
```

(g) Example 15: BCB using attacker controlled pointer.

```
void victim_function_nested_ifs(unsigned x) {
    unsigned val1, val2;
    if (x < array1_size) {
        val1 = array1[x];
        if (val1 & 1) {
            _mm_lfence();
            val2 = array2[0];
        }
    }
}
```

(h) Example NI: BCB with nested if statements.

Figure 11: Examples that were verified for secure speculation with `__mm__lfence()` implemented. In all code snippets assume that that arguments `x` and `k` are untrusted (low-security) inputs to the trusted (high-security) victim functions.

transient execution vulnerabilities. The insight in CheckMate is that happens-before graphs encode information about the orders in which instructions can be executed. By searching for patterns in the graph where branches are followed by dependent loads, an architectural model can be analyzed for susceptibility to Spectre/Meltdown. A key difference between CheckMate and our approach is that we are not matching patterns of vulnerable instructions. Our verification is semantic, not pattern-based. In particular, the example showing conditional vulnerability in Figure 3(c) cannot be precisely captured by CheckMate.

Another closely related effort is by McIlroy et al. [60] who introduce a formal model of speculative execution in modern processors and analyze it for transient execution vulnerabilities. Similar to our work, they too introduce speculative operational semantics and their model includes indirect jumps and a timer. An important difference between their semantics and ours is that their semantics are based on a microarchitectural model of execution. In contrast, our semantics capture an abstract notion of speculation that: (i) does not prescribe any specific microarchitectural implementation and (ii) is more amenable to verification due to its abstract nature. Further, they do not present a automated verification approach for finding transient execution vulnerabilities.

In concurrent work to ours, Guarnieri et al. [61] introduce SPECTECTOR which is also a principled verification methodology for the detection of Spectre-like vulnerabilities. They introduce the notion of speculative non-interference which is defined as follows: for every pair of initial configurations of the program, if these configurations are low-equivalent and their non-speculative traces have the same observations, then their speculative traces must also have the same observations. This is similar to our secure speculation property. Note that we also introduce TPOD which is a generalization of secure speculation/speculative non-interference and could be used to reason about the interaction between arbitrary microarchitectural side-channels: e.g. prefetching and value prediction.

One difference between our work and SPECTECTOR is that the latter only considers terminating programs while our methodology is applicable to non-terminating programs. This is because SPECTECTOR analyzes only finite-length traces. This also implies that in the case of non-terminating programs, SPECTECTOR can only find violations, not prove the absence of vulnerabilities. In contrast, our verification methodology can indeed prove the absence of vulnerabilities. A important insight in the SPECTECTOR work is that is that non-speculative traces are sub-sequences of the speculative traces *for in-order processors*. The allows SPECTECTOR to convert the 4-safety secure speculation property into into a 2-safety property. While this is an important and useful optimization that improves scalability, it does not appear to be applicable to out-of-order speculative semantics. While our current implementation and semantics do not model out-of-order execution, they are built to be extensible to this scenario.

The Spectre vulnerability was discovered by Kocher et al. [2, 3] while Meltdown was discovered by Lipp et al. [1]. Their public disclosure has triggered an avalanche of new transient execution vulnerabilities, notable among which are Foreshadow [4] which attacked enclave platforms and virtual machine monitors, SpectreRSB [7] and Ret2Spec [6]. A thorough study of transient execution vulnerabilities was done by Canella et al. [5]. These vulnerabilities build on the rich literature of microarchitectural side-channel attacks [14–20, 62–65]. Verification of mitigations to these "traditional" side-channel attacks is well-studied [54, 55, 66–74].

TPOD in general and secure speculation in particular are examples of hyperproperties [40]. A large body of work has studied hyperproperties that encode secure information flow. Influential exemplars of this line of work include non-interference [41], separability [75] and observational determinism [42–44]. Our verification method is based on self-composition which has been well-studied; see, for example, Barthe et al. [45, 56]. While we take a straightforward approach to using self-composition, more sophisticated approaches are also possible in some cases (e.g., [76]).

IX. CONCLUSION

This paper presented a formal approach for secure speculative execution on modern processors, a key part of which is a formal specification of secure speculation that abstracts away from the particulars of specific vulnerabilities. Our secure speculation formulation is an instance of trace property-dependent observational determinism, a new class of information flow security properties introduced by this work. We introduced an adversary model and an automated approach to verifying secure speculation and demonstrated the approach on several programs that have been used to illustrate the Spectre class of vulnerabilities. To the best of our knowledge, ours is the first effort to formalize and automatically prove secure speculation. In future work, we plan to evaluate our approach on larger programs and more complex platforms including out-of-order processors.

Acknowledgments

This work was supported in part by the ADEPT Center, SRC tasks 2867.001 and 2854.001, the iCyPhy Center, NSF grants CNS-1739816 and CNS-1646208, the Science and Engineering Research Board and a gift from Microsoft Research.

REFERENCES

[1] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *ArXiv e-prints*, Jan. 2018.

[2] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *ArXiv e-prints*, Jan. 2018.

[3] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *Proceedings of the IEEE Symposium on Security and Privacy*, pp. 19–37, 2019.

[4] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, 2018, p. 991–1008.

[5] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A Systematic Evaluation of Transient Execution Attacks and Defenses," *CoRR*, vol. abs/1811.05441, 2018. [Online]. Available: http://arxiv.org/abs/1811.05441

[6] G. Maisuradze and C. Rossow, "Ret2Spec: Speculative Execution Using Return Stack Buffers," in *Proc. of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, 2018.

[7] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre Returns! Speculation Attacks using the Return Stack Buffer," in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. Baltimore, MD: USENIX Association, 2018.

[8] J. Horn, "Read privileged memory with a side-channel," 2018. [Online]. Available: https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html

[9] Intel, "Bounds Check Bypass / CVE-2017-5753 / INTEL-SA-00088," 2018. [Online]. Available: https://software.intel.com/security-software-guidance/software-guidance/bounds-check-bypass

[10] ——, "Branch Target Injection / CVE-2017-5715 / INTEL-SA-00088," 2018. [Online]. Available: https://software.intel.com/security-software-guidance/software-guidance/branch-target-injection

[11] ——, "Rogue System Register Read / CVE-2018-3640 / INTEL-SA-00115," 2018. [Online]. Available: https://software.intel.com/security-software-guidance/software-guidance/rogue-system-register-read

[12] ——, "Speculative Store Bypass / CVE-2018-3639 / INTEL-SA-00115," 2018. [Online]. Available: https://software.intel.com/security-software-guidance/software-guidance/speculative-store-bypass

[13] ——, "L1 Terminal Fault / CVE-2018-3615 , CVE-2018-3620,CVE-2018-3646 / INTEL-SA-00161," 2018. [Online]. Available: https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault

[14] C. Percival, "Cache missing for fun and profit," 2005.

[15] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert, "Predicting secret keys via branch prediction," in *Cryptographers Track at the RSA Conference*. Springer, 2007, pp. 225–242.

[16] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The Spy in the Sandbox - Practical Cache Attacks in Javascript," *CoRR*, vol. abs/1502.07373, 2015. [Online]. Available: http://arxiv.org/abs/1502.07373

[17] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks Are Practical," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2015, pp. 605–622. [Online]. Available: http://dx.doi.org/10.1109/SP.2015.43

[18] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing – and Its Application to AES," in *IEEE Symposium on Security and Privacy*, May 2015, pp. 591–604.

[19] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware Guard Extension: Using SGX to Conceal Cache Attacks," *CoRR*, vol. abs/1702.08719, 2017. [Online]. Available: http://arxiv.org/abs/1702.08719

[20] S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing," *CoRR*, vol. abs/1611.06952, 2016. [Online]. Available: http://arxiv.org/abs/1611.06952

[21] O. Oleksenko, B. Trach, T. Reiher, M. Silberstein, and C. Fetzer, "You shall not bypass: Employing data dependencies to prevent bounds check bypass," *arXiv preprint arXiv:1805.08506*, 2018.

[22] Intel, "Deep Dive: Analyzing Potential Bounds Check Bypass Vulnerabilities," 2018. [Online]. Available: https://software.intel.com/security-software-guidance/insights/deep-dive-analyzing-potential-bounds-check-bypass-vulnerabilities

[23] Microsoft, "Spectre mitigations in MSVC," 2018. [Online]. Available: https://devblogs.microsoft.com/cppblog/spectre-mitigations-in-msvc/

[24] P. Turner, "Retpoline: a software construct for preventing branch-target-injection," 2018. [Online]. Available: https://support.google.com/faqs/answer/7625886

[25] Intel, "Deep Dive: Managed Runtime Speculative Execution Side Channel Mitigations," 2018. [Online]. Available: https://software.intel.com/security-software-guidance/insights/deep-dive-managed-runtime-speculative-execution-side-channel-mitigations

[26] ——, "Deep Dive: Mitigation Overview for Side Channel Exploits in Linux," 2018. [Online]. Available: https://software.intel.com/security-software-guidance/insights/deep-dive-mitigation-overview-side-channel-exploits-linux

[27] B. Stuart, "Current state of mitigations for spectre within operating systems," in *Proceedings of the 4th Wiesbaden Workshop on Advanced Microkernel Operating Systems*, 2018.

[28] Microsoft, "ADV180012 — Microsoft Guidance for Speculative Store Bypass," 2018. [Online]. Available: https://portal.msrc.microsoft.com/en-US/security-guidance/advisory/ADV180012

[29] J. Cable, "Update on Spectre and Meltdown security updates for Windows devices," 2018. [Online]. Available: https://blogs.windows.com/windowsexperience/2018/03/01/update-on-spectre-and-meltdown-security-updates-for-windows-devices/

[30] P. Kocher, "Spectre Mitigations in Microsoft's C/C++ Compiler," Feb 2018. [Online]. Available: https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html

[31] J. E. Smith and G. S. Sohi, "The Microarchitecture of Superscalar Processors," *Proc. IEEE*, vol. 83, no. 12, pp. 1609–1624, Dec 1995.

[32] J. P. Shen and M. Lipasti, *Fundamentals of Superscalar Processor Design*. McGraw-Hill, 2003.

[33] T.-Y. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in *Proc. of the 24th Annual International Symposium on Microarchitecture*, ser. MICRO 24, 1991, pp. 51–61.

[34] S. McFarling, "Combining branch predictors," Technical Report TN-36, Digital Western Research Laboratory, Tech. Rep., 1993.

[35] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. Hwu, "Dynamic Memory Disambiguation Using the Memory Conflict Buffer," in *Proc. of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VI, 1994, pp. 183–193.

[36] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value locality and load value prediction," in *Proc. of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[37] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "Dawg: A defense against cache timing attacks in speculative execution processors," Cryptology ePrint Archive, Report 2018/418, 2018, https://eprint.iacr.org/2018/418.

[38] M. Larabel, "Benchmarking The Work-In-Progress Spectre/STIBP Code On The Way For Linux 4.20," 2018. [Online]. Available: https://www.phoronix.com/scan.php?page=article&item=linux-420wip-stibp&num=1

[39] L. Tung, "Linus Torvalds: After big Linux performance hit, Spectre v2 patch needs curbs," 2018. [Online]. Available: https://www.zdnet.com/article/linus-torvalds-after-big-linux-performance-hit-spectre-v2-patch-needs-curbs/

[40] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *Journal of Computer Security*, vol. 18, no. 6, pp. 1157–1210, Sep. 2010.

[41] J. A. Goguen and J. Meseguer, "Security Policies and Security Models," in *IEEE Symposium on Security and Privacy*, 1982, pp. 11–20.

[42] J. Mclean, "Proving Noninterference and Functional Correctness Using Traces," *Journal of Computer Security*, vol. 1, pp. 37–58, 1992.

[43] A. W. Roscoe, "CSP and Determinism in Security Modelling," in *Proceedings of the 1995 IEEE Symposium on Security and Privacy, Oakland, California, USA, May 8-10, 1995*, 1995, pp. 114–127.

[44] S. Zdancewic and A. C. Myers, "Observational Determinism for Concurrent Program Security," in *Proc. of the 16th IEEE Computer Security Foundations Workshop*. IEEE, 2003, pp. 29–43.

[45] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure Information Flow by Self-Composition," in *17th IEEE Computer Security Foundations Workshop, (CSFW-17)*, 2004, pp. 100–114.

[46] T. Terauchi and A. Aiken, "Secure Information Flow as a Safety Problem," in *Static Analysis Symposium (SAS '05)*, ser. LNCS 3672, 2005, pp. 352–367.

[47] S. A. Seshia and P. Subramanyan, "UCLID5: Integrating modeling, verification, synthesis and learning," in *Proceedings of the 16th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, October 2018.

[48] UCLID5 Verification and Synthesis System, "Available at http://github.com/uclid-org/uclid/."

[49] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: a sandbox for portable, untrusted x86 native code," *Communications of the ACM*, vol. 53, no. 1, pp. 91–99, 2010. [Online]. Available: http://doi.acm.org/10.1145/1629175.1629203

[50] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A Binary Analysis Platform," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, ser. CAV'11, 2011, pp. 463–469.

[51] Binary Analysis Platform (BAP) Repository, "Available at https://github.com/BinaryAnalysisPlatform/bap," 2019.

[52] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow Integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, ser. CCS '05, 2005, pp. 340–353.

[53] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, and G. Pike, "Enforcing Forward-edge Control-flow Integrity in GCC & LLVM," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14, 2014, pp. 941–955.

[54] G. Doychev, D. Feld, B. Kopf, L. Mauborgne, and J. Reineke, "CacheAudit: A Tool for the Static Analysis of Cache Side Channels," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX, 2013, pp. 431–446.

[55] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke, "CacheAudit: A tool for the static analysis of cache side channels," *ACM Transactions on Information and System Security (TISSEC)*, vol. 18, no. 1, p. 4, 2015.

[56] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure information flow by self-composition," *Mathematical Structures in Computer Science*, vol. 21, no. 6, pp. 1207–1252, 2011.

[57] R. E. Bryant, S. K. Lahiri, and S. A. Seshia, "Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions," in *Computer-Aided Verification (CAV'02)*, ser. LNCS 2404, July 2002, pp. 78–92.

[58] Microsoft, "/qspectre," Oct 2018. [Online]. Available: https://docs.microsoft.com/en-us/cpp/build/reference/qspectre?view=vs-2017

[59] C. Trippel, D. Lustig, and M. Martonosi, "Checkmate: Automated synthesis of hardware exploits and security litmus tests," in *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018*, 2018, pp. 947–960.

[60] R. Mcilroy, J. Sevcik, T. Tebbi, B. L. Titzer, and T. Verwaest, "Spectre is here to stay: An analysis of side-channels and speculative execution," *arXiv preprint arXiv:1902.05178*, 2019.

[61] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "SPECTECTOR: principled detection of speculative information flows," *CoRR*, vol. abs/1812.08639, 2018. [Online]. Available: http://arxiv.org/abs/1812.08639

[62] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR," in *Proc. of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, 2016, pp. 368–379.

[63] Y. Shin, H. C. Kim, D. Kwon, J. H. Jeong, and J. Hur, "Unveiling Hardware-based Data Prefetcher, a Hidden Source of Information Leakage," in *Proc. of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, 2018, pp. 131–145.

[64] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 955–972.

[65] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for cross-cpu attacks," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 565–581.

[66] J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hriţcu, K. Bhargavan, C. Fournet *et al.*, "Verified low-level programming embedded in f," *Proceedings of the ACM on Programming Languages*, vol. 1, no. ICFP, p. 17, 2017.

[67] B. Bond, C. Hawblitzel, M. Kapritsos, K. R. M. Leino, J. R. Lorch,

B. Parno, A. Rane, S. Setty, and L. Thompson, "Vale: Verifying high-performance cryptographic assembly code," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 917–934.

[68] J. B. Almeida, M. Barbosa, J. S. Pinto, and B. Vieira, "Formal verification of side-channel countermeasures using self-composition," *Science of Computer Programming*, vol. 78, no. 7, pp. 796–812, 2013.

[69] G. Barthe, B. Grégoire, and V. Laporte, "Secure compilation of side-channel countermeasures: the case of cryptographic constant-time," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 2018, pp. 328–343.

[70] J. B. Almeida, M. Barbosa, G. Barthe, and F. Dupressoir, "Verifiable side-channel security of cryptographic implementations: constant-time mee-cbc," in *International Conference on Fast Software Encryption*. Springer, 2016, pp. 163–184.

[71] G. Barthe, G. Betarte, J. Campo, C. Luna, and D. Pichardie, "System-level non-interference for constant-time cryptography," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1267–1279.

[72] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 53–70.

[73] H. Eldib, C. Wang, and P. Schaumont, "Formal verification of software countermeasures against side-channel attacks," *ACM Transactions on Software Engineering and Methodology*, vol. 24, no. 2, p. 11, 2014.

[74] ——, "SMT-based verification of software countermeasures against side-channel attacks," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2014, pp. 62–77.

[75] J. M. Rushby, "Proof of separability: A verification technique for a class of a security kernels," in *Proceedings of the International Symposium on Programming, 5th Colloquium, Torino, Italy*, 1982, pp. 352–367.

[76] M. Sousa and I. Dillig, "Cartesian Hoare Logic for verifying k-safety properties," in *Proc. of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16, 2016, pp. 57–69.

APPENDIX

```
// block 1
000001ba: sub victim_function_v01()
00000172:
... // function prologue
00000178: RAX := mem[0x601050, el]:u64
0000017a: CF := mem[RBP + 0xFFFFFFFFFFFFFFF8, el]:
      u64 < RAX
00000180: when ~CF goto %000001b4
00000181: goto %00000182
```

(a) Example 1: Block 1 BIL

```
procedure do_block1()
    modifies (...);
{
    ... // function prologue
    call (RAX) = load_mem(array1_size_addr);
    CF = common.lessthan(RDI, RAX);
    call branch(!CF, block3, block2);
}
```

(b) Example 1: Block 1 UCLID5 procedure.

```
// block 2
00000182:
00000183: RAX := mem[0x601040, el]:u64
00000184: RDX := mem[0x601038, el]:u64
00000185: RCX := mem[RBP + 0xFFFFFFFFFFFFFFF8, el]:
      u64
00000187: RCX := RCX << 3
0000018f: v274 := RCX
00000190: RDX := RDX + v274
00000197: RDX := mem[RDX, el]:u64
00000199: RDX := RDX << 0xC
000001a1: v284 := RDX
000001a2: RAX := RAX + v284
000001a9: RDX := mem[RAX, el]:u64
000001aa: RAX := mem[0x601058, el]:u64
000001ab: RAX := RAX & RDX
000001b2: mem := mem with [0x601058, el]:u64 <- RAX
000001b3: goto %000001b4
```

(c) Example 1: Block 2 BIL

```
procedure do_block2()
  modifies (...);
{
  var v274, v284 : word_t;
  assume (common.read(mem, temp_addr) == common.read
     (common.mem_init, temp_addr));
  if (lfence && spec_level != common.spec_idx0) {
    call do_resolve();
  } else {
    call (RAX) = load_mem(array2_addr);
    call (RDX) = load_mem(array1_addr);
    RCX = RDI;
    RCX = common.left_shift(common.val0x3, RCX);
    v274 = RCX;
    RDX = common.add(RDX, v274);
    call (RDX) = load_mem(RDX);
    RDX = common.left_shift(common.val0xC, RDX);
    v284 = RDX;
    RAX = common.add(RAX, v284);
    call (RDX) = load_mem(RAX);
    call (RAX) = load_mem(temp_addr);
    RAX = common.land(RAX, RDX);
    call store_mem(temp_addr, RAX);
    pc = block3;
    br_pred_state = common.update_br_pred(
     br_pred_state, true);
  }
}
```

(d) Example 1: Block 2 UCLID5 procedure.

```
// block 3
000001b4:
... // function epilogue
000001b9: return v295
```

(e) Example 1: Block 3 BIL

```
procedure do_block3()
    modifies (...);
{
  RAX = common.val0x0;
  RDX = common.val0x0;
  ... // function epilogue
  pc = halt;
  br_pred_state = common.update_br_pred(
     br_pred_state, true);
}
```

(f) Example 1: Block 3 UCLID5 procedure.

Figure 12: Translation from BIL to UCLID5 of the victim function in example 1 from Paul Kocher's list. The left side shows the BIL representation of the x86 binary and the right side shows the corresponding translation from a BIL block to a UCLID5 procedure.

Table II: Glossary of Symbols

Symbol	**Description**
$\lambda x.\ expr$	Function with argument x; computes $expr$.
$s.fld$	Field fld in the tuple s.
π, π_1, π_2, etc.	Traces or trace variables.
π^i	The ith element of trace π.
$\rightsquigarrow$	Transition relation.
$\approx_{\mathcal{L}}$	Low-equivalence relation (overloaded over both states and traces).
$\not\approx_{\mathcal{L}}$	Negation of the the low-equivalence relation $\approx_{\mathcal{L}}$.
$op_{\mathcal{L}}(s)$	Operation executed by the untrusted low-security component in state s. Defined to be $\perp$ if the low-security component is not being executed.
$op_{\mathcal{H}}(s)$	Operation executed by the trusted high-security component in state s. Defined to be $\perp$ if the high-security component is not being executed.
$op_{\mathcal{L}}(\pi)$	Trace of operations executed by the low-security component.
$op_{\mathcal{H}}(\pi)$	Trace of operations executed by the high-security component.
$\langle \Pi, \Delta, \mu, pc, \omega, \beta, n, \iota \rangle$	Machine state.
Π	Program memory: a map from instruction addresses to instructions.
n	Speculation level. $n = 0$ refers to non-speculative (architectural) execution, higher levels indicate (possibly nested) misspeculation.
Δ	Register state. A map from the tuple (n, i) to register values, where n is the speculation level n and i the register index.
μ	Memory state. A map from the tuple (n, a) to memory values, where n is the speculation level and a the memory address.
pc	Program counter state. A map from the speculation level to the program counter value at that level.
ω	Trace of instruction and data memory addresses accessed by the program.
β	Branch predictor state. Left abstract in this paper.
ι	The current instruction.
$\mathcal{T}_\rho$	The set of instruction memory addresses that contain the trusted program.
$\mathcal{EP}$	The entrypoint to the trusted program (instruction address of starting instruction).
$\mathcal{S}_{\mathcal{T}}$	The set of confidential memory addresses.
$\mathcal{P}_{\mathcal{T}}$	Values stored in public memory addresses. These are the non-speculative valuations of all addresses not in $\mathcal{S}_{\mathcal{T}}$.
$\mathcal{U}_\mu^{rd}$	Set the addresses the adversary can (non-speculatively) read from.
$\mathcal{U}_\mu^{wr}$	Set of addresses the adversary can (non-speculatively) write to.
$init_{\mathcal{T}}(s)$	Satisfied when s is a valid initial state of the trusted program.