Constant-Time Foundations for the New Spectre Era

Sunjay Cauligi[†] Craig Disselkoen[†] Klaus v. Gleissenthall[†]
Dean Tullsen[†] Deian Stefan[†] Tamara Rezk[★] Gilles Barthe[♠♣]

[†]UC San Diego, USA [★]INRIA Sophia Antipolis, France
[♠]MPI for Security and Privacy, Germany [♣]IMDEA Software Institute, Spain

Abstract

The constant-time discipline is a software-based countermeasure used for protecting high assurance cryptographic implementations against timing side-channel attacks. Constant-time is effective (it protects against many known attacks), rigorous (it can be formalized using program semantics), and amenable to automated verification. Yet, the advent of microarchitectural attacks makes constant-time as it exists today far less useful.

This paper lays foundations for constant-time programming in the presence of speculative and out-of-order execution. We present an operational semantics and a formal definition of constant-time programs in this extended setting. Our semantics eschews formalization of microarchitectural features (that are instead assumed under adversary control), and yields a notion of constant-time that retains the elegance and tractability of the usual notion. We demonstrate the relevance of our semantics in two ways: First, by contrasting existing Spectre-like attacks with our definition of constant-time. Second, by implementing a static analysis tool, Pitchfork, which detects violations of our extended constant-time property in real world cryptographic libraries.

CCS Concepts: • **Security and privacy → Formal security models**; **Side-channel analysis and countermeasures**.

Keywords: Spectre; speculative execution; semantics; static analysis

ACM Reference Format:
Sunjay Cauligi, Craig Disselkoen, Klaus v. Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2020. Constant-Time Foundations for the New Spectre Era. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3385412.3385970

PLDI '20, June 15–20, 2020, London, UK

ACM ISBN 978-1-4503-7613-6/20/06.
https://doi.org/10.1145/3385412.3385970

1 Introduction

Protecting secrets in software is hard. Security and cryptography engineers must write programs that protect secrets, both at the source level and when they execute on real hardware. Unfortunately, hardware too easily divulges information about a program's execution via *timing side-channels*—e.g., an attacker can learn secrets by simply observing (via timing) the effects of a program on the hardware cache [16].

The most robust way to deal with timing side-channels is via *constant-time programming*—the paradigm used to implement almost all modern cryptography [2, 11, 12, 26, 27]. Constant-time programs can neither branch on secrets nor access memory based on secret data.[1] These restrictions ensure that programs do not leak secrets via timing side-channels on hardware *without* microarchitectural features.

Unfortunately, these guarantees are moot for most modern hardware: Spectre [20], Meltdown [22], ZombieLoad [29], RIDL [32], and Fallout [5] are all dramatic examples of attacks that exploit microarchitectural features. These attacks reveal that code that is deemed constant-time in the usual sense may, in fact, leak information on processors with microarchitectural features. The decade-old constant-time recipes are no longer enough.[2]

In this work, we lay the foundations for constant-time in the presence of microarchitectural features that have been exploited in recent attacks: out-of-order and speculative execution. We focus on constant-time for two key reasons. First, *impact*: constant-time programming is largely used in real-world crypto libraries—and high-assurance code—where developers already go to great lengths to eliminate leaks via side-channels. Second, *foundations:* constant-time programming is already rooted in foundations, with well-defined semantics [4, 8]. These semantics consider very powerful attackers—e.g., attackers in [4] have control over the cache and the scheduler. An advantage of considering powerful attackers is that the semantics can overlook many hardware details—e.g., since the cache is adversarially controlled, there is no point in modeling it precisely—making constant-time amenable to automated verification and enforcement.

Contributions. We first define a semantics for an abstract, three-stage (fetch, execute, and retire) machine. Our machine

[1]More generally, constant-time programs cannot use secret data as input to any variable-time operation—e.g., floating-point multiplication.
[2]OpenSSL found this situation so hopeless that they recently updated their security model to explicitly exclude "physical system side channels" [25].

supports out-of-order and speculative execution by modeling *reorder buffers* and *transient instructions*, respectively. We assume that attackers have complete control over microarchitectural features (e.g., the branch target predictor) when executing a victim program and model the attacker's control over predictors using *directives*. This keeps our semantics simple yet powerful: our semantics abstracts over all predictors when proving security—of course, assuming that predictors themselves do not leak secrets. We further show how our semantics can be extended to capture new predictors—e.g., a hypothetical *memory aliasing* predictor.

We then define *speculative constant-time*, an extension of constant-time for machines with out-of-order and speculative execution. This definition allows us to discover microarchitectural side channels in a principled way—all four classes of Spectre attacks as classified by Canella et al. [6], for example, manifest as violations of our constant-time property.

We further use our semantics as the basis for a prototype analysis tool, Pitchfork, built on top of the `angr` symbolic execution engine [30]. Like other symbolic analysis tools, Pitchfork suffers from path explosion, which limits the depth of speculation we can analyze. Nevertheless, we are able to use Pitchfork to detect multiple Spectre bugs in real code. We use Pitchfork to detect leaks in the well-known Kocher test cases [19] for Spectre v1, as well as our more extensive test suite which includes Spectre v1.1 variants. More significantly, we use Pitchfork to analyze—and find leaks in—real cryptographic code from the libsodium, OpenSSL, and curve25519-donna libraries.

Open source. Pitchfork and our test suites are open source and available at https://pitchfork.programming.systems.

2 Motivating Examples

In this section, we show why classical constant-time programming is insufficient when attackers can exploit microarchitectural features. We do this via two example attacks and show how these attacks are captured by our semantics.

Classical constant time is not enough. Our first example consists of 3 lines of code, shown in Figure 1 (top right). The program, a variant of the classical Spectre v1 attack [20], branches on the value of register r_a (line $\underline{1}$). If r_a's value is smaller than 4, the program jumps to program location $\underline{2}$, where it uses r_a to index into a public array A, saves the value into register r_b, and uses r_b to index into another public array B. If r_a is larger than or equal to 4 (i.e., the index is out of bounds), the program skips the two load instructions and jumps to location $\underline{4}$. In a sequential execution, this program neither loads nor branches on secret values. It thus trivially satisfies the constant-time discipline.

However, modern processors do not execute sequentially. Instead, they continue fetching instructions before prior instructions are complete. In particular, a processor may continue fetching instructions beyond a conditional branch, before evaluating the branch condition. In that case, the processor *guesses* which branch will be taken. For example, the processor may erroneously guess that the branch condition at line $\underline{1}$ evaluates to true, even though r_a contains value 9. It will therefore continue down the "true" branch speculatively. In hardware, such guesses are made by a branch prediction unit, which may have been mistrained by an adversary.

These guesses, as well as additional choices such as execution order, are directly supplied by the adversary in our semantics. We model this through a series of *directives*, as shown on the bottom left of Figure 1. The directive fetch: true instructs our model to speculatively follow the true branch and to place the fetched instruction at index $\overline{1}$ in the *reorder buffer*. Similarly, the two following fetch directives place the loads at indices $\overline{2}$ and $\overline{3}$ in the buffer. The instructions in the reorder buffer, called *transient instructions*, do not necessarily match the original instructions, but can contain additional information (see Table 1). For instance, the transient version of the branch instruction records which branch has been speculatively taken.

In our example, the attacker next instructs the model to execute the first load, using the directive execute $\overline{2}$. Because the bounds check has not yet been executed, the load reads from the secret element $Key[1]$, placing the value in r_b. The attacker then issues directive execute $\overline{3}$ to execute the following load; this load's address is calculated as $44 + Key[1]$. Accessing this address affects externally visible cache state, allowing the attacker to recover $Key[1]$ through a cache side-channel attack [16]. This is encoded by the leakage observation shown in red on the bottom right. Though this secret leakage cannot happen under sequential execution, our semantics clearly highlights the possible leak when we account for microarchitectural features.

Modeling hypothetical attacks. Next, we give an example of a hypothetical class of Spectre attack captured by our extended semantics. The attack is based on a microarchitectural feature which would allow processors to speculate whether a store and load pair might operate on the same address, and forward values between them [18, 28].

We demonstrate this attack in Figure 2. The reorder buffer, after all instructions have been fetched, is shown in the top right. The program stores the value of register r_b into the $secretKey_{\text{sec}}$ array and eventually loads two values from public arrays. The attacker first issues the directive execute $\overline{2}$: value; this results in a buffer where the store instruction at $\overline{2}$ has been modified to record the resolved value x_{sec}. Next, the attacker issues the directive execute $\overline{7}$: fwd $\overline{2}$, which causes the model to mispredict that the load at $\overline{7}$ aliases with the store at $\overline{2}$, and thus to forward the value x_{sec} to the load. The forwarded value x_{sec} is then used in the address $a = 48 + x_{\text{sec}}$ of the load instruction at index $\overline{8}$. There, the loaded value X is irrelevant, but the address a is leaked to the attacker, allowing them to recover the secret value x_{sec}. The speculative

Registers

r	$\rho(r)$
r_a	9_{pub}

Memory

a	$\mu(a)$
40..43	*array* A_{pub}
44..47	*array* B_{pub}
48..4B	*array* Key_{sec}

Program

n	$\mu(n)$
$\underline{1}$	$\text{br}(>, (4, r_a), \underline{2}, \underline{4})$
$\underline{2}$	$(r_b = \text{load}([40, r_a], \underline{3}))$
$\underline{3}$	$(r_c = \text{load}([44, r_b], \underline{4}))$
$\underline{4}$	…

Speculative execution:

Directive	Effect on reorder buffer	Leakage
fetch: true	$\overline{1} \mapsto \text{br}(>, (4, r_a), \underline{2}, (\underline{2}, \underline{4}))$	
fetch	$\overline{2} \mapsto (r_b = \text{load}([40, r_a]))$	
fetch	$\overline{3} \mapsto (r_c = \text{load}([44, r_b]))$	
execute $\overline{2}$	$\overline{2} \mapsto (r_b = Key[1]_{\text{sec}})$	read 49_{pub}
execute $\overline{3}$	$\overline{3} \mapsto (r_c = X)$ where $a = Key[1]_{\text{sec}} + 44$	read a_{sec}

Figure 1. Example demonstrating a Spectre v1 attack. The branch at $\underline{1}$ acts as bounds check for array A. The execution speculatively ignores the bounds check, and leaks a byte of the secret Key.

Registers

r	$\rho(r)$
r_a	2_{pub}
r_b	x_{sec}

Memory

a	$\mu(a)$
40..43	$secretKey_{\text{sec}}$
44..47	$pubArrA_{\text{pub}}$
48..4B	$pubArrB_{\text{pub}}$

Reorder buffer

i	$buf(i)$
$\overline{2}$	$\text{store}(r_b, [40, r_a])$
	…
$\overline{7}$	$(r_c = \text{load}([45]))$
$\overline{8}$	$(r_c = \text{load}([48, r_c]))$

Speculative execution

Directive	Effect on *buf*	Leakage
execute $\overline{2}$: value	$\overline{2} \mapsto \text{store}(x_{\text{sec}}, [40, r_a])$	
execute $\overline{7}$: fwd $\overline{2}$	$\overline{7} \mapsto (r_c = \text{load}([45], x_{\text{sec}}, \overline{2}))$	
execute $\overline{8}$	$\overline{8} \mapsto (r_c = X\{\bot, a\})$	read a_{sec}
execute $\overline{2}$: addr	$\overline{2} \mapsto \text{store}(r_b, 42_{\text{pub}})$	fwd 42_{pub}
execute $\overline{7}$	$\{\overline{7}, \overline{8}\} \notin buf$ where $a = x_{\text{sec}} + 48$	rollback, fwd 45_{pub}

Figure 2. Example demonstrating a hypothetical attack abusing an aliasing predictor. This attack differs from prior speculative data forwarding attacks in that branch misprediction is not needed.

execution continues and rolls back when the misprediction is detected (details on this are given in Section 3), but at this point, the secret has already been leaked.

As with the example in Figure 1, the program in this example follows the (sequential) constant-time discipline, yet leaks during speculative execution. But, both examples are insecure under our new notion of *speculative constant-time* as we discuss next.

3 Speculative Semantics and Security

In this section we define the notion of speculative constant time, and propose a speculative semantics that models execution on modern processors. We start by laying the groundwork for our definitions and semantics.

Configurations. A configuration $C \in \text{Confs}$ represents the state of execution at a given step. It is defined as a tuple (ρ, μ, n, buf) where:

- $\rho : \mathcal{R} \rightharpoonup \mathcal{V}$ is a map from a finite set of register names $\mathcal{R}$ to values;
- $\mu : \mathcal{V} \rightharpoonup \mathcal{V}$ is a memory;
- $n : \mathcal{V}$ is the current program point;
- $buf : \mathbb{N} \rightharpoonup \text{TransInstr}$ is the reorder buffer.

Values and labels. As a convention, we use n for memory addresses that map to instructions, and a for addresses that map to data. Each value is annotated with a label from a lattice of security labels with join operator $\sqcup$. For brevity, we sometimes omit public label annotation on values.

Using labels, we define an equivalence $\simeq_{\text{pub}}$ on configurations. We say that two configurations are equivalent if they coincide on public values in registers and memories.

Reorder buffer. The *reorder buffer* maps buffer indices (natural numbers) to transient instructions. We write $buf(i)$ to denote the instruction at index i in buffer buf, if i is in buf's domain. We write $buf[i \mapsto \underline{\text{instr}}]$ to denote the result of extending buf with the mapping from i to $\underline{\text{instr}}$, and $buf \setminus buf(i)$ for the function formed by removing i from buf's domain. We write $buf[j : j < i]$ to denote the restriction of buf's domain to all indices j, s.t. $j < i$ (i.e., removing all mappings at indices i and greater). Our rules add and remove indices in a way that ensures that buf's domain will always be contiguous.

Notation. We let $\text{MIN}(M)$ (resp. $\text{MAX}(M)$) denote the minimum (maximum) index in the domain of a mapping M. We denote the empty mapping as $\emptyset$ and let $\text{MIN}(\emptyset) = \text{MAX}(\emptyset) = 0$.

For a formula φ, we may discuss the bounded highest (lowest) index for which a formula holds. We write $max(j) < i : \varphi(j)$ to mean that j is the highest index less than i for which φ holds, and define $min(j) > i : \varphi(j)$ analogously.

Register resolve function. In Figure 3, we define the *register resolve function*, which we use to determine the value of a register in the presence of transient instructions in the reorder buffer. For index i and register r, the function may **(1)** return the latest assignment to r prior to position i in the buffer, if the corresponding operation is already resolved; **(2)** return the value from the register map ρ, if there are no pending assignments to r in the buffer; or **(3)** be undefined. Note that if the latest assignment to r is yet unresolved then $(buf +_i \rho)(r) = \bot$. We extend this definition to values by

Table 1. Instructions and their transient instruction form.

	Instruction	Transient form(s)	
arithmetic operation (op specifies opcode)	$(r = \text{op}(op, \overrightarrow{rv}, n'))$	$(r = \text{op}(op, \overrightarrow{rv}))$ $(r = v_\ell)$	*(unresolved op)* *(resolved value)*
conditional branch	$\text{br}(op, \overrightarrow{rv}, n^{\text{true}}, n^{\text{false}})$	$\text{br}(op, \overrightarrow{rv}, n_0, (n^{\text{true}}, n^{\text{false}}))$ $\text{jump } n_0$	*(unresolved conditional)* *(resolved conditional)*
memory load (at program point n)	$(r = \text{load}(\overrightarrow{rv}, n'))$	$(r = \text{load}(\overrightarrow{rv}))^n$ $(r = \text{load}(\overrightarrow{rv}, (v_\ell, j)))^n$ $(r = v_\ell\{\bot, a\})^n$ $(r = v_\ell\{j, a\})^n$	*(unresolved load)* *(partially resolved load with dependency on j)* *(resolved load without dependencies)* *(resolved load with dependency on j)*
memory store	$\text{store}(rv, \overrightarrow{rv}, n')$	$\text{store}(rv, \overrightarrow{rv})$ $\text{store}(v_\ell, a_\ell)$	*(unresolved store)* *(resolved store)*
indirect jump	$\text{jmpi}(\overrightarrow{rv})$	$\text{jmpi}(\overrightarrow{rv}, n_0)$	*(unresolved jump predicted to n_0)*
function calls	$\text{call}(n_f, n_{ret})$ ret	call ret	*(unresolved call)* *(unresolved return)*
speculation fence	fence n	fence	*(no resolution step)*

$$(buf +_i \rho)(r) = \begin{cases} v_\ell & \text{if } max(j) < i : buf(j) = (r = \_) \wedge \\ & \quad buf(j) = (r = v_\ell) \\ \rho(r) & \text{if } \forall j < i : buf(j) \neq (r = \_) \\ \bot & \text{otherwise} \end{cases}$$

Figure 3. Definition of the register resolve function.

defining $(buf +_i \rho)(v_\ell) = v_\ell$ for all $v_\ell \in \mathcal{V}$, and lift it to lists of registers or values using a pointwise lifting.

3.1 Speculative Constant-Time

We present our new notion of constant-time security in terms of a small-step semantics, which relates program configurations, observations, and attacker directives.

Our semantics does not directly model caches, nor any of the predictors used by speculative semantics. Rather, we model externally visible effects—memory accesses and control flow—by producing a sequence of *observations*. We can thus reason about *any* possible cache implementation, as any cache eviction policy can be expressed as a function of the sequence of observations. Furthermore, exposing control flow observations directly in our semantics makes it unnecessary for us to track various other side channels. Indeed, while channels such as port contention or register renaming produce distinct measurable effects [20], they only serve to leak the path taken through the code—and thus modeling these observations separately would be redundant. For the same reason, we do not model a particular branch prediction strategy; we instead let the attacker resolve scheduling non-determinism by supplying a series of *directives*.

This approach has two important consequences. First, the use of observations and directives allows our semantics to remain *tractable* and *amenable to verification*. For instance, we do not need to model the behavior of the cache or any branch predictor. Second, our notion of speculative constant-time is *robust*, i.e., it holds for all possible branch predictors and replacement policies—assuming that they do not leak secrets directly, a condition that is achieved by all practical hardware implementations.

Given an attacker directive d, we use $C \xrightarrow[d]{o} C'$ to denote the execution step from configuration C to configuration C' that produces observation o. Program execution is defined from the small-step semantics in the usual style. We use $C \; {}_O\Downarrow^N_D \; C'$ to denote a sequence of execution steps from C to C'. Here D and O are the concatenation of the single-step directives and leakages, respectively; N is the number of retired instructions, i.e., $N = \#\{d \in D \mid d = \text{retire}\}$. When such a big step from C to C' is possible, we say D is a *well-formed* schedule of directives for C. We omit D, N, or O when not used.

Definition 3.1 (Speculative constant-time)**.** We say a configuration C with schedule D satisfies *speculative constant-time* (SCT) with respect to a low-equivalence relation $\simeq_{\text{pub}}$ iff for every C' such that $C \simeq_{\text{pub}} C'$:

$$C \; {}_D\Downarrow_O \; C_1 \text{ iff } C' \; {}_D\Downarrow_{O'} \; C_1' \text{ and } C_1 \simeq_{\text{pub}} C_1' \text{ and } O = O'.$$

A program satisfies SCT iff every initial configuration satisfies SCT under any schedule.

Aside, on sequential execution. Processors work hard to create the illusion that assembly instructions are executed sequentially. We validate our semantics by proving equivalence with respect to sequential execution. Formally, we

define *sequential schedules* as schedules that execute and retire instructions immediately upon fetching them. We attach to each program a canonical sequential schedule and write $C \Downarrow^N_{seq} C'$ to model execution under this canonical schedule. Our sequential validation is defined relative to an equivalence $\approx$ on configurations. Informally, two configurations are equivalent if their memories and register files are equal, even if their speculative states may be different.

Theorem 3.2 (Sequential equivalence)**.** *Let C be an initial configuration and D a well-formed schedule for C. If $C \Downarrow^N_D C_1$, then $C \Downarrow^N_{seq} C_2$ and $C_1 \approx C_2$.*

Complete definitions, more properties, and proofs are given in Appendix B.

3.2 Overview of the Semantics

As shown in Table 1, each instruction has a *physical* form and one or more *transient* forms. Our semantics operates on these instructions similar to a multi-stage processor pipeline. Physical instructions are *fetched* from memory and become transient instructions in the reorder buffer. They are then *executed* until they are fully resolved. Finally they are *retired*, updating the non-speculative state in the configuration.

In the rest of this section, we show how we model speculative execution (Section 3.3), memory operations (Section 3.4), aliasing prediction (Section 3.5), and fence instructions (Section 3.6). We also briefly describe indirect jumps and function calls (Section 3.7), which are presented in full in Appendix A.

Our semantics captures a variety of existing Spectre variants, including v1 (Figure 1), v1.1 (Figure 6), and v4 (Figure 7), as well as a new hypothetical variant (Figure 2). Additional variants (e.g., v2 and *ret2spec*) can be expressed with the extended semantics given in Appendix A. Our semantics shows that these attacks violate SCT by producing observations depending on secrets.

3.3 Speculative Execution

We start with the semantics for *conditional branches* which introduce speculative execution.

Conditional branching. The physical instruction for conditional branches has the form $\mathsf{br}(op, \overrightarrow{rv}, n^{\mathsf{true}}, n^{\mathsf{false}})$, where op is a Boolean operator whose result determines whether or not to execute the jump, $\overrightarrow{rv}$ are the operands to op, and n^{true} and n^{false} are the program points for the *true* and *false* branches, respectively.

We show br's transient counterparts in Table 1. The unresolved form extends the physical instruction with a program point n_0, which is used to record the branch that is executed (n^{true} or n^{false}) speculatively, and may or may not correspond to the branch that is actually taken once op is resolved. The resolved form contains the final jump target.

Fetch. We give the rule for the fetch stage below.

$$\textsc{cond-fetch}\quad \frac{\mu(n) = \mathsf{br}(op, \overrightarrow{rv}, n^{\mathsf{true}}, n^{\mathsf{false}}) \qquad i = \mathsf{MAX}(buf) + 1 \qquad buf' = buf[i \mapsto \mathsf{br}(op, \overrightarrow{rv}, n^{\mathsf{true}}, (n^{\mathsf{true}}, n^{\mathsf{false}}))]}{(\rho, \mu, n, buf) \xhookrightarrow[\text{fetch: true}]{} (\rho, \mu, n^{\mathsf{true}}, buf')}$$

The COND-FETCH rule speculatively executes the branch determined by a Boolean value b given by the directive. We show the case for $b = \mathsf{true}$; the case for false is analogous. The rule updates the current program point n, allowing execution to continue along the specified branch. The rule then records the chosen branch n^{true} (resp. n^{false}) in the transient jump instruction.

This semantics models the behavior of most modern processors. Since the target of the branch cannot be resolved in the fetch stage, speculation allows execution to continue and not stall until the branch target is resolved. In hardware, a branch predictor chooses which branch to execute; in our semantics, the directives fetch: true and fetch: false determine which of the rules to execute. This allows us to abstract over all possible predictor implementations.

Execute. Next, we describe the rules for the execute stage.

$$\textsc{cond-execute-correct}\quad \frac{buf(i) = \mathsf{br}(op, \overrightarrow{rv}, n_0, (n^{\mathsf{true}}, n^{\mathsf{false}})) \qquad \forall j < i : buf(j) \neq \mathsf{fence} \qquad (buf +_i \rho)(\overrightarrow{rv}) = \overrightarrow{v_\ell} \qquad [\![op(\overrightarrow{v_\ell})]\!] = \mathsf{true}_\ell \qquad n^{\mathsf{true}} = n_0 \qquad buf' = buf[i \mapsto \mathsf{jump}\ n^{\mathsf{true}}]}{(\rho, \mu, n, buf) \xhookrightarrow[\text{execute } i]{\mathsf{jump}\ n^{\mathsf{true}}_\ell} (\rho, \mu, n, buf')}$$

$$\textsc{cond-execute-incorrect}\quad \frac{buf(i) = \mathsf{br}(op, \overrightarrow{rv}, n_0, (n^{\mathsf{true}}, n^{\mathsf{false}})) \qquad \forall j < i : buf(j) \neq \mathsf{fence} \qquad (buf +_i \rho)(\overrightarrow{rv}) = \overrightarrow{v_\ell} \qquad [\![op(\overrightarrow{v_\ell})]\!] = \mathsf{true}_\ell \qquad n^{\mathsf{true}} \neq n_0 \qquad buf' = buf[j : j < i][i \mapsto \mathsf{jump}\ n^{\mathsf{true}}]}{(\rho, \mu, n, buf) \xhookrightarrow[\text{execute } i]{\mathtt{rollback},\mathtt{jump}\ n^{\mathsf{true}}_\ell} (\rho, \mu, n^{\mathsf{true}}, buf')}$$

Both rules evaluate the condition op via an evaluation function $[\![\cdot]\!]$. In both, the function produces true; but the false rules are analogous. The rules then compare the actual branch target n_{true} against the speculatively chosen target n_0 from the fetch stage.

If the *correct* path was chosen during speculation, i.e., n_0 agrees with the correct branch n^{true}, rule COND-EXECUTE-CORRECT updates buf with the fully resolved jump instruction and emits an observation: $\mathtt{jump}\ n^{\mathsf{true}}_\ell$. This models an attacker that can observe control flow, e.g., by timing executions along different paths. The leaked observation n^{true} has label ℓ, propagated from the evaluation of the condition.

In case the *wrong* path was taken during speculation, i.e., the calculated branch n^{true} *disagrees* with n_0, the semantics must roll back all execution steps along the erroneous path. For this, rule COND-EXECUTE-INCORRECT removes all entries

(a) Predicted correctly

i	Initial $buf(i)$	$buf(i)$ after execute $\overline{4}$
$\overline{3}$	$(r_b = 4)$	$(r_b = 4)$
$\overline{4}$	$\text{br}(<, (2, r_a), \underline{9}, (\underline{9}, \underline{12}))$	$\text{jump } \underline{9}$
$\overline{5}$	$(r_c = \text{op}(+, (1, r_b)))$	$(r_c = \text{op}(+, (1, r_b)))$

(b) Predicted incorrectly

i	Initial $buf(i)$	$buf(i)$ after execute $\overline{4}$
$\overline{3}$	$(r_b = 4)$	$(r_b = 4)$
$\overline{4}$	$\text{br}(<, (2, r_a), \underline{12}, (\underline{9}, \underline{12}))$	$\text{jump } \underline{9}$
$\overline{5}$	$(r_d = \text{op}(*, (r_g, r_h)))$	-

Figure 4. Correct and incorrect branch prediction. Initially, $r_a = 3$. In (b), the misprediction causes a rollback to $\overline{4}$.

in *buf* that are newer than the current instruction (i.e., all entries $j \geq i$), sets the program point n to the correct branch, and updates *buf* at index i with correct value for the resolved jump instruction. Since an attacker can observer misspeculation through instruction timing [20], the rule issues a `rollback` observation in addition to the `jump` observation.

Retire. The rule for the retire stage is shown below; its only effect is to remove the jump instruction from the buffer.

JUMP-RETIRE

$$\frac{\text{MIN}(buf) = i \qquad buf(i) = \text{jump } n_0 \qquad buf' = buf \setminus buf(i)}{(\rho, \mu, n, buf) \xhookrightarrow[\text{retire}]{} (\rho, \mu, n, buf')}$$

Examples. Figure 4 shows how branch prediction affects the reorder buffer. In part (a), the branch at index $\overline{4}$ is predicted correctly. The jump instruction is resolved, and execution proceeds as normal. In part (b), the branch at index $\overline{4}$ is incorrectly predicted. Upon executing the branch, the misprediction is detected, and *buf* is rolled back to index $\overline{4}$.

3.4 Memory Operations

The physical instruction for loads is $(r = \text{load}(\overrightarrow{rv}, n'))$, while the form for stores is $\text{store}(rv, \overrightarrow{rv}, n')$. As before, n' is the program point of the next instruction. For loads, r is the register receiving the result; for stores, rv is the register or value to be stored. For both loads and stores, $\overrightarrow{rv}$ is a list of operands (registers and values) which are used to calculate the operation's target address.

Transient counterparts of load and store are given in Table 1. We annotate unresolved load instructions with the program point of the physical instruction that generated them; we omit annotations whenever not used. Unresolved and resolved store instructions share the same syntax, but for resolved stores, both address and operand are required to be single values.

Address calculation. We assume an arithmetic operator *addr* which calculates target addresses for stores and loads from its operands. We leave this operation abstract in order to model a large variety of architectures. For example, in a simple addressing mode, $[\![addr(\vec{v})]\!]$ might compute the sum of its operands; in an x86-style address mode, $[\![addr([v_1, v_2, v_3])]\!]$ might instead compute $v_1 + v_2 \cdot v_3$.

Store forwarding. Multiple transient load and store instructions may exist concurrently in the reorder buffer. In particular, there may be unresolved loads and stores that will read or write to the same address in memory. Under a naive model, we must wait to execute load instructions until all prior store instructions have been retired, in case they write to the address we will load from. Indeed, some real-world processors behave exactly this way [10].

For performance, most modern processors implement *store-forwarding* for memory operations: if a load reads from the same address as a prior store and the store has already been resolved, the processor can *forward* the resolved value to the load. The load can then proceed without waiting for the store to commit to memory [34].

To model these store forwarding semantics, we use annotations to recall if a load was resolved from memory or forwarding. A resolved load has the form $(r = v_\ell\{j, a\})^n$, where the index j records either the buffer index of the store instruction that forwarded its value to the load, or $\perp$ if the value was taken from memory. We also record the memory address a associated with the data, and retain the program point n of the load instruction that generated the value instruction. The resolved load otherwise behaves as a resolved value instruction (e.g., for the register resolve function).

Fetch. We now discuss the inference rules for memory operations, starting with the fetch stage.

SIMPLE-FETCH

$$\frac{\mu(n) \in \{\text{op}, \text{load}, \text{store}, \text{fence}\} \qquad n' = next(\mu(n)) \qquad i = \text{MAX}(buf) + 1 \qquad buf' = buf[i \mapsto transient(\mu(n))]}{(\rho, \mu, n, buf) \xhookrightarrow[\text{fetch}]{} (\rho, \mu, n', buf')}$$

Given a fetch directive, rule SIMPLE-FETCH extends the reorder buffer *buf* with a new transient instruction (see Table 1). Other than load and store, the rule also applies to op and fence instructions. The $transient(\cdot)$ function simply translates the physical instruction at $\mu(n)$ to its unresolved transient form. It inserts the new, transient instruction at the first empty index in *buf*, and sets the current program point to the next instruction n'. Note that $transient(\cdot)$ annotates the transient load instruction with its program point.

Load execution. Next, we cover the rules for the load execute stage.

LOAD-EXECUTE-NODEP

$$\frac{buf(i) = (r = \text{load}(\overrightarrow{rv}))^n \qquad \forall j < i : buf(j) \neq \text{fence} \qquad (buf +_i \rho)(\overrightarrow{rv}) = \overrightarrow{v_\ell} \qquad [\![addr(\overrightarrow{v_\ell})]\!] = a \qquad \ell_a = \sqcup\vec{\ell} \qquad \forall j < i : buf(j) \neq \text{store}(\_, a) \qquad \mu(a) = v_\ell \qquad buf' = buf[i \mapsto (r = v_\ell\{\perp, a\})^n]}{(\rho, \mu, n, buf) \xrightarrow[\text{execute } i]{\text{read } a_{\ell_a}} (\rho, \mu, n, buf')}$$

LOAD-EXECUTE-FORWARD

$$\frac{buf(i) = (r = \mathsf{load}(\overrightarrow{rv}))^n \qquad \forall j < i : buf(j) \neq \mathsf{fence} \\ (buf +_i \rho)(\overrightarrow{rv}) = \overrightarrow{v_\ell} \qquad [\![addr(\overrightarrow{v_\ell})]\!] = a \qquad \ell_a = \sqcup\overrightarrow{\ell} \\ max(j) < i : buf(j) = \mathsf{store}(\_, a) \wedge buf(j) = \mathsf{store}(v_\ell, a,) \\ buf' = buf[i \mapsto (r = v_\ell\{j, a\})^n]}{(\rho, \mu, n, buf) \xhookrightarrow[\text{execute } i]{\mathtt{fwd}\ a_{\ell_a}} (\rho, \mu, n, buf')}$$

Given an execute directive for buffer index i, under the condition that i points to an unresolved load, rule LOAD-EXECUTE-NODEP applies if there are no prior store instructions in buf that have a resolved, matching address. The rule first resolves the operand list $\overrightarrow{rv}$ into a list of values $\overrightarrow{v_\ell}$, and then uses $\overrightarrow{v_\ell}$ to calculate the target address a. It then retrieves the current value v_ℓ at address a from memory, and finally adds to the buffer a resolved value instruction assigning v_ℓ to the target register r. We annotate the value instruction with the address a and $\bot$, signifying that the value comes from memory. Finally, the rule produces the observation $\mathtt{read}\ a_{\ell_a}$, which renders the memory read at address a with label ℓ_a visible to an attacker.

Rule LOAD-EXECUTE-FORWARD applies if the most recent store instruction in buf with a resolved, matching address has a resolved data value. Instead of accessing memory, the rule forwards the value from the store instruction, annotating the new value instruction with the calculated address a and the index j of the originating store instruction. The rule produces a fwd observation with the labeled address a_{ℓ_a}. This observation captures that the attacker can determine (e.g., by observing the *absence* of memory access using a cache timing attack) that a forwarded value from address a was found in the buffer instead of loaded from memory.

Importantly, neither of the rules has to wait for prior stores to be resolved and can proceed speculatively. This can lead to memory hazards when a more recent store to the load's address has not been resolved yet; we show how to deal with hazards in the rules for the store instruction.

Store execution. We show the rules for stores below.

STORE-EXECUTE-VALUE

$$\frac{buf(i) = \mathsf{store}(rv, \overrightarrow{rv}) \qquad \forall j < i : buf(j) \neq \mathsf{fence} \\ (buf +_i \rho)(rv) = v_\ell \qquad buf' = buf[i \mapsto \mathsf{store}(v_\ell, \overrightarrow{rv})]}{(\rho, \mu, n, buf) \xhookrightarrow[\text{execute } i\text{: value}]{} (\rho, \mu, n, buf')}$$

STORE-EXECUTE-ADDR-OK

$$\frac{buf(i) = \mathsf{store}(rv, \overrightarrow{rv}) \qquad \forall j < i : buf(j) \neq \mathsf{fence} \\ (buf +_i \rho)(\overrightarrow{rv}) = \overrightarrow{v_\ell} \qquad [\![addr(\overrightarrow{v_\ell})]\!] = a \qquad \ell_a = \sqcup\overrightarrow{\ell} \\ \forall k > i : buf(k) = (r = \ldots\{j_k, a_k\}) : \\ (a_k = a \implies j_k \geq i) \wedge (j_k = i \implies a_k = a) \\ buf' = buf[i \mapsto \mathsf{store}(rv, a_{\ell_a})]}{(\rho, \mu, n, buf) \xhookrightarrow[\text{execute } i\text{: addr}]{\mathtt{fwd}\ a_{\ell_a}} (\rho, \mu, n, buf')}$$

STORE-EXECUTE-ADDR-HAZARD

$$\frac{buf(i) = \mathsf{store}(rv, \overrightarrow{rv}) \qquad \forall j < i : buf(j) \neq \mathsf{fence} \\ (buf +_i \rho)(\overrightarrow{rv}) = \overrightarrow{v_\ell} \qquad [\![addr(\overrightarrow{v_\ell})]\!] = a \qquad \ell_a = \sqcup\overrightarrow{\ell} \\ min(k) > i : buf(k) = (r = \ldots\{j_k, a_k\})^{n_k} : \\ (a_k = a \wedge j_k < i) \vee (j_k = i \wedge a_k \neq a) \\ buf' = buf[j : j < k][i \mapsto \mathsf{store}(rv, a_{\ell_a})]}{(\rho, \mu, n, buf) \xhookrightarrow[\text{execute } i\text{: addr}]{\mathtt{rollback},\mathtt{fwd}\ a_{\ell_a}} (\rho, \mu, n_k, buf')}$$

The execution of store is split into two steps: value resolution, represented by the directive execute i : value, and address resolution, represented by the directive execute i : addr; a schedule may have either step first. Either step may be skipped if data or address are already in immediate form.

Rule STORE-EXECUTE-ADDR-OK applies if no misprediction has been detected, i.e., if no load instruction forwarded data from an outdated store. We check this by requiring that all value instructions *after* the current index (indices $k > i$) with an address a matching the current store must be using a value forwarded from a store *at least as recent* as this one ($a_k = a \implies j_k \geq i$). We define $\bot < n$ for any index n—that is, if a future load matches the address of the current store but loaded its value from memory, we consider this a hazard.

If there is indeed a hazard, i.e., if there was a resolved load with an outdated value, the rule STORE-EXECUTE-ADDR-HAZARD picks the *earliest* such instruction (index k) and restarts execution by resetting the instruction pointer to the program point n_k of this instruction. It then discards all transient instructions at indices at least k from the reorder buffer. As in the case of misspeculation, the rule issues a rollback observation.

Retire. Resolved loads are retired using the following rule.

VALUE-RETIRE

$$\frac{\mathsf{MIN}(buf) = i \qquad buf(i) = (r = v_\ell) \\ \rho' = \rho[r \mapsto v_\ell] \qquad buf' = buf \setminus buf(i)}{(\rho, \mu, n, buf) \xhookrightarrow[\text{retire}]{} (\rho', \mu, n, buf')}$$

This is the same retire rule used for simple value instructions (e.g., resolved op instructions). The rule updates the register map ρ with the new value, and removes the instruction from the reorder buffer.

Stores are retired using the rule below.

STORE-RETIRE

$$\frac{\mathsf{MIN}(buf) = i \qquad buf(i) = \mathsf{store}(v_\ell, a_{\ell_a}) \\ \mu' = \mu[a \mapsto v_\ell] \qquad buf' = buf \setminus buf(i)}{(\rho, \mu, n, buf) \xhookrightarrow[\text{retire}]{\mathtt{write}\ a_{\ell_a}} (\rho, \mu', n, buf')}$$

A fully resolved store instruction retires similarly to a value instruction. However, instead of updating the register map ρ, rule STORE-RETIRE updates the memory μ. Since an attacker can observe memory writes, the rule produces the observation $\mathtt{write}\ a_{\ell_a}$ with the labeled address of the store.

Example. Figure 5 gives an example of store-to-load forwarding. In the starting configuration, the store at index

Registers	$\rho(r_a) = 40_{\text{pub}}$
Directives	D= execute $\overline{4}$; execute $\overline{3}$: addr
Leakage for D	fwd 43_{pub}; rollback, fwd 43_{pub}

starting *buf*	*buf* after execute $\overline{4}$	*buf* after D
$\overline{2}$ store$(12, 43_{\text{pub}})$	$\overline{2}$ store$(12, 43_{\text{pub}})$	$\overline{2}$ store$(12, 43_{\text{pub}})$
$\overline{3}$ store$(20, [3, r_a])$	$\overline{3}$ store$(20, [3, r_a])$	$\overline{3}$ store$(20, 43_{\text{pub}})$
$\overline{4}$ $(r_c = \text{load}([43]))$	$\overline{4}$ $(r_c = 12\{\overline{2}, 43\})$	

Figure 5. Store hazard caused by late execution of store addresses. The store address for $\overline{3}$ is resolved too late, causing the later load instruction to forward from the wrong store. When $\overline{3}$'s address is resolved, the execution must be rolled back. In this example, $[\![addr(\cdot)]\!]$ adds its arguments.

Registers

r	$\rho(r)$
r_a	5_{pub}
r_b	x_{sec}

Memory

a	$\mu(a)$
40..43	$secretKey_{\text{sec}}$
44..47	$pubArrA_{\text{pub}}$
48..4B	$pubArrB_{\text{pub}}$

Reorder buffer

i	$buf(i)$
$\overline{1}$	br$(>, (4, r_a), \underline{2}, (\underline{2}, \underline{4}))$
$\overline{2}$	store$(r_b, [40, r_a])$
	...
$\overline{7}$	$(r_c = \text{load}([45]))$
$\overline{8}$	$(r_c = \text{load}([48, r_c]))$

Directive	Effect on *buf*	Leakage
execute $\overline{2}$: addr	$\overline{2} \mapsto \text{store}(r_b, 45_{\text{pub}})$	fwd 45_{pub}
execute $\overline{2}$: value	$\overline{2} \mapsto \text{store}(x_{\text{sec}}, 45_{\text{pub}})$	
execute $\overline{7}$	$\overline{7} \mapsto (r_c = x_{\text{sec}}\{\overline{2}, 45\})$	fwd 45_{pub}
execute $\overline{8}$	$\overline{8} \mapsto (r_c = X\{\bot, a\})$	read a_{sec}
	where $a = x_{\text{sec}} + 48$	

Figure 6. Example demonstrating a store-to-load Spectre v1.1 attack. A speculatively stored value is forwarded and then leaked using a subsequent load instruction.

Registers

r	$\rho(r)$
r_a	40_{pub}

Memory

a	$\mu(a)$
40..43	$secretKey_{\text{sec}}$
44..47	$pubArrA_{\text{pub}}$

Reorder buffer

i	$buf(i)$
$\overline{2}$	store$(0, [3, r_a])$
$\overline{3}$	$(r_c = \text{load}([43]))$
$\overline{4}$	$(r_c = \text{load}([44, r_c]))$

Directive	Effect on *buf*	Leakage
execute $\overline{3}$	$\overline{3} \mapsto (r_c = secretKey[3]\{\bot, 43\})$	read 43_{pub}
execute $\overline{4}$	$\overline{4} \mapsto (r_c = X\{\bot, a\})$	read a_{sec}
execute $\overline{2}$: addr	$\{\overline{3}, \overline{4}\} \notin buf$ $\overline{2} \mapsto \text{store}(0, 43_{\text{pub}})$	rollback, fwd 43_{pub}
	where $a = secretKey[3]_{\text{sec}} + 44$	

Figure 7. Example demonstrating a v4 Spectre attack. The store is executed too late, causing later load instructions to use outdated values.

$\overline{2}$ is fully resolved, while the store at index $\overline{3}$ has an unresolved address. The first directive executes the load at $\overline{4}$. This load accesses address 43, which matches the store at index $\overline{2}$. Since this is the most recent such store and has a resolved value, the load gets the value 12 from this store. The following directive resolves the address of the store at index $\overline{3}$. This store also matches address 43. As this store is more recent than store $\overline{2}$, this directive triggers a hazard for the load at $\overline{4}$, leading to the rollback of the load from the reorder buffer.

Capturing Spectre. We now have enough machinery to capture several variants of Spectre attacks.

We discussed how our semantics model Spectre v1 in Section 2 (Figure 1). Figure 6 shows a simple disclosure gadget using forwarding from an out-of-bounds write. In this example, a secret value x_{sec} is supposed to be written to *secretKey* at an index r_a as long as r_a is within bounds. However, due to branch misprediction, the store instruction is executed despite r_a being too large. The load instruction at index $\overline{7}$, normally benign, now aliases with the store at index $\overline{2}$, and receives the secret x_{sec} instead of a public value from *pubArrA*. This value is then used as the address of another load instruction, causing x_{sec} to leak.

Figure 7 shows a Spectre v4 vulnerability caused when a store *fails* to forward to a future load. In this example, the load at index $\overline{3}$ executes before the store at $\overline{2}$ calculates its address. As a result, this execution loads the outdated secret value at address 43 and leaks it, instead of using the public zeroed-out value that would be written.

3.5 Aliasing Prediction

We extend the memory semantics from the previous section to model aliasing prediction by introducing a new transient instruction $(r = \text{load}(\overrightarrow{rv}, (v_\ell, j)))^n$. This instruction represents a *partially resolved* load with speculatively forwarded data. As before, r is the target register, $\overrightarrow{rv}$ is the list of arguments for address calculation, and n is the program point of the physical load instruction. The new parameters are v_ℓ, the forwarded data, and j, the index of the originating store.

Forwarding via prediction.

LOAD-EXECUTE-FORWARDED-GUESSED

$$\frac{buf(i) = (r = \text{load}(\overrightarrow{rv}))^n \qquad j < i \qquad \forall k < i : buf(k) \neq \text{fence} \qquad buf(j) = \text{store}(v_\ell, \overrightarrow{rv}_j) \qquad buf' = buf[i \mapsto (r = \text{load}(\overrightarrow{rv}, (v_\ell, j)))^n]}{(\rho, \mu, n, buf) \xrightarrow[\text{execute } i: \text{fwd } j]{} (\rho, \mu, n, buf')}$$

Rule LOAD-EXECUTE-FORWARDED-GUESSED implements forwarding in the presence of unresolved target addresses. Instead of forwarding the value from a store with a matching address, as in Section 3.4, the attacker can now freely choose to forward from *any* store with a resolved value—even if its target address is not known yet. Given a choice of which store j to forward from—supplied via directive—the rule updates the reorder buffer with the new partially resolved load

and records both the forwarded value v_l and the buffer index j of the store instruction.

Register resolve function. We extend the register resolve function $(buf +_i \rho)$ to allow using values from partially resolved loads. In particular, whenever the register resolve function computes the latest resolved assignment to some register r, it now considers not only fully resolved value instructions, but also our new partially resolved load: whenever the latest assignment in the buffer is a partially resolved load, the register resolve function returns its value.

We now discuss the execution rules, where partially resolved loads may fully resolve against either the originating store or against memory.

Resolving when originating store is in the reorder buffer.

LOAD-EXECUTE-ADDR-OK

$$\frac{\begin{array}{c} buf(i) = (r = \mathsf{load}(\overrightarrow{rv}, (v_\ell, j)))^n \\ (buf +_i \rho)(\overrightarrow{rv}) = \overrightarrow{v_\ell} \qquad [\![addr(\overrightarrow{v_\ell})]\!] = a \\ \ell_a = \sqcup\vec{\ell} \qquad buf(j) = \mathsf{store}(v_\ell, \overrightarrow{rv}_j) \wedge (\overrightarrow{rv}_j = a' \Rightarrow a' = a) \\ \forall k : (j < k < i) : buf(k) \neq \mathsf{store}(\_, a) \\ buf' = buf[i \mapsto (r = v_\ell\{j, a\})^n] \end{array}}{(\rho, \mu, n, buf) \xrightarrow[\text{execute } i]{\mathsf{fwd}\ a_{\ell_a}} (\rho, \mu, n, buf')}$$

LOAD-EXECUTE-ADDR-HAZARD

$$\frac{\begin{array}{c} buf(i) = (r = \mathsf{load}(\overrightarrow{rv}, (v_\ell, j)))^{n'} \\ (buf +_i \rho)(\overrightarrow{rv}) = \overrightarrow{v_\ell} \qquad [\![addr(\overrightarrow{v_\ell})]\!] = a \\ \ell_a = \sqcup\vec{\ell} \qquad (buf(j) = \mathsf{store}(v_\ell, a') \wedge a' \neq a) \vee \\ (\exists k : j < k < i \wedge buf(k) = \mathsf{store}(\_, a)) \\ buf' = buf[j : j < i] \end{array}}{(\rho, \mu, n, buf) \xrightarrow[\text{execute } i]{\mathsf{rollback},\mathsf{fwd}\ a_{\ell_a}} (\rho, \mu, n', buf')}$$

To resolve $(r = \mathsf{load}(\overrightarrow{rv}, (v_\ell, j)))^n$ when its originating store is still in buf, we calculate the load's actual target address a and compare it against the target address of the originating store at $buf(j)$. If the store is not followed by later stores to a, and either **(1)** the store's address is resolved and its address is indeed a, or **(2)** the store's address is still unresolved, we update the reorder buffer with an annotated value instruction (rule LOAD-EXECUTE-ADDR-OK).

If, however, either the originating store resolved to a *different* address (mispredicted aliasing) or a later store resolved to the same address (hazard), we roll back our execution to just before the load (rule LOAD-EXECUTE-ADDR-HAZARD).

We allow the load to execute even if the originating store has not yet resolved its address. When the store does finally resolve its address, it must check that the addresses match and that the forwarding was correct. The gray formulas in STORE-EXECUTE-ADDR-OK and STORE-EXECUTE-ADDR-HAZARD (Section 3.4) perform these checks: For forwarding to be correct, all values forwarded from a store at $buf(i)$ must have a matching annotated address ($\forall k > i : j_k = i \Rightarrow a_k = a$). Otherwise, if any value annotation has a mismatched address, then the instruction is rolled back ($j_k = i \wedge a_k \neq a$).

Resolving when originating store is not in the buffer. We must also consider the case where we have delayed resolving the load address to the point where the originating store has already retired, and is no longer available in buf. If this is the case, and no other prior store instructions have a matching address, then we must check the forwarded data against memory.

LOAD-EXECUTE-ADDR-MEM-MATCH

$$\frac{\begin{array}{c} buf(i) = (r = \mathsf{load}(\overrightarrow{rv}, v_\ell, j))^n \\ j \notin buf \qquad (buf +_i \rho)(\overrightarrow{rv}) = \overrightarrow{v_\ell} \qquad \ell_a = \sqcup\vec{\ell} \\ [\![addr(\overrightarrow{v_\ell})]\!] = a \qquad \forall k < i : buf(k) \neq \mathsf{store}(\_, a) \\ \mu(a) = v_\ell \qquad buf' = buf[i \mapsto (r = v_\ell\{\bot, a\})^n] \end{array}}{(\rho, \mu, n, buf) \xrightarrow[\text{execute } i]{\mathsf{read}\ a_{\ell_a}} (\rho, \mu, n, buf')}$$

LOAD-EXECUTE-ADDR-MEM-HAZARD

$$\frac{\begin{array}{c} buf(i) = (r = \mathsf{load}(\overrightarrow{rv}, v_\ell, j))^{n'} \\ j \notin buf \qquad (buf +_i \rho)(\overrightarrow{rv}) = \overrightarrow{v_\ell} \qquad \ell_a = \sqcup\vec{\ell} \\ [\![addr(\overrightarrow{v_\ell})]\!] = a \qquad \forall k < i : buf(k) \neq \mathsf{store}(\_, a) \\ \mu(a) = v'_{\ell'} \qquad v'_{\ell'} \neq v_\ell \qquad buf' = buf[j : j < i] \end{array}}{(\rho, \mu, n, buf) \xrightarrow[\text{execute } i]{\mathsf{rollback},\mathsf{read}\ a_{\ell_a}} (\rho, \mu, n', buf')}$$

If the originating store has retired, and no intervening stores match the same address, we must load the value from memory to ensure we were originally forwarded the correct value. If the value loaded from memory matches the value we were forwarded, we update the reorder buffer with a resolved load annotated as if it had been loaded from memory (rule LOAD-EXECUTE-ADDR-MEM-MATCH).

If a store *different* from the originating store overwrote the originally forwarded value, the value loaded from memory may not match the value we were originally forwarded. In this case we roll back execution to just before the load (rule LOAD-EXECUTE-ADDR-MEM-HAZARD).

We demonstrate these semantics in the attack shown in Figure 2. An earlier draft of this paper [7] incorrectly claimed to have a proof-of-concept exploit for this attack on real hardware.

3.6 Speculation Barriers

We extend our semantics with a *speculation barrier* instruction, fence n, that prevents further speculative execution until all prior instructions have been retired.

FENCE-RETIRE

$$\frac{\mathsf{MIN}(buf) = i \qquad buf(i) = \mathsf{fence} \qquad buf' = buf \setminus buf(i)}{(\rho, \mu, n, buf) \xrightarrow[\text{retire}]{} (\rho, \mu, n, buf')}$$

The fence instruction uses SIMPLE-FETCH as its fetch rule, and its rule for retire only removes the instruction from the buffer. It does not have an execute rule. However, fence instructions affect the execution of all instructions in the reorder buffer that come *after* them. In prior sections, execute rules have the highlighted condition $\forall j < i : buf(j) \neq \mathsf{fence}$.

Before executing $\overline{1}$	
i	$buf[i]$
$\overline{1}$	$\mathsf{br}(>, (4, r_a), \underline{2}, (\underline{2}, \underline{5}))$
$\overline{2}$	fence
$\overline{3}$	$(r_b = \mathsf{load}([\mathtt{40}, r_a]))$
$\overline{4}$	$(r_c = \mathsf{load}([\mathtt{44}, r_b]))$

After	
i	$buf[i]$
$\overline{1}$	jump $\underline{5}$

Figure 8. Example demonstrating fencing mitigation against Spectre v1 attacks. The fence instruction prevents the load instructions from executing before the br.

This condition ensures that as long as a fence instruction remains in buf, any instructions fetched after the fence cannot be executed.

We use fence instructions to restrict out-of-order execution in our semantics. Notably, we can use it to prevent attacks of the forms shown in Figures 1, 6 and 7.

Example. The example in Figure 8 shows how placing a fence instruction just after the br instruction prevents the Spectre v1 attack from Figure 1. The fence in this example prevents the load instructions at $\overline{2}$ and $\overline{3}$ from executing and forces the br to be resolved first. Evaluating the br exposes the misprediction and causes the two loads (as well as the fence) to be rolled back.

3.7 Indirect Jumps and Return Address Prediction

Finally, we briefly discuss two additional extensions to our semantics. First, we extend our semantics with *indirect jumps*. Rather than specifying jump targets *directly* as with the br instruction in Section 3.3, indirect jumps compute the target from a list of argument operands. The indirect jump instruction has the form $\mathsf{jmpi}(\overrightarrow{rv})$, where $\overrightarrow{rv}$ is the list of operands for calculating the jump target. The transient form of jmpi is $\mathsf{jmpi}(\overrightarrow{rv}, n_0)$, where n_0 is the predicted jump target. To fetch a jmpi instruction, we use fetch: n', where n' is the speculated jump target. In all other respects, the rules for indirect jump instructions are similar to the rules for conditional branches.

Second, we extend our semantics with *call* and *ret* instructions. The call instruction has the form $\mathsf{call}(n_f, n_{ret})$, where n_f is the callee program point and n_{ret} is the program point to return to. The return instruction is simply ret. Both the call and ret instructions have the simple transient forms call and ret. However, when fetched, they are unpacked into multiple transient instructions. Fetching a call produces the call transient instruction as well as transient instructions which will increment a stack pointer and store the return program point to memory. Fetching a ret produces a corresponding load, decrement, and jump as well as the ret transient instruction. Furthermore, the call and ret instructions respectively push and pop program points to an additional configuration state representing the *return stack buffer* (RSB). The RSB is used to predict the new program point upon fetching a ret.

In Appendix A, we present detailed rules for indirect jumps, functions calls, and returns. We also show how both Spectre v2 [20] and ret2spec [23] attacks can be expressed in our semantics, as well as the *retpoline* mitigation [31] against Spectre v2 attacks.

4 Detecting Violations

We develop a tool Pitchfork based on our semantics to check for SCT violations. Pitchfork first generates a set of schedules representing various *worst-case* attackers. This set of schedules is far smaller than the set of all possible schedules for the program, but is nonetheless sound: if there is an SCT violation in any possible schedule, then there will be an SCT violation in one of the worst-case schedules. Pitchfork then checks for secret leakage by symbolically executing the program under each schedule.

Pitchfork only exercises a subset of our semantics; it does not detect SCT violations based on alias prediction, indirect jumps, or return stack buffers (Sections 3.5 and 3.7). Doing so would require it to generate a prohibitively large number of schedules. Nevertheless, Pitchfork still exposes attacks based on Spectre variants 1, 1.1, and 4.

We describe our schedule generation in Section 4.1, and evaluate Pitchfork on several crypto libraries in Section 4.2.

4.1 Schedule Generation

Given a program, Pitchfork generates a set of schedules representing various worst-case attackers. Pitchfork's schedule generation is parametrized by a *speculation bound*, which limits the size of the reorder buffer, and thus the depth of speculation.

In general, Pitchfork constructs worst-case schedules to maximize speculation. These schedules eagerly fetch instructions until the reorder buffer is full, i.e., the size of the reorder buffer equals the speculation bound. Once the reorder buffer is full, the schedules only retire instructions as necessary to fetch new ones.

When conditional branches are to be fetched, Pitchfork constructs schedules containing both possible outcomes: one where the branch is guessed true (fetch: true) and one where the branch is guessed false (fetch: false). For the mispredicted outcome, Pitchfork's schedules execute the branch as late as possible (i.e., it is the oldest instruction in the reorder buffer and the reorder buffer is full), which delays the rollback of mispredicted paths.

To account for the load-store forwarding hazards described in Section 3.4, Pitchfork constructs schedules containing all possible forwarding outcomes. For every load instruction l in the program, Pitchfork finds all prior stores s_i within the speculation bound that would resolve to the same address. Then, for each such store, Pitchfork constructs a schedule that would cause that store to forward its data to l. That is, Pitchfork constructs separate schedules [execute s_1 :

addr; execute l], [execute s_2 : addr; execute l], and so on. Additionally, Pitchfork constructs a schedule where none of the prior stores s_i have resolved addresses, forcing the load instruction to read from memory.

For all instructions other than conditional branches and memory operations, Pitchfork only constructs schedules where these instructions are executed eagerly and in order. Reordering of these instructions is uninteresting: either the instructions naturally commute, or data dependencies prevent the reordering (i.e., the reordered schedule is invalid for the program). This intuition matches with the property that any out-of-order execution of a given program has the same final result regardless of its schedule.

We formalize the soundness of Pitchfork's schedule construction in more detail in Appendix B.3.

4.2 Implementation and Evaluation

We implement Pitchfork on top of the `angr` binary-analysis tool [30]. Pitchfork uses `angr` to symbolically execute a given program according to each of its worst-case schedules, flagging any resulting secret leakage.

To sanity check Pitchfork, we create and analyze a set of Spectre v1 and v1.1 test cases, and ensure we flag their SCT violations. Our test cases are based off the well-known Kocher Spectre v1 examples [19]. Since many of the Kocher examples exhibit violations even during sequential execution, we create a new set of Spectre v1 test cases which only exhibit violations when executed speculatively. We also develop a similar set of test cases for Spectre v1.1 data attacks.

Pitchfork necessarily inherits the limitations of `angr`'s symbolic execution. For instance, `angr` concretizes addresses for memory operations instead of keeping them symbolic. Furthermore, exploring every speculative branch and potential store-forward within a given speculation bound leads to an explosion in state space. In our tests, we were able to support speculation bounds of up to 20 instructions. We were able to increase this bound to 250 instructions when we disabled checking for store-forwarding hazards. Though these bounds do not capture the speculation depth of some modern processors, Pitchfork still correctly finds SCT violations in all our test cases, as well as SCT violations in real-world crypto code. We consider the design and implementation of a more scalable tool future work.

4.2.1 Evaluation Procedure. To evaluate Pitchfork on real-world crypto implementations, we use the same case studies as FaCT [8], a domain-specific language and compiler for constant-time crypto code. We use FaCT's case studies for two reasons: these implementations have been verified to be (sequentially) constant-time, and their inputs have already been annotated by the FaCT authors with secrecy labels.[3]

We analyzed both the FaCT-generated binaries and the corresponding C binaries for the case studies. For each binary, we ran Pitchfork without forwarding hazard detection—only looking for Spectre v1 and v1.1 violations—and with a speculation bound of 250 instructions. If Pitchfork did not flag any violations, we re-enabled forwarding hazard detection—looking for Spectre v4 violations—and ran Pitchfork with a reduced bound of 20 instructions. The reduced bound ensured that the analysis was tractable.

[3] https://github.com/PLSysSec/fact-eval

Table 2. A 👻 indicates Pitchfork found an SCT violation. A $👻^f$ indicates the violation was found only with forwarding hazard detection.

Case Study	C	FaCT
curve25519-donna	✓	✓
libsodium `secretbox`	👻	✓
OpenSSL ssl3 record validate	👻	$👻^f$
OpenSSL MEE-CBC	👻	$👻^f$

```
1  for (int cnt = nlist - 1; cnt >= 0; --cnt) {
2    iov[cnt].iov_base = (char *) list->str;
3      // ...
4    list = list->next;
5  }
```

Figure 9. Vulnerable snippet from `__libc__message()`.[4]

4.2.2 Detected Violations. Table 2 shows our results. Pitchfork did not flag any SCT violations in the curve25519-donna implementations; this is not surprising, as the curve25519-donna library is a straightforward implementation of crypto primitives. Pitchfork did, however, find SCT violations (without forwarding hazard detection) in both the libsodium and OpenSSL codebases. Specifically, Pitchfork found violations in the C implementations of these libraries, in code ancillary to the core crypto routines. This aligns with our intuition that crypto primitives will not themselves be vulnerable to Spectre attacks, but higher-level code that interfaces with these primitives may still leak secrets. Such higher-level code is not present in the corresponding FaCT implementations, and Pitchfork did not find any violations in the FaCT code with these settings. However, with forwarding hazard detection, Pitchfork was able to find vulnerabilities even in the FaCT versions of the OpenSSL implementations. We describe two of the violations Pitchfork flagged next.

C libsodium secretbox. The libsodium codebase compiles with stack protection [15] turned on by default. This means that, for certain functions (e.g., functions with stack allocated `char` buffers), the compiler inserts code in the function epilogue to check if the stack was "smashed". If so, the program displays an error message and aborts. As part of printing the error message, the program calls a function `__libc_message`, which does `printf`-style string formatting.

```
1  aesni_cbc_encrypt(/* ... */);
2  // (len _out) is in %r14
3  secret mut uint32 pad = _out[len _out - 1];
4  public uint32 maxpad = tmppad > 255 ? 255 : tmppad;
5  if (pad > maxpad) {
6    pad = maxpad;
7    ret = 0; // overwrites %r14
8  }
9  // ...
10 _sha1_update(/* ... */); // can return to line 3
```

Figure 10. Vulnerable snippet from the FaCT OpenSSL MEE implementation.[5]

Figure 9 shows a snippet from this function which traverses a linked list. When running the C `secretbox` implementation speculatively, the processor may misspeculate on the stack tampering check and jump into the error handling code, eventually calling `__libc_message`. Again due to misspeculation, the processor may incorrectly proceed through the loop extra times, traversing non-existent links, eventually causing secret data to be stored into `list` instead of a valid address (line 4). On the following iteration of the loop, dereferencing `list` (line 2) causes a secret-dependent memory access.

FaCT OpenSSL MEE. In Figure 10, we show the code from the FaCT port of OpenSSL's authenticated encryption implementation. The FaCT compiler transforms the branch at lines 5-7 into straight-line constant-time code, since the variable `pad` is considered **`secret`**.

Initially, register `%r14` holds the length of the array `_out`. The processor leaks this value due to the array access on line 3; this is not a security violation, as the length is public. On line 7, the value of `%r14` is overwritten with `0` if `pad > maxpad`, or 1 (the initial value of `ret`) otherwise. Afterwards, the processor calls `_sha1_update`.

To return from `_sha1_update`, the processor must first load the return address from memory. When forwarding hazard detection is enabled, Pitchfork allows this load to speculatively receive data from stores *older* than the most recent store to that address (see Section 3.4). Specifically, it may receive the prior value that was stored at that location: the return address for the call to `aesni_cbc_encrypt`.

After the speculative return, the processor executes line 3 a second time. This time, `%r14` does not hold the public value **`len`** `_out`; it instead holds the value of `ret`, which was derived from the secret condition `pad > maxpad`. The processor thus accesses either `_out[0]` or `_out[-1]`, leaking information about the secret value of `pad` via cache state.

[4]Code snippet taken from https://github.com/lattera/glibc/blob/895ef79e04a953cac1493863bcae29ad85657ee1/sysdeps/posix/libc_fatal.c

[5]Code snippet taken from https://github.com/PLSysSec/fact-eval/blob/888bc6c6898a06cef54170ea273de91868ea621e/openssl-mee/20170717_latest.fact

5 Related Work

Prior work on modeling speculative or out-of-order execution is concerned with correctness rather than security [1, 21]. We instead focus on security and model side-channel leakage explicitly. Moreover, we abstract away the specifics of microarchitectural features, considering them to be adversarially controlled.

Disselkoen et al. [13] explore speculation and out-of-order effects through a relaxed memory model. Their semantics sits at a higher level, and is orthogonal to our approach. They do not define a semantic notion of security that prevents Spectre-like attacks, and do not provide support for verification.

Mcilroy et al. [24] reason about micro-architectural attacks using a multi-stage pipeline semantics (though they do not define a formal security property). Their semantics models branch predictor and cache state explicitly. However, they do not model the effects of speculative barriers, nor other microarchitecture features such as store-forwarding. Thus, their semantics can only capture Spectre v1 attacks.

Both Guarnieri et al. [17] and Cheang et al. [9] define speculative semantics that are supported by tools. Their semantics handle speculation through branch prediction—where the predictor is left abstract—but do not capture more general out-of-order execution nor other types of speculation. These works also propose new semantic notions of security (different from SCT); both essentially require that the speculative execution of a program not leak more than its sequential execution. If a program is sequentially constant-time, this additional security property is equivalent to our notion of speculative constant-time. Though our property is stronger, it is also simpler to verify: we can directly check SCT without first checking if a program is sequentially constant-time. And since we focus on cryptographic code, we directly require the stronger SCT property.

Balliu et al. [3] define a semantics in a style similar to ours. Their semantics captures various Spectre attacks, including an attack similar to our alias prediction example (Figure 2), and a new attack based on their memory ordering semantics, which our semantics cannot capture.

Finally, several tools detect Spectre vulnerabilities, but do not present semantics. The oo7 static analysis tool [33], for example, uses taint tracking to find Spectre attacks and automatically insert mitigations for several variants. Wu and Wang [35], on the other hand, perform cache analysis of LLVM programs under speculative execution, capturing Spectre v1 attacks.

6 Conclusion

We introduced a semantics for reasoning about side-channels under adversarially controlled out-of-order and speculative

execution. Our semantics capture existing transient execution attacks—namely Spectre—but can be extended to future hardware predictors and potential attacks. We also defined a new notion of constant-time code under speculation—speculative constant-time (SCT)—and implemented a prototype tool to check if code is SCT. Our prototype, Pitchfork, discovered new vulnerabilities in real-world crypto libraries.

There are several directions for future work. Our immediate plan is to use our semantics to prove the effectiveness of existing countermeasures (e.g., retpolines) and to justify new countermeasures.

Acknowledgments

We thank the anonymous PLDI and PLDI AEC reviewers and our shepherd James Bornholt for their suggestions and insightful comments. We thank David Kaplan from AMD for his detailed analysis of our proof-of-concept exploit that we incorrectly thought to be abusing an aliasing predictor. We also thank Natalie Popescu for her aid in editing and formatting this paper. This work was supported in part by gifts from Cisco and Fastly, by the NSF under Grant Number CCF-1918573, by ONR Grant N000141512750, and by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. 2009. The Semantics of Power and ARM Multiprocessor Machine Code. In *Proceedings of the 4th Workshop on Declarative Aspects of Multicore Programming*.

[2] Arm Mbed. [n.d.]. mbed TLS. Retrieved May 16, 2018 from https://github.com/armmbed/mbedtls

[3] Musard Balliu, Mads Dam, and Roberto Guanciale. 2019. *InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis.* arXiv:1911.00868 [cs.CR]

[4] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. 2014. System-level non-interference for constant-time cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM.

[5] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. 2019. Fallout: Leaking Data on Meltdown-resistant CPUs. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM.

[6] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *28th USENIX Security Symposium*. USENIX Association.

[7] Sunjay Cauligi, Craig Disselkoen, Klaus von Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2019. *Towards Constant-Time Foundations for the New Spectre Era.* arXiv:1910.01755v2

[8] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Gregoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. 2019. FaCT: A DSL for timing-sensitive computation. In *40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM.

[9] Kevin Cheang, Cameron Rasmussen, Sanjit Seshia, and Pramod Subramanyan. 2019. A Formal Approach to Secure Speculation. Cryptology ePrint Archive, Report 2019/310.

[10] Tien-Fu Chen and Jean-Loup Baer. 1992. Reducing Memory Latency via Non-blocking and Prefetching Caches. In *5th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM.

[11] Cryptography Coding Standard. 2016. Coding Rules. Retrieved June 9, 2017 from https://cryptocoding.net/index.php/Coding_rules

[12] Frank Denis. 2019. libsodium. Retrieved May 16, 2018 from https://github.com/jedisct1/libsodium

[13] Craig Disselkoen, Radha Jagadeesan, Alan Jeffrey, and James Riely. 2019. The Code That Never Ran: Modeling Attacks on Speculative Evaluation. In *40th IEEE Symposium on Security and Privacy*. IEEE.

[14] Dmitry Evtyushkin, Ryan Riley, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2018. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *23rd International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM.

[15] GCC Team. 2019. Using the GNU Compiler Collection (GCC): Instrumentation Options. Retrieved November 21, 2019 from https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html

[16] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2018. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering* (2018).

[17] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. 2019. *SPECTECTOR: Principled Detection of Speculative Information Flows.* arXiv:1812.08639

[18] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. 2019. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. In *28th USENIX Security Symposium*. USENIX Association.

[19] Paul Kocher. 2018. Spectre mitigations in Microsoft's C/C++ compiler. Retrieved April 6, 2020 from https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html

[20] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy*. IEEE.

[21] Shuvendu K. Lahiri, Sanjit A. Seshia, and Randal E. Bryant. 2002. Modeling and Verification of Out-of-Order Microprocessors in UCLID. In *International Conference on Formal Methods in Computer-Aided Design*. Springer.

[22] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium*. USENIX Association.

[23] Giorgi Maisuradze and Christian Rossow. 2018. ret2spec: Speculative Execution Using Return Stack Buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM.

[24] Ross McIlroy, Jaroslav Sevcik, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. 2019. *Spectre is here to stay: An analysis of side-channels and speculative execution.* arXiv:1902.05178

[25] OpenSSL. 2019. Security Policy. Retrieved April 6, 2020 from https://www.openssl.org/policies/secpolicy.html

[26] Thomas Pornin. 2016. Why Constant-Time Crypto? Retrieved November 15, 2018 from https://www.bearssl.org/constanttime.html

[27] Thomas Pornin. 2018. Constant-Time Toolkit. Retrieved November 15, 2018 from https://github.com/pornin/CTTK

[28] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. 2019. *Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs.* arXiv:1905.05725

[29] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM.

[30] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *37th IEEE Symposium on Security and Privacy*. IEEE.

[31] Paul Turner. 2019. Retpoline: a software construct for preventing branch-target-injection. Retrieved April 6, 2020 from https://support.google.com/faqs/answer/7625886

[32] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-Flight Data Load. In *40th IEEE Symposium on Security and Privacy*. IEEE.

[33] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. 2019. oo7: Low-overhead Defense against Spectre Attacks via Program Analysis. *IEEE Transactions on Software Engineering* (2019).

[34] Henry Wong. 2014. Store-to-Load Forwarding and Memory Disambiguation in X86 Processors. Retrieved April 6, 2020 from https://blog.stuffedcow.net/2014/01/x86-memory-disambiguation/

[35] Meng Wu and Chao Wang. 2019. Abstract Interpretation under Speculative Execution. In *40th SIGPLAN ACM Conference on Programming Language Design and Implementation*. ACM.

A Extended semantics

A.1 Indirect jumps

Semantics. The semantics for jmpi are given below:

JMPI-FETCH

$$\frac{\mu(n) = \mathsf{jmpi}(\overrightarrow{rv}) \qquad i = \mathsf{MAX}(buf) + 1 \qquad buf' = buf[i \mapsto \mathsf{jmpi}(\overrightarrow{rv}, n')]}{(\rho, \mu, n, buf) \xrightarrow[\text{fetch: } n']{} (\rho, \mu, n', buf')}$$

JMPI-EXECUTE-CORRECT

$$\frac{buf(i) = \mathsf{jmpi}(\overrightarrow{rv}, n_0) \qquad \forall j < i : buf(j) \neq \mathsf{fence} \qquad (buf +_i \rho)(\overrightarrow{rv}) = \overrightarrow{v_\ell} \qquad \ell = \sqcup\overrightarrow{\ell} \qquad [\![addr(\overrightarrow{v_\ell})]\!] = n_0 \qquad buf' = buf[i \mapsto \mathsf{jump}\ n_0]}{(\rho, \mu, n, buf) \xrightarrow[\text{execute } i]{\mathsf{jump}\ n_{0\ell}} (\rho, \mu, n, buf')}$$

JMPI-EXECUTE-INCORRECT

$$\frac{buf(i) = \mathsf{jmpi}(\overrightarrow{rv}, n_0) \qquad \forall j < i : buf[j] \neq \mathsf{fence} \qquad (buf +_i \rho)(\overrightarrow{rv}) = \overrightarrow{v_\ell} \qquad \ell = \sqcup\overrightarrow{\ell} \qquad [\![addr(\overrightarrow{v_\ell})]\!] = n' \neq n_0 \qquad buf' = buf[j : j < i][\, i \mapsto \mathsf{jump}\ n']}{(\rho, \mu, n, buf) \xrightarrow[\text{execute } i]{\mathsf{rollback}, \mathsf{jump}\ n'_\ell} (\rho, \mu, n', buf')}$$

When fetching a jmpi instruction, the schedule guesses the jump target n'. The rule records the operands and the guessed program point in a new buffer entry. In a real processors, the jump target guess is supplied by an indirect branch predictor; as branch predictors can be arbitrarily influenced by an adversary [14], we model the guess as an attacker directive.

Registers	
r	$\rho(r)$
r_a	1_{pub}
r_b	8_{pub}

Memory	
a	$\mu(a)$
44..47	*array* B_{pub}
48..4B	*array* Key_{sec}

Program	
n	$\mu(n)$
$\underline{1}$	$(r_c = \mathsf{load}([48, \mathsf{r_a}], \underline{2}))$
$\underline{2}$	fence $\underline{3}$
$\underline{3}$	jmpi$([12, r_b])$
	…
$\underline{16}$	fence $\underline{17}$
$\underline{17}$	$(r_d = \mathsf{load}([44, \mathsf{r_c}], \underline{18}))$

Directive	Effect on *buf*	Leakage
fetch	$\overline{1} \mapsto r_c = \mathsf{load}(48 + r_a)$	
fetch	$\overline{2} \mapsto$ fence	
execute $\overline{1}$	$\overline{1} \mapsto r_c = Key[1]_{\mathsf{sec}}$	read 49_{pub}
fetch: $\underline{17}$	$\overline{3} \mapsto \mathsf{jmpi}([12, r_b], \underline{17})$	
fetch	$\overline{4} \mapsto r_d = \mathsf{load}([44, r_c])$	
retire	$\overline{1} \notin buf$	
retire	$\overline{2} \notin buf$	
execute $\overline{4}$	$\overline{4} \mapsto r_d = X$ where $a = Key[1]_{\mathsf{sec}} + 40$	read a_{sec}

Figure 11. Example demonstrating Spectre v2 attack from a mistrained indirect branch predictor. Speculation barriers are not a useful defense against this style of attack.

In the execute stage, we calculate the actual jump target and compare it to the guess. If the actual target and the guess match, we update the entry in the reorder buffer to the resolved jump instruction jump n_0. If actual target and the guess do not match, we roll back the execution by removing all buffer entries larger or equal to i, update the buffer with the resolved jump to the correct address, and set the next instruction.

Like conditional branch instructions, indirect jumps leak the calculated jump target.

Examples. The example in Figure 11 shows how a mistrained indirect branch predictor can lead to disclosure vulnerabilities. After loading a secret value into r_c at program point $\underline{1}$, the program makes an indirect jump. An adversary can mistrain the predictor to send execution to $\underline{17}$ instead of the intended branch target, where the secret value in r_c is immediately leaked. Because indirect jumps can have arbitrary branch target locations, fence instructions do not prevent these kinds of attacks; an adversary can simply retarget the indirect jump to the instruction after the fence, as is seen in this example.

A.2 Return address prediction

Next, we discuss how our semantics models function calls.

Instructions. We introduce the following two physical instructions: call(n_f, n_{ret}), where n_f is the target program point of the call and n_{ret} is the return program point; and the return instruction ret. Their transient forms are simply call and ret.

Call stack. To track control flow in the presence of function calls, our semantics explicitly maintains a call stack in memory. For this, we use a dedicated register r_{sp} which points to

	Program	n	$\underline{1}$	$\underline{2}$	$\underline{3}$
		$\mu(n)$	call($\underline{3}, \underline{2}$)	ret	ret

Directive	n	buf	σ
fetch	$\underline{1} \to \underline{3}$	$\overline{1} \mapsto$ call	$\overline{1} \mapsto push\ \underline{2}$
		$\overline{2} \mapsto r_{sp} = \text{op}(succ, r_{sp})$	
		$\overline{3} \mapsto \text{store}(2, [r_{sp}])$	
fetch	$\underline{3} \to \underline{2}$	$\overline{4} \mapsto$ ret	$\overline{4} \mapsto pop$
		$\overline{5} \mapsto r_{tmp} = \text{load}([r_{sp}])$	
		$\overline{6} \mapsto r_{sp} = \text{op}(pred, r_{sp})$	
		$\overline{7} \mapsto \text{jmpi}([r_{tmp}], \underline{2})$	
fetch: $\underline{n}$	$\underline{2} \to \underline{n}$	$\overline{8} \mapsto$ ret	$\overline{8} \mapsto pop$
		$\overline{9} \mapsto r_{tmp} = \text{load}([r_{sp}])$	
		$\overline{10} \mapsto r_{sp} = \text{op}(pred, r_{sp})$	
		$\overline{11} \mapsto \text{jmpi}([r_{tmp}], \underline{n})$	

Figure 12. Example demonstrating a ret2spec-style attack [23]. The attacker is able to send (speculative) execution to an arbitrary program point, shown in red.

the top of the call stack, and which we call the *stack pointer register*.

On fetching a call instruction, we update r_{sp} to point to the address of the next element of the stack using an abstract operation *succ*. It then saves the return address to the newly computed address. On returning from a function call, our semantics transfers control to the return address at r_{sp}, and then updates r_{sp} to point to the address of the previous element using a function *pred*. This step makes use of a temporary register r_{tmp}.

Using abstract operations *succ* and *pred* rather than committing to a concrete implementation allows our semantics to capture different stack designs. For example, on a 32-bit x86 processor with a downward-growing stack, $\text{op}(succ, r_{sp})$ would be implemented as $r_{sp} - 4$, while $\text{op}(pred, r_{sp})$ would be implemented as $r_{sp} + 4$; on an upward growing system, the reverse would be true.

Note that the stack register r_{sp} is not protected from illegal access and can be updated freely.

Return stack buffer. For performance, modern processors speculatively predict return addresses. To model this, we extend configurations with a new piece of state called the *return stack buffer* (RSB), written as σ. The return stack buffer contains the expected return address at any execution point. Its implementation is simple: for a call instruction, the semantics pushes the return address to the RSB, while for a ret instruction, the semantics pops the address at the top of the RSB. Similar to the reorder buffer, we address the RSB through indices and roll it back on misspeculation or memory hazards.

We model return prediction directly through the return stack buffer rather than relying on attacker directives, as most processors follow this simple strategy, and the predictions therefore cannot be influenced by an attacker.

We now present the step rules for our semantics.

Calling.

CALL-DIRECT-FETCH

$$\frac{\begin{array}{c}\mu(n) = \text{call}(n_f, n_{ret}) \qquad i = \text{MAX}(buf) + 1 \\ buf_1 = buf[i \mapsto \text{call}][i+1 \mapsto (r_{sp} = \text{op}(succ, r_{sp}))] \\ buf' = buf_1[i+2 \mapsto \text{store}(n_{ret}, [r_{sp}])] \\ \sigma' = \sigma[i \mapsto push\ n_{ret}] \qquad n' = n_f\end{array}}{(\rho, \mu, n, buf, \sigma) \xhookrightarrow[\text{fetch}]{} (\rho, \mu, n', buf', \sigma')}$$

CALL-RETIRE

$$\frac{\begin{array}{c}\text{MIN}(buf) = i \\ buf(i) = \text{call} \qquad buf(i+1) = (r_{sp} = v_\ell) \\ buf(i+2) = \text{store}(n_{ret}, a_{\ell_a}) \qquad \rho' = \rho[r_{sp} \mapsto v_\ell] \\ \mu' = \mu[a \mapsto n_{ret}] \qquad buf' = buf[j : j > i+2]\end{array}}{(\rho, \mu, n, buf, \sigma) \xhookrightarrow[\text{retire}]{\text{write } a_{\ell_a}} (\rho', \mu', n, buf', \sigma)}$$

On fetching a call instruction, we add three transient instructions to the reorder buffer to model pushing the return address to the in-memory stack. The first transient instruction, call, simply serves as an indication that the following two instructions come from fetching a call instruction. The remaining two instructions advance r_{sp} to point to a new stack entry, then store the return address n_{ret} in the new entry. Neither of these transient instructions are fully resolved—they will need to be executed in later steps. We next add a new entry to the RSB, signifying a push of the return address n_{ret} to the RSB. Finally, we set our program point to the target of the call n_f.

When retiring a call, all three instructions generated during the fetch are retired together. The register file is updated with the new value of r_{sp}, and the return address is written to physical memory, producing the corresponding leakage.

The semantics for direct calls can be extended to cover indirect calls in a straightforward manner by imitating the semantics for indirect jumps. We omit them for brevity.

Evaluating the RSB. We define a function $top(\sigma)$ that retrieves the value at the top of the RSB stack. For this, we let $[\![\sigma]\!]$ be a function that transforms the RSB stack σ into a stack in the form of a partial map $(st : \mathcal{N} \rightharpoonup \mathcal{V})$ from the natural numbers to program points, as follows: the function $[\![\cdot]\!]$ applies the commands for each value in the domain of σ, in the order of the indices. For a *push n* it adds n to the lowest empty index of st. For *pop*, it and removes the value with the highest index in st, if it exists. We then define $top(\sigma)$ as $st(\text{MAX}(st))$, where $st = [\![\sigma]\!]$, and $\bot$, if the domain of st is empty. For example, if σ is given as $\emptyset[1 \mapsto\ push\ \underline{4}][2 \mapsto\ push\ \underline{5}][3 \mapsto pop]$, then $[\![\sigma]\!] = \emptyset[1 \mapsto \underline{4}]$, and $top(\sigma) = \underline{4}$.

Returning.

RET-FETCH-RSB

$$\frac{\mu(n) = \text{ret} \qquad top(\sigma) = n' \qquad i = \text{MAX}(buf) + 1 \qquad buf_1 = buf[i \mapsto \text{ret}] \qquad buf_2 = buf_1[i+1 \mapsto (r_{tmp} = \text{load}([r_{sp}]))] \qquad buf_3 = buf_2[i+2 \mapsto (r_{sp} = \text{op}(pred, r_{sp}))] \qquad buf_4 = buf_3[i+3 \mapsto \text{jmpi}([r_{tmp}], n')] \qquad \sigma' = \sigma[i \mapsto pop]}{(\rho, \mu, n, buf, \sigma) \xhookrightarrow[\text{fetch}]{} (\rho, \mu, n', buf_4, \sigma')}$$

RET-FETCH-RSB-EMPTY

$$\frac{\mu(n) = \text{ret} \qquad top(\sigma) = \bot \qquad i = \text{MAX}(buf) + 1 \qquad buf_1 = buf[i \mapsto \text{ret}] \qquad buf_2 = buf_1[i+1 \mapsto (r_{tmp} = \text{load}([r_{sp}]))] \qquad buf_3 = buf_2[i+2 \mapsto (r_{sp} = \text{op}(pred, r_{sp}))] \qquad buf_4 = buf_3[i+3 \mapsto \text{jmpi}([r_{tmp}], n')] \qquad \sigma' = \sigma[i \mapsto pop]}{(\rho, \mu, n, buf, \sigma) \xhookrightarrow[\text{fetch: } n']{} (\rho, \mu, n', buf_4, \sigma')}$$

RET-RETIRE

$$\frac{\text{MIN}(buf) = i \qquad buf(i) = \text{ret} \qquad buf(i+1) = (r_{tmp} = v_{1\ell_1}) \qquad buf(i+2) = (r_{sp} = v_{2\ell_2}) \qquad buf(i+3) = \text{jump } n' \qquad \rho' = \rho[r_{sp} \mapsto v_{2\ell_2}] \qquad buf' = buf[j : j > i+3]}{(\rho, \mu, n, buf, \sigma) \xhookrightarrow[\text{retire}]{} (\rho', \mu, n, buf', \sigma)}$$

On a fetch of ret, the next program point is set to the predicted return address, i.e., the top value of the RSB, $top(\sigma)$. Just as with call, we add the transient ret instruction to the reorder buffer, followed by the following (unresolved) instructions: we load the value at address r_{sp} into a temporary register r_{tmp}, we "pop" r_{sp} to point back to the previous stack entry, and then add an indirect jump to the program point given by r_{tmp}. Finally, we add a *pop* entry to the RSB. As with call instructions, the set of instructions generated by a ret fetch are retired all at once.

When the RSB is empty, the attacker can supply a speculative return address via the directive fetch: n'. This is consistent with the behavior of existing processors. In practice, there are several variants on what processors actually do when the RSB is empty [23]:

- AMD processors refuse to speculate. This can be modeled by defining $top(\sigma)$ to be a failing predicate if it would result in $\bot$.
- Intel Skylake/Broadwell processors fall back to using their branch target predictor. This can be modeled by allowing arbitrary n' for the fetch: n' directive for the RET-FETCH-RSB-EMPTY rule.
- "Most" Intel processors treat the RSB as a circular buffer, taking whichever value is produced when the RSB over- or underflows. This can be modeled by having $top(\sigma)$ always produce an according value, and never producing $\bot$.

Registers

r	$\rho(r)$
r_b	8_{pub}
r_{sp}	7C_{pub}

Program

n	$\mu(n)$
$\underline{3}$	call($\underline{5}$, $\underline{4}$)
$\underline{4}$	fence $\underline{4}$
$\underline{5}$	$r_d = \text{op}(addr, [12, r_b], \underline{6})$
$\underline{6}$	store(r_d, $[r_{sp}]$, $\underline{7}$)
$\underline{7}$	ret

Effect of successive fetch directives

n	buf	σ
$\underline{3} \to \underline{5}$	$\overline{3} \mapsto \text{call}$	$\overline{3} \mapsto push\ \underline{4}$
	$\overline{4} \mapsto r_{sp} = \text{op}(succ, r_{sp})$	
	$\overline{5} \mapsto \text{store}(4, [r_{sp}])$	
$\underline{5} \to \underline{6}$	$\overline{6} \mapsto r_d = \text{op}(addr, [12, r_b])$	
$\underline{6} \to \underline{7}$	$\overline{7} \mapsto \text{store}(r_d, [r_{sp}])$	
$\underline{7} \to \underline{4}$	$\overline{8} \mapsto \text{ret}$	$\overline{8} \mapsto pop$
	$\overline{9} \mapsto r_{tmp} = \text{load}([r_{sp}])$	
	$\overline{10} \mapsto r_{sp} = \text{op}(pred, r_{sp})$	
	$\overline{11} \mapsto \text{jmpi}([r_{tmp}], \underline{4})$	
$\underline{4} \to \underline{4}$	$\overline{12} \mapsto \text{fence}$	

Directive	Effect on buf	Leakage
execute $\overline{4}$	$\overline{4} \mapsto r_{sp} = 7\text{B}$	
execute $\overline{6}$	$\overline{6} \mapsto r_d = 20$	
execute $\overline{7}$: value	$\overline{7} \mapsto \text{store}(20, [r_{sp}])$	
execute $\overline{7}$: addr	$\overline{7} \mapsto \text{store}(20, 7\text{B})$	`fwd 7B`
execute $\overline{9}$	$\overline{9} \mapsto r_{tmp} = 20$	`fwd 7B`
execute $\overline{11}$	$\overline{12} \notin buf$	`rollback`,
	$\overline{11} \mapsto \text{jump } \underline{20}$	`jump` $\underline{20}$

Figure 13. Example demonstrating "retpoline" mitigation against Spectre v2 attack. The program is able to jump to program point $12 + r_b = \underline{20}$ without the schedule influencing prediction.

Examples. We present an example of an RSB underflow attack in Figure 12. After fetching a call and paired ret instruction, the RSB will be "empty". When one more (unmatched) ret instruction is fetched, since $top(\sigma) = \bot$, the program point n is no longer set by the RSB, and is instead set by the (attacker-controlled) schedule.

Retpoline mitigation. A mitigation for Spectre v2 attacks is to replace indirect jumps with *retpolines* [31]. Figure 13 shows a retpoline construction that would replace the indirect jump in Figure 11. The call sends execution to program point $\underline{5}$, while adding $\underline{4}$ to the RSB. The next two instructions at $\underline{5}$ and $\underline{6}$ calculate the same target as the indirect jump in Figure 11 and overwrite the return address in memory with this jump target. When executed speculatively, the ret at $\underline{7}$ will pop the top value off the RSB, $\underline{4}$, and jump there, landing on a fence instruction that loops back on itself. Thus speculative execution cannot proceed beyond this point. When the transient instructions in the ret sequence finally execute, the indirect jump target $\underline{20}$ is loaded from memory, causing a roll back. However, execution is then directed to the proper

jump target. Notably, at no point is an attacker able to hijack the jump target via misprediction.

B Full proofs

B.1 Consistency

Lemma B.1 (Determinism). *If* $C \xhookrightarrow[d]{o'} C'$ *and* $C \xhookrightarrow[d]{o''} C''$ *then* $C' = C''$ *and* $o' = o''$.

Proof. The tuple (C, d) fully determines which rule of the semantics can be executed. □

Definition B.2 (Initial/terminal configuration). A configuration C is an *initial* (or *terminal*) configuration if $|C.buf| = 0$.

Definition B.3 (Sequential schedule). Given a configuration C, we say a schedule D is *sequential* if every instruction that is fetched is executed and retired before further instructions are fetched.

Definition B.4 (Sequential execution). $C \; {}_O\!\Downarrow^N_D \; C'$ is a sequential execution if C is an initial configuration, D is a sequential schedule for C, and C' is a terminal configuration.

We write $C \; {}_O\!\Downarrow^N_{seq} \; C'$ if we execute sequentially.

Lemma B.5 (Sequential equivalence). *If* $C \; {}_{O_1}\!\Downarrow^N_{D_1} \; C_1$ *is sequential and* $C \; {}_{O_2}\!\Downarrow^N_{D_2} \; C_2$ *is sequential, then* $C_1 = C_2$.

Proof. Suppose $N = 0$. Then neither D_1 nor D_2 may contain any retire directives. Since we assume that both $C_1.buf$ and $C_2.buf$ have size 0, neither D_1 nor D_2 may contain any fetch directives. Therefore, both D_1 and D_2 are empty; both C_1 and C_2 are equal to C.

We proceed by induction on N.

Let D_1' be a sequential prefix of D_1 up to the $N-1$th retire, and let D_1'' be the remainder of D_1. That is, $\#\{d \in D_1' \mid d = \text{retire}\} = N-1$ and $D_1' \| D_1'' = D_1$. Let D_2' and D_2'' be similarly defined.

By our induction hypothesis, we know $C \; {}_{O_1'}\!\Downarrow^{N-1}_{D_1'} \; C'$ and $C \; {}_{O_2'}\!\Downarrow^{N-1}_{D_2'} \; C'$ for some C'. Since D_1' (resp. D_2') is sequential and $|C'.buf| = 0$, the first directive in D_1'' (resp. D_2'') must be a fetch directive. Furthermore, $C' \; {}_{O_1''}\!\Downarrow^1_{D_1''} \; C_1$ and $C' \; {}_{O_2''}\!\Downarrow^1_{D_2''} \; C_2$.

We can now proceed by cases on $C'.\mu[C'.n]$, the final instruction to be fetched.

- For op, the only valid sequence of directives is (fetch, execute i, retire) where i is the sole valid index in the buffer. Similarly for fence, with the sequence {fetch, retire}.
- For load, alias prediction is not possible, as no prior stores exist in the buffer. Therefore, just as with op, the only valid sequence of directives is (fetch, execute i, retire).
- For store, the only possible difference between D_1'' and D_2'' is the ordering of the execute i : value and execute i : addr directives. However, both orderings will result in the same configuration since they independently resolve the components of the store.
- For br, D_1'' and D_2'' may have different guesses for their initial fetch directives. However, both COND-EXECUTE-CORRECT and COND-EXECUTE-INCORRECT will result in the same configuration regardless of the initial guess, as the br is the only instruction in the buffer. Similarly for jmpi.
- For call and ret, the ordering of execution of the resulting transient instructions does not affect the final configuration.

Thus for all cases we have $C_1 = C_2$. □

To make our discussion easier, we will say that a directive d *applies to* a buffer index i if when executing a step $C \xhookrightarrow[d]{o} C'$:

- d is a fetch directive, and would fetch an instruction into index i in buf.
- d is an execute directive, and would execute the instruction at index i in buf.
- d is a retire directive, and would retire the instruction at index i in buf.

We would like to reason about schedules that do not contain *misspeculated steps*, i.e., directives that are superfluous due to their effects getting wiped away by rollbacks.

Definition B.6 (Misspeculated steps). Given an execution $C \; {}_O\!\Downarrow^N_D \; C'$, we say that D contains *misspeculated steps* if there exists $d \in D$ such that $D' = D \setminus d$ and $C \; {}_{O'}\!\Downarrow^N_{D'} \; C'' = C'$.

Given an execution $C \; {}_O\!\Downarrow^N_D \; C'$ that may contain rollbacks, we can create an alternate schedule D^* without any rollbacks by removing all misspeculated steps. Note that sequential schedules have no misspeculated steps[6] as defined in Definition B.6.

Theorem B.7 (Equivalence to sequential execution). *Let* C *be an initial configuration and* D *a well-formed schedule for* C. *If* $C \; {}_{O_1}\!\Downarrow^N_D \; C_1$, *then* $C \; {}_{O_2}\!\Downarrow^N_{seq} \; C_2$ *and* $C_1 \approx C_2$. *Furthermore, if* C_1 *is terminal then* $C_1 = C_2$.

Proof. Since we can always remove all misspeculated steps from any well-formed execution without affecting the final configuration, we assume D_1 has no misspeculated steps.

Suppose $N = 0$. Then the theorem is trivially true. We proceed by induction on N.

Let D_1' be the subsequence of D_1 containing the first $N-1$ retire directives and the directives that apply to the same indices of the first $N-1$ retire directives. Let D_1'' be the complement of D_1' with respect to D_1. All directives in D_1'' apply to indices later than any directive in D_1', and thus cannot affect the execution of directives in D_1'. Thus D_1' is a well-formed schedule and produces execution $C \; {}_{O_1'}\!\Downarrow^{N-1}_{D_1'} \; C_1'$.

Since D_1 contains no misspeculated steps, the directives in D_1'' can be reordered after the directives in D_1'. Thus

[6] Sequential schedules may still misspeculate on conditional branches but the rollback does not imply removal of any reorder buffer instructions as defined in Definition B.6.

D_1'' is a well-formed schedule for C_1', producing execution $C_1' \; {}_{O_1''}\!\Downarrow^1_{D_1''} C_1''$ with $C_1'' \approx C_1$. If C_1 is terminal, then C_1'' is also terminal and $C_1'' = C_1$.

By our induction hypothesis, we know there exists D'_{seq} such that $C \; {}_{O_2'}\!\Downarrow^{N-1}_{D'_{seq}} C_2'$. Since D_1' contains equal numbers of fetch and retire directives, ends with a retire, and contains no misspeculated steps, C_1' is terminal. Thus $C_1' = C_2'$.

Let D''_{seq} be the subsequence of D_1'' containing the retire directive in D_1'' and the directives that apply to the same index. D''_{seq} is sequential with respect to C_1' and produces execution $C_1' \; {}_{O_2''}\!\Downarrow^1_{D''_{seq}} C_2''$ with $C_2'' \approx C_1'' \approx C_1$. If C_1'' is terminal, then $D''_{seq} = D_1''$ and thus $C_2'' = C_1'' = C_1$.

Let $D_{seq} = D'_{seq} \| D''_{seq}$. D_{seq} is thus itself sequential and produces execution $C \; {}_{(O_2' \| O_2'')}\!\Downarrow^N_{seq} C_2''$, completing our proof. □

Corollary B.8 (General consistency)**.** *Let C be an initial configuration. If $C \; {}_{O_1}\!\Downarrow^N_{D_1} C_1$ and $C \; {}_{O_2}\!\Downarrow^N_{D_2} C_2$, then $C_1 \approx C_2$. Furthermore, if C_1 and C_2 are both terminal then $C_1 = C_2$.*

Proof. By Theorem B.7, there exists D'_{seq} such that executing with C produces $C_1' \approx C_1$ (resp. $C_1' = C_1$). Similarly, there exists D''_{seq} that produces $C_2' \approx C_2$ (resp. $C_2' = C_2$). By Lemma B.5, we have $C_1' = C_2'$. Thus $C_1 \approx C_2$ (resp. $C_1 = C_2$). □

B.2 Security

Theorem B.9 (Label stability)**.** *Let ℓ be a label in the lattice $\mathcal{L}$. If $C \; {}_{O_1}\!\Downarrow^N_{D_1} C_1$ and $\forall o \in O_1 : \ell \notin o$, then $C \; {}_{O_2}\!\Downarrow^N_{seq} C_2$ and $\forall o \in O_2 : \ell \notin o$.*

Proof. Let D_1^* be the schedule given by removing all misspeculated steps from D_1. The corresponding trace O_1^* is a subsequence of O_1, and hence $\forall o \in O_1^* : \ell \notin o$. We thus proceed assuming that execution of D_1 contains no misspeculated steps.

Our proof closely follows that of Theorem B.7. When constructing D_1' and D_1'' from D_1 in the inductive step, we know that all directives in D_1'' apply to indices later than any directive in D_1', and cannot affect execution of any directive in D_1'. This implies that O_1' is the subsequence of O_1 that corresponds to the mapping of D_1' to D_1.

Reordering the directives in D_1'' after D_1' do not affect the observations produced by most directives. The exceptions to this are execute directives for load instructions that would have received a forwarded value: after reordering, the store instruction they forwarded from may have been retired, and they must fetch their value from memory. However, even in this case, the address a_{ℓ_a} attached to the observation does not change. Thus $\forall o \in O_2'' : \ell \notin o$.

Continuing the proof as in Theorem B.7, we create schedule D'_{seq} (with trace O_2') from the induction hypothesis and D''_{seq} (with trace O_2'') as the subsequence of D_1'' of directives applying to the remaining instruction to be retired. As noted before, executing the subsequence of a schedule produces the corresponding subsequence of the original trace; hence $\forall o \in O_2'' : \ell \notin o$.

The trace of the final (sequential) schedule $D_{seq} = D'_{seq} \| D''_{seq}$ is $O_2' \| O_2''$. Since O_2' satisfies the label stability property via the induction hypothesis, we have $\forall o \in O_2' \| O_2'' : \ell \notin o$. □

By letting ℓ be the label secret, we get the following corollary:

Corollary B.10 (Secrecy)**.** *If speculative execution of C under schedule D produces a trace O that contains no secret labels, then sequential execution of C will never produce a trace that contains any secret labels.*

With this, we can prove the following proposition:

Proposition B.11. *For a given initial configuration C and well-formed schedule D, if C is SCT with respect to D, and execution of C with D results in a terminal configuration C_1, then C is also sequentially constant-time.*

Proof. Since C is SCT, we know that for all $C' \simeq_{\text{pub}} C$, we have $C \; {}_{O}\!\Downarrow^N_D C_1$ and $C' \; {}_{O'}\!\Downarrow^N_D C_1'$ where $C_1 \simeq_{\text{pub}} C_1'$ and $O = O'$. By Theorem B.7, we know there exist sequential executions such that $C \; {}_{O_{seq}}\!\Downarrow^N_{seq} C_2$ and $C' \; {}_{O'_{seq}}\!\Downarrow^N_{seq} C_2'$. Note that the two sequential schedules need not be the same.

C_1 is terminal by hypothesis. Execution of C' uses the same schedule D, so C_1' is also terminal. Since we have $C_1 = C_2$ and $C_1' = C_2'$, we can lift $C_1 \simeq_{\text{pub}} C_1'$ to get $C_2 \simeq_{\text{pub}} C_2'$.

To prove the trace property $O_{seq} = O'_{seq}$, we note that if $O_{seq} \neq O'_{seq}$, then since $C_2 \simeq_{\text{pub}} C_2'$, it must be the case that there exists some $o \in O_{seq}$ such that secret $\in O_{seq}$. Since this is also true for O and O', we know that there exist no observations in either O or O' that contain secret labels. By Corollary B.10, it follows that no secret labels appear in either O_{seq} or O'_{seq}, and thus $O_{seq} = O'_{seq}$. □

B.3 Soundness of Pitchfork

Definition B.12 (Affecting an index)**.** We say a directive d *affects an index i* if:

- d is a fetch-type directive and would produce a new mapping in *buf* at index i.
- d is an execute-type directive and specifies index i directly (e.g., execute i).
- d is a retire directive and would cause the instruction at i in *buf* to be removed.

Definition B.13 (Path function)**.** The function $Path(C, D)$ produces the sequence of branch choice (from fetching br instructions) and store-forwarding information (when executing load instructions) when executing D with initial configuration C. That is, for a schedule D without misspeculated

steps:

$$Path(C, \emptyset) = []$$

$$Path(C, D\|d) = \begin{cases} Path(C, D); (i, b), & d = \text{fetch: } b \\ Path(C, D); (i, j), & d \text{ produces } v_\ell\{j, a\} \\ Path(C, D); (i, \bot), & d \text{ produces } v_\ell\{\bot, a\} \\ Path(C, D), & \text{otherwise} \end{cases}$$

where d affects index i. If D has misspeculated steps, then $Path(C, D) = Path(C, D^*)$ where D^* is the subset of D with misspeculated steps removed. We write simply $Path(D)$ when C is obvious.

For the Lemmas B.14, B.16 and B.17, we start with the following shared assumptions:

- C is an initial configuration.
- D_1 and D_2 are nonempty schedules.
- $C \; {}_{D_1}\!\Downarrow_{O_1} C_1$ and $C \; {}_{D_2}\!\Downarrow_{O_2} C_2$.
- $Path(C, D_1) = Path(C, D_2)$.
- $D_1 = D_1'\|d_1$ and $D_2 = D_2'\|d_2$ and $d_1 = d_2$.
- d_1 and d_2 affect the same index i in the their respective reorder buffers.

Let o_1 (resp. o_2) be the observation produced during execution of d_1 (resp. d_2).

Lemma B.14 (Fetch)**.** *If d_1 and d_2 are both fetch-type directives, then $C_1.n = C_2.n$ and $C_1.buf[i] = C_2.buf[i]$.*

Proof. Since fetches happen in-order, the index i of a given physical instruction along a control flow path is deterministic. Both D_1 and D_2 both have the same (control flow) path. Since by hypothesis both d_1 and d_2 affect the same index i, d_1 and d_2 must necessarily both be fetching the same physical instruction. Furthermore, since $Path(D_1) = Path(D_2)$, if the fetched instruction is a br instruction, then both d_1 and d_2 must have made the same guess. The lemma statements all hold accordingly. □

Corollary B.15. *If D_1^* and D_2^* are nonempty schedules such that $C \; {}_{D_1^*}\!\Downarrow C_1^*$ and $C \; {}_{D_2^*}\!\Downarrow C_2^*$ and $Path(C, D_1^*) = Path(C, D_2^*)$, then: For any $i \in C_1^*.buf$, if $i \in C_2^*.buf$, then both $C_1^*.buf[i]$ and $C_2^*.buf[i]$ were derived from the same physical instruction.*

Proof. Let D_1 be the prefix of D_1^* such that the final directive in D_1 is the latest fetch that affects i. Let D_2 be similarly defined w.r.t. D_2^*. Then by Lemma B.14, D_1 and D_2 both fetch the same physical instruction to index i. □

Lemma B.16. *If d_1 and d_2 are both execute-type directives, then $C_1.buf[i] = C_2.buf[i]$ and $o_1 = o_2$.*

Proof. We proceed by full induction on the size of D_1.

For the base case: if $|D_1| = 1$, then the lemma statements are trivial regardless of the directive d_1.

We know from Corollary B.15 that since d_1 and d_2 both affect the same index i, the two transient instruction must be derived from the same physical instruction, and thus has the same register dependencies. For each register dependency r, if the register was calculated by a transient instruction at a prior index j, we can create prefixes $D_{1,j}$ and $D_{2,j}$ of D_1 and D_2 respectively that end at the execute directive that resolves r at buffer index j. By our induction hypothesis, both $D_{1,j}$ and $D_{2,j}$ calculate the same value v_ℓ for r.

We now proceed by cases on the transient instruction being executed.

Op, Store (value). Since all dependencies calculate the same values, both instructions calculate the same value.

Store (address). Both instructions calculate the same address. Since $Path(D_1) = Path(D_2)$, both schedules have the same pattern of store-forwarding behavior. Thus execution of d_1 causes a hazard if and only if d_2 causes a hazard.

Load. Both instructions calculate the same address, producing the same observations o_1 and o_2. Since $Path(D_1) = Path(D_2)$, either d_1 and d_2 cause the values to be retrieved from the same prior stores, or they both load values from the same address in memory. By our induction hypothesis, these values will be the same, so both instructions will resolve to the same value.

Branch. Both instructions calculate the same branch condition, producing the same observations o_1 and o_2. Since $Path(D_1) = Path(D_2)$, execution of d_1 causes a misspeculation hazard if and only if d_2 also causes misspeculation hazard. □

Lemma B.17. *If d_1 and d_2 are both retire directives, then $o_1 = o_2$.*

Proof. From Lemmas B.14 and B.16 we know that for both d_1 and d_2, the transient instructions to be retired are the same. Thus the produced observations o_1 and o_2 are also the same. □

We now formally define the set of schedules examined by Pitchfork:

Definition B.18 (Tool schedules)**.** Given an initial configuration C and a speculative window size n, we define the set of *tool schedules* $D_T(n)$ recursively as follows: The empty schedule $\emptyset$ is in $D_T(n)$. If $D_0 \in D_T(n)$ and $C \; {}_{D_0}\!\Downarrow C_0$ and $|C_0.buf| < n$, then based on the next instruction to be fetched (and where i is the index of the fetched instruction):

- op: $D_0\|$fetch; execute $i \in D_T(n)$.
- load: $D_0\|$fetch; execute $i \in D_T(n)$.
- store: $D_0\|$fetch; execute i : value $\in D_T(n)$ and $D_0\|$fetch; execute i : value; execute i : addr $\in D_T(n)$.
- br: Let b be the "correct" path for the branch condition. Then $D_0\|$fetch: b; execute $i \in D_T(n)$ and $D_0\|$fetch: $\neg b \in D_T(n)$.

Otherwise, if $|C_0.buf| = n$, then we instead extend based on the oldest instruction in the reorder buffer. If the oldest instruction is a store with an unresolved address, and will not cause a hazard, then $D_0\|$execute i : addr; retire $\in D_T(n)$.

Otherwise, if the oldest instruction is fully resolved, then $D_0 \| \mathsf{retire} \in D_T(n)$.

Proposition B.19 (Path coverage)**.** *If D_1 is a well-formed schedule for C whose reorder buffer never grows beyond size n, then $\exists D_2 : Path(D_1) = Path(D_2) \wedge D_2 \in D_T(n)$.*

Proof. The proof stems directly from the definition of $D_T(n)$; at every branch, both branches are added to the set of schedules, and every load is able to "skip" any combination of prior stores. □

Theorem B.20 (Soundness of tool)**.** *If speculative execution of C under a schedule D with speculation bound n produces a trace O that contains at least one secret label, then there exists a schedule $D_t \in D_T(n)$ that produces a trace O_t that also contains at least one secret label.*

Proof. We can truncate D to a schedule D^* that ends at the first directive to produce a secret observation. By Proposition B.19 there exists a schedule $D_0 \in D_T(n)$ such that $Path(D_t) = Path(D^*)$. By following construction of tool schedules as given in Definition B.18, we can find a schedule $D_t \in D_T(n)$ that satisfies the preconditions for Lemma B.16. Then by that same lemma, D_t produces the same final observation as D^*, which contains a secret label. □