Towards Constant-Time Foundations for the New Spectre Era

Sunjay Cauligi[†] Craig Disselkoen[†] Klaus v. Gleissenthall[†]
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 micro-architectural 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. Our first contribution is an operational semantics, and a formal definition of constant-time programs in this extended setting. Our semantics eschews formalization of micro-architectural 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 by contrasting existing Spectre-like attacks with our definition of constant-time and by exhibiting a new (theoretical) class of Spectre attacks based on alias prediction. Our second contribution is a static analysis tool, Pitchfork, which detects violations of our extended constant-time property (for a subset of the semantics presented in this paper).

KEYWORDS

Spectre; speculative execution; semantics; static analysis

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 [13].

The most robust way to deal with timing side-channels in software is via *constant-time* programming—the paradigm used to implement almost all modern cryptography [2, 3, 10, 26, 27]. Programs that are constant-time can neither branch on secrets nor access memory based on secret data.[1] Together these restrictions ensure that programs cannot leak secret information via timing side-channels by construction [5]—at least on hardware *without* out-of-order execution and speculative execution. Modern processors, however, *do* have such complicated microarchitectural features. These features can indeed be used as timing side channels, as the still-growing number of Spectre-like attacks is showing. Unfortunately, since existing constant-time models do not capture these details, programs deemed to be constant-time at the ISA level, may in fact leak information on processors that execute instructions out-of-order or speculatively. The decade-old constant-time recipes are no longer enough. OpenSSL found this situation so hopeless that they recently updated their security model to explicitly exclude "physical system side channels" [25].

In this paper, we lay the foundations for constant-time in the presence of speculative and out-of-order execution. We focus on constant-time for two key reasons. First, *impact*: constant-time programming is largely used in narrow, high-assurance code—mostly cryptographic implementations—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. These semantics consider very powerful attackers—e.g., attackers in [5] have control over the cache and the scheduler. A nice effect of considering powerful attackers is that the semantics can already 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. Adapting these semantics to account for attacker-controlled microarchitectural features makes it possible for us to both reason about code running on modern hardware, and uncover how code can leak secrets via microarchitectural side channels in a principled way.

To this end, we define a semantics for an abstract, three-stage—fetch, execute, and retire—machine. Our machine retires instructions in-order, but supports out-of-order and speculative execution by modeling *reorder buffers* and *transient instructions*, respectively. We do not, however, model the cache or most other microarchitectural features (e.g., the branch target predictor). We instead leave them abstract in our model and give the attacker complete control over them via execution *directives*. This keeps our semantics simple yet powerful: it allows us to abstract over all possible implementations (e.g., all possible cache-eviction or branch prediction policies) when reasoning about security.

We define a notion of constant time for this abstract, speculative machine in terms of a new property, *speculative observational non-interference*—an extension of the classical *observational non-interference* that accounts for leaks during speculative and out-of-order 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 violation of our constant-time property. Our semantics even revealed a new (theoretical) Spectre variant that exploits the aliasing predictor.

To detect violations of the constant time property in real code, we built a prototype, static analysis tool, Pitchfork, which captures potential leakages in binaries (for a subset of our semantics). We

[1] More generally, constant-time programs cannot use secret data as input to any variable-time operation, including instructions like floating-point multiplication.

verify that Pitchfork detects leakages in the well-known Kocher test cases [18] for Spectre v1, as well our more extensive test suite which includes Spectre v1.1 variants. We will make both the tool and test suites available under an open source license.

Our contributions are:

- An operational semantics for speculative and out-of-order execution which abstracts over microarchitectural features that can be controlled by attackers.
- A formal definition of constant-time under speculative and out-of-order semantics.
- A new theoretical Spectre variant, Spectre-MOB, discovered using our formal semantics.
- A prototype static analysis tool, Pitchfork, that implements our semantics to detect speculative constant-time violations. A revised set of Spectre v1 tests based on the Kocher test cases, and a new set of test cases for Spectre v1.1, accompanying the tool.

2 MICROARCHITECTURE: AN OVERVIEW

Modern processors are considerably more sophisticated than the simple sequential execution models assumed by most work on constant-time. We briefly review some of the underlying, microarchitectural features in this section.

Out-of-order execution. Modern processors support *out-of-order* execution by placing instructions in a *reorder buffer* which maintains a queue of all *in-flight* instructions—instructions that have begun but not yet finished execution. This allows the instructions to be executed in any order, as their operands become available.

Speculation. To continue execution before memory dependencies are fully resolved, processors *speculate* on many conditions, including the results of loads. When wrong, the microarchitectural state is *rolled back* to the point where the incorrect guess was made.

The most fundamental form of speculation is *branch prediction*. When fetching a conditional branch, the processor guesses whether the branch will be taken or not. This allows it to continue fetching and executing subsequent instructions, before it knows the actual branch direction. If the predictor was wrong, microarchitectural state is rolled back, and execution resumes on the correct path.

Processors also speculate on the targets of indirect jumps using a structure called the *branch target buffer (BTB)* [19]; on the targets of returns using the *return stack buffer (RSB)* [20, 22]; and on whether certain memory operations (loads and stores) will alias or not, i.e., whether they access the same memory address [17]. The latter, aliasing prediction is often used to perform load operations before fully resolving the addresses of previous stores.

Consistency. Processors work hard to create consistency: the illusion that in spite of their complexity, assembly instructions are executed sequentially. For instance, dedicated circuits track dependencies between instructions, ensuring that each instruction only executes when all of its operands' values are available. Similarly, dedicated hardware ensures that *hazards* are avoided—e.g., instructions do not incorrectly read stale values from registers or memory.

Speculative attacks. Recent Spectre attacks [17, 19, 21, 23] have shown that the intended abstraction of sequential execution is not perfectly maintained: a carefully crafted program can observe the effects of the underlying microarchitectural mechanisms. The original Spectre attack [19] exploits branch prediction to leak secrets. Subsequent attacks show that many other predictors can similarly be used to leak secrets and violate the sequential execution abstraction—e.g., Spectre v2 [12] exploits the BTB, ret2spec [20] exploits the RSB, while Spectre v4 [15] exploits the aliasing predictor.

3 LANGUAGE, SEMANTICS, AND SECURITY

We formalize speculative execution with a simplified assembly language ASM. ASM abstracts over microarchitectural details that are unnecessary for our purpose of modeling side-channel attacks.

Values and labels. We assume that values of ASM belong to a set $\mathcal{V}$. By convention, we use n to denote values that are used as memory addresses which point to code (*program points*), a to denote values used as memory addresses which point to data, and v for any kind of values. For modeling security violations, it is necessary to distinguish which values in an execution are secrets. Thus, we assume each value is annotated with a label from a lattice $\mathcal{L}$ of security labels, and use $\sqcup$ for its join operator. Let ℓ range over $\mathcal{L}$. We use v_ℓ to denote the value v annotated with label ℓ and interpret omitted labels as the least label in the lattice. We overload $\mathcal{V}$ to refer to the set of labeled values. Our semantics is parametric in $\mathcal{L}$; in our examples and security analysis, we use the simple lattice public $\sqsubset$ secret to distinguish public and secret data.

Instructions. We consider two sets of instructions for ASM: a set of *physical* instructions Instr and set of *transient* instructions TransInstr. Physical instructions exist in memory and are the components of the program to be run, while transient instructions only exist in speculative state. We will introduce the physical and transient instructions in ASM in the later sections.

Reorder buffer. Rather than being executed instantaneously, transient instructions are arranged in a *reorder buffer*, which 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. To define our semantics, we will need to modify buf by adding and removing entries. 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 write $buf \setminus buf(i)$ to denote 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 $\mathsf{MIN}(M)$ (resp. $\mathsf{MAX}(M)$) denote the minimum (maximum) index in the domain of a mapping M. We denote an empty mapping as $\emptyset$ and define $\mathsf{MIN}(\emptyset) = \mathsf{MAX}(\emptyset) = 0$.

For a formula φ, we sometimes want to talk about the highest (resp. lowest) index the formula holds on. We write $max(j) < i : \varphi(j)$ to mean that j is the highest index for which φ holds, i.e., $\varphi(j) \wedge \forall j' : j < j' < i : \neg\varphi(j')$. Analogously, we define $min(j) > i : \varphi(j)$.

Configurations. A configuration $C \in \text{Confs}$ for ASM 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 labeled 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 a reorder buffer.

We treat physical instructions as values in memory, and retrieve instructions from the memory by addressing it with program points. Thus, we assume that for any program point n, $\mu(n)$ maps to a physical instruction.

Formal small-step semantics. Our semantics is modeled after the stages of a processor pipeline. For each program instruction, we distinguish three execution stages: fetch, execute, and retire. Splitting up execution into stages allows us to model out-of-order execution. We model out-of-order execution (Section 3.1), speculative execution (Section 3.2), memory operations (Section 3.3), aliasing prediction (Section 3.4), indirect jumps (Section 3.5), calls and returns (Section 3.6), and finally fence instructions (Section 3.7).

Observations and directives. Although our goal is to reason about microarchitectural side-channels, our semantics does not directly model cache, nor any of the predictors used by speculative semantics. Rather, our semantics models externally visible effects—memory accesses and control flow—by producing a sequence of *observations*. Modeling the cache through these observations rather than a concrete eviction policy allows us to reason about *any* possible cache implementation, as any such policy can be expressed as a function of the sequence of observations.

Furthermore, exposing control flow observations directly in our semantics makes it unnecessary to track a variety of other side channels. For example, while channels such as port contention or register renaming [19] produce distinct measurable effects, they serve only to leak the path taken through the code; as such these observations are redundant.

Similarly, instead of modeling a particular branch prediction strategy, we let a hypothetical attacker resolve scheduling nondeterminism arbitrarily by supplying a series of *directives*. These directives include fetch, execute i, and retire (among others), allowing the attacker to choose when instructions are fetched and retired, and what order instructions are executed in. Not all directives are allowed at all times—for instance, retire is not a valid directive if the reorder buffer is empty.

Step relation. Our semantics is given through a step relation $\hookrightarrow$, where a step of the form $C \overset{o}{\underset{d}{\hookrightarrow}} C'$ denotes that given directive d, an execution step starting from configuration C leads to configuration C' and produces an observation o. If a directive d is not compatible with the current configuration, we write this as $C \overset{o}{\underset{d}{\hookrightarrow}} \bot$.

3.1 Basic out-of-order execution

Instructions. The op instruction has the form $(r = \text{op}(op, \overrightarrow{rv}, n'))$, where op indicates which arithmetic operation to perform, $\overrightarrow{rv}$ is the list of operands (each operand is either a register or a labeled value), r is the target register, and n' is the program point of the next instruction. The transient form of op can be either:

$$\begin{array}{ll} (r = \text{op}(op, \overrightarrow{rv})) & \textit{(unresolved op)} \\ (r = v_\ell) & \textit{(resolved value)} \end{array}$$

where v_ℓ is a (labeled) value. The resolved form of op represents an operation that has already been executed, but is yet to be retired.

Register resolve function. In Figure 1, we define a function $(.+_{.}.)()$, called the *register resolve function*, that allows us to determine the value of a register in the presence of pending operations in the reorder buffer. It determines the correct value of a given register r at "time" i, i.e., considering only operations that should be visible to the instruction at index i in the reorder buffer buf. 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 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.

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

Figure 1: Definition of the register resolve function.

Inference rules for op.

SIMPLE-FETCH

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

OP-EXECUTE

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

VALUE-RETIRE

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

Fetch. Given a fetch directive, rule SIMPLE-FETCH extends the reorder buffer buf with a new transient instruction. 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'. The SIMPLE-FETCH rule applies to load, store, and fence instructions (described in later sections) as well.

Execute. Given an execute i directive, rule OP-EXECUTE executes the operation at buffer index i. For this, it first computes the values of all argument registers using the register resolve function. This yields a list of operand values $\overrightarrow{v_\ell}$ used to evaluate operation op via an evaluation function $[\![\cdot]\!]$. This function $[\![\cdot]\!]$ is supplied as a parameter to our semantics, i.e., our semantics is not tied to any particular implementation. Finally, rule OP-EXECUTE updates the location i in buf with the resolved instruction $(r = v_\ell)$, where v_ℓ is the result of the evaluation function. The rule contains a condition on fences shown in gray, but this condition can be ignored for now; we will explain its significance in Section 3.7, where we introduce the semantics for memory barrier instructions.

(a) No dependence

Program		Reorder buffer *buf* after D	
$\underline{1}$	$(r_a = \text{op}(+, (1, r_a), \underline{2}))$	$\overline{1}$	$(r_a = \text{op}(+, (1, r_a)))$
$\underline{2}$	$(r_b = \text{op}(+, (4, r_b), \underline{3}))$	$\overline{2}$	$(r_b = 4)$

Directives: fetch; fetch; execute $\overline{2}$;

(b) With dependence

Program		Reorder buffer *buf* after D	
$\underline{1}$	$(r_a = \text{op}(+, (1, r_a), \underline{2}))$	$\overline{1}$	$(r_a = \text{op}(+, (1, r_a)))$
$\underline{2}$	$(r_b = \text{op}(+, (4, r_a), \underline{3}))$	$\overline{2}$	$(r_b = \text{op}(+, (4, r_a)))$

Directives: fetch; fetch; ~~execute $\overline{2}$;~~

Figure 2: Out-of-order execution with and without a data dependence. In part (b), the directive execute $\overline{2}$ is invalid.

Retire. Given a retire directive, and under the condition that the oldest instruction in *buf* (i.e., the instruction with the least index) is fully resolved, rule VALUE-RETIRE updates the register map ρ by assigning the resolved value to its target. Finally, it removes the instruction from *buf*.

Examples. Consider the program, buffer and directives shown in Figure 2a. For clarity, we write program points in underline and reorder buffer indices with overline, to separate them from regular values. We assume that all registers are initialized to 0. In the example, the instruction at program point $\underline{2}$ can be executed before instruction at program point $\underline{1}$, since instruction $\underline{2}$ does not depend on register r_a—the target register of the instruction at $\underline{1}$. In particular, since r_b is not assigned in any previous instruction in the buffer, the invocation of the register resolve function in rule OP-EXECUTE takes the second branch, i.e., reads the value 0 for r_b from the register map.

Next, consider the example given in Figure 2b. Because instruction at program counter $\underline{2}$ depends on register r_a which is assigned in instruction $\underline{1}$, we cannot execute instruction $\underline{2}$ before $\underline{1}$ has been fully resolved. Attempting the directive execute $\overline{2}$ will result in stuck execution ($\bot$), as the register resolve function in rule OP-EXECUTE takes the third branch, i.e., is undefined.

3.2 Speculative execution

Next, we discuss our semantics for *conditional branches*, and introduce speculative execution.

Instructions. The physical instruction for conditional branches has the following form $\text{br}(op, \overrightarrow{rv}, n^{\text{true}}, n^{\text{false}})$, 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 below. The unresolved form extends the physical instruction by a program point n_0, which is used to record the branch that is executed speculatively, and may or may not correspond to the branch that is taken, once op is resolved. The resolved form contains the final jump target.

$\text{br}(op, \overrightarrow{rv}, n_0, (n^{\text{true}}, n^{\text{false}}))$	*(unresolved conditional)*
$\text{jump } n_0$	*(resolved conditional)*

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

COND-FETCH

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

The COND-FETCH rule speculatively executes the branch determined by the Boolean value $b \in \{\text{true}, \text{false}\}$ specified by the directive. 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} or n^{false} in the transient jump instruction.

This semantics models the behavior of most modern processors. Since the value of the next program point n' may not be resolved in the fetch stage, speculation allows execution to continue rather than stalling the pipeline until the branch target is resolved. In hardware, the choice of which branch to execute is made by a branch predictor; our semantics instead allows the schedule to choose which of the rules to execute through the directives fetch: true and fetch: false. This allows us to abstract over any possible predictor implementation.

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

COND-EXECUTE-CORRECT

$$\frac{buf(i) = \text{br}(op, \overrightarrow{rv}, n_0, (n^{\text{true}}, n^{\text{false}})) \qquad \forall j < i : buf(j) \neq \text{fence} \qquad (buf +_i \rho)(\overrightarrow{rv}) = \overrightarrow{v_\ell} \qquad [\![op(\overrightarrow{v_\ell})]\!] = b_\ell \qquad n^b = n_0 \qquad buf' = buf[i \mapsto \text{jump } n^b]}{(\rho, \mu, n, buf) \xrightarrow[\text{execute } i]{\text{jump } n^b_\ell} (\rho, \mu, n, buf')}$$

COND-EXECUTE-INCORRECT

$$\frac{buf(i) = \text{br}(op, \overrightarrow{rv}, n_0, (n^{\text{true}}, n^{\text{false}})) \qquad \forall j < i : buf(j) \neq \text{fence} \qquad (buf +_i \rho)(\overrightarrow{rv}) = \overrightarrow{v_\ell} \qquad [\![op(\overrightarrow{v_\ell})]\!] = b_\ell \qquad n^b \neq n_0 \qquad buf' = buf[j : j < i][i \mapsto \text{jump } n^b]}{(\rho, \mu, n, buf) \xrightarrow[\text{execute } i]{\text{rollback, jump } n^b_\ell} (\rho, \mu, n^b, buf')}$$

Both rules evaluate the condition op, as in Section 3.1, resulting in a Boolean value b. The rules then compare the actual branch target n_b 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^b, rule COND-EXECUTE-CORRECT updates *buf* with the fully resolved jump instruction, and emits an observation of $\texttt{jump } n^b_\ell$. This models the fact that an attacker can see which branch was taken, e.g., by timing executions along different paths. The leaked observation n^b 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^b *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 in *buf* that are newer than the current instruction (i.e., all entries $j > i$), sets the program point n to the correct branch, and updates *buf* at index i with correct value for the resolved jump instruction. Since misspeculation can be externally visible through instruction timing [19], the rule issues a `rollback` observation in addition to the `jump` observation.

(a) Predicted correctly

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

(b) Predicted incorrectly

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

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

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

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

Examples. Figure 3 shows how branch prediction affects the reorder buffer. In part (a), the branch at index $\overline{4}$ was 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, causing *buf* to be rolled back to index $\overline{4}$.

3.3 Memory operations

We show the physical instructions for load and store below.

$$(r = \mathsf{load}(\overrightarrow{rv}, n'))$$
$$\mathsf{store}(rv, \overrightarrow{rv}, n')$$

As before, n' is the program point of the next instruction. For loads, r is the register receiving the result, while 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 instructions. The transient counterparts of load and store have the form:

$$\begin{array}{cc} (r = \mathsf{load}(\overrightarrow{rv}))^n & \textit{(unresolved load)} \\ \mathsf{store}(rv, \overrightarrow{rv}) & \textit{(unresolved store)} \\ \mathsf{store}(v_\ell, a_\ell) & \textit{(resolved store)} \end{array}$$

Unresolved loads represent transient load instructions that wait to be executed; upon execution they turn into resolved value instructions, which were introduced in Section 3.1. 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, $[\![addr(\overrightarrow{v})]\!]$ 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. With out-of-order semantics, it is possible for our execution state to have multiple load and store transient instructions concurrently. 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 overwrite the address we will load from. Indeed, some real-world processors behave exactly this way [8].

However, for performance reasons, 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 rather than wait for the store to commit to memory [34].

To properly model forwarding semantics, we modify the syntax of resolved value transient instructions. We add annotations to instructions $(r = v_\ell)$ to recall if they were generated by a load $(r = v_\ell\{i, a\})^n$, where the index i records either the buffer index of the store instruction that forwarded its value to the load, or $\bot$, if there was no such instruction and the value was taken from memory. We also record the memory address a associated with the data, and the program point n of the load instruction that generated the value instruction. The annotation is only relevant for memory rules and can be ignored elsewhere.

Fetch. The fetch stages for load and store are straightforward analogues of the fetch stage for op, and thus share the same rule SIMPLE-FETCH: the current instruction is translated to its transient form and inserted into the buffer at the next largest index. 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.

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

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

Given an execute directive for buffer index i, and 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 $\texttt{read}\ a_{\ell_a}$, 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. Rather than producing a `read` observation, 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, but can instead proceed speculatively. This can lead to mispredicted state when a more recent store to the load's address has not been resolved yet; we show how to deal with misprediction in the rules for the store instruction.

Store execution. We show the rules for stores below.

STORE-EXECUTE-VALUE

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

STORE-EXECUTE-ADDR-OK

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

STORE-EXECUTE-ADDR-HAZARD

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

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. As the order of execution is determined by the directives, a schedule is free to determine whether to resolve address or data 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. This is checked by requiring that all value instructions *after* the current index (i.e., with 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 (i.e., $a_k = a \Rightarrow j_k \geq i$). For this check, we define $\bot < n$, for any index n. That is, if a future load matches addresses with the current store but took its value from memory, this is always considered a misprediction.

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

Retire. Stores are retired using the rule below.

STORE-RETIRE

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

A fully resolved store instruction retires similarly to a value instruction. Instead of updating the register map ρ, rule STORE-RETIRE updates the memory. Since memory writes are observable to an attacker, the rule produces an observation of $\texttt{write}\ a_{\ell_a}$ containing the store's address and security label.

As mentioned previously, load instructions resolve to value instructions, and are retired using rule VALUE-RETIRE from the last section: annotations are ignored when values are retired.

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

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

Figure 4: 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.

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

Retire. Stores are retired using the rule below.

STORE-RETIRE

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

A fully resolved store instruction retires similarly to a value instruction. Instead of updating the register map ρ, rule STORE-RETIRE updates the memory. Since memory writes are observable to an attacker, the rule produces an observation of `write` a_{ℓ_a} containing the store's address and security label.

As mentioned previously, load instructions resolve to value instructions, and are retired using rule VALUE-RETIRE from the last section: annotations are ignored when values are retired.

Examples. Figure 4 gives an example of store-to-load forwarding. In the starting configuration, the store at index $\overline{2}$ is fully resolved, while the store at index $\overline{3}$ has an unresolved address. The first step of the schedule 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 step resolves the address of the store at index $\overline{3}$. This store also matches address 43, but has a more recent index than the store the load forwarded from. This triggers a hazard, leading to the rollback of the load from the reorder buffer.

Spectre examples. We now have enough machinery to capture several variants of Spectre attacks. Figure 5 demonstrates how our semantics models a Spectre v1 attack [19]. The branch at program point $\underline{1}$ serves as a bounds check for array A. Even though the check $4 > r_a$ fails in a sequential execution, the speculative execution proceeds, as the schedule predicts that the true branch will be taken.

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

Directive	Effect on *buf*	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 $\overline{3}$	$\overline{3} \mapsto (r_c = X)$	read a_{sec}
	where $a = Key[1]_{\text{sec}} + 44$	

Figure 5: Example demonstrating a v1 Spectre attack. The branch at $\underline{1}$ acts as bounds check for array A. The execution speculatively ignores the bounds check, resulting in leaking a byte of the secret Key.

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

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.

The execution performs an an out-of-bounds access which aliases to array $Key[1]$. This value is then used in a following load operation, leaking the byte to a public observer.

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.

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

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.

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

3.4 Aliasing prediction

Modern hardware can further speculate on load-store aliasing by including *aliasing prediction*, in which a load can be *predicted* to alias with a past store even if that store's address has not yet been resolved [16, 28]. This provides greater performance whenever a store to a slow-to-calculate address is followed by a load from the same address, e.g., in a tight loop. Aliasing prediction allows the processor to guess the correct store-to-load forwarding before the addresses resolve.

We extend the memory semantics from Section 3.3 to model aliasing prediction by introducing a new transient instruction $(r = \text{load}(\overrightarrow{rv}, (v_\ell, j)))$, which represents a partially executed load with forwarded data. As before, r is the target register and $\overrightarrow{rv}$ is the list of arguments for address calculation. The new parameters are v_ℓ, the forwarded data, and j, the buffer index of the originating store. We now discuss the step-rules for the new instruction.

Forwarding via attacker directives.

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 the latest resolved store to the same address, as in Section 3.3, 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 which store to forward from—supplied via a directive—the rule updates the reorder buffer with the new

partially resolved load and records both the forwarded value and the buffer index of the store the value was forwarded from.

Register resolve function. To state our rules for executing the new partially resolved load instruction, we need to extend the register resolve function $(.+_{.}.)()$ to account for partially executed loads with forwarded data. In particular, whenever the register resolve function computes the latest resolved assignment to some register r, it now needs to not only consider resolved value instructions, but it also needs to account for our new partially executed load: whenever the latest resolved assignment in the buffer is a partially executed load, the register resolve function returns its value.

We can now discuss the execution rules, where the partially executed load may resolve against either the forwarded store or against memory.

Resolving when originating store is in the reorder buffer.

LOAD-EXECUTE-ADDR-OK

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

LOAD-EXECUTE-ADDR-HAZARD

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

To resolve $(r = \mathsf{load}(\overrightarrow{rv}, (v_\ell, j)))^n$ when its forwarding store is still in $buf(j)$, we calculate the load actual target address a and compare it against the target address of the store at $buf(j)$ that was used for forwarding. If the store is not followed by later stores to a, and either **(1)** the store's address is resolved and its address is 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 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 anymore. 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

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

LOAD-EXECUTE-ADDR-MEM-HAZARD

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

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 value instruction annotated as if it had originally 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).

New (theoretical) Spectre attack. Thanks to the alias prediction semantics, we discover a new class of Spectre vulnerability, which we call Spectre-MOB[2], using the Canella et al. naming scheme [6]. An attacker mistrains the alias predictor to cause a store to incorrectly forward (secret) data to an unrelated load, which then proceeds to leak the forwarded data. If an attacker is able to mistrain the alias predictor, this can lead to more serious vulnerability than Spectre v1.1, as it does not depend on an out-of-bounds write gadget already existing in the program for the attacker to cause arbitrary speculative writes. This attack is distinct from the "Microarchitectural Data Samping" (MDS) suite of attacks, as explained in Section 6.

Examples. The example in Figure 8 demonstrates this new attack. The attack is similar to the Spectre v1.1 attack shown in Figure 6, but does not require the store instruction to be within a mispredicted branch. The alias predictor is mistrained and guesses that the load at $\overline{7}$ aliases with the store at $\overline{2}$ before either instruction has resolved its address. The forwarded value x_{sec} is then used in the load at $\overline{8}$, leaking x_{sec} to the attacker.

3.5 Indirect jumps

Next, we extend our semantics with *indirect jumps*. Rather than specifying jump targets *directly* as with the br instruction in Section 3.2, 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

[2]From *Memory Order Buffer* [16].

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

Directive	Effect on buf	Leakage
execute $\overline{2}$: value	$\overline{2} \mapsto$ store(x_{sec}, [40, r_a])	
execute $\overline{7}$: fwd $\overline{2}$	$\overline{7} \mapsto (r_c$ = 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	$\overline{2} \mapsto$ store(r_b, 42_{pub})	fwd 42_{pub}
execute $\overline{7}$	$\{\overline{7}, \overline{8}\} \notin buf$	rollback, fwd 45_{pub}

where $a = x_{\text{sec}} + 48$

Figure 8: Example demonstrating the new Spectre-MOB attack. This attack differs from Spectre v1.1 in that branch misprediction is not needed.

the list of operands for calculating the jump target. The transient form of jmpi is jmpi($\overline{rv}, n_0$), where n_0 is the predicted jump target. To fetch a jmpi instruction, the schedule must issue a new directive, 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.

In Appendix A, we present the detailed rules for indirect jumps, and show how Spectre v2 can be expressed in our semantics.

3.6 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 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, op($succ, r_{sp}$) would be implemented as $r_{sp} - 4$, while 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

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

CALL-RETIRE

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

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 $st(\mathsf{MAX}(st))$, 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

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

RET-FETCH-RSB-EMPTY

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

RET-RETIRE

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

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; we discuss several implementation choices of real processors in Appendix B.

Examples. We present an example of an RSB underflow attack in Figure 9. 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.

3.7 Speculation barriers

Finally, our semantics contains a *speculation barrier* instruction, fence n, that prevents further speculative execution until all prior instructions have been retired. Its transient form is simply fence with no additional parameters; we show its inference rules below.

FENCE-RETIRE

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

The fence instruction uses SIMPLE-FETCH as its fetch rule, and its rule for retire only removes the instruction from the buffer. The

Program

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

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

Figure 9: Example demonstrating a ret2spec-style attack [22]. The attacker is able to send (speculative) execution to an arbitrary program point, shown in red.

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

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

Figure 10: Example demonstrating fencing mitigation against Spectre v1 attacks. The fence instruction prevents the load instructions from executing before the br.

fence instruction does not have an execute stage. However, fence instructions affect the execution of all instructions in the reorder buffer that come *after* them. In prior sections, all execute stage rules contained the condition $\forall j < i : buf(j) \neq \mathsf{fence}$. This condition ensures that as long as a fence instruction remains un-retired in buf, any instructions fetched after the fence cannot be executed.

With this property, we can use fence instructions to restrict out-of-order execution in our semantics. Notably, we can use it to prevent the styles of attacks shown in Figures 5 to 7.

Examples. The example in Figure 10 shows how placing a fence instruction just after the br instruction prevents the Spectre v1 attack from Figure 5. 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. At this point, the misprediction is caught, and the two loads (as well as the fence) are rolled back.

4 PROOFS OF CONSISTENCY AND SECRECY

Our semantics is strictly more general than sequential execution. That is, any property of sequential execution can be captured by our speculative semantics. We underpin this claim by formally proving the following properties:

- If sequential execution of a program produces a physical state μ and ρ, then *any* well-formed speculative schedule will also produce the same physical state (consistency).
- If sequential execution of a program can result in secret leakage, then *any* well-formed speculative schedule will also produce the same secret leakage, or more (secrecy).

Formal execution. We start by defining execution traces. We write $C \ {}_O\Downarrow^N_D \ C'$ if there is a sequence of execution steps from C to C', D and O are the concatenation of the directives and leakages at each execution step, and N is the number of retired instructions. That is, $N = \#\{d \in D \mid d = \text{retire}\}$.

We define *sequential schedules* as schedules that execute and retire instructions immediately upon fetching them. *Sequential execution* is then any execution using a sequential schedule. We write $C \ {}_O\Downarrow^N_{seq} \ C'$ if we execute sequentially.

Consistency. To prove consistency of our semantics, we define the relation $C_1 \approx C_2$ which relates configurations whose physical components are equal. That is, $C_1.\rho = C_2.\rho \wedge C_1.\mu = C_2.\mu$.

We now formally state our theorem:

Theorem 4.1 (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 $C_1 \approx C_2$.*

Our semantics provides the same guarantees as an out-of-order processor: Even though execution may proceed speculatively and out-of-order, the effects on memory and the register file will still be as if we executed sequentially.

Secrecy. Due to speculation and instruction reordering, an observation trace from a speculative schedule may differ from that of a sequential trace. We can, however, show that if a given label appears somewhere in the sequential trace of a program, then it must also appear in every speculative trace of the program. We define $\ell \notin o$ to mean that label ℓ appears nowhere in the expression for o, and prove the following:

Theorem 4.2 (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 $\forall o \in O_2 : \ell \notin o$.*

We can use Theorem 4.2 to prove a slightly stronger statement: If a program is free from Spectre attacks, then it is also constant-time. That is, freedom from Spectre attacks is a strictly stronger property than constant-time.

We first define constant-time and Spectre freedom as non-interference properties in our semantics:

Definition 4.3 (Sequential observational non-interference). We say a configuration C satisfies *sequential observational non-interference* (ONI) with respect to a low-equivalence relation $\simeq_{\text{pub}}$ iff for every C' such that $C \simeq_{\text{pub}} C'$:

$$C \ {}_O\Downarrow^N_{seq} \ C_1 \text{ iff } C' \ {}_{O'}\Downarrow^N_{seq} \ C_1' \text{ and } C_1 \simeq_{\text{pub}} C_1' \text{ and } O = O'.$$

We can extend this notion to speculative execution as follows:

Definition 4.4 (Speculative observational non-interference). We say a configuration C with schedule D satisfies *speculative observational non-interference* (SONI) with respect to a low-equivalence relation $\simeq_{\text{pub}}$ iff for every C' such that $C \simeq_{\text{pub}} C'$:

$$C \ {}_O\Downarrow^N_D \ C_1 \text{ iff } C' \ {}_{O'}\Downarrow^N_D \ C_1' \text{ and } C_1 \simeq_{\text{pub}} C_1' \text{ and } O = O'.$$

Note that when determining ONI, the sequential schedules for C and C' may not be the same, e.g., if C has a secret-dependent branch. On the other hand, when determining SONI, we must fix a schedule D along with the initial configuration C. This definition of SONI corresponds to a threat model where an adversary that can animate the execution of programs must distinguish between two executions with different initial configurations.

We can now prove:

Proposition 4.5. *For a given initial configuration C and schedule D, if C is SONI with respect to D, then C is also ONI.*

As a final remark, note that our notion of SONI implicitly assumes that the program output is private. Following [4], one can modify our definition to consider the case where the program output is public.

Complete definitions and proofs are given in Appendix C.

5 PITCHFORK

We present a static analysis tool, Pitchfork, built on the `angr` binary-analysis suite [30]. Pitchfork is based on a subset of our semantics; we do not detect Spectre violations based on alias prediction, indirect jumps, or return stack buffers (Sections 3.4 to 3.6). However, Pitchfork still allows us to detect attacks based on Spectre variants 1, 1.1, and 4—which includes the load-store forwarding described in Section 3.3. We give intuition for the soundness of Pitchfork's algorithm, and present the results of running Pitchfork against both traditional Spectre test cases and new test cases we developed.

Pitchfork algorithm. Pitchfork examines only a subset of the possible schedules for the input program. It refrains from considering all possible reorderings of instructions, instead focusing on a much smaller set of schedules which represent worst-cases. Specifically, we assume we are given a *speculation bound*, which limits the size of the reorder buffer (and thus the amount of speculation). If any Spectre violations exist in a given program for the given bound, then those violations will manifest in one or more of the worst-case schedules examined by Pitchfork.

Pitchfork examines schedules that keep the reorder buffer full whenever possible, delaying retiring instructions as long as it can. This has no effect on correct-path execution but results in maximizing mispredicted paths. Worst-case schedules thus also involve conditional branches remaining unresolved until just before they are retired. Conversely, most instructions other than conditional branches are resolved eagerly, i.e., they are executed immediately after being fetched.

To account for the load-store forwarding hazards described in Section 3.3, for every store instruction in the program, Pitchfork examines two sets of schedules: one where the store address is resolved eagerly and one where the store address is resolved as late as possible. This ensures that we consider both cases: the case where the store can forward data to a subsequent load, and the case where subsequent loads may instead read stale values.

Reorderings of instructions other than conditional branches and stores are 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. Therefore, we need only consider schedules where instructions other than conditional branches and stores are executed in order.

For the Spectre variants covered by Pitchfork, examining this resulting set of schedules is sufficient to capture any vulnerabilities.

We formalize Pitchfork schedules in more detail in Appendix D.

Pitchfork evaluation. We evaluate Pitchfork by running it against three sets of Spectre test cases. The first set of test cases is the well-known Kocher Spectre v1 examples [18]. The test cases are given in source form; we first compile them to binary using the `clang` compiler at optimization level `-O3`, then run Pitchfork on the resulting files. Pitchfork detects violations in all of the Kocher test cases except #8. In that test case, our `clang` compiler emits an x86 `cmov` instruction rather than a vulnerable conditional branch; modern processors do not perform prediction on `cmov` instructions [9], meaning that the binary code for #8 is indeed safe from Spectre v1 and that Pitchfork is correct to not report a violation.

Pitchfork can also be run without speculative execution, i.e., with purely sequential semantics. Ideally, there would be no violations on any of the Spectre test cases when executing non-speculatively. Unfortunately, many of the Kocher test cases still exhibit violations during sequential execution, often because of out-of-bounds errors which exist even on the non-speculative path. Therefore, we have created a revised set of Spectre v1 test cases based on the Kocher test cases, which do not exhibit violations when executed sequentially.

Thirdly, we have also developed a similar set of test cases for Spectre v1.1 data attacks. Again, on these test cases, Pitchfork reports violations in every case, but reports no violations when executing sequentially.

Limitations. Our static analysis necessarily involves several approximations which make it incomplete, i.e., it will not necessarily find *all* possible Spectre violations, and cannot be used to prove the absence of Spectre vulnerabilities. For instance, Pitchfork does not fully consider all possible addresses for memory write operations, and does not reason about speculative writes which overwrite function return addresses, meaning it will not find certain Spectre v1.1 violations. Nonetheless, despite its approximations, Pitchfork still finds violations in all of the vulnerable test cases discussed above.

6 RELATED WORK

Speculative semantics/foundations. The semantics given by Mcilroy et al. [23] are the most similar to ours. Their semantics uses a three-stage issue/execute/commit system, which is analogous to our fetch/execute/retire stages. However, they choose to explicitly model branch predictor and cache state, and evaluate side-channels using a step-timer; their security analysis is thus based on counts of execution steps rather than non-interference or label-based analysis. Furthermore, they do not model the effects of speculative barriers, nor other microarchitecture features such as store-forwarding. Although they give examples of a variety of Spectre attacks, their presented semantics can only model Spectre v1 attacks.

Both Guarnieri et al. [14] and Cheang et al. [7] define a speculative semantics and a corresponding notion of speculative security. Both semantics only handle speculation through branch prediction, where the predictor is left abstract, and does not allow for general out-of-order execution nor other types of speculation. Guarnieri et al. use a transactional model, where instructions are successively grouped up once speculation starts, and are later either committed or rolled back. They reason about speculation bounds by having their branch prediction oracle specify how many instructions to continue down a branch before checking whether to roll back or commit. Cheang et al. instead keep track of a *speculation level* that increases on misprediction and decreases on branch resolution; all architectural state in their semantics is indexed by speculation level. Their execution model is similar to that of Pitchfork's restricted schedules: instructions are evaluated eagerly, except for mispredicted branches, which are evaluated as late as possible.

Both [14] and [7] also implement binary static analysis tools based on their semantics and are able to detect Spectre v1 attacks. As such, their evaluation is solely on the Kocher test cases.

Disselkoen et al. [11] explore speculation effects on memory through a Spectre-aware relaxed memory model rather than architecture-level semantics. This allows them to reason about concurrent shared accesses to memory, which our semantics does not. Their abstraction sits at a higher level, and is orthogonal to our approach.

Spectre detection tools. Several recent static analysis tools detect Spectre vulnerabilities, but do not present semantics.

The oo7 static analysis tool [33] uses a set of taint propagation rules to detect Spectre attacks for several Spectre variants, and is able to detect all of the variants our semantics capture except for Spectre-MOB. It is additionally able to automatically insert mitigations for Spectre variants 1, 1.1, and 1.2. The oo7 tool detects Spectre vulnerabilities stemming from *untrusted inputs* rather than secret data; the oo7 authors consider all external inputs, such as network or file data, to be untrusted.

Wu and Wang [36] extend *must-hit* cache analysis to the speculative domain. Their tool, which performs abstract interpretation of LLVM programs, is limited to Spectre v1 attacks.

Constant-time analysis tools. We prove in Section 4 that speculative non-interference is a strictly stronger property than constant-time. Thus our work can be seen as an improvement on prior work that statically checks code for constant-time violations [1, 4, 35].

Recent speculative execution attacks. Several speculative execution attacks were published on May 14, when Intel's embargo on the vulnerability ended. The ZombieLoad [29], RIDL [32], and Fallout [24] attacks are all variants of "Microarchitectural Data Sampling" (MDS) attacks. The MDS attacks allow an attacker to forward data directly from transient victim load and store instructions to attacker-issued load instructions. Notably, the target address of the attacker's load instructions need not be related to the victim memory instructions. Although this is similar in effect to our Spectre-MOB attack, the mechanism of the MDS attacks relies on triggering memory faults to induce the incorrect forwarding behavior, as opposed to a mistrained predictor. This places the MDS attacks in the Meltdown [21] family of attacks rather than Spectre—indeed, the ZombieLoad authors classify their attack as *Meltdown-MCA* in the Canella et al. naming scheme.

Another attack released on the same day, Store-to-Leak Forwarding [28], describes a "Meltdown-like" vulnerability the authors term

Data Bounce. Although Data Bounce, like Spectre-MOB, relies on store-forwarding, the Data Bounce vulnerability only affects store-load pairs that are *correctly* predicted to alias; Data Bounce exploits leakage in virtual-to-physical address translation.

ACKNOWLEDGMENTS

We thank Natalie Popescu for her aid in editing and formatting this paper. This work was supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

REFERENCES

[1] 2010. Checking that functions are constant time with Valgrind. https://github.com/agl/ctgrind/. (2010). https://github.com/agl/ctgrind/

[2] 2016. Coding Rules. https://cryptocoding.net/index.php/Coding_rules. (2016). Retrieved June 9, 2017 from https://cryptocoding.net/index.php/Coding_rules

[3] 2019. mbed TLS. https://github.com/armmbed/mbedtls. (2019). Retrieved May 16, 2018 from https://github.com/armmbed/mbedtls

[4] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *Proceedings of the 25th USENIX Security Symposium*. USENIX Association. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida.

[5] 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, 1267–1279.

[6] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2018. A Systematic Evaluation of Transient Execution Attacks and Defenses. *CoRR* abs/1811.05441 (2018). arXiv:1811.05441 http://arxiv.org/abs/1811.05441

[7] Kevin Cheang, Cameron Rasmussen, Sanjit Seshia, and Pramod Subramanyan. 2019. A Formal Approach to Secure Speculation. Cryptology ePrint Archive, Report 2019/310. (2019). https://eprint.iacr.org/2019/310.

[8] Tien-Fu Chen and Jean-Loup Baer. 1992. Reducing Memory Latency via Non-blocking and Prefetching Caches. *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (1992).

[9] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. 2009. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *30th IEEE Symposium on Security and Privacy,*. IEEE, 45–60.

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

[11] Craig Disselkoen, Radha Jagadeesan, Alan Jeffrey, and James Riely. 2019. The Code That Never Ran: Modeling Attacks on Speculative Evaluation. In *S&P 2019*.

[12] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. 2018. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 693–707. https://doi.org/10.1145/3173162.3173204

[13] 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* 8, 1 (2018), 1–27.

[14] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. 2018. SPECTECTOR: Principled Detection of Speculative Information Flows. *CoRR* abs/1812.08639 (2018). arXiv:1812.08639 http://arxiv.org/abs/1812.08639

[15] Jann Horn. 2018. Issue 1528: speculative execution, variant 4: speculative store bypass. (2018). https://bugs.chromium.org/p/project-zero/issues/detail?id=1528

[16] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gülmezoglu, Thomas Eisenbarth, and Berk Sunar. 2019. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. *CoRR* abs/1903.00446 (2019). arXiv:1903.00446 http://arxiv.org/abs/1903.00446

[17] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative Buffer Overflows: Attacks and Defenses. *CoRR* abs/1807.03757 (2018). arXiv:1807.03757 http://arxiv.org/abs/1807.03757

[18] Paul Kocher. 2018. Spectre mitigations in Microsoft's C/C++ compiler. (2018). https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html

[19] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA. https://doi.org/10.1109/SP.2019.00002

[20] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. USENIX Association, Baltimore, MD. https://www.usenix.org/conference/woot18/presentation/koruyeh

[21] 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 Security 18)*. USENIX Association, Baltimore, MD, 973–990.

[22] 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 (CCS '18)*. ACM, New York, NY, USA, 2109–2122. https://doi.org/10.1145/3243734.3243761

[23] Ross McIlroy, Jaroslav Sevcík, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. 2019. Spectre is here to stay: An analysis of side-channels and speculative execution. *CoRR* abs/1902.05178 (2019). arXiv:1902.05178 http://arxiv.org/abs/1902.05178

[24] Marina Minkin, Daniel Moghimi, Moritz Lipp, Michael Schwarz, Jo Van Bulck, Daniel Genkin, Daniel Gruss, Berk Sunar, Frank Piessens, and Yuval Yarom. 2019. Fallout: Reading Kernel Writes From User Space. (2019).

[25] OpenSSL. 2019. Security policy 12th May 2019. https://www.openssl.org/policies/secpolicy.html. (2019). https://www.openssl.org/policies/secpolicy.html

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

[27] Thomas Pornin. 2018. Constant-Time Toolkit. https://github.com/pornin/CTTK. (2018). 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. (2019). https://cpu.fail/store_to_leak_forwarding.pdf

[29] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. (2019). https://zombieloadattack.com

[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 *IEEE Symposium on Security and Privacy*.

[31] Paul Turner. 2019. Retpoline: a software construct for preventing branch-target-injection. (2019). 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 *S&P*.

[33] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. 2018. oo7: Low-overhead Defense against Spectre Attacks via Binary Analysis. *CoRR* abs/1807.05843 (2018). arXiv:1807.05843 http://arxiv.org/abs/1807.05843

[34] Henry Wong. 2014. Store-to-Load Forwarding and Memory Disambiguation in X86 Processors. (2014). http://blog.stuffedcow.net/2014/01/x86-memory-disambiguation/

[35] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. 2018. Eliminating Timing Side-channel Leaks Using Program Repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*. ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/3213846.3213851

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

A INDIRECT JUMP SEMANTICS

Semantics. The semantics for jmpi are given below:

JMPI-FETCH

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

JMPI-EXECUTE-CORRECT

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

JMPI-EXECUTE-INCORRECT

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

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 [12], we model the guess as an attacker directive.

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 1, the program makes an indirect jump. An adversary can mistrain the predictor to send execution to 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 mitigation for Spectre v2 attacks is to replace indirect jumps with *retpolines* [31]. Figure 12 shows a retpoline construction that would replace the indirect jump in Figure 11. The call sends execution to program point 5, while adding 4 to the RSB. The next two instructions at 5 and 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 7 will pop the top value off the RSB, 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 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.

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	$\mu(n)$
1	$(r_c = \text{load}([48, \texttt{r\_a}], 2))$
2	fence 3
3	jmpi($[12, r_b]$)
…	
16	fence 17
17	$(r_d = \text{load}([44, \texttt{r\_c}], 18))$

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

B IMPLEMENTATION CHOICES WHEN THE RETURN STACK BUFFER IS EMPTY

In practice, there are several variants on what processors actually do when the RSB is empty [22]:

- 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 $\bot$.

C FULL PROOFS

C.1 Consistency

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

Registers	
r	$\rho(r)$
r_b	8_{pub}
r_{sp}	$7\mathsf{C}_{\text{pub}}$

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

Directive	Effect on buf	Leakage
execute $\overline{4}$	$\overline{4} \mapsto r_{sp} = 7\mathsf{B}$	
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	$\overline{7} \mapsto \text{store}(20, 7\mathsf{B})$	`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 20`

Figure 12: 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.

Definition C.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 C.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.

As stated in Section 4, we write $C\ {}_O\Downarrow^N_{seq}\ C'$ if we execute sequentially.

Lemma C.5 (Sequential consistency). *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 affect independent parts of the buffer in the store(value and address).
- 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 C.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 [3] as defined in Definition C.6.

Theorem C.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 $C_1 = C_2$.*

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

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 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 C.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 C.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 Theorem C.5, we have $C_1' = C_2'$. Thus $C_1 \approx C_2$ (resp. $C_1 = C_2$). □

C.2 Security

Theorem 4.2 (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 C.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 C.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 secret label, we get the following corollary:

Corollary C.9 (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 C.10. *For a given initial configuration C and well-formed schedule D, if C is SONI with respect to D, and execution of C with D results in a terminal configuration C_1, then C is also ONI.*

Proof. Since C is SONI, 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 C.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 Theorem C.9, it follows that no secret labels appear in either O_{seq} or O'_{seq}, and thus $O_{seq} = O'_{seq}$. □

D FORMALIZING PITCHFORK

As described in Section 5, Pitchfork examines only a subset of the possible schedules of the given program; for brevity, we will refer to these as the set of *Pitchfork schedules*. In this appendix, we formalize the definition of the Pitchfork schedules using a recursive construction.

Let there be an initial configuration C and a speculation bound k; we define $PFork(C, k)$ to be the set of Pitchfork schedules for C and k. For the purposes of the recursive construction, we also define $PForkPrefix(C, k, n)$ as the set of all schedules of length n directives which are prefixes of any schedules in the set $PFork(C, k)$. As a base case, $PForkPrefix(C, k, 0) = \{\emptyset\}$ for all C and k: there is a single Pitchfork schedule prefix of length 0, namely, the empty set.

For the inductive case, we define

$$PForkPrefix(C, k, n) = \bigcup \{\{D{+}{+}d \mid d \in succ(D, C, k)\} \mid D \in PForkPrefix(C, k, n-1)\}$$

where ++ is the concatenation operator, and $succ(D, C, k)$ is the set of possible choices for the nth directive which Pitchfork will examine. That is, given each possible schedule D of length $n-1$, we compute the possible choices for the nth directive (using a function *succ* which we will define shortly) and consider D concatenated with each of them. This forms the set of Pitchfork schedules of length n.

We define the function $succ(D, C, k)$ as follows. Recall that D is a schedule in the set $PForkPrefix(C, k, n-1)$—that is, D is a sequence of $n-1$ directives. Let C' be given by $C \Downarrow_D C'$, that is, the state after executing the $n-1$ directives in D. Further let y be $\mathsf{MAX}(C'.buf)$, and let z be $\mathsf{MIN}(C'.buf)$ (i.e., $buf[z]$ is the oldest outstanding instruction in $C'.buf$).

(1) If $buf[y]$ is unresolved and $buf[y]$ is not a br, fence, or store, then $succ(D, C, k)$ is {execute y}—that is, Pitchfork considers only one possibility for the nth directive, namely, executing y.
(2) Otherwise, if $buf[y]$ is a store with unresolved value, then $succ(D, C, k)$ is {execute y : value}.
(3) Otherwise, if $buf[y]$ is a store with unresolved address and the previous directive was fetch,[4] then $succ(D, C, k)$ is {execute y : addr} $\cup D'$, where D' is the result of applying whichever of the following rules applies. That is, Pitchfork considers both the possibility that we eagerly resolve the store's address, or that we do not (in which case the store's address will eventually be resolved by rule 6 once it reaches the end of the speculation window).
(4) Otherwise, if there are fewer than k instructions in $C'.buf$ and $buf[y]$ is not fence, then $succ(D, C, k)$ is {fetch}, unless the next instruction is a br, in which case $succ(D, C, k)$ is {fetch: true, fetch: false}—that is, Pitchfork considers both the possible branch predictions.
(5) Otherwise, if $buf[z]$ is an unresolved br, then $succ(D, C, k)$ is {execute z}.
(6) Otherwise, if $buf[z]$ is a store with unresolved address, then $succ(D, C, k)$ is {execute z : addr}.
(7) Otherwise, $succ(D, C, k)$ is {retire}.

These rules have the effect that conditional branches are always resolved as late as possible, stores are either resolved as early as possible or as late as possible, and all other instructions are always resolved as early as possible. As argued in Section 5, examining the resulting set of Pitchfork schedules is sufficient to capture any Spectre vulnerabilities that may exist (of the variants covered by Pitchfork, i.e., variants 1, 1.1, and 4).

[4]The restriction on the previous directive is necessary to avoid triggering this rule multiple times for the same store. Each store should only get one chance to eagerly resolve: immediately after it was fetched.