Towards Constant-Time Foundations for the New Spectre Era

Sunjay Cauligi† Craig Desselken‡ Klaus v. Gleissenthal†
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. 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 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
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 prediction 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 \( \sqsupseteq \) for its join operator. Let \( \ell \) range over \( \mathcal{L} \). We use \( v_\ell \) to denote the value \( v \) annotated with label \( \ell \) 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 \( \text{Instr} \) and set of transient instructions \( \text{TransInstr} \). 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 \( \text{buf}(i) \) to denote the instruction at index \( i \) in buffer \( \text{buf} \), if \( i \) is in \( \text{buf} \)’s domain. To define our semantics, we will need to modify \( \text{buf} \) by adding and removing entries. We write \( \text{buf}(i \mapsto \text{instr}) \) to denote the result of extending \( \text{buf} \) with the mapping from \( i \) to \( \text{instr} \), and write \( \text{buf}\setminus\text{buf}(i) \) to denote the function formed by removing \( i \) from \( \text{buf} \)’s domain. We write \( \text{buf}[j : j < i] \) to denote the restriction of \( \text{buf} \)’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 \( \text{buf} \)’s domain will always be contiguous.

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

For a formula \( \varphi \), 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 \( \varphi \) holds, i.e., \( \varphi(j) \land \forall j’ : j < j’ < i \Rightarrow \neg \varphi(j’) \). Analogously, we define \( \text{min}(j) \) as \( i : \varphi(j) \).

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

- \( \rho : \mathcal{R} \to \mathcal{V} \) is a map from a finite set of register names \( \mathcal{R} \) to labeled values;
- \( \mu : \mathcal{V} \to \mathcal{V} \) is a memory;
- \( n : \mathcal{V} \) is the current program point;
**buf : \mathbb{N} \rightarrow \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 \( r \), \( \mu(n) \) 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 \( \xrightarrow{\sigma} \), where a step of the form \( C \xrightarrow{d} C' \) 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 \( C \xrightarrow{d} \bot \).

### 3.1 Basic out-of-order execution

**Instructions.** The op instruction has the form \( (r = \text{op}(\rho, \overline{\nu}, n')) \), where \( \text{op} \) indicates which arithmetic operation to perform, \( \overline{\nu} \) 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:

\[
\begin{align*}
(r = \text{op}(\rho, \overline{\nu}, n')) & \quad (\text{unresolved op}) \\
(r = \nu_i) & \quad (\text{resolved value})
\end{align*}
\]

where \( \nu_i \) 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 \( \langle \cdot, \cdot \rangle \), 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 \( \text{buf} \). 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 \( \rho \), 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 \( (\text{buf} + i_r)(r) = \bot \). We extend this definition to values by defining \( (\text{buf} + i_r)(\nu_i) = \nu_i \), for all \( \nu_i \in \mathcal{V} \), and lift it to lists of registers or values using a pointwise lifting.

**Inference rules for op.**

**fetch**

\[
\begin{align*}
\text{simple-fetch} \\
\mu(n) \in \{ \text{op}, \text{load, store, fence} \} \\
\text{MAX}(\text{buf}) + 1 \\
\text{buf}^i = \text{buf}[i \mapsto \text{transient}(\mu(n))]
\end{align*}
\]

\[
\begin{array}{c}
(p, \mu, n, \text{buf}) \xrightarrow{\text{fetch}} (p, \mu, n', \text{buf}^i)
\end{array}
\]

**execute**

\[
\begin{align*}
\forall j < i : \text{buf}(j) \neq \text{fence} \\
\text{buf}^i & = \text{buf}[i \mapsto \text{transient}(\mu(n))]
\end{align*}
\]

\[
\begin{array}{c}
\text{op-execute} \\
\text{buf}(i) = (r = \text{op}(\rho, \overline{\nu})) \\
\nu_j = \nu_i \\
\text{buf}^i = \text{buf}[i \mapsto (r = \nu_i)]
\end{array}
\]

**retire**

\[
\begin{align*}
\text{value-retire} \\
\text{MIN}(\text{buf}) = i \\
\rho'_r = \rho[r \mapsto \nu_i] \\
\text{buf}^i = \text{buf} \setminus \text{buf}(i)
\end{align*}
\]

\[
\begin{array}{c}
(p, \mu, \text{buf}) \xrightarrow{\text{retire}} (p', \mu, \text{buf}^i)
\end{array}
\]

**Fetch.** Given a fetch directive, rule **simple-fetch** extends the reorder buffer \( \text{buf} \) 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 \( \text{buf} \), 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 \( \overline{\nu} \) used to evaluate operation \( \text{op} \) 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 \( \text{buf} \) with the resolved instruction \( (r = \nu_i) \), where \( \nu_i \) 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.
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 $\rho$ 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 2 can be executed before instruction at program point 1, since instruction 2 does not depend on register $r_a$—the target register of the instruction at 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 2 depends on register $r_a$ which is assigned in instruction 1, we cannot execute instruction 2 before 1 has been fully resolved. Attempting the directive execute 2 will result in stuck execution ($\perp$), 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 $\text{br}(op, \overline{\ell}, \text{true}, \text{false})$, where $op$ is a Boolean operator whose result determines whether or not to execute the jump. $\overline{\ell}$ are the operands to $op$, and $n_{\text{true}}$ and $n_{\text{false}}$ 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.

\[
\begin{align*}
\text{br}(op, \overline{\ell}, n_{\text{true}}, n_{\text{false}}) & \quad \text{(unresolved conditional)} \\
\text{jump } n_0 & \quad \text{(resolved conditional)}
\end{align*}
\]

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

\[
\begin{align*}
\text{cond-fetch} \\
\mu(n) &= \text{br}(op, \overline{\ell}, n_{\text{true}}, n_{\text{false}}) \\
& \text{where } i = \max(\mu(\text{buf})) + 1 \\
buf' &= \text{buf}[i \mapsto \text{br}(op, \overline{\ell}, n_{\text{true}}, n_{\text{false}})] \\
\mu(n) &= \text{buf}[i \mapsto \text{br}(op, \overline{\ell}, n_{\text{true}}, n_{\text{false}})] \\
\text{fetch: } b
\end{align*}
\]

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 $n_{\text{true}}$ or $n_{\text{false}}$ 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.

\[
\begin{align*}
\text{cond-execute-correct} \\
\forall j < i : \text{buf}(j) \neq \text{fence} & \quad (b_i \downarrow \rho(\overline{\ell}) = \overline{\ell}) \\
\text{buf}(i) &= \text{br}(op, \overline{\ell}, n_{\text{true}}, n_{\text{false}}) \\
& \quad \text{where } i = \max(\mu(\text{buf})) + 1 \\
& \quad \text{buf}[i \mapsto \text{br}(op, \overline{\ell}, n_{\text{true}}, n_{\text{false}})] \\
& \quad \text{fence: } b
\end{align*}
\]

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 $n_0$ 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 $\text{jump } n^b$. 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 $\ell$, 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.
The jump instruction is resolved, and execution proceeds as normal.

Towards Constant-Time Foundations for the New Spectre Era

store have the form: $J(\mu, r_r, r_g, \ell)$ calculates target addresses for stores and loads from its operands. For example, in an x86-style address calculation, $\lceil \text{addr}(\ell) \rceil$ might compute the sum of its operands; in an x86-style address mode, $\lceil \text{addr}(i, v_2, v_3) \rceil$ 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 $(r = \nu_{\ell})$ to recall if they were generated by a load $(r = \nu_{\ell}(i, a))^a$, where the index $i$ records either the buffer index of the store instruction that forwarded its value to the load, or ↓, 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 transpose form and inserted into the buffer at the next largest index. Note that transient(·) annotates the transient load instruction with its program point.

### Load execution

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

#### Load-execute-nodep

$$\text{buf}(i) = (r = \text{load}(\ell))\quad \forall j < i : \text{buf}(j) \neq \text{fence}$$

$$\ell_a = \text{load}(\ell) \quad \lceil \text{addr}(\ell) \rceil = a$$

$$\text{buf}(i) = \text{store}(\ell) \quad \forall j < i : \text{buf}(j) \neq \text{store}(\ell) \quad (r = \nu_{\ell}(i, a))^a$$

$$\text{buf}(i) = (r = \text{load}(\ell))\quad \forall j < i : \text{buf}(j) \neq \text{fence}$$

$$\ell_a = \text{load}(\ell) \quad \lceil \text{addr}(\ell) \rceil = a$$

$$\text{buf}(i) = \text{store}(\ell) \quad \forall j < i : \text{buf}(j) \neq \text{store}(\ell) \quad (r = \nu_{\ell}(i, a))^a$$

$$\text{buf}(i) = (r = \text{load}(\ell))\quad \forall j < i : \text{buf}(j) \neq \text{fence}$$

$$\ell_a = \text{load}(\ell) \quad \lceil \text{addr}(\ell) \rceil = a$$

$$\text{buf}(i) = \text{store}(\ell) \quad \forall j < i : \text{buf}(j) \neq \text{store}(\ell) \quad (r = \nu_{\ell}(i, a))^a$$

**Examples.** Figure 3 shows how branch prediction affects the reorder buffer. In part (a), the branch at index 4 was predicted correctly. The branch instruction is resolved, and execution proceeds as normal.

In part (b), the branch at index 4 is incorrectly predicted. Upon executing the branch, the misprediction is detected, causing buf to be rolled back to index 4.

### 3.3 Memory operations

We show the physical instructions for load and store below.

$$(r = \text{load}(\ell))$$

$$(\text{store}(\ell))$$

As before, $n'$ is the program point of the next instruction. For loads, $r$ is the register receiving the result, while for stores, $r_c$ is the register or value to be stored. For both loads and stores, $\overrightarrow{\ell}$ 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:

$$(r = \text{load}(\overrightarrow{\ell}))\quad \text{store}(\ell_1, \overrightarrow{\ell})$$

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, $\lceil \text{addr}(\ell) \rceil$
memory. Finally, the rule produces the observation \( \text{read}\ a_{r_{\alpha}} \), which renders the memory read at address \( a \) with label \( \ell_{a} \) visible to an attacker.

Rule **load-execute-forward** applies if the most recent store instruction in \( \text{buf} \) 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 \( \text{fwd} \) observation with the labeled address \( a_{r_{\alpha}} \). 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**

\[
\begin{align*}
\text{buf}(i) &= \text{store}(\tau, \overline{\tau}, \overline{\tau}) \\
\text{buf}(i) &= \text{store}(\tau, \overline{\tau}, \overline{\tau}) \\
\text{buf} &\Rightarrow \text{store}(\overline{\tau}, \overline{\tau}, \overline{\tau}) \\
\end{align*}
\]

\[\forall j < i : \text{buf}(j) \neq \text{fence} \]

\[(\rho, \mu, n, \text{buf}) \Rightarrow (\rho, \mu, n, \text{buf}^*) \]

**store-execute-addr-ok**

\[
\begin{align*}
\text{buf}(i) &= \text{store}(\tau, \overline{\tau}, \overline{\tau}) \\
\forall k > i : \text{buf}(k) &= \text{store}(\overline{\tau}, \overline{\tau}, \overline{\tau}) \\
\forall k > i : \text{buf}(k) &= \text{store}(\overline{\tau}, \overline{\tau}, \overline{\tau}) \\
\end{align*}
\]

\[(\rho, \mu, n, \text{buf}) \Rightarrow \text{store}(\overline{\tau}, \overline{\tau}, \overline{\tau}) \]

\[\forall j < i : \text{buf}(j) \neq \text{fence} \]

\[\forall k > i : \text{buf}(k) = \text{store}(\overline{\tau}, \overline{\tau}, \overline{\tau}) \]

\[a_k = a \Rightarrow j_k \geq i \wedge (j_k = i \Rightarrow a_k = a) \]

\[\text{buf}^* = \text{buf}[i \mapsto \text{store}(\tau, a_{r_{\alpha}})] \]

\[\text{fwd}_{a_{r_{\alpha}}} \Rightarrow (\rho, \mu, n, \text{buf}^*) \]

**store-execute-addr-hazard**

\[
\begin{align*}
\text{buf}(i) &= \text{store}(\tau, \overline{\tau}, \overline{\tau}) \\
\forall k > i : \text{buf}(k) &= \text{store}(\overline{\tau}, \overline{\tau}, \overline{\tau}) \\
\forall k > i : \text{buf}(k) &= \text{store}(\overline{\tau}, \overline{\tau}, \overline{\tau}) \\
\end{align*}
\]

\[\forall j < i : \text{buf}(j) \neq \text{fence} \]

\[\forall k > i : \text{buf}(k) = \text{store}(\overline{\tau}, \overline{\tau}, \overline{\tau}) \]

\[a_k = a \land j_k < i \vee (j_k = i \land k < a_k = a) \]

\[\text{buf}^* = \text{buf}[j : j < k][i \mapsto \text{store}(\tau, a_{r_{\alpha}})] \]

\[\text{rollback}_{a_{r_{\alpha}}} \Rightarrow (\rho, \mu, n, \text{buf}^*) \]

The execution of the store is split into two steps: value resolution, represented by the directive execute \( i \), and address resolution, represented by the directive execute \( i \). As the order of execution is determined by the directives, a schedule is free to determine whether to resolve data or address 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 \( \perp < n \), 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 \( j \)), restarts the execution by resetting the next instruction pointer to the program point \( n_j \) 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 misprediction, the rule issues a rollback observation.

**Retire.** Stores are retired using the rule below.

\[
\begin{align*}
\text{store-retire} \\
\text{store-retire} \\
\end{align*}
\]

\[\text{MIN}(\text{buf}) = i \\
\mu = \text{store}(\overline{\tau}, a_{r_{\alpha}}) \\
\mu = \text{store}(\overline{\tau}, a_{r_{\alpha}}) \\
\]

\[\text{buf}^* = \text{buf} \setminus \text{buf}(i) \\
\text{buf}^* = \text{buf} \setminus \text{buf}(i) \\
\]

\[\text{store(expr)} = \text{store}(\overline{\tau}, a_{r_{\alpha}}) \\
\text{store(expr)} = \text{store}(\overline{\tau}, a_{r_{\alpha}}) \\
\]

\[\text{write}_{a_{r_{\alpha}}} \Rightarrow (\rho, \mu, n, \text{buf}^*) \\
\text{write}_{a_{r_{\alpha}}} \Rightarrow (\rho, \mu, n, \text{buf}^*) \\
\]

A fully resolved store instruction retires similarly to a value instruction. Instead of updating the register map \( \rho \), rule **store-retire** updates the memory. Since memory writes are observable to an attacker, the rule produces an observation of write \( a_{r_{\alpha}} \), 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 7 is resolved fully, while the store at index 3 has an unresolved address. The first step of the schedule executes the load at 4. This load addresses access 43, which matches the store at index 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 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 1 serves as a bounds check for array \( A \). Even though the check \( 4 > r_{\alpha} \) fails in a sequential execution, the speculative execution proceeds, as the schedule predicts that the true branch will be taken.

---

Figure 4: Store hazard caused by late execution of store addresses. The store address for 3 is resolved too late, causing the later load instruction to forward from the wrong store. When 3’s address is resolved, the execution must be rolled back. In this example, \([addr()]\) adds its arguments.
The execution performs an out-of-bounds access which aliases to array \texttt{Key[1]}. This value is then used as the address of another load instruction, resulting in leaking a byte of the secret \texttt{Key[1]}

![Figure 5: Example demonstrating a v1 Spectre attack. The branch at 1 acts as bounds check for array \text{A}. The execution speculatively ignores the bounds check, resulting in leaking a byte of the secret \texttt{Key.}](image)

![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.](image)

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 $(r = \text{load}(\overline{\rho} w, \overline{s} j))$, which represents a partially executed load with forwarded data. As before, $r$ is the target register and $\overline{\rho} w$ is the list of arguments for address calculation. The new parameters are $\overline{s} j$, the forwarded data, and $i$, the buffer index of the originating store. We now discuss the step-rules for the new instruction.

**Forwarding via attacker directives.**

$\text{LOAD-EXECUTE-FORWARDED-GUESSED}$

$$\text{buf}(i) = (r = \text{load}(\overline{\rho} w))$$

$$\begin{align*}
\forall k < i : \text{buf}(k) &\neq \text{fence} \\
\text{buf}(j) &\leftarrow \text{store}(\overline{\rho} w, \overline{s} j) \\
\text{buf}' &= \text{buf}[i \mapsto (r = \text{load}(\overline{\rho} w, \overline{s} j))] \\
\end{align*}$$

**Rule load-execute-forwarded-guesed**

When forward-execute-guesed 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 \( (+, \ldots) \) 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.**

\[
\text{LOAD-EXECUTE-ADDR-OK} \quad \text{buf}(i) = (r = \text{load}(\overrightarrow{u}, (v_i, j)))^n \\
(\text{buf } + j + 1)(\overrightarrow{u}) = \overrightarrow{u}^j \\
(\text{addr}(\overrightarrow{u}^j)) = a \\
\forall k : (k < j < i) : \text{buf}(k) \not= \text{store}(\_j, a) \\
\text{buf}' = \text{buf}[i \mapsto (r = v_i, j, a)]^n \\
(\rho, \mu, n, \text{buf}') \xrightarrow{\text{execute } i} (\rho, \mu, n, \text{buf}')
\]

\[
\text{LOAD-EXECUTE-HAZARD} \quad \text{buf}(i) = (r = \text{load}(\overrightarrow{u}, (v_i, j)))^n \\
(\text{buf } + j + 1)(\overrightarrow{u}) = \overrightarrow{u}^j \\
(\text{addr}(\overrightarrow{u}^j)) = a \\
\forall k : (k < j < i) : \text{buf}(k) = \text{store}(\_j, a) \\
\text{buf}' = \text{buf}[i \mapsto (r = v_i, j, a)]^n \\
\text{rollback, fwd } a_k \\
(\rho, \mu, n, \text{buf}) \xrightarrow{\text{execute } i} (\rho, \mu, n', \text{buf}')
\]

To resolve \((r = \text{load}(\overrightarrow{u}, (v_i, j)))^n\) when its forwarding store is
still in \(\text{buf}(j)\), we calculate the load actual target address \(a\)
and compare it against the target address of the store at \(\text{buf}(j)\) 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-
ADD-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-ADD-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 \(\text{buf}(i)\) 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 \((k_j = i \land a_k \not= a)\).

**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 \(\text{buf}\). If this is the case, and no
other prior store instructions have a matching address, then we
must check the forwarded data against memory.

\[
\text{LOAD-EXECUTE-ADDR-MATCH} \quad \text{buf}(i) = (r = \text{load}(\overrightarrow{u}, (v_i, j)))^n \\
j \not\in \text{buf} \quad (\text{buf } + 1)(\overrightarrow{u}) = \overrightarrow{u}^j \\
(\text{addr}(\overrightarrow{u}^j)) = a \\
\forall \ell < j : \text{buf}(\ell) \not= \text{store}(\_j, a) \\
\mu(a) = v_t \quad \text{buf}' = \text{buf}[i \mapsto (r = v_t, j, a)]^n \\
(\rho, \mu, n, \text{buf}') \xrightarrow{\text{read } a_k \text{ execute } i} (\rho, \mu, n, \text{buf}')
\]

\[
\text{LOAD-EXECUTE-ADDR-HAZARD} \quad \text{buf}(i) = (r = \text{load}(\overrightarrow{u}, (v_i, j)))^n \\
j \not\in \text{buf} \quad (\text{buf } + 1)(\overrightarrow{u}) = \overrightarrow{u}^j \\
(\text{addr}(\overrightarrow{u}^j)) = a \\
\forall \ell < j : \text{buf}(\ell) \not= \text{store}(\_j, a) \\
\mu(a) = v_{\ell'} \quad v_{\ell'} \not= v_t \\
\text{buf}' = \text{buf}[j \mapsto j < i] \\
(\rho, \mu, n, \text{buf}) \xrightarrow{\text{rollback, fwd } a_k \text{ execute } i} (\rho, \mu, n', \text{buf}')
\]

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-
ADD-ADDR-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-ADD-
ADDR-HAZARD).

**New (theoretical) Spectre attack.** Thanks to the alias prediction
semantics, we discover a new class of Spectre vulnerability, which we
call Spectre-MOB², using the Canella et al. naming scheme [6].
An attacker mistrains the alias predictor to cause a store to in-
correctly forward (secret) data to an unrelated load, which then
proceeds to leak the forwarded data. If an attacker is able to mis-

train 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 “Mic-
roarchitectural 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 \(\overrightarrow{a}\) aliases with the store at \(\overrightarrow{b}\) before either instruction has resolved
its address. The forwarded value \(x_{\text{sec}}\) is then used in the load at \(\overrightarrow{a}\),
leaking \(x_{\text{sec}}\) to the attacker.

### 3.5 Indirect jumps

Next, we extend our semantics with indirect jumps. Rather than spec-
ifying 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 jmp(\(\overrightarrow{u}\)), where \(\overrightarrow{u}\) is

²From Memory Order Buffer [16].
Towards Constant-Time Foundations for the New Spectre Era

<table>
<thead>
<tr>
<th>Registers</th>
<th>Reorder buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r )</td>
<td>( \rho(r) )</td>
</tr>
<tr>
<td>( r_a )</td>
<td>( Z_{pub} )</td>
</tr>
<tr>
<td>( r_b )</td>
<td>( x_{sec} )</td>
</tr>
<tr>
<td>( a )</td>
<td>( \mu(a) )</td>
</tr>
</tbody>
</table>

| 40..43 | secretKey_{sec} |
| 44..47 | pubArr_{r_{pub}} |
| 48..4B | pubArr_{r_{pub}} |

<table>
<thead>
<tr>
<th>Directive</th>
<th>Effect on ( buf )</th>
<th>Leakage</th>
</tr>
</thead>
<tbody>
<tr>
<td>execute ( 2 ): value</td>
<td>( 2 \mapsto store(x_{sec}, [40, r_a]) )</td>
<td>( \text{read } a_{sec} )</td>
</tr>
<tr>
<td>execute ( 7 ): fwd ( 2 )</td>
<td>( 7 \mapsto (r_c = load([45], x_{sec}, 2)) )</td>
<td>( \text{rollback} )</td>
</tr>
<tr>
<td>execute ( 8 )</td>
<td>( 8 \mapsto (r_c = X(\ldots, a)) )</td>
<td>( \text{fwd } 42_{pub} )</td>
</tr>
<tr>
<td>execute ( 7 ): addr</td>
<td>( 7 \mapsto store(r_b, 42_{pub}) )</td>
<td>( \text{fwd } 45_{pub} )</td>
</tr>
<tr>
<td>execute ( 7 )</td>
<td>( (7, 8) \notin buf )</td>
<td>( \text{where } a = x_{sec} + 48 )</td>
</tr>
</tbody>
</table>

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(\( \overline{a} \), \( n_0 \)), 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: \( \text{call}(\sigma_f, n_{ret}) \), where \( \sigma_f \) is the target program point of the call and \( n_{ret} \) is the return program point; and the return instruction \( \text{ret} \). 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 \( sp \).

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 \( \text{suc} \). 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 \( \text{pred} \). This step makes use of a temporary register \( r_{mp} \).

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

\[
\begin{align*}
\text{call-direct-fetch} & \quad \mu(n) = \text{call}(\sigma_f, n_{ret}) \quad i = \text{MAX}(buf(i)) + 1 \\
& \quad \text{buf}_1 = \text{buf}[i \mapsto \text{call}][i + 1 \mapsto (r_{sp} = \rho[\text{succ}, r_{sp}])] \\
& \quad \text{buf}_2 = \text{buf}_1[i + 2 \mapsto \text{store}(n_{ret}, r_{sp})] \\
& \quad \sigma' = \sigma[i \mapsto \text{push } n_{ret}] \\
& \quad n' = n_f
\end{align*}
\]

\[
\begin{align*}
\text{call-retrieve} & \quad \text{MIN}(\text{buf}) = i \\
& \quad \text{buf}(i + 1) = \text{call}(\sigma_f, n_{ret}) \quad \rho' = \rho[\text{succ}, r_{sp} \mapsto \nu_f] \\
& \quad \mu' = \mu[\text{push } n_{ret}] \\
& \quad (\rho', \mu', n, \text{buf}', \sigma') \mapsto (\rho, \mu, n, \text{buf}, \sigma)
\end{align*}
\]

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

**Returning.**

\[
\begin{align*}
\text{ret-fetch-RSB} & \quad \mu(n) = \text{ret} \\
\text{top}(\sigma) = n' & \quad i = \text{MAX}(\text{buf}f) + 1 \quad \text{buf}f_i = \text{buf}[i \mapsto \text{ret}] \\
\text{buf}f_2 & = \text{buf}f_1[i + 1 \mapsto (\text{rmp} = \text{load}(\text{reg}])] \\
\text{buf}f_3 & = \text{buf}f_2[i + 2 \mapsto (\text{sp} = \text{op}(\text{pred}, \text{reg}))] \\
\text{buf}f_4 & = \text{buf}f_3[i + 3 \mapsto \text{jmp}(\text{rmp}, n')] \\
\sigma' & = \sigma[i \mapsto \text{pop}] \\
\text{fetch} & \quad (p, \mu, n, \text{buf}, \sigma) \xrightarrow{\text{fetch}} (p, \mu, n', \text{buf}, \alpha') \\
\text{ret-reire} & \quad \text{MIN}(\text{buf}) = i \\
\text{buf}(i) & = \text{fence} \\
\text{buf}' & = \text{buf} \setminus \text{buf}(i) \\
\text{reire} & \quad (p, \mu, n, \text{buf}, \alpha') \xrightarrow{\text{reire}} (p, \mu, n, \text{buf}')
\end{align*}
\]

The fence instruction uses \text{simple-fetch} as its fetch rule, and its rule for retire only removes the instruction from the buffer. The

**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 \( \text{top}(\sigma) = \bot \), 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 \text{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.

\[
\begin{align*}
\text{fence-reire} & \quad \text{MIN}(\text{buf}) = i \\
\text{buf}(i) & = \text{fence} \\
\text{buf}' & = \text{buf} \setminus \text{buf}(i) \\
\text{reire} & \quad (p, \mu, n, \text{buf}, \alpha') \xrightarrow{\text{reire}} (p, \mu, n, \text{buf}')
\end{align*}
\]

The 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 \( \forall j < i : \text{buf}(j) \neq \text{fence} \). This condition ensures that as long as a fence instruction remains un-retired in \( \text{buf} \), 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.

**Experiences.** The example in Figure 10 shows how placing a fence instruction just after the \text{br} instruction prevents the Spectre v1 attack from Figure 5. The fence in this example prevents the load instructions at \( 2 \) and \( 3 \) from executing and forces the \text{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 $\mu$ and $\rho$, 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 $C_{\varnothing} \xrightarrow{N} C'$ 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 $C_{\varnothing} \xrightarrow{N}_{\text{seq}} C'$ if we execute sequentially.

**Consistency.** To prove consistency of our semantics, we define the relation $C_1 \simeq C_2$ which relates configurations whose physical components are equal. That is, $C_1 = C_2$ if $\forall \alpha \in O_1 : \alpha \notin o \rightarrow C_1.\rho = C_2.\rho \wedge C_1.\mu = C_2.\mu$.

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 $C_{\varnothing} \xrightarrow{N} C_1$, then $C_{\varnothing} \xrightarrow{N}_{\text{seq}} C_2$ and $C_1 \simeq C_2$.

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 $\ell$ appears nowhere in the expression for $o$, and prove the following:

**Theorem 4.2 (Label stability).** Let $\ell$ be a label in the lattice $\mathcal{L}$. If $C_{\varnothing} \xrightarrow{N}_{\text{seq}} C_1$ and $\forall o \in O_1 : \ell \notin o$, then $C_{\varnothing} \xrightarrow{N}_{\text{seq}} C_2$ 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 $\cong_{\text{pub}}$ iff for every $C'$ such that $C \cong_{\text{pub}} C'$:

$$C_{\varnothing} \xrightarrow{N} C_1 \text{ iff } C'_{\varnothing} \xrightarrow{N} C'_1$$

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 $\cong_{\text{pub}}$ iff for every $C'$ such that $C \cong_{\text{pub}} C'$:

$$C_{\varnothing} \xrightarrow{N}_{\text{seq}} C_1 \text{ iff } C'_{\varnothing} \xrightarrow{N}_{\text{seq}} C'_1$$

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,
which exist even on the non-speculative path. Therefore, we have violations. Nonetheless, despite its approximations, Pitchfork still finds violations in every case, but reports no violations when executing non-speculatively. Purely sequential semantics can only model Spectre v1 attacks. Though they give examples of a variety of Spectre attacks, their three-stage issue/execute/commit system, which is analogous to 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 speculative 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...
Towards Constant-Time Foundations for the New Spectre Era

Data Bounce. Although Data Bounce, like Spec-TE-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

[12] Jann Horn. 2018. Issue 1528: speculative execution, variant 4: speculative 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.
A INDIRECT JUMP SEMANTICS

Semantics. The semantics for jmpi are given below:

\[ \mu(n) = \text{jmpi}(\pi) \]
\[ i = \text{MAX}(\text{buf}i) + 1 \]
\[ \text{buf}' = \text{buf}[i \mapsto \text{jmpi}(\pi, n')] \]
\[ (p, \mu, n, \text{buf}, \sigma) \overset{\text{fetch} i}{\rightarrow} (p, \mu, n', \text{buf}') \]

JMPI-EXECUTE-CORRECT

\[ \forall j < i : \text{buf}(j) \neq \text{fence} \quad (\text{buf} + i + j)(\pi) = \overline{\ell} \]
\[ \ell = \overline{\ell} \quad [addr(\overline{\ell})] = n_0 \quad \text{buf}' = \text{buf}[i \mapsto \text{jump } n_0] \]
\[ (p, \mu, n, \text{buf}, \sigma) \overset{\text{jump } n_0}{\rightarrow} (p, \mu, n, \text{buf}') \]

JMPI-EXECUTE-INCORRECT

\[ \forall j < i : \text{buf}(j) \neq \text{fence} \quad (\text{buf} + i + j)(\pi) = \overline{\ell} \]
\[ [addr(\overline{\ell})] = n' \neq n_0 \quad \text{buf}' = \text{buf}[j \mapsto i][i \mapsto \text{jump } n'] \]
\[ (p, \mu, n, \text{buf}, \sigma) \overset{\text{rollback, jump } n'}{\rightarrow} (p, \mu, n', \text{buf}') \]

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 \( r_x \) at program point \( i \), the program makes an indirect jump. An adversary can mistrain the predictor to send execution to \( n ' \) instead of the intended branch target, where the indirect jump to the instruction after the fence 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 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.

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 \( \text{top}(\sigma) \) 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 \text{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 \( \text{top}(\sigma) \) always produce an according value, and never producing \( \\bot \).

C FULL PROOFS

C.1 Consistency

Lemma C.1 (Determinism). If \( C, \sigma \xrightarrow{d} C' \) and \( C, \sigma' \xrightarrow{d} C'' \) then \( C' = C'' \) and \( \sigma' = \sigma'' \).

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 \( |C, \text{buf}| = 0 \).

![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.](image-url)
Towards Constant-Time Foundations for the New Spectre Era

<table>
<thead>
<tr>
<th>Registers</th>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_b )</td>
<td>8pub</td>
</tr>
<tr>
<td>( r_{sp} )</td>
<td>7Cpub</td>
</tr>
<tr>
<td>( n )</td>
<td>( \mu(n) )</td>
</tr>
<tr>
<td>3</td>
<td>call(5, 4)</td>
</tr>
<tr>
<td>4</td>
<td>fence 4</td>
</tr>
<tr>
<td>5</td>
<td>( r_d = \text{op}(\text{addr}, [12, r_b]) )</td>
</tr>
<tr>
<td>6</td>
<td>store(( r_d, [r_{sp}] ))</td>
</tr>
<tr>
<td>7</td>
<td>ret</td>
</tr>
</tbody>
</table>

Effect of successive fetch directives

\[
\begin{array}{|c|c|c|}
\hline
\text{Directive} & \text{Effect on buf} & \text{Leakage} \\
\hline
\text{execute 4} & 3 \rightarrow \text{call} & 3 \rightarrow \text{push 4} \\
\text{execute 5} & 4 \rightarrow \text{push 4} & 4 \rightarrow \text{ret} \\
\text{execute 6} & 5 \rightarrow \text{store}(4, [r_{sp}] ) & 5 \rightarrow \text{ret} \\
\text{execute 7} & 6 \rightarrow \text{store}(r_d, [r_{sp}] ) & 6 \rightarrow \text{ret} \\
\text{execute 8} & 7 \rightarrow \text{store}(7B, [r_{sp}] ) & 7 \rightarrow \text{ret} \\
\hline
\end{array}
\]

**Figure 12:** Example demonstrating “retpoline” mitigation against Spectre v2 attack. The program is able to jump to program point \( 12 + r_b = 20 \) 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).** \( C_{\|D} \) 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 \( C_{\|D} \) if we execute sequentially.

**Lemma C.5 (Sequential consistency).** If \( C_{\|D_1} \) and \( C_{\|D_2} \) are sequential and \( C_{\|D_1 \cap D_2} \) 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 \( C_1 \) and \( C_2 \) 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 \( N \).

By our induction hypothesis, we know \( C_{\|D_1} \) and \( C_{\|D_2} \) for some \( C' \). Since \( D_1' \) (resp. \( D_2' \)) is sequential and \( |C'. \text{buf}| = 0 \), the first directive in \( D_1' \) (resp. \( D_2' \)) must be a fetch directive. Furthermore, \( \text{cond-exec-correct} \) \( C_1 \) and \( \text{cond-exec-correct} \) \( C_2 \).

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

- For \( \text{op} \), the only valid sequence of directives is (fetch, execute, ret), where \( i \) is the sole valid index in the buffer. Similarly for fence, with the sequence (fetch, ret).
- 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, ret).
- 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 \( \text{cond-exec-correct} \) result in the same configuration regardless of the initial guess, as the br is the only instruction in the buffer. Similarly for jmp.
- 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 \( C \xleftarrow{d} C' \):

- \( d \) is a fetch directive, and would fetch an instruction into index \( i \), \( \text{buf} \).
- \( d \) is an execute directive, and would execute the instruction at index \( i \), \( \text{buf} \).
- \( d \) is a retire directive, and would retire the instruction at index \( i \), \( \text{buf} \).

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 \( C_{\|D} \), we say that \( D \) contains **misspeculated steps** if there exists \( d \in D \) such that \( D' = D \setminus d \) and \( C_{\|D} \) is not equivalent to \( C' \).

Given an execution \( C_{\|D} \), we can create an alternate schedule \( D' \) without any rollbacks by removing all misspeculated steps. Note that sequential schedules have no misspeculated steps 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 \( C_{\|D} \), then \( C_{\|D'} \) and \( C_1 = C_2 \). Furthermore, if \( C_1 \) is terminal then \( C_1 = C_2 \).

\(^{3}\text{Sequential schedules may still misspeculate on conditional branches but the rollback does not imply removal of any reorder buffer instructions as defined in Definition C.6.}\)
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 $D'_1$.

By induction hypothesis, we have $\exists D'_2$ such that $C(i) = O_p(i)$ for all $i > 0$.

Proof. By Theorem C.7, there exists $D'_1$ such that executing with $C$ produces $C_1 = C_1'$ (resp. $C_1'' = C_1'$). Similarly, there exists $D''_1$ such that executes $C_2 = C_2'$ (resp. $C_2'' = C_2'$). By Theorem C.5, we have $C_1 = C_2$. Thus $C_1 = C_2$ (resp. $C_1' = C_2'$).

C.2 Security

Theorem 4.2 (Label stability). Let $\ell$ be a label in the lattice $L$. If $C(i) = C(i)$ for all $i$, then $C(i) = C(i)$ for all $i$.

Proof. Let $D'_1$ be the schedule given by removing all misspeculated steps from $D_1$. The corresponding trace $O_p'$ is a subsequence of $O_p$, and hence $O_p' \subseteq O_p$. We thus assume 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 can conclude 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 $O_p'$ is the subsequence of $O_p$ 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 $a_{\ell}$ 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 $D'_1$ by removing all misspeculations from the induction hypothesis and $D''_1$ by removing all misspeculations from the subsequence of $D''_1$ to 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 $D = D'_1 \parallel D''_1$ is $O'_2 \parallel O'_2$. Since $O'_2$ satisfies the label stability property via the induction hypothesis, we have $\forall o \in O'_2 \parallel O'_2$.

By letting $\ell$ 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 $C' \approx_{pub} C$ and $C' \approx_{pub} C_1$ where $C_1 \approx_{pub} C'$ and $O = O'$. By Theorem C.7, we know there exist sequential executions such that $C_{seq} \parallel C_{seq}$ and $C' \approx_{seq} C'$. Note that the two sequential schedules need not be the same.

If $C$ 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 $C_1 = C'_1$ to get $C_2 = C'_2$.

To prove the trace property $O_{seq} = O_{seq}$, we note that if $O_{seq} \neq O_{seq}$, then $C_1 \approx_{pub} C_2$, and it must be the case that there exists some $o \in O_{seq}$ such that $o \notin O_{seq}$. 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 \textit{Pitchfork}(C, k) to be the set of Pitchfork schedules for $C$ and $k$. For the purposes of the recursive construction, we also define \textit{PitchforkPrefix}(C, k, n) as the set of all schedules of length $n$ directives which are prefixes of any schedules in the set \textit{Pitchfork}(C, k). As a base case, \textit{PitchforkPrefix}(C, k, 0) = \{\} 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 \( n \)th directive which Pitchfork will examine. That is, given each possible schedule \( D \) of length \( n-1 \), we compute the possible choices for the \( n \)th 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 \( C \cup D \), that is, the state after executing the \( n - 1 \) directives in \( D \). Further let \( y \) be MAX(\( C'.buf \)), and let \( z \) be 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 \( y \) : value\}—that is, Pitchfork considers only one possibility for the \( n \)th 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.