AUTOMATICALLY ELIMINATING SPECULATIVE LEAKS WITH BLADE

ANONYMOUS AUTHOR(S)

Abstract We introduce BLADE, a new approach to automatically and efficiently synthesizing provably correct repairs for transient execution vulnerabilities like Spectre. BLADE is built on the insight that to stop speculative execution attacks, it suffices to *cut* the dataflow from expressions that speculatively introduce secrets (*sources*) to those that leak them through the cache (*sinks*), rather than prohibiting speculation altogether. We formalize this insight in a *static type sytem* that *(1)* types each expression as either *transient*, *i.e.,* possibly containing speculative secrets or as being *stable*, and *(2)* prohibits speculative leaks by requiring that all *sink* expressions are stable. We introduce `protect`, a new abstract primitive for fine grained speculation control that can be implemented via existing architectural mechanisms, and show how our type system can automatically synthesize a *minimal* number of `protect` calls needed to ensure the program is secure. We evaluate BLADE by using it to repair several verified, yet vulnerable WebAssembly implementations of cryptographic primitives. BLADE can fix existing programs that leak via speculation *automatically*, without user intervention, and *efficiently* using two orders of magnitude fewer fences than would be added by existing compilers, thereby and ensuring security with minimal performance overhead.

1 Introduction

Implementing secure cryptographic algorithms is hard. The code must not only be functionally correct and memory safe, it must avoid divulging secrets indirectly through side channels like control-flow, memory-access patterns, or execution time. Consequently, much recent work focuses on how to ensure implementations do not leak secrets *e.g.,* via type systems [12, 39], verification[4], and program transformations [6].

Unfortunately, these efforts are foiled by speculative execution. Even if secrets are closely controlled via guards and access checks, the processor can simply ignore those checks when executing speculatively. An attacker can exploit this to leak secrets in turn.

In principle, memory fences block speculation, and hence, offer a way to recover the original security guarantees. In practice, however, fences pose a confounding dilemma. Programmers can either rely on heuristic approaches for inserting fences [37], but then forgo guarantees about the absence of side-channels. Alternatively, they can recover security guarantees by conservatively inserting fences after every load, but endure the huge performance costs.

In this paper, we introduce BLADE, a new approach to automatically, provably and efficiently eliminate speculation-based leakage. BLADE is based on the key insight that to prevent leaking data via speculative execution, it is unnecessary to stop *all* speculation as done by traditional memory fences. Instead, it suffices to *cut* the data flow from expressions (sources) that speculatively introduce secrets to those that leak them through the cache (sinks). We develop this insight into an automatic enforcement algorithm via four contributions.

1. A Semantics for Speculation. Our first contribution is a formal operational semantics for a simple While language that precisely captures how speculation can occur and what an attacker can observe via speculation (§ 3). To prevent leakage, we propose and formalize the semantics of an abstract primitive called `protect` that does not exist in today's hardware but captures the essence of several primitives proposed in recent work [2, 32]. Furthermore, this primitive can be implemented in software *e.g.,* via *speculative load hardening* [30]. Crucially, and in contrast to a regular fence which stops *all* speculation, `protect` only stops speculation for a given *variable*. For example x:=**protect**(e) ensures that e's value is only assigned to x *after* e has been assigned its *stable*, non-speculative value.

2. A Type System for Speculation. Our second contribution is an approach to conservatively approximating the dynamic semantics of speculation via a *static type sytem* that types each expression as either transient (**T**), *i.e.,* expressions that may contain speculative secrets, or stable (**S**), *i.e.,* those that cannot (§ 4.1). Our system prohibits speculative leaks by requiring that all *sink* expressions that can influence intrinsic attacker visible behavior (*e.g.,* cache addresses) are typed as stable. We connect the static and dynamic semantics by proving that well-typed programs are indeed secure, *i.e.,* satisfy a correctness condition called speculative non-interference [17] which states that the program does not leak under speculative execution more than it would under sequential execution.

3. Automatic Protection. Existing programs that are free of `protect` statements are likely insecure under speculation and will be rejected by our type system. Thus, our third contribution is an algorithm that automatically synthesizes a *minimal* number of `protect` statements to ensure that the program satisfies speculative non-interference. To this end, we extend the type checker to construct a *def-use graph* that captures the data-flow between program expressions. A *cut-set* in the graph is a set of variables whose removal eliminates

```
1  void SHA2_update_last(int *input_len, ...)
2  {
3    if (! valid(input_len)) { ... }
4    int len = *input_len;
5    int *dst3 = ... + len;
6    _mm_lfence();
7    int *dst3_safe = protect(.. + len);
8    ...
9    *dst3_safe = pad;
10   ...
11 }
```

Figure 1. Code fragment from the HACL* SHA2 implementation, containing a potential speculative execution vulnerability that leaks *explicitly* through the cache by writing memory at a secret-tainted address (line 9). A naive patch is shown is shown in **red**, the patch computed by BLADE is shown in **green**.

all paths from secret-sources to observable-sinks. We show that inserting a `protect` statement for each variable in a cut-set suffices to yield a program that is well-typed, and hence, secure with respect to speculation (§5.3). Happily, finding such cuts is an instance of the classic max-flow/min-cut problem, so existing polynomial time algorithms let us efficiently synthesize `protect` statements that resolve the dilemma of enforcing security with minimal performance overhead.

4. Evaluation. Our final contribution is an implementation of our method in a tool called BLADE, and an evaluation using BLADE to repair verified yet vulnerable (to transient execution attacks) programs: the WebAssembly implementations of the signal messaging Protocol and its respective cryptographic libraries [29], and a number of verified cryptographic algorithms from [38] (§ 6). Our evaluation shows that BLADE can automatically compute fixes for existing programs. Compared to an existing fully automatic protection as implemented in existing compilers (notably Clang), BLADE inserts two orders of magnitude fewer fences and thus imposes negligible performance overhead.

2 Overview

In this section, we present two potential speculative execution vulnerabilities in HACL*— a verified cryptographic library — that were discovered by BLADE and discuss how BLADE repairs the vulnerabilities by inserting `protect` statements. We then show how BLADE computes the repairs via our minimal fence inference algorithm and finally how BLADE proves that the repairs are indeed correct, via our transient-flow type system.

2.1 Two Speculation Bugs and Their Fixes

Figure 1 shows a code fragment from a function in the implementation of the SHA2 hash in HACL*. Though BLADE operates on WebAssembly, we present equivalent simplified C code for readability. The function takes as input a pointer `input_len`, validates the input (line 3), loads from memory the public length of the hash (line 4), calculates a target address `dst3` (line 5), and finally pads the buffer pointed to by `dst3` (line 9).

1. Leaking Through a Memory Write. During normal, sequential execution this code is not a problem: the function validates the input to prevent classic buffer overflows vulnerabilities. However, an attacker can exploit the function to leak senstive data during speculation. To do this, the attacker first has to modify the value that the pointer `input_len` holds during speculation. Since `input_len` is a function parameter, this can be achieved *e.g.,* by calling the function repeatedly with legitimate addresses, training the branch predictor to predict the next input to be valid. After (mis)training the branch predictor, the attacker manipulates `input_len` to point to an address containing secret data (e.g., the secret key used by the hash function) and calls the function again, this time with an invalid pointer. As a result of the mistraining, the branch predictor causes the processor to skip validation and erroneously load the secret into `len`, which in turn, is used to calculate pointer `dst3`. The buffer pointed to by `dst3` is then written in line 9, completing the attack. Even though pointer `dst3` is incorrect due to misprediction and the write will therefore be squashed, its side-effects persist, and therefore remain visible to the attacker. The attacker can then extract the target address — and thereby the secret via cache timing measurements [16].

Preventing the Attack: Memory Fences. Since the attack exploits the fact that input validation is speculatively skipped, we can prevent it by making sure that the buffer in line 9 is not written until the input has been validated. To mitigate these class of attacks, Intel [19] and AMD [5] recommend inserting a speculation barrier after critical validation check-points. Following this strategy, we would place a *memory fence* on line 6. This fence stops all speculative execution past the fence, *i.e.,* no statements after the fence are executed until all previous statements (including input validation) have been resolved. While the effects of the fence prevent the attack, they are more restrictive than necessary and incur high performance cost [33].

Preventing the Attack Efficiently. We propose an alternative way to stop speculation from reaching the write in line 9 through a new primitive called `protect`. Rather than eliminate *all* speculation, `protect` only stops speculation along *a particular data-path.* We use `protect` to patch the program in line 7. Instead of assigning pointer `dst3` directly as in line 5, the expression that computes the address is guarded by a `protect` statement. This ensures that the value assigned

```
1  void SHA2_update_last(int *input_len,...)
2   {
3     if (! valid(input_len)) { ... }
4     int len = *(input_len);
5     ...
6     int len_safe = protect(*input_len)
7     for ( i = 0; i < len_safe + ...)
8       dst2[i] = 0;
9     ...
10  }
```

Figure 2. SHA2 code fragment containing a potential speculative execution vulnerability that leaks *implicitly* through a control-flow dependency.

to `dst3_safe` is always guaranteed to use `len`'s final, non-speculative value. Therefore, writing to `dst3_safe` in line 9 prevents any invalid secret-*tainted* address from speculatively reaching the store, where it could be leaked to the attacker.

The `protect` primitive offers an abstract interface for fine grained control of speculation. There are a number of possible implementations for this interface. For example, `protect` could be implemented in hardware. While unfortunately, today's hardware does not offer an equivalent instruction to `protect`, similar functionalities have been proposed in recent work [2, 32]. Alternatively, `protect` can be implemented in software (a similar proposal has been made in [30]). In general, `protect` can be implemented through a fence instruction. However, better solutions exist for reading arrays. For example, Speculative Load Hardening (SLH), a mitigation deployed in the code generated by Clang [10], stalls individual array reads until the corresponding bounds-check condition gets resolved. We model software implementations of `protect` through a restricted primitive called `safe_read`, which can only be applied to array reads. We then formalize an implementation of `safe_read` via SLH in the supplementary material, and evaluate the number of `protect` and `safe_read` needed to patch HACL* and their overhead in Section 6.

2. Leaking Through a Control-Flow Dependency. Figure 2 shows a code fragment taken from the same function as in Figure 1. The code contains a second potential vulnerability, but in contrast to Figure 1 the vulnerability leaks secrets *implicitly*, through a control-flow dependency.

The function reads from memory a (public) integer `len` (line 4), which determines the number of initialization rounds in the condition of the for-loop (line 7). Like the previous vulnerability, the function is harmless under sequential execution, but leaks under speculation. As before, the attacker manipulates the pointer `input_len` to point to a secret after mistraining the branch predictor to skip validation. But instead of leaking the secret directly through the data cache, they can leak the value indirectly through a control-flow dependency, e.g., via the instruction cache and non-secret dependent lines of the data cache. In particular, the secret determines how often the initialization loop (line 7) is executed during speculation, and therefore an attacker can make secret dependent observations via instruction- and data-cache timing attacks. Like the previous vulnerability, this vulnerability can be fixed via the `protect` primitive, as shown in lines 6 and 7.

2.2 Computing Fixes Via Minimal Fence Inference

BLADE automatically infers the placement of these `protect` statements. We illustrate this process using a simple running example Ex1 shown in Figure 3. The code reads two values from an array ($x := a[i_1]$ and $y := a[i_2]$), adds them ($z := x + y$), and indexes another array with the result ($w := b[z]$). We assume that all array operations are implicitly bounds-checked and thus no explicit validation code is needed.

Like the examples above, Ex1 contains a speculative execution vulnerability: the array reads may skip their bounds check and so x and y can contain transient secrets (*i.e.,* secrets introduced by misspeculation). This secret data then flows to z, and finally leaks through the data cache by the array read $b[z]$.

Def-Use Graph. To secure the program, we need to *cut the dataflow* between the array reads which could introduce *transient* secret values into the program, and the index in the array read where they are leaked through the cache. For this, we first build a def-use graph whose nodes and directed edges capture the data dependencies between the expressions and variables of a program. For example, consider the def-use graph of program Ex1 in Figure 4. In the graph, the edge $x \longrightarrow x+y$ indicates that x is used to compute $x+y$.[1] To track how transient values propagate in the def-use graph, we extend the graph with the special circle node **T**, which represents the *source* of *transient* values of the program. Since reading memory creates transient values, we connect the **T** node to all nodes containing expressions that explicitly read memory, e.g., $\mathbf{T} \longrightarrow a[i_1]$. Following the data dependencies along the edges of the def-use graph, we can see that node **T** is transitively connected to node z, which indicates that z can contain transient data at run-time. To detect insecure uses of transient values, we then extend the graph with the special circle node **S**, which represents the *sink* of *stable* (i.e., non-transient) values of a program. Intuitively, this node draws all the values of a program that *must* be stable to avoid transient execution attacks. Therefore, we connect all expression used as array indices in the program to the **S** node, e.g., $z \longrightarrow \mathbf{S}$. The fact that the graph in Figure 3 contains a *path* from **T** to **S** indicates that transient data flows through data dependencies into (what should be) a stable index expression and thus the program is insecure.

Cutting the Dataflow. In order to make the program safe, we need to *cut* the data-flow between **T** and **S** by introducing

[1] To avoid ambiguities in the graph, we assume that each variable is assigned at most *once, i.e.,* the code is in static single assignment form.

$x := a[i_1]$ $\quad x := \textbf{protect}(a[i_1])$
$y := a[i_2]$ $\quad y := \textbf{protect}(a[i_2])$
$z := x + y$ $\quad z := \textbf{protect}(x + y)$
$w := b[z]$

Figure 3. Ex1: Running Example. The optimal patch computed by BLADE is shown in **green**. A sub-optimal patch is shown in **orange**.

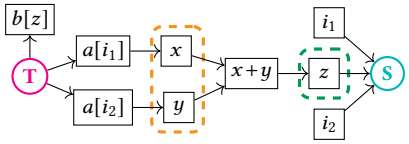

Figure 4. Def-use Graph of Ex1. We omit some irrelevant edges for readability. The Figure contains two choices of cut-sets, shown as dashed lines. The left cut requires removing two nodes and thus, inserting two `protect` statements. The right cut shows a minimal solution, which only requires removing a single node.

as few `protect` statements as necessary. This problem can be equivalently restated as follows: find a *minimal cut-set*, *i.e.,* a minimal set of variables, such that removing the variables from the graph eliminates all paths from **T** from **S**. Each choice of cut-set defines a way to repair the program: simply add a `protect` statement for each variable in the set. Figure 4 contains two choices of cut-sets, shown as dotted lines. The cut-set on the left requires two protect statements, for variables x and y respectively, corresponding to the **orange** patch in Figure 3. The cut-set on the right is minimal, it requires only a single protect, for variable z, and corresponds to the **green** patch in Figure 3. In general, the a minimal cut-set can be computed as a solution to the Min-Cut/Max-Flow problem, for which efficient polynomial-time algorithms exist [1].

2.3 Proving Correctness Via Transient-Flow Types

To formalize and verify the correctness of the patch computed by cutting the def-use graph, we define a transient-flow type system and construct the def-use graph for a given program from the type-constraints generated during type inference.

Typing Judgement. The type system statically assigns a transient-flow type to each variable: a variable is typed as *transient* (written as **T**), if it can contain transient data (i.e., potential secrets) at run-time, and as *stable* (written as **S**), otherwise. Given a typing environment Γ which assigns a transient flow type to each variable, and a command c, the type system defines a judgement $\Gamma \vdash c$ saying that c is free of speculative execution bugs. The type system enforces that transient expressions may not be used in positions that may leak their value by affecting memory reads and writes, *e.g.,* they may not be used as array indices and in loop conditions. Additionally, it requires that transient expressions may not be assigned to stable variables, except through the use of `protect`. To show that our type system indeed prevents speculative execution attacks, we define a semantics for speculative execution of a while language (Section 3) and prove that well-typed programs do not leak speculatively more than *sequentially*, that is by executing their statements in-order and without speculation (see Section 5).

Type Inference. Given an input program, we construct the corresponding def-use graph by collecting the type constraints generated during type inference. Type inference is formalized by a typing-inference judgment $\Gamma, \text{Prot} \vdash c \Rightarrow k$, which extends the typing judgment from above with (1) a set of protected variables Prot (the *cut-set*), and (2) a set of type-constraints k (the *def-use graph*). At a high level, type inference has 3 steps: *(i)* generate a set of constraints under an initial typing environment and protected set that allow any program to type-check, *(ii)* construct the def-use graph from the constraints and find a cut-set, and *(iii)* compute the resulting typing environment. To characterize the security of a still *unrepaired* program after type inference, we define a typing judgment $\Gamma, \text{Prot} \vdash c$, where unprotected variables are explicitly accounted for in the Prot set.[2] Intuitively, the program is secure if we promise to insert a `protect` statement for each variable in Prot.

To repair programs, we simply honor the promise of inserting protect statements for each for each variable in the protected set of the typing judgment obtained above. Once repaired, the program type checks under an empty protected set and with the same typing environment.

2.4 Attacker Model

Before moving to the details of our semantics and transient type system, we discuss the attacker model considered in this work. The attacker runs cryptographic code on a speculative out-of-order processor and, as usual, can choose the values of public inputs and observe public outputs, but may not read secret data (e.g., cryptographic keys) in registers and memory. Additionally, the attacker can influence how programs are speculatively executed through the branch predictor and choose the instructions execution order in the processor pipeline. The effects of these actions are observable through the cache and are otherwise invisible at the ISA level. In particular, while programs run, the attacker can take precise timing measurements through the data- and instruction-cache with a cache-line granularity, which may disclose secret data covertly. These features allow the attacker to mount Spectre-PHT [20, 21] and Spectre-STL [9] attacks and leak data through FLUSH+RELOAD [43] and PRIME+PROBE [34] cache side-channels attacks. We do not consider speculative attacks that rely on the Return Stack Buffer (e.g., Ret2Spec [25]

[2] The judgment $\Gamma \vdash c$ is just a short-hand for $\Gamma, \varnothing \vdash c$.

Value	v	::=	$n \mid b \mid a$
Expr.	e	::=	$v \mid x \mid e_1 + e_2 \mid e_1 \leqslant e_2$
			$\mid \mathit{length}(e) \mid \mathit{base}(e)$
Rhs.	r	::=	$e \mid *e \mid e[e]$
Cmd.	c	::=	$\mathbf{skip} \mid x := r \mid *e = e \mid e_1[e_2] := e_3$
			$\mid \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2$
			$\mid \mathbf{while}\ e\ \mathbf{do}\ c \mid \mathbf{fail} \mid c_1; c_2$
			$\mid x := \mathbf{stable\_read}(e_1, e_2)$
			$\mid x := \mathbf{protect}(r)$

Figure 5. Surface Syntax.

and [22]) or the Branch Target Buffer (Spectre-BTB [21]). We similarly do not consider attacks that do not use the cache to exfiltrate data, e.g., port contention (SMoTherSpectre [7]) and Meltdown attacks [9, 24], since hardware fixes address them.

3 A Semantics for Speculation

We now formalize the concepts presented in the overview. We start by giving a formal semantics for a while language with speculative execution. Figure 5 presents the language's surface syntax. Values consist of Booleans b, pointers n represented as natural numbers, and arrays a. Array length and base address are given by functions $\mathit{length}(\cdot)$ and $\mathit{base}(\cdot)$. In addition to variable assignments, pointer dereferences, array stores, conditionals and loops, our language features two special commands that help prevent transient execution attacks. Command $x := \mathbf{protect}(r)$ evaluates r and assigns its value to x, only after the value is *stable* (i.e., non-transient). Command $x := \mathbf{stable\_read}(e_1, e_2)$ is a restricted version of $\mathbf{protect}(\cdot)$ that only applies to array reads (see Section 3.4) Lastly, **fail** triggers a memory violation error (caused by reading or writing an array out-of-bounds) and aborts the program.

Processor Instructions. Our semantics translates the surface syntax into an abstract set of processor instructions shown in Figure 6. Our processor instructions do not contain branching, they represent a *single* predicted path through the control flow. The prediction choices are represented by a sequence of *guard* instructions representing pending branch points. Guard instructions have form $\mathbf{guard}(e^b, cs, p)$, which records the branch condition e, its predicted truth value b and a unique guard identifier p, used in our security analysis (Section 5). Each guard attests the fact that the current execution is valid only if the branch condition gets resolved as predicted. In order to enable a roll-back in case of a misprediction, guards additionally record the set of commands cs along the alternative branch.

Directives and Observations. Instructions do not have to be executed in sequence, they can be executed in any order, enabling out-of-order execution. We use a simple three stage processor pipeline: the execution of each instruction is split into **fetch**, **exec**, and **retire**. We do not fix the order in which

Instr.	i	::=	$\mathbf{nop} \mid x := e \mid x := \mathbf{load}(e)$
			$\mid \mathbf{store}(e_1, e_2) \mid x := \mathbf{protect}(e)$
			$\mid \mathbf{guard}(e^b, cs, p) \mid \mathbf{fail}$
Dir.	d	::=	$\mathbf{fetch} \mid \mathbf{fetch}\ b \mid \mathbf{exec}\ n$
			$\mid \mathbf{retire}$
Obs.	o	::=	$\epsilon \mid \mathbf{load}(n, ps) \mid \mathbf{store}(n, ps)$
			$\mid \mathbf{fail} \mid \mathbf{rollback}(p)$
Prediction		b	$\in \{\mathbf{true}, \mathbf{false}\}$
Guard Id.		p	$\in \mathbb{N}$
Reorder Buffer		is	$::= i : is \mid [\,]$
Cmd Stack		cs	$::= c : cs \mid [\,]$
Memory Store		μ	$\in \mathbb{N} \rightharpoonup \mathit{Value}$
Variables Map		ρ	$\in \mathit{Var} \rightharpoonup \mathit{Value}$
Configuration		C	$::= \langle is, cs, \mu, \rho \rangle$

Figure 6. Processor Syntax.

instructions, and their individual stages are executed, nor do we supply a model of the branch predictor to decide which control flow path to follow. Instead, we let the attacker supply those decisions through a set of *directives* [11] shown in Fig. 6. For example, directive **fetch true** fetches the **true** branch of a conditional and **exec** n executes the nth instruction in the reorder buffer. Executing an instruction generates an *observation* (Fig. 6) which records attacker observable behavior. Observations include *speculative* memory reads and writes (i.e., $\mathbf{load}(n, ps)$ and $\mathbf{store}(n, ps)$ issued while guards ps are pending), rollbacks (i.e., $\mathbf{rollback}(p)$ due to misspeculation of guard p), and memory violations (**fail**). Most instructions generate the empty observation ϵ.

Configurations and Reduction Relation. We formally specify our semantics as a reduction relation between processor configurations. A configuration $\langle is, cs, \mu, \rho \rangle$ consists of a queue of in-flight instructions is called the reorder buffer, a stack of commands cs, a memory μ, and map from variables to values ρ. A reduction step $C \xrightarrow{d}_o C'$ denotes that, under directive d, configuration C is transformed into C' and generates observation o. To execute a program c with initial memory μ and variable map ρ, the processor initializes the configuration with an empty reorder buffer and inserts the program into the command stack, i.e., $\langle [\,], [\,c\,], \mu, \rho \rangle$. Then, the execution proceeds until both the reorder buffer and the stack in the configuration are empty , i.e., we reach a configuration of the form $\langle [\,], [\,], \mu', \rho' \rangle$, for some final memory store μ' and variable map ρ'.

We now discuss the semantics rules of each execution stage and then those for our security primitives.

3.1 Fetch Stage

The fetch stage flattens the input command into a sequence of instructions which it stores in the reorder buffer. Figure 7 presents selected rules; the remaining rules are in Appendix A. Rule [Fetch-Seq] pops command $c_1; c_2$ from the commands

FETCH-SEQ

$$\langle is, (c_1;c_2):cs, \mu, \rho\rangle \xrightarrow{\textbf{fetch}}_{\epsilon} \langle is, c_1:c_2:cs, \mu, \rho\rangle$$

FETCH-ASGN

$$\langle is, x:=e:cs, \mu, \rho\rangle \xrightarrow{\textbf{fetch}}_{\epsilon} \langle is{+}{+}[x:=e], cs, \mu, \rho\rangle$$

FETCH-PTR-LOAD

$$\langle is, x:=*e:cs, \mu, \rho\rangle \xrightarrow{\textbf{fetch}}_{\epsilon} \langle is{+}{+}[x:=\textbf{load}(e)], cs, \mu, \rho\rangle$$

FETCH-ARRAY-LOAD

$$\frac{c = x:=e_1[e_2] \qquad e = e_2 < \mathit{length}(e_1) \qquad \mathrm{fresh}(p) \\ e' = \mathit{base}(e_1)+e_2 \qquad c' = \textbf{if } e \textbf{ then } x:=*e' \textbf{ else fail}}{\langle is, c:cs, \mu, \rho\rangle \xrightarrow{\textbf{fetch}}_{\epsilon} \langle is, c':cs, \mu, \rho\rangle}$$

FETCH-IF-TRUE

$$\frac{c = \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \\ \mathrm{fresh}(p) \qquad i = \textbf{guard}(e^{\textbf{true}}, c_2:cs, p)}{\langle is, c:cs, \mu, \rho\rangle \xrightarrow{\textbf{fetch true}}_{\epsilon} \langle is{+}{+}[i], c_1:cs, \mu, \rho\rangle}$$

Figure 7. Fetch stage (selected rules).

stack and pushes the two sub-commands for further processing. [FETCH-ASGN] pops an assignment from the commands stack and appends the corresponding processor instruction ($x := e$) at the end of the reorder buffer.[3] Rule [FETCH-PTR-LOAD] is similar and simply translates pointer dereferences to the corresponding load instruction. Arrays provide a memory-safe interface to read and write memory: the processor injects bounds-checks when fetching commands that read and write arrays. For example, rule [FETCH-LOAD-TRUE] expands command $x:=e_1[e_2]$ into the corresponding pointer dereference, but guards the command with a bounds-check condition. First, the rule generates the condition $e = e_2 < \mathit{length}(e_1)$ and calculates the address of the indexed element $e' = \mathit{base}(e_1)+e_2$. Then, it replaces the array read on the stack with command **if** e **then** $x := *e'$ **else fail** to abort the program and prevent the buffer overrun if the bounds check fails. Later, we show that speculative out-of-order execution can simply ignore the bounds check guard and cause the processor to transiently read memory at an invalid address. Rule [FETCH-IF-TRUE] fetches a conditional branch from the stack and, following the prediction provided in directive **fetch true**, speculates that the condition e will evaluate to **true**. Thus, the processor inserts the corresponding instruction $\textbf{guard}(e^{\textbf{true}}, c_2:cs, p)$ with a fresh guard identifier p in the reorder buffer and pushes the then-branch c_1 onto the stack cs. Importantly, the guard instruction stores the else-branch together with a copy of

[3] Notation $[i_1, \ldots, i_n]$ represents a list of n elements, $is_1 ++ is_2$ denotes list concatenation, and $|is|$ computes the length of the list is.

EXECUTE

$$\frac{|is_1| = n-1 \qquad \rho' = \phi(is_1, \rho) \qquad \langle is_1, i, is_2, cs\rangle \xrightarrow{(\mu, \rho', o)} \langle is', cs'\rangle}{\langle is_1{+}{+}[i]{+}{+}is_2, cs, \mu, \rho\rangle \xrightarrow{\textbf{exec } n}_{o} \langle is', cs', \mu, \rho\rangle}$$

EXEC-ASGN

$$\frac{i = (x:=e) \qquad v = [\![e]\!]^{\rho} \qquad i' = (x:=v)}{\langle is_1, i, is_2, cs\rangle \xrightarrow{(\mu, \rho, \epsilon)} \langle is_1{+}{+}[i']{+}{+}is_2, cs\rangle}$$

EXEC-BRANCH-OK

$$\frac{i = \textbf{guard}(e^b, cs', p) \qquad [\![e]\!]^{\rho} = b}{\langle is_1, i, is_2, cs\rangle \xrightarrow{(\mu, \rho, \epsilon)} \langle is_1{+}{+}[\textbf{nop}]{+}{+}is_2, cs\rangle}$$

EXEC-BRANCH-MISPREDICT

$$\frac{i = \textbf{guard}(e^b, cs', p) \qquad [\![e]\!]^{\rho} \neq b}{\langle is_1, i, is_2, cs\rangle \xrightarrow{(\mu, \rho, \textbf{rollback}(p))} \langle is_1, cs'\rangle}$$

EXEC-LOAD

$$\frac{i = (x:=\textbf{load}(e)) \qquad \textbf{store}(\_,\_) \notin is_1 \\ n = [\![e]\!]^{\rho} \qquad ps = (\!|is_1|\!) \qquad i' = (x:=\mu(n))}{\langle is_1, i, is_2, cs\rangle \xrightarrow{(\mu, \rho, \textbf{read}(n, ps))} \langle is_1{+}{+}[i']{+}{+}is_2, cs\rangle}$$

Figure 8. Execute stage (selected rules).

the current commands stack (*i.e.,* $c_2:cs$) as a rollback stack to restart the execution in case of misprediction.

3.2 Execute Stage

In the execute stage, the processor evaluates the operands of instructions in the reorder buffer and rolls back the program state whenever it detects a misprediction.

Transient Variable Map. To evaluate operands in the presence of out-of-order execution, we need to take into account how previous, possibly unresolved assignments in the reorder buffer affect the variable map. In particular, we need to ensure that an instruction cannot execute if it depends on a preceding assignment whose value is still unknown. To update variable map ρ with the pending assignments in reorder buffer is, we define a function $\phi(is, \rho)$, called the *transient variable map*. The function walks through the reorder buffer, registers each resolved assignment instruction ($x:=v$) in the variable map (through function update $\rho[x \mapsto v]$) and marks variables from pending assignments (*i.e.,* $x:=e$, $x:=\textbf{load}(e)$, and $x := \textbf{protect}(r)$) as *undefined* ($\rho[x \mapsto \bot]$), making their respective values unavailable to following instructions.

Execute Rule and Auxiliary Relation. Step rules for the reduction relation are shown in Figure 8. Rule [EXECUTE] executes the n-th instruction in the reorder buffer, following the directive **exec** n. For this, the rule splits the reorder buffer into

prefix is_1, n-th instrucion i and suffix is_2. Next, it computes the transient variable map $\phi(is_1, \rho)$ and executes a transition step under the new map using an auxiliary relation $\rightsquigarrow$. Notice that [Execute] does not update the store or the variable map (the transient map is simply discarded). These changes are performed later in the retire stage.

The rules for the auxiliary relation are shown in Fig. 8. The relation transforms a tuple $\langle is_1, i, is_2, cs\rangle$ consisting of prefix, suffix and current instruction i into a tuple $\langle is', cs'\rangle$ specifying the reorder buffer and command stack obtained by executing i. For example, rule [Exec-Asgn] evaluates the right-hand side of the assignment $x := e$ where $[\![e]\!]^\rho$ denotes the value of e under ρ. The premise $v = [\![e]\!]^\rho$ ensures that the expression is defined *i.e.,* it does not evaluate to $\perp$. Then, the rule substitutes the computed value into the assignment $(x := v)$, and reinserts the instruction back into its original position in the reorder buffer.

Guards and Rollback. Rules [Exec-Branch-Ok] and [Exec-Branch-Mispredict] resolve guard instructions. In rule [Exec-Branch-Ok], the predicted and computed value of the guard expression match, and the processor only has to replace the guard with a **nop**. In contrast, in rule [Exec-Branch-Mispredict] the predicted and computed value differ ($[\![e]\!]^\rho \neq b$). This causes the processor to revert the program state and issue a rollback observation. For the rollback, the processor discards the instructions *past* the guard (*i.e., is_2*) and substitutes the current commands stack cs with the rollback stack cs' which causes execution to revert to the alternative branch.

Loads. Rule [Exec-Load] executes a memory load. The rule computes the address $(n = [\![e]\!]^\rho)$, retrieves the value at that address from memory $(\mu(n))$ and rewrites the load into an assignment $(x := \mu(n))$. Inserting the assignment into the reorder buffer allows transiently forwarding the loaded value to later instructions. The premise $\mathbf{store}(\_,\_) \notin is_1$ prevents the processor from reading stale data from memory: if the load aliases with a preceding (but pending) store, ignoring the store would produce a stale read. To record that the load is issues *speculatively*, the observation $\mathbf{read}(n, ps)$ stores list ps containing the identifiers of the guards still pending in the reorder buffer. Function $\langle\!|is|\!\rangle$ simply extracts the identifiers of the guard instructions in the buffer is.

3.3 Retire Stage

The retire stage removes completed instructions from the reorder buffer and propagates their changes to variable map and memory store. While instructions are executed out-of-order, they are retired in-order to preserve the illusion of sequential execution to the user. Figure 9 presents the rules for the retire stage. Rule [Retire-Nop] removes **nop**. Rules [Retire-Asgn] and [Retire-Store] remove the resolved assignment $x := v$ and instruction $\mathbf{store}(n, v)$ from the reorder buffer and update the variable map $(\rho[x \mapsto v])$ and the memory store $(\mu[n \mapsto v])$

Retire-Nop

$$\langle \mathbf{nop} : is, cs, \mu, \rho\rangle \xrightarrow{\mathbf{retire}}_\epsilon \langle is, cs, \mu, \rho\rangle$$

Retire-Asgn

$$\langle x := v : is, cs, \mu, \rho\rangle \xrightarrow{\mathbf{retire}}_\epsilon \langle is, cs, \mu, \rho[x \mapsto v]\rangle$$

Retire-Store

$$\frac{i = \mathbf{store}(n, v)}{\langle i : is, cs, \mu, \rho\rangle \xrightarrow{\mathbf{retire}}_\epsilon \langle is, cs, \mu[n \mapsto v], \rho\rangle}$$

Retire-Fail

$$\langle \mathbf{fail} : is, cs, \mu, \rho\rangle \xrightarrow{\mathbf{retire}}_{\mathbf{fail}} \langle [\,], [\,], \mu, \rho\rangle$$

Figure 9. Retire stage.

respectively. Rule [Retire-Fail] aborts the program by emptying reorder buffer and command stack and generates a **fail** observation, simulating a processor raising an exception (e.g., a page fault).

We demonstrate how the attacker can leak a secret from program Ex1 (Fig. 3) in our model. First, the attacker instructs the processor to fetch all the instructions, suppling prediction **true** for all bounds-check conditions. Figure 10 shows the resulting buffer and how it evolves after each attacker directive, which instruct the processor to speculatively execute the load instructions and the assignment (but not the guard instructions). Memory μ and variable map ρ are shown on the right. Directive **exec** 4 transiently reads array a past its bound, at index 2, reading into the memory $(\mu(3) = 42)$ of secret array $s[0]$ and generates the corresponding observation. Finally, the processor forwards the values of x and y to compute their sum in the fifth instruction, $(z := 42)$, which is then used as an index in the last instruction and leaked to the attacker via observation $\mathbf{read}(42, [1,2,3])$.

3.4 Security Primitives

Next, we turn to the rules describing our security primitives.

Protect. Instruction $x := \mathbf{protect}(r)$ assigns the value of r, only after all previous **guard** instructions have been executed, *i.e.,* when the value has become stable and no more rollbacks are possible. Figure 11 formalizes this intuition. Rule [Fetch-Protect-Expr] fetches protect commands involving simple expressions $(x := \mathbf{protect}(e))$ and inserts the corresponding protect instruction in the reorder buffer. Rule [Fetch-Protect-Array] piggy-backs on the previous rule by splitting a protect of an array read $(x := \mathbf{protect}(e_1[e_2]))$ into a separate assignment of the array value $(x := e_1[e_2])$ and protect of the variable $(x := \mathbf{protect}(x))$. Rules [Exec-Protect$_1$] and [Exec-Protect$_2$] extend auxiliary relation $\rightsquigarrow$. Rule [Exec-Protect$_1$] evaluates the expression $(v = [\![e]\!]^\rho)$

	Reorder Buffer	exec 2	exec 4	exec 5	exec 7
1	$\mathbf{guard}((i_1 < \mathit{length}(a))^{\mathbf{true}}, [\mathbf{fail}], 1)$				
2	$x{:=}\mathbf{load}(\mathit{base}(a)+i_1)$	$x{:=}\mu(2)$			
3	$\mathbf{guard}((i_2 < \mathit{length}(a))^{\mathbf{true}}, [\mathbf{fail}], 2)$				
4	$y{:=}\mathbf{load}(\mathit{base}(a)+i_2)$		$y{:=}\mu(3)$		
5	$z{:=}x+y$			$z{:=}42$	
6	$\mathbf{guard}((z < \mathit{length}(b))^{\mathbf{true}}, [\mathbf{fail}], 3)$				
7	$w{:=}\mathbf{load}(\mathit{base}(b)+z)$				$w{:=}\mu(42)$
	Observations:	$\mathbf{read}(2,[1])$	$\mathbf{read}(3,[1,2])$	ϵ	$\mathbf{read}(42,[1,2,3])$

Memory Layout

$\mu(0)=0$	$b[0]$
$\mu(1)=0$	$a[0]$
$\mu(2)=0$	$a[1]$
$\mu(3)=42$	$s[0]$
...	...

Variable Map

$\rho(i_1)=1$
$\rho(i_1)=2$

Figure 10. Leaking execution of running example Ex1.

$$\textsc{Fetch-Protect-Array}\quad \frac{c=(x{:=}\mathbf{protect}(e_1[e_2])) \qquad c_1=(x{:=}e_1[e_2]) \qquad c_2=(x{:=}\mathbf{protect}(x))}{\langle is, c:cs, \mu, \rho\rangle \xrightarrow{\mathbf{fetch}}_{\epsilon} \langle is, c_1:c_2:cs, \mu, \rho\rangle}$$

$$\textsc{Fetch-Protect-Expr}\quad \frac{c=(x{:=}\mathbf{protect}(e)) \qquad i=(x{:=}\mathbf{protect}(e))}{\langle is, c:cs, \mu, \rho\rangle \xrightarrow{\mathbf{fetch}}_{\epsilon} \langle is{+}{+}[i], cs, \mu, \rho\rangle}$$

$$\textsc{Exec-Protect}_1\quad \frac{i=(x{:=}\mathbf{protect}(e)) \qquad v=[\![e]\!]^{\rho} \qquad i'=(x{:=}\mathbf{protect}(v))}{\langle is_1, i, is_2, cs\rangle \overset{(\mu,\rho,\epsilon)}{\rightsquigarrow} \langle is_1{+}{+}[i']{+}{+}is_2, cs\rangle}$$

$$\textsc{Exec-Protect}_2\quad \frac{i=(x{:=}\mathbf{protect}(v)) \qquad \mathbf{guard}(\_,\_,\_) \notin is_1 \qquad i'=(x{:=}v)}{\langle is_1, i, is_2, cs\rangle \overset{(\mu,\rho,\epsilon)}{\rightsquigarrow} \langle is_1{+}{+}[i']{+}{+}is_2, cs\rangle}$$

Figure 11. Semantics of **protect**(·) (selected rules).

and reinserts the instruction in the reorder buffer as if it were a normal assignment. However, the processor leaves the value wrapped inside the protect instruction in the reorder buffer, i.e., $x := \mathbf{protect}(v)$, to prevent forwarding the value to the later instructions via the the transient variable map. When no guards are pending in the reorder buffer ($\mathbf{guard}(\_,\_,\_) \notin is_1$), rule [Exec-Protect$_2$] transforms the instruction into a normal assignment, so that the processor can propagate and commit its value.

Example. Consider again Ex1 and the execution shown in Figure 10. In the repaired program, $x + y$ is wrapped in a **protect** statement. As a result, directive **exec** 5 produces value $z{:=}\mathbf{protect}(42)$, instead of $z{:=}42$ which prevents instruction 7 from executing (as its target address is undefined), until all guards are resolved. This in turn prevents the leaking of the transient value.

Stable Read. Unfortunately, current processors do not provide the means to implement protect in its full generality. Our semantics therefore contains a primitve $\mathbf{stable\_read}(e_1, e_2)$ that implements a restricted version of $\mathbf{protect}(e_1[e_2])$ for array reads. While **protect**(·) prevents forwarding loaded values until all pending branches are resolved, **stable_read**(·) stalls memory loads until individual bounds-check conditions have been resolved. **stable_read**(·) can be implemented using today's hardware, for example through speculative Load Hardening (SLH) [10], the spectre mitigation proposed by and deployed in the Clang compiler. We provide formal semantics in Appendix B.

Example. Consider again Ex1. Instead of using **protect**(·), we can repair the example by inserting **stable_read**. Instead of a single **protect**(·) for expression $x+y$, we however need to insert two **stable_read** for $a[i_1]$ and $a[i_2]$, respectively.

4 Type System and Inference

In Section 4.1, we present a transient-flow type system which statically rejects programs that can potentially leak through transient execution attacks. Given an unannotated program, we apply constraint-based type inference [3, 27] to generate its use-def graph and reconstruct type information (Section 4.2). Then, reusing off-the-shelf Max-Flow/Min-Cut algorithms, we analyze the graph and locate potential speculative vulnerabilities in the form of a variable min-cut set. Finally, using a simple program repair algorithm we patch the program by inserting a minimum number of **protect** so that it does not leak speculatively anymore (Figure 13).

4.1 Type System

Our transient-flow type system prevents programs from leaking transient values via cache timing channels. To this end, the type system assigns a *transient-flow type* to expressions and tracks how transient values propagate within programs, rejecting programs in which transient values reach commands which may leak them. An expression can either be typed as *stable* (**S**) indicating that it cannot contain transient values during execution, or as *transient* (**T**) indicating that it can. These types form a 2-point lattice [23], which allows stable expressions to be typed as transient, but not vice versa, i.e., we define a can-flow-to relation $\sqsubseteq$ such that $\mathbf{S} \sqsubseteq \mathbf{T}$, but $\mathbf{T} \not\sqsubseteq \mathbf{S}$.

$$\textsc{Value}\quad \Gamma \vdash v:\tau \;\Rightarrow \emptyset \qquad \textsc{Var}\quad \frac{\Gamma(x)=\tau}{\Gamma \vdash x:\tau \;\Rightarrow x \sqsubseteq \alpha_x}$$

$$\textsc{Bop}\quad \frac{\Gamma \vdash e_1:\tau_1 \;\Rightarrow k_1 \qquad \Gamma \vdash e_2:\tau_2 \;\Rightarrow k_2 \qquad \tau_1 \sqsubseteq \tau \qquad \tau_2 \sqsubseteq \tau}{\Gamma \vdash e_1 \oplus e_2:\tau \;\Rightarrow k_1 \cup k_2 \cup (e_1 \sqsubseteq e_1 \oplus e_2) \cup (e_2 \sqsubseteq e_1 \oplus e_2)}$$

$$\textsc{Array-Read}\quad \frac{\Gamma \vdash e_1:\mathbf{S} \;\Rightarrow k_1 \qquad \Gamma \vdash e_2:\mathbf{S} \;\Rightarrow k_2}{\Gamma \vdash e_1[e_2]:\mathbf{T} \;\Rightarrow k_1 \cup k_2 \cup (e_1 \sqsubseteq \mathbf{S}) \cup (e_2 \sqsubseteq \mathbf{S}) \cup (\mathbf{T} \sqsubseteq e_1[e_2])}$$

(a) Typing Rules for Expressions and Arrays.

$$\textsc{Asgn}\quad \frac{\Gamma \vdash r:\tau \;\Rightarrow k \qquad \tau \sqsubseteq \Gamma(x)}{\Gamma,\mathrm{Prot} \vdash x := r \;\Rightarrow k \cup (r \sqsubseteq x)}$$

$$\textsc{Protect}\quad \frac{\Gamma \vdash r:\tau \;\Rightarrow k}{\Gamma,\mathrm{Prot} \vdash x := \mathbf{protect}(r) \;\Rightarrow k} \qquad \textsc{Asgn-Prot}\quad \frac{\Gamma \vdash r:\tau \;\Rightarrow k \qquad x \in \mathrm{Prot}}{\Gamma,\mathrm{Prot} \vdash x := r \;\Rightarrow k \cup (r \sqsubseteq x)}$$

$$\textsc{Stable-Read}\quad \frac{\Gamma \vdash e_1:\mathbf{S} \;\Rightarrow k \qquad \Gamma \vdash e_2:\mathbf{S}}{\Gamma,\mathrm{Prot} \vdash x := \mathbf{stable\_read}(e_1,e_2) \;\Rightarrow k \cup (e_1 \sqsubseteq \mathbf{S}) \cup (e_2 \sqsubseteq \mathbf{S})}$$

$$\textsc{If-Then-Else}\quad \frac{\Gamma \vdash e:\mathbf{S} \;\Rightarrow k \qquad \Gamma,\mathrm{Prot} \vdash c_1 \;\Rightarrow k_1 \qquad \Gamma,\mathrm{Prot} \vdash c_2 \;\Rightarrow k_2}{\Gamma,\mathrm{Prot} \vdash \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \;\Rightarrow k \cup k_1 \cup k_2 \cup (e \sqsubseteq \mathbf{S})}$$

(b) Typing Rules fo Commands.

Figure 12. Transient flow type system and type constraints generation (selected rules).

Typing Expressions. Given a typing environment for variables $\Gamma \in Var \rightarrow \{\mathbf{S},\mathbf{T}\}$, the typing judgement $\Gamma \vdash r:\tau$ assigns a transient-flow type τ to r. Figure 12 presents selected rules (see Appendix C for the rest). The shaded part of the rules generates type constraints during type inference and are explained later. Values can assume any type. Variables are assigned their respective type from the environment. Rule [Bop] propagates the type of the operands to the result of binary operators $\oplus \in \{+,<\}$. Finally, rule [Array-Read] assigns the transient type to array reads as the array may potentially be indexed out of bounds during speculation. Importantly, the rule requires the array index to be stable to prevent programs from leaking through the cache.

Typing Commands. Given a set of protected variables Prot, we define a typing judgment $\Gamma, \mathrm{Prot} \vdash c$ for commands. Intuitively, a command c is well-typed under environment Γ and set Prot, if c does not leak, under the assumption that the expressions assigned to all variables in Prot are protected using the **protect**(·) primitive. Figure 12b shows our typing rules. Rule [Asgn] disallows assignments from transient to stable variables (as $\mathbf{T} \not\sqsubseteq \mathbf{S}$). Rule [Protect] relaxes this policy as long as the right-hand side is explicitly protected. Intuitively, the result of **protect**(·) is stable and it can thus flow securely to variables of any type. Rule [Asgn-Prot] is similar, but instead of requiring an explicit **protect**(·) statement, it demands that the variable is accounted for in the protected set Prot. This is secure because all assignments to variables in Prot will eventually be protected through the repair function discussed later in this section. Since primitive $x :=$ **stable_read**(e_1, e_2) corresponds to the array read $e_1[e_2]$, rule [Stable-Read] requires the array and the index argument to be stable like in rule [Array-Read]. Similar to **protect**(·), the result of **stable_read**(·) is stable and thus the type of the variable needs no constraints.

Implicit Flows. To prevent programs from leaking data *implicitly* through their control flow, rule [If-Then-Else] requires the branch condition to be stable. This might seem overly restrictive, at first: why can't we accept a program that branches on transient data, as long as it does not perform any attacker-observable operations (e.g., memory reads and writes) along the branches? Indeed, classic information-flow control (IFC) type systems (e.g., [36]) take this approach by keeping track of an explicit program counter label. Unfortunately, such permissiveness is *unsound* under speculation. Even if a branch does not contain observable behavior, the value of the branch condition can be leaked by the instructions that *follow* a mispredicted branch. In particular, the rollback caused by a misprediction may cause to *repeat* load and store instructions after the mispredicted branch, thus revealing whether the attacker guessed the value of the branch condition.

Example. Consider the following program: **if** *tr* **then** $x :=$ 0 **else skip**; $y := a[0]$. The program can leak the value of *tr* during speculative execution. To see that, assume that the processor predicts that *tr* will evaluate to **true**. Then, the processor speculatively executes the then-branch (x:=0) and the load instruction ($y := a[0]$), before resolving the condition. If *tr* is **true**, the memory trace of the program contains a single read observation. However, if *tr* is **false**, the processor detects a misprediction, restarts the execution from the other branch (**skip**) and executes the array read, producing a rollback and *two* read observations. From these observations, an attacker could potentially make inferences about the value of *tr*. Consequently, if *tr* is typed as **T**, our type system rejects the program as unsafe.

4.2 Type Inference

We now present our type inference algorithm.

Constraints. We start by collecting a set of constraints k via typing judgement $\Gamma, \text{Prot} \vdash s \Rightarrow k$. For this, we define a dummy environment Γ^* and protected set Prot^*, such that $\Gamma^*, \text{Prot}^* \vdash c \Rightarrow k$ holds for any command c, (*i.e.,* we let $\Gamma^* = \lambda x.\mathbf{S}$ and include *all* variables in the cut-set) and use it to extract the set of constraints k. The syntax for constraints is shown in Figure 21. The constraints relate *atoms* which represent the unknown type of variables, *i.e.,* α_x for x, and expression, *i.e.,* r. Constraints record can-flow-to relationships between the atoms and lattice values $\mathbf{T}$ and $\mathbf{S}$. They are accumulated via operator $\cup$, where we identify $k_1 \cup \cdots \cup k_n$ with the set $\{k_1, \ldots, k_n\}$.

Solutions and Satisfiability. We define the solution to a set of constraints as a function σ from atoms to flow types, *i.e.,* $\sigma \in \textsc{Atoms} \mapsto \{\mathbf{T}, \mathbf{S}\}$, and extend solutions to map $\mathbf{T}$ and $\mathbf{S}$ to themselves. For a set of constraints k and a solution function σ, we write $\sigma \vdash k$ to say that the constraints k are satisfied under solution σ. A solution σ satisfies k, if all can-flow-to constraints hold, when the atoms are replaced by their values under σ. We say that a set of constraints k is satisfiable, if there is a solution σ such that $\sigma \vdash k$.

Def-Use Graph & Paths. The constraints generated by our type system give rise to the def-use graph of the type-checked program. For a set of constraints k, we call a sequence of atoms $a_1 \ldots a_n$ a *path* in k, if $a_i \sqsubseteq a_{i+1} \in k$ for $i \in \{1, \ldots, n-1\}$ and say that a_1 is the path's entry and a_n its exit. A $\mathbf{T}$-$\mathbf{S}$ path is a path with entry $\mathbf{T}$ and exit $\mathbf{S}$. A set of constraints k is satisfiable if and only if there is no $\mathbf{T}$-$\mathbf{S}$ path in k, as such a path would correspond to a derivation of false. If k is satisfiable, we can compute a solution $\sigma(k)$ by letting $\sigma(k)(a) = \mathbf{T}$, if there is a path with entry $\mathbf{T}$ and exit a, and $\mathbf{S}$ otherwise.

Cuts. If a set of constraints is unsatisfiable, we can make it satisfiable by removing some of the nodes in its graph or equivalently protecting some of the variables. A set of atoms A *cuts* a path $a_1 \ldots a_n$, if some $a \in A$ occurs along the path, *i.e.,* there exists $a \in A$ and $i \in \{1, \ldots, n\}$ such that $a_i = a$. We call A a cut-set for a set of constraints k, if A cuts all $\mathbf{T}$-$\mathbf{S}$ paths in k. A cut-set A is minimal for k, if all other cut-sets A' contain as many or more atoms than A, *i.e.,* $\#A \leqslant \#A'$.

Extracting Types From Cuts. From a set of variables A such that A is a cut-set of constraints k, we can extract a typing environment $\Gamma(k, A)$ as follows: for an atom α_x, we define $\Gamma(k,A)(x) = \mathbf{T}$, if there is a path with entry $\mathbf{T}$ and exit α_x in k that is not cut by A, and let $\Gamma(k,A)(x) = \mathbf{S}$ otherwise.

Proposition 1 (Type Inference). *If* $\Gamma^*, \text{Prot}^* \vdash c \Rightarrow k$ *and* A *is a set of variables that cut* k*, then* $\Gamma(k,A), A \vdash s$.

Remark. To infer a repair using **stable_read** instead of **protect**, we can restrict our cut-set to only include variables that are assigned from an array read.

Atom	a	$::=$	$\alpha_x \mid r$
Constraint	k	$::=$	$a \sqsubseteq \mathbf{S} \mid \mathbf{T} \sqsubseteq a \mid a \sqsubseteq a \mid k \cup k \mid \emptyset$
Solution	σ	$\in$	$\textsc{Atoms} \mapsto \{\mathbf{S}, \mathbf{T}\}$

Figure 13. Constraint Syntax.

Example. Consider again Ex1 in Figure 3. The graph defined by the constraints k, given by $\Gamma^*, \text{Prot}^* \vdash \text{Ex1} \Rightarrow k$ is shown in Figure 4, where we have omitted α-nodes. The constraints are not satisfiable, since there are $\mathbf{T}$-$\mathbf{S}$ paths. Both $\{x,y\}$ and $\{z\}$ are cut-sets, since they cut each $\mathbf{T}$-$\mathbf{S}$ path, however, the set $\{z\}$ contains only one element and is therefore minimal. The typing environment $\Gamma(k, \{x,y\})$ extracted from the sub-obptimal cut $\{x,y\}$ types all variables as $\mathbf{S}$, while the typing extracted from the optimal cut, i.e., $\Gamma(k, \{z\})$ types x and y as $\mathbf{T}$ and z, i_1 and i_2 as $\mathbf{S}$. By Proposition 2 both $\Gamma(k, \{x,y\}), \{x,y\} \vdash \text{Ex1}$ and $\Gamma(k, \{z\}), \{z\} \vdash \text{Ex1}$ hold.

4.3 Program Repair

As a final step, our repair algorithm $repair(c, \text{Prot})$ traverses program c and inserts a **protect**$(\cdot)$ statement for each variable in the cut-set Prot. Since we assume that programs are in static single assignment form, there is a single assignment $x := r$ for each variable $x \in \text{Prot}$, and our repair algorithm simply replaces it with $x :=$ **protect**(r).

5 Consistency and Security

We now present two formal results about our speculative semantics and the security of the type system. Our full definitions and proofs can be found in Appendix D.

Consistency. We write $C \Downarrow_O^D C'$ for the complete speculative execution of configuration C to final configuration C', which generates a trace of observations O under list of directives D. Similarly, we write $\langle \mu, \rho \rangle \Downarrow_O^c \langle \mu', \rho' \rangle$ for the sequential execution of program c with initial memory μ and variable map ρ resulting in final memory μ' and variable map ρ'. To relate speculative and sequential observations, we define a projection function, written $O\downarrow$, which removes prediction identifiers, rollbacks, and mispeculated loads and stores.

Theorem 5.1 (Consistency). *For all programs* c*, initial memory stores* μ*, variable maps* ρ*, and directives* D*, such that* $\langle \mu, \rho \rangle \Downarrow_O^c \langle \mu', \rho' \rangle$ *and* $\langle [\,], [\,c\,], \mu, \rho \rangle \Downarrow_{O'}^D \langle [\,], [\,], \mu'', \rho'' \rangle$*, then* $\mu' = \mu''$*,* $\rho' = \rho''$*, and* $O \cong O'\downarrow$.

The theorem ensures equivalence of the final memory stores, variable maps, and observation traces from the sequential and the speculative execution. Notice that trace equivalence is up to *permutation*, i.e., $O \cong O'\downarrow$, because the processor can execute load and store instructions out-of-order.

Speculative Non-Interference. Speculative non-interference is parametric in the security policy that specifies which variables and part of the memory are controlled by the attacker [17]. In the following, we write L for the set of public

variables and memory locations that are *observable* by the attacker. Two variable maps are *indistinguishable* to the attacker, written $\rho_1 \approx_L \rho_2$, if and only if $\rho_1(x) = \rho_2(x)$ for all $x \in L$. Similarly, memory stores are related pointwise, i.e., $\mu_1 \approx_L \mu_2$ iff $\mu_1(n) = \mu_2(n)$ for all $n \in L$.

Definition 1 (Speculative Non-Interference). *A program c satisfies speculative non-interference if and only if for all directives D, memory stores and variable maps such that* $\mu_1 \approx_L \mu_2$ *and* $\rho_1 \approx_L \rho_2$, *let* $C_i = \langle [\,], [c], \mu_i, \rho_i \rangle$ *for* $i \in \{1,2\}$, *such that* $C_1 \Downarrow^D_{O_1} C_1'$, $C_2 \Downarrow^D_{O_2} C_2'$, *if* $O_1\downarrow = O_2\downarrow$, *then* $O_1 = O_2$.

In the defintion above, programs *leak* by producing different observations starting from memories and variables indistinguishable to the attacker. Speculative non-interference requires showing absence of leaks for the speculative traces ($O_1 = O_2$) assuming that the program does not already leak sequentially ($O_1\downarrow = O_2\downarrow$). Notice that here we consider syntactic equivalence for the traces because both executions follow the same list of directives. We now present our soundness theorem: well-typed programs satisfy speculative non-interference.

Theorem 5.2 (Soundness). *For all programs c, if* $\Gamma \vdash c$ *then c satisfies speculative non-interference.*

We conclude with a corollary that combines all the components of our protection chain (type inference, type checking and automatic repair via our security primitives) and shows that repaired programs satisfy speculative non-interference.

Corollary 5.3. *For all programs c, there exists a set of constraints k such that* $\Gamma^*, \mathrm{Prot}^* \vdash c \Rightarrow k$. *Let A be a set of variables that cut k, then the repaired program repair*(c, A) *satisfies speculative non-interference.*

6 Implementation and Evaluation

We now describe our implementation and evaluate BLADE on an implementation of the Signal secure messaging protocol and various cryptographic algorithms. Our evaluation shows that BLADE can secure existing software systems against speculative execution attacks automatically. Moreover, BLADE introduces two orders of magnitude less fences than a baseline algorithm implemented in Clang. As a result, the repairs computed by BLADE incur only a minimal performance overhead.

6.1 Implementation

We implemented BLADE in 3500 lines of Haskell code. BLADE takes as input a WebAssembly program, computes a repaired program that is safe under speculative execution and verifies its correctness via type-checking. Internally, BLADE proceeds in three stages. First, BLADE converts the WebAssembly program into an intermediate representation similar to the While language in Figure 5. This simplifies further processing as WebAssembly is a stack-based language, *i.e.,* arguments are not represented directly, but instead kept on an argument stack. Second, BLADE builds the use-def graph (§4.1) of the input program, infers a minimal cut-set (§4.2), and computes the repair (§4.3). Finally, in the last stage, BLADE extracts a typing-environment from the use-def graph and type-checks the repaired program (§4). This independent checking step provides extra confidence that the repaired program indeed does not leak more speculatively, than it does sequentially (§5). Source code will be made available under an open source license.

6.2 Evaluation

We evaluate BLADE by answering three questions: ***(Q1)*** Can we apply BLADE to secure existing software? ***(Q2)*** How many `protect` statements does BLADE have to insert in order to secure those systems? and ***(Q3)*** How do the inserted fences affect performance?

(Q1) Applicability. To evaluate BLADE's applicability, we run it on crypto code, which is already carefully written to eschew cache-timing side channels. Our benchmarks are taken from two main sources: first, a verified implementation [29] of the Signal messaging protocol [15], and second, verified implementations of several crypto primitives taken from [38]. In particular, our benchmarks consist of

- The messaging algorithm implemented in module Signal Core and common cryptographic constructions implemented in module Signal Crypto and used in Signal.
- The HACL* SHA2 hash, AES block cypher, Curve25519 elyptic curve function, and ED25519 digital signature used in Signal.
- The SALSA20 stream cypher, SHA2 hash, and TEA block cypher from [38].

The original implementations of our benchmarks are *provably* free from cache and timing side-channel. However, those proofs considered only a sequential execution model and therefore do not account for the speculative execution vulnerabilities addressed in this work.

Results. Table 1 shows the code size in Webssembly text format, and the runtime of BLADE on each benchmark. The runtime includes translation, repair and type-checking. The results are encouraging: the execution time scales proportionally with the code size and the analysis completes fairly quickly, even for large benchmarks (>60k WASM LOC): the runtime is less than 10s for all of our benchmarks.

(Q2) Number of Fences. Next, we evaluate how many fences the analysis has to insert to make the programs secure. The results are shown in Table 1. Column **B** contains our baseline, which replaces all non-constant array reads, *i.e.,* reads whose address depends on a variable, with statement `stable_read` (Section 3.4), implementing a SLH-like mitigation that masks the address with the array bounds-check condition. This is the proposed mitigation in the Clang compiler [10]. Column **P** shows the number of `protect` inserted by BLADE. All benchmarks are modified by the baseline, except for TEA, which is a simple, toy encryption algorithm

Name	B	P	S	P/B	LOC	Time	
CRYPTO [29]	92	1	2	1.1	3386	181.0	ms
CORE [29]	47	1	2	2.1	6595	347.8	ms
SHA2 [29]	156	18	34	11.5	7310	286.7	ms
AES[29]	48	0	0	0	6284	28.95	ms
CURVE [29]	2214	0	0	0	59921	5.571	s
ED25519 [29]	2403	6	10	0.2	60308	8.797	s
SALSA 20 [38]	7	0	0	0	529	20.20	ms
SHA 256 [38]	23	0	0	0	334	11.23	ms
TEA [38]	0	0	0	-	112	3.036	ms
Total	**4990**	**26**	**48**	**0.5**	**144779**	-	

Table 1. (**B**) contains our baseline, *i.e.,* the number of `stable_read`, if every non-constant read is protected; (**P**) contains the number of `protect` statements insert by BLADE; (**S**) contains the number of `stable_read` inserted, if `stable_read` is used to implement `protect`; (**P/B**) contains the ratio of `protect` statments to the baseline fences in %; (**LOC**) contains the number of lines of WASM code in text format; (**Time**) shows the mean timing for fence inference, repair, and typechecking over 15 trials; Experiments were run on a 12" Macbook with 8GB RAM.

(that should not be used in practice). In particular, for five of the nine programs, BLADE does not need to insert any fences. Column **P/B** shows the ratio of protect statements to baseline read masks in percent. For most benchmarks, our analysis has to insert under 3% of fences compared to the baseline. For the SHA2 implementation of HACL* this rises to 11.5%. Across all benchmarks, the number of fences is two orders of magnitude lower than the baseline. Since `protect` statements are an idealized primitive that are not available in todays hardware, we show the number of `stable-read` primitives that are needed to implement the `protect` in column **S**. The table shows that using stable reads requires inserting more fences by a factor of 1.8x, which underlines the benefits of a hardware implementation of `protect`.

(Q3) Performance Impact of Fences. To evaluate the performance impact of our repair, we compared how a *naive* placement of fences—applying speculative load hardening to every load of a non-constant address—compares against our approach. We picked the SHA2-512 hash function for this test, and used inputs of size 4KB. Naive fence placement introduced 44 fences while ours introduced only 5. Our measurements showed that while the naive repair algorithm caused 13.9% overhead, the overhead of our minimal fence replacement algorithm was only 0.42%. We used a sample size of 500, and found the relative margin of error of our measurements were less than 0.07%.

7 Related Work

Transient Execution Attacks. Since Spectre [21] and Meltdown [24] were announced, many transient execution attacks exploiting different microarchitectural components and side-channels have been discovered and new ones come to light at a steady pace. These attacks leak data across arbitrary security boundaries, including SGX enclaves [14, 35], hypervisors and virtual machines [40], and even remotely over a network [31]. We refer to [9] for a comprehensive systematization.

Detection and Repair. Wu and Wang [41] detect cache side channels via abstract interpretation by augmenting the control-flow to accomodate for speculation. Spectector [17] and Pitchfork [11] use symbolic execution on x86 binaries to detect speculative vulnerabilities. Cheang et al. [13] and Bloem et al. [8] apply bounded model checking to detect potential speculative vulnerabilities respectively via 4-ways self-composition and taint-tracking. Almost all these efforts [8, 11, 13, 17, 41] consider only *in-order* execution (except Pitchfork [11]) for a *fixed* speculation bound, and focus on vulnerability detection but do not propose techniques to *repair* vulnerable programs. In contrast, our type system enforces speculative non-interference even when program instructions are executed *out-of-order* with *unbounded* speculation and automatically synthesizes repairs. Given a set of untrusted input source, oo7 Wang et al. [37] statically analyzes a binary to detect vulnerable patterns and inserts fences in turn. Our tool, BLADE, not only repairs vulnerable programs without user annotation, but ensures that program patches contain a minimum number of fences. Furthermore, BLADE formally guarantees that repaired programs are free from speculation-based attacks.

Speculative Execution Semantics. There have been several recent proposals for speculative execution semantics [11, 13, 17, 26]. Of those, [11] is closest to ours, and inspired our semantics (e.g., we share the 3-stages pipeline, attacker-supplied directives and the instruction reorder buffer). However their semantics targets an assembly language with direct jumps, while we reason about speculative execution of imperative programs with structured control-flow.

Hardware Mitigations and Secure Design. Both AMD AMD [5] and Intel Intel [19] recommend inserting serializing, fence instructions after bounds checks to protect against Spectre v1 attacks and some compilers followed suit [18, 28]. Unfortunately, these defenses causes significant performance degradation [9]. Taram et al. [32] propose context-sensitive fencing, a hardware-based mitigation that dynamically inserts fences in the instruction stream when dangerous conditions arise. Several secure hardware designs have been studied to remove speculative attacks from future processors. InvisiSpec Yan et al. [42] is a new micro-architecture design that features a special *speculative buffer* to prevent speculative loads from polluting the cache. STT [2] tracks speculative taints inside the processor micro-architecture and prevent speculative values from reaching instructions that could serve as covert channels. We think our approach could be applied to guide such hardware mitigations by pinpointing the program parts that need to be protected.

References

[1] *Flows in Networks*. Princeton University Press, 1962.

[2] Speculative taint tracking (stt): A comprehensive protection for speculatively accessed data. In *MICRO*, 2019.

[3] Alex Aiken. Constraint-based program analysis. In Radhia Cousot and David A. Schmidt, editors, *Static Analysis*, pages 1–1, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. ISBN 978-3-540-70674-8.

[4] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *Usenix Security*, 2016.

[5] AMD. Software techniques for managing speculation on AMD processors. https://developer.amd.com/wp-content/resources/Managing-Speculation-on-AMD-Processors.pdf, 2018.

[6] GILLES BARTHE, SANDRINE BLAZY, BENJAMIN GRÉGOIRE, RÉMI HUTIN, VINCENT LAPORTE, DAVID PICHARDIE, and ALIX TRIEU. Formal verification of a constant-time preserving c compiler. In *POPL*, 2020.

[7] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. Smotherspectre: Exploiting speculative execution through port contention. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 785–800, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6747-9. doi: 10.1145/3319535.3363194. URL http://doi.acm.org/10.1145/3319535.3363194.

[8] Roderick Bloem, Swen Jacobs, and Yakir Vizel. Efficient information-flow verification under speculative execution. In Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza, editors, *Automated Technology for Verification and Analysis*, pages 499–514, Cham, 2019. Springer International Publishing. ISBN 978-3-030-31784-3.

[9] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, pages 249–266, Berkeley, CA, USA, 2019. USENIX Association. ISBN 978-1-939133-06-9. URL http://dl.acm.org/citation.cfm?id=3361338.3361356.

[10] Chandler Carruth. Speculative load hardening. https://llvm.org/docs/SpeculativeLoadHardening.html, 2019.

[11] Sunjay Cauligi, Craig Disselkoen, Klaus von Gleissenthall, Deian Stefan, Tamara Rezk, and Gilles Barthe. Towards constant-time foundations for the new spectre era. *CoRR*, abs/1910.01755, 2019. URL http://arxiv.org/abs/1910.01755.

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

[13] Kevin Cheang, Cameron Rasmussen, Sanjit A. Seshia, and Pramod Subramanyan. A formal approach to secure speculation. In *Proceedings of the Computer Security Foundations Symposium (CSF)*, 2019.

[14] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten-Hwang Lai. Sgxpectre attacks: Leaking enclave secrets via speculative execution. *CoRR*, abs/1802.09085, 2018. URL http://arxiv.org/abs/1802.09085.

[15] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. In *EuroS&P*, 2017.

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

[17] Marco Guarnieri, Boris Koepf, José Francisco Morales, Jan Reineke, and Andrés Sánchez. Spectector: Principled detection of speculative information flows. In *Proc. IEEE Symp. on Security and Privacy*, SSP '20, 2020.

[18] Intel. Using intel compilers to mitigate speculative execution side-channel issues. https://software.intel.com/en-us/articles/using-intel-compilers-to-mitigate-speculative-execution-side-, 2018.

[19] Intel. Intel analysis of speculative execution side channels. https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf, 2018.

[20] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses. *CoRR*, abs/1807.03757, 2018. URL http://arxiv.org/abs/1807.03757.

[21] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[22] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *Proceedings of the 12th USENIX Conference on Offensive Technologies*, WOOT'18, pages 3–3, Berkeley, CA, USA, 2018. USENIX Association. URL http://dl.acm.org/citation.cfm?id=3307423.3307426.

[23] J. Landauer. A lattice of information. In *CSFW*, 1993.

[24] 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. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[25] Giorgi Maisuradze and Christian Rossow. Ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 2109–2122, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5693-0. doi: 10.1145/3243734.3243761. URL http://doi.acm.org/10.1145/3243734.3243761.

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

[27] Hanne Riis Nielson and Flemming Nielson. Flow logics for constraint based analysis. In Kai Koskimies, editor, *Compiler Construction*, pages 109–127, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg. ISBN 978-3-540-69724-4.

[28] Andrew Pardoe. Spectre mitigations in msvc. https://devblogs.microsoft.com/cppblog/spectre-mitigations-in-msvc/, 2018.

[29] Jonathan Protzenko, Benjamin Beurdouche, Denis Merigoux, and Karthikeyan Bhargavan. Formally verified cryptographic web applications in webassembly. In *Security and Privacy*, 2019.

[30] G. Romer and C. Carruth. C++ proposal, 2019. URL http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0928r0.pdf.

[31] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. Netspectre: Read arbitrary memory over network. *CoRR*, abs/1807.10535, 2018. URL http://arxiv.org/abs/1807.10535.

[32] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Context-sensitive fencing: Securing speculative execution via microcode customization. In *ASPLOS'19*.

[33] Vadim Tkachenko. 20-30% performance hit from the spectre bug fix on ubuntu. https://www.percona.com/blog/2018/01/23/20-30-performance-hit-spectre-bug-fix-ubuntu/, Jan 2018.

[34] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures. *J. Cryptol.*, 23(1):37–71, January 2010. ISSN 0933-2790. doi: 10.1007/s00145-009-9049-y. URL http://dx.doi.org/10.1007/s00145-009-9049-y.

[35] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom,

and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, August 2018. See also technical report Foreshadow-NG [40].

[36] D. Volpano, G. Smith, and C. Irvine. A Sound Type System for Secure Flow Analysis. *J. Computer Security*, 4(3):167–187, 1996.

[37] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead defense against spectre attacks via binary analysis. *CoRR*, abs/1807.05843, 2018. URL http://arxiv.org/abs/1807.05843.

[38] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. Ct-wasm: Type-driven secure cryptography for the web ecosystem. In *POPL*, 2019.

[39] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. Ct-wasm: Type-driven secure cryptography for the web ecosystem. *Proc. ACM Program. Lang.*, 3(POPL):77:1–77:29, January 2019. ISSN 2475-1421. doi: 10.1145/3290390. URL http://doi.acm.org/10.1145/3290390.

[40] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. *Technical report*, 2018. See also USENIX Security paper Foreshadow [35].

[41] Meng Wu and Chao Wang. Abstract interpretation under speculative execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 802–815, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6712-7. doi: 10.1145/3314221.3314647. URL http://doi.acm.org/10.1145/3314221.3314647.

[42] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, pages 428–441, Piscataway, NJ, USA, 2018. IEEE Press. ISBN 978-1-5386-6240-3. doi: 10.1109/MICRO.2018.00042. URL https://doi.org/10.1109/MICRO.2018.00042.

[43] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, August 2014. USENIX Association. ISBN 978-1-931971-15-7. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

A Full Calculus

FETCH-SKIP
$$\langle is, \mathbf{skip} : cs, \mu, \rho\rangle \xrightarrow{\mathbf{fetch}}_{\epsilon} \langle is +\!+ [\mathbf{nop}], cs, \mu, \rho\rangle$$

FETCH-ASGN
$$\langle is, x := e : cs, \mu, \rho\rangle \xrightarrow{\mathbf{fetch}}_{\epsilon} \langle is +\!+ [x := e], cs, \mu, \rho\rangle$$

FETCH-SEQ
$$\langle is, c_1; c_2 : cs, \mu, \rho\rangle \xrightarrow{\mathbf{fetch}}_{\epsilon} \langle is, c_1 : c_2 : cs, \mu, \rho\rangle$$

FETCH-PTR-LOAD
$$\langle is, x := *e : cs, \mu, \rho\rangle \xrightarrow{\mathbf{fetch}}_{\epsilon} \langle is +\!+ [x := \mathbf{load}(e)], cs, \mu, \rho\rangle$$

FETCH-PTR-STORE
$$\langle is, *e_1 := e_2 : cs, \mu, \rho\rangle \xrightarrow{\mathbf{fetch}}_{\epsilon} \langle is +\!+ [\mathbf{store}(e_1, e_2)], cs, \mu, \rho\rangle$$

FETCH-FAIL
$$\langle is, \mathbf{fail} : cs, \mu, \rho\rangle \xrightarrow{\mathbf{fetch}}_{\epsilon} \langle is +\!+ [\mathbf{fail}], cs, \mu, \rho\rangle$$

FETCH-ARRAY-LOAD
$$\frac{c = x := e_1[e_2] \qquad e = e_2 < \mathit{length}(e_1) \qquad \mathrm{fresh}(p) \\ e' = \mathit{base}(e_1) + e_2 \qquad c' = \mathbf{if}\ e\ \mathbf{then}\ x := *e'\ \mathbf{else}\ \mathbf{fail}}{\langle is, c : cs, \mu, \rho\rangle \xrightarrow{\mathbf{fetch}}_{\epsilon} \langle is, c' : cs, \mu, \rho\rangle}$$

FETCH-ARRAY-STORE
$$\frac{c = e_1[e_2] := e_3 \qquad e = e_2 < \mathit{length}(e_1) \qquad \mathrm{fresh}(p) \\ e' = \mathit{base}(e_1) + e_2 \qquad c' = \mathbf{if}\ e\ \mathbf{then}\ *e' := e\ \mathbf{else}\ \mathbf{fail}}{\langle is, c : cs, \mu, \rho\rangle \xrightarrow{\mathbf{fetch}}_{\epsilon} \langle is, c' : cs, \mu, \rho\rangle}$$

FETCH-IF-TRUE
$$\frac{c = \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \\ \mathrm{fresh}(p) \qquad i = \mathbf{guard}(e^{\mathbf{true}}, c_2 : cs, p)}{\langle is, c : cs, \mu, \rho\rangle \xrightarrow{\mathbf{fetch\,true}}_{\epsilon} \langle is +\!+ [i], c_1 : cs, \mu, \rho\rangle}$$

FETCH-IF-FALSE
$$\frac{c = \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \\ \mathrm{fresh}(p) \qquad i = \mathbf{guard}(e^{\mathbf{false}}, c_1 : cs, p)}{\langle is, c : cs, \mu, \rho\rangle \xrightarrow{\mathbf{fetch\,false}}_{\epsilon} \langle is +\!+ [i], c_2 : cs, \mu, \rho\rangle}$$

FETCH-WHILE
$$\frac{c_1 = c; \mathbf{while}\ e\ c \qquad c_2 = \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ \mathbf{skip}}{\langle is, \mathbf{while}\ e\ c : cs, \mu, \rho\rangle \xrightarrow{\mathbf{fetch}}_{\epsilon} \langle is, c_2 : cs, \mu, \rho\rangle}$$

Figure 14. Fetch stage.

EXECUTE

$$\frac{|is_1| = n-1 \qquad \rho' = \phi(is_1,\rho) \qquad \langle is_1,i,is_2,cs\rangle \xrightarrow{(\mu,\rho',o)} \langle is',cs'\rangle}{\langle is_1 {+\!+} [\,i\,] {+\!+} is_2,cs,\mu,\rho\rangle \xrightarrow{\mathbf{exec}\,n}_o \langle is',cs',\mu,\rho\rangle}$$

EXEC-ASGN

$$\frac{i = (x := e) \qquad v = [\![e]\!]^\rho \qquad i' = (x := v)}{\langle is_1,i,is_2,cs\rangle \xrightarrow{(\mu,\rho,\epsilon)} \langle is_1 {+\!+} [\,i'\,] {+\!+} is_2,cs\rangle}$$

EXEC-BRANCH-OK

$$\frac{i = \mathbf{guard}(e^b,cs',p) \qquad [\![e]\!]^\rho = b}{\langle is_1,i,is_2,cs\rangle \xrightarrow{(\mu,\rho,\epsilon)} \langle is_1 {+\!+} [\,\mathbf{nop}\,] {+\!+} is_2,cs\rangle}$$

EXEC-BRANCH-MISPREDICT

$$\frac{i = \mathbf{guard}(e^b,cs',p) \qquad [\![e]\!]^\rho \neq b}{\langle is_1,i,is_2,cs\rangle \xrightarrow{(\mu,\rho,\mathbf{rollback}(p))} \langle is_1,cs'\rangle}$$

EXEC-LOAD

$$\frac{i = x := \mathbf{load}(e) \qquad \mathbf{store}(\_,\_) \notin is_1 \qquad n = [\![e]\!]^\rho \qquad ps = (\!|is_1|\!) \qquad i' = (x := \mu(n))}{\langle is_1,i,is_2,cs\rangle \xrightarrow{(\mu,\rho,\mathbf{read}(n,ps))} \langle is_1 {+\!+} [\,i'\,] {+\!+} is_2,cs\rangle}$$

EXEC-STORE-ADDR

$$\frac{i = \mathbf{store}(e_1,e_2) \qquad n = [\![e_1]\!]^\rho \qquad i' = \mathbf{store}(n,e_2)}{\langle is_1,i,is_2,cs\rangle \xrightarrow{(\mu,\rho,\epsilon)} \langle is_1 {+\!+} [\,i'\,] {+\!+} is_2,cs\rangle}$$

EXEC-STORE-VALUE

$$\frac{i = \mathbf{store}(n,e) \qquad v = [\![e]\!]^\rho \qquad ps = (\!|is_1|\!) \qquad i' = \mathbf{store}(n,v)}{\langle is_1,i,is_2,cs\rangle \xrightarrow{(\mu,\rho,\mathbf{write}(n,ps))} \langle is_1 {+\!+} [\,i'\,] {+\!+} is_2,cs\rangle}$$

Figure 15. Execute stage.

RETIRE-NOP

$$\langle \mathbf{nop}: is,cs,\mu,\rho\rangle \xrightarrow{\mathbf{retire}}_\epsilon \langle is,cs,\mu,\rho\rangle$$

RETIRE-ASGN

$$\langle x := v: is,cs,\mu,\rho\rangle \xrightarrow{\mathbf{retire}}_\epsilon \langle is,cs,\mu,\rho[x \mapsto v]\rangle$$

RETIRE-STORE

$$\frac{i = \mathbf{store}(n,v)}{\langle i: is,cs,\mu,\rho\rangle \xrightarrow{\mathbf{retire}}_\epsilon \langle is,cs,\mu[n \mapsto v],\rho\rangle}$$

RETIRE-FAIL

$$\langle \mathbf{fail}: is,cs,\mu,\rho\rangle \xrightarrow{\mathbf{retire}}_{\mathbf{fail}} \langle [\,],[\,],\mu,\rho\rangle$$

Figure 16. Retire stage.

$$\begin{aligned}
&\phi(\rho,[\,]) = \rho\\
&\phi(\rho,(x := v): is) = \phi(\rho[x \mapsto v],is)\\
&\phi(\rho,(x := e): is) = \phi(\rho[x \mapsto \bot],is)\\
&\phi(\rho,(x := \mathbf{load}(e)): is) = \phi(\rho[x \mapsto \bot],is)\\
&\phi(\rho,(x := \mathbf{protect}(e)): is) = \phi(\rho[x \mapsto \bot],is)\\
&\phi(\rho,i: is) = \phi(\rho,is)
\end{aligned}$$

(a) Transient Variable Map.

$$\begin{aligned}
&[\![v]\!]^\rho = v\\
&[\![x]\!]^\rho = \rho(x)\\
&[\![length(e)]\!]^\rho = length([\![e]\!]^\rho)\\
&[\![base(e)]\!]^\rho = base([\![e]\!]^\rho)\\
&[\![e_1 + e_2]\!]^\rho = [\![e_1]\!]^\rho + [\![e_2]\!]^\rho\\
&[\![e_1 \leqslant e_2]\!]^\rho = [\![e_1]\!]^\rho \leqslant [\![e_2]\!]^\rho
\end{aligned}$$

(b) Evaluation Function.

$$\begin{aligned}
&(\!|[\,]|\!) = [\,]\\
&(\!|\mathbf{guard}(e^b,cs,p): is|\!) = p: (\!|is|\!)\\
&(\!|i: is|\!) = (\!|is|\!)
\end{aligned}$$

(c) Pending Guard Identifiers.

Figure 17. Helper functions.

$$\textsc{Fetch-Protect-Ptr}\quad \frac{c = x := \mathbf{protect}(*e) \qquad c_1 = x := *e \qquad c_2 = x := \mathbf{protect}(x)}{\langle is, c : cs, \mu, \rho\rangle \xrightarrow{\mathbf{fetch}}_{\epsilon} \langle is, c_1 : c_2 : cs, \mu, \rho\rangle}$$

$$\textsc{Fetch-Protect-Array}\quad \frac{c = x := \mathbf{protect}(e_1[e_2]) \qquad c_1 = x := e_1[e_2] \qquad c_2 = x := \mathbf{protect}(x)}{\langle is, c : cs, \mu, \rho\rangle \xrightarrow{\mathbf{fetch}}_{\epsilon} \langle is, c_1 : c_2 : cs, \mu, \rho\rangle}$$

$$\textsc{Fetch-Protect-Expr}\quad \frac{c = x := \mathbf{protect}(e) \qquad i = x := \mathbf{protect}(e)}{\langle is, c : cs, \mu, \rho\rangle \xrightarrow{\mathbf{fetch}}_{\epsilon} \langle is +\!+ [\,i\,], cs, \mu, \rho\rangle}$$

$$\textsc{Exec-Protect}_1\quad \frac{i = x := \mathbf{protect}(e) \qquad v = [\![e]\!]^{\rho} \qquad i' = x := \mathbf{protect}(v)}{\langle is_1, i, is_2, cs\rangle \xrightarrow{(\mu,\rho,\epsilon)} \langle is_1 +\!+ [\,i'\,] +\!+ is_2, cs\rangle}$$

$$\textsc{Exec-Protect}_2\quad \frac{i = x := \mathbf{protect}(v) \qquad \mathbf{guard}(\_,\_,\_) \notin is_1 \qquad i' = (x := v)}{\langle is_1, i, is_2, cs\rangle \xrightarrow{(\mu,\rho,\epsilon)} \langle is_1 +\!+ [\,i'\,] +\!+ is_2, cs\rangle}$$

Figure 18. Semantics of **protect**(·).

B Semantics of Stable Read

Current processors do not provide a protect primitive instruction nor the means to implement it on top of existing instructions, in its full generality. However, for array reads, it is possible to replicate the effects of protect by exploiting the same data-dependencies tracking capabilities at the core of the processor pipeline. Indeed, Speculative Load Hardening (SLH), a mitigation technique deployed in the code generated by the CLANG compiler, relies on data-dependencies to secure memory loads automatically [10]. Using our formal model, we give rigorous semantics to SLH and show that it can stop transient execution attacks.

At a high level, SLH injects *artificial* data-dependencies between the conditions used in branch instructions and the addresses loaded in the following instructions to transform control-flow dependencies into data-flow dependencies. Intuitively, these data-dependencies validate control-flow decisions at runtime by stalling speculative loads until the processor resolves the conditions. Using branch conditions, SLH *masks* the address of loads instructions in such a way that the processor zeroes out the address if the condition is mispredicted, preventing misloads.

To formalize this mechanism, we extend our processor model as follows. We introduce a new processor instruction $x := e\,?\,e_1 : e_2$, which corresponds to the conditional move instruction CMOV on x86 processors. This instruction simply assigns the value of e_1 (resp. e_2) to variable x, if the condition e evaluates to true (resp. false). Importantly, this instruction is not subject to speculation: the processor must first evaluate the condition before it can resolve the assignment. We also extend expressions with the standard bitwise AND operator (&) and write $\overline{0}$ and $\overline{1}$ for bit words consisting of all 0 and 1. As usual bitmask $\overline{0}$ and $\overline{1}$ are respectively the zero and identity element for &, i.e., $[\![e\&\overline{0}]\!]^\rho = \overline{0}$ and $[\![e\&\overline{1}]\!]^\rho = [\![e]\!]^\rho$.

Figure 19 presents the semantics rules for CMOV and for the stable read command implemented using SLH. Rule [Exec-CMOV] evaluates the condition ($b = [\![\rho]\!]^e$) of the conditional assignment $x := e?:e_{\mathbf{true}}:e_{\mathbf{false}}$ and assigns the corresponding expressions ($x := e_b$). Rule [Fetch-Stable-Read-SLH] fetches command $x := \mathbf{stable\_read}(e_1, e_2)$, computes the bounds check condition, the address of the indexed element, and push on the stack the following command.

$$
\begin{array}{l}
r := e_1 \leqslant \mathit{length}(e_2) \\
\mathbf{if}\ r\ \mathbf{then} \\
\quad r := r\,?\,\overline{1} : \overline{0}; \\
\quad x := *((\mathit{base}(e_1) + e_2)\,\&\,r); \\
\mathbf{else} \\
\quad \mathbf{fail}
\end{array}
$$

The code above is similar to the code generated by a regular array read, but additionally stores the result of the bounds-check condition in reserved variable r. In the then-branch, the condition is then converted into a suitable bitmask using using the non-speculative CMOV instruction i.e., $r := r\,?\,\overline{1} : \overline{0}$, which then masks the address loaded, i.e., $*((\mathit{base}(e_1) + e_2)\,\&\,r)$. As a result, the value of the address remains undefined until the processor evaluates the bounds check condition. When the condition resolves, if the index is inbound $r = \overline{1}$ and the program reads the correct address $[\![e\&\overline{1}]\!]^\rho = [\![e]\!]^\rho$ If the index is out-of-bounds, instead, $r = \overline{0}$ and the load can only read speculatively from a constant address ($x := \mu(0)$), thus closing the leak.[4]

Exec-CMOV

$$
\frac{i = x := e?:e_{\mathbf{true}}:e_{\mathbf{false}} \qquad b = [\![e]\!]^\rho \qquad i' = x := e_b}{\langle is_1, i, is_2, cs\rangle \overset{(\mu,\rho,\epsilon)}{\rightsquigarrow} \langle is_1 +\!+ [\,i'\,] +\!+ is_2, cs\rangle}
$$

Fetch-Stable-Read-SLH

$$
\frac{\begin{array}{ll} c = x := \mathbf{stable\_read}(e_1, e_2) & e = e_2 < \mathit{length}(e_1) \\ e' = \mathit{base}(e_1) + e_2 & c_1 = r := e \qquad c_2 = r := r\,?\,\overline{1} : \overline{0} \\ c_3 = x := *(e'\,\&\,r) & c' = c_1; \mathbf{if}\ r\ \mathbf{then}\ c_2; c_3\ \mathbf{else}\ \mathbf{fail} \end{array}}{\langle is, c : cs, \mu, \rho\rangle \xrightarrow{\mathbf{fetch}}_\epsilon \langle is, c' : cs, \mu, \rho\rangle}
$$

Figure 19. Semantics of $x := \mathbf{stable\_read}(e_1, e_2)$.

Revisited Example. Consider again running example Ex1 in Figure 3, where instead of standard array reads, we employ the **stable_read**(·) primitive from above. After fetching the program, the addresses of the loads are masked with the respective array bounds-check conditions. Assuming the same memory layout and content as in Figure 10 (except for the fact that arrays are shifted by one position since $\mu(0) = 0$ is reserved), the processor resolves the first bounds check and reads the array within its bounds, i.e., $x := \mu(3) = 0$. The second load attempts to read the array out of bounds ($y := a[2]$), and our countermeasure prevents the buffer overrun by redirecting the load to the dummy value stored at address 0. First, the processor resolves the bounds check, i.e., $r := \overline{0}$, and forwardes it to the load $y := \mathbf{load}((\mathit{base}(a) + i_2)\,\&\,r)$. Then, the condition zeros out the address and the processor assigns the dummy value to variable y, i.e., $y := \mu(0)$. As a result, we always read array b at index $z = 0$ and close the leak.

[4] We assume that the first memory cell is reserved to the processor, which initializes it with dummy data, e.g., $\mu(0) = 0$.