Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade

MARCO VASSENA, CISPA Helmholtz Center for Information Security, Germany
CRAIG DISSELKOEN, UC San Diego, USA
KLAUS VON GLEISSENTHALL, Vrije Universiteit Amsterdam, Netherlands
SUNJAY CAULIGI, UC San Diego, USA
RAMI GÖKHAN KICI, UC San Diego, USA
RANJIT JHALA, UC San Diego, USA
DEAN TULLSEN, UC San Diego, USA
DEIAN STEFAN, UC San Diego, USA

We introduce Blade, a new approach to automatically and efficiently eliminate speculative leaks from cryptographic code. Blade is built on the insight that to stop leaks via speculative execution, it suffices to *cut* the dataflow from expressions that speculatively introduce secrets (*sources*) to those that leak them through the cache (*sinks*), rather than prohibit 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. Blade relies on a new abstract primitive, **protect**, to halt speculation at fine granularity. We formalize and implement **protect** using existing architectural mechanisms, and show how Blade's type system can automatically synthesize a *minimal* number of **protect**s to provably eliminate speculative leaks. We implement Blade in the Cranelift WebAssembly compiler and evaluate our approach by repairing several verified, yet vulnerable WebAssembly implementations of cryptographic primitives. We find that Blade can fix existing programs that leak via speculation *automatically*, without user intervention, and *efficiently* even when using fences to implement **protect**.

CCS Concepts: • **Security and privacy** → **Formal security models**.

Additional Key Words and Phrases: Speculative execution, Spectre, Constant-time, Type system.

ACM Reference Format:
Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kıcı, Ranjit Jhala, Dean Tullsen, and Deian Stefan. 2021. Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade. *Proc. ACM Program. Lang.* 5, POPL, Article 49 (January 2021), 30 pages. https://doi.org/10.1145/3434330

1 INTRODUCTION

Implementing secure cryptographic algorithms is hard. The code must not only be functionally correct, memory safe, and efficient, it must also avoid divulging secrets indirectly through side channels like control-flow, memory-access patterns, or execution time. Consequently, much recent

Authors' addresses: Marco Vassena, CISPA Helmholtz Center for Information Security, Germany, marco.vassena@cispa.saarland; Craig Disselkoen, UC San Diego, USA, cdisselk@cs.ucsd.edu; Klaus von Gleissenthall, Vrije Universiteit Amsterdam, Netherlands, k.freiherrvongleissenthal@vu.nl; Sunjay Cauligi, UC San Diego, USA, scauligi@eng.ucsd.edu; Rami Gökhan Kıcı, UC San Diego, USA, rkici@eng.ucsd.edu; Ranjit Jhala, UC San Diego, USA, jhala@cs.ucsd.edu; Dean Tullsen, UC San Diego, USA, tullsen@cs.ucsd.edu; Deian Stefan, UC San Diego, USA, deian@cs.ucsd.edu.

2475-1421/2021/1-ART49
https://doi.org/10.1145/3434330

work focuses on how to ensure implementations do not leak secrets e.g., via type systems [Cauligi et al. 2019; Watt et al. 2019], verification [Almeida et al. 2016; Protzenko et al. 2019], and program transformations [Barthe et al. 2019].

Unfortunately, these efforts can be foiled by speculative execution. Even if secrets are closely controlled via guards and access checks, the processor can simply ignore these checks when executing speculatively. This, in turn, can be exploited by an attacker to leak protected secrets.

In principle, memory fences block speculation, and hence, offer a way to recover the original security guarantees. In practice, however, fences pose a dilemma. Programmers can restore security by conservatively inserting fences after every load (e.g., using Microsoft's Visual Studio compiler pass [Donenfeld 2020]), but at huge performance costs. Alternatively, they can rely on heuristic approaches for inserting fences [Wang et al. 2018], but forgo guarantees about the absence of side-channels. Since missing even one fence can allow an attacker to leak any secret from the address space, secure runtime systems—in particular, browsers like Chrome and Firefox—take yet another approach and isolate secrets from untrusted code in different processes to avoid the risk altogether [Blog 2010; Mozilla Wiki 2018]. Unfortunately, the engineering effort of such a multi-process redesign is huge—e.g., Chrome's redesign took five years and roughly 450K lines of code changes [Reis et al. 2019].

In this paper, we introduce Blade, a new, fully automatic approach to provably and efficiently eliminate speculation-based leakage from constant-time cryptographic code. Blade is based on the key insight that to prevent leaking data via speculative execution, it is not necessary to stop *all* speculation. Instead, it suffices to *cut* the data flow from expressions (sources) that could speculatively introduce secrets to those that leak them through the cache (sinks). We develop this insight into an automatic enforcement algorithm via four contributions.

A JIT-Step Semantics for Speculation. A key aim of BLADE is to enable *source*-level reasoning about the absence of speculation-based information leaks. This is crucial to let the source-level type system use control- and data-flow information to optimally prevent leaks. High-level reasoning requires a source-level semantic model of speculation which has, so far, proven to be challenging as the effects of speculation manifest at the very low *machine*-level: e.g., as branch-prediction affects the *streams* of low-level instructions that are fetched and (speculatively) executed. We address this challenge with our first contribution: a JIT-step semantics for a high-level While language that lets us reconcile the tension between high-level reasoning and low-level speculation. These semantics translate high-level *commands* to low-level machine *instructions* in a *just-in-time* (JIT) fashion, whilst tracking control-flow predictions at branch points and modeling the speculative execution path of the program as well as the "paths not taken" as stacks of high-level commands.

Our low-level instructions are inspired by a previous formal model of a speculative and out-of-order processor [Cauligi et al. 2020], and let us model the essence of speculation-based attacks—in particular Spectre-PHT [Canella et al. 2019; Kiriansky and Waldspurger 2018; Kocher et al. 2019]—by modeling precisely 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 embodies several hardware mechanisms proposed in recent work [Taram et al. 2019; Yu et al. 2019]. Crucially, and in contrast to a regular fence which stops *all* speculation, **protect** only stops speculation for a given *variable*. For example $x := \textbf{protect}(e)$ ensures that the value of e is assigned to x only *after* e has been assigned its *stable*, non-speculative value. Though we encourage hardware vendors to implement **protect** in future processors, for backwards compatibility, we implement and evaluate two versions of **protect** on existing hardware—one using fences, another using *speculative load hardening* (SLH) [Carruth 2019].

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. The type system does *not* rely on user-provided security annotations to identify sensitive sources and public sinks. Instead, we *conservatively* reject programs that exhibit any flow of information from transient sources to stable sinks. This, in turn, allows us to automatically identify speculative vulnerabilities in existing cryptographic code (where secrets are not explicitly annotated). We connect the static and dynamic semantics by proving that well-typed constant-time programs are secure, i.e., they are also *speculative constant-time* [Cauligi et al. 2020] (§ 5). This result extends the pre-existing guarantees about sequential constant-time execution of verified cryptographic code to our speculative execution model.

Automatic Protection. Existing programs that are free of **protect** statements are likely insecure under speculation (see Section 7 and [Cauligi et al. 2020]) and will be rejected by our type system. Thus, our third contribution is an algorithm that finds potential speculative leaks and automatically synthesizes a *minimal* number of **protect** statements to ensure that the program is speculatively constant-time (§ 4.2). To this end, we extend the type checker to construct a *def-use graph* that captures the data-flow between program expressions. The presence of a *path* from transient sources to stable sinks in the graph indicates a potential speculative leak in the program. To repair the program, we only need to identify a *cut-set*, a set of variables whose removal eliminates all the leaky paths in the graph. 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. 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* number of protections.

Blade Tool. Our final contribution is an automatic push-button tool, Blade, which eliminates potential speculative leaks using this min-cut algorithm. Blade extends the Cranelift compiler [Bytecode Alliance 2020], which compiles WebAssembly (Wasm) to x86 machine code; thus, Blade operates on programs expressed in Wasm. However, operating on Wasm is not fundamental to our approach—we believe that Blade's techniques can be applied to other programming languages and bytecodes.

We evaluate Blade by repairing verified yet vulnerable (to transient execution attacks) constant-time cryptographic primitives from Constant-time Wasm (CT-Wasm) [Watt et al. 2019] and HACL* [Zinzindohoué et al. 2017] (§ 7). Compared to existing fully automatic speculative mitigation approaches (as notably implemented in Clang), Blade inserts an order of magnitude fewer protections (fences or SLH masks). Blade's fence-based implementation imposes modest performance overheads: (geometric mean) 5.0% performance overhead on our benchmarks to defend from Spectre v1, or 15.3% overhead to also defend from Spectre v1.1. Both results are significant gains over current solutions. Our fence-free implementation, which automatically inserts SLH masks, is faster in the Spectre v1 case—geometric mean 1.7% overhead—but slower when including Spectre v1.1 protections, imposing 26.6% overhead.

2 OVERVIEW

This section gives a brief overview of the kinds of speculative leaks that Blade identifies and how it repairs such programs by careful placement of **protect** statements. We then describe how Blade

```
1  void SHA2_update_last(int *input_len, ...)
2  {
3    if (! valid(input_len)) { return; }  // Input validation
4    int len = protect(*input_len);  // Can speculatively read secret data
5    ...
6    int *dst3 = len + base;  // Secret-tainted address
7    ...
8    *dst3 = pad;  // Secret-dependent memory access
9    ...
10   for ( i = 0; i < len + ...)  // Secret-dependent branch
11     dst2[i] = 0;
12   ...
13 }
```

Fig. 1. Code fragment adapted from the HACL* SHA2 implementation, containing two potential speculative execution vulnerabilities: one through the data cache by writing memory at a secret-tainted address, and one through the instruction cache via a secret-tainted control-flow dependency. The patch computed by Blade is shown in **green**.

(1) automatically repairs existing programs using our minimal **protect** inference algorithm and (2) proves that the repairs are correct using our transient-flow type system.

2.1 Two Kinds of Speculative Leaks

Figure 1 shows a code fragment of the `SHA2_update_last` function, a core piece of the SHA2 cryptographic hash function implementation, adapted (to simplify exposition) from the HACL* library. This function takes as input a pointer `input_len`, validates the input (line 3), loads from memory the public length of the hash (line 4, ignore **`protect`** for now), calculates a target address `dst3` (line 6), and pads the buffer pointed to by `dst3` (line 8). Later, it uses `len` to determine the number of initialization rounds in the condition of the for-loop on line 10.

Leaking Through a Memory Write. During normal, sequential execution this code is not a problem: the function validates the input to prevent classic buffer-overflow vulnerabilities. However, during speculation, an attacker can exploit this function to leak sensitive data. To do this, the attacker first has to *mistrain* the branch predictor to predict the next input to be valid. Since `input_len` is a function parameter, the attacker can do this by, e.g., calling the function repeatedly with legitimate addresses. After mistraining the branch predictor this way, the attacker manipulates `input_len` to point to an address containing secret data 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 load the secret into `len`, which in turn is used to calculate pointer `dst3`. The location pointed to by `dst3` is then written on line 8, leaking the secret data. Even though pointer `dst3` is invalid and the subsequent write will not be visible at the program level (the processor disregards it), the side-effects of the memory access persist in the cache and therefore become visible to the attacker. In particular, the attacker can extract the target address—and thereby the secret—using cache timing measurements [Ge et al. 2018].

Leaking Through a Control-Flow Dependency. The code in Figure 1 contains a second potential speculative vulnerability, which leaks secrets through a control-flow dependency instead of a memory access. To exploit this vulnerability, the attacker can similarly manipulate 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, the attacker can leak the value *indirectly* through a control-flow dependency: in this case, the secret determines how often the initialization loop (line 10) is executed during speculation. The attacker can then infer the value of the secret from a timing attack on the instruction cache or (much more easily) on iteration-dependent lines of the data cache.

2.2 Eliminating Speculative Leaks

Preventing the Leak using Memory Fences. Since these leaks exploit the fact that input validation is speculatively skipped, we can prevent them by making sure that dangerous operations such as the write on line 8 or the loop condition check on line 10 are not executed until the input has been validated. Intel [2018a], AMD [2018], and others [Donenfeld 2020; Pardoe 2018] recommend doing this by inserting a speculation barrier after critical validation check-points. This would place a *memory fence* after line 3, but anywhere between lines 3 and 8 would work. This fence would stop speculation over the fence: statements after the fence will not be executed until all statements up to the fence (including input validation) executed. While fences can prevent leaks, using fences as such is more restrictive than necessary—they stop speculative execution of all following instructions, not only of the instructions that leak—and thus incur a high performance cost [Taram et al. 2019; Tkachenko 2018].

Preventing the Leak Efficiently. We do not need to stop all speculation to prevent leaks. Instead, we only need to ensure that potentially secret data, when read speculatively, cannot be leaked. To this end, we propose an alternative way to stop speculation from reaching the operations on line 8 and line 10, 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 on line 4. Instead of assigning the value `len` directly from the result of the load, the memory load is guarded by a **protect** statement. This ensures that the value assigned to `len` is always guaranteed to use the `input_len` pointer's final, nonspeculative value. This single **protect** statement on line 4 is sufficient to fix both of the speculative leaks described in Section 2.1—it prevents any speculative, secret data from reaching lines 8 or 10 where it could be leaked to the attacker.

Implementation of protect. Our **protect** primitive provides an *abstract interface* for fine grained control of speculation. This allows us to eliminate speculation-based leaks precisely and only when needed. However, whether **protect** can eliminate leaks with tolerable runtime overhead depends on its concrete implementation. We consider and formalize two implementations: an ideal implementation and one we can implement on today's hardware.

To have fine grain control of speculation, **protect** must be implemented in *hardware* and exposed as part of the ISA. Though existing processors provide only coarse grained control over speculation through memory fence instructions, this might change in the future. For example, recently proposed microprocessor designs [Taram et al. 2019; Yu et al. 2019] provide new hardware mechanisms to control speculation, in particular to restrict targeted types of speculation while allowing other speculation to proceed: this suggests that **protect** could be implemented efficiently in hardware in the future.

Even if future processors implement **protect**, we still need to address Spectre attacks on existing hardware. Hence, we formalize and implement **protect** in *software*, building on recent Spectre attack mitigations [Schwarz et al. 2020]. Specifically, we propose a self-contained approach inspired by Clang's Speculative Load Hardening (SLH) [Carruth 2019]. At a high level, Clang's SLH stalls speculative load instructions in a conditional block by inserting artificial data-dependencies between loaded addresses and the value of the condition. This ensures that the load is not executed before the branch condition is resolved. Unfortunately, this approach unnecessarily stalls *all* non-constant conditional load instructions, regardless of whether they can influence a subsequent load and thus can actually cause speculative data leaks. Furthermore, this approach is unsound—it can also miss some speculative leaks, e.g., if a load instruction is not in the same conditional block that validates its

Example	Example Patched
$x := a[i_1]$	$x :=$ **protect**$(a[i_1])$
$y := a[i_2]$	$y :=$ **protect**$(a[i_2])$
$z := x + y$	$z :=$ **protect**$(x + y)$
$w := b[z]$	$w := b[z]$

Fig. 2. Running example. Program Example is shown on the left and the patched program is shown on the right. The **orange** patch is sub-optimal because it requires more **protect** statements than the optimal **green** patch.

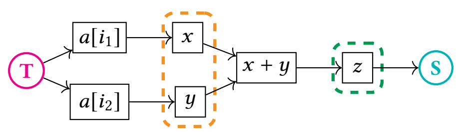

Fig. 3. *Subset* of the def-use graph of Example. The dashed lines identify two valid choices of cut-sets. The **left cut** requires removing two nodes and thus inserting two **protect** statements. The **right cut** shows a minimal solution, which only requires removing a single node.

address (like the code in Figure 1). In contrast to Clang, our approach applies SLH *selectively*, only to individual load instructions whose result flows to an instruction which might leak, and *precisely*, by using accurate bounds-check conditions to ensure that only data from valid addresses can be loaded.

2.3 Automatically Repairing Speculative Leaks via protect-Inference

Blade automatically finds potential speculative leaks and synthesizes a *minimal* number of **protect** statements to eliminate the leaks. We illustrate this process using the simple program Example in Figure 2 as a running example. The program 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 ($w := b[z]$). This program is written using our formal calculus in which all array operations are implicitly bounds-checked and thus no explicit validation code is needed.

Like the SHA2 example from Figure 1, Example contains a speculative execution vulnerability: the speculative array reads could bypass their bounds checks and so x and y can contain transient secrets (i.e., secrets introduced by misprediction). This secret data then flows to z, and finally leaks through the data cache when reading $b[z]$.

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 (a subset of) the def-use graph of program Example in Figure 3. In the graph, the edge $x \longrightarrow x + y$ indicates that x is used to compute $x + y$. To track how transient values propagate in the def-use graph, we extend the graph with the special 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., **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 runtime. To detect insecure uses of transient values, we then extend the graph with the special 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., $z \longrightarrow$ **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 may be leaky.

Cutting the Dataflow. In order to make the program safe, we need to *cut* the data-flow between **T** and **S** by introducing **protect** statements. This problem can be equivalently restated as follows: find a *cut-set*, i.e., a set of variables, such that removing the variables from the graph eliminates all paths from **T** to **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 3 contains two choices of cut-sets, shown as dashed lines. The cut-set on the left requires two protect statements, for variables x and y respectively, corresponding to the **orange** patch in Figure 2. 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 2. Intuitively, minimal cut-sets result in patches that introduce as few **protect**s as needed and therefore allow more speculation. Luckily, the problem of finding a minimal cut-set is an instance of the classic Min-Cut/Max-Flow problem, which can be solved using efficient, polynomial-time algorithms [Ford and Fulkerson 2010]. For simplicity, Blade adopts a uniform cost model and therefore synthesizes patches that contain a minimal *number* of **protect** statements, regardless of their position in the code and how many times they can be executed. Though our evaluation shows that even this simple model imposes modest overheads (§7), our implementation can be easily optimized by considering additional criteria when searching for a minimal cut set, with further performance gain likely. For example, we could assign weights proportional to execution frequency, or introduce penalties for placing **protect** inside loops.

2.4 Proving Correctness via Transient-Flow Types

To ensure that we add **protect** statements in all the right places (without formally verifying our repair algorithm), we use a type system to *prove* that patched programs are secure, i.e., they satisfy a *semantic* security condition. The type system simplifies the security analysis—we can reason about program execution directly rather than through generic flows of information in the def-use graph. Moreover, restricting the security analysis to the type system makes the security proofs independent of the specific algorithm used to compute the program repairs (e.g., the Max-Flow/Min-Cut algorithm). As long as the repaired program type checks, Blade's formal guarantees hold. To show that the patches obtained from cutting the def-use graph of a given program are *correct* (i.e., they make the program well-typed), our transient-flow type system constructs its def-use graph from the type-constraints generated during type inference.

Typing Judgement. Our 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 runtime, and as *stable* (written as **S**), otherwise. For instance, in program Example (Fig. 3) variables x and y (and hence z) are typed as *transient* because they may temporarily contain secret data originated from speculatively reading the array out of bounds. 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 (§ 4.1). The type system ensures that transient expressions are not 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 ensures that transient expressions are not written to stable variables, except via **protect**. For example, our type system rejects program Example because it uses *transient* variable z as an array index, but it accepts program Example Patched in which z can be typed *stable* thanks to the **protect** statements. We say that variables whose assignment is guarded by a **protect** statement (like z in Example Patched) are "protected variables".

To study the security guarantees of our type system, we define a JIT-step semantics for speculative execution of a high-level While language (§ 3), which resolves the tension between source-level reasoning and machine-level speculation. We then show that our type system indeed prevents speculative execution attacks, i.e., we prove that well-typed programs remain constant-time under the speculative semantics (§ 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 $\Gamma, \text{Prot} \vdash c \Rightarrow k$ (§ 4.2), which extends the typing judgment from above with (1) a set of implicitly protected variables Prot (the *cut-set*), and (2) a set of type-constraints k (the *def-use graph*). Intuitively, the set Prot identifies the variables of program c that contain *transient* data and that must be protected (i.e., they must be made *stable* using **protect**) to avoid leaks. 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 (the *final* protected set), and *(iii)* compute the *final* typing environment which types the variables in the cut-set as stable. 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.[1] Intuitively, the program is secure if we promise to insert a **protect** statement for each variable in Prot. To repair programs, we simply honor this promise and insert a protect statement for each variable in the protected set of the typing judgment obtained above. Once repaired, the program type checks under an *empty* protected set.

2.5 Attacker Model

We delineate the extents of the security guarantees of our type system and repair algorithm by discussing the attacker model considered in this work. We assume an attacker model where the attacker runs cryptographic code, written in Wasm, on a speculative out-of-order processor; the attacker can influence how programs are speculatively executed using the branch predictor and choose the instruction execution order in the processor pipeline. The attacker can make predictions based on control-flow history and memory-access patterns similar to real, adaptive predictors. Though these predictions *can* depend on secret information in general, we assume that they are secret-independent for the constant-time programs repaired by Blade.

We assume that the attacker can observe the effects of their actions on the cache, even if these effects are otherwise invisible at the ISA level. In particular, while programs run, the attacker can take precise timing measurements of the data- and instruction-cache with a cache-line granularity, and thus infer the value of secret data. These features allow the attacker to mount Spectre-PHT attacks [Kiriansky and Waldspurger 2018; Kocher et al. 2019] and exfiltrate data through FLUSH+RELOAD [Yarom and Falkner 2014] and PRIME+PROBE [Tromer et al. 2010] cache side-channel attacks. We do not consider speculative attacks that rely on the Return Stack Buffer (e.g., Spectre-RSB [Koruyeh et al. 2018; Maisuradze and Rossow 2018]), Branch Target Buffer (Spectre-BTB [Kocher et al. 2019]), or Store-to-Load forwarding misprediction (Spectre-STL [Horn 2018], recently reclassified as a Meltdown attack [Moghimi et al. 2020]). We similarly do not consider Meltdown attacks [Lipp et al. 2018] or attacks that do not use the cache to exfiltrate data, e.g., port contention (SMoTherSpectre [Bhattacharyya et al. 2019]).

3 A JIT-STEP SEMANTICS FOR SPECULATION

We now formalize the concepts discussed in the overview, presenting a semantics in this section and a type system in Section 4. Our semantics are explicitly conservative and abstract, i.e., they only

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

$$
\begin{array}{rl}
\text{Values } v ::= & n \mid b \mid a \\
\text{Expr. } e ::= & v \mid x \mid e + e \mid e < e \\
 & \mid e \otimes e \mid e\,?\,e : e \\
 & \mid \mathit{length}(e) \mid \mathit{base}(e) \\
\text{Rhs. } r ::= & e \mid *e \mid a[e] \\
\text{Cmd. } c ::= & \textbf{skip} \mid x := r \mid *e = e \\
 & \mid a[e] := e \mid \textbf{fail} \mid c; c \\
 & \mid \textbf{if } e \textbf{ then } c \textbf{ else } c \\
 & \mid \textbf{while } e \textbf{ do } c \\
 & \mid x := \textbf{protect}(r)
\end{array}
$$

(a) Source syntax.

$$
\begin{array}{rl}
\text{Instructions } i ::= & \textbf{nop} \mid x := e \mid x := \textbf{load}(e) \\
 & \mid \textbf{store}(e, e) \mid x := \textbf{protect}(e) \\
 & \mid \textbf{guard}(e^b, cs, p) \mid \textbf{fail}(p) \\
\text{Directives } d ::= & \textbf{fetch} \mid \textbf{fetch } b \mid \textbf{exec } n \mid \textbf{retire} \\
\text{Observations } o ::= & \epsilon \mid \textbf{read}(n, ps) \mid \textbf{write}(n, ps) \\
 & \mid \textbf{fail}(p) \mid \textbf{rollback}(p) \\
\text{Predictions } b \in & \{\textbf{true}, \textbf{false}\} \\
\text{Guard Fail Ids. } p \in & \mathbb{N} \\
\text{Instr. Buffers } is ::= & i : is \mid [\,] \\
\text{Cmd Stacks } cs ::= & c : cs \mid [\,] \\
\text{Stores } \mu \in & \mathbb{N} \rightharpoonup \mathit{Value} \\
\text{Var. Maps } \rho \in & \mathit{Var} \to \mathit{Value} \\
\text{Config. } C ::= & \langle is, cs, \mu, \rho \rangle
\end{array}
$$

(b) Processor syntax.

Fig. 4. Syntax of the calculus.

model the *essential* features required to capture Spectre-PHT attacks on modern microarchitectures—namely, speculative and out-of-order execution. We do not model microarchitectural features exploited by other attacks (e.g., Spectre-BTB and Spectre-RSB), which require a more complex semantics model [Cauligi et al. 2020; Guanciale et al. 2020].

Language. We start by giving a formal just-in-time step semantics for a While language with speculative execution. We present the language's source syntax in Figure 4a. Its values consist of Booleans b, pointers n represented as natural numbers, and arrays a. The calculus has standard expression constructs: values v, variables x, addition $e + e$, and a comparison operator $e < e$. To formalize the semantics of the software (SLH) implementation of $\textbf{protect}(\cdot)$ we rely on several additional constructs: the bitwise AND operator $e \otimes e$, the *non-speculative* conditional ternary operator $e\,?\,e : e$, and primitives for getting the length ($\mathit{length}(\cdot)$) and base address ($\mathit{base}(\cdot)$) of an array. We do not specify the size and bit-representation of values and, instead, they satisfy standard properties. In particular, we assume that $n \otimes \mathbf{0} = \mathbf{0} = \mathbf{0} \otimes n$, i.e., the bitmask $\mathbf{0}$ consisting of all 0s is the zero element for $\otimes$ and corresponds to number 0, and $n \otimes \mathbf{1} = n = \mathbf{1} \otimes n$, i.e., the bitmask $\mathbf{1}$ consisting of all 1s is the unit element for $\otimes$ and corresponds to some number $n' \neq 0$. Commands include variable assignments, pointer dereferences, array stores,[2] conditionals, and loops.[3] Beyond these standard constructs, we expose a special command that is used to prevent transient execution attacks: the command $x := \textbf{protect}(r)$ evaluates r and assigns its value to x, but only after the value

[2]The syntax for reading and writing arrays requires the array to be a constant value (i.e., $a[e]$) to simplify our security analysis. This simplification does not restrict the power of the attacker, who can still access memory at arbitrary addresses speculatively through the index expression e.

[3] The branch constructs of our source language can be used to directly model Wasm's branch constructs (e.g., conditionals and while loops can map to Wasm's **br_if** and **br**) or could be easily extended (e.g., we can add **switch** statements to support Wasm's branch table **br_table** instruction) [Haas et al. 2017]. To keep our calculus small, we do not replicate Wasm's (less traditional) low-level branch constructs which rely on block labels (e.g., **br** *label*). Since Wasm programs are statically typed, these branch instructions switch execution to statically known, well-specified program points marked with labels. Our calculus simply avoids modeling labels explicitly and replaces them with the code itself.

is *stable* (i.e., non-transient). Lastly, **fail** triggers a memory violation error (caused by trying to read or write an array out-of-bounds) and aborts the program.

JIT-Step Semantics. Our operational semantics formalizes the execution of source programs on a pipelined processor and thus enables *source*-level reasoning about speculation-based information leaks. In contrast to previous semantics for speculative execution [Cauligi et al. 2020; Cheang et al. 2019; Guarnieri et al. 2020; McIlroy et al. 2019], our processor abstract machine does not operate directly on fully compiled assembly programs. Instead, our processor translates high-level commands into low-level instructions *just in time*, by converting individual commands into corresponding instructions in the first stage of the processor pipeline. To support this execution model, the processor *progressively* flattens structured commands (e.g., **if**-statements and **while** loops) into predicted straight-line code and maintains a *stack* of (partially flattened) commands to keep track of the program execution path. In this model, when the processor detects a misspeculation, it only needs to replace the command stack with the sequence of commands that should have been executed instead to restart the execution on the correct path.

Processor Instructions. Our semantics translates source commands into an abstract set of processor instructions shown in Figure 4b. Most of the processor instructions correspond directly to the basic source commands. Notably, the processor instructions do not include an explicit jump instruction for branching. Instead, a sequence of *guard* instructions represents a series of *pending* branch points along a *single* predicted path. Guard instructions have the 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 to 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 missprediction, guards additionally record the sequence 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 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* [Cauligi et al. 2020] shown in Fig. 4b. For example, directive **fetch true** fetches the **true** branch of a conditional and **exec** n executes the n-th instruction in the reorder buffer. Executing an instruction generates an *observation* (Fig. 4b) which records attacker observable behavior. Observations include *speculative* memory reads and writes (i.e., $\mathbf{read}(n, ps)$ and $\mathbf{write}(n, ps)$ issued while the guard and fail instructions identified by ps are pending), rollbacks (i.e., $\mathbf{rollback}(p)$ due to misspeculation of guard p), and memory violations (i.e., $\mathbf{fail}(p)$ due to instruction $\mathbf{fail}(p)$). Most instructions generate the *silent* observation ϵ. Like [Cauligi et al. 2020], we do not include observations about branch directions. Doing so would *not* increase the power of the attacker: a constant-time program that leaks through the branch direction would also leak through the presence or absence of rollback events, i.e., different predictions produce different rollback events, capturing leaks due to branch direction.

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 representing the current *execution path*, a memory μ, and a map ρ from variables to values. 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, 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

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

Fetch-If-True

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

Fig. 5. Fetch stage (selected rules).

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 primitive.

3.1 Fetch Stage

The fetch stage flattens the input commands into a sequence of instructions which it stores in the reorder buffer. Figure 5 presents selected rules; the remaining rules are in the extended version of this paper [Vassena et al. 2020]. Rule [Fetch-Seq] pops command $c_1; c_2$ from the commands 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.[4] 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-Array-Load] expands command $x := a[e]$ into the corresponding pointer dereference, but guards the command with a bounds-check condition. First, the rule generates the condition $e_1 = e < length(a)$ and calculates the address of the indexed element $e_1 = base(a) + e$. Then, it replaces the array read on the stack with command **if** e_1 **then** $x := *e_2$ **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 the current commands stack (i.e., $c_2 : cs$) as a rollback stack to restart the execution in case of misprediction.

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

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

(a) Transient variable map.

Execute

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

(b) Execute rule.

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-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-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

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

(c) Auxiliary relation (selected rules).

Fig. 6. Execute stage.

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. Since instructions can be executed out-of-order, when we evaluate operands 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 this end, we define a function $\phi(\rho, is)$, called the *transient variable map* (Fig. 6a), which updates variable map ρ with the pending assignments in reorder buffer is. 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., $x := e$, $x := \mathbf{load}(e)$, and $x := \mathbf{protect}(r)$) as *undefined* ($\rho[x \mapsto \bot]$), making their respective values unavailable to following instructions.

Execute Rule and Auxiliary Relation. Figure 6 shows selected rules for the execute stage. Rule [Execute] executes the nth instruction in the reorder buffer, following the directive **exec** n. For this, the rule splits the reorder buffer into prefix is_1, nth instruction 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. 6c. 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 $\bot$. 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.

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 ($x := \mu(n)$). By inserting the resolved assignment into the reorder buffer, the rule allows the processor to transiently forward the loaded value to later instructions through the transient variable map. To record that the load is issued *speculatively*, the observation $\mathbf{read}(n, ps)$ stores list ps containing the identifiers of the guard and fail instructions still pending in the reorder buffer. Function $(\!|is_1|\!)$ simply extracts these identifiers from the guard and fail instructions in prefix is_1. In the rule, premise $\mathbf{store}(\_, \_) \notin is_1$ prevents the processor from reading potentially stale data from memory: if the load aliases with a preceding (but pending) store, ignoring the store could produce a stale read.

Store Forwarding. Alternatively, instead of exclusively reading fresh values from memory, it would be also possible to *forward* values waiting to be written to memory at the same address by a pending, aliasing store. For example, if buffer is_1 contained a resolved instruction $\mathbf{store}(n, v)$, rule [Exec-Load] could *directly* propagate the value v to the load, which would be resolved to $x := v$, without reading memory and thus generating silent event ϵ instead of $\mathbf{read}(n, ps)$.[5] The Spectre v1.1 variant relies on this particular microarchitectural optimization to leak data—for example by speculatively writing transient data out of the bounds of an array, forwarding that data to an aliasing load.

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 ($[\![e]\!]^{\rho} = b$), and thus the processor only replaces the guard with **nop**. In contrast, in rule [Exec-Branch-Mispredict] the predicted and computed value differ ($[\![e]\!]^{\rho} = b'$ and $b' \neq b$). This causes the processor to revert the program state and issue a rollback observation ($\mathbf{rollback}(p)$). 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.

3.3 Retire Stage

The retire stage removes completed instructions from the reorder buffer and propagates their changes to the 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. For this reason, the rules for the retire stage in Figure 7 always remove the *first* instruction in the reorder buffer. For example, rule [Retire-Nop] removes **nop** from the front of the reorder buffer. Rules [Retire-Asgn]

[5] Capturing the semantics of *store-forwarding* would additionally require some bookkeeping about the freshness of (forwarded) values in order to detect reading stale data. We refer the interested reader to [Cauligi et al. 2020] for full details.

$$\textsc{Retire-Nop} \quad \langle \mathbf{nop} : is, cs, \mu, \rho \rangle \xrightarrow{\mathbf{retire}}_{\epsilon} \langle is, cs, \mu, \rho \rangle$$

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

$$\textsc{Retire-Store} \quad \langle \mathbf{store}(n, v) : is, cs, \mu, \rho \rangle \xrightarrow{\mathbf{retire}}_{\epsilon} \langle is, cs, \mu[n \mapsto v], \rho \rangle$$

$$\textsc{Retire-Fail} \quad \langle \mathbf{fail}(p) : is, cs, \mu, \rho \rangle \xrightarrow{\mathbf{retire}}_{\mathbf{fail}(p)} \langle [\,], [\,], \mu, \rho \rangle$$

Fig. 7. Retire stage.

Memory Layout

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

Variable Map

$\rho(i_1) = 1$

$\rho(i_2) = 2$

$\dots$

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

Fig. 8. Leaking execution of running program Example.

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 ($\mu[n \mapsto v]$) respectively. Rule [Retire-Fail] aborts the program by emptying reorder buffer and command stack and generates a $\mathbf{fail}(p)$ observation, simulating a processor raising an exception (e.g., a segmentation fault).

Example. We demonstrate how the attacker can leak a secret from program Example (Fig. 2) in our model. First, the attacker instructs the processor to fetch all the instructions, supplying prediction **true** for all bounds-check conditions. Figure 8 shows the resulting buffer and how it evolves after each attacker directive; the memory μ and variable map ρ are shown above. The attacker directives instruct the processor to speculatively execute the load instructions and the assignment (but not the guard instructions). Directive **exec** 2 executes the first load instruction by computing the memory address $2 = \llbracket base(a) + i_1 \rrbracket^{\rho}$ and replacing the instruction with the assignment $x := \mu(2)$ containing the loaded value. Directive **exec** 4 transiently reads *public* 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 through the *transient* variable map $\rho[x \mapsto \mu(2), y \mapsto \mu(3)]$ 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 Protect

Next, we turn to the rules that formalize the semantics of **protect** as an ideal hardware primitive and then its software implementation via speculative-load-hardening (SLH).

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

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

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

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

(a) Semantics of **protect** as a hardware primitive (selected rules).

$$\textsc{Fetch-Protect-SLH}\quad \frac{\begin{array}{cccc} & c = x := \textbf{protect}(a[e]) & e_1 = e < \mathit{length}(a) & e_2 = \mathit{base}(a) + e \\ c_1 = m := e_1 & c_2 = m := m\,?\,\mathbf{1} : \mathbf{0} & c_3 = x := *(e_2 \otimes m) & c' = c_1; \textbf{if } m \textbf{ then } c_2; c_3 \textbf{ else fail}\end{array}}{\langle is, c : cs, \mu, \rho\rangle \xrightarrow{\textbf{fetch}}_{\epsilon} \langle is, c' : cs, \mu, \rho\rangle}$$

(b) Software implementation of **protect**($a[e]$).

Fig. 9. Semantics of **protect**.

Protect in Hardware. Instruction x:=**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 9a formalizes this intuition. Rule [Fetch-Protect-Expr] fetches protect commands involving simple expressions (x := **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 := **protect**($a[e]$)) into a separate assignment of the array value ($x := a[e]$) and protect of the variable (x := **protect**(x)). Rules [Exec-Protect$_1$] and [Exec-Protect$_2$] extend the auxiliary relation $\rightsquigarrow$. Rule [Exec-Protect$_1$] evaluates the expression ($v = [\![e]\!]^{\rho}$) 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 := **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 (**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 Example and the execution shown in Figure 8. In the repaired program, $x + y$ is wrapped in a **protect** statement. As a result, directive **exec** 5 produces value z:=**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 leaking the transient value.

Protect in Software. The software implementation of **protect** applies SLH to array reads. Intuitively, we rewrite array reads by injecting *artificial* data-dependencies between bounds-check conditions and the corresponding addresses in load instructions, thus transforming control-flow dependencies into data-flow dependencies.[6] These data-dependencies validate control-flow decisions at runtime by stalling speculative loads until the processor resolves their bounds check conditions.[7] Formally, we replace rule [Fetch-Protect-Array] with rule [Fetch-Protect-SLH] in Figure 9b. The rule computes the bounds check condition $e_1 = e < \mathit{length}(a)$, the target address $e_2 = \mathit{base}(a) + e$, and generates commands that abort the execution if the check fails, like for regular array reads. Additionally, the rule generates *regular commands* that (i) assign the result of the bounds check to a reserved variable m ($c_1 = m := e_1$), (ii) conditionally update the variable with a bitmask consisting of all 1s or 0s ($c_2 = m := m\ ?\ \mathbf{1} : \mathbf{0}$), and (iii) mask off the target address with the bitmask ($c_3 = x := *(e_2 \otimes m)$).[8] Since the target address in command c_3 *depends* on variable m, the processor cannot read memory until the bounds check is resolved. If the check succeeds, the bitmask $m = \mathbf{1}$ leaves the target address unchanged ($[\![e']\!]^\rho = [\![e' \otimes \mathbf{1}]\!]^\rho$) and the processor reads the correct address normally. Otherwise, the bitmask $m = \mathbf{0}$ zeros out the target address and the processor loads speculatively only from the constant address $0 = [\![e' \otimes \mathbf{0}]\!]^\rho$. (We assume that the processor reserves the first memory cell and initializes it with a dummy value, e.g., $\mu(0) = 0$.) Notice that this solution works under the assumption that the processor does not evaluate the conditional update $m := m\ ?\ \mathbf{1} : \mathbf{0}$ speculatively. We can easily enforce that by compiling conditional updates to non-speculative instructions available on commodity processors (e.g., the conditional move instruction CMOV on x86).

Example. Consider again Example. The optimal patch $\mathbf{protect}(x+y)$ cannot be executed on existing processors without support for a generic **protect** primitive. Nevertheless, we can repair the program by applying SLH to the individual array reads, i.e., $x := \mathbf{protect}(a[i_1])$ and $y := \mathbf{protect}(a[i_2])$.

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. These speculative leaks arise in programs as transient data flows from source to sink expressions. Our type system does *not* rely on user annotations to identify secret. Indeed, our typing rules simply ignore security annotations and instead, conservatively reject programs that exhibit any source-to-sink data flows. Intuitively, this is because security annotations are not trustworthy when programs are executed speculatively: public variables could contain transient secrets and secret variables could flow speculatively into public sinks. Additionally, our soundness theorem, which states that well-typed programs are *speculatively* constant-time, assumes that programs are sequentially constant-time. We do *not* enforce a *sequential* constant-time discipline; this can be done using existing constant-time type systems or interfaces (e.g., [Watt et al. 2019; Zinzindohoué et al. 2017]).

Given an unannotated program, we apply constraint-based type inference [Aiken 1996; Nielson and Nielson 1998] 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

[6] Technically, applying SLH to array reads is a program transformation. In our implementation (§ 6), the SLH version of **protect** inserts additional instructions into the program to perform the conditional update and mask operation described in rule [Fetch-Protect-SLH].

[7] A fully-fledged security tool could apply static analysis techniques to infer array bounds. Our implementation of Blade merely simulates SLH using a static constant instead of the actual lengths, as discussed in Section 6.

[8] Alternatively, it would be also possible to mask the loaded value, i.e., $c_3 = x := (*e_2) \otimes m$. However, this alternative mitigation would still introduce transient data in various processor internal buffers, where it could be leaked. In contrast, we conservatively mask the address, which has the effect of stalling the load and preventing transient data from even entering the processor, thus avoiding the risk of leaking it altogether.

$$\textsc{Value}\quad \Gamma \vdash v : \tau \;\boxed{\Rightarrow\ \emptyset}$$

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

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

$$\textsc{Array-Read}\quad \frac{\Gamma \vdash e : \mathbf{S} \;\boxed{\Rightarrow\ k}}{\Gamma \vdash a[e] : \mathbf{T} \;\boxed{\Rightarrow\ k \cup (e \sqsubseteq \mathbf{S}) \cup (\mathbf{T} \sqsubseteq a[e])}}$$

(a) Typing rules for expressions and arrays.

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

$$\textsc{Protect}\quad \frac{\Gamma \vdash r : \tau \;\boxed{\Rightarrow\ k}}{\Gamma, \mathrm{Prot} \vdash x := \mathbf{protect}(r) \;\boxed{\Rightarrow\ k}}$$

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

$$\textsc{Array-Write}\quad \frac{\Gamma \vdash e_1 : \mathbf{S} \;\boxed{\Rightarrow k_1} \qquad \Gamma \vdash e_2 : \tau \;\boxed{\Rightarrow k_2}}{\Gamma, \mathrm{Prot} \vdash a[e_1] := e_2 \;\boxed{\Rightarrow k_1 \cup k_2 \cup (e_1 \sqsubseteq \mathbf{S})}}$$

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

(b) Typing rules for commands.

Fig. 10. Transient-flow type system and constraints generation.

repair algorithm we patch the program by inserting a minimum number of **protect** so that it cannot leak speculatively anymore (Section 4.3).

4.1 Type System

Our 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 [Landauer and Redmond 1993], 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 $\mathbf{T} \not\sqsubseteq \mathbf{S}$.

Typing Expressions. Given a typing environment for variables $\Gamma \in Var \rightarrow \{\mathbf{S}, \mathbf{T}\}$, the typing judgment $\Gamma \vdash r : \tau$ assigns a transient-flow type τ to r. Figure 10 presents selected rules (see [Vassena et al. 2020] for the rest). The shaded part of the rules generates type constraints during type inference and are explained later. Values can assume any type in rule [Value] and variables are assigned their respective type from the environment in rule [Var]. Rule [Bop] propagates the type of the operands to the result of binary operators $\oplus \in \{+, <, \otimes\}$. Finally, rule [Array-Read] assigns the *transient* type $\mathbf{T}$ to array reads as the array may potentially be indexed out of bounds during speculation. Importantly, the rule requires the index expression to be typed *stable* ($\mathbf{S}$) to prevent programs from leaking through the cache.

Typing Commands. Given a set of implicitly 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 10b 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.[9] 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. Rule [Array-Write] requires the index expression used in array writes to be typed *stable* ($\mathbf{S}$) to avoid leaking through the cache, similarly to rule [Array-Read]. Notice that the rule does not prevent storing transient data, i.e., the stored value can have any type. While this is sufficient to mitigate Spectre v1 attacks, it is inadequate to defend against Spectre v1.1. Intuitively, the store-to-load forward optimization enables additional *implicit* data-flows between stored data to aliasing load instructions, thus enabling Spectre v1.1 attacks. Luckily, to protect against these attacks we need only modify one clause of rule [Array-Write]: we change the type system to conservatively treat stored values as *sinks* and therefore require them to be typed *stable* ($\Gamma \vdash e_2 : \mathbf{S}$).[10]

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., [Volpano et al. 1996]) 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, upon a rollback, the processor may *repeat* some load and store instructions after the mispredicted branch and thus generate additional observations, which can implicitly reveal the value of the branch condition.

Example. Consider the program $\{\textbf{if } tr \textbf{ then } x := 0 \textbf{ else skip}\}; y := a[0]$. The program can leak the transient 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 observation 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 *again*, 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 $\mathbf{T}$, our type system rejects the program as unsafe.

4.2 Type Inference

Our type-inference approach is based on type-constraints satisfaction [Aiken 1996; Nielson and Nielson 1998]. Intuitively, type constraints restrict the types that variables and expressions may

[9] Readers familiar with information-flow control may see an analogy between **protect** and the **declassify** primitive of some IFC languages [Myers et al. 2004].

[10] Both variants of rule [Array-Write] are implemented in Blade (§ 6) and evaluated (§ 7).

assume in a program. In the constraints, the possible types of variables and expressions are represented by *atoms* consisting of unknown types of expressions and variables. Solving these constrains requires finding a *substitution*, i.e., a mapping from atoms to concrete transient-flow types, such that all constraints are *satisfied* if we instantiate the atoms with their type.

Our type inference algorithm consists of 3 steps: (i) generate a set of type constraints under an initial typing environment and protected set that under-approximates the solution of the constraints, (ii) construct the def-use graph from the constraints and find a cut-set, and (iii) cut the transient-to-stable dataflows in the graph and compute the resulting typing environment. We start by describing the generation of constraints through the typing judgment from Figure 10.

Type Constraints. Given a typing environment Γ, a protected set Prot, the judgment $\Gamma, \text{Prot} \vdash r \Rightarrow k$ type checks r and generates type constraints k. The syntax for constraints is shown in Figure 11. Constraints are sets of can-flow-to relations involving concrete types (**S** and **T**) and *atoms*, i.e., type variables corresponding to program variables (e.g., α_x for x) and unknown types for expressions (e.g., r). In rule [Var], constraint $x \sqsubseteq \alpha_x$ indicates that the type variable of x should be at least as transient as the unknown type of x. This ensures that, if variable x is transient, then α_x can only be instantiated with type **T**. Rule [Bop] generates constraints $e_1 \sqsubseteq e_1 \oplus e_2$ and $e_2 \sqsubseteq e_1 \oplus e_2$ to reflect the fact that the unknown type of $e_1 \oplus e_2$ should be at least as transient as the (unknown) type of e_1 and e_2. Notice that these constraints correspond exactly to the premises $\tau_1 \sqsubseteq \tau$ and $\tau_2 \sqsubseteq \tau$ of the same rule. Similarly, rule [Array-Read] generates constraint $e \sqsubseteq \mathbf{S}$ for the unknown type of the array index, thus forcing it to be typed **S**.[11] In addition to these, the rule generates also the constraint $\mathbf{T} \sqsubseteq a[e]$, which forces the type of $a[e]$ to be **T**. Rule [Asgn] generates the constraint $r \sqsubseteq x$ disallowing transient to stable assignments. In contrast, rule [Protect] does *not* generate the constraint $r \sqsubseteq x$ because r is explicitly protected. Rule [Asgn-Prot] generates the same constraint as rule [Asgn], because type inference ignores the protected set, which is computed in the next step of the algorithm. The constraints generated by the other rules follow the same intuition. In the following we describe the inference algorithm in more detail.

Generating 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.

Solutions and Satisfiability. We define the solution to a set of constraints as a function σ from atoms to flow types, i.e., $\sigma \in \text{Atoms} \mapsto \{\mathbf{T}, \mathbf{S}\}$, and extend solutions to map **T** and **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 (Fig. 11). 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 $a_1 \dots a_n$ a *path* in k, if $a_i \sqsubseteq a_{i+1} \in k$ for $i \in \{1, \dots, n-1\}$ and say that a_1 is the path's entry and a_n its exit. A **T**-**S** path is a path with entry **T** and exit **S**. A set of constraints k is satisfiable if and only if there is no **T**-**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 **T** and exit a, and **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

[11] No constraints are generated for the type of the array because our syntax forces the array to be a *static*, constant value. If we allowed arbitrary expressions for arrays the rule would require them to by typed as stable.

$$\begin{array}{rrl} \text{Atoms } a & ::= & \alpha_x \mid r \\ \text{Constraints } k & ::= & a \sqsubseteq \mathbf{S} \mid \mathbf{T} \sqsubseteq a \mid a \sqsubseteq a \\ & & \mid k \cup k \mid \emptyset \\ \text{Solutions } \sigma & \in & \text{ATOMS} \uplus \{\mathbf{S},\mathbf{T}\} \mapsto \{\mathbf{S},\mathbf{T}\} \\ & & \text{where } \sigma(\mathbf{S}) = \mathbf{S} \text{ and } \sigma(\mathbf{T}) = \mathbf{T} \end{array}$$

$$\textsc{Sol-Empty}\ \ \sigma \vdash \emptyset \qquad \textsc{Sol-Set}\ \ \frac{\sigma \vdash k_1 \quad \sigma \vdash k_2}{\sigma \vdash k_1 \cup k_2} \qquad \textsc{Sol-Trans}\ \ \frac{\mathbf{T} \sqsubseteq \sigma(a_2)}{\sigma \vdash \mathbf{T} \sqsubseteq a_2}$$

$$\textsc{Sol-Stable}\ \ \frac{\sigma(a_1) \sqsubseteq \mathbf{S}}{\sigma \vdash a_1 \sqsubseteq \mathbf{S}} \qquad \textsc{Sol-Flow}\ \ \frac{\sigma(a_1) \sqsubseteq \sigma(a_2)}{\sigma \vdash a_1 \sqsubseteq a_2}$$

Fig. 11. Type constraints and satisfiability.

$a_1 \ldots a_n$, 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 **T**-**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 **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* $\Gamma(k, A), A \vdash c$.

Remark. To infer a repair using exclusively SLH-based **protect** statements, we simply restrict our cut-set to only include variables that are assigned from an array read.

Example. Consider again EXAMPLE in Figure 2. The graph defined by the constraints k, given by $\Gamma^*, \text{Prot}^* \vdash \text{EXAMPLE} \ \Rightarrow k$ is shown in Figure 3, where we have omitted α-nodes. The constraints are not satisfiable, since there are **T**-**S** paths. Both $\{x, y\}$ and $\{z\}$ are cut-sets, since they cut each **T**-**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 **S**, while the typing extracted from the optimal cut, i.e., $\Gamma(k, \{z\})$ types x and y as **T** and z, i_1 and i_2 as **S**. By Proposition 1 both $\Gamma(k, \{x, y\}), \{x, y\} \vdash \text{EXAMPLE}$ and $\Gamma(k, \{z\}), \{z\} \vdash \text{EXAMPLE}$ hold.

4.3 Program Repair

As a final step, our repair algorithm *repair*(c, Prot) traverses program c and inserts a **protect** statement for each variable in the cut-set Prot. For simplicity, we assume that programs are in static single assignment (SSA) form.[12] Therefore, for each variable $x \in \text{Prot}$ there is a single assignment $x := r$, and our repair algorithm simply replaces it with $x := \mathbf{protect}(r)$.

5 CONSISTENCY AND SECURITY

We now present two formal results about our speculative semantics and the security of our type system. First, we prove that the semantics from Section 3 is *consistent* with sequential program execution (Theorem 1). Intuitively, programs running on our processor produce the same results (with respect to the memory store and variables) as if their commands were executed in-order and without speculation. The second result establishes that our type system is *sound* (Theorem 2). We prove that our transient-flow type system in combination with a standard constant-time type system (e.g., [Protzenko et al. 2019; Watt et al. 2019]) enforces constant time under speculative

[12]We make this assumption to simplify our analysis and security proof. We also omit phi-nodes from our calculus to avoid cluttering the semantics and remark that this simplification does not affect our *flow-insensitive* analysis. Our implementation operates on Cranelift's SSA form (§ 6).

execution [Cauligi et al. 2020]. We provide full definitions and proofs in the extended version of this paper [Vassena et al. 2020].

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 misspeculated loads and stores.

Theorem 1 (Consistency). *For all programs c, initial memory stores μ, variable maps ρ, and directives D, if $\langle \mu, \rho \rangle \Downarrow_O^c \langle \mu', \rho' \rangle$ 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 Constant Time. In our model, an attacker can leak information through the architectural state (i.e., the variable map and the memory store) and through the cache by supplying directives that force the execution of an otherwise constant-time cryptographic program to generate different traces. In the following, the relation $\approx_L$ denotes L-equivalence, i.e., equivalence of configurations with respect to a security policy L that specifies which variables and arrays are public (L) and attacker observable. Initial and final configurations are L-equivalent ($C_1 \approx_L C_2$) if the values of public variables in the variable maps and the content of public arrays in the memories coincide, i.e., $\forall\ x \in L\,.\ \rho_1(x) = \rho_2(x)$ and $\forall\ a \in L$ and addresses $n \in \{\, base(a), \ldots, base(a) + length(a) - 1\,\}$, $\mu_1(n) = \mu_2(n)$, respectively.

Definition 1 (Speculative Constant Time). *A program c is speculative constant time with respect to a security policy L, written $SCT_L(c)$, iff for all directives D and initial configurations $C_i = \langle [\,], [\,c\,], \mu_i, \rho_i \rangle$ for $i \in \{1, 2\}$, if $C_1 \approx_L C_2$, $C_1 \Downarrow_{O_1}^D C_1'$, and $C_2 \Downarrow_{O_2}^D C_2'$, then $O_1 = O_2$ and $C_1' \approx_L C_2'$.*

In the definition above, we consider syntactic equivalence of traces because both executions follow the same list of directives. We now present our soundness theorem: well-typed programs satisfy speculative constant-time. Our approach focuses on side-channel attacks through the observation trace and therefore relies on a separate, but standard, type system to control leaks through the program control-flow and architectural state. In particular, we write $CT_L(c)$ if c follows the (sequential) constant time discipline from [Protzenko et al. 2019; Watt et al. 2019], i.e., it is free of secret-dependent branches and memory accesses.

Theorem 2 (Soundness). *For all programs c and security policies L, if $CT_L(c)$ and $\Gamma \vdash c$, then $SCT_L(c)$.*

As mentioned in Section 4, our transient flow-type system is oblivious to the security policy L, which is only required by the constant-time type system and the definition of speculative constant time.

We conclude with a corollary that combines all the components of our protection chain (type inference, type checking and automatic repair) and shows that repaired programs satisfy speculative constant time.

Corollary 1. *For all sequential constant-time programs $CT_L(c)$, 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, it follows that $SCT_L(repair(c, A))$.*

6 IMPLEMENTATION

We implement Blade as a compilation pass in the Cranelift [Bytecode Alliance 2020] Wasm code-generator, which is used by the Lucet compiler and runtime [McMullen 2020]. Blade first identifies all sources and sinks. Then, it finds the cut points using the Max-Flow/Min-Cut algorithm (§4.2), and either inserts fences at the cut points, or applies SLH to all of the loads which feed the cut point in the graph. This difference is why SLH sometimes requires code insertions in more locations.

Our SLH prototype implementation does not track the length of arrays, and instead uses a static constant for all array lengths when applying masking. Once compilers like Clang add support for conveying array length information to Wasm (e.g., via Wasm's custom section), our compilation pass would be able to take this information into account. This simplification in our experiments does not affect the sequence of instructions emitted for the SLH masks and thus Blade's performance overhead is accurately measured.

Our Cranelift Blade pass runs after the control-flow graph has been finalized and right before register allocation.[13] Placing Blade before register allocation allows our implementation to remain oblivious of low-level details such as register pressure and stack spills and fills. Ignoring the memory operations incurred by spills and fills simplifies Blade's analysis and reduces the required number of **protect** statements. This, importantly, does not compromise the security of its mitigations: In Cranelift, spills and fills are always to constant addresses which are inaccessible to ordinary Wasm loads and stores, even speculatively. (Cranelift uses guard pages—not conditional bounds checks—to ensure that Wasm memory accesses cannot access anything outside the linear memory, such as the stack used for spills and fills.) As a result, we can treat stack spill slots like registers. Indeed, since Blade runs before register allocation, it already traces def-use chains across operations that will become spills and fills. Even if a particular spill-fill sequence would handle potentially sensitive transient data, Blade would insert a **protect** between the original transient source and the final transient sink (and thus mitigate the attack).

Our implementation implements a single optimization: we do not mark constant-address loads as transient sources. We assume that the program contains no loads from out-of-bounds constant addresses, and therefore that loads from constant (Wasm linear memory) addresses can never speculatively produce invalid data. As we describe below, however, we omit this optimization when considering Spectre v1.1.

At its core, our repair algorithm addresses Spectre v1 attacks based on PHT mispredictions. To also protect against Spectre variant 1.1 attacks, which exploit store forwarding in the presence of PHT mispredictions,[14] we perform two additional mitigations. First, we mark constant-address loads as transient sources (and thus omit the above optimization). Under Spectre v1.1, a load from a constant address may speculatively produce transient data, if a previous speculative store wrote transient data to that constant address—and, thus, Blade must account for this. Second, our SLH implementation marks all stored *values* as sinks, essentially preventing any transient data from being stored to memory. This is necessary when considering Spectre v1.1 because otherwise, ensuring that a load is in-bounds using SLH is insufficient to guarantee that the produced data is not transient—again, a previous speculative store may have written transient data to that in-bounds address.

[13] More precisely: The Cranelift register allocation pass modifies the control-flow graph as an initial step; we insert our pass after this initial step but before register allocation proper.

[14] Spectre v1 and Spectre v1.1 attacks are both classified as Spectre-PHT attacks [Canella et al. 2019].

7 EVALUATION

We evaluate Blade by answering two questions: ***(Q1)*** How many **protect**s does Blade insert when repairing existing programs? ***(Q2)*** What is the runtime performance overhead of eliminating speculative leaks with Blade on existing hardware?

Benchmarks. We evaluate Blade on existing cryptographic code taken from two sources. First, we consider two cryptographic primitives from CT-Wasm [Watt et al. 2019]:

- ▹ The Salsa20 stream cipher, with a workload of 64 bytes.
- ▹ The SHA-256 hash function, with workloads of 64 bytes (one block) or 8192 bytes (128 blocks).

Second, we consider automatically generated cryptographic primitives and protocols from the HACL* [Zinzindohoué et al. 2017] library. We compile the automatically generated C code to Wasm using Clang's Wasm backend. (We do not use HACL*'s Wasm backend since it relies on a JavaScript embedding environment and is not well suited for Lucet.) Specifically, from HACL* we consider:

- ▹ The ChaCha20 stream cipher, with a workload of 8192 bytes.
- ▹ The Poly1305 message authentication code, with workloads of 1024 or 8192 bytes.
- ▹ ECDH key agreement using Curve25519.

We selected these primitives to cover different kinds of modern crypto workloads (including hash functions, MACs, encryption ciphers, and public key exchange algorithms). We omitted primitives that had inline assembly or SIMD since Lucet does not yet support either; we also omitted the AES from HACL* and TEA from CT-Wasm—modern processors implement AES in hardware (largely because efficient software implementations of AES are generally not constant-time [Osvik et al. 2006]), while TEA is not used in practice. All the primitives we consider have been verified to be constant-time—free of cache and timing side-channels. However, the proofs assume a sequential execution model and do not account for speculative leaks as addressed in this work.

Experimental Setup. We conduct our experiments on an Intel Xeon Platinum 8160 (Skylake) with 1TB of RAM. The machine runs Arch Linux with kernel `5.8.14`, and we use the Lucet runtime version `0.7.0-dev` (Cranelift version `0.62.0` with our modifications) compiled with `rustc` version `1.46.0`. We collect benchmarks using the Rust `criterion` crate version `0.3.3` [Heisler and Aparicio 2020] and report the point estimate for the mean runtime of each benchmark.

Reference and Baseline Comparisons. We compare Blade to a reference (unsafe) implementation and a baseline (safe) implementation which simply **protect**s every Wasm memory load instruction. We consider two baseline variants: The baseline solution with Spectre v1.1 mitigation **protect**s every Wasm load instruction, while the baseline solution with only Spectre v1 mitigation **protect**s only Wasm load instructions with non-constant addresses. The latter is similar to Clang's Spectre mitigation pass, which applies SLH to each non-constant array read [Carruth 2019]. We evaluate both Blade and the baseline implementation with Spectre v1 protection and with both v1 and v1.1 protections combined. We consider both fence-based and SLH-based implementations of the **protect** primitive. In the rest of this section, we use Baseline-F and Blade-F to refer to fence-based implementations of their respective mitigations and Baseline-S and Blade-S to refer to the SLH-based implementations.

Results. Table 1 summarizes our results. With Spectre v1 protections, both Blade-F and Blade-S insert very few **protect**s and have negligible performance overhead on most of our benchmarks—the geometric mean overheads imposed by Blade-F and Blade-S are 5.0% and 1.7%, respectively. In contrast, the baseline passes insert between 3 and 1862 protections and incur significantly higher overheads than Blade—the geometric mean overheads imposed by Baseline-F and Baseline-S are 80.2% and 19.4%, respectively.

Table 1. **Ref**: Reference implementation with no Spectre mitigations; **Baseline-F**: Baseline mitigation inserting fences; **Blade-F**: Blade using fences as **protect**; **Baseline-S**: Baseline mitigation using SLH; **Blade-S**: Blade using SLH; **Overhead**: Runtime overhead compared to Ref; **Defs**: number of fences inserted (Baseline-F and Blade-F), or number of loads protected with SLH (Baseline-S and Blade-S)

		Without v1.1 protections			With v1.1 protections		
Benchmark	**Defense**	**Time**	**Overhead**	**Defs**	**Time**	**Overhead**	**Defs**
Salsa20 (CT-Wasm), 64 bytes	Ref	4.3 us	-	-	4.3 us	-	-
	Baseline-F	4.6 us	7.2%	3	8.6 us	101.7%	99
	Blade-F	4.4 us	1.9%	0	4.3 us	1.7%	0
	Baseline-S	4.4 us	2.7%	3	5.3 us	24.3%	99
	Blade-S	4.3 us	0.5%	0	5.4 us	26.4%	99
SHA-256 (CT-Wasm), 64 bytes	Ref	13.7 us	-	-	13.7 us	-	-
	Baseline-F	19.8 us	43.8%	23	20.3 us	48.0%	54
	Blade-F	13.8 us	0.2%	0	14.5 us	5.4%	3
	Baseline-S	15.0 us	9.1%	23	15.1 us	10.0%	54
	Blade-S	13.9 us	0.8%	0	15.2 us	10.9%	54
SHA-256 (CT-Wasm), 8192 bytes	Ref	114.6 us	-	-	114.6 us	-	-
	Baseline-F	516.6 us	350.6%	23	632.6 us	451.8%	54
	Blade-F	113.7 us	-0.8%	0	193.3 us	68.6%	3
	Baseline-S	187.4 us	63.4%	23	208.0 us	81.5%	54
	Blade-S	115.2 us	0.5%	0	216.5 us	88.9%	54
ChaCha20 (HACL*), 8192 bytes	Ref	43.7 us	-	-	43.7 us	-	-
	Baseline-F	85.2 us	94.8%	136	85.4 us	95.3%	142
	Blade-F	44.4 us	1.5%	3	45.4 us	3.8%	7
	Baseline-S	52.8 us	20.8%	136	53.3 us	21.9%	142
	Blade-S	43.6 us	-0.3%	3	53.8 us	22.9%	142
Poly1305 (HACL*), 1024 bytes	Ref	5.5 us	-	-	5.5 us	-	-
	Baseline-F	6.3 us	15.9%	133	6.4 us	17.2%	139
	Blade-F	5.5 us	1.4%	3	5.6 us	2.2%	9
	Baseline-S	5.6 us	1.8%	133	5.7 us	4.4%	139
	Blade-S	5.5 us	1.0%	3	5.6 us	2.5%	139
Poly1305 (HACL*), 8192 bytes	Ref	15.1 us	-	-	15.1 us	-	-
	Baseline-F	21.3 us	41.1%	133	21.4 us	41.2%	139
	Blade-F	15.1 us	-0.0%	3	15.2 us	0.8%	9
	Baseline-S	16.2 us	7.2%	133	16.3 us	7.6%	139
	Blade-S	15.2 us	0.7%	3	16.2 us	7.1%	139
ECDH Curve25519 (HACL*)	Ref	354.3 us	-	-	354.3 us	-	-
	Baseline-F	989.8 us	179.3%	1862	1006.4 us	184.0%	1887
	Blade-F	479.9 us	35.4%	235	497.8 us	40.5%	256
	Baseline-S	507.0 us	43.1%	1862	520.4 us	46.9%	1887
	Blade-S	386.1 us	9.0%	1419	516.8 us	45.9%	1887
Geometric means	Ref		-			-	
	Baseline-F		80.2%			104.8%	
	Blade-F		5.0%			15.3%	
	Baseline-S		19.4%			25.8%	
	Blade-S		1.7%			26.6%	

With both v1 and v1.1 protections, Blade-F inserts an order of magnitude fewer protections than Baseline-F, and has correspondingly low performance overhead—the geometric mean overhead of Blade-F is 15.3%, whereas Baseline-F's is 104.8%. The geometric mean overhead of both Blade-S and Baseline-S, on the other hand, is roughly 26%. Unlike Blade-F, Blade-S must mark all stored values as sinks in order to eliminate Spectre v1.1 attacks; for these benchmarks, this countermeasure requires Blade-S to apply protections to every Wasm load, just like Baseline-S. Indeed, we see in the table that Baseline-S and Blade-S make the exact same number of additions to the code.

We make three observations from our measurements. First, and somewhat surprisingly, Blade does not insert any **protect**s for Spectre v1 on any of the CT-Wasm benchmarks. We attribute this to the style of code: the CT-Wasm primitives are hand-written and, moreover, statically allocate variables and arrays in the Wasm linear memory—which, in turn, results in many constant-address loads. This is unlike the HACL* primitives which are written in F*, compiled to C and then Wasm—and thus require between 3 and 235 **protect**s.

Second, the benchmarks with short reference runtimes tend to have overall lower overheads, particularly for the baseline schemes. This is because for short workloads, the overall runtime is dominated by sandbox setup and teardown, which Blade does not introduce much overhead for. In contrast, for longer workloads, the execution of the Wasm code becomes the dominant portion of the benchmark—and exposes the overhead imposed by the different mitigations. We explore the relationship between workload size and performance overhead in more detail in [Vassena et al. 2020].

And third, we observe for the Spectre v1 version that SLH gives overall better performance than fences, as expected. This is true even in the case of Curve25519, where implementing **protect** using SLH (Blade-S) results in a significant increase in the number of protections versus the fence-based implementation (Blade-F). Even in this case, the more targeted restriction of speculation, and the less heavyweight impact on the pipeline, allows SLH to still prevail over the fewer fences. However, this advantage is lost when considering both v1 and v1.1 mitigation: The sharp increase in the number of **protect**s required for v1.1 ends up being slower than using (fewer) fences. A hybrid approach that uses both fences and SLH could potentially outperform both Blade-F and Blade-S.

In reality, though, both versions are inadequate software emulations of what the **protect** primitive should be. Fences take a heavy toll on the pipeline and are far too restrictive of speculation, while SLH pays a heavy instruction overhead for each instance, and can only be applied directly to loads, not to arbitrary cut points. A hardware implementation of the **protect** primitive could combine the best of Blade-F and Blade-S: targeted restriction of speculation, minimal instruction overhead, and only as many defenses as Blade-F, without the inflation in insertion count required by Blade-S.

However, even without any hardware assistance, both versions of the Blade tool provide significant performance gains over the current state of the art in mitigating Spectre v1, and over existing fence-based solutions when targeting v1 or both v1 and v1.1.

8 RELATED WORK

Speculative Execution Semantics. Several semantics models for speculative execution have been proposed recently [Cauligi et al. 2020; Cheang et al. 2019; Disselkoen et al. 2019; Guanciale et al. 2020; Guarnieri et al. 2020; McIlroy et al. 2019]. Of those, [Cauligi et al. 2020] 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—and, indeed, the semantics of most of the other works—are exclusively for low-level assembly-like languages.[15] In contrast, our JIT semantics bridges the gap between high-level commands and low-level instructions, which allows us to reason about speculative execution of source-level imperative programs through straightforward typing rules, while being faithful to low-level microarchitectural details. Moreover, the idea of modeling

[15] The one exception, Disselkoen et al. [2019], present a Spectre-aware relaxed memory model based on pomsets, which is even further abstracted from the microarchitectural features of real processors.

speculative and out-of-order execution using stacks of progressively flattened commands is novel and key to enable source-level reasoning about the low-level effects of speculation.

Detection and Repair. Wu and Wang [2019] detect cache side channels via abstract interpretation by augmenting the program control-flow to accommodate for speculation. Spectector [Guarnieri et al. 2020] and Pitchfork [Cauligi et al. 2020] use symbolic execution on x86 binaries to detect speculative vulnerabilities. Cheang et al. [2019] and Bloem et al. [2019] apply bounded model checking to detect potential speculative vulnerabilities respectively via 4-ways self-composition and taint-tracking. These efforts assume a *fixed* speculation bound, and they focus on vulnerability detection rather than proposing techniques to *repair* vulnerable programs. Furthermore, many of these works consider only *in-order* execution. In contrast, our type system enforces *speculative constant-time* when program instructions are executed *out-of-order* with *unbounded* speculation—and our tool Blade automatically synthesizes repairs. Separately, oo7 [Wang et al. 2018] statically analyzes a binary from a set of untrusted input sources, detecting vulnerable patterns and inserting 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.

Concurrent to our work, Intel proposed a mitigation for a new class of LVI attacks [Intel 2020; Van Bulck et al. 2020]. Like Blade, they implement a compiler pass that analyzes the program to determine an optimal placement of fences to cut source-to-sink data flows. While we consider an abstract, ideal **protect** primitive, they focus on the optimal placement of fences in particular. This means that they optimize the fence placement by taking into account the coarse-grained effects of fences—e.g., one fence providing a speculation barrier for multiple independent data-dependency chains.[16] This also means, however, their approach does not easily transfer to using SLH for cases where SLH would be faster.

Hardware-based Mitigations. To eliminate speculative attacks, several secure hardware designs have been proposed. Taram et al. [2019] propose context-sensitive fencing, a hardware-based mitigation that dynamically inserts fences in the instruction stream when dangerous conditions arise. InvisiSpec [Yan et al. 2018] features a special *speculative buffer* to prevent speculative loads from polluting the cache. STT [Yu et al. 2019] tracks speculative taints *dynamically* inside the processor micro-architecture and stalls instructions to prevent speculative leaks. Schwarz et al. [2020] propose ConTExT, a whole architecture change (applications, compilers, operating systems, and hardware) to eliminate *all* Spectre attacks. Though Blade can benefit from a hardware implementation of **protect**, this work also shows that Spectre-PHT on existing hardware can be automatically eliminated in pure software with modest performance overheads.

9 LIMITATIONS AND FUTURE WORK

Blade only addresses Spectre-PHT attacks and does so at the Wasm-layer. Extending Blade to tackle other Spectre variants and the limitations of operating on Wasm is future work.

Other Spectre Variants. The Spectre-BTB variant [Kocher et al. 2019] mistrains the Branch Target Buffer (BTB), which is used to predict indirect jump targets, to hijack the (speculative) control-flow of the program. Although Wasm does not provide an unrestricted indirect jump instruction, the indirect function call instruction—which is used to call functions registered in a function table—can be abused by an attacker. To address (in-process) Spectre-BTB, we could extend our type system to restrict the values used as indices into the function table to be typed as *stable*.

[16] Unlike our approach, their resulting optimization problem is NP-hard—and only sub-optimal solutions may be found through heuristics.

The other Spectre variant, Spectre-RSB [Koruyeh et al. 2018; Maisuradze and Rossow 2018], abuses the return stack buffer. To mitigate these attacks, we could analyze Wasm code to identify potential RSB over/underflows and insert fences in response, or use mitigation strategies like RSB stuffing [Intel 2018b]. A more promising approach, however, is to use Intel's recent shadow stack, which ensures that returns cannot be speculatively hijacked [Shanbhogue et al. 2019].

Detecting Spectre Gadgets at the Binary Level. Blade operates on Wasm code—or more precisely, on the Cranelift compiler's IR—and can thus miss leaks inserted by the compiler passes that run after Blade—namely, register allocation and instruction selection. Though these passes are unlikely to introduce such leaks, we leave the validation of the generated binary code to future work.

Spectre Resistant Compilation. An alternative to repairing existing programs is to ensure they are compiled securely from the start. Recent works have developed verified constant-time-preserving optimizing compilers for generating correct, efficient, and secure cryptographic code [Almeida et al. 2017; Barthe et al. 2019]. Doing this for speculative constant-time, and understanding which optimizations break the SCT notion, is an interesting direction for future work.

Bounds Information. Blade-S relies on array bounds information to implement the speculative load hardening. For a memory safe language, this information can be made available to Blade when compiling to Wasm (e.g., as a custom section). When compiling languages like C, where arrays bounds information is not explicit, this is harder—and we would need to use program analysis to track array lengths statically [Venet and Brat 2004]. Although such an analysis may be feasible for cryptographic code, it is likely to fall short for other application domains (e.g., due to dynamic memory allocation and pointer chasing). In these cases, we could track array lengths at runtime (e.g., by instrumenting programs [Nagarakatte et al. 2009]) or, more simply, fall back to fences (especially since the overhead of tracking bounds information at runtime is typically high).

10 CONCLUSION

We presented Blade, a fully automatic approach to provably and efficiently eliminate speculation-based leakage in unannotated cryptographic code. Blade statically detects data flows from transient sources to stable sinks and synthesizes a minimal number of fence-based or SLH-based **protect** calls to eliminate potential leaks. Our evaluation shows that Blade inserts an order of magnitude fewer protections than would be added by today's compilers, and that existing crypto primitives repaired with Blade impose modest overheads when using both fences and SLH for **protect**.

ACKNOWLEDGEMENTS

We thank the reviewers and our shepherd Aseem Rastogi for their suggestions and insightful comments. Many thanks to Shravan Narayan, Ravi Sahita, and Anjo Vahldiek-Oberwagner for fruitful discussions. This work was supported in part by gifts from Fastly, Fujitsu, and Cisco; by the NSF under Grant Number CNS-1514435 and CCF-1918573; by ONR Grant N000141512750; by the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity; and, by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

REFERENCES

Alex Aiken. 1996. Constraint-based program analysis. In *Static Analysis*, Radhia Cousot and David A. Schmidt (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–1.

José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-Assurance and High-Speed Cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. Association for Computing Machinery, New York, NY, USA, 1807–1823. https://doi.org/10.1145/3133956.3134078

Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 53–70. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida

AMD. 2018. Software Techniques For Managing Speculation On AMD Processors. https://developer.amd.com/wp-content/resources/Managing-Speculation-on-AMD-Processors.pdf.

Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. 2019. Formal Verification of a Constant-Time Preserving C Compiler. *Proc. ACM Program. Lang.* 4, POPL, Article 7 (Dec. 2019), 30 pages. https://doi.org/10.1145/3371075

Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: Exploiting Speculative Execution Through Port Contention. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. ACM, New York, NY, USA, 785–800. https://doi.org/10.1145/3319535.3363194

Roderick Bloem, Swen Jacobs, and Yakir Vizel. 2019. Efficient Information-Flow Verification Under Speculative Execution. In *Automated Technology for Verification and Analysis*, Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza (Eds.). Springer International Publishing, Cham, 499–514.

Google Security Blog. 2010. Mitigating Spectre with Site Isolation in Chrome. https://security.googleblog.com/2018/07/mitigating-spectre-with-site-isolation.html.

Bytecode Alliance. 2020. Cranelift Code Generator. https://github.com/bytecodealliance/wasmtime/tree/main/cranelift.

Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *Proceedings of the 28th USENIX Conference on Security Symposium (SEC'19)*. USENIX Association, Berkeley, CA, USA, 249–266. http://dl.acm.org/citation.cfm?id=3361338.3361356

Chandler Carruth. 2019. Speculative Load Hardening. https://llvm.org/docs/SpeculativeLoadHardening.html.

Sunjay Cauligi, Craig Disselkoen, Klaus von Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2020. Constant-Time Foundations for the New Spectre Era. In *Proc. ACM Conference on Programming Language Design and Implementation*.

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

Kevin Cheang, Cameron Rasmussen, Sanjit A. Seshia, and Pramod Subramanyan. 2019. A Formal Approach to Secure Speculation. In *Proceedings of the Computer Security Foundations Symposium (CSF)*.

Craig Disselkoen, Radha Jagadeesan, Alan Jeffrey, and James Riely. 2019. The Code That Never Ran: Modeling Attacks on Speculative Evaluation. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. 1238–1255. https://doi.org/10.1109/SP.2019.00047

Daniel Donenfeld. 2020. More Spectre mitigations in MSVC. https://devblogs.microsoft.com/cppblog/more-spectre-mitigations-in-msvc/.

D. R. Ford and D. R. Fulkerson. 2010. *Flows in Networks*. Princeton University Press, USA.

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

Roberto Guanciale, Musard Balliu, and Mads Dam. 2020. InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20)*. Association for Computing Machinery, New York, NY, USA, 1853–1869. https://doi.org/10.1145/3372297.3417246

Marco Guarnieri, Boris Koepf, José Francisco Morales, Jan Reineke, and Andrés Sánchez. 2020. Spectector: Principled Detection of Speculative Information Flows. In *Proc. IEEE Symp. on Security and Privacy (SSP '20)*.

Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 185–200. https://doi.org/10.1145/3062341.3062363

Brook Heisler and Jorge Aparicio. 2020. Criterion.rs: Statistics-driven Microbenchmarking in Rust. https://crates.io/crates/criterion.

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

Intel. 2018a. 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.

Intel. 2018b. Retpoline: A Branch Target Injection Mitigation. https://software.intel.com/security-software-guidance/api-app/sites/default/files/Retpoline-A-Branch-Target-Injection-Mitigation.pdf.

Intel. 2020. An Optimized Mitigation Approach for Load Value Injection. https://software.intel.com/security-software-guidance/insights/optimized-mitigation-approach-load-value-injection.

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

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

Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks Using the Return Stack Buffer. In *Proceedings of the 12th USENIX Conference on Offensive Technologies (WOOT'18)*. USENIX Association, Berkeley, CA, USA, 3–3. http://dl.acm.org/citation.cfm?id=3307423.3307426

J. Landauer and T. Redmond. 1993. A lattice of information. In *[1993] Proceedings Computer Security Foundations Workshop VI*. 65–70. https://doi.org/10.1109/CSFW.1993.246638

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

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

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

Tyler McMullen. 2020. Lucet: A Compiler and Runtime for High-Concurrency Low-Latency Sandboxing. Principles of Secure Compilation (PriSC).

Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. 2020. Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi-medusa

Mozilla Wiki 2018. Security/Sandbox. https://wiki.mozilla.org/Security/Sandbox.

A. C. Myers, A. Sabelfeld, and S. Zdancewic. 2004. Enforcing robust declassification. In *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.* 172–186. https://doi.org/10.1109/CSFW.2004.1310740

Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 245–258. https://doi.org/10.1145/1542476.1542504

Hanne Riis Nielson and Flemming Nielson. 1998. Flow logics for constraint based analysis. In *Compiler Construction*, Kai Koskimies (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 109–127.

Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA conference on Topics in Cryptology (CT-RSA'06)*. Springer-Verlag.

Andrew Pardoe. 2018. Spectre mitigations in MSVC. https://devblogs.microsoft.com/cppblog/spectre-mitigations-in-msvc/.

Jonathan Protzenko, Benjamin Beurdouche, Denis Merigoux, and Karthikeyan Bhargavan. 2019. Formally Verified Cryptographic Web Applications in WebAssembly. In *Security and Privacy*.

Charles Reis, Alexander Moshchuk, and Nasko Oskov. 2019. Site Isolation: Process Separation for Web Sites within the Browser. In *USENIX Security Symposium*.

Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. 2020. ConTExT: A Generic Approach for Mitigating Spectre. In *Proc. Network and Distributed System Security Symposium*. https://doi.org/10.14722/ndss.2020.24271

Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. 2019. Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity. In *International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*.

Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. 2019. Context-Sensitive Fencing: Securing Speculative Execution via Microcode Customization. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 395–410. https://doi.org/10.1145/3297858.3304060

Vadim Tkachenko. 2018. 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/.

Eran Tromer, Dag Arne Osvik, and Adi Shamir. 2010. Efficient Cache Attacks on AES, and Countermeasures. *J. Cryptol.* 23, 1 (Jan. 2010), 37–71. https://doi.org/10.1007/s00145-009-9049-y

Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. 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. See also technical report Foreshadow-NG [Weisse et al. 2018].

Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *41th IEEE Symposium on Security and Privacy (S&P'20)*.

Marco Vassena, Craig Disselkoen, Klaus V. Gleissenthall, Sunjay Cauligi, Rami Gökhan Kici, Ranjit Jhala, Dean Tullsen, and Deian Stefan. 2020. Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade. *CoRR* abs/2005.00294 (2020). arXiv:2005.00294 https://arxiv.org/abs/2005.00294

Arnaud Venet and Guillaume Brat. 2004. Precise and Efficient Static Array Bound Checking for Large Embedded C Programs. *SIGPLAN Not.* 39, 6 (June 2004), 231–242. https://doi.org/10.1145/996893.996869

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

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

Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. 2019. CT-wasm: Type-driven Secure Cryptography for the Web Ecosystem. *Proc. ACM Program. Lang.* 3, POPL, Article 77 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290390

Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. 2018. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. *Technical report* (2018). See also USENIX Security paper Foreshadow [Van Bulck et al. 2018].

Meng Wu and Chao Wang. 2019. Abstract Interpretation Under Speculative Execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, New York, NY, USA, 802–815. https://doi.org/10.1145/3314221.3314647

Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. 2018. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-51)*. IEEE Press, Piscataway, NJ, USA, 428–441. https://doi.org/10.1109/MICRO.2018.00042

Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 719–732. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. 2019. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. 954–968. https://doi.org/10.1145/3352460.3358274

Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL * : A Verified Modern Cryptographic Library. In *ACM Conference on Computer and Communications Security (CCS)*. Dallas, United States. https://hal.inria.fr/hal-01588421