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 system* 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 **protects** 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*, imposing less than 20% overhead even when using fences to implement **protect**.

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

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 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. 2019a].

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 [Microsoft 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.

1. 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; Kririansky 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 := 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].

2. A Type System for Speculation. Our second contribution is an approach to conservatively approximating the dynamic semantics of speculation via a static type system 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.

3. 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
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.   ... int *dst3 = len + base; // Secret-tainted address
6.   ... *dst3 = pad; // Secret-dependent memory access
7.   ... for (i = 0; i < len + ... ) /* Secret-dependent branch */
8.     dst2[i] = 0;
9.   ... }

Fig. 1. Code fragment adopted 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.

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

4 **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: (geomean) 3.8% performance overhead on our benchmarks to defend from Spectre v1, or 11.7% 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—geomean 1.4% overhead—but slower when including Spectre v1.1 protections, imposing 18.4% overhead.
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, adopted (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.

A. 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. 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].

B. 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,
Example

\[
\begin{align*}
    x &:= a[i_1] \\
    y &:= a[i_2] \\
    z &:= x + y \\
    w &:= b[z]
\end{align*}
\]

Example Patched

\[
\begin{align*}
    x &:= \text{protect}(a[i_1]) \\
    y &:= \text{protect}(a[i_2]) \\
    z &:= \text{protect}(x + y) \\
    w &:= b[z]
\end{align*}
\]

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.

\textbf{protect} only stops speculation along a particular data-path. We use \textbf{protect} to patch the program on line 4. Instead of assigning the value \texttt{len} directly from the result of the load, the memory load is guarded by a \textbf{protect} statement. This ensures that the value assigned to \texttt{len} is always guaranteed to use the \texttt{input_len} pointer’s final, nonspeculative value. This single \textbf{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.

\textbf{Implementation of protect}. Our \textbf{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 \textbf{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, \textbf{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 \textbf{protect} could be implemented efficiently in hardware in the future.

Even if future processors implement \textbf{protect}, we still need to address Spectre attacks on existing hardware. Hence, we formalize and implement \textbf{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 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.

\subsection*{2.3 Automatically Repairing Speculative Leaks via \textbf{protect}-Inference}

\textsc{Blade} automatically finds potential speculative leaks and synthesizes a minimal number of \textbf{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 → 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 → a[i_1].

Following the data dependencies along the edges of the def-use graph, we can see that node T is transitively connected to node z, which indicates that z can contain transient data at run-time. To detect insecure uses of transient values, we then extend the graph with the special 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 → 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 protects 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-constants 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 $\mathsf{T}$), if it can contain transient data (i.e., potential secrets) at run-time, and as stable (written as $\mathsf{S}$), 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 $\Gamma$ 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.

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 $\Gamma, \mathsf{Prot} \vdash c \Rightarrow k$ (§ 4.2), which extends the typing judgment from above with (1) a set of protected variables $\mathsf{Prot}$ (the cut-set), and (2) a set of type-constants $k$ (the def-use graph). At a high level, type inference has 3 steps: (i) generate a set of constraints under an initial typing environment and protected set that allow any program to type-check, (ii) construct the def-use graph from the constraints and find a cut-set (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 unpatched program after type inference, we define a typing judgment $\Gamma, \mathsf{Prot} \vdash c$, where unprotected variables are explicitly accounted for in the $\mathsf{Prot}$ set.$^1$ Intuitively, the program is secure if we promise to insert a protect statement for each variable in $\mathsf{Prot}$. To repair programs, we simply honor this promise and insert a protect statement for each variable in the protected set.

$^1$The judgment $\Gamma \vdash c$ is just a short-hand for $\Gamma, \emptyset \vdash c$. 

We now formalize the concepts discussed in the overview, presenting a semantics in this section.

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 observe the effects of these 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 Rosow 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.

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 \( a \). It includes primitives to obtain array length and base address, represented by functions \( \text{length}(\cdot) \) and \( \text{base}(\cdot) \); and its commands include variable assignments, pointer dereferences, array stores, conditionals, and loops.

It also features a special command that helps prevent transient execution attacks: the command

![Fig. 4. Syntax of the calculus.](image-url)
\( x := \text{protect}(r) \) evaluates \( r \) and assigns its value to \( x \), but only after the value is \textit{stable} (i.e., non-transient). Lastly, \textit{fail} triggers a memory violation error (caused by trying to read or write an array out-of-bounds) and aborts the program.

\textbf{JIT-Step Semantics.} Our operational semantics formalizes the execution of source programs on a pipelined processor and thus enables \textit{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 \textit{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 \textit{progressively} flattens structured commands (e.g., if-statements and while loops) into predicted straight-line code and maintains a \textit{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.

\textbf{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 \textit{guard} instructions represents a series of \textit{pending} branch points along a \textit{single} predicted path. Guard instructions have the form \text{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 misspecification, guards additionally record the sequence of commands \( cs \) along the alternative branch.

\textbf{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 \textit{fetch}, \textit{exec}, and \textit{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 \textit{directives} [Cauligi et al. 2020] shown in Fig. 4b. For example, directive \textit{fetch true} fetches the \textit{true} branch of a conditional and \textit{exec n} executes the \( n \)-th instruction in the reorder buffer. Executing an instruction generates an \textit{observation} (Fig. 4b) which records attacker observable behavior. Observations include \textit{speculative} memory reads and writes (i.e., \text{read}(n, ps) and \text{write}(n, ps) issued while guards \( ps \) are pending), rollbacks (i.e., \text{rollback}(p) due to misspeculation of guard \( p \), and memory violations (fail). Most instructions generate the \textit{silent} observation \( \epsilon \).

\textbf{Configurations and Reduction Relation.} We formally specify our semantics as a reduction relation between processor configurations. A configuration \((is, cs, \mu, \rho)\) consists of a queue of in-flight instructions \text{is} the \textit{reorder buffer}, a stack of commands \text{cs} representing the current \textit{execution path}, a memory \( \mu \), and a map \( \rho \) from variables to values. A reduction step \( C \xrightarrow{d} C' \) denotes that, under directive \( d \), configuration \( C \) is transformed into \( C' \) and generates observation \( o \). To execute a program \( c \) with initial memory \( \mu \) and variable map \( \rho \), the processor initializes the configuration with an empty reorder buffer and inserts the program into the command stack, i.e., \([(\emptyset, [c], \mu, \rho)]. 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 \(\mu'\) and variable map \(\rho'\).

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 Appendix A. Rule \(\text{Fetch-Seq}\) pops command \(c_1; c_2\) from the commands stack and pushes the two sub-commands for further processing. \(\text{Fetch-Asgn}\) pops an assignment from the commands stack and appends the corresponding processor instruction \((x := e)\) at the end of the reorder buffer.\(^2\) Rule \(\text{Fetch-Ptr-Load}\) 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 on the stack with command \(\text{Fetch-Array-Load}\) 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 \(e_1 = e < \text{length}(a)\) and calculates the address of the indexed element \(e_1 = \text{base}(a) + e\).\(^3\) Then, it replaces the array read on the stack with command \(\text{if } e_1 \text{ then } x := *e_2 \text{ else fail}\) 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 \(\text{Fetch-If-True}\) fetches a conditional branch from the stack and, following the prediction provided in directive \text{fetch true}, speculates that the condition \(e\) will evaluate to \text{true}. Thus, the processor inserts the corresponding instruction \(\text{guard}(e^{\text{true}}, c_2 : cs, p)\) 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.

---

\(^2\) Notation \([i_1, \ldots, i_n]\) represents a list of \(n\) elements, \(i_1 + i_2\) denotes list concatenation, and \(|is|\) computes the length of the list \(is\).

\(^3\) The syntax for array accesses allows only static, constant arrays values (e.g., \(a[i]\)) to simplify our security analysis. This simplification does not restrict the power of the attacker, who can still access arbitrary memory addresses through the dynamic index expression \(e\).
\[\phi(\rho, [\_]) = \rho\]
\[\phi(\rho, (x := v) : is) = \phi(\rho[x \mapsto v], is)\]
\[\phi(\rho, (x := e) : is) = \phi(\rho[x \mapsto \bot], is)\]
\[\phi(\rho, (x := \text{load}(e)) : is) = \phi(\rho[x \mapsto \bot], is)\]
\[\phi(\rho, (x := \text{protect}(e)) : is) = \phi(\rho[x \mapsto \bot], is)\]
\[\phi(\rho, i : is) = \phi(\rho, is)\]

(a) Transient variable map.

(b) Execute rule.

\[
\text{EXEC-ASGN} \quad i = (x := e) \quad v = [e]^\rho \quad i' = (x := v)
\]
\[
\langle \text{is}_1, i, \text{is}_2, cs \rangle \xrightarrow{\phi} \langle \text{is}_1 + [i'] + \text{is}_2, cs \rangle
\]

\[
\text{EXEC-LOAD} \quad i = (x := \text{load}(e)) \quad \text{store}(-, -) \notin \text{is}_1 \quad n = [e]^\rho \quad ps = ([\text{is}_1]) \quad i' = (x := \mu(n))
\]
\[
\langle \text{is}_1, i, \text{is}_2, cs \rangle \xrightarrow{\phi} \langle \text{is}_1 + [i'] + \text{is}_2, cs \rangle
\]

(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 \(\phi(is, \rho)\), called the transient variable map (Fig. 6a), which updates variable map \(\rho\) 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., \(x := e\), \(x := \text{load}(e)\), and \(x := \text{protect}(r)\)) 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 is1, nth instruction i, and suffix is2. Next, it computes the transient variable map $\phi$ (is1, $\rho$) and executes a transition step under the new map using an auxiliary relation $\leadsto$. 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 $(is1, i, is2, cs)$ consisting of prefix, suffix and current instruction i into a tuple $(is', cs')$ 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 $\rho$. 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 read(n, ps) stores list ps containing the identifiers of the guards still pending in the reorder buffer. Function $(\{is\})$ simply extracts these identifiers from the guard instructions in prefix is1. In the rule, premise store($\bot$, $\bot$) $\neq$ is1 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 is1 contained a resolved instruction 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 e instead of read(n, ps).4 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 (rollback($p$)). For the rollback, the processor discards the instructions past the guard (i.e., is2) 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.

\[\text{Capturing the semantics of store-forwarding would additionally require some bookkeeping about the freshness of (forwarded) values in order to detect reading stale data. We refer the interested reader to } [\text{Cauligi et al. 2020}] \text{ for full details.}\]
For example, rule \texttt{[Retire-Nop]} removes \texttt{nop} from the front of the reorder buffer. Rules \texttt{[Retire-Asgn]} and \texttt{[Retire-Store]} remove the resolved assignment \(x := v\) and instruction \texttt{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 \texttt{[Retire-Fail]} aborts the program by emptying reorder buffer and command stack and generates a \texttt{fail} observation, simulating a processor raising an exception (e.g., a segmentation fault).

\textbf{Example.} We demonstrate how the attacker can leak a secret from program \texttt{EXAMPLE} (Fig. 2) in our model. First, the attacker instructs the processor to fetch all the instructions, supplying prediction \texttt{true} for all bounds-check conditions. Figure 8 shows the resulting buffer and how it evolves after each attacker directive; the memory \(\mu\) and variable map \(\rho\) are shown above. The attacker directives instruct the processor to speculatively execute the load instructions and the assignment (but not the guard instructions). Directive \texttt{exec 2} executes the first load instruction by computing the memory address \(2 = \lceil \text{base}(a) + i_1 \rceil^\rho\) and replacing the instruction with the assignment \(x := \mu(2)\) containing the loaded value. Directive \texttt{exec 4} transiently reads \texttt{public} array \(a\) past its bound, at index 2, reading into the memory \((\mu(3) = 42)\) of \texttt{secret} array \(s[0]\) and generates the corresponding observation. Finally, the processor and forwards the values of \(x\) and \(y\) through the \texttt{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 \texttt{read(42, [1, 2, 3])}.  

\begin{figure}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
\textbf{Reorder Buffer} & \texttt{exec 2} & \texttt{exec 4} & \texttt{exec 5} & \texttt{exec 7} \\
\hline
1 & \texttt{guard((i_1 < length(a))true, [fail], 1)} & \texttt{x := \mu(2)} & \texttt{y := \mu(3)} & \texttt{z := 42} \\
2 & \texttt{x := load(base(a) + i_1)} & \texttt{y := \mu(3)} & \texttt{z := 42} & \texttt{w := \mu(42)} \\
3 & \texttt{guard((i_2 < length(a))true, [fail], 2)} & \texttt{y := \mu(3)} & \texttt{z := 42} & \texttt{w := \mu(42)} \\
4 & \texttt{y := load(base(a) + i_2)} & \texttt{z := 42} & \texttt{w := \mu(42)} & \texttt{w := \mu(42)} \\
5 & \texttt{z := x + y} & \texttt{z := 42} & \texttt{w := \mu(42)} & \texttt{w := \mu(42)} \\
6 & \texttt{guard((z < length(b))true, [fail], 3)} & \texttt{z := 42} & \texttt{w := \mu(42)} & \texttt{w := \mu(42)} \\
7 & \texttt{w := load(base(b) + z)} & \texttt{w := \mu(42)} & \texttt{w := \mu(42)} & \texttt{w := \mu(42)} \\
\hline
\end{tabular}
\caption{Leaking execution of running program \texttt{EXAMPLE}.}
\end{figure}
Consider again the instruction into a normal assignment, so that the processor can propagate and commit its value. The guards are pending in the reorder buffer (\(\text{guard}(\_\_\_ \neq \text{is}_1)\)). When no leaves the value wrapped inside the protect instruction in the reorder buffer, i.e., \(\text{exec} = \text{protect}(v)\), the instruction in the reorder buffer as if it were a normal assignment. However, the processor \(\Rightarrow \) \(\Rightarrow \rightarrow \rightarrow \) the auxiliary relation \(\Rightarrow \rightarrow \rightarrow \rightarrow \) and protect of the variable (\(\Rightarrow \Rightarrow \rightarrow \rightarrow \) of an array read (\(\Rightarrow \Rightarrow \rightarrow \rightarrow \) involving simple expressions (\(\Rightarrow \Rightarrow \rightarrow \rightarrow \) possible. Figure 9a formalizes this intuition. Rule \(\Rightarrow \rightarrow \rightarrow \rightarrow \) transforms \(\Rightarrow \Rightarrow \rightarrow \rightarrow \) protects \(\Rightarrow \Rightarrow \rightarrow \rightarrow \) the reorder buffer. Rule \(\Rightarrow \rightarrow \rightarrow \rightarrow \) fetches \(\Rightarrow \Rightarrow \rightarrow \rightarrow \) protects \(\Rightarrow \Rightarrow \rightarrow \rightarrow \) and inserts the corresponding protect instruction in the reorder buffer.

\[
\begin{align*}
\text{Fetch-Protect-Array} & \quad c = (x := \text{protect}(a[e])) \\
& \quad \quad c_1 = (x := a[e]) \\
& \quad \quad c_2 = (x := \text{protect}(x)) \\
& \quad \quad \langle \text{is}, c : cs, \mu, \rho \rangle \xrightarrow{\text{fetch}} \langle \text{is}, c_1 : c_2 : cs, \mu, \rho \rangle \\
\end{align*}
\]

\[
\begin{align*}
\text{Fetch-Protect-Expr} & \quad c = (x := \text{protect}(e)) \\
& \quad i = (x := \text{protect}(e)) \\
& \quad \langle \text{is}, c : cs, \mu, \rho \rangle \xrightarrow{\text{fetch}} \langle \text{is} + [+ [i]], cs, \mu, \rho \rangle \\
\end{align*}
\]

\[
\begin{align*}
\text{Exec-Protect}_1 & \quad i = (x := \text{protect}(e)) \\
& \quad v = [e]^\rho \\
& \quad i' = (x := \text{protect}(v)) \\
& \quad \langle \text{is}_1, i, \text{is}_2, cs \rangle \xrightarrow{\text{\(\mu, \rho, e\)}} \langle \text{is}_1 + [+ [i'] + \text{is}_2, cs] \rangle \\
\end{align*}
\]

\[
\begin{align*}
\text{Exec-Protect}_2 & \quad i = (x := \text{protect}(v)) \\
& \quad \text{guard}(\_\_\_ \neq \text{is}_1) \quad i' = (x := v) \\
& \quad \langle \text{is}_1, i, \text{is}_2, cs \rangle \xrightarrow{\text{\(\mu, \rho, e\)}} \langle \text{is}_1 + [+ [i'] + \text{is}_2, cs] \rangle \\
\end{align*}
\]

(a) Semantics of \text{protect} as a hardware primitive (selected rules).

\[
\begin{align*}
\text{Fetch-Protect-SLH} & \quad c = x := \text{protect}(a[e]) \\
& \quad e_1 = e < \text{length}(a) \\
& \quad e_2 = \text{base}(a) + e \\
& \quad c_1 = m := e_1 \\
& \quad c_2 = m := m ? 1 : 0 \\
& \quad c_3 = x := a[e] \\
& \quad c' = c_1; \text{if} \; m \; \text{then} \; c_2; \text{c\_3} \; \text{else} \; \text{fail} \\
& \quad \langle \text{is}, c : cs, \mu, \rho \rangle \xrightarrow{\text{fetch}} \langle \text{is}, c' : cs, \mu, \rho \rangle \\
\end{align*}
\]

(b) Software implementation of \text{protect}(a[e]).

Fig. 9. Semantics of \text{protect}.

### 3.4 Protect

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

**Protect in Hardware.** Instruction \(x := \text{protect}(r)\) 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 \(\text{Fetch-Protect-Expr}\) fetches protect commands involving simple expressions \((x := \text{protect}(e))\) and inserts the corresponding protect instruction in the reorder buffer. Rule \(\text{Fetch-Protect-Array}\) piggy-backs on the previous rule by splitting a protect of an array read \((x := \text{protect}(a[e]))\) into a separate assignment of the array value \((x := a[e])\) and protect of the variable \((x := \text{protect}(x))\). Rules \(\text{Exec-Protect}_1\) and \(\text{Exec-Protect}_2\) extend the auxiliary relation \(\Rightarrow \rightarrow \rightarrow \rightarrow \) and \(\Rightarrow \rightarrow \rightarrow \rightarrow \rightarrow \) the reorder buffer. Rule \(\text{Exec-Protect}_1\) transforms 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., \(x := \text{protect}(v)\), to prevent forwarding the value to the later instructions via the transient variable map. When no guards are pending in the reorder buffer \((\text{guard}(\_\_\_ \neq \text{is}_1))\), rule \(\text{Exec-Protect}_2\) 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 \text{protect} statement. As a result, directive \text{exec} 5 produces value \(z := \text{protect}(42), \ldots\)
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. These data-dependencies validate control-flow decisions at runtime by stalling speculative loads until the processor resolves their bounds check conditions.\(^5\) Formally, we replace rule [FETCH-PROTECT-ARRAY] with rule [FETCH-PROTECT-SLH] in Figure 9b. The rule computes the bounds check condition $e_1 = e < \text{length}(a)$, the target address $c_2 = (\text{base} \ a) + e$, 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 ($c_2 = m := m \ 1 : 0$), and (iii) mask off the target address with the bitmask ($c_3 = x := s(e_2 \otimes m)$).\(^6\) 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 $m = 1$ leaves the target address unchanged ($[[e']]^p = [[e' \otimes 1]^0]$) and the processor reads the correct address normally. Otherwise, the bitmask $m = 0$ zeros out the target address and the processor loads speculatively only from the constant address $0 = [[e' \otimes 0]^0]$. (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 $m := m \ 1 : 0$ 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 **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., $x := \text{protect}(a[i_1])$ and $y := \text{protect}(a[i_2])$.

### 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. Importantly, the type system does not rely on user annotations to identify secret sources and public sinks, but conservatively rejects programs that exhibit any source-to-sink data flows.\(^7\) This type system does not address the problem of enforcing a sequential constant-time discipline, which is the goal of existing type systems [Watt et al. 2019] and secure interfaces [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

\(^5\)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.

\(^6\)Alternatively, it would be also possible to mask the loaded value, i.e., $c_3 = x := (\text{base} e_2) \otimes m$. 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.

\(^7\)We remark that secrecy annotations would not meaningfully improve the precision of the type system unless secret data is partitioned separately from public data.
repair algorithm we patch the program by inserting a minimum number of \texttt{protect} so that it cannot leak speculatively anymore (Section 4.3).

4.1 Type System

Our type system assigns a \textit{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 \textit{stable} (S) indicating that it cannot contain transient values during execution, or as \textit{transient} (T) indicating that it can. These types form a 2-point lattice [Landauer 1993], which allows stable expressions to be typed as transient, but not vice versa, i.e., we define a can-flow-to relation \( \subseteq \) such that \( S \subseteq T \), but \( T \nsubseteq S \).

\textbf{Typing Expressions.} Given a typing environment for variables \( \Gamma \in \text{Var} \rightarrow \{S, T\} \), the typing judgment \( \Gamma \vdash r : \tau \) assigns a transient-flow type \( \tau \) to \( r \). Figure 10 presents selected rules (see Appendix A.1 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 [\texttt{Value}] and variables are assigned their respective type from the environment in rule [\texttt{Var}]. Rule [\texttt{Bop}] propagates the
type of the operands to the result of binary operators $\oplus \in \{+, \cdot, \land\}$. Finally, rule [ARRAY-READ] assigns the \textit{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 \textit{stable} ($\mathbf{S}$) to prevent programs from leaking through the cache.

\textbf{Typing Commands.} Given a set of protected variables Prot, we define a typing judgment $\Gamma, \text{Prot} \vdash c$ for commands. Intuitively, a command $c$ is well-typed under environment $\Gamma$ and set Prot, if $c$ does not leak, under the assumption that the expressions assigned to all variables in Prot are protected using the $\text{protect}$ primitive. Figure 10b shows our typing rules. Rule [ASGN-PROT] disallows assignments from \textit{transient} to \textit{stable} variables (as $\mathbf{T} \not\subseteq \mathbf{S}$). Rule [PROTECT] relaxes this policy as long as the right-hand side is explicitly protected.\footnote{Readers familiar with information-flow control may see an analogy between \texttt{protect} and the \texttt{declassify} primitive of some IFC languages [Myers et al. 2004].} Intuitively, the result of \texttt{protect} is \textit{stable} and it can thus flow securely to variables of any type. Rule [ASGN-PROT] is similar, but instead of requiring an explicit \texttt{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 \textit{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 \textit{sinks} and therefore require them to be typed \textit{stable} ($\Gamma \vdash e_2 : \mathbf{S}$).

\textbf{Implicit Flows.} To prevent programs from leaking data \textit{implicitly} through their control flow, rule [IF-THEN-ELSE] requires the branch condition to be \textit{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 \textit{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 \textit{follow} a mispredicted branch. In particular, the rollback caused by a misprediction may cause to \textit{repeat} load and store instructions after the mispredicted branch, thus revealing whether the attacker guessed the value of the branch condition.

\textbf{Example.} Consider the program $\{\text{if } tr \text{ then } x := 0 \text{ else skip}; y := a[0].\}$ 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 \texttt{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 \texttt{true}, the observation trace of the program contains a single read observation. However, if $tr$ is \texttt{false}, the processor detects a misprediction, restarts the execution from the other branch (\texttt{skip}) and executes the array read \texttt{again}, producing a rollback and \textit{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 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 constraints 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 Γ, Prot ⊢ r ⇒ 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., αx for x) and unknown types for expressions (e.g., r). In rule [Var], constraint x ⊑ αx 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 αx can only be instantiated with type T. Rule [Bop] generates constraints e1 ⊑ e1 ⊕ e2 and e2 ⊑ e1 ⊕ e2 to reflect the fact that the unknown type of e1 ⊕ e2 should be at least as transient as the (unknown) type of e1 and e2. Notice that these constraints correspond exactly to the premises τ1 ⊑ τ and τ2 ⊑ τ of the same rule. Similarly, rule [ARRAY-READ] generates constraint e ⊑ S for the unknown type of the array index, thus forcing it to be typed S. In addition to these, the rule generates also the constraint T ⊑ a[e], which forces the type of a[e] to be T. Rule [ASGN] generates the constraint r ⊑ x disallowing transient to stable assignments. In contrast, rule [PROTECT] does not generate the constraint r ⊑ 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 Γ, Prot ⊢ s ⇒ k. For this, we define a dummy environment Γ∗ and protected set Prot∗, such that Γ∗, Prot∗ ⊢ c ⇒ k holds for any command c (i.e., we let Γ∗ = λx: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., σ ∈ ATOMS ⇒ (T, S), and extend solutions to map T and S to themselves. For a set of constraints k and a solution function σ, we write σ ⊨ 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 σ ⊨ 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 a1 . . . an a path in k, if a1 ⊑ a1+1 ∈ k for i ∈ {1, . . . , n − 1} and say that a1 is the path’s entry and an 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

---

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 be typed as stable.
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 \( \sigma(k)(a) = T \), 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 \( a_1, \ldots, a_n \), 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., \( \#A \leq \#A' \).

**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 \( \alpha_x \), we define \( \Gamma(k, A)(x) = T \), if there is a path with entry T and exit \( \alpha_x \) in k that is not cut by A, and let \( \Gamma(k, A)(x) = S \) otherwise.

**PROPOSITION 1 (TYPE INFERENCE).** If \( \Gamma^* \), Prot* + c \( \Rightarrow k \) and A is a set of variables that cut k, then \( \Gamma(k, A), A \vdash s \).

**Remark.** To infer a repair using 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 \( \alpha \)-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-optimal 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. Therefore, for each variable \( x \in \text{Prot} \) there is a single assignment \( x := r \), and our repair algorithm simply replaces it with \( x := \text{protect}(r) \).

## 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 execution [Cauligi et al. 2020]. We provide full definitions and proofs in Appendix B.

**Consistency.** We write $C \triangleright_{O} C'$ 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 $(\mu, \rho) \triangleright_{O} (\mu', \rho')$ for the sequential execution of program $c$ with initial memory $\mu$ and variable map $\rho$ resulting in final memory $\mu'$ and variable map $\rho'$. 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 $\mu$, variable maps $\rho$, and directives $D$, if $(\mu, \rho) \triangleright_{O} (\mu', \rho')$ and $(L, [c], \mu, \rho) \triangleright_{O} (L', [c], \mu', \rho'')$, then $\mu' = \mu''$, $\rho' = \rho''$, and $O \equiv O' \downarrow$.

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., $O \equiv O' \downarrow$, 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 low-equivalence, i.e., equivalence of configurations, with respect to the value of public ($L$) variables and arrays.

**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 $C_1 = ([], [c], \mu_i, \rho_i)$ for $i \in \{1, 2\}$, if $C_1 \approx_{L} C_2$, $C_1 \triangleright_{O_1} C_1'$, and $C_2 \triangleright_{O_2} C_2'$, 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 $\Gamma^*, \text{Prot}^* \vdash c \Rightarrow k$. Let $A$ be a set of variables that cut $k$. Then, it follows that $SCT_{L}(\text{repair}(c, A))$.

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. 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. 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, 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.

7 EVALUATION

We evaluate Blade by answering two questions: (Q1) How many protects 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.

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.

Spectre v1 and Spectre v1.1 attacks are both classified as Spectre-PHT attacks [Canella et al. 2019].
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Defense</th>
<th>Without v1.1 protections</th>
<th>With v1.1 protections</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time</td>
<td>Overhead</td>
</tr>
<tr>
<td>Salsa20 (CT-Wasm), 64 bytes</td>
<td>Ref</td>
<td>6.9 us</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Baseline-F</td>
<td>7.2 us</td>
<td>3.4%</td>
</tr>
<tr>
<td></td>
<td>BLADE-F</td>
<td>7.0 us</td>
<td>0.8%</td>
</tr>
<tr>
<td></td>
<td>Baseline-S</td>
<td>7.0 us</td>
<td>1.4%</td>
</tr>
<tr>
<td></td>
<td>BLADE-S</td>
<td>7.0 us</td>
<td>0.7%</td>
</tr>
<tr>
<td>SHA-256 (CT-Wasm), 64 bytes</td>
<td>Ref</td>
<td>20.7 us</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Baseline-F</td>
<td>25.8 us</td>
<td>24.7%</td>
</tr>
<tr>
<td></td>
<td>BLADE-F</td>
<td>20.6 us</td>
<td>-0.3%</td>
</tr>
<tr>
<td></td>
<td>Baseline-S</td>
<td>21.5 us</td>
<td>4.0%</td>
</tr>
<tr>
<td></td>
<td>BLADE-S</td>
<td>20.6 us</td>
<td>-0.3%</td>
</tr>
<tr>
<td>SHA-256 (CT-Wasm), 8192 bytes</td>
<td>Ref</td>
<td>137.5 us</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Baseline-F</td>
<td>137.6 us</td>
<td>0.1%</td>
</tr>
<tr>
<td></td>
<td>Baseline-S</td>
<td>191.7 us</td>
<td>39.4%</td>
</tr>
<tr>
<td></td>
<td>BLADE-S</td>
<td>139.3 us</td>
<td>1.3%</td>
</tr>
<tr>
<td>ChaCha20 (HACL(^{\ast})), 8192 bytes</td>
<td>Ref</td>
<td>45.7 us</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Baseline-F</td>
<td>82.6 us</td>
<td>81.0%</td>
</tr>
<tr>
<td></td>
<td>BLADE-F</td>
<td>46.7 us</td>
<td>2.2%</td>
</tr>
<tr>
<td></td>
<td>Baseline-S</td>
<td>55.0 us</td>
<td>20.5%</td>
</tr>
<tr>
<td></td>
<td>BLADE-S</td>
<td>46.3 us</td>
<td>1.3%</td>
</tr>
<tr>
<td>Poly1305 (HACL(^{\ast})), 1024 bytes</td>
<td>Ref</td>
<td>7.6 us</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Baseline-F</td>
<td>8.4 us</td>
<td>11.1%</td>
</tr>
<tr>
<td></td>
<td>BLADE-F</td>
<td>7.6 us</td>
<td>0.1%</td>
</tr>
<tr>
<td></td>
<td>Baseline-S</td>
<td>7.8 us</td>
<td>2.8%</td>
</tr>
<tr>
<td></td>
<td>BLADE-S</td>
<td>7.6 us</td>
<td>0.2%</td>
</tr>
<tr>
<td>Poly1305 (HACL(^{\ast})), 8192 bytes</td>
<td>Ref</td>
<td>16.3 us</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Baseline-F</td>
<td>22.0 us</td>
<td>34.7%</td>
</tr>
<tr>
<td></td>
<td>BLADE-F</td>
<td>16.3 us</td>
<td>0.1%</td>
</tr>
<tr>
<td></td>
<td>Baseline-S</td>
<td>17.4 us</td>
<td>6.6%</td>
</tr>
<tr>
<td></td>
<td>BLADE-S</td>
<td>16.3 us</td>
<td>0.2%</td>
</tr>
<tr>
<td>ECDH Curve25519 (HACL(^{\ast}))</td>
<td>Ref</td>
<td>405.7 us</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Baseline-F</td>
<td>964.5 us</td>
<td>137.7%</td>
</tr>
<tr>
<td></td>
<td>BLADE-F</td>
<td>512.1 us</td>
<td>26.2%</td>
</tr>
<tr>
<td></td>
<td>Baseline-S</td>
<td>546.3 us</td>
<td>34.7%</td>
</tr>
<tr>
<td></td>
<td>BLADE-S</td>
<td>433.9 us</td>
<td>6.9%</td>
</tr>
</tbody>
</table>

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)

- 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\(^{\ast}\) [Zinzindohoué et al. 2017] library. We compile the automatically generated C code to Wasm using Clang’s Wasm backend. (We do not use HACL\(^{\ast}\)’s Wasm backend since it relies on a JavaScript embedding environment and is not well suited for Lucet.) Specifically, from HACL\(^{\ast}\) 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 Core i7-6700K (Skylake) with 64GB of RAM. The machine runs Arch Linux with kernel 5.5.8, and we use the Lucet runtime version 0.7.0-dev (Cranelift version 0.62.0 with our modifications) compiled with rustc version 1.40.0. 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 protects every Wasm memory load instruction.

We consider two baseline variants: The baseline solution with Spectre v1.1 mitigation protects every Wasm load instruction, while the baseline solution with only Spectre v1 mitigation protects 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 protects and have negligible performance overhead on most of our benchmarks—the geometric mean overheads imposed by BLADE-F and BLADE-S are 3.8% and 1.4%, 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 62.1% and 14.7%, respectively.

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 11.7%, whereas Baseline-F’s is 79.1%. The geometric mean overhead of both BLADE-S and Baseline-S, on the other hand, is roughly 19%. 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 two observations from our measurements. First, and somewhat surprisingly, BLADE does not insert any protects 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 protects.

Second, 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, that advantage is lost when considering both v1 and v1.1 mitigation. In this case, the
sharp increase in the number of protects required for this solution end up making the fenced
version more performant overall.
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,
oto 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 signifi-
cant 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 [Balliu et al. 2019; Cauligi et al. 2020; Cheang et al. 2019; Disselkoen et al. 2019;
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 for low-level, assembly-like languages, while our JIT semantics allows us to
reason about speculative execution of higher level imperative programs.12
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
12Disselkoen 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.
chains. 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 this function table to be typed as stable.

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. 2019b]. Doing this for speculative constant-time, and understanding which optimizations break the SCT notion, is an interesting direction for future work.

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.

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


A FULL CALCULUS

Arrays: $a \ ::= \ \{ \ \text{base } n, \ \text{length } n, \ \text{label } \ell \} \$

Values: $v \ ::= \ n \mid b \mid a$

Expressions: $e \ ::= \ v \mid x \mid e_1 + e_2 \mid e_1 \leq e_2 \mid e_1 ? : e_2 \mid e \otimes e$

Right-hand Sides: $r \ ::= \ e \mid (\ast \ell \ e) \mid a[e]$

Commands: $c \ ::= \ \text{skip} \mid x := r \mid (\ast \ell \ e) = e \mid a[e_1] := e_2 \mid x := \text{protect}(r)$

Instructions: $i \ ::= \ \text{nop} \mid x := e \mid x := \text{load}_\ell(e) \mid \text{store}_\ell(e_1, e_2) \mid x := \text{protect}(e)$

Predictions: $b \in \{ \text{true}, \text{false} \}$

Guard Identifiers: $p \in \mathbb{N}$

Directives: $d \ ::= \ \text{fetch} \mid \text{fetch } b \mid \text{exec } n \mid \text{retire}$

Schedules: $D \ ::= \ [] \mid d : D$

Observations: $o \ ::= \ e \mid \text{fail} \mid \text{read}(n, ps) \mid \text{write}(n, ps) \mid \text{rollback}(p)$

Observation Traces: $O \ ::= \ e \mid o \cdot O$

Reorder Buffers: $i s \ ::= \ i : i s \mid []$

Command Stacks: $cs \ ::= \ c : cs \mid []$

Memory Stores: $\mu \in \mathbb{N} \rightarrow \text{Value}$

Variable Maps: $\rho \in \text{Var} \rightarrow \text{Value}$

Configurations: $C \ ::= \ \langle is, cs, \mu, \rho \rangle$

Fig. 12. Decorated syntax. Pointer operators (e.g., $\ast \ell \ e$) and memory instructions (e.g., $\text{load}_\ell(e)$) are annotated with a security label $\ell$, which represents the sensitivity of the data stored at the corresponding memory accesses.
Fig. 13. Full semantics.
Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade

Fig. 13. Full semantics (continued).
\[ \phi(\rho, [\varepsilon]) = \rho \]
\[ \phi(\rho, (x := v) : is) = \phi(\rho[x \mapsto v], is) \]
\[ \phi(\rho, (x := e) : is) = \phi(\rho[x \mapsto \bot], is) \]
\[ \phi(\rho, (x := \text{load}(e)) : is) = \phi(\rho[x \mapsto \bot], is) \]
\[ \phi(\rho, (x := \text{protect}(e)) : is) = \phi(\rho[x \mapsto \bot], is) \]
\[ \phi(\rho, i : is) = \phi(\rho, is) \]

(d) Transient Variable Map.

\[
\begin{align*}
\{ [] \} &= [] \\
\{ \text{guard}(e^b, cs, p) : is \} &= p : \{ is \} \\
\{ i : is \} &= \{ is \}
\end{align*}
\]

(e) Pending Guard Identifiers.

\[
\begin{align*}
[\varepsilon] &\rightarrow \nu \\
[x] &\rightarrow \rho(x) \\
[e_1 + e_2] &\rightarrow [e_1] + [e_2] \\
[e_1 \leq e_2] &\rightarrow [e_1] \leq [e_2] \\
[e_1 \otimes e_2] &\rightarrow [e_1] \otimes [e_2] \\
[e_1 ? e_2 : e_3] &\rightarrow \begin{cases}
[e_2] & \text{if } [e_1] = \text{true} \\
[e_3] & \text{if } [e_1] = \text{false} \\
\bot & \text{if } [e_1] = \bot
\end{cases}
\end{align*}
\]

(f) Evaluation Function.

\[
\begin{align*}
\text{base}(e) &\rightarrow \begin{cases}
n & \text{if } [e] = \{ n, \bot, \bot \} \\
\bot & \text{if } [e] = \bot
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{length}(e) &\rightarrow \begin{cases}
n & \text{if } [e] = \{ \bot, n, \bot \} \\
\bot & \text{if } [e] = \bot
\end{cases}
\end{align*}
\]

(g) Helper functions.

Fig. 13. Full semantics (continued).
Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade

Fig. 13. Full semantics (continued).

A.1 Full Type System

**Lemma 1 (Transient-Flow Type Preservation).** If $\Gamma \vdash C$ and $C \xrightarrow{d} C'$, then $\Gamma \vdash C'$.

**Proof.** By case analysis on the reduction step and typing judgment. $\square$
Transient-flow Lattice: $\mathcal{T} = \{ S, T \}$ where $S \subseteq \tau, T \subseteq T, T \not\subseteq S$

Transient-flow types: $\tau \in \mathcal{T}$

Typing Context: $\Gamma \in \text{Var} \rightarrow \mathcal{T}$

(a) Transient-flow lattice and syntax.

(b) Typing Rules for Expressions and Arrays.
Fig. 14. Transient-flow type system and type constraints generation, and constraints satisfiability.
Nop \quad Γ \vdash \text{nop} \quad \text{FAIL} \quad Γ \vdash \text{fail} \quad \text{ASGN} \quad \frac{Γ \vdash e : τ \quad τ \subseteq Γ(x)}{Γ \vdash x := e} \quad \text{PROTECT} \quad Γ \vdash e : τ \quad Γ \vdash x := \text{protect}(e) \quad \text{LOAD} \quad Γ \vdash e : S \quad Γ \vdash x := \text{load}_ℓ(e)

Store \quad Γ \vdash e_1 : S \quad Γ \vdash e_2 : τ \quad Γ \vdash \text{store}_ℓ(e_1, e_2)

Store-Spectre-1.1 \quad Γ \vdash e_1 : S \quad Γ \vdash e_2 : S \quad Γ \vdash \text{store}_ℓ(e_1, e_2)

Guard \quad Γ \vdash e : S \quad Γ \vdash cs \quad Γ \vdash \text{guard}(e^b, cs, p)

(a) Instructions $Γ \vdash i$.

Cmd-Stack-Empty \quad Γ \vdash \mathbf{[]} \quad \text{CMD-STACK-CONS} \quad Γ \vdash c \quad Γ \vdash cs \quad Γ \vdash c : cs \quad \text{RB-EMPTY} \quad Γ \vdash \mathbf{[]} \quad \text{RB-CONS} \quad Γ \vdash i \quad Γ \vdash is

(b) Command stack $Γ \vdash cs$ and reorder buffer $Γ \vdash is$.

Conf \quad Γ \vdash is \quad Γ \vdash cs \quad Γ \vdash \langle is, cs, μ, ρ \rangle

(c) Configurations $Γ \vdash C$.

Fig. 15. Typing rules for the speculative processor.
Security Lattice: \( \mathcal{L} = \{ L, H \} \) where \( L \subseteq \ell, H \subseteq H, H \not\subseteq L \)

Security Labels: \( \ell \in \mathcal{L} \)

Typing Context: \( \Gamma \in \text{Var} \rightarrow \mathcal{L} \)

(a) Two-point security lattice and syntax.

<table>
<thead>
<tr>
<th>VAR</th>
<th>ARRAY</th>
<th>VAL</th>
<th>PROJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma \vdash \text{ct} x : \Gamma(x) )</td>
<td>( \text{val} a = { - \ell } )</td>
<td>( \text{val} v \neq a )</td>
<td>( f \in { \text{length}(\cdot), \text{base}(\cdot) } )</td>
</tr>
<tr>
<td>BOP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash \text{ct} e_1 : \ell )</td>
<td>( \Gamma \vdash \text{ct} e_2 : \ell )</td>
<td>( \Gamma \vdash \text{ct} (e_1 \oplus e_2) : \ell )</td>
<td></td>
</tr>
<tr>
<td>SELECT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash \text{ct} e_1 : L )</td>
<td>( \Gamma \vdash \text{ct} e_2 : \ell )</td>
<td>( \Gamma \vdash \text{ct} e_3 : \ell )</td>
<td></td>
</tr>
<tr>
<td>PTR-READ</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash \text{ct} e : L )</td>
<td>( \Gamma \vdash \text{ct} e : \ell )</td>
<td>( \Gamma \vdash \text{ct} e : \ell )</td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash \text{ct} e_1 : \ell_1 )</td>
<td>( \Gamma \vdash \text{ct} e_2 : L )</td>
<td>( \Gamma \vdash \text{ct} e : \ell )</td>
<td>( \ell \subseteq \ell_1 )</td>
</tr>
</tbody>
</table>

(b) Expressions: \( \Gamma \vdash \text{ct} e : \ell \).

<table>
<thead>
<tr>
<th>SKIP</th>
<th>FAIL</th>
<th>ASGN</th>
<th>PROTECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma \vdash \text{ct} \text{skip} )</td>
<td>( \Gamma \vdash \text{ct} \text{fail} )</td>
<td>( \Gamma \vdash \text{ct} \text{r : } \ell )</td>
<td>( \ell \subseteq \Gamma(x) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \Gamma \vdash \text{ct} x := \text{r} )</td>
<td>( \Gamma \vdash \text{ct} x := \text{protect(r)} )</td>
</tr>
<tr>
<td>ARRAY-WRITE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash \text{ct} e_1 : \ell_1 )</td>
<td>( \Gamma \vdash \text{ct} e_2 : L )</td>
<td>( \Gamma \vdash \text{ct} e : \ell )</td>
<td>( \ell \subseteq \ell_1 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \Gamma \vdash \text{ct} e_1[e_2] := e )</td>
<td></td>
</tr>
<tr>
<td>PTR-WRITE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash \text{ct} e_1 : L )</td>
<td>( \Gamma \vdash \text{ct} e_2 : \ell_2 )</td>
<td>( \ell_2 \subseteq \ell )</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \Gamma \vdash \text{ct} (*\ell) e_1 := e_2 )</td>
<td></td>
</tr>
<tr>
<td>WHILE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash \text{ct} e : L )</td>
<td>( \Gamma \vdash \text{ct} c )</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \Gamma \vdash \text{ct} \text{while } e \ c )</td>
<td></td>
</tr>
<tr>
<td>IF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash \text{ct} e : L )</td>
<td>( \Gamma \vdash \text{ct} c_1 )</td>
<td>( \Gamma \vdash \text{ct} c_2 )</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \Gamma \vdash \text{ct} \text{if } e \ \text{then } c_1 \ \text{else } c_2 )</td>
<td></td>
</tr>
<tr>
<td>SEQ</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \Gamma \vdash \text{ct} c_1 )</td>
<td>( \Gamma \vdash \text{ct} c_2 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \Gamma \vdash \text{ct} c_1 ; c_2 )</td>
<td></td>
</tr>
</tbody>
</table>

(c) Commands \( \Gamma \vdash \text{ct} \ c \).

### A.2 Constant-Time Type System

The patches computed by BLADE enforce speculative constant time (Definition 1) only for programs that are already sequential constant-time. Enforcing sequential constant time is not a goal of BLADE because this problem has already been addressed in previous work [Protzenko et al. 2019; Watt et al. 2019]. Therefore, the soundness guarantees of BLADE (Theorem 2) rely on a separate, but standard, type-system to enforce sequential constant time. To this end, we simply adopt the constant-time programming discipline from [Protzenko et al. 2019; Watt et al. 2019] by disallowing secret-dependent branches and memory accesses. Given a security policy \( L \) that specifies the set
of public variables and arrays (memory addresses containing public data), a program satisfies sequential constant time if it is well-typed according to the type-system from Figure 16. Figure 16a defines the classic two-point lattice consisting of public (L) and secret (H) security levels, which disallows secret-to-public flows of information (H \nsubseteq L). The type-system relies on a typing environment \( \Gamma \) to map variables and arrays to their security level, which we derive from the given security policy, as follows:

\[
CT_L(c) \triangleq \Gamma \vdash_{\text{ct}} c \quad \text{where} \quad \Gamma(x) = \begin{cases} L, & \text{if } x \in L \\ H, & \text{otherwise} \end{cases}
\]

The typing rules from Figure 16b and 16c are fairly standard. Figure 16b defines the typing judgment for expressions, i.e., \( \Gamma \vdash_{\text{ct}} e : \ell \), which indicates that expression \( e \) has sensitivity at most \( \ell \) under typing context \( \Gamma \). Rule [Var] types variables and arrays\(^{14}\) according to the typing context \( \Gamma \) and ground values can assume any label in rule [Value]. These labels can be upgraded via rule [Sub] and are otherwise propagated in rules [Fun,Bop,Select,Array-Read,Ptr-Read].\(^{15}\)

\(^{14}\)For simplicity, we assume that arrays are always referenced through variables, which determine their security level.

\(^{15}\)Technically, our type-system does not prevent an attacker from forging a public pointer to secret memory via rules [Val] and [Ptr-Read]. In the following, we assume that pointers are not directly casted from integers, but they only appear when evaluating array accesses and thus have the same security level of the array.
Additionally, rule [ARRAY-READ] disallows memory reads that may depend on secret data by typing the index $e_2$ as public ($L$).

Figure 16c defines the typing judgment for commands, i.e., $\Gamma \vdash_{ct} c$, which indicates that program $c$ is constant time under sequential execution. Rules [ASGN,PROTECT] disallow (protected) secret-to-public assignments ($H \not\sqsubseteq L$). Rule [ARRAY-WRITE] is similar but additionally disallows memory-writes that depend on secret data by typing the index $e_2$ as public ($L$), like rule [ARRAY-READ].

The combination of rules [ARRAY-READ] and [ARRAY-WRITE] makes programs that exhibit secret-dependent memory accesses ill-typed. In a similar way, rules [IF] and [WHILE] forbid branches that may depend on secret data by typing the conditional $e$ as public ($L$).

**Lemma 2 (Constant-Time Type Preservation).** If $\Gamma \vdash_{ct} C$ and $C \xrightarrow{d} C'$, then $\Gamma \vdash_{ct} C'$.

**Proof.** By case analysis on the reduction step and typing judgment. □
Fig. 17. Sequential big-step semantics with observations.

B PROOFS

B.1 Consistency

Notation. In the following, we write \textbf{rollback} \not\in O to denote that no rollback observation occurs in the observation trace O, i.e., \forall p. \textbf{rollback}(p) \not\in O. The notation \( C \xrightarrow{D} O \)\( C' \) denotes a multi-step speculative reduction, i.e., a sequence of zero or more small-step reductions from configuration \( C \) to \( C' \) that follow the directives in schedule \( D \) and generate observation trace \( O \). We consider equivalence of observation traces \textit{up to silent observations} \( \epsilon \), i.e., \( \forall O. \epsilon \cdot O = O \cdot \epsilon = O \).

We begin by defining the notion of \textit{valid directive}, i.e., a directive that can be followed by a given configuration without getting stuck.

Definition 2 (Valid Directive). A directive \( d \) is valid for a configuration \( C \) iff there exists a configuration \( C' \) and an observation \( o \) such that \( C \xrightarrow{d} o C' \).
Similarly, we define the notion of valid schedule, i.e., a list of valid directives that completely evaluates a given program without getting the processor stuck and re-executing any instruction.

**Definition 3 (Valid Schedule).** A schedule $D$ is valid for a configuration $C$ iff there exists a final configuration $C'$ and a sequence of observations $O$ such that $C \mathrel{\mathcal{O}} D C'$ and every fetched instruction is executed at most once.

**Lemma 3 (Consistency of Valid Schedules).** For all schedules $D_1$ and $D_2$ valid for configuration $C$, if $C \mathrel{\mathcal{O}} D_1 C_1, C \mathrel{\mathcal{O}} D_2 C_2$, rollback $\neq O_1$ and rollback $\neq O_2$, then $C_1 = C_2$ and $O_1 \equiv O_2$.

**Proof (Sketch).** By design, our processor fetch and retire instructions in-order and can only execute them out-of-order. In this case, we know additionally that the schedules $D_1$ and $D_2$ are valid (Def. 3) and no rollbacks occur during the executions, therefore we deduce that (1) each fetched instruction is executed exactly once before it is eventually retired, and (2) the fetch and retire directives in the schedules fetch and retire corresponding instructions in the same order. Therefore, to prove that the executions reach the same final configurations ($C_1 = C_2$), it suffices to show that the executions evaluate the operands of corresponding instructions to the same value. Intuitively, the particular order with which instructions are executed does not affect the value of their operands, which is determined only by the data they depend on. In particular, if the schedule does not respect these data dependencies the processor simply gets stuck. However the two executions cannot get stuck because schedules $D_1$ and $D_2$ are valid by assumption, and thus each exec directive in the schedules is also valid (Def. 2). Therefore, in the executions $C \mathrel{\mathcal{O}} D_1 C_1$ and $C \mathrel{\mathcal{O}} D_2 C_2$ above, the dependencies of each instruction executed must have already been resolved (out-of-order) or committed (in-order) by previous exec and retire directives in schedules $D_1$ and $D_2$. Thus, the operands of each instruction must evaluate to the same value and we conclude that $C_1 = C_2$.

To prove that the observation traces are equal up to permutation, i.e., $O_1 \equiv O_2$, we match individual observations generated by corresponding directives and instructions. The observations generated by fetch and retire directives are easier to match because these directives proceed in-order. To relate their observations, we compute the pairs of matching fetch and retire directives from the schedules and use the indexes of individual directives to relate the corresponding observations. Formally, let $D_1 = \{(d_1)_1, \ldots, (d_1)_n\}$ and $D_2 = \{(d_2)_1, \ldots, (d_2)_n\}$ be the list of directives of the first and second schedule, respectively. Let $((d_1)_i, (d_2)_j)_k$ be the pair matching the $k$-th fetch directives, i.e., where directives $(d_1)_i = \text{fetch} = (d_2)_j$ (or $(d_1)_i = \text{fetch} = (d_2)_j = \text{fetch} = b = (d_2)_j$) are the $k$-th fetch directives in schedules $D_1$ and $D_2$. Then, the indexes $i \leftrightarrow j$ in each pair indicate that the $i$-th and $j$-th observations in the observation traces $O_1$ and $O_2$ are related. In particular, for each fetch matching pair $((d_1)_i, (d_2)_j)_k$ in $D_1$ and $D_2$, we relate the individual observations $(o_1)_i \leftrightarrow (o_2)_j$ (where $o_1 = e = o_2$) of the observation traces $O_1 = (o_1)_1 \ldots (o_1)_n$ and $O_2 = (o_2)_1 \ldots (o_2)_n$. The procedure for matching observations generated by retire directives is similar. Since the schedules can execute instruction out-of-order, we cannot apply exactly the same technique to exec directives. Instead, we annotate the instructions in the reorder buffer with a unique index when they are fetched and relate the observations generated by the instructions with the same index. Intuitively, these indexes identify uniquely matching observations because each fetched instruction is executed exactly once (as explained above). Formally, for each instruction $(i_1)_j$ whose execution generates the observation $(o_1)_j$ in the first execution, we have a corresponding instruction $(i_2)_j$ with observation $(o_2)_j$ in the second execution, where in general $j \neq j'$ because instructions can be executed out-of-order. Since these instructions are annotated with the same index $j$, they are identical, i.e., $i_1 = i_2$, their

\[16\text{As explained in Section 3, the transient variable map and the semantics rules of the processor inhibit (1) instructions whose dependencies have not yet been resolved, which manifests as the value of an operand $e$ being undefined, i.e., $[e]^= \bot$, and (2) instructions that may access stale data, i.e., rule [EXEC-LOAD] requires that no store is pending in the buffer.} \]
operands evaluate to the same value (as explained above), and thus generate the same observations
\( o_1 = o_2 \) and hence \( (o_1)\rho' \leftrightarrow (o_2)\rho' \).

**Definition 4 (Sequential Schedule).** A schedule \( D \) is sequential for a configuration \( C \) iff there exists a final configuration \( C' \) and observation trace \( O \) such that \( C \notr D C' \) and:

1. all instructions are executed in-order, as soon as they are fetched;
2. all instructions are retired as soon as they are executed;
3. no misprediction occurs, i.e., rollack \( \notin O \).

The following lemma shows that we can simulate a sequential execution on our processor through a sequential schedule.

**Lemma 4 (Sequential Consistency).** Given a sequential execution \( \langle \mu, \rho \rangle \notr C \langle \mu', \rho' \rangle \), there exists a valid, sequential schedule \( D \), such that \( \langle I, [\cdot], \mu, \rho \rangle \notr D \langle I, [\cdot], \mu', \rho' \rangle \).

**Proof.** The proof is constructive. We perform induction on the sequential execution judgment \( \langle \mu, \rho \rangle \notr C \langle \mu', \rho' \rangle \) defined in Figure 17 and we construct the corresponding sequential schedule for it. Then, we translate the individual sequential reductions in the derivation tree into corresponding speculative reductions that follow the schedule. In the following, we ignore the security labels that annotate commands.

Case [Skip]. We define the sequential schedule \( D = [\text{fetch}, \text{retire}] \) and construct the corresponding speculative big-step reductions consisting of [Step] applied to [Fetch-Skip], [Step] applied to [Retire-Nop] and [Done]. These reductions form the big-step \( \langle I, [\cdot], \mu, \rho \rangle \notr D \langle I, [\cdot], \mu, \rho \rangle \) that corresponds to the sequential reduction \( \langle \mu, \rho \rangle \notr \text{skip} \langle \mu, \rho \rangle \).

Case [Fail]. Analogous to the previous case, but using rules [Fetch-Fail] and [Retire-Fail]. Notice that in this case the observation trace generated in the sequential execution contains only observation fail, which corresponds to the trace generated by the sequential schedule up to silent events, i.e., \( \epsilon \cdot \text{fail} = \text{fail} \).

Case [Asgn]. We define the sequential schedule \( D = [\text{fetch}, \text{exec 1}, \text{retire}] \) and construct the corresponding speculative big-step reduction using rules [Fetch-Asgn], [Execute-Asgn], and [Retire-Asgn]. These reductions form the big-step \( \langle I, [x := e, \mu, \rho] \rangle \notr D \langle I, [\cdot], \mu, \rho[x \mapsto [e]\rho'] \rangle \), where \( \rho' \) is the transient variable map computed in rule [Execute]. Since the reorder buffer is initially empty, the transient map is identical to the initial variable map, i.e., \( \rho' = \phi(\rho, []) = \rho \), and thus the speculative reduction corresponds to the sequential reduction \( \langle \mu, \rho \rangle \notr \text{exec} \langle \mu, \rho[x \mapsto [e]\rho'] \rangle \).

Case [Ptr-Read]. The translation is similar to the previous case and follows the same sequential schedule \( D = [\text{fetch}, \text{exec 1}, \text{retire}] \). First we apply rule [Fetch-Ptr-Read], which fetches command \( x := e \) from the command stack and inserts the corresponding instruction \( x := \text{load}(e) \) in the empty reorder buffer. Then, we apply rule [Exec-Load], which evaluates \( e \) to the same address \( n \) obtained in [Ptr-Read] (as explained above). Notice that the premise \( \text{store}(_-, _) \notin S_1 \) of rule [Exec-Load] holds trivially, because the reorder buffer is initially empty and thus \( S_1 = [] \). The memory store in the sequential and speculative reductions are equal by assumption, therefore the same value \( v = \mu(n) \) is loaded from memory and assigned to variable \( x \) and the final variable maps remain equal after applying rule [Retire-Asgn] like in the previous case. As a result, we obtain the speculative reduction \( \langle I, [x := \ast e], \mu, \rho \rangle \notr D \langle I, [\cdot], \mu, \rho[x \mapsto \mu(n)] \rangle \) corresponding to the sequential

\[ \notr \]
reduction \( \langle \mu, \rho \rangle \xrightarrow{x \mapsto e} \langle \mu, \rho[x \mapsto \mu(n)] \rangle \). Notice that the observations \( \texttt{read}(n, \mathbb{I}) \) and \( \texttt{read}(n) \) are considered equivalent because the list of guard identifiers is \texttt{empty}, i.e., \( \langle \{s_1\} \rangle = \mathbb{I} \) from \( s_1 = \mathbb{I} \) in rule \([\texttt{Exec-Load}]\).

Case \([\texttt{Ptr-Write}]\). Analogous to case \([\texttt{Ptr-Read}]\).

Case \([\texttt{Array-Read}]\). The sequential reduction reveals that the array is read \textit{in-bounds}, therefore we ensure that the speculative execution follows the correct branch after the bounds check by supplying prediction \texttt{true} for the bounds-check condition. Formally, we define the sequential schedule \( D = [ \texttt{fetch, fetch true, exec 1, retire, fetch, exec 1, retire} ] \) for rule \([\texttt{Array-Read}]\). The first \texttt{fetch} directive is processed by rule \([\texttt{Fetch-Array-Load}]\), which converts command \( x := a[e_1] \) into the corresponding bounds-checked pointer read, i.e., command \( \texttt{if \ e \ then \ x := e' else fail} \), where expression \( e = e_1 < \texttt{length}(a) \) is the bounds-check condition and expression \( e' = \texttt{base}(a) + e_1 \) computes the memory address for the pointer. Then, directive \texttt{fetch true} is processed by rule \([\texttt{Fetch-If-True}]\), which speculatively follows the \texttt{then} branch and produces the guard \( \texttt{guard}(e \texttt{true}, \texttt{fail}, p) \) for some fresh guard identifier \( p \), and pushes command \( x := e' \) on the empty command stack. The guard instruction is then immediately resolved by rule \([\texttt{Execute}]\) applied to \([\texttt{Exec-Branch-Ok}]\) which consumes the first directive \texttt{exec 1}. In rule \([\texttt{Array-Read}]\), the premises \( n = [e_2]^\rho \) and \( n \leq \texttt{length}(a) \) ensure that the bounds-check condition succeeds as predicted in rule \([\texttt{Exec-Branch-Ok}]\), i.e., \( [e]^{\rho} = \texttt{true} \), which then rewrites the guard instruction to \texttt{nop}. Then, \texttt{nop} is retired by the next \texttt{retire} directive and the remaining directives \([\texttt{fetch, exec 1, retire}]\) process the pointer read as in case \([\texttt{Ptr-Read}]\).

Case \([\texttt{Array-Write}]\). Analogous to case \([\texttt{Array-Read}]\).

Case \([\texttt{Array-Read-Fail}]\). The sequential reduction reveals that the array is read \textit{out-of-bounds}, therefore we supply prediction \texttt{false} to the bounds-check guard in the speculative reduction. Formally, we define the sequential schedule \( D = [ \texttt{fetch, fetch false, exec 1, retire, fetch, retire} ] \) for rule \([\texttt{Array-Read-Fail}]\). The first \texttt{fetch} directive is processed by rule \([\texttt{Fetch-Array-Load}]\) as in case \([\texttt{Array-Read}]\). Then, directive \texttt{fetch false} is processed by rule \([\texttt{Fetch-If-False}]\), which generates guard \( \texttt{guard}(e \texttt{false}, \texttt{fail}, p) \) and command stack \( \texttt{fail} \), instead. Similarly to the previous case, the \texttt{exec 1} directive resolves the guard correctly via rule \([\texttt{Exec-Branch-Ok}]\), rewriting it to \texttt{nop}, which is then retired by the next \texttt{retire} directive. Finally, directive \texttt{fetch} pops \texttt{fail} from the stack, which is then retired by directive \texttt{retire}, halting the processor with observation \texttt{fail}.

Case \([\texttt{Array-Write-Fail}]\). Analogous to case \([\texttt{Array-Read-Fail}]\).

Case \([\texttt{If-Then-Else}]\). Analogous to case \([\texttt{Asgn}]\), \([\texttt{Array-Read}]\), and \([\texttt{Array-Read-Fail}]\).

Case \([\texttt{If-Then-Else}]\). Let \( c = \texttt{if \ e \ then \ ctrue \ else \ cfalse} \) be the command executed in rule \([\texttt{If-Then-Else}]\) and let \( b = [e]^{\rho} \) be the value of the conditional under initial variable map \( \rho \). Then, we define \( D_1 = [ \texttt{fetch \ b, exec 1, retire} ] \) as the first part of the corresponding sequential schedule. First, we consume directive \texttt{fetch b} via rule \([\texttt{Fetch-If-b}]\), which follows the correct prediction and inserts instruction \( \texttt{guard}(e^b, [c^{-b}], p) \) in the empty buffer and pushes command \( c_0 \) on the empty command stack. Then, we process directive \texttt{exec 1} through rule \([\texttt{Execute}]\) applied to \([\texttt{Exec-Branch-Ok}]\), which rewrites the guard to instruction \texttt{nop}, which is lastly retired by rule \([\texttt{Retire-Nop}]\). By composing these small-steps, we obtain the multi-step reduction \( \langle \mathbb{I}, [c], \rho, \mu \rangle \xrightarrow{D_1} e \)}
Next, we apply our induction hypothesis to the reduction \(\langle \rho, \mu \rangle \downarrow^{\text{ fail}}_{\text{ O}} \langle \rho', \mu' \rangle\), giving us the second part of the sequential schedule, i.e., \(D_2\), and speculative big-step \(\langle \rho, c, \mu, \rho \rangle \downarrow^{D_2}_{\text{ O}} \langle \rho', \mu' \rangle\). We then conclude the proof by defining the complete sequential schedule \(D = D_1 + D_2\) and composing the multi-step reduction \(\langle \rho, c, \mu, \rho \rangle \downarrow^{D_1 + D_2}_{\text{ O}} \langle \rho', \mu' \rangle\) with the big-step reduction \(\langle \rho, c, \mu, \rho \rangle \downarrow^{D_2}_{\text{ O}} \langle \rho', \mu' \rangle\), thus obtaining the big-step \(\langle \rho, c, \mu, \rho \rangle \downarrow^{D}_{\text{ O}} \langle \rho', \mu' \rangle\).

Case [While-b]. Both the sequential and the speculative semantics progressively unroll while loops into a sequence of conditionals, therefore this case follows similarly to case [If-Then-Else], i.e., using the conditional value \(b\) computed in the sequential reduction as prediction in the fetch block directive of the sequential schedule.

Case [Seq]. From the sequential reduction \(\langle \rho, \mu \rangle \downarrow^{c_1, c_2}_{\text{ O}} \langle \mu'', \rho'' \rangle\), we have two sub-reductions \(\langle \rho, \mu \rangle \downarrow^{c_1}_{\text{ O}} \langle \mu', \rho' \rangle\) and \(\langle \rho, \mu \rangle \downarrow^{c_2}_{\text{ O}} \langle \mu'', \rho'' \rangle\), where \(\text{fail} \notin O_1\). First, we apply our induction hypothesis to the first reduction \(\langle \rho, \mu \rangle \downarrow^{c_1}_{\text{ O}} \langle \mu', \rho' \rangle\), obtaining the first part of the sequential schedule, i.e., \(D_1\), and a big-step reduction \(\langle \rho, \mu \rangle \downarrow^{D_1}_{\text{ O}} \langle \rho', \mu' \rangle\). Intuitively, we can lift this reduction to use initial stack \([c_1, c_2]\) and obtain the multi-step reduction \(\langle \rho, \mu \rangle \downarrow^{D_1 + D_2}_{\text{ O}} \langle \rho', \mu' \rangle\), where the second command \(c_2\) remains unchanged on the resulting stack because no failures or rollbacks occur during the execution, i.e., \(\text{fail} \notin O_1\) and \(\text{rollback} \notin O_1\). Then, we apply our induction hypothesis to the second reduction \(\langle \rho, \mu \rangle \downarrow^{c_2}_{\text{ O}} \langle \mu'', \rho'' \rangle\) and obtain the second part of the sequential schedule, i.e., \(D_2\), and a big-step reduction \(\langle \rho, \mu \rangle \downarrow^{D_2}_{\text{ O}} \langle \rho', \mu' \rangle\). By composing these reductions, we obtain the big-step reduction \(\langle \rho, \mu \rangle \downarrow^{D_1 + D_2}_{\text{ O}} \langle \rho', \mu' \rangle\) and obtain the second part of the sequential schedule, i.e., \(D_2\), and a big-step reduction \(\langle \rho, \mu \rangle \downarrow^{D_2}_{\text{ O}} \langle \rho', \mu' \rangle\). Finally, we define the complete sequential schedule \(D = [\text{fetch}] + D_1 + D_2\) and compose the small-step \([\text{fetch}] \langle \rho, \mu \rangle \downarrow^{c_1, c_2}_{\text{ O}} \langle \mu'', \rho'' \rangle\) obtained via rule [Fetch-Seq] with the big-step reduction \(\langle \rho, \mu \rangle \downarrow^{D_2}_{\text{ O}} \langle \rho', \mu' \rangle\), thus obtaining the corresponding big-step reduction \(\langle \rho, \mu \rangle \downarrow^{D}_{\text{ O}} \langle \rho', \mu' \rangle\).

Case [Seq-Fail]. Analogous to case [Seq]. First, we apply our induction hypothesis to the reduction \(\langle \rho, \mu \rangle \downarrow^{c_1, c_2}_{\text{ O}} \langle \mu', \rho' \rangle\) where \(\text{fail} \in O_1\), and obtain part of the sequential schedule, i.e., \(D_1\), and a big-step reduction \(\langle \rho, \mu \rangle \downarrow^{D_1}_{\text{ O}} \langle \rho', \mu' \rangle\). In contrast to the previous case, when we lift this reduction to use initial stack \([c_1; c_2]\), we obtain the big-step reduction \(\langle \rho, \mu \rangle \downarrow^{D_1 + D_2}_{\text{ O}} \langle \rho', \mu' \rangle\), because a failure occurs during the execution of command \(c_1\), i.e., \(\text{rollback} \in O_1\), which aborts the execution and empties the command stack. Finally, we define the complete sequential schedule \(D = [\text{fetch}] + D_1\) and compose the small-step \([\text{fetch}] \langle \rho, \mu \rangle \downarrow^{c_1, c_2}_{\text{ O}} \langle \mu', \rho' \rangle\) obtained via rule [Fetch-Seq] with the big-step reduction \(\langle \rho, \mu \rangle \downarrow^{D_1}_{\text{ O}} \langle \rho', \mu' \rangle\), thus obtaining the corresponding big-step reduction \(\langle \rho, \mu \rangle \downarrow^{D}_{\text{ O}} \langle \rho', \mu' \rangle\).

\(\Box\)

**j-Equivalence.** In order to prove consistency between the speculative and sequential semantics, we make use of an auxiliary relation called \(j\)-equivalence, defined in Figure 18. This relation captures program configurations of two executions that may have diverged due to a single misprediction (without loss of generality we assume that the second configuration in the relation is at fault). If \(j = \bot\), then rule [Sync] ensures that the configurations are identical, i.e., the two executions have not diverged. If \(j \in \mathbb{N}\), the two executions have diverged on the \(j + 1\)-th (guard) instruction in the reorder buffer, but agree on the first \(j\) instructions and have identical architectural state (memory...
\[
\begin{align*}
\text{Zero} & \quad i_1 = \text{guard}(e^{b_1}, cs^2, p) \quad i_2 = \text{guard}(e^{b_2}, cs_1, p) \quad b_1 \neq b_2 \\
\langle i_1 : [], cs_1, \mu, p \rangle \Rightarrow_0 \langle i_2 : is_2, cs_2, \mu, p \rangle \\
\text{Suc} & \quad i_1 = i_2 \quad \langle is_1, cs_1, \mu, p \rangle \Rightarrow_j \langle is_2, cs_2, \mu, p \rangle \\
\langle i_1 : is_1, cs_1, \mu, p \rangle \Rightarrow_{1+j} \langle i_2 : is_2, cs_2, \mu, p \rangle \\
\text{Synch} & \quad C_1 = C_2 \\
\langle i_1 : is_1, cs_1, \mu, p \rangle & \Rightarrow \langle i_2 : is_2, cs_2, \mu, p \rangle \\
\langle i_1 : is_1, cs_1, \mu, p \rangle & \Rightarrow_0 \langle i_2 : is_2, cs_2, \mu, p \rangle
\end{align*}
\]

Fig. 18. J-Equivalence \( C_1 \equiv_j C_2 \), where \( j \in \mathbb{N} \cup \{\bot\} \).

store and variable map). Formally, rule [Suc] ensures that the first \( J \) instructions in the buffers are equal i.e., \( i_1 = i_2 \), while the command stacks are unrelated because the executions have followed different paths after they have diverged, i.e., in general \( cs_1 \neq cs_2 \). The \( J+1 \) instruction in the buffer of the first configuration (representing the correct execution) is the pending guard instruction, which is mispredicted in the second configuration. This instruction represents a synchronization point for the two executions, therefore rule [Zero] requires the guard instructions in the configurations to be unresolved, i.e., \( i_1 = \text{guard}(e^{b_1}, cs_1, p) \) and \( i_2 = \text{guard}(e^{b_2}, cs_1, p) \), with different predictions, i.e., \( b_1 \neq b_2 \), and that the rollback command stack in \( i_2 \) is identical to the stack of the first configuration, i.e., \( cs_1 \). Notice that the first buffer contains only the guard instruction buffer, which simulates the first execution waiting for the second execution to catch up. This ensures that the two executions will fully synchronize again via rule [Synch] after the pending guard is executed.
(3) Let is\(_1 = is'\) +=[i\(_1\)] ++is'' such that |is'\(_1\)| = n − 1 and thus i\(_1\) is the instruction executed by C\(_1\) and let is\(_2 \) in C\(_2\) be similarly decomposed according to rule [EXECUTE] (Fig. 13b). Since C\(_1 \) = J C\(_2\), then the configurations have equal memory stores, i.e., \(\mu_1 = \mu_2\), variable maps, i.e., \(\rho_1 = \rho_2\), executed instructions \(i_1 = i_2\), and instruction prefixes is'\(_1\) = is'\(_2\) since \(n < J + 1\). Therefore, the transient variable maps computed in rule [EXECUTE] are the same, i.e., \(\rho'_1 = \phi(is_1, \rho_1) = \phi(is_2, \rho_2) = \rho'_2\). Thus, the configurations C\(_1 \) = J C\(_2\) execute the same instruction \(i_1 = i_2\), under the same transient variable maps \(\rho'_1 = \rho'_2\) and memories \(\mu_1 = \mu_2\), i.e., \(\langle is'_1, i_1, is''_1, c\(_1\) \rangle \sim \langle is''_2, c\(_2\) \rangle\) and \(\langle is'_2, i_2, is''_2, c\(_2\) \rangle \sim \langle is''_1, c\(_1\) \rangle\). Since rule [EXECUTE] does not commit changes to the variable map and memory store, it suffices to show that \(o_1 = o_2\) and that there exists a J′ such that the first J′ instructions in is'\(_1\) and is'\(_2\) are identical. Since is'\(_1 = is'_2\), \(i_1 = i_2\), and \(\rho'_1 = \rho'_2\), the configurations step according to the same rule, evaluate equal expressions with equal transient maps and produce equal resolved instructions, i.e., \(i'_1 = i'_2\) in rules [EXEC-ASGN], [EXEC-LOAD], [EXEC-STORE] and nop in rule [EXEC-BRANCH-OK] and [EXEC-BRANCH-MISPREDICT], and thus \(o_1 = o_2\). Furthermore, for rule [EXEC-BRANCH-MISPREDICT] the subfixes is''\(_1\) and is''\(_2\) containing the mispredicted guard are flushed from is''\(_1\) and is''\(_2\) (since \(n < J + 1\)), and thus the configurations become fully synchronized, i.e., C\(_1 \) = J C\(_2\). For all the other rules, the resulting buffers is''\(_1\) and is''\(_2\) are identical to is\(_1\) and is\(_2\), respectively, except for the resolved instructions \(i'_1 = i'_2\), therefore the configurations remain J-equivalent, i.e., C\(_1 \) = J C\(_2\).

Before proving consistency between the sequential and speculative semantics, we define the filtering function \(O \downarrow\), which rewrites rollback and mispredicted observations to the silent observation \(\epsilon\).

**DEFINITION 5 (Filtering Function).** Given an observation trace \(O\), let \(P\) be the set of identifiers of mispredicted guards in \(O\), i.e., \(P = \{p \mid \text{rollback}(p) \in O\}\). Then, we define the filtering function \(O \downarrow\) by case analysis on \(O\):

\[
\epsilon \downarrow = \epsilon
\]

\[
(o \cdot O) \downarrow = \begin{cases} 
\epsilon \cdot O \downarrow & \text{if } o = \text{rollback}(p) \\
\epsilon \cdot O \downarrow & \text{if } o = \text{load}(n, ps) \land ps \land P \neq \emptyset \\
\epsilon \cdot O \downarrow & \text{if } o = \text{store}(n, ps) \land ps \land P \neq \emptyset \\
o \cdot O \downarrow & \text{otherwise}
\end{cases}
\]

The proof of consistency relies on the following more general lemma.

**LEMMA 7 (General Consistency).** For all configurations C\(_1\) and C\(_2\) such that C\(_1 \) = J C\(_2\), valid schedules D\(_1\) and D\(_2\), observations O\(_1\) and O\(_2\) such that \(\text{rollback} \notin O_1\), if C\(_1 \) \(\downarrow^{D_1}_{O_1}\) C\(_1\)' and C\(_2 \) \(\downarrow^{D_2}_{O_2}\) C\(_2\)' then C\(_1\) = J C\(_2\) and O\(_1\) \(\equiv\) O\(_2\).\(\downarrow\)

**Proof Outline.** The proof consists in synchronizing the two executions by carefully reordering the directives of one execution after the other. The fact that the second schedule may interleave directives concerning both correct and misspeculated paths complicates the proof. Therefore, we rely on J-equivalence to deal with configurations that are only partially synchronized. In particular, J-equivalence denotes whether two configurations are fully synchronized (\(J = \perp\)) or synchronized up to J instructions in the reorder buffers (\(J \in \mathbb{N}\)). Using J-equivalence, we can distinguish directives that must be synchronized (e.g., those that retire or execute an instruction at index \(n < J + 1\)), from those that belong to a misspeculated path (e.g., executing a mispredicted instruction...
at index \( n > J + 1 \), as well as synchronization points (i.e., resolving the pending mispredicted guard at index \( n = J + 1 \)).

**Proof.** By induction on the big-step reductions and the \( J \)-equivalence relation. If the first reduction is \([\text{DONE}]\), then \( C_{1} = ([\_], [\_], \mu, \rho) = C'_{1} \) and \( O_{1} = \varepsilon \). Since the reorder buffer in \( C_{1} \) is empty and \( C_{2} = \emptyset \), we deduce that \( J = \bot \) (only rule [\text{SYNCH}] applies to empty buffers) and thus the buffer of \( C_{2} \) is also empty. Therefore, the second reduction is also \([\text{DONE}]\), thus \( C_{2} = C'_{2} \) and \( O_{2} = \varepsilon \), and hence \( C_{1}' = C_{2}' \) and \( O_{1} = O_{2} \bot \). In the inductive case both big-step reductions follow rule [\text{STEP}], therefore for \( i \in \{1, 2\} \), we have schedules \( D_{i} = d_{i} : D'_{i} \), observation traces \( O_{i} = o_{i} : O'_{i} \), and two pairs of small- and big-steps, i.e., \( C_{1} \xrightarrow{d_{i}} o_{i} \ C'_{1} \) and \( C'_{1} \models_{O'_{i}} D'_{i} \ C''_{1} \). We will refer to these names in the rest of the proof.

If \( d_{1} = d_{2} \), then we apply Lemma 6 (if the index \( J \) and the directives \( d_{1} = d_{2} \) fulfill either of the conditions of the lemma)\(^{18}\) and induction. In the following, we focus on the general case where \( d_{1} \neq d_{2} \) and rely on \( J \)-equivalence to determine whether the configurations are fully synchronized \((J = \bot)\) or a mispredicted guard is pending in the second configuration \((J \in \mathbb{N})\) at index \( J + 1 \).

\[(1) \ J = \bot \text{ and } d_{1} = \text{fetch } b_{1}, d_{2} = \text{fetch } b_{2}, \text{ and } b_{1} \neq b_{2} \text{. Since the first execution does not contain mispredictions (\text{rollback} \neq O_{1}) \text{, we know that the second configuration enters a mispredicted branch and diverges temporarily from the first. The initial configurations } C_{1} \text{ and } C_{2} \text{ are synchronized } C_{1} \text{ and } C_{2} \text{ by the command fetched, i.e., the } C_{1} \text{ and } C_{2} \text{ are identical, i.e., } C_{1} = C_{2} \text{ and } C_{1} \models_{O'_{1}} D'_{1} \text{ and } C_{2} \models_{O'_{2}} D'_{2} \text{ and conclude that the final configurations are identical, i.e., } C''_{1} = C''_{2} \text{, and the observation traces are equal up to permutation and filtering, i.e., } O'_{1} = O'_{2} \bot \text{ thus } \varepsilon \cdot O'_{1} \equiv \varepsilon \cdot O'_{2} \bot \text{, i.e., } O_{1} = O_{2} \bot \text{.}
\]

\[(2) \ J = \bot \text{ and } d_{2} = \text{fetch } (d_{2} = \text{fetch } b_{2}) \text{ and } d_{1} \neq d_{2} \text{. Let } c_{2} \text{ be the command fetched, i.e., the command on top of the stack } cs_{2} \text{ of configuration } C_{2} \text{. Similarly, let } cs_{1} \text{ be the stack of the first configuration } C_{1} \text{. Since the configurations are } J \text{-equivalent, i.e., } C_{1} = C_{2} \text{ and synchronized } (J = \bot), \text{ then } cs_{1} = cs_{2} \text{ from rule [\text{SYNCH}] \text{, therefore there exists a command } c_{1} = c_{2} \text{ on top of } cs_{1} \text{.}
\]

Then, we anticipate the next \text{fetch} directive in the first schedule (one must exist because schedule \( D_{1} \) is valid) in order to fetch \( c_{1} \) and preserve \( J \)-equivalence. Formally, let \( D_{1} = D'_{1} \lvert \text{fetch} \rvert \lvert D''_{1} \), such that sub-schedule \( D'_{1} \) contains no \text{fetch} directives and let \( D''_{1} = \lvert \text{fetch} \rvert \lvert D'_{1} \rvert \lvert D''_{1} \) be the alternative schedule where directive \text{fetch} is anticipated. Intuitively, schedule \( D''_{1} \) remains valid (Def. 3) for \( C_{1} \) because the new instruction fetched end up at the end of the reorder buffer \( is_{1} \) and thus do not interfere with the \text{retire} directive in \( D'_{1} \), which operate on \( is_{1} \), and with the \text{exec} \( n \) instructions in \( D'_{1} \), which remain valid at their original index \( 1 \leq n \leq |is_{1}| \). Since \( D_{1} \) and \( D''_{1} \) are valid, we apply Lemma 3 and deduce that there exists an intermediate configuration \( C''_{1} \) such that \( C_{1} \xrightarrow{\text{fetch}} \varepsilon \ C''_{1} \), which then reduces to the same final configuration \( C''_{1} \), i.e., \( C''_{1} \models_{O'_{1} \bot} D'_{1} \) for some observation trace \( O'_{1} \). Furthermore, Lemma 3 ensures that the observation trace \( O'_{1} = \varepsilon \cdot O'_{1} \)

\(^{18}\) Otherwise, if \( d_{1} = d_{2} \) but \( J \neq \bot \), see case 4. If \( J \in \mathbb{N} \), \( d_{1} = d_{2} = \text{exec } n \), but \( n = J + 1 \), see case 7. In this case, notice that \( n \) cannot be greater than \( J + 1 \), i.e., \( n \neq J + 1 \), because the buffer of \( C_{1} \) has only \( J + 1 \) instructions, which contradicts the step \( C_{1} \xrightarrow{\text{exec } n} \).

generated by the alternative schedule $D''_0'$ is equivalent up to permutation to the observation trace $O_1 = o_1 \cdot O_1'$ generated by the original schedule $D_1$, i.e., we have that $O_1 \equiv O''_1'$. Since $C_1 = J C_2$, $J = \perp, C_1 \xrightarrow{\text{fetch}} C''_1'$, and $C_2 \xrightarrow{\text{fetch}} C''_2'$, we apply Lemma 6 and obtain that there exists a $J'$ such that the intermediate configurations are $J'$-equivalent, i.e., $C''_1' = J C''_2'$. Then, we apply our induction hypothesis to reductions $C''_1' \xrightarrow{\text{D}} C''_1'$ and $C''_2' \xrightarrow{\text{D}} C''_2'$, deduce that the final configurations are identical, i.e., $C''_1' = C''_2'$, and the observation traces are equivalent up to permutation and filtering $O''_1' \equiv O''_2' \perp$ and thus $\epsilon \cdot O''_1' \equiv \epsilon \cdot O''_2'$, i.e., $O''_1' \equiv O''_2' \perp$. Finally, from $O_1 \equiv O''_1'$ and $O''_1' \equiv O''_2'$ we derive $O_1 \equiv O_2 \perp$ by transitivity. Notice that if $d_2 = \text{fetch } b_2$, then the next fetch directive in $D_1$ is of the same type, i.e., $\text{fetch } b_1$, and the proof follows as above if $b_1 = b_2$ and as in case 1) if $b_1 \neq b_2$.

(3) $J = \perp$ and $d_1 = \text{fetch } (d_1 = \text{fetch } b_1)$ and $d_1 \neq d_2$. Symmetric to the previous case.

(4) $J \neq \perp$ and $d_2 = \text{fetch } (d_2 = \text{fetch } b)$. Since $J \neq \perp$, a mispredicted guard is pending in the buffer of $C_2$ at index $J + 1$ and thus the instruction fetched in the step $C_2 \xrightarrow{\text{fetch}} C''_2'$ is going to be flushed eventually. We observe that this instruction is appended at the end of the buffer, thus at an index greater than $J + 1$, and therefore $C_1$ remains $J$-equivalent with $C''_2'$, i.e., $C_1 = J C''_2'$. We apply our induction hypothesis to $C_1 \xrightarrow{\text{D}} C''_1'$ and $C''_2' \xrightarrow{\text{D}} C''_2'$ and deduce that the final configurations are identical, i.e., $C''_1' = C''_2'$, and the observation traces are equivalent up to permutation and filtering, i.e., $O_1 \equiv O''_2' \perp$. Notice that the step from $C_2$ to $C''_2'$ generates a silent observation $\epsilon$, i.e., $O_2 = \epsilon \cdot O''_2' = O''_2'$, and therefore $O_1 \equiv O''_2' \perp$ implies $O_1 \equiv O_2 \perp$.

(5) $d_2 = \text{exec } n$ and $n > J + 1$. The instruction executed in the step $C_2 \xrightarrow{\text{exec } n} C''_2'$ is at index $n > J + 1$ and thus it belongs to a mispredicted path and will be squashed. In particular, the second buffer contains a pending, mispredicted guard instruction at index $J + 1$ with prediction identifier $p$ such that $\text{rollback}(p) \in O''_2'$. Therefore the observation $o_2$ generated in the step is tainted with the prediction identifier $p$ of the guard and thus rewritten to silent event $\epsilon$ (Def. 5), i.e., $(o_2 \cdot O''_2') \perp = \epsilon \cdot O''_2' \perp = O''_2'.\perp$. The resolved instruction is reinserted in the buffer at its original index $n > J + 1$ and thus $C_1$ remains $J$-equivalent to $C''_2'$, i.e., $C_1 = J C''_2'$, and the lemma follows by induction on $C_1 \xrightarrow{\text{D}} C''_1'$ and $C''_2' \xrightarrow{\text{D}} C''_2'$ as in the previous case.

(6) $d_2 = \text{exec } n$ and $n < J + 1$ or $J = \perp$. Since $n < J + 1$ or $J = \perp$, the instruction executed in step $C_2 \xrightarrow{\text{exec } n} C''_2'$ will not be squashed and thus we synchronize the first schedule to execute it as well. Let $i\epsilon_2 = i\epsilon_2' + i\epsilon_2''$ be the reorder buffer of $C_2$ such that $|i\epsilon_2'| = n - 1$ and thus $i\epsilon_2$ is the executed instruction at index $n$. From $J$-equivalence, i.e., $C_1 = J C_2$, we deduce that the buffer $i\epsilon_1$ of $C_1$ can be decomposed like $i\epsilon_1 = i\epsilon_1' + [i\epsilon_1]$ where $|i\epsilon_1'| = n - 1 < J$ and thus $i\epsilon_1' = i\epsilon_2'$ and $i_1 = i_2$. Then, we observe that directive $d_2 = \text{exec } n$ is valid also for $C_1$ (Def. 2) and we adjust the schedule $D_1$ to anticipate the directive that executes that instruction.\footnote{In particular, the fact that $d_2 = \text{exec } n$ is valid for $C_2$ implies that the data-dependencies of $i_1 = i_2$ are also resolved in the prefix $i\epsilon_2' = i\epsilon_2''$ and thus $d_2$ is valid for $C_1$.} Formally, let $D_1 = D_1' + [d_1''] + D_1''$ be the original schedule, where $d_1''$ is the exec $n'$ directive that evaluates $i_1$ and let $D_1''' = [\text{exec } n] + D_1'' + D_1''$ be the alternative schedule where the execution of $i_1$ is anticipated.\footnote{The position $n'$ where instruction $i_1$ lies in the buffer when it gets executed following $D_1$ can be different from $n$ because other instructions may be retired in $D_1''$, i.e., in general $n \neq n'$.} Intuitively, schedule $D_1'''$ is valid because directive $\text{exec } n$ does not cause any rollback...
(rollback $\not\in O_1$) and thus it cannot interfere with any fetch, exec, or retire directive in $D'_1$, which were already valid in the original schedule. Since $D'_1$ and $D''_1$ are valid, we apply Lemma 3 and deduce that there exists an intermediate configuration $C''_1$ such that $C_1 \xrightarrow{\text{exec } n} C''_1$, which then reduces to the same final configuration $C'_1$, i.e., $C''_1 \Downarrow_{O_1''} D'_1 \Downarrow_{O_1''} C'_1$ for some observation trace $O'_1$.

Furthermore, Lemma 3 ensures that the observation trace $O'_1 = a'_1; O'_1$ generated by the alternative schedule $D''_1$ is equivalent up to permutation to the observation trace $O_1 = a_1; O'_1$ generated by the original schedule $D_1$, i.e., $O_1 \equiv O'_1$. Since $C_1 = f_{C_2, C_1} \xrightarrow{\text{exec } n} C''_1$, and $C_2 \xrightarrow{\text{exec } n} C''_2$, we apply Lemma 6 and obtain that the observations generated are equivalent, i.e., $O'_1 = O'_2$, and there exists $f'$ such that the intermediate configurations are $f'$-equivalent, i.e., $C''_1 = f'_1 \Downarrow C''_2$. Then, we apply our induction hypothesis to reductions $C''_1 \Downarrow_{O_1''} D'_1 \Downarrow_{O_1''} C''_1$ and $C'_1 \Downarrow_{O_2''} D'_2 \Downarrow_{O_2''} C''_2$ and deduce that the final configurations are identical, i.e., $C'_1 = C''_2$, and the observation traces are equivalent up to permutation and filtering, i.e., $O'_1 = O'_2 \downarrow$. From $O'_1 \equiv O'_2 \downarrow$ and $O'_1 = O'_2$, we have $O'_1 = O'_2 \equiv O'_2 \downarrow$, i.e., $O''_1 = O_2 \downarrow$. Finally, from $O_1 \equiv O''_1$ and $O''_1 \equiv O_2 \downarrow$ we derive $O_1 \equiv O_2 \downarrow$ by transitivity.

(7) $d_2 = \text{exec } n$ and $n = f + 1$. Since $n = f + 1$, the second configuration $C_2$ resolves the mispredicted guard and performs a rollback, i.e., the second small-step is $C_2 \xrightarrow{\text{exec } n} C'_2$, where $p$ is the identifier of the guard. Then, we anticipate the execution of the corresponding guard instruction in the first configuration $C_1$, so that the two executions synchronize again. Let $i_s = i'_s + [i_z] + i'_s$ be the buffer of $C_2$, where $|i'_s| = n - 1$ and thus $i_s$ is the mispredicted guard instruction. From $f$-equivalence, i.e., $C_1 = f_{C_2, C_1}$, we deduce that the buffer $i_s$ of $C_1$ can be decomposed similarly, i.e., $i_s = i'_s + [i_z]$, where $|i'_s| = n - 1$ and thus $i'_s = i'_s$ and $i_z$ is the corresponding guard instruction, but correctly predicted. In particular, from rule [Zero], we learn that $i_z = \text{guard} (e^{b_1}, c_s, p)$ and $i_z = \text{guard} (e^{b_2}, c_s, p)$, where $b_1 \neq b_2$ and $c_s$ is the command stack of $C_1$. Let $D_1 = O_1 \Downarrow_{O_1''} C''_1 \Downarrow$ be the original schedule where $D'_1$ is the $\text{exec } n'$ directive that evaluates the guard instruction $i_z$ and let $D''_1 = [\text{exec } n] \Downarrow D'_1 \Downarrow$ be the valid, alternative schedule that anticipates it. We apply Lemma 3 to valid schedules $D_1$ and $D''_1$ and deduce that there exists an intermediate configuration $C''_1$ such that $C_1 \xrightarrow{\text{exec } n} C''_1$ via rule [EXEC-BRANCH-OK], which then reduces to the same final configuration $C''_1$, i.e., $C''_1 \Downarrow_{O_1''} D'_1 \Downarrow_{O_1''} C''_1$ for some observation trace $O'_1$. Furthermore, the lemma ensures that the observation trace $O''_1 = e \cdot O'_1$ generated by the alternative schedule $D''_1$ is equivalent up to permutation to the observation trace $O_1 = o_1; O'_1$ generated by the original schedule $D_1$, i.e., $O_1 \equiv O''_1$. Notice that rule [EXEC-BRANCH-OK] rewrites the buffer $i_s$ in $C_1$ to $i'_s = i_z + [\text{nop}] + [\text{nop}]$ in $C'_2$, which is identical to the buffer $i'_s = i_z + [\text{nop}]$ obtained from the step $C_2 \xrightarrow{\text{exec } n} \text{rollback}(p)$ $C'_2$ via rule [EXEC-BRANCH-MISPREDICT], which additionally replaces the command stack of $C_2$ with $c_s$ in $C'_2$. As a result, the intermediate configurations $C''_1$ and $C''_2$ have identical reorder buffers ($i'_s = i'_s$) and command stacks ($c_s = c_s$), memory stores and variable maps, which are equal in $C_1 = f_{C_2, C_1}$ and left unchanged in $C''_1$ and $C''_2$ by the small steps. Therefore, the configurations $C''_1$ and $C''_2$ are fully synchronized again, i.e., $C''_1 =_{\Downarrow} C''_2$ via rule [SYNC] and we can apply our induction hypothesis to the reductions $C''_1 \Downarrow_{O_1''} D'_1 \Downarrow_{O_1''} C''_1$ and $C''_2 \Downarrow_{O_2''} D''_2 \Downarrow_{O_2''} C''_2$. Then, we deduce that the final configurations are identical, i.e., $C''_1 = C''_2$, and that the observation traces are equal up to permutation and filtering, i.e., $O''_1 \equiv O'_2 \downarrow$. Notice that the filter function rewrites the rollback observation $\text{rollback}(p)$ to the silent observation $e$ (Def 5) in the second observation trace, i.e., $O_2 \downarrow \equiv (\text{rollback}(p) \cdot O'_2) \downarrow \equiv e \cdot O'_2 \downarrow \equiv O'_2 \downarrow$, and that the alternative observation trace starts with a silent observation as well, i.e., $O''_1 \equiv e \cdot O'_1$, therefore from $O''_1 \equiv O'_2 \downarrow$ above we obtain $O''_1 \equiv O_2 \downarrow$. Finally, from $O_1 \equiv O''_1$ and $O''_1 \equiv O_2 \downarrow$ we derive $O_1 \equiv O_2 \downarrow$ by transitivity.
(8) \(d_2 = \text{retire}\). Let \(i_2\) be the instruction that gets retired, i.e., the first instruction in the reorder buffer of \(C_2\), and let \(i_1\) be the first instruction in the buffer of \(C_1\). From \(J\)-equivalence, i.e., \(C_1 \equiv_J C_2\), we know that either the two configurations are fully synchronized, i.e., \(J = \perp\), or \(i_2\) precedes the mispredicted guard at index \(J + 1\), i.e., \(J > 0\).\(^{21}\) Therefore, from \(C_1 \equiv_J C_2\) it follows that \(i_1 = i_2\) and thus also instruction \(i_1\) is resolved and can be retired, i.e., directive retire is valid for \(C_1\). Then, we anticipate the next retire directive in the first schedule (one must exist because \(D_1\) is valid), in order to retire \(i_2\) and preserve \(J\)-equivalence. Formally, let \(D_1 = D'_1 + [\text{retire}] + D''_1\) be the original schedule, where \(D'_1\) does not contain any retire directive. Then, we define the alternative, valid schedule \(D_1'' = [\text{retire}] + D'_1 \uparrow + D''_1\), where \(D'_1 \downarrow\) decreases by 1 the indexes of all execute directives in \(D'_1\) to account for the early retirement of \(i_1\). We apply Lemma 3 to valid schedules \(D_1\) and \(D_1''\) and deduce that there exists an intermediate configuration \(C_1'''\) such that \(C_1 \xrightarrow{\text{retire}} C_1''\), which then reduces to the same final configuration \(C''\), i.e., \(C_1''' \downarrow_{O''_1}^{D'_1 + D''_1} C''\), for some observation trace \(O''_1\). Furthermore, Lemma 3 ensures that the observation trace \(O''_1 = C'_{\text{exec}} \otimes C'_{\text{synch}}\) generated by the alternative schedule \(D_1''\) is equivalent up to permutation to the observation trace \(O_1 = C_1 \otimes C_2\) generated by the original schedule \(D_1\), i.e., we have \(O_1 \equiv O''_1\). Since \(C_1 \equiv_J C_2\), \(C_1 \xrightarrow{\text{retire}} C_1''\), and \(C_2 \xrightarrow{\text{retire}} C_2''\), we apply Lemma 6 and obtain that the observations generated are equivalent, i.e., \(O_2 \equiv O''_2\), and there exists \(j'\) such that the intermediate configurations remain \(j'\)-equivalent, i.e., \(C_1'' \equiv_{j'} C_2''\). Then, we apply our induction hypothesis to reductions \(C_1'' \downarrow_{O''_1}^{D'_1 + D''_1} C''\) and \(C_2'' \downarrow_{O''_2}^{D''_2} C''\) and deduce that the final configurations are identical, i.e., \(C_1'' = C_2''\), and the observation traces are equivalent up to permutation and filtering, i.e., \(O_1'' \equiv O_2''\). From \(O_1'' \equiv O_2''\) and \(O_2 \equiv O''_2\), we have \(O_1 \equiv O''_1\). Finally, from \(O_1 \equiv O''_1\) and \(O''_1 \equiv O_2\), we derive \(O_1 \equiv O_2\) by transitivity.

\[\square\]

**Theorem 3 (Speculative Consistency).** For all programs \(c\), initial memory stores \(\mu\), variable maps \(\rho\), and valid directives \(D\), such that \(\langle \mu, \rho \rangle \downarrow_{O}^{D_1} \langle \mu', \rho' \rangle\) and \(\langle \mu, \rho \rangle \downarrow_{O}^{D_2} \langle \mu'', \rho'' \rangle\), then \(\mu' = \mu''\) and \(\rho' = \rho''\), and \(O \equiv O'\).

**Proof.** First, we apply Lemma 4 (sequential consistency) to the sequential reduction \(\langle \mu, \rho \rangle \downarrow_{O}^{D_1} \langle \mu', \rho' \rangle\) and obtain a valid, sequential schedule \(D'\) such that \(\langle \mu, \rho \rangle \downarrow_{O}^{D'} \langle \mu', \rho' \rangle\) where rollback \(\not\in O\). Let \(C' = \langle \mu, \rho \rangle, C' = \langle \mu', \rho' \rangle\), and \(C'' = \langle \mu'', \rho'' \rangle\). We now have two speculative reductions \(C \downarrow_{O}^{D'} C'\) and \(C \downarrow_{O}^{D''} C'\), where \(C \equiv_J C\) from rule [SYNCH]. Then, we apply Lemma 7 (general consistency) to these reductions and obtain \(O \equiv O'\) and \(C' = C''\), which implies \(\mu' = \mu''\) and \(\rho' = \rho''\).

\[\square\]

**B.2 Security**

**L-Equivalence.** The soundness proof of our type system relies on a relation called \(L\)-equivalence, which intuitively relates configurations that are indistinguishable to an attacker that can observe only public data, which we identify with security label \(L\). Figure 19 formally defines \(L\)-equivalence for the categories of our calculus. Two configurations \(C_1 = \langle is_1, cs_1, \mu_1, \rho_1 \rangle\) and \(C_2 = \langle is_2, cs_2, \mu_2, \rho_2 \rangle\) are \(L\)-equivalent under typing environment \(\Gamma\), written \(\Gamma \vdash C_1 \approx_L C_2\), if their reorder buffers are \(L\)-equivalent, i.e., \(\Gamma \vdash is_1 \approx_L is_2\), they have the same command stacks, i.e., \(cs_1 = cs_2\), and their architectural state, i.e., variable maps and memory stores, are related, i.e., \(\mu_1 \approx_L \mu_2\) and \(\rho_1 \approx_L \rho_2\), in particular, if \(J = 0\), then rule [ZERO] implies that instruction \(i_2\) is the mispredicted guard, which cannot be retired, contradicting the step \(C_2 \xrightarrow{\text{retire}} C_2'\).

\[\square\]
respectively. Syntactic equivalence for the command stacks ensures that programs cannot leak secret data through the instruction cache, e.g., by branching on secret data. For the architectural state $L$-equivalence is standard; values of public variables and stored at public memory addresses must be equal (rules $\text{VARMAP}$ and $\text{MEMORY}$) in Fig. 19c).

Figure 19b defines pointwise $L$-equivalence for reorder buffers, which ensures that related buffers have the same length. Instructions are related ($\Gamma \vdash i_1 \approx_L i_2$ in Fig. 19a) only if they are of the same kind, e.g., both instructions are guards, loads, assignments etc. For most instructions, $L$-equivalence is just syntactic equivalence, e.g., for rules $\text{NOP}$ and $\text{FAIL}$. In particular, to avoid leaking through the instruction cache, we demand syntactic equivalence for guard instructions (rule $\text{GUARD}$), where all components (condition $e$, predicted value $b$, rollback stack $cs$, and prediction identifier $p$) must be identical. Similarly, to avoid leaks through the data cache, rule $\text{LOAD}$ relates load instructions as long as they read the same address and update the same variable. We impose a similar restriction for store instructions ([$\text{STOREL}$, $\text{STOREH}$]) and additionally require equal expressions for stores that update public addresses ([$\text{STOREH}$]), which are identified by the label annotation $L$ that decorates the instruction itself. Protect instructions allow their argument to be different because they allow their arguments are allowed to be evaluated transiently ($\text{PROTECT}$). Assignments are related as long as they update the same variable. If the variable is public and typed stable in the typing environment, the values assigned must be identical ([$\text{ASGN}_{\text{LAS}}$]), but can be different for secret or transient variables ([$\text{ASGN}_{\text{HAT}}$]). This relaxation permits public, but transient, variables to temporarily assume different, secret values resulting from off-bounds array reads. Later, we prove that when these assignments are retrieved, these values are necessarily public, and therefore equal.

Since transient variable maps are computed from the pending assignments in the reorder buffer (Fig. 13d), we relax $L$-equivalence in rule $\text{TRANSIENT-VARMAP}$ similarly to rule $\text{ASGN}_{\text{LAS}}$.

**Lemma 8 ($L$-Equivalence for Transient Variable Maps).** If $\Gamma \vdash i_1 \approx_L is_2$ and $\Gamma \vdash p_1 \approx_L p_2$, then $\Gamma \vdash \phi(i_1, p_1) \approx_L \phi(is_2, p_2)$.

**Proof.** By induction on the $L$-equivalence judgment for reorder buffers (Fig. 19b). The base case $\text{RB-EMPTY}$ follows by hypothesis ($\Gamma \vdash p_1 \approx_L p_2$). In the inductive case $\text{RB-CON}$, we know that the next instructions in the buffers are equivalent, i.e., $\Gamma \vdash i_1 \approx_L i_2$ and the rest of the buffers are equivalent, i.e., $\Gamma \vdash is_1 \approx_L is_2$. In order to apply the induction hypothesis, we need to show first that updating equivalent maps with equivalent instructions gives equivalent variable maps. We proceed by case analysis on the judgment for equivalent instruction ($\Gamma \vdash i_1 \approx_L i_2$ in Fig. 19a). By definition of the judgment, $i_1$ and $i_2$ are the same kind of instruction and cases $\text{NOP}$, $\text{FAIL}$, $\text{GUARD}$, $\text{STORE}$ do not update the transient variable maps (they are handled by the last case of the function $\phi$ in Fig. 13d), which remain equivalent ($\Gamma \vdash p_1 \approx_L p_2$). In case $\text{LOAD}$, both instructions are identical unresolved loads ($i_1 = i_2 = x := \text{load}(e)$), therefore variable $x$ becomes undefined in both variable maps, which remain equivalent regardless of the sensitivity and type of $x$, i.e., $\Gamma \vdash p_1[x \mapsto \bot] \approx_L p_2[x \mapsto \bot]$ from rule $\text{TRANSIENT-VARMAP}$. Similarly, in case $\text{ASGN}_{\text{LAS}},$ the two instructions are identical assignments ($i_1 = i_2 = x := e$) and variable $x$ is public ($x \in L$) and stable ($\Gamma(x) = S$). Since the assignments are identical, variable $x$ gets updated in the same way in the respective variable maps, which remain equivalent whether the assignments are resolved ($e = v$ and $\Gamma \vdash p_1[x \mapsto v] \approx_L p_2[x \mapsto v]$) or not ($e \neq v$ and $\Gamma \vdash p_1[x \mapsto \bot] \approx_L p_2[x \mapsto \bot]$). In case $\text{ASGN}_{\text{HAT}}$, the two instructions update variable $x$ with possibly different expressions ($i_1 = x := e_1 \neq x := e_2 = i_2$). This includes the cases where one instruction is resolved (e.g., $e_1 = v_1$) and the other is not ($e_2 \neq v_2$), or both are resolved, but to different values ($e_1 = v_1 \neq v_2 = e_2$). In these cases, variable $x$ assumes different values in the resulting transient variable maps, but the maps remain nevertheless equivalent according to rule $\text{TRANSIENT-VARMAP}$, e.g., $\Gamma \vdash p_1[x \mapsto v_1] \approx_L p_2[x \mapsto \bot]$, because $x$ is secret ($x \notin L$) or transient ($\Gamma(x) = T$). In case $\text{PROTECT}$, both variables are protected, i.e.,
\( \Gamma \vdash x := \text{protect}(e_1) \approx_L x := \text{protect}(e_2) \), and therefore, they are undefined in the transient variable maps (regardless of whether the expressions are resolved or not) and thus the maps remain related, i.e., \( \Gamma \vdash \rho_1[x \mapsto \bot] \approx_L \rho_2[x \mapsto \bot] \).

Now that we have established that after processing the next instructions in the buffers the resulting transient variable maps are \( L \)-equivalent, we conclude the proof of the lemma by applying the induction hypothesis.

\[\square\]

**Lemma 9 (Evaluations of Public Stable Expressions).** If \( \Gamma \vdash e : S, \vdash_{\text{ct}} e : L, \) and \( \Gamma \vdash \rho_1 \approx_L \rho_2, \) then \( \llbracket e \rrbracket^{\rho_1} = \llbracket e \rrbracket^{\rho_2} \).

**Proof.** By induction on the typing judgments, where the transient-flow and constant-time type systems ensure that the sub-expressions of \( e \) are also typed as \( L \) and \( S \). (The length and the base of secret arrays are assumed to be public information like in [Protzenko et al. 2019]). All the inductive cases follow directly by induction hypothesis. The base case for values is trivial and we derive equality in the base case for variables from \( L \)-equivalence of the transient variable maps (rule [Transient-V arMap] in Fig. 19e).

\[\square\]

**Lemma 10 (Equal Guard Identifiers).** If \( \Gamma \vdash i_{s_1} \approx_L i_{s_2}, \) then \( \llbracket i_{s_1} \rrbracket = \llbracket i_{s_2} \rrbracket \).

**Proof.** By induction on the \( L \)-equivalence judgment for reorder buffers (Fig. 19b). The only interesting case is the inductive case, when the \( L \)-instructions at the beginning of the reorder buffer are guard instructions. By inspection of the definitions from Figure 19a, we see that when this occurs, both instructions are guards and share the same prediction identifier (rule [Guard]).

In the following we write \( \Gamma \vdash_{\text{sct}} C \) to indicate that configuration \( C \) is well-typed with respect to both the constant-time type system (Fig. 16), i.e., \( \Gamma \vdash_{\text{ct}} C \), and the transient-flow type system (Fig. 15), i.e., \( \Gamma \vdash C \), under the respective typing contexts.

**Lemma 11 (L-equivalence Preservation).** If \( \Gamma \vdash_{\text{sct}} C_1, \Gamma \vdash_{\text{sct}} C_2, \Gamma \vdash C_1 \approx_L C_2, C_1 \xrightarrow{d_{o_1}} C_1', \) and \( C_2 \xrightarrow{d_{o_2}} C_2', \) then \( C_2 \approx_L C_2' \) and \( o_1 = o_2 \).

**Proof.** By case analysis on the reduction steps. From \( \Gamma \vdash C_1 \approx_L C_2 \), we know that the respective components of the configurations are also \( L \)-equivalent. In particular, \( \Gamma \vdash i_{s_1} \approx_L i_{s_2} \) and \( cs_1 = cs_2 \), thus the reorder buffers and the command stacks have the same structure, length, and are point-wise related. Then, since the directive \( d \) is the same in both reduction, equivalent commands (\( c_1 = c_2 \)) are fetched from the command stacks and \( L \)-equivalent instructions (\( \Gamma \vdash i_1 \approx_L i_2 \)) are executed and retired.

\( \triangleright \) Fetch Stage (\( d = \text{fetch} \) or \( d = \text{fetch b} \)). The rules from this stage (Fig. 13a) always generate the empty event, i.e., \( o_1 = e = o_2 \), and affect only the reorder buffers and the command stacks. Thus, to prove \( \Gamma \vdash C_1' \approx_L C_2' \), we only need to show that the resulting buffers and stacks are related. First, we observe that the rules only pop and push commands from the command stacks and append instructions at the end of the reorder buffer. Since the definition of \( L \)-equivalence for buffers is structural (Fig. 19b) and for stacks is syntactic equivalence, these operations preserve \( (L\text{-})\)equivalence as long as they are applied to \((L\text{-})\)equivalent arguments. This is exactly the case, for control-flow commands, which are popped, flattened into simpler, equal commands, and pushed back on the stack, e.g., in rules [Fetch-Seq] and [Fetch-While]. Similarly, the other commands are translated directly into related instructions (Fig. 19a), e.g., by rule [Nop] for [Fetch-Skip], [Fail] for [Fetch-Fail], and [Guard] for [Fetch-If-True] and [Fetch-If-False], where the prediction \( b = \text{true} \).
or \( b = \text{false} \) is determined from the same attacker-provided directive \texttt{fetch} \( b \).ootnote{We assume that the generation of the fresh prediction identifier (fresh(\( p \))) is deterministic and secret independent. For example, the configuration could contain a counter \( p \) containing the next fresh identifier, which gets incremented each time a conditional is fetched.} When fetching assignments, i.e., rule [\texttt{Fetch-Asgn}], the corresponding assignments instructions appended at the end of the reorder buffers are related, either by rule [\texttt{Asgn}_{\text{LAS}}] or [\texttt{Asgn}_{\text{HVT}}] depending on the sensitivity and type of the variable. For rule [\texttt{Fetch-Ptr-Store}], we rely on the label annotation that decorates the command (i.e., label \( \ell \) for \( \ast_{\ell} e_{1} = e_{2} \)) to relate the corresponding instructions via rule [\texttt{STORE}_{\ell}] if \( \ell = L \), or [\texttt{STORE}_{\text{H}}]), otherwise. For array reads ([\texttt{Fetch-Array-Load}]) and writes ([\texttt{Fetch-Array-Store}]), we observe that the \textit{same} bounds-checking code is generated and pushed on the commands stack. Rules [\texttt{Fetch-Protect-Ptr}, \texttt{Fetch-Protect-Array}, \texttt{Fetch-Protect-Expr}, \texttt{Fetch-Protect-SLH}] fetch \textit{identical} protect commands and decompose them into equal commands ([\texttt{Fetch-Protect-Ptr}, \texttt{Fetch-Protect-Array}, \texttt{Fetch-Protect-SLH}]) or equivalent \textit{protect} instructions ([\texttt{Fetch-Protect-Expr}]).ootnote{We prove security for both the hardware- and SLH-based implementation of \texttt{protect}, but we assume that the same implementation is used in both executions.}

\> Execute Stage (\( d = \text{exec} \ n \)). Instruction [\texttt{Execute}] from Figure 13b, selects the \( n \)-th instruction from the reorder buffer, computes the transient variable map, and relies on a separate reduction relation to generate an observation and compute the resulting reorder buffer and command stack. Since the initial buffers are \textit{structurally} related, i.e., \( \Gamma \vdash is_{1} \approx_{L} is_{2} \) and the directive \texttt{exec} \( n \) is the same in both reductions, then the buffers are split in related components, i.e., prefixes \( \Gamma \vdash is'_{1} \approx_{L} is'_{2} \) such that \( |is'_{1}| = |is'_{2}| = n - 1 \), suffixes \( \Gamma \vdash is''_{1} \approx_{L} is''_{2} \), and \( n \)-th instructions \( \Gamma \vdash i_{1} \approx_{L} i_{2} \). Therefore, their transient variable maps are related, i.e., \( \Gamma \vdash \phi(is'_{1}, \rho_{1}) \approx_{L} \phi(is'_{2}, \rho_{2}) \) by Lemma 8 (notice that \( \rho_{1} \approx_{L} \rho_{2} \) implies \( \Gamma \vdash \rho_{1} \approx_{L} \rho_{2} \) and we conclude by applying Lemma 12.

\> Retire Stage (\( d = \text{retire} \)). The rules from this stage (Fig. 13c) remove the resolved instruction at the beginning of the reorder buffer, and update the variable map and the memory store accordingly. Therefore, to prove \( \Gamma \vdash C'_{2} \approx_{L} C'_{1} \), we only need to show that the resulting reorder buffers, variable maps, and memory stores are \( L \)-equivalent. Since the reorder buffers are initially related, we have \( \Gamma \vdash (i_{1} : is_{1}) \approx_{L} (i_{2} : is_{2}) \), and from rule [\texttt{RB-Cons}] we learn that the instructions retired are related, i.e., \( \Gamma \vdash i_{1} \approx_{L} i_{2} \), and the tails of the buffers remain related in the resulting configuration, i.e., \( \Gamma \vdash is_{1} \approx_{L} is_{2} \). Then, we prove that the architectural state (variable maps and memory stores) remain related when updated by the related instructions. Since the instructions are related, the two configurations perform the same reduction step.

Cases [\texttt{Retire-Nop}] and [\texttt{Retire-Fail}] are trivial. We only point out that in either case the two reductions produce the same event (i.e., \( o_{1} = \epsilon = o_{2} \) and \( o_{1} = \text{fail} = o_{2} \)) and update (or empty) the reorder buffers and the command stack in the same way. For [\texttt{Retire-Asgn}], if the assignments involve a \textit{public} variable (e.g., \( x \in L \)) and the variable is stable (i.e., \( \Gamma(x) = S \)), then the resolved values are the same, i.e., \( i_{1} = (x := v_{1}) = i_{2} \) from rule [\texttt{Asgn}_{\text{LAS}}], and the resulting variable maps are \( L \)-equivalent, i.e., \( \rho_{1}[x \mapsto v_{1}] \approx_{L} \rho_{2}[x \mapsto v_{2}] \) from rule [\texttt{VARMAP}] in Fig. 19c. If the variable is \textit{secret} (\( x \notin L \)), then the assignments may have been resolved to different values, i.e., \( i_{1} = (x := v_{1}) \neq (x := v_{2}) = i_{2} \) from rule [\texttt{Asgn}_{\text{HVT}})] and the variable maps remain \( L \)-equivalent, i.e., \( \rho_{1}[x \mapsto v_{1}] \approx_{L} \rho_{2}[x \mapsto v_{2}] \). If the variable is \textit{public} (\( x \in L \)), but typed \textit{transient} according to our transient-flow type system (i.e., \( \Gamma(x) = T \)), we show that the assignments necessarily must have been resolved to the same value. Intuitively, public variables may assume different, secret values (e.g., \( v_{1} \neq v_{2} \)) only \textit{transiently} via a data-flow dependency to a \textit{previous} command that reads a public array \textit{off-bounds}. Therefore, the array read command must have been fetched first via rule [\texttt{Fetch-Array-Load}] and then secret data must have been transiently loaded via rule [\texttt{Exec-Load}].

Since rule [Fetch-Array-Load] automatically prepends bounds-checking instructions to the load, its guard instruction must necessarily have been fetched, executed, and retired before the current assignment can reach the beginning of the reorder buffer and be retired. From this, we deduce that the bounds-check guard must have succeeded, the subsequent load instruction has (correctly) read public data from memory, and therefore ([MEMORY] in Fig. 19c) the current values are equal, i.e., \( \nu_1 = \nu_2 \) and the variable maps remain \( L \)-equivalent, i.e., \( \rho_1[x \mapsto \nu_1] \approx_L \rho_2[x \mapsto \nu_1] \).

In case [Retire-Store], two related store instructions are retired, i.e., \( \Gamma \vdash \text{store}(n, \nu) \approx_L \text{store}(n', \nu') \).

If the label that decorates the instructions is public, (\( \ell = L \)), then the same public memory address is updated with the same values, i.e., \( n = n' \) and \( \nu = \nu' \) from rule [STOREL], and the memory stores remain related, i.e., \( \mu_1[n \mapsto \nu] \approx_L \mu_2[n \mapsto \nu] \) from [MEMORY]. Otherwise (\( \ell = H \)), the instruction update the same secret location with possibly different values, i.e., \( \Gamma \vdash \text{store}(n, \nu) \approx_L \text{store}(n, \nu_2) \) from rule [STOREH], and the stores remain related, \( \mu_1[n \mapsto \nu] \approx_L \mu_2[n \mapsto \nu_2] \).

\[ \square \]

**Lemma 12 (L-equivalence Preservation (Execute Stage)).** Let \( C_1 = (i s_1, i_1, i s'_1, e s_1) \) and \( C_2 = (i s_2, i_2, i s'_2, e s_2) \), such that \( \Gamma \vdash_{\text{SCT}} C_1 \) and \( \Gamma \vdash_{\text{SCT}} C_2 \). If \( \Gamma \vdash C_1 \approx_L C_2 \), then \( \mu_1 \approx_L \mu_2 \), and \( \rho_1 \approx_L \rho_2 \), then \( \alpha_1 = \alpha_2 \) and \( \Gamma' \approx_L C_2' \).\(^{24}\)

**Proof.** By case analysis on the \( L \)-equivalence judgment \( \Gamma \vdash i_1 \approx_L i_2 \) and the reduction steps. Since the judgment relates only instructions of the same kind, the two configurations perform the same reduction step.

\( \triangleright \) [Asgn\(_{LAS} \)] and [Exec-Asgn]. Let the \( L \)-equivalent assignment instructions executed be \( \Gamma \vdash x := e_1 \approx_L x := e_2 \), where \( x \) is public (\( x \in L \)) and stable (\( \Gamma(x) = S \)). From \( L \)-equivalence, we know that the expressions are identical \( e_1 = e_2 \) and public and stable from typing, therefore \( [e_1]^{\rho_1} = [e_2]^{\rho_2} \), by Lemma 9 and the resolved instructions reinserted in the reorder buffers are related, i.e., \( \Gamma \vdash x := [e]^{\rho_1} \approx_L x := [e]^{\rho_2} \) by rule [Asgn\(_{LAS} \)].

\( \triangleright \) [Asgn\(_{HV/T} \)] and [Exec-Asgn]. Similar to the previous case, but variable \( x \) is secret (\( x \notin L \)) or transient (\( \Gamma(x) = T \)). In this case, the expressions are different and may evaluate to different values, but the resolved instructions remain related, i.e., \( \Gamma \vdash x := [e_1]^{\rho_1} \approx_L x := [e_2]^{\rho_2} \) by rule [Asgn\(_{HV/T} \)].

\( \triangleright \) [Guard] and [Exec-Brench-Ok]. Let the \( L \)-equivalent guard instructions be \( \Gamma \vdash \text{guard}(e_{b_1}^{h_1}, c s_1, p_1) \approx_L \text{guard}(e_{b_2}^{h_2}, c s_2, p_2) \). From \( L \)-equivalence, we know that the guard instructions are identical (i.e., same condition expression \( e_1 = e_2 \), predicted outcome \( b_1 = b_2 \), roll-back stack \( c s_1 = c s_2 \), and prediction identifier \( p_1 = p_2 \)) and from typing we know that the guard expression is public, i.e., \( \Gamma \vdash_{\text{CT}} e_{b_1}^{h_1} \) and stable \( \Gamma \vdash e_1 : S \) from rule [Guard] in Figure 16d and Fig. 15a, respectively. Then, the prediction is correct in both reductions, i.e., \( [e_1]^{\rho_1} = b_1 = b_2 = [e_2]^{\rho_2} \) by Lemma 9, and both guard instructions are resolved to \( \Gamma \vdash \text{nop} \approx_L \text{nop} \) and the resulting buffers remain related.

\( \triangleright \) [Guard] and [Exec-Brench-Mispredict]. Similar to the previous case. For the same reasons, both predictions are wrong, i.e., \( [e_1]^{\rho_1} = b' \neq b \neq b' \neq b \), and the rules generate the same observations \( \text{rollback}(p) \), flush the rest of the reorder buffer (the guard instruction and the suffix) and restore the same rollback command stack.

\( \triangleright \) [Load] and [Exec-Load]. Let the \( L \)-equivalent prefixes of the reorder buffers be \( \Gamma \vdash is_1 \approx_L is_2 \). From \( L \)-equivalence, no store is pending in either buffers (one configuration steps if and only if the other steps too), and the same guards are pending in each buffer, i.e., \( \ll [is_1] \rr = \ll [is_2] \rr \) by Lemma 10.

Let the \( L \)-equivalent load instructions executed in the steps be \( \Gamma \vdash x := \text{load}(e_1) \approx_L x := \text{load}(e_2) \). From \( L \)-equivalence, both instructions the address expressions are identical, i.e., \( e_1 = e_2 \), and

\(^{24}\)Also for these configurations, \( L \)-equivalence and typing is defined component-wise.
public and stable (from typing), and therefore evaluate to the same address \([e_1]^{\rho_1} = n = [e_2]^{\rho_2}\) by Lemma 9. As a result, both reductions generate the same observation \(o_1 = \text{read}(n, ps) = o_2\). Lastly, the resolved instructions are \(L\)-equivalent, i.e., \(\Gamma \vdash x := \mu_1(n) \approx_L x := \mu_2(n)\) by rule [\text{ASGN}_{HT}] because the variable \(x\) is \textit{transient} \((\Gamma(x) = T)\) by typing (Fig. 15a).

\[ \text{[Store} \ell] \text{ and [Exec-Store]. Let the two } L\text{-equivalent store instructions be } \text{store}_\ell(e_1, e_1') \approx_L \text{store}_\ell(e_2, e_2'). \text{ First, notice that the expressions that compute the address of the stores are identical in both instructions from rules [Store} \ell]. \text{ Furthermore, we know that these expressions are public and stable from typing, i.e., } \Gamma \vdash e_1 : L \text{ and } \Gamma \vdash e_1 : S \text{ from rule [Store} \ell] \text{ in Fig. 15a and 16d, respectively, and therefore the expression evaluates to the same address by Lemma 9, i.e., } [e_1]^{\rho_1} = n = [e_2]^{\rho_2}. \text{ We then proceed by further case distinction on the security label } \ell \text{ that decorates the store instructions. If } \ell = L, \text{ then } e_1' = e_2' \text{ from [Store}_L], \text{ and thus evaluate to the same value, i.e., } [e_1']^{\rho_1} = v = [e_2']^{\rho_1} \text{ by Lemma 9, and the resolved instructions in the resulting buffers remain related, i.e., } \Gamma \vdash \text{store}_L(n, v) \approx_L \text{store}_L(n, v) \text{ by rule [Store}_L]. \text{ Otherwise } \ell = H, \text{ then } e_1' \neq e_2' \text{ and the resolved instructions remain related because the store writes a secret address, i.e., } \Gamma \vdash \text{store}_H(n, [e_1']^{\rho_1}) \approx_L \text{store}_H(n, [e_2']^{\rho_2}) \text{ by rule [Store}_H].\]

\[ \text{[Protect] and [Exec-Protect]}. \text{ Let the two } L\text{-equivalent protect instructions be } \Gamma \vdash x := \text{protect}(e_1) \approx_L \text{protect}(e_2). \text{ By rule [Exec-Protect], we know that } v_1 = [e_1]^{\rho_1} \text{ and } v_2 = [e_2]^{\rho_2} \text{ and thus the resulting instructions } \Gamma \vdash x := \text{protect}(v_1) \approx_L x := \text{protect}(v_2) \text{ are } L\text{-equivalent by rule [Protect].} \]

\[ \text{[Protect] and [Exec-Protect} 2\text{]. Let the two } L\text{-equivalent protect instructions be } \Gamma \vdash x := \text{protect}(v_1) \approx_L x := \text{protect}(v_2) \text{ and we must show that } \Gamma \vdash x := v_1 \approx_L x := v_2. \text{ If variable } x \text{ is secret, i.e., } x \notin L, \text{ or transient, i.e., } \Gamma(x) = T, \text{ then the two resolved instructions are related by rule [\text{ASGN}_{HT}]. If variable } x \text{ is public, i.e., } x \in L, \text{ and stable, i.e., } \Gamma(x) = S, \text{ then we must show that } v_1 = v_2 \text{ to apply rule [\text{ASGN}_{LAS}]. Intuitively, public variables may assume different, secret values (e.g., } v_1 \neq v_2), \text{ only transiently, due to a data-flow dependency to a previous command that reads a public array off-bounds. Array reads are automatically guarded by bounds check instructions when fetched ([\text{Fetch-Array-Load}]) and by rule [\text{Exec-Protect} 2], we know that no guards are pending in either prefix of the reorder buffer, i.e., } \text{guard}(_-_-) \notin i_1 \text{ and } \text{guard}(_-_-) \notin i_2. \text{ From this, we conclude that all previous bounds-check have been successfully resolved and retired, and therefore the values } v_1 \text{ and } v_2 \text{ are truly public and stable, and thus equal, i.e., } v_1 = v_2, \text{ and the corresponding, resolved assignments are likewise related, i.e., } \Gamma \vdash x := v_1 \approx_L x := v_1. \]

\[ \text{Lemma 13 (Big-step } L\text{-equivalence Preservation). If } \Gamma \vdash_{\text{SCT}} C_1, \Gamma \vdash_{\text{SCT}} C_2, \Gamma \vdash C_1 \approx_L C_2, C_1 \left\lvert D_{\Omega_1} \right. C_1', \text{ and } C_2 \left\lvert D_{\Omega_2} \right. C_2', \text{ then } \Gamma \vdash C_2 \approx_{\text{L}} C_2' \text{ and } O_1 = O_2. \]

\[ \text{Proof. By induction on the speculative big-step reductions (Fig. 13j). Since the reductions follow the same list of directives } D, \text{ either both configurations } C_1 \text{ and } C_2 \text{ reduce, i.e., rule [Step], or have terminated, i.e., rule [Done]. The base case [Done] is trivial; the lists of observation are empty, i.e., } O_1 = \epsilon = O_2 \text{ and } C_1 = C_1', C_2 = C_2', \text{ and thus } \Gamma \vdash C_1 \approx_{\text{L}} C_2 \text{ implies } \Gamma \vdash C_1' \approx_{\text{L}} C_2'. \text{ In the inductive case [Step], both configurations perform a small step, i.e., } C_1 \xrightarrow{d_{\Omega_1}} C_1' \text{ and } C_2 \xrightarrow{d_{\Omega_2}} C_2', \text{ and a big-step, i.e., } C_1'' \left\lvert D_{\Omega_1} \right. C_1', \text{ and } C_2'' \left\lvert D_{\Omega_2} \right. C_2'. \text{ To prove that the observations } o_1 : O_1 \text{ and } o_2 : O_2 \text{ are identical and the final configurations are } L\text{-equivalent, we first apply } L\text{-equivalence preservation for small-steps, i.e., Lemma 11, and deduce that } o_1 = o_2 \text{ and that the intermediate configurations are } L\text{-equivalent, i.e., } \Gamma \vdash C_1'' \approx_{\text{L}} C_2''. \text{ Furthermore, these configurations remain well-typed, i.e., } \Gamma \vdash_{\text{SCT}} C_1'' \text{ and } \Gamma \vdash_{\text{SCT}} C_2'', \text{ by typing preservation, i.e., Lemma 1 and Lemma 2. At this point, we}\]

can apply the induction hypothesis to the big-steps $C''_1 \Downarrow_{D_1} C'_1$ and $C''_2 \Downarrow_{D_2} C'_2$, and conclude that $O_1 = O_2$, and thus $o_1 : O_1 = o_2 : O_2$, and $\Gamma \vdash C'_1 \approx_L C'_2$.

\[ \square \]

**Theorem 4 (Speculative Constant Time).** For all programs $c$ and security policies $L$, if $CT_L(c)$ and $\Gamma \vdash c$, then $SCT_L(c)$.

**Proof.** First, we expand the conclusion $SCT_L(c)$ (Definition 1) and let $C_i = \langle [], [c], \mu_i, \rho_i \rangle$ for $i \in \{1, 2\}$. Then, we assume $\Gamma \vdash C_1 \approx_L C_2$, $C_1 \Downarrow_{D_1} C'_1$, and $C_2 \Downarrow_{D_2} C'_2$ and derive $O_1 = O_2$ and $C'_1 \approx_L C'_2$ from Lemma 13.

\[ \square \]
\begin{align*}
\text{Nop} & \vdash \text{nop} \approx_L \text{nop} \\
\text{Fail} & \vdash \text{fail} \approx_L \text{fail} \\
\text{Guard} & \vdash \text{guard}(e^b, cs, p) \approx_L \text{guard}(e^b, cs, p) \\
\text{Asgn}_{\text{LAS}} & \quad x \in L \quad \Gamma(x) = S \\
& \quad \vdash x := e \approx_L x := e \\
\text{Asgn}_{\text{HT}} & \quad x \notin L \lor \Gamma(x) = T \\
& \quad \vdash x := e_1 \approx_L x := e_2 
\end{align*}

\begin{align*}
\text{Protect} & \quad \vdash x := \text{protect}(e_1) \approx_L x := \text{protect}(e_2) \\
\text{Load} & \quad \vdash x := \text{load}(e) \approx_L x := \text{load}(e) \\
\text{Store}_{\text{H}} & \quad \vdash \text{store}_{\text{H}}(e, e_1) \approx_L \text{store}_{\text{H}}(e, e_2) \\
\text{Store}_{\text{L}} & \quad \vdash \text{store}_{\text{L}}(e_1, e_2) \approx_L \text{store}_{\text{L}}(e_1, e_2) 
\end{align*}

(a) Instructions $\Gamma \vdash i_1 \approx_L i_2$. 

\begin{align*}
\text{RB-Empty} & \quad \Gamma \vdash [ ] \approx_L [ ] \\
\text{RB-Cons} & \quad \Gamma \vdash i_1 \approx_L i_2 \quad \Gamma \vdash i_{s_1} \approx_L i_{s_2} \quad \vdash (i_1 : i_{s_1}) \approx_L (i_2 : i_{s_2}) 
\end{align*}

(b) Reorder buffers $\Gamma \vdash is_1 \approx_L is_2$. 

\begin{align*}
\text{VarMap} & \quad \forall x \in L. \rho_1(x) = \rho_2(x) \\
& \quad \rho_1 \approx_L \rho_2 \\
\text{Memory} & \quad \forall n \in L. \mu_1(n) = \mu_2(n) \\
& \quad \mu_1 \approx_L \mu_2 
\end{align*}

(c) Variable maps ($\rho_1 \approx_L \rho_2$) and memories ($\mu_1 \approx_L \mu_2$). 

\begin{align*}
\text{Conf} & \quad \vdash is_1 \approx_L is_2 \quad cs_1 = cs_2 \quad \rho_1 \approx_L \rho_2 \quad \mu_1 \approx_L \mu_2 \\
& \quad \vdash (is_1, cs_1, \mu_1, \rho_1) \approx_L (is_2, cs_2, \mu_2, \rho_2) 
\end{align*}

(d) Configurations $\Gamma \vdash c_1 \approx_L c_2$. 

\begin{align*}
\text{Transient-VarMap} & \quad \forall x \in L \land \Gamma(x) = S. \rho_1(x) = \rho_2(x) \\
& \quad \vdash \rho_1 \approx_L \rho_2 
\end{align*}

(e) Transient variable map $\Gamma \vdash \rho_1 \approx_L \rho_2$. 

Fig. 19. $L$-equivalence.