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

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

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

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

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

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

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

We connect the static and dynamic semantics by proving that well-typed programs are indeed secure, i.e., satisfy a correctness condition called speculative non-interference [17] which states that the program does not leak under speculative execution more than it would under sequential execution.

3. Automatic Protection. Existing programs that are free of protect statements are likely insecure under speculation and will be rejected by our type system. Thus, our third contribution is an algorithm that automatically synthesizes a minimal number of protect statements to ensure that the program satisfies speculative non-interference. To this end, we extend the type checker to construct a def-use graph that captures the data-flow between program expressions. A cut-set in the graph is a set of variables whose removal eliminates all paths from secret-sources to observable-sinks. We show that inserting a protect statement for each variable in a cut-set suffices to yield a program that is well-typed, and hence, secure with respect to speculation (§ 5.3). Happily, finding such cuts is an instance of the classic max-flow/min-cut problem, so existing polynomial time algorithms let us efficiently
1 void SHA2_update_last(int *input_len, ...)  
2 {  
3     if (! valid(input_len)) { ... }  
4     int len = *input_len;  
5     int *dst3 = ... + len;  
6     _mm_lfence();  
7     int *dst3_safe = protect(. + len);  
8     ...  
9     *dst3_safe = pad;  
10    ...  
11 }

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

synthesize protect statements that resolve the dilemma of enforcing security with minimal performance overhead.

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

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

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

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

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

Preventing the Attack Efficiently. We propose an alternative way to stop speculation from reaching the write in line 9 through a new primitive called protect. Rather than eliminate all speculation, protect only stops speculation along a particular data-path. We use protect to patch the program in line 7. Instead of assigning pointer dst3 directly as in line 5, the expression that computes the address is guarded by a protect statement. This ensures that the value assigned to dst3_safe is always guaranteed to use len’s final, non-speculative value. Therefore, writing to dst3_safe in line 9 prevents any invalid secret-tainted address from speculatively reaching the store, where it could be leaked to the attacker.

The protect primitive offers an abstract interface for fine grained control of speculation. There are a number of possible implementations for this interface. For example,
void SHA2_update_last(int *input_len,...)
{
  if (! valid(input_len)) { ... }
  int len = *(input_len);
  ...
  int len_safe = protect(*input_len)
  for (i = 0; i < len_safe + ...)
  dst2[i] = 0;
  ...
}

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

could be implemented in hardware. While unfortunately, today’s hardware does not offer an equivalent instruction to protect, similar functionalities have been proposed in recent work [2, 32]. Alternatively, protect can be implemented in software (a similar proposal has been made in [36]). In general, protect can be implemented through a fence instruction. However, better solutions exist for reading arrays. For example, Speculative Load Hardening (SLH), a mitigation deployed in the code generated by Clang [10], stalls individual array reads until the corresponding bounds-check condition gets resolved. We model software implementations of protect through a restricted primitive called safe_read, which can only be applied to array reads. We then formalize an implementation of safe_read via SLH in the supplementary material, and evaluate the number of protect and safe_read needed to patch HACL* and their overhead in Section 6.

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

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

2.2 Computing Fixes Via Minimal Fence Inference

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

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

Def-Use Graph. To secure the program, we need to cut the dataflow between the array reads which could introduce transient secret values into the program, and the index in the array read where they are leaked through the cache. For this, we first build a def-use graph whose nodes and directed edges capture the data dependencies between the expressions and variables of a program. For example, consider the def-use graph of program Ex1 in Figure 3. In the graph, the edge \( x \rightarrow x + y \) indicates that \( x \) is used to compute \( x + y \).\(^1\) To track how transient values propagate in the def-use graph, we extend the graph with the special circle node \( T \), which represents the source of transient values of the program. Since reading memory creates transient values, we connect the \( T \) node to all nodes containing expressions that explicitly read memory, e.g., \( T \rightarrow a[i] \).

Following the data dependencies along the edges of the def-use graph, we can see that node \( T \) is transitively connected to node \( z \), which indicates that \( z \) can contain transient data at run-time. To detect insecure uses of transient values, we then extend the graph with the special circle node \( S \), which represents the sink of stable (i.e., non-transient) values of a program. Intuitively, this node draws all the values of a program that must be stable to avoid transient execution attacks. Therefore, we connect all expression used as array indices in the program to the \( S \) node, e.g., \( z \rightarrow S \). The fact that the graph in Figure 3 contains a path from \( T \) to \( S \) indicates that transient data flows through data dependencies into (what should be) a stable index expression and thus the program is insecure.

Cutting the Dataflow. In order to make the program safe, we need to cut the data-flow between \( T \) and \( S \) by introducing as few protect statements as necessary. This problem can be equivalently restated as follows: find a minimal cut-set, i.e., a minimal set of variables, such that removing the variables from the graph eliminates all paths from \( T \) 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 4

\(^1\)To avoid ambiguities in the graph, we assume that each variable is assigned at most once, i.e., the code is in static single assignment form.
To formalize and verify the correctness of the patch computed by BLADE is shown in green. A sub-optimal patch is shown in orange.

2.3 Proving Correctness Via Transient-Flow Types

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

2.4 Attacker Model

Before moving to the details of our semantics and transient type system, we discuss the attacker model considered in this work. The attacker runs cryptographic code on a speculative out-of-order processor and, as usual, can choose the values of public inputs and observe public outputs, but may not read secret data (e.g., cryptographic keys) in registers and memory. Additionally, the attacker can influence how programs are speculatively executed through the branch predictor and choose the instructions execution order in the processor pipeline. The effects of these actions are observable through the cache and are otherwise invisible at the ISA level. In particular, while programs run, the attacker can take precise timing measurements through the data- and instruction-cache with a cache-line granularity, which may disclose secret data covertly. These features allow the attacker to mount Spectre-PHT [20, 21] and Spectre-STL [9] attacks and leak data through FLUSH+RELOAD [43] and PRIME+PROBE [34] cache side-channels attacks. We do not consider speculative attacks that rely on the Return Stack Buffer (e.g., Ret2Spec [25] and [22]) or the Branch Target Buffer (Spectre-BTB [21]). We similarly do not consider attacks that do not use the cache to exfiltrate data, e.g., port contention (SMoTHERSpectre [7]) and Meltdown attacks [9, 24], since hardware fixes address them.

2.3 Proving Correctness Via Transient-Flow Types

To prove correctness, we consider a typing-inference judgment \( \Gamma, \text{Prot} \vdash c \Rightarrow k \), which extends the typing judgment from above with (1) a set of protected variables Prot (the cut-set), and (2) a set of type-constraints \( k \) (the def-use graph). At a high level, type inference has 3 steps: (i) generate a set of constraints under an initial typing environment and protected set that allow any program to type-check, (ii) construct the def-use graph from the constraints and find a cut-set, and (iii) compute the resulting typing environment. To characterize the security of a still unrepaired program after type inference, we define a typing judgment \( \Gamma, \text{Prot} \vdash c \), where unprotected variables are explicitly accounted for in the Prot set. 2

Intuitively, the program is secure if we promise to insert a protect statement for each variable in Prot.

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

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

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

**Directives and Observations.** Instructions do not have to be executed in sequence, they can be executed in any order, enabling out-of-order execution. We use a simple three stage processor pipeline: the execution of each instruction is split into fetch, exec, and retire. We do not fix the order in which instructions, and their individual stages are executed, nor do we supply a model of the branch predictor to decide which control flow path to follow. Instead, we let the attacker supply those decisions through a set of directives [11] shown in Fig. 6. For example, directive $\text{fetch \ true}$ fetches the true branch of a conditional and $\text{exec \ n}$ executes the $n$th instruction in the reorder buffer. Executing an instruction generates an observation (Fig. 6) which records attacker observable behavior. Observations include speculative memory reads and writes (i.e., $\text{load}(n, ps)$ and $\text{store}(n, ps)$ issued while guards $ps$ are pending), rollbacks (i.e., $\text{rollback}(p)$ due to misspeculation of guard $p$), and memory violations ($\text{fail}$). Most instructions generate the empty observation $\epsilon$.

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

3.1 Fetch Stage

The fetch stage flattens the input command into a sequence of instructions which it stores in the reorder buffer. Figure 7 presents selected rules; the remaining rules are in Appendix A. Rule $\text{[Fetch-Seq]}$ pops command $c_1, c_2$ from the commands stack and pushes the two sub-commands for further processing. $\text{[Fetch-Asyn]}$ pops an assignment from the commands stack and appends the corresponding processor instruction.

---

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

\(^3\) Notation \([i_1, \ldots, i_n]\) represents a list of \(n\) elements, \(i_{s+n}++ is\) denotes list concatenation, and \(|is|\) computes the length of the list \(is\).

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

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

**Execute Rule and Auxiliary Relation.** Step rules for the reduction relation are shown in Figure 8. Rule [EXECUTE] executes the \(n\)-th instruction in the reorder buffer, following the directive exec \(n\). For this, the rule splits the reorder buffer into prefix \(is_1\), \(n\)-th instruction \(i\) and suffix \(is_2\). Next, it computes the transient variable map \(\phi(is_1, \rho)\) and executes a transition step under the new map using an auxiliary relation \(\rightsquigarrow\). Notice
that [Execute] does not update the store or the variable map (the transient map is simply discarded). These changes are performed later in the retire stage.

The rules for the auxiliary relation are shown in Fig. 8. The relation transforms a tuple \((is_1, i, is_2, cs)\) 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.

### Guards and Rollback

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

### Loads

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

### 3.3 Retire Stage

The retire stage removes completed instructions from the reorder buffer and propagates their changes to variable map and memory store. While instructions are executed out-of-order, they are retired in-order to preserve the illusion of sequential execution to the user. Figure 9 presents the rules for the retire stage. Rule [Retire-Nop] removes \(\text{nop}\). Rules [Retire-Asgn] and [Retire-Store] remove the resolved assignment \((x := v)\) and instruction \(\text{store}(n, v)\) from the reorder buffer and update the variable map \((\rho[x \mapsto v])\) and the memory store \((\mu[n \mapsto v])\) respectively. Rule [Retire-Fail] aborts the program by emptying reorder buffer and command stack and generates a \(\text{fail}\) observation, simulating a processor raising an exception \(e.g., a\ page\ fault)\).

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

### 3.4 Security Primitives

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

#### Protect

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 11 formalizes this intuition. Rule [Fetch-Protect-Expr] fetches protect commands involving simple expressions \((x := \text{protect}(e))\) and inserts the corresponding protect instruction in the reorder buffer. Rule [Fetch-Protect-Array] piggy-backs on the previous rule by splitting a protect of an array read \((x := \text{protect}(e_1[1]))\) into a separate assignment of the array value \((x := e_1[1])\) and protect of the variable \((x := \text{protect}(e))\). Rules [Exec-Protect1] and [Exec-Protect2] extend auxiliary relation \(\leadsto\). Rule [Exec-Protect1] extends auxiliary relation \(\leadsto\). Rule [Exec-Protect2] evaluates the expression \((v = [e]^\rho)\) and reinserts the instruction in the reorder buffer as if it were a normal assignment. However, the processor leaves the value wrapped inside the protect instruction in the reorder buffer, \(i.e., x := \text{protect}(v), to prevent forwarding the value to the
stalls memory loads until individual bounds-check conditions have been resolved. \texttt{stable\_read}(-) can be implemented using today’s hardware, for example through speculative Load Hardening (SLH) [10], the specret mitigation proposed by and deployed in the Clang compiler. We provide formal semantics in Appendix B.

\textbf{Example.} Consider again Ex1. Instead of using \texttt{protect}(-), we can repair the example by inserting \texttt{stable\_read}. Instead of a single \texttt{protect}(-) for expression \(x + y\), we however need to insert two \texttt{stable\_read} for \(a[i_j]\) and \(a[i_2]\), respectively.

\section{Type System and Inference}

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

\subsection{Type System}

Our transient-flow type system prevents programs from leaking transient values via cache timing channels. To this end, the type system assigns a \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 \texttt{stable} (S) indicating that it cannot contain transient values during execution, or as \texttt{transient} (T) indicating that it can. These types form a 2-point lattice [23], which allows stable expressions to be typed as transient, but not vice versa, i.e., we define a can-flow-to relation \(\subseteq\) 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 judgement \(\Gamma \vdash r:\tau\) assigns a transient-flow type \(\tau\) to \(r\). Figure 12 presents selected rules (see Appendix C for the rest). The shaded part of the rules generates type constraints during type inference and are explained.
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}(\cdot) primitive. Figure 12b shows our typing rules. Rule [Asgn-Prot] disallows assignments from transient to stable variables (as \( T \not\in S \)). Rule [Protect] relaxes this policy as long as the right-hand side is explicitly protected. Intuitively, the result of \text{protect}(\cdot) is stable and it can thus flow securely to variables of any type. Rule [Asgn-Prot] is similar, but instead of requiring an explicit \text{protect}(\cdot) statement, it demands that the variable is accounted for in the protected set Prot. This is secure because all assignments to variables in Prot will eventually be protected through the repair function discussed later in this section. Since primitive \( x := \text{stable_read}(e_1, e_2) \) corresponds to the array read \( e_1[e_2] \), rule [STABLE-READ] requires the array and the index argument to be stable like in rule [ARRAY-READ]. Similar to \text{protect}(\cdot), the result of \text{stable_read}(\cdot) is \text{stable} and thus the type of the variable needs no constraints.

\textbf{Implicit Flows.} To prevent programs from leaking data implicitly through their control flow, rule [If-Then-Else] requires the branch condition to be \text{stable}. This might seem overly restrictive, at first: why can’t we accept a program that branches on transient data, as long as it does not perform any attacker-observable operations (e.g., memory reads and writes) along the branches? Indeed, classic information-flow control (IFC) type systems (e.g., [36]) take this approach by keeping track of an explicit program counter label. Unfortunately, such permissiveness is \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 following program: \( \text{if } tr \text{ then } x := 0 \text{ else skip; } y := a[0] \). The program can leak the value of \( tr \) during speculative execution. To see that, assume that the processor predicts that \( tr \) will evaluate to \( true \). Then, the processor speculatively executes the then-branch \((x:=0)\) and the load instruction \((y:=a[0])\), before resolving the condition. If \( tr \) is \( true \), the memory trace of the program contains a single read observation. However, if \( tr \) is \( false \), the processor detects a misprediction, restarts the execution from the other branch (\text{skip}) and executes the array read, producing a rollback and two read observations. From these observations, an attacker could potentially make inferences about the value of \( tr \). Consequently, if \( tr \) is typed as \( T \), our type system rejects the program as unsafe.

\subsection*{4.2 Type Inference}

We now present our type inference algorithm.

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

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

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

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

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

**Example.** Consider again Ex1 in Figure 3. The graph defined by the constraints $k$, given by $\Gamma \vdash k$, $\text{Prot} \vdash \text{Ex1} \Rightarrow k$ is shown in Figure 4, 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 $x$ as $T$ and $z, i_1$ and $i_2$ in $S$. By Proposition 2 both $\Gamma(k,\{x,y\}), \{x,y\} \vdash \text{Ex1}$ and $\Gamma(k,\{z\}), \{z\} \vdash \text{Ex1}$ hold.

**4.3 Program Repair**

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

**5 Consistency and Security**

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

**Consistency.** We write $C \cup_D C'$ for the complete speculative execution of configuration $C$ to final configuration $C'$, which generates a trace of observations $O$ under list of directives $D$. Similarly, we write $(\mu,\rho) \cup_{O} (\mu',\rho')$ 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 $O|_{\mu}$, which removes prediction identifiers, rollbacks, and mispeculated loads and stores.

**Theorem 5.1 (Consistency).** For all programs $c$, initial memory stores $\mu$, variable maps $\rho$, and directives $D$, such that $(\mu,\rho) \cup_{O} (\mu',\rho')$ and $(\mu,\rho) \cup_{D} (\mu''',\rho''')$, then $\mu' = \mu''$, $\rho' = \rho'''$, and $O \equiv O|_{\mu}$.

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., $O \equiv O|_{\mu}$, because the processor can execute load and store instructions out-of-order.

**Speculative Non-Interference.** Speculative non-interference is parametric in the security policy that specifies which variables and part of the memory are controlled by the attacker [17]. In the following, we write $L$ for the set of public variables and memory locations that are observable by the attacker. Two variable maps are indistinguishable to the attacker, written $\mu_1 \approx_L \mu_2$, if and only if $\mu_1(x) = \mu_2(x)$ for all $x \in L$. Similarly, memory stores are related pointwise, i.e., $\mu_1 \approx_L \mu_2$ iff $\mu_1(n) = \mu_2(n)$ for all $n \in L$.

**Definition 1 (Speculative Non-Interference).** A program $c$ satisfies speculative non-interference if and only if for all directives $D$, memory stores and variable maps such that $\mu_1 \approx_L \mu_2$.
We implemented and evaluated our implementation of BLADE, which automatically eliminates speculative leaks. We now describe our implementation and evaluate the soundness and applicability of our protection chain.

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

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

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

### 6 Implementation and Evaluation

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

#### 6.1 Implementation

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

#### 6.2 Evaluation

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

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

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

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

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

**Number of Fences.** Next, we evaluate how many fences the analysis has to insert to make the programs secure. The results are shown in Table 1. Column **P/B** contains our baseline, which replaces all non-constant array reads, i.e., reads whose address depends on a variable, with statement stable_read (Section 3.4), implementing a SLH-like mitigation that masks the address with the array bounds-check condition. This is the proposed mitigation in the Clang compiler [10]. Column **P** shows the number of protect inserted by BLADE. All benchmarks are modified by the baseline, except for TEA, which is a simple, toy encryption algorithm (that should not be used in practice). In particular, for five of the nine programs, BLADE does not need to insert any fences. Column **P/B** shows the ratio of protect statements to baseline read masks in percent. For most benchmarks, our analysis has to insert under 3% of fences compared to the baseline. For the SHA2 implementation of HACL* this rises to 11.5%. Across all benchmarks, the number of fences is two orders of magnitude lower than the baseline. Since protect statements are an idealized primitive that are not available in today’s hardware,
To evaluate the performance impact of our repair, we compared how a naive placement of fences—applying speculative load hardening to every load of a non-constant address—compares against our approach. We picked the SHA2-512 hash function for this test, and used inputs of size 4KB. Naive fence placement introduced 44 fences while ours introduced only 5. Our measurements showed that while the naive repair algorithm caused 13.9% overhead, the overhead of our minimal fence replacement algorithm was only 0.42%. We used a sample size of 500, and found the relative margin of error of our measurements were less than 0.07%.

7 Related Work

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

We refer to [9] for a comprehensive systematization of transient execution (except Pitchfork [11]) for a fixed speculation bound, and focus on vulnerability detection but do not propose techniques to repair vulnerable programs. In contrast, our type system enforces speculative non-interference even when program instructions are executed out-of-order with unbounded speculation and automatically synthesizes repairs. Given a set of untrusted input source, oo7 Wang et al. [37] statically analyzes a binary to detect vulnerable patterns and inserts fences in turn. Our tool, BLADE, not only repairs vulnerable programs without user annotation, but ensures that program patches contain a minimum number of fences. Furthermore, BLADE formally guarantees that repaired programs are free from speculation-based attacks.

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

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

---

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

<table>
<thead>
<tr>
<th>Name</th>
<th>B</th>
<th>P</th>
<th>S/B</th>
<th>LOC</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRYPTO [29]</td>
<td>92</td>
<td>1</td>
<td>2</td>
<td>1.1</td>
<td>3386</td>
</tr>
<tr>
<td>CORE [29]</td>
<td>47</td>
<td>1</td>
<td>2</td>
<td>2.1</td>
<td>6595</td>
</tr>
<tr>
<td>SHA2 [29]</td>
<td>156</td>
<td>18</td>
<td>34</td>
<td>11.5</td>
<td>7310</td>
</tr>
<tr>
<td>AES [29]</td>
<td>48</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6284</td>
</tr>
<tr>
<td>CURVE [29]</td>
<td>2214</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>59921</td>
</tr>
<tr>
<td>ED25519 [29]</td>
<td>2403</td>
<td>10</td>
<td>0</td>
<td>0.2</td>
<td>60308</td>
</tr>
<tr>
<td>SALSA 20 [38]</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>529</td>
</tr>
<tr>
<td>SHA 256 [38]</td>
<td>23</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>334</td>
</tr>
<tr>
<td>TEA [38]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>112</td>
</tr>
</tbody>
</table>

**Total** | 4990 | 26 | 48  | 0.5 | 144779 |

---

**References**


Automatically Eliminating Speculative Leaks with BLADE


A Full Calculus

\[
\begin{align*}
\text{Fetch-Skip} & \quad (is, skip: cs, \mu, \rho) \xrightarrow{fetch} (is + [\text{nop}], cs, \mu, \rho) \\
\text{Fetch-Asgn} & \quad (is, x := e: cs, \mu, \rho) \xrightarrow{fetch} (is + [x := e], cs, \mu, \rho) \\
\text{Fetch-Seq} & \quad (is, c_1; c_2: cs, \mu, \rho) \xrightarrow{fetch} (is, c_1; c_2: cs, \mu, \rho) \\
\text{Fetch-Ptr-Load} & \quad (is, x := *e: cs, \mu, \rho) \xrightarrow{fetch} (is + [x := \text{load}(e)], cs, \mu, \rho) \\
\text{Fetch-Ptr-Store} & \quad (is, *e_1 := c_2: cs, \mu, \rho) \xrightarrow{fetch} (is + [\text{store}(e_1, e_2)], cs, \mu, \rho) \\
\text{Fetch-Fail} & \quad (is, \text{fail}: cs, \mu, \rho) \xrightarrow{fetch} (is + [\text{fail}], cs, \mu, \rho) \\
\text{Fetch-Array-Load} & \quad (is, c; e_1[e_2][c] := e := e_2 < \text{length}(e_1) \quad \text{fresh}(p) \\
& \quad e' = \text{base}(e_1) + e_2 \quad e' = \text{if } e \text{ then } x := *e' \text{ else fail}) \xrightarrow{fetch} (is, c': cs, \mu, \rho) \\
\text{Fetch-Array-Store} & \quad (is, c; e_1[e_2][c] := e := e_2 < \text{length}(e_1) \quad \text{fresh}(p) \\
& \quad e' = \text{base}(e_1) + e_2 \quad e' = \text{if } e \text{ then } *e' := e \text{ else fail}) \xrightarrow{fetch} (is, c': cs, \mu, \rho) \\
\text{Fetch-If-True} & \quad (is, c; \text{if } e \text{ then } c_1 \text{ else } c_2 \\
& \quad \text{fresh}(p) \quad i = \text{guard}(e, c_2: cs, \mu, \rho) \xrightarrow{fetchtrue} (is + [i], c_1: cs, \mu, \rho) \\
\text{Fetch-If-False} & \quad (is, c; \text{if } e \text{ then } c_1 \text{ else } c_2 \\
& \quad \text{fresh}(p) \quad i = \text{guard}(e, c_1: cs, \mu, \rho) \xrightarrow{fetchfalse} (is + [i], c_2: cs, \mu, \rho) \\
\text{Fetch-While} & \quad (is, \text{while } e \text{ c} \quad c_2 = \text{if } e \text{ then } c_1 \text{ else skip} \xrightarrow{fetch} (is, c_2: cs, \mu, \rho)
\end{align*}
\]

Figure 14. Fetch stage.
\[\phi(\rho, i) = \rho\]
\[\phi(\rho, (x := v)) = \phi(\rho[x \mapsto v], is)\]
\[\phi(\rho, (x := e)) = \phi(\rho[x \mapsto \perp], is)\]
\[\phi(\rho, (x := \text{load}(e)) = \phi(\rho[x \mapsto \perp], is)\]
\[\phi(\rho, (x := \text{protect}(e)) = \phi(\rho[x \mapsto \perp], is)\]
\[\phi(\rho, i) = \phi(\rho, is)\]

(a) Transient Variable Map.

\[\perp \mapsto v\]
\[x \mapsto \rho(x)\]
\[\text{length}(e) \mapsto \text{length}(\rho(e))\]
\[\text{base}(e) \mapsto \text{base}(\rho(e))\]
\[e_1 + e_2 \mapsto e_1 \rho + e_2 \rho\]
\[e_1 \leq e_2 \rho \leq e_2 \rho\]

(b) Evaluation Function.

\[(\mathbb{I}) = \mathbb{I}\]
\[(\text{guard}(e, b, c, p) : i) = p\langle i \rangle\]
\[(i : is) = \langle is \rangle\]

(c) Pending Guard Identifiers.

Figure 16. Retire stage.

\[\text{Retire-Nop} \quad \langle \text{nop} : is, cs, \mu, \rho \rangle \xrightarrow{\text{retire}} \langle cs, \mu, \rho \rangle\]

\[\text{Retire-Asgn} \quad \langle x := v : is, cs, \mu, \rho \rangle \xrightarrow{\text{retire}} \langle is, cs, \mu, \rho[x \mapsto v] \rangle\]

\[\text{Retire-Store} \quad i = \text{store}(n, v) \quad \xrightarrow{\text{retire}} \langle is, cs, \mu, \rho[n \mapsto v] \rangle\]

\[\text{Retire-Fail} \quad \langle \text{fail} : is, cs, \mu, \rho \rangle \xrightarrow{\text{fail}} \langle \perp \rangle\]

Figure 17. Helper functions.
Figure 18. Semantics of $\text{protect}(\cdot)$.
B Semantics of Stable Read

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

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

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

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

\[
\begin{align*}
& r := e_1 \leq \text{length}(e_2) \\
& \quad \text{if } \text{true} \text{ then} \\
& \quad \quad r := r ? \text{true} : 0; \\
& \quad \quad x := (((\text{base}(e_1) + e_2) \& r); \\
& \text{else} \text{ fail}
\end{align*}
\]

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

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

\(^4\)We assume that the first memory cell is reserved to the processor, which initializes it with dummy data, e.g., \( \mu(0) = 0 \).
C Full Type System

Constraints. Our typing judgement $\Gamma, \text{Prot} \vdash s \Rightarrow c$ creates a set of constraints $c$. The syntax for constraints is shown in Figure 21. The constraints relate atoms which either represent the unknown type of a variable $x$ ($a_x$), or the unknown type of an expression ($\rho$). Constraints record can-flow-to relationships between the atoms and lattice values $T$ and $S$. They are accumulated via operator $\cup$, where we identify $c_1 \cup \ldots \cup c_n$ with the set $\{c_1, \ldots, c_n\}$.

Solutions and Satisfiability. We define the solution to a set of constraints as a function $\sigma$ from atoms to flow types, i.e., $\sigma \in \text{Atoms} \mapsto (T, S)$, and extend solutions to map $T$ and $S$ to themselves. For a set of constraints $c$ and a solution function $\sigma$, we write $\sigma \vdash c$ to say that the constraints $c$ are satisfied under solution $\sigma$. The definition of $\sigma \vdash c$ is shown in the lower part of Figure 21. In short, solution $\sigma$ satisfies $c$, if all can-flow-to constraints hold, when the atoms are replaced by their values under $\sigma$. We say that a set of constraints $c$ is satisfiable, if there is a solution $\sigma$ such that $\sigma \vdash c$.

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 $c$, we call a sequence of atoms $a_1, \ldots, a_n$ a path in $c$, if $a_i \in\subseteq a_{i+1} \in c$ for $i = 1, \ldots, n - 1$ and say that $a_i$ is the path’s entry and $a_n$ its exit. A T-S path is a path with entry $T$ and exit $S$. A set of constraints $c$ is satisfiable if and only if there is no T-S path in $c$, as such a path would correspond to a derivation of false. If $c$ is satisfiable, we can compute a solution $\sigma(c)$ by letting $\sigma(c)(a) = T$, 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 expressions. 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 $c$, if $A$ cuts all T-S paths in $c$, and say that $A$ is minimal for $c$, if all other cut-sets $A'$ contain as many or more atoms that $A$, i.e., $\#A \leq \#A'$. The problem of finding a minimal cut-set is an instance of them Min-Cut/Max-Flow problem, and we can reuse existing efficient algorithms [1] to compute a solution.

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

Type Inference. To infer typing environment $\Gamma$ and protected set Prot for a statement $s$, we first define a dummy environment $\Gamma^*$ and protected set Prot*, such that $\Gamma^*, \text{Prot}^* \vdash s \Rightarrow c$ holds for any statement $s$, and use it to extract the set of constraints $c$. For this, we define $\Gamma^*$ as the environment that maps all variables to $S$, and Prot* the set of all variables. We then compute a minimal set of variables $A$ such that $A$ is a cut-set of $c$, extract environment $\Gamma(c, A)$ and use $A$ as protected set. Statement $s$ is then guaranteed to type check under the inferred environment.

Proposition 2 (Type Inference). If $\Gamma^*, \text{Prot}^* \vdash s \Rightarrow c$ and $A$ is a set of variables that cut $c$, then $\Gamma(c, A), A+ s$.

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

Example. Consider again Ex1 in Figure 3. The graph defined by the constraints $c$, given by $\Gamma^*, \text{Prot}^* \vdash \text{Ex}1$ is shown in Figure 4, 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. Finally, environment $\Gamma(c, \{x, y\})$ types all variables as $S$ and $\Gamma(c, \{z\})$ types $x$ and $y$ as $T$ and $z$ as $S$, and by Proposition 2 both $\Gamma(c, \{x, y\}), \{x, y\} + \text{Ex}1$ and $\Gamma(c, \{z\}), \{z\} + \text{Ex}1$ hold.

Example. Next, consider the following example Ex3.

\[
x := a[i]; b[y] := x; \text{if } 0 \leq x \text{ then } z := y \text{ else skip}
\]

We show the corresponding graph in Figure 22. As before, the constraints are unsatisfiable due to the path from $T$ to $S$. The set $\{x\}$ is a minimal cut-set producing environment $\Gamma(c, \{x\})$ which types all variables as $S$. Finally, the typing judgement $\Gamma, \{x\} + \text{Ex}3$ holds, indicating that the program is secure, given the promise that $x$ will be protected.

C.1 Examples for Repair

Example. Consider again Ex1 in Figure 3 from Section 2. The cut-set shown on the right in Figure 4 produces the repair shown in the comments of Figure 3.

Example. Consider again Ex2 and its dataflow graph shown in Figure 22. The cut-set $\{x\}$ produces the repaired program below.

\[
x := \text{protect}(a[i]); b[y] := x; \text{if } 0 \leq x \text{ then } z := y \text{ else skip}
\]

C.2 Type Inference

Our type-inference approach is based on type-constraints satisfaction. 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, unknown types of (sub-)expressions and type variables that can be instantiated with any type that satisfies the constraints. Solving these constrains requires finding a substitution, i.e., a mapping from atoms to concrete transient-flow type, such that all constraints are satisfied if we instantiate the atoms with their type.

Type inference consists of 3 steps: (i) generate a set of constraints under an initial typing environment and protected set that under-approximates the solution of the constraints,
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.

**Constraint Generation.** We describe the generation of constraints through the typing judgment from Figure 12. Given a typing environment $\Gamma$, a protected set Prot, the judgment $\Gamma, \text{Prot} \vdash r \Rightarrow k$ type checks $r$ and generates type constraints $k$. The syntax for constraints is shown in Figure 13. 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., $\alpha_x$ for $x$) and unknown types for expressions (e.g., $r$). In rule [Var], constraint $x \subseteq \alpha_x$ indicates that the type variable of $x$ should be at least as transitive as the unknown type of $x$. This ensures that, if variable $x$ is transient, then $\alpha_x$ can only be instantiated with type $T$. Rule [Bop] generates constraints $e_1 \subseteq e_1 \oplus e_2$ and $e_2 \subseteq e_1 \oplus e_2$ to reflect the fact that the unknown type of $e_1 \oplus e_2$ should be at least as transitive as the (unknown) type of $e_1$ and $e_2$. Notice that these constraints correspond exactly to the premises $\tau_1 \subseteq \tau$ and $\tau_2 \subseteq \tau$ of the same rule. Similarly, rule [Array-Read] generates constraints $e_1 \subseteq S$ and $e_2 \subseteq S$ for the unknown type of the array and the index respectively. In addition to these, the rule generates also the constraint $\Gamma \vdash e_1 \subseteq e_2$, which forces the type of $e_1[e_2]$ to be transient. Rule [Asgn] and [Asgn-Prot] generate the same constraint $r \subseteq x$ because we ignore the protected set during constraint generation, as explained in the footnote.

---

Figure 20. Transient flow type system and type constraints generation.
we compute a substitution that solves the constraints
we first generate a set of constraints
Appendix C for a formal account of the mathematical con-
struction.

\[
\begin{align*}
\text{Atom} & \quad a & := & \alpha_x | r \\
\text{Constraint} & \quad k & := & a \in S \mid T \subseteq a \quad a \in \{ k \cup k \} \mid \emptyset \\
\text{Solution} & \quad \sigma & \in & \text{Atoms} \rightarrow \{ S, T \} \\
& \sigma(S) = S & \quad & \sigma(T) = T \\
& \sigma(a_1) \subseteq a_2 \\
& \sigma + a_1 \subseteq a_2 \\
\ \sigma + a_1 \subseteq a_2 \\
& \sigma + \{ c_1, \ldots, c_n \} \\
\end{align*}
\]

**Figure 21.** Type Constraints and Satisfiability.

In contrast, rule [PROTECT] does not generate the constraint 
\( r \subseteq x \) because \( r \) is explicitly protected. In the other rules, the 
constraints are generated following a similar scheme.

From the set of constraints, we can construct the use-def 
graph of the program as outlined in Section 2.3. We refer 
Appendix C for a formal account of the mathematical con-
struction.

**Type Inference.** To perform type inference on a program \( c \), we first generate a set of constraints \( k \) using the judgment described above, with appropriate initial values for the typing 
environment and the protected set. Specifically, we start with 
an environment that types all variables as stable, i.e., \( \Gamma^* = \lambda x. S \) and include all variables in the cut-set, i.e., \( Prot^* = \text{Vars}(c) \) and generate a set of constraints \( k \) for \( c \), i.e., \( \Gamma^*, Prot^* \vdash s \Rightarrow k \).

From the constraints \( k \), we construct the def-graph and 
compute a cut-set \( Prot \), e.g., by applying the Min-Cut/Max-Flow 
algorithm. Then, from the cut-set \( Prot \) and the program \( c \), 
we compute a substitution that solves the constraints \( k \), as 
follows. We remove from the graph all nodes in the cut-set 
\( Prot \) (and their corresponding edges), and type all variables 
reachable from node \( T \) as \( \text{transient} \), and all other variables as 
\( \text{stable} \). We update the initial typing environment with these 
type assignments and obtain the resulting environment \( \Gamma \). Un-
der new environment \( \Gamma \) and protected set \( Prot \), the unrepair 
program type checks, i.e., \( \Gamma, Prot \vdash c \).

**D Proofs**

**D.1 Security**

In the following, we write \( \Gamma \vdash C \) to indicate that the program 
being executed on the processor is well-typed according to the 
transient-flow type-system.

**Non-Speculative Projection of Observations.** Function \( Q \) 
computes the non-speculative projection of observations \( O \). 
To do that, it applies function \( C(o, ps) \) pointwise. Function 
\( C(o, ps) \) takes as input a single observation \( o \) and \( ps \), a set of 
identifiers of mispredicted guards. The function then removes 
prediction identifiers from observations correctly speculated 
and replaces mispredicted load, store and rollbacks with the 
empty observation \( e \). Function \( R(O) \) collects the identifiers of 
rollbacked guards from events \( \text{rollback}(p) \).

\[
R(O) = \{ p \mid \text{rollback}(p) \in O \} \\
C(\text{load}(n, ps_1), ps_2) = \\
\begin{cases}
| ps_1 \cap ps_2 \equiv \emptyset \Rightarrow \text{load}(n) \\
\text{otherwise} = e
\end{cases}
C(\text{store}(n, ps_1), ps_2) = \\
\begin{cases}
| ps_1 \cap ps_2 \equiv \emptyset \Rightarrow \text{store}(n) \\
\text{otherwise} = e
\end{cases}
C(\text{rollback}(p)), ps) = e
C(o\_o) = o
Q = \{ C(o, R(O)) \mid o \in O \}
\]

**Definition 2 (L-equivalence).** 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, if and only if \( is_1 = is_2, \) 
\( cs_1 = cs_2, \mu_1 = \mu_2, \) and \( \rho_1 \approx L \rho_2. \)

**Lemma D.1 (L-equivalence 1-step preservation).** Let \( ps \) be 
the set of guard identifiers rollbacked in the rest of the execution 
of well-typed configurations \( \Gamma^* \vdash C_1 \) and \( \Gamma^* \vdash C_2. \) If \( C_1 \approx L \) and 
\( C_2 \approx L \) if C(01, ps) = C(02, ps), then \( 01 \approx L 02. \)

**Proof.** By case analysis on the two small-step reductions. Since \( C_1 \approx L C_2 \), 
then their reorder buffer and commands stack are 
equal, i.e., \( is_1 = is_2 \) and \( cs_1 = cs_2. \) Thus, the two configurations 
execute the process the same instruction in the same stage. 
All the instructions that generate the empty observation \( e \) or 
fail are trivial. This include all the rules in the fetch and in the 
\text{retire} stage. The interesting cases that can leak occur speculatively 
and out-of-order, i.e., the \( \rightarrow \) stage. By inspecting rule [EXECUTE], 
we notice both configurations execute the same n-th 
instruction from the attacker supplied directive and with \( L \)-
equivalent transient variable maps, \( \phi(is, \rho_1) \approx L \phi(is, \rho_2) \) from 
\( r_1 \approx L r_1. \) Then, we consider the instructions that can leak during 
the execute stage: guards \( guard(e, cs, ps) \), loads \( x := \text{load}(e) \), 
and stores \( \text{store}(n, e) \). The guard instruction can result in a 
rollback (rule [EXEC-BRANCH-MISPREDICT]) or resolved suc-
sessfully (rule [EXEC-BRANCH-Ok]) if the two execution differ 
the attacker gains information by observing a \text{rollback}(p) or 
not (e.g., through the data cache). We show that this cannot 
happen because the guard expression is typed \( S. \)
We need to prove \( \llbracket e \rrbracket^{p_1} = \llbracket e \rrbracket^{p_2} \), with \( p_1 \approx L p_2 \). If \( e \) contains a secret variable, then \( \llbracket e \rrbracket^{p_1} \neq \llbracket e \rrbracket^{p_2} \), however the secret value would be leaked during sequential execution as well, i.e., it contradicts the hypothesis \( C(o_1, ps) = C(o_2, ps) \). If \( e \) contains only public variables, the outcome of the two conditions may still differ. In particular transient secrets may taint public variables and from there transmitted to the condition through the transient function map. However, by the rules of our type system \( \Gamma \vdash e : S \), which means that there must be a \text{protect}(-) in between the transient source and the stable sink. Since \text{protect}(-) forbids values forwarding, the value of the condition is undefined \( e(p_1) = e(p_2) = \text{bot} \) and this case is void.

The reasoning for rules \text{[Exec-Load]} and \text{[Exec-Store-Value]} is similar.

\[ \square \]

**Theorem D.2 (Soundness).** For all programs \( c, \text{if}\Gamma \vdash c \) then \( c \) satisfies speculative non-interference.

**Proof.** Let \( \mu_1 \) and \( \mu_2 \) be memories such that \( \mu_1 \approx L \mu_2 \) and similarly \( p_1 \approx L p_2 \). Let \( C_i = \{ L, [ ] \}_{\mu_i, p_i} \) for \( i \in \{ 1, 2 \} \) and let \( D \) be a valid schedule such that \( C_1 \llbracket D \rrbracket C_1' \) and \( C_2 \llbracket D \rrbracket C_2' \). We now assume \( O_{1L} = O_{2L} \) and show that \( O_1 = O_2 \) by induction on the typing judgment. The base case (\( \{ \text{Done} \} \)) is trivial. In the inductive case, we two pairs of small and big steps: reductions. A pair of small-step reductions \( \langle is_i, cs_i, \mu_i, p_i \rangle \overset{d}{\rightarrow} \langle is_i', cs_i', \mu_i', p_i \rangle \) and a pair of big-step reductions \( \langle is_i, cs_i, \mu_i, p_i \rangle \llbracket D \rrbracket \langle is_i', cs_i', \mu_i', p_i' \rangle \) for \( i \in \{ 1, 2 \} \). Assuming that the program does not leak sequentially, we have \( O_1 = O_2 \) and \( O_{1L} = O_{2L} \). By induction hypothesis on the big-step we obtain \( O_1 = O_2 \) and derive \( o_1 = o_2 \) by Lemma D.1 applied to small-step reductions and the set of mispredicted guard identifiers \( R_{O_i} \).