















Spectre Declassified: Reading from the Right Place at the Wrong Time


Basavesh Ammanaghatta Shivakumar , Jack Barnes , Gilles Barthe  ,
 Sunjay Cauligi , Chitchanok Chuengsatiansup , Daniel Genkin 
 Sioli O’Connell , Peter Schwabe  , Rui Qi Sim , Yuval Yarom 

 Georgia Institute of Technology, Atlanta, United States

 IMDEA Software Institute, Madrid, Spain

 MPI-SP, Bochum, Germany

 Radboud University, Nijmegen, The Netherlands

 The University of Adelaide, Adelaide, Australia

Abstract—Practical information-flow programming languages commonly allow *controlled leakage* via a *declassify* construct—programmers can use this construct to declare intentional leakage. For instance, cryptographic signatures and ciphertexts, which are computed from private keys, are viewed as secret by information-flow analyses. Cryptographic libraries can use declassify to make this data public, as it is no longer sensitive.

In this paper, we study the interaction between speculative execution and declassification. We show that speculative execution leads to unintended leakage from declassification sites. Concretely, we present a PoC that recovers keys from AES implementations. Our PoC is an instance of a Spectre attack, and remains effective even when programs are compiled with *speculative load hardening* (SLH), a widespread compiler-based countermeasure against Spectre. We develop formal countermeasures against these attacks, including a significant improvement to SLH we term *selective speculative load hardening* (selSLH). These countermeasures soundly enforce *relative non-interference* (RNI): Informally, the speculative leakage of a protected program is limited to the existing sequential leakage of the original program. We implement our simplest countermeasure in the FaCT language and compiler—which is designed specifically for high-assurance cryptography—and we see performance overheads of at most 10%. Finally, although we do not directly implement selSLH, our preliminary evaluation suggests a significant reduction in performance cost for cryptographic functions as compared to traditional SLH.

I. INTRODUCTION

Cryptographic software can be vulnerable to devastating side-channel attacks, allowing malicious parties to recover cryptographic keys from observing the timing behavior of programs. A common means to limit such attacks is to follow the cryptographic constant-time policy, which states that programs do not leak confidential data through an ideal model of cache-based timing side-channels. However, writing efficient constant-time cryptographic software is notoriously hard [25]. The challenges of writing constant-time cryptographic software are partially alleviated by dedicated verification or mitigation frameworks. One example of such a framework is FaCT [14], a security enhancing compiler that transforms typable programs into constant-time programs. The FaCT compiler features a constrained information-flow

```

1 public uint8 otp_and_decode(
2     secret uint8 m,
3     secret uint8 otp) {
4
5     secret mut uint8 c = m;
6     for (uint8 i from 0 to 8) {
7         c ^= (otp & (1 << i));
8     }
9
10    public uint8 d = declassify(c);
11    return decode[d];
12 }

```

Listing 1: One-time pad into table-based decoder. Skipping the for loop (due to misspeculation) directly leaks the secret m .

type system with formal guarantees, but these guarantees have a limited scope: First, they only hold for programs *without declassification*. In practice, however, cryptographic software must be able to release information that is technically typed secret, such as encrypted ciphertexts or verification checks, in a secure way. Second, FaCT only considers a *sequential model of execution*. Unfortunately, modern platforms implement aggressive optimizations such as *speculative execution*—the root of the recent and devastating Spectre attacks [27]. Prior work studies declassification and speculative execution in isolation; see [40] for a survey on declassification and [16] for an overview on Spectre countermeasures. However, we are not aware of any work that studies the interactions between declassification and speculative execution.

Unfortunately, we show that the interaction between declassification and speculative execution can cause unintentional leaks. This is demonstrated by the FaCT program in Listing 1. In this program, a secret message m is encrypted by repeatedly performing a bitwise one-time pad; the resulting ciphertext c is fed into a table-based decoder. Since the ciphertext depends on the secret message, it is also typed secret. For the FaCT compiler to accept this program, c must be declassified before it is fed to the decoder, as array indices can leak to an attacker

(e.g., via the cache). Assuming one-time pads are uniformly distributed, it is easy to see that the program does not leak: Indeed, the ciphertext is uniformly distributed and independent from secrets, and thus the leakage is also independent of secrets.

On the other hand, if the program in Listing 1 were to somehow bypass the loop, it would trivially leak m via c . Such a scenario is clearly impossible during sequential execution; however, under adversarial speculative execution, an attacker with control over branch predictions can cause the loop condition to mispredict, skipping the loop body entirely. In fact, this attack is notably different from prior Spectre attacks: Existing Spectre attacks use speculation to bypass safety checks, allowing attackers to leak data from *unintended locations*, i.e., out-of-bounds memory or type-confused structures. Conversely, the attacker described here leaks values from *intended locations*—the value c in Listing 1 is explicitly declassified, after all—but *before these locations are fully secure*. In other words, the leakage happens due to reading from the **right place** (within bounds) but at the **wrong time** (before it should be public).

FaCT’s formal security guarantees cannot capture or prevent these speculative attacks. To remedy this gap, we ideally want the speculative behavior of cryptographic software to reflect the existing (sequential) security guarantees that frameworks like FaCT provide. One such method of “hardening” programs against adversarial speculation is *speculative load hardening* (SLH) [13]: Informally, SLH protects programs by masking the values of speculatively executed loads, ensuring memory safety even during speculative execution. Unfortunately, applying SLH to Listing 1 does not offer any protection, as the attack does not depend on unsafe loads from misspeculated addresses.

We define the undesirable interaction between declassification and speculative execution in terms of a formal threat model (§III), and we define a simple language, semantics, and type system (§IV) to analyze this new vulnerability. To protect vulnerable programs, we formalize security in terms of a property called *relative non-interference* (RNI) and we develop an “ideal” speculative semantics for which well-typed programs are RNI (§V). We realize this idealized semantics by developing two program transformations: *Selective speculative load hardening* (selSLH), an optimization over SLH, which only masks values speculatively loaded into publicly-typed variables; and *masked declassification*, which masks values declassified during speculation. These countermeasures simulate the idealized semantics during adversarially controlled speculative execution of a program—well-typed programs transformed with these countermeasures thus satisfy RNI. We also briefly consider other countermeasures which combine (selective) SLH and add speculation fences before declassification.

We evaluate the simplest form of our countermeasure—SLH with fenced declassification—by modifying the FaCT compiler (§VI). We find that the overhead of our countermeasure on the FaCT cryptographic benchmarks is under 10%. In addition, we heuristically evaluate the performance savings of selSLH over SLH by comparing the number of public- and secret-typed loads in different cryptographic routines. Our results show that selSLH can reduce the number of masked loads in

these programs by 80% or more.

Finally, we demonstrate the practical importance of RNI and our theoretical model (§VII): We develop a proof-of-concept (PoC) attack against a FaCT implementation of AES. Even when this implementation is compiled with SLH, our PoC can recover the cryptographic keys. We further demonstrate PoC attacks against two implementations of AES from OpenSSL.

We have made all software associated with this paper, including the PoC attack against AES code, the FaCT benchmarks, and the heuristic analysis of selSLH, available online at <https://github.com/0xADE1A1DE/Spectre-Declassified>.

II. BACKGROUND

A. Microarchitectural channels

Modern processor microarchitectures aim to improve the performance of software, usually by predicting the future behavior of programs. For example, data caches in the processor store data that a program has recently accessed, allowing them to exploit temporal and spatial locality in software. Similarly, a processor’s branch predictors monitor conditional jumps that the software executes, aiming to predict whether or not future jumps will be taken and what the destination address will be. Although the state of the microarchitecture does not affect the computation results, it does affect the program performance. Consequently, a program that monitors its own performance, e.g., by measuring the time it takes to perform certain operations, can deduce the state of the microarchitecture. Moreover, because the state of the microarchitecture is determined by program execution, monitoring the microarchitecture will leak information about past execution. *Microarchitectural side-channel attacks* exploit these information leaks: An attacker can monitor the microarchitectural components it shares with a victim program to determine the victim’s behavior [21].

Microarchitectural attacks have exploited a variety of microarchitectural components, including data caches [29, 31, 47, 49], branch predictors [1, 20], translation lookaside buffers [22], and return address stacks [17]. These attacks have devastating consequences for the security of software—including breaking cryptography [1, 31, 49], performing keystroke monitoring [23], reversing machine-learning models [48], reconstructing databases [41], and fingerprinting websites [42, 43].

Of particular relevance to cryptographic code are *cache attacks*, which can determine which memory addresses a victim program has accessed. For example, in a FLUSH+RELOAD [49] attack, the attacker evicts a monitored memory location from the cache before the victim program executes. After the victim finishes running, the attacker measures the time to access the previously evicted memory location. If the victim accessed the monitored location, then the location will have been cached, and the access time will be short. Otherwise, the contents of the memory location will have to come from the the memory, and the access time will be long.

B. Spectre attacks

Modern processors use *speculative execution* as a means to improve performance. Under speculative execution, the

processor fetches and executes instructions before knowing if these computations are required rather than waiting for the results of preceding computations. A simple case of speculative execution is for branching statements: Instead of waiting for the result of a branch condition, the processor may use a *branch predictor* to guess which path will be taken and then execute the predicted branch. Later, it will check whether the branch predictor was correct, *rolling back* execution if this was not the case. The rollback mechanism ensures that architectural effects remain correct—e.g., registers and other architectural states are reset to the initial point of misspeculation. However, the *microarchitectural* state, like the cache, is not rolled back. As a consequence, speculative execution can leak data that would be protected under sequential (non-speculative) execution. Attacks that exploit these predictors are known as *Spectre attacks* [27].

```

1  void access(public uint8[] array,
2             public uint64 index) {
3      if (index < len array) {
4          public uint8 res = array[index];
5          leak(res);
6      }
7  }

```

Listing 2: A FaCT program vulnerable to the basic Spectre-PHT attack. The function checks that the index refers to an item within the array before it accesses the array with the index, leaking the result.

Listing 2 shows a program snippet that is vulnerable to a basic Spectre attack. Because this style of attack targets the processor’s *pattern history table*—which is responsible for branch predictions—it is termed a Spectre-PHT attack [12]. The snippet in Listing 2 ensures that all memory accesses are within the bounds of the input array. Under a sequential execution model, this snippet succeeds at its goal: If an out-of-bounds index is provided, the condition on line 3 will be false and the access will not be executed. Under a speculative execution model, however, it is possible for the body of the branch to be executed *before* the branch condition—even if the condition is ultimately revealed to be false. As these values are obtained from incorrect speculation, any execution that depends on them will eventually be rolled back. However, an attacker can encode these values into the cache (or other microarchitectural state) before the rollback, allowing them to use a side-channel such as FLUSH+RELOAD to recover the value. We refer the reader to the literature for a more extensive account of Spectre attacks [12].

III. THREAT MODEL

We distinguish between the *theoretical* threat model, which we use to prove the correctness of our countermeasure, and the *practical* threat model, which instantiates our theoretical threat model and which we use for our case study.

Our theoretical analysis is inspired by standard models for reasoning about side-channel leakage: We assume a co-located attacker whose goal is to coerce the victim program into leaking

sensitive data. The attacker can use the public interface of the victim and can observe the victim’s control flow and memory trace, but cannot otherwise execute code in the victim context and cannot directly read or write the victim’s memory or registers. The attacker can, however, influence and observe the microarchitectural state of the victim program. Following the standard Spectre-PHT threat model [15, 24], we allow the attacker full control over the prediction of conditional branches.

Our model makes two additional conservative assumptions regarding unsafe memory accesses and declassified values: First, we assume that if the victim program makes an out-of-bounds access, the target address can be arbitrarily controlled by the attacker; this allows us to abstract over the memory layout of the victim program. Second, we assume that the attacker immediately observes declassified values. These conservative assumptions lead to a stronger notion of security that is not tied to specific architectural models.

Our case study is carried out in an instantiation of this attack model, inspired by Patrignani and Guarnieri [32]. In this instantiation, the attacker program invokes a victim function, which has access to secret data; however, the attacker is not allowed to directly access these secrets, even during misspeculated execution. This limitation can be enforced, e.g., through instrumentation [30, 45] or hardware mechanisms [26, 44, 46]. For our case study, we do not use such enforcing mechanisms; instead, our attacker simply does not make any such accesses.

IV. SEMANTICS AND TYPING

For the theoretical analysis, we analyze programs in terms of a core imperative while-language. We give our language a semantics that handles speculative execution and a simple type system for security labels.

A. Language syntax

We present the formal syntax of our language in Figure 1. Our language is a while-language with speculation fences and explicit declassification. For simplicity, we assume that memory can only be accessed through fixed-size arrays. Our language also features (constant-time) conditional expressions, which we use for our countermeasures (see Section V-C).

We let $v \in \mathcal{V}$ range over values, $x \in \mathcal{X}$ range over registers, and $a \in \mathcal{A}$ range over arrays. We assume all values are either integers or booleans and we let $|a|$ denote the size of array a . The state of a program during execution is then given by the tuple $\langle c, \rho, \mu, b \rangle$: The program, $c \in \text{Com}$, is the next command (or sequence of commands) to execute. The register map, $\rho : \mathcal{X} \rightarrow \mathcal{V}$, maps register names to values; we write $\llbracket e \rrbracket_\rho$ for evaluating expression e with the register mapping ρ , and we write $\rho[x := v]$ to update register x with value v . The memory, $\mu : \mathcal{A} \times \mathcal{V} \rightarrow \mathcal{V}$, maps addresses—pairs of array names and valid indices—to values; given $i \in [0, |a|)$, we write $\mu[(a, i)]$ to retrieve the value at index i in array a and $\mu[(a, i) := v]$ to update the array a at index i with value v . Finally, the *speculation flag*, b , is a Boolean value; we set b to \top when the program is misspeculating.

$e \in \text{Expr} ::=$	v	value
	x	register
	$\text{op}(e, \dots, e)$	operator
	$e?e : e$	conditional expression
$c \in \text{Com} ::=$	skip	empty, do nothing
	$c; c$	sequence
	$x := e$	assignment
	$x :=_{\text{declassify}} e$	declassification
	$x := a[e]$	load from array a offset e
	$a[e] := e$	store to array a offset e
	$\text{if } t \text{ then } c \text{ else } c$	conditional
	$\text{while } t \text{ do } c$	while loop
	fence	fence

Figure 1: Syntax of programs.

B. Speculative semantics

Formally, we model speculative execution as an instrumented adversarial semantics inspired by [8, 15]. This style of semantics departs from classic semantics by using explicit *observations* to model side-channel leakage and *adversarial directives* to model adversarial control over branch prediction and out-of-bounds accesses.

One-step execution of programs is given by a labeled transition relation between states:

$$\langle c, \rho, \mu, b \rangle \xrightarrow[d]{o} \langle c', \rho', \mu', b' \rangle$$

The directive d and observation o are taken from the following syntaxes:

$$\begin{aligned} d \in \text{Dir} &::= \text{step} \mid \text{force} \mid \text{load } a, i \mid \text{store } a, i \\ o \in \text{Obs} &::= \bullet \mid \text{read } a, v \mid \text{write } a, v \mid \text{branch } b \mid \text{decl } v \end{aligned}$$

Each observation o represents a potential leak of information in the standard constant-time model [7].¹ The read a, v and write a, v observations, respectively, capture information that is leaked via the cache or other memory side-channel attacks [43, 49]. Similarly, the branch b observation captures information an attacker can recover from the control flow of the program, such as through port contention [2] or instruction cache analysis [1, 20]. In addition, we include an observation $\text{decl } v$ that immediately leaks any explicitly declassified values, as we conservatively assume that the attacker can learn any declassified information.

The adversarial directives d allow our modeled attackers to control the speculative behavior of a program during execution. For example, to represent an attacker that causes a conditional branch to misspeculate, we have the attacker supply the directive force, which forces the program down the wrong branch. Similarly, when a program is about to perform an unsafe load or store, we conservatively allow the attacker to control the address that is read from (or written to) with the directive load a, i (resp. store a, i). Otherwise, the attacker

¹For simplicity, we assume that instructions have no data-depending timing.

supplies the directive step, which simply executes the program as per the usual semantics.

We provide our execution rules in Figure 2. Our rules are similar to usual semantics for a simple while-language: Rules [SEQ-SKIP] and [SEQ] allow empty commands and command sequencing, while rule [ASSIGN] evaluates a given expression e using the register file ρ and updates the register x accordingly. We describe in detail our divergence from usual semantics:

Conditional branching. Conditional branches (rule [IF]) evaluate their branch expression t , continuing down the associated branch. The value of the condition is leaked to the attacker via the observation branch $\llbracket t \rrbracket_\rho$. While loops (rule [WH]) proceed similarly, leaking the loop condition on each iteration. We also allow the attacker to force conditional branches and while loops to misspeculate the result of their respective conditions (rules [IF-S] and [WH-S]). When the attacker issues the directive force instead of step, these rules cause execution to proceed down the *incorrect* branch. Accordingly, we update the speculation flag b to \top to signal that we have misspeculated. Because the force directive always forces the incorrect path, we know that $b = \top$ if and only if we have diverged from sequential execution.

Memory operations. All memory operations in our semantics are given as indices into discrete arrays. For *safe* accesses (rules [LD] and [ST]) where the evaluated index $\llbracket e \rrbracket_\rho$ is in-bounds for a , we leak the memory address via observation read $a, \llbracket e \rrbracket_\rho$ (resp. write $a, \llbracket e \rrbracket_\rho$). We assume all programs are memory-safe during sequential execution. However, if a program has misspeculated, an attacker may coerce the program into performing *unsafe* memory operations (rules [LD-U] and [ST-U]). If, during a misspeculated memory operation, the index e is out-of-bounds (that is, $\llbracket e \rrbracket_\rho \notin [0, |a|)$), then we conservatively allow the attacker to specify where the out-of-bounds address leads. In this case, the attacker issues the directive load a', i (resp. store a', i) to pick the address that will be loaded from (or stored to)—this may use a different array entirely, but must be valid (i.e., $i \in [0, |a'|)$).

Speculation barriers. Modern processors include *speculation barrier* instructions that halt execution and wait for all speculation to properly resolve. We model speculation barriers in our language with the fence command (rule [FEN]), which only allows execution to continue if the current execution has never misspeculated (that is, $b = \perp$). Otherwise (if $b = \top$), we let execution become stuck.

Declassification. Declassified assignment (rule [DECL]) is semantically similar to regular assignment. However, we explicitly leak the value of the expression e to the attacker via the observation $\text{decl } \llbracket e \rrbracket_\rho$.

The following lemma summarizes key structural properties of execution:

Lemma 1. *If $\langle c, \rho, \mu, b \rangle \xrightarrow[d]{o} \langle c', \rho', \mu', b' \rangle$ then:*

- If $b = \top$ then $b' = \top$.
- If $b = \perp$ and $b' = \top$ then $d = \text{force}$.
- If $d = \text{load } a, v$ or $\text{store } a, v$ then $b = \top$.

$\frac{\rho' = \rho[x := \llbracket e \rrbracket_\rho]}{\langle x := e, \rho, \mu, b \rangle \xrightarrow[\text{step}]{\bullet} \langle \text{skip}, \rho', \mu, b \rangle} \text{ [ASSIGN]}$
$\frac{\llbracket e \rrbracket_\rho \in [0, a] \quad \rho' = \rho[x := \mu[(a, \llbracket e \rrbracket_\rho)]]}{\langle x := a[e], \rho, \mu, b \rangle \xrightarrow[\text{step}]{\text{read } a, \llbracket e \rrbracket_\rho} \langle \text{skip}, \rho', \mu, b \rangle} \text{ [LD]}$
$\frac{\llbracket e \rrbracket_\rho \notin [0, a] \quad i \in [0, a'] \quad \rho' = \rho[x := \mu[(a', i)]]}{\langle x := a[e], \rho, \mu, \top \rangle \xrightarrow[\text{load } a', i]{\text{read } a, \llbracket e \rrbracket_\rho} \langle \text{skip}, \rho', \mu, \top \rangle} \text{ [LD-U]}$
$\frac{\llbracket e \rrbracket_\rho \in [0, a] \quad \mu' = \mu[(a, \llbracket e \rrbracket_\rho) := \llbracket e' \rrbracket_\rho]}{\langle a[e] := e', \rho, \mu, b \rangle \xrightarrow[\text{step}]{\text{write } a, \llbracket e \rrbracket_\rho} \langle \text{skip}, \rho, \mu', b \rangle} \text{ [ST]}$
$\frac{\llbracket e \rrbracket_\rho \notin [0, a] \quad i \in [0, a'] \quad \mu' = \mu[(a', i) := \llbracket e' \rrbracket_\rho]}{\langle a[e] := e', \rho, \mu, \top \rangle \xrightarrow[\text{store } a', i]{\text{write } a, \llbracket e \rrbracket_\rho} \langle \text{skip}, \rho, \mu', \top \rangle} \text{ [ST-U]}$
$\frac{\langle c_1, \rho, \mu, b \rangle \xrightarrow[d]{o} \langle c'_1, \rho', \mu', b' \rangle}{\langle c_1; c_2, \rho, \mu, b \rangle \xrightarrow[d]{o} \langle c'_1; c_2, \rho', \mu', b' \rangle} \text{ [SEQ]}$
$\frac{\langle c_1, \rho, \mu, b \rangle \xrightarrow[d]{o} \langle \text{skip}, \rho', \mu', b' \rangle}{\langle c_1; c_2, \rho, \mu, b \rangle \xrightarrow[d]{o} \langle c_2, \rho', \mu', b' \rangle} \text{ [SEQ-SKIP]}$
$\frac{}{\langle \text{if } t \text{ then } c_\top \text{ else } c_\perp, \rho, \mu, b \rangle \xrightarrow[\text{step}]{\text{branch } \llbracket t \rrbracket_\rho} \langle c_{\llbracket t \rrbracket_\rho}, \rho, \mu, b \rangle} \text{ [IF]}$
$\frac{}{\langle \text{if } t \text{ then } c_\top \text{ else } c_\perp, \rho, \mu, b \rangle \xrightarrow[\text{force}]{\text{branch } \llbracket t \rrbracket_\rho} \langle c_{\neg \llbracket t \rrbracket_\rho}, \rho, \mu, \top \rangle} \text{ [IF-S]}$
$\frac{c_\perp = \text{skip} \quad c_\top = c; \text{while } t \text{ do } c}{\langle \text{while } t \text{ do } c, \rho, \mu, b \rangle \xrightarrow[\text{step}]{\text{branch } \llbracket t \rrbracket_\rho} \langle c_{\llbracket t \rrbracket_\rho}, \rho, \mu, b \rangle} \text{ [WH]}$
$\frac{c_\perp = \text{skip} \quad c_\top = c; \text{while } t \text{ do } c}{\langle \text{while } t \text{ do } c, \rho, \mu, b \rangle \xrightarrow[\text{force}]{\text{branch } \llbracket t \rrbracket_\rho} \langle c_{\neg \llbracket t \rrbracket_\rho}, \rho, \mu, \top \rangle} \text{ [WH-S]}$
$\frac{}{\langle x := \text{declassify } e, \rho, \mu, b \rangle \xrightarrow[\text{step}]{\text{decl } \llbracket e \rrbracket_\rho} \langle \text{skip}, \rho[x := \llbracket e \rrbracket_\rho], \mu, b \rangle} \text{ [DECL]}$
$\frac{}{\langle \text{fence}, \rho, \mu, \perp \rangle \xrightarrow[\text{step}]{\bullet} \langle \text{skip}, \rho, \mu, \perp \rangle} \text{ [FEN]}$

Figure 2: One-step (adversarial) semantics.

The first item states that there is no way for b to reset to \perp once it has been set; we purposefully do not model speculative rollback, as it is unnecessary when considering all possible execution paths as in our analysis [8]. The second item states that the only way for b to become \top (i.e., for execution to misspeculate) is through the force directive. Thus every execution follows sequential semantics up until the point of misspeculation (if any). The last item states that unsafe accesses can only happen after the program has misspeculated.

Typing	
$\frac{\Gamma(e) \leq \Gamma(x)}{\Gamma \vdash x := e} \text{ [ASSIGN]}$	
$\frac{\Gamma(e) = L \quad \Gamma(a) \leq \Gamma(x)}{\Gamma \vdash x := a[e]} \text{ [LD]}$	
$\frac{\Gamma(e) = L \quad \Gamma(e') \leq \Gamma(a)}{\Gamma \vdash a[e] := e'} \text{ [ST]}$	
$\frac{\Gamma(t) = L \quad \Gamma \vdash c_1 \quad \Gamma \vdash c_2}{\Gamma \vdash \text{if } t \text{ then } c_1 \text{ else } c_2} \text{ [IF]}$	
$\frac{\Gamma(t) = L \quad \Gamma \vdash c}{\Gamma \vdash \text{while } t \text{ do } c} \text{ [WH]}$	
$\frac{\Gamma \vdash c_1 \quad \Gamma \vdash c_2}{\Gamma \vdash c_1; c_2} \text{ [SEQ]}$	
$\frac{\Gamma(x) = L}{\Gamma \vdash x := \text{declassify } e} \text{ [DECL]}$	

Figure 3: Standard constant-time typing with declassification.

Complete executions. We let $\langle c, \rho, \mu, b \rangle \xrightarrow[D]{o} \langle c', \rho', \mu', b' \rangle$ denote the labeled reflexive-transitive closure of single-step execution. Moreover, we write $\langle c, \rho, \mu, \perp \rangle \Downarrow_D^o$ when $\langle c, \rho, \mu, \perp \rangle \xrightarrow[D]{o} \langle \text{skip}, \rho', \mu', b' \rangle$ or $\langle c, \rho, \mu, \perp \rangle \xrightarrow[D]{o} \langle \text{fence}, \rho', \mu', \top \rangle$: The first case corresponds to a complete execution that has terminated and the second to a misspeculated execution that has become stuck. When execution remains entirely sequential, i.e., all directives are step, we write $\langle c, \rho, \mu \rangle \Downarrow^o$ instead of $\langle c, \rho, \mu, \perp \rangle \Downarrow_D^o$.

C. Typing environment and speculation

We assume that every register and array is tagged with a security level. For simplicity, we only consider the lattice of security levels $L \leq H$, where H is secret data and L is public data: Public values can be treated as secret, but not vice versa (unless explicitly declassified). Other choices of security lattices are possible, but are not considered in this paper. Additionally, we do not consider arrays with mixed sensitivity—arrays are either entirely public or entirely secret.

We use Γ to denote the static typing environment; $\Gamma(x)$ and $\Gamma(a)$ represent the security levels of registers x and arrays a respectively. We extend Γ to expressions by defining $\Gamma(e) = \max_{x \in \text{Vars}(e)} \Gamma(x)$, where $\text{Vars}(e)$ is the set of variables contained in e .

We present our typing rules in Figure 3. As usual, we allow public values to be assigned to secret variables, but not vice-versa: Rule [ASSIGN] specifies $\Gamma(e) \leq \Gamma(x)$ for $x := e$ to be well-typed. Rules [LD] and [ST] enforce similar constraints. In addition, memory and control-flow commands use constant-time typing rules [14, 15]: Array indices must be public ($\Gamma(e) = L$ in rules [LD] and [ST]), since memory addresses are leaked to the attacker during execution. Similarly,

control flow can only depend on public branching conditions ($\Gamma(t) = L$ in rules [IF] and [WH]) since branch conditions are also leaked. Finally, explicit declassification (rule [DECL]) allows secret values to be assigned to public variables with no constraints on the type of e .

Under this type system, well-typed programs (without declassification) are secure under standard execution, since any leakage observations can only be of public values. We formalize this claim using the standard notion of low-equivalence:

Definition 1 (Low equivalence). *For a well-typed program $\Gamma \vdash c$, we have $(\rho_1, \mu_1) \sim (\rho_2, \mu_2)$ iff $\rho_1(x) = \rho_2(x)$ for every $x \in \mathcal{X}$ such that $\Gamma(x) = L$ and $\mu_1(a) = \mu_2(a)$ for every $a \in \mathcal{A}$ such that $\Gamma(a) = L$.*

Lemma 2. *Given a well-typed program c , if c is declassify-free and $(\rho_1, \mu_1) \sim (\rho_2, \mu_2)$ and $\langle c, \rho_1, \mu_1 \rangle \Downarrow^{\mathcal{O}_1}$ and $\langle c, \rho_2, \mu_2 \rangle \Downarrow^{\mathcal{O}_2}$ then $\mathcal{O}_1 = \mathcal{O}_2$.*

Lemma 2 states that if two sequential executions of a program have low-equivalent initial states—i.e., that they agree on all public values—then they will have identical observation traces. An attacker thus cannot recover any secret information. However, under *speculative* execution, this no longer holds: Even if a program is well-typed, an attacker can force the program down misspeculated paths to reveal secret information.

The program in [Listing 2](#) demonstrates this exact scenario: Sequentially, the program can only access (and leak) the public values from `array`. However, if the attacker can control the value of `index` and forces the branch on line 3 to misspeculate, they can leak *any* arbitrary value from memory—including secret values from elsewhere in the program.

A more subtle problem is that attackers can cause *unintended declassification*, as seen in [Listing 1](#). Sequentially, the masking result `c` is declassified only after being properly masked. Even though this result is leaked via the access to `table`, the attacker normally only learns the final obfuscated value. However, if the attacker forces the loop condition to misspeculate (and thus skip the loop body entirely), the result `c` that gets declassified and leaked is exactly the original secret input `m`.

V. RELATIVE NON-INTERFERENCE

Broadly speaking, it is a standard practice to model security policies as information-flow policies. These policies can be *direct* or *relative*: While direct policies enforce an explicit notion of security (e.g., “secret-typed data should not leak”), relative policies enforce security in terms of existing behavior (e.g., “regardless of type, speculative execution should *not leak more* information than sequential execution”) [16]. Direct policies offer stronger guarantees than relative policies, but cannot always be achieved.

In the classic sequential setting, most security policies enforce *non-interference*: Informally, non-interference states that an attacker cannot distinguish between two executions of the same program that use different secret inputs (but identical public inputs). However, simple non-interference is insufficient for programs that use declassification, as an attacker can trivially distinguish traces with declassified secret values.

Instead, we frame our security policy as a relative property. Concretely, we evaluate the speculative security of a program relative to its sequential behavior; during speculative execution, a program should not reveal more information to an attacker than it would have sequentially. We formalize this property as *relative non-interference* (RNI).

A. Relative non-interference

We define relative non-interference (RNI) as a relative policy which contrasts the speculative and sequential leakage observations of a program. Our notion is inspired by prior speculative non-interference properties [16, 24] and is a form of robust declassification [50] in the speculative domain.

Definition 2. *A program c is RNI iff for every pair of executions $\langle c, \rho_1, \mu_1, \perp \rangle \Downarrow_D^{\mathcal{O}_1}$ and $\langle c, \rho_2, \mu_2, \perp \rangle \Downarrow_D^{\mathcal{O}_2}$ such that $(\rho_1, \mu_1) \sim (\rho_2, \mu_2)$ we have:*

$$\mathcal{O}_1^* = \mathcal{O}_2^* \implies \mathcal{O}_1 = \mathcal{O}_2$$

where $\langle c, \rho_1, \mu_1, \perp \rangle \Downarrow_{D^*}^{\mathcal{O}_1^*}$ (resp. \mathcal{O}_2^*) and D^* is the longest prefix of D that does not contain the directive force.

Formally, RNI requires that for every sequence of directives D , given any pair of executions of program c from equivalent states, if the resulting traces \mathcal{O}_1 and \mathcal{O}_2 are equal up to the first force directive (i.e., the point of first misspeculation) then the traces must remain equal for the remainder of the execution.

As we see in [Listings 1](#) and [2](#), well-typed programs can still fail to satisfy RNI. However, given a well-typed program c , we can transform it into a program c' that not only satisfies RNI, but remains *sequentially equivalent* to c : Under sequential execution, c' will produce the same output and the same sequence of observations as c . We formalize this transformation in two steps: We first define an idealized semantics and show that well-typed programs satisfy RNI under this idealized semantics. Then, we show that the idealized semantics can be implemented by a program transformation.

B. Idealized semantics

The idealized semantics protects instructions that would speculatively leak secrets—namely, declassify and load instructions. We find, however, that load instructions which operate on secret arrays (i.e., $\Gamma(a) = H$) do not need to be protected: Regardless of adversarial misspeculation, the result of a secret load remains typed secret, and thus cannot be leaked from a well-typed program. As a result, to create the idealized semantics, we only need to modify the original semantics in the following two ways:

- **Public loads:** The target register is updated with a default value when the speculation flag is set to true.
- **Declassify:** The target register is updated with a default value when the speculation flag is set to true.

We formalize the idealized semantics in [Figure 4](#) with a new step relation $\langle c, \rho, \mu, b \rangle \xrightarrow[d]{\circ} \langle c', \rho', \mu', b' \rangle$ and we write complete executions of the idealized semantics as $\langle c, \rho, \mu, b \rangle \Downarrow_D^{\mathcal{O}} \langle c', \rho', \mu', b' \rangle$. As before, we omit b and D when considering a sequential execution of the program.

$\frac{\llbracket e \rrbracket_\rho \in [0, a] \quad \rho' = \rho[x := \mu[(a, \llbracket e \rrbracket_\rho)]]}{\langle x := a[e], \rho, \mu, \perp \rangle \xrightarrow[\text{step}]{\text{read } a, \llbracket e \rrbracket_\rho} \langle \text{skip}, \rho', \mu, \perp \rangle} \text{ [LD]}$
$\frac{\Gamma(a) = H \quad \llbracket e \rrbracket_\rho \notin [0, a] \quad i \in [0, a'] \quad \rho' = \rho[x := \mu[(a', i)]]}{\langle x := a[e], \rho, \mu, \top \rangle \xrightarrow[\text{load } a', i]{\text{read } a, \llbracket e \rrbracket_\rho} \langle \text{skip}, \rho', \mu, \top \rangle} \text{ [LD-U]}$
$\frac{\Gamma(a) = L \quad \rho' = \rho[x := 0]}{\langle x := a[e], \rho, \mu, \top \rangle \xrightarrow[\text{step}]{\text{read } a, \llbracket e \rrbracket_\rho} \langle \text{skip}, \rho', \mu, \top \rangle} \text{ [LD-PROT]}$
$\frac{\rho' = \rho[x := \llbracket e \rrbracket_\rho]}{\langle x := \text{declassify } e, \rho, \mu, \perp \rangle \xrightarrow[\text{step}]{\text{decl } \llbracket e \rrbracket_\rho} \langle \text{skip}, \rho', \mu, \perp \rangle} \text{ [DECL]}$
$\frac{\rho' = \rho[x := 0]}{\langle x := \text{declassify } e, \rho, \mu, \top \rangle \xrightarrow[\text{step}]{\text{decl } 0} \langle \text{skip}, \rho', \mu, \top \rangle} \text{ [DECL-PROT]}$

Figure 4: Selected rules for idealized semantics.

We can now show the sequential equivalence of the original and idealized semantics:

Lemma 3 (Sequential equivalence of semantics). *Well-typed programs have equivalent sequential leakage and functional behavior under both semantics:*

$$\langle c, \rho, \mu \rangle \Downarrow^\circ \langle c', \rho', \mu' \rangle \text{ iff } \langle c, \rho, \mu \rangle \Downarrow \langle c', \rho', \mu' \rangle.$$

The proof of the lemma is by inspection of the semantic rules for single-step execution followed by induction on the length of the complete execution.

Next, we show that well-typed programs are RNI under idealized semantics:

Proposition 1 (RNI with idealized semantics). *If $\vdash c$ then c is RNI under the idealized semantics.*

We prove this proposition via two unwinding lemmas [37]—one each for sequential and speculative execution.

Lemma 4 (Unwinding lemma for sequential execution). *Let $d = \text{step}, \text{force}$. If $\vdash c$, then for every pair of execution steps:*

$$\begin{aligned} \langle c, \rho_1, \mu_1, \perp \rangle &\xrightarrow[d]{o_1} \langle c'_1, \rho'_1, \mu'_1, b'_1 \rangle \\ \langle c, \rho_2, \mu_2, \perp \rangle &\xrightarrow[d]{o_2} \langle c'_2, \rho'_2, \mu'_2, b'_2 \rangle \end{aligned}$$

we have:

$$\begin{aligned} (\rho_1, \mu_1) &\sim (\rho_2, \mu_2) \wedge o_1 = o_2 \\ \implies (\rho'_1, \mu'_1) &\sim (\rho'_2, \mu'_2) \wedge c'_1 = c'_2 \wedge b'_1 = b'_2. \end{aligned}$$

The unwinding lemma for sequential execution considers two single-step executions with the same directive and observation, and shows that the respective register maps and memories remain equivalent. We present the full proof in [Appendix A](#).

Lemma 5 (Unwinding lemma for idealized speculative execution). *If $\vdash c$ then for every pair of execution steps:*

$$\begin{aligned} \langle c, \rho_1, \mu_1, \top \rangle &\xrightarrow[d]{o_1} \langle c'_1, \rho'_1, \mu'_1, \top \rangle \\ \langle c, \rho_2, \mu_2, \top \rangle &\xrightarrow[d]{o_2} \langle c'_2, \rho'_2, \mu'_2, \top \rangle \end{aligned}$$

we have:

$$\rho_1 \sim \rho_2 \implies o_1 = o_2 \wedge \rho'_1 \sim \rho'_2 \wedge c'_1 = c'_2$$

The unwinding lemma for speculative execution considers two executions with the same directive and shows preservation of equivalence for register maps and observations. Again, we present the full proof in [Appendix A](#).

With these two lemmas, we can now prove [Proposition 1](#): *Proof.* W.l.o.g. we can decompose the two executions as:

$$\begin{aligned} \langle c, \rho_1, \mu_1, \top \rangle &\xrightarrow[D']{O_1: o_1} \langle c'_1, \rho'_1, \mu'_1, \perp \rangle \xrightarrow[D'']{O'_1} \langle \text{fence}, \rho'_1, \mu'_1, \perp \rangle \\ \langle c, \rho_2, \mu_2, \top \rangle &\xrightarrow[D']{O_2: o_2} \langle c'_2, \rho'_2, \mu'_2, \top \rangle \xrightarrow[D'']{O'_2} \langle \text{fence}, \rho'_2, \mu'_2, \perp \rangle \end{aligned}$$

where $D' = \text{step}^n :: \text{force}$ and $D = D' :: D''$. Assume that $O_1 = O_2$. By repeated applications of [Lemma 4](#), it follows that the instructions and the memories *before* executing the force step are equivalent. Since force executes on a public branching instruction, the two force steps leak the same observations, i.e., $o_1 = o_2$, and hence by one final application of [Lemma 4](#) we conclude that $c'_1 = c'_2$ and that $(\rho'_1, \mu'_1) \sim (\rho'_2, \mu'_2)$. By repeated applications of [Lemma 5](#), we conclude that $O'_1 = O'_2$, and hence $O_1 :: o_1 :: O'_1 = O_2 :: o_2 :: O'_2$, as desired. \square

C. Program transformation

To implement the idealized semantics, we define a concrete program transformation $\langle c \rangle$. We present the transformation rules in [Figure 5](#). Our transformations make the misspeculation flag b concrete, instrumenting programs to track this value in a (unique) architectural register \tilde{b} . In particular, we update \tilde{b} when entering a branch or a loop body and after exiting a loop. We use \tilde{b} to implement the *selective SLH* and *masked declassify* countermeasures, which respectively protect memory loads and declassification statements.

selSLH, or *selective speculative load hardening*, masks the results of public memory loads against \tilde{b} . Hence if \tilde{b} is \top —i.e., the program has misspeculated—then the result of the load becomes 0. However, *selSLH* explicitly does *not* transform secret loads; this is an improvement over traditional SLH, which masks *all* loads. *selSLH* is particularly relevant for cryptographic programs since such programs mainly operate on secret data: The intuition is that secret loads *already* handle sensitive data, and thus our typing rules already protect any data from these loads from leaking. Interestingly, this means *selSLH* benefits from “over-labeling” inputs as secret (as long as the program remains well-typed), as this allows us to further reduce the number of transformation sites. We present some preliminary results of the potential savings in [Section VI-C](#).

Masked declassification is an even simpler mitigation: At every *declassify* statement, we mask the result against \tilde{b} .

$$\begin{aligned}
\langle (x := e) \rangle &= x := e \\
\langle (a[e] := e') \rangle &= a[e] := e' \\
\langle (x := a[e]) \rangle &= x := a[e]; x := \tilde{b} : 0?x \quad , \quad \Gamma(x) = L \\
\langle (x := a[e]) \rangle &= x := a[e] \quad , \quad \Gamma(x) = H \\
\langle (x :=_{\text{declassify}} e) \rangle &= x := e; x := \tilde{b} : 0?x \\
\langle (\text{if } t \text{ then } c_1 \text{ else } c_2) \rangle &= \text{if } t \text{ then } (\tilde{b} := t?\tilde{b} : \top; \langle c_1 \rangle) \text{ else } (\tilde{b} := t?\top : \tilde{b}; \langle c_2 \rangle) \\
\langle (\text{while } t \text{ do } c) \rangle &= \text{while } t \text{ do } (\tilde{b} := t?\tilde{b} : \top; \langle c \rangle); \tilde{b} := t?\top : \tilde{b} \\
\langle (c_1; c_2) \rangle &= \langle c_1 \rangle; \langle c_2 \rangle
\end{aligned}$$

Figure 5: Selective SLH and masked declassification countermeasures. Transformation of loads is predicated on the type of x .

Just as with selSLH, if the program has misspeculated, the result becomes 0, thus preventing transient values from being improperly declassified.

Finally, we prove that our transformation properly implements the idealized semantics: Given a program c , the ideal execution of c agrees with the execution of the transformed program $\langle c \rangle$. We present the formal definition of agreement as [Lemma 7 in Appendix A](#) and state the correctness here in a simplified form:

Lemma 6 (Implementation of idealized semantics, simplified). *The following are equivalent:*

- $\langle c, \rho, \mu, b \rangle \Downarrow_D^{\mathcal{O}} \langle c', \rho', \mu', b' \rangle$
- $\langle \langle c \rangle, \rho[\tilde{b} := b], \mu, b \rangle \Downarrow_D^{\mathcal{O}} \langle c', \rho'[\tilde{b} := b'], \mu', b' \rangle$

The lemma is proved by induction on the length of execution.

Although we focus on selSLH and masked declassification, in practice developers may be forced to fall back to alternatives. In particular, without compiler support for public/secret type information, we must conservatively mask every array access. Similarly, it is not feasible to implement masked declassification without proper compiler support. In this case, we can instead use *fenced declassification*: Instead of masking the result of declassification, we insert a fence instruction before each declassify statement—fences prevent misspeculated execution from proceeding, so unintended values cannot be declassified. Although less efficient, these alternative countermeasures are still sound: We can easily define corresponding idealized semantics and transformations for each combination of countermeasures; the proofs of these countermeasures proceed in exactly the same way.

VI. IMPLEMENTATION AND EVALUATION

We evaluate the performance of SLH and fenced declassification using FaCT [14], a domain-specific framework for writing efficient constant-time cryptographic routines. FaCT is an ideal target for implementing our countermeasures: The FaCT language already supports information-flow typing and declassification, and the FaCT compiler uses the LLVM compiler infrastructure, which already supports SLH.

A. FaCT implementation

FaCT is a framework for writing efficient constant-time code. The framework consists of two components: The FaCT lan-

Table I: Case study summary: Lines of code in FaCT and uses of declassify (#D).

Case study	LoC	#D
libsodium secretbox	1068	1
curve25519-donna-c64	342	1
OpenSSL record validate	91	1
OpenSSL MEE-CBC	219	1

guage, a domain-specific language supported by an information-flow type system; and the FaCT compiler, which generates efficient constant-time code. In order to ease programming, FaCT explicitly allows secret-dependent control flow. The FaCT compiler uses type-directed transformations to remove any potential timing leaks; the resulting programs are constant-time and well-typed in a system similar to [Figure 3](#).

The FaCT distribution includes ports of code from several well-known cryptographic libraries, including the *secretbox* authenticated encryption suite from libsodium [19]; the *donna-c64* implementation of the Curve25519 elliptic-curve primitive [28]; and SSLv3 and TLS packet verification code from OpenSSL [33]. We provide an indication of the size of each port and the number of declassify statements in [Table I](#).

Implementing SLH and fenced declassification required two modifications to the FaCT compiler:

- We modified code generation to insert a fence instruction before each declassification. Concretely, our implementation inserts the *llvm.x86.sse2.ifence* LLVM intrinsic before changing the security label.
- We upgraded FaCT’s backend to LLVM 11 to make use of LLVM’s `-mspeculative-load-hardening` option.

B. Performance evaluation

Our performance evaluation uses the case studies from FaCT. We made the following modifications to their benchmarks:

- We collect measurements in terms of CPU cycle counts and report the median and quartiles of repeated measurements.. This is a standard practice to eliminate outliers due to system interrupts.
- We declassify the outputs of Curve25519 and *secretbox* encryption. This is not required by the FaCT type system, but it reflects that the outputs of Curve25519 public-key generation and of *secretbox* authenticated encryption are indeed public, and must be safe even if leaked by the caller.

Table II: Benchmarks summary: Lower quartile, median, and upper quartile for each implementation.

Implementation	Cpucycle counts: P_{25} , P_{50} , P_{75}								
	FaCT (plain)			FaCT w/ SLH			FaCT w/ SLH+Fence		
donna	1.96e5	1.96e5	1.96e5	2.13e5	2.13e5	2.13e5	2.13e5	2.13e5	2.13e5
secretbox ref enc	2.03e3	2.03e3	2.03e3	2.20e3	2.20e3	2.20e3	2.23e3	2.23e3	2.23e3
secretbox ref dec	2.93e3	2.93e3	2.93e3	3.12e3	3.12e3	3.12e3	3.14e3	3.14e3	3.15e3
secretbox vec enc	1.93e3	1.93e3	1.94e3	2.04e3	2.04e3	2.04e3	2.07e3	2.07e3	2.07e3
secretbox vec dec	2.83e3	2.83e3	2.83e3	2.97e3	2.97e3	2.97e3	2.98e3	2.98e3	2.98e3
mee 256mb	2.31e9	2.31e9	2.31e9	2.31e9	2.31e9	2.31e9	2.32e9	2.33e9	2.33e9
mee 1gb	9.22e9	9.23e9	9.23e9	9.25e9	9.27e9	9.29e9	9.32e9	9.34e9	9.37e9
mee 4gb	3.68e10	3.69e10	3.69e10	3.70e10	3.70e10	3.71e10	3.71e10	3.72e10	3.73e10
ssl3 256mb	3.09e9	3.10e9	3.10e9	3.09e9	3.10e9	3.10e9	3.11e9	3.12e9	3.12e9
ssl3 1gb	1.23e10	1.23e10	1.24e10	1.24e10	1.24e10	1.24e10	1.24e10	1.24e10	1.25e10
ssl3 4gb	4.95e10	4.95e10	4.96e10	4.95e10	4.96e10	4.96e10	4.98e10	4.99e10	4.99e10

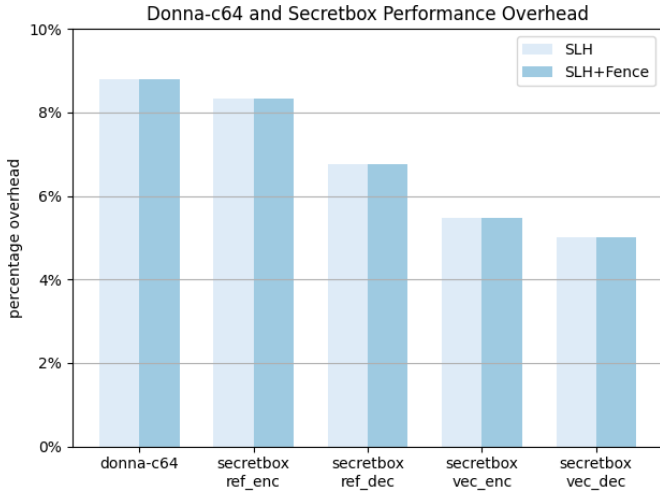


Figure 6: Median of FaCT with SLH and SLH+Fence mitigations over unmodified FaCT in Curve25519 donna-c64 and libsodium’s secretbox. We omit quartiles as the measurement variance was negligible.

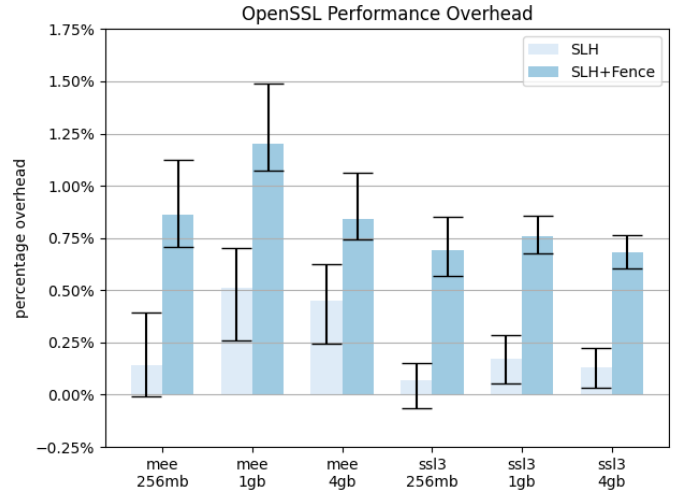


Figure 7: Median overhead of FaCT with SLH and SLH+Fence mitigations over unmodified FaCT in OpenSSL operations, with upper and lower quartiles displayed.

We measured each case study with unmodified FaCT, FaCT with LLVM’s SLH enabled, and FaCT with both SLH and fenced declassification. We used a machine with an Intel i7-9700K CPU and 64GB RAM; this CPU does not feature hyperthreading and we disabled the TurboBoost feature for consistency. We present our results in Figures 6 and 7 and we report the absolute cycle counts in Table II. We find that the overhead introduced by SLH is significant, but the additional overhead introduced by fenced declassification is very small: In cryptographic software, declassification is required, although rare, and usually only upon the final output. The OpenSSL benchmarks in Figure 7 show a smaller overhead compared to the libsodium benchmarks in Figure 6; this is because the OpenSSL benchmarks are much less CPU-bound.

C. Performance of selSLH

Implementing selSLH requires major changes to the LLVM compiler—including implementing a security type system for the LLVM IR—and is out of scope for this paper. Instead, we estimate selSLH’s improvement over standard SLH by classify-

ing memory loads: We analyzed the reference implementation of ChaCha20 [9], as an example of a primitive that has a public variable input length; the donna-c64 implementation of scalar multiplication on Curve25519 [28], as an example of a primitive with all inputs and outputs of fixed length; and the reference implementation of the Ed25519 public-key signature scheme [10], as an example of a higher-level API. We compiled all implementations with gcc-10.2 and optimization flags `-fomit-frame-pointer -march=native` on a machine with an Intel i7-6500U CPU.

For our evaluation, we modified the Pitchfork analysis tool [15], which uses symbolic execution to verify binaries for sequential (and speculative) constant-time. More importantly for us, Pitchfork propagates the security types of values through execution by using the initial types of any inputs and global data. We instrumented Pitchfork to classify and count each load it encounters during sequential execution.

With selSLH, the security type of inputs has a direct impact on performance, as secret values do not need to be protected by SLH mitigations. For our evaluation, we declare *all* inputs

Table III: Counts of public and secret loads of various cryptographic routines. We report the percentage reduction of mitigations that selective SLH would provide as compared to traditional SLH.

Impl.	# public	# secret	# total	SLH saved
ChaCha20 (512 B)	192	766	958	79.96%
ChaCha20 (1024 B)	384	1,518	1,902	79.81%
donna-c64	2,054	43,663	45,717	95.51%
Ed25519 keypair	14,103	348,878	362,981	96.11%
Ed25519 sign	12,200	349,221	361,421	96.62%
Ed25519 verify	1,127	65,409	66,536	98.31%

to be secret unless they *must* be public to be well-typed (i.e., values that get leaked during normal execution). For example, while the fixed basepoint of Curve25519 in donna-c64 is public in principle, declaring it as secret is not unsound, and results in fewer public loads.

We present our results in Table III. Our experiments show that for typical cryptographic code, we can indeed safely and soundly omit the majority of SLH protections. In the ChaCha20 implementation, for example, nearly 80% of the loads are for secret data, and thus need not be protected by SLH; the remaining loads access public pointers and loop counters spilled to the stack (and must remain protected). For the donna-c64 Curve25519 implementation and the Ed25519 signature functions, the savings reach more than 95%—selSLH is able to remove nearly *all* SLH protections. Although these measurements do not translate directly to performance improvements, they serve as a useful indication of how much performance selSLH can recover from the overhead of standard SLH mitigation.

VII. CASE STUDY: AES

Section V demonstrates that interactions between declassification and speculative execution may breach the security guarantee of a program. In this section we demonstrate some of the risks that this may cause in realistic scenarios. We investigate declassification in the common case of AES encryption, where declassification is required to allow transmitting the ciphertext. We demonstrate that due to speculative execution, the ciphertext may be declassified too early leading to disclosure of an improperly encrypted message. We leave a description of how to recover a key from such messages to Appendix B.

More specifically, we look at an implementation of AES written in FaCT that uses the AES-NI instruction set to perform the encryption rounds. Successive rounds are implemented as straight-line code with two branches that exit early after ten or twelve rounds to allow for the different key lengths of AES. We further look at two implementations that are part of OpenSSL: The default implementation, which uses AES-NI, and a machine-independent version that uses T-tables.

A. AES Background

The Advanced Encryption Standard (AES) is a symmetric block-cipher, operating on 128-bit block size using keys of size

128, 192, or 256 bits. AES follows a substitution-permutation network design whose construction consists of multiple rounds to produce a ciphertext.

AES Round Overview. The 128-bit AES state is written as a 4×4 byte matrix and in each round transformed through 4 operations: SubBytes, ShiftRows, MixColumns and AddRoundKey. The final round (10, 12, or 14 depending on the key size) does not perform MixColumns. The SubBytes operation replaces each byte by another byte according to a predefined lookup table. ShiftRows circularly rotates row i to the left by i . MixColumns is a linear transformation of the columns with bytes interpreted as elements of $\text{GF}(2^8)$. AddRoundKey performs an exclusive-or with a round key. Note that SubBytes is the only non-linear transformation.

AES-NI. The Advanced Encryption Standard instruction set (AES-NI) is an extension of the x86-64 instruction set, which implements the steps of the AES encryption. It offers both better performance and enhanced security than software implementations. For AES encryption, AES-NI supports two main instructions: AESENC, which performs a full AES round consisting of SubBytes, ShiftRows, MixColumns, and AddRoundKey, and AESENCLAST, which performs SubBytes, ShiftRows, and AddRoundKey for the last round.

B. PoC Attack Overview

The high-level idea behind the attack is to train the branch prediction unit (BPU) to speculatively exit the AES implementation after performing fewer rounds than required. We flush the key length from the cache then invoke the encryption. The BPU predicts which branches to take in the implementation and by extension predicts how many rounds to apply. Because it takes time for the processor to retrieve the key length from the memory, the processor does not immediately detect the misprediction, allowing the code to return speculatively to the attacker, who then leaks the ciphertext via a cache-based covert channel. At some later time, the processor retrieves the correct number of rounds and squashes all of the mispredicted execution, including any later execution of the attacker. It then restarts execution from the first mispredicted branch, completing the correct number of rounds. However, squashing instructions does not revert any changes made to the cache. This allows an attacker to measure the state of the cache using the FLUSH+RELOAD attack [49] and retrieve the partially encrypted ciphertext.

C. PoC Attack on AES

We begin with a description of our PoC attack on Listing 3. The `aes_round` and `aes_final_round` functions are backed by compiler intrinsics that replace the functions with the AESENC and AESENCLAST instructions. The `key` variable contains the full key expansion along with the number of rounds that needs to be performed.

Step 1: Branch Predictor Training. Our goal in this attack is to train the branch history buffer, which predicts the conditions of conditional branches, to abort the encryption early, allowing the attacker to leak the ciphertext of a reduced-round AES that

```

1  export void unrolled_fact (secret uint64[2] plaintext,
2      public mut uint64[2] ciphertext, mut AES_KEY key) {
3      secret mut uint64<2> state = load_le(plaintext);
4      secret mut uint64<2> rd_key = load_le(view(key.rd_key, 0, 2));
5      public uint32 rounds = uint32(key.rounds);
6      assume(rounds < 15);
7      state = state ^ rd_key;      rd_key = load_le(view(key.rd_key, 2, 2));
8      state = aesenc(state, rd_key); rd_key = load_le(view(key.rd_key, 4, 2));
9      state = aesenc(state, rd_key); rd_key = load_le(view(key.rd_key, 6, 2));
10     state = aesenc(state, rd_key); rd_key = load_le(view(key.rd_key, 8, 2));
11     state = aesenc(state, rd_key); rd_key = load_le(view(key.rd_key, 10, 2));
12     state = aesenc(state, rd_key); rd_key = load_le(view(key.rd_key, 12, 2));
13     state = aesenc(state, rd_key); rd_key = load_le(view(key.rd_key, 14, 2));
14     state = aesenc(state, rd_key); rd_key = load_le(view(key.rd_key, 16, 2));
15     state = aesenc(state, rd_key); rd_key = load_le(view(key.rd_key, 18, 2));
16     state = aesenc(state, rd_key); rd_key = load_le(view(key.rd_key, 20, 2));
17     if (rounds > 10) {
18         state = aesenc(state, rd_key); rd_key = load_le(view(key.rd_key, 22, 2));
19         state = aesenc(state, rd_key); rd_key = load_le(view(key.rd_key, 24, 2));
20         if (rounds > 12) {
21             state = aesenc(state, rd_key); rd_key = load_le(view(key.rd_key, 26, 2));
22             state = aesenc(state, rd_key); rd_key = load_le(view(key.rd_key, 28, 2));
23         }
24     }
25     state = aesenclast(state, rd_key);
26     store_le(ciphertext, declassify(state));
27 }

```

Listing 3: FaCT implementation of unrolled AES encryption. The `aesenc` and `aesenclast` functions are compiler intrinsics that map to the `AESENC` and `AESENCLAST` x86 instructions.

can be cryptanalyzed to recover the key. Listing 4 shows the pseudocode of the attack. It starts by creating two keys, one which trains the BPU and one which emulates a secret key the attacker does not have access to. We repeatedly call `encrypt` using the training key, training the BPU to predict `false` for the condition on line 17 in Listing 3. This causes `encrypt` to exit after applying ten rounds. We note that in practice, an attacker can use other means to train the BPU. In particular, the attacker can rely on aliasing in the BPU, using a branch that the attacker controls to train prediction within `encrypt`.

```

1  function attack() {
2      training_key = create_aes128_key();
3      secret_key = create_aes192_key();
4
5      for (int i = 0; i < 127; i++) {
6          encrypt(plaintext, training_key);
7      }
8
9      flush(secret_key.rounds);
10     ciphertext = encrypt(plaintext, secret_key);
11
12     sidechannel_send(ciphertext);
13
14     return sidechannel_recv();
15 }

```

Listing 4: Pseudocode of our attack on AES. For clarity, we show training and victim execution as separate steps. In practice, our code does both of these steps in the same loop, using constant-time select to switch between inputs.

Step 2: Triggering Misspeculation. After training the branch predictor, we flush the field `key.rounds` from the cache. The field controls the branches on lines 17 and 20 of Listing 3

which determine the number of rounds to apply. Thus, flushing it from the cache delays the evaluation of these branches until the field is retrieved from memory. In the meanwhile, the processor uses the `false` prediction for the condition in line 17, as trained earlier, and proceeds speculatively along the mispredicted execution path. This delay is necessary to allow the speculative execution to perform the final round (Listing 3, line 25) and return to the attacker code, which leaks the reduced-round ciphertext through a microarchitectural side channel (Listing 4, line 12). Eventually, the processor will retrieve `key.rounds` from the memory, perform the comparisons for the conditions, detect that the branch was mispredicted, squash the ensuing speculative execution, and resume execution with the correct `true` condition in line 17. However, by this time, the reduced-round ciphertext has already leaked through the side channel.

Step 3: Recovering the Reduced-Round Ciphertext. Finally, execution once again returns from `encrypt` and control flows to `sidechannel_recv`. This function acts as the receiver of the side channel and receives the reduced-round ciphertext from the transiently executed `sidechannel_send`. We implement `sidechannel_recv` using `FLUSH+RELOAD` [49], a cache side-channel that can determine if a particular address has been accessed. We leak the incorrect ciphertext one byte at a time, by selecting a byte and using it to access a 256-page array. We can then check which of the 256 pages has been accessed to recover one byte of the reduced-round ciphertext from the previously squashed execution. This process is repeated for each of the 16 bytes of the reduced-round ciphertext. We note that multiple side channels have been demonstrated in the

context of transient-execution attacks [4, 11, 35, 36], and the choice of channel is not limited to FLUSH+RELOAD.

Attack Accuracy. We test two victims. The first uses a default LLVM back-end for code generation and the second uses LLVM with SLH enabled. We repeat the attack 1 000 times with each victim, each time recording whether the reduced-round ciphertext is recovered correctly. On average the attack succeeds with a probability of 95% irrespective of the victim.

D. PoC Attack on OpenSSL AES

We further demonstrated leakage from two AES implementations provided in OpenSSL. The first implementation uses T-tables for implementing the round function and the second uses AES-NI. The T-table implementation follows the same general structure of the FaCT implementation in Listing 3, but uses precomputed tables for performing the round function. The T-table implementation is known to be vulnerable to cache attacks [31], but our PoC does not exploit this vulnerability; we use the same strategy (and leakage channel) described previously. The PoC works as expected when SLH is disabled—enabling SLH prevents the leak, since SLH poisons the table accesses executed in the last round.

The second implementation we test is the default OpenSSL implementation for computers that support AES-NI. The implementation, which is written in x86-64 assembly, uses the AESENC instruction in a loop, and then invokes AESENCLAST for the last round. To determine the number of iterations, the implementation uses the value of `key.rounds`. Our PoC trains the loop to stop after one iteration, resulting in a two-round encryption. We only test this implementation without SLH, because LLVM SLH does not apply to assembly code. Appendix B describes how to recover the key from the information we obtain.

E. Attack Practicality

The proof-of-concepts we present in this section serve to show that ignoring declassification can result in leakage from otherwise protected code. Several aspects may make our attacks difficult in practice. Specifically, while intra-process isolation is an active research area [26, 44, 46], some real-world applications seem to be moving in the opposite direction [34]. To perform the attack across process boundaries, the attacker will have to overcome branch predictor flushing in modern processors and to find a leaky gadget that leaks the mispredicted declassified values. The former could be achieved through confused deputy attacks [5] and the latter through automated search of vulnerable gadgets [11]. We leave implementing these to future work.

VIII. RELATED WORK

Robust Declassification. Relative non-interference is closely related to *robust declassification*² [50]. Robust declassification requires that active attackers—formalized as adversarial transition steps—do not observe any more information from

²In their setting, “declassification” corresponds to *all* leakages, not just explicit decl observations.

a program than passive attackers. Although the definition of robust declassification predates Spectre attacks by nearly two decades, it can be instantiated to our setting: The labeling of registers and arrays in our setting corresponds to their lattice of security domains, and their equivalence classes of states at each transition step are represented by our sequences of observations O . Passive attackers correspond to sequential executions limited to step transitions, while active attackers correspond to speculative executions governed by adversarial directives $d \in \text{Dir}$. Under this framework, robust declassification states that if the sequential execution of any two initial states produces equivalent observations, then the speculative traces must produce equivalent observations as well.

Information Flow and Constant-time. There is a significant body of work on information flow and declassification, see e.g., [38, 40]. Sabelfeld and Myers [39] introduce delimited information release, and show how it can be enforced by a classic information-flow type system. We model security of countermeasures in a spirit that is close to delimited release. However, their formal definition only considers sequential semantics, whereas our definition contrasts sequential and speculative semantics. In addition, we reason about leakage, whereas (as is common for information-flow type systems) they reason about equality of outputs.

There is a growing body of work that uses type systems and static program analyses to enforce that programs are constant-time; see [6] for a recent survey. However, these works either do not consider or do not provide guarantees for declassification. The only exception is [3], which supports *public outputs*, a form of declassification that is required by cryptographic libraries, e.g., to disclose the length of a string. However, [3]’s focus on declassification is complementary to ours: They show how to relax the verification algorithm so that leaks that do not reveal more than the public output are not considered insecure; their relaxation allows for more efficient code to be written. Our work provides instead a means to protect against unintended leakage caused by speculative execution.

Speculative Semantics. There is a growing body of work that applies language-based techniques to reason about security under speculative execution; see [16] for a recent survey. Our adversarial semantics is inspired by [15] and by [8]; however, we refine their adversarial directives: Attackers in their semantics explicitly specify which direction each branch speculates, e.g., with directives like `force \top` , and so may not always *misspeculate*. In contrast, our `force` directive *always* takes the incorrect branch, letting us easily delineate misspeculated execution and simplify our proofs. In addition, our semantics differs from [8] in its treatment of unsafe memory accesses: Indeed, the semantics of [8] immediately aborts execution and leaks the complete memory whenever an unsafe access is performed. Their semantics is thus too coarse for reasoning about the security of our countermeasure (or speculative load hardening). In contrast, our semantics resolves unsafe memory accesses with adversarial directives.

Prior semantics serve as the basis for defining security

of applications under speculative execution. These speculative security properties fall into two categories: *Direct* and *relative* properties [16]. Direct notions, such as *speculative constant-time* [15], are specified as 2-trace non-interference hyperproperties, and prevent (explicitly typed) secret data from leaking to an attacker. Relative notions, such as *speculative non-interference* (SNI) [24] and *trace property-dependent observational determinism* (TPOD) [18] instead simply prevent an attacker from learning *more* information than they would sequentially. These properties are thus typically specified in terms of *four* traces—two speculative traces for the leakage trace of the program, and two sequential traces to determine the baseline leakage to compare against. Our security property, RNI, is also a form of relative property; however, we require only two execution traces in our definition. As a result, it is stronger than 4-trace properties such as SNI: If a program satisfies RNI then it satisfies SNI. In addition, RNI explicitly handles declassification; we are not aware of any prior work that explicitly connects relative notions with declassification.

Blade [45] is an automated tool that eliminates speculative leaks via hardening loads or inserting fences. Their approach views secret inputs as sources and observations as sinks, and applies classic graph algorithms to infer where protections must be inserted. Their approach is evaluated on WebAssembly implementations of cryptographic algorithms.

Patrignani and Guarnieri [32] study the security impact of compiler transformations and countermeasures in a model of speculative execution. They propose two criteria, called robust speculative safety (RSS) and robust speculative non-interference (RSNI), and evaluate proposed (both theoretical and deployed) countermeasures w.r.t. these criteria. One main difference with our work is that they emphasize *robust compilation*, i.e., properties that are preserved even when the program is linked with arbitrary code. Robustness is an important concern but is not considered in our work. Another key difference is that our language features an explicit construct with declassification; this construct is essential in our setting to capture programmer’s intent, but not considered in [32].

IX. CONCLUSION

We showed that, despite current defenses like SLH, attackers can exploit speculative execution and its interaction with declassification to leak sensitive cryptographic information—we demonstrate these attacks concretely against various implementations of AES. We developed a formal framework to capture speculative declassification and its pitfalls and to define our formal security property *relative non-interference*. We further proved that our proposed countermeasures, *fenced declassification* and *selective SLH*, soundly enforce RNI without imposing a significant performance overhead.

Our preliminary evaluation suggests that selSLH is itself of independent interest and can drastically reduce the overhead of protecting programs against Spectre attacks as compared to traditional SLH. One potential direction for future work is to implement selSLH in an existing compiler framework to

build an efficient, formally verified cryptographic library that achieves the guarantees offered by RNI.

ACKNOWLEDGEMENTS

This research was supported by the Air Force Office of Scientific Research (AFOSR) under award number FA9550-20-1-0425; an ARC Discovery Early Career Researcher Award (project number DE200101577); an ARC Discovery Project (project number DP210102670); the Blavatnik ICRC at Tel-Aviv University; the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA; the Deutsche Forschungsgemeinschaft (DFG, German research Foundation) as part of the Excellence Strategy of the German Federal and State Governments – EXC 2092 CASA - 390781972; the European Commission through the ERC Starting Grant 805031 (EPOQUE); the National Science Foundation under grant CNS-1954712; and gifts from AMD, Google, Intel, and Qualcomm.

REFERENCES

- [1] Onur Aciğmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *CT-RSA*, pages 225–242, 2007. doi: [10.1007/11967668_15](https://doi.org/10.1007/11967668_15). 2, 4
- [2] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *IEEE SP*, pages 870–887, 2019. doi: [10.1109/SP.2019.00066](https://doi.org/10.1109/SP.2019.00066). 4
- [3] José Bacerlar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *USENIX Security*, pages 53–70, 2016. <https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida>. 12
- [4] Ben Amos, Niv Gilboa, and Arbel Levy. Spectre without shared memory. In *SAC*, pages 1944–1951, 2019. doi: [10.1145/3297280.3297470](https://doi.org/10.1145/3297280.3297470). 12
- [5] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. Branch history injection: On the effectiveness of hardware mitigations against cross-privilege Spectre-v2 attacks. In *USENIX Security*, 2022. <https://www.usenix.org/system/files/sec22-barberis.pdf>. 12
- [6] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. SoK: Computer-aided cryptography. In *IEEE SP*, pages 777–795, 2021. doi: [10.1109/SP40001.2021.00008](https://doi.org/10.1109/SP40001.2021.00008). 12
- [7] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In *CCS*, pages 1267–1279, 2014. doi: [10.1145/2660267.2660283](https://doi.org/10.1145/2660267.2660283). 4
- [8] Gilles Barthe, Sunjay Cauligi, Benjamin Grégoire, Adrien Koutsos, Kevin Liao, Tiago Oliveira, Swarn Priya, Tamara Rezk, and Peter Schwabe. High-assurance cryptography in the Spectre era. In *IEEE SP*, pages 1884–1901, 2021. doi: [10.1109/SP40001.2021.00046](https://doi.org/10.1109/SP40001.2021.00046). 4, 5, 12
- [9] Daniel J. Bernstein. ChaCha, a variant of Salsa20. In *Workshop record of SASC*, volume 8, pages 3–5. Lausanne, Switzerland, 2008. <https://cr.yp.to/chacha/chacha-20080120.pdf>. 9
- [10] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of cryptographic engineering*, 2(2):77–89, 2012. doi: [10.1007/s13389-012-0027-1](https://doi.org/10.1007/s13389-012-0027-1). 9
- [11] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMOtherSpectre: Exploiting speculative execution through port contention. In *CCS*, pages 785–800, 2019. doi: [10.1145/3319535.3363194](https://doi.org/10.1145/3319535.3363194). 12
- [12] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses.

- In *USENIX Security*, pages 249–266, 2019. <https://www.usenix.org/system/files/sec19-canella.pdf>. 3
- [13] Chandler Carruth. Speculative load hardening – a Spectre variant #1 mitigation technique. LLVM documentation. <https://llvm.org/docs/SpeculativeLoadHardening.html>. 2
- [14] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. FaCT: a DSL for timing-sensitive computation. In *PLDI*, pages 174–189, 2019. doi: 10.1145/3314221.3314605. 1, 5, 8
- [15] Sunjay Cauligi, Craig Disselkoben, Klaus von Gleissenthall, Dean M. Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. Constant-time foundations for the new Spectre era. In Alastair F. Donaldson and Emina Torlak, editors, *PLDI*, pages 913–926, 2020. doi: 10.1145/3385412.3385970. 3, 4, 5, 9, 12, 13
- [16] Sunjay Cauligi, Craig Disselkoben, Daniel Moghimi, Gilles Barthe, and Deian Stefan. SoK: Practical foundations for Spectre defenses. In *IEEE SP*, pages 666–680, 2022. doi: 10.1109/SP46214.2022.9833707. 1, 6, 12, 13
- [17] Anirban Chakraborty, Sarani Bhattacharya, Manaa Alam, Sikhar Patranabis, and Debdeep Mukhopadhyay. RASSLE: return address stack based side-channel leakage. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(2):275–303, 2021. doi: 10.46586/tches.v2021.i2.275-303. 2
- [18] Kevin Cheang, Cameron Rasmussen, Sanjit Seshia, and Pramod Subramanyan. A formal approach to secure speculation. In *CSF*, 2019. doi: 10.1109/CSF.2019.00027. 13
- [19] Frank Denis. libsodium. <https://github.com/jedisct1/libsodium>. 8
- [20] Dmitry Evtvyushkin, Dmitry V. Ponomarev, and Nael B. Abu-Ghazaleh. Jump over ASLR: attacking branch predictors to bypass ASLR. In *MICRO*, pages 40:1–40:13, 2016. doi: 10.1109/MICRO.2016.7783743. 2, 4
- [21] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptogr. Eng.*, 8(1):1–27, 2018. doi: 10.1007/s13389-016-0141-6. 2
- [22] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *USENIX Security Symposium*, pages 955–972, 2018. <https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-gras.pdf>. 2
- [23] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security Symposium*, pages 897–912, 2015. <https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-gruss.pdf>. 2
- [24] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. Spectector: Principled detection of speculative information flows. In *IEEE SP*, pages 1–19, 2020. doi: 10.1109/SP40000.2020.00011. 3, 6, 13
- [25] Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. “They’re not that hard to mitigate”: What cryptographic library developers think about timing attacks. In *IEEE SP*, pages 632–649, 2022. doi: 10.1109/SP46214.2022.9833713. 1
- [26] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. PKRU-Safe: Automatically locking down the heap between safe and unsafe languages. In *EuroSys*, pages 132–142, 2022. doi: 10.1145/3492321.3519582. 3, 12
- [27] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE SP*, pages 1–19, 2019. doi: 10.1109/SP.2019.00002. 1, 3
- [28] Adam Langley. curve25519-donna. <https://github.com/agl/curve25519-donna>. 8, 9
- [29] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *IEEE SP*, pages 605–622, 2015. doi: 10.1109/SP.2015.43. 2
- [30] Shравan Narayan, Craig Disselkoben, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, and Deian Stefan. Swivel: Hardening WebAssembly against Spectre. In *USENIX Security Symposium*, pages 1433–1450, 2021. <https://www.usenix.org/system/files/sec21-narayan.pdf>. 3
- [31] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *CT-RSA*, pages 1–20, 2006. doi: 10.1007/11605805_1. 2, 12
- [32] Marco Patrignani and Marco Guarnieri. Exorcising Spectres with secure compilers. In *CCS*, pages 445–461, 2021. doi: 10.1145/3460120.3484534. 3, 13
- [33] The OpenSSL Project. openssl. <https://github.com/openssl/openssl>. 8
- [34] Charles Reis, Alexander Moshchuk, and Nasko Oskov. Site isolation: process separation for web sites within the browser. In *USENIX Security*, pages 1661–1678, 2019. <https://www.usenix.org/system/files/sec19-reis.pdf>. 12
- [35] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M. Tullsen, and Ashish Venkat. I see dead μ ops: Leaking secrets via Intel/AMD micro-op caches. In *ISCA*, pages 361–374, 2021. doi: 10.1109/ISCA52012.2021.00036. 12
- [36] Stephen Röttger and Arthur Janc. A Spectre proof-of-concept for a Spectre-proof web. <https://security.googleblog.com/2021/03/a-spectre-proof-of-concept-for-spectre.html>, 2021. 12
- [37] John Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-02, SRI International, Computer Science Laboratory, 1992. <http://www.csl.sri.com/papers/csl-92-2/csl-92-2.pdf>. 7
- [38] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE J. Sel. Areas Commun.*, 21(1):5–19, 2003. doi: 10.1109/JASC.2002.806121. 12
- [39] Andrei Sabelfeld and Andrew C. Myers. A model for delimited information release. In *ISSS*, pages 174–191, 2003. doi: 10.1007/978-3-540-37621-7_9. 12
- [40] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *J. Comput. Secur.*, 17(5):517–548, 2009. doi: 10.3233/JCS-2009-0352. 1, 12
- [41] Aria Shahverdi, Mahammad Shirinov, and Dana Dachman-Soled. Database reconstruction from noisy volumes: A cache side-channel attack on SQLite. In *USENIX Security Symposium*, pages 1019–1035, 2021. <https://www.usenix.org/system/files/sec21-shahverdi.pdf>. 2
- [42] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. In *USENIX Security Symposium*, pages 639–656, 2019. <https://www.usenix.org/system/files/sec19-shusterman.pdf>. 2
- [43] Anatoly Shusterman, Ayush Agarwal, Sioli O’Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. Prime+Probe 1, JavaScript 0: Overcoming browser-based side-channel defenses. In *USENIX Security Symposium*, pages 2863–2880, 2021. <https://www.usenix.org/system/files/sec21-shusterman.pdf>. 2, 4
- [44] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: secure, efficient in-process isolation with protection keys (MPK). In *USENIX Security Symposium*, pages 1221–1238, 2019. https://www.usenix.org/system/files/sec19-vahldiek-oberwagner_0.pdf. 3, 12
- [45] Marco Vassena, Craig Disselkoben, Klaus v. Gleissenthall, Sunjay Cauligi, Rami Gökhan Kıcı, Ranjit Jhala, Dean Tullsen, and Deian Stefan. Automatically eliminating speculative leaks from cryptographic code with Blade. In *POPL*, pages 1–30, 2021. doi: 10.1145/3434330. 3, 13
- [46] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. You shall not (by)pass! practical, secure, and fast PKU-based sandboxing. In *EuroSys*, pages 266–282, 2022. doi: 10.1145/3492321.3519560. 3, 12
- [47] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher W. Fletcher, Roy H. Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *IEEE SP*, pages 888–904, 2019. doi: 10.1109/SP.2019.00004. 2
- [48] Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas. Cache telepathy: Leveraging shared resource attacks to learn DNN architectures.

- [49] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, pages 719–732, 2014. <https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-yarom.pdf>. 2, 4, 10, 11
- [50] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *CSFW*, pages 15–23, 2001. doi: 10.1109/CSFW.2001.930133. 6, 12

APPENDIX A ADDITIONAL PROOFS

Lemma 4 (Unwinding lemma for sequential execution). *Let $d = \text{step}$, force. If $\vdash c$, then for every pair of execution steps:*

$$\begin{aligned} \langle c, \rho_1, \mu_1, \perp \rangle &\xrightarrow[d]{o_1} \langle c'_1, \rho'_1, \mu'_1, b'_1 \rangle \\ \langle c, \rho_2, \mu_2, \perp \rangle &\xrightarrow[d]{o_2} \langle c'_2, \rho'_2, \mu'_2, b'_2 \rangle \end{aligned}$$

we have:

$$\begin{aligned} (\rho_1, \mu_1) &\sim (\rho_2, \mu_2) \wedge o_1 = o_2 \\ \implies (\rho'_1, \mu'_1) &\sim (\rho'_2, \mu'_2) \wedge c'_1 = c'_2 \wedge b'_1 = b'_2. \end{aligned}$$

Proof. We prove that $(\rho'_1, \mu'_1) \sim (\rho'_2, \mu'_2)$ and $c'_1 = c'_2$ and $b'_1 = b'_2$ by case analysis on the structure of c . The second and third item are immediate to establish so we focus on the first.

- skip and fence are trivial as the memory and register maps are left unchanged.
- $x := e$ and $\Gamma(x) = H$. By assumption, $(\rho_1, \mu_1) \sim (\rho_2, \mu_2)$. As $\Gamma(x) = H$, we have $\rho_1 \sim \rho'_1$ and $\rho_2 \sim \rho'_2$, where $\rho'_i = \rho_i[x := \llbracket e \rrbracket_{\rho_i}]$. Hence by transitivity $(\rho'_1, \mu_1) \sim (\rho'_2, \mu_2)$.
- $x := e$ and $\Gamma(x) = L$. By assumption, $(\rho_1, \mu_1) \sim (\rho_2, \mu_2)$. The typing rule guarantees that $\Gamma(e) \leq \Gamma(x)$ and hence $\Gamma(e) = L$. It follows that $\llbracket e \rrbracket_{\rho_1} = \llbracket e \rrbracket_{\rho_2}$ and hence that $\rho'_1 \sim \rho'_2$, where $\rho'_i = \rho_i[x := \llbracket e \rrbracket_{\rho_i}]$. Hence by transitivity $(\rho'_1, \mu_1) \sim (\rho'_2, \mu_2)$.
- $x := a[e]$. By assumption, $(\rho_1, \mu_1) \sim (\rho_2, \mu_2)$. The typing rule guarantees that $\Gamma(e) = L$ and $\Gamma(a) \leq \Gamma(x)$. Since $(\rho_1, \mu_1) \sim (\rho_2, \mu_2)$, it follows that $\llbracket e \rrbracket_{\rho_1} = \llbracket e \rrbracket_{\rho_2}$. Moreover $\rho'_1 \sim \rho'_2$, where $\rho'_i = \rho_i[x := \mu_i[(a, \llbracket e \rrbracket_{\rho_i})]]$. Hence by transitivity $(\rho'_1, \mu_1) \sim (\rho'_2, \mu_2)$.
- $a[e] := e'$ and $\Gamma(a) = H$. By assumption, $(\rho_1, \mu_1) \sim (\rho_2, \mu_2)$. The typing rule guarantees that $\Gamma(e) = L$, hence $\llbracket e \rrbracket_{\rho_1} = \llbracket e \rrbracket_{\rho_2}$. Moreover $\mu'_1 \sim \mu'_2$, where $\mu'_i = \mu_i[(a, \llbracket e \rrbracket_{\rho_i}) := \llbracket e' \rrbracket_{\rho_i}]$. Hence by transitivity $(\rho_1, \mu'_1) \sim (\rho_2, \mu'_2)$;
- $a[e] := e'$ and $\Gamma(a) = L$. By assumption, $(\rho_1, \mu_1) \sim (\rho_2, \mu_2)$. The typing rule guarantees that $\Gamma(e) = L$ and $\Gamma(x) = L$, hence $\llbracket e \rrbracket_{\rho_1} = \llbracket e \rrbracket_{\rho_2}$. Moreover $\llbracket e' \rrbracket_{\rho_1} = \llbracket e' \rrbracket_{\rho_2}$ and hence $\mu'_1 \sim \mu'_2$, where $\mu'_i = \mu_i[(a, \llbracket e \rrbracket_{\rho_i}) := \llbracket e' \rrbracket_{\rho_i}]$. Hence by transitivity $(\rho_1, \mu'_1) \sim (\rho_2, \mu'_2)$.
- $x := \text{declassify } e$. By assumption, $(\rho_1, \mu_1) \sim (\rho_2, \mu_2)$. The first execution leaks $\llbracket e \rrbracket_{\rho_1}$ and the second execution leaks $\llbracket e \rrbracket_{\rho_2}$. Again by assumption, the two observations o_1 and

o_2 are equal, hence $\llbracket e \rrbracket_{\rho_1} = \llbracket e \rrbracket_{\rho_2}$. Therefore $\rho'_1 \sim \rho'_2$, where $\rho'_i = \rho_i[x := \llbracket e \rrbracket_{\rho_i}]$. By transitivity $(\rho'_1, \mu_1) \sim (\rho'_2, \mu_2)$.

- if t then c' else c'' and $d = \text{step}$. By assumption, $(\rho_1, \mu_1) \sim (\rho_2, \mu_2)$. The first execution leaks $\llbracket t \rrbracket_{\rho_1}$ and the second execution leaks $\llbracket t \rrbracket_{\rho_2}$. Again by assumption, the two observations o_1 and o_2 are equal, hence $\llbracket t \rrbracket_{\rho_1} = \llbracket t \rrbracket_{\rho_2}$, and so both executions take the same branch (corresponding to $\llbracket t \rrbracket_{\rho_i}$). The memory and register maps are left unchanged.
- if t then c' else c'' and $d = \text{force}$. By assumption, $(\rho_1, \mu_1) \sim (\rho_2, \mu_2)$. The first execution leaks $\llbracket t \rrbracket_{\rho_1}$ and the second execution leaks $\llbracket t \rrbracket_{\rho_2}$. Again by assumption, the two observations o_1 and o_2 are equal, hence $\llbracket t \rrbracket_{\rho_1} = \llbracket t \rrbracket_{\rho_2}$, and so both executions take the same branch (corresponding to $\neg \llbracket t \rrbracket_{\rho_i}$). The memory and register maps are left unchanged.
- while t do c' proceeds similar to if.
- $c'; c''$. The result follows from induction on c' and c'' . □

Lemma 5 (Unwinding lemma for idealized speculative execution). *If $\vdash c$ then for every pair of execution steps:*

$$\begin{aligned} \langle c, \rho_1, \mu_1, \top \rangle &\xrightarrow[d]{o_1} \langle c'_1, \rho'_1, \mu'_1, \top \rangle \\ \langle c, \rho_2, \mu_2, \top \rangle &\xrightarrow[d]{o_2} \langle c'_2, \rho'_2, \mu'_2, \top \rangle \end{aligned}$$

we have:

$$\rho_1 \sim \rho_2 \implies o_1 = o_2 \wedge \rho'_1 \sim \rho'_2 \wedge c'_1 = c'_2$$

Proof. By case analysis on the structure of c .

- skip is trivial.
- fence does not apply, since the rule requires $b = \perp$.
- $x := e$ and $\Gamma(x) = H$. By assumption, $\rho_1 \sim \rho_2$. As $\Gamma(x) = H$, we have $\rho_1 \sim \rho'_1$ and $\rho_2 \sim \rho'_2$, where $\rho'_i = \rho_i[x := \llbracket e \rrbracket_{\rho_i}]$. Hence by transitivity $\rho'_1 \sim \rho'_2$.
- $x := e$ and $\Gamma(x) = L$. By assumption, $\rho_1 \sim \rho_2$. The typing rule guarantees that $\Gamma(e) \leq \Gamma(x)$ and hence $\Gamma(e) = L$. It follows that $\llbracket e \rrbracket_{\rho_1} = \llbracket e \rrbracket_{\rho_2}$ and hence that $\rho'_1 \sim \rho'_2$, where $\rho'_i = \rho_i[x := \llbracket e \rrbracket_{\rho_i}]$.
- $x := a[e]$, $\Gamma(x) = H$, and $\llbracket e \rrbracket_{\rho} \in [0, |a|)$. By assumption, $\rho_1 \sim \rho_2$. The typing rule guarantees that $\Gamma(e) = L$, hence $\llbracket e \rrbracket_{\rho_1} = \llbracket e \rrbracket_{\rho_2}$ and the observations o_1 and o_2 coincide. We also have $\rho_1 \sim \rho'_1$ and $\rho_2 \sim \rho'_2$, hence by transitivity $\rho'_1 \sim \rho'_2$ where $\rho'_i = [x := \mu_i[(a, \llbracket e \rrbracket_{\rho_i})]]$.
- $x := a[e]$, $\Gamma(x) = H$, and $\llbracket e \rrbracket_{\rho} \notin [0, |a|)$. By assumption, $\rho_1 \sim \rho_2$. The typing rule guarantees that $\Gamma(e) = L$, hence $\llbracket e \rrbracket_{\rho_1} = \llbracket e \rrbracket_{\rho_2}$ and the observations o_1 and o_2 coincide. We also have $\rho_1 \sim \rho'_1$ and $\rho_2 \sim \rho'_2$ where $\rho'_i = [x := v_i]$ and v_i is chosen adversarially; hence by transitivity $\rho'_1 \sim \rho'_2$.
- $x := a[e]$, $\Gamma(x) = L$. By assumption, $\rho_1 \sim \rho_2$. The typing rule guarantees that $\Gamma(e) = L$, hence $\llbracket e \rrbracket_{\rho_1} = \llbracket e \rrbracket_{\rho_2}$

and the observations o_1 and o_2 coincide. Moreover in the idealized semantics, we have $\rho'_i = \rho_i[x := 0]$; thus $\rho'_1 \sim \rho'_2$.

- $a[e] := e'$. By assumption, $\rho_1 \sim \rho_2$. The typing rule guarantees that $\Gamma(e) = L$, hence $\llbracket e \rrbracket_{\rho_1} = \llbracket e \rrbracket_{\rho_2}$ and the observations o_1 and o_2 coincide. The register map is left unchanged.
- $x :=_{\text{declassify}} e$. By assumption, $\rho_1 \sim \rho_2$. In the idealized semantics, both executions leak $\text{decl } 0$ and hence the two observations o_1 and o_2 are equal. Moreover $\rho'_i = \rho_i[x := 0]$; thus $\rho'_1 \sim \rho'_2$.
- if t then c' else c'' and $d = \text{step}$. By assumption, $\rho_1 \sim \rho_2$. The typing rule guarantees that $\Gamma(t) = L$, hence $\llbracket t \rrbracket_{\rho_1} = \llbracket t \rrbracket_{\rho_2}$ and the observations o_1 and o_2 coincide. Both executions take the same branch, corresponding to $\llbracket t \rrbracket_{\rho_i}$. The register maps are left unchanged.
- if t then c' else c'' and $d = \text{force}$. By assumption, $\rho_1 \sim \rho_2$. The typing rule guarantees that $\Gamma(t) = L$, hence $\llbracket t \rrbracket_{\rho_1} = \llbracket t \rrbracket_{\rho_2}$ and the observations o_1 and o_2 coincide. Both executions take the same branch, corresponding to $\neg \llbracket t \rrbracket_{\rho_i}$. The register maps are left unchanged.
- while t do c' proceeds similar to if.
- $c'; c''$. The result follows from induction on c' and c'' .

□

The correctness of the transformation as stated in [Lemma 6](#) is imprecise, because the transformed program performs “administrative” steps to update the speculation flag. The correctness is stated precisely using an erasure function $|\cdot|$ that takes as input a list of directives and a list of observations of the same length, and filters out all entries that contain a • observation. Formally, $|\cdot|$ is defined inductively by the clauses:

$$\begin{aligned} |(\epsilon, \epsilon)| &= (\epsilon, \epsilon) \\ |(d :: D, \bullet :: \mathcal{O})| &= |(D, \mathcal{O})| \\ |(d :: D, o :: \mathcal{O})| &= \text{let } (D', \mathcal{O}') = |(D, \mathcal{O})| \\ &\quad \text{in } (d :: D', o :: \mathcal{O}') \quad \text{if } o \neq \bullet \end{aligned}$$

Lemma 7 (Implementation of idealized semantics). *If*

$$\langle c, \rho, \mu, b \rangle \Downarrow_D^{\mathcal{O}} \langle c', \rho', \mu', b' \rangle$$

then there exists D' such that

$$\langle \llbracket c \rrbracket, \rho[\tilde{b} := b], \mu, b \rangle \Downarrow_{D'}^{\mathcal{O}'} \langle c', \rho'[\tilde{b} := b'], \mu', b' \rangle$$

and $|(D, \mathcal{O})| = |(D', \mathcal{O}')|$. Conversely, if

$$\langle \llbracket c \rrbracket, \rho[\tilde{b} := b], \mu, b \rangle \Downarrow_{D'}^{\mathcal{O}'} \langle \llbracket c' \rrbracket, \rho'[\tilde{b} := b'], \mu', b' \rangle$$

then there exists D such that

$$\langle c, \rho, \mu, b \rangle \Downarrow_D^{\mathcal{O}} \langle c', \rho', \mu', b' \rangle$$

and $|(D, \mathcal{O})| = |(D', \mathcal{O}')|$.

Both implications are proved by induction on the length of the execution. The base case is proved by inspection on the semantics.

Definition 2 presents *2-trace RNI*. We show that this is a stronger property than the typical 4-trace hyperproperty for speculative security.

Definition 3 (4-trace RNI). *A program c is 4-trace RNI iff for every pair of executions $\langle c, \rho_1, \mu_1, \perp \rangle \Downarrow_D^{\mathcal{O}_1}$ and $\langle c, \rho_2, \mu_2, \perp \rangle \Downarrow_D^{\mathcal{O}_2}$ we have:*

$$\mathcal{O}'_1 = \mathcal{O}'_2 \implies \mathcal{O}_1 = \mathcal{O}_2$$

where $\langle c, \rho_1, \mu_1 \rangle \Downarrow^{\mathcal{O}'_1}$ (resp. \mathcal{O}'_2) is a complete sequential execution.

Lemma 8. *If a program c satisfies 2-trace RNI, then it also satisfies 4-trace RNI.*

Proof. Suppose c satisfies 2-trace RNI. Let $\langle c, \rho_1, \mu_1, \perp \rangle \Downarrow_D^{\mathcal{O}_1}$ and $\langle c, \rho_2, \mu_2, \perp \rangle \Downarrow_D^{\mathcal{O}_2}$ be two executions of c such that $\langle c, \rho_1, \mu_1 \rangle \Downarrow^{\mathcal{O}'_1}$ and $\langle c, \rho_2, \mu_2 \rangle \Downarrow^{\mathcal{O}'_2}$ and $\mathcal{O}'_1 = \mathcal{O}'_2$. Since \mathcal{O}'_1 (resp. \mathcal{O}'_2) is the sequential trace of c , any sequential prefix of D will produce a prefix of \mathcal{O}'_1 (resp. \mathcal{O}'_2) when executed from the same initial state. Thus for any sequential prefix D^* and $\langle c, \rho_1, \mu_1, \perp \rangle \Downarrow_{D^*}^{\mathcal{O}'_1}$ and $\langle c, \rho_2, \mu_2, \perp \rangle \Downarrow_{D^*}^{\mathcal{O}'_2}$ we have $\mathcal{O}^*_1 = \mathcal{O}^*_2$. Fix D^* as the longest sequential prefix of D . By assumption, c satisfies 2-trace RNI, so $\mathcal{O}^*_1 = \mathcal{O}^*_2 \implies \mathcal{O}_1 = \mathcal{O}_2$. □

APPENDIX B AES KEY RECOVERY

A. Further Background on AES

As described in [Section VII-A](#), AES represents the state as a 4×4 matrix of bytes. Each of these bytes represents an element in $\text{GF}(2^8)$, with the reducing polynomial $x^8 + x^4 + x^3 + x + 1$. A byte with a value $b = \sum_{i=0}^7 b_i 2^i$ represents the polynomial $\sum_{i=0}^7 b_i x^i$. Thus, for example, a byte value of 3 represents the polynomial $x + 1$.

The `MixColumns` operation computes the product of the state and the fixed matrix

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}.$$

Thus, each output column is a linear transformation of the input column. We note that `MixColumns` is the only AES operation to provide diffusion between state bytes.

B. Key Recovery of Reduced-Round AES

In this section we describe how we recover keys from messages disclosed by the attack described in [Section VII](#).

A two-round AES consists of one internal round and one final round. Recall that an internal round consists of the operations `SubBytes`, `ShiftRows`, `MixColumns`, and `AddRoundKey`, whereas the final round only consists of `SubBytes`, `ShiftRows`, and `AddRoundKey`. Hence, with the addition of the key blinding step before the encryption, we have that for a plaintext P , the reduced-round ciphertext is $RRC = k^2 \oplus SR(SB(k^1 \oplus MC(SR(SB(k^0 \oplus P))))$, where k^i is the round key for round i .

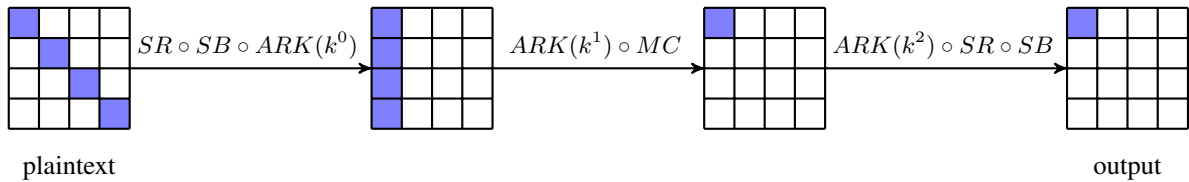


Figure 8: The blue tiles represents bytes that are affected by fixing $k_0^0, k_5^0, k_{10}^0, k_{15}^0$ of the first round key, k_0^1 of the second round key, and k_0^2 of the third round key. The operations AddRoundKey, SubBytes, ShiftRows, MixColumns are shortened to ARK, SB, SR, MC respectively.

We assume that the attacker knows P and recovers RRC through the transient-execution attack. To recover the key, we use a divide-and-conquer strategy. Specifically, we note that the MixColumns operation, which is the only AES step that mixes data between state bytes, only appears once in the derivation of RRC . Consequently, diffusion across a two-round AES is extremely limited, and each byte of the reduced-round ciphertext depends on exactly four plaintext bytes. This is depicted in Figure 8, where we see that plaintext bytes $P_0, P_5, P_{10},$ and P_{15} are the only plaintext bytes that affect RRC_0 .

We can therefore split the keys to only those bytes that affect a targeted ciphertext byte and recover them independently of other key bits. For example, observing Figure 8, we see that we need to determine the key bytes at the positions of the shaded tiles for k^0 , i.e. $k_0^0, k_5^0, k_{10}^0,$ and k_{15}^0 . We further need to recover the key bytes that affect RRC_0 in rounds 1 and 2, i.e. k_0^1 and k_0^2 . A straightforward approach would be to brute force these by first collecting a number of plaintexts and matching reduced-round ciphertexts, and then rejecting a guess of key bytes that does not match one or more of the pairs. However, this requires guessing 48 bits for each quarter of the state, to a total expected complexity of 2^{50} .

We can improve the attack complexity significantly by considering pairs of encryptions that agree on a byte. Assume we have a pair of plaintexts P and P' , such that for the corresponding reduced-round ciphertexts we have $RRC_0 = RRC'_0$. For such a pair, we can guess key bytes $k_0^0, k_5^0, k_{10}^0,$ and k_{15}^0 and check whether after the MixColumns step we get the same value at state byte 0. We find that on average we need 6 pairs that match on a byte in a column to recover the key bytes that match the column. On average, we need to try 26 plaintexts to find the required number of pairs. Finally, because we can reuse the plaintexts to attack all columns, the total number of plaintexts we need to encrypt is 31.

Brute forcing four key bytes requires at most 2^{32} tries and takes on the order of a few minutes on a typical laptop. Thus, the total complexity of recovering k^0 is 2^{34} with 31 encryption samples on average. For AES-128, k^0 is identical to the master key. For AES-192 and AES-256, we need to also recover k^1 . Fortunately, having recovered k^0 , we can guess one byte each of k^1 and k^2 and compare against the resulting byte of RRC . Thus, with an additional complexity of $2^{16} \cdot 16 = 2^{20}$ we can recover the 16 bytes of k^1 . On average, we need to try 3 plaintexts to recover each byte of k^1 . The plaintexts used to attack k^0 previously can be reused. So, there are no additional

samples needed to find k^1 .

C. Related-Cipher Attack

We also consider the attack scenario where the attacker can extend the number of rounds AES performs. For example, in the case of AES-128, this can occur when we use a standard AES-192 key for the training. The use of an AES-192 key trains the branch predictor that AES is performed with 12 rounds. In practice, however, we find implementations access precomputed round keys and that such accesses are protected by SLH after misprediction. Nevertheless, we still present the cryptanalysis of recovering the AES-128 key from a hypothetical implementation that computes round keys on-the-fly.

Let S denote the cipher state after performing nine rounds. For simplicity, we also use $S' = SR(SB(S))$. In the normal execution of AES, the cipher now performs a final round to calculate the ciphertext. Hence, the ciphertext is $C = k^{10} \oplus S'$. Thus, if we can determine S' , we can find the round key k^{10} .

Next, we note that when the cipher continues for two additional rounds before the final round executes we get $C' = k^{12} \oplus SR(SB(k^{11} \oplus MC(SR(SB(k^{10} \oplus MC(S'))))))$. For the key recovery, we assume that memory is scrubbed before it is used for key material. Consequently, in AES-128 the keys for rounds 11 and 12 are not initialized and remain 0. We can, therefore, compute

$$\begin{aligned} \hat{C} &= SB^{-1}(SR^{-1}(MC^{-1}(k^{11} \oplus SB^{-1}(SR^{-1}(k^{12} \oplus C'))))) \\ &= k^{10} \oplus MC(S') \end{aligned}$$

We now have $C \oplus \hat{C} = (k^{10} \oplus S') \oplus (k^{10} \oplus MC(S')) = S' \oplus MC(S')$. Recall that for a state S' , we have

$$MC(S') = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \cdot S'$$

Thus,

$$\begin{aligned} C \oplus \hat{C} &= S' \oplus MC(S') \\ &= S' \oplus \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \cdot S' = \begin{bmatrix} 3 & 3 & 1 & 1 \\ 1 & 3 & 3 & 1 \\ 1 & 1 & 3 & 3 \\ 3 & 1 & 1 & 3 \end{bmatrix} \cdot S' \end{aligned}$$

Unfortunately, the matrix

$$\begin{bmatrix} 3 & 3 & 1 & 1 \\ 1 & 3 & 3 & 1 \\ 1 & 1 & 3 & 3 \\ 3 & 1 & 1 & 3 \end{bmatrix}$$

is singular. Hence, we cannot invert it to find S' from $C \oplus \hat{C}$. However, we find that for each of the columns of S' there are exactly 2^8 values that can satisfy the equation. Specifically, for each column we can select a value for one of the bytes and use the linear relationship between S' and $C \oplus \hat{C}$ to determine the other values of the column. For each combination of values for the four columns of S' we get a guess of k^{10} . We can now reverse the key expansion to get a guess of the key, and test whether the guess is correct. Overall, we need to test 2^{32} such combinations, recovering the key in less than 10 minutes on a typical laptop.