Robust Constant-Time Cryptography

MATTHEW KOLOSICK, UC San Diego, USA
BASAVESH AMMANAGHATTA SHIVAKUMAR, MPI-SP, Germany
SUNJAY CAULIGI, MPI-SP, Germany
MARCO PATRIGNANI, University of Trento, Italy
MARCO VASSENA, Utrecht University, Netherlands
RANJIT JHALA, UC San Diego, USA
DEIAN STEFAN, UC San Diego, USA

Cryptographic *library* developers take care to ensure their library does not leak secrets even when there are (inevitably) exploitable vulnerabilities in the *applications* the library is linked against. To do so, they choose some class of application vulnerabilities to defend against and hardcode protections against those vulnerabilities in the library code. A single set of choices is a poor fit for all contexts: a chosen protection could impose unnecessary overheads in contexts where those attacks are impossible, and an ignored protection could render the library insecure in contexts where the attack is feasible.

We introduce RoboCop, a new methodology and toolchain for building secure *and* efficient applications from cryptographic libraries, via four contributions. First, we present an operational semantics that describes the behavior of a (cryptographic) library executing in the context of a potentially vulnerable application so that we can precisely specify what different attackers can observe. Second, we use our semantics to define a novel security property, *Robust Constant Time* (RCT), that defines when a cryptographic library is secure *in the context of* a vulnerable application. Crucially, our definition is parameterized by an attacker model, allowing us to factor out the classes of attackers that a library may wish to secure against. This refactoring yields our third contribution: a compiler that can synthesize bespoke cryptographic libraries with security tailored to the specific application context against which the library will be linked, guaranteeing that the library is RCT in that context. Finally, we present an empirical evaluation that shows the RoboCop compiler can automatically generate code to efficiently protect a wide range (over 500) of cryptographic library primitives against three classes of attacks: read gadgets (due to application memory safety vulnerabilities), speculative read gadgets (due to application speculative execution vulnerabilities), and concurrent observations (due to application threads), with performance overhead generally under 2% for protections from read gadgets and under 4% for protections from speculative read gadgets, thus freeing library developers from making one-size-fits-all choices between security and performance.

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

Additional Key Words and Phrases: Spectre, speculative execution, secure compilation, cryptography

ACM Reference Format:
Matthew Kolosick, Basavesh Ammanaghatta Shivakumar, Sunjay Cauligi, Marco Patrignani, Marco Vassena, Ranjit Jhala, and Deian Stefan. 2025. Robust Constant-Time Cryptography. *Proc. ACM Program. Lang.* 9, PLDI, Article 207 (June 2025), 25 pages. https://doi.org/10.1145/3729310

Authors' Contact Information: Matthew Kolosick, UC San Diego, San Diego, USA; Basavesh Ammanaghatta Shivakumar, MPI-SP, Bochum, Germany; Sunjay Cauligi, MPI-SP, Bochum, Germany; Marco Patrignani, University of Trento, Trento, Italy; Marco Vassena, Utrecht University, Utrecht, Netherlands; Ranjit Jhala, UC San Diego, San Diego, USA; Deian Stefan, UC San Diego, San Diego, USA.

ACM 2475-1421/2025/6-ART207
https://doi.org/10.1145/3729310

```
1  int stream(u8 *c, u64 clen, u8 *n, u8 *k) {
2      ... u8 kcopy[32]; ...
3      for (i = 0; i < 32; i++) { kcopy[i] = k[i]; }
4      ...
5      while (clen >= 64) { crypto_core_salsa20(c, in, kcopy, NULL); ... }
6      ...
7      sodium_memzero(kcopy, sizeof kcopy);
8      return 0;
9  }
```

Fig. 1. An excerpt from the reference implementation of Salsa20 in LibSodium.

1 Introduction

"Don't roll your own crypto" is a well known adage directed at application developers when they are considering using cryptography in their code. Instead developers are exhorted to use cryptographic libraries written by experts who have hopefully learned the hard-won lessons of decades of cryptographic and software security research and practice. For such cryptographic library developers, as well as algorithm designers, one of those hard-won lessons is that cryptographic code must be *constant time* [10]. More recently (due to microarchitectural attacks like Spectre [40]) this lesson has been expanded to the requirement that, under certain circumstances, it is also crucial for cryptographic code to be *speculatively constant time* [15]. Together these ensure that the code in the library does not leak secrets (such as cryptographic keys) through either timing channels or speculative execution, respectively, and significant work has gone into developing theory, tools, and rules of thumb to ensure that cryptographic code is (speculatively) constant time [6, 16].

Attacks via Application Vulnerabilities. Sadly, this is not enough. While a constant time cryptographic library will not *itself* leak secrets, it is but one component executing within the context of a larger *application*. Security vulnerabilities in application code could themselves lead to inadvertent disclosure of secrets, no matter how careful library authors were to avoid vulnerabilities.

Library authors are keenly aware of this problem and routinely add protections to harden their code against application vulnerabilities. For example, consider the excerpt of the implementation of the `Salsa20` stream cipher [11] from `LibSodium` [25] shown in Figure 1. We might hope that the (elided) body being constant time suffices to ensure that the secrets in `kcopy` are not leaked. However the *classic* constant time guarantee only ensures that `stream` does not leak the secrets: it makes no guarantees about what happens if there is a memory safety vulnerability in the application linked against the library. Such a vulnerability can lead to a *read-gadget* that may be used to exfiltrate the secrets in `kcopy`! To defend against such gadgets, `LibSodium`'s developers take care to zero the intermediate memory used in `stream` (Line 7) to ensure those secrets cannot be leaked through memory vulnerabilities that reside in the application. Unfortunately, optimizations like dead-store elimination can remove secret scrubbing and nullify the efforts of `LibSodium`'s developers [70].

Security vs. Performance. Read gadgets are but one of several possible attacks that library developers must defend against. For each attack, the library developer must either manually add relevant defenses to their code, or make an explicit choice *not* to do so (e.g. due to prohibitive overheads). Consequently, library code "bakes in" a subset of possible protections against application vulnerabilities: these may be unnecessary or insufficient depending upon the application context. For example, `LibSodium`'s zeroization protection against read gadgets is unnecessary when linked against a memory safe Rust application. On the other hand, `LibSodium`'s protections are insufficient against Spectre attacks [63]. The library's authors chose to elide an appropriate fence needed to protect against speculative read gadgets as *all* the clients of the library would have to suffer the corresponding performance degradation, not just the ones where Spectre was a legitimate concern.

In a nutshell, cryptographic library developers are currently in a difficult position: they must make one-size-fits-all security-performance trade offs by manually inserting fragile protections orthogonal to the cryptographic algorithms and protocols they are implementing. Luckily, secure compilers offer a promising solution to escape this dilemma [50]. Library users are in a better position to make security-performance trade offs and can provide a compiler with security policies to be automatically enforced through inserted protections. To realize this vision, we introduce RoboCop, a new methodology and toolchain for building secure *and* efficient applications from cryptographic libraries. We develop RoboCop via four concrete contributions:

1. Abstraction: Libraries and Attackers (§3, §4.3). Our first contribution is a formal operational semantics describing the behavior of a library executing within a potentially vulnerable application. We further define a semantics capturing a high-level, abstract model of speculative execution, based on the notion of a speculation oracle that "guesses" the result of evaluating an expression and then later rolling back or committing if the guesses were correct. These semantics provide a unified setting for precisely stating different attackers' observations, guiding the design of our protections.

2. Specification: Robust Constant Time (§4.2). Our second contribution is a novel security property, *robust constant time* (RCT), using our model to precisely define security for a cryptographic library running *in the context of* a potentially vulnerable application. Crucially, our definition is parameterized by an attacker model, capturing the set of attackers that a library is supposed to be secure against. We further define a speculative version, *robust speculative constant time*, capturing constant time in the presence of speculation.

3. Implementation: The RoboCop Compiler (§5). By factoring out assumptions about the context, our notion of RCT enables our third contribution: a compiler that takes a cryptographic library and synthesizes a bespoke binary tailored to the application against which the library will be linked. To do so, we show how to map each kind of attacker to a concrete code transform that provably, with respect to our definition of RCT, protects against that attacker. Thus, our RoboCop compiler lets library developers focus on implementing constant-time cryptographic algorithms, without having to worry about baking in potentially inefficient or insecure protections against application vulnerabilities. Instead, protections can be automatically inserted based on the application context, thereby ensuring the same library code can be securely *and* efficiently reused in all contexts.

4. Evaluation: SUPERCOP (§6). Finally, our fourth contribution is an empirical evaluation that shows that our RoboCop compiler can automatically generate protections for a wide range of cryptographic library code defending against a variety of attacks. Here we modify the `SUPERCOP` [65] cryptographic benchmarking suite. We instrument `SUPERCOP` to generate and measure the overhead of protecting against three classes of attacks: read gadgets (due to memory safety vulnerabilities in the application), speculative read gadgets (due to speculative execution in the application), and concurrent observations (due to threads in the application). In our test suite of 507 different implementations of cryptographic operations, we show that our RoboCop compiler can automatically generate code that is secure against application vulnerabilities with the majority of overheads under 2% when protecting against read gadgets and under 4% when protecting against speculative read gadgets, thereby demonstrating that RCT reconciles the tension between security and efficiency when reusing cryptographic libraries in different application contexts.

2 Overview

Every application that works with sensitive or personal user data uses cryptography to ensure the confidentiality or integrity of the data. These cryptographic operations typically rely upon sophisticated mathematics and are notoriously difficult to get right: bugs or security vulnerabilities within cryptographic code risk leaking critical secret keys or data, which could compromise the security of the whole application. Thus, cryptographic operations are typically implemented and encapsulated within *libraries* that are carefully authored and audited by cryptographic experts. These libraries provide trusted implementations of cryptographic operations (e.g. `LibSodium` [25]) or protocols (e.g. `OpenSSL` [5]), that can then be widely reused by developers—without requiring cryptographic expertise—to build secure applications.

In addition to correctly implementing cryptographic algorithms, the developers of cryptographic libraries must carefully ensure their code meets certain *generic* security requirements. For example, they must ensure that their libraries are *constant time*, meaning that the execution time of the library must be independent of the values of the secret data that the library operates over. Otherwise, an attacker can measure the timing variations to learn whether a secret conditioned branch or operation went one way or the other, and from that, eventually recover the data itself. There has been significant work [15, 17] on characterizing when a cryptographic library is constant-time and designing recipes to write constant time code, and this work guides both the design of cryptographic algorithms and their implementation in cryptographic libraries.

2.1 Application (Attacker) Assumptions

Sadly, timing leaks are not the only attacker capability cryptographic library developers need worry about. Ultimately, libraries are not executed in isolation: they are linked against *applications* written by non-expert developers: another source of vulnerabilities through which secrets can be leaked.

Defending Against Application Vulnerabilities. Consider an application written in C that uses the `LibSodium` library. If this application has a *buffer overflow* leading to a memory read then an attacker targeting the whole program now has the ability to read any cryptographic secrets that are left accessible in memory, completely bypassing the need for timing channel based attacks.

In fact, the developers of `LibSodium` are keenly aware of the need to defend against these potential application vulnerabilities. Figure 1 shows an excerpt of `LibSodium`'s reference implementation of the `Salsa20` stream cipher. `stream` takes as input a key `k` and a nonce `n` and outputs a length `clen` stream of pseudo random bytes in the buffer `c`. The developers make a copy, `kcopy`, of the key buffer, `k`. This copied buffer contains secret data that must not be left on the stack, as otherwise a buffer overflow in the application would let an attacker read the secret key off the stack. Thus, in anticipation of `LibSodium` being used in the context of a vulnerable application, on Line 7 the `LibSodium` developers invoke `sodium_memzero` to *zero* out the contents of `kcopy`, thereby ensuring that its value is inaccessible regardless of memory vulnerabilities from the application.

Attacker (Application) Assumptions. In general, cryptographic libraries must defend against vulnerabilities in their host applications. We capture these threats (attacker capabilities) as *assumptions* about application behavior. We specifically adopt the threat models assumed by real world cryptographic libraries: applications may be vulnerable to (speculative) read gadgets, but those vulnerable to control flow hijacking (e.g., because of write gadgets) are out of scope as most cryptographic libraries assume the application is not *completely* under attacker control.[1]

1. Attacks via Memory Unsafety. The first class of assumptions, illustrated by the code in Figure 1, is that owing to memory unsafety, there exist *memory read gadgets* within the application. Zeroing buffers (like `kcopy`) is one key defense against such gadgets, but does nothing to prevent the attacker from reading the original version of the key `k` which remains unzeroed. To protect `k` from read gadgets, libraries like `Libsodium` offer memory protection APIs which must be manually inserted and toggled on and off by application developers, and hence, are prone to incorrect usage.

[1]Applications are also starting to employ software and hardware CFI techniques such as Clang's CFI [62] and Intel's CET [20].

2. Attacks via Speculation. If the application is written in a memory safe language like Rust, then the library developer need not fret about read gadgets, and can avoid the overhead of zeroing out secrets. However, the library developer must still contend with the spectre of hardware speculation and its attendant vulnerabilities [34, 40, 41, 44, 60, 69]. There is work on extending constant time protections to Spectre vulnerabilities, but it focuses on protecting cryptographic code *itself* from speculatively leaking secrets. Owing to the overheads imposed by such protections, it is currently unreasonable to apply the same protections to the entirety of application code. As such, library developers may have to contend with attacks based on Spectre vulnerabilities *in the application* [45].

Indeed, in the case of `stream`, speculation leads to a potential security issue with the clearing of `kcopy`: The `LibSodium` developers forgo an appropriate memory fence in `sodium_memzero`, leading to the possibility of `kcopy` being read by speculatively executing application code *before* it is zeroed [63].[2] The fence was eschewed due to its performance overhead: *all* clients of `Libsodium` would suffer the performance degradation for Spectre protections. By manually adding or changing protections, `Libsodium`'s developers implicitly restrict their defenses to certain classes of attackers: i.e. they assume that the only application vulnerabilities are *non-speculative* read gadgets.

3. Attacks via Concurrency. The last category of application assumptions that we consider is whether the application is *concurrent*. In a concurrent context, work like Spectre-Declassified [60] has shown that an attacker can recover secret information if they can observe intermediate results, thus enhancing the reach of (speculative) read gadgets. In fact, the possibility of such observations are cited by the `LibSodium` developers as a reason to forgo the memory fence in `sodium_memzero` [63].

2.2 Robust Constant Time

Unlike with the property of constant time, there does not exist a security property capturing when a cryptographic library is secure when running *in the context of* a potentially vulnerable application. To address this and to capture the different application assumptions that a cryptographic library developer needs to consider we introduce the notion of *robust (speculative) constant time* (RCT). Like other robustness properties [1, 28, 29, 51, 52, 58, 61], a cryptographic library being robustly constant time captures that the library does not leak secrets *when linked against a context (application).*

RCT serves as a formal security model for how cryptographic code is actually developed and used: protections are applied to the libraries with the goal that their use within an as yet unknown application will remain secure. Attacker capabilities can then be directly expressed as assumptions about the contexts in which the library code will run. For instance, `LibSodium`'s implicit protections can be explicitly specified as providing RCT assuming only the presence of single-threaded attacks based on memory unsafety (i.e. the presence of gadgets that can perform out-of-bounds reads).

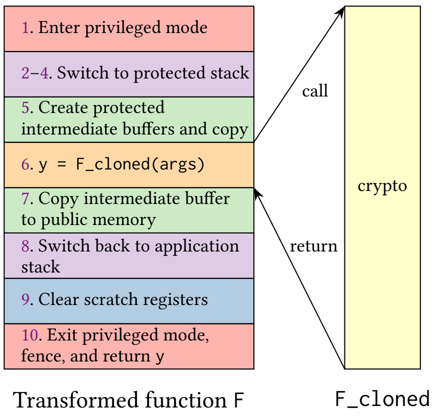
2.3 A Robust Constant Time Compiler

By factoring out security assumptions about the context, our notion of RCT lets us design and develop a compiler that takes as input: (1) an implementation of a cryptographic library[3] and (2) an explicit set of *assumptions* about application/attacker capabilities and then synthesizes a protected library that is guaranteed to be robustly constant time with respect to the given attacker. Library developers can then focus on implementing cryptographic algorithms and their library can be used securely and efficiently in a variety of contexts.

[2]Well-documented issues with zeroing functions in high-level code make the zeroing best effort regardless of speculation [53].
[3]Our compiler assumes that the library is already (speculatively) constant-time. In §4.2 we discuss how RoboCop's robust protections are orthogonal to existing (speculative) constant time protections, thus allowing library developers to use existing automated tools or manual technique to meet this assumption.

```
1  int stream(u8 *c, u64 clen, u8 *n, u8 *k) {
2    mpk_allow_access();
3    switch_to_protected_stack();
4    u8 *c_internal = mpk_malloc(clen);
5    int y = stream_cloned(c_internal, clen, n, k);
6    memcpy(c_internal, c, clen);
7    switch_to_unprotected_stack();
8    clear_scratch_registers();
9    mpk_disable_access();
10   fence;
11   return y;
12 }
```

(a) Protections applied to LibSodium's Salsa20.


1. Enter privileged mode
2–4. Switch to protected stack
5. Create protected intermediate buffers and copy
6. y = F_cloned(args)
7. Copy intermediate buffer to public memory
8. Switch back to application stack
9. Clear scratch registers
10. Exit privileged mode, fence, and return y
call
return
crypto
Transformed function F
F_cloned

(b) Wrapping of cryptographic function.

Fig. 2. RoboCop protections.

Bespoke Protection. Cryptographic libraries like `LibSodium` are designed to be used in a broad range of applications. As we saw with `libsodium_memzero`, the current state of manually baking protections into library code means that a single decision is made trading off between performance and which contexts the library is robust against. Instead, by factoring the protections out of the library and placing them in the compiler, RoboCop, developers can tailor protections to exactly the level required by the particular application context. For example, when used in a memory safe Rust application we can omit the unnecessary zeroing from `LibSodium`.

Efficient Protection via Wrapping and MPK. Our notion of RCT lets us use modern hardware memory protections, in particular Intel™ Memory Protection Keys (MPK), to protect library code incredibly efficiently. Our key insight in RoboCop is to allocate secret keys and carry out secret computation within a protected memory region. To do so, RoboCop wraps the library's external API functions, as illustrated by the protected version of `stream` in Figure 2a. We first enable access to the protected memory region and switch to a stack within protected memory. This protected stack ensures all intermediate computation remains protected. Line 4 shows the parameterized nature of RoboCop where we allocate a copy of the output buffer within protected memory. This allocation is *only* needed when (1) the underlying implementation uses the buffer for intermediate computation, and (2) the application context is *concurrent*. If these conditions are not met, RoboCop does not generate the extra allocation (the intermediate results *cannot* be observed by an attacker and hence, do not need to be protected). The wrapper function then calls the original library implementation, `stream_cloned`. After encryption, we copy the internal buffer back to the publicly visible `c` buffer (again, only if the attacker assumptions require.) The wrapper then switches back to the unprotected stack, clears scratch registers (if in speculative protection mode where these registers may contain secret values that could be speculatively read), and finally disables access to the protected memory region before returning to the application (additionally fencing if in speculative protection mode).

3 Security Semantics

To formally ground robust constant time and the attacker models we develop a high-level, stateful calculus, λ_{spec}, whose syntax is shown in Figure 3. Syntactically, λ_{spec} is relatively standard, following λ_{rust}, CompCert, and others [38, 43] in employing a block-based memory model: memory is structured as "disjoint", fixed width blocks addressed via a block label and offset ($z_b[z_o]$). New blocks are allocated with $\texttt{new}_p\ e$ with e the size of the block and the protection label p determines whether

$$
\begin{array}{rcll}
\text{labels} & \ell & ::= & \texttt{app} \mid \texttt{lib} \\
\text{protection} & p & ::= & \texttt{public} \mid \texttt{protected} \\
\text{values} & v & ::= & z \mid z[z] \\
\text{expressions} & e & ::= & v \mid x \mid x\{v\} \mid \mathsf{op}(\overline{e}) \mid e[e] \mid\ !\,e \mid \mathsf{new}_p\ e \mid e \coloneqq e \mid \texttt{protect}_p \mid e; e \\
 & & \mid & \texttt{get-block}\ e \mid \texttt{get-offset}\ e \mid e(\overline{e}) \mid \mathsf{fence} \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
\text{states} & S & : & p \times (\mathbb{Z} \rightharpoonup m) \\
\text{memory slots} & m & : & \{size : \mathbb{Z}, p : p, v : [size] \rightharpoonup v\} \mid \lambda_\ell \overline{x}.e \\
 & & & \\
\text{events} & \epsilon & ::= & \delta^\ell \mid \tau^{\ell \to \ell} \\
\text{transition events} & \tau & ::= & \mathsf{call}\ z_f \mid \mathsf{ret}\ v \mid \mathsf{begin} \mid \mathsf{end}\ v \\
\text{domain events} & \delta & ::= & \mu \mid \mathsf{call}\ z \mid \mathsf{ret}\ v \mid \mathsf{branch}\ v \mid \mathsf{fence} \mid 0 \\
\text{memory label} & b & ::= & \mathsf{ib} \mid \mathsf{oob} \\
\text{memory events} & \mu & ::= & \mathsf{new}_p\ z@z \mid \mathsf{read}_b\ v \hookleftarrow z[z] \mid \mathsf{write}_b\ v \mapsto z[z] \mid \mathsf{protect}_p \\
\text{observable events} & c & ::= & 0 \mid \mathsf{branch}\ v \mid \mathsf{read} \hookleftarrow z[z] \\
 & & \mid & \mathsf{write} \mapsto z[z] \mid \mathsf{new}_p\ z@z \mid \mathsf{call}\ z \mid \mathsf{end}\ v \\
 & & & \\
\text{speculation frame} & \Xi & : & \widehat{(S, \overline{K} :: e, \overline{\delta})} \mid (S, \overline{K} :: e, \overline{\delta}, \mu) \\
\text{speculative states} & \Phi & : & \{S : S, a : A, \Xi : \overline{\Xi}\} \\
\text{speculation directives} & d & ::= & \mathsf{nonspec} \mid \mathsf{spec}\ v \mid \mathsf{fence}
\end{array}
$$

Fig. 3. λ_{spec} syntax

the allocation is in a protected or unprotected memory "page". The size, set of values, and memory page are tracked in the second component of the non-speculative states (S). Correspondingly there is a protection operation, $\texttt{protect}_p$, which models hardware memory protection. $\texttt{protect}_p$ sets the memory access policy in the first component of the state: `public` only allows access to the public memory page whereas `protected` allows access to all memory. Dereferences are written $!\,e$ and assignments are written $e_{ptr} \coloneqq e_{val}$. Functions are also stored (immutably) in memory.

We equip λ_{spec} with these semantics: a non-speculative semantics capturing a trace of events that we use to define our attacker models (§3.1) a novel, high-level speculative semantics (§3.2), and (speculative) concurrent semantics capturing a passive observer. Throughout this work we use $\bullet$ for an empty list, $\diamond$ for list concatenation, and an overline (e.g. $\overline{e}$) for a list of elements.

3.1 Non-Speculative Trace Semantics

We first describe the structure of the traces the non-speculative semantics is designed to capture.

Traces. We are interested in capturing three aspects of the execution: (1) *who* (which party, `app` or `lib`) is executing code at a given time as well as the transfer of control between the parties, (2) *what memory* each party accesses, and (3) the internal branching which will be used to define the various constant time properties. To this end each reduction is labeled with an event, ϵ, whose syntax is shown in Figure 3. Labels are also attached to every function ($\lambda_\ell \overline{x}.e$) to distinguish application and library code. An event is either a labeled *domain event*, δ^ℓ, which captures an event executed by the party ℓ or a *transition event*, $\tau^{\ell \to \ell}$, which captures the transfer of control between the two parties.

The main transition events are call z_f and ret v which represent a call to the function at address z_f and returning from a function with return value v. Beyond the call and return events, there are begin and end v events that capture the (implicit) beginning and end of a trace. Domain events are either a *memory event* (μ), a call z_f and its corresponding ret v events (capturing a function call that stays within a single domain), a branch v event (capturing branching on the value v), or the empty event 0. Memory events are one of an allocation, a protection operation, a read, or a write

$$\boxed{\langle S \mid \overline{K^\ell} :: e^\ell \rangle \overset{\epsilon}{\Longrightarrow} \langle S \mid \overline{K^\ell} :: e^\ell \rangle}$$

RED-CALL

$$\frac{S(z) = \lambda_{\ell_f}\overline{x}.e \qquad \epsilon = \begin{cases}(\text{call } z)^\ell & \text{when } \ell_f = \ell \\ (\text{call } z)^{\ell\to\ell_f} & \text{otherwise}\end{cases}}{\langle S \mid \overline{K'^{\ell'}} :: K[z(\overline{v})]^\ell \rangle \overset{\epsilon}{\Longrightarrow} \langle S \mid \overline{K'^{\ell'}} :: K^\ell :: e\overline{[v/x]}^{\ell_f}\rangle}$$

RED-RET

$$\frac{\epsilon = \begin{cases}(\text{ret } v)^\ell & \text{when } \ell_K = \ell\\ (\text{ret } v)^{\ell\to\ell_K} & \text{otherwise}\end{cases}}{\langle S \mid \overline{K'^{\ell'}} :: K^{\ell_K} :: v^\ell\rangle \overset{\epsilon}{\Longrightarrow} \langle S \mid \overline{K'^{\ell'}} :: K[v]^{\ell_K}\rangle}$$

RED-β

$$\frac{\langle S \mid e\rangle \xrightarrow{\delta} \langle S' \mid e'\rangle}{\langle S \mid \overline{K'^{\ell'}} :: K[e]^\ell\rangle \overset{\delta^\ell}{\Longrightarrow} \langle S' \mid \overline{K'^{\ell'}} :: K[e']^\ell\rangle}$$

$$\boxed{\langle S \mid e\rangle \xrightarrow{\delta} \langle S \mid e\rangle}$$

β-DEREF

$$\frac{\begin{array}{c}\text{accessible}(S, z_b)\\ z_o \in [S(z_b).size)] \qquad v = S(z_b).v(z_o)\end{array}}{\langle S \mid !\,(z_b[z_o])\rangle \xrightarrow{\text{read}_{\text{ib}}\ v \mapsfrom z_b[z_o]} \langle S \mid v\rangle}$$

β-NEW

$$\frac{\begin{array}{c}z > 0 \qquad z_b = \text{fresh}(S) \qquad S.p \sqsubseteq p\\ S' = S[z_b := \{size = z, v = \bot, p = p\}]\end{array}}{\langle S \mid \text{new}_p\ z\rangle \xrightarrow{\text{new}_p\ z@z_b} \langle S' \mid z_b[0]\rangle}$$

β-DEREF-OOB

$$\frac{\begin{array}{c}z_b \notin \text{dom}(S) \lor z_o \notin [S(z_b).size] \qquad z'_b \in \text{dom}(S)\\ z'_o \in [S(z'_b).size)] \qquad v = S(z'_b).v(z'_o) \qquad \text{accessible}(S, z'_b)\end{array}}{\langle S \mid !\,(z_b[z_o])\rangle \xrightarrow{\text{read}_{\text{oob}}\ v \mapsfrom z'_b[z'_o]} \langle S \mid v\rangle}$$

β-SUBST

$$\frac{}{\langle S \mid x\{v\}\rangle \xrightarrow{0} \langle S \mid v\rangle}$$

Fig. 4. Non-speculative trace semantics excerpts

and track the data associated with each operation (e.g. the value read/written, the location it was read/written from/to, and whether the memory access was in bounds or out of bounds).

Transition Operational Semantics. Our non-speculative semantics are split between two labeled reduction judgments: top-level transition reductions and domain reductions (Figure 4). Transition reductions are of the form $\langle S \mid \overline{K^\ell} :: e^\ell \rangle \overset{\epsilon}{\Longrightarrow} \langle S \mid \overline{K^\ell} :: e^\ell \rangle$ where K are the standard call-by-value evaluation contexts. The state S tracks the memory and the current access level. To explain the control stack $\overline{K^\ell} :: e^\ell$ we examine the rule RED-CALL. The top of our stack (e^ℓ) is the mid-reduction body of the currently executing function (with label ℓ). For RED-CALL we are reducing a function call in the evaluation context K. We push this continuation (where the called function will return to) onto the stack of labeled continuations ($\overline{K'^{\ell'}}$) and then use the substituted body as the new execution frame. If the current label, ℓ, and the label of the function we are calling, ℓ_f, differ then we emit a transition event with label $\ell\to\ell_f$ otherwise we omit a domain event for the call.

Dually, rule RED-RET handles returning from a function. This is a transfer of control flow from ℓ, the label of the currently executing function, back to ℓ_K, the function caller. The return value is plugged into the top continuation on the stack and a corresponding return event is generated (we ignore same domain returns). The last "transition" reduction rule, RED-β, dispatches to the domain reduction relation ($\langle S \mid e\rangle \xrightarrow{\delta} \langle S \mid e\rangle$), and labels the domain event δ with the current label.

$$\boxed{\langle \Phi \mid \overline{K} :: e\rangle \overset{\overline{\delta}}{\lhook\joinrel\twoheadrightarrow} \langle \Phi \mid \overline{K} :: e\rangle}$$

SPEC-NONSPEC

$$\frac{(a', \text{nonspec}) = \mathit{spec}(\Phi.a, e) \qquad \langle \Phi \mid \overline{K} :: e\rangle \overset{\delta}{\hookrightarrow} \langle \Phi' \mid \overline{K'} :: e'\rangle}{\langle \Phi \mid \overline{K} :: e\rangle \overset{\delta}{\lhook\joinrel\twoheadrightarrow} \langle \Phi'[a := a'] \mid \overline{K'} :: e'\rangle}$$

SPEC-TRY-COMMIT

$$\frac{(a', \text{fence}) = \mathit{spec}(\Phi.a, e) \qquad \text{fence } \langle \Phi \mid \overline{K} :: e\rangle \text{ to } \langle \Phi' \mid \overline{K'} :: e'\rangle_{\overline{\delta}}}{\langle \Phi \mid \overline{K} :: e\rangle \overset{\overline{\delta}}{\lhook\joinrel\twoheadrightarrow} \langle \Phi'[a := a'] \mid \overline{K'} :: e'\rangle}$$

SPEC-SPEC

$$\frac{(a', \text{spec } v) = \mathit{spec}(\Phi.a, K[e]) \qquad \langle \Phi \mid \bullet :: e\rangle \overset{\overline{\delta}}{\hookrightarrow}{}^{*} \langle \Phi' \mid \bullet :: v'\rangle \qquad \overline{\Xi'} = \text{makeFrame}_{v=v'}(\Phi.S, \overline{K'} :: K[e], \overline{\delta}) :: \Phi.\Xi \qquad e \neq \texttt{fence}}{\langle \Phi \mid \overline{K'} :: K[e]\rangle \overset{0}{\lhook\joinrel\twoheadrightarrow} \langle \Phi[\Xi := \overline{\Xi'}, a := a'] \mid \overline{K'} :: K[v]\rangle}$$

$$\boxed{\langle \Phi \mid \overline{K} :: e\rangle \overset{\delta}{\hookrightarrow} \langle \Phi \mid \overline{K} :: e\rangle}$$

SPEC-β

$$\frac{\langle \Phi.S \mid \overline{K^{\ell}} :: e^{\ell}\rangle \overset{\epsilon}{\Rightarrow} \langle S' \mid \overline{K'^{\ell'}} :: e'^{\ell'}\rangle \qquad \neg\text{stalled}(\Phi.\Xi, \Phi.S, \text{unlabel}(\epsilon)) \qquad \overline{\Xi} = \text{addEvent}(\Phi.\Xi, \text{unlabel}(\epsilon))}{\langle \Phi \mid \overline{K} :: e\rangle \xhookrightarrow{\text{unlabel}(\epsilon)} \langle \Phi[S := S', \Xi := \overline{\Xi}] \mid \overline{K'} :: e'\rangle}$$

Fig. 5. Small step speculative semantics

Domain Operational Semantics. Most of the domain rules are standard and produce an empty trace event. Conditionals are also standard, but produce a branch event based on the condition. The rule β-NEW takes a protection domain p in which to allocate a new block of size z, checking that our current access level allows writing to the domain p using the "can-access" judgment $S.p \sqsubseteq p$.

The most notable of the reduction rules are those related to dereferencing and writing through pointers. The rules mirror each other so we will focus on dereferencing as it is simpler. In the rule β-DEREF we are dereferencing the pointer $z_b[z_o]$ with block label z_b and offset into that block z_o. To actually obtain the value at that location the following must hold: (1) the block must be accessible (accessible(S, z_b)), (2) the offset must be within the allocated size of the block ($z_o \in [S(z_b).\mathit{size}]$), and (3) a value must have been written to the block at the offset z_o ($v = S(z_b).v(z_o)$). When these conditions are met the value is read and a corresponding *in bounds* read trace event is generated. On the other hand, if either of the latter two conditions are not met the dereference is considered out-of-bounds and the rule β-DEREF-OOB applies instead. Here, we model the out-of-bounds read as a nondeterministic read from an arbitrary, valid, and accessible location $z'_o[z'_b]$.

3.2 Speculative Semantics

To model Spectre and speculative execution broadly we define a second operational semantics for λ_{spec}. Instead of directly modeling a specific version of Spectre we seek to capture a high-level essence of speculation, namely that speculation is the combination of guessing how an expression might evaluate and then either rolling back or committing if the guess was correct. That is, different types of speculation can be captured as the following sequence: First, instead of evaluating an expression, make up the value you think it would evaluate to and continue running using that value instead. While running "under speculation" check if any operation invalidates the speculative guess. Then, once you hit a "fence", rollback to the speculation point if the guess was invalid, or commit

$$\boxed{\mathsf{stalled}(\overline{\Xi}, S, \delta) : \overline{\Xi} \times S \times \delta \longrightarrow \mathbb{2}}$$

$$\begin{aligned}
\mathsf{stalled}(\bullet, S, \mathsf{fence}) &= \top \\
\mathsf{stalled}(\Xi :: \overline{\Xi}, S, \mathsf{read}_b\ v \hookleftarrow z_b[z_o]) &= (\mathsf{protect}_p \in \Xi.\overline{\delta} \diamond \Xi.\overline{\mu} \wedge S(z_b).p = \mathsf{protected}) \\
&\quad \vee (\mathsf{stalled}(\overline{\Xi}, S, \mathsf{read}_b\ v \hookleftarrow z_b[z_o]))
\end{aligned}$$

$$\boxed{\mathsf{addEvent}(\Xi, \delta) : \Xi \times \delta \longrightarrow \Xi}$$

$$\begin{aligned}
\mathsf{addEvent}((S,\ \overline{K} :: e,\ \overline{\delta},\ \overline{\mu}), \mathsf{read}_b\ v \hookleftarrow v_r) &= \widehat{(S,\ \overline{K} :: e,\ \overline{\delta} \diamond \overline{\mu})} \quad \text{when } v_r \in \mathsf{writeLocs}(\overline{\delta}) \\
\mathsf{addEvent}((S,\ \overline{K} :: e,\ \overline{\delta},\ \overline{\mu}), \mathsf{write}_b\ v \mapsto v_w) &= (S,\ \overline{K} :: e,\ \overline{\delta},\ \overline{\mu} \diamond \mathsf{write}_b\ v \mapsto v_w)
\end{aligned}$$

$$\boxed{\mathsf{fence}\ \langle \Phi \mid \overline{K} :: e \rangle\ \mathsf{to}\ \langle \Phi \mid \overline{K} :: e \rangle_{\overline{\delta}}}$$

FENCE-ROLLBACK

$$\frac{\begin{array}{c}\Phi.\Xi = \widehat{(S,\ \overline{K'} :: e',\ \overline{\delta})} :: \overline{\Xi} \\ \Phi' = \Phi[S := S, \Xi := \overline{\Xi}]\end{array}}{\mathsf{fence}\ \langle \Phi \mid \overline{K} :: e \rangle\ \mathsf{to}\ \langle \Phi' \mid \overline{K'} :: e' \rangle_{\bullet}}$$

FENCE-COMMIT

$$\frac{\begin{array}{cc}\Phi.\Xi = (S,\ \overline{K'} :: e',\ \overline{\delta},\ \overline{\mu}) :: \overline{\Xi} & S' = \mathsf{commit}(S, \overline{\delta} \diamond \overline{\mu}) \\ \overline{\Xi'} = \mathsf{addEvents}(\overline{\Xi}, \overline{\delta} \diamond \overline{\mu}) & \Phi' = \Phi[S := S', \Xi := \overline{\Xi'}]\end{array}}{\mathsf{fence}\ \langle \Phi \mid \overline{K} :: e \rangle\ \mathsf{to}\ \langle \Phi' \mid \overline{K} :: e \rangle_{\overline{\delta}}}$$

Fig. 6. Speculative semantics auxiliary definitions

the effects of the skipped expression and continue evaluating. In Appendix B we step through λ_{spec} models Spectre-PHT, -BTB, -STL, and (with choices of encoding)-RSB [21–23, 36, 40, 44].

We capture (excerpts of) these notions in Figures 5 and 6. Figure 5 defines the top-level reduction relation $\langle \Phi \mid e \rangle \overset{\overline{\delta}}{\hookrightarrow\!\!\!\!\rightarrow} \langle \Phi \mid e \rangle$. Speculative states, Φ, extend the non-speculative state (S) with a microarchitectural state (a) and a stack of speculation frames (Ξ). Speculation frames come in two forms, a "mispeculation" frame, $\widehat{(S,\ e,\ \overline{\delta})}$, capturing that the current speculation is invalid and will be rolled back to the state S and expression e and "in progress" frames, $(S,\ e,\ \overline{\delta},\ \overline{\mu})$, capturing that the current speculation is potentially valid. The $\overline{\delta}$ and $\overline{\mu}$ components capture the trace of events of the skipped expression and the subsequent memory events, respectively. The skipped events are used if we commit a speculative frame, "replaying" the events that were speculatively passed over, and the memory events are used to determine whether speculation is invalid (discussed below).

The decision of whether and how to speculate is handled by a microarchitectural state "speculation" function ($spec : A \times e \longrightarrow A \times d$). The function takes the current microarchitectural state and the current state of the executing function, updates the microarchitectural state, and returns a speculation directive, d. This directive says whether we will be speculating with the guessed value v (spec v), not speculating (nonspec), or performing a fence operation (fence).

Speculation. To see how speculation plays out we will go over the three corresponding rules: SPEC-NONSPEC, SPEC-SPEC, and SPEC-TRY-COMMIT and how they capture speculation in the Spectre-PHT bypassing of a bounds check when evaluating `if` $i\{100\} < 10$ `then` $!\,b[i\{100\}]$ `else` 0 (the size of the block b is 10). The term $i\{100\}$ captures a delayed substitution: earlier in the execution the value 100 was substituted for i.[4] We first evaluate $i\{100\}$ and apply the rule SPEC-NONSPEC: the speculator returns that it does not want to speculate on this expression and evaluation proceeds normally (via the SPEC-β rule), so our branch condition is now $100 < 10$. Here the speculator consults its microarchitectural state, sees that every other time we have done this check the result has been true, and thus returns spec 1 (and a new microarchitectural state) and the rule SPEC-SPEC applies. SPEC-SPEC runs the skipped expression, capturing any memory events (writes, reads, etc.) but does

[4]We use delayed substitutions (β-SUBST) to capture the fact that, when executing on hardware, argument substitution will be compiled to a register access or memory lookup and as such should not be treated as an immediate value.

not commit them, instead saving them in a new speculation frame via $\mathsf{makeFrame}$. $\mathsf{makeFrame}$ checks if the speculated value matches the real value: in this case it does not and therefore our new frame is a mispeculation frame saving the current state and continuation on our stack Ξ. Evaluation then continues with the speculated value 1 and another two SPEC-NONSPEC steps evaluate $!\,b[100]$.

In these SPEC-NONSPEC steps the rule SPEC-β handles two additional aspects beyond the evaluation. First, it checks that the instruction is not stalled ($\neg\mathsf{stalled}(\Phi.\Xi, \Phi.S, \mathsf{unlabel}(\epsilon))$).[5] This captures that fence instructions will not execute speculatively and that, with MPK, writes and reads to `protected` memory will not execute if the protection level is speculatively uncertain (the rules for fences and reads are shown in Figure 6). SPEC-β also adds the new event to the speculative stack ($\overline{\Xi} = \mathsf{addEvent}(\Phi.\Xi, \mathsf{unlabel}(\epsilon))$), updating whether the current speculation is invalid or not. Excerpts of $\mathsf{addEvent}$ are shown in Figure 6. For reads it checks if the read is to the location of a speculatively skipped write: if so the current speculation is invalid and will be rolled back (but continues executing until a fence). The displayed rule for writes shows how other events are added to an in progress frame to be used to check the validity of previous speculation.

Back in our example, were we to continue evaluating we would reach the classic Spectre out of bounds read, however we will instead assume that the speculator decides to stop speculating and returns the fence directive. The rule SPEC-FENCE thus applies and we turn to the judgment $\mathsf{fence}\ \langle \Phi \mid\ !\,b[100]\rangle$ to $\langle \Phi_1 \mid e_1\rangle_{\overline{\delta}}$. This judgment, defined in Figure 6, returns the state and expression with which we will continue evaluation as well as the trace of events from evaluating the speculatively skipped expression. If the speculation was invalidated (FENCE-ROLLBACK) then we will return to the continuation where speculation began (this applies to our example and we return to the saved continuation `if` $100 < 10$ `then` $!\,b[i\{100\}]$ `else` 0 and the respective state at the time of speculation). If the speculation had been valid then FENCE-COMMIT would apply. This commits any memory events that were speculatively skipped and checks whether the new, now committed events invalidate previous speculation (we allow nested speculation thus the speculative "stack").

3.3 Concurrent Observer Semantics

To define concurrent observer capabilities we layer another semantics on top of the speculative and non-speculative semantics, capturing the intuitive notion that a read-only attacker in a concurrent thread may, at *any* time, observe *any* visible memory location. We show the new judgment for the concurrent, speculative semantics below, consisting of a singular rule that, before any step, adds a read event for every memory location visible to a concurrent thread. As MPK guarantees thread local protection this consists of every location that is not in the `protected` memory region, even if our "main" thread currently has access to protected memory. The non-speculative version is defined similarly. This models the attacker making *all* possible concurrent observations.

$$\frac{\langle \Phi \mid \overline{K} :: e\rangle \overset{\overline{\delta}}{\hookrightarrow\!\!\!\!\twoheadrightarrow} \langle \Phi' \mid \overline{K'} :: e'\rangle \qquad \overline{\delta'} = [\mathsf{branch}\ v \mid \langle \Phi[S.p := \mathtt{public}] \mid\ !\,z_b[z_o]\rangle \overset{\mu}{\hookrightarrow} \langle \Phi' \mid v\rangle]}{\langle \Phi \mid \overline{K} :: e\rangle \xhookrightarrow{\overline{\delta'} \diamond \overline{\delta}}\!\!\!\!\twoheadrightarrow_C \langle \Phi' \mid \overline{K'} :: e'\rangle}$$

4 Robust Constant Time

Next we formalize the security semantics of cryptographic libraries when used within an application.

4.1 Programs and Traces

Figure 7 shows the syntax for defining libraries and applications.

[5]The $\mathsf{unlabel}$ function removes labels and is defined in Appendix A.

$$
\begin{array}{rrcl rrcl}
\text{API contexts} & \Gamma & ::= & \overline{(f : z)} & \text{app traces} & A & ::= & \tau^{\texttt{lib}\to\texttt{app}} \diamond \overline{\delta^{\texttt{app}}} \diamond \tau^{\texttt{app}\to\texttt{lib}} \\
\text{libraries} & L & ::= & \overline{f \mapsto (z_f, \lambda\overline{x}.e)} & \text{lib traces} & L & ::= & \tau^{\texttt{app}\to\texttt{lib}} \diamond \overline{\delta^{\texttt{lib}}} \diamond \tau^{\texttt{lib}\to\texttt{app}} \\
\text{secret contexts} & \Delta & ::= & \overline{x \mapsto (z_{loc}, z_{len})} & \text{traces} & T & ::= & A \circ L \circ T \\
\text{heaplets} & H & : & \mathbb{Z} \rightharpoonup m & & & | & \tau^{\texttt{lib}\to\texttt{app}} \diamond \overline{\delta^{\texttt{app}}} \diamond \mathsf{end}\ v^{\texttt{app}\to\texttt{lib}}
\end{array}
$$

Fig. 7. Syntax of programs and traces

Libraries. An *API context*, Γ is a map from function names to the number of arguments that function takes and defines the external API for a library. Given an API context Γ a *library* L is a set of functions and their heap locations for each of the external names in Γ. We capture this with the well-formedness judgment $\Gamma \vDash L$, which additionally allows L to contain internal functions (defined in Figure 17 in Appendix C).

Applications. We assume that secrets are stored in memory. A *secret context* Δ is a map from secret variable names x (used to refer to that secret block in application code) to a pair of integer values denoting the location (address) and length of the corresponding block. Given an API context Γ and a secret context Δ, an *application* is a pair of a *heaplet* H (a partial heap containing initial state and application functions), and a "main" expression e with free variables from Γ and Δ. We define a well-formedness judgment for applications $\Gamma, \Delta \vdash (H,\ e)$ (detailed in Figure 17 in Appendix C).

Programs. A *whole program* is then composed of a library L plugged into an application $(H,\ e)$. To define this we build a judgment $L \mid \Delta \mid H \vDash S$ (defined in Figure 17 in Appendix C). This captures that the initial state S (1) contains all of the functions defined in L, (2) contains locations for all of the secrets in Δ, and (3) contains the application heaplet H. We then define substitution operations $e[\Delta][L]$ and $S[\Delta][L]$ replacing the API variable names from Γ with the function locations in L and the secret variables in Δ with their actual block locations (defined in Figure 18 in Appendix C).

Program Traces. Our operational semantics captures a sequence of events, but for whole programs we factor this sequence into a *program trace* (T) with additional structure (shown in Figure 7). This trace captures the decomposition of execution into alternating sequences of application and library domain events, with transition events the boundaries between them. Application traces A are thus a transition event from library to application, followed by a sequence of application domain events, and then a transition back to the library. Library traces are defined similarly and the trace of an entire program is then alternating sequences of these application and library traces. For program traces we define a gluing concatenation operator $\overline{\epsilon_1}\epsilon \circ \epsilon\overline{\epsilon_2}$ which matches the sequence $\overline{\epsilon_1}\epsilon\overline{\epsilon_2}$. We use this to capture that the transition event ending an application trace and starting the subsequent library trace are in fact the same transition event. We then define trace and speculative trace metafunctions capturing the set of all program traces for a given program (the corresponding concurrent versions are as expected). These use the respective non-speculative and speculative termination judgments $\downarrow^{T}$ and $\downarrow^{\overline{\delta}}_{S}$ (which capture the terminating trace and are defined in Appendix C):

$$
\begin{array}{rcl}
\mathsf{traces}(\langle S \mid e\rangle) & \triangleq & \{\mathsf{begin} \diamond T \diamond \mathsf{end}\ v \mid \langle S \mid e\rangle \downarrow^{T} \langle S' \mid v\rangle\} \\
\mathsf{specTraces}(\langle \Phi \mid e\rangle) & \triangleq & \{\mathsf{begin} \diamond \overline{\delta} \diamond \mathsf{end}\ v \mid \langle \Phi \mid e\rangle \downarrow^{\overline{\delta}}_{S} \langle \Phi' \mid v\rangle\}
\end{array}
$$

4.2 Robust Constant Time

Our core security property, robust constant time, much like classic constant time, comes in two flavors: speculative and non-speculative (we also separate the associated concurrent versions).

Attacker Predicates. Intuitively, robust constant time captures that a library behaves correctly when plugged into an "unknown" context (in this case an application) representing an attacker. For cryptographic libraries the attacker attempts to exploit vulnerabilities in the application to extract secrets, so we parameterize our definition by an *attacker*: a predicate on applications,

$\text{pred}_{\Gamma,\Delta} : \wp(H,\ e)$ that captures a vulnerability as a set of applications with that vulnerability. We instantiate $\text{pred}_{\Gamma,\Delta}$ in Section 4.3 to capture read-only and memory-safe attackers.

Robust Constant Time. Robust constant time, then, is a parameterized, robust version of classic constant time definitions. Let $\text{ct} : \overline{e} \longrightarrow \overline{c}$ be a meta-function that erases the components of the trace that are not visible from timing-based attacks (defined by the syntactic class of observable events in Figure 3). Robust constant time can thus be defined as follows:

Definition 1 (Robust constant time). *We say a library* $\Gamma \vDash L$ *is* robustly constant time *for an attacker class* $\text{pred}_{\Gamma,\Delta}$ *if, for all secret contexts* Δ*, applications* $\Gamma, \Delta \vdash (H,\ e)$ *such that* $\text{pred}_{\Gamma,\Delta}(H,\ e)$*, and initial states* S_0, S_0' *such that* $L \mid \Delta \mid H \vDash S_0 = S_0'$ *we have that* $\text{ct}(\text{traces}(\langle S_0[\Delta][L] \mid e[\Delta][L]\rangle)) = \text{ct}(\text{traces}(\langle S_0'[\Delta][L] \mid e[\Delta][L]\rangle))$.

The judgment $L \mid \Delta \mid H \vDash S_0 = S_0'$ is a variant of our well-formedness judgment for initial states capturing that S_0 and S_0' *only* vary at the secret locations contained in Δ. Robust speculative constant time is defined similarly, however we consider the speculative traces and a speculative attacker oracle $\text{spec} : A \times e \longrightarrow A \times d$ (capturing varying speculative attacks). The concurrent versions simply use the set of concurrent traces. These definitions can be found in Appendix C.1.

4.3 Attackers

As discussed in Section 2.1 we wish to protect libraries against different kinds of application vulnerabilities: read-only vulnerabilities (corresponding to the attacker model libraries like `LibSodium` assume), speculative vulnerabilities, and concurrent observers. Speculative vulnerabilities and concurrent observers are captured by our speculative and concurrent semantics leaving us with non-speculative read-only attackers (which we contrast with memory-safe attackers such as those written in memory-safe languages). We will use the fact that our traces T capture the back and forth of actions of the application and library and the hand-offs between them to instantiate the predicate of Definition 1. We then define read-only and memory-safe attackers by restricting the set of unsafe behaviors during the application subtraces: Read-only attackers can read memory they were not given access to but cannot write to it (and thus cannot carry out *active* attacks). In contrast, memory-safe attackers may neither read nor write memory they were not given access to. We define both read-only and memory-safe attackers as excluding control-flow exploits.

We wish to capture sets of attackers by their trace properties, however the attacker predicate is defined as a property of applications which are only a partial program and cannot be run. As such we must first link with a library before we can assess the application's (mis)behavior. But we cannot simply take an arbitrary library: an ill-formed library can break application invariants. To untangle this knot we simultaneously define the relevant *restrictions* on the application we are classifying with the *assumptions* that it may make about the library that it is running against.

Definition 2 (Read-only attackers). *We say an application* $\Gamma, \Delta \vdash (H,\ e)$ *is a* read-only attacker *if, for all libraries* $\Gamma \vDash L$*, initial states* S_0 *such that* $L \mid \Delta \mid H \vDash S_0$*, and traces* $T \in \text{traces}(\langle S_0[\Delta][L] \mid e[\Delta][L]\rangle)$*,* $\Gamma \vdash \text{read-only}\ T$*.*

Figure 8 shows an excerpt of the judgment $\Gamma \vdash \text{read-only}\ T$ which handles the restrictions on the application and assumptions on the library components of the trace T. The rule read-only-rec captures the back and forth of assumptions and restrictions: it decomposes the next application and library trace sequences, imposes the read-only restrictions on the application events (wf-read-only $\overline{\delta_A}$), requires that the call into the library is an API function, and then, under the assumption that the library events do not write out of bounds, inductively requires that the rest of the trace is read-only. The definition of wf-read-only can be found in Figure 19 in Appendix C: it captures that the application trace can only contain in bounds write events, but may contain

$$\boxed{\Gamma \vdash \text{read-only } T}$$

READ-ONLY-REC

$$\frac{\begin{array}{c} A = \tau_1{}^{\texttt{lib}\to\texttt{app}} \diamond \overline{\delta_A}^{\texttt{app}} \diamond \tau_2{}^{\texttt{app}\to\texttt{lib}} \qquad L = \tau_2{}^{\texttt{app}\to\texttt{lib}} \diamond \overline{\delta_L}^{\texttt{lib}} \diamond \tau_3{}^{\texttt{lib}\to\texttt{app}} \\ \tau_2 = \text{call } z_f \Rightarrow z_f \in \text{dom}(\Gamma) \qquad \text{wf-read-only } \overline{\delta_A} \qquad \text{wf-read-only } \overline{\delta_L} \implies \Gamma \vdash \text{read-only } T \end{array}}{\Gamma \vdash \text{read-only } A \circ L \circ T}$$

Fig. 8. Excerpt of read-only attacker model definition

arbitrary read events. Memory-safe attackers are defined similarly, with the predicate adjusted to also preclude reads from unexposed blocks.

5 A Robust Compiler

Guided by our formal models we develop RoboCop, a compiler providing robust constant time protections for cryptographic libraries against different attackers. RoboCop is built on top of the LLVM framework [42] and uses Intel™ Memory Protection Keys (MPK) to guarantee that secret data (cryptographic secrets and the data derived from them) is only accessible while executing trusted cryptographic library code. While RoboCop mainly operates on the library, there are two components that involve the application. First, cryptographic secrets often originate and are managed by application code, and it is therefore necessary to allocate these secrets in protected memory. To do so we provide manual MPK allocation APIs and also adapt techniques from CryptoMPK [37] to provide an alternative, automatic application transformation that securely allocates secrets. Further, for the efficiency of our library protections, we reuse a single stack allocated in protected memory. To enable this we develop a simple LLVM pass that allocates this stack on program entry.

5.1 Making Libraries Robust

Cryptographic code operates directly on secret data and, to prevent timing-based leaks, is required to be constant time. With this baseline security requirement we operate under the assumption that cryptographic code is *trusted*. The task of RoboCop then is to ensure that the secret data remains inaccessible even if there are vulnerabilities within the client application code. These protections are provided in three steps: (1) Cryptographic developers label the external API functions. (2) RoboCop wraps these API functions to handle the memory isolation. (3) RoboCop replaces all dynamic memory functions (`malloc` and similar) with custom MPK compatible versions, ensuring that all memory allocated within the cryptographic library is kept within the protected domain.

Figure 2b shows our wrapping of cryptographic API functions. For every exported function `F` in the library, a clone, `F_cloned`, is generated containing the original implementation of `F`. *Internal* calls to `F` are replaced with calls to `F_cloned`: `F` becomes the external API wrapper for use by the client. `F` is responsible for switching into and out of the protected memory region.

The new `F` takes the following domain switching steps: (1) We enable access to the protected memory region with a `wrpkru` instruction. The specification for MPK [24] ensures this is speculatively secure: `wrpkru` will not execute speculatively and protected memory cannot be accessed until `wrpkru` is committed. (2) We get the address of the protected stack and save the current stack pointer. (3) We copy any stack arguments from the unprotected frame to the new protected stack. (4) We switch the stack pointer to the protected stack frame. (5) If concurrent protections are enabled, we allocate an internal copy buffer for the external buffers (discussed below). (6) We call `F_cloned`. (7) After the cryptographic function returns, we copy any internal buffers back to the original, public buffers. (8) We restore the previous stack pointer. (9) We clear all scratch registers which may potentially contain transient secret computation. (10) We disable access to protected memory, fence if speculative protections are enabled, and return to the application. Together these ensure that all data produced and used by the cryptographic code is within the protected memory region and the region is inaccessible to application code.

Concurrent Protections. Rather than allocate extra memory, cryptographic algorithms sometimes carry out intermediate computations within output (e.g. ciphertext) buffers. In a single-threaded context this is safe: there is no way for client code to access these buffers before they contain their final, cryptographically secure value. In a concurrent context, work like Spectre Declassified [60] has shown that an attacker can recover secret information by observing intermediate results. To defend against these attacks we add a concurrent protection option to RoboCop. Here library developers annotate API arguments that are used for intermediate computation, and RoboCop allocates a buffer in protected memory for the arguments, performs the intermediate work within the protected domain, and then copies the declassified result back out to the unprotected memory.

5.2 Proving RoboCop Secure

On its own (concurrent) robust (speculative) constant time serves as a useful security property for understanding when library protections are secure against different attackers. However we are interested in automatically providing robust protections via RoboCop. To do so we need to understand our "source language": the assumptions we make about the source of our compilation. For RoboCop we assume that the source is *classically* (speculative) constant time and will prove that RoboCop then guarantees robust (speculative) constant time. This gives library developers flexibility: they can safely use any tool or handwritten technique to guarantee that their library implementation is constant time and then RoboCop lifts this to the *robust* counterpart.

In defining a "source language" that captures classic constant time, note that the classic notion of constant time is that executing a library function produces invariant traces. As such classic constant time is in fact a restricted form of robust constant time, where, for a given API context Γ, the main function is defined by the grammar $e_\Gamma ::= f(\overline{v})$ for $f \in \Gamma$. That is, "source applications" are simply individual function calls into the library. There is a slight caveat: classical constant time also requires that secrets have not leaked into the application state. With this in mind we define classical constant time as the following variation on robust constant time:

Definition 3 (Classical constant time). *We say a library* $\Gamma \vDash L$ *is* classically constant time *if, for all secret contexts* Δ*, classical "applications"* $\Gamma, \Delta \vdash (H, e_\Gamma)$*, and initial states* S_0, S_0' *such that* $L \mid \Delta \mid H \vDash S_0 = S_0'$*, we have that for all traces* $\langle S_0[\Delta][L] \mid e_\Gamma[\Delta][L]\rangle \downarrow^{\overline{\delta}} \langle S_1 \mid v\rangle$ *there exists a trace* $\langle S_0'[\Delta][L] \mid e_\Gamma[\Delta][L]\rangle \downarrow^{\overline{\delta'}} \langle S_1' \mid v\rangle$ *such that* $\mathrm{ct}(\overline{\delta}) = \mathrm{ct}(\overline{\delta'})$ *and* $S_1(\mathsf{dom}(H)) = S_1'(\mathsf{dom}(H))$.

The speculative version is defined similarly and can be found in Appendix C.1. Our compiler correctness properties are then that the compiler guarantees (speculative) (concurrent) robust constant under the assumption that the library is (speculatively) classically constant time.[6]

A Formal Model of RoboCop. Formally, we represent RoboCop as a parameterized compiler $\mathbb{C}$ which transforms source libraries ($\Gamma \vDash L$) into protected targets L. We have four compilers: (1) $\mathbb{C}_{\text{ro}}$, which protects libraries from read-only attackers, (2) $\mathbb{C}_{\text{spec}}$ which protects against speculative attackers, and (3) $\mathbb{C}_{\text{ro-co}}$, and (4) $\mathbb{C}_{\text{spec-co}}$ which also protect against concurrent observers. $\mathbb{C}_{\text{ro}}$ transforms all internal uses of $\mathtt{new}_p\ e$ into $\mathtt{new}_{\mathtt{protected}}\ e$, and wraps each external API function with $\mathtt{protect}_{\mathtt{protected}}$ and $\mathtt{protect}_{\mathtt{public}}$. $\mathbb{C}_{\text{spec}}$ is the same as $\mathbb{C}_{\text{ro}}$ but also inserts a fence before returns. The concurrent compilers additionally reallocate all external buffers used by the library within protected memory, and insert memory copying to and from the external and protected buffers.

[6]This property can be equivalently viewed as preservation of robust constant time from our source language of classical "applications" to the target language containing the contexts of our attacker model.

To capture that the application manages the secret buffers and must allocate them in protected memory we slightly modify the initial state well-formedness judgment to capture that each secret block in Δ is allocated in the `protected` memory region. We additionally assume that applications and libraries do not contain any $\texttt{protect}_p$ expressions prior to being run through our compiler. We prove that each compiler is secure and guarantees its corresponding robust constant time property. The theorem statement that $\mathbb{C}_{\text{ro}}$ guarantees read-only robust constant time is shown below (the remaining theorems are in Appendix E):

Theorem 1 ($\mathbb{C}_{\text{ro}}$ guarantees read-only robust constant time). *If $\Gamma \vDash L$ is classically constant time and does not contain any $\texttt{protect}_p$ subterms, then $\mathbb{C}_{\text{ro}}(\Gamma \vDash L)$ is robustly constant time for read-only attackers (that do not contain $\texttt{protect}_p$).*

Proof Sketches. To prove Theorem 1 we must show that, for *any* read-only attackers and any two initial states that vary only in the values of secrets, when we plug our compiled library into the application the execution under both states does not vary (up to the observable leakage). The difficulty arises in two places: firstly, we must reason about a (mostly) arbitrary application. Secondly, the inherent nondeterminism of our semantics (due to allocation and out of bounds memory semantics), means that we are dealing with sets of traces rather than just a single trace.

To handle these challenges we rely on the fact that our attacker model is defined as properties on the structured traces T. Our proof begins with the two initial states S and S' with the same whole program e and some trace T for $\langle S \mid e\rangle$. We must then show that there is a corresponding trace for $\langle S' \mid e\rangle$. To do so we inductively split the *read-only* trace T into its attacker and library subsequences and define a semantic interpretation that captures that there is a corresponding trace T' for $\langle S' \mid e\rangle$. The interpretation enforces the following invariant during application subtraces: (1) the access level in both states is `public`, (2) all memory that varies between the two states (contains secrets) is within protected memory, and (3) the same memory locations have been allocated in both states. These conditions and our assumption that our application is read-only allow us to prove that, for all of the application subsequences A of T, there is a T' that contains the *exact same* subsequences A. The first two conditions ensure that the memory that the application can access (even if it reads out of bounds) does not contain secrets and is therefore identical, and the third condition aligns the non-determinism between the two traces. For each non-deterministic choice in one application subtrace we can then always show that we can make the exact same non-deterministic choice in our other trace. Our interpretation of library subtraces requires that there exists a corresponding library trace with the same observable events, which follows from assuming the library is classically constant time. Put together these let us prove Theorem 1 (details are in Appendix E.1).

Our proof of the corresponding theorem for $\mathbb{C}_{\text{spec}}$ uses a similar technique of semantically interpreting traces as relations between the two states. Here we split the speculative traces into subsequences of non-speculative events (where there was no speculation), mispeculated events (where the speculation was rolled back), and "correctly speculated" events (where the speculative guess was eventually committed). We then show that the non-speculative and "correctly" speculated events are related to an underlying non-speculative trace and thus constant time by the proof of Theorem 1. To show the latter, we rely on the inserted `fence` instruction ensuring that we cannot speculatively return into the application. For mispeculated events we rely on the invariant that the application cannot change the protection level and a lemma that speculative execution cannot read protected memory unless the access level was changed non-speculatively (details are in Appendix E.2). The proof requires strengthening the statement of classical speculative constant time in two ways, both of which could go overlooked in a classical setting when not considering running cryptographic code in an application context. Firstly, we require that the library be speculatively constant time under an arbitrary initial microarchitectural state. This ensures that the protections take into account any speculative (mis)behavior from the application before the handoff to the library.[7] Secondly, we strengthen the classical statement to add the assumption that the library is speculatively constant time *under an extended function context*, i.e. in the presence of unprotected application functions. This requires that the speculative protections account for *cross-domain* speculation attacks, for example, speculatively jumping to unprotected application code.

The corresponding proofs for concurrent observers strengthen the invariant to capture that unprotected buffers never contain secrets, even during library subtraces. Beyond this the proofs follow the same (non-)speculative reasoning as for $\mathbb{C}_{\mathsf{ro}}$ and $\mathbb{C}_{\mathsf{spec}}$.

6 Evaluation

To evaluate the cost of guaranteeing robust constant time we ask the following questions:

Q1: What is the overhead of *robust* constant time against read-only attackers? (§6.1)

Q2: What is the overhead of *robust* constant time against speculative attackers? (§6.1)

Q3: What is the overhead of *robust* constant time against concurrent observers? (§6.2)

Benchmarks. To study the performance of RoboCop on a wide range of cryptographic code we modify the SUPERCOP [65] cryptographic benchmarking tool. SUPERCOP's benchmarking suite is broken down into operations: we focus on its collections of implementations of authenticated encryption (**aead**), Diffie-Hellman key exchange (**dh**), public key encryption (**encrypt**), key encapsulation mechanism (**kem**), public key signatures (**sign**), and stream cipher (**stream**) algorithms. Within each of these operations SUPERCOP collects multiple implementations of each algorithm: e.g. the **stream** data set contains several implementations of both Salsa20 and ChaCha20. The original design intent of SUPERCOP is then to find the fastest each cryptographic algorithm can run on a given machine. To this end its benchmarking tries every implementation of an algorithm with every compiler in its test set. These implementations are benchmarked multiple times across a range of input data sizes, recording data for the fastest implementation-compiler pair.

RoboCop is implemented as a set of LLVM passes, so we restrict the compiler test set to Clang with the `-O3`, `-O2`, `-Os`, and `-O` flags. We manually remove all non-C/C++ implementations, as RoboCop does not handle assembly or Rust code. Beyond this manual removal, SUPERCOP automatically prunes implementations that fail to compile or run, e.g. implementations that rely on non-standard behavior of the GCC compiler and do not compile with Clang. The findings presented below contain all data remaining after the manual removal of non-C/C++ implementations and SUPERCOP's automatic pruning, all of which occurred before data analysis.

With its broad suite of algorithms and its find-the-fastest methodology, SUPERCOP is a more *robust* means of benchmarking cryptographic software security techniques and we encourage future designers to use it for benchmarking. We found its selecting from multiple compilation levels particularly beneficial as, due to the cascading effects of optimizations, comparing at the same optimization level is not truly a head-to-head comparison. Indeed we found that sometimes the fastest optimization level would differ between RoboCop and the baseline, with several instances of `-O3` producing significantly slower code in combination with the protections than `-O`.

Machine and Software Setup. We run all benchmarks on a 13th Gen Intel® Core™ i9-13900KS, with 125GB RAM, and running Linux kernel version 6.3.0. We run SUPERCOP configured to collect data only on cores with the same frequency and our data is collected from 5.6 GHz cores. RoboCop adds new passes to LLVM 16.0.2 and is split into two passes: the library pass adds the protections and the application pass allocates a stack in protected memory on program entry. SUPERCOP defines API functions for each operation: these are annotated as the external API for RoboCop.

[7]We discuss an example where this was overlooked in an existing tool in Section 8.

Table 1. Overheads for read-only and speculative protections vs. unprotected baseline. N is the number of algorithms in the dataset, Size is the size of the operation's input in bytes, Q_1 and Q_3 are the first and third quartile overheads, and the median overhead is of the overheads of the mean runtimes.

			READ-ONLY			SPECULATIVE		
Benchmark	N	Size	Q_1	**Median overhead**	Q_3	Q_1	**Median overhead**	Q_3
aead	385	2048	-0.26%	**0.16%**	0.65%	-0.15%	**0.15%**	0.71%
dh	9	fixed	-0.11%	**0.32%**	0.56%	0.36%	**0.46%**	1.50%
encrypt	15	4237	-0.57%	**-0.09%**	0.41%	-3.61%	**0.48%**	3.38%
kem	47	fixed	-0.59%	**0.04%**	0.55%	-0.82%	**0.00%**	0.49%
sign	40	4237	-1.00%	**-0.11%**	0.67%	-1.00%	**-0.22%**	0.97%
stream	11	4096	0.24%	**0.88%**	1.45%	0.51%	**0.92%**	1.90%

We manually label secret key buffers and insert protected allocations for them in the protected versions. For the concurrent protection benchmarks we label the ciphertext arguments on the **dh** and **stream** APIs as being used for internal computation. The speculative protections differ from the read-only protections solely in the insertion of a single fence before returning from the library. We use a modified version of jemalloc 5.2.0 [37] patched to provide MPK allocation functions. Our baseline replaces the libc malloc implementation with an unpatched version of jemalloc.

Summary of Results. We find that robust constant time protections can generally be guaranteed with minimal overhead (less than 5% in almost all cases for both read-only and speculative protections), with a small number of outliers with a peak of 40% overhead. At small data sizes highly optimized stream ciphers also carry a large overhead (with a median around 33% for read-only and 37% for speculative protections), however these workloads take on the order of a few hundred cycles. We also find that concurrent protections add additional overhead but the resulting costs remain minimal (the majority have overheads under 6%).

6.1 Read-Only and Speculative Attackers

We measure the cost of ensuring robust constant time against read-only and speculative attackers across six data sets (shown in Table 1 and Figure 9). For the largest data size for each benchmark we find that the median overhead across all benchmarked algorithms is below 1% for both read-only and speculative protections and that 75% of all implementations for each primitive have overheads under 2% for read-only protections and 4% for speculative protections. For algorithms with varying input lengths, SUPERCOP measures performance across a wide range of lengths. We show the varying overheads across these sizes for stream ciphers in Table 2. The overhead increases as data sizes get smaller, with a median overhead of 33% for encrypting a single byte with read-only protections and 37% for speculative protections. Fortunately, at this data size encryption only takes a few hundred cycles so this high relative overhead remains a minimal raw cost.

Outliers. For 75% of implementations, read-only protections have overheads below 2% and speculative protections below 4%, there are some outliers above 10%: one **kem** implementation has a read-only overhead of 17% and a speculative overhead of 26%; two **sign** implementations have read-only overheads of 15% and 16% and speculative overheads of 16% and 35%; and four **aead** implementations have read-only overheads between 26% and 37% and speculative overheads between 28% and 42%. The two **sign** outliers are based on the learning with errors problem [3] and exhibit high runtime variance in the baseline that includes the measured overhead of RoboCop. For the remaining outliers, the only consistent measure we observe in the performance statistics (as measured using the perf tool) are an increase on the order of 10% in page faults. We hypothesize the observed speedups are due to (1) noise as the baseline number of cycles is often small and (2) the

[8]For clarity: 21/385 read-only and 23/385 speculative **aead** implementations lie above the whiskers and below the cutoff.

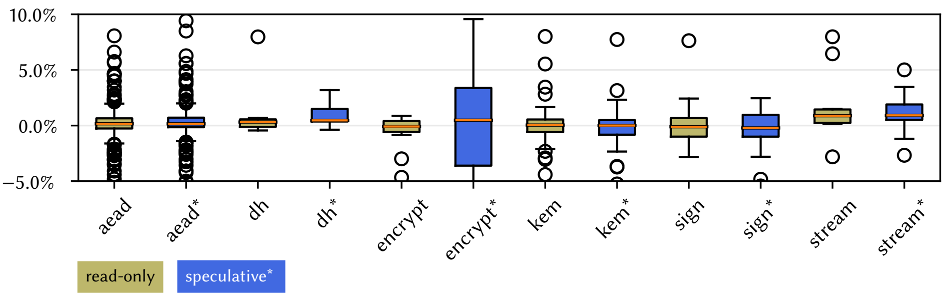


Fig. 9. Box plot of overheads for read-only and speculative protections* compared to unprotected baseline. Cutoff at 10% overhead, outliers beyond this point are discussed directly.[8]

Table 2. Read-only and speculative protection overheads for stream ciphers with varying plaintext sizes. Baseline cycles is the mean number of cycles for the unprotected baseline.

	READ-ONLY			SPECULATIVE			
Size	Q_1	**Median overhead**	Q_3	Q_1	**Median overhead**	Q_3	Baseline cycles
1	29.07%	**33.40%**	55.56%	27.63%	**37.49%**	55.43%	4.29e+02
128	15.08%	**21.66%**	24.78%	15.63%	**21.25%**	25.35%	7.69e+02
256	8.23%	**12.01%**	17.52%	9.50%	**13.48%**	15.79%	1.44e+03
512	5.04%	**6.71%**	9.88%	5.03%	**6.80%**	8.81%	2.11e+03
1024	3.39%	**5.27%**	7.28%	3.48%	**4.94%**	6.60%	4.11e+03
2048	1.54%	**2.03%**	2.73%	1.60%	**2.45%**	3.82%	7.72e+03
4096	0.24%	**0.88%**	1.45%	0.51%	**0.92%**	1.90%	1.54e+04

separate protected allocator. We've previously observed better locality properties with separate library allocators.

6.2 Concurrent Attackers

To protect against concurrent attackers it is necessary for buffers containing intermediate values derived from secrets to remain within the MPK protected memory. In its read-only attacker protections mode RoboCop ensures all memory originating from cryptographic code meets this requirement, however some cryptographic implementations use external, public buffers as internal, intermediate (private) buffers. To measure the cost of protecting these intermediate buffers we benchmark the **dh** and **stream** data sets with RoboCop's concurrent protections mode. We annotate the top-level SUPERCOP API functions `crypto_dh` and `crypto_stream` to mark the ciphertext argument as being used for intermediate computation. In the case of **stream** the size of this buffer is dynamically determined so we mark the plaintext length argument as the size for allocating a secure intermediate buffer. Table 3 shows the median overheads for the read-only protections compared to the median overheads for the concurrent protections. For our **dh** data set protecting these intermediate buffers increases the median overhead from 0.32% to 0.74% for read-only attackers and from 0.46% to 0.69% for speculative attackers. For our stream cipher data set at the largest data size the increase is greater, with the median increasing from 0.88% to 2.39% for read-only attacker and 0.92% to 2.14% for speculative attackers. We hypothesize that the higher overhead for stream ciphers is due to the dynamic allocation whereas the statically sized buffer of the Diffie-Hellman API allows optimizations in both allocation and copying back to the external buffer.

Our benchmarking treats every algorithm within each data set as if they use public buffers for intermediate computations. In practice RoboCop lets library designers label specifically which/if buffers are used for intermediate computations. This ensures that code that does not use the external

Table 3. Overhead of read-only and speculative protections vs. overhead with concurrent protections.

(a) Diffie-Hellman key exchange (**dh**) algorithms

	NON-CONCURRENT			CONCURRENT		
Base protections	Q_1	**Median overhead**	Q_3	Q_1	**Median overhead**	Q_3
read-only	-0.11%	**0.32%**	0.56%	0.24%	**0.74%**	1.26%
speculative	0.36%	**0.46%**	1.50%	0.10%	**0.69%**	1.07%

(b) Stream (**stream**) ciphers

		NON-CONCURRENT			CONCURRENT		
Base protections	Size	Q_1	**Median overhead**	Q_3	Q_1	**Median overhead**	Q_3
read-only	4096	0.24%	**0.88%**	1.45%	1.77%	**2.39%**	5.67%
speculative	4096	0.51%	**0.92%**	1.90%	1.61%	**2.14%**	2.60%

buffers never has to pay the price for the extra allocation and copying and that the same library code base can be used in all contexts: in single-threaded mode RoboCop can omit the allocation and copy but in concurrent mode it handles the creation of a protected intermediate buffer.

7 Limitations

Our implementation of RoboCop has a few limitations: (1) While RoboCop handles observers in concurrent threads it does not handle running concurrent *cryptographic library* code. To handle this we could allocate a single protected stack per thread. It would be safe to reuse a single MPK protection key across all threads as concurrent threads would only have access while running cryptographic library code. (2) RoboCop assumes that no other code is using MPK. (3) RoboCop does not handle ensuring that protected memory does not get dumped on crashes, written to disk, or that it is cleared at the end of execution. These could be handled in one place by operating on MPK protected pages. (4) RoboCop assumes the speculative behavior of MPK follows Intel's specification and `wrpkru` will never execute speculatively and protected memory will never be accessed speculatively until protections have been fully committed [24]. Hardware bugs such as meltdown-pk [14] violate these assumptions, and we rely on hardware fixes for such bugs (for instance official patches have already been provided for the machine used for our evaluation). (5) RoboCop currently only works on C/C++ code. While we believe compilers are an effective means of both separating security concerns and de-duplicating implementation work, it should be possible for library authors to implement parameterized RCT protections via preprocessor macros, templates, traits, etc. We encourage cryptographic library developers to use our attacker models to ground their protections and to offer more efficient, parameterized protections moving forward.

8 Related Work

Memory Isolation. MPK has been used to provide in-process memory isolation [35, 64], including between safe and unsafe Rust code [33, 39, 56]. MPK protections can be modified by unprivileged instructions, as such RoboCop assumes that applications do not have access to these instructions and including through control-flow hijacks. Countermeasures like binary rewriting [64], system call filtering [59, 67], and Cfi schemes [12, 13] could be used to lift these assumptions.

Similar to RoboCop, CryptoMPK [37] leverages MPK to protect the confidentiality of secret cryptographic data from memory disclosure vulnerabilities. We believe that it can guarantee robust constant time against read-only attackers, though their work is not framed in this manner. CryptoMPK differs from RoboCop in several fundamental ways: First, it is instead a whole-program analysis and transformation, identifying "crypto buffers" throughout the program and toggling MPK protection around use sites. As such it inserts significantly more context switches than RoboCop, and dynamically allocates and frees secret stack buffers. RoboCop's trusted library model avoids these costs, allowing a single context switch on library entry and exit, completely avoiding the need to dynamically allocate stack buffers and leading to significant performance gains. Second, by avoiding a whole program analysis RoboCop's model allows a much simpler implementation. This allows RoboCop to be applied to more complicated code and even hard to analyze assembly code (though we have not implemented this). Lastly, CryptoMPK does not handle robust speculative protections nor robust protections against concurrent attackers. As an optimization, CryptoMPK chooses not to securely allocate small secret stack buffers and instead inserts zeroing code. This zeroing code has the same trade offs between protecting against Spectre and performance as in `LibSodium`, and the buffers are also visible to concurrent attackers. CryptoMPK provides a `mxor` annotation to ensure that eventually declassified buffers are not marked as tainted when they are mixed with secret key data. This has the result of exposing these buffers to concurrent attackers.

Secure zeroization is often deployed as a manual protection in cryptographic libraries. Unfortunately, implementing secure memory zeroing in a high-level language is essentially impossible [53, 57, 70]. Recent work [49] shows how to develop a compiler pass to implement secure zeroization. RoboCop avoids these issues by restricting all secret data to a protected memory region, thus avoiding the need for zeroization (apart from register zeroing).

(Speculative) Constant Time. Many domain-specific languages and compilers have been developed to produce high-assurance cryptographic code [4, 7, 17, 55, 68]. Spectre attacks [40] significantly reduced the guarantees of these tools and prompted defenses against speculative leaks [16]. Jasmin implements Selective Speculative Load Hardening to protect against Spectre-PHT [60], performing stack zeroization and register clearing [49]. Blade inserts a minimum number of protections to prevent leaks via Spectre-PHT gadgets [66]. Swivel hardens WebAssembly sandboxes against speculative sandbox breakout and sandbox poisoning attacks [48, 71]. Serberus mitigates all currently known Spectre variants in code that follows the *static* constant-time discipline, which additionally prohibits secret function arguments and return values [47]. These tools serve as complements to RoboCop: in combination they can be used to guarantee end-to-end protections against speculative attackers as discussed in §4.2. Notably, there is an exception to this statement in the case of Serberus: In handling Spectre-RSB [41], Serberus makes an implicit assumption that the return stack buffer is empty when entering cryptographic code, arising from an assumption that the cryptographic code represents the entire program. This can be remedied by RSB filling on entry to cryptographic code.

Foundations for Cryptographic Software Security. Researchers have developed trace-based leakage models to reason about timing leaks in cryptographic code [9, 46]. These models have then been extended with prediction oracles [32], speculative semantics, and directives [15, 18, 31] to capture leaks via (combinations of) different Spectre gadgets [27]. Cauligi et al. [16] survey a number of these semantic models. Of the medium and high-level semantics they survey [8, 19, 26, 52, 54, 66], all model Spectre-PHT [40], while only Barthe et al. [8] and Ponce-de Leon and Kinder [54] model Spectre-STL [36]. None model Spectre-BTB [40] or Spectre-RSB [44]: λ_{spec} answers the call to capture additional Spectre variants in higher-level models. Our speculative semantics also serve as the first model of the speculative behavior of MPK, outside of the description in Intel's manual [24]. Guanciale et al. [30], present a low-level, microarchitectural, speculative semantics based on speculatively predicting the value of microinstructions: λ_{spec} can be viewed as a lifting of this idea to a high-level language. Our notion of RCT is inspired by previous work on secure compilers [2], which are formally defined as compilers that preserve classes of (hyper)-properties in adversarial contexts. Patrignani and Guarnieri [52] develop secure robust compilation criteria to formally examine the security guarantees of protections inserted by major compilers against Spectre-PHT, our approach to a secure compiler property follows this line of work.

Acknowledgments

We thank the reviewers for their suggestions and insightful comments. This material is based upon work supported by the National Science Foundation under Grant Nos. 2120642, 2120696, 2154964, 2155235, and 2327336; and by gifts from Mozilla, Intel, and the Google V8 team. Thanks to the Italian Ministry of Education through funding for the Rita Levi Montalcini grant (call of 2019). Funded by the European Union - NextGenerationEU, M4 C2 Inv. 1.1 - PRIN 2022 PNRR Prot. n. P2022EPPHM, CUP: E53D23016320001. Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or European Commission. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] Martín Abadi. 1999. Secrecy by Typing in Security Protocols. *J. ACM* 46, 5 (Sept. 1999), 749–786. https://doi.org/10.1145/324133.324266

[2] Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, and Jeremy Thibault. 2019. Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)* (2019-06). IEEE, 256–25615. https://doi.org/10.1109/CSF.2019.00025

[3] Erdem Alkim, Paulo S. L. M. Barreto, Nina Bindel, Juliane Krämer, Patrick Longa, and Jefferson E. Ricardini. 2020. The Lattice-Based Digital Signature Scheme qTESLA. In *Applied Cryptography and Network Security*, Mauro Conti, Jianying Zhou, Emiliano Casalicchio, and Angelo Spognardi (Eds.). Springer International Publishing, 441–460. https://doi.org/10.1007/978-3-030-57808-4_22

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

[5] OpenSSL Project Authors. 2023. *OpenSSL*. OpenSSL Project. https://www.openssl.org/

[6] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. 2021. SoK: Computer-Aided Cryptography. In *2021 IEEE Symposium on Security and Privacy (SP)* (2021-05). IEEE, 777–795. https://doi.org/10.1109/SP40001.2021.00008 ISSN: 2375-1207.

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

[8] Gilles Barthe, Sunjay Cauligi, Benjamin Grégoire, Adrien Koutsos, Kevin Liao, Tiago Oliveira, Swarn Priya, Tamara Rezk, and Peter Schwabe. 2021. High-Assurance Cryptography in the Spectre Era. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1884–1901. https://doi.org/10.1109/SP40001.2021.00046 ISSN: 2375-1207.

[9] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. 2018. Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time". In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)* (2018-07). IEEE, 328–343. https://doi.org/10.1109/CSF.2018.00031 ISSN: 2374-8303.

[10] Daniel J. Bernstein. 2005. *Cache-timing attacks on AES*. Technical Report. The University of Illinois at Chicago. https://api.semanticscholar.org/CorpusID:2217245

[11] Daniel J. Bernstein. 2008. The Salsa20 Family of Stream Ciphers. In *New Stream Cipher Designs: The eSTREAM Finalists (Lecture Notes in Computer Science)*, Matthew Robshaw and Olivier Billet (Eds.). Springer, 84–97. https://doi.org/10.1007/978-3-540-68351-3_8

[12] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-Flow Integrity: Precision, Security, and Performance. *Comput. Surveys* 50 (April 2017), 16:1–16:33. https://doi.org/10.1145/3054924

[13] Nathan Burow, Xinping Zhang, and Mathias Payer. 2019. SoK: Shining Light on Shadow Stacks. In *2019 IEEE Symposium on Security and Privacy (SP)* (2019-05). IEEE, 985–999. https://doi.org/10.1109/SP.2019.00076 ISSN: 2375-1207.

[14] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. USENIX Association, 249–266. https://www.usenix.org/conference/usenixsecurity19/presentation/canella

[15] Sunjay Cauligi, Craig Disselkoen, Klaus v. Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2020. Constant-time foundations for the new spectre era. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (2020-06-11) *(PLDI 2020)*. Association for Computing Machinery, 913–926. https://doi.org/10.1145/3385412.3385970

[16] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. 2022. SoK: Practical Foundations for Software Spectre Defenses. In *2022 IEEE Symposium on Security and Privacy (SP)* (2022-05). IEEE, 666–680. https://doi.org/10.1109/SP46214.2022.9833707 ISSN: 2375-1207.

[17] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Gregoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. 2019. FaCT: A DSL for timing-sensitive computation. In *Programming Language Design and Implementation (PLDI)*. ACM SIGPLAN. https://doi.org/10.1145/3314221.3314605

[18] Kevin Cheang, Cameron Rasmussen, Sanjit Seshia, and Pramod Subramanyan. 2019. A Formal Approach to Secure Speculation. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. IEEE, 288–28815. https://doi.org/10.1109/CSF.2019.00027

[19] Robert J. Colvin and Kirsten Winter. 2020. An Abstract Semantics of Speculative Execution for Reasoning About Security Vulnerabilities. In *Formal Methods. FM 2019 International Workshops*, Emil Sekerinski, Nelma Moreira, José N. Oliveira, Daniel Ratiu, Riccardo Guidotti, Marie Farrell, Matt Luckcuck, Diego Marmsoler, José Campos, Troy Astarte, Laure Gonnord, Antonio Cerone, Luis Couto, Brijesh Dongol, Martin Kutrib, Pedro Monteiro, and David Delmas (Eds.). Springer International Publishing, 323–341. https://doi.org/10.1007/978-3-030-54997-8_21

[20] Intel Corporation. 2018. *Complex Shadow-Stack Updates, Intel® Control-flow Enforcement Technology Specification.* Technical Report. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf

[21] Intel Corporation. 2018. CVE-2017-5715. Available from NIST NVD, CVE-ID CVE-2017-5715. https://nvd.nist.gov/vuln/detail/CVE-2017-5715

[22] Intel Corporation. 2018. CVE-2017-5753. Available from NIST NVD, CVE-ID CVE-2017-5753. https://nvd.nist.gov/vuln/detail/CVE-2017-5753

[23] Intel Corporation. 2018. CVE-2018-3639. Available from NIST NVD, CVE-ID CVE-2018-3639. https://nvd.nist.gov/vuln/detail/CVE-2018-3639

[24] Intel Corporation. 2023. *Intel® 64 and IA-32 Architectures Software Developer's Manual.* Technical Report. https://www.intel.com/content/www/us/en/content-details/774476/intel-64-and-ia-32-architectures-software-developer-s-manual-volume-1-basic-architecture.html

[25] Frank Denis. 2023. *libsodium.* libsodium. https://doc.libsodium.org/

[26] Craig Disselkoen, Radha Jagadeesan, Alan Jeffrey, and James Riely. 2019. The Code That Never Ran: Modeling Attacks on Speculative Evaluation. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1238–1255. https://doi.org/10.1109/SP.2019.00047 ISSN: 2375-1207.

[27] Xaver Fabian, Marco Guarnieri, and Marco Patrignani. 2022. Automatic Detection of Speculative Execution Combinations. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*. Association for Computing Machinery, New York, NY, USA, 965–978. https://doi.org/10.1145/3548606.3560555

[28] Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. 2007. A Type Discipline for Authorization Policies. *ACM Trans. Program. Lang. Syst.* 29, 5, Article 25 (Aug. 2007). https://doi.org/10.1145/1275497.1275500

[29] Andrew D. Gordon and Alan Jeffrey. 2003. Authenticity by Typing for Security Protocols. *J. Comput. Secur.* 11, 4 (July 2003), 451–519. http://dl.acm.org/citation.cfm?id=959088.959090

[30] Roberto Guanciale, Musard Balliu, and Mads Dam. 2020. InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security.* ACM, 1853–1869. https://doi.org/10.1145/3372297.3417246

[31] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2021. Hardware-Software Contracts for Secure Speculation. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021.* IEEE, 1868–1883. https://doi.org/10.1109/SP40001.2021.00036

[32] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. 2020. Spectector: Principled Detection of Speculative Information Flows. In *2020 IEEE Symposium on Security and Privacy (SP)* (2020-05). IEEE, 1–19. https://doi.org/10.1109/SP40000.2020.00011 ISSN: 2375-1207.

[33] Merve Gülmez, Thomas Nyman, Christoph Baumann, and Jan Tobias Mühlberg. 2023. Friend or Foe Inside? Exploring In-Process Isolation to Maintain Memory Safety for Unsafe Rust. *arXiv preprint arXiv:2306.08127* (2023).

[34] Enes Göktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. 2020. Speculative Probing: Hacking Blind in the Spectre Era. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (2020-11-02) *(CCS '20)*. Association for Computing Machinery, 1871–1885. https://doi.org/10.1145/3372297.3417289

[35] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-process isolation for high-throughput data plane libraries. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019.* USENIX Association.

[36] Jann Horn. 2018. *Speculative execution, variant 4: speculative store bypass.* https://project-zero.issues.chromium.org/issues/42450580

[37] Xuancheng Jin, Xuangan Xiao, Songlin Jia, Wang Gao, Hang Zhang, Dawu Gu, Siqi Ma, Zhiyun Qian, and Juanru Li. 2021. Annotating, Tracking, and Protecting Cryptographic Secrets with CryptoMPK. IEEE Computer Society, 473–488. https://doi.org/10.1109/SP46214.2022.9833650 ISSN: 2375-1207.

[38] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.* 2, POPL (2017). https://doi.org/10.1145/3158154

[39] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. 2022. PKRU-safe: automatically locking down the heap between safe and unsafe languages. In *Proceedings of the Seventeenth European Conference on Computer Systems*. Association for Computing Machinery, New York, NY, USA, 132–148. https://doi.org/10.1145/3492321.3519582

[40] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy (SP)* (2019-05). IEEE, 1–19. https://doi.org/10.1109/SP.2019.00002 ISSN: 2375-1207.

[41] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. USENIX Association. https://www.usenix.org/conference/woot18/presentation/koruyeh

[42] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004)*. IEEE Computer Society, 75–88. https://doi.org/10.1109/CGO.2004.1281665

[43] Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363–446. https://doi.org/10.1007/s10817-009-9155-4

[44] Giorgi Maisuradze and Christian Rossow. 2018. ret2spec: Speculative Execution Using Return Stack Buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018-10-15) *(CCS '18)*. Association for Computing Machinery, 2109–2122. https://doi.org/10.1145/3243734.3243761

[45] Andrea Mambretti, Alexandra Sandulescu, Alessandro Sorniotti, William Robertson, Engin Kirda, and Anil Kurmus. 2021. Bypassing memory safety mechanisms through speculative control flow hijacks. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)* (2021-09). IEEE, 633–649. https://doi.org/10.1109/EuroSP51992.2021.00048

[46] David Molnar, Matt Piotrowski, David Schultz, and David A. Wagner. 2005. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *Information Security and Cryptology - ICISC 2005 (Lecture Notes in Computer Science, Vol. 3935)*, Dongho Won and Seungjoo Kim (Eds.). Springer, 156–168. https://doi.org/10.1007/11734727_14

[47] Nicholas Mosier, Hamed Nemati, John C. Mitchell, and Caroline Trippel. 2023. Serberus: Protecting Cryptographic Code from Spectres at Compile-Time. *ArXiv* abs/2309.05174 (2023). https://api.semanticscholar.org/CorpusID:261682113

[48] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, and Deian Stefan. 2021. Swivel: Hardening WebAssembly against Spectre. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 1433–1450. https://www.usenix.org/conference/usenixsecurity21/presentation/narayan

[49] Santiago Arranz Olmos, Gilles Barthe, Ruben Gonzalez, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Lechenet, Tiago Oliveira, and Peter Schwabe. 2023. High-assurance zeroization. Cryptology ePrint Archive, Paper 2023/1713. https://eprint.iacr.org/2023/1713

[50] Marco Patrignani, Amal Ahmed, and Dave Clarke. 2019. Formal Approaches to Secure Compilation: A Survey of Fully Abstract Compilation and Related Work. *ACM Comput. Surv.* 51, 6, Article 125 (feb 2019), 36 pages. https://doi.org/10.1145/3280984

[51] Marco Patrignani and Sam Blackshear. 2023. Robust Safety for Move. In *2023 IEEE 36th Computer Security Foundations Symposium (CSF)*. IEEE Computer Society, Los Alamitos, CA, USA, 308–323. https://doi.org/10.1109/CSF57540.2023.00045

[52] Marco Patrignani and Marco Guarnieri. 2021. Exorcising Spectres with Secure Compilers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer Communications Security (CCS '21)*. Association for Computing Machinery, New York, NY, USA, 445–461. https://doi.org/10.1145/3460120.3484534

[53] Colin Percival. 2014. *Zeroing buffers is insufficient.* https://www.daemonology.net/blog/2014-09-06-zeroing-buffers-is-insufficient.html

[54] Hernán Ponce-de Leon and Johannes Kinder. 2022. Cats vs. Spectre: An Axiomatic Approach to Modeling Speculative Execution Attacks. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 235–248. https://doi.org/10.1109/SP46214.2022.9833774 ISSN: 2375-1207.

[55] Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017. Verified Low-Level Programming Embedded in F*. *Proc. ACM Program. Lang.* 1, ICFP, Article 17 (aug 2017), 29 pages.

https://doi.org/10.1145/3110261

[56] Elijah Rivera, Samuel Mergendahl, Howard E. Shrobe, Hamed Okhravi, and Nathan Burow. 2021. Keeping Safe Rust Safe with Galeed. In *ACSAC '21: Annual Computer Security Applications Conference*. ACM, 824–836. https://doi.org/10.1145/3485832.3485903

[57] RustCrypto. 2023. *Zeroize*. RustCrypto. https://docs.rs/zeroize/1.7.0/zeroize/

[58] Michael Sammler, Deepak Garg, Derek Dreyer, and Tadeusz Litak. 2020. The high-level benefits of low-level sandboxing. *PACMPL* 4, POPL (2020), 32:1–32:32. https://doi.org/10.1145/3371100

[59] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. 2022. Jenny: Securing Syscalls for PKU-based Memory Isolation Systems. In *31st USENIX Security Symposium, USENIX Security 2022*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 936–952. https://www.usenix.org/conference/usenixsecurity22/presentation/schrammel

[60] Basavesh Ammanaghatta Shivakumar, Jack Barnes, Gilles Barthe, Sunjay Cauligi, Chitchanok Chuengsatiansup, Daniel Genkin, Sioli O'Connell, Peter Schwabe, Rui Qi Sim, and Yuval Yarom. 2023. Spectre Declassified: Reading from the Right Place at the Wrong Time. In *2023 IEEE Symposium on Security and Privacy (SP)* (2023-05). IEEE, 1753–1770. https://doi.org/10.1109/SP46215.2023.10179355 ISSN: 2375-1207.

[61] David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and compositional verification of object capability patterns. *Proc. ACM Program. Lang.* 1, OOPSLA, 89:1–89:26. https://doi.org/10.1145/3133913

[62] The LLVM Foundation. 2021. *Control Flow Integrity, Clang 12 documentation*. https://clang.llvm.org/docs/ControlFlowIntegrity.html

[63] Reini Urban. 2019. *libsodium_memzero with memory barrier · Issue #802 · jedisct1/libsodium*. https://github.com/jedisct1/libsodium/issues/802

[64] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *28th USENIX Security Symposium, USENIX Security 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 1221–1238. https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner

[65] VAMPIRE. 2021. *SUPERCOP: System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives*. https://bench.cr.yp.to/supercop.html

[66] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kıcı, Ranjit Jhala, Dean Tullsen, and Deian Stefan. 2021. Automatically eliminating speculative leaks from cryptographic code with blade. *Proc. ACM Program. Lang.* 5 (2021), 49:1–49:30. Issue POPL. https://doi.org/10.1145/3434330

[67] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. 2022. You shall not (by)pass!: practical, secure, and fast PKU-based sandboxing. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*, Yérom-David Bromberg, Anne-Marie Kermarrec, and Christos Kozyrakis (Eds.). ACM, 266–282. https://doi.org/10.1145/3492321.3519560

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

[69] Johannes Wikner and Kaveh Razavi. 2022. RETBLEED: Arbitrary Speculative Code Execution with Return Instructions. USENIX Association, 3825–3842. https://www.usenix.org/conference/usenixsecurity22/presentation/wikner

[70] Zhaomo Yang, Brian Johannesmeyer, Anders Trier Olesen, Sorin Lerner, and Kirill Levchenko. 2017. Dead Store Elimination (Still) Considered Harmful. USENIX Association, 1025–1040. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/yang

[71] Hosein Yavarzadeh, Mohammadkazem Taram, Shravan Narayan, Deian Stefan, and Dean M. Tullsen. 2023. Half&Half: Demystifying Intel's Directional Branch Predictors for Fast, Secure Partitioned Execution. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 1220–1237. https://doi.org/10.1109/SP46215.2023.10179415