

Protecting Cryptographic Code Against Spectre-RSB

(and, in Fact, All Known Spectre Variants)

Santiago Arranz Olmos

MPI-SP
Bochum, Germany
santiago.arranz-olmos@mpi-sp.org

Gilles Barthe

MPI-SP
Bochum, Germany
IMDEA Software Institute
Madrid, Spain
gilles.barthe@mpi-sp.org

Chitchanok Chuengsatiansup

University of Melbourne
Melbourne, Australia
c.chuengsatiansup@unimelb.edu.au

Benjamin Grégoire

Inria
Sophia Antipolis, France
benjamin.gregoire@inria.fr

Vincent Laporte

Université de Lorraine, CNRS, Inria,
LORIA
F-54000 Nancy, France
vincent.laporte@inria.fr

Tiago Oliveira

SanboxAQ
Palo Alto, USA
tiago.oliveira@sandboxquantum.com

Peter Schwabe

MPI-SP
Bochum, Germany
Radboud University
Nijmegen, The Netherlands
peter@cryptojedi.org

Yuval Yarom

Ruhr University Bochum
Bochum, Germany
yuval.yarom@rub.de

Zhiyuan Zhang*

MPI-SP
Bochum, Germany
zhiyuan.zhang@mpi-sp.org

Abstract

Spectre attacks void the guarantees of constant-time cryptographic code by leaking secrets during speculative execution. Recent research shows that such code can be protected from Spectre-v1 attacks with minimal overhead, but leaves open the question of protecting against other Spectre variants.

In this work, we design, validate, implement, and verify a new approach to protect cryptographic code against all known classes of Spectre attacks, in particular Spectre-RSB. Our approach combines a new value-dependent information-flow type system that ensures that no secrets leak even under speculative execution and a compiler transformation that enables it on the generated low-level code.

We first prove the soundness of the type system and the correctness of the compiler transformation using the Coq proof assistant. We then implement our approach in the Jasmin framework for high-assurance cryptography and demonstrate that the overhead incurred by all Spectre protections is below 2% for most cryptographic primitives and reaches only about 5–7% for the more complex post-quantum key-encapsulation mechanism Kyber.

*Part of this work was done while the author was affiliated with the University of Melbourne.



This work is licensed under a [Creative Commons Attribution 4.0 International License](https://creativecommons.org/licenses/by/4.0/).

CCS Concepts: • Security and privacy → Formal security models; Logic and verification.

Keywords: Spectre-RSB, Type system, Compiler, Speculative execution, Coq

1 Introduction

In this paper, we present techniques to systematically protect high-performance cryptographic software against all known classes of Spectre attacks. Our work draws from the computer-aided cryptography paradigm [11], i.e., we employ methodologies and tools from formal methods to build efficient and formally verified cryptographic software. More specifically, we build our solution as part of Jasmin [1, 4], a programming language and framework that has been used to produce highly optimized and machine-checked implementations of symmetric cryptography [4], elliptic-curve cryptography [1], hash functions [5] and recently also post-quantum cryptography [3]. The Jasmin compiler is proven in Coq to preserve semantics, and offers tools to ensure properties relating to implementation security such as memory safety, thread safety, and absence of branches and memory accesses dependent on secrets. It furthermore offers an interface to the EasyCrypt [23] proof assistant to prove functional correctness of implementations, that can be further connected to computer-verified reductionist cryptographic proofs of security, as, for instance, in [6, 12, 13].

An S&P 2023 paper [9] made an important first step toward integrating systematic protections against Spectre attacks into Jasmin. In short, that paper proposes a type system to ensure that typable programs are protected against Spectre-v1 attacks, i.e., attacks exploiting misspeculated conditional branches. Furthermore, it presents extensions to the Jasmin language to protect programs (and thus make them typable). These protections mostly consist of selectively employing speculative load hardening (SLH) [20] as proposed in [38], incurring a remarkably small overhead of (typically) below 1%.

However, as that paper addresses only one class of Spectre attacks, the conclusion hopes that the work “*will be a starting point to upgrade the gold standard of constant-time cryptography, and will help deliver new post-quantum implementations that are not only protected against attacks by future large quantum computers, but also against the most common classes of speculative attacks.*”

Contributions. In this paper, we first show that Jasmin programs are very easily, even naturally, protected against most other classes of Spectre attacks. We identify one major remaining challenge in protecting against *all* known classes of Spectre attacks: Spectre-RSB [29, 31], i.e., attacks exploiting the return stack buffer.

We then present the main contribution of our paper: efficient and systematic protections against Spectre-RSB and their integration into the Jasmin framework. Our solution is a hybrid approach that combines selective speculative load hardening (selSLH) [9] with program transformation. First, we propose language constructs to enforce selSLH on a programming language with calls and returns. Second, we propose a program transformation that replaces calls and returns by conditional *direct* jumps, which we call *return-table insertion*. This transformation, which is inspired from prior work on return-oriented programming (ROP) [35], removes all Spectre-RSB gadgets. The combination of program transformation and selective speculative load hardening guarantees that transformed programs do not leak secrets through timing even during speculative execution. Prior work refers to this property as *speculative constant-time* [22]. Compared to previous work, we complete this property by considering all known Spectre attacks. We provide a precise definition of speculative constant-time in [Section 5](#).

The next step is to check that programs are correctly instrumented. For this purpose, we define an information-flow type system for *source* programs in the spirit of the approach taken for Spectre-v1 in [9]. Being cognizant that source programs will be transformed, the type system can check that program instrumentation will be sufficient to track misspeculation and guide the programmer to systematically protect the code against speculative leakage. We use the Coq proof assistant to formalize our approach for a core language: we define the source language and its speculative operational

semantics, the return-table insertion, and the type system. Our main result is a proof that the compilation of a well-typed program is speculative constant-time.

Next, we integrate our approach into the Jasmin framework. We extend the Jasmin language with an annotation for function calls and we modify the existing speculative constant-time type system from [9] so that inserting return tables in well-typed programs yields programs that leak no secrets even under speculation. Our implementation addresses practical issues omitted by our core language in the setting of a full-blown programming language. Last, we extend the Jasmin compiler with a new return-table insertion pass; we consider different variants, allowing for instance return addresses to be stored in MMX registers or on the stack.

Finally, we perform an in-depth evaluation of the impact of our approach on cryptographic software. The evaluation is carried out on a set of Jasmin implementations of cryptographic algorithms from libjade,¹ that had already been protected against Spectre-v1 in [9]. We use these routines to measure the overhead of our approach, both in terms of programmer effort and performance overhead. We show that the overhead for full Spectre protections is below 2% for most primitives and reaches only about 5–7% for the more complex post-quantum scheme Kyber [17].

Artifacts. We produce three artifacts: the Coq formalization, a new version of the Jasmin framework, and the protected version of libjade. These artifacts are available from <https://doi.org/10.5281/zenodo.14773254>. The items that can be found in the Coq formalization are marked with . In this work, we make several simplifications and restrictions for clarity of presentation—which we point out in the text—that are not present in our Coq development or Jasmin.

Supplementary Material. We provide formal definitions of the semantics and compilation scheme, proof statements for soundness and correctness of the approach, and an illustrative example of the proof method in the appendices of the extended version of this paper <https://eprint.iacr.org/2024/1070>.

2 Background

In this section, we first introduce the constant-time (CT) paradigm and how Spectre attacks challenge it. Then, we introduce how Jasmin enforces the CT paradigm and tackles three out of four variants of Spectre attacks.

CT Paradigm. The CT paradigm requires that no secrets flow into memory addresses or branch conditions under sequential execution. It is widely regarded as a standard baseline defense mechanism against timing attacks [27], as it defeats most traditional timing attacks, where the attacker

¹See <https://github.com/formosa-crypto/libjade>.

knows the trace of all accessed memory locations and the complete control flow of the program.

Spectre Attacks. The CT paradigm protects cryptographic code under the assumption that instructions are executed sequentially. However, the performance of modern CPUs relies heavily on speculative execution, which predicts the instructions to be executed next and *speculatively* executes them. Speculative execution is problematic for protecting cryptographic code from timing attacks, as an attacker could force a misprediction and speculatively forward data from secrets to memory indexes or branch conditions. Attacks that exploit speculative execution are known as Spectre attacks [28], and have four general variants:

- Spectre-v1 [28, Sec. IV] attacks mistrain the conditional branch predictor and capitalize on speculative execution following a mispredicted conditional jump.
- Spectre-v2 [28, Sec. V] manipulates the prediction of indirect branches to speculatively jump to almost anywhere in the victim memory space.
- Spectre-v4 exploits load instructions that target speculative data. This can happen when a load operation reads from an address that has unresolved store operations [28, Sec. VI], or because the CPU wrongly predicted a store-to-load forward [7].
- Spectre-RSB [29, 31, 39] attacks exploit the RSB (Return Stack Buffer) to mispredict the address of return instructions. Similar to Spectre-v2 attacks, an attacker could speculatively jump to almost anywhere in the victim’s memory space.

Jasmin. In Jasmin, the constant-time paradigm is enforced through an information-flow type system at source level. In short, all variables are typed as either *secret* or *public* and the type system enforces that operations taking *secret* inputs produce *secret* outputs. Memory addresses and branch conditions have to be *public*. Compiler preservation of the constant-time property has been formally proven in [14].

Since the Jasmin language does not support indirect branches, Jasmin programs are naturally protected from Spectre-v2 attacks. Furthermore, Spectre-v4 attacks are prevented by setting the SSBD (Speculative Store Bypass Disable) flag on Intel and AMD processors. As we will see in Section 9, the performance impact of setting this flag on cryptographic code is very small.

To protect against Spectre-v1 attacks, recent work [9] proposes to extend the current type system by adding an additional security level, *transient*, for variables that are always *public* in sequential execution but may contain *secret* data in speculative execution after a mispredicted branch. Variables typed as *transient* are not allowed to influence memory addresses or branch conditions, but they can be lowered to

public through one of two mechanisms: terminating speculative execution by inserting an *lfence* instruction; or masking the variable with a misspeculation flag that is updated through arithmetic instructions at each branch. This second technique is selective speculative load hardening [38]. It is supported in Jasmin through three instructions:

- `init_msf()` inserts an *lfence* instruction and sets a special register *msf* to the neutral value of masking *NOMASK*. This register is used to track speculation and we call it the *misspeculation flag* (MSF).²
- `update_msf(e)` conditionally updates the misspeculation flag *msf* to *MASK*, depending on the boolean expression *e*; it is essentially $msf = e ? msf : MASK$, implemented as a conditional move instruction (e.g., `CMOV` in x86) that takes the value of the expression *e*, updates the *msf* atomically, and does not speculate. We insert this instruction after branches to compare with the prediction.
- `x = protect(y)` protects register *y* conditioned on the value of *msf*. Specifically, if the value of *msf* is *NOMASK*, register *x* receives the value of *y*, but if it is *MASK*, it gets the default value of the masking *MASK*. This instruction is used to lower the type of *y* from *transient* to *public*.

3 Threat Model

We assume the OS is fully patched with the latest microcode updates, and sets the SSBD flag to disable Spectre-v4 attacks.

Since we aim to protect cryptographic code against all Spectre attacks, we assume a strong attacker who cannot compromise the OS, but can perform all known variants of Spectre attacks. That is, the attacker can manipulate the prediction of conditional branches [28], indirect branches [28], and return instructions [29]. We remark that this attacker is stronger than in previous work [9], which focuses only on Spectre-v1 attacks and thus assumes an attacker who can only manipulate the prediction of conditional branches.

In terms of the attacker’s ability to monitor the victim program, we share the assumptions of previous work. Specifically, we assume the attacker can observe the addresses of all memory accesses and control-flow under sequential execution, which aligns with the CT paradigm [2]. Furthermore, we assume the attacker also has the same ability under speculative execution. This assumption is consistent with existing work on protecting cryptographic code against Spectre-v1 [9].

4 Design Overview

Our countermeasure against Spectre-RSB comprises two ingredients: first, the developer instruments the program with

²We will, for simplicity of presentation, assume that *msf* is a distinguished variable that does not occur in the program. This restriction is unnecessary and not present in our Coq development or the Jasmin language.

protections resembling the ones for Spectre-v1 [9], and then the compiler replaces function calls with direct jumps and returns with return tables, implemented as nested conditional direct jumps.

To understand the instrumentation expected from programmers, let us first consider the transformation: Figure 1a shows a source program, and Figure 1b shows its compilation. The transformed program is trivially protected against Spectre-RSB attacks, as there are no RET instructions. However, after the transformation, the program is vulnerable to Spectre-v1 attacks because the branch in `id` might be mispredicted, and speculative execution after the second invocation of `id` might proceed from the first call site and thus leak `sec`. Nevertheless, the transformation greatly reduces the attack surface of a program that would otherwise have a RET instruction: it ensures that speculative execution can no longer be directed to an *arbitrary* location; instead, it can only be directed to a well-defined, known set of possible locations: the set of all call sites of the function we are returning from. The idea of compiling RET instructions to a sequence of direct conditional jumps is not new; it was mentioned in [15, Sec. 7] as a potential Spectre-RSB countermeasure, but not implemented. Figure 1c shows our program with selSLH protections, protecting it against all attacks in our threat model. For a high-level description of selSLH and its implementation in Jasmin, see Section 2.

We combine these two ingredients in a way that integrates well with the Jasmin workflow and is, in spirit, similar to the type system presented in [9]. We perform security typing at the source level, i.e., *with* function returns. Afterwards, during compilation, we apply the transform from RET instructions to nested conditional jumps. This means that the speculative semantics and the type system from [9] are insufficient to capture all effects of speculative execution.

Therefore, the structure of this work is as follows. First, we define a language featuring calls and returns in addition to conditional statements and loops, together with a speculative semantics that captures at the source level the protections offered by return tables (Section 5). We then present a type system that enforces speculative constant-time under these semantics (Section 6) and a compilation scheme that realizes them and preserves leakage (Section 7). Finally, Section 8 discusses how we implement this type system and compilation scheme in Jasmin, and Section 9 how we use that to protect libjade with little overhead.

5 Language

In this section we introduce a core imperative language with function calls and returns and primitives for selective speculative load hardening. This language allows us to define our security model and notion of speculative constant-time.

We denote the booleans true and false as \top and \perp . Given a function $f : X \rightarrow Y$, we denote $f[x \leftarrow y]$ the function that

maps x to y and every other x' to $f(x')$. Given two functions $f, g : X \rightarrow Y$ and a relation $\leq \subseteq Y \times Y$, we write $f \leq g$ when $f(x) \leq g(x)$ for each x .

A program is a set of pairs of function names and code, where there is one distinguished pair that is the entry point. The entry point has no callers, and execution halts after reaching its return.

For simplicity, we consider function calls without local variables, arguments, or return values. The syntax for instructions and code is as follows, where e is an expression (either an integer, a boolean, a register variable, or an operation between expressions), x a register variable, a an array variable, f a function name, and b a boolean:

$$\begin{aligned} I ::= & x = e \mid x = a[e] \mid a[e] = x \\ & \mid \text{if } e \text{ then } c \text{ else } c \mid \text{while } e \text{ do } c \mid \text{call}_b f \\ & \mid \text{init_msf}() \mid \text{update_msf}(e) \mid x = \text{protect}(x) \\ c ::= & [] \mid I; c. \end{aligned}$$

We assume that each array comes with its size $|a|$. The definitions are standard, except for the call instruction and the selSLH instructions (which are described in Section 2). The $\text{call}_\top f$ instruction calls f and performs an MSF update on the register `msf` upon return. On the other hand, the instruction $\text{call}_\perp f$ is a usual assembly CALL f instruction, which does not update the misspeculation flag. The programmer annotates call instructions with a boolean b because we will compile returns to tables of conditional jumps, which may trigger misspeculation. Instead of interleaving several MSF updates in the table, as is usual in SLH, we can perform just one at the return site.

Semantics. To formalize our security model, we define for every function f its set of continuations $C(f)$, consisting of triples (c, g, b) , where c is the code—potentially many instructions—that remains to be executed after returning from a call to f , and g is the caller, i.e., the function that contains the instruction $\text{call}_b f$. As an illustration, let c' be the body of function `g` in Figure 2. There are two continuations of f in c' : the first one is $(x = x + 1; c', g, \top)$, i.e., when returning from f to the first call site we need to finish executing the loop body and then reenter the loop; and the second one is $(x = 0, g, \perp)$, i.e., when returning from f to the second call site we only need to execute the last assignment to x .

We define *directives*, using continuations, to model the attacker’s power to influence execution as follows:

$$\text{Dir} ::= \text{step} \mid \text{force } b \mid \text{mem } a \ i \mid \text{return } c \ f \ b.$$

The directive `step` is a usual sequential step, `force b` takes the b branch of a conditional, `mem a i` forces an unsafe memory access to read from or write to the address (a, i) instead, and `return c f b` forces the function to return to a continuation (c, f, b) . On the other hand, *observations* model execution

<pre> 1 id { 2 return 3 } 4 5 main { 6 x = pub 7 call id 8 leak(x) 9 x = sec 10 call id 11 ... // do not leak x 12 }</pre> <p style="text-align: center;">(a)</p>	<pre> 1 id: 2 if ra = 0 jump ℓ_0 3 jump ℓ_1 4 5 main: 6 x = pub 7 ra = 0 8 jump id 9 ℓ_0: leak(x) 10 x = sec 11 ra = 1 12 jump id 13 ℓ_1: ... // do not leak x</pre> <p style="text-align: center;">(b)</p>	<pre> 1 id: 2 if ra = 0 jump ℓ_0 3 jump ℓ_1 4 5 main: 6 init_msf() 7 x = pub 8 ra = 0 9 jump id 10 ℓ_0: update_msf(ra = 0) 11 x = protect(x) 12 leak(x) 13 x = sec 14 ra = 1 15 jump id 16 ℓ_1: ... // do not leak x</pre> <p style="text-align: center;">(c)</p>
---	--	--

Figure 1. (a) This program leaks `sec` speculatively: an attacker can force the second call to `id` to return to the `leak(x)` instruction, thus leaking `x` which holds the value `sec`. (b) Compiled program using return tables. This program does not use `RET` instructions. The second time the `id` block executes, when `x` gets a secret value, an attacker can mistrain the conditional jump predictor to predict that the jump to ℓ_0 will be taken, and speculatively leak `x`. (c) Compiled program with `selSLH` protections. This program is protected because the value of `x` is masked before leaking it. If the attacker mounts the attack discussed in the previous snippet, only a default masked value gets leaked.

```

1 g {
2   while (x < 10) do {
3     call $_{\top}$  f
4     x = x + 1
5   }
6   call $_{\perp}$  f
7   x = 0
8 }
```

Figure 2. The function `g` has two continuations of `f`, one after each call site.

leakage from control flow and memory accesses, and are defined as follows:

$$\text{Obs} ::= \bullet \mid \text{branch } b \mid \text{addr } a \ i.$$

The observation \bullet corresponds to no observation, branch b indicates that the condition of a conditional evaluated to b , and `addr $a \ i$` that a memory access to array a in position i occurred. We model addresses as pairs (a, i) where the first element indicates the base pointer of array a and the second the offset i .

We define the single-step semantics of our language as an indexed relation between states. The indices are directives and observations: $s \xrightarrow[o]{d} s'$ expresses that the directive d makes a state s step to another state s' and produce an observation o . A state is a 6-tuple $\langle c, f, cs, \rho, \mu, ms \rangle$ that consists of the code being executed c , the name of the function being executed f , the call stack cs , the register map ρ , the memory μ , and the misspeculation status ms . A call stack is a list

of pairs of code and function names, a register map maps register names to values, a memory maps arrays and valid indices to values, and a misspeculation status is a boolean indicating whether there has (ever) been misspeculation. The only instructions that push elements to the call stack are of the form `call $_b$ f`, which means that call stack elements are continuations of f .

Figure 3 presents selected rules of the semantics for our language. The **N-LOAD** rule correspond to a safe memory load. It requires that the offset e evaluates to an integer i , which is in bounds of the array a . It leaks the address of the load, producing the observation `addr $a \ i$` . Since the step is safe, it ignores its directive. On the other hand, the **S-LOAD** corresponds to an unsafe load, i.e., one where the offset is out-of-bounds. Since out-of-bounds accesses may access different regions of memory at different times, we give the attacker the power to choose any address for the load, through the directive `mem $a' \ j$` . The observation is as before.

The **CALL** rule states that a function call fetches the body c_g and name g of the callee from the program p , places them as the code and function name under execution, and pushes the caller's code and function name to the call stack.

The **N-RET** rule represents a normal return, during which execution is transferred to the caller, i.e., the top of the call stack. The **S-RET** rule forces execution to continue to a continuation (c, g, b) of f (of the adversary's choosing, different from the top of the call stack), sets the misspeculation status to \top , and, if b is \top , sets msf to `MASK`. Note that the call stack plays no role during speculative execution and hence

$$\begin{array}{c}
 \text{N-LOAD} \\
 \frac{\llbracket e \rrbracket_\rho = i \quad i \in [0, |a|] \quad \rho' = \rho[x \leftarrow \mu(a, i)]}{\langle x = a[e]; c, f, cs, \rho, \mu, ms \rangle \xrightarrow[\text{mem } a' j]{\text{addr } a i} \langle c, f, cs, \rho', \mu, ms \rangle} \\
 \\
 \text{CALL} \\
 \frac{(g, cg) \in p}{\langle \text{call}_b g; c, f, cs, \rho, \mu, ms \rangle \xrightarrow[\text{step}]{\bullet} \langle c_g, g, (c, f) :: cs, \rho, \mu, ms \rangle} \\
 \\
 \text{S-LOAD} \\
 \frac{i \notin [0, |a|] \quad \llbracket e \rrbracket_\rho = i \quad j \in [0, |a'|] \quad \rho' = \rho[x \leftarrow \mu(a', j)]}{\langle x = a[e]; c, f, cs, \rho, \mu, \top \rangle \xrightarrow[\text{mem } a' j]{\text{addr } a i} \langle c, f, cs, \rho', \mu, \top \rangle} \\
 \\
 \text{N-RET} \\
 \frac{}{\langle [], f, (c, g) :: cs, \rho, \mu, ms \rangle \xrightarrow[\text{return } c g b]{\bullet} \langle c, g, cs, \rho, \mu, ms \rangle} \\
 \\
 \text{S-RET} \\
 \frac{(c, g, b) \in C(f) \quad cs \neq (c, g) :: cs' \quad \rho' = \text{if } b \text{ then } \rho[\text{msf} \leftarrow \text{MASK}] \text{ else } \rho}{\langle [], f, cs, \rho, \mu, ms \rangle \xrightarrow[\text{return } c g b]{\bullet} \langle c, g, [], \rho', \mu, \top \rangle}
 \end{array}$$

Figure 3. Selected rules of the small step operational semantics.

is discarded. This rule captures a misspeculation in the return table of a compiled function.

The multi-step semantics $s \xrightarrow[D]{O} s'$ represents $|D|$ steps of execution, under a sequence of directives D , starting from state s and ending in state s' , producing the sequence of observations O . It is the reflexive transitive closure of the single-step semantics, accumulating directives and observations—it follows that $|D| = |O|$. We provide the rest of the rules, which are standard, in the supplementary material.

We now have all the ingredients to rigorously define *speculative constant-time*. The definition is parameterized by a relation on states, which determines which data is public. It is meaningful when it requires that the code, function name, call stack and misspeculation status of the states coincide, and puts restrictions on the register map and memory. For example, if a program expects a public nonce in register n and a secret key in register k , an appropriate relation would be that the states coincide on n , i.e., $\rho_1(n) = \rho_2(n)$. It should not restrict the value of k since, as a secret, it can vary between runs and should not produce an observable difference. An arbitrary relation is more flexible than classifying variables as secret or public, since it can capture fine grained relations such as “ n contains a public nonce with a Hamming weight of eight.”

Definition 1 (Speculative constant-time, ϕ -SCT). *Given ϕ a relation on states, a program p is speculative constant-time w.r.t. ϕ , denoted ϕ -SCT, if and only if executions starting from ϕ -related states produce the same observations under any directives. That is, for any $D, O_1, O_2, s_1, s_2, s'_1$, and s'_2 ,*

$$s_1 \phi s_2 \wedge s_1 \xrightarrow[D]{O_1} s'_1 \wedge s_2 \xrightarrow[D]{O_2} s'_2 \implies O_1 = O_2.$$

This definition captures our intuition of resistance against Spectre attacks since it ensures that executions starting from the same public data, and under any adversarially controlled prediction behavior (modeled by D), produce no observable

difference in the measurements the attacker can perform (modeled by O_1 and O_2). The sequence of directives D captures predictions from both the branch predictor and the RSB. The same sequence of directives guides both runs as is standard in non-interference definitions.

The definition of SCT states that both related states must step under the same directives. At first sight, if one of the states steps but the other one gets stuck, it is not guaranteed that leakage is independent from secrets. We prove that this is impossible: if a typable state can step under directives D , then all indistinguishable states can step under the same directives \clubsuit . More precisely, we show that if the program is well typed, we have that

$$s_1 \phi s_2 \wedge s_1 \xrightarrow[D]{O_1} s'_1 \implies \exists O_2 s'_2. s_2 \xrightarrow[D]{O_2} s'_2.$$

6 Type System

In this section, we introduce a type system for speculative constant-time. We then prove that all initial states of a well-typed program that coincide in their public parts produce the same observations under all sequences of adversarial directives \clubsuit .

Our type system uses security types to track the confidentiality of data and detect possible violations, both under sequential and speculative execution. Concretely, we attach *security levels* to data (in our case, a confidentiality lattice $\{P, S\}$ with $P \leq S$, corresponding to public and secret data) and define *types* as either a level or a type variable α (this is the case when the same register or memory location is used to hold data of different levels at different times).³ Finally, registers and memory locations get *security types*, which consist of a type (that represents the confidentiality of the data under sequential execution) and a level (that represents

³In our artifact, we implement types as either S or a set of type variables. The empty set corresponds to P and $\{\alpha, \beta, \dots\}$ to the maximum of α, β, \dots (thus, the type α is encoded as $\{\alpha\}$).

the maximum confidentiality of the data under all possible speculative executions), as follows:

$$\begin{aligned} \text{level} &::= S \mid P \\ \text{type} &::= \text{level} \mid \alpha \\ \text{stype} &::= \langle \text{type}, \text{level} \rangle. \end{aligned}$$

Given a security type τ , we write τ_n to refer to the first component (called nominal or sequential), and τ_s for the second component (called speculative). Thus, a public variable has security type $\langle P, P \rangle$, a secret one $\langle S, S \rangle$, and a transient one $\langle P, S \rangle$. A speculatively public (resp. secret) variable is one where the speculative component of its type is P (resp. S). Allowing polymorphism in the speculative component of a security type makes the type system unsound, as we discuss next.

Polymorphism. Our type system has polymorphism over security types to allow function calls in contexts where variables have different security types. It is critical in our model where variables and memory are global since it is unrealistic that calling a function requires that all registers and memory locations have some fixed security type. Nevertheless, we need some care when a function call occurs in different contexts because we do not know to which call site it will return. We need to distinguish between sequential and speculative types, as we do not want to allow instantiating as public a register that may, due to misspeculation, contain secret data.

Recall the example in [Figure 1a](#). If we assigned a type such as $\{x : \langle \alpha, \beta \rangle\} \rightarrow \{x : \langle \alpha, \beta \rangle\}$ to `id`, with a polymorphic variable in the speculative component, then we would be able to type this program (choosing θ such that $\theta(\alpha) = \theta(\beta) = P$ for the first call site, and θ' such that $\theta'(\alpha) = \theta'(\beta) = S$ for the second). We need to ensure that under speculation, the output type of the function is the maximum of all possible instantiations we need for the program: $\langle \alpha, \max_{\theta \in P} \{\theta(\beta)\} \rangle$. Since we only have two levels, this restriction means that if at *any* call site we need to instantiate x as speculatively secret, we need to assume that it could be secret in *all* return sites. Conversely, if we guarantee that at *all* call sites x is speculatively public, we can assume this at every return site. In this way, the example is no longer typable: the type must be of the form $\{x : \langle \alpha, t \rangle\} \rightarrow \{x : \langle \alpha', t' \rangle\}$ where $\alpha \leq \alpha'$ and $t \leq t'$. We need t' to be P for the first call site to be typable since we leak x after it, but we need t' to be S for the second call site to be typable since it is after the assignment of a secret value to x . There is no way of typing this example 🐛. We can type this example where we protect x after the first call site because of subtyping: we can choose $\langle \alpha, S \rangle \rightarrow \langle \alpha, S \rangle$ as the type of `id` and instantiate first with P and then with S 🐛. Doing so means that x is speculatively S after the first call site, but the protect ensures we do not leak any secrets.

$$\begin{aligned} \text{FV}(\Sigma) &\triangleq \begin{cases} \text{FV}(e) & \text{if } \Sigma \text{ is outdated}(e), \\ \emptyset & \text{otherwise,} \end{cases} \\ \text{to_lvl}(t) &\triangleq \begin{cases} P & \text{if } t \text{ is } P, \\ S & \text{otherwise,} \end{cases} \\ \Sigma|_e &\triangleq \begin{cases} \text{outdated}(e) & \text{if } \Sigma \text{ is updated,} \\ \text{unknown} & \text{otherwise} \end{cases} \\ \Sigma \sqsubseteq \Sigma' &\triangleq \Sigma = \text{unknown} \vee \Sigma = \Sigma' \end{aligned}$$

Figure 4. Auxiliary definitions for type rules. The free variables of an MSF type are the free variables of its condition if it is outdated, and empty otherwise. The order for MSF types is flat with unknown as the bottom element.

Misspeculation Flag Type. Our type system also needs to keep track of the misspeculation flag to detect whether protections will be effective. For this, we define the MSF type

$$\Sigma ::= \text{unknown} \mid \text{updated} \mid \text{outdated}(e).$$

The MSF type unknown expresses that the program does not know whether the state is misspeculating. The MSF type updated means that the variable `msf` accurately tracks speculation: `msf` holds the value NOMASK if execution has been sequential, and MASK if there has been misspeculation. Performing an `init_msf()` takes us to updated since this instruction executes a speculation fence. Lastly, the MSF type outdated(e) expresses that `msf` holds a value that can be updated to track speculation accurately. After a conditional jump on e in state updated, the MSF type transitions to outdated(e), and we need to execute an `update_msf(e)` to recover the MSF.⁴

Typing Rules. We use the typing judgment $\Sigma, \Gamma \vdash c : \Sigma', \Gamma'$, where Γ and Γ' are mappings from register and array variables to security types (which have a sequential and a speculative component), Σ and Σ' are MSF types, and c is code. [Figure 5](#) presents the typing rules, with some auxiliary definitions in [Figure 4](#).

Type checking requires a static signature for all functions of the program, which associates each function name f with a signature $\Sigma_f, \Gamma_f \rightarrow \Sigma'_f, \Gamma'_f$ of input and output MSF types and contexts. Signatures may contain type variables that get instantiated at each call site. The purpose of the signature is to fix the type of each function, as functions may be typable with different types; they also allow modular verification.

The first three rules are straightforward. Firstly, the **ASSIGN** rule assigns to x the type of the expression e . We must ensure that the assigned variable does not occur in Σ to be

⁴For clarity, we depart from the notation in [9]: what that work denotes ms we write as updated, and for $\text{ms}|_e$ we write outdated(e). Furthermore, in our artifact the MSF type carries one more piece of information: the location of the MSF, i.e., which register it is in.

$$\begin{array}{c}
 \text{ASSIGN} \\
 \frac{\Gamma \vdash e : \tau \quad x \notin \text{FV}(\Sigma)}{\Sigma, \Gamma \vdash x = e : \Sigma, \Gamma[x \leftarrow \tau]} \\
 \\
 \text{LOAD} \\
 \frac{\Gamma \vdash e : P \quad x \notin \text{FV}(\Sigma)}{\Sigma, \Gamma \vdash x = a[e] : \Sigma, \Gamma[x \leftarrow \langle \Gamma(a)_n, S \rangle]} \\
 \\
 \text{STORE} \\
 \frac{\Gamma \leq \Gamma' \quad \Gamma \vdash e : P \quad \Gamma(x) \leq \Gamma'(a) \quad \forall a' \neq a. \Gamma(x)_s \leq \Gamma'(a')_s}{\Sigma, \Gamma \vdash a[e] = x : \Sigma, \Gamma'} \\
 \\
 \text{WHILE} \\
 \frac{\Gamma \vdash e : P \quad \Sigma|_e, \Gamma \vdash c : \Sigma, \Gamma}{\Sigma, \Gamma \vdash \text{while } e \text{ do } c : \Sigma|_e, \Gamma} \\
 \\
 \text{INIT-MSF} \\
 \frac{\Gamma'(v) = \langle \Gamma(v)_n, \text{to_lvl}(\Gamma(v)_n) \rangle \quad \text{for each } v \in \text{Var}}{\Sigma, \Gamma \vdash \text{init_msf}() : \text{updated}, \Gamma'} \\
 \\
 \text{SEQ} \\
 \frac{\Sigma, \Gamma \vdash I : \Sigma', \Gamma' \quad \Sigma', \Gamma' \vdash c : \Sigma'', \Gamma''}{\Sigma, \Gamma \vdash I; c : \Sigma'', \Gamma''} \\
 \\
 \text{UPDATE-MSF} \\
 \frac{}{\text{outdated}(e), \Gamma \vdash \text{update_msf}(e) : \text{updated}, \Gamma} \\
 \\
 \text{PROTECT} \\
 \frac{\tau = \langle \Gamma(x)_n, \text{to_lvl}(\Gamma(x)_n) \rangle}{\text{updated}, \Gamma \vdash y = \text{protect}(x) : \text{updated}, \Gamma[y \leftarrow \tau]} \\
 \\
 \text{COND} \\
 \frac{\Gamma \vdash e : P \quad \Sigma|_e, \Gamma \vdash c_\top : \Sigma', \Gamma' \quad \Sigma|_{!e}, \Gamma \vdash c_\perp : \Sigma', \Gamma'}{\Sigma, \Gamma \vdash \text{if } e \text{ then } c_\top \text{ else } c_\perp : \Sigma', \Gamma'} \\
 \\
 \text{WEAK} \\
 \frac{\Sigma_0 \sqsubseteq \Sigma \quad \Sigma_0, \Gamma_0 \vdash c : \Sigma'_0, \Gamma'_0 \quad \Sigma' \sqsubseteq \Sigma'_0 \quad \Gamma \leq \Gamma_0 \quad \Gamma'_0 \leq \Gamma'}{\Sigma, \Gamma \vdash c : \Sigma', \Gamma'} \\
 \\
 \text{CALL-}\perp \\
 \frac{\text{Sig}(f) = \Sigma_f, \Gamma_f \rightarrow \Sigma'_f, \Gamma'_f}{\Sigma_f, \theta(\Gamma_f) \vdash \text{call}_\perp f : \text{unknown}, \theta(\Gamma'_f)} \\
 \\
 \text{CALL-}\top \\
 \frac{\text{Sig}(f) = \Sigma_f, \Gamma_f \rightarrow \text{updated}, \Gamma'_f}{\Sigma_f, \theta(\Gamma_f) \vdash \text{call}_\top f : \text{updated}, \theta(\Gamma'_f)} \\
 \\
 \text{NIL} \\
 \frac{}{\Sigma, \Gamma \vdash [] : \Sigma, \Gamma}
 \end{array}$$

Figure 5. Type system. Here $\theta : \text{typeVar} \rightarrow \text{level}$ is an instantiation of type variables and Sig is the signature for functions of the program.

able to accurately update the MSF later on; however, if we do not want to update the MSF, we can always choose to weaken Σ to unknown with the **WEAK** rule and make this restriction vacuous. Secondly, the **LOAD** rule ensures that the array index is public, even speculatively. Variable x gets its sequential type from the array (recall that $\Gamma(a)_n$ is the first component of the type of a), but since the index might be speculatively out of bounds, we need to overapproximate the speculative type as S . Lastly, the **STORE** rule also ensures that the index is public, and we update the type of the array to the that of x . Similarly to the case for load, the index might be out of bounds, so we need to update the speculative types of all other arrays (recall that $\Gamma(a)_s$ is the second component of the type of a).

Next come the rules for selSLH instructions. The **INIT-MSF** rule sets the MSF type to updated, and sets the type of each register and array variable to its sequential component, overapproximating polymorphic type variables with S : that is, $\langle P, t \rangle$ goes to $\langle P, P \rangle$, $\langle S, t \rangle$ goes to $\langle S, S \rangle$, and $\langle \alpha, t \rangle$ goes to $\langle \alpha, S \rangle$. We define this $\text{to_lvl}(\cdot)$ overapproximation in **Figure 4**. This instruction is the only way of exiting the state unknown. Secondly, the **UPDATE-MSF** rule expects the MSF type to be outdated (which occurs when we enter a conditional or a loop) and updates it if the condition is the same. Lastly, the **PROTECT** rule requires that the MSF is updated, and sets the security type of y to the sequential counterpart

of the one for x , similarly to the **INIT-MSF** rule but for one variable only.

The **COND** rule ensures that the condition of an if instruction is public, and that each branch is typable with an outdated MSF type w.r.t. the appropriate condition. We define this, denoted $\Sigma|_e$, in **Figure 4**. Thus, the then-branch will need to perform an MSF update with respect to e if it needs to use protect, and similarly for the else-branch with $!e$. The **WHILE** rule is analogous.

The **CALL- \perp** and **CALL- \top** rules ensure that before every call site of f , the current MSF type and context are what f expects them to be, according to its signature. The resulting MSF type depends on the boolean parameter of the instruction: if the programmer wants the output type to be updated, they choose the b parameter to be \top and ensure that the output MSF type of the function is updated. This rule allows instantiating the type variables in Γ with an instantiation θ —inferred by the type checker—that assigns a level to type variables.

The **WEAK** rule allows to compose typing judgments by weakening them, and the **NIL** and **SEQ** rules chain judgments in the usual way. Finally, a program is well-typed if the body of each function is typable with its signature.

Soundness. Our soundness theorem states that executions of a typable program depend only on public data. Recall that the definition of SCT is parameterized by a relation. The relation we need is indistinguishability of states, which holds

when two states coincide on their public values. More specifically, we say that two states are indistinguishable if

1. their code, function name, call stack, and misspeculation status are the same;
2. their registers and memory coincide on speculatively public variables (i.e., if τ_s is P, where τ is the type of a variable v , the states coincide on v); and
3. if they are not misspeculating, the registers and memories coincide also on sequentially public variables.

This definition depends, therefore, on a typing context.

Theorem 1 (Soundness \clubsuit). *If a program is safe and typable, and the initial type for its entry point is $(\text{unknown}, \Gamma)$, then every pair of indistinguishable initial states (i.e., that coincide on their public parts and their misspeculation statuses) produce, under a given list of directives, the same observations.*

Informally, the notion of *safety* is that any reachable state (starting from an initial state of the program) is either misspeculating, final, or there is a directive that allows a step of execution. The notion imposes the usual conditions only when the reached state is not misspeculating, i.e., under sequential execution. More precisely, a state s' is *reachable* from another state s if there exist directives and observations such that $s \xrightarrow[D]{O} s'$. A state is *misspeculating* if and only if its misspeculation status is \top , and it is *final* if its code and call stack are empty.

The proof of the soundness theorem needs an important lemma, for single-step execution soundness. It says that if a state is well-typed, which intuitively means that the code being executed and the contents of the call stack are typable, the resulting state after one execution step is also well-typed and the observations are the same for all indistinguishable states. Naturally we also need to show that initial states are well-typed.

7 Return-Table Insertion

In this section, we present return-table insertion and show that it preserves speculative constant-time. We compile our structured source language into an unstructured linear language with the same base instructions (assignments, loads, stores, and selSLH instructions) but without the control flow primitives (i.e., if statements, while loops, and function calls). The linear language has only two control flow constructs, conditional and unconditional direct jumps, and programs are lists of labeled instructions.

Linear states are similar to source ones, but instead of code being executed we have a *program counter*, which is a label pointing to an instruction in the program. We denote t_{pc} to the program counter of t . The operational semantics of this language is standard, with similar directive and observation behavior as the source.

$$\begin{aligned} (\llbracket \text{call}_\top f \rrbracket) &\triangleq \begin{array}{l} ra_f = \ell_{ret} \\ \text{jump } f \\ \ell_{ret} : \text{update_msf}(ra_f = \ell_{ret}) \end{array} \\ (\llbracket \{\ell_r\} \rrbracket_{\text{rettbl}}^f) &\triangleq \text{jump } \ell_r \\ (\llbracket \{\ell_r\} \cup \ell^* \rrbracket_{\text{rettbl}}^f) &\triangleq \text{if } ra_f = \ell_r \text{ jump } \ell_r; (\llbracket \ell^* \rrbracket_{\text{rettbl}}^f) \end{aligned}$$

Figure 6. Compilation of function calls and return tables. We write $\llbracket c \rrbracket$ for the compilation of c , and $\llbracket \ell^* \rrbracket_{\text{rettbl}}^f$ for the compilation of the return table of f whose entries are ℓ^* . We omit the labels of instructions that do not need them. The register ra_f is where the function f expects its return address.

Figure 6 presents the case of non-recursive function calls in our compilation scheme. Compiling assignments, memory and selSLH instructions is trivial since the languages coincide on these constructs. We compile conditionals and loops in the standard way with a conditional jump. For a non-recursive function f , we pass its return address in a dedicated return address register ra_f ; we discuss this restriction in Section 8.

Figure 6 also shows the compilation of return tables as a sequence of conditional direct jumps, given a non-empty set of return labels. If the set has only one return label, we generate a direct jump to it. Otherwise, we generate conditional branches comparing return addresses with each return label.

Compiling a program p , denoted $\llbracket p \rrbracket$, entails compiling for each function f its body at a distinguished label ℓ_f , followed by its return table at ℓ_{ret_f} . We derive the set of return labels of a function from its set of continuations, that is, the program points following a call to it. As mentioned in Section 5, the entry point has no callers, and thus no return table. The last instruction of the code of the entry points to a distinguished, invalid label, indicating program termination.

Preservation of SCT. We use a definition of SCT similar to Definition 1 for linear programs, and define a similar indistinguishability relation between states. Our compilation scheme ensures that the compilation of a typable program is not vulnerable to Spectre attacks.

Theorem 2 (Preservation of SCT \clubsuit). *If p is typable then $\llbracket p \rrbracket$ is SCT.*

We prove this theorem by showing that the leakage of an execution depends only on the public part of the initial state and a corresponding source execution. In order to do this, we define a directive transformer, which computes source directives given the program counter and a target directive, and a leakage transformer, which computes target leakage

given the program counter and source leakage:

$$T_{\text{Dir}} : \text{label} \times \text{Dir} \rightarrow \text{Dir}^*, \quad T_{\text{Obs}} : \text{label} \times \text{Obs}^* \rightarrow \text{Obs}.$$

The proof is a standard simulation, where the key lemma is as follows:

Lemma 1 (Single-step leakage transformation ). *For all single-step executions $t \xrightarrow[\text{d}]{\text{o}_t} t'$, if there exists a typable source state $s \sim t$, then there exists a state s' and observations O_s such that $s \xrightarrow[\text{T}_{\text{Dir}}(t_{pc}, d)]{O_s} s'$, $T_{\text{Obs}}(t_{pc}, O_s) = o_t$, and $s' \sim t'$.*

Here the relation $s \sim t$ indicates that the linear state t is the compilation of the source state s . Note that the source directives $T_{\text{Dir}}(t_{pc}, d)$ can be empty, corresponding to a silent step (in the source). It is usual that backward simulations for compiler correctness proofs require showing that there are a finite number of silent steps, but for security proofs like ours this is not needed—silent steps do not leak secrets.

We give a formalization of the target language and the trivial compilation cases, together with details on the proof of single-step leakage transformation and an intuition for these transformers in the supplementary material.

8 Implementation in the Jasmin Compiler

In this section, we describe how we implemented, within the Jasmin compiler [1, 4], the type system and compilation scheme presented in Sections 6 and 7.

Changes to the SCT Checker. Jasmin provides an automatic checker for Spectre-v1 vulnerabilities that takes into account selSLH instructions. We extend this checker to consider also Spectre-RSB, following the type system from Section 6. We must now bridge the gap between the model presented in this work and Jasmin. Firstly, function calls in our model are nothing more than a transfer of control to the body of the function, while in Jasmin, functions have local variables, arguments, and results. Since, at the source level, we do not know which variables of the callers of a function will be allocated to which registers, we need to be coarser than in Section 6 and consider that, after a function call, all public variables become transient. Secondly, we introduce an annotation #update_after_call for function calls, corresponding to the boolean parameter of call instructions. Using this annotation corresponds to setting the b parameter of call instructions to \top , and omitting it corresponds to \perp . Thus, we have that

$$\text{call}_{\perp} f \text{ is } x = f(y); \quad \text{call}_{\top} f \text{ is } \begin{array}{l} \text{\#update_after_call} \\ x, \text{msf} = f(y, \text{msf}); \end{array}$$

Recall that, the MSF variable is explicit in Jasmin, as mentioned in Section 2. Keeping the MSF in different locations (registers or MMX registers) allows us to spill it when register pressure is high. Finally, we modify the type system so that only public data flows into MMX registers, even speculatively. This is not strictly necessary, but proved extremely

1 callee:	1 caller:
2 ...	2 ...
3 CMP ra, ℓ	3 MOV ℓ , ra
4 JMPeq ℓ	4 JMP callee
5 JMPlt LT_branch	5 ℓ : MOV MASK, tmp
6 // GT_branch	6 CMOVne tmp, msf

Figure 7. A return table implemented as a tree and a return site that reuses the comparison made in the table.

convenient, since it frees us from keeping an MSF in cases when we have only a few values to protect. We expand on this below.

Changes to the Compiler. We adapt the Jasmin compiler to use direct jumps instead of CALL and RET instructions. As discussed in Section 7, we compile unannotated function calls as two instructions (to save the return address and perform a direct jump). In contrast, calls annotated with #update_after_call issue a third one, an MSF update.

We implement return tables as trees, which means that the number of comparisons is logarithmic in the number of callers of a function. Moreover, at return sites, in most cases, the MSF update can reuse the flags that we set in the last comparison before jumping. The most frequent instance of this is the one depicted in Figure 7, where at the return site ℓ we have an MSF update with condition $\text{ra} = \ell$. We need not introduce a CMP instruction for this update, since the flags set before jumping correctly reflect the condition (in this case EQ).

The compiler is flexible in passing return addresses in different ways. For libjade, using MMX registers was the best option, as cryptographic implementations seldom use these registers. By modifying the type checker to ensure that all writes to MMX registers are public, even speculatively, we never need to protect them. This restriction is also beneficial when we only need to protect a few values because we can place them in MMX registers and thus avoid keeping an MSF. Since using these registers can be expensive and register pressure can be high in some of the programs in libjade, the compiler also allows passing return addresses on the stack or in general-purpose registers. This, however, requires some care: when passing them in an arbitrary location, we need to be mindful of speculative writes to this location.

Figure 8 shows how naively passing the return address is insecure. Note how a return table leaks its return address since it performs conditional branches on it. In this example, the problem is that the return table in f leaks the secret that evil puts into the register ra_f . The function g cannot modify register ra_f because one of its callers, f , uses it. However, a different caller, evil , can put a secret there, and when it calls g , the attacker can force g to return to f . The return table in f then leaks ra_f as remarked.

1 f:	1 g:
2 ...	2 ...
3 $ra_g = f_0$	3 if $ra_g = f_0$ jump f_0
4 jump g	4 jump $evil_0$
5 f_0 : ...	1 evil:
6 if $ra_f = \ell$ jump ℓ	2 $ra_f = secret$
7 jump ℓ'	3 $ra_g = evil_0$
	4 jump g
	5 $evil_0$: ...

Figure 8. How a secret may leak as a return tag.

Protecting the return address using an MSF mitigates the problem: the leaked comparisons will be against a default value instead of a secret. Note that there is no risk of a speculative write to the return address forcing execution to an invalid program point since the table will perform comparisons on it, but the targets of all jumps are hard coded valid labels. We can pass the return address on the stack for recursive functions, but this is unnecessary for Jasmin as it does not support them. The drawback of protecting return addresses is that we need an MSF at each return site that needs the protection. This entails keeping an MSF updated, which means more instructions and data dependencies, and, therefore, a greater overhead. Fortunately, MMX registers are free from this drawback.

9 Evaluation

Now we overview the changes to libjade and evaluates the computational cost of Spectre-RSB countermeasures. libjade is a high-assurance cryptographic library written in Jasmin and extended by [9] to be Spectre-v1 protected. The present work uses the artifact from [9] as our starting point, which contains the constant-time implementations (with no countermeasures against Spectre attacks) and the corresponding Spectre-v1 protected implementations.

9.1 Modifications to libjade

We started by updating these implementations to be compatible with recent versions of the Jasmin compiler and then added RSB protections to the Spectre-v1 protected version. The primitive that required the largest amount of changes was Kyber [17]; in particular, no other primitive required the `#update_after_call` annotation. Kyber512 and Kyber768 share a significant part of the code, which is generic on the algorithm’s parameters. We needed to annotate 49 out of 51 call sites in Kyber512 with `#update_after_call`, and 56 out of 58 in Kyber768. A rejection-sampling routine is the main reason for the difference in the number of call sites between Kyber512 and Kyber768 (it accounts for six call sites).

Sometimes, we can avoid keeping an MSF by applying one of four different strategies when protecting our code; the libjade implementations of Kyber have examples of all

four. First, we inline function calls if the code size penalty is minor; this is the case for two function calls in Kyber. Second, we spill public values to MMX registers—these are guaranteed to remain public—when the performance penalty allows; this is the case for all calls to SHAKE in Kyber. Third, we enforce that some function arguments are always public, since, in some cases, the type system (soundly) generalizes too much and gives false positives. An example of this is the `id` function in Figure 1a: the type system will greedily assign a polymorphic type to this function, $\alpha \rightarrow \alpha$, and thus will we need to protect its result after calling it. If in our program we notice that we only call it with public arguments, we can annotate this function as `id(#public x) -> #public`, which is a more restrictive type, but frees us from having to protect its result after calling it. To justify the fourth and last strategy, let us remark that if a function does not modify a particular register, which is guaranteed to be public at every call site of the function, we can safely assume that it is public at each return site—this is an extension of the third strategy. We can capitalize on this realization at the Jasmin level by making such functions take extra arguments, *annotating them as public*, and returning them unmodified. In this way, the type system will enforce that these variables are always public at every call site, even speculatively, and register allocation will force these variables to the same architectural register since they are an argument to a function.

The `keypair` and `enc` functions of Kyber each use a call to an external `randombytes` function that serves as a wrapper around a `getrandom` system call. These calls to external functions (with actual `RET` instructions) violate the assumptions of our security arguments; they are currently being replaced by a re-implementation of `randombytes` for an upcoming Jasmin release. We expect no significant performance difference from this upcoming change to Jasmin.

9.2 Performance of libjade

Table 1 reports benchmarks of highly optimized implementations of various cryptographic primitives in libjade with different Spectre protections. The benchmarks in the table are the median cycle count of 10000 executions on a single core of an Intel Core i7 11700K (Rocket Lake) CPU running at 3600 MHz with TurboBoost and hyper-threading disabled. The benchmarking state is running Debian 6.1.76, and we compiled our benchmarking code using GCC 12.2.

In addition to the implementations of libjade, we include results for alternative cryptographic libraries for comparison purposes (column “Alt.”). For each Jasmin primitive, the leftmost column (“plain”) cycle count is the baseline CT implementation with no Spectre protections. As a first step (“+SSBD”), we set the SSBD CPU flag to protect against Spectre v4 attacks. In a next step (“+SSBD+v1”), we additionally add the `selSLH` protections against Spectre-v1 as described in [9]. Finally, we report cycle counts with the full protections described in this paper (“+SSBD+v1+RSB”).

Table 1. libjade benchmarks on Intel Core i7 11700K (most optimized implementation of each primitive). “Alt.”: cycles of alternative cryptographic libraries; “plain”: cycles without any Spectre protections; “+SSBD”: with SSBD CPU flag set; “+SSBD+v1”: with SSBD CPU flag set and v1 countermeasures from [9]; “+SSBD+v1+RSB”: with full Spectre protection as described in this paper; “increase”: relative increase in CPU cycles between unprotected (“plain”) and fully protected (+SSBD+v1+RSB).

Primitive	Impl.	Operation	Alt.	plain	+SSBD	+SSBD+v1	+SSBD+v1+RSB	increase (%)
ChaCha20	avx2	1 KiB	-	1198	1202	1244	1246	4.01
		1 KiB xor	1230	1208	1212	1248	1250	3.48
		16 KiB	-	19040	19052	19066	19068	0.15
		16 KiB xor	18960	19070	19086	19096	19110	0.21
Poly1305	avx2	1 KiB	704	670	672	720	718	7.16
		1 KiB verif	-	674	676	726	724	7.42
		16 KiB	8590	8942	8948	8990	8986	0.49
		16 KiB verif	-	8942	8984	8984	8984	0.47
XSalsa20Poly1305	avx2	128 B	1834	1206	1212	1250	1246	3.32
		128 B open	2698	1964	1970	2044	2046	4.18
		1 KiB	5956	3140	3142	3190	3188	1.53
		1 KiB open	6858	3900	3904	3988	3988	2.26
		16 KiB	82642	32598	32574	32604	32602	0.01
		16 KiB open	83582	33292	33274	33358	33362	0.21
X25519	mulx	smult	121730	102848	104150	104424	104428	1.54
Kyber512	avx2	keypair	28802	27676	28106	28040	28090	1.50
		enc	31032	37050	38332	38876	38792	4.70
		dec	38816	29302	30444	30590	30714	4.82
Kyber768	avx2	keypair	48036	43432	45708	45860	46548	7.17
		enc	49016	57006	59316	60028	60674	6.43
		dec	60682	46138	48418	48532	49294	6.84

The alternative implementations of ChaCha20, Poly1305, and X25519 are taken from OpenSSL 3.4.0. For ChaCha20 and Poly1305, we omit the data corresponding to the production of a stream (ChaCha20) and, for Poly1305, MAC verification. This is because OpenSSL provide no interface that corresponds directly to these operations. We developed an interface for compatibility with our benchmarking framework, and omit this data as it is not part of OpenSSL. We report the cycle counts for XSalsa20Poly1305 (not available in OpenSSL) from libsodium 1.0.20. The fastest implementation of this primitive in libsodium is not avx2 (although it uses 128-bit vectorization instructions); hence, the leftmost label avx2 (256-bit vectorization) applies only to the Jasmin implementations. OpenSSL 3.4.0 does not implement Kyber, so we include cycles from mlkem-native, which provides updated and optimized implementations of ML-KEM. ML-KEM differs slightly from Kyber, but these algorithmic differences do not significantly affect the presented data.

We see that for the symmetric primitives, i.e., ChaCha20, Poly1305, and XSalsa20Poly1305, the overhead for full Spectre protection is solidly below 1% when processing sufficiently long messages. The rather large overhead for short

messages is due to the fixed cost of the initial fence; this is consistent with the observations reported in [9].

For X25519 [16], an elliptic-curve Diffie-Hellman key exchange, we see a slightly larger overhead, which is almost entirely due to Spectre-v4 protections, i.e., setting the SSBD flag. This is not surprising, because the active data set in the speed-critical main loop of X25519 is considerably larger than in the symmetric primitives. The main loop thus involves more loads and stores that potentially benefit from speculative store bypass and may thus be slowed down by SSBD.

The most interesting measurement results are those for Kyber512 and Kyber768. Kyber is the most complex scheme in our benchmarks in terms of code size, number of function calls, and size of the active data set throughout the speed-critical computations. Consequently, it is not surprising to see a slightly higher overhead from Spectre protections in Kyber than, e.g., X25519. Given that the generation of the 3×3 matrix in Kyber768 does not vectorize as well as for the 2×2 one in Kyber512, it is also expected that the overhead for the former is higher. The results for the `keypair` operation are surprising: it has the smallest overhead in Kyber512 but

the largest in Kyber768. We will continue to investigate the reasons for this behavior.

10 Related Work

Return Tables as Compiler Optimizations. Calder and Grunwald [18] show that return tables can improve performance in object-oriented programs. Yang, Coopriider, and Regehr show that return tables can reduce RAM usage in embedded code [40]. Both transformations keep some indirect jumps, since their goal is efficiency, and thus are inadequate as mitigations against Spectre.

ROP Countermeasures. Return-oriented programming is an exploitation technique that targets return instructions to force program execution to jump to arbitrary program points [36]. In contrast to Spectre-RSB, ROP does not exploit speculative execution. There are many ROP countermeasures; our work is closely related to countermeasures that remove calls and returns [30, 34] and replace them with indirect jumps. Arthur et al. [10] is the closest since it introduces only *direct* branches. The main difference with our work is that we make this transformation resistant to speculative execution attacks and compatible with selective speculative load hardening. Other countermeasures harden return instructions by using return indirection or randomizing return addresses. Unfortunately, these transformations are ineffective in our scenario.

Spectre Countermeasures. There is a large body of work that proposes countermeasures and verification approaches against Spectre. We refer the reader to two recent surveys for background [19, 21] and focus on closely related work.

Swivel [33] is a software-only compiler framework (with a hardware-assisted variant) for WebAssembly that tackles Spectre-v1, v2, and RSB. Similarly to Venkman [37], Swivel enforces coarse-grained control-flow integrity (CFI) under speculation by starting from a clean branch target buffer (BTB) and RSB and restricting jumps to the beginning of basic blocks. It implements various mitigations on top of this, including disjoint memory regions for blocks (enforced by masking), a Spectre-protected shadow stack (using either guard pages or Intel CET), masking of addresses, and flushes of the BTB (on every transition into and out of the sandbox). In contrast to our work, Swivel incurs significant overhead and lacks formal guarantees.

Serberus [32] is a comprehensive approach to protect programs against all known Spectre attacks. Serberus uses CFI protections to constrain the attacker’s power over speculative control flow, and a sequence of program transformations to eliminate speculative leakage. One main difference with our approach is that Serberus requires hardware and operating system support. Specifically, Serberus derives its CFI protection from Intel’s CET [24] and DOIT [25], and requires the operating system to perform RSB stuffing on

context switches. Another main difference is that Serberus needs to use fences and to spill all function arguments as its primary protection mechanisms against speculative leakage, rather than selective speculative load hardening. This is reflected in the experimental evaluation, which reports a 21.3% overhead. In contrast, the overhead of our approach is minimal, and it uses hardware support only for Spectre-v4, for precisely this reason.

Retpoline [26] is an early software-based countermeasure against Spectre-v2—and some variants of Spectre-RSB—that replaces indirect jumps by return instructions. This mitigation leverages knowledge of how the RSB is implemented—as a LIFO buffer—to insert fences at the points where execution would continue if the predictor is wrong. Unfortunately, Wikner and Razavi [39] show that the assumptions that retpoline relies on are incorrect. JumpSwitches [8] improves the performance of retpolines by generating partial tables of return targets. Switchpoline [15] builds on that to produce a software-based countermeasure that replaces indirect jumps with tables of direct ones. It instruments them with a JIT compiler that generates new entries (i.e., new direct jumps) at runtime for the targets that were not inferred statically. Although Switchpoline targets Spectre-v2 in ARM, the transformation is very similar to ours. A critical difference between Switchpoline and our approach is that the former does not consider how to combine its transformation with efficient countermeasures to protect programs. In particular, our proof shows that return tables introduce leakage, which needs mitigation.

11 Limitations

One limitation of our approach is that it applies only to full programs because an (unprotected) external function call can exploit the RSB to bypass protections. This limitation is common to other approaches, such as Serberus [32], Switchpoline [15], Swivel [33], and Venkman [37]. In particular, our approach does not allow separate compilation of programs, which is not supported in Jasmin, since the compiler must statically construct return tables. Thus, even if our type system and transformation, as well as selSLH, are general and could apply to mainstream languages, for the former to be secure, we would need to give up on certain features such as function pointers and separate compilation.

Another limitation of our approach is that it does not account for declassification. We are confident that our results extend with declassification, at the cost of switching from speculative constant-time to relative speculative constant-time. In the future, we hope to leverage a formalization of the type system in Coq to extend our results to declassification.

12 Conclusion

We have proposed an approach to protect against all known forms of Spectre attacks. Our approach consists of an enriched set of program primitives and a compiler pass. We provide a machine-checked proof of the correctness of our approach in Coq, implement it in the Jasmin compiler, and evaluate the impact on the library of cryptographic primitives `libjade`.

Our implementation is currently limited to Jasmin programs. A pragmatic solution to carry the essence of our techniques to mainstream languages would be to instrument existing compilers with a pass for return table instructions and to develop assembly-level type systems for checking speculative constant-timeness.

Acknowledgments

We thank the anonymous reviewers, and specially our shepherd Guy Steele, who were very generous with their time and many insightful suggestions. This research was supported an ARC Discovery Early Career Researcher Award (project number DE200101577); an ARC Discovery Project (project number DP210102670); by Deutsche Forschungsgemeinschaft (DFG, German research Foundation) as part of the Excellence Strategy of the German Federal and State Governments – EXC 2092 CASA - 390781972; by the European Commission through the ERC Starting Grant 805031 (EPOQUE); by the German Federal Ministry of Education and Research (BMBF) in the course of the 6GEM research hub under grant number 16KISK038; by the *Agence Nationale de la Recherche* (French National Research Agency) as part of the France 2030 programme – ANR-22-PECY-0006. Author Peter Schwabe is a member of the advisory boards of Bitmark Inc., PQShield, Neutrality, and SciEngines.

References

- [1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-Assurance and High-Speed Cryptography. In *ACM CCS*. 1807–1823. <https://doi.org/10.1145/3133956.3134078>
- [2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *USENIX Security*. 53–70. <https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida>
- [3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Tiago Oliveira, Hugo Pacheco, Miguel Quaresma, Peter Schwabe, Antoine Séré, and Pierre-Yves Strub. 2023. Formally verifying Kyber Episode IV: Implementation correctness. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2023, 3 (2023), 164–193. <https://doi.org/10.46586/tches.v2023.i3.164-193>
- [4] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. 2020. The Last Mile: High-Assurance and High-Speed Cryptographic Implementations. In *IEEE S&P*. 965–982. <https://doi.org/10.1109/SP40000.2020.00028>
- [5] José Bacelar Almeida, Cécile Baritel-Ruet, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Alley Stoughton, and Pierre-Yves Strub. 2019. Machine-Checked Proofs for Cryptographic Standards: Indifferentiability of Sponge and Secure High-Assurance Implementations of SHA-3. In *ACM CCS*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). 1607–1622. <https://doi.org/10.1145/3319535.3363211>
- [6] José Bacelar Almeida, Santiago Arranz Olmos, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, Jean-Christophe Léchenet, Cameron Low, Tiago Oliveira, Hugo Pacheco, Miguel Quaresma, Peter Schwabe, and Pierre-Yves Strub. 2024. Formally verifying Kyber Episode V: Machine-checked IND-CCA security and correctness of ML-KEM in EasyCrypt. In *IACR CRYPTO*. to appear. <https://eprint.iacr.org/2024/843>
- [7] AMD. 2021. *Security Analysis of AMD Predictive Store Forwarding*. Technical Report. <https://www.amd.com/system/files/documents/security-analysis-predictive-store-forwarding.pdf>
- [8] Nadav Amit, Fred Jacobs, and Michael Wei. 2019. JumpSwitches: Restoring the Performance of Indirect Branches In the Era of Spectre. In *USENIX Security*. 285–300. <https://www.usenix.org/conference/atc19/presentation/amit>
- [9] Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Swarn Priya, Peter Schwabe, and Lucas Tabary-Maujean. 2023. Typing High-Speed Cryptography against Spectre v1. In *IEEE S&P*. 1094–1111. <https://doi.org/10.1109/SP46215.2023.10179418>
- [10] William Arthur, Ben Mehne, Reetuparna Das, and Todd Austin. 2015. Getting in control of your control flow with control-data isolation. In *IEEE/ACM CGO*. 79–90. <https://ieeexplore.ieee.org/abstract/document/7054189>
- [11] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. 2021. SoK: Computer-Aided Cryptography. In *IEEE S&P*. 777–795. <https://doi.org/10.1109/SP40001.2021.00008>
- [12] Manuel Barbosa, Gilles Barthe, Christian Doczkal, Jelle Don, Serge Fehr, Benjamin Grégoire, Yu-Hsuan Huang, Andreas Hülsing, Yi Lee, and Xiaodi Wu. 2023. Fixing and Mechanizing the Security Proof of Fiat-Shamir with Aborts and Dilithium. In *IACR CRYPTO*. 358–389. <https://doi.org/10.1007/978-3-031-38554-4%5F12>
- [13] Manuel Barbosa, François Dupressoir, Benjamin Grégoire, Andreas Hülsing, Matthias Meijers, and Pierre-Yves Strub. 2023. Machine-Checked Security for XMSS as in RFC 8391 and SPHINCS+. In *IACR CRYPTO*. 421–454. <https://doi.org/10.1007/978-3-031-38554-4%5F14>
- [14] Gilles Barthe, Benjamin Grégoire, Vincent Laporte, and Swarn Priya. 2021. Structured Leakage and Applications to Cryptographic Constant-Time and Cost. In *ACM CCS*. 462–476. <https://doi.org/10.1145/3460120.3484761>
- [15] Markus Bauer, Lorenz Hetterich, Christian Rossow, and Michael Schwarz. 2024. Switchpoline: A Software Mitigation for Spectre-BTB and Spectre-BHB on ARMv8. In *ACM AsiaCCS*. <https://misc0110.net/files/switchpoline%5Fasiaccs24.pdf>
- [16] Daniel J. Bernstein. 2006. Curve25519: new Diffie-Hellman speed records. In *IACR PKC*. 207–228. <https://cr.yp.to/papers.html#curve25519>
- [17] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrede Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2018. CRYSTALS-Kyber: A CCA-Secure Module-Lattice-Based KEM. In *IEEE EuroS&P*. 353–367. <https://doi.org/10.1109/EuroSP.2018.00032>
- [18] Brad Calder and Dirk Grunwald. 1994. Reducing indirect function call overhead in C++ programs. In *ACM POPL*. 397–408. <https://dl.acm.org/doi/10.1145/174675.177973>

- [19] Claudio Canella, Jo Van Bulck, Michael Schwarz, Frank Piessens, Dmitry Evtvushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security*. 249–266. <https://www.usenix.org/conference/usenixsecurity19/presentation/canella>
- [20] Chandler Carruth. [n.d.]. Speculative Load Hardening – A Spectre Variant #1 Mitigation Technique. LLVM documentation. <https://llvm.org/docs/SpeculativeLoadHardening.html>
- [21] Sunjay Cauligi, Craig Disselkoe, Daniel Moghimi, Gilles Barthe, and Deian Stefan. 2022. SoK: Practical Foundations for Software Spectre Defenses. In *IEEE S&P*. 666–680. <https://doi.org/10.1109/SP46214.2022.9833707>
- [22] Sunjay Cauligi, Craig Disselkoe, Klaus von Gleissenthall, Dean M. Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2020. Constant-time foundations for the new Spectre era. In *ACM PLDI*. 913–926. <https://doi.org/10.1145/3385412.3385970>
- [23] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. 2011. Computer-Aided Security Proofs for the Working Cryptographer. In *IACR CRYPTO*. 71–90. <https://iacr.org/archive/crypto2011/68410071/68410071.pdf>
- [24] Intel. [n.d.]. A Technical Look at Intel’s Control-flow Enforcement Technology. <https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html>
- [25] Intel documentation. [n.d.]. Data Operand Independent Timing Instruction Set Architecture (ISA) Guidance. <https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html>
- [26] Intel Labs. 2018. Retpoline: a software construct for preventing branch-target-injection. <https://www.intel.com/content/dam/develop/external/us/en/documents/retpoline-a-branch-target-injection-mitigation.pdf>
- [27] Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. 2022. “They’re not that hard to mitigate”: What Cryptographic Library Developers Think About Timing Attacks. In *IEEE S&P*. 632–649. <https://eprint.iacr.org/2021/1650>
- [28] 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 *IEEE S&P*. 1–19. <https://spectreattack.com/spectre.pdf>
- [29] Esmail Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *USENIX WOOT*. <https://www.usenix.org/conference/woot18/presentation/koruyeh>
- [30] Jinku Li, Zhi Wang, Xuxian Jiang, Michael C. Grace, and Sina Bahram. 2010. Defeating return-oriented rootkits with “Return-Less” kernels. In *ACM EuroSys*. 195–208. <https://doi.org/10.1145/1755913.1755934>
- [31] Giorgi Maisuradze and Christian Rossow. 2018. ret2spec: Speculative Execution Using Return Stack Buffers. In *ACM CCS*. 2109–2122. <https://dl.acm.org/doi/10.1145/3243734.3243761>
- [32] Nicholas Mosier, Hamed Nemat, John C. Mitchell, and Caroline Trippel. 2024. Serberus: Protecting Cryptographic Code from Spectres at Compile-Time. In *IEEE S&P*. <https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00048>
- [33] Shrahan Narayan, Craig Disselkoe, 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 *USENIX Security*. 1433–1450. <https://www.usenix.org/conference/usenixsecurity21/presentation/narayan>
- [34] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. 2010. G-Free: defeating return-oriented programming through gadget-less binaries. In *ACM ACSAC*. 49–58. <https://doi.org/10.1145/1920261.1920269>
- [35] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Transactions on Information and System Security* 15, 1 (2012), 2:1–2:34. <https://dl.acm.org/doi/10.1145/2133375.2133377>
- [36] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM CCS*. 552–561. <https://doi.org/10.1145/1315245.1315313>
- [37] Zhuojia Shen, Jie Zhou, Divya Ojha, and John Criswell. 2019. Restricting Control Flow During Speculative Execution with Venkman. <https://arxiv.org/abs/1903.10651v1>
- [38] 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 *IEEE S&P*. 1753–1770. <https://doi.org/10.1109/SP46215.2023.10179355>
- [39] Johannes Wikner and Kaveh Razavi. 2022. RETBLED: Arbitrary Speculative Code Execution with Return Instructions. In *USENIX Security*. 3825–3842. <https://www.usenix.org/conference/usenixsecurity22/presentation/wikner>
- [40] Xuejun Yang, Nathan Coopridge, and John Regehr. 2009. Eliminating the call stack to save RAM. In *ACM LCTES*. 60–69. <https://doi.org/10.1145/1542452.1542461>

A Artifact Appendix

A.1 Abstract

This is the artifact for the paper “Protecting Cryptographic Code Against Spectre-RSB.” It contains a Coq formalization of the approach presented in the paper, a version of the Jasmin compiler that protects programs against Spectre-RSB, a version of the libjade crypto library protected against all known Spectre variants, and benchmarks for the updated version of libjade. The main contributions in this artifact are a new SCT type system for Jasmin that checks for Spectre-RSB, the Coq formalization and proof of our approach, and high-assurance crypto implementations protected against all known Spectre variants.

To build the Coq formalization and the Jasmin compiler, we provide instructions using nix-shell. In addition, we provide a Docker image with the Jasmin compiler already installed. To run the benchmarks, we provide standard Makefiles (that require the Jasmin compiler).

The result of building this artifact is high confidence on the security of our approach and evidence of its overhead being minimal.

A.2 Artifact check-list (meta-information)

- **Program:** Benchmarks are included.
- **Compilation:** The Jasmin compiler is included.
- **Transformations:** The “protect calls” pass in the Jasmin compiler is included.
- **Run-time environment:** Not specific. Requires nix-shell or Docker.

- **Hardware:** A modern Intel laptop with a firmware update to enable SSBD.
- **Metrics:** CPU cycle counts.
- **Output:** Console output for the formalization, type checker, and library. Assembly code for the compiler. PDF tables for the benchmarks.
- **How much disk space required (approximately)?:** 5GB
- **How much time is needed to prepare workflow (approximately)?:** One hour.
- **How much time is needed to complete experiments (approximately)?:** One hour.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** MIT.
- **Workflow framework used?:** Nix, Dune, Docker, GNU Makefile.
- **Archived (provide DOI)?:** <https://doi.org/10.5281/zenodo.14773254>

A.3 Description

A.3.1 How to access. Download from <https://doi.org/10.5281/zenodo.14773254> (compressed ~5MB).

A.3.2 Software dependencies. A recent version of Docker, or, alternatively, nix-shell, make, and GCC.

A.4 Installation And Basic Tests

The README.md at the top level of our artifact contains instructions with useful details. We recommend that for an easier and customized build. It also contains instructions for doing all of these steps inside Docker, which we omit here. This artifact contains the following directories:

1. formalization, a contribution of this work: a formalization in Coq of the source and target speculative semantics, the type system, the compilation using jump table for return, and of the theorem 1 and 2 of the paper.
2. jasmin, a contribution of this work: an improved version of the Jasmin compiler and SCT checker.
3. sslh_rsb, a contribution of this work: a fully protected version of libjade.
4. scripts, bash scripts to perform several utility tasks such as preparing or running the benchmarks.

Coq Formalization. The recommended way to build the modified Jasmin compiler is using nix. To build the formalization and test it works, run the following in the formalization directory:

```
$ nix-shell
$ dune build
```

Jasmin. To build Jasmin using nix, run the following commands in the jasmin directory:

```
$ nix-shell
$ make
$ export JASMIN=$PWD/compiler/jasminc
```

To perform a basic test, run

```
$ $JASMIN -help
```

libjade. To build and perform a basic test, run

```
$ make -C sslh_rsb/src -j$(nproc) libjade.a
```

You can now use the library, `sslh_rsb/src/libjade.a`, and the corresponding header file, `sslh_rsb/src/libjade.h` in your projects.

A.5 Experiment workflow

Before running the benchmarks, ensure that the machine is configured for this purpose to get stable results. Run:

```
$ ./scripts/bench-prepare
```

To run the complete benchmark setup, run

```
$ cd bench-prepared/
$ ./bench-run
```

A.6 Evaluation and expected results

To include the benchmark results into the tables/main.pdf file (where CPU is the name of your CPU):

```
$ cp -r results/ ../tables/data/raw/CPU
$ make -C ../tables/ all
```

Note: the chosen name will be used in a `\label{(...)CPU(...)}` \LaTeX command for the generated table, characters restrictions must be considered (e.g., avoid special characters).

The main.pdf file should now contain the new tables.