Protecting cryptographic code against Spectre-RSB
(and, in fact, all known Spectre variants)

Santiago Arranz Olmos[1], Gilles Barthe[2]
Chitchanok Chuengsatiansup[3], Benjamin Grégoire[4], Vincent Laporte[5]
Tiago Oliveira[6], Peter Schwabe[7], Yuval Yarom[8], Zhiyuan Zhang[9]

MPI-SP, Bochum, Germany
Inria, Sophia Antipolis, France
SandboxAQ, USA
Radboud University, Nijmegen, The Netherlands
Ruhr University Bochum, Bochum, Germany
IMDEA Software Institute, Madrid, Spain
University of Melbourne, Melbourne, Australia
Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France

[1]santiago.arranz-olmos@mpi-sp.org [2]gilles.barthe@mpi-sp.org
[3]c.chuengsatiansup@unimelb.edu.au [4]benjamin.gregoire@inria.fr
[5]vincent.laporte@inria.fr [6]tiago.oliveira@sandboxquantum.com
[7]peter@cryptojedi.org [8]yuval.yarom@rub.de [9]zhiyuanz5@student.unimelb.edu.au

***Abstract*—It is fundamental that executing cryptographic software must not leak secrets through side-channels. For software-visible side-channels, it was long believed that “constant-time” programming would be sufficient as a systematic countermeasure. However, this belief was shattered in 2018 by attacks exploiting speculative execution—so called Spectre attacks.**

Recent work shows that language support suffices to protect cryptographic code with minimal overhead against one class of such attacks, Spectre v1, but leaves an open question of whether this result can be extended to also cover other classes of Spectre attacks.

In this paper, we answer this question in the affirmative: We design, validate, implement, and verify an approach to protect cryptographic implementations against all known classes of Spectre attacks—the main challenge in this endeavor is attacks exploiting the return stack buffer, which are known as Spectre-RSB. Our approach combines a new value-dependent information-flow type system that enforces speculative constant-time in an idealized model of transient execution and a compiler transformation that realizes this idealized model on the generated low-level code. Using the Coq proof assistant, we prove that the type system is sound with respect to the idealized semantics and that the compiler transformation preserves speculative constant-time.

We implement our approach in the Jasmin framework for high-assurance cryptography and demonstrate that the overhead incurred by full 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.

1. Introduction

In this paper we present techniques to systematically protect high-performance cryptographic software against all known classes of Spectre attacks. Our work uses the computer-aided cryptography paradigm [12], i.e., we employ methodologies and tools from formal methods to build very efficient and formally verified cryptographic software. More specifically, we build our solution as part of Jasmin [4], [7], a programming language and framework that has been used to produce highly optimized and machine checked implementations of symmetric cryptography [7], elliptic-curve cryptography [4], hash functions [6] and very recently also post-quantum cryptography [5]. The Jasmin compiler is formally proven in Coq to preserve semantics, and offers tools to ensure properties relating to implementation security like memory safety, thread safety, and absence of secret-dependent branches and secretly indexed memory access; and furthermore offers an interface to the EasyCrypt [26] interactive theorem prover for proofs of functional correctness of implementations, that can be further connected to computer-verified reductionist cryptographic proofs of security, as, for instance, in [13], [14], [8].

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 secure against a specific class of Spectre attacks, namely attacks exploiting misspeculated conditional branches (aka Spectre-v1 or Spectre-PHT attacks). 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) [21] as proposed in [39], incurring a remarkably small overhead of (typically) below 1%.

However, as that paper addresses only one class of Spectre attacks, the conclusion expresses hope that the work may serve as *"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 other classes of Spectre attacks. We then identify one major remaining challenge in protecting against *all* known classes of Spectre attacks. This remaining class is Spectre-RSB, i.e., attacks exploiting the return stack buffer.

We then present the main contribution of our paper, namely 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) with program transformation. First, we replace calls and returns by conditional *direct* jumps, a transformation we call *return table insertion*. This transformation, which is inspired from prior work on return-oriented programming (ROP) [36], removes all Spectre-RSB gadgets. Second, we instrument programs with (an enhanced set of) language constructs to enforce selSLH on the source program (with calls). The combination of program transformation and selective speculative load hardening guarantees that transformed programs are speculative constant-time [23], i.e., do not leak secrets through timing even during speculative execution.

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 to eliminate sources of speculative leakage through masking. 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 speculative constant-time programs. Our implementation addresses practical issues ignored 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 on a set of Jasmin implementations of cryptographic algorithms. These routines are derived from existing implementations from [9] that are already protected against Spectre-PHT. 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 [18].

Artifacts. We produce three artifacts: the Coq formalization, a new version of the Jasmin framework, and the libjade implementations. All artifacts are submitted as supplementary material and will be made publicly available. Items marked with 🐓 can be found in the Coq formalization. The artifacts are in https://artifacts.formosa-crypto.org/data/rsbsecure.tar.bz2.

2. Background

As a starting point of this work we consider cryptographic software following the "constant-time" (CT) paradigm, i.e., software that, in sequential execution, systematically avoids data flow from secrets into memory addresses or branch conditions. The CT paradigm is widely regarded as a standard baseline defense mechanism against timing attacks [28]. The motivation for this paradigm is a leakage model that captures most traditional timing attacks by leaking the trace of all accessed memory locations and the complete control flow of the program. The goal of this paper is to systematically avoid leakage of secret data in this leakage model *also during speculative execution*. We specifically aim to protect cryptographic software in the Jasmin framework for high-assurance cryptography [4].

In Jasmin, the constant-time paradigm is enforced through an information-flow type system on source level. In short, all variables are typed as either secret or public and the type system enforces that operations taking secret inputs also produce secret outputs. Memory addresses and branch conditions have to be public. Preservation of the constant-time property through compilation has been formally proven in [15].

Spectre attacks, i.e., attacks that exploit leakage of secrets during speculative execution, are commonly classified based on the type of speculation. Let us briefly recall the nomenclature of Spectre attacks, based on [29] and [20], and review how Jasmin implementations protect against them.

Spectre v1 (PHT). Spectre v1 attacks [29, Sec. IV]—also called "Spectre-PHT" because they employ the *Pattern History Table*—exploit speculative execution following a mispredicted conditional branch. The simplest Spectre v1 gadget is a speculative bounds-check bypass, which results in speculatively reading outside a buffer. An efficient approach to protect CT cryptographic code in Jasmin against Spectre v1 was recently proposed in [9]. The high-level idea is to extend Jasmin's CT type system to systematically avoid secretly indexed memory access and secret branch conditions even under speculative execution following a mispredicted conditional branch. The type system is extended by an additional security level, transient, for variables that are always public in sequential (non-mispredicted) 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: inserting an lfence instruction introduces a speculation barrier and thus turns all transient variables to public; alternatively, a programmer can choose to lower a single transient variable to public by masking the variable with a misspeculation flag that is updated through arithmetic instructions at each branch. This second technique is selective speculative load hardening [39]. It is supported in Jasmin through three instructions:

- `init_msf()` inserts a speculation fence and sets a special register msf to the neutral value of masking NOMASK. We will use this register to track speculation and call it the *misspeculation flag* (MSF).[1]
- `update_msf(`e`)` conditionally updates the misspeculation flag msf to MASK, depending on the boolean expression e; it is essentially $\mathit{msf} = e \ ? \ \mathit{msf} \ : \ \mathsf{MASK}$, implemented as an atomic conditional move instruction CMOV (which is not subject to the PHT speculation mechanism). We use this instruction after branches to check if execution has proceeded sequentially (and therefore the MSF should not be modified) or if the branch predictor has misspeculated (and therefore the MSF should become MASK).
- $x =$ `protect(`y`)` protects, i.e., masks, register x conditioned on the value of the misspeculation flag msf; that is, 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. This instruction is used to lower the type of x from transient to public.

Spectre v2 (BTB). Spectre v2 attacks [29, Sec. V]—also called "Spectre-BTB" because they employ the *Branch Target Buffer*—exploit speculative execution following a mispredicted indirect branch. A variant is BHI (for Branch History Injection), also called Spectre-BHB [11].

The Jasmin language does not support indirect branches, which means that Jasmin programs are inherently protected against Spectre v2.

Spectre v4 (STL & PSF). We define the class of Spectre v4 attacks as attacks where load instructions retrieve speculative data. This can be the case either because not all earlier store addresses have been resolved (speculative store bypass, STL) [29, Sec. VI], or because the CPU wrongly predicted a store-to-load forward (predictive store forwarding, PSF) [3]. These attacks are prevented by disabling the respective speculation, which, on Intel and AMD CPUs, is accomplished by setting the SSBD (speculative store bypass disable) flag in the CPU. As we will see in Section 8, the performance impact of setting this flag on cryptographic code is very small.

The one remaining class of Spectre attacks exploits the return stack buffer (RSB), so for the remainder of this paper we will assume that Spectre v2 and v4 attacks are addressed and focus on countermeasures against Spectre-RSB and their integration with countermeasures against Spectre v1.

3. Overview

Spectre-RSB was discovered independently by [30] and [32]. It exploits speculative execution when returning from a function; this speculation uses entries in the return stack buffer (RSB), hence the name. As the RSB is shared between processes, it can be trained by an attacker to direct speculative execution after a RET to *anywhere in the program*. Consider, e.g., the program in Figure 1a, which is vulnerable because when we call id (with sec as an argument), speculative execution may return to the leak(x) instruction and leak the secret.

Our countermeasure against Spectre-RSB uses two ingredients. The first is to use a program transformation that replaces all calls with direct jumps and all returns with return tables, i.e., nested conditional direct jumps. Figure 1b shows the transformation of the source program in Figure 1a. 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-PHT 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. At first glance, it might look like we have gained nothing. However, the transformation 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. This idea of rewriting RET instructions to a sequence of direct conditional jumps is not new; it was mentioned (but not implemented) as a potential Spectre-RSB countermeasure in [16, Sec. 7]. The second ingredient we use to protect the program is selective speculative load hardening. For a high-level description of selSLH and its implementation in Jasmin, see Section 2; for an example of its application to our simple example program, see Figure 1c.

We combine these two ingredients in a way that integrates well with the Jasmin workflow and is in spirit very

[1] We 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.

```
id {
    return
}

main {
    x = pub
    call id
    leak(x)
    x = sec
    call id
    ... // do not leak x
}
```

(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.

```
id:
    if ra = 0 jump $\ell_0$
    jump $\ell_1$

main:
    x = pub
    ra = 0
    jump id
$\ell_0$: leak(x)
    x = sec
    ra = 1
    jump id
$\ell_1$: ... // do not leak x
```

(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.

```
id:
    if ra = 0 jump $\ell_0$
    jump $\ell_1$

main:
    init_msf()
    x = pub
    ra = 0
    jump id
$\ell_0$: update_msf(ra = 0)
    x = protect(x)
    leak(x)
    x = sec
    ra = 1
    jump id
$\ell_1$: ... // do not 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.

Figure 1: (a) Source program, (b) program with return tables, (c) protected program with return tables.

similar to the type system presented in [9]: We perform security typing on source level *with* function returns, and apply the transform from RET instructions to nested conditional jumps *afterward*, during compilation. This, however, means that the speculative semantics and the type system from [9] are insufficient to capture all effects of speculative execution. Specifically, they only capture misspeculation of source level conditionals (i.e., if/else constructions) and while loops. For these, the control-flow graph of the program is the same for sequential and for speculative execution. This is different for function returns that have been transformed into sequences of branch instructions: in sequential execution, the control-flow graph has only one edge, to the *actual* call site, while under speculative execution it has edges to *every* call site of the function in the whole program. This is a situation that the type system from [9] does not handle.

We therefore first define a language featuring calls and returns in addition to conditional statements and loops, together with a speculative semantics that captures at source level the protections offered by return tables (Section 4). We then present a type system that enforces speculative constant-time under these semantics (Section 5) and a compilation scheme that realizes the semantics and preserves leakage (Section 6). After this, Section 7 discusses how we implement this type system and compilation scheme in Jasmin, and Section 8 how we use that to protect libjade with little overhead. Lastly, we compare and contrast with related work in Section 9, discuss limitations (Section 10) and conclude (Section 11).

4. 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.

4.1. Security model

As stated in Section 2, we are considering Spectre-RSB and Spectre-PHT attacks; however, as compilation transforms function returns to return tables, our security model does not aim to capture the full power of a Spectre-RSB attacker at source level. Opting for a weaker security model at source level is beneficial, because it minimizes the amount of protections that programmers need to write in their code, and leads to more efficient programs. Moreover, the choice of a weaker model has no negative security implication, as our method still ensures that compiled programs are protected against all Spectre attacks.

4.2. Syntax

For simplicity, we consider function calls without local variables, arguments, or return values. The syntax of expressions, instructions and code is as follows:

$$
\begin{aligned}
e &::= n \mid b \mid x \mid \texttt{op1}(e) \mid \texttt{op2}(e, e) \\
i &::= x = e \mid x = a[e] \mid a[e] = x \\
&\quad \mid \texttt{if } e \texttt{ then } c \texttt{ else } c \mid \texttt{while } e \texttt{ do } c \mid \texttt{call}_b\ f \\
&\quad \mid \texttt{init\_msf}() \mid \texttt{update\_msf}(e) \mid x = \texttt{protect}(x) \\
c &::= [] \mid i; c
\end{aligned}
$$

where n is an integer, b a boolean, x a register variable, and a an array variable. We assume that each array comes with its size $|a|$. The definitions are standard, except for the call

```
g {
    while x < 10 do {
        call⊤ f
        x = x + 1
    }
    call⊥ f
    x = 0
}
```

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

instruction and the selSLH instructions (which are described in Section 3). The $\texttt{call}_\top\ f$ instruction calls f and performs an MSF update on the register msf upon return. On the other hand, the instruction $\texttt{call}_\bot\ f$ is a usual assembly CALL f instruction, which does not update the misspeculation flag. We need this MSF update in call instructions because we will compile returns to tables of conditional branches, 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.

A program is a set of pairs of function names and code, i.e., $p \coloneqq \mathcal{P}(f \times c)$, where there is one distinguished pair that is the entry point. The entry point has no callers, and execution halts after reaching its return.

4.3. Semantics

To formalize our security model, we define for every function f its set of continuations $\mathcal{C}(f)$, consisting of triples (c, g, b), where g is a function that contains an instruction $\texttt{call}_b\ f$ and c is the code that remains to be executed after returning from the call. Intuitively, if the code of g is of the form $\ldots; \texttt{call}_b\ f; c$ then (c, g, b) is a continuation of f; however, some care is needed to deal with function calls within loops.

Let c be the body of function g in Figure 2. We note that there are two continuations of f in c: the first one is $(\mathsf{x = x + 1}; c,\ \mathsf{g},\ \top)$, 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 $(\mathsf{x = 0},\ \mathsf{g},\ \bot)$, i.e., when returning from f to the second call site we only need to execute the last assignment to x. We present the precise definition of continuations in Appendix A.

Directives use continuations to model the attacker's power to influence execution. 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 leakage from control flow and memory accesses. 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.

$$\begin{aligned}\mathsf{Dir} &::= \mathsf{step} \mid \mathsf{force}\ b \mid \mathsf{mem}\ a\ i \mid \mathsf{return}\ c\ f\ b\\ \mathsf{Obs} &::= \bullet \mid \mathsf{branch}\ b \mid \mathsf{addr}\ a\ i\end{aligned}$$

We define the single-step semantics of our language with a judgment $m \xrightarrow[d]{o} m'$ that expresses that the directive d makes the machine m step to m' and produce an observation o. Machines are 6-tuples $\langle c,\ f,\ cs,\ \rho,\ \mu,\ ms\rangle$ that consist of the code being executed, the name of the function being executed, the call stack, the register map, the memory, and the misspeculation status. 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. Since we will only deal with one program at a time, we will always leave the program implicit.

Figure 3 presents the semantics for loads, protects, conditionals, calls and returns. The N-LOAD rule is the usual rule for memory loads, where the expression needs to evaluate to an integer that is in bounds for the array. This rule has addr a i as an observation because memory accesses leak their addresses, and ignores its directive. On the other hand, the S-LOAD rule only applies if the access is out of bounds, and allows the attacker to read data from *any* array and index. We require the misspeculation status to be $\top$ because we assume the program to be sequentially safe.

The PROTECT rule shows how we can use the MSF to mask a value. If the variable msf is set to MASK, the result of a protection is a default value.

The COND rule shows that the attacker controls the execution of conditionals via the force b directive. The condition of the statement only plays a role in the observation, branch $[\![\, e \,]\!]_\rho$.

The CALL rule states that a function call puts the body and name of the callee as the code and function name under execution, and pushes the current 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 is discarded. This rule captures a misspeculation in the return table of a compiled function.

Figure 3 also defines multi-step semantics as the reflexive transitive closure of the single-step one, accumulating the directives and observations. The rest of the rules are similar, and shown in Appendix A.

4.4. Speculative constant-time

We now have all the ingredients to define *speculative constant-time*. The definition is parameterized by an equivalence relation on machines.

Definition 1 (Speculative constant-time, ϕ-SCT)**.** *Let ϕ be an equivalence relation on machines. A program p is speculative constant-time with respect to ϕ, denoted ϕ-SCT, if and only if all executions of machines that are related by ϕ following a given list of directives produce the same*

$$\frac{\llbracket e \rrbracket_\rho = i \qquad 0 \le i < |a| \qquad \mu(a,i) = v}{\langle x = a[e]; c,\ f,\ cs,\ \rho,\ \mu,\ ms\rangle \xrightarrow[\mathsf{mem}\ b\ j]{\mathsf{addr}\ a\ i} \langle c,\ f,\ cs,\ \rho[x \leftarrow v],\ \mu,\ ms\rangle}\ \textbf{N-LOAD}$$

$$\frac{\llbracket e \rrbracket_\rho = i \qquad \neg\,(0 \le i < |a|) \qquad 0 \le j < |b| \qquad \mu(b,j) = v}{\langle x = a[e]; c,\ f,\ cs,\ \rho,\ \mu,\ \top\rangle \xrightarrow[\mathsf{mem}\ b\ j]{\mathsf{addr}\ a\ i} \langle c,\ f,\ cs,\ \rho[x \leftarrow v],\ \mu,\ \top\rangle}\ \textbf{S-LOAD}$$

$$\frac{v = \mathsf{if}\ \rho(\mathit{msf}) = \mathsf{MASK}\ \mathsf{then}\ \mathsf{MASK}\ \mathsf{else}\ \rho(y)}{\langle x = \mathtt{protect}(y); c,\ f,\ cs,\ \rho,\ \mu,\ ms\rangle \xrightarrow[\mathsf{step}]{\bullet} \langle c,\ f,\ cs,\ \rho[x \leftarrow v],\ \mu,\ ms\rangle}\ \textbf{PROTECT}$$

$$\frac{\llbracket e \rrbracket_\rho = b'}{\langle \mathtt{if}\ e\ \mathtt{then}\ c_\top\ \mathtt{else}\ c_\bot; c,\ f,\ cs,\ \rho,\ \mu,\ ms\rangle \xrightarrow[\mathsf{force}\ b]{\mathsf{branch}\ b'} \langle c_b; c,\ f,\ cs,\ \rho,\ \mu,\ ms \vee (b \neq b')\rangle}\ \textbf{COND}$$

$$\frac{(g, c_g) \in p}{\langle \mathtt{call}_b\ g; c,\ f,\ cs,\ \rho,\ \mu,\ ms\rangle \xrightarrow[\mathsf{step}]{\bullet} \langle c_g,\ g,\ (c, f) :: cs,\ \rho,\ \mu,\ ms\rangle}\ \textbf{CALL}$$

$$\frac{}{\langle [],\ f,\ (c,g) :: cs,\ \rho,\ \mu,\ ms\rangle \xrightarrow[\mathsf{return}\ c\ g\ b]{\bullet} \langle c,\ g,\ cs,\ \rho,\ \mu,\ ms\rangle}\ \textbf{N-RET}$$

$$\frac{(c,g,b) \in \mathcal{C}(f) \qquad cs \neq (c,g) :: cs' \qquad \rho' = \mathsf{if}\ b\ \mathsf{then}\ \rho[\mathit{msf} \leftarrow \mathsf{MASK}]\ \mathsf{else}\ \rho}{\langle [],\ f,\ cs,\ \rho,\ \mu,\ ms\rangle \xrightarrow[\mathsf{return}\ c\ g\ b]{\bullet} \langle c,\ g,\ [],\ \rho',\ \mu,\ \top\rangle}\ \textbf{S-RET}$$

$$\frac{}{m \overset{[]}{\underset{[]}{\twoheadrightarrow}} m}\ \textbf{NIL} \qquad\qquad \frac{m \xrightarrow[d]{o} m' \qquad m' \overset{os}{\underset{ds}{\twoheadrightarrow}} m''}{m \overset{o::os}{\underset{d::ds}{\twoheadrightarrow}} m''}\ \textbf{CONS}$$

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

observations. That is, for every pair of related machines $m_1 \ \phi \ m_2$ *with executions* $m_i \overset{O_i}{\underset{D}{\twoheadrightarrow}} m_i'$ *we get* $O_1 = O_2$.

5. Type system

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

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 $\{\mathsf{L}, \mathsf{H}\}$ with $\mathsf{L} \le \mathsf{H}$, 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). 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 the maximum confidentiality of the data under all possible speculative executions)

$$\mathsf{level} ::= \mathsf{H} \mid \mathsf{L} \quad \mathsf{type} ::= \mathsf{level} \mid \alpha \quad \mathsf{stype} ::= \langle \mathsf{type}, \mathsf{level}\rangle$$

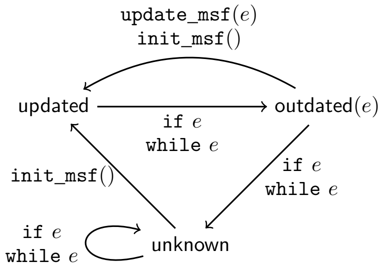

Figure 4: MSF type automaton.

and we write τ_n to refer to the normal (first) component, and τ_s for the speculative (second) component of the security type τ. Thus, a public variable has type $\langle \mathsf{L}, \mathsf{L}\rangle$, a secret one $\langle \mathsf{H}, \mathsf{H}\rangle$, and a transient one $\langle \mathsf{L}, \mathsf{H}\rangle$. Allowing polymorphism in the speculative component of a security type makes the type system unsound, see Appendix A for a detailed explanation.

5.1. Misspeculation type

Our type system also needs to keep track of the misspeculation flag to detect whether protections will be effective.

$$\mathsf{FV}(\Sigma) := \begin{cases} \mathsf{FV}(e) & \text{if } \Sigma \text{ is } \mathsf{outdated}(e) \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathsf{to\_lvl}(t) := \begin{cases} \mathsf{L} & \text{if } t \text{ is } \mathsf{L} \\ \mathsf{H} & \text{otherwise} \end{cases}$$

$$\Sigma|_e := \begin{cases} \mathsf{outdated}(e) & \text{if } \Sigma \text{ is updated} \\ \mathsf{unknown} & \text{otherwise} \end{cases}$$

$$\Sigma \subseteq \Sigma' := \Sigma = \mathsf{unknown} \vee \Sigma = \Sigma'$$

Figure 5: 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. Note that $\Sigma|_e$ corresponds to the `if` e and `while` e arrows of the MSF type automaton (Figure 4). The order for MSF types is flat with unknown as the bottom element.

For this, we define the MSF type

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

Figure 4 gives an intuition of the behavior of these types. The MSF type unknown expresses that we do not know whether the machine 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, we transition to outdated(e), and we need to execute an `update_msf`(e) to recover the MSF.[2]

The typing judgment is of the form $\Gamma, \Sigma \vdash c : \Gamma', \Sigma'$, where Γ and Γ' are mappings from register and array variables to security types (which have a normal and a speculative component), Σ and Σ' are a MSF types, and c is code. Figure 6 presents the typing rules, with some auxiliary definitions in Figure 5.

Type checking requires a static signature for all functions of the program, which associates each function name f with a signature $\Sigma_f, \Gamma_f \to \Sigma'_f, \Gamma'_f$ 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 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 normal type from the array, but since the index might be speculatively out of bounds, we need to overapproximate the speculative type as H. Lastly, the STORE rule also ensures that the index is public, and we update the normal type of the array to the normal type of x, but similarly to the case for load, the index might be out of bounds, so we need to update the speculative types of all arrays.

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 counterpart, overapproximating polymorphic type variables with H: $\langle \mathsf{L}, t\rangle$ goes to $\langle \mathsf{L}, \mathsf{L}\rangle$, $\langle \mathsf{H}, t\rangle$ goes to $\langle \mathsf{H}, \mathsf{H}\rangle$, and $\langle \alpha, t\rangle$ goes to $\langle \alpha, \mathsf{H}\rangle$. We define this $\mathsf{to\_lvl}(\cdot)$ overapproximation in Figure 5. Secondly, the UPDATE-MSF rule expects the MSF type to be outdated, 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 5. This means that 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 rule ensures 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 we want the type to be updated, we need the parameter to be $\top$ and the output MSF type of the function to be updated. This rule allows instantiating the type variables in Γ with θ.

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

5.2. 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 machines, which holds when two machines coincide on their public values.

We interpret types as relations between values

$$x =_\tau y := \tau = \mathsf{L} \implies x = y$$
$$\rho =_f \rho' := \forall x.\, \rho(x) =_{f(x)} \rho'(x)$$
$$\mu =_f \mu' := \forall a\, i.\, \mu(a, i) =_{f(a)} \mu'(a, i)$$

where τ can be a type variable: we show that this is not a problem since a program that is typable under Γ is also typable under $\theta(\Gamma)$ for every instantiation θ, in particular when $\theta(\tau)$ is L 🐓.

We use these notions to define the indistinguishability relation between machines, extending [9].

[2] For clarity, we depart from the notation in [9]: what that work denotes ms we write as updated, and for $\mathsf{ms}|_\mathsf{e}$ we write outdated(e).

$$\frac{\Gamma \vdash e : \tau \qquad x \notin \mathsf{FV}(\Sigma)}{\Sigma, \Gamma \vdash x = e : \Sigma, \Gamma[x \leftarrow \tau]}\ \textbf{ASSIGN}$$

$$\frac{\Gamma \vdash e : \mathsf{L} \qquad x \notin \mathsf{FV}(\Sigma)}{\Sigma, \Gamma \vdash x = a[e] : \Sigma, \Gamma[x \leftarrow \langle \Gamma(a)_n, \mathsf{H} \rangle]}\ \textbf{LOAD}$$

$$\frac{\begin{array}{cc} \Gamma \leq \Gamma' & \Gamma(x) \leq \Gamma'(a) \\ \Gamma \vdash e : \mathsf{L} & \forall b \neq a.\, \Gamma(x)_s \leq \Gamma'(b)_s \end{array}}{\Sigma, \Gamma \vdash a[e] = x : \Sigma, \Gamma'}\ \textbf{STORE}$$

$$\frac{\Gamma \vdash e : \mathsf{L} \qquad \Sigma|_e, \Gamma \vdash c : \Sigma, \Gamma}{\Sigma, \Gamma \vdash \texttt{while}\ e\ \texttt{do}\ c : \Sigma|_{!e}, \Gamma}\ \textbf{WHILE}$$

$$\frac{\Sigma, \Gamma \vdash i : \Sigma_i, \Gamma_i \qquad \Sigma_i, \Gamma_i \vdash c : \Sigma', \Gamma'}{\Sigma, \Gamma \vdash i; c : \Sigma', \Gamma'}\ \textbf{CONS}$$

$$\frac{}{\Sigma, \Gamma \vdash [] : \Sigma, \Gamma}\ \textbf{NIL}$$

$$\frac{\Gamma' = \{\, v : \langle \Gamma(v)_n, \mathsf{to\_lvl}(\Gamma(v)_n) \rangle \mid \text{for each } v \,\}}{\Sigma, \Gamma \vdash \texttt{init\_msf}() : \mathsf{updated}, \Gamma'}\ \textbf{INIT-MSF}$$

$$\frac{}{\mathsf{outdated}(e), \Gamma \vdash \texttt{update\_msf}(e) : \mathsf{updated}, \Gamma}\ \textbf{UPDATE-MSF}$$

$$\frac{\tau = \langle \Gamma(x)_n, \mathsf{to\_lvl}(\Gamma(x)_n) \rangle}{\mathsf{updated}, \Gamma \vdash y = \texttt{protect}(x) : \mathsf{updated}, \Gamma[y \leftarrow \tau]}\ \textbf{PROTECT}$$

$$\frac{\Gamma \vdash e : \mathsf{L} \qquad \Sigma|_e, \Gamma \vdash c_\top : \Sigma', \Gamma' \qquad \Sigma|_{!e}, \Gamma \vdash c_\bot : \Sigma', \Gamma'}{\Sigma, \Gamma \vdash \texttt{if}\ e\ \texttt{then}\ c_\top\ \texttt{else}\ c_\bot : \Sigma', \Gamma'}\ \textbf{COND}$$

$$\frac{\begin{array}{c} Sig(f) = \Sigma_f, \Gamma_f \to \Sigma'_f, \Gamma'_f \\ \text{if } b \text{ then } \Sigma' = \mathsf{unknown} \text{ else } \Sigma' = \Sigma'_f = \mathsf{updated} \end{array}}{\Sigma_f, \theta(\Gamma_f) \vdash \texttt{call}_b\ f : \Sigma', \theta(\Gamma'_f)}\ \textbf{CALL}$$

$$\frac{\Sigma_0 \subseteq \Sigma \qquad \Sigma' \subseteq \Sigma'_0 \qquad \Gamma \leq \Gamma_0 \qquad \Gamma'_0 \leq \Gamma' \qquad \Sigma_0, \Gamma_0 \vdash c : \Sigma'_0, \Gamma'_0}{\Sigma, \Gamma \vdash c : \Sigma', \Gamma'}\ \textbf{WEAK}$$

Figure 6: Type system. Here $\theta : \mathsf{typeVar} \to \mathsf{level}$ is an instanciation of type variables and Sig is the signature for functions of the program. We define $\Gamma \leq \Gamma'$ pointwise.

Definition 2 (Machine indistinguishability). *Two machines are indistinguishable under Γ, denoted $\langle c, f, cs, \rho, \mu, ms \rangle =_\Gamma \langle c', f', cs', \rho', \mu', ms' \rangle$, if their code, function name, call stack and misspeculation status are the same*

$$c = c' \quad f = f' \quad cs = cs' \quad ms = ms'$$

they coincide on their speculatively public parts

$$\rho =_{\Gamma_s} \rho' \qquad \mu =_{\Gamma_s} \mu'$$

and they coincide on their sequentially public parts if they are not misspeculating

$$ms \vee (\rho =_{\Gamma_n} \rho' \wedge \mu =_{\Gamma_n} \mu')$$

The relation we will use for our main theorem states that the machines are indistinguishable and are executing the entry point of the program

Definition 3 (Initial machine). *An initial machine for a program p is of the form $\langle c, f, [], \rho, \mu, ms \rangle$, where c is the code for the entry function and f its name.*

Definition 4 (Initial SCT relation). *We will say that two machines are in the relation ϕ_Γ if they are initial and indistinguishable under Γ.*

Theorem 1 (Soundness). *If a program is safe and typable, and the initial type for its entry point is* $(\mathsf{unknown}, \Gamma)$*, then it is ϕ_Γ-SCT (that is, every pair of initial machines that coincide on their public parts and their misspeculation statuses produce the same observations, under the assumption that the directives are the same).*

Informally, the notion of *safety* is that any reachable state (starting from an initial state of the program) is either misspeculating, final (i.e., the program has been executed fully), or there is a directive that allows a step of execution. Remark that the notion imposes the usual conditions only when the reached state is not misspeculating, i.e., under sequential execution.

The proof of the soundness theorem needs two important lemmas, for single and multi-step execution soundness. These say that if a machine is well-typed, which intuitively means that the code being executed and the contents of the call stack are typable, the resulting machine after one (resp. many) execution step (resp. steps) is also well-typed and the observations are the same for all indistinguishable machines. Naturally we also need to show that initial machines are well-typed.

The definition of SCT states that both related machines must step under the same directives. At first sight, if one of the machines 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 machine can step under directives D, then all indistinguishable machines can step under the same directives.

6. Return table insertion

In this section we present return table insertion and show that this transformation 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., conditional, while, and function calls). The linear language has only two control flow constructs, conditional and unconditional direct jumps,

$$
\begin{aligned}
&(\!|\, \texttt{call}_\top\ f \,|\!) \coloneqq \\
&\qquad\qquad ra_f = \ell_{ret} \\
&\qquad\qquad \texttt{jump}\ f \\
&\qquad \ell_{ret}:\ \ \texttt{update\_msf}(ra_f = \ell_{ret})
\end{aligned}
$$

$$
\begin{aligned}
(\!|\, \{\ell_r\} \,|\!)^f_{\mathsf{rettbl}} &\coloneqq \texttt{jump}\ \ell_r \\
(\!|\, \{\ell_r\} \cup \ell^* \,|\!)^f_{\mathsf{rettbl}} &\coloneqq \texttt{if}\ ra_f = \ell_r\ \texttt{jump}\ \ell_r;\ (\!|\, \ell^* \,|\!)^f_{\mathsf{rettbl}}
\end{aligned}
$$

Figure 7: Compilation of function calls and return tables. We write $(\!| c |\!)$ for the compilation of c, and $(\!| \ell^* |\!)^f_{\mathsf{rettbl}}$ 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.

and programs are lists of labeled instructions:

$$
\begin{aligned}
li &::= x = e \mid x = a[e] \mid a[e] = x \\
&\quad \mid \texttt{init\_msf}() \mid \texttt{update\_msf}(e) \mid x = \texttt{protect}(x) \\
&\quad \mid \texttt{jump}\ \ell \mid \texttt{if}\ e\ \texttt{jump}\ \ell \\
lc &::= [] \mid (\ell : li); lc
\end{aligned}
$$

Linear machines, while similar to source ones, are 4 tuples $\langle \ell, \rho, \mu, ms \rangle$, where instead of code being executed we have the *program counter*, which is a label pointing to an instruction in the program. If t is a linear machine, t_{pc} denotes its program counter. The operational semantics of this language is standard, with similar directive and observation behavior as the source. We show the precise rules in Appendix B.

Figure 7 presents our compilation scheme for non-recursive function calls. Compiling assignments, memory and selSLH instructions is trivial since the two 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 7.

Figure 7 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. We show rigorous definitions for the compilation scheme, including basic instructions, in Appendix B.

Compiling a program p, denoted $(\!| p |\!)$, 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 4, the entry point has no callers, and thus no return table.

Preservation of SCT. We use a similar definition of SCT to the one in Section 4.4 for linear programs and define a similar relation ϕ^l_Γ which expresses the same indistinguishability relation between states. Note that while the source relation requires that the code of the machines are equal, the linear relation requires that the program counters point to the same instruction. Our compilation scheme preserves SCT for typable source programs.

Theorem 2 (Preservation of SCT). *If p is typable then $(\!| p |\!)$ is ϕ^l_Γ-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 $T_{\mathsf{Dir}}(\cdot,\cdot) : \mathsf{label} \times \mathsf{Dir} \to \mathsf{Dir}^*$ that computes source directives given the program counter and a target directive, and a leakage transformer $T_{\mathsf{Obs}}(\cdot,\cdot) : \mathsf{label} \times \mathsf{Obs}^* \to \mathsf{Obs}$ that computes target leakage given the program counter and source leakage. For single step executions, we prove

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

Here the relation $s \sim t$ indicates that the linear machine t is the compilation of the source machine s. Note that $T_{\mathsf{Dir}}(t_{pc}, d)$ can be empty, which corresponds 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.

Extending this to multi-step executions allows us to prove Theorem 2: given two target executions with leakages O_t and O'_t, we know that there exist source executions with leakages O_s and O'_s such that $T^*_{\mathsf{Obs}}(t_{pc}, O_s) = O_t$ and $T^*_{\mathsf{Obs}}(t_{pc}, O'_s) = O'_t$. By soundness of the source type system we know $O_s = O'_s$, and therefore $O_t = O'_t$.

Let us now give an intuition for the transformers for directives and observations. Recall the example in Figure 1a and its compilation in Figure 1b. In Figure 8 we present the attack discussed before, showing the current instruction and directive in the second and third columns. Recall that the attack consists of executing sequentially until the second call to id, and when we return for the second time, we mispredict and go to the first call site (we mark sequential execution with thin blue directives and misspeculating execution with **thick red** directives). The transformers tell us how, from the current program counter pc and the target directive d, we can produce a list of source directives $T_{\mathsf{Dir}}(pc, d)$, such that their leakage O_s predicts the target observation o. We have two interesting cases in this execution: first, instructions that store return addresses are not present in the source, so the source directives are $[]$, i.e., the source machine does not step (we mark these with $\rightsquigarrow$). In these cases the target observation is always $\bullet$. Second, return tables consist of conditional branches, so we need to translate force directives into return directives (we mark these with $\longrightarrow$). The leakage here depends on the value of ra, which we can compute since

	pc	$(\!| p |\!)_{pc}$	d	$T_{\mathsf{Dir}}(pc, d)$	O_s	$T_{\mathsf{Obs}}(pc, O_s) = o$
	main:1	x = pub	step	[step]	[•]	•
⇝	main:2	ra = 0	step	[]	[]	•
	main:3	jump id	step	[step]	[•]	•
→	id:1	if ra = 0 jump ℓ_0	force ⊤	[return κ]	[•]	branch ⊤
	main:4	leak(x)	step	[step]	[pub]	pub
	main:5	x = sec	step	[step]	[•]	•
⇝	main:6	ra = 1	step	[]	[]	•
	main:7	jump id	step	[step]	[•]	•
→	id:1	if ra = 0 jump ℓ_0	**force ⊤**	[return κ]	[•]	branch ⊥
	main:4	leak(x)	**step**	[step]	[sec]	sec

Figure 8: The attack from the example in Figure 1b. Here κ is the continuation after the first call to id, that is (leak(x); . . . , main, ⊥).

the CFG is public. We present the precise definition of the directive and observation transformers in Appendix B.

7. Implementation in the Jasmin compiler

In this section, we describe how we implemented the type system and compilation scheme, presented in Sections 5 and 6, in Jasmin [4], [7], a low-level programming language for high-assurance high-speed cryptography. The Jasmin language is designed to write efficient and verification-friendly assembly code. It directly exposes assembly instructions to the programmer, except for jumps, since control flow in Jasmin consists of if statements, while loops, for loops (which are always unrolled), and function calls. Jasmin also provides several zero-cost abstractions over assembly, such as register and stack variables, that the compiler allocates to architectural registers and stack offsets.

The Jasmin compiler consists of around thirty passes and is written and verified in Coq; it outputs AMD64 (i.e., x86-64) assembly and has experimental support for ARMv7-M. Some exemplary passes are: Inlining, which removes function calls annotated as #inline by the bodies of their functions; Unrolling, which unrolls all for loops; Stack Allocation, which replaces operations on stack variables by memory accesses; Register Allocation, which renames and reuses variables to require only architectural registers; and Linearization, which replaces structured control flow by jumps. For the present work, we introduce a new pass after Linearization that replaces all CALL and RET instructions with direct jumps and return tables.

The Jasmin compiler can also produce a representation of the program to reason in the EasyCrypt [26] proof assistant. Finally, Jasmin comes with three static analyzers: 1) a safety checker that attempts to prove that a program terminates, accesses only valid memory, and produces no arithmetic errors like division by zero; 2) a constant-time checker that ensures that a program does not leak secrets through timing side-channels in sequential execution; and 3) a speculative constant-time checker that ensures that a program is constant-time even under speculative execution.

Changes to the SCT checker. Jasmin provides an automatic checker for Spectre-PHT vulnerabilities that takes into account selSLH instructions. We extend this checker to consider also Spectre-RSB, following the type system from Section 5. 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 5 and consider all variables as speculatively secret after a function call. Secondly, we introduce an annotation for function calls corresponding to the boolean parameter of call instructions, where ⊥ is the default and ⊤ requires an annotation

call$_\perp$ f	⇒	x = f(y);
call$_\top$ f	⇒	#update_after_call x, msf = f(y, msf);

Note that in Jasmin, the MSF variable is explicit, and, as mentioned in Section 2, in both the Coq development and the Jasmin compiler, we can keep the MSF in any location (registers, MMX registers, or on the stack), which allows us to spill it when register pressure is high.

Changes to the compiler. We adapt the Jasmin compiler to use direct jumps instead of CALL and RET instructions. As discussed in Section 6, 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, we 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 9, where at the return site ℓ we can reuse the EQ condition in the MSF update without performing another comparison.

The compiler is flexible in passing return addresses

```
callee:
    ...
    CMP ra, ℓ
    JMPeq ℓ
    JMPlt LT_branch
    // GT_branch
```

```
caller:
    ...
    MOV ℓ, ra
    JMP callee
ℓ:  MOV MASK, tmp
    CMOVne tmp, msf
```

Figure 9: A return table implemented as a tree and a return site that reuses the comparison made in the table.

```
f:
    ...
    ra_g = f_0
    jump g
f_0: ...
    if ra_f = ℓ jump ℓ
    jump ℓ′
```

```
g:
    ...
    if ra_g = f_0 jump f_0
    jump evil_0
```

```
evil:
    ra_f = secret
    ra_g = evil_0
    jump g
evil_0: ...
```

Figure 10: How a secret may leak as a return tag.

in different ways. For libjade, using MMX registers was the best option. 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 since 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 10 shows how naively passing the return address is insecure. Remark that 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.

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, or more generally reentrant, 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.

8. Evaluation

This section overviews the changes to libjade and evaluates the computational cost of applying Spectre-RSB countermeasures. Libjade is a high-assurance cryptographic library written in Jasmin and extended by [9] to be Spectre-PHT protected. The present work uses the artifact from that work as our starting point, which contains the constant-time implementations (without countermeasures against any Spectre attack) and the corresponding Spectre-PHT protected implementations.

8.1. 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-PHT protected version. Most of the changes were in the context of Kyber, and no implementations other than Kyber 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. The rejection sampling algorithm is the main reason for the difference in the number of call sites between Kyber512 and Kyber768 (it accounts for six call sites). Very briefly and informally, Kyber512 samples a 2×2 matrix of polynomials, while Kyber768 a 3×3 one. AVX2 256-bit registers and instructions work nicely to perform four samplings at a time, which means that Kyber768 follows the same pattern as Kyber512 for the first eight polynomials and then needs to sample an extra one separately.

Sometimes, we can avoid keeping an MSF by applying one of four different strategies while still protecting our code; Kyber, in both versions, has examples of all four. First, we inline function calls when 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 is minor; 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, $\alpha \to \alpha$, 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 weaker 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.

We note that 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.

8.2. Performance of libjade

Table 1 reports benchmarks of highly optimized implementations of various cryptographic primitives in libjade with different Spectre protections. The reported benchmarks are median cycle counts 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 machine is running Debian 6.1.76, and we compiled our benchmarking code using GCC 12.2.

For each primitive, the leftmost ("plain") cycle count is the baseline CT implementation without any 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 as described in this paper ("+SSBD+v1+RSB").

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 lfence; this is consistent with the observations reported in [9].

For the elliptic-curve Diffie-Hellman key exchange X25519 [17] 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 measurements 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. As explained above, given that the generation of a 3×3 matrix in Kyber768 does not vectorize quite as straightforwardly as the generation of a 2×2 matrix in Kyber512, it is also expected that the overhead for Kyber768 is higher than for Kyber512. The keypair operation is surprising: it has the smallest overhead in Kyber512 but the largest in Kyber768.

9. Related work

Return tables in optimizing compilers. Calder and Grunwald [19] show that return tables can improve performance in object-oriented programs. Yang, Cooprider, and Regehr [43] show that return tables can reduce RAM usage in embedded code. Both transformations keep some indirect jumps, since their goal is efficiency, and thus are inadequate as mitigations against Spectre.

ROP countermeasures. Return-oriented programming (ROP) [37] is an exploitation technique that targets return instructions to force program execution to jump to arbitrary program points. In contrast to Spectre-RSB, ROP does not exploit speculative execution. There are many countermeasures against ROP. Our work is closely related to countermeasures that remove calls and returns [35], [31] 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 [20], [22] and focus on closely related work.

Swivel [34] is a software-only compiler framework (with a hardware-assisted variant) for WebAssembly that tackles Spectre-PHT, Spectre-BTB, and Spectre-RSB. Similarly to Venkman [38], it enforces a coarse-grained CFI under speculation by starting from a clean 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 with 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 [33] is a comprehensive approach to protect programs against all known Spectre attacks. Serberus uses control flow integrity (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 hardware mechanisms, e.g., Intel CET [1] and DOIT [2], and requires the operating system to perform RSB filling on context

TABLE 1: libjade benchmarks on Intel Core i7 11700K (most optimized implementation of each primitive). “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.	Op.	plain	+SSBD	+SSBD+v1	+SSBD+v1+RSB	increase
ChaCha20	avx2	128 B	344	344	398	398	15.70%
	avx2	128 B xor	350	350	402	400	14.29%
	avx2	1 KiB	1198	1202	1244	1246	4.01%
	avx2	1 KiB xor	1208	1212	1248	1250	3.48%
	avx2	16 KiB	19040	19052	19066	19068	0.15%
	avx2	16 KiB xor	19070	19086	19096	19110	0.21%
Poly1305	avx2	128 B	138	142	182	180	30.43%
	avx2	128 B verif	142	146	180	178	25.35%
	avx2	1 KiB	670	672	720	718	7.16%
	avx2	1 KiB verif	674	676	726	724	7.42%
	avx2	16 KiB	8942	8948	8990	8986	0.49%
	avx2	16 KiB verif	8942	8984	8984	8984	0.47%
XSalsa20Poly1305	avx2	128 B	1206	1212	1250	1246	3.32%
	avx2	128 B open	1964	1970	2044	2046	4.18%
	avx2	1 KiB	3140	3142	3190	3188	1.53%
	avx2	1 KiB open	3900	3904	3988	3988	2.26%
	avx2	16 KiB	32598	32574	32604	32602	0.01%
	avx2	16 KiB open	33292	33274	33358	33362	0.21%
X25519	mulx	smult	102848	104150	104424	104428	1.54%
Kyber512	avx2	keypair	27676	28106	28040	28090	1.50%
	avx2	enc	37050	38332	38876	38792	4.70%
	avx2	dec	29302	30444	30590	30714	4.82%
Kyber768	avx2	keypair	43432	45708	45860	46548	7.17%
	avx2	enc	57006	59316	60028	60674	6.43%
	avx2	dec	46138	48418	48532	49294	6.84%

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

Retpoline [27] is a software-based countermeasure against Spectre-BTB—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 [41] show that the assumptions that retpoline relies on are incorrect. Switchpoline [16] is a software-based countermeasure that replaces indirect jumps with direct ones, and a JIT compiler in some cases. Although Switchpoline targets Spectre-BTB in ARM, the transformation is very similar to ours. A critical difference between Switchpoline and our approach is that they do not consider how to combine their transformation with efficient countermeasures to protect programs. Furthermore, our proof shows that return tables introduce leakage, which needs mitigation.

Other attacks and countermeasures. There are many micro-architectural attacks beyond Spectre, see e.g., [40], [42], [24]. There exist initial efforts to develop formal foundations, verification techniques and countermeasures against these attacks, see e.g., [25]. Extending our approach to integrate these attacks and countermeasures is an exciting direction for future work.

10. Limitations

One limitation of our approach is that it applies only to full programs, because an (unprotected) external function call can exploit RSB to bypass protections. This limitation is common to other approaches, including Serberus [33], Switchpoline [16], Swivel [34], and Venkman [38].

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.

11. Conclusion

We have proposed an approach to protect Spectre programs against all known forms of Spectre attacks. An important direction for future work is to reduce the Trusted Computing Base of our extension by using the formalization presented in this work to prove in Coq that the Jasmin type system is sound, the return table insertion pass is correct, and the Jasmin compiler preserves speculative constant-time.

Our approach is currently limited to Jasmin programs. A pragmatic solution to carry 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

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] https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html.

[2] https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html.

[3] "Security analysis of AMD predictive store forwarding," AMD, Tech. Rep., 2021, https://www.amd.com/system/files/documents/security-analysis-predictive-store-forwarding.pdf.

[4] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P. Strub, "Jasmin: High-assurance and high-speed cryptography," in *ACM CCS*, 2017, pp. 1807–1823, https://doi.org/10.1145/3133956.3134078.

[5] J. B. Almeida, M. Barbosa, G. Barthe, B. Grégoire, V. Laporte, J. Léchenet, T. Oliveira, H. Pacheco, M. Quaresma, P. Schwabe, A. Séré, and P. Strub, "Formally verifying Kyber episode IV: implementation correctness," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2023, no. 3, pp. 164–193, 2023, https://doi.org/10.46586/tches.v2023.i3.164-193.

[6] J. B. Almeida, C. Baritel-Ruet, M. Barbosa, G. Barthe, F. Dupressoir, B. Grégoire, V. Laporte, T. Oliveira, A. Stoughton, and P. Strub, "Machine-checked proofs for cryptographic standards: Indifferentiability of sponge and secure high-assurance implementations of SHA-3," in *ACM CCS*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds., 2019, pp. 1607–1622, https://doi.org/10.1145/3319535.3363211.

[7] J. B. Almeida, M. Barbosa, G. Barthe, B. Grégoire, A. Koutsos, V. Laporte, T. Oliveira, and P.-Y. Strub, "The last mile: High-assurance and high-speed cryptographic implementations," in *IEEE S&P*, 2020, pp. 965–982, https://doi.org/10.1109/SP40000.2020.00028.

[8] J. B. Almeida, S. A. Olmos, M. Barbosa, G. Barthe, F. Dupressoir, B. Grégoire, V. Laporte, J.-C. Léchenet, C. Low, T. Oliveira, H. Pacheco, M. Quaresma, P. Schwabe, and P.-Y. Strub, "Formally verifying Kyber episode V: Machine-checked IND-CCA security and correctness of ML-KEM in EasyCrypt," in *IACR CRYPTO*, 2024, p. to appear, https://eprint.iacr.org/2024/843.

[9] B. Ammanaghatta Shivakumar, G. Barthe, B. Grégoire, V. Laporte, T. Oliveira, S. Priya, P. Schwabe, and L. Tabary-Maujean, "Typing high-speed cryptography against Spectre v1," in *IEEE S&P*, 2023, pp. 1094–1111, https://doi.org/10.1109/SP46215.2023.10179418.

[10] W. Arthur, B. Mehne, R. Das, and T. Austin, "Getting in control of your control flow with control-data isolation," in *IEEE/ACM CGO*, 2015, pp. 79–90, https://ieeexplore.ieee.org/abstract/document/7054189.

[11] E. Barberis, P. Frigo, M. Muench, H. Bos, and C. Giuffrida, "Branch history injection: On the effectiveness of hardware mitigations against cross-privilege Spectre-v2 attacks," in *USENIX Security*, 2022, pp. 971–988, https://www.usenix.org/conference/usenixsecurity22/presentation/barberis.

[12] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno, "SoK: Computer-aided cryptography," in *IEEE S&P*, 2021, pp. 777–795, https://doi.org/10.1109/SP40001.2021.00008.

[13] M. Barbosa, G. Barthe, C. Doczkal, J. Don, S. Fehr, B. Grégoire, Y. Huang, A. Hülsing, Y. Lee, and X. Wu, "Fixing and mechanizing the security proof of Fiat-Shamir with aborts and Dilithium," in *IACR CRYPTO*, 2023, pp. 358–389, https://doi.org/10.1007/978-3-031-38554-4_12.

[14] M. Barbosa, F. Dupressoir, B. Grégoire, A. Hülsing, M. Meijers, and P. Strub, "Machine-checked security for XMSS as in RFC 8391 and SPHINCS$^+$," in *IACR CRYPTO*, 2023, pp. 421–454, https://doi.org/10.1007/978-3-031-38554-4_14.

[15] G. Barthe, B. Grégoire, V. Laporte, and S. Priya, "Structured leakage and applications to cryptographic constant-time and cost," in *ACM CCS*, 2021, pp. 462–476, https://doi.org/10.1145/3460120.3484761.

[16] M. Bauer, L. Hetterich, C. Rossow, and M. Schwarz, "Switchpoline: A software mitigation for Spectre-BTB and Spectre-BHB on ARMv8," in *ACM AsiaCCS*, 2024, https://misc0110.net/files/switchpoline_asiaccs24.pdf.

[17] D. J. Bernstein, "Curve25519: new Diffie-Hellman speed records," in *IACR PKC*, 2006, pp. 207–228, https://cr.yp.to/papers.html#curve25519.

[18] J. W. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, "CRYSTALS-Kyber: A CCA-secure module-lattice-based KEM," in *IEEE EuroS&P*, 2018, pp. 353–367, https://doi.org/10.1109/EuroSP.2018.00032.

[19] B. Calder and D. Grunwald, "Reducing indirect function call overhead in C++ programs," in *ACM POPL*, 1994, pp. 397–408, https://dl.acm.org/doi/10.1145/174675.177973.

[20] C. Canella, J. V. Bulck, M. Schwarz, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *USENIX Security*, 2019, pp. 249–266, https://www.usenix.org/conference/usenixsecurity19/presentation/canella.

[21] C. Carruth, "Speculative load hardening – a Spectre variant #1 mitigation technique," LLVM documentation, https://llvm.org/docs/SpeculativeLoadHardening.html.

[22] S. Cauligi, C. Disselkoen, D. Moghimi, G. Barthe, and D. Stefan, "SoK: Practical foundations for software spectre defenses," in *IEEE S&P*, 2022, pp. 666–680, https://doi.org/10.1109/SP46214.2022.9833707.

[23] S. Cauligi, C. Disselkoen, K. von Gleissenthall, D. M. Tullsen, D. Stefan, T. Rezk, and G. Barthe, "Constant-time foundations for the new Spectre era," in *ACM PLDI*, 2020, pp. 913–926, https://doi.org/10.1145/3385412.3385970.

[24] B. Chen, Y. Wang, P. Shome, C. W. Fletcher, D. Kohlbrenner, R. Paccagnella, and D. Genkin, "GoFetch: Breaking constant-time cryptographic implementations using data memory-dependent prefetchers," in *USENIX Security*, 2024, p. to appear, https://gofetch.fail/files/gofetch.pdf.

[25] M. Flanders, R. K. Sharma, A. E. Michael, D. Grossman, and D. Kohlbrenner, “Avoiding instruction-centric microarchitectural timing channels via binary-code transformations,” in *ACM ASPLOS*, 2024, pp. 120–136, https://doi.org/10.1145/3620665.3640400.

[26] G. Gilles Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin, “Computer-aided security proofs for the working cryptographer,” in *IACR CRYPTO*, 2011, pp. 71–90, https://iacr.org/archive/crypto2011/68410071/68410071.pdf.

[27] Intel Labs, “Retpoline: a software construct for preventing branch-target-injection,” 2018, https://www.intel.com/content/dam/develop/external/us/en/documents/retpoline-a-branch-target-injection-mitigation.pdf.

[28] J. Jancar, M. Fourné, D. D. A. Braga, M. Sabt, P. Schwabe, G. Barthe, P.-A. Fouque, and Y. Acar, ““They’re not that hard to mitigate”: What cryptographic library developers think about timing attacks,” in *IEEE S&P*, 2022, pp. 632–649, https://eprint.iacr.org/2021/1650.

[29] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre attacks: Exploiting speculative execution,” in *IEEE S&P*, 2019, pp. 1–19, https://spectreattack.com/spectre.pdf.

[30] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, “Spectre returns! speculation attacks using the return stack buffer,” in *USENIX WOOT*, 2018, https://www.usenix.org/conference/woot18/presentation/koruyeh.

[31] J. Li, Z. Wang, X. Jiang, M. C. Grace, and S. Bahram, “Defeating return-oriented rootkits with "*Return-Less*" kernels,” in *ACM EuroSys*, 2010, pp. 195–208, https://doi.org/10.1145/1755913.1755934.

[32] G. Maisuradze and C. Rossow, “ret2spec: Speculative execution using return stack buffers,” in *ACM CCS*, 2018, pp. 2109–2122, https://dl.acm.org/doi/10.1145/3243734.3243761.

[33] N. Mosier, H. Nemati, J. C. Mitchell, and C. Trippel, “Serberus: Protecting cryptographic code from spectres at compile-time,” in *IEEE S&P*, 2024, https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00048.

[34] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen, and D. Stefan, “Swivel: Hardening WebAssembly against Spectre,” in *USENIX Security*, 2021, pp. 1433–1450, https://www.usenix.org/conference/usenixsecurity21/presentation/narayan.

[35] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, “G-free: defeating return-oriented programming through gadget-less binaries,” in *ACM ACSAC*, 2010, pp. 49–58, https://doi.org/10.1145/1920261.1920269.

[36] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, “Return-oriented programming: Systems, languages, and applications,” *ACM Transactions on Information and System Security*, vol. 15, no. 1, pp. 2:1–2:34, 2012, https://dl.acm.org/doi/10.1145/2133375.2133377.

[37] H. Shacham, “The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86),” in *ACM CCS*, 2007, pp. 552–561, https://doi.org/10.1145/1315245.1315313.

[38] Z. Shen, J. Zhou, D. Ojha, and J. Criswell, “Restricting control flow during speculative execution with Venkman,” https://arxiv.org/abs/1903.10651v1, 2019.

[39] B. A. Shivakumar, J. Barnes, G. Barthe, S. Cauligi, C. Chuengsatiansup, D. Genkin, S. O’Connell, P. Schwabe, R. Q. Sim, and Y. Yarom, “Spectre declassified: Reading from the right place at the wrong time,” in *IEEE S&P*, 2023, pp. 1753–1770, https://doi.org/10.1109/SP46215.2023.10179355.

[40] J. R. S. Vicarte, P. Shome, N. Nayak, C. Trippel, A. Morrison, D. Kohlbrenner, and C. W. Fletcher, “Opening Pandora’s box: A systematic study of new ways microarchitecture can leak private data,” in *ACM/IEEE ISCA 2021*, 2021, pp. 347–360, https://doi.org/10.1109/ISCA52012.2021.00035.

[41] J. Wikner and K. Razavi, “RETBLEED: Arbitrary speculative code execution with return instructions,” in *USENIX Security*, 2022, pp. 3825–3842, https://www.usenix.org/conference/usenixsecurity22/presentation/wikner.

[42] J. Wikner, D. Trujillo, and K. Razavi, “Phantom: Exploiting decoder-detectable mispredictions,” in *IEEE/ACM MICRO*, 2023, pp. 49–61, https://doi.org/10.1145/3613424.3614275.

[43] X. Yang, N. Cooprider, and J. Regehr, “Eliminating the call stack to save RAM,” in *ACM LCTES*, 2009, pp. 60–69, https://doi.org/10.1145/1542452.1542461.

Appendix

1. Source language and soundness

This appendix gives precise definitions of the source language, semantics, type system, and soundness proof discussed in the main paper.

Continuations. Figure 11 defines the continuations $\mathcal{C}(f)$ of a function f as the set of code that follows a call to f, along with a boolean that indicates whether the MSF gets updated.

Source semantics. Figure 12 presents the rest of the semantics. For speculative returns, that is the S-RET rule, we pin down the whole continuation $(cont, f, b)$ in the directive, instead of just the code, because $cont$ might come from different program points, with different MSF updates. We require $cs \neq (f, cont) :: cs'$ simply to make the semantics deterministic: our approach does not need determinism, but it makes presenting the rules and proofs easier.

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 since 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 the type $\{x : \langle\alpha, \beta\rangle\} \to \{x : \langle\alpha, \beta\rangle\}$ to id, with a polymorphic variable in the speculative component, then we would be able to type this program (by choosing as instantiations $\{\alpha, \beta \to \mathsf{L}\}$ for the first call site and $\{\alpha, \beta \to \mathsf{H}\}$ for the second). What we need is 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 $\{x : \langle\alpha, t\rangle\} \to \{x : \langle\alpha', t'\rangle\}$ where $\alpha \leq \alpha'$ and $t \leq t'$. We

$$C \oplus c \coloneqq \{\, (\mathit{cont}; c,\ b) \mid (\mathit{cont},\ b) \in C \,\}$$

$$\mathcal{C}_{\mathsf{i}}(f, i) \coloneqq \begin{cases} \mathcal{C}_{\mathsf{c}}(f, c_\top) \cup \mathcal{C}_{\mathsf{c}}(f, c_\bot) & \text{if } i \text{ is } \texttt{if } e \texttt{ then } c_\top \texttt{ else } c_\bot \\ \mathcal{C}_{\mathsf{c}}(f, c) \oplus i & \text{if } i \text{ is } \texttt{while } e \texttt{ do } c \\ \{([], b)\} & \text{if } i \text{ is } \texttt{call}_b\ f \\ \varnothing & \text{otherwise} \end{cases}$$

$$\mathcal{C}_{\mathsf{c}}(f, c) \coloneqq \begin{cases} \varnothing & \text{if } c \text{ is } [] \\ \mathcal{C}_{\mathsf{i}}(f, i) \oplus c' \cup \mathcal{C}_{\mathsf{c}}(f, c') & \text{if } c \text{ is } i; c' \end{cases}$$

$$\mathcal{C}(\mathit{callee}) \coloneqq \bigcup_{(\mathit{caller},\ c) \in p} \{\, (\mathit{cont},\ \mathit{caller},\ b) \mid (\mathit{cont}, b) \in \mathcal{C}_{\mathsf{c}}(\mathit{callee}, c) \,\}$$

Figure 11: Continuations. Here $\mathcal{C}_{\mathsf{i}}(f, i)$, where f is a function name and i an instruction, returns a set of pairs: the code that follows (in the execution, not syntactically) each function call to f in i and the boolean argument of the call. The operation $\oplus$ takes a set of code and appends its second argument to each element in the set. The definition of $\mathcal{C}_{\mathsf{c}}(f, c)$ is analogous to the one of $\mathcal{C}_{\mathsf{i}}(f, i)$, where c is code. Finally, $\mathcal{C}(f)$ aggregates all the continuations of the whole program, adding the name of the caller.

$$\frac{}{\langle x = e; c,\ f,\ cs,\ \rho,\ \mu,\ ms\rangle \xrightarrow[\mathsf{step}]{\bullet} \langle c,\ f,\ cs,\ \rho[x \leftarrow [\![\, e \,]\!]_\rho],\ \mu,\ ms\rangle}\ \textbf{ASSIGN}$$

$$\frac{[\![\, e \,]\!]_\rho = i \qquad 0 \le i < |a|}{\langle a[e] = x; c,\ f,\ cs,\ \rho,\ \mu,\ ms\rangle \xrightarrow[\mathsf{mem}\ b\ j]{\mathsf{addr}\ a\ i} \langle c,\ f,\ cs,\ \rho,\ \mu[(a, i) \leftarrow \rho(x)],\ ms\rangle}\ \textbf{N-STORE}$$

$$\frac{[\![\, e \,]\!]_\rho = i \qquad \neg\,(0 \le i < |a|) \qquad 0 \le j < |b|}{\langle a[e] = x; c,\ f,\ cs,\ \rho,\ \mu,\ \top\rangle \xrightarrow[\mathsf{mem}\ b\ j]{\mathsf{addr}\ a\ i} \langle c,\ f,\ cs,\ \rho,\ \mu[(b, j) \leftarrow \rho(x)],\ \top\rangle}\ \textbf{S-STORE}$$

$$\frac{[\![\, e \,]\!]_\rho = b' \qquad c' = \mathsf{if}\ b\ \mathsf{then}\ c_0; \texttt{while}\ e\ \texttt{do}\ c_0; c\ \mathsf{else}\ c}{\langle \texttt{while}\ e\ \texttt{do}\ c_0; c,\ f,\ cs,\ \rho,\ \mu,\ ms\rangle \xrightarrow[\mathsf{force}\ b]{\mathsf{branch}\ b'} \langle c',\ f,\ cs,\ \rho,\ \mu,\ ms \vee (b \neq b')\rangle}\ \textbf{WHILE}$$

$$\frac{}{\langle \texttt{init\_msf}(); c,\ f,\ cs,\ \rho,\ \mu,\ \bot\rangle \xrightarrow[\mathsf{step}]{\bullet} \langle c,\ f,\ cs,\ \rho[\mathit{msf} \leftarrow \mathsf{NOMASK}],\ \mu,\ \bot\rangle}\ \textbf{INIT-MSF}$$

$$\frac{v = \mathsf{if}\ [\![\, e \,]\!]_\rho\ \mathsf{then}\ \rho(\mathit{msf})\ \mathsf{else}\ \mathsf{MASK}}{\langle \texttt{update\_msf}(e); c,\ f,\ cs,\ \rho,\ \mu,\ ms\rangle \xrightarrow[\mathsf{step}]{\bullet} \langle c,\ f,\ cs,\ \rho[\mathit{msf} \leftarrow v],\ \mu,\ ms\rangle}\ \textbf{UPDATE-MSF}$$

Figure 12: Rest of the semantics.

need t' to be L for the first call site to be typable since we leak x after it, but we need t' to be H 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 $\langle \alpha, \mathsf{H}\rangle \to \langle \alpha, \mathsf{H}\rangle$ as the type of `id` and instantiate first with L and then with H 🐓. Doing so means that x is speculatively H after the first call site, but the protect ensures we do not leak any secrets.

Soundness. To prove that typable programs are speculatively constant-time, we show that pairs of single step executions preserve a property that includes equality of observations. This property entails the code of the machines and their call stack being typable, the MSF being synchronized with its type, and the machines being indistinguishable. Figure 13 presents the formal definitions of these properties. The precise statement of the lemma is as follows

Lemma 2 (Single-step soundness 🐓). *If two machines m_1 and m_2 step under directive d to m_1' and m_2' respectively, are typable under (Σ, Γ), are synchronized with Σ, and are indistinguishable under Γ, then the observation of both steps are the same, and the resulting machines are typable, synchronized, and indistinguishable. That is, we will prove that there exist Σ', and Γ' such that*

$$\begin{array}{c} m_1 \xrightarrow[d]{o_1} m_1' \\ m_2 \xrightarrow[d]{o_2} m_2' \\ \Sigma, \Gamma \vdash m_1 \\ \mathsf{synced}(\Sigma, \rho_1, ms_1) \\ \mathsf{synced}(\Sigma, \rho_2, ms_2) \\ m_1 =_\Gamma m_2 \end{array} \implies \begin{array}{l} o_1 = o_2 \\ \wedge\, \Sigma', \Gamma' \vdash m_1' \\ \wedge\, \mathsf{synced}(\Sigma', \rho_1', ms_1') \\ \wedge\, \mathsf{synced}(\Sigma', \rho_2', ms_2') \\ \wedge\, m_1' =_{\Gamma'} m_2' \end{array}$$

We show an analogous property for multi-step execu-

$$\frac{}{\Sigma, \Gamma \vdash []}\ \textbf{CS-NIL} \qquad \mathsf{synced}(\Sigma, \rho, ms) := \begin{cases} \top & \text{if } \Sigma \text{ is } \mathsf{unknown} \\ ms \iff \rho(msf) = \mathsf{MASK} & \text{if } \Sigma \text{ is } \mathsf{updated} \\ \neg\llbracket\, e \,\rrbracket_\rho \vee \rho(msf) = \mathsf{MASK} & \text{if } \Sigma \text{ is } \mathsf{outdated}(e) \text{ and } ms \\ \llbracket\, e \,\rrbracket_\rho \wedge \rho(msf) = \mathsf{NOMASK} & \text{if } \Sigma \text{ is } \mathsf{outdated}(e) \text{ and } \neg ms \end{cases}$$

$$\frac{\Sigma, \Gamma \vdash c : \Sigma'_g, \Gamma'_g \qquad \Sigma'_g, \Gamma'_g \vdash cs}{\Sigma, \Gamma \vdash (g, c) :: cs}\ \textbf{CS-CONS} \qquad \frac{\Sigma, \Gamma \vdash c : \Sigma'_f, \theta(\Gamma'_f) \qquad \Sigma'_f, \theta(\Gamma'_f) \vdash cs}{\Sigma, \Gamma \vdash \langle c,\ f,\ cs,\ \rho,\ \mu,\ ms \rangle}\ \textbf{MACH}$$

Figure 13: A machine with registers ρ and misspeculation state ms is *synchronized* with an MSF type Σ, denoted $\mathsf{synced}(\Sigma, \rho, ms)$, if the MSF accurately tracks misspeculation. A call stack cs is typable under Σ and Γ if $\Sigma, \Gamma \vdash cs$ holds. A machine m is *typable* under Σ and Γ, denoted $\Sigma, \Gamma \vdash m$, if there exists an instantiation θ such that its code and its call stack are typable as shown in the rule. For all functions f, we write Σ_f, Γ_f, Σ'_f, and Γ'_f assuming $Sig(f) = \Sigma_f, \Gamma_f \to \Sigma'_f, \Gamma'_f$.

$$\frac{\ell : x = e \to \ell' \qquad \llbracket\, e \,\rrbracket_\rho = v}{\langle \ell,\ \rho,\ \mu,\ ms \rangle \xrightarrow[\mathsf{step}]{\bullet} \langle \ell',\ \rho[x \leftarrow v],\ \mu,\ ms \rangle}\ \textbf{ASSIGN}$$

$$\frac{\ell : \mathsf{jump} \to \ell'}{\langle \ell,\ \rho,\ \mu,\ ms \rangle \xrightarrow[\mathsf{step}]{\bullet} \langle \ell',\ \rho,\ \mu,\ ms \rangle}\ \textbf{JUMP}$$

$$\frac{\ell : \mathsf{if}\ e\ \mathsf{jump} \to \ell_\top, \ell_\bot \qquad \llbracket\, e \,\rrbracket_\rho = b'}{\langle \ell,\ \rho,\ \mu,\ ms \rangle \xrightarrow[\mathsf{force}\ b]{\mathsf{branch}\ b'} \langle \ell_b,\ \rho,\ \mu,\ ms \vee (b \neq b') \rangle}\ \textbf{CJUMP}$$

Figure 14: Selected rules of the semantics of the linear language.

$$\frac{\begin{array}{c}(callee, cont, caller, b) \leftrightarrow_{\mathcal{L}} \ell_r \\ \ell \ \ : \ \ \mathsf{jump}_{\mathsf{ret}} \ \to \ \ell_r\end{array}}{\ell : \llparenthesis\, \{\, (cont, caller, b) \,\}\, \rrparenthesis^{callee}_{\mathsf{rettbl}}}$$

$$\frac{\begin{array}{c}(callee, cont, caller, b) \leftrightarrow_{\mathcal{L}} \ell_r \\ \begin{array}{lll} \ell & : & \mathsf{if}\ (ra_{callee} = \ell_r)\ \mathsf{jump}_{\mathsf{ret}} \quad \to \ \ell_r, \ell_0 \\ \ell_0 & : & \llparenthesis\, \ell^* \,\rrparenthesis^{callee}_{\mathsf{rettbl}} \end{array}\end{array}}{\ell : \llparenthesis\, \{\, (cont, caller, b) \,\} \cup \ell^* \,\rrparenthesis^{callee}_{\mathsf{rettbl}}}$$

Figure 15: Compilation relation for return tables.

tions, which allows us to conclude Theorem 1.

2. Compilation and preservation

This appendix gives precise definitions of the compilation scheme, directive and observation transformers, and preservation proof discussed in the main paper.

Linear semantics. Figure 14 presents some rules of the semantics of the linear language. The remaining rules, for loads, stores, and the selSLH instructions, are similar to the ones for the source language. We write $\ell : li \to \ell^*$ when li is at position ℓ and its successors are ℓ^*. All instructions have exactly one successor except conditional jumps, which have two.

Compilation. Figure 16 presents the compilation scheme as a relation between source code an a linear program, denoted $\ell : \llparenthesis\, c \,\rrparenthesis^f_{cont} \to \ell'$, that holds when the target program contains compilation of c, starting at label ℓ and ending at label ℓ'. We keep track of a context which contains the function name f and current continuation $cont$ to be able to compile function calls.

We compile basic instructions, conditionals, and loops in the natural way. For function calls, that is the CALL-$\bot$ and CALL-$\top$ rules, we annotate the jumps to the callee; this is just a jump instruction with a special annotation. We introduce an MSF update only if the second argument of the call is $\top$.

To create the return table for each function we need to label all its continuations. We model this with a relation $(callee, cont, caller, b) \leftrightarrow_{\mathcal{L}} \ell$, which relates each continuation of *callee* with a unique label. This means

$$\begin{aligned} &(callee, cont, caller, b) \leftrightarrow_{\mathcal{L}} \ell \\ &\quad \Longrightarrow \ (cont, caller, b) \in \mathcal{C}(callee) \end{aligned}$$

with ℓ unique per callee.

Figure 15 presents the compilation of return tables. We need no rule for zero call sites since the only function with no call sites is the entry point, which does not require a return table.

Finally, the linear program is the compilation of the source program, denoted $\llparenthesis\, p \,\rrparenthesis$, if the bodies of all functions are compiled at their labels, and internal functions have their return tables. That is,

$$\begin{aligned} &\forall (f, c_f) \in p. \\ &\quad \ell_f : \llparenthesis\, c_f \,\rrparenthesis^f_{[]} \to \ell_{ret_f} \wedge f \neq \mathsf{main} \Rightarrow \ell_{ret_f} : \llparenthesis\, \mathcal{C}(f) \,\rrparenthesis^f_{\mathsf{rettbl}} \end{aligned}$$

where the entry point is main.

Preservation. Figure 17 defines the directive and observation transformers. The directive transformer $T_{\mathsf{Dir}}(\ell, d)$ determines what the source machine does given the current

$$\frac{}{\ell : (\!|\, [\,] \,|\!)^{f}_{cont} \to \ell}\ \textbf{NIL} \qquad \frac{\begin{array}{lcll}\ell & : & (\!|\, i \,|\!)^{f}_{c;cont} & \to\ \ell_i \\ \ell_i & : & (\!|\, c \,|\!)^{f}_{cont} & \to\ \ell'\end{array}}{\ell : (\!|\, i; c \,|\!)^{f}_{cont} \to \ell'}\ \textbf{CONS} \qquad \frac{\ell\ :\ \mathcal{A} = \mathcal{B}\ \to\ \ell'}{\ell : (\!|\, \mathcal{A} = \mathcal{B} \,|\!)^{f}_{cont} \to \ell'}\ \begin{array}{l}\textbf{ASSIGN}\\ \textbf{LOAD}\\ \textbf{STORE}\\ \textbf{INIT-MSF}\\ \textbf{UPDATE-MSF}\\ \textbf{PROTECT}\end{array}$$

$$\frac{\begin{array}{lcll}\ell & : & \mathsf{if}\ e\ \mathsf{jump} & \to\ \ell_\top, \ell_\bot \\ \ell_\top & : & (\!|\, c_\top \,|\!)^{f}_{cont} & \to\ \ell'_\top \\ \ell'_\top & : & \mathsf{jump} & \to\ \ell' \\ \ell_\bot & : & (\!|\, c_\bot \,|\!)^{f}_{cont} & \to\ \ell'\end{array}}{\ell : (\!|\, \texttt{if}\ e\ \texttt{then}\ c_\top\ \texttt{else}\ c_\bot \,|\!)^{f}_{cont} \to \ell'}\ \textbf{COND} \qquad \frac{w = \texttt{while}\ e\ \texttt{do}\ c \qquad \begin{array}{lcll}\ell & : & \mathsf{if}\ e\ \mathsf{jump} & \to\ \ell_0, \ell' \\ \ell_0 & : & (\!|\, c \,|\!)^{f}_{w;cont} & \to\ \ell_1 \\ \ell_1 & : & \mathsf{jump} & \to\ \ell\end{array}}{\ell : (\!|\, \texttt{while}\ e\ \texttt{do}\ c \,|\!)^{f}_{cont} \to \ell'}\ \textbf{WHILE}$$

$$\frac{\begin{array}{c}(callee, cont, f, \bot) \leftrightarrow_{\mathcal{L}} \ell' \\ \begin{array}{lcll}\ell & : & ra_{callee} = \ell' & \to\ \ell_0 \\ \ell_0 & : & \mathsf{jump}_{\mathsf{call}} & \to\ \ell_{callee}\end{array}\end{array}}{\ell : (\!|\, \texttt{call}_\bot\ \ callee \,|\!)^{f}_{cont} \to \ell'}\ \textbf{CALL-}\bot \qquad \frac{\begin{array}{c}(callee, cont, f, \top) \leftrightarrow_{\mathcal{L}} \ell' \\ \begin{array}{lcll}\ell & : & ra_{callee} = \ell_r & \to\ \ell_0 \\ \ell_0 & : & \mathsf{jump}_{\mathsf{call}} & \to\ \ell_{callee} \\ \ell_r & : & \texttt{update\_msf}(ra_{callee} = \ell_r) & \to\ \ell'\end{array}\end{array}}{\ell : (\!|\, \texttt{call}_\top\ \ callee \,|\!)^{f}_{cont} \to \ell'}\ \textbf{CALL-}\top$$

Figure 16: Compilation relation.

$$T_{\mathsf{Dir}}(\ell, d) := \begin{cases} [\,] & \text{if } \ell : \mathsf{jump} \to \ell' \\ [\,] & \text{if } \ell : ra_f = \ell_r \to \ell' \\ [\,] & \text{if } \ell : \texttt{update\_msf}(ra_f = \ell_r) \to \ell' \\ [\,] & \text{if } \ell : \mathsf{if}\ ra_f = \ell_r\ \mathsf{jump}_{\mathsf{ret}} \to \ell_r, \ell_0 \text{ and } d = \mathsf{force}\ \bot \\ \mathsf{return}\ c\ g\ b & \text{if } \ell : \mathsf{jump}_{\mathsf{ret}} \to \ell_r \text{ and } (f, c, g, b) \leftrightarrow_{\mathcal{L}} \ell_r \\ \mathsf{return}\ c\ g\ b & \text{if } \ell : \mathsf{if}\ ra_f = \ell_r\ \mathsf{jump}_{\mathsf{ret}} \to \ell_r, \ell_0 \text{ and } d = \mathsf{force}\ \top \text{ and } (f, c, g, b) \leftrightarrow_{\mathcal{L}} \ell_r \\ [d] & \text{otherwise} \end{cases}$$

$$T_{\mathsf{Obs}}(\ell, \rho^*, O) := \begin{cases} \bullet & \text{if } \ell : \mathsf{jump} \to \ell' \\ \bullet & \text{if } \ell : ra_f = \ell_r \to \ell' \\ \bullet & \text{if } \ell : \texttt{update\_msf}(ra_f = \ell_r) \to \ell' \\ \bullet & \text{if } \ell : \mathsf{jump}_{\mathsf{ret}} \to \ell_r \\ \mathsf{branch}\ [\![\, ra_f = \ell_r \,]\!]_{\rho^*} & \text{if } \ell : \mathsf{if}\ ra_f = \ell_r\ \mathsf{jump}_{\mathsf{ret}} \to \ell_r, \ell_0 \\ \mathsf{head}\ O & \text{otherwise} \end{cases}$$

Figure 17: Directive and observation transformers. The transformer are undefined in the cases not shown here.

instruction (at position ℓ) and the target directive d. The first four cases say that the source machine skips a step when the target executes jumps, return address stores, MSF updates after a function call, and entries of the return table that are not taken. The next two cases are when an entry of the return table entry is taken, and it produces a return $c\ g\ b$ directive, which is uniquely identified by $(f, c, g, b) \leftrightarrow_{\mathcal{L}} \ell_r$. In all other cases, the source and target machines behave identically.

On the other hand, the transformer for observations $T_{\mathsf{Obs}}(\ell, \rho^*, O)$ determines the observation of the target machine given the current instruction (at position ℓ), the values of the return addresses, and the source observations. For the first four cases, which are intermediate steps in the compilation of a source instruction, it predicts that the target observation will be $\bullet$. The next case is an intermediate jump in a return table: it uses the value of the return addresses to determine the observation of this comparison. We are able to provide the values of the return addresses because they are uniquely determined from the public part of the initial state and the directives executed until now. Similarly to the directive transformer, in all other cases, the source and target machines behave identically.

In order to prove Theorem 2, we need to extend the transformers to multi-step executions. This is straightforward if we note that from the current instruction and directive we can predict the next instruction: it is the following label, unless it is a conditional jump, in which case the directive is force b and the label ℓ_b follows.