



HAL
open science

Preservation of Speculative Constant-time by Compilation

Santiago Arranz Olmos, Gilles Barthe, Lionel Blatter, Benjamin Grégoire,
Vincent Laporte

► **To cite this version:**

Santiago Arranz Olmos, Gilles Barthe, Lionel Blatter, Benjamin Grégoire, Vincent Laporte. Preservation of Speculative Constant-time by Compilation. Proceedings of the ACM on Programming Languages, 2025, 9 (POPL), pp.1293 - 1325. 10.1145/3704880 . hal-04663857v2

HAL Id: hal-04663857

<https://hal.univ-lorraine.fr/hal-04663857v2>

Submitted on 13 Jan 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License



Preservation of Speculative Constant-Time by Compilation

SANTIAGO ARRANZ OLMOS, MPI-SP, Germany

GILLES BARTHE, MPI-SP, Germany and IMDEA Software Institute, Spain

LIONEL BLATTER, MPI-SP, Germany

BENJAMIN GRÉGOIRE, Inria, France

VINCENT LAPORTE, Inria, France and Université de Lorraine, France

Compilers often weaken or even discard software-based countermeasures commonly used to protect programs against side-channel attacks; worse, they may also introduce vulnerabilities that attackers can exploit. The solution to this problem is to develop compilers that preserve such countermeasures. Prior work establishes that (a mildly modified version of) the CompCert and Jasmin formally verified compilers preserve constant-time, an information flow policy that ensures that programs are protected against timing side-channel attacks. However, nothing is known about preservation of speculative constant-time, a strengthening of the constant-time policy that ensures that programs are protected against Spectre-v1 attacks. We first show that preservation of speculative constant-time fails in practice by providing examples of secure programs whose compilation is not speculative constant-time using GCC (GCC -O0 and GCC -O1) and Jasmin. Then, we define a proof-of-concept compiler that distills some of the critical passes of the Jasmin compiler and use the Coq proof assistant to prove that it preserves speculative constant-time. Finally, we patch the Jasmin speculative constant-time type checker and demonstrate that all cryptographic implementations written in Jasmin can be fixed with minimal impact.

CCS Concepts: • **Software and its engineering** → **Formal software verification; Compilers**; • **Security and privacy** → **Formal security models**.

Additional Key Words and Phrases: Preservation of security, Speculative execution, Spectre

ACM Reference Format:

Santiago Arranz Olmos, Gilles Barthe, Lionel Blatter, Benjamin Grégoire, and Vincent Laporte. 2025. Preservation of Speculative Constant-Time by Compilation. *Proc. ACM Program. Lang.* 9, POPL, Article 44 (January 2025), 33 pages. <https://doi.org/10.1145/3704880>

1 Introduction

Timing attacks are a class of side-channel attacks that collect and analyze timing behavior of programs to learn about confidential data manipulated during execution [38]. Over the last decades, timing attacks have been used extensively to retrieve secret keys from popular cryptographic implementations, see, e.g., [3, 4, 11, 19, 29, 31, 45, 47, 48, 50, 59]. One recipe to protect cryptographic libraries against such attacks is to follow the constant-time (CT) discipline. The CT discipline is simple to state: a program is constant-time if its control flow and memory accesses do not depend on secrets. Unfortunately, writing efficient constant-time code is error-prone. To avoid such errors, developers have access to a broad range of tools for checking that a program is

Authors' Contact Information: [Santiago Arranz Olmos](mailto:santiago.arranz-olmos@mpi-sp.org), MPI-SP, Bochum, Germany, santiago.arranz-olmos@mpi-sp.org; [Gilles Barthe](mailto:gilles.barthe@mpi-sp.org), MPI-SP, Bochum, Germany and IMDEA Software Institute, Madrid, Spain, gilles.barthe@mpi-sp.org; [Lionel Blatter](mailto:lionel.blatter@mpi-sp.org), MPI-SP, Bochum, Germany, lionel.blatter@mpi-sp.org; [Benjamin Grégoire](mailto:benjamin.gregoire@inria.fr), Inria, Sophia Antipolis, France, benjamin.gregoire@inria.fr; [Vincent Laporte](mailto:vincent.laporte@loria.fr), Inria, Nancy, France and Université de Lorraine, Nancy, France, vincent.laporte@loria.fr.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2025 Copyright held by the owner/author(s).

ACM 2475-1421/2025/1-ART44
<https://doi.org/10.1145/3704880>

constant-time [13, 30, 34]. Many tools operate at source level. This allows analyzing fine-grained properties (if needed by means of interactive tools) and significantly eases developer feedback. However, this is at the cost of ignoring the possibility of compiler security bugs [28, 58]. They can have disastrous consequences, as demonstrated by Purnal in the recent side-channel attack of the reference implementation of ML-KEM [46], a post-quantum key encapsulation mechanism under standardization by the National Institute of Standards and Technology. Purnal’s discovery is an example of a compiler—in this case Clang—undoing programmers’ countermeasures in a code snippet with non-trivial control flow—specifically, nested loops.

An alternative is to check for constant-time at target level, immunizing against compiler bugs compromising security. However, targeting low-level programs makes it harder to carry interactive fine-grained analyses and to provide useful feedback to developers. Fortunately, one can obtain the best of the two worlds by proving that the compiler preserves constant-time, i.e., that the compilation of a constant-time program is constant-time. Preservation of constant-time allows us to check for constant-time on source programs, thereby simplifying analysis and programmer feedback, and obtain guarantees on assembly programs, thus ensuring the absence of compiler induced security vulnerabilities. Recent work shows that preservation of constant-time can be practical. For instance, CompCertCT [15] shows how CompCert [39], a formally verified optimizing C compiler not designed with security in mind, can be made to preserve constant-time with only mild adjustments during instruction selection and assembly generation. Another example is the Jasmin compiler, used to generate efficient cryptographic implementations [5, 6], which has been shown to preserve constant-time [10, 18].

Unfortunately, the constant-time policy is no longer the golden standard for cryptographic libraries. Indeed, there is a long line of work showing that constant-time cryptographic code is vulnerable against transient execution attacks [8, 26, 36, 53–55]. In this paper, we focus on Spectre-v1 [37], the most basic form of Spectre attack, in which an attacker hijacks the branch predictor in order to force the victim program to execute a control flow path of their choice, independently of the values of the guards. Spectre-v1 is a powerful attack, allowing, in particular, an attacker to force a program to skip security-critical code, such as bounds checks. Accounting for Spectre-v1 leads to the notion of speculative constant-time (SCT), which, informally, ensures that an attacker with full control over the branch predictor and full ability to observe program leakage learns nothing about the victim’s secrets [25]. Since speculative constant-time is even more error-prone than constant-time, researchers have developed many different tools for checking that a program is SCT [24]. These tools support different protection methods, such as fence insertion [16, 52] and speculative load hardening (SLH) [23, 44, 60]. The most promising approach to date is selective speculative load hardening (selSLH), a refinement of SLH that minimizes the amount of protections inserted in programs based on a fine-grained information flow analysis [8]. Selective SLH is implemented in the Jasmin language [5, 6], in the form of new language primitives that programmers can use at their discretion [9], with the help of an information flow type system that ensures that the program contains sufficiently many and well-placed mitigations. Selective SLH primitives allow programmers to write SCT programs with minimal overhead; for instance, protecting a state-of-the-art ML-KEM implementation against Spectre-v1 incurs a performance penalty of about 1%. Unfortunately, these primitives are added to source programs, creating the risk that the Jasmin compiler produces assembly programs that are not speculative constant-time. This risk is not limited to the Jasmin compiler; indeed, early work by Patrignani and Guarnieri [44] shows that mainstream compilers produce assembly code that is insecure under speculative execution.

Problem statement and contributions. The main contribution of this paper is to prove preservation of speculative constant-time for a proof-of-concept compiler inspired by the Jasmin language. We specifically target Spectre-v1.

As a preliminary observation, we show that preservation of speculative constant-time fails in practice, either because optimizations tamper with source-level countermeasures, or because they introduce leakage that does not exist at source level. We consider two examples:

- GCC, where we show that manual instrumentation of speculative load hardening in C programs is discarded by GCC -O1. This suggests to use GCC -O0 to preserve security. Unfortunately, we dispel this suggestion by providing an example of a speculative constant-time C program that is compiled using GCC -O0 into an assembly program that is not. In the second case, the culprit is *spilling*, which introduces new memory accesses and thus opportunities for Spectre attacks; and
- Jasmin, where we show an example of a Jasmin program that is speculative constant-time—and, in fact, typable with the type system of [9]—whose compilation is not. In this case, the culprit is memory reuse, and concretely the stack allocation pass that introduces array reuse.

These observations may come as no surprise, since the compilers did not claim preservation of speculative constant-time. Yet, they show that such a claim requires careful analysis or ideally formal proofs.

Then, we make the following contributions:

- We develop general results for preservation of speculative constant-time. We lift the well-known notion of backward simulation to the speculative setting, and show a completeness result for this and a related approach;
- We define a core language with primitives for protecting against Spectre-v1, and endow the language with two speculative semantics. The first semantics provides an explicit treatment of initialized values that is used to address the issue with array reuse, and overcomes another incongruity of prior semantics (see Section 5.2). The second semantics imposes additional requirements on directives, which simplifies simulation proofs. We prove the equivalence of these semantics with respect to speculative constant-time;
- We prove preservation of speculative constant-time for nine compiler passes of interest used in the Jasmin compiler, including array reuse and array concatenation. Then, we prove a composition result and conclude that the combination of all these passes preserves SCT; and
- We adapt the SCT type system of [9] both for our core language and for the Jasmin language. For our core language, we show an end-to-end theorem stating that typable programs are compiled into speculative constant-time programs. For the Jasmin language, we show that the practical impact of our adaptation is minimal on existing Jasmin implementations. Specifically, we show that all implementations of the libjade library are conveniently patched with little effort and no overhead.

For the sake of concreteness, our presentation discusses exclusively on (selective) speculative load hardening. However, the simplest (but expensive) protection against Spectre-v1 is to insert memory fences. Therefore, it is reasonable to ask if compilers preserve speculative constant-time when programs are protected with fences. All the preservation results of this paper extend *mutatis mutandis* to fences, as the `init_msf()` instruction used in this paper is a form of fence instruction.

Artifacts. All our results are formally verified in the Coq proof assistant. The full Coq formalization is available as supplementary material on <https://artifacts.formosa-crypto.org/data/sct-preservation.tbz>.

Concurrent Work. van der Wall and Meyer [51] developed a different SCT preservation technique simultaneously and independently. We compare and contrast the approaches in Section 9.

2 Background

Program execution involves intricate interactions between multiple processor components, collectively known as the micro-architecture. These interactions are bidirectional: micro-architectural components are updated during execution, and they influence execution, e.g., through execution time. An important part of the micro-architecture is the state of caches and predictors, continuously evolving during program execution, e.g., by adding or evicting a cache entry, or by updating branch prediction counters. Additionally, the state of the cache or of branch prediction counters may lead to cache misses and mispredictions, and result in timing differences of executions. Moreover, effects of transient execution may persist after misspeculation has been detected and misspeculated execution has been squashed. While these discrepancies have no influence on the correct execution of programs, attackers can exploit them to recover secrets through side-channel attacks. This section provides a brief review of attacks, policies, and countermeasures.

2.1 Cache Attacks and the Constant-Time Discipline

Cache attacks [19, 45, 50] is a class of side-channel attacks that exploit cache latency, and more concretely the latency between a cache hit and a cache miss, to recover cryptographic keys. Examples of devastating cache attacks include [11, 31, 47, 48, 59].

The constant-time discipline is a programming practice to protect implementations against cache attacks by requiring that a program's control flow and memory accesses do not depend on secrets. One main appeal of the constant-time discipline is that many cryptographic implementations can be implemented in constant-time with minimal overhead, see [35] for an early example. Another advantage of the constant-time discipline is that it enforces system-level security while simultaneously being expressible neatly in terms of an instrumented semantics of the form $s \xrightarrow{o} s'$, where s and s' are execution states, and o is an observation [14]. In the baseline constant-time model, observations are either none (denoted \bullet , for instructions that do not leak), a boolean value (denoted branch b , for branching instructions), or a memory address (denoted $\text{addr } a \ i$, for load and store instructions). Then, a program is constant-time w.r.t. some relation ϕ on initial states if and only if any two executions starting from related states yield equal leakage. Constant-time is an instance of observational non-interference, and as such can be checked or enforced using techniques from language-based security. We refer the reader to [13, 30, 34] for a recent overview of some of the main tools for constant-time verification.

2.2 Speculative Attacks and the Speculative Constant-Time Discipline

Unfortunately, constant-time programs remain vulnerable to speculative attacks, and in particular Spectre attacks [37], in which an attacker takes control of speculation mechanisms and makes a victim program leak sensitive information during misspeculated execution.

Speculative attacks can be broadly described by an instrumented semantics of the form $s \xrightarrow[d]{o} s'$ where s , s' , and o are, as before, two states and an observation, and d is a *directive* that is used for resolving (under-specified) speculative choices and hence models an adversary controlling speculative execution. Given such a speculative semantics, we can define the general notion of *speculative constant-time*, with respect to a relation ϕ on initial states: it states that a program does not leak sensitive information in spite of speculative execution.

Several approaches have been proposed to mitigate speculative attacks, and in particular Spectre-v1, which exploits branch prediction. The simplest one is to insert a fence after every branch, effectively blocking any misspeculated execution. Another approach is index masking, which prevents speculative out-of-bounds accesses by ensuring that every array access is of the form

<pre> 1 #define update_msf(msf, e) (e ? msf : 0xff...ff) 2 #define protect(x, msf) (x msf) 3 4 uint64_t init_msf() { 5 asm volatile ("lfence" ::: "memory"); 6 return 0; 7 } 8 9 uint64_t load(uint64_t* p, uint64_t i) { 10 uint64_t msf = init_msf(); 11 uint64_t x = 0; 12 if (i < 10) { 13 msf = update_msf(msf, i < 10); 14 x = p[i]; 15 x = protect(x, msf); 16 } else { 17 msf = update_msf(msf, !(i < 10)); 18 } 19 return p[x]; 20 } </pre>	<pre> 1 load: 2 lfence 3 cmpq \$9, %rsi 4 ja .L 5 movq (%rdi,%rsi,8), %rax 6 movq (%rdi,%rax,8), %rax 7 retq 8 .L: 9 xorl %eax, %eax 10 movq (%rdi,%rax,8), %rax 11 retq </pre>
---	---

Fig. 1. A function with SLH mitigations, before and after compilation with GCC 12.3.0 at optimization level 1. The array p has size ten.

$a[e \bmod n]$, where n is the size of the array. However, both approaches have a significant performance overhead. A more efficient approach, named *speculative load hardening*, creates control flow dependencies (which may be abused speculatively) through data dependencies (which are not affected by speculation) [23, 44, 60]. In this way, leaking misspeculated execution steps are blocked until the end of the speculation window. SLH induces quite a large run-time overhead. A more refined approach to SLH, called *selective SLH* exploits knowledge about which data is secret to reduce the need for protection: in many cases, it is unnecessary to detect misspeculation.

3 Motivating Examples

This section illustrates issues with preservation of speculative constant-time for GCC, a widely used compiler that is not designed with security in mind, and Jasmin, a domain-specific compiler designed with security in mind.

3.1 Case Study I: GCC

Spectre-v1 attacks typically exploit unsafe memory accesses during speculative execution. The essence of speculative load hardening is to mask such memory accesses such that they do not leak secrets. To this end, it is necessary to instrument programs so that they can track whether speculation has gone wrong. This is done using a register msf (the *misspeculation flag* or MSF) to track misspeculation, and updating this register after each branching statement to check if execution proceeded into the correct branch or misspeculated. Figure 1 shows an example of a C program instrumented with speculative load hardening, taken from Figure 3a in [9]. The first instruction of the procedure `load` sets the msf register to zero using the `init_msf` instruction, one of the three selSLH instructions from the same work. This instruction inserts a fence, which has the effect of stopping speculative execution, and sets the misspeculation flag to zero, to indicate that the program is executing sequentially—i.e., it has not misspeculated. Then, the flag is updated immediately after entering each branch of a conditional. This is carried out using the instruction

<pre> 1 uint8_t* tbl; 2 3 uint8_t example(uint64_t sec, uint64_t pub) { 4 uint64_t r = 0; 5 uint64_t p[1] = { pub }; 6 if (pub != pub) { p[1] = sec; } 7 return tbl[r]; 8 } </pre>	<pre> example: ... movq %rdi, -8(%rbp) ;; push sec movq \$0, -24(%rbp) ;; init r ... je .L movq -8(%rbp), %rax ;; read sec movq %rax, -24(%rbp) ;; write r .L: movq tbl(%rip), %rax ;; load tbl movq -24(%rbp), %rcx ;; load r movb (%rax,%rcx), %al ;; tbl[r] ... </pre>
--	---

Fig. 2. Spilling does not preserve speculative constant-time.

`update_msf`. Critically, this instruction updates the MSF using a conditional move, which, contrary to conditional jumps, is not affected by speculation. This way, `update_msf` ensures that the register `msf` accurately tracks misspeculation. Based on this, we can protect vulnerable memory accesses using the `protect` instruction, which is the last `selSLH` primitive from [9]. This instruction masks values if execution has misspeculated, so that arrays are accessed to a default index in that case. Overall, the instrumentation ensures that the program from Figure 1 is speculative constant-time.

Speculative load hardening mitigations are redundant computations in sequential execution; the GCC compiler (version 12.3.0) detects this at optimization level 1. It notices that the updates to the `msf` variable are superfluous and therefore removes them, as witnessed by the generated assembly, on the right side of Figure 1: the compiler removes the `protects` and the initialization of the MSF. As a result, the output assembly is not speculative constant-time.

It is often tempting to address security issues introduced by compilers simply by turning off optimizations. However, Figure 2 provides an example of a speculative constant-time program that is compiled by GCC `-O0` into an assembly program that is not. Indeed, the listing on the left side of Figure 2 presents a sequentially safe program (note the out-of-bounds access inside the never-taken then-branch) that is speculative constant-time (even under misspeculation, the value of `r` is zero). When compiled with GCC `-O0`, we get the listing on the right side of Figure 2. By default, the compiler spills function arguments on the stack, allowing an attacker to exploit unsafe memory accesses to overwrite the local variable `r` with secret value `sec`, even though this was not possible in the source program. A dead branch like this represents, for instance, iteration $n + 1$ of a loop with n iterations.

3.2 Case Study II: Jasmin

Jasmin [5, 6] is a framework for high-speed, high-assurance cryptographic software. Its main components are the Jasmin language and certified compiler. The language is based on the “assembly in the head” paradigm; it features a combination of low-level constructs, such as vectorized instructions, and high-level constructs, such as structured control flow (conditionals, loops, and functions), and zero-cost abstractions, such as explicit variable names and functional arrays. These high-level constructs and zero-cost abstractions vastly simplify reasoning about program correctness and security. At the same time, the Jasmin framework is designed to generate efficient assembly code, so its compiler eliminates source-level conveniences. For instance, the compiler checks that arrays are used linearly, allowing them to be modified in-place and ensuring that there are no run-time

```

1 export
2 fn main(#secret reg u64 sec) → #secret reg u64 {
3   stack u64[N] tmp;
4   tmp[0] = sec; // spill secret
5   ... // Computation that uses all registers.
6   sec = tmp[0]; // tmp is dead now
7   stack u64[N] p;
8   reg u64 i = 0;
9   while (i < N) {
10    p[i] = 0; // Public computation
11    i += 1;
12  }
13  i = p[0];
14  [i] = 0; // Raw memory access
15  return sec;
16 }

```

```

1 export
2 fn sign(reg u64 m len k) {
3   _ = #init_msf();
4   reg u128 key = preprocess_secret_key(k);
5   stack u8[N] state;
6   reg u64 i = 0;
7   while (i < N) { state[i] = 0; i += 1; }
8   i = 0;
9   while (i + B < len) {
10    state = absorb(state, m, i);
11    i += B;
12  }
13 }

```

Fig. 3. Array reuse does not preserve speculative constant-time.

copies. The compiler also minimizes stack usage through array reuse, allowing two arrays with different live ranges to share the same stack region.

Recently, Ammanaghatta Shivakumar et al. [9] extended Jasmin with a small set of language constructs to support selSLH and a type system to verify that programmers use them correctly. The type system recognizes different countermeasures: it can type check programs protected with fence insertion, index masking, speculative load hardening, and selective speculative load hardening. Moreover, this approach has proven effective in protecting optimized cryptographic primitives with negligible performance overhead. Unfortunately, some Jasmin programs are typable (and hence speculative constant-time according to the definition of said work) but compile into insecure assembly programs.

Let us illustrate the problem with a constructed example first. Consider the program on the left side of Figure 3. This program is (sequentially) safe and speculative constant-time, according to [9]; however, its compilation is vulnerable to Spectre-v1 attacks. It starts by spilling its (secret) argument to a stack array `tmp` (line 4), a common occurrence in code with high register pressure. It later unspills the secret, rendering the stack array dead (line 6). Next, it performs public computation on a different stack array, `p`, filling it with zeros (lines 9 to 12). Lastly, it leaks the contents of `p` with a memory access (lines 13 to 14). The vulnerability arises during compilation: the compiler reuses the stack region of `tmp` for `p`, which is standard behavior since the former is dead when the latter is initialized. In doing so, the compiler enables an attacker to skip the public computation on `p` by mistraining the CPU to predict “not taken.” Thus, the last two instructions would leak the secret argument. An essential component of this attack is that the leaked value comes from a speculatively unsafe load. That is, the access `p[0]` is always safe (and especially initialized) under sequential execution, but not under speculation.

The solution to this problem is to modify the semantics of source programs so that every load from an uninitialized memory cell is adversarially controlled—since tracking initialization with a type system is challenging. Thus, programmers must assume that an attacker can load a value from a memory cell of their choice on an uninitialized memory access. Under such a semantics, the source program is no longer speculative constant-time and requires the protections that allow the preservation of SCT. We emphasize that our solution makes the type system more restrictive, it does not affect our model of assembly programs, for which we ultimately seek guarantees. This is

because the semantics of assembly programs assumes a single memory that is initialized at the beginning of execution.

As mentioned above, our example is artificial. Nevertheless, it reflects a real-world issue, as illustrated by the program on the right side of Figure 3, which presents the high-level structure of a signing procedure. This function takes as arguments a pointer m to a *public* message, together with its length len , and a pointer k to a secret key. First, the `preprocess_secret_key` function loads the key from memory; then, we initialize a stack array `state` to a public initialization vector (here, just zeros); finally, we hash the message using `absorb`, block by block, incrementally updating the `state` variable. As the state depends on public data only (the message, its length, and the initialization vector), we would expect it to be typed as *public*. Indeed, the type system for SCT for Jasmin [9] infers that it is *public*, even during misspeculated executions. However, the reader will recognize the pattern of the previous example: `preprocess_secret_key` may store parts of the secret key in local stack variables. Then, the compiler may decide to share the stack regions for that function’s local stack and the `state` variable since they are never simultaneously live. In this—rather typical—situation, an attacker who bypasses the initialization of the `state` (at line 7) would make it depend on secrets at the assembly level.

This highlights that to account for the effects of compilation properly (in this case, allocating local variables in stack frames and sharing the same address space for secret and public data), the notion of security must accommodate uninitialized memory and conservatively assume that such memory contains secret data.

4 Security

In this section, we recast the definition of SCT in an abstract setting, and extend the well-known notion of backward simulations to establish preservation of SCT.

A compiler often handles programs written in different programming languages: source, target, and intermediate representations. The formalism below keeps programming languages fairly unspecified and enables relating programs written in several languages. Syntactically speaking, a programming language is modeled as a set \mathcal{L} of programs. We assume given a set \mathcal{I} of inputs, common to all programming languages. Inputs describe how a program interacts with (reads from) its environment. For concreteness, we will simply consider that inputs are lists of values, to be taken as arguments by the program entry point.

So as to be able to define the security notions of interest, we consider semantics that are *instrumented* with observations and directives. Observations model data that leaks through side-channels and may be available to an adversary. Directives model the ability of an adversary to influence program execution and in particular decisions that are taken speculatively; in essence, directives resolve all non-deterministic choices that may occur during program execution.

The semantics of a programming language is given by a set \mathcal{M} of execution states (referred to as *machines*), a set \mathcal{D} of directives, a set \mathcal{O} of observations, a set \mathcal{F} of outcomes, an initialization function $\mathcal{L} \times \mathcal{I} \rightarrow \mathcal{M}$, and a single-step relation $m \xrightarrow[d]{o} c$ where m is a machine, d a directive, o an observation, and c a *configuration*. A configuration is either a *final* outcome f , denoted $\text{Final}(f)$, or a machine from which execution might continue. We conveniently conflate a program P with the initialization function: $P(i)$ is the initial machine given by the program P and input i , i.e., the initialization function applied on P and i .

We extend the single-step relation to a multi-step relation $c \xrightarrow[d]^{\vec{o}}^* c'$ between machine configurations reading a list of directives and producing a list of observations as shown in Figure 4. When

$$\begin{array}{c}
 \text{NIL} \\
 \hline
 c \xrightarrow[\epsilon]{\epsilon}^* c
 \end{array}
 \qquad
 \begin{array}{c}
 \text{CONS} \\
 \frac{m \xrightarrow[d]{o} c \quad c \xrightarrow[\vec{d}]{\vec{o}}^* c'}{m \xrightarrow[d \cdot \vec{d}]{o \cdot \vec{o}}^* c'}
 \end{array}$$

Fig. 4. Multi-step speculative execution semantics.

there are no directives, execution performs no step and produces no observation (rule **NIL**). Otherwise, a first step is taken according to the first directive, then execution continues, accumulating the observations in a list (rule **CONS**). We denote the empty list as ϵ and use \cdot for concatenation.

The SCT security notion is defined relative to a relation on input values, often denoted ϕ , analogous to the *low-equivalence* relation usually found in the definition of constant-time (or non-interference in general). It expresses what part of the input is public. Frequently, we consider simple relations such as “the first argument is public,” which may be formally defined as $(i_0 \dots i_n) \phi_0 (i'_0 \dots i'_n) := i_0 = i'_0$. However, this formalism is much more general and can express a number of precise conditions. Consider for instance a program that checks that user input (its first argument) matches a predefined secret code (its second argument). This program complies with the following security policy: $(x_1, y_1) \phi_1 (x_2, y_2) := x_1 = x_2 \wedge (x_1 = y_1 \iff x_2 = y_2)$, which states that the value x of the first input is public and that the tested condition is also public (i.e., declassified).

Definition 1 (ϕ -SCT). Let $\phi \subseteq I \times I$ be a relation on inputs. We denote ϕ -SCT the set of programs P (in any language) satisfying

$$\forall i_1 i_2 \vec{d} \vec{o}_1 \vec{o}_2 c_1 c_2, i_1 \phi i_2 \wedge P(i_1) \xrightarrow[\vec{d}]{\vec{o}_1}^* c_1 \wedge P(i_2) \xrightarrow[\vec{d}]{\vec{o}_2}^* c_2 \implies \vec{o}_1 = \vec{o}_2.$$

We now turn to preservation. In the more general setting, we consider transformations from a source language \mathcal{L}_s to a target language \mathcal{L}_t . A transformation $\mathcal{T} \subseteq \mathcal{L}_s \times \mathcal{L}_t$ that relates source and target programs *preserves* ϕ -SCT when it relates ϕ -SCT source programs with ϕ -SCT target programs exclusively, that is, $P \mathcal{T} Q \wedge P \in \phi\text{-SCT} \implies Q \in \phi\text{-SCT}$ for every P and Q . We will assume that the semantics for the source and target languages is instrumented with source and target directives and observations. The transformations presented in this work preserve all relations on inputs, i.e., every ϕ .

To establish this property, we notice that given a program transformation, the observation trace of the target program can usually be explained (i.e., computed) from an observation trace of the source program. Since the semantics cover speculative behaviors and are driven by directives, in order to pick the source execution that explains the target execution, source directives have to be constructed from target directives and public inputs.

Definition 2 (Program ϕ -simulation). There is a *program ϕ -simulation* between two programs $P \in \mathcal{L}_s$ and $Q \in \mathcal{L}_t$ if there exist two functions $\mathcal{T}_d^* : I \times \mathcal{D}_t^* \rightarrow \mathcal{D}_s^*$ and $\mathcal{T}_o^* : I \times \mathcal{D}_t^* \times \mathcal{O}_s^* \rightarrow \mathcal{O}_t^*$ such that

$$\forall i_1 i_2 \vec{d} \vec{o}_t t, i_1 \phi i_2 \wedge Q(i_1) \xrightarrow[\vec{d}]{\vec{o}_t}^* t \implies \exists s \vec{o}_s, P(i_1) \xrightarrow[\mathcal{T}_d^*(i_2, \vec{d})]{\vec{o}_s}^* s \wedge \vec{o}_t = \mathcal{T}_o^*(i_2, \vec{d}, \vec{o}_s).$$

Note that the first argument of the transformers is *any* input i_2 indistinguishable from the one under consideration i_1 . This captures the notion of “public inputs.” Using this definition, we can achieve secure compilation with the following result.

THEOREM 1 (SECURE COMPILATION). *For any programs $P \in \mathcal{L}_s$ and $Q \in \mathcal{L}_t$, and reflexive relation ϕ , if P is ϕ -SCT and there is a program ϕ -simulation between P and Q , then Q is ϕ -SCT.*

PROOF. This follows from the more general result we prove in Theorem 3. \square

Lemma `transformer_is_sct`, in file `completeness.v` of the CoQ development, corresponds to the one above.

In Section 4.1 we provide a completeness result showing that if P is ϕ -SCT and Q is ϕ -SCT then there is a program ϕ -simulation between P and Q . However, for the purposes of proving our compiler secure, we do not need transformers to depend on inputs, the sequence of directives is enough. This less powerful alternative of Definition 2 does not require the relation to be reflexive.

Theorem 1 remains complex to apply directly. However, we can employ a standard technique of certified compilation: verifying a program transformation often entails identifying a relation between execution states of the source and target programs that is maintained during their executions. Such a relation is called a simulation relation, and proving it amounts to establishing a simulation diagram. One of the simplest forms of such a diagram arises when source and target executions progress simultaneously and when directives and observations are transformed similarly at every step.

Definition 3 (Backward Lock-Step Simulation). For any programs $P \in \mathcal{L}_s$ and $Q \in \mathcal{L}_t$, a *lock-step simulation* is a relation \sim (between source and target configurations), together with a directive transformer $\mathcal{T}_d : \mathcal{D}_t \rightarrow \mathcal{D}_s$ and an observation transformer $\mathcal{T}_o : \mathcal{D}_t \rightarrow \mathcal{O}_s \rightarrow \mathcal{O}_t$, where

- (1) initial configurations are related, i.e., $\forall i, P(i) \sim Q(i)$; and
- (2) it is preserved at every step, that is, the following diagram holds:

$$\begin{array}{ccc} \forall d \ o_t \ s \ t \ t', s \sim t \wedge t \xrightarrow[d]{o_t} t' \implies & & \begin{array}{ccc} s & \xrightarrow[\mathcal{T}_d(d)]{o_s} & s' \\ \vdots & & \vdots \\ t & \xrightarrow[d]{\mathcal{T}_o(d, o_s)} & t' \end{array} \\ \exists s' \ o_s, s \xrightarrow[\mathcal{T}_d(d)]{o_s} s' \wedge o_t = \mathcal{T}_o(d, o_s) \wedge s' \sim t'. & & \end{array}$$

(In the diagram, hypotheses are in **thin blue** and conclusions in **bold red**.)

In many situations, this diagram is too strong. Extending it to a multi-step execution in the source helps cover most cases, yet certain passes in Section 6 require an even more general notion. We must allow transformers to depend on the history of execution, as realized by the list of (target) directives that have been followed from an initial state to reach the target state under consideration. Thus, the simulation relation will now be indexed by the previous directives.

Definition 4 (Backward Simulation). For any programs $P \in \mathcal{L}_s$ and $Q \in \mathcal{L}_t$, a *simulation* is a relation \sim (between source and target configurations, indexed by lists of directives), together with a directive transformer $\mathcal{T}_d : \mathcal{D}_t^* \times \mathcal{D}_t \rightarrow \mathcal{D}_s^*$ and an observation transformer $\mathcal{T}_o : \mathcal{D}_t^* \times \mathcal{D}_t \times \mathcal{O}_s^* \rightarrow \mathcal{O}_t$, where

- (1) initial configurations are related (with initial history), i.e., $\forall i, P(i) \sim_\epsilon Q(i)$; and
- (2) it is preserved at every step (accumulating the directive as history), that is, the following diagram holds:

$$\begin{array}{ccc} \forall \vec{h} \ d \ o_t \ s \ t \ t', s \sim_{\vec{h}} t \wedge t \xrightarrow[d]{o_t} t' \implies & & \begin{array}{ccc} s & \xrightarrow[\mathcal{T}_d(\vec{h}, d)]{\vec{o}_s} & s' \\ \vdots & & \vdots \\ t & \xrightarrow[d]{\mathcal{T}_o(\vec{h}, d, \vec{o}_s)} & t' \end{array} \\ \exists s' \ \vec{o}_s, s \xrightarrow[\mathcal{T}_d(\vec{h}, d)]{\vec{o}_s} s' \wedge o_t = \mathcal{T}_o(\vec{h}, d, \vec{o}_s) \wedge s' \sim_{\vec{h}.d} t'. & & \end{array}$$

(Note that \vec{h} is the list of directives executed until this point.)

We can now show the main theorem we use for each pass of our compiler. As mentioned above, we will construct a program simulation whose transformers depend on directives only, ignoring inputs.

THEOREM 2 (SOUNDNESS OF BACKWARD SIMULATIONS). *Given two programs P and Q , ϕ a relation on inputs, and \sim , \mathcal{T}_d , and \mathcal{T}_o satisfying the backward simulation conditions (Definition 4), then if P is ϕ -SCT, Q is ϕ -SCT.*

PROOF. This proof consists in showing that we can apply Theorem 1 by lifting the backward simulation to a program simulation (Definition 2), which we do by induction on the target directives.

Thus, we first define auxiliary multi-step transformers for directives \mathcal{T}_d^{aux} and observations \mathcal{T}_o^{aux} by accumulating the results of the single-step transformers. Intuitively, to match a target execution under directives \vec{d} , we must produce the source steps for d_1 , then for $d_1 \cdot d_2$, and so on until \vec{d} . More precisely,

$$\begin{aligned} \mathcal{T}_d^{aux}(\vec{h}, \vec{d}) &:= \mathcal{T}_d(\vec{h}, d_1) \cdot \mathcal{T}_d(\vec{h} \cdot d_1, d_2) \cdot \dots \cdot \mathcal{T}_d(\vec{h} \cdot \vec{d}_{1:n-1}, d_n) \\ \mathcal{T}_o^{aux}(\vec{h}, \vec{d}, \vec{o}) &:= \mathcal{T}_o(\vec{h}, d_1, \vec{o}_{1:ss_1}) \cdot \\ &\quad \mathcal{T}_o(\vec{h} \cdot d_1, d_2, \vec{o}_{ss_1+1:ss_2}) \cdot \\ &\quad \dots \cdot \\ &\quad \mathcal{T}_o(\vec{h} \cdot \vec{d}_{1:n-1}, d_n, \vec{o}_{ss_{n-1}+1:ss_n}), \end{aligned}$$

where n is the length of \vec{d} , we use $\vec{v}_{i:j}$ to denote $v_i \cdot \dots \cdot v_j$, and $ss_k := \left| \mathcal{T}_d^{aux}(\vec{h}, d_1 \cdot \dots \cdot d_k) \right|$ is the number of *source steps* matching the first k target steps. Note that $\vec{o}_{ss_{k-1}+1:ss_k}$ are the observations of the source execution matching the k -th target step. In applying the theorem, we will show that these indices are valid, i.e., that \vec{o} is sufficiently long. Now we can define the multi-step transformer for directives $\mathcal{T}_d^*(\vec{d}) := \mathcal{T}_d^{aux}(\epsilon, \vec{d})$ and for observations $\mathcal{T}_o^*(\vec{d}, \vec{o}) := \mathcal{T}_o^{aux}(\epsilon, \vec{d}, \vec{o})$.

Applying Theorem 1 with these transformers, it now suffices to show

$$\forall \vec{d} \vec{o}_t \ i \ t', Q(i) \xrightarrow[\vec{d}]{\vec{o}_t}^* t' \implies \exists s' \ \vec{o}_s, \begin{cases} P(i) \xrightarrow[\mathcal{T}_d^{aux}(\epsilon, \vec{d})]{\vec{o}_s}^* s', & \text{(A)} \\ \vec{o}_t = \mathcal{T}_o^{aux}(\epsilon, \vec{d}, \vec{o}_s). & \text{(B)} \end{cases}$$

We now proceed by induction on \vec{d} , generalizing $P(i)$ and $Q(i)$ as s and t , together with the fact that $P(i) \sim_\epsilon Q(i)$ and the initial history ϵ . The base case follows from choosing s and ϵ , as there are no steps to perform.

In the inductive case, we will use the backward simulation hypothesis to perform the step dictated by d , and then the inductive hypothesis to perform the rest—i.e., those dictated by \vec{d}' . The resulting observations will be $\mathcal{T}_o(\vec{h}, d, \vec{o}_s)$ and $\mathcal{T}_o^{aux}(\vec{h} \cdot d, \vec{d}', \vec{o}'_s)$, which match the required $\mathcal{T}_o^{aux}(\vec{h}, d \cdot \vec{d}', \vec{o}_s \cdot \vec{o}'_s)$ since the number of directives an observations in an execution is the same. We provide the details of this proof in the appendix. \square

We refer the reader to lemma `lift_step_preserves_sim`, in file `language.v` of the Coq development, for a mechanized proof of the theorem above.

For some passes of the compiler, we need to assume that the source program is safe, in a sense that we make precise in Section 6. In those cases, we use a weaker version of the diagrams and soundness theorem, with this additional hypothesis.

4.1 Completeness

In this section, we show that the method of simulations with transformers introduced above is complete in the following sense: given P that is ϕ -SCT, Q is ϕ -SCT if and only if there exists a program ϕ -simulation between P and Q . Furthermore, we show the equivalence of another methodology, speculative constant-time cubes, a natural extension of constant-time cubes [17] to the speculative setting. We will assume that the semantics is deterministic, which is the case for our model, as proven in lemma `deterministic_sem`, in file `semantics_facts.v` of the COQ development.

We first formalize an alternative definition of speculative constant-time in Lemma 1: the leakage of an SCT program depends only on public inputs and directives. In the context of an arbitrary relation ϕ , this means that *any* indistinguishable input suffices to compute the leakage. We require that ϕ is reflexive and relates inputs that accept the same directives (Definition 5).

Definition 5 (Co-Safety). Given a program P , an input i *accepts* directives \vec{d} if there exists \vec{o} and c such that $P(i) \xrightarrow{\vec{d}}^* c$. We say that a pair of inputs are *co-safe* if they accept the same directives.

This requirement is not problematic, as, in practice, we must show that ϕ distinguishes inputs with different termination behavior regardless. Moreover, we can design our language—particularly, the semantics—so that every relation fulfills this, as we show in Lemma 2.

LEMMA 1. *Let ϕ be reflexive and relate co-safe inputs of P . P is ϕ -SCT if and only if there exists a leakage function $L : \mathcal{I} \times \mathcal{D}^* \rightarrow \mathcal{O}^*$ that computes the observations of a program given public inputs and directives. In symbols, P is ϕ -SCT $\iff \exists L, \forall i_1 i_2 \vec{d} \vec{o} c, i_1 \phi i_2 \wedge P(i_1) \xrightarrow{\vec{d}}^* c \implies \vec{o} = L(i_2, \vec{d})$.*

PROOF. For the forward implication, we define $L(i, \vec{d}) := \vec{o}$ if there exists c such that $P(i) \xrightarrow{\vec{d}}^* c$, and leave it undefined otherwise. This function is well-defined since we assume determinism. Let us now assume that $i_1 \phi i_2$ and $P(i_1) \xrightarrow{\vec{d}}^* c_1$. To show that \vec{o}_1 is $L(i_2, \vec{d})$, we first notice that ϕ relates co-safe inputs, and, therefore, there is an execution $P(i_2) \xrightarrow{\vec{d}}^* c_2$. We then have that $\vec{o}_2 = L(i_2, \vec{d})$, and can apply the ϕ -SCT hypothesis.

For the backward implication, we have L , two executions $P(i_k) \xrightarrow{\vec{d}}^* c_k$ with $k = 1, 2$, we know that $i_1 \phi i_2$, and we will show that the observations are the same. First, note that \vec{o}_2 is $L(i_1, \vec{d})$, by the property of L , the fact that i_1 and i_2 are related, and the second execution. Additionally, we can also show $\vec{o}_1 = L(i_1, \vec{d})$, given the first execution and that $i_1 \phi i_1$ holds by reflexivity. \square

We refer the reader to lemmas `is_sct_fun_is_sct` and `is_sct_is_sct_fun`, in file `completeness.v` of the COQ development, for a mechanized proof of the lemma above.

THEOREM 3 (COMPLETENESS). *Let ϕ be reflexive and relate co-safe inputs of P . For every $P \in \mathcal{L}_s$ and $Q \in \mathcal{L}_t$, if P is ϕ -SCT, the following are equivalent:*

- (1) Q is ϕ -SCT;
- (2) There is a program ϕ -simulation between P and Q ; and

(3) There is a ϕ -SCT cube between P and Q , that is,

$$\forall i_1 i_2 \vec{d}_t \vec{o}_{t_1} \vec{o}_{t_2} t_1 t_2, i_1 \phi i_2 \wedge Q(i_k) \xrightarrow[\vec{d}_t]{o_{i_k}^*} t_k \text{ with } k = 1, 2 \implies \\ \exists \vec{d}_s \vec{o}_{s_1} \vec{o}_{s_2} s_1 s_2, P(i_k) \xrightarrow[\vec{d}_s]{o_{s_k}^*} s_k \text{ with } k = 1, 2 \wedge (\vec{o}_{s_1} = \vec{o}_{s_2} \implies \vec{o}_{t_1} = \vec{o}_{t_2}).$$

PROOF. We show the implications in order.

Case: (1) implies (2). Since Q is ϕ -SCT, Lemma 1 gives us a leakage function L for Q . Note that we do not need the source to step and produce observations, since the directives and public inputs already provide the observations we need. We therefore choose $\mathcal{T}_d^*(i, \vec{d}) := \epsilon$ and $\mathcal{T}_o^*(i, \vec{d}, \vec{o}) := L(i, \vec{d})$ as transformers. We have that $i_1 \phi i_2$ and a target execution $Q(i_1) \xrightarrow[\vec{d}]{o_{i_1}^*} t$. Let us choose $P(i_1)$ and ϵ as source machine and observations. The source execution trivially holds by **NIL**, and we need only show $\vec{o}_t = L(i_2, \vec{d})$. We use the property of L with the target execution.

Case: (2) implies (3). We have that $i_1 \phi i_2$ and two target executions $Q(i_k) \xrightarrow[\vec{d}_t]{o_{i_k}^*} t'_k$. Applying the program ϕ -simulation on both, where in the second one we use reflexivity to show that $i_2 \phi i_2$, we get a pair of source executions under $\mathcal{T}_d^*(i_2, \vec{d}_t)$. Choosing the latter directives and the resulting source machines for the existential finishes this case.

Case: (3) implies (1). We have that $i_1 \phi i_2$ and two target executions $Q(i_k) \xrightarrow[\vec{d}]{o_{i_k}^*} c_k$. Applying the ϕ -SCT cube to these executions, we get that $\vec{o}_{s_1} = \vec{o}_{s_2} \implies \vec{o}_{t_1} = \vec{o}_{t_2}$ for a pair of source executions $P(i_k) \xrightarrow[\vec{d}_s]{o_{s_k}^*} s'_k$. Since P is ϕ -SCT, the source observations are equal, and therefore the target ones as well. \square

We refer the reader to lemmas `is_sct_transformer`, `transformer_cube`, and `cube_is_sct`, in file `completeness.v` of the Coq development, for a mechanized proof of the theorem above.

5 Language

We instantiate the framework of the previous section with a core language: programs are written in a simple imperative language with arrays and `selSLH` constructs. The syntax of instructions and code is as follows:

$$i ::= x = e \mid x = a[e] \mid a[e] = x \mid \text{if } e \text{ then } c \text{ else } c \mid \text{while } e \text{ do } c \\ \mid \text{init_msf}() \mid \text{update_msf}(e) \mid x = \text{protect}(x) \\ c ::= \epsilon \mid i; c$$

where e are expressions, x register variables, and a array variables. We assume that each array comes with its size $|a|$. We also assume a distinguished register `msf`.

5.1 Semantics

Sequential Semantics. The sequential semantics of programs is mainly standard. However, our semantics accounts for uninitialized values, and requires that arrays are initialized before their use.

Sequential machines are triples $\langle c, \rho, \mu \rangle$ consisting of the code being executed, a register map mapping register names to optional values, and a memory mapping pairs of arrays and valid indices

to optional values. We use optional values to track initialization: $\mu(a, j) = \perp$ means that the array a is not initialized at index j (and that j is within bounds, i.e., $0 \leq j < |a|$). The operational semantics of programs is given by the relation $m \rightarrow m'$; the rules are standard and omitted. The semantics implicitly assumes that programs are sequentially safe, i.e., all sequential array accesses are within bounds, and arrays are initialized before being read.

Instrumented Semantics. Our instrumented semantics lets execution produce observations. Observations correspond to the kind of side-channels we consider, we will use the classic constant-time leakage model:

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

The observation \bullet corresponds to the observation of a single execution step but no leakage, branch b indicates that the condition of a conditional evaluated to b , $\text{addr } a \ i$ that a memory access to array a in position i occurred. The instrumented semantics of programs is given by the relation $m \xrightarrow{O} m'$; the rules are standard and omitted.

Speculative Semantics. To adapt the instrumented semantics to the speculative setting, we extend sequential machines: we will use 4-tuples $\langle c, \rho, \mu, ms \rangle$, where the last component is a *misspeculation status* that tracks whether misspeculation has (ever) occurred. Note that this means that ms starts as \perp and once it becomes \top it never goes back.

Additionally, we introduce *directives* to model the attacker's power to influence speculation, given by the following syntax:

$$\mathcal{D} ::= \text{step} \mid \text{force } b \mid \text{mem } a \ i.$$

The step directive is used for the execution of atomic instructions, i.e., assignments and SLH instructions. The force b directive is used for the execution of conditional and loops and forces the execution of the b branch, reflecting the attacker's control of the branch predictor. Finally, the mem $a \ i$ directive is used for loads and stores and forces these instructions to read from or write to the adversarially chosen address (a, i) if the instruction performs an unsafe or uninitialized memory access, reflecting our conservative choice to give attackers full control over memory accesses in such cases. Note that this includes constant offsets: an access $b[3]$ allows the attacker to choose any target if 3 is out of bounds. If the access is in bounds and the cell is initialized, the directive is ignored. We assume that, in a memory directive, the index is always in bounds of the array.

The speculative semantics of programs $m \xrightarrow[\mathcal{D}]{O} m'$ and its multi-step version are defined in Figure 5 and Figure 4 respectively. We briefly comment on the rules.

The **ASSIGN** rule models the usual semantics of an assignment, where the only possible directive is step and the observation is always \bullet .

The **N-LOAD** rule models the semantics of loads. The index must be in bounds of the array, and the corresponding cell initialized. The retrieved value is stored into a register, and the address (not the value) is leaked via the observation $\text{addr } a \ i$. As mentioned before, the directive is ignored in this case. The dual of this rule is **S-LOAD**, which models the semantics of loads under misspeculation. This is the case when the index is out-of-bounds or the cell is uninitialized. The directive mem $b \ j$ causes the instruction to load a value from memory address $b[j]$ instead, and store it in a register. Execution still leaks observation $\text{addr } a \ i$. Note that the rule requires that the misspeculation status is \top , as we assume that memory accesses in sequential execution are safe.

The **N-STORE** rule models the semantics of sequential stores. Similarly to the **N-LOAD** case, the index must be in bounds and the cell initialized. On the other hand, the **S-STORE** rule is analogous to the **S-LOAD** rule.

$$\begin{array}{c}
 \text{ASSIGN} \\
 \frac{\rho' = \rho[x \leftarrow \llbracket e \rrbracket_\rho]}{\langle x = e; c, \rho, \mu, ms \rangle \xrightarrow[\text{step}]{\bullet} \langle c, \rho', \mu, ms \rangle} \\
 \\
 \text{N-LOAD} \\
 \frac{\begin{array}{l} \llbracket e \rrbracket_\rho = i \quad i \in [0, |a|) \\ \mu(a, i) = v \quad \rho' = \rho[x \leftarrow v] \end{array}}{\langle x = a[e]; c, \rho, \mu, ms \rangle \xrightarrow[\text{mem } b \ j]{\text{addr } a \ i} \langle c, \rho', \mu, ms \rangle} \\
 \\
 \text{S-LOAD} \\
 \frac{\begin{array}{l} \llbracket e \rrbracket_\rho = i \quad i \notin [0, |a|) \vee \mu(a, i) = \perp \quad j \in [0, |b|) \quad \mu(b, j) = v \end{array}}{\langle x = a[e]; c, \rho, \mu, \top \rangle \xrightarrow[\text{mem } b \ j]{\text{addr } a \ i} \langle c, \rho[x \leftarrow v], \mu, \top \rangle} \\
 \\
 \text{N-STORE} \\
 \frac{\begin{array}{l} \llbracket e \rrbracket_\rho = i \quad i \in [0, |a|) \\ \mu' = \mu[(a, i) \leftarrow \rho(x)] \end{array}}{\langle a[e] = x; c, \rho, \mu, ms \rangle \xrightarrow[\text{mem } b \ j]{\text{addr } a \ i} \langle c, \rho, \mu', ms \rangle} \\
 \\
 \text{S-STORE} \\
 \frac{\begin{array}{l} \llbracket e \rrbracket_\rho = i \quad i \notin [0, |a|) \\ j \in [0, |b|) \quad \mu' = \mu[(b, j) \leftarrow \rho(x)] \end{array}}{\langle a[e] = x; c, \rho, \mu, \top \rangle \xrightarrow[\text{mem } b \ j]{\text{addr } a \ i} \langle c, \rho, \mu', \top \rangle} \\
 \\
 \text{INIT-MSF} \\
 \frac{}{\langle \text{init_msf}(); c, \rho, \mu, \perp \rangle \xrightarrow[\text{step}]{\bullet} \langle c, \rho[\text{msf} \leftarrow \text{NOMASK}], \mu, \perp \rangle} \\
 \\
 \text{UPDATE-MSF} \\
 \frac{v = \text{if } \llbracket e \rrbracket_\rho \text{ then } \rho(\text{msf}) \text{ else MASK}}{\langle \text{update_msf}(e); c, \rho, \mu, ms \rangle \xrightarrow[\text{step}]{\bullet} \langle c, \rho[\text{msf} \leftarrow v], \mu, ms \rangle} \\
 \\
 \text{PROTECT} \\
 \frac{v = \text{if } \rho(\text{msf}) = \text{MASK} \text{ then MASK else } \rho(y)}{\langle x = \text{protect}(y); c, \rho, \mu, ms \rangle \xrightarrow[\text{step}]{\bullet} \langle c, \rho[x \leftarrow v], \mu, ms \rangle} \\
 \\
 \text{COND} \\
 \frac{\begin{array}{l} i = \text{if } e \text{ then } c_\top \text{ else } c_\perp \\ \llbracket e \rrbracket_\rho = b' \quad ms' = ms \vee (b \neq b') \end{array}}{\langle i; c, \rho, \mu, ms \rangle \xrightarrow[\text{force } b]{\text{branch } b'} \langle c_b; c, \rho, \mu, ms' \rangle} \\
 \\
 \text{WHILE} \\
 \frac{\begin{array}{l} i = \text{while } e \text{ do } c_w \quad \llbracket e \rrbracket_\rho = b' \quad c_\perp = \epsilon \\ c_\top = c_w; \text{while } e \text{ do } c_w \quad ms' = ms \vee (b \neq b') \end{array}}{\langle i; c, \rho, \mu, ms \rangle \xrightarrow[\text{force } b]{\text{branch } b'} \langle c_b; c, \rho, \mu, ms' \rangle}
 \end{array}$$

Fig. 5. Single-step speculative execution semantics. Differences with earlier semantics are highlighted in **bold red**.

The following three rules are for the selSLH instructions. The **INIT-MSF** rule shows a speculation fence, which means that the machine must be in sequential execution, and assigns the initial value NOMASK to a distinguished register *msf*. The **UPDATE-MSF** rule is a conditional update of *msf*: it updates its value to MASK if and only if the condition is false. The **PROTECT** rule shows how a value is masked with *msf*. If the value of *msf* is MASK, the output is the default value MASK, and otherwise just the value of the input variable.

The rules for control flow instructions, **COND** and **WHILE**, always follow the adversary's choice of branch given in the force *b* directive. The observation branch *b'* leaks the evaluation of the condition, and the misspeculation status is updated accordingly. Finally, the **NIL** and **CONS** rules in

Table 1. An intuitively insecure program accepted by the previous definition of SCT.

Directive	sec is even	sec is odd
step	$\langle x = a[\text{sec}\%2], \dots \rangle \xrightarrow[\text{step}]{\text{addr } a \ 0} \dots$	stuck
mem	stuck	$\langle x = a[\text{sec}\%2], \dots \rangle \xrightarrow[\text{mem } b \ j]{\text{addr } a \ 1} \dots$

Figure 4 define the semantics for code as the reflexive transitive closure of the one for instructions, accumulating the directives and observations.

Finally, we define the notion of sequential safety for programs, to state the usual safety condition restricted to sequential executions.

Definition 6 (Sequential Safety). A program is *sequentially safe* w.r.t. ϕ if all inputs related by ϕ give sequentially safe initial machines, where *sequentially safe* machines are either final, misspeculating, or can perform a step

$$\begin{aligned} \text{seqsafe}(P, \phi) &:= \forall i_1 \ i_2, i_1 \ \phi \ i_2 \implies \text{seqsafe}(P(i_1)) \wedge \text{seqsafe}(P(i_2)), \\ \text{seqsafe}(m) &:= \forall \vec{d} \ \vec{o} \ m', m \xrightarrow[\vec{d}]{\vec{o}}^* m' \implies m'_c = \epsilon \vee m'_{ms} \vee \exists d \ o \ m'', m' \xrightarrow[d]{o} m''. \end{aligned}$$

(We use subscripts to refer to the components of a machine.)

5.2 Comparison with Prior Semantics

Our semantics differs from [9] in two ways. First, in the **S-LOAD** rule, the clause for uninitialized memory prevents undesirable interactions between uninitialized memory and memory reuse. As a consequence, the program from Figure 3 is speculative constant-time according to [9], but not according to our definition—the load in line 13 is unsafe.

Second, in the **N-LOAD** and **N-STORE** rules, our semantics uses the directive `mem b j` for all load and store operations. This is in contrast with [9], where load and store operations also accept the `step` directive. Unfortunately, using different directives depending on whether the execution step is sequential or misspeculating leads to an unhappy interaction with the notion of SCT defined above. Let us take as an example the program `if (false) then x = a[sec % 2]`, where $|a|$ is 1 and `sec` is secret. Despite clearly leaking the parity of `sec`, this program is speculative constant-time under the semantics from [9]. This is because, for a given sequence of directives, the executions of this program that would expose a difference in the value of the secret are stuck; thus, the (relational) definition of SCT holds vacuously. If the directive for the load instruction is `step`, executions where `sec` is even produce the observation `addr a 0`, and the rest are stuck—since the access is out-of-bounds. Conversely, if the directive is `mem b j`, only executions where `sec` is odd produce an observation, `addr a 1`, while the rest are stuck—since the access is in-bounds. Table 1 illustrates this situation. Since no directive causes the machines to generate different observations, the program is SCT. The semantics presented in the current work does not suffer from this: as memory operations always take a `mem b j` directive, execution leaks the parity of `sec` and the program is thus not SCT.

The semantics from this work rules out this problem:

LEMMA 2. *For any ϕ , if a program is ϕ -SCT, and sequentially safe (Definition 6), then its inputs are co-safe (Definition 5).*

Lemma `is_sct_eq_exec`, in file `both_step.v` of the COQ development, corresponds to the one above.

$$\begin{array}{c}
\text{ERR-N-LOAD} \\
\llbracket e \rrbracket_\rho = i \quad i \in [0, |a|) \quad \mu(a, i) = v \\
m' = \begin{cases} \langle c, \rho[x \leftarrow v], \mu, ms \rangle & \text{if } a, i = b, j \\ \text{err}_M & \text{otherwise} \end{cases} \\
\hline
\langle x = a[e]; c, \rho, \mu, ms \rangle \xrightarrow[\text{mem } b \ j]{\text{addr } a \ i} m'
\end{array}
\qquad
\begin{array}{c}
\text{ERR-N-STORE} \\
\llbracket e \rrbracket_\rho = i \quad i \in [0, |a|) \\
m' = \begin{cases} \langle c, \rho, \mu[(a, i) \leftarrow \rho(x)], ms \rangle & \text{if } a, i = b, j \\ \text{err}_M & \text{otherwise} \end{cases} \\
\hline
\langle a[e] = x; c, \rho, \mu, ms \rangle \xrightarrow[\text{mem } b \ j]{\text{addr } a \ i} m'
\end{array}$$

Fig. 6. Semantics with error.

5.3 Error Semantics

The proof of the two most complex passes of this work, array reuse and concatenation, require a refinement of the semantics from Figure 5 that is more informative in the rules for memory operations, but equivalent for the purposes of showing SCT. The refinement replaces the **N-LOAD** and **N-STORE** rules as shown in Figure 6. The new rules enforce that if the memory operation is sequential, in bounds, and initialized, the directive is not ignored but required to match exactly the memory address in the code. It introduces a distinguished “error machine” err_M , to indicate a mismatch. The advantage of this semantics is that both the control flow and the memory accesses of an execution are uniquely determined by the sequence of directives. The disadvantage is that it is not immediately clear that restricting the semantics in this way faithfully models speculative execution. However, since the definition of speculative constant-time quantifies over all sequences of directives, we are able to prove that for the purposes of proving SCT they are equivalent. That is, a program is speculative constant-time under the semantics presented in Figure 5 if and only if it is speculative constant-time under the error semantics (i.e., with the rules from Figure 6).

LEMMA 3 (EQUIVALENCE OF ERROR SEMANTICS). *A program is ϕ -SCT if and only if it is ϕ -SCT $_{\text{err}_M}$.*

We refer the reader to lemma `std_sem_sem` and lemma `seqsafe_new_stfd`, in file `equiv_sem.v` of the COQ development, for a proof of this lemma.

6 Preservation of SCT by Compilation

In this section, we consider nine standard (simplified) compiler passes: constant folding, constant propagation, dead assignment elimination, loop peeling, dead branch elimination, array reuse, register allocation, array concatenation, and linearization, and prove that they preserve SCT. In broad terms, the proofs of all passes of our compiler involve two main components: first, we ensure a degree of functional correctness in sequential execution such that the evaluation of expressions leaked by the program is the same. Second, we prove that the compiler does not introduce new opportunities for speculative leakage, by showing that target leakage can be computed from source leakage. The proof of some of the passes of this work require sequential safety of the source, as mentioned at the end of Section 4.

All the proofs presented in this section use backward simulations and Theorem 2, instantiated with the semantics with error machines from Section 5.3. By convention, we lift every simulation relation \sim to optional machines in the conclusion of each backward lemma as follows:

$$s \sim_{\text{err}_M} t := \begin{cases} \top & \text{if } t \text{ is } \text{err}_M, \\ s \neq \text{err}_M \wedge s \sim t & \text{otherwise.} \end{cases}$$

In other words, an error in the target means that there is nothing to prove, otherwise we must show that the source is not an error and that they are related.

We now establish preservation of speculative constant-time for individual passes of the compiler. For each pass, we first briefly explain its goals and mechanics and then turn to the proof of preservation of SCT; in particular, we provide details about the simulation and the directive and observation transformers. Note that many of our compiler passes are defined abstractly, relative to oracles that perform standard analyses, such as liveness analysis for dead assignment elimination or valid renamings for register allocation. Such an abstract view of compiler passes is compatible and even beneficial with our goal to explore the feasibility of preserving SCT. Moreover, instantiating these oracles with concrete analyzers has no implication on security.

Finally, note that all but the last pass are source-to-source translations of the language described in Section 5. On the other hand, linearization produces a program in a language that features the same basic instructions but changes the representation of a program to a directed graph, as explained below.

Constant Folding. This pass replaces composite constant expressions in the program with their evaluation. For instance, the expression $x + (3 - 1)$ goes to $x + 2$. It does not introduce or remove instructions, it replaces only expressions in the program. As this pass is quite simple, we do not need to parameterize over any analyses.

A lock-step simulation (Definition 3) suffices to prove SCT preservation for this pass, using the soundness of backward simulations (Theorem 2). The simulation relation for this pass states that the code of the target machine is the compilation of the source's and that their states are equal. The transformers for directives and observations are the identity, so we only need to show that the machines are in lock-step and that their leakage is the same. The former comes from the pass not adding or removing instructions, the latter from leakage containing only concrete values—the leakage of $x = a[3 - 2]$ and $x = a[1]$ is the same, $\text{addr } a$.

Constant Propagation. This pass replaces variables with their values when they are constant. For example, $x = 3; y += x + 4$ gets transformed into $x = 3; y += 3 + 4$. It does not introduce or remove instructions; it replaces the expressions in the program—specifically, in assignments, memory operations, MSF updates, conditionals, and loops. This pass is parameterized by an oracle that attaches to each program point a map from variables to optional values, meaning that at this point, a variable is guaranteed to hold the associated value—e.g., in the second instruction of the example from before, we would get a map $\{x \rightarrow 3, y \rightarrow \perp\}$.

We prove the simulation under the assumption that the oracle is functionally correct, the standard requirement for a correct compiler. We establish a lock-step simulation, where the relation states that the code of the target is the compilation of the source's, that their states are the same, and that the map given by the oracle is correct w.r.t. the state—i.e., that variables have their predicted values. The directive and observation transformers are the identity, so we only need to prove that they are in lock-step and the evaluation of leaked expressions coincide.

Dead Assignment Elimination. This pass removes some instructions that are not needed for the program. Specifically, it removes assignments, MSF updates, and protects when they do not contribute to the result or leakage of the program; it does not remove any other instruction. We remark that we can remove MSF updates (resp. protects) without compromising security only if the MSF is unused (resp. the protected value is not used). It is parameterized by an oracle that returns the set of needed variables at each program point; we guarantee the correctness of this information with a verified checker. A straightforward extension of this pass can remove protects when the value is used for computation but not leaked.

The checker ensures that the liveness information from the oracle is at most that of the standard (sequential) liveness analysis, except in the case of loads. The problem with loads is that to issue

a mem a i directive we need $i \in [0, |a|)$, which is only possible if we know that there exists an array with size greater than zero in the program. The checker verifies that the oracle O satisfies $O(i, S) \subseteq \text{live}(i, S)$ at each instruction i , where

$$\begin{aligned} \text{live}(x = e, S) &:= \begin{cases} (S \setminus \{x\}) \cup \text{FV}(e) & \text{if } x \in S \\ S & \text{otherwise} \end{cases} & \text{live}(a[e] = x, S) &:= S \cup \{x\} \cup \text{FV}(e) \\ \text{live}(\text{update_msf}(e), S) &:= \begin{cases} S \cup \text{FV}(e) & \text{if } \text{msf} \in S \\ S & \text{otherwise} \end{cases} & \text{live}(x = a[e], S) &:= (S \setminus \{x\}) \cup \text{FV}(e) \\ & & \text{live}(\text{init_msf}(), S) &:= S \setminus \{\text{msf}\} \\ \text{live}(x = \text{protect}(y), S) &:= \begin{cases} (S \setminus \{x\}) \cup \{y, \text{msf}\} & \text{if } x \in S \\ S & \text{otherwise} \end{cases} \end{aligned}$$

and the cases for conditional and loops are recursive as usual, where $\text{FV}(e)$ are the free variables of e .

This pass requires a multi-step simulation without history. This simulation relation states that the code of the target is the compilation of the source's, the register maps coincide on the set of needed variables, and the memories and misspeculation statuses are the same, that is

$$t_c = \langle s_c, S \rangle, \quad t_\rho = s_\rho, \quad t_\mu = s_\mu, \quad \text{and} \quad t_{ms} = s_{ms}$$

where S is given by the oracle and $\rho =_S \rho'$ means that ρ and ρ' coincide on the registers in S . Its transformers remove some step directives and \bullet observations in the target, corresponding to dead instructions

$$\langle d_t, c \rangle_{\mathcal{D}} := \begin{cases} \text{step} \cdot \langle d_t, c' \rangle_{\mathcal{D}} & \text{if } c = i; c' \wedge \text{dead}(i, c'), \\ d_t & \text{otherwise,} \end{cases} \quad \langle \vec{o}_s \rangle_O := \text{last}(\vec{o}_s),$$

where $\text{dead}(i, c)$ means that the written variables of i are not needed by c , as determined by the oracle. This means that the source will perform several steps for each target step, for the dead assignments preceding the target instruction: if the target performs a step under directive d_t and observation o_t , the source needs $\text{step} \dots \text{step} \cdot d_t$ and $\bullet \dots \bullet \cdot o_t$.

Loop Peeling. This pass performs one unrolling of some loops. Specifically, it transforms loops while e do c into conditionals if e then c' ; while e do c' else ϵ where c' is the transformation of c . We assume that certain loops are annotated as candidates for unrolling and perform this peeling only on them—this annotation may come, for instance, from an analysis that expects the loop range to be known at compile time. We can unroll some loops by combining this pass with constant propagation, folding, and dead assignment and branch elimination. Since the semantics is preserved exactly, we need no constraints on the oracle.

The simulation for this pass is lock-step, where the relation states that the target code is either the compilation of the source's or a peeled loop after its first iteration

$$\frac{}{\epsilon \sim \epsilon} \quad \frac{c_s \sim c_t}{i_s; c_s \sim \langle i_s, O \rangle; c_t} \quad \frac{c'_s \sim c'_t \quad c_s \sim c_t}{\text{while } e \text{ do } c'_s; c_s \sim \text{while } e \text{ do } c'_t; c_t}$$

where O is the oracle indicating which loops should be peeled. Note that the requirement for code is not simply $t_c = \langle s_c, O \rangle$ as usual; this is because, in the first iteration, the source executes a loop and the target executes a conditional (the compilation of said loop), however, for all the following iterations both execute the same loop. The simulation also requires that their states are equal. Since the directive for both conditionals and loops is force b , and we do not change any instruction that updates the state, the directive and observation transformers are the identity, and the machines are

in lock-step. As evinced by the simulation for code, there are three subcases for loops: a loop that was not peeled, a peeled loop in its first iteration, and one after the first iteration.

Dead Branch Elimination. This pass simplifies conditionals and loops whose guards are constant. That is, the pass transforms conditionals $\text{if } b \text{ then } c_{\top} \text{ else } c_{\perp}$ into the transformation of c_b and loops $\text{while } \perp \text{ do } c$ into ϵ . It does not use oracles, as it simply matches expressions on exact constants.

The simulation for this pass is multi-step, with history. The relation is straightforward: the code of the target is the compilation of the source's, and the states are equal. The directive and observation transformers must account for the removed branches, akin to dead assignment elimination. That is, when the target executes the then-branch of $\text{if } \top \text{ then } \dots \text{ else } \dots$, the source needs to do an extra step to enter said branch, namely force \top . Consequently, the transformers use history to compute the code of the source machine and thus the prefix of force b directives. Hence, we define pre to compute this prefix: it returns a pair of a boolean, indicating whether all instructions have been removed, and the prefix of removed force b directives until now. Then, we define the transformers as follows:

$$\begin{aligned} \text{pre}(i) &:= \begin{cases} (r, \text{force } b \cdot \vec{d}) & \text{if } i = \text{if } b \text{ then } c_{\top} \text{ else } c_{\perp} \wedge \text{pre}(c_b) = (r, \vec{d}), \\ (\perp, \text{force } \perp) & \text{if } i = \text{while } \perp \text{ do } c, \\ (\perp, \epsilon) & \text{otherwise,} \end{cases} \\ \text{pre}(c) &:= \begin{cases} (r, \vec{d}_i \cdot \vec{d}_c) & \text{if } c = i; c' \wedge \text{pre}(i) = (\top, \vec{d}_i) \wedge \text{pre}(c') = (r, \vec{d}_c), \\ (\perp, \vec{d}_i) & \text{if } c = i; c' \wedge \text{pre}(i) = (\perp, \vec{d}_i), \\ (\top, \epsilon) & \text{otherwise,} \end{cases} \\ \langle d_t, c \rangle_{\mathcal{D}} &:= \vec{d}_c \cdot d_t \quad \text{where } (_, \vec{d}_c) = \text{pre}(c). \end{aligned}$$

Intuitively, the directive transformer consumes as many conditionals and loops with constant guards from the code of the source machine as possible, for which it issues force b directives, and ends the sequence with the target directive. Schematically, if the target executes under a directive d_t and produces an observation o_t , the source will do so under directives $\text{force } b_1 \cdot \dots \cdot \text{force } b_n \cdot d_t$ and observations $\text{branch } b_1 \cdot \dots \cdot \text{branch } b_n \cdot o_t$. The transformer for observations is identical to the one of dead assignment elimination.

We split the backward SCT simulation proof of this pass into two lemmas. Let s and t be a source and target machine, such that they are in the simulation. We will first show that if the pass does not remove the first instruction of s , they both step under the same directive and produce the same observation. Then, we show that it is always possible for s to execute a (possibly empty) sequence of force b steps until we fall in the first case. Note that the last lemma, when the source stutters, need not say anything about the source observations. Indeed, these are irrelevant since the last execution step of the source always comes from the execution in the first lemma, when both the source and target step, and $\langle \vec{o}_s \rangle_{\mathcal{O}} = \text{last}(\vec{o}_s) = o_t$.

Array Reuse. This pass aims to reduce (stack) memory usage by reusing dead arrays. As an illustration, let a and b be two arrays of equal size in a program such that its first part does not use b and its second part does not use a —that is, the two arrays are never simultaneously live. In this situation, the compiler can safely replace all uses of a and b with a new array of the same size, c , thus requiring one memory region instead of two. The first two columns of Table 2, “Source” and “Target,” illustrate this example. Array reuse, therefore, consists of simply renaming array variables in loads and stores at annotated points. We restrict this pass to replace only arrays of equal size for simplicity.

Table 2. An example of array reuse where it is challenging to define the directive transformer, where the size of arrays a , b , and c is two.

Source	Target	uninit	$\text{rtbl}^{-1}(c)$	\mathcal{D}_t	\mathcal{D}_s	$\text{wtbl}(c)$
$a[0] = s;$	$c[0] = s;$	$\{a, b\}$	–	mem c 0	mem a 0	$[a, a]$
$b[1] = 1;$	$c[1] = 1;$	$\{b\}$	a	mem c 1	mem b 0	$[a, a]$
if \perp then	if \perp then	\emptyset	b	force \top	force \top	$[a, b]$
$x = b[0];$	$x = c[0];$	\emptyset	b	mem c 0	mem a 0	$[a, b]$

This pass requires an oracle, a *reuse (or renaming) table*, denoted rtbl , which indicates the renaming that the compiler should perform. With this notation, array reuse consists of replacing each array a in the program by $\text{rtbl}(a)$. We remark that the point of this pass is for rtbl *not* to be injective. Naturally, we need to ensure that this table is valid w.r.t. the source program, which means that if the compiler wants to rename a and b into c , as in the example, no cell of b can be initialized while its counterpart in a is live. Consequently, we require an analysis that computes two pieces of information at each program point: a set, denoted uninit , of the arrays with no initialized cells at this point; and a mapping, denoted rtbl^{-1} , from target arrays to source arrays, such that each target coincides with its mapping on its initialized cells. If the instruction at the current program point is $a[e] = x$, then the correctness of the analysis entails that either $a \in \text{uninit}$ or $\text{rtbl}^{-1}(\text{rtbl}(a)) = a$. Moreover, the analysis after this store (i.e., for the next instruction) removes a from uninit —because one of its cells is now initialized—and asserts $\text{rtbl}^{-1}(\text{rtbl}(a)) = a$. A similar condition holds for loads, except that the analysis after the instruction is unchanged.

The conditions presented above are sufficient to prove functional correctness but not preservation of SCT. Figure 2 illustrates the problem: Recall that the first two columns depict the example program from before, where source arrays a and b get renamed into c . Note that this program is sequentially safe, and the renaming is sequentially correct, but the source performs an uninitialized memory access under misspeculation. The second and third columns show the static information from the compile-time analysis needed to check the validity of the transformation. The “uninit” column shows the set of uninitialized source arrays at each program point, and the “ $\text{rtbl}^{-1}(c)$ ” column shows the source array that was renamed into c at that point, if any. The fourth and fifth columns show dynamic information corresponding to an example execution. The “ \mathcal{D}_t ” column lists directives for the target program—note that the branch misspeculates. At the end of this target execution, x will hold the value s . In order to explain this behavior at the source level, we need to construct the source directives shown in the “ \mathcal{D}_s ” column. For safe loads, we can use rtbl^{-1} to transform the first two directives. However, the static information we have is not enough to transform the last directive: we need to remember that the first cell of c , which at that point is the renaming of b , has the same value as the first cell of a . This information is not available from the static data alone, as there may be more than two arrays renamed to c , and, in fact, may depend on the specific execution.

Nevertheless, there is still hope for our proof: using the semantics with error presented in Section 5.3, we can determine which source array corresponds to unsafe memory accesses from the previous directives of the execution. This means that we will prove a lock-step simulation indexed by history. In our example, we know that at the time we executed the previous write with target directive mem c 0, the analysis gave $\text{rtbl}(a) = c$, so the source array that coincides with c at position 0 is a . This is what the *write table* does, denoted “ $\text{wtbl}(c)$,” which we show in the last column: the transformers use the history of directives to compute, for each cell of each target array,

which source array contains the corresponding value (from the last write to the cell). Contrary to rtbl^{-1} , this information cannot be computed statically but can be derived—in the semantics with error—from the history of directives. In Table 2, we see that after the first store, with $\text{mem } c \ 0$ and $\text{rtbl}(a) = c$, we update $\text{wtbl}(c)$ with a in the first cell (in **bold red**). After the second store, with $\text{mem } c \ 1$ and $\text{rtbl}(b) = c$, we update it with b in the second cell (also in **bold red**). Other instructions do not update this table.

In this manner, we can parameterize the directive transformer with a write table, and compute the next one after each execution step, from the target directive, the source code, the reuse table, and the current write table, all of which are public data. The precise definition of the transformer is

$$\langle d, c, \text{wtbl} \rangle_{\mathcal{D}} = \begin{cases} \text{mem } \text{wtbl}(b)[j] \ j & \text{if } d = \text{mem } b \ j \wedge c = x = a[e]; c', \\ \text{mem } \text{rtbl}^{-1}(b) \ j & \text{if } d = \text{mem } b \ j \wedge c = a[e] = x; c' \wedge \text{rtbl}(a) \neq b, \\ \text{mem } a \ j & \text{if } d = \text{mem } b \ j \wedge c = a[e] = x; c' \wedge \text{rtbl}(a) = b, \\ d & \text{otherwise.} \end{cases}$$

The transformer for observations is straightforward: it renames a source observation $\text{addr } a \ i$ into the target observation $\text{addr } \text{rtbl}(a) \ i$ and leaves other observations unchanged.

The simulation relation of array reuse is that the code of the target is the compilation of the source's, that the states are equal modulo the renaming of arrays given by the oracle, and that the information from the write table is correct. Concretely, we say that a source machine s is in relation with a target machine t with respect to wtbl if

$$\text{valid}(s_c), \quad t_c = \langle s_c, \text{rtbl} \rangle, \quad t_\rho = s_\rho, \quad s_\mu, t_\mu \models \text{uninit}, \text{rtbl}^{-1}, \text{wtbl}, \quad \text{and} \quad t_{ms} = s_{ms}$$

where $s_\mu, t_\mu \models \text{uninit}, \text{rtbl}^{-1}, \text{wtbl}$ means that the memories validate the information from the analyses and the write table.

Register Allocation. This pass renames the register variables of a program. It does not introduce or remove instructions. It can be used, for instance, to make a program require only architectural registers (such as RAX, RDI, RSI, etc.) after a different pass has introduced the appropriate spilling; and also in standard compiler passes such as inlining, which we do not consider in the present work. It has two components: an untrusted function that performs the renaming and a verified checker. The function takes and returns a program, and the checker ensures they are alpha equivalent. We do not implement the renaming, only the checker—i.e., the pass is parameterized over the renaming function.

The simulation for this pass is lock-step, where the relation states that the code of the target machine is a valid renaming of that of the source and that the states are equal modulo this renaming (note that the renaming is of register variables only, so the condition on states for memories and misspeculation statuses is equality). Specifically, we require $t_\rho(x) = s_\rho(O(x))$ for each register variable x , where O is the renaming function at the current point. The directive and leakage transformers are the identity. Thus, we show that the machines are in lock-step and that source and target expressions evaluate to the same value.

Array Concatenation. This pass renames array accesses to use one large array instead of many small ones. That is, instead of using a and b with $|a| = 2$ and $|b| = 3$, we can use c with $|c| = 5$ and replace all instances of $a[e]$ with $c[e]$ and $b[e]$ with $c[e + 2]$. Compilers usually feature a pass like this to introduce stack frames with the local data of a function. This pass is parameterized by an oracle that maps source arrays to target arrays and offsets: in the example from before, we get $\{a \rightarrow (c, 0), b \rightarrow (c, 2)\}$.

As usual, we must ensure some level of functional correctness. Under sequential execution, we need two conditions on the oracle O for this pass to be correct: that the target array can fit the source array, that is, that for every source array a , target array c , and offset n , we ensure that

$$O(a) = (c, n) \implies [n, n + |a|] \subseteq [0, |c|],$$

and that source arrays do not overlap inside the target, that is, that for every two source arrays a and b , target array c , and offsets n and m , we require that

$$O(a) = (c, n) \wedge O(b) = (c, m) \implies [n, n + |a|] \cap [m, m + |b|] = \emptyset.$$

Unfortunately, these correctness conditions are insufficient to prove preservation of speculative constant-time. Since an adversary might be able to speculatively read from the target array c at any offset, we must also require that this pass is no more than a renaming of memory regions and that we are not exposing “new” memory for the attacker to read. That is to say, we need every cell of each target array to correspond to a cell in a source array—put differently, there must be no holes between source arrays: for each target array c and position i we need that

$$i \in [0, |c|] \implies \exists a n, O(a) = (c, n) \wedge i \in [n, n + |a|].$$

This restriction looks problematic, for instance, if the compiler needs to align source arrays, but it is not so. As explained before, arrays in our model are disjoint memory regions; this condition states that the compiler must introduce no new regions. It is always possible to consider that the source program has more regions than it uses, uninitialized “dummy” arrays, that will serve as padding between other source arrays.

Using these three hypotheses, we can show that there exists an inverse function O^{-1} of the oracle, satisfying that for every source array a , target array c , offset n , and position j ,

$$j \in [n, n + |a|] \wedge O(a) = (c, n) \implies O^{-1}(c, j) = (a, j - n).$$

We can now establish a lock-step simulation for this pass, with the relation saying that the code of the target is the compilation of the source’s and that their states are equal modulo the renaming of memory addresses from the oracle. More precisely, we ask that their register maps and misspeculation statuses are equal, and that $t_\mu(c, j) = s_\mu(O^{-1}(c, j))$ for each target array c and position j . The directive and observation transformers rename mem $a i$ directives and addr $a i$ observations according to the oracle, and leave the other directives and observations unmodified

$$\langle d_t \rangle_{\mathcal{D}} := \begin{cases} \text{mem } a i & \text{if } d_t = \text{mem } c j \\ & \text{and } O^{-1}(c, j) = (a, i), \\ d_t & \text{otherwise,} \end{cases} \quad \langle o_s \rangle_{\mathcal{O}} := \begin{cases} \text{addr } c (n + i) & \text{if } o_s = \text{addr } a i \\ & \text{and } O(a) = (c, n), \\ o & \text{otherwise.} \end{cases}$$

Linearization. This pass removes structured control flow (i.e., conditionals and loops) by using a graph representation of the program. The target language has the same basic instructions as the source, e.g., assignments, loads, and stores, but the edges of the graph capture control flow. A program in the target language is a directed graph where the nodes are instructions and out-neighbors are successor instructions; instructions have exactly one successor except conditional jumps, which have two. Programs also have two distinguished labels, entry and exit, to start and halt execution. Machines in this language are similar to source machines, with a program label instead of code, i.e., $\langle \ell, \rho, \mu, ms \rangle$. We denote $\langle \ell, \rho, \mu, ms \rangle_{pc}$ for the program counter of this machine, ℓ .

We characterize this pass as a relation between source and target programs, where $\ell : i_t \rightarrow \vec{\ell}$ means that label ℓ points to the target instruction i_t , and its successors are $\vec{\ell}$, and we define $\ell : (i_s) \rightarrow \vec{\ell}$ similarly but for the linearization of the source instruction i_s . As the specification of

this pass is a relation, our compiler is parametric on the exact way linearization is done as long as it satisfies the relation. The nontrivial cases of the compilation scheme are

$$\frac{\begin{array}{l} \ell : \text{if } e \text{ jump} \rightarrow \ell_{\top}, \ell_{\perp} \\ \ell_{\top} : \langle c_{\top} \rangle \rightarrow \ell' \\ \ell_{\perp} : \langle c_{\perp} \rangle \rightarrow \ell' \end{array}}{\ell : \langle \text{if } e \text{ then } c_{\top} \text{ else } c_{\perp} \rangle \rightarrow \ell'} \quad \frac{\begin{array}{l} \ell : \text{if } e \text{ jump} \rightarrow \ell_{\text{body}}, \ell' \\ \ell_{\text{body}} : \langle c \rangle \rightarrow \ell \end{array}}{\ell : \langle \text{while } e \text{ do } c \rangle \rightarrow \ell'}$$

This pass should be followed by one that places basic blocks sequentially instead of having a graph representation. Such a pass simply needs to find program points with more than one in-neighbor and introduce an unconditional jump in one of the branches. It would preserve speculative constant-time since unconditional jumps do not allow speculation or leak information.

The simulation for this pass is lock-step, relating machines when the label of the target machine points to either the compilation of the code of the source machine or the exit label of the program

$$c \sim pc := \begin{cases} pc = \ell_{\text{exit}} & \text{if } c \text{ is } \epsilon, \\ pc : \langle i \rangle \rightarrow \ell \wedge c' \sim \ell & \text{if } s_c \text{ is } i; c', \end{cases}$$

and that their states are equal. The directive and observation transformers are the identity, hence we show that the machines are in lock-step.

Safety. The proof of preservation of SCT of array sharing and array concatenation requires the source program to be sequentially safe. Therefore, we must show that previous passes preserve this property to compose the proofs. It is a standard property, usually satisfied by correct compilers, and can be proved with established methods such as forward simulations for functional correctness. Hence, in our CoQ development, we include the proof of sequential safety preservation of register allocation as an illustration and assume it for the rest of the passes.

Composition. Composing the proof of preservation of each pass yields a proof of preservation of SCT for the whole compiler, assuming that the source program is sequentially safe. The proof of preservation of SCT imposes no restrictions on the order in which our passes are composed.

PROPOSITION 1 (COMPILER PRESERVATION OF SCT). *Let ϕ be a relation on inputs, P a source program, and Q its compilation. If P is ϕ -SCT and sequentially safe, Q is ϕ -SCT.*

We refer the reader to lemma `compose_sct`, file `security.v` in the CoQ development, for a proof of Proposition 1.

7 Type System

This section defines an information flow type system that enforces SCT with respect to our new semantics. The type system is taken from [9], but the notion of typable program puts an additional requirement on the initial type.

For simplicity of exposition, we present the type system in a simplified form. In particular, our presentation does not allow for label polymorphism and does not follow the constraint-based style used in type inference.

Type System. We now describe an information flow type system that ensures that observations do not depend on secrets, for any adversarial choice of directives—i.e., that the program is SCT. Typing judgments are of the form $\Sigma, \Gamma \vdash c : \Sigma', \Gamma'$, where Γ and Γ' are security environments that map registers and arrays to pairs of security levels taken from a 2-elements lattice $\{\text{Public}, \text{Secret}\}$ with the usual order $\text{Public} \leq \text{Secret}$. The components of these pairs respectively track the confidentiality level under sequential execution and under all possible speculative executions. By convention,

$$\begin{array}{c}
\frac{\Gamma \vdash e : \tau \quad x \notin \text{FV}(\Sigma)}{\Sigma, \Gamma \vdash x = e : \Sigma, \Gamma[x \leftarrow \tau]} \\
\frac{\Gamma \vdash e : \text{Public} \quad x \notin \text{FV}(\Sigma)}{\Sigma, \Gamma \vdash x = a[e] : \Sigma, \Gamma[x \leftarrow \langle \Gamma(a)_n, \text{Secret} \rangle]} \\
\frac{\Gamma \leq \Gamma' \quad \Gamma \vdash e : \text{Public} \quad \Gamma(x) \leq \Gamma'(a) \quad \forall b \neq a, \Gamma(x)_s \leq \Gamma'(b)_s}{\Sigma, \Gamma \vdash a[e] = x : \Sigma, \Gamma'} \\
\frac{\Gamma' = \{v : \langle \Gamma(v)_n, \Gamma(v)_n \rangle \mid \text{for each } v\}}{\Sigma, \Gamma \vdash \text{init_msf}() : \text{updated}, \Gamma'} \\
\frac{\text{outdated}(e), \Gamma \vdash \text{update_msf}(e) : \text{updated}, \Gamma \quad \tau = \langle \Gamma(x)_n, \Gamma(x)_n \rangle}{\text{updated}, \Gamma \vdash y = \text{protect}(x) : \text{updated}, \Gamma[y \leftarrow \tau]} \\
\frac{\Gamma \vdash e : \text{Public} \quad \Sigma|_e, \Gamma \vdash c_{\top} : \Sigma', \Gamma' \quad \Sigma|_e, \Gamma \vdash c_{\perp} : \Sigma', \Gamma'}{\Sigma, \Gamma \vdash \text{if } e \text{ then } c_{\top} \text{ else } c_{\perp} : \Sigma', \Gamma'} \\
\frac{\Gamma \vdash e : \text{Public} \quad \Sigma|_e, \Gamma \vdash c : \Sigma, \Gamma}{\Sigma, \Gamma \vdash \text{while } e \text{ do } c : \Sigma|_e, \Gamma} \\
\frac{}{\Sigma, \Gamma \vdash \epsilon : \Sigma, \Gamma} \\
\frac{\Sigma, \Gamma \vdash i : \Sigma_i, \Gamma_i \quad \Sigma_i, \Gamma_i \vdash c : \Sigma', \Gamma'}{\Sigma, \Gamma \vdash i; c : \Sigma', \Gamma'} \\
\frac{\Sigma_0, \Gamma_0 \vdash c : \Sigma'_0, \Gamma'_0 \quad \Sigma_0 \subseteq \Sigma \quad \Sigma' \subseteq \Sigma'_0 \quad \Gamma \leq \Gamma_0 \quad \Gamma'_0 \leq \Gamma'}{\Sigma, \Gamma \vdash c : \Sigma', \Gamma'}
\end{array}$$

$$\begin{array}{l}
\text{FV}(\Sigma) := \begin{cases} \text{FV}(e) & \text{if } \Sigma \text{ is outdated}(e) \\ \emptyset & \text{otherwise} \end{cases} \\
\Sigma \subseteq \Sigma' := \Sigma = \text{unknown} \vee \Sigma = \Sigma' \\
\Sigma|_e := \begin{cases} \text{outdated}(e) & \text{if } \Sigma \text{ is updated} \\ \text{unknown} & \text{otherwise} \end{cases} \\
\Gamma \leq \Gamma' := \forall x, \Gamma(x) \leq \Gamma'(x)
\end{array}$$

Fig. 7. Typing rules and auxiliary definitions.

we write Γ_n and Γ_s for the first and second component of Γ . Given that speculative execution encompasses sequential execution, we implicitly assume the invariant $\Gamma_n \leq \Gamma_s$. Moreover, Σ and Σ' are misspeculation flag (MSF) types taken from the following grammar:

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

Informally, MSF types capture partial information about the way the MSF tracks misspeculation. We briefly explain each type in turn. The MSF type `unknown` means that nothing is known about misspeculation. The MSF type `updated` means that the register `msf` correctly captures misspeculation status, i.e., `msf` is equal to `MASK` if execution is misspeculating and `NOMASK` otherwise. The type `outdated(e)` means that the register `msf` will correctly capture misspeculation status after being updated with the expression e .

Figure 7 presents the core typing rules. We refer to [9] for further explanations. In addition to these rules, the type system features specific rules for safe loads and stores. These rules relax the typing constraints of loads and stores, under the assumption that they are speculatively safe. Informally, a load or store is speculatively safe if for every execution, including speculative executions of the program, the index of the load or store remains within bounds. Under this assumption, one does not need to set the speculative type of the target register of a load to `Secret`. Similarly, one does not need to update the speculative type of all arrays when typing a speculatively safe store. It is possible to adapt these rules to our setting, by strengthening the notion of speculative safety to encompass correct initialization.

Soundness. For every program P and relation ϕ that determines secrets and public inputs, let $\Gamma_{P,\phi}$ be the security environment that maps to $\langle \text{Secret}, \text{Secret} \rangle$ every register and array that does not store a public input. A key observation is that, whenever $\Gamma_{P,\phi}$ is used as an initial type, the typing rules ensure that only variables and arrays initialized with public data can be public. This is a critical difference with the type system in [9], and suffices to prove that the type system ensures SCT.

PROPOSITION 2 (SOUNDNESS). *If unknown, $\Gamma_{P,\phi} \vdash P : \Sigma', \Gamma'$, for some Σ' and Γ' , then P is ϕ -SCT.*

We refer the reader to lemma `typedp_soundness`, file `typesystem_security.v` in the COQ development, for a mechanized proof of Proposition 2.

End-to-End Security. By composing Proposition 2 and Proposition 1, we obtain the following end-to-end theorem:

THEOREM 4 (END-TO-END SECURITY). *Let P be a source program, Q its compilation, and ϕ a relation on inputs. If P is sequentially safe and unknown, $\Gamma_{P,\phi} \vdash P : \Sigma', \Gamma'$, for some Σ' and Γ' , then Q is ϕ -SCT.*

We refer the reader to lemma `end_to_end`, file `compiler.v` in the COQ development, for a mechanized proof of Theorem 4.

8 Applications to Jasmin and Libjade

In this section, we explore the significance of our results for the Jasmin language. We discuss the impact of our findings on high-assurance cryptographic software written in Jasmin, including the libjade library.

The Jasmin Compiler. Table 3 lists the principal passes of the Jasmin compiler and briefly discusses preservation of speculative constant-time; whenever possible, we contrast the Jasmin pass with one of the compiler passes considered in this paper. The overall conclusion is that most passes are either similar to the ones considered in this work or clearly preserve speculative constant-time.

For the sake of clarity, this work focuses on a simple imperative language and compiler. Unfortunately, it remains a significant engineering endeavor to prove preservation of speculative constant-time for the Jasmin compiler. The first reason for this is that the proof of preservation of (sequential) constant-time [18] is a formalization (of over ten thousand lines according to *loc. cit.*) that uses complex objects, including structured leakage transformers. Moreover, this formalization has not been updated to the latest versions of the Jasmin compiler. We conjecture that a proof of preservation of SCT would be even larger and more complex. Second, the compiler and the proof of compiler correctness are under major refactoring, in particular, to switch from the current big-step semantics to an alternative semantics based on interaction trees [57].

Jasmin Implementations. The Jasmin framework has been used to write efficient and formally verified implementations of many cryptographic primitives, including the hash function SHA3 and the key encapsulation mechanism ML-KEM, formerly Kyber. Most of these implementations are typable with the type system of [9] and [12]. However, we have seen that this type system accepts Jasmin programs that compile into assembly that is not SCT. To avoid the problem, we have patched the type system to make additional requirements on the initial type, see Section 7. To understand the impact of this requirement, we have adapted the implementation of the Jasmin type system so that it enforces our additional constraints. Pleasingly, the implementations of libjade require no modifications, with the exception of ML-KEM. The type system rejects ML-KEM because the rejection sampling procedure uses local arrays, which must be initially considered as secret in our type system. We manually inspected the sources of failure and concluded that they correspond to reading public values in initialized arrays. The type system accepts the program after being

Table 3. Jasmin compiler passes and their effect on leakage (we refer to the passes presented in this work when applicable). We expect that all passes preserve SCT.

Jasmin pass	Transformers for directives and observations
Inlining	Add step directives and \bullet observations before and after function calls.
Function pruning	The identity (this pass removes unreachable parts of the program).
Constant propagation	Remove some force b directives and branch b observations (this pass is a combination of constant folding, propagation, and dead branch elimination from this work).
Dead code elimination	Remove some step directives and \bullet observations (the same as dead assignment elimination from this work).
Unrolling	The identity (this pass is repeated loop peeling).
Register array expansion	Source mem a i directives and addr a i observations become step and \bullet in the target when a is a register array.
Instruction selection	Add step directives and \bullet observations for complex assignments.
SLH instruction selection	The identity.
Inline propagation	The identity (this is constant propagation on inline variables only).
Stack allocation	Renaming of mem a i directives and addr a i observations (this is a combination of array reuse and partial array concatenation from this work).
Register allocation	The identity (the same as register allocation from this work).
Linearization	Add some step directives and \bullet observations (a combination of the linearization pass of this work and a pass that inserts intermediate direct unconditional jumps).
Stack zeroization	Add mem RSP i directives and addr RSP i observations, where i is constant.
Tunneling	Remove some step directives and \bullet observations.
Assembly generation	Add step directives and \bullet observations for complex operations.

instructed that these reads are legitimate, using the `declassify` annotation of the Jasmin language. As declassification has no computational content, the modifications have no effect on performance.

We need to manually instruct the type system that these loads are legitimate because, as discussed in Section 7, the type of uninitialized variables must be secret. The type checker implements some heuristics to improve on this, specifically for registers and stack variables, since once it finds a write to these, it can assume that they are initialized. For arrays, it is more complex; see, for instance the example of the listing on the right of Figure 3. In the case of ML-KEM, manual code inspection reveals that the array in question must be initialized before rejection sampling.

9 Related and Future Work

Enforcement of Speculative Constant-Time. There is a large body of work that develops automated techniques for detecting speculative leaks, using a variety of approaches, including type systems, symbolic execution, and fuzzing, and having a broad range of targets, including source programs,

intermediate representations, assembly, and binary code. We refer the reader to the survey [24] for further information and an overview of the tools up to 2022.

From the point of view of this paper, the most important distinction among these works is the security definition they target. In this work, we consider speculative constant-time, and require that programs do not leak through observations. An alternative is to require that speculative execution does not leak more than sequential execution. We refer to this policy as relative speculative constant-time (RSCT), although the terms relative non-interference [24] or speculative non-interference [33] are sometimes used in the literature. RSCT can be defined as an instance of SCT, by letting ϕ be the maximal relation between memories for which sequential executions of a given program coincide. It follows that a compiler pass preserves RSCT if and only if it preserves SCT for an arbitrary relation ϕ and reflects CT for an arbitrary relation ϕ . However, it is unclear that all transformations of interest reflect constant-time. In general, RSCT and all conditional policies (i.e., policies stated as implications) seem less amenable to preservation proofs.

Preservation of Constant-Time. Over the last few years, researchers have developed new techniques to prove that compilers preserve constant-time. The first proof of preservation of constant-time introduces CT-simulation, a technique that extends the usual commuting diagram of forward simulation from compiler correctness proofs into a commuting cube, that involves two related executions at source level and two corresponding executions at target level [17]. The technique, while general, is overly powerful for many common optimizations of interest. Therefore, subsequent works explore a simpler technique, which consists in proving that the leakage of a target program can be computed from the leakage of the corresponding source program—and potentially the public part of the initial state. This suffices to establish, using a simple argument, that the compiler preserves constant-time. This technique (in some variant forms) was used to verify formally that the CompCert and the Jasmin compilers, which constitute two prime examples of formally verified compilers, preserve constant-time [15, 18].

Preservation of Speculative Constant-Time. Patrignani and Guarnieri [44] were among the first to explore systematically the interactions between compilers and Spectre attacks. Similar to ours, their exploration is contrasted. On the negative side, they observe that existing compilers, including compilers that implement SLH, generate code that leaks speculatively. On the positive side, they show that with adequate care it is possible to generate code that is SCT. However, their work does not consider common compiler optimizations as done in this paper.

An alternative to preservation of SCT is to implement compiler-based mitigations that make programs SCT. Blade [52] is a prime example of compiler-based mitigation that has been used to protect cryptographic code written in WASM against Spectre v1. Informally, Blade identifies secrets as sources and leaking instructions as leaks, and implements an efficient algorithm for the Max-Flow/Min-Cut problem to achieve optimal placement of protections.

SNiP [51] is a closely related effort with some interesting differences. It is also a study on the preservation of SCT; it presents a different proof methodology and focuses in depth on two compiler passes. More specifically, it proposes a new approach to tackle the SCT cube directly. In contrast, and in line with [15, 18], our work proposes an equivalent (see Theorem 3) technique based on transformers for directives and observations; the motivation for our choice is that it simplifies proofs and endows compiler passes with a concise and abstract contract for speculation and leakage. SNiP delves into two compiler passes: dead code elimination (an extension of dead assignment elimination from Section 6) and register allocation (with spilling). As spilling does not preserve SCT, SNiP develops a dedicated static analysis to introduce selSLH mitigations and enable spilling. Complementarily, our work is more comprehensive, studying nine different passes and their transformers, implementing them, and mechanizing the framework and their proofs in Coq.

There are two less substantial differences related to the chosen setting. First, the object languages are similar, and both feature directives and observations. SNIp deals with explicit rollbacks after a fence, while we build on *secure forward consistency* [16] to simplify the language and proofs—we need only consider forward executions. Second, the non-interference property in SNIp is equivalent to the one in our work. However, the one in SNIp (called *SNi*) requires that indistinguishable inputs (called *initial states* there) are receptive to the same directives, in addition to the leakage being equal. We tackle this in the design of our semantics (see Lemma 2).

Secure Compilation. Secure compilation is a research area that focuses on the interactions between compilation and security. Since compilers are complex objects and security policies are more intricate than trace-based properties, many of the results in this area are formalized within a proof assistant. One main line of work aims to develop a framework to define, analyze and contrast rigorously different notions of secure compilation [1, 2, 43]. The focus of these works is definitional, and therefore their general aim is generality rather than on specific compilers or properties. Another main line of work focuses on specific compilers and specific properties, including various forms of safety and integrity [20, 27, 40, 41, 61], and resource usage [7, 21, 22, 32, 42, 56]. Other works approach secure compilation from a broader perspective, and in particular consider compiler security in a wider context [28, 49, 58].

Declassification. This work does not include declassification, even though libjade uses it to avoid unnecessary mitigations. There are two main purposes for declassification. On the one hand, it can ameliorate imprecision in the type system. We sometimes use declassification to overcome limitations such as homogeneous-level arrays—i.e., that all cells of an array have the same confidentiality—or incomplete analysis for array initialization. Refining our type system to address such imprecisions would free us from declassification in these cases.

On the other hand, cryptographic programs are often SCT only up to computational assumptions—e.g., leaking ciphertext. In such cases, declassification interfaces SCT verification and computational security proofs. Thus, this case is essentially different from the previous one: here, the aim is to guarantee SCT of the compiled program up to some computational assumptions about the adversary. We plan to study this in future work by showing that the leakage of assembly code compiled from well-typed programs depends only on directives, public data, and declassified values.

10 Conclusion

Our paper introduces a framework for proving preservation of speculative constant-time, and shows that it can be instantiated to many common compiler optimizations from the literature. Our overall conclusion is that preservation of speculative constant-time for a realistic compiler like Jasmin is a plausible target, provided sufficient care is taken to align the source-level notion of security with compiler passes that may introduce additional leakage. A natural direction for future work is to carry a more systematic exploration of source-level type systems, in order to achieve the best performance of generated code. Another direction for future work is to extend our results to probabilistic programs, which satisfy a probabilistic notion of speculative constant-time, and to programs which use declassification.

Acknowledgments

This research was supported by the *Deutsche Forschungsgemeinschaft* (DFG, German research Foundation) as part of the Excellence Strategy of the German Federal and State Governments – EXC 2092 CASA - 390781972; and by the *Agence Nationale de la Recherche* (French National Research Agency) as part of the France 2030 programme – ANR-22-PECY-0006.

References

- [1] Carmine Abate, Roberto Blanco, Ștefan Ciobăcă, Adrien Durier, Deepak Garg, Catalin Hrițcu, Marco Patrignani, Éric Tanter, and Jérémy Thibault. 2021. An Extended Account of Trace-relating Compiler Correctness and Secure Compilation. *ACM Trans. Program. Lang. Syst.* 43, 4 (2021), 14:1–14:48. <https://doi.org/10.1145/3460860>
- [2] Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hrițcu, Marco Patrignani, and Jérémy Thibault. 2019. Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25–28, 2019*. IEEE, 256–271. <https://doi.org/10.1109/CSF.2019.00025>
- [3] Martin R. Albrecht and Kenneth G. Paterson. 2016. Lucky Microseconds: A Timing Attack on Amazon’s s2n Implementation of TLS. In *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8–12, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9665)*, Marc Fischlin and Jean-Sébastien Coron (Eds.). Springer, 622–643. https://doi.org/10.1007/978-3-662-49890-3_24
- [4] Nadjem J. AlFardan and Kenneth G. Paterson. 2013. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19–22, 2013*. IEEE Computer Society, 526–540. <https://doi.org/10.1109/SP.2013.42>
- [5] José Baccelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-Assurance and High-Speed Cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 1807–1823. <https://doi.org/10.1145/3133956.3134078>
- [6] José Baccelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. 2020. The Last Mile: High-Assurance and High-Speed Cryptographic Implementations. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18–21, 2020*. IEEE, 965–982. <https://doi.org/10.1109/SP40000.2020.00028>
- [7] Roberto M. Amadio, Nicholas Ayache, François Bobot, Jaap Boender, Brian Campbell, Ilias Garnier, Antoine Madet, James McKinna, Dominic P. Mulligan, Mauro Piccolo, Randy Pollack, Yann Régis-Gianas, Claudio Sacerdoti Coen, Ian Stark, and Paolo Tranquilli. 2013. Certified Complexity (CerCo). In *Foundational and Practical Aspects of Resource Analysis—Third International Workshop, FOPARA 2013, Bertinoro, Italy, August 29–31, 2013, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 8552)*, Ugo Dal Lago and Ricardo Peña (Eds.). Springer, 1–18. https://doi.org/10.1007/978-3-319-12466-7_1
- [8] Basavesh Ammanaghatta Shivakumar, Jack Barnes, Gilles Barthe, Sunjay Cauligi, Chitchanok Chuengsatiansup, Daniel Genkin, Sioli O’Connell, Peter Schwabe, Rui Qi Sim, and Yuval Yarom. 2023. Spectre Declassified: Reading from the Right Place at the Wrong Time. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21–25, 2023*. IEEE, 1753–1770. <https://doi.org/10.1109/SP46215.2023.10179355>
- [9] Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Swarn Priya, Peter Schwabe, and Lucas Tabary-Maujean. 2023. Typing High-Speed Cryptography against Spectre v1. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21–25, 2023*. IEEE, 1094–1111. <https://doi.org/10.1109/SP46215.2023.10179418>
- [10] Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, and Swarn Priya. 2022. Enforcing Fine-grained Constant-time Policies. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7–11, 2022*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM, 83–96. <https://doi.org/10.1145/3548606.3560689>
- [11] Diego F. Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. 2020. LadderLeak: Breaking ECDSA with Less than One Bit of Nonce Leakage. In *CCS ’20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9–13, 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM, 225–242. <https://doi.org/10.1145/3372297.3417268>
- [12] Santiago Arranz Olmos, Gilles Barthe, Chitchanok Chuengsatiansup, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Peter Schwabe, Yuval Yarom, and Zhiyuan Zhang. 2024. Protecting cryptographic code against Spectre-RSB. Cryptology ePrint Archive, Paper 2024/1070. <https://eprint.iacr.org/2024/1070> <https://eprint.iacr.org/2024/1070>
- [13] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. 2021. SoK: Computer-Aided Cryptography. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24–27 May 2021*. IEEE, 777–795. <https://doi.org/10.1109/SP40001.2021.00008>
- [14] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. 2014. System-level Non-interference for Constant-time Cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3–7, 2014*, Gail-Joon Ahn, Moti Yung, and Ninghui Li (Eds.). ACM, 1267–1279. <https://doi.org/10.1145/2660267.2660283>

- [15] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. 2020. Formal verification of a constant-time preserving C compiler. *Proc. ACM Program. Lang.* 4, POPL (2020), 7:1–7:30. <https://doi.org/10.1145/3371075>
- [16] Gilles Barthe, Sunjay Cauligi, Benjamin Grégoire, Adrien Koutsos, Kevin Liao, Tiago Oliveira, Swarn Priya, Tamara Rezk, and Peter Schwabe. 2021. High-Assurance Cryptography in the Spectre Era. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 1884–1901. <https://doi.org/10.1109/SP40001.2021.00046>
- [17] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. 2018. Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time". In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*. IEEE Computer Society, 328–343. <https://doi.org/10.1109/CSF.2018.00031>
- [18] Gilles Barthe, Benjamin Grégoire, Vincent Laporte, and Swarn Priya. 2021. Structured Leakage and Applications to Cryptographic Constant-Time and Cost. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 462–476. <https://doi.org/10.1145/3460120.3484761>
- [19] Daniel J Bernstein. 2005. Cache-timing attacks on AES.
- [20] Frédéric Besson, Sandrine Blazy, Alexandre Dang, Thomas P. Jensen, and Pierre Wilke. 2019. Compiling Sandboxes: Formally Verified Software Fault Isolation. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11423)*, Luis Caires (Ed.). Springer, 499–524. https://doi.org/10.1007/978-3-030-17184-1_18
- [21] Sandrine Blazy, André Oliveira Maroneze, and David Pichardie. 2013. Formal Verification of Loop Bound Estimation for WCET Analysis. In *Verified Software: Theories, Tools, Experiments - 5th International Conference, VSTTE 2013, Menlo Park, CA, USA, May 17-19, 2013, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 8164)*, Ernie Cohen and Andrey Rybalchenko (Eds.). Springer, 281–303. https://doi.org/10.1007/978-3-642-54108-7_15
- [22] Quentin Carboneaux, Jan Hoffmann, Tahina Ramananandro, and Zhong Shao. 2014. End-to-end verification of stack-space bounds for C programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom—June 09—11, 2014*, Michael F. P. O’Boyle and Keshav Pingali (Eds.). ACM, 270–281. <https://doi.org/10.1145/2594291.2594301>
- [23] Chandler Carruth. 2018. RFC: Speculative Load Hardening (a Spectre variant #1 mitigation). <https://lists.lvm.org/pipermail/lvm-dev/2018-March/122085.html>.
- [24] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. 2022. SoK: Practical Foundations for Software Spectre Defenses. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 666–680. <https://doi.org/10.1109/SP46214.2022.9833707>
- [25] Sunjay Cauligi, Craig Disselkoen, Klaus von Gleissenthall, Dean M. Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2020. Constant-time foundations for the new spectre era. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 913–926. <https://doi.org/10.1145/3385412.3385970>
- [26] Boru Chen, Yingchen Wang, Pradyumna Shome, Christopher W. Fletcher, David Kohlbrenner, Riccardo Paccagnella, and Daniel Genkin. 2024. GoFetch: Breaking Constant-Time Cryptographic Implementations Using Data Memory-Dependent Prefetchers. In *USENIX Security*.
- [27] John Criswell, Nathan Dautenhahn, and Vikram S. Adve. 2014. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*. IEEE Computer Society, 292–307. <https://doi.org/10.1109/SP.2014.26>
- [28] Vijay D’Silva, Mathias Payer, and Dawn Xiaodong Song. 2015. The Correctness-Security Gap in Compiler Optimization. In *2015 IEEE Symposium on Security and Privacy Workshops, SPW 2015, San Jose, CA, USA, May 21-22, 2015*. IEEE Computer Society, 73–87. <https://doi.org/10.1109/SPW.2015.33>
- [29] Qian Ge, Yuval Yarom, David A. Cock, and Gernot Heiser. 2018. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptogr. Eng.* 8, 1 (2018), 1–27. <https://doi.org/10.1007/S13389-016-0141-6>
- [30] Antoine Geimer, Mathéo Vergnolle, Frédéric Recoules, Lesly-Ann Daniel, Sébastien Bardin, and Clémentine Maurice. 2023. A Systematic Evaluation of Automated Tools for Side-Channel Vulnerabilities Detection in Cryptographic Libraries. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda (Eds.). ACM, 1690–1704. <https://doi.org/10.1145/3576915.3623112>
- [31] Daniel Genkin, Luke Valenta, and Yuval Yarom. 2017. May the Fourth Be With You: A Microarchitectural Side Channel Attack on Several Real-World Applications of Curve25519. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 845–858. <https://doi.org/10.1145/3133956>

3134029

- [32] Alejandro Gómez-Londoño, Johannes Åman Pohjola, Hira Taqdees Syeda, Magnus O. Myreen, and Yong Kiam Tan. 2020. Do you have space for dessert? a verified space cost semantics for CakeML programs. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 204:1–204:29. <https://doi.org/10.1145/3428272>
- [33] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. 2020. Spectector: Principled Detection of Speculative Information Flows. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1–19. <https://doi.org/10.1109/SP40000.2020.00011>
- [34] Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. 2022. "They're not that hard to mitigate": What Cryptographic Library Developers Think About Timing Attacks. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 632–649. <https://doi.org/10.1109/SP46214.2022.9833713>
- [35] Emilia Käsper and Peter Schwabe. 2009. Faster and Timing-Attack Resistant AES-GCM. In *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings (Lecture Notes in Computer Science, Vol. 5747)*, Christophe Clavier and Kris Gaj (Eds.). Springer, 1–17. https://doi.org/10.1007/978-3-642-04138-9_1
- [36] Jason Kim, Stephan van Schaik, Daniel Genkin, and Yuval Yarom. 2023. iLeakage: Browser-based Timerless Speculative Execution Attacks on Apple Devices. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda (Eds.). ACM, 2038–2052. <https://doi.org/10.1145/3576915.3616611>
- [37] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 1–19. <https://doi.org/10.1109/SP.2019.00002>
- [38] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings (Lecture Notes in Computer Science, Vol. 1109)*, Neal Koblitz (Ed.). Springer, 104–113. https://doi.org/10.1007/3-540-68697-5_9
- [39] Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, J. Gregory Morrisett and Simon L. Peyton Jones (Eds.). ACM, 42–54. <https://doi.org/10.1145/1111037.1111042>
- [40] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. 2012. RockSalt: better, faster, stronger SFI for the x86. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 395–404. <https://doi.org/10.1145/2254064.2254111>
- [41] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2009. SoftBound: highly compatible and complete spatial memory safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 245–258. <https://doi.org/10.1145/1542476.1542504>
- [42] Zoe Paraskevopoulou and Andrew W. Appel. 2019. Closure conversion is safe for space. *Proc. ACM Program. Lang.* 3, ICFP (2019), 83:1–83:29. <https://doi.org/10.1145/3341687>
- [43] Marco Patrignani and Deepak Garg. 2017. Secure Compilation and Hyperproperty Preservation. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*. IEEE Computer Society, 392–404. <https://doi.org/10.1109/CSF.2017.13>
- [44] Marco Patrignani and Marco Guarnieri. 2021. Exorcising Spectres with Secure Compilers. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 445–461. <https://doi.org/10.1145/3460120.3484534>
- [45] Colin Percival. 2005. Cache missing for fun and profit.
- [46] Antoon Purnal. 2024. PQShield plugs timing leaks in Kyber / ML-KEM to improve PQC implementation maturity. <https://pqshield.com/pqshield-plugs-timing-leaks-in-kyber-ml-kem-to-improve-pqc-implementation-maturity/>
- [47] Eyal Ronen, Kenneth G. Paterson, and Adi Shamir. 2018. Pseudo Constant Time Implementations of TLS Are Only Pseudo Secure. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 1397–1414. <https://doi.org/10.1145/3243734.3243775>
- [48] Florian Sieck, Zhiyuan Zhang, Sebastian Berndt, Chitchanok Chuengsatiansup, Thomas Eisenbarth, and Yuval Yarom. 2024. TeeJam: Sub-Cache-Line Leakages Strike Back. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2024, 1 (2024), 457–500. <https://doi.org/10.46586/TCHES.V2024.I1.457-500>

- [49] Laurent Simon, David Chisnall, and Ross J. Anderson. 2018. What You Get is What You C: Controlling Side Effects in Mainstream C Compilers. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*. IEEE, 1–15. <https://doi.org/10.1109/EuroSP.2018.00009>
- [50] Eran Tromer, Dag Arne Osvik, and Adi Shamir. 2010. Efficient Cache Attacks on AES, and Countermeasures. *J. Cryptol.* 23, 1 (2010), 37–71. <https://doi.org/10.1007/S00145-009-9049-Y>
- [51] Sören van der Wall and Roland Meyer. 2025. SNIP: Speculative Execution and Non-Interference Preservation for Compiler Transformations. *Proc. ACM Program. Lang.* POPL (2025), To appear. Issue POPL.
- [52] Marco Vassena, Craig Disselkoe, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kici, Ranjit Jhala, Dean M. Tullsen, and Deian Stefan. 2021. Automatically eliminating speculative leaks from cryptographic code with blade. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–30. <https://doi.org/10.1145/3434330>
- [53] Jose Rodrigo Sanchez Vicarte, Michael Flanders, Riccardo Paccagnella, Grant Garrett-Grossman, Adam Morrison, Christopher W. Fletcher, and David Kohlbrenner. 2022. Augury: Using Data Memory-Dependent Prefetchers to Leak Data at Rest. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 1491–1505. <https://doi.org/10.1109/SP46214.2022.9833570>
- [54] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. 2022. Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 679–697. <https://www.usenix.org/conference/usenixsecurity22/presentation/wang-yingchen>
- [55] Yingchen Wang, Riccardo Paccagnella, Alan Wandke, Zhao Gang, Grant Garrett-Grossman, Christopher W. Fletcher, David Kohlbrenner, and Hovav Shacham. 2023. DVFS Frequently Leaks Secrets: Hertzbleed Attacks Beyond SIKE, Cryptography, and CPU-Only Data. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 2306–2320. <https://doi.org/10.1109/SP46215.2023.10179326>
- [56] Yuting Wang, Pierre Wilke, and Zhong Shao. 2019. An abstract stack based approach to verified compositional compilation to machine code. *Proc. ACM Program. Lang.* 3, POPL (2019), 62:1–62:30. <https://doi.org/10.1145/3290375>
- [57] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 51:1–51:32. <https://doi.org/10.1145/3371119>
- [58] Jianhao Xu, Kangjie Lu, Zhengjie Du, Zhu Ding, Linke Li, Qiushi Wu, Mathias Payer, and Bing Mao. 2023. Silent Bugs Matter: A Study of Compiler-Introduced Security Bugs. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association, 3655–3672. <https://www.usenix.org/conference/usenixsecurity23/presentation/xu-jianhao>
- [59] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2017. CacheBleed: a timing attack on OpenSSL constant-time RSA. *J. Cryptogr. Eng.* 7, 2 (2017), 99–112. <https://doi.org/10.1007/S13389-017-0152-Y>
- [60] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. 2023. Ultimate SLH: Taking Speculative Load Hardening to the Next Level. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association, 7125–7142. <https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-zhiyuan-slh>
- [61] Lu Zhao, Guodong Li, Bjorn De Sutter, and John Regehr. 2011. ARMor: fully verified software fault isolation. In *Proceedings of the 11th International Conference on Embedded Software, EMSOFT 2011, part of the Seventh Embedded Systems Week, ESWeek 2011, Taipei, Taiwan, October 9-14, 2011*, Samarjit Chakraborty, Ahmed Jerraya, Sanjoy K. Baruah, and Sebastian Fischmeister (Eds.). ACM, 289–298. <https://doi.org/10.1145/2038642.2038687>

Received 2024-07-11; accepted 2024-11-07