ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

ProSpeCT: Provably Secure Speculation for the Constant-Time Policy

Lesly-Ann Daniel[1], Marton Bognar[1], Job Noorman[1], Sébastien Bardin[2], Tamara Rezk[3] and Frank Piessens[1]

[1]imec-DistriNet, KU Leuven, 3001 Leuven, Belgium
[2]CEA, List, Université Paris Saclay, France
[3]INRIA, Université Côte d'Azur, Sophia Antipolis, France

Abstract

We propose ProSpeCT, a generic formal processor model providing provably secure speculation for the constant-time policy. For constant-time programs under a *non-speculative* semantics, ProSpeCT guarantees that speculative and out-of-order execution cause no microarchitectural leaks. This guarantee is achieved by tracking secrets in the processor pipeline and ensuring that they do not influence the microarchitectural state during speculative execution. Our formalization covers a broad class of speculation mechanisms, generalizing prior work. As a result, our security proof covers all known Spectre attacks, including load value injection (LVI) attacks.

In addition to the formal model, we provide a prototype hardware implementation of ProSpeCT on a RISC-V processor and show evidence of its low impact on hardware cost, performance, and required software changes. In particular, the experimental evaluation confirms our expectation that for a compliant constant-time binary, enabling ProSpeCT incurs no performance overhead.

1 Introduction

It is well-understood that microarchitectural optimization techniques commonly used in processors can lead to security vulnerabilities [102]. One of the most recent and challenging problems in this space is the family of Spectre attacks [21], which abuse speculative execution to leak secrets to an attacker that can observe parts of the microarchitectural state of the platform on which the victim is executing.

In response to the discovery of Spectre, a wide range of countermeasures has already been proposed [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 39]. It is an important and difficult challenge to understand the trade-offs offered by these mitigations in terms of security, performance, and applicability to legacy hardware or software.

On the one hand, software countermeasures targeting specific transient execution attacks can still leave other attacks unmitigated [18], and they must be patched every time new speculation mechanisms are introduced (e.g., the predictive store forwarding feature newly introduced in AMD Zen3 processors [19]). On the other hand, mainstream hardware mitigations have been recently shown ineffective [20] against Spectre-v2 (BTB) attacks [21].

Hardware-based secure speculation. In a recent paper, Guarnieri et al. [22] propose *hardware-software contracts* to compare hardware-based mechanisms for secure speculation and better understand how these defenses can enable software to provide end-to-end security guarantees. For instance, they show that certain types of hardware-level taint tracking [9, 13, 15] provide secure speculation for the *sandboxing* policy. On processors implementing one of these mechanisms, the software can simply enforce the sandboxing policy under a non-speculative semantics and does not need to consider the (error-prone and possibly expensive) placement of software speculation barriers.

However, none of the hardware defenses studied under the hardware-software contract framework enable secure speculation for the constant-time policy, except for completely disabling speculative execution. Hence, the classic cryptographic constant-time programming model [23] does not suffice to guarantee security on processors with these countermeasures, and significantly more complex and costly software programming models are required to recover security [18, 24, 25, 26, 27, 28, 29].

Problem statement. In this paper, we investigate how to provide efficient provably secure speculation for the constant-time policy under a wide range of speculation mechanisms. Specifically, we apply the hardware-software contract framework to another class of hardware taint-tracking mechanisms explicitly tracking *secrecy* of data in the microarchitecture (e.g., systems like ConTExT [30], SpectreGuard [31], or SPT [32]). In such systems, a constant-time program informs the processor about which memory cells contain secret data. Using this additional information, hardware-based taint-tracking can provide *stronger* security guarantees than sandboxing approaches [22]. Additionally, we consider a wide variety of speculation mechanisms, whereas the model of Guarnieri et al. considers only speculation on conditional branches.

Our proposal. The main contribution of this paper is ProSpeCT, a generic processor model formalizing the essence of such secrecy-tracking hardware mechanisms and a proof that it provides secure speculation for the constant-time policy. Specifically, off-the-shelf constant-time cryptographic libraries can be run securely on ProSpeCT without additional protections for transient execution attacks.

ProSpeCT is modular in the implementation of predictors and covers a broad class of speculation mechanisms, including branch prediction and store-to-load forwarding. As a novel aspect, ProSpeCT additionally covers new mechanisms like predictive store forwarding [19] and even mechanisms that are not (yet) implemented in commercial processors, such as load value prediction [33] or value prediction [34]. In particular, we rigorously show that ProSpeCT protects against Spectre-v2 (BTB) attacks [21], for which mainstream hardware mitigations have recently been shown ineffective [20]. As evidence for generality, we show that our mechanism even protects against Load Value Injection (LVI) attacks [43], which are particularly challenging to mitigate.

Another novel aspect of our formalization is the statement of our security condition, which allows a program to declassify a ciphertext while still requiring the processor to make sure that the attacker does not learn anything about the plaintext or the key used to compute the ciphertext.

To demonstrate the viability of our proposed mechanism, we extend a RISC-V processor to be ProSpeCT-compliant and quantify the hardware costs. Results show that the overhead of ProSpeCT in area usage and critical path is reasonable. We also demonstrate that the required software changes to cryptographic code are minimal and that the performance impact is negligible if secrets are precisely annotated. Our prototype is the first non-simulated hardware implementation of a speculative and out-of-order processor that implements secure speculation for the constant-time policy.

Contribution. In summary, our contributions are:

- We present ProSpeCT, the first formal processor model providing provably secure speculation for the constant-time policy (Section 4). We propose a formal model of a processor that tracks secrets during execution and temporarily blocks speculative execution if secrets could leak. The model is generic; it supports a wide range of speculation mechanisms and formalizes the guarantees provided by prior hardware-based secrecy tracking mechanisms [30, 31, 32].

- We formally prove that ProSpeCT provides secure speculation for the constant-time policy, i.e., programs that comply with the classic cryptographic constant-time discipline will not leak secrets through microarchitectural channels (Theorem 1), including in the presence of declassification (Theorem 2). The proof holds for a large variety of speculation mechanisms, encompassing all known Spectre and LVI attacks.

- We are the first to consider load value speculation. Interestingly, our formal analysis reveals that executions resulting from *correct* load value speculation must sometimes be rolled-back to avoid attacks based on *implicit resolution-based channels* [15]. We prove this formally (Theorem 1) and provide an example in Section 4.6.

- We provide the first non-simulated hardware implementation of a processor offering secure speculation. We implement ProSpeCT on a RISC-V processor supporting speculation (Section 6.1) and evaluate the costs of the proposed mechanism in terms of hardware, performance, and manual effort for precisely marking secret data (Section 6.2).

Availability. Our implementation and the experimental evaluation are open-sourced at `https://github.com/proteus-core/prospect`. A technical report containing the full formalization and proofs is available at [35].

2 Problem statement

2.1 Transient execution attacks

Modern processors rely on heavy optimizations to improve performance. They can execute instructions out-of-order to avoid stalling the pipeline when the operands of an instruction are not available. Additionally, they employ *speculation* mechanisms to predict the instruction stream. The execution of instructions resulting from a misprediction, called *transient execution*, is reverted at the architectural level, but effects on the microarchitectural state (e.g., the cache) are persistent.

Spectre attacks [21] exploit these speculation mechanisms to force a victim to leak secrets during transient execution. An attacker can mistrain predictors to force a victim into transiently executing a sequence of instructions, called a Spectre gadget, chosen to encode secrets in the microarchitectural state. Finally, the attacker can use microarchitectural attacks to extract the secret. To this day, many variants of Spectre attacks have been discovered, exploiting a wide variety of speculation mechanisms [19, 21, 36, 37, 38, 39, 40].

Transient execution may also arise from incorrect data being forwarded by faulting instructions. For instance, on some processors, the result of unauthorized loads is transiently forwarded to subsequent instructions before the load is rolled back. This mechanism has first been exploited in Meltdown-style attacks [41, 42] to exfiltrate secret data from another security domain. It is generally accepted that Meltdown-style attacks should be mitigated in hardware by preventing such forwarding from faulting loads. We consider Meltdown-style attacks out of scope for this paper.

However, these faulting loads have also been exploited to *inject* incorrect data into the victim's transient execution, and, similarly to Spectre attacks, lead the victim to leak their secrets into the microarchitectural state. In particular, these so-called load value injection (LVI) attacks [43] are still possible in the presence of Meltdown mitigations zeroing out the results of faulting loads at the silicon level (i.e., LVI-NULL). LVI attacks are related to Spectre attacks that would exploit *value speculation* during loads.

We illustrate variants of Spectre and LVI attacks in Listing 1, where programs in Listings 1c to 1e abuse different sources of transient execution (👻) to encode `SecretVal` in the cache using the **`leak`** function in Listing 1b. After encoding, the attacker can extract the secret from the cache using cache attacks. Note that while we illustrate these attacks using a cache side-channel, transient execution vulnerabilities are independent of the microarchitectural side-channel they exploit, such as branch predictor state [44], SIMD units [45], port contention [46, 47], micro-op cache [48], etc. Consequently, the **`leak`**`(x)` function can be replaced with any other function that reveals information on the value of `x` via a timing or microarchitectural side-channel.

The **Spectre-PHT** (Pattern History Table) or Spectre-v1 variant [21] exploits the conditional branch predictor to transiently execute the wrong side of a conditional branch. For instance, in Listing 1c, an attacker can first mistrain the conditional branch predictor to take the branch and then call the piece of code with `idx = 16` to make the victim transiently execute the branch, accessing `SecretVal` at line 5 and encoding it to the microarchitectural state at line 6.

The **Spectre-BTB** (Branch Target Buffer) or Spectre-v2 variant [21] exploits indirect branch prediction to transiently redirect the control flow to an attacker-chosen location. For example, the program in Listing 1d calls a trusted function, which performs secure computations using `SecretVal`. An attacker can mistrain the branch predictor such that, after line 9, the victim transiently jumps to the **`leak`** function instead of the trusted function and leaks `SecretVal`. The **Spectre-RSB** (Return Stack Buffer) variant [36, 37] is similar to Spectre-BTB but exploits target predictions for `ret` instructions.

The **Spectre-STL** (Store-To-Load-forwarding) or Spectre-v4 variant [38] exploits the fact that load instructions can speculatively bypass preceding stores. In Listing 1e, the secret located at `ptr_s` is overwritten at line 10, followed by a **`load`** to the same address, which should return `0`. With Spectre-STL, the **`load`** may bypass the **`store`** at line 10 and transiently load `SecretVal`, which would then be leaked to the microarchitectural state at line 12.

Finally, **LVI** (Load Value Injection) attacks [43] exploit a faulting **`load`** to directly inject incorrect data into the victim's execution. For instance, in Listing 1f, an attacker can prepare the microarchitectural state so that the value `16` is forwarded to `idx` by the **`load`** instruction at line 13, hence accessing `SecretVal` at line 14 and leaking it at line 15.

2.2 Secure speculation approaches

Since transient execution attacks were discovered, several studies have focused on adapting program semantics, security policies, and verification tools to take into account the *speculative semantics* of programs and place extra software-level protections against Spectre attacks, e.g., [18, 26, 27, 28, 29, 49, 50, 51, 52, 53, 54, 55, 103]. However, reasoning about transient execution attacks at the software level only can be burdensome and fragile. Firstly, it necessitates knowledge of microarchitectural details that are often not publicly available. Secondly, it requires changing security policies and applying software patches every time new speculation mechanisms are introduced (e.g., the predictive store forwarding feature newly introduced in AMD Zen3 processors [19]). Finally, software countermeasures targeting specific transient execution attacks can still leave the door open to other attacks [18].

Instead, we argue that, for a given policy P, enforcement mechanisms at the software level should only consider an architectural (non-speculative) semantics, while the hardware should guarantee that transient execution does not introduce additional vulnerabilities. We call this approach *hardware-based secure speculation for P*.

Hardware-based secure speculation for sandboxing. A sandboxing policy isolates a potentially malicious application by restricting the memory range it can access. A program is said to be *sandboxed* if it never accesses memory outside its authorized address range. Sandboxed programs are vulnerable to Spectre attacks, as out-of-bounds memory locations may still be accessed transiently and have their contents leaked to the microarchitectural state. As an example, the program in Listing 1c is sandboxed but can still access and leak out-of-bounds data when the condition is misspeculated.

Some hardware taint-tracking mechanisms [9, 13, 15] have been shown to enable secure speculation for sandboxing [22]. For instance, Speculative Taint Tracking (STT) [15] taints speculatively accessed data and prevents tainted values from being forwarded to instructions that may form a covert channel. In Listing 1c, STT taints the variable `x` at line 5 until the condition at line 4 is resolved. As `x` is tainted, its value is not forwarded to the insecure **`load`** in the **`leak`** function.

Unfortunately, hardware-based secure speculation for sandboxing *only protects speculatively accessed data*, meaning that secret data loaded in registers during sequential execution may still be transiently leaked. For instance, STT does not protect the program in Listing 1d against Spectre-BTB. At line 5, a secret is loaded during sequential execution. As a result, `x` is not tainted by STT, and its value can still be forwarded to an insecure instruction if the **`jmp`** is misspeculated. Hardware-based secure speculation for sandboxing is therefore insufficient to guarantee security for programs that compute on secrets, such as cryptographic primitives. To protect these programs, we need to enable hardware-based secure speculation for the constant-time policy.

```
    0 - 15: A[16]
ptr_s (16): SecretVal
17 - 16400: B[256 * 64]
```

(a) Memory

```
void leak(x):
  idx ← x * 64
  y ← load B + idx ☠
```

(b) Encode x into the cache.

```
if (idx < size_A) 👻
  x ← load A + idx
  leak(x)
```

(c) Spectre-PHT (v1)

```
f ← trusted_func
x ← load ptr_s
jmp f(x) 👻
```

(d) Spectre-BTB (v2)

```
store ptr_s 0
x ← load ptr_s 👻
leak(x)
```

(e) Spectre-STL (v4)

```
idx ← load trusted_idx 👻
x ← load A + idx
leak(x)
```

(f) LVI

Listing 1: Examples of code snippets vulnerable to transient execution attacks. The memory layout given in Listing 1a where SecretVal is the only secret input and ptr_s = 16 is common to Listings 1c to 1f. 👻 indicates instructions triggering transient executions and ☠ indicates a leakage.

Hardware-based secure speculation for constant-time. A constant-time policy specifies that program secrets should not leak through timing or microarchitectural side channels. Before the advent of transient execution attacks, the constant-time policy was enforced with a coding discipline ensuring that the *control-flow of the program*, *addresses of memory accesses*, and *operands of variable-time instructions* do not depend on secret data. This coding discipline is the de facto standard for writing cryptographic code; it has been adopted in many cryptographic libraries [56, 57, 58, 59] and is supported by many tools, e.g., [23, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69].

A standard definition for constant-time programs (i.e., programs adhering to the constant-time policy), and the one we use in this paper, is the following:

Definition 1 (Constant-time program)**.** A program is constant-time if the observation trace that it produces during *sequential execution* is independent of secret data (where the observation trace records the control flow and memory accesses).[1]

Unfortunately, adhering to this definition is insufficient to guarantee security on modern processors vulnerable to transient execution attacks like Spectre or LVI. Indeed, all programs in Listing 1 are constant-time according to Definition 1, but they are vulnerable to transient execution attacks.

Hardware-based countermeasures guaranteeing secure speculation for sandboxing do not guarantee secure speculation for the constant-time policy. Therefore, to enforce the constant-time policy on speculative processors, it is still necessary to insert specific protections (typically `fence` instructions or retpolines [2]) to protect against transient execution attacks. Software developers still have to reason about speculation when they want to enforce the constant-time policy. In this paper, we address the problem of *providing hardware-based provably secure speculation for the constant-time policy*.

3 Informal overview of ProSpeCT

In this section, we motivate our design choices, make explicit what guarantees have to be enforced by software, and sketch the requirements the hardware must enforce. Finally, we illustrate how ProSpeCT protects the programs in Listing 1.

3.1 Design choices

ProSpeCT relies on a hardware-software co-design where developers annotate their secret data, and the hardware guarantees that no information about these secrets can leak during transient execution. The design of ProSpeCT is motivated by two main objectives. The first objective is to support existing constant-time code with minimal software changes. To this end, we base our annotation and declassification mechanism on ConTExT [30] in which developers partition the memory into public and secret regions and can declassify secrets by writing them to public memory. The second objective is to support secure code while maintaining full performance benefits of speculative and out-of-order execution. Specifically, ProSpeCT delays speculative execution only when a secret is about to be leaked; hence *in constant-time programs* (which do not leak secrets) ProSpeCT *only blocks mispredicted instructions*.

Software contracts. Software developers must comply with three contracts:

Contract 1. Secret memory locations are labeled.

For instance, in Listing 1, address `16` is labeled as *secret* (or *high*), denoted `H`, whereas other addresses are labeled as *public* (or *low*), denoted `L`.

Contract 2. The program is constant-time.

Contract 3. Secret values written to public memory are *intentionally declassified* by the program.

Contract 3 allows, for instance, cryptographic code to declassify ciphertexts. However, software developers must make sure to not unintentionally declassify secrets by writing them to public memory.

[1] We give a formal definition in Section 4.5. For simplicity, we do not include variable-time instructions in our security proofs but discuss how to handle them with ProSpeCT.

We prove in Section 4.5 that if programs comply with these three contracts, then execution on ProSpeCT does not leak secrets through timing and microarchitectural side channels.

Hardware requirements. On the hardware side, ProSpeCT must realize the following:

Requirement 1. During the execution of a program, the processor tracks *security levels*. Concretely, it labels values loaded from memory with their corresponding security level (`L` or `H`) and soundly propagates these security levels during computations.

Requirement 2. The processor prevents values with security level `H` to be leaked during speculative execution. Hardware developers identify *insecure instructions* that may leak data through (1) changing the microarchitectural state, (2) influencing the program counter, or (3) exhibiting operand-dependent timing. The processor prevents these instructions from being speculatively executed with secret operands.

Requirement 3. Predictions do not leak secret data, in particular: (1) predictor states are only updated using public values, and (2) speculations are rolled back (even the *correct* ones) when their outcome depends on secrets (otherwise, it would leak whether the public prediction is equal to the secret value).

3.2 ProSpeCT through illustrative examples

Consider the program in Listing 1c, assuming that `idx = 16` and the condition is misspeculated to *true*. When executing the **`load`** instruction at line 5, ProSpeCT tags the register `x` with the security level corresponding to address `16`, denoted $\texttt{x} \mapsto (\texttt{SecretVal}{:}\texttt{H})$ (by Req. 1).[2] Then, when the **`leak`** function is executed, the **`load`** instruction (line 3, Listing 1b) is blocked because it would leak a secret-labeled value during speculative execution (by Req. 2). Conversely, if register `x` contains a public-labeled value, i.e., $\texttt{x} \mapsto (\texttt{v}{:}\texttt{L})$, the **`load`** instruction is not blocked. ProSpeCT only blocks speculative execution in a few restricted cases, namely when secret data is about to be leaked.

Notice that, contrary to sandboxing-based approaches, ProSpeCT also protects secrets loaded in architectural registers from being transiently leaked. For example, in Listing 1d, when the secret is loaded at line 8, `x` is labeled with `H`, which prevents the secret from being transiently leaked later (by Req. 2) if the **`jmp`** instruction at line 9 transiently jumps to the **`leak`** function.

So far, we have seen examples of ProSpeCT applied to Spectre-PHT and Spectre-BTB (Spectre-RSB is similar to the latter). The protection generalizes to any other source of speculation, such as load value prediction (which encompasses LVI and Spectre-STL). Take, for instance, the program in Listing 1f. Here, the source of speculation is the **`load`** instruction, which transiently forwards an incorrect value at line 13. Until the **`load`** is resolved, ProSpeCT considers the following instructions speculative. Consequently, (by Req. 2) it does not forward the secrets to the **`load`** in the **`leak`** function (Listing 1b, line 3) and prevents the LVI attack.

Finally, ProSpeCT also guarantees (by Req. 3) that predicted values do not depend on secrets. In particular, secret values cannot be speculatively forwarded to other instructions. For example, in Listing 1e, the **`load`** instruction at line 11 cannot speculatively load `SecretVal`, because the corresponding address is labeled as secret. Notice that ProSpeCT still allows forwarding public values.

4 Formalization and theorems

This section presents one of the core contributions of this paper, the formalization of ProSpeCT. The ProSpeCT semantics builds on prior semantics [22], extended to consider a broader spectrum of prediction mechanisms. Moreover, it generalizes the standard constant-time leakage model; in addition to disclosing control-flow and memory accesses, our semantics also discloses *all public-labeled data*. Concretely, all public-labeled data can influence predictions and is observable by an attacker.

4.1 ISA language

The ISA is modeled using a small assembly language called μAsm [27], described in Fig. 1. $\mathcal{V}$ is a set of values, including memory addresses and program locations, and we let `v` and `l` range over $\mathcal{V}$. `pc` denotes the program counter register, and `r`, `x` denote registers with $\texttt{x} \neq \texttt{pc}$. A program P is a partial mapping from locations to instructions. We use $P[\texttt{l}]$ to denote the instruction at location `l`.

$$\begin{array}{lrl} \text{(Expressions)} & e ::= & \texttt{v} \mid \texttt{r} \mid e_1 \otimes e_2 \\ \text{(Instructions)} & inst ::= & \texttt{x} \leftarrow e \mid \textbf{jmp}\ e \mid \textbf{beqz}\ e\ \texttt{l} \mid \\ & & \texttt{x} \leftarrow \textbf{load}\ e \mid \textbf{store}\ e_a\ e_v \end{array}$$

Figure 1: Syntax of μAsm programs where $\otimes$ denotes a binary operation.

Security levels. We assume a lattice Λ with two security levels: public (low, `L`) and secret (high, `H`). We let $\texttt{s}, \texttt{s}', \texttt{s}_0, \ldots$ range over security levels from Λ. $\sqcup$ denotes the least upper bound operation on the lattice, with $\texttt{L} \sqcup \texttt{H} = \texttt{H}$. Additionally, we let $(\texttt{v}{:}\texttt{s})$ denote a value `v` with security level `s`, ranging over the set $\hat{\mathcal{V}} = \mathcal{V} \times \Lambda$. For simplicity, we restrict our description to this 2-level security lattice, but this work generalizes to arbitrary security lattices.

[2] A more conservative design choice, adopted by ConTExT [30], would be to prevent such speculative loads from accessing secret memory locations and to prevent the execution of line 5. However, we formally show that secure speculation is possilbe with this more liberal design choice.

Location of secret data. As stated in Contract 1, programmers annotate secret data in the code. In our formal semantics, we assume that they do so by specifying a *security memory partition* s_m, which maps memory addresses to security levels in Λ. We assume this mapping to be fixed, it cannot change over time. Hence, for the sake of readability, we do not explicitly include it in the configurations of the semantics rules.

4.2 Hardware configurations

Hardware configurations are of the form $\langle m, r, buf, \mu \rangle$, where m is the memory, which maps addresses to values in $\mathcal{V}$; r is the register map, which maps registers to pairs of a value and a security level in $\hat{\mathcal{V}}$; buf is the reorder buffer, which is a sequence of (possibly transient) instructions; and μ is the *microarchitectural context*.

Microarchitectural context. The microarchitectural context μ can be thought of as the part of the microarchitectural state that the attacker controls. It is an abstract component that models both the *observations* that the attacker can make and the *influence* the attacker has on predictions and scheduling. Formally, it is a stateful deterministic component offering three functions:

- *update*, called by the semantics whenever microarchitectural state possibly leaks to the attacker;
- *predict*, giving the attacker control over predictions, for instance, to predict jump targets or load values;
- *next*, giving the attacker control over scheduling decisions that determine the next processor step to execute.

Hence, our definition of the semantics must make sure that for each computation step, μ is updated with all information that could leak from that computation. In particular, updates should include the program counter and the addresses of memory accesses (which *directly* influence the instruction and data cache), but also operands of variable-time instructions (which do not directly influence the microarchitectural state, but might do so *indirectly* via timing variations [70]).

Importantly, to satisfy Req. 2 and 3, our evaluation rules must satisfy the following invariant: if the program is constant-time (as stated in Contract 2), secret data should never leak to μ. In particular, the *update* function should never be given secret data as input during speculative execution. It follows from our security theorems that this is indeed the case.

Reorder buffer. In out-of-order processors, program instructions are fetched in order and placed in a *reorder buffer* (ROB) where they can be executed out-of-order. Contrary to ISA instructions, ROB instructions, defined in Fig. 2, keep track of the security levels of values. In addition, **jmp** and **beqz** are directly translated to pc assignments when they are fetched and thus are not part of ROB instructions. This also implies that, contrary to ISA instructions, assignments in the ROB can target pc. Finally, instructions in the ROB with predicted values are tagged with the address l of the instruction that the prediction resulted from; otherwise, they are tagged with ε.

$$\begin{aligned}
&\text{(Tags)} & T &::= \mathtt{l} \mid \varepsilon \\
&\text{(ROB exp)} & e &::= (\mathtt{v{:}s}) \mid \mathtt{r} \mid e_1 \otimes e_2 \\
&\text{(ROB inst)} & i &::= \mathtt{r} \leftarrow e@T \mid \mathtt{x} \leftarrow \mathbf{load}\ e@T \mid \\
& & &\qquad \mathbf{store}\ e_a\ e_v@T
\end{aligned}$$

Figure 2: ROB instructions.

During the execution, changes that occur in the ROB are applied to the registers using the function *apl* [22]. When the value of an assignment in the reorder buffer is not resolved yet, the corresponding register is mapped to a special symbol $\bot$, meaning that it is undefined. Thus, the function *apl* generates a new register map where the value of some registers is undefined.

Definition 2 (Apply function *apl*)**.** For all register maps r and reorder buffers buf:

$$\begin{aligned}
apl(\varepsilon, r) &= r \\
apl(\mathtt{r} \leftarrow (\mathtt{v{:}s})@T \cdot buf, r) &= apl(buf, r[\mathtt{r} \mapsto (\mathtt{v{:}s})]) \\
apl(\mathtt{r} \leftarrow e@T \cdot buf, r) &= apl(buf, r[\mathtt{r} \mapsto \bot]) \text{ if } e \notin \hat{\mathcal{V}} \\
apl(\mathtt{x} \leftarrow \mathbf{load}\ e@T \cdot buf, r) &= apl(buf, r[\mathtt{x} \mapsto \bot]) \\
apl(\ \mathbf{store}\ e_a\ e_v@T \cdot buf, r) &= apl(buf, r)
\end{aligned}$$

4.3 Sanitization of secret values

An important feature of ProSpeCT is the ability to *sanitize* secret values during speculative execution and predictions. To achieve sanitization, we define a low-projection for values denoted $(\mathtt{v{:}s})|_{\mathtt{L}}$. It discloses public values but replaces secret values with $\bot$.

Definition 3 (Low-projection)**.**

$$(\mathtt{v{:}L})|_{\mathtt{L}} = (\mathtt{v{:}L}) \qquad (\mathtt{v{:}H})|_{\mathtt{L}} = \bot \qquad \bot|_{\mathtt{L}} = \bot$$

We let $r|_{\mathtt{L}}$ be the point-wise extension of $\cdot|_{\mathtt{L}}$ to register maps. Hence, a sanitized register map $r|_{\mathtt{L}}$ maps registers to either their value when the associated security level is public or to $\bot$ when the value is unresolved, or when it is secret.

Definition 4 (Low memory projection)**.** We define the low-projection of a memory $m|_{\mathtt{L}}$ such that for all addresses a:

$$m|_{\mathtt{L}}(\mathtt{a}) = \begin{cases} m(\mathtt{a}) & \text{if } s_m(\mathtt{a}) = \mathtt{L} \\ \bot & \text{otherwise} \end{cases}$$

The low-projection of a reorder buffer buf, denoted $\lfloor buf \rfloor_{\mathtt{L}}$, discloses all low values in buf. Values with security level H are replaced by $\bot$. On the other hand, values with security level L, unresolved expressions, and tags are not replaced. Details are deferred to the technical report [35].

Sanitizing secrets in speculative execution. We define a function *aplsan* that selectively sanitizes the register map returned by *apl*. In sequential execution (i.e., when no instruction in buf results from a prediction), it directly returns the result of *apl*. During speculative execution (i.e., where at least one instruction in buf results from a prediction), it returns the low-projection of *apl*, in which secrets are replaced with $\bot$.

Definition 5 (Apply function $aplsan(\cdot,\cdot)$)**.**

$$aplsan(buf,r) = \begin{cases} apl(buf,r) & \text{if } \forall inst@T \in buf.\ T = \varepsilon \\ apl(buf,r)|_{\mathtt{L}} & \text{if } \exists inst@T \in buf.\ T \neq \varepsilon \end{cases}$$

Concretely, the function *aplsan* is a crucial part of Req. 2; it acts as a filter that prevents forwarding secret data to insecure instructions (which *update* the microarchitectural context) during speculative execution.

4.4 Evaluation rule

Expression evaluation. The evaluation of an expression e with a register map r, denoted $[\![e]\!]_r$, is a partial function from expressions to labeled values in $\hat{\mathcal{V}}$. It is undefined if one of the sub-expressions is undefined. Importantly, the evaluation of a binary operation propagates the security level of its operands in a conservative way (cf. Req. 1); if at least one of the operands has security level $\mathtt{H}$, then the resulting security level is $\mathtt{H}$. Details are available in the technical report [35].

Instruction evaluation. The hardware semantics is given by a main relation $c_1 \rightarrow c_2$ and an auxiliary relation $c_1' \xrightarrow[\mathsf{d}]{} c_2'$ where c_1 and c_2 are hardware configurations and d is a directive. Leaks are highlighted in the rules and _ is used to denote that there exists an expression, but this expression is not important in the context.

The directive determines which processor step to execute: the fetch directive fetches an instruction and places it at the end of the ROB, $\mathsf{execute}$ i executes the i^{th} instruction in the ROB, retire removes the oldest instruction from the ROB and commits its changes to the register map and memory.

The STEP rule selects the next directive to apply from the microarchitectural context using the function $next(\mu)$ and updates the hardware configuration accordingly:

STEP

$$\frac{\begin{array}{c} \mu' \triangleq update(m|_{\mathtt{L}}, r|_{\mathtt{L}}, \lfloor buf \rfloor_{\mathtt{L}}, \mu) \\ \mathsf{d} \triangleq next(\mu') \qquad \langle m, r, buf, \mu' \rangle \xrightarrow[\mathsf{d}]{} \langle m', r', buf', \mu'' \rangle \end{array}}{\langle m, r, buf, \mu \rangle \rightarrow \langle m', r', buf', \mu'' \rangle}$$

The rule updates the microarchitectural context using public-labeled values from the memory $m|_{\mathtt{L}}$, the register map $r|_{\mathtt{L}}$, and the reorder buffer $\lfloor buf \rfloor_{\mathtt{L}}$. This means that *any public-labeled value can influence subsequent predictions*. In Section 5, we show how this abstraction captures existing prediction mechanisms. It also means that any public-labeled value is leaked to the attacker, which effectively leaks *more* than the standard constant-time leakage model corresponding to Definition 1.

We present some evaluation rules for each directive, the full set of rules is available in our technical report [35].

Fetch directive. The instructions are fetched in program order and placed in the ROB. We provide here the rule FETCH-PREDICT-BRANCH-JMP, which applies when the instruction to fetch is a branch or jump.

FETCH-PREDICT-BRANCH-JMP

$$\frac{\begin{array}{c} (\mathtt{l}{:}\_) \triangleq [\![\mathtt{pc}]\!]_{apl(buf,r)} \\ P[\mathtt{l}] \in \{\, \mathbf{\texttt{beqz}}\ e\ \_,\ \mathbf{\texttt{jmp}}\ e \} \qquad \mathtt{l}' \triangleq predict(\mu) \end{array}}{\langle m, r, buf, \mu \rangle \xrightarrow[\mathsf{fetch}]{} \langle m, r, buf \cdot \mathtt{pc} \leftarrow (\mathtt{l}'{:}\mathtt{L})@\mathtt{l}, \mu \rangle}$$

The rule predicts the next program location $\mathtt{l}'$ and updates $\mathtt{pc}$ in the ROB accordingly.[3] Notice that the next location $\mathtt{l}'$ is added to the ROB with security level $\mathtt{L}$, hence it will be leaked to the microarchitectural context in the next STEP rule. In the same way, the current location $\mathtt{l}$, added as a speculation tag in the ROB, is also leaked.

Execute assignments. The rule EXECUTE-ASSIGN evaluates an expression e and updates the value of a register $\mathtt{r}$ in the ROB accordingly. The rule EXECUTE-ASSIGN assumes that the evaluation of the expression e does not leak information about its operands—in particular, the execution time of the instruction is independent of the value of its operands. In this case, the instruction can securely execute using secret data. Therefore, the rule uses the non-sanitized register map $apl(buf,r)$ to evaluate e (cf. the boxed hypothesis). For *variable-time (insecure) instructions* [71, 72, 73], we can simply replace the function $apl(buf,r)$ in the box with the function $aplsan(buf,r)$ to prevent the instruction from executing using secret data during speculative execution (cf. Req. 2).

EXECUTE-ASSIGN

$$\frac{\begin{array}{c} |buf| = i-1 \qquad e \notin \hat{\mathcal{V}} \qquad inst = \mathtt{r} \leftarrow e@T \\ \boxed{(\mathtt{v}{:}\mathtt{s}) \triangleq [\![e]\!]_{apl(buf,r)}} \qquad inst' \triangleq \mathtt{r} \leftarrow (\mathtt{v}{:}\mathtt{s})@T \end{array}}{\langle m, r, buf \cdot inst \cdot buf', \mu \rangle \xrightarrow[\mathsf{execute}\ i]{} \langle m, r, buf \cdot inst' \cdot buf', \mu \rangle}$$

Execute loads. The rule EXECUTE-LOAD-PREDICT predicts the value of a $\mathbf{\texttt{load}}$ instruction and updates the ROB with this predicted value. Note that the predicted value $\mathtt{v}$ is added to the ROB with security level $\mathtt{L}$ and is hence observable. This is consistent with the fact that predictions can only depend on public data (cf. Req. 3).

EXECUTE-LOAD-PREDICT

$$\frac{\begin{array}{c} |buf| = i-1 \\ inst = \mathtt{x} \leftarrow \mathbf{\texttt{load}}\ e@T \qquad (\mathtt{l}{:}\_) \triangleq [\![\mathtt{pc}]\!]_{apl(buf,r)} \\ \mathtt{v} \triangleq predict(\mu) \qquad inst' \triangleq \mathtt{x} \leftarrow (\mathtt{v}{:}\mathtt{L})@\mathtt{l} \end{array}}{\langle m, r, buf \cdot inst \cdot buf', \mu \rangle \xrightarrow[\mathsf{execute}\ i]{} \langle m, r, buf \cdot inst' \cdot buf', \mu \rangle}$$

[3] Note that what we call prediction here also covers store-to-load forwarding, load value prediction, and load value injection, as discussed in Section 5.

The rule EXECUTE-LOAD-COMMIT commits the result of a predicted **load** instruction to the ROB when the prediction is correct. It also leaks the address of the load to the microarchitectural context. The rule uses the *sanitized* value $\mathtt{a}$ of the address, which effectively prevents the rule to be applied during speculative execution when the address is secret (Req. 2). Additionally, notice that the rule can only be applied if the address $\mathtt{a}$ corresponds to public memory (i.e., $s_m(\mathtt{a}) = \mathtt{L}$). If the address $\mathtt{a}$ maps to secret memory (meaning that $m(\mathtt{a})$ is secret), a rollback is performed to prevent leaking whether the secret value $m(\mathtt{a})$ is equal to the predicted value $\mathtt{v}$, as described in Req. 3 (a detailed example is provided in Section 4.6).

EXECUTE-LOAD-COMMIT

$$\frac{\begin{array}{c} |buf| = i-1 \qquad inst = \mathtt{x} \leftarrow (\mathtt{v}{:}\_)@\mathtt{l_0} \\ P[\mathtt{l_0}] = \mathtt{x} \leftarrow \mathbf{load}\, e \qquad \mathbf{store}\ \_\ \_ \notin buf \\ (\mathtt{a}{:}\_) \triangleq [\![e]\!]_{aplsan(buf,r)} \qquad m(\mathtt{a}) = \mathtt{v} \qquad s_m(\mathtt{a}) = \mathtt{L} \\ inst' = \mathtt{x} \leftarrow (\mathtt{v}{:}s_m(\mathtt{a}))@\varepsilon \qquad \mu' \triangleq update(\mu, \mathtt{a}) \end{array}}{\langle m, r, buf \cdot inst \cdot buf', \mu \rangle \xrightarrow[\text{execute } i]{} \langle m, r, buf \cdot inst' \cdot buf', \mu' \rangle}$$

The complementary rule EXECUTE-LOAD-ROLLBACK is applied when the prediction is incorrect or when $m(\mathtt{a})$ is secret. It commits the correct value to the ROB and drops younger instructions from buf' (excluding the corresponding $\mathtt{pc}$ update).

Retire directives. Rules RETIRE-STORE-LOW and RETIRE-STORE-HIGH retire a store instruction on top of the ROB and update the memory accordingly. They also leak the address of the store to the microarchitectural context, following the constant-time leakage model. The rule RETIRE-STORE-LOW is evaluated if the address of the store corresponds to public memory (i.e., $s_m(\mathtt{a}) = \mathtt{L}$). In this case, regardless of its original security level, the value $\mathtt{v}$ becomes visible to the attacker: it is *declassified*. As stated in Contract 3, it is the responsibility of the developer to make sure that such declassification is intentional. In Section 4.6, we illustrate declassification in ProSpeCT with an example. The declassified value is recorded above the evaluation relation, denoted $\xrightarrow{\mathtt{v}}$. It does not affect the semantics but will be used in the theorems of Section 4.5. For secret store locations, the rule RETIRE-STORE-HIGH is applied, which produces an empty declassification trace.

RETIRE-STORE-LOW

$$\frac{\begin{array}{c} buf = \ \mathbf{store}\ (\mathtt{a}{:}\_)\ (\mathtt{v}{:}\_)@\varepsilon \cdot buf' \\ \mu' = update(\mu, \mathtt{a}) \qquad s_m(\mathtt{a}) = \mathtt{L} \end{array}}{\langle m, r, buf, \mu \rangle \xrightarrow[\text{retire}]{\mathtt{v}} \langle m[\mathtt{a} \mapsto \mathtt{v}], r, buf', \mu' \rangle}$$

RETIRE-STORE-HIGH

$$\frac{\begin{array}{c} buf = \ \mathbf{store}\ (\mathtt{a}{:}\_)\ (\mathtt{v}{:}\_)@\varepsilon \cdot buf' \\ \mu' = update(\mu, \mathtt{a}) \qquad s_m(\mathtt{a}) = \mathtt{H} \end{array}}{\langle m, r, buf, \mu \rangle \xrightarrow[\text{retire}]{\varepsilon} \langle m[\mathtt{a} \mapsto \mathtt{v}], r, buf', \mu' \rangle}$$

4.5 Theorems

We first define a security theorem for ProSpeCT that can be applied to constant-time programs without declassification, and then extend it to capture declassification. The full proofs are available in our technical report [35].

An architectural configuration $\alpha = \langle m, r \rangle$ is the subset of a hardware configuration, consisting of the memory and the register map. The (sequential) architectural semantics is given as a relation $\alpha \overset{\delta}{\underset{o}{\rightsquigarrow}} \alpha'$, which evaluates an architectural configuration α to another architectural configuration α'. It produces a sequence of observations o that contains the control flow (changes to the program counter) and the addresses of memory accesses. It also produces a declassification trace δ, which is the sequence of all values written to public memory. Evaluation rules are otherwise standard and are provided in the technical report [35].

Architectural configurations are said to be *low-equivalent*, written $\alpha|_{\mathtt{L}} = \alpha'|_{\mathtt{L}}$, if they are identical in the low-projections of their register maps and memories.

We let $c_0 \xrightarrow{\delta}{}^{n} c_n$ denote an n-step execution from a hardware configuration c_0 to a configuration c_n, which produces a *declassification trace* δ . When δ is not needed in the context, it is omitted. Similarly, we let $\alpha_0 \overset{\delta}{\underset{o}{\rightsquigarrow}}{}^{n} \alpha_n$ denote an n-step execution in the architectural semantics.

Security for constant-time programs. ProSpeCT guarantees that if a program is constant-time (Contract 2) and does not declassify secret data (Contracts 1 and 3), then it does not leak secret data when running on ProSpeCT.

The following definition formalizes the constant-time and no-declassification policy that we expect to be enforced on the software side. In that respect, it establishes a *hardware-software security contract* [22].

Definition 6 (Constant-time program)**.** A program is constant-time if for any initial architectural configurations α_0 and α_0' such that $\alpha_0|_{\mathtt{L}} = \alpha_0'|_{\mathtt{L}}$, and number of steps n:

$$\alpha_0 \overset{\delta}{\underset{o}{\rightsquigarrow}}{}^{n} \alpha_n \implies \alpha_0' \overset{\delta'}{\underset{o'}{\rightsquigarrow}}{}^{n} \alpha_n' \land o = o'$$

Additionally, if $\delta = \delta'$ for all possible α_0 and α_0', we say that the program *does not declassify secret data*.

The goal of the attacker is to distinguish two low-equivalent initial configurations c and c', by providing a *strategy*, i.e., a concrete initial microarchitectural context μ_0 and implementations for *predict*, *update*, and *next* (designed by the attacker to distinguish the configurations). Under any such *deterministic* strategy, the attacker should have the same observations when running in c and c'.[4]

Hypothesis 1. The functions *predict*, *update*, and *next* are deterministic.

[4] Note that nondeterministic strategies would produce different observations due to nondeterminism, not due to differences in secrets being leaked.

Under this hypothesis, the hardware semantics is deterministic.

The following theorem establishes end-to-end security for constant-time programs without declassification, running on ProSpecT.

Theorem 1 (Security for constant-time programs.)**.** *For any constant-time program that does not declassify secret data, microarchitectural state μ_0, initial configurations $c_0 = \langle m_0, r_0, \varepsilon, \mu_0 \rangle$ and $c'_0 = \langle m'_0, r'_0, \varepsilon, \mu_0 \rangle$ such that $\langle m_0, r_0 \rangle|_{\mathtt{L}} = \langle m'_0, r'_0 \rangle|_{\mathtt{L}}$ and number of steps n,*

$$c_0 \rightarrow^n c_n \implies c'_0 \rightarrow^n c'_n \ \wedge\ \mu_n = \mu'_n$$

where μ_n and μ'_n are the microarchitectural contexts in configurations c_n and c'_n, respectively.

Security with declassification. It is common for cryptographic programs to declassify ciphertexts after an encryption primitive. As we show in the technical report [35], classic definitions of declassification [74, 75, 76, 77] allow a program to declassify more information than expected in the context of cryptographic code. Indeed, with such definitions, declassifying `f(m)` implicitly declassifies `m` when `f` is an injective function (e.g., a cryptographic permutation). In contrast, we propose a novel definition of security *up to* declassification that captures the following intuition: if a program only declassifies ciphertexts, then plaintexts and keys remain indistinguishable to an attacker (because they are *cryptographically* indistinguishable) and should not be leaked.

As described above, the declassification trace of an execution $c \xrightarrow{\delta}{}^n c'$ is the sequence of all values stored to low (public) memory by the rule RETIRE-STORE-LOW. To express security up to declassification, we introduce a notion of *patched execution*, denoted $(c, \delta) \hookrightarrow (c', \delta')$, which replaces values stored to low-memory by values from a declassification trace δ (usually obtained by a low-equivalent run in the standard semantics $\rightarrow$). The patched execution ensures that the low-memories of two low-equivalent executions remain equal, achieved by patching the second execution with the declassification trace of the first execution. For instance, a ciphertext that is declassified in the standard semantics can be used to patch another (low-equivalent) execution, to obtain two executions with the same declassified ciphertext (and to make sure that they leak the same values). Concretely, the patched execution only differs from the standard execution by the rule RETIRE-STORE-LOW, which is replaced by the following:

$$\textsc{retire-store-patched}\quad \frac{\begin{array}{c} buf = \ \mathtt{\textbf{store}}\ (\mathtt{a}{:}\_)\ (\mathtt{v}{:}\_)@\varepsilon \cdot buf' \\ \mu' = update(\mu, \mathtt{a}) \qquad \delta = \mathtt{v}' \cdot \delta' \qquad s_m(\mathtt{a}) = \mathtt{L} \end{array}}{(\langle m, r, buf, \mu \rangle, \delta) \xhookrightarrow[\text{retire}]{} (\langle m[\mathtt{a} \mapsto \mathtt{v}'], r, buf', \mu' \rangle, \delta')}$$

We use $(c, \delta) \hookrightarrow^n (c', \delta')$ to denote the evaluation of n steps in the patched execution. Similarly, we define a patched execution for the architectural semantics denoted $(\alpha, \delta) \underset{o}{\rightsquigarrow} (\alpha', \delta')$ and provide the evaluation rules in the technical report [35].

Definition 7 (Constant-time up to declassification)**.** A program is constant-time up to declassification if for any pair of initial architectural configurations α_0 and α'_0 such that $\alpha_0|_{\mathtt{L}} = \alpha'_0|_{\mathtt{L}}$, and number of steps n:

$$\alpha_0 \overset{\delta}{\underset{o}{\rightsquigarrow}}{}^n \alpha_n \implies (\alpha'_0, \delta) \underset{o'}{\rightsquigarrow}{}^n (\alpha'_n, \varepsilon) \ \wedge\ o = o'$$

The following theorem establishes end-to-end security for constant-time programs up to declassification running on ProSpecT.

Theorem 2. *For any constant-time program up to declassification, microarchitectural state μ_0, initial configurations $c_0 = \langle m_0, r_0, \varepsilon, \mu_0 \rangle$ and $c'_0 = \langle m'_0, r'_0, \varepsilon, \mu_0 \rangle$ such that $\langle m_0, r_0 \rangle|_{\mathtt{L}} = \langle m'_0, r'_0 \rangle|_{\mathtt{L}}$ and number of steps n,*

$$c_0 \xrightarrow{\delta}{}^n c_n \implies (c'_0, \delta) \hookrightarrow^n (c'_n, \varepsilon) \ \wedge\ \mu_n = \mu'_n$$

where μ_n and μ'_n are the microarchitectural contexts in configurations c_n and c'_n, respectively.

In the next section, we provide an example to demonstrate how patched execution works.

4.6 Examples

This section showcases key aspects of ProSpecT's semantics through small examples. Example 1 illustrates how declassification works and how ProSpecT prevents forwarding secrets to potential side channels during speculative execution. Example 2 demonstrates that when a predicted load value is resolved and the actual value is secret, speculative execution must be rolled back, even if the prediction was correct.

Example 1 (Declassification)**.** Consider an execution of the program in Listing 2 (in the standard hardware semantics $\rightarrow$), where the register `s` evaluates to $(\mathtt{s_1}{:}\mathtt{H})$ in the initial configuration. Moreover, we assume that all conditions are predicted to be *true*, but only $\mathtt{c_1}$ can architecturally evaluate to *true*. Under this hypothesis, the program is constant-time up to declassification.

```
1  store a_L f(s)        // Declassify f(s)
2  d ← load a_L // Load declassified value
3  if c1 { x ← load d }        // Allowed
4  if c2 { x ← load s }        // Blocked
5  if c3 { x ← load f(s) }      // Blocked
```

Listing 2: Illustration of declassification where `s` is a secret input, `f` is a one-way function, and $\mathtt{a_L}$ is an address to a public memory location ($s_m(\mathtt{a_L}) = \mathtt{L}$). For readability, **beqz** instructions are replaced with **if** constructs.

At line 1, the program computes $\mathtt{f(s_1)}$ and declassifies the result by storing it to public memory. By the rule RETIRE-STORE-LOW, it also produces a declassification trace $\mathtt{f(s_1)}$.

At line 2, the program loads the declassified value in register `d`. By the rule EXECUTE-LOAD-COMMIT, `d` has security level $s_m(\texttt{a}_\texttt{L}) = \texttt{L}$, hence ProSpecT can speculatively execute the **load** on line 3 to speed up computations before $\texttt{c}_1$ is resolved. While it speculatively leaks $\texttt{d}$ to the cache, it is not a security concern because the value has been intentionally declassified (cf. Contract 3).

Notice that because $\texttt{f}$ is a one-way function, declassifying $\texttt{f(s}_1\texttt{)}$ does not reveal information about $\texttt{s}_1$. In particular, the **load** on line 4 should certainly not be allowed to execute speculatively. ProSpecT faithfully enforces this policy; the rule EXECUTE-LOAD-COMMIT uses *aplsan* to compute the address of the load, and because this happens during speculative execution and $\texttt{s}$ is labeled secret, we get $[\![\texttt{s}]\!]_{aplsan(buf,r)} = \bot$ (cf. Definitions 3 and 5). Hence, the **load** cannot be executed. Similarly, the **load** on line 5 is also blocked because the value $\texttt{f(s)}$ is recomputed and inherits the secret label from $\texttt{s}$.

Now, to illustrate Theorem 2 (security up to declassification), consider a second execution of the program in the *patched semantics* starting with a configuration low-equivalent to the previous one, but where $\texttt{s}$ evaluates to $(\texttt{s}_2{:}\texttt{H})$ and where declassified values are patched with the declassification trace of the first execution, i.e., $\texttt{f(s}_1\texttt{)}$. According to Theorem 2, the leakage of the first execution and this second patched execution should be the same.

At line 1, the execution evaluates the rule RETIRE-STORE-LOW-PATCHED, which stores the value $\texttt{f(s}_1\texttt{)}$ at address $\texttt{a}_\texttt{L}$ (instead of storing $\texttt{f(s}_2\texttt{)}$).

At line 2, the value $\texttt{f(s}_1\texttt{)}$ is loaded into the register $\texttt{d}$ and is assigned security level $s_m(\texttt{a}_\texttt{L}) = \texttt{L}$.

At line 3 (cf. rule EXECUTE-LOAD-COMMIT), the value of $\texttt{d}$ can be forwarded to the **load** because its security level is $\texttt{L}$. Notice that because the second execution has been patched with the declassification trace of the first execution, the update to the microarchitectural context (i.e., the leakage) is the same in both executions.

At line 4, similarly to the first execution, ProSpecT does not forward the value of $\texttt{s}$ to the **load** because its security level is $\texttt{H}$. By contrast, if we would consider an insecure microarchitecture that forwards the value, the microarchitectural context would be updated with $\texttt{s}_1$ in the first execution and $\texttt{s}_2$ in the second execution, which would violate Theorem 2.

Perhaps less intuitively, the leakage at line 5 would also be considered insecure w.r.t. Theorem 2, even though it is semantically equivalent to the leakage at line 3. Indeed, the leakage is not intentional w.r.t. Contract 3, which our security definition takes into account.

In summary, using the notion of *patched execution* allows us to express that there are no microarchitectural leaks beyond public information and explicitly declassified values. Even if some other secrets in the program are information-theoretically derivable from declassified values (but, for instance, are cryptographically protected against such derivation), they should still be considered secret. In particular, our declassification condition guarantees that any attack exposing program secrets has to do so based on public information and explicitly declassified values and hence does not rely on any Spectre attack. We believe that for a cryptographic primitive, our definition of declassification can be composed with a standard notion of cryptographic indistinguishability to obtain a stronger notion of cryptographic indistinguishability for the execution of the primitive on ProSpecT.

Example 2 (Rolling back a correct prediction)**.** Consider the following program, where $\texttt{a}_\texttt{H}$ is an address to a secret memory location ($s_m(\texttt{a}_\texttt{H}) = \texttt{H}$):

```
x ← load aH // Load secret value
y ← x + 4
```

We illustrate step-by-step a (possible) execution flow, focusing on the evolution of buf and highlighting changes at each step.

- Consider that the scheduler first fetches all instructions: $buf = \texttt{x} \leftarrow \textbf{load}\ a_\texttt{H}@\varepsilon \cdot \texttt{pc} \leftarrow (2{:}\texttt{L})@\varepsilon \cdot \texttt{y} \leftarrow \texttt{x}+4@\varepsilon$
- The scheduler then applies the rule EXECUTE-LOAD-PREDICT and the predictor predicts the loaded value to be 0. Notice that the prediction is public, so we assume that the attacker knows (and can even influence) its value: $buf = \texttt{x} \leftarrow 0@1 \cdot \texttt{pc} \leftarrow (2{:}\texttt{L})@\varepsilon \cdot \texttt{y} \leftarrow \texttt{x}+4@\varepsilon$
- Next, the scheduler applies the rule EXECUTE-ASSIGN, which computes $\texttt{y} \leftarrow \texttt{x}+4$: $buf = \texttt{x} \leftarrow 0@1 \cdot \texttt{pc} \leftarrow (2{:}\texttt{L})@\varepsilon \cdot \texttt{y} \leftarrow 4@\varepsilon$
- When resolving the prediction, because we have $s_m(\texttt{a}_\texttt{H}) = \texttt{H}$, the rule EXECUTE-LOAD-COMMIT cannot be applied, and the execution is rolled back, even if the predicted value (0) was correct: $buf_{rollback} = \texttt{x} \leftarrow 0@\varepsilon \cdot \texttt{pc} \leftarrow (2{:}\texttt{L})@\varepsilon$

Importantly, unconditionally rolling back the execution does not leak information about the secret value $m(a_\texttt{H})$, whereas allowing EXECUTE-LOAD-COMMIT to proceed would leak whether $m(a_\texttt{H}) = 0$. Indeed, if $m(a_\texttt{H}) \neq 0$, then a rollback would happen, and the final ROB would be $buf_{rollback}$. If $m(a_\texttt{H}) = 0$, the final ROB would be $buf_{commit} = \texttt{x} \leftarrow 0@\varepsilon \cdot \texttt{pc} \leftarrow (2{:}\texttt{L})@\varepsilon \cdot \texttt{y} \leftarrow 4@\varepsilon$.

Concretely, such a conditional rollback introduces a so-called *implicit resolution-based channel* [15]: the rollback case and the commit case lead to distinct timing behaviors (indeed, contrary to the commit case, the rollback case has to recompute the instructions following the load, which takes extra cycles). While prior solutions [15, 32] address implicit resolution-based channels (e.g., from memory disambiguation) by delaying the squashing of the implicit branch until prior speculations are resolved, this solution does not apply to load value prediction. Indeed, in our example, the implicit branch is already non-speculative (the processor knows for sure that the value can be committed).

5 Discussion

This section discusses prediction mechanisms supported by ProSpeCT, limitations, and compatibility with legacy code.

Prediction mechanisms. Prior work [15, 78] already stressed the importance of making predictions a function of public data. The novelty of ProSpeCT is to allow predictions to depend on *any public data*, hence generalizing standard models of speculative execution. Indeed, public values are used in the STEP rule to update the microarchitectural context μ, which is always given as an argument of *predict*. We now discuss how this model encompasses known prediction strategies.

Because the program counter is always public and therefore part of μ (cf. the technical report [35]), *predict*(μ) always has access to the current (and past) values of the program counter and corresponding instructions. Hence, it can make control-flow predictions based on the full control-flow history, which encompasses existing prediction strategies for conditional branches [79], indirect branches [80], and return targets.

Speculation on memory disambiguation, related to memory-disambiguation machine clears [40] (i.e., Speculative-store-bypass [38] and (predictive) store-to-load-forwarding [19]), is enabled by the rule EXECUTE-LOAD-PREDICT. However, ProSpeCT forwards only *public values* from the memory and microarchitectural buffers (the store buffer in particular). This restriction could be lifted by propagating security levels in the store buffer and forwarding predicted values with their security level, as done by SPT [32]. We leave this optimization for future work.

Via the rule EXECUTE-LOAD-PREDICT, our semantics also encompasses the more futuristic load value prediction [33], which can be implemented as forwarding a simple constant or forwarding a value based on the *public* history of load operations. Value prediction [34] (also related to floating-point machine clears [40]), which we do not formalize here for simplicity, is similar to load value prediction. In particular, the prediction should not depend on secrets, and the execution must always be rolled back if the actual value is secret.

Finally, *predict* might also return any arbitrary value, which accounts for predictor states that have been poisoned by an attacker or that forward dummy values, hence encompassing Spectre, as well as LVI (and LVI-NULL) attacks.

Limitations of ProSpeCT. Memory-ordering machine clears [40] are another source of transient execution, sometimes requiring rolling back memory operations to preserve memory consistency for concurrent programs. Even though our semantics does not support concurrency, this kind of mechanism could be supported by tracking whether memory instructions might be rolled back w.r.t. to some memory consistency model, similar to speculations. For instance, to support the total-store-ordering model (TSO), the rule EXECUTE-LOAD-COMMIT should additionally make sure that all prior **`load`** operations in the ROB are resolved (e.g., with an additional hypothesis $\_ \leftarrow \texttt{load}\ \_ \notin \mathit{buf}$).

Self-modifying code machine clears [40] can also cause transient execution in self-modifying code when the instruction cache (queried in the fetch stage) and the data cache (modified by prior **`store`** instructions) are desynchronized. Because our semantics assumes instruction memory to be fixed, it does not apply to self-modifying code.

Legacy software compatibility. ProSpeCT is fully compatible with legacy software. Code without secret annotations works on ProSpeCT as is, but without additional security over the base processor. Security and performance can also be traded off: the entire memory (or the stack) can be marked as secret, but it will likely result in additional performance overhead. Finally, to achieve security and optimal performance, only secret-handling code needs to be annotated. For instance, an annotated cryptosystem could be linked securely with (memory-safe) legacy code if the legacy code architecturally accesses only public or declassified information.

6 Implementation and evaluation

6.1 Implementation

To better understand the costs and benefits of ProSpeCT, we built a prototype hardware implementation using the Proteus RISC-V processor framework.[5] Proteus is implemented in SpinalHDL [81], a Scala-based hardware description language (HDL). SpinalHDL generates Verilog or VHDL code that can be run in a simulator or synthesized for an FPGA.

From the many existing open-source RISC-V CPU implementations,[6] we selected Proteus because it is designed to be extended with new hardware mechanisms via a plugin system (inspired by VexRiscv [82]). It is easily configurable in the number of ROB entries and execution units, and it supports branch target prediction and speculative execution, making it vulnerable to Spectre-PHT, -BTB, and -RSB attacks.

ProSpeCT is implemented as a Proteus plugin, with some additional modifications in the base processor. In future work, we are looking into combining ProSpeCT with other security extensions on Proteus. Our implementation is open-sourced at https://github.com/proteus-core/prospect.

Simplifications. Our prototype adopts memory partitioning: secrets are co-located in one or more secret memory regions, and we manually inform the hardware of the region boundaries. While it is possible to hardcode the boundaries of arbitrary secret regions in hardware, we implemented a more flexible approach that enables the configuration of secret region boundaries via control and status registers (CSRs). The number of CSRs can affect the hardware costs of the implementation; we report on setups allowing one and two secret regions. Note that in our benchmarks, we could co-locate

[5] https://github.com/proteus-core/proteus

[6] https://github.com/riscv/riscv-isa-manual/blob/master/marchid.md

all secrets in a single region when the stack is public and in two regions when the stack is secret. In future work, we plan to develop compiler support for co-locating all secrets in a single region and automatically setting up the secret region boundaries through CSRs.

The prototype takes a conservative approach and stalls every speculative instruction operating on secret data, not only *insecure instructions* (cf. Req. 2). Finally, interrupts are not supported.

Code size. The full implementation of a ProSpeCT-enabled processor consists of 5275 lines of SpinalHDL code, which generates approximately 104,000 lines of Verilog code. The ProSpeCT plugin is written in 90 lines of SpinalHDL, and approximately 270 additional lines of the base Proteus code were modified. For our evaluation, we use 5 execution units and a reorder buffer of size 16. To encourage further experimentation with different configurations, we open-source our evaluation setup.

6.2 Evaluation

The security benefit of ProSpeCT comes with a tradeoff in three different aspects:

1. *Hardware cost*: ProSpeCT uses additional hardware to track secret data and restrict its propagation. This can have an impact on hardware cost metrics like area used and critical path.
2. *Runtime overhead*: ProSpeCT can delay the forwarding of secret data, which might impact the execution time of applications.
3. *Labeling of secrets*: Software needs to declare what data is secret, possibly requiring additional developer effort to avoid unintentional declassification (cf. Contract 3).

To validate the security claims of the implementation [83], we executed code samples vulnerable against Spectre on both the base Proteus and the ProSpeCT-extended implementation and did not observe leakage in the latter case.

Hardware cost. To assess the overhead of the area used and the critical path, we synthesized Proteus without and with the ProSpeCT modifications for an `Artix-7 XC7A35T` FPGA with a speed grade of -1 using Xilinx Vivado. Our experiments showed a reasonable overhead for ProSpeCT. The version supporting a single secret region increases the number of slice LUTs from 16,847 to 19,728 (+17%) and slice registers from 11,913 to 12,600 (+6%). The critical path increases from 30.1 ns to 30.7 ns (+2%). We did not observe any additional increase in these numbers when adding a second set of CSRs to support a second secret region.

Runtime overhead. The runtime overhead of ProSpeCT depends on two main factors. First, the amount of data marked as secret: if a program only accesses public data, ProSpeCT incurs no overhead, while if the whole memory is marked as secret, the overhead is maximal. Second, the performance benefit of speculative execution on the program: if the performance of a program heavily benefits from speculative execution, the overhead of ProSpeCT will be higher than if the program does not benefit from speculative execution.

Table 1: Relative SpectreGuard benchmark performance on ProSpeCT.

Setting	25**S**/75**C**	50**S**/50**C**	75**S**/25**C**	90**S**/10**C**
baseline	100%	100%	100%	100%
P(`key`)	100%	100%	100%	100%
P(`all`)	110%	125%	136%	145%

Making a binary ProSpeCT-compliant requires co-locating secrets in memory, which could impact performance, e.g. via caching effects. We leave an evaluation of these secondary effects for future work.

Configuring the secret regions from software using CSRs only requires a few additional instructions (loading the boundary addresses into the CSRs before starting the program), resulting in negligible overhead.

We evaluate the first two main factors using the synthetic benchmarks from SpectreGuard [31]. These benchmarks simulate a mix of computations on public data (whose performance heavily benefits from speculative execution) and an encryption routine (whose performance benefits less from speculative execution) with different fractions of speculation/crypto (i.e., **S**/**C**). We modify the benchmarks in two ways: (1) because we specifically target constant-time code, we replace the non-constant-time `AES` primitive with the constant-time `chacha20` primitive from HACL* [57], (2) we annotate not only the key and plaintext as secret but also all variables that may contain secrets to avoid unintentional declassification (cf. Contract 3).

We ran the benchmarks in three different configurations: the base processor with no ProSpeCT extension (our baseline), precisely defining the secret region and defining a secret region to cover the entire address space. More precisely, for the second configuration, `P(key)`, we co-locate all secrets in a single region and load the boundaries of this secret region into the CSRs. In the third configuration, `P(all)`, we load the first and last address of the memory into the CSRs, protecting the entire address space. Results are given in Table 1, where the percentages denote the relative execution time compared to the baseline for each configuration.

In line with our expectations, our results show that for a ProSpeCT-compliant binary, enabling the defense incurs no runtime overhead when secret values are only accessed in constant-time code. When marking the whole memory secret, the overhead ranges from 10% to 45%, which is comparable (but lower) to the overhead of SpectreGuard [31] when enabled for the entire address space.[7] Overall, we conclude that ProSpeCT incurs a low overhead when secret data is precisely annotated, especially for programs where only a restricted part of the code computes on secrets, which is a common scenario [30, 31] (in SSH clients, web servers, etc.).

Labeling of secrets. To benefit from the security guarantees provided by our security theorems, code must be verified to be constant-time in the sequential execution model. Fortunately, verified constant-time implementations of cryptographic primitives are readily available [57], and such verified code also makes explicit what program data should be marked as secret. However, it is still not trivial to identify which memory addresses should be marked as secret. For instance, if the compiler spills secret arguments on the stack, that stack memory must also be marked as secret to avoid unintentional declassification and comply with Contract 3. While it is secure to conservatively over-approximate the memory areas marked as secret (and, for instance, always mark the stack as secret), this has a performance cost.

Hence, we evaluate how difficult it is to obtain precise information about which memory addresses should be labeled secret for a set of representative constant-time cryptographic primitives given in Table 2. To do so, we manually annotate (in the C code) all local variables that may contain secret data, to place them in a dedicated memory section. Additionally, we patch the generated assembly code to clear secret values from registers after declassification. The number of required annotations and assembly lines is reported in Table 2. As a sanity check, we validate that secret data is not written by the compiler to public memory locations outside of the dedicated declassification memory.[8] In the worst case, the time required to annotate secret variables and to validate that no secrets are written on the stack was less than 1 hour.

Table 2: Cryptographic primitives used for the experimental evaluation, reporting the lines of C code (LoC), whether the stack (S) is labeled public (L) or secret (H), the number of annotations manually (A_m), and automatically (A_a) inserted for marking variables, and the number of assembly instructions (I) manually inserted.

	LoC	S	A_m	A_a	I	Description
`djbsort` [84]	246	L	3	0	6	Constant-time sort
`sha256` [57]	1795	L	34	0	6	Hash function
`chacha20` [57]	1864	L	51	0	6	Encryption
`curve25519` [57]	3026	H	9	67	0	Elliptic curve

Interestingly, in 3 of the 4 cryptographic primitives, secret registers are not spilled on the stack by the primitive itself but by the surrounding code. Therefore, for these examples, it is possible to isolate secrets from public data and *keep the stack public* to minimize performance impact.

Finally, for the more complex primitive `curve25519`, secret register spilling cannot easily be avoided manually. Hence, we label the stack as secret, and instead of annotating secret variables, we annotate *public* variables to place them out of the stack and limit the performance impact. Notice that because the program is constant-time, pointers are public and we can automate their annotation in 67 cases.

In summary, with reasonable manual effort, we were able to keep the stack public for 3 out of 4 cryptographic primitives and isolate public variables from the (secret) stack for the remaining primitive. We expect that this manual effort can easily be automated with compiler support along the lines of existing work for `x86` [30, 85].

7 Related work and conclusion

We discussed transient execution attacks throughout the paper, more details can be found in existing surveys [39, 40].

Formal microarchitectural semantics. Many studies have proposed operational semantics for speculative execution to formally reason about Spectre attacks (see [86] for a detailed comparison up to 2021). Most previous semantics only capture Spectre-PHT, with a few capturing other variants such as Spectre-STL [18, 25, 26, 87, 88], Spectre-BTB, and Spectre-RSB [25, 26]. Contrary to previous operational semantics, ProSpeCT handles *arbitrary* load value prediction [33] and can additionally capture LVI [43]. Using *axiomatic semantics*, Ponce-de-León and Kinder [89] can accommodate new leakage models for different prediction mechanisms and thus cover such cases. In contrast to our work, they can also model attacks based on memory-ordering machine clears [40]. Yet, it is an open question how to model non-interference and declassification, as in our work, using axiomatic semantics.

Declassification definitions. Existing leakage models for secure speculation do not permit any kind of leakage [86] that depends on secrets. Yet, in practice, it is common to treat encrypted secrets as observable. In the literature before transient execution attacks, security properties with intentional leakage (i.e., declassification) have been widely studied [90]. When declassifying ciphertexts is a goal, declassification definitions that are not built for this, such as delimited release [75], may consider programs that unintentionally leak secrets (including the cryptographic keys), as secure, as we show in our technical report [35]. Our definition does not suffer from this limitation and is closer to cryptographically masked flows [91] since it considers a symbolic model for declassified values. Laud [92] pioneered work in the area of security conditions composable with indistinguishability properties of encryption, which was later generalized to other cryptographic primitives [93, 94]. Later, Laud [95] devised necessary conditions to compose cryptographically masked flows with standard cryptographic indistinguishability properties. We stipulate that our security

[7] Of course, no direct comparison can be made as the compiled programs, architectures, and microarchitectures are different.

[8] We track whether secret data is written to non-secret locations using a hardware plugin that we built on top of ProSpeCT.

property will require similar conditions to compose. None of these previous works consider transient execution attacks.

Hardware defenses against Spectre. Many defenses specifically target the cache hierarchy [6, 7, 10, 11, 12, 14, 17, 96, 97, 98], yet, these defenses are still vulnerable to attacks exploiting other side channels [44, 45, 46, 47, 48].

Speculative taint-tracking approaches [9, 13, 15, 78] delay instructions that depend on speculatively loaded data. As shown in [22], these approaches enforce hardware-based secure speculation for sandboxing, but offer no protection for non-speculatively accessed data. Hence, they do not provide secure speculation for the constant-time policy.

DOLMA [99] additionally protects non-speculatively accessed data during speculation, but its performance relies on optimizations allowing (under certain conditions) speculative execution of variable-time instructions and memory operations, which might still be exploited via resource contention.

Data oblivious ISA extensions (OISA) [100] are a hardware-based secrecy-tracking mechanism that prevents secret data from leaking, *including during non-speculative execution*. Software must be updated to use the ISA extensions to make sure that secret data is not used as an unsafe operand. In contrast, ProSpeCT can be retrofitted into existing ISAs and supports existing (constant-time) cryptographic code with minimal (software and hardware) changes.

Secure speculation for the constant-time policy. The idea of propagating security levels from software to hardware and using this information to delay speculative instructions originates from ConTExT [30] and SpectreGuard [31]. Our work contributes the formalization, security proof, and hardware implementation. While our implementation tracks secret memory regions via CSRs, ConTExT and SpectreGuard track secrets at a page-level granularity through, for instance, page table entry bits. ConTExT also tracks public security labels in the cache to reduce over-tainting e.g., when public values are written to the secret stack. As a minor difference, ConTExT and SpectreGuard completely block the forwarding of secret data during speculative execution, whereas ProSpeCT allows executing instructions that do not leak information on their operands. For instance, in the program `if(c){h ← h + 1}`, if `h` is secret, ConTExT and SpectreGuard would stall the instruction `h ← h + 1` until speculations are resolved, whereas ProSpeCT would allow `h ← h + 1` to speculatively execute. Given a whitelist of secure instructions (e.g., [101]), ConTExT and SpectreGuard could adopt this less conservative approach while still being secure. Finally, contrary to ConTExT and SpectreGuard, our work addresses load value speculation and shows that correct predictions must sometimes be rolled back for security.

Speculative Privacy Tracking (SPT) [32] is another taint-tracking mechanism providing secure speculation for the constant-time policy, but without requiring support from applications. SPT initially considers all data as secret, and whenever a register is architecturally leaked, SPT declassifies (i.e., untaints) the register and propagates the information through the microarchitecture with (forward and backward) untainting. SPT also tracks security labels dynamically in the L1D cache. As an example, consider the program in Listing 3 such that only public values are accessed. When x_1 and x_2 are loaded from memory for the first time, SPT marks them as secret. At line 2, x_1 is architecturally leaked to the microarchitectural state, meaning that it gets untainted. Hence, the **load** at line 4 can be executed speculatively. However, because x_2 is tainted, the **load** at line 5 cannot be executed speculatively. In contrast, on ProSpeCT, if the **load** at line 2 corresponds to a public location, then x_2 is set to public and the **load** at line 5 can be executed speculatively. Contrary to SPT, ProSpeCT requires annotations but can label data more precisely.

```
1 x1 ← load a      // x1:H
2 x2 ← load x1     // x2:H, x1:L
3 if (c) 👻
4    z1 ← load x1 // Continue
5    z2 ← load x2 // Stall
```

Listing 3: Taint tracking in SPT where $x_i : H$ indicates that the register x_i gets a secret label and $x_i : L$ indicates that the register x_i gets untainted.

The above defenses have been implemented in simulators and target the x86 platform. In contrast, we provide a hardware implementation for RISC-V, which allows us to evaluate hardware costs. ProSpeCT generalizes these prior efforts with a formalization supporting a wide range of microarchitectural optimizations capturing recent attacks, and a proof that this enables secure speculation for the constant-time policy.

Acknowledgments

We are grateful for the valuable feedback of our shepherd and the other reviewers, which helped us improve our paper. This research was partially funded by the ORSHIN project (Horizon Europe grant agreement No. 101070008) and the Flemish Research Programme Cybersecurity. It has also received funding from ANR TAVA, Carnot Flexsecurity, and PEPR Cyber/Secureval.

References

[1] Nadav Amit, Fred Jacobs, and Michael Wei. "Jump-Switches: Restoring the Performance of Indirect Branches in the Era of Spectre". In: *USENIX Annual Technical Conference*. 2019.

[2] Paul Turner. *Retpoline: A Software Construct for Preventing Branch-Target-Injection*. URL: https://support.google.com/faqs/answer/7625886 (visited on 08/17/2021).

[3] Josh Poimboeuf. *[PATCH v2 0/4] Static Calls [LWN.Net]*. 26, 2018. URL: https://lwn.net/ml/linux-kernel/cover.1543200841.git.jpoimboe@redhat.com/ (visited on 08/24/2021).

[4] Chandler Carruth. *Speculative Load Hardening*. LLVM documentation. URL: https://llvm.org/docs/SpeculativeLoadHardening.html (visited on 02/16/2022).

[5] F.Pizlo. *What Spectre and Meltdown Mean for WebKit*. 2018. URL: https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/ (visited on 07/17/2020).

[6] Mohammadkazem Taram, Ashish Venkat, and Dean M. Tullsen. "Context-Sensitive Fencing: Securing Speculative Execution via Microcode Customization". In: *ASPLOS*. 2019.

[7] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael B. Abu-Ghazaleh. "SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation". In: *DAC*. 2019.

[8] Peinan Li, Lutan Zhao, Rui Hou, Lixin Zhang, and Dan Meng. "Conditional Speculation: An Effective Approach to Safeguard out-of-Order Execution against Spectre Attacks". In: *HPCA*. 2019.

[9] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. "SpecShield: Shielding Speculative Data from Microarchitectural Covert Channels". In: *PACT*. 2019.

[10] Sam Ainsworth and Timothy M. Jones. "MuonTrap: Preventing Cross-Domain Spectre-like Attacks by Capturing Speculative State". In: *ISCA*. 2020.

[11] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. "Efficient invisible speculative execution through selective delay and value prediction". In: *ISCA*. 2019.

[12] Gururaj Saileshwar and Moinuddin K. Qureshi. "CleanupSpec: An "Undo" Approach to Safe Speculation". In: *MICRO*. 2019.

[13] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch, and Baris Kasikci. "NDA: Preventing Speculative Execution Attacks at Their Source". In: *MICRO*. 2019.

[14] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. "InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy". In: *MICRO*. 2018.

[15] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. "Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data". In: *MICRO*. 2019.

[16] Jan Philipp Thoma, Jakob Feldtkeller, Markus Krausz, Tim Güneysu, and Daniel J. Bernstein. "BasicBlocker: ISA Redesign to Make Spectre-Immune CPUs Faster". In: *RAID*. 2021.

[17] Sungkeun Kim, Farabi Mahmud, Jiayi Huang, Pritam Majumder, Neophytos Christou, Abdullah Muzahid, Chia-Che Tsai, and Eun Jung Kim. "ReViCe: Reusing Victim Cache to Prevent Speculative Cache Leakage". In: *SecDev*. 2020.

[18] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. "Hunting the Haunter - Efficient Relational Symbolic Execution for Spectre with Haunted RelSE". In: *NDSS*. 2021.

[19] AMD. *Security Analysis of AMD Predictive Store Forwarding*. 2021. URL: https://www.amd.com/system/files/documents/security-analysis-predictive-store-forwarding.pdf.

[20] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. "Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks". In: *USENIX Security*. Intel Bounty Reward. 2022.

[21] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *IEEE Symposium on Security and Privacy*. 2019.

[22] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. "Hardware-Software Contracts for Secure Speculation". In: *IEEE Symposium on Security and Privacy*. 2021.

[23] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. "Verifying Constant-Time Implementations". In: *USENIX Security Symposium*. 2016.

[24] Ross McIlroy, Jaroslav Sevcík, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. "Spectre Is Here to Stay: An Analysis of Side-Channels and Speculative Execution". In: *CoRR* abs/1902.05178 (2019).

[25] Roberto Guanciale, Musard Balliu, and Mads Dam. "InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis". In: *CCS*. 2020.

[26] Sunjay Cauligi, Craig Disselkoen, Klaus von Gleissenthall, Dean M. Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. "Constant-time foundations for the new spectre era". In: *PLDI*. 2020.

[27] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. “Spectector: Principled Detection of Speculative Information Flows”. In: *IEEE Symposium on Security and Privacy*. 2020.

[28] Gilles Barthe, Sunjay Cauligi, Benjamin Grégoire, Adrien Koutsos, Kevin Liao, Tiago Oliveira, Swarn Priya, Tamara Rezk, and Peter Schwabe. “High-Assurance Cryptography in the Spectre Era”. In: *IEEE Symposium on Security and Privacy*. 2021.

[29] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kici, Ranjit Jhala, Dean M. Tullsen, and Deian Stefan. “Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade”. In: *Proc. ACM Program. Lang.* 5 (POPL 2021).

[30] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. “ConTExT: A Generic Approach for Mitigating Spectre”. In: *NDSS*. 2020.

[31] Jacob Fustos, Farzad Farshchi, and Heechul Yun. “SpectreGuard: An Efficient Data-Centric Defense Mechanism against Spectre Attacks”. In: *DAC*. 2019.

[32] Rutvik Choudhary, Jiyong Yu, Christopher W. Fletcher, and Adam Morrison. “Speculative Privacy Tracking (SPT): Leaking Information from Speculative Execution without Compromising Privacy”. In: *MICRO*. 2021.

[33] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. “Value Locality and Load Value Prediction”. In: *ASPLOS*. 1996.

[34] Mikko H. Lipasti and John Paul Shen. “Exceeding the Dataflow Limit via Value Prediction”. In: *MICRO*. 1996.

[35] Lesly-Ann Daniel, Marton Bognar, Job Noorman, Sébastien Bardin, Tamara Rezk, and Frank Piessens. *ProSpeCT: Provably Secure Speculation for the Constant-Time Policy (Extended version)*. 2023. URL: https://arxiv.org/abs/2302.12108.

[36] Giorgi Maisuradze and Christian Rossow. “Ret2spec: Speculative Execution Using Return Stack Buffers”. In: *CCS*. 2018.

[37] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael B. Abu-Ghazaleh. “Spectre Returns! Speculation Attacks Using the Return Stack Buffer”. In: *WOOT @ USENIX Security Symposium*. 2018.

[38] Jann Horn. *Speculative Execution, Variant 4: Speculative Store Bypass*. 2018. URL: https://bugs.chromium.org/p/project-zero/issues/detail?id=1528 (visited on 10/12/2020).

[39] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. “A Systematic Evaluation of Transient Execution Attacks and Defenses”. In: *USENIX Security Symposium*. 2019.

[40] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. “Rage Against the Machine Clear: A Systematic Analysis of Machine Clears and Their Implications for Transient Execution Attacks”. In: *USENIX Security Symposium*. 2021.

[41] Moritz Lipp et al. “Meltdown: Reading Kernel Memory from User Space”. In: *USENIX Security Symposium*. 2018.

[42] Jo Van Bulck et al. “Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution”. In: *USENIX Security Symposium*. 2018.

[43] Jo Van Bulck et al. “LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection”. In: *IEEE Symposium on Security and Privacy*. 2020.

[44] Md Hafizul Islam Chowdhuryy and Fan Yao. “Leaking Secrets through Modern Branch Predictor in the Speculative World”. In: *IEEE Transactions on Computers* (2021).

[45] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. “NetSpectre: Read Arbitrary Memory over Network”. In: *ESORICS (1)*. 2019.

[46] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. “SMoTherSpectre: Exploiting Speculative Execution through Port Contention”. In: *CCS*. 2019.

[47] Jacob Fustos, Michael Garrett Bechtel, and Heechul Yun. “SpectreRewind: Leaking Secrets to Past Instructions”. In: *ASHES@CCS*. 2020.

[48] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M Tullsen, and Ashish Venkat. “I See Dead μops: Leaking Secrets via Intel/AMD Micro-Op Caches”. In: *ICSA* (2021).

[49] Kevin Cheang, Cameron Rasmussen, Sanjit A. Seshia, and Pramod Subramanyan. “A Formal Approach to Secure Speculation”. In: *CSF*. 2019.

[50] Emmanuel Pescosta, Georg Weissenbacher, and Florian Zuleger. “Bounded Model Checking of Speculative Non-Interference”. In: *ICCAD*. 2021.

[51] Meng Wu and Chao Wang. “Abstract Interpretation under Speculative Execution”. In: *PLDI*. 2019.

[52] Guanhua Wang, Sudipta Chattopadhyay, Arnab Kumar Biswas, Tulika Mitra, and Abhik Roychoudhury. “KLEESpectre: Detecting Information Leakage through Speculative Cache Attacks via Symbolic Execution”. In: *ACM Trans. Softw. Eng. Methodol.* 29.3 (2020).

[53] Shengjian Guo, Yueqi Chen, Peng Li, Yueqiang Cheng, Huibo Wang, Meng Wu, and Zhiqiang Zuo. “SpecuSym: Speculative Symbolic Execution for Cache Timing Leak Detection”. In: ICSE 2020 Technical Papers. 2020.

[54] Zhenxiao Qi, Qian Feng, Yueqiang Cheng, Mengjia Yan, Peng Li, Heng Yin, and Tao Wei. “SpecTaint: Speculative Taint Analysis for Discovering Spectre Gadgets”. In: *NDSS*. 2021.

[55] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. “Oo7: Low-overhead Defense against Spectre Attacks via Program Analysis”. In: *IEEE Transactions on Software Engineering* (2020).

[56] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. “The Security Impact of a New Cryptographic Library”. In: *LATINCRYPT*. 2012.

[57] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. “HACL*: A Verified Modern Cryptographic Library”. In: *CCS*. 2017.

[58] *BearSSL - Constant-Time Crypto*. URL: https://bearssl.org/constanttime.html (visited on 05/07/2019).

[59] José Bacelar Almeida et al. “Jasmin: High-assurance and High-Speed Cryptography”. In: *CCS*. 2017.

[60] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. “MicroWalk: A Framework for Finding Side Channels in Binaries”. In: *ACSAC* (San Juan, PR, USA). 2018.

[61] Qinkun Bao, Zihao Wang, Xiaoting Li, James R. Larus, and Dinghao Wu. “Abacus: Precise Side-Channel Analysis”. In: *ICSE*. 2021.

[62] Goran Doychev and Boris Köpf. “Rigorous Analysis of Software Countermeasures against Cache Attacks”. In: *PLDI*. 2017.

[63] Adam Langley. *ImperialViolet - Checking That Functions Are Constant Time with Valgrind*. 2010. URL: https://www.imperialviolet.org/2010/04/01/ctgrind.html (visited on 03/14/2019).

[64] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. “Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level”. In: *IEEE Symposium on Security and Privacy*. 2020.

[65] Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannesmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. “FaCT: A Flexible, Constant-Time Programming Language”. In: *SecDev*. 2017.

[66] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut T. Kandemir. “CaSym: Cache Aware Symbolic Execution for Side Channel Detection and Mitigation”. In: *IEEE Symposium on Security and Privacy*. 2019.

[67] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. “CacheAudit: A Tool for the Static Analysis of Cache Side Channels”. In: *USENIX Security Symposium*. 2013.

[68] Shaobo He, Michael Emmi, and Gabriela F. Ciocarlie. “Ct-Fuzz: Fuzzing for Timing Leaks”. In: *ICST*. 2020.

[69] Jan Wichelmann, Florian Sieck, Anna Pätschke, and Thomas Eisenbarth. “Microwalk-CI: Practical Side-Channel Analysis for JavaScript Applications”. In: *CoRR* abs/2208.14942 (2022).

[70] Mohammad Behnia et al. “Speculative Interference Attacks: Breaking Invisible Speculation Schemes”. In: *ASPLOS*. 2021.

[71] Johann Großschädl, Elisabeth Oswald, Dan Page, and Michael Tunstall. “Side-Channel Analysis of Cryptographic Software via Early-Terminating Multiplications”. In: *ICISC*. 2009.

[72] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. “On Subnormal Floating Point and Abnormal Timing”. In: *IEEE Symposium on Security and Privacy*. 2015.

[73] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. “Practical Mitigations for Timing-Based Side-Channel Attacks on Modern X86 Processors”. In: *IEEE Symposium on Security and Privacy*. 2009.

[74] Gilles Barthe, Pedro R. D’Argenio, and Tamara Rezk. “Secure Information Flow by Self-Composition”. In: *CSFW*. 2004.

[75] Andrei Sabelfeld and Andrew C. Myers. “A Model for Delimited Information Release”. In: *Software Security - Theories and Systems, Second Mext-NSF-JSPS International Symposium, ISSS 2003, Tokyo, Japan, November 4-6, 2003, Revised Papers*. 2003.

[76] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. “Towards a Logical Account of Declassification”. In: *PLAS*. 2007.

[77] Aslan Askarov and Andrei Sabelfeld. “Gradual Release: Unifying Declassification, Encryption and Key Release Policies”. In: *IEEE Symposium on Security and Privacy*. 2007.

[78] Jiyong Yu, Namrata Mantri, Josep Torrellas, Adam Morrison, and Christopher W. Fletcher. “Speculative Data-Oblivious Execution: Mobilizing Safe Prediction for Safe and Efficient Speculative Execution”. In: *ISCA*. 2020.

[79] James E. Smith. “A Study of Branch Prediction Strategies”. In: *ISCA*. 1981.

[80] Johnny F. K. Lee and Alan Jay Smith. “Branch Prediction Strategies and Branch Target Buffer Design”. In: *Computer* 17.1 (1984).

[81] Charles Papon. *SpinalHDL, A Scala based HDL*. https://github.com/SpinalHDL/SpinalHDL.

[82] Charles Papon. *VexRiscv, A FPGA friendly 32 bit RISC-V CPU implementation*. https://github.com/SpinalHDL/VexRiscv.

[83] Marton Bognar, Jo Van Bulck, and Frank Piessens. “Mind the Gap: Studying the Insecurity of Provably Secure Embedded Trusted Execution Architectures”. In: *IEEE Symposium on Security and Privacy*. 2022.

[84] Daniel J. Bernstein. *djbsort*. URL: https://sorting.cr.yp.to/ (visited on 03/27/2022).

[85] Laurent Simon, David Chisnall, and Ross J. Anderson. “What You Get Is What You C: Controlling Side Effects in Mainstream C Compilers”. In: *EuroS&P*. 2018.

[86] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. “SoK: Practical Foundations for Software Spectre Defenses”. In: *IEEE Symposium on Security and Privacy*. 2022.

[87] Gilles Barthe, Sunjay Cauligi, Benjamin Grégoire, Adrien Koutsos, Kevin Liao, Tiago Oliveira, Swarn Priya, Tamara Rezk, and Peter Schwabe. “High-Assurance Cryptography in the Spectre Era”. In: *IEEE Symposium on Security and Privacy*. 2021.

[88] Xaver Fabian, Marco Guarnieri, and Marco Patrignani. “Automatic Detection of Speculative Execution Combinations”. In: *CCS*. 2022.

[89] Hernán Ponce de León and Johannes Kinder. “Cats vs. Spectre: An Axiomatic Approach to Modeling Speculative Execution Attacks”. In: *IEEE Symposium on Security and Privacy, CA, USA*. 2022.

[90] Andrei Sabelfeld and David Sands. “Declassification: Dimensions and principles”. In: *J. Comput. Secur.* 17.5 (2009).

[91] Aslan Askarov, Daniel Hedin, and Andrei Sabelfeld. “Cryptographically-masked flows”. In: *Theor. Comput. Sci.* 402.2-3 (2008).

[92] Peeter Laud. “Semantics and Program Analysis of Computationally Secure Information Flow”. In: *ESOP*. 2001.

[93] Cédric Fournet and Tamara Rezk. “Cryptographically sound implementations for typed information-flow security”. In: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. 2008.

[94] Cédric Fournet, Jérémy Planul, and Tamara Rezk. “Information-flow types for homomorphic encryptions”. In: *CCS*. 2011.

[95] Peeter Laud. “On the computational soundness of cryptographically masked flows”. In: *POPL*. 2008.

[96] Chang Liu, Austin Harris, Martin Maas, Michael W. Hicks, Mohit Tiwari, and Elaine Shi. “GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation”. In: *ASPLOS*. 2015.

[97] Vladimir Kiriansky, Ilia A. Lebedev, Saman P. Amarasinghe, Srinivas Devadas, and Joel S. Emer. “DAWG: A Defense against Cache Timing Attacks in Speculative Execution Processors”. In: *MICRO*. 2018.

[98] Sam Ainsworth. “GhostMinion: A Strictness-Ordered Cache System for Spectre Mitigation”. In: *MICRO*. 2021.

[99] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. “DOLMA: Securing Speculation with the Principle of Transient Non-Observability”. In: *USENIX Security Symposium*. 2021.

[100] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W. Fletcher. “Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing”. In: *NDSS*. 2019.

[101] Intel. *Data Operand Independent Timing Instructions*. 2022. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/resources/data-operand-independent-timing-instructions.html.

[102] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. “A survey of microarchitectural timing attacks and countermeasures on contemporary hardware”. In: *J. Cryptogr. Eng.* 8.1 (2018).

[103] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. “SpecFuzz: Bringing Spectre-type Vulnerabilities to the Surface”. In: *USENIX Security Symposium*. 2020.

A Artifact Appendix

A.1 Abstract

The artifact contains the source code of the base Proteus processor extended with PROSPECT, alongside the benchmarks and security tests from our paper. All materials (except for the tool required for hardware cost measurements) are bundled into a Docker container and distributed on GitHub.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None, our artifact is contained in a Docker container, it does not perform any attacks against the host system and it does not use user data.

A.2.2 How to access

The artifact is available on GitHub at the following URL: https://github.com/proteus-core/prospect/tree/usenix_artifact.

A.2.3 Hardware dependencies

None.

A.2.4 Software dependencies

Our artifact uses the following two tools, which are available for both Windows and Linux.

- Docker and 7 GB of disk space for the container (https://docs.docker.com/engine/install/).

- Xilinx Vivado 2022.2 Standard Edition, requiring approximately 55 GB of disk space (https://www.xilinx.com/products/design-tools/vivado/vivado-ml.html).

A.2.5 Benchmarks

Our evaluation uses modified benchmarks from the SpectreGuard paper, which are included in our artifact.

A.3 Set-up

A.3.1 Installation

1. Install the two dependencies (Docker and Vivado). Our repository contains detailed instructions on setting up Vivado to minimize the required disk space.

2. Clone our GitHub repository or download the Dockerfile from the root directory (https://github.com/proteus-core/prospect/tree/usenix_artifact).

3. Build the Docker container by following the instructions in the `README.md` of the repository (building takes approximately 2 hours on a mid-range desktop).

A.3.2 Basic Test

The security evaluation can be run from the Docker container using the following commands:

```
// first, launch the container
$ docker run -i -t prospect

// inside the container, run the tests
# cd /prospect/tests/spectre-tests/
# ./eval.py /proteus-base/sim/build/base \
    /prospect/sim/build/prospect
TEST secret-before-branch
SECURE VARIANT:   Secret did not leak!
INSECURE VARIANT: Secret leaked!
[...]
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): PROSPECT prevents the leakage of secrets from well-annotated programs via Spectre attacks. This is shown by experiment (E1) described in Section 6.2, which executes programs vulnerable to Spectre on the baseline and the extended secure implementation.

(C2): PROSPECT incurs no overhead on precisely annotated constant-time code. This is shown by experiment (E2), described in Section 6.2 (Runtime overhead) and Table 1.

(C3): PROSPECT only incurs a small overhead in terms of hardware cost. This is shown by experiment (E3), described in Section 6.2 (Hardware cost).

A.4.2 Experiments

(E1): [Security tests, 5 human-minutes]:

How to: The experiment is performed in the container by launching a script (identical to the basic test A.3.2).

Preparation: Launch the container with `docker run -i -t prospect` and navigate to the experiment with `cd /prospect/tests/spectre-tests`.

Execution: Run the following command:

```
./eval.py /proteus-base/sim/build/base \
/prospect/sim/build/prospect
```

This will run and evaluate the experiments with both the baseline implementation (first argument) and the PROSPECT-extended version (second argument).

Results: The results are displayed as text. The security evaluation should fail with the baseline implementation and succeed with the extension, validating claim (C1).

(E2): [Runtime overhead, 5 human-minutes + 9 compute-hours]:

How to: The experiment is performed in the container by launching a script.

Preparation: Launch the container with `docker run -i -t prospect` and navigate to the experiment with `cd /prospect/tests/synthetic-benchmark`.

Execution: Run the following command:

```
./eval.py \
/proteus-base/sim/build/base_nodump \
/prospect/sim/build/prospect_nodump
```

This will run and evaluate the experiments with both the baseline implementation (first argument) and the ProSpeCT-extended version (second argument), using the variants compiled with no waveform dumping to save disk space.

Results: The results are displayed as text. The generated table should reflect Table 1 from the paper, validating claim (C2).

(E3): [Hardware cost, 1 human-hour + 2 compute-hours]:

How to: The experiment is performed in Vivado, using generated Verilog files from the Docker container.

Preparation: Follow the instructions under the heading *Hardware overhead* in `README.md` to obtain the Verilog files used for the synthesis and to set up the Vivado project (*Creating the Vivado project*).

Execution: Follow the instructions under the heading *Running the Vivado evaluation* in `README.md` to (iteratively) obtain the hardware costs of both the baseline and the ProSpeCT-extended hardware design.

Results: The results of the synthesis should be interpreted according to the description under the heading *Interpreting the results* in the `README.md` and compared to the reported numbers in the paper under the heading *Hardware cost* (Section 6.2).

A.5 Notes on Reusability

Using the `newlib` board support package included in this repository and building on the scripts used for our benchmarks, it is possible to run other benchmarks on Proteus and ProSpeCT, making additional benchmarking and security tests possible. The source code of ProSpeCT can also be modified to investigate tradeoffs or to extend the offered security guarantees.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2023/`.