dfence: Fine-Grained Speculation Barriers for Efficient and Effective Hardware-Software Protection in the Spectre Era

Davide Davoli
davide.davoli@mpi-sp.org
MPI-SP
Bochum, Germany

Marton Bognar
marton.bognar@kuleuven.be
DistriNet, KU Leuven
Leuven, Belgium

Lesly-Ann Daniel
lesly-ann.daniel@eurecom.fr
EURECOM
Sophia Antipolis, France

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

Frank Piessens
frank.piessens@kuleuven.be
DistriNet, KU Leuven
Leuven, Belgium

Tamara Rezk
tamara.rezk@inria.fr
Inria
Sophia Antipolis, France

Abstract

Speculative execution attacks such as Spectre-PHT and Spectre-STL remain a critical security concern in modern processors. While software-based mitigations like Speculative Load Hardening (SLH) offer effective protection against Spectre-PHT, they are limited in scope and require software-managed speculative masks, which can be error-prone and costly. Defenses against Spectre-STL, such as the Speculative Store Bypass Disable bit (SSBD), incur additional performance overhead and lack fine-grained control. In this work, we introduce **dfence**, a new CPU instruction that generalizes SLH to mitigate both Spectre-PHT and Spectre-STL with minimal hardware support. **dfence** enables developers to annotate sensitive registers, with the hardware ensuring that these values do not leak transiently. We implement **dfence** in the Proteus CPU and evaluate its security and performance, demonstrating less than 1% average performance overhead for our benchmarks. In addition, to support easy and secure adoption, we design a type system that statically verifies the correct placement of **dfence** instructions in code.

CCS Concepts

• **Security and privacy** → **Formal methods and theory of security**; **Security in hardware**.

Keywords

Spectre defense, speculation barriers, RISC-V, type system

ACM Reference Format:
Davide Davoli, Marton Bognar, Lesly-Ann Daniel, Benjamin Grégoire, Frank Piessens, and Tamara Rezk. 2026. **dfence**: Fine-Grained Speculation Barriers for Efficient and Effective Hardware-Software Protection in the Spectre Era. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3830454.3832748

CCS '26, The Hague, Netherlands

1 Introduction

After the discovery of Spectre attacks [60], vendors scrambled to develop rapid software patches to mitigate these hardware-based vulnerabilities. A common mitigation strategy has been to repurpose synchronization primitives, such as fence instructions, as speculation barriers to block the leakage of secrets during speculative execution. On x86 systems, this is typically implemented using the **lfence** instruction [54], while Arm processors use the **dsb** instruction [65] for similar purposes. However, these full-fence approaches, while effective, impose significant performance overheads when applied broadly across speculative execution paths.

To address this performance penalty, hardware vendors have advocated for more fine-grained solutions. For example, Arm recommends techniques such as software-based index masking to mitigate Spectre-PHT (Spectre-v1) [66], which selectively restricts speculative execution rather than halting it entirely. Similarly, Intel and AMD recommend hybrid approaches that combine fences with other software mitigations such as retpolines [6, 48, 50, 51]. These techniques aim to balance security with performance by limiting the scope of speculation barriers to the necessary minimum.

One of these fine-grained mitigations, Speculative Load Hardening (SLH), was introduced in LLVM [25] to address Spectre-PHT attacks. SLH is implemented at the compiler level and is based on conditional masking. SLH maintains a global misspeculation flag in a dedicated register, updated for each branch condition that could be mispredicted. This misspeculation flag is used to mask operands of unsafe instructions, preventing data leakage during transient execution. Initially, SLH was used to automatically mask load addresses or values, but this approach was found to be incomplete and was later extended to mask all potential sources of leakage [73, 91]. This extended approach incurs an average performance overhead of approximately 150% in the SPEC2017 benchmark when protecting all branches—still a noticeable improvement over the 300% overhead of full fence insertion [91]. An alternative approach is to let the user manually insert SLH instrumentations, guided by a type system ensuring that their placement prevents confidential information leakage [7]. This strategy reduces the number of inserted protections and the resulting overhead, but it demands manual management of the misspeculation flag and often requires extensive code modifications.

More importantly, SLH can only be applied to Spectre-PHT due to the inherent limitation of detecting mispredictions from software. For instance, in the case of Spectre-STL (v4), it is impossible to detect from software when a load uses transiently forwarded sensitive data from a prior store. The vendor-recommended defense for Spectre-STL is Speculative Store Bypass Disable (SSBD), a hardware-based mitigation that fully disables STL speculation, incurring additional performance overhead. To address the lack of generalization and performance penalties of current mitigations, hardware modifications like taint-tracking solutions [30, 36, 85, 90] have been proposed, but their adoption is hindered by often requiring extensive hardware changes, increasing implementation complexity and cost.

In this work, we investigate the following research question: *Can we apply the principles of SLH to prevent Spectre-STL in addition to Spectre-PHT with minimal hardware changes?*

We answer this question affirmatively by introducing a new assembly instruction called **dfence**.[1] This instruction generalizes the core principles of SLH beyond Spectre-PHT and makes it more lightweight to use for programmers by alleviating the need to track speculation in software. The software—following the principle of SLH—can use **dfence** x to protect the contents of register x from transient leakage. The hardware tracks the sources of speculative execution and ensures that x does not leak speculatively. Unlike SLH, **dfence** eliminates the need to track a speculative mask in software. This reduces the likelihood of errors, as it removes the need to continuously update the misspeculation flag and provides greater flexibility compared to SLH, where the register holding the misspeculation cannot be spilled. Additionally, tracking speculation in hardware enables detecting additional types of speculation that are invisible to software (such as Spectre-STL). To protect programs, we propose and implement a low-level type system for the Jasmin [3] language, which automatically verifies that **dfence** protections (or standard serializing fences) are correctly inserted. We formally prove the effectiveness of our defense against Spectre-PHT, and variants of Spectre-STL—namely Speculative Store Bypass (SSB) and Predictive Store Forwarding (PSF). We also discuss extensions to other Spectre variants, including the recent SLAP attack [58], which **dfence** can mitigate with very small changes (§9).

To validate our solution, we implement **dfence** in a prototype RISC-V CPU, Proteus [21], showcasing its low hardware cost and effectiveness in real-world scenarios. We evaluate its security by using non-interference testing, and its performance on different cryptographic schemes such as ML-DSA [40] and ML-KEM [22]. We demonstrate that our generalized approach incurs lower performance overhead than the combined costs of SLH and SSBD or serializing fences, offering a practical and scalable alternative. By achieving robust protection against speculative execution vulnerabilities without prohibitive costs or complexity, our work paves the way for high-assurance software-hardware Spectre defenses.

In summary, our contributions are the following:

- Introducing **dfence**, generalizing SLH to mitigate Spectre-PHT and -STL. **dfence** requires minimal hardware support (§3) and simplifies software instrumentation, resulting in fewer constraints and lower complexity (§6);
- Developing a formally proven type system that enables verification of correct **dfence** insertion (§4);
- Implementing **dfence** in the Proteus RISC-V core and demonstrating its minimal hardware overhead (§5);
- Evaluating **dfence**'s security and performance, providing evidence for protection against Spectre-PHT, -SSB and -PSF, and an average overhead of less than 1%, smaller than standard protection mechanisms (§6).

An extended version of this paper with the operational semantics of **dfence**, additional proofs, and benchmark results is available [37].

Open science. To facilitate reproducing and building on our research, we archive all implementation and evaluation materials at https://doi.org/10.5281/zenodo.20770661, and will incorporate most components in the upstream Proteus ecosystem at https://github.com/proteus-core.

2 Background

In this section, we provide some background information on Spectre attacks and their mitigations.

2.1 Spectre Attacks

Microarchitectural attacks exploit microarchitectural side effects of a program's execution to extract sensitive information. Any computation affecting the microarchitectural state in a way that can be measured by an attacker reveals information about data processed by a victim (via e.g., timing [61], cache [12], predictor state [31], micro-op cache [76], and other microarchitectural resource contention [2, 19, 41, 78]). For instance, a memory load `x := a[E]` affects the cache state based on the accessed address `a[E]`. An attacker can exploit this change through cache side channels to infer the address. We refer to instruction operands that expose information about their values through microarchitectural side-channels as *unsafe operands*. For simplicity, in this paper, we assume the set of unsafe operands, a.k.a. the *leakage model*, to be restricted to memory operands and guards of control-flow instructions. However, our results can trivially be extended to other types of unsafe operands, such as variable-time divisions or multiplications [43].

Processors can execute instructions out-of-order and employ various *speculation* mechanisms to predict, for instance, the next instruction to execute. If the prediction is wrong, the processor simply reverts the incorrectly executed instructions—called *transient* instructions—and continues along the correct path. Spectre attacks [60] exploit these speculation mechanisms to force a victim to leak secrets during transient execution, as this still produces visible microarchitectural side effects. By mistraining predictors, a victim can be forced into transiently executing a sequence of instructions chosen to encode secrets in the microarchitectural state, which can later be extracted. Many Spectre attack variants exist [5, 47, 60, 63, 71, 72, 74], classified in various surveys [24, 74, 75] according to which speculation mechanisms they exploit. In this paper, we focus on *Spectre-PHT* (also known as Spectre-v1) [60] and *Spectre-STL*. We use Spectre-STL as an umbrella term for *Store-to-Load dependency speculation*, which encompasses two distinct variants: *Spectre-SSB* [47, 49] (also known as Spectre-v4) and *Spectre-PSF* [5, 52]. We illustrate all these variants in Listing 1.

[1] A play on 'defense', with the 'd' loosely suggesting its selective or discerning behavior.

```
  0 - 15: a[16]
      16: s[1]
17 - 272: b[256]
```

(a) Memory layout.

```
4  if (i < |a|)
5    x := a[i]
6    y := b[x]
```

(b) Spectre-PHT (Spectre-v1).

```
7  s[0] := 0
8  x := s[0]
9  y := b[x]
```

(c) Spectre-SSB (Spectre-v4).

```
10 s[0] := s[0] + 1
11 x := a[0]
12 y := b[x]
```

(d) Spectre-PSF.

Listing 1: Examples of code snippets vulnerable to transient execution attacks. All programs share the memory layout in Listing 1a where s is secret. The symbol indicates an instruction triggering transient execution and indicates leakage.

Spectre-PHT (Pattern History Table) exploits the conditional branch outcome predictor. In Listing 1b, an attacker can train the branch predictor to predict the guard to be true (line 4), execute the code with an out-of-bounds index i = 16 that transiently accesses s[0] (line 5), and leak it to the microarchitectural state (line 6).

Spectre-SSB (Speculative Store Bypass) exploits the ability of a load instruction to speculatively bypass preceding stores with unresolved addresses. In Listing 1c, the load of s[0] (line 8) may speculatively bypass the prior store to s[0] (line 7). This enables transient access to the secret value before it is zeroed out in memory, leading to its leakage (line 9).

Spectre-PSF (Predictive Store Forwarding) exploits the fact that a store can speculatively forward its value to a subsequent load. In Listing 1d, the store (line 10) can transiently forward the secret s[0] + 1 to the next load (line 11), which leaks at line 12.

2.2 Countermeasures

Spectre attacks are possible when secrets speculatively flow to unsafe operands. Two main policies have been proposed to protect against Spectre [26, 44]: *speculative constant-time* [27] and *speculative sandboxing* [44].

Speculative constant-time is an extension of *constant-time programming*, the traditional defense against (non-speculative) microarchitectural attacks, already largely followed by cryptographic libraries [4, 17]. While constant-time requires that program secrets do not flow to unsafe operands (architecturally), speculative constant-time extends this requirement to speculative paths as well.

Speculative sandboxing requires that programs do not speculatively leak data that is not architecturally accessed (in this model, we can consider transiently accessed data to be secret). Thus, if a program reads out-of-bounds data during speculation, speculative sandboxing forbids this data from speculatively flowing to an unsafe instruction. Speculative sandboxing is weaker than speculative constant-time as it does not protect architecturally accessed secrets; however, contrary to speculative constant-time, it does not require identifying explicit program secrets and is therefore better suited for protecting general-purpose programs.

Both policies disallow secret data to speculatively flow to unsafe instructions. To enforce this, a first solution is to insert speculation barriers (e.g., **lfence**) to cut speculative paths before they can leak secrets. For instance, all programs in Listing 1 can be protected

```
1 m := 0; c := i < |a|
2 if (c)
3   m := m&(−c);
4   x := a[i]
5   x := x&m;
6   y := b[x]
```

Listing 2: Listing 1b secured with SLH.

```
1 if (i < |a|)
2   x := a[i]
3   dfence x
4   y := b[x]
```

(a) Listing 1b protected with dfence.

```
5 s[0] := s[0] + 1
6 x := a[0]
7 dfence x
8 y := b[x]
```

(b) Listing 1d protected with dfence.

Listing 3: Spectre-PHT and Spectre-PSF gadgets protected with dfence. Spectre-SSB can be handled similarly to Spectre-PSF.

by inserting a barrier right before the last load. However, existing fences needlessly restrict speculative execution of *all* instructions after the barrier, increasing the performance overhead. To alleviate this performance cost, a more efficient defense, *Speculative Load Hardening* (SLH), is usually employed [7, 25, 91]. As illustrated in Listing 2, this defense consists of keeping track of possibly misspeculated branch conditions as a mask in a dedicated register (line 3) and using this to mask the secrets before they can reach an unsafe operand (line 5).

Problem: SLH faces two significant limitations. First, its *generality* is restricted: it is effective only against Spectre-PHT; protecting against Spectre-STL requires either inserting costly **fence** instructions or disabling SSB and PSF predictions (e.g., using SSBD [49] on Intel processors). In our benchmarks, SSBD alone introduces up to 56% overhead. Second, SLH requires a dedicated register to maintain the misspeculation flag and extra instructions, increasing register pressure and degrading performance. This effect is reflected in our results, where SLH alone introduces up to 9% overhead.

3 Design of dfence

We propose a new instruction, **dfence** x, which, in the spirit of SLH, can be used to protect a register x before it speculatively reaches an unsafe operand.

> **Specification of dfence**
>
> **dfence** x acts as a *selective register fence*: it guarantees that its operand x is not forwarded to subsequent instructions until this value becomes *non-speculative*.

Listing 3 shows how to protect the programs from Listing 1 with **dfence**. The idea is to protect the register x, which can transiently contain secret data, before it reaches the unsafe load operand. The final load instruction can therefore only be executed when x becomes non-speculative—i.e., in Listing 3a, the branch is resolved, and in Listing 3b, dependencies between the load (line 6) and store (line 5) are resolved—at which point x only contains public data.

3.1 Design Choices

The **dfence** instruction relies on a hardware-software co-design: to achieve end-to-end security, the responsibilities are split between hardware and software, following a security contract [44]. On the hardware side, in addition to implementing **dfence**, hardware vendors must provide a *leakage model* determining unsafe operands. This leakage model is similar to the list of data-independent-timing instructions already provided by hardware vendors such as AMD and Intel in their DIT/DOIT modes [53, 67]. On the software side, developers are required to (R1) handle architectural leakage, and (R2) correctly insert **dfence** to make sure that secrets do not flow speculatively to unsafe operands. How to achieve these requirements depends on the policy. In the case of speculative constant-time, requirement R1 is achieved by following the standard constant-time programming discipline, and requirement R2 is achieved by additionally protecting secrets on speculative paths. In the case of speculative sandboxing, requirement R1 is achieved by ensuring (architectural) memory safety, and requirement R2 is achieved by protecting the result of potentially speculative loads from leaking.

Type system. To address the contract requirements for software and assist developers in correctly inserting **dfence** protections, we propose a type system that accepts only programs adhering to the constant-time discipline (based on secrecy annotations) with properly placed **dfence** instructions. Our type system considers the results of speculative loads to be secret and ensures that they are protected with **dfence**. This approach guarantees compliance with the speculative constant-time policy (§4). Moreover, our type system still guarantees compliance with the speculative sandboxing policy for general-purpose programs without secrecy annotations.

Our type system tracks the confidentiality level, public (L) or secret (H), of registers and arrays. For instance, in Listing 3, array a initially has type (L), while s has type (H). Registers have two types to account for both *non-speculative* and *speculative* execution.

We assume that programs are memory-safe in normal execution. Importantly, to account for the fact that load instructions can be memory-unsafe in transient execution, or speculatively get forwarded secret values, the speculative type of their destination register is always set to H. For instance, in Listing 3, after the load from array a (lines 2 and 6), x is typed (L, H): non-speculatively public but speculatively secret.

After a **dfence** x instruction, our type system sets the speculative type of x to its non-speculative type. This captures the intuition that, after the fence, the value of x is non-speculative and cannot speculatively contain a secret if x is public. Therefore, in Listing 3, the **dfence** x instructions set the type of x to (L, L) (lines 3 and 7).

The type system accepts instructions with unsafe operands (e.g., loads) only if the unsafe operand is typed (L, L). Therefore, the insecure programs in Listing 1, without **dfence**, do not typecheck as x has type (L, H). However, the secured versions in Listing 3 do typecheck as x has type (L, L).

We implemented the type system in Jasmin [3], a low-level language designed for writing high-speed cryptography.

Hardware. We propose an implementation of **dfence** in hardware (§5). The processor tracks all sources of speculative execution, and upon encountering a **dfence** x instruction, delays forwarding x until older sources of speculation have been resolved. We also implement and discuss other strategies in §5.3 and §9.2.

3.2 Evaluation Setting and Limitations

We evaluate **dfence** on cryptographic code written in the Jasmin low-level programming language. In this setting, our type system makes assumptions about memory safety and predictability of jump targets, which significantly improve the precision of static analysis for the protected program. These assumptions are standard and practically relevant in this context, as they typically hold for assembly generated from high-level languages and cryptographic software. This choice was motivated by the goal of demonstrating that **dfence** can enforce strong security guarantees (SCT) while incurring modest performance costs.

However, these limitations should not be regarded as inherent to the approach: **dfence** is not restricted to languages or settings where (i) security annotations are available, (ii) jump targets are predictable, and (iii) programs are sequentially safe. When such assumptions do not hold, **dfence** may be combined with more conservative static analyses or target weaker security properties, such as relative SCT, as also considered by Blade [82] (cf. §7) and SLH [25]. In addition, compiler-based hardening passes such as SESES (LLVM's `x86-seses*` flags), USLH [91], and Eclipse [32] use a static analysis to identify where an SLH-style protection mechanism should be inserted, providing protection guarantees that are often more suitable to non-cryptographic code, not requiring security annotations. The **dfence** instruction can serve as a drop-in replacement for the protection primitive in these compiler-based hardening passes, likely improving performance (cf. §6.2). Evaluating **dfence** beyond SCT is an interesting area for future work.

Finally, our implementation is currently limited to the RISC-V architecture, specifically the Proteus processor. Consequently, the results of our evaluation are representative of this platform. However, **dfence** is not inherently tied to RISC-V. The proposed mechanism is ISA-agnostic and can be integrated into other architectures and (commercial) processors.

3.3 Threat Model

We assume sequential safety—i.e., memory safety during non-speculative execution—but safety during speculative execution is not required. For speculative constant-time (but not for speculative sandboxing), we assume a programming model where developers annotate secret data in their program (typical for cryptographic implementations).

Our threat model includes microarchitectural timing attacks. In particular, an attacker can measure execution time with clock cycle precision and observe changes to the shared microarchitectural state, restricted by the leakage contract. For simplicity, we assume the standard constant-time leakage model (leaking memory operands and the program counter), but our results can be extended to other leakage models. This paper focuses on PHT and STL speculation. Although we model indirect branches, we assume RSB and BTB to be mitigated with complementary defenses (cf. §9.1). We discuss extensions to other forms of speculation in §9.1.

We exclude physical side channels like power consumption or electromagnetic emissions. These attacks assume stronger attacker models and are subject to orthogonal defenses like masking [20, 79]. Additionally, dynamic frequency throttling attacks (e.g., Hertzbleed) [70, 84] are left out of scope; this feature should be disabled when computing on secret data. Data memory-dependent prefetch attacks [28, 83] are also out of scope.

Integration with non-cryptographic code. The **dfence** primitive guarantees that values explicitly marked as secret cannot be transiently propagated to unsafe operands, even in the presence of speculative execution. These guarantees, however, apply only to the protected and annotated portion of the code. When cryptographic components are linked with generic non-protected code, secret data may still reside in shared memory and therefore become accessible to unprotected components. If the program is linked to another program, we assume that the linking process robustly preserves speculative noninterference [73].

This guarantee is largely orthogonal to those provided by **dfence**. The role of **dfence** is to prevent secret leakage within the protected cryptographic component itself, whereas the surrounding non-cryptographic code must be secured through complementary mechanisms that prevent the introduction of exploitable speculative gadgets. This approach is consistent with existing practice, where different protection techniques are combined to secure different layers of the system. For instance, cryptographic routines can be isolated using compartmentalization mechanisms akin to Intel MPK so that unprotected components cannot access secure compartments either architecturally or speculatively without going through specific API calls [62].

4 Type System for dfence

In this section, we propose a type system that ensures that software is correctly protected using **dfence**. For simplicity, our formalization focuses on the stronger speculative constant-time policy. We discuss sandboxing in §9.3.

Overview of the type system. Our type system is designed to integrate with Jasmin [3], drawing inspiration from its existing type system for speculative constant-time [7]. While the Jasmin type system is tailored for SLH protection and focuses solely on Spectre-PHT, our type system is designed for **dfence** and provides protection against Spectre-PHT, Spectre-SSB, and Spectre-PSF. The main idea of the type system is to associate two security levels (L or H) with each program variable. The first level corresponds to non-speculative execution, the second to speculative execution. A register with type (L, L) indicates that its contents are public during both non-speculative and speculative execution, meaning the value is not dependent on any secret data and can safely leak. The type (L, H) signals that the contents of a register are public during non-speculative execution but may depend on secret data during speculative execution. In this case, the variable must be protected with a **dfence** before its value can be used as an unsafe operand. If a variable has type (H, H), its value is considered secret-dependent in both execution contexts and must not leak.

Program language. We formalize our type system over a simple imperative `while` language (Figure 1). Memory operations are modeled as accesses to non-overlapping fixed-size arrays. More precisely, we model memory as a collection of arrays $\mathsf{a} \in \mathsf{Arr}$, each one associated with a length, $|\mathsf{a}|$. This approach supports the design of a more precise type system, by associating each memory operation with the security level of the corresponding region.

Expr ∋ E, F ::=	v	values
	x	register
	op(E$_1$, …, E$_n$)	operation
Instr ∋ I, J ::=	x := E	assignment
	x := a[E]	memory load
	a[E] := F	memory store
	fence	regular fence
	dfence x	selective fence
	if E **then** P **else** Q **fi**	conditional
	while E **do** P **od**	while loop
	jmp E	indirect jump
Prg ∋ P, Q ::=	ϵ | I; P	programs

Figure 1: Syntax of the language. Here $\mathsf{v} \in \mathbb{V}$ is a value, $\mathsf{x} \in \mathsf{Reg}$ is a register variable and $\mathsf{a} \in \mathsf{Arr}$ is an array name.

Expressions $\mathsf{E} \in \mathsf{Expr}$ can either be a ground value $v \in \mathbb{V}$, a register $\mathsf{x} \in \mathsf{Reg}$, or the application of an n-ary operation $\mathsf{op} \in \mathsf{Ops}$ on expressions $\mathsf{E}_1, \ldots, \mathsf{E}_n$.

Programs $\mathsf{P} \in \mathsf{Prg}$ are sequences of instructions. The assignment x := E updates the register x with the value of E. The load instruction x := a[E] evaluates E to an offset i and loads the i-th entry of the array a in the register x. The store instruction a[E] := F evaluates F and writes its value to the array location a[E]. We assume that our programs enjoy basic memory safety properties: the offsets of arrays always evaluate to natural numbers $n \in \mathbb{N}$, and $0 \leq n \leq |\mathsf{a}|$ always holds in non-speculative execution. Finally, our language features structured control flow instructions, the full speculation barrier **fence**, and the fine-grained barrier **dfence**.

For brevity's sake, we omit the operational semantics of the language, which can be found in our extended paper [37].

Types and environments. Now that we introduced the program language targeted by our type system, we introduce its types and typing environments. Our *ground types* describe the confidentiality levels of registers and arrays. A variable is assigned type L if it contains public data, or H if it may contain secret data. These ground types form a security lattice with the ordering $\mathsf{L} \leq \mathsf{H}$, indicating that public data can always be treated as secret if necessary. A type environment Γ maps registers $\mathsf{x} \in \mathsf{Reg}$ to pairs $\sigma = (\tau_1, \tau_2)$, where τ_1 and τ_2 represent the security level of x during non-speculative and speculative execution, respectively. Additionally, Γ maps arrays $\mathsf{a} \in \mathsf{Arr}$ to a single security level τ, corresponding to the security level of the array's contents (i.e., its cells) during non-speculative execution. Speculative types for arrays are not tracked because the type system assumes that values loaded speculatively from memory are always secret (H). For two given pairs $\sigma = (\tau_1, \tau_2)$ and $\sigma' = (\tau_1', \tau_2')$ we say that $\sigma \leq \sigma'$ whenever $\tau_1 \leq \tau_1'$ and $\tau_2 \leq \tau_2'$. We

stipulate that two environments satisfy $\Gamma \leq \Sigma$, if and only if for every array or register variable z, we have $\Gamma(z) \leq \Sigma(z)$.

Judgments. The type system establishes judgments of the form $\Gamma \vdash \mathsf{P} : \Sigma$, which expresses that executing the program P in a state of type Γ transitions to a state of type Σ. Moreover, the judgment guarantees that all values leaked during the execution of P are public. To deal with indirect jumps, our type system internally uses *auxiliary judgments* such as $\Phi \mid \Gamma \vdash \mathsf{P} : \Sigma$, where the *jump context* Φ is a set of judgments of the form $\Gamma \vdash \mathsf{P} : \Sigma$. Intuitively, the context records previously established typings for programs that may be reached through indirect jumps. Ordinary judgments are then defined as auxiliary judgments with empty jump context $\emptyset$. That is, we write $\Gamma \vdash \mathsf{P} : \Sigma$ as shorthand for $\emptyset \mid \Gamma \vdash \mathsf{P} : \Sigma$. Jump contexts support compositional reasoning about indirect jumps: when typing an indirect jump, the type system can look up the typing of the target continuation in Φ instead of recomputing it. This mechanism also allows the type system to handle programs with cyclic patterns of indirect jumps.

Typing rules. The typing rules are detailed in Figure 2. The rules for expressions propagate the security levels of the registers to the resulting value of the expression, producing judgments of the form $\Gamma \vdash \mathsf{E} : \sigma$. This judgment indicates that evaluating the expression E in a state of type Γ will produce a value of type σ. Rule [TSub] reflects the principle that public data can always be treated as secret. For simplicity's sake, we assume that applying an operator op to a sequence of sub-expressions takes the same amount of time regardless of the values involved, so that no information can leak through the timing of expression evaluation itself. This assumption can be easily relaxed by imposing that arguments of expressions with data-dependent timing have type (L, L).

The remaining rules handle the typing of programs, ensuring that the types of all variables are consistently tracked at each program point. Rule [TAsgn] updates the type of the assigned variable x to match the type of the expression E. For control flow instructions, the rules follow standard patterns ([TIf], [TWhile], [TEmpty], [TSeq]). The condition $\Phi \mid \Gamma \vdash \mathsf{E} : (\mathsf{L}, \mathsf{L})$ in rules [TIf] and [TWhile] ensures that the value leaked by the conditional instruction is public. [TWhile] imposes that the environment is a fixed point: the output environment must be identical to the input environment. This can be achieved using the weakening rule [TWeak].

The operational semantics [37] handles indirect jumps by means of a *continuation map* ϕ, associating each program location ℓ with the corresponding continuation $\phi(\ell)$. Rule [TJmp] types indirect jump instructions by assuming that these are annotated with a set V of possible jump targets under normal and speculative execution. The rule requires all the jump continuations $\phi(\ell)$ for $\ell \in V$ to be typable, described by the triple $(\Gamma \vdash \phi(\ell) : \Sigma) \in \Phi$. This rule is only significant in combination with Rule [TCont]. When typing a program P, this rule allows populating a jump context with a collection Φ of auxiliary judgments $\Gamma' \vdash \mathsf{Q} : \Sigma'$. In the premise on the left, the validity of each judgment in Φ is established under the assumption that the type of jump targets can be retrieved from Φ. Once the validity of judgments in Φ is established, these judgments can be used to type P in the right premise.

The most significant rules correspond to loads, stores, **fence**, and **dfence**. Rule [TLoad] ensures that the array's offset has type (L, L) to prevent leakage due to using secrets as memory addresses. It also enforces that the new type of register x is $(\Gamma(a), \mathsf{H})$. In other words, the type system uses the non-speculative type of a for the register x in non-speculative execution and H during speculation. This approach is justified by several factors; because of Spectre-PHT, the array access can overflow, which means the loaded value might be secret. Even if the access is within bounds, PSF speculation allows for loading any value from the store buffer. Consequently, the type system conservatively approximates the speculative type to H. Rule [TStore] similarly requires that the offset has type (L, L). Since only a single array entry is modified, the new type of the array is the least upper bound of the existing element types and the updated value, i.e., $\max(\pi_1(\sigma_{\mathsf{F}}),\ \Gamma(\mathsf{a}))$, where π_1 is the first projection of the pair of types (i.e., the non-speculative type of F).

The last two rules account for instructions that (partially) stop speculation. Rule [TFence] sets the speculative type of all register variables to be equal to their non-speculative type. This rule is sound because the **fence** instruction and any subsequent instructions can only be executed when the state is not in misspeculation. Array types are unchanged because our type system does not keep track of their speculative type. Rule [TDfence] is similar to Rule [TFence], but it only updates the protected variable's speculative type.

Theorem 1 states the main property of our type system, i.e., that well-typed programs do not leak confidential data. This is captured by the notion of $\cong_\Gamma$-SCT, stating that executing P on states that store the same values in public memory regions—i.e., $\cong_\Gamma$-related—yields identical leakage. In the relation $\cong_\Gamma$, the environment Γ specifies which parts of the state (register variables or arrays) are considered secret and which of them are public. For brevity's sake, we omit the complete definitions of $\cong_\Gamma$ and SCT, which are given in [37].

Theorem 1 (Correctness of the type system). *If* $\Gamma \vdash \mathsf{P} : \Sigma$ *then* P *is* $\cong_\Gamma$*-SCT.*

Compilation. Note that Theorem 1 guarantees that typable programs are speculative constant-time only *at the source level*. This guarantee will translate to binaries only if the compiler preserves the notion of speculative constant-time. Proving that secure compilers like Jasmin preserve speculative constant-time is still an active research topic [9]. In particular, to preserve speculative constant-time, it is crucial that the compiler does not introduce new memory operations (as often happens when register spilling is required). If spilling is necessary, the programmer (or a policy-aware compiler) should handle it by storing or loading registers into an array, as done in the Jasmin language. Alternatively, the type system should be applied after the memory operations are introduced.

Example of the type system on a Spectre v1.1 gadget. Spectre v1.1 is a class of transient execution attacks which relies on transient out-of-bounds store operations and store-to-load forwarding to leak confidential data in a victim program [59], as demonstrated in Listing 4. In the example, `addr_of_line_2` is a constant that corresponds to the address of the unsafe instruction at line 2. An attacker can induce transient execution of the conditional instruction with `i = 16`, overlapping `a[i]` with the return address stored in `sp[0]`. As a result, the out-of-bounds store at line 5 speculatively forwards `addr_of_line_2` to the return instruction at line 8, redirecting the control flow to line 2, where the value of x is leaked.

Typing rules for expressions:

$$\textsc{TVar}\ \frac{}{\Gamma \vdash \mathsf{x} : \Gamma(x)} \qquad \textsc{TVal}\ \frac{}{\Gamma \vdash \mathsf{v} : (\mathsf{L},\mathsf{L})} \qquad \textsc{TOp}\ \frac{\Gamma \vdash \mathsf{E}_i : \sigma}{\Gamma \vdash \mathsf{op}(\mathsf{E}_1,\ldots,\mathsf{E}_k) : \sigma} \qquad \textsc{TSub}\ \frac{\Gamma \vdash E : \sigma \quad \sigma \le \sigma'}{\Gamma \vdash E : \sigma'}$$

Typing rules for instructions:

$$\textsc{TAsgn}\ \frac{\Gamma \vdash \mathsf{E} : \sigma}{\Phi \mid \Gamma \vdash \mathsf{x} := \mathsf{E} : \Gamma[\mathsf{x} \leftarrow \sigma]} \qquad \textsc{TLoad}\ \frac{\Gamma \vdash \mathsf{E} : (\mathsf{L},\mathsf{L}) \quad \sigma = (\Gamma(\mathsf{a}),\mathsf{H})}{\Phi \mid \Gamma \vdash \mathsf{x} := \mathsf{a}[\mathsf{E}] : \Gamma[\mathsf{x} \leftarrow \sigma]}$$

$$\textsc{TStore}\ \frac{\Gamma \vdash \mathsf{E} : (\mathsf{L},\mathsf{L}) \quad \Gamma \vdash \mathsf{F} : \sigma_{\mathsf{F}}}{\Phi \mid \Gamma \vdash \mathsf{a}[\mathsf{E}] := \mathsf{F} : \Gamma[\mathsf{a} \leftarrow \max(\pi_1(\sigma_{\mathsf{F}}),\ \Gamma(\mathsf{a}))]}$$

$$\textsc{TIf}\ \frac{\Gamma \vdash \mathsf{E} : (\mathsf{L},\mathsf{L}) \quad \Phi \mid \Gamma \vdash \mathsf{P}_1 : \Gamma' \quad \Phi \mid \Gamma \vdash \mathsf{P}_2 : \Gamma'}{\Phi \mid \Gamma \vdash \mathbf{if}\ \mathsf{E}\ \mathbf{then}\ \mathsf{P}_1\ \mathbf{else}\ \mathsf{P}_2\ \mathbf{fi} : \Gamma'} \qquad \textsc{TWhile}\ \frac{\Gamma \vdash \mathsf{E} : (\mathsf{L},\mathsf{L}) \quad \Phi \mid \Gamma \vdash \mathsf{P} : \Gamma}{\Phi \mid \Gamma \vdash \mathbf{while}\ \mathsf{E}\ \mathbf{do}\ \mathsf{P}\ \mathbf{od} : \Gamma}$$

$$\textsc{TJmp}\ \frac{\Gamma \vdash \mathsf{E} : (\mathsf{L},\mathsf{L}) \quad \forall \ell \in V.\ (\Gamma \vdash \phi(\ell) : \Sigma_\ell) \in \Phi}{\Phi \mid \Gamma \vdash \mathbf{jmp}\ \mathsf{E}_V : \bigsqcup_{\ell \in V} \Sigma_\ell}$$

$$\textsc{TFence}\ \frac{\forall \mathsf{x} \in \mathrm{Reg}.\ \Sigma(\mathsf{x}) = (\pi_1(\Gamma(\mathsf{x})), \pi_1(\Gamma(\mathsf{x}))) \quad \forall \mathsf{a} \in \mathrm{Arr}.\ \Sigma(\mathsf{a}) = \Gamma(\mathsf{a})}{\Phi \mid \Gamma \vdash \mathbf{fence} : \Sigma} \qquad \textsc{TDfence}\ \frac{\sigma = (\pi_1(\Gamma(\mathsf{x})), \pi_1(\Gamma(\mathsf{x})))}{\Phi \mid \Gamma \vdash \mathbf{dfence}\mathsf{x} : \Gamma[\mathsf{x} \leftarrow \sigma]}$$

Typing rules for programs:

$$\textsc{TEmpty}\ \frac{}{\Phi \mid \Gamma \vdash \epsilon : \Gamma} \qquad \textsc{TSeq}\ \frac{\Phi \mid \Gamma_1 \vdash \mathsf{I} : \Gamma_2 \quad \Phi \mid \Gamma_2 \vdash \mathsf{P}_2 : \Gamma_3}{\Phi \mid \Gamma_1 \vdash \mathsf{I}; \mathsf{P}_2 : \Gamma_3} \qquad \textsc{TWeak}\ \frac{\Phi \mid \Gamma' \vdash \mathsf{P} : \Sigma' \quad \Gamma \le \Gamma' \quad \Sigma' \le \Sigma}{\Phi \mid \Gamma \vdash \mathsf{P} : \Sigma}$$

$$\textsc{TCont}\ \frac{\forall\, (\Gamma' \vdash \mathsf{Q} : \Sigma') \in \Phi.\ \Phi \mid \Gamma' \vdash \mathsf{Q} : \Sigma' \quad \Phi \mid \Gamma \vdash \mathsf{P} : \Sigma}{\emptyset \mid \Gamma \vdash \mathsf{P} : \Sigma}$$

Figure 2: Type system.

```
  0 - 15: a[16]
 16 - 271: sp[256]
```

(a) Memory

```
1  x := 0;          // x : (L, L)
2  y := a[x];       // vulnerable instruction
3  x := secret;     // x : (H, H)
4  if (in_bounds(a, i)){
5    a[i] := addr_of_line_2;
6  }
7  ra := sp[0];     // load return address
8  jmp ra;          // return
```

(b) Spectre-v1.1

Listing 4: Snippet vulnerable to Spectre v1.1. In RISC-V, returns are encoded as unconditional jumps to the address held in the `ra` register.

In our type system, programs vulnerable to Spectre v1.1 are not typable. Indeed, for typing the program in Listing 4, the rule [TJmp] for the jump instruction in line 9 requires each target of the jump instruction to typable. Notice that the program starting in line 2 is reachable, as the out-of-bounds write instruction may transiently overwrite the stack pointer. Consequently, by rule [TJmp], the instruction at line 2 must be typable under the assumption that x has type (H, H), which is not the case.

It is important to note that our type system can only protect against Spectre v1.1 when jump targets can be statically over-approximated. In Jasmin, this requirement is not problematic, since the only form of indirect jumps it supports is return instructions. During normal execution, the set of return targets corresponds to the instructions immediately following each call site. Thus, if all return instructions in a program are preceded by a **dfence** instruction protecting the register that holds the return address, then in the absence of return address speculation the set of possible jump targets V remains the same under both speculative and normal execution. See [37] for a detailed proof of this fact. This also means that Jasmin programs for RISC-V can be protected against Spectre v1.1 by simply replacing every return instruction with `dfence ra; jmp ra`.

Software implementation. We implemented our type system in the Jasmin framework. To this end, we integrated the **dfence** instruction into its language and extended Jasmin's RISC-V compiler to emit the corresponding assembly instruction. To compare **dfence** with Jasmin's manual SLH mechanism, we extended the compiler with support for SLH and **fence** instructions in RISC-V, which were only supported for the x86 architecture.

The primary benefit of our approach is that it enables generating multiple versions of the same cryptographic primitives, each protected by a different mitigation mechanism (e.g., SLH, **dfence**, or **fence**), as demonstrated in §6.

5 Hardware Implementation

To assess the feasibility of **dfence** in real hardware and evaluate its performance impact, we implement it as an extension of the Proteus [21] RISC-V CPU, which has already been used for evaluating Spectre countermeasures [36]. We extend version 25.08-O of Proteus featuring out-of-order execution and speculation primitives. This CPU features both SSB- and PSF-type memory disambiguation speculation, and a branch target buffer (BTB) that can induce both Spectre-PHT and -BTB type speculation.

5.1 Speculative Execution on Proteus

Proteus features a reorder buffer (ROB) with a configurable number of entries and register renaming. Execution units are managed by reservation stations that wait for unavailable operands, enforcing data dependency resolution. Load operations are issued to dedicated load units after their target addresses have been computed. Instructions in the ROB are retired in program order when ready, and stores are committed to memory during this retirement phase.

PHT speculation. Proteus includes a BTB for control-flow speculation. The BTB is a table that maps partial instruction addresses—independent of instruction type—to predicted target addresses. Additionally, Proteus supports straight-line speculation. These mechanisms are employed in the instruction fetch stage to predict the next instructions to fetch and execute.

Note that the control-flow speculation implemented in Proteus is more permissive than what is assumed in our threat model and formal semantics (§4), which only consider PHT-based speculation on conditional branches. As a result, achieving full security on Proteus requires either complementing our defense with an orthogonal BTB-specific mitigation that restricts speculation to PHT-style behavior, or extending our defense as discussed in §9.1. Nevertheless, this broader speculation model does not undermine our security claims. Our evaluation (§6.1) focuses on PHT-style leakage and demonstrates that our defense effectively mitigates these.

SSB and PSF speculation. Proteus supports both SSB and PSF speculation. In the case of SSB, a load may execute speculatively even in the presence of an older store with an unresolved address. Once the store instruction retires, a check is performed to detect any address aliasing. If the addresses alias, the store was incorrectly bypassed, and the pipeline is flushed. The PSF implementation speculatively forwards the value from the most recent store to subsequent loads with unresolved addresses. Once the load address is computed, the actual load address is compared with the address of the store. If a mismatch is detected, the pipeline is flushed. These predictors are also equipped with a small history table to avoid repeated mispredictions on the same addresses, similar to how similar predictors can be dynamically disabled in Intel processors [39].

5.2 RISC-V Encoding

The standard RISC-V **`fence`** instructions are encoded as register-immediate (I-type) instructions with separate `fm` identifiers for `fence` and `fence.tso`, and two—currently unused—register identifiers (`rd` and `rs1`). For **`dfence`**, we choose a new `fm` identifier and refine the semantics to use two registers instead of one. Concretely, **`dfence`** `rd, rs1` introduces a dependency between `rd` and `rs1`. Instructions using the value of the protected `rd` will be delayed until the fence forwards its value. Therefore, a register `x` can be protected using **`dfence`** `x, x`. If a different register is used in `rs1`, the **`dfence`** instruction copies the value of `rs1` into `rd` upon retirement. In this case, only instructions that use `rd` (and not `rs1`) are blocked from transiently using the value. This encoding simplifies the hardware changes, as it enables reusing existing dependency tracking mechanisms. We also extended the LLVM RISC-V assembler and disassembler with this encoding to simplify writing code.

5.3 Speculative Fence Implementations

We implement different speculative execution barriers that comply with our specification (§3). These implementations offer the same security, but different trade-offs in terms of performance and hardware costs, which we evaluate in §6.2 and §6.4, respectively.

Serializing fence. As a baseline, we implemented **`fence`** as a fully serializing instruction, analogous to **`lfence`** on x86. All subsequent instructions are blocked from executing until the fence retires. This implementation is stricter than the RISC-V specification, which only specifies ordering among certain I/O and memory operations. This limited ordering would not prevent leakage through non-memory side channels.

Unoptimized **`dfence`**. We first implement a **`dfence`** variant that does not rely on hardware speculation tracking. In this design, the value of the destination register is not forwarded until the **`dfence`** instruction retires, regardless of whether the processor is executing speculatively. This approach simplifies the implementation by eliminating the need for tracking speculation while still complying with our specification, at a potential performance penalty.

Optimized **`dfence`**. For the optimized **`dfence`** variant, we reuse existing facilities in Proteus for speculation tracking. Proteus taints instructions executing speculatively, regardless of the source of speculation (BTB, unresolved control-flow instructions, SSB, and PSF). These taints propagate over the pipeline and are recorded in the ROB. When an instruction uses the result of a tainted instruction, it also becomes tainted. Taints in the ROB are cleared upon instruction retirement or a pipeline flush. A **`dfence`** instruction is blocked in its reservation station before it can start executing if it follows a speculative branch or if its source register is tainted. It is only allowed to execute—and forward the register value—once all prior speculation has been resolved. This ensures that instructions following the **`dfence`** depending on its output value also cannot start executing (and potentially leak their operands) until the **`dfence`** is retired.

6 Evaluation

We evaluate our **`dfence`** implementation in multiple aspects: security, runtime overhead compared to other state-of-the-art protections, software instrumentation complexity, and hardware cost. Our security and performance evaluations are conducted in the cycle-accurate Proteus simulator.

6.1 Security Evaluation

First, we perform a security evaluation to increase our confidence in the correctness of the implementation and validate that, with the correct placement of **`dfence`** instructions, secrets do not leak through microarchitectural channels. To this end, we develop a template to generate vulnerable code snippets and protect them using various defense strategies. Specifically, our test cases cover: (1) four speculation strategies (SSB, PSF, and two variants of PHT), (2) leakage through five different insecure instructions (load, store, branch, jump, division), and (3) instrumentation using secure instructions (**`fence`**, **`dfence`**), and insecure strategies (no protection,

or dfence with an incorrect register). This template generates a total of 40 secure and 40 insecure programs.

Program executions on Proteus produce an output with the time-ordered value changes of the processor's signals, which can be used to detect security violations [36, 86, 87]. We conduct our security evaluation by monitoring two sets of signals: a *conservative* set following an allowlist approach and a *liberal* set following a blocklist approach. The liberal set includes manually selected observable signals (e.g., cache metadata, memory bus addresses, program counter), while the conservative set includes all signals except those known not to be observable (e.g., register and cache data).

We run each program twice, using different values for the secret input. Each pair of executions with different secrets (s_1, s_2) produces a pair of signals sets $(\text{signal}(s_1), \text{signal}(s_2))$. If $\text{signal}(s_1) \neq \text{signal}(s_2)$, the program leaks; if $\text{signal}(s_1) = \text{signal}(s_2)$, the program does not leak (for our specific inputs). Our goal is to demonstrate that (**G1**) *most* insecure programs leak on Proteus, and (**G2**) our fence and dfence protections effectively close the leaks for *all* programs using a secure strategy.

Results. (**G1**): With both signal sets, 20/40 insecure programs were found to leak in practice. This confirms that these programs are insecure on Proteus and that our security evaluation can accurately identify insecure programs. (**G2**:) All 40 programs were found to be secure with both signal sets, providing evidence that our protections effectively close the leaks for all secure programs.

6.2 Performance Evaluation

We evaluate the runtime overhead of dfence by comparing it to an unprotected baseline and standard protections for Spectre-PHT and Spectre-STL. As benchmark programs, we use cryptographic algorithms distributed as part of the `libjade` library [33–35]. In addition to the existing ML-DSA (Dilithium) [40] code, we ported the Jasmin sources of ML-KEM (Kyber) [22], Curve25519 [14], Keccak$f$-1600 [18], ChaCha20 [15], Poly1305 [13] and Gimli [16] from Arm to 32-bit RISC-V. Notably, Dilithium and Kyber are NIST standards for post-quantum cryptography since 2024.

Methodology. Our tests are conducted on the dfence implementation built on Proteus, as described in §5. Our benchmarks execute the aforementioned cryptographic primitives on Proteus under various configurations:

- *Baseline.* The cryptographic primitives are executed without applying protection mechanisms or disabling speculative execution features.
- dfence. The cryptographic primitives are protected exclusively using dfence, validated by our Jasmin type system. These benchmarks are executed on both the unoptimized and optimized implementations from §5.3. To measure the impact of defending against Spectre v1.1 with dfence, we also consider a configuration where return instructions are not protected with dfence, named dfence *without v1.1*.
- *Regular* fence. The cryptographic primitives are protected solely using regular serializing fence instructions and are validated by our type system. In particular Spectre v1.1 is prevented by replacing RISC-V `ret` instructions with `fence; ret`. We evaluate the impact protecting against Spectre v1.1 by also considering a variant of this configuration where return instructions are not protected with fence, named *regular* fence *without v1.1*.
- *SLH and SSBD.* Jasmin's existing type system is used to insert dependencies between the guards of conditionals and the potentially leaked values, following the SLH approach. To achieve security comparable to the previous two configurations, we disable SSB and PSF to prevent leaks via these prediction mechanisms. Additionally, we report on the isolated impact of SLH and SSBD. Notably, this configuration does not prevent Spectre v1.1 attacks that are not covered by the threat model of Jasmin's existing type system.

Table 1: Geometric mean and maximum overhead measured over the evaluated benchmarks.

Configuration	Geometric mean	Maximum
Optimized dfence	-0.23%	1.76%
Optimized dfence w/o v1.1	0.13%	2.52%
Unoptimized dfence	-0.19%	1.79%
Unoptimized dfence w/o v1.1	0.28%	3.55%
Regular fence	1.67%	4.39%
Regular fence w/o v1.1	1.85%	4.73%
SLH and SSBD	13.53%	59.72%
SLH only	2.45%	9.28%
SSBD only	12.24%	56.96%

Our benchmarks were executed on a setup of Proteus with 48 ROB entries, 12 ALUs, 3 MUL/DIV units, and 3 load units with a 64-cycle memory load delay.

6.2.1 Results. The results of our performance evaluation are summarized in Table 1. Individual measurements are included in our extended paper [37]. Overall, the experiments show that the overhead of dfence consistently low. In particular, the overhead induced by dfence to prevent Spectre v1.1 is negative on average for both the variants. Negative overheads have also been reported for other hardware-based protection mechanisms that partially restrict speculative execution to prevent leakage [46, 56, 77]. We hypothesize that in our case, these negative overheads result from dfence limiting the incorrect propagation of values during transient execution. This behavior may reduce the cost associated with squashing transient instructions compared to the baseline.

Our results also show that the unoptimized implementation of dfence shows a good trade-off between implementation cost and performance overhead: while its performance overhead is greater than the one of the optimized dfence, it is still negligible and its hardware cost is smaller (see §6.4).

Finally, observe that disabling SSB and PSF speculation on Proteus brings an average performance cost of over 10%. This cost is unavoidable when using SLH to still provide protection against Spectre-STL, which is offered by dfence out of the box.

6.3 Software Instrumentation Cost

Table 2 reports the number of lines modified to obtain secure implementations to measure the complexity involved in instrumenting the different countermeasures. The effort required to implement

Table 2: Line additions (Lines) and protection sites (Prot) required to protect our benchmarks. Percentages are relative to the total LOC of each benchmark.

		dfence		fence		SLH	
Benchmark	LOC	Lines	Prot	Lines	Prot	Lines	Prot
ML-DSA	10,755	171 (1.6%)	160 (1.5%)	93 (0.9%)	91 (0.8%)	1786 (16.6%)	149 (1.4%)
ML-KEM	2,665	98 (3.7%)	51 (1.9%)	63 (2.4%)	45 (1.7%)	634 (23.8%)	55 (2.1%)
Curve25519	443	3 (0.7%)	3 (0.7%)	2 (0.5%)	1 (0.2%)	2 (0.5%)	1 (0.2%)
ChaCha20	369	19 (5.1%)	10 (2.7%)	10 (2.7%)	2 (0.5%)	14 (3.8%)	2 (0.5%)
Keccakf1600	384	43 (11.2%)	8 (2.1%)	45 (11.7%)	6 (1.6%)	76 (19.8%)	6 (1.6%)
Poly1305	195	5 (2.6%)	4 (2.1%)	2 (1.0%)	1 (0.5%)	2 (1.0%)	1 (0.5%)
Gimli	87	2 (2.3%)	1 (1.1%)	2 (2.3%)	1 (1.1%)	2 (2.3%)	1 (1.1%)

protections using **dfence** or **fence** is comparable, and is approximately one 1 line change per protection site, with fluctuations (see e.g. on Keccakf1600), due to orthogonal aesthetics changes in the source files. From our measurement, it is also evident that the integration cost for SLH is significantly higher, primarily due to maintaining a misspeculation flag throughout the execution and updating it at every branching instruction and passing it to and from functions. This flag must also remain in a register throughout the execution. In practice, this means that the compiler must treat the flag as a special variable that cannot be temporarily spilled to the stack. This constraint also has the side effect of increasing register pressure. This is less of an issue for RISC-V due to its large register file, but it can be more problematic on architectures like the ARM Cortex-M, which have fewer available registers.

6.4 Hardware Cost

We also perform measurements to determine the cost of implementing **dfence** in hardware using the EVAL-HD workflow [38]. Implementing the baseline core with the configuration of §6.2 uses a chip area of 0.7856 mm^2. The unoptimized **dfence** implementation measures 0.7872 mm^2 (+0.2%), while the optimized dfence with speculation tracking is 0.8053 mm^2 (+2.5%). The critical path remains unchanged for all variants at 17.13 ns.

7 Comparison with Blade

Blade [82] introduces a **protect** primitive designed to serve as a fine-grained speculation barrier, with semantics similar to our **dfence**, and two software implementations. The first is a software-level instrumentation, which ensures that array accesses remain in bounds, putting Spectre-STL out of scope. The second one compiles it to **lfence**, resulting in a larger performance overhead [82].

Our work overcomes these limitations by providing two hardware implementations of **dfence**, capable of protecting against multiple forms of speculation while incurring low instrumentation and performance cost. Overall, compared to Blade's **protect**, **dfence** brings several benefits in terms of security and control over the placement of protections, that we discuss in the following.

Guarantees of the type systems. Although the type systems of Blade and **dfence** both ensure the correct placement of protections, they provide different security guarantees. The type system of **dfence** only accepts programs that are speculative constant-time. In contrast, the programs accepted by Blade's type system are speculative constant-time only if they comply to a *static* notion of constant-time. More precisely, this notion is an *under-approximation* of constant-time, defined as a typability condition under a second type system which relies on security annotations. Therefore, while Blade's authors claim to *"eliminate speculation-based leakage in unannotated cryptographic code"*, this is not exactly the case. More precisely, using Blade on unannotated programs that do not comply with the above-mentioned static notion of constant-time does not guarantee their security at run time. To illustrate this, consider the following example:

```
1 if (E) { y := a[x];}
```

Listing 5: Example of program that is typable by Blade but not by dfence.

In this program, we assume that the guard `E` always evaluates to `false` when `x` contains secret data. Thus, the program is constant-time. Since `x` is a register variable, its value is not loaded from memory, and therefore the program is accepted by Blade's type system. Consequently, Blade does not insert any **protect** instruction on this program. However, due to direct branch speculation, the body of the conditional may be transiently executed with `x` holding secret data that leaks[2]. With **dfence**'s type system, `x` would have the type `H` in normal execution, as it may store secret data. Therefore, this vulnerable program would be directly rejected.

Blade's SLH. The main difference between Blade's SLH implementation and other forms of SLH—like the original LLVM version and several variants [7, 73, 91]—is that Blade's SLH does not keep track of the misspeculation flag in a dedicated register. Therefore, it cannot selectively mask values based on the speculative state of the processor. Instead, it masks arrays accesses based on whether the index is in bounds or not. Consider the following program:

```
1 if (E) {
2   x := protect(a[i]);
3   y := b[x];
4 }
```

Listing 6: Blade's `protect` to prevent a Spectre-v1 vulnerability.

Using Blade's variant of SLH, the **protect** instruction is replaced with a form of non-speculative index masking:

```
1 if (E) {
2   mask := in_bounds(a, i);
3   mask := mask ? 0xFFFF : 0x0000;
4   x := *((a + i) & mask);
5   y := b[x];
6 }
```

Listing 7: Result of the compilation of Listing 6 using Blade's SLH.

Line 2 checks whether the index `i` is in bounds for `a`. Then, the *non-speculative* conditional assignment at line 3 sets the variable `mask` to `0x0000` when `i` is out of bounds, and to `0xFFFF` otherwise. The load address is then masked at line 4 to prevent transient out-of-bounds accesses.

[2]This example does not invalidate the soundness of Blade's type system, as the program in consideration does not comply with Blade's static notion of constant-time. However, it shows that to safely use Blade, one should ideally first verify that the program complies with this notion of constant-time via a second type system.

First limitation: the need for array bounds. Blade's SLH is only effective when precise array bounds are available. For instance, if loose bounds are used in Listing 7 at line 2, Blade's SLH does not prevent the speculative out-of-bounds access at line 4, leading to potential leakage of confidential data at line 5. In contrast to Blade's SLH, **dfence** protects *values* rather than *accesses*, therefore it does not require array bounds information. As an example, Listing 6 can be protected with **dfence** as follows:

```
1 if (E) {
2   x := a[i];
3   dfence x;
4   y := b[x];
5 }
```

Listing 8: Program in Listing 6 protected with dfence.

Here, the **dfence** instruction at line 3 prevents the transient execution of the load at line 4, thereby preventing the leakage of secret data originating from out-of-bounds memory loads at line 2.

Second limitation: specificity to out-of-bounds accesses. As Blade's SLH performs index masking, it can only prevent attacks that exploit out-of-bounds accesses. To see this, assume that the condition E of Listing 7 always evaluates to false and that the array a contains secret data. Under these assumptions, the program in Listing 7 trivially does not leak secret data in normal execution—i.e., it is constant-time—but it can leak secret data during transient execution. More precisely, it is sufficient to trigger the transient execution of the conditional's body with any value of i within the bounds of a to cause the leakage of confidential information at line 5.

To see how **dfence** can prevent this issue, assume the same scenario in Listing 8, i.e., E always evaluates to false and that a contains secret data. Even if an attacker triggers the transient execution of the conditional's body, the leaking instruction at line 4 would not be executed because **dfence** would withhold the value of x.

Protection against Spectre v1.1. Blade can mitigate the Spectre v1.1 data variant [82], which leaks confidential data that is stored during transient execution by forwarding it to an aliasing load that would otherwise access only public data during normal execution. However, the variant of Spectre v1.1 illustrated in Listing 4, corresponding to the original formulation of v1.1 [59], is not part of Blade's threat model because Blade's type system does not model indirect jumps or return instructions, and consequently it imposes no restrictions on jump targets. With **dfence**, we can prevent both variants with a negligible overhead (§6). Using Blade, the geometric mean overhead for preventing Spectre v1.1 is more than 15% [82].

8 Related Work

We focus the related work on software and hardware defenses that are suitable at least for Spectre-PHT or -STL.

Speculation barriers in hardware. Commercial ISAs include speculation barrier instructions designed for various purposes [54, 55, 65], primarily to address concurrency issues and ensure program correctness. Intel and AMD recommend using fences—alongside other techniques such as retpolines—to mitigate speculative vulnerabilities [6, 48, 50, 51]. In contrast to **dfence**, which discerns what to block, traditional fences block all forms of speculation, introducing a performance overhead. Another important caveat with fence-based approaches is that the programmer is often responsible for determining where to insert fences, possibly leading to unnecessary slowdowns or security vulnerabilities. Our type system (§4) is of independent interest from **dfence**, as it can be used as an automatic verification tool for the smart insertion of all speculation barriers to effectively protect against Spectre-PHT and STL (as we have used it for our evaluation in §6).

Building on speculation barriers, multiple academic proposals emerged, introducing fence-like instructions with stronger security guarantees (including this work). Context-sensitive fences [81] are inserted dynamically by the CPU using decoder-level dynamic information flow tracking, reducing the performance overhead compared to **lfence**s. In contrast to their work, our approach requires fewer hardware modifications. Temporal fences (`fence.t`) [88] are instructions that flush the microarchitectural state, effectively preventing secret leakage with only a 2% performance overhead. However, this protection does not address end-to-end timing and same-address-space attacks.

Concurrent to our work, Herinomena et al. [8] designed and implemented a RISC-V instruction similar to **dfence**. Their proposed instruction, **fence.spec** rd, rs1, establishes a dependency between registers rd and rs1, and "*completes execution only in a non-speculative state, i.e. when the core is certain that it will commit*". To mitigate speculative execution attacks, the authors developed a compiler pass that can insert **fence.spec** instructions before and/or after every load and/or store. Due to the coarse-grained nature of these insertion policies, their performance evaluation reports overheads exceeding 100% in the most conservative policies. Based on these results, the authors conclude that "*there is no way to use them [**fence.spec** instructions] correctly and efficiently, which significantly hinders their practical adoption*". Our results challenge this conclusion by demonstrating that, by using a more fine-grained protection strategy, our **dfence** implementation can prevent speculative attacks based on PHT, STL, and PSF speculation with negligible overhead.

Software-based defenses. The state-of-the-art software-only mitigations for Spectre-PHT are index-masking and SLH, both recommended by CPU vendors [66]. Academic works have observed that prior SLH techniques were insecure, and hence, proposed extensions of SLH—backed-up by formal guarantees—that still offer better performance than **lfence**s [7, 11, 91]. In particular, Shivakumar et al. [7] define a type system to verify that protections based on an optimized form of SLH are correctly applied at the source level. Their type system only covers PHT speculation, in which well-typed programs must explicitly track the value of the misspeculation predicate in a dedicated register, with the type system ensuring that this register is updated correctly after each branching instruction. In comparison, our type system does not impose this requirement, as the **dfence** instruction operates independently of any software-level misspeculation predicate, and we also cover SSB and PSF speculation and indirect jumps. Olmos et al. [9] focus on ensuring that compilers preserve the speculative constant-time property, tackling a critical aspect of compiler interference with

security properties. Other works have also evaluated the performance impact of deployed mitigations on real-world operating systems [23]. More recently, a provably secure software-only approach was proposed that mitigates Spectre-v1, -v2 and -v4 variants at a 2-7% performance overhead on commercial hardware [10].

Hardware-based defenses. Several works have proposed novel processor models [29, 30, 36, 85, 90] to defend against transient execution attacks. These works can significantly reduce the performance impact (sometimes even leading to performance gains [45]) and often achieve stronger security guarantees than fence-insertion techniques [36]. In particular, from the works using taint tracking, ProSpeCT [36] has been implemented in hardware. In comparison to our proposal, ProSpeCT has a larger hardware resource overhead, recently estimated at 20.87% on an older Proteus baseline [38]. Additionally, our approach necessitates fewer software changes: in contrast to inserting **dfence** instructions, the entire software stack must track secrets stored in secret memory areas with ProSpeCT. STT [90] and SPT [30] are other secure speculation mechanisms based on taint tracking that prevent the transient leakage of values that are considered *unsafe*. Although these mechanisms do not impose software modifications, they have a much higher runtime overhead and more expensive hardware changes than **dfence**.

Moreover, while taint-tracking mechanisms implement the protection logic in hardware, **dfence** fully delegates it to software. This delegation also provides more flexibility: in our design, the software can use **dfence** to implement different protection mechanisms (e.g., SCT or speculative sandboxing). This is typically not possible with hardware-level taint tracking.

Other recent work has proposed SpecLFB [29], implemented in the BOOM RISC-V processor with a hardware overhead of only +0.77% LUTs and +0.31% FFs. SpecLFB is limited to cache side-channel attacks, whereas **dfence** can also prevent other side channels such as operand-dependent timing for division or multiplication, or port contention. OISA [89] is a design also based on BOOM that facilitates writing constant-time (data-oblivious) code, also enforced during speculation, by introducing new leakage-free instructions to the ISA. While this design enforces strong security guarantees, it introduces significant changes to software (new ISA) and hardware, such as using oblivious RAM [80], and slows down execution by a factor of 3-40 depending on the configuration, making it impractical in many applications.

9 Discussion

In this section, we discuss extensions of **dfence** to other Spectre variants, other possible improvements, and generalizations of this work, such as the application of our type system to speculative sandboxing and automatic insertion mechanisms.

9.1 Extension to other Spectre Variants

Load address prediction. Load address prediction (LAP) has recently been documented in Apple processors and successfully exploited to leak secrets [58]. The processor can speculate on the address of a load and incorrectly forward secret data (including non-architecturally accessed data) to subsequent instructions. This speculation mechanism is similar to PSF and SSB in the sense that a load instruction can transiently forward stale or secret data. Consequently, **dfence** and our type system can address Spectre-LAP without any software modifications. On the hardware side, the only required modification is to extend our speculation tracking to include this new speculation source—which should be a simple extension of PSF and SSB tracking.

Control-flow prediction. **dfence** requires complementary defenses to secure programs in the presence of *arbitrary* control flow speculation, e.g., in the presence of BTB or RSB predictors. In these cases, even if all unsafe operands are protected by a **dfence**, the attacker can abuse speculative execution to redirect a victim's control flow to an arbitrary program location, bypassing the **dfence** instruction and leaking secrets.

A first solution to handle these forms of speculative execution is to replace indirect jumps and returns with conditional direct jumps. This solution has been shown to incur low overhead on cryptographic benchmarks [10]. A second solution is to complement **dfence** with existing hardware defenses for (speculative) control-flow integrity [1, 64]. Intel's Control-flow Enforcement Technology (CET) can restrict the potential targets of control-flow speculation to a set of well-defined landing pads. On a processor with such a defense, it would be possible to also handle control-flow speculation like Spectre-BTB or Spectre-RSB with **dfence**. In hardware, the speculation-tracking mechanism would need to capture these new speculation sources. In software, the semantics and type system would need to capture these extra speculative paths and ensure that unsafe operands are protected along them.

Value-based PSF and SSB resolution. In this paper, we assumed that PSF and SSB are purely based on address speculation and not on value speculation, which is consistent with the Proteus configuration we used. In theory, PSF and SSB could resolve speculation based on load *values*: only roll back the execution if the predicted load value does not match the actual value, regardless of the memory addresses. This creates a resolution-based timing channel [90] dependent on whether the load value and forwarded store values are equal. Such a value-based resolution mechanism would turn both loads and store values into unsafe operands, breaking the common assumption that load and store values do not leak, and requiring orthogonal defenses. We are not aware of any currently deployed processor featuring such a value-based resolution mechanism for SSB and PSF speculation.

Other value prediction. Value prediction [42, 68, 69] speculates on the value of instructions' operands. For example, Intel CPUs can speculate that the upper 64 bits of a 128-bit dividend are zero. Another example is load value prediction, which was recently exploited by FLOP [57] attacks. Whether **dfence** alone can address value-based speculation depends on which instruction is impacted. These predictions introduce resolution-based channels [90] dependent on whether the predicted value equals the actual value, making predicted operands unsafe. Thus, we need to adapt our leakage model to forbid passing secret values to unsafe operands. Adapting software to comply with the new leakage model might not be feasible—e.g., if load/store values become unsafe—in which case orthogonal mitigations are needed. For divisions, however, it is possible to consider

them as unsafe (many cryptographic implementations already do) and adapt our speculation model accordingly.

9.2 Possible Improvements

Type system. Our type system could be refined to reduce the number of protected operands. First, the type system could be equipped with a speculation window to avoid unnecessarily protecting registers outside this window. Second, depending on the target CPU, we could model memory disambiguation speculation more precisely to identify cases where loads can only access public values speculatively, thereby reducing the number of registers to protect. For example, in the code `fence; x := a[y]`, the write buffer is empty after the `fence`, so if `y` is a public in-bounds address, `x` can only contain public data; yet our current type system conservatively labels `x` as secret.

Hardware. Any conservative implementation of `dfence` w.r.t. its specification (§3) is secure, such as a serializing `fence` or the unoptimized `dfence` from §5. However, there is a performance trade-off associated with different implementations: more conservative approaches are likely cheaper in hardware, but may unnecessarily decrease instruction throughput. On the other hand, a more optimized implementation of `dfence x` could allow `x` to be forwarded to *safe* operands while blocking forwarding to *unsafe* operands. This approach has the advantage of enabling further speculative out-of-order execution of instructions past the fence. However, it increases hardware complexity, as it must track dependencies between protected registers and the instructions using them. This requires propagating a protection taint and ensuring that unsafe instructions do not speculatively use tainted registers, akin to speculative taint-tracking mechanisms [90].

End-to-end verification. There is currently a gap between our Jasmin-level security contract and its hardware implementation. To close this gap and achieve end-to-end security, future work could focus on (i) proving that the Jasmin compiler preserves the security contract (SCT), building on recent theoretical foundations and (ii) verifying that a hardware implementation satisfies the contract.

9.3 Generalization

Application to speculative sandboxing. Even though we focused our formalization on speculative constant-time, `dfence` can also be used to reason about speculative sandboxing. We detail here how to apply our type system to this model. Speculative sandboxing considers architecturally accessed values (or values inside a sandbox) as public and transiently accessed values (or values outside a sandbox) as secret. Therefore, applying our type system with the memory annotated as such trivially ensures speculative sandboxing. However, we remark that applying our type system without any secret annotation (i.e., with the entire memory marked as public) also guarantees speculative sandboxing. Indeed, assuming the program is architecturally sandboxed, it can only access secrets during speculative execution, via a transient load. Since our type system types all transient load values as secret, a program can only type-check if transient load values are protected before speculatively flowing to unsafe instructions, which guarantees speculative sandboxing.

Automatic insertion of `dfence`. The type system presented in §4 can aid developers in detecting security violations and deciding where to *manually* insert protections. Building on this, it is possible to implement an *automated* mechanism that adopts heuristics to automatically insert `dfence` instructions when violations are detected. For instance, if $\Gamma \vdash \mathsf{E} : (\mathsf{L}, \mathsf{H})$ violates the constraint $\Gamma \vdash \mathsf{E} : (\mathsf{L}, \mathsf{L})$, such a heuristic can apply `dfence` protections to all variables within E with type (L, H) right before the violation site. We conducted an initial exploration where we implemented this and similar heuristics in Jasmin. We were able to automatically protect Curve25519, Keccak$f$-1600, ChaCha20, Poly1305 and Gimli. We believe that these preliminary experiments are promising, and investigating to what extent such an approach could reduce—or even eliminate—developer effort is interesting future work.

Other languages. As we already observed in §3.2, while our paper focuses on the Jasmin language, `dfence` is not specific to Jasmin and can generally replace SLH. Moreover, although the implementation of our type system is specific to Jasmin, its core ideas can be adapted to other languages. For safe languages, information about array bounds and possible jump targets, used by our type system, can be gathered at compile time. For unsafe languages, adaptation is more difficult, but `dfence` can be coupled with control-flow integrity mechanisms, like Intel CET, that restrict jump targets to a set of well-defined landing pads.

10 Conclusion

In this paper, we introduced `dfence`, a novel instruction generalizing SLH to mitigate both Spectre-PHT and Spectre-STL. We demonstrated that `dfence` requires minimal hardware support, simplifies software instrumentation, and provides strong security guarantees. Our formally proven type system ensures correct insertion of `dfence` instructions, and our evaluation using the Proteus RISC-V core shows that `dfence` offers effective security with negligible performance and hardware impact.

Acknowledgments

This research was supported by the Agence Nationale de la Recherche (French National Research Agency) as part of the France 2030 programme – ANR-22-PECY-0006, the Internal Funds KU Leuven, and the Cybersecurity Research Program Flanders.

Ethical considerations

This research involved only the authors and did not rely on user data or interactions. End-users, companies, and public institutions may benefit from this research. The main ethical principle guiding our work is beneficence, since the results aim to improve computer security without creating risks of harm or rights violations.

Generative AI usage. In this work, generative AI has been used as a support for writing and proofreading parts of the paper.

References

[1] Intel 2024. *Hardware Features and Behaviors Related to Speculative Execution.* Intel. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/hardware-behavior-related-to-speculative-execution.html

[2] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2019. Port Contention for Fun and Profit. In *IEEE S&P*. IEEE. doi:10.1109/SP.2019.00066

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-Assurance and High-Speed Cryptography. In *CCS*. ACM. doi:10.1145/3133956.3134078

[4] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *USENIX Security*. USENIX Association. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida

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

[6] AMD. 2023. *Software Techniques for Managing Speculation on AMD Processors*. White Paper Revision 5.09.23. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/software-techniques-for-managing-speculation.pdf

[7] Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Swarn Priya, Peter Schwabe, and Lucas Tabary-Maujean. 2023. Typing High-Speed Cryptography against Spectre v1. In *IEEE S&P*. IEEE. doi:10.1109/SP46215.2023.10179418

[8] Herinomena Andrianatrehina, Ronan Lashermes, Joseph Paturel, Simon Rokicki, and Thomas Rubiano. 2025. Exploring speculation barriers for RISC-V selective speculation. In *ARES*. ACM. https://hal.science/hal-05061555

[9] Santiago Arranz Olmos, Gilles Barthe, Lionel Blatter, Benjamin Grégoire, and Vincent Laporte. 2025. Preservation of Speculative Constant-Time by Compilation. *Proceedings of the ACM on Programming Languages* 9, POPL (2025). doi:10.1145/3704880

[10] Santiago Arranz Olmos, Gilles Barthe, Chitchanok Chuengsatiansup, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Peter Schwabe, Yuval Yarom, and Zhiyuan Zhang. 2025. Protecting Cryptographic Code Against Spectre-RSB: (and, in Fact, All Known Spectre Variants). In *ASPLOS*, Vol. 2. ACM. doi:10.1145/3676641.3716015

[11] Jonathan Baumann, Roberto Blanco, Léon Ducruet, Sebastian Harwig, and Catalin Hritcu. 2025. FSLH: Flexible Mechanized Speculative Load Hardening. In *IEEE CSF*. IEEE, 569–584. doi:10.1109/CSF64896.2025.00023

[12] Daniel J. Bernstein. 2005. Cache-timing attacks on AES. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[13] Daniel J. Bernstein. 2005. The Poly1305-AES Message-Authentication Code. In *FSE*. Springer. doi:10.1007/11502760_3

[14] Daniel J. Bernstein. 2006. Curve25519: New Diffie-Hellman Speed Records. In *PKC*. Springer. doi:10.1007/11745853_14

[15] Daniel J. Bernstein. 2008. ChaCha, a variant of Salsa20. In *SASC*, Vol. 8. ECRYPT. https://www.ecrypt.eu.org/stvl/sasc2008/SASCRecord.zip

[16] Daniel J. Bernstein, Stefan Kölbl, Stefan Lucks, Pedro Maat Costa Massolino, Florian Mendel, Kashif Nawaz, Tobias Schneider, Peter Schwabe, François-Xavier Standaert, Yosuke Todo, and Benoît Viguier. 2017. Gimli : A Cross-Platform Permutation. In *CHES*. Springer. doi:10.1007/978-3-319-66787-4_15

[17] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. 2012. The Security Impact of a New Cryptographic Library. In *LATINCRYPT*. Springer. doi:10.1007/978-3-642-33481-8_9

[18] Guido Bertoni, Joan Daemens, Michaël Peeters, and Gilles Van Assche. 2011. *The Keccak Reference*. Reference Specification Version 3.0. https://keccak.team/files/Keccak-reference-3.0.pdf

[19] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: Exploiting Speculative Execution through Port Contention. In *CCS*. ACM. doi:10.1145/3319535.3363194

[20] G. R. Blakley. 1979. Safeguarding cryptographic keys. In *MARK*. IEEE. doi:10.1109/MARK.1979.8817296

[21] Marton Bognar, Job Noorman, and Frank Piessens. 2023. Proteus: An Extensible RISC-V Core for Hardware Extensions. Presented at "RISC-V Summit Europe". https://lirias.kuleuven.be/4088721

[22] Joppe Bos, Leo Ducas, Eike Kiltz, T Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehle. 2018. CRYSTALS - Kyber: A CCA-Secure Module-Lattice-Based KEM. In *IEEE EuroS&P*. IEEE. doi:10.1109/EuroSP.2018.00032

[23] Lucy Bowen and Chris Lupo. 2020. The Performance Cost of Software-based Security Mitigations. In *ICPE*. ACM. doi:10.1145/3358960.3379139

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

[25] Chandler Carruth. 2018. *Speculative Load Hardening*. https://llvm.org/docs/SpeculativeLoadHardening.html

[26] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. 2022. SoK: Practical Foundations for Software Spectre Defenses. In *IEEE S&P*. IEEE. doi:10.1109/SP46214.2022.9833707

[27] Sunjay Cauligi, Craig Disselkoen, Klaus von Gleissenthall, Dean M. Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2020. Constant-time foundations for the new spectre era. In *PLDI*. ACM. doi:10.1145/3385412.3385970

[28] Boru Chen, Yingchen Wang, Pradyumna Shome, Christopher W. Fletcher, David Kohlbrenner, Riccardo Paccagnella, and Daniel Genkin. 2024. GoFetch: Breaking Constant-Time Cryptographic Implementations Using Data Memory-Dependent Prefetchers. In *USENIX Security*. USENIX Association. https://www.usenix.org/conference/usenixsecurity24/presentation/chen-boru

[29] Xiaoyu Cheng, Fei Tong, Hongyu Wang, Zhe Zhou, Fang Jiang, and Yuxing Mao. 2024. SpecLFB: Eliminating Cache Side Channels in Speculative Executions. In *USENIX Security*. USENIX Association. https://www.usenix.org/conference/usenixsecurity24/presentation/cheng-xiaoyu

[30] Rutvik Choudhary, Jiyong Yu, Christopher Fletcher, and Adam Morrison. 2021. Speculative Privacy Tracking (SPT): Leaking Information From Speculative Execution Without Compromising Privacy. In *MICRO*. ACM. doi:10.1145/3466752.3480068

[31] Md Hafizul Islam Chowdhuryy and Fan Yao. 2022. Leaking Secrets Through Modern Branch Predictors in the Speculative World. *IEEE Trans. Comput.* 71, 9 (2022). https://doi.org/10.1109/TC.2021.3122830

[32] Neophytos Christou, Alexander J. Gaidis, Vaggelis Atlidakis, and Vasileios P. Kemerlis. 2024. Eclipse: Preventing Speculative Memory-error Abuse with Artificial Data Dependencies. In *CCS*. ACM. doi:10.1145/3658644.3690201

[33] Formosa Crypto. 2025. *formosa-mldsa*. https://github.com/formosa-crypto/formosa-mlkem

[34] Formosa Crypto. 2025. *formosa-mlkem*. https://github.com/formosa-crypto/formosa-mldsa/tree/armv7m-riscv32

[35] Formosa Crypto. 2025. *libjade*. https://github.com/formosa-crypto/libjade

[36] Lesly-Ann Daniel, Marton Bognar, Job Noorman, Sébastien Bardin, Tamara Rezk, and Frank Piessens. 2023. ProSpeCT: Provably Secure Speculation for the Constant-Time Policy. In *USENIX Security*. USENIX Association. https://www.usenix.org/conference/usenixsecurity23/presentation/daniel

[37] Davide Davoli, Marton Bognar, Lesly-Ann Daniel, Benjamin Grégoire, Frank Piessens, and Tamara Rezk. 2026. dfence: Fine-Grained Speculation Barriers for Efficient and Effective Hardware-Software Protection in the Spectre Era (Extended Version). arXiv:2608.06124 [cs.CR]

[38] Jesse De Meulemeester, Quinten Norga, Frank Piessens, Ingrid Verbauwhede, and Marton Bognar. 2025. Hardware Cost Evaluation in Systems Security. In *ACM REP*. doi:10.1145/3736731.3746155

[39] Jack Doweck. 2006. Intel Smart Memory Access and the Energy-Efficient Performance of the Intel Core Microarchitecture. https://web.archive.org/web/20231026102745/https://www.all-electronics.de/wp-content/uploads/migrated/document/196371/413ei0507-intel-sma.pdf

[40] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. 2018. CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018, 1 (2018). doi:10.13154/TCHES.V2018.I1.238-268

[41] Jacob Fustos, Michael Garrett Bechtel, and Heechul Yun. 2020. SpectreRewind: Leaking Secrets to Past Instructions. In *ASHES@CCS*. ACM. doi:10.1145/3411504.3421216

[42] Freddy Gabbay. 1996. *Speculative Execution Based on Value Prediction*. Technical Report 1080. Technion — Israel Institute of Technology, Electrical Engineering Department.

[43] Johann Großschädl, Elisabeth Oswald, Dan Page, and Michael Tunstall. 2009. Side-Channel Analysis of Cryptographic Software via Early-Terminating Multiplications. In *ICISC*. Springer. doi:10.1007/978-3-642-14423-3_13

[44] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2021. Hardware-Software Contracts for Secure Speculation. In *IEEE S&P*. IEEE. doi:10.1109/SP40001.2021.00036

[45] Ali Hajiabadi and Trevor E. Carlson. 2024. Providing High-Performance Execution with a Sequential Contract for Cryptographic Programs. arXiv:2406.04290 [cs.CR]

[46] Ali Hajiabadi and Trevor E. Carlson. 2025. Cassandra: Efficient Enforcement of Sequential Execution for Cryptographic Programs. In *ISCA*. ACM. doi:10.1145/3695053.3731048

[47] Jann Horn. 2018. *Speculative Execution, Variant 4: Speculative Store Bypass*. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528

[48] Intel. 2018. *Bounds Check Bypass / CVE-2017-5753 / INTEL-SA-00088*. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/bounds-check-bypass.html

[49] Intel. 2018. *Speculative Store Bypass / CVE-2018-3639 / INTEL-SA-00115*. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/speculative-store-bypass.html

[50] Intel. 2018. *Using Intel® Compilers to Mitigate Speculative Execution Side-Channel Issues*. https://www.intel.com/content/www/us/en/developer/articles/troubleshooting/using-intel-compilers-to-mitigate-speculative-execution-side-channel-issues.html

[51] Intel. 2020. *An Optimized Mitigation Approach for Load Value Injection.* https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/optimized-mitigation-approach-load-value-injection.html

[52] Intel. 2022. *Fast Store Forwarding Predictor.* Intel. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/fast-store-forwarding-predictor.html

[53] Intel. 2025. *Data Operand Independent Timing ISA Guidance.* Intel Corporation. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html

[54] Intel. 2025. *Intel ®64 and IA-32 Architectures Software Developer's Manual.*

[55] RISC-V International. 2024. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA version 20240411.* RISC-V.

[56] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2019. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. In *DAC*. ACM. doi:10.1145/3316781.3317903

[57] Jason Kim, Jalen Chuang, Daniel Genkin, and Yuval Yarom. 2025. FLOP: Breaking the Apple M3 CPU via False Load Output Predictions. In *USENIX Security*. USENIX Association. https://www.usenix.org/conference/usenixsecurity24/presentation/chen-boru

[58] Jason Kim, Daniel Genkin, and Yuval Yarom. 2025. SLAP: Data Speculation Attacks via Load Address Prediction on Apple Silicon. In *IEEE S&P*. IEEE. doi:10.1109/SP61157.2025.00098

[59] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative Buffer Overflows: Attacks and Defenses. arXiv:1807.03757 [cs.CR]

[60] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *IEEE S&P*. IEEE. doi:10.1109/SP.2019.00002

[61] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*. Springer. doi:10.1007/3-540-68697-5_9

[62] Matthew Kolosick, Basavesh Ammanaghatta Shivakumar, Sunjay Cauligi, Marco Patrignani, Marco Vassena, Ranjit Jhala, and Deian Stefan. 2025. Robust Constant-Time Cryptography. *Proceedings of the ACM on Programming Languages* 9, PLDI (2025), 1491–1515. doi:10.1145/3729310

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

[64] Esmaeil Mohammadian Koruyeh, Shirin Haji Amin Shirazi, Khaled N. Khasawneh, Chengyu Song, and Nael B. Abu-Ghazaleh. 2020. SpecCFI: Mitigating Spectre Attacks using CFI Informed Speculation. In *IEEE S&P*. IEEE. doi:10.1109/SP40000.2020.00033

[65] Arm Limited. 2015. *Programmer's Guide for ARMv8-A.*

[66] Arm Limited. 2018. *Addressing Spectre Variant 1 (CVE-2017-5753) in Software.* White Paper Version 1.0. https://developer.arm.com/documentation/102820/latest/

[67] Arm Limited. 2020. *DIT, Data Independent Timing.* Arm Limited. https://developer.arm.com/documentation/ddi0601/2020-12/AArch64-Registers/DIT--Data-Independent-Timing

[68] Mikko H. Lipasti and John Paul Shen. 1996. Exceeding the Dataflow Limit via Value Prediction. In *MICRO*. IEEE. doi:10.1109/MICRO.1996.566464

[69] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. 1996. Value Locality and Load Value Prediction. In *ASPLOS*. ACM. doi:10.1145/237090.237173

[70] Chen Liu, Abhishek Chakraborty, Nikhil Chawla, and Neer Roggel. 2022. Frequency Throttling Side-Channel Attack. In *CCS*. ACM. doi:10.1145/3548606.3560682

[71] Giorgi Maisuradze and Christian Rossow. 2018. ret2spec: Speculative Execution Using Return Stack Buffers. In *CCS*. ACM. doi:10.1145/3243734.3243761

[72] Oleksii Oleksenko, Marco Guarnieri, Boris Köpf, and Mark Silberstein. 2023. Hide and Seek with Spectres: Efficient discovery of speculative information leaks with random testing. In *IEEE S&P*. IEEE. doi:10.1109/SP46215.2023.10179391

[73] Marco Patrignani and Marco Guarnieri. 2021. Exorcising Spectres with Secure Compilers. In *CCS*. ACM. doi:10.1145/3460120.3484534

[74] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. 2021. Rage against the Machine Clear: A Systematic Analysis of Machine Clears and Their Implications for Transient Execution Attacks. In *USENIX Security*. USENIX Association. https://www.usenix.org/conference/usenixsecurity21/presentation/ragab

[75] Allison Randal. 2023. This is How You Lose the Transient Execution War. arXiv:2309.03376 [cs.CR]

[76] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M. Tullsen, and Ashish Venkat. 2021. I See Dead μops: Leaking Secrets via Intel/AMD Micro-Op Caches. In *ISCA*. IEEE. doi:10.1109/ISCA52012.2021.00036

[77] Philipp Schmitz, Tobias Jauch, Alex Wezel, Mohammad R. Fadiheh, Thore Tiemann, Jonah Heller, Thomas Eisenbarth, Dominik Stoffel, and Wolfgang Kunz. 2024. Okapi: Efficiently Safeguarding Speculative Data Accesses in Sandboxed Environments. arXiv:2312.08156 [cs.CR]

[78] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. 2019. NetSpectre: Read Arbitrary Memory over Network. In *ESORICS*. Springer. doi:10.1007/978-3-030-29959-0_14

[79] Adi Shamir. 1979. How to Share a Secret. *Commun. ACM* 22, 11 (1979). doi:10.1145/359168.359176

[80] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: An Extremely Simple Oblivious RAM Protocol. In *CCS*. ACM, 299–310. doi:10.1145/2508859.2516660

[81] Mohammadkazem Taram, Ashish Venkat, and Dean M. Tullsen. 2019. Context-Sensitive Fencing: Securing Speculative Execution via Microcode Customization. In *ASPLOS*. ACM. doi:10.1145/3297858.3304060

[82] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kici, Ranjit Jhala, Dean M. Tullsen, and Deian Stefan. 2021. Automatically eliminating speculative leaks from cryptographic code with blade. *Proceedings of the ACM on Programming Languages* 5, POPL (2021). doi:10.1145/3434330

[83] Jose Rodrigo Sanchez Vicarte, Michael Flanders, Riccardo Paccagnella, Grant Garrett-Grossman, Adam Morrison, Christopher W. Fletcher, and David Kohlbrenner. 2022. Augury: Using Data Memory-Dependent Prefetchers to Leak Data at Rest. In *IEEE S&P*. IEEE. doi:10.1109/SP46214.2022.9833570

[84] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. 2022. Hertzbleed: Turning Power Side-Channel Attacks into Remote Timing Attacks on X86. In *USENIX Security*. USENIX Association. https://www.usenix.org/conference/usenixsecurity22/presentation/wang-yingchen

[85] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch, and Baris Kasikci. 2019. NDA: Preventing Speculative Execution Attacks at Their Source. In *MICRO*. ACM. doi:10.1145/3352460.3358306

[86] Hans Winderix, Marton Bognar, Lesly-Ann Daniel, and Frank Piessens. 2024. Libra: Architectural Support For Principled, Secure And Efficient Balanced Execution On High-End Processors. In *CCS*. ACM. doi:10.1145/3658644.3690319

[87] Hans Winderix, Marton Bognar, Job Noorman, Lesly-Ann Daniel, and Frank Piessens. 2024. Architectural Mimicry: Innovative Instructions to Efficiently Address Control-Flow Leakage in Data-Oblivious Programs. In *IEEE S&P*. IEEE. doi:10.1109/SP54263.2024.00047

[88] Nils Wistoff, Moritz Schneider, Frank K. Gürkaynak, Luca Benini, and Gernot Heiser. 2021. Microarchitectural Timing Channels and their Prevention on an Open-Source 64-bit RISC-V Core. In *DATE*. IEEE. doi:10.23919/DATE51398.2021.9474214

[89] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W. Fletcher. 2019. Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing. In *NDSS*. https://www.ndss-symposium.org/ndss-paper/data-oblivious-isa-extensions-for-side-channel-resistant-and-high-performance-computing/

[90] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. 2019. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *MICRO*. ACM. doi:10.1145/3352460.3358274

[91] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. 2023. Ultimate SLH: Taking Speculative Load Hardening to the Next Level. In *USENIX Security*. USENIX Association. https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-zhiyuan-slh

A Open Science

We believe that our prototype is an important contribution, which we release as open software. At https://doi.org/10.5281/zenodo.20770661, we archive the following:

- The prototype implementation of **dfence** in Proteus;
- The Jasmin implementation of the benchmark suite;
- A version of the Jasmin compiler with support for **dfence** instructions;
- A Docker environment for replicating the evaluation.

We will also incorporate most components in the upstream Proteus ecosystem at https://github.com/proteus-core.