Defeating Transient Execution Attacks by Limiting Secret Reachability through REGISTER HIDING and SHADOWCFI

Daniël Trujillo
MIT CSAIL
danieltr@mit.edu

Jagadish Kotra
AMD
jagadish.kotra@amd.com

David Kaplan
AMD
david.kaplan@amd.com

Mengjia Yan
MIT CSAIL
mengjiay@mit.edu

Abstract—**Modern processors incorporate aggressive branch prediction mechanisms for indirect branches, offering various unanticipated ways to influence speculative behavior during a transient execution attack. Existing mitigations against these so-called Spectre v2-style attacks are often ad-hoc, highly specific to the discovered attack and the targeted microarchitecture, and thus fail to generalize. In this paper, we identify a core requirement previously overlooked that all of these attacks share: *secret reachability*. Building upon this, we propose REGISTER HIDING and SHADOWCFI, two complementary but independent software-based and hardware-agnostic techniques which target the attacker's ability to reach secrets in registers and memory. REGISTER HIDING hides the architectural register state before a misprediction can occur, while SHADOWCFI ensures the architectural register state can only be restored at the correct target. To demonstrate their merit, we implement a fully functional patch for Linux kernel version 6.8.0, protecting against known and futuristic Spectre v2-style attacks, including all those which target indirect jumps, indirect calls and returns. We provide a security analysis and corresponding scanner to verify that an attacker cannot restore the register state during misprediction in our proof-of-concept. Replacing the most recently deployed Spectre v2 defenses with REGISTER HIDING and SHADOWCFI reduces the overall mitigation overhead on AMD Zen 4 from 114.1% to 75.9% for LEBench, and from 33.4% to 25.8% on average across server workloads.**

1. Introduction

Speculative execution is paramount for the performance of pipelined processors. However, mispredictions can be dangerous, potentially leaving secret-dependent traces behind. Indirect branch mispredictions (Spectre v2-type) are one of the most dangerous, allowing an attacker to execute arbitrary gadgets in the victim's address space. Various mitigations have been proposed and implemented to protect against these attacks, targeting different attack ingredients. In this paper, we identify an ingredient that remained untargeted so far, and propose to tackle this as a defense against Spectre v2-style attacks.

Indirect branch mispredictions. Although speculative execution is beneficial for performance, it has been less favorable for security. By mistraining predictors in the CPU's Branch Prediction Unit (BPU), attackers can leak data through side-channels that they architecturally have no access to [1]. A large number of these so-called transient execution attacks have been discovered, either targeting different predictors or training them in distinct ways [2], [3], [4], [5], [6], [7], [8], [9]. By mistraining the BPU, an attacker causes mispredictions to a disclosure gadget during victim execution, leaving secret-dependent traces behind in microarchitectural structures such as the cache. Spectre v2-style attacks target indirect branches and are particularly powerful, allowing any sequence of bytes in the victim's address space to serve as a disclosure gadget. Recent attacks targeting indirect branches such as Retbleed [3], BHI [10], Inception [4] and BPI [8] show the relevance of these attacks, even today.

Mitigations. Various mitigations against transient execution attacks targeting indirect branches have been proposed and implemented in recent years. While some work focused on removing disclosure gadgets from the victim's address space [11], most defenses attempt to prevent disclosure gadgets from being reached under speculation, either in software (e.g., retpoline) or in hardware (e.g., IBRS) [12], [13], [14], [15], [16], [17], [18], [19]. Other defenses propose to remove secrets from the victim's address space [20], [10]. Lastly, a large number of academic works proposed hardware changes to prevent side-channels due to transiently executed instructions [21], [22], [23], [24], [25].

Attack ingredients. Surveying previously proposed mitigations of Spectre attacks targeting indirect branches, we realize that they all overlook one ingredient: *secret reachability*. Previous work assumes that if a secret exists, it is automatically reachable as well, i.e., the disclosure gadget can find a way to consume the desired secret. During our analysis of previous Spectre attacks, we come to the conclusion that they all achieve secret reachability by relying on the architectural register state, i.e., the register state committed prior to the misprediction. Can we limit secret reachability for transient execution attacks by blocking access to the architectural register state during mispredictions?

Register Hiding and ShadowCFI. Instead of limiting speculation or fighting side-channels, we propose to tackle secret reachability. Before each potentially vulnerable branch, Register Hiding empties the register state, restoring it only at the destination of a branch. In this way, the attacker has very limited options for reaching secrets, significantly raising the bar for exploitation. To ensure we only release the register state at the correct target, we furthermore introduce ShadowCFI, a control-flow integrity mechanism that compares the speculative target to the architectural target, preventing the register state from being released at an incorrect target.

Our defense is general, software-only and hardware-agnostic. Furthermore, we validate the kernel to be free of any gadgets which allow an attacker to restore the register state, or bypass ShadowCFI, speculatively. Unlike currently deployed mitigations, our defense does not force mispredictions, and therefore enjoys high throughput. Our benchmarking results show that we obtain significantly higher performance than default mitigations in the Linux kernel, despite the additional instructions we introduce. Again in contrast to recently deployed mitigations, our defense protects against indirect mispredictions regardless of the attack used to trigger them. Owing to this generality, it defeats all known transient execution attacks targeting indirect branches, and offers resilience against potential future variants, without any hardware support or hardware assumptions.

Contributions

In summary, our contributions are:

- We identify a core ingredient of transient execution attacks mispredicting indirect branches, not targeted by any previous defense: *secret reachability*.
- We introduce Register Hiding, a novel approach to protect against Spectre v2-style attacks by ensuring an empty register state upon a misprediction.
- We present ShadowCFI, a run-time control-flow integrity mechanism which exposes the speculative instruction pointer to software, allowing comparison against the architectural branch target.
- We build a proof-of-concept for Linux kernel version `6.8.0`. LEBench results on AMD Zen 4 demonstrate a reduction in overall mitigation overhead from 114.1% to 75.9% when protecting against Spectre v2 attacks using Register Hiding and ShadowCFI.
- We validate our defense using a security analysis and corresponding gadget scanner, verifying the inability of an attacker to restore the architectural register state at an incorrect target under speculation. This allows the attacker to cause misprediction to any byte-aligned address, even those not corresponding to an architectural instruction boundary.

Our proof-of-concept defense using Register Hiding and ShadowCFI, along with the validating gadget analyzer, is available at https://github.com/MATCHA-MIT/register-hiding-and-shadowcfi.

2. Background

2.1. Branching and speculative execution

Branch instructions determine the next instruction pointer, but may have dependencies and take time to complete. To deal with this, the processor could just stall, waiting for the outcome of the branch. Instead, virtually all CPUs opt for a more performant solution. By predicting the next instruction pointer, the pipeline can be filled up with instructions that can be worked on while the branch is being resolved. This technique, called speculative execution, improves processor performance considerably. When the prediction is confirmed to be correct, instructions executed after the prediction can commit to the architectural state. If mispredictions occur, the processor eventually re-steers to the correct execution path, discarding instructions executed after the mispredicted branch.

Among the different branch types, indirect branches are most flexible with their context-dependent targets. Indirect jumps update the instruction pointer to a new target read from a register or from memory, while indirect calls additionally push a return address onto the stack. Return instructions update the instruction pointer to the target last pushed onto the stack by a call instruction (i.e., from memory), and are thus also considered indirect branches.

Given that indirect branches update the instruction pointer to arbitrary targets, mispredictions can cause speculative execution of an intractable number of paths. In this paper, we focus on all indirect branches – jumps, calls and returns.

2.2. Branch prediction for indirect branches

To best guess the next instruction pointer, the front-end of the processor is embodied with a BPU. The BPU's task is to have a new instruction pointer ready at all times, facilitating new instructions being fetched and fed into the pipeline. Given the different types of branches that exist, the BPU itself consists of various components that help with its decisions.

Branch Target Buffer. The Branch Target Buffer (BTB) records previously seen branch targets, and typically contains thousands of entries [26]. On AMD CPUs, this includes targets for all branches except return targets [15]. On Intel CPUs, the BTB also records previously seen targets of return instructions [16]. Reverse engineering efforts show that the BTB is normally indexed with a hash of the current instruction pointer [27], [3], [28], [1]. As a result, collisions may occur in the BTB between independent branches, causing mispredictions. In addition to a target, the BTB is also known to predict the type of the branch on some microarchitectures [5], [4].

Return Stack Buffer. To help predict the outcome of return instructions, the BPU is equipped with a dedicated buffer called the Return Stack Buffer (RSB), which acts as a microarchitectural stack. When a return address is pushed onto the stack (due to a direct or indirect call), the return address is additionally pushed on the RSB. A return instruction pops from this stack to obtain a target for speculative execution. The RSB is typically 16 or 32 entries big [16], [15], and may operate as a circular stack. Due to its limited size, an underflow can happen. On Intel CPUs, the BTB is used for return instructions upon an RSB underflow [29].

Branch History Buffer. A Branch History Buffer (BHB) records the most recent branches in the execution flow. With the value recorded in the BHB, a choice can be made between multiple target candidates in the BTB for indirect branches. The value is commonly a concatenation or combination of instruction pointer hashes for the recently executed branches [6].

2.3. Transient execution attacks

While improving performance, speculative execution can lead to unintended information disclosure when the processor mispredicts. In 2018, Kocher et al. revealed Spectre, showing that speculative execution can be abused to leak data across security domains [1]. By carefully executing code snippets, the CPU can be trained to take certain predictions in the future. A misprediction triggers *transient execution*, opening a *transient window* of instructions which are wrongfully executed under speculation. If these instructions constitute a so-called *disclosure gadget*, mispredictions can lead to information disclosure across security domains. A *universal disclosure gadget* is one that allows the attacker to leak arbitrary secrets in the system.

Spectre attacks that poison the BTB to cause indirect branch mispredictions are referred to as Spectre v2 attacks. Being able to mispredict to arbitrary targets, it becomes significantly easier for the attacker to transiently execute a disclosure gadget. A large number of Spectre v2-style attacks followed after its initial discovery. Spectre-RSB triggers mispredictions of return instructions by mistraining the RSB [7], [9]. In [30], it was shown that fence instructions do not protect indirect branches from starting transient windows large enough for data leakage. In Retbleed, researchers showed that return instructions can mispredict due to BTB-poisoning as well [3]. Branch History Injection (BHI) showed the BHB can be manipulated across security domains on Intel CPUs, triggering mispredictions of indirect branches without poisoning the BTB [6]. Phantom showed that even non-branch instructions can mispredict due to BTB poisoning on x86 microarchitectures [5]. Inception transiently trains the RSB with Phantom to hijack return instructions on AMD microarchitectures. InSpectre [31] showed that non-trivial disclosure gadgets can lead to information disclosure. Training Solo revealed that an attacker can realistically trick the Linux kernel into training its own indirect branches on Intel CPUs [2]. Lastly, Branch Privilege Injection (BPI) showed that BTB updates are made asynchronously on Intel CPUs, allowing indirect branch predictions to be inserted with incorrect privilege tags [8].

2.4. Transient execution attack mitigations

Software mitigations. The original Spectre v2 attack was mitigated with retpoline [13], transforming indirect branches into return instructions, forcefully mispredicting them using the RSB. On AMD microarchitectures, retpoline was replaced with a different variant which simply placed an `lfence` before the indirect branch [32]. After this variant was proven to be insecure, AMD switched to the default repoline as well. To protect against Spectre-RSB, the RSB is *stuffed*, i.e., it is filled up with benign targets after potential attacker influence [14].

When retpoline was shown to be insufficient due to Retbleed, it was enhanced with call-depth tracking on Intel CPUs, which prevents RSB underflows [29]. For AMD CPUs, retpoline was replaced by *jmp2ret*, a mitigation which centralizes all returns into a single location [17]. This single return is "untrained" upon kernel entry, guaranteeing it is free of attacker influence. When *jmp2ret* was bypassed by Inception, AMD replaced it with *saferet* [18]. This mitigation furthermore executes a call before the centralized return, providing the guarantee that the top-of-the-RSB value is known and benign. However, this comes at a performance cost: all returns are mispredicted.

Hardware mitigations. In addition to software mitigations, CPU vendors released a number of hardware mitigations. Indirect Branch Prediction Barrier (IBPB) provides a barrier such that indirect branches following it are free of influence of those executed prior to the barrier [15], [16]. Single Thread Indirect Branch Predictors (STIBP) furthermore isolates these predictions between sibling hyperthreads. Likewise, Indirect Branch Restricted Speculation (IBRS) limits its indirect branch predictions from being influenced by lower-level privilege software. Newer AMD and Intel CPUs provide Automatic IBRS and Enhanced IBRS respectively, which are IBRS that are active in the background, without the need of explicitly triggering them on a privilege switch. Furthermore, many of the shown attacks triggered microcode updates which provide ad-hoc mitigations [33], [18], [34].

2.5. Control-flow integrity for software security

Control-Flow Integrity (CFI) mechanisms help defend against software attacks in hardware. Newer Intel CPUs support Control-flow Enforcement Technology (CET), which compares stack values against a Shadow Stack for returns [35]. In addition, indirect branches are protected using Indirect Branch Tracking (IBT), restricting indirect branch targets to those which start with an `endbr` instruction. On newer AMD CPUs, Shadow Stack is supported, but protection for indirect branches is not.

While Shadow Stack and IBT were not designed to defend against transient execution attacks, IBT on Intel CPUs provide some guarantee about the maximum number of instructions that may execute after a missing `endbr`. Older Intel CPUs may execute less than 8 instructions transiently, while newer Intel CPUs are not supposed to transiently execute past a missing `endbr` at all [35]. Recent offensive work showed, however, that even newer Intel CPUs allow for a single load to execute after a missing `endbr` [31].

FineIBT furthermore restricts indirect branch targets to a smaller and more precise set on Intel CPUs, mainly for software security [36]. It also aimed to reduce the speculative attack surface, although it was later bypassed [31].

3. Threat model

We assume a strong and realistic threat model, covering and going beyond what previous Spectre attacks targeting indirect branches have demonstrated. We assume the attacker is able to execute unprivileged but arbitrary code, aiming to leak data from a victim program on the same machine. In particular, we focus on the Linux kernel, which has access to all mapped memory on the system, but our techniques could be applied to other victims as well.

We consider misprediction to occur while executing any indirect branch (jump, call, return). We broadly consider all possible ways for an attacker to establish the desired branch predictor state: the attacker may be training across privilege domains out-of-place, or the victim may even be training itself (in-place/same-domain). We do not assume the predictor which serves the speculative branch target either. This covers a broader threat model than many previously deployed mitigations [17], [18], [13], [15], [16].

Furthermore, we allow the attacker to arbitrarily choose any address to speculate into. This address may not be the start of a function (i.e., we do not rely on IBT or the like [35]), and it may not even be the start of an existing instruction. That is, we allow the attacker to pick any byte-aligned address in the victim's address space. This is again going beyond what previous mitigations assume [35], [36].

4. Motivation

In this section, we outline the motivation behind REGISTER HIDING. First, in Section 4.1, we explore the requirements for Spectre v2-style attacks that have been targeted by previous defensive work. In Section 4.2 we identify that one requirement – secret reachability – was overlooked by previous work, serving as the motivation behind our defense. Lastly, in Section 4.3 we discuss how secret reachability is dependent on the architectural register state, providing a path toward a practical mechanism to limit it.

4.1. Attack ingredients

Previous defense work can be categorized by which ingredient of Spectre v2-style attacks they target. We recognize that previous work has targeted four distinct ingredients: ① *gadget availability*, ② *gadget reachability*, ③ *secret availability* and ④ *secret transmittability*. To explain what these ingredients mean, we discuss each in detail below. We use the example attack shown in Figure 1 throughout this explanation, which depicts a cross-privilege Spectre v2-style attack, leaking an arbitrary secret from the privileged kernel.

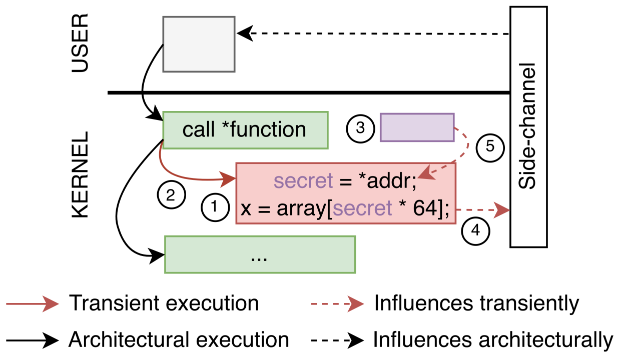

Figure 1: Example attack. Previous defenses target ① gadget availability, ② gadget reachability, ③ secret availability, and ④ secret transmittability. What about ⑤ secret reachability?

① Gadget availability. The first requirement is rather trivial: a disclosure gadget must be present in the victim's address space. The one shown in Figure 1 first loads the secret pointed to by `addr` (under control of the attacker in user-space) into `secret`, and then accesses a `secret`-dependent entry in the array pointed to by `array`. However, this is by no means the only type of disclosure gadget that can result in data leakage [37], [31]. Any code snippet that performs secret-dependent operations may introduce a side-channel and serve as a disclosure gadget.

Previous work built scanners to determine the presence of disclosure gadgets in a victim binary for exploitation [3], [38], [31], [39]. Recent defensive work proposed to map the kernel only partially in the virtual address space, decreasing the number of disclosure gadgets that can be exploited by an attacker [11].

② Gadget reachability. In addition to the existence of a disclosure gadget, it must also be reachable under speculation. In other words, the victim must be able to trigger a misprediction during its execution, causing transient execution of the disclosure gadget. In Figure 1, an indirect call mispredicts to the disclosure gadget, because the attacker was able to manipulate the branch predictor from user-space.

A large number of offensive works have been published on novel ways to achieve gadget reachability [1], [3], [5], [4], [6], [7]. On the defensive side, the majority of mitigations against transient execution attacks target gadget reachability, including retpoline, IBRS, and *saferet* [12], [13], [17], [7], [14], [15], [16], [18], [29], [30], [40].

③ Secret availability. Having a disclosure gadget and the ability to reach it transiently, the attacker needs a secret to leak. In Figure 1, the secret is available in the kernel address space, allowing operations to be performed on the secret during transient execution.

Prior defense work proposed to make secrets unavailable by unmapping them from the address space [10], [11]. Similarly, Address Space Isolation (ISO) was proposed for the Linux kernel, unmapping memory which is not immediately needed [20].

④ **Secret transmittability.** Lastly, the gadget must be able to transmit the secret to the attacker using a side-channel. In Figure 1, the secret is used to access an offset in the array pointed to by `array`, leaving a secret-dependent trace in a shared microarchitectural structure. This can be recovered by the attacker in user space, finalizing the attack.

To combat side-channels due to speculative executed instructions, hardware changes were proposed by previous defensive work [21], [22], [23], [24], [25]. In addition, prior work has proposed to isolate victim and attacker by eliminating shared microarchitectural resources [41].

4.2. Secret reachability

Our categorization of prior work highlights an important requirement that has been overlooked. While previous defenses have made a clear distinction between gadget availability and reachability, they have failed to do so for secrets. We define secret reachability as the disclosure gadget's ability to consume the secret desired by the attacker. For example, Figure 1 labels secret reachability with ⑤, since the disclosure gadget consumes the secret by de-referencing a pointer to memory.

Existing defenses that focus on secrets solely target their *availability* [10], [11], [20], assuming that if a secret exists, it can be reached by the attacker. This belief is even reflected in widely used terminology; disclosure gadgets consisting of two dependent loads are known as *universal* read gadgets [42], taking secret reachability for granted. While address-space randomization techniques such as ASLR complicate secret reachability, they were not designed as a transient execution defense, and are typically easily bypassed under speculation [3], [5], [4].

We have identified a previously underexplored requirement of transient execution attacks. This observation is especially interesting, since the far majority of deployed defenses target gadget reachability, which often comes at high performance overhead, and are tailored to specific attacks and predictors. By instead targeting secret reachability, it becomes possible to design a more generic and performant defense against Spectre attacks – an approach we will discuss in the remainder of this paper.

4.3. Secret reachability and the architectural register state

We realize that there exist three ways of achieving secret reachability, which we discuss in detail below. Figure 2 shows example disclosure gadgets for each of the three cases, where green depicts architectural execution, i.e., execution which eventually commits, and red denotes transient execution. We note that each instruction in reality may consist of multiple instructions, and that they furthermore can be separated by branches, meaning that the entire gadget is executed in nested speculation (as in [43]).

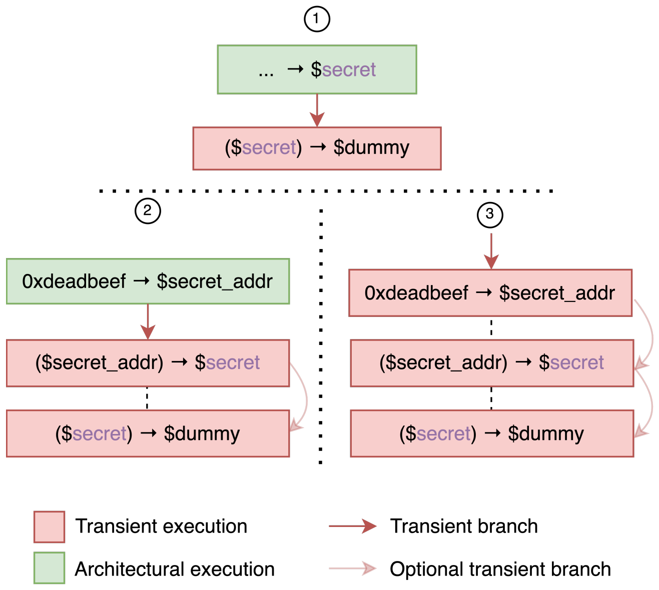

Figure 2: Three cases of secret reachability. ① **architectural register-state**: the secret is available in a register, ② **architectural register-state referenced**: the secret is in memory, and it is referenced by a register, or ③ **architectural register-state independent**: the secret is not depending on the architectural register state.

① **Architectural register-state.** The secret may be readily available in the architectural register state, such as when it has been fetched from memory during architectural execution, prior to the misprediction [1]. An example is cryptographic code; registers may contain the secret key while performing encryption or decryption. In this case, an attacker can use this register to compute a secret-dependent address during a misprediction, allowing a single load to leak the secret value.

② **Architectural register-state referenced.** This case is more dangerous, allowing even arbitrary secret leakage if the architectural register state is controlled by the attacker. In this case, the secret resides in memory, and therefore, it must first be loaded into a register before it can be leaked. A plethora of previous offensive works leak arbitrary secrets from memory by relying on a register as the (base) address [3], [4], [1], [6], [5], [44], [31]. In most cases, these are controlled by the attacker from user-space.

③ **Architectural register-state independent.** In this case, the secret is completely independent of the architectural register state. The CPU transiently computes the address of the desired secret out of nowhere, before it discloses the secret using a regular disclosure gadget. The attacker is able to obtain a reference to the secret "out of thin air", by only using the victim's code, independently of the architectural register state. Needless to say, this requires extraordinary circumstances which are unlikely to exist for arbitrary secrets in a reasonable victim program.

After reviewing previous offensive work, we come to the conclusion that all previous offensive work uses ① *architectural register-state* or ② *architectural register-state dependent* secret reachability. That is, in all of them, an attacker relies on the architectural register state during the transient window. In our example attack shown in Figure 1, we use ② *architectural register-state dependent* secret reachability.

We are unaware of any previous offensive work that uses ③ *architectural register-state independent* secret reachability. Having derived this insight, this paper proposes to limit secret reachability by preventing access to the architectural register state during mispredictions.

5. Register Hiding & ShadowCFI

In this section, we present Register Hiding and ShadowCFI, two complementary but independent techniques which together thwart transient execution attacks that target indirect branches. First, in Section 5.1 we discuss our high-level idea and the challenges we need to overcome. In Section 5.2 and Section 5.3 we discuss the design of Register Hiding and ShadowCFI respectively. Lastly, Section 5.4 provides a discussion of the alternative configuration and design options.

5.1. Overview

To leverage our insight, we propose to hide the architectural register state during mispredictions. Prior to executing any indirect branch, we hide the architectural register state, and at the correct target only, we release the architectural register state. As a result, any code executed transiently has to work without the architectural register state, blocking all previously used ways of achieving secret reachability.

Challenges. There are a number of challenges to be solved before we can actually limit secret reachability by hiding the architectural register state. As a first challenge, we need to find a way to efficiently hide the architectural register state:

Challenge C1.

Hiding the register state during mispredictions.

In Section 5.2 we present Register Hiding, which moves the architectural register state to a *hidden storage* and clears the register state prior to any indirect branch. At valid branch targets (i.e., function entry or return site), the architectural register state is restored from the hidden storage.

While this prevents access to the architectural state for mispredictions to addresses which are not branch targets, an attacker may still mispredict to a valid but incorrect branch target. Therefore, our second challenge is:

Challenge C2.

Ensuring control-flow integrity to prevent mispredictions to incorrect but valid targets.

In Section 5.3 we introduce ShadowCFI, a control-flow integrity mechanism that only releases the register state when the speculative target matches the correct architectural branch target, blocking an attacker which triggers mispredictions to incorrect, but valid targets.

Register Hiding and ShadowCFI are only secure if the hidden storage cannot be accessed transiently during victim execution. This is a non-trivial property to verify, especially since the attacker can speculate into any byte-aligned target, potentially introducing *non-architectural* instructions which can access the hidden storage. Therefore, our last challenge is:

Challenge C3.

Validating the inaccessibility of the hidden storage during misprediction through (non-architectural) instruction analysis.

In Section 6 we provide a security analysis using induction to identify the software requirements necessary to ensure the hidden storage is inaccessible at an incorrect target. We discuss the design of our sound and complete instruction analyzer which can validate that these requirements are satisfied.

5.2. Register Hiding

Our goal is to hide the architectural register state during mispredictions. In order to do this, we propose to store a snapshot of the architectural register state in a *hidden storage* prior to taking the branch. It should be infeasible for an attacker to access the hidden storage speculatively (i.e., it should be hidden). After creating the snapshot, we can safely clear the register state and execute the indirect branch. Creating the snapshot and clearing the architectural register state is done by a *Register Hide* step, which we insert prior to each indirect branch in the victim binary.

At each valid branch target, we furthermore insert a *Register Restore* step, which restores the architectural register state using the snapshot stored in the hidden storage. This ensures correct execution of the branch target. Valid branch targets can be easily inferred: every instruction following a call is a valid return site, and every valid function target is labeled by the compiler to support IBT. Figure 3 depicts the high-level operation of Register Hiding. During transient

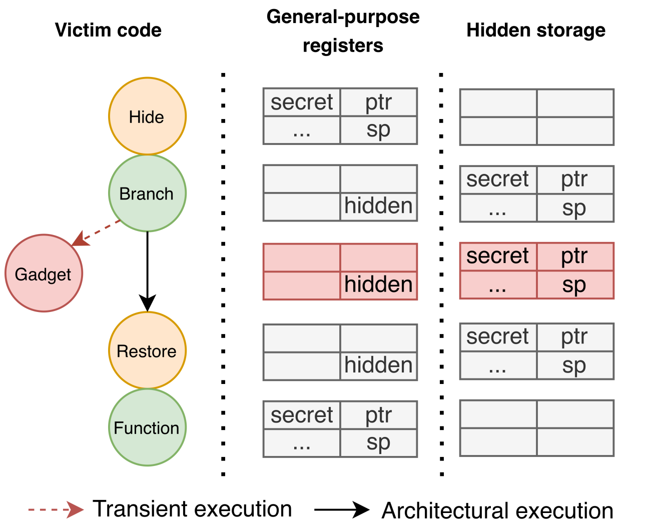

Figure 3: The design of REGISTER HIDING. The architectural register state is cleared prior to a misprediction.

execution, the register state is hidden (row 3). Only when reaching a *Register Restore*, the register state is recovered from the hidden storage.

The only register we cannot safely clear is the stack pointer (RSP), since interrupts push to the stack, and they can happen while the architectural register state is hidden. However, we should not leave RSP intact, since this register may be used to achieve secret reachability (case ② in Figure 2). Therefore, in the *Register Hide* step, we store RSP in the hidden storage, and overwrite it with a pointer to an empty memory region. This per-core *hidden stack* is allocated at boot-times, specifically for interrupt handlers to use. Section 7 expands on our operation when an interrupt occurs in a hidden state.

SSE as hidden storage. We are left with finding a suitable candidate for our hidden storage. We realize that during execution of many applications, including the Linux kernel, a part of the x86 ISA is largely unused. In particular, we find that Streaming SIMD Extensions (SSE) are rarely used in the Linux kernel, and we can pick a build configuration such that they are not used at all. SSE introduces a large number of extra registers: in its original implementation, XMM0-XMM15 provide sixteen 128-bit registers during 64-bit execution, easily large enough to store a snapshot of the general-purpose register state. Later SSE versions introduced more and larger registers, but for simplicity and compatibility, we will exclusively focus on XMM registers. The first x86 processors with SSE were shipped decades ago, and thus any x86 processor currently in use offers its support.

Leveraging this underutilization, our design uses SSE registers as the hidden storage. In our *Register Hide* step, we move the general-purpose register state to the SSE registers using movq instructions, after which we clear the register

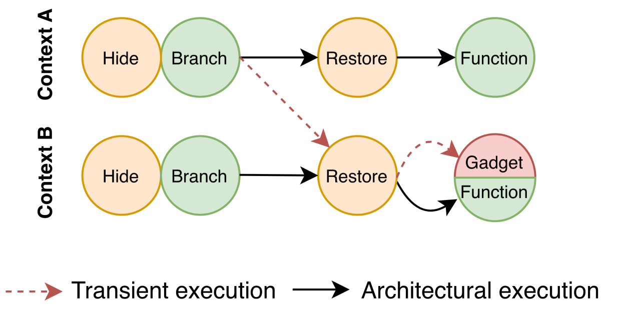

Figure 4: An attacker can cause a misprediction to an incorrect target (the target from context B). In context A, this target serves as a disclosure gadget.

state using xor instructions. In the *Register Restore* step, we simply move the values back from the SSE registers into the general-purpose registers. While this introduces a large number of additional instructions, we explore some optimizations in Section 7.2 that reduce the number of registers that need to be restored.

In rare cases where the victim program does need to use SSE, conflicts with REGISTER HIDING only arise if 1) the same SSE registers in use by REGISTER HIDING are also used by the victim program, and if 2) these SSE registers need to remain consistent *across* indirect branches for correct program execution. Fortunately, SSE usage in the kernel is limited and mostly isolated to cryptographic modules, where typically at most a small subset of SSE registers need to persist across control-flow boundaries, remaining compatible with REGISTER HIDING. If broader consistency is required, code would need to be manually patched to preserve the SSE state across an indirect branch.

5.3. SHADOWCFI

REGISTER HIDING ensures that mispredictions to addresses which are not valid targets execute without access to the architectural register state, limiting secret reachability. However, with our *Register Restore* step, we have inserted gadgets that restore the register state at all valid targets in our victim binary. An attacker can thus still trigger mispredictions to incorrect, but valid targets, leaving a remaining attack surface similar as in BHI [6] or as in software attacks on an IBT-protected machine [35]. We thus have context-*insensitive* CFI. If there happens to be a disclosure gadget at a valid target, an attacker can abuse the *Register Restore* step at this target to restore the register state before the disclosure gadget executes. Figure 4 depicts this scenario. When executing context A, a misprediction to the target of the branch in context B restores the register state and consequently allows for secret reachability.

To mitigate this scenario, we require a context-*sensitive* CFI mechanism which helps to restore the register state exclusively at the correct target of the indirect branch. Our approach is to expose the *speculative instruction pointer* to our software defense, i.e., the instruction pointer we are

***Register Hide* with SHADOWCFI**

```
movq (%rsp), %target                        SHADOWCFI
movq %rax, %xmm0                      REGISTER HIDING
movq %rbx, %xmm1
...
xor %rax, %rax
xor %rbx, %rbx
...
ret                                            Branch
```

***Register Restore* with SHADOWCFI**

```
0x810777:
movq %xmm0, %rax                      REGISTER HIDING
movq %xmm1, %rbx
...
movq %target, %temp                         SHADOWCFI
xor $0x810777, %temp
sub $1, %temp
sbb %temp, %temp
and %temp, %rax
and %temp, %rbx
...
...                                          Function
```

Figure 5: REGISTER HIDING enhanced with SHADOWCFI. `temp` is a temporary register not in use by the target. `target` is in the hidden storage.

currently performing speculative execution at. This allows comparison against the *architectural branch target*, i.e., the actual target the indirect branch is supposed to branch into. We are essentially comparing a ready-to-commit *data value* from the CPU's back end (architectural branch target) to a *control-flow prediction* from the front end (speculative instruction pointer). The architectural branch target is available in the hidden storage, and the speculative instruction pointer can be obtained by reading the virtual address of the *Register Restore* step under speculative execution.

Figure 5 shows the instructions in the *Register Hide* and *Register Restore* step, enhanced by SHADOWCFI. The architectural branch target is stored in SSE register `target` at the start of the *Register Hide* step. After restoring the architectural register state in the *Register Restore* step, we `xor` the current speculative instruction pointer (`0x810777` in the example) with the architectural branch target stored in `target`. The current speculative instruction pointer is baked in to the *Register Hide* step, and is unique for each target. If the two values are equal, subtracting `1` of the resulting value sets the carry flag (`CF`), which allows us to create a mask of `-1` using the `sbb` instruction. If the two values are not equal, `CF` is not set, and `sbb` produces a mask of `0`. This mask allows us to perform a logical `and` on all general-purpose registers, leaving them untouched at the correct target, but masking them out at a wrong target. There are many possible implementations of SHADOWCFI that may have better or worse performance than this implementation, depending on the underlying microarchitecture. For example, it is also possible to use `cmov` instructions, conditionally overwriting the restored registers with `0`. Note that to avoid mispredictions, SHADOWCFI's implementation must be free of any branches.

Comparison to traditional CFI. SHADOWCFI is a control-flow integrity mechanism specifically designed for speculation, and is significantly different than traditional work that aims to enforce CFI. In particular, prior work accomplishes CFI by comparing the speculative or architectural instruction pointer against a branch target derived through *static analysis*, which cannot be fully context-sensitive.

SHADOWCFI does not use static analysis and is fully context-sensitive, by leveraging data dependencies on the correct resolution of a branch. This data dependency is also used by Swivel's register interlocking for WebAssembly [45], clearing the heap- and stack pointer during misprediction, and by speculative load-hardening for Spectre v1 [46], which masks array indices during mispredictions.

5.4. Variations & configurations

Section 5.2 introduces REGISTER HIDING, clearing the architectural register state prior to each indirect branch. Section 5.3 introduces SHADOWCFI, preventing the architectural register state from being restored at an incorrect, but valid target. We have now presented the entire design of REGISTER HIDING and SHADOWCFI, which together thwart transient execution attacks targeting indirect branches by limiting secret reachability. We will now proceed to discuss a few configuration options in our design, allowing flexibility to adapt to the underlying hardware and assumptions.

Alternative hidden storage. While we identify SSE registers as a suitable candidate for our hidden storage, we note that our design is independent of the exact choice of hidden storage. For example, one could also opt to use memory as hidden storage, which has as a benefit that we avoid SSE register state corruption on each indirect branch. Furthermore, SHADOWCFI can be implemented more efficiently by only masking the memory pointer referencing the architectural register state, as opposed to each individual register as shown in Figure 5. However, using memory as hidden storage may make it more difficult or impossible to validate the security of our defense, as we will do in Section 6. In contrast, SSE registers are only interacted with using well-defined instruction sequences, enabling complete validation. Nevertheless, we evaluate the performance feasibility of a memory-based REGISTER HIDING variant in Section 8.3.

Modular configurability. We point out that our defense is heavily modular. Our proposed technique is designed to require minimal assumptions about the underlying hardware features or mitigations. This design choice ensures that our approach remains applicable across a wide range of platforms, including those where speculation is largely unconstrained due to known attacks or missing hardware mitigations. However, we realize that depending on the underlying hardware, varying levels of speculative control can be assumed. As such, our defense is modular and can be selectively combined and/or adapted with hardware mechanisms to reduce redundancy or improve performance. For example, our techniques could be applied to only indirect jumps and calls, assuming returns are protected using a different mechanism. Likewise, speculation restriction mechanisms providing context-insensitive CFI may be enhanced with SHADOWCFI only, offering more comprehensive protection.

To highlight and fully demonstrate the software-only and hardware-agnostic nature of our defense, we choose to evaluate REGISTER HIDING and SHADOWCFI on a platform equipped with minimal attack-proof hardware mitigations against indirect branch misprediction, while still being new. We therefore implement and evaluate our defense on AMD Zen 4, which does not support IBT, allowing large speculation windows at arbitrary speculative targets. Furthermore, return instructions on AMD Zen 4 CPUs are known to be vulnerable to attacker influence [4], and there exists no efficient hardware mitigation against this vulnerability.

Additionally, to show the generality and modularity of our defense, we evaluate the performance overhead of complementing IBT with SHADOWCFI on an Intel CPU. While IBT prevents long speculation windows at targets not starting with an `endbr` [35], [31], SHADOWCFI furthermore masks out the register state at any incorrect `endbr` destination, effectively making IBT context-sensitive. On machines which exclusively mispredict to valid kernel targets, SHADOWCFI thus limits secret reachability for attacks that trick the victim into mispredicting to a valid but incorrect target (e.g., BHI attacks [6]).

6. Security analysis & gadget validation

In this section, we present a security analysis based on induction to identify the software requirements that ensure the register state can be restored exclusively at the correct target. To help do this explicitly and comprehensively, we aim to satisfy speculative non-interference (SNI) with respect to branch misprediction as our security goal, assuming that an attacker cannot transiently reach secrets using architectural register-independent operations (③ in Section 4.3). Specifically, for any two architectural register states that yield identical observation traces under sequential execution, speculative execution should produce indistinguishable microarchitectural observation traces. We then show that the derived software requirements are satisfied in our REGISTER HIDING and SHADOWCFI protected kernel using a binary scanner (Section 6.5).

We define our SNI property in Section 6.2, and explain how it slightly deviates from prior work [47], [48]. For readability, we provide an overview of our machine semantics and the high-level inductive reasoning, deferring the complete formal discussion to Appendix A.

6.1. Machine semantics

We define speculative machine states and sequential machine states, along with their respective operations.

Speculative machine. The speculative machine has state:

$(R, H, H_{snapshot}, hidedepth, PC, PC_{snapshot}, misspec, B)$

where R is the current register state, H is the hidden storage, $hidedepth$ is a logical counter that keeps track of the number of times the register state was hidden in H, PC is the current program counter, and B is the binary file under execution by the machine.

The snapshot-related components and $misspec$ are necessary for re-steering speculative execution. Since the security analysis focuses on analyzing information leakage during misprediction, we omit these snapshot components and simply write the speculative state as:

$$S = (R, H, hidedepth, PC, B)$$

For simplicity, we do not model memory. Instead, we assume all secrets are held in registers directly (protecting against ① in Section 4.3). Our analysis would follow a similar structure if memory was modeled as well.

The speculative machine can support the following operations, whose precise semantics are defined in Appendix A:

1) `hide` – *Register Hide* step: saves the current register state R into H, and increments $hidedepth$.
2) `restore` – *Register Restore* step: restores H into R and decrements $hidedepth$ if it has reached the correct target; otherwise resets $R = 0$.
3) `branch`: sets the next PC to its speculative target.
4) $\text{read}_{\mathcal{H}}$: reads from hidden storage H into R.
5) $\text{write}_{\mathcal{H}}$: writes from R or H into H.
6) $\text{nop}_{\mathcal{H}}$: represents any operations that do not interact with H, but may operate on R. There are two subtypes: $\text{nop}_{\mathcal{H}}^{transmitter}$ and $\text{nop}_{\mathcal{H}}^{nontransmitter}$, where the first leaks their operands through a microarchitectural side-channel, and the latter does not.

Following our threat model, the speculative machine can mispredict a branch to any byte-aligned address. Consequently, the speculative machine can fetch and execute instructions from locations that do not correspond to intended instruction boundaries, giving rise to *non-architectural* instructions. In our analysis, we consider binaries that do not contain *architectural* $\text{read}_{\mathcal{H}}$ and $\text{write}_{\mathcal{H}}$ operations.

Sequential machine. The sequential machine has state

$$A = (R, H, PC, B)$$

The sequential machine supports only a subset of operations: `hide`, `restore`, `branch`, and $\text{nop}_{\mathcal{H}}$. In the sequential model, branches always jump to the architecturally correct target, and the machine does not execute $\text{read}_{\mathcal{H}}$ and $\text{write}_{\mathcal{H}}$ operations, as these may only appear as non-architectural instructions in the binary.

6.2. Security goal

Our speculative non-interference (SNI) deviates from prior work [47], [48], particularly in how we define observation traces and establish the mapping relationship between speculative and sequential machines.

Observation traces. We define what can be observed by an attacker when executing the speculative machine and sequential machine described above. Among the operations supported by the machines, we consider `branch` and $\text{nop}_{\mathcal{H}}^{transmitter}$ as potential transmitters and thus their arguments (the target of `branch` and any operands of $\text{nop}_{\mathcal{H}}^{transmitter}$) are included in the observation trace denoted as τ.

An architectural trace τ_A represents the sequence of observations produced when executing the sequential machine $A \xrightarrow{\tau_A}^{*} A'$, starting from an initial architectural state A and reaching a final state A' after multiple steps. Likewise, a speculative trace τ_S is defined as $S \xrightarrow{\tau_S}^{*} S'$.

Speculative non-interference. We instantiate two pairs of machines, where each pair contains a sequential machine and a speculative machine, whose states are denoted as A_1, S_1, A_2, and S_2. We initialize the four machines with identical PC and binary file B. For each pair, we initialize the register state using R_1 and R_2 respectively. Our goal is to show that for any B, R_1 and R_2, if the sequential machine states A_1 and A_2 produce indistinguishable traces, then the speculative machine states S_1 and S_2 do so as well:

Security goal: Speculative non-interference

$$\tau_{A_1} = \tau_{A_2} \Rightarrow \tau_{S_1} = \tau_{S_2}$$

Mapping relationship between speculative and sequential machines. We define a mapping from a speculative machine state to a corresponding sequential machine state. If a speculative machine and an architectural machine are initialized with the same PC, R, and B, the following relationships hold:

- When $hidedepth = 0$, there exists a valid architectural state A where $A.R = S.R$ and $A.PC = S.PC$. This corresponds to the situation when no `hide` operation has occurred or a `restore` has reached the correct architectural target. In this case, speculative and sequential executions converge, and SNI naturally holds.
- When $hidedepth > 0$, there exists a valid architectural state A such that $A.H = S.H_{snapshot}$ and $A.PC = S.PC_{snapshot}$. We focus on this case in our security analysis, since it represents an ongoing misprediction following a `hide`.

6.3. Software requirements

To assist with systematically identifying software requirements, we categorize each program counter (architectural or non-architectural) into one of three classes. This classification captures the control-flow relationships between operations in a control-flow graph, where every indirect branch's outgoing edges may target any byte-aligned address, matching our speculative execution semantics.

- **ReadSafe:** If there exists a $\text{nop}_{\mathcal{H}}$ which consumes the location to which a previous $\text{read}_{\mathcal{H}}$ wrote, or there exists a `hide` operation as a successor of this PC, then a `restore` must be on the path to this successor. Intuitively, *ReadSafe* guarantees that any value in R speculatively restored from the hidden storage will be lost before it can be consumed and potentially leaked by a $\text{nop}_{\mathcal{H}}$, and before it can be consumed by a `hide`.
- **WriteSafe:** If there exists a `restore` operation as a successor of this PC, then a `hide` must be on the path to this successor. Intuitively, *WriteSafe* guarantees that any value speculatively written to the hidden storage H will be overwritten before it can be consumed by a `restore`.
- **Unknown:** A PC is *Unknown* if no security guarantees can be inferred. In this case, interacting with the hidden storage H may result in leakage.

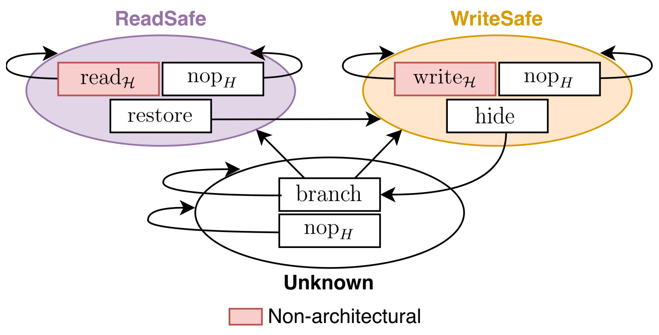

Figure 6: Categorization of PCs in the victim binary with the allowed transitions according to Property 1, Property 2, and their definitions. Arrows indicate allowed transitions between categories.

Having defined these three PC classes, we derive the following two software properties, which must hold for all binaries instrumented with Register Hiding and ShadowCFI, as validated by our binary scanner. The scanner analyzes code byte-by-byte to enforce the properties below on any non-architectural $\text{read}_{\mathcal{H}}$ and $\text{write}_{\mathcal{H}}$ operation.

Property 1:

Every $\text{read}_{\mathcal{H}}$ and its successors up to and including the next `restore` are *ReadSafe*.

Property 2:

Every $\text{write}_{\mathcal{H}}$ and its successors up to and including the next `hide` are *WriteSafe*.

Figure 6 depicts the three PC classes, given their definitions, Property 1, and Property 2. Any $\text{read}_{\mathcal{H}}$, `restore` and $\text{nop}_{\mathcal{H}}$ in between must be *ReadSafe*. Any $\text{write}_{\mathcal{H}}$, `hide` and $\text{nop}_{\mathcal{H}}$ in between must be *WriteSafe*.

A PC within some class may be followed by any other PC in the same class, except for two cases. First, a PC that decodes as a `hide` forces the next PC to be *Unknown* (`branch`). Second, a PC that decodes as a `restore` forces the next PC to be *WriteSafe*. Note that by definition of *ReadSafe* and *WriteSafe*, a `branch` operation must be *Unknown*, as such operation poses no restriction on the next PC to be executed. Likewise, any operation which is *ReadSafe* may not be *WriteSafe* and vice versa, due to conflicting definitions.

6.4. Overview of inductive reasoning

We use induction to show that the software requirements defined above are sufficient to ensure speculative non-interference (SNI) on our speculative machine model. The key invariant in our induction is register equivalence, which states that for all program counters not classified as *ReadSafe*, the register states of S_1 and S_2 remain equivalent and therefore cannot leak information. Additional helper invariants and the inductive steps are provided in Appendix A.

Invariant 1: Register equivalence

$$(\text{PC} \notin \textit{ReadSafe} \land \text{hidedepth} > 0) \Rightarrow R_{S_1} = R_{S_2}$$

Invariant 1, Property 1 and our mapping relation together show that our security goal holds.

- If $hidedepth > 0$ and the PC is not *ReadSafe*, the two register states of the speculative machine are equal (Invariant 1), and thus $\tau_{S_1} = \tau_{S_2}$.
- If $hidedepth > 0$ and the PC is *ReadSafe*, there exists no $\text{nop}_{\mathcal{H}}$ which consumes a value restored from the hidden storage (Property 1), and thus $\tau_{S_1} = \tau_{S_2}$.
- Lastly, if $hidedepth = 0$, $\tau_{S_1} = \tau_{A_1}$ and $\tau_{S_2} = \tau_{A_2}$ (mapping relation). Since the property assumes $\tau_{A_1} = \tau_{A_2}$, we have $\tau_{S_1} = \tau_{S_2}$.

6.5. Gadget validation

Having identified the software requirements necessary to achieve speculative non-interference, we now proceed to validate Properties 1 and 2 by statically scanning from `_text` to `_end` (our configuration has no loadable modules), verifying that every read is *ReadSafe*, and every write is *WriteSafe*. We discharge all cases that could be verified automatically, and manually check the remainder. To reduce the number of non-architectural SSE-reading and SSE-writing instructions, we insert `nop` instructions at some locations that non-architecturally decode as a SSE-read or SSE-write instruction. While our induction in the previous subsections disregard memory for simplicity, our scanner *does* take memory into account to ensure full coverage of potential gadgets.

Table 1 shows the results for our configuration. We find 20 non-architectural instructions that read from SSE registers, of which one is followed by bytes that do not successfully decode. The remaining 19 sites are manually inspected, which shows that none of them can be followed by an instruction that consumes the read value, prior to the next *Register Restore*. In particular, most of them either write to addresses which are not read afterwards (e.g., `0x4801e983(%rax)`), or write to registers which

	Total	**Auto. *Safe***	**Manual *Safe***	**Not *Safe***
SSE read	20	1	19	0
SSE write	785	785	0	0

TABLE 1: Validation results of REGISTER HIDING and SHADOWCFI using non-architectural instruction analysis. *Safe* indicates *ReadSafe* for SSE reads, and *WriteSafe* for SSE writes.

are afterwards overwritten. Likewise, all of the 785 non-architectural instructions that write to SSE registers are *WriteSafe*. That is, we automatically determine that none of the SSE writes can be speculatively followed by *Register Restore* prior to the next *Register Hide*. Upon executing a *Register Restore*, the SSE registers are thus guaranteed to contain the values hidden by the previous *Register Restore*. Note that KASLR causes some immediates in the text section to differ across boots, leading to slight variations in the numbers presented in Table 1.

We successfully validated the security of our defense: there should be no way to restore and use the architectural register state anywhere except at the correct branch target.

7. Implementation

We implement a proof-of-concept of REGISTER HIDING and SHADOWCFI as a patch for Linux kernel version `6.8.0`. Below we discuss some implementation details.

7.1. Functionality

GCC changes. To support REGISTER HIDING, we patch GCC version `14.0.0` to insert the *Register Hide* step before each indirect branch. Furthermore, we insert the *Register Restore* step at all targets of indirect calls and indirect jumps. Likewise, each return target is followed by our *Register Restore* step.

Manual patching. To support BPF, its framework is patched with instrumented instructions for REGISTER HIDING and SHADOWCFI. Furthermore, the Linux kernel contains static calls which are patched in during boot. To support REGISTER HIDING and SHADOWCFI for these calls as well, the static calls need to be patched in with instrumentation. Lastly, assembly code is not automatically instrumented using GCC, and need to be patched manually.

Patching *Register Restore*. To support SHADOWCFI, we need to patch the *Register Restore* steps with their unique targets, as shown in Figure 5. For this, we use self-modifying code using the alternatives framework, allowing us to patch each *Register Restore* step during boot time.

Using SSE freely in the kernel. Before using SSE registers, kernel code is required to call an API (`kernel_fpu_begin()`). This call has significant overhead, since it stores the entire FPU register state, and starts non-preemptive code execution. To freely use SSE registers, we patch the entry points of the kernel to push the SSE register state onto the stack. Likewise, the exit paths are patched to restore the SSE register state by popping from the stack. To furthermore implement support for virtual machines (VMs), we patch KVM to store the SSE registers in memory upon exiting the VM, allowing us to restore them when re-entering the VM.

Interrupt support. An interrupt may occur while the architectural register state is hidden. This has three implications. First, we need to save the SSE registers on the stack, to ensure we can safely overwrite them by executing indirect branches inside the interrupt. Second, while hiding the architectural register state, `RSP` is set to a hidden stack, as explained in Section 5.2. To ensure we do not overwrite data pushed upon interrupt entry (e.g., using nested interrupts), we switch back to the architectural stack as soon as an interrupt occurs, moving already-pushed data. Third, interrupts often happen in bursts. To prevent having to switch off the hidden stack repeatedly, we restore all registers on the kernel thread originally interrupted. Note that this requires moving the instruction pointer of the original kernel thread passed the indirect branch that was intended to be executed after the *Register Hide* step completes, as otherwise an indirect branch executes with restored registers after the interrupt finishes. We can find the target in the hidden storage, and update the instruction pointer accordingly.

7.2. Performance trade-offs

Callee-saved registers. While *Register Hide* and *Register Restore* need to save and restore all registers upon indirect jumps and indirect calls, they only need to save and restore callee-saved registers for return instructions. Since return instructions are the most common type of indirect branch, we patch Linux to turn `R12`-`R15` into caller-saved registers instead. This significantly reduces the size of the *Register Hide* and *Register Restore* steps for return instructions. Note that we always clear the entire architectural register state.

Global Register Hide. Instead of adding `movq` and `xor` instructions prior to every indirect branch in the kernel, we opt for a single *Global Register Hide* step to which we branch before every indirect branch. Note that this branch is direct and thus does not require protection itself, and the return of this function is executed with a cleared register state. Furthermore, the indirect branch remains inlined, supporting accurate prediction. This significantly decreases the kernel's code footprint.

Global Register Restore. The *Register Restore* step is short for targets of returns, especially with a reduced number of callee-saved registers. However, for targets of other indirect branches, the *Register Restore* steps increase the kernel's code footprint significantly.

On some microarchitectures, it may be possible to ensure safe speculation for indirect branches at certain addresses.

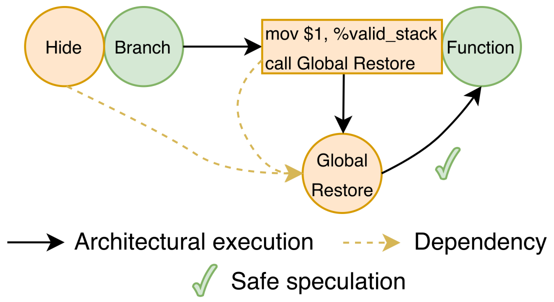

Figure 7: We have a single *Global Register Restore* for indirect branches which are not returns. `valid_stack` is an SSE register. The return in the *Global Register Restore* is untrained upon kernel entry.

On AMD CPUs, it is possible to untrain a particular return in the kernel upon entry, guaranteeing it is free of attacker influence [18], [17]. Therefore, we call into a *Global Register Restore* function at targets of indirect branches which are not returns. Before this branch, we move a value into `valid_stack` (part of hidden storage) which indicates that the top-of-the-stack value was pushed by a real call to the *Global Register Restore*. This facilitates ShadowCFI without a baked-in target (as in Figure 5). That is, the *Global Register Restore* will compare `target` from the *Register Hide* step against `*RSP` to create a mask if `valid_stack` has the expected value (else the mask is 0). This technique ensures we do not violate our security validation presented in Section 6.

Figure 7 depicts how the *Global Register Restore* is called to reduce the instruction footprint of the kernel. By executing the *Global Register Restore* once at kernel entry with a cleared register state, we untrain its return instruction, ensuring safe speculation back to the target after restoring the architectural state. Note that unlike the *saferet* [18] mitigation, our defense allows the return in the *Global Register Restore* to be predicted correctly using the RSB, while still ensuring its security.

8. Evaluation

Having built a proof-of-concept of Register Hiding and ShadowCFI, we now proceed to evaluate its security and performance.

8.1. Experiment setup

Experiments are conducted on a dual-socket system equipped with two AMD EPYC 9124 16-core processors (Zen 4), providing a total of 32 physical cores and 64 hardware threads. The system was configured with 512 GB of DDR5-4800 memory and ran Ubuntu 22.04.4 LTS. We use microcode version `0x0a101148`. We consider three different kernel configurations:

① **Insecure baseline**: we compile Linux `6.8.0` and boot it with `mitigations=off`. This makes the kernel

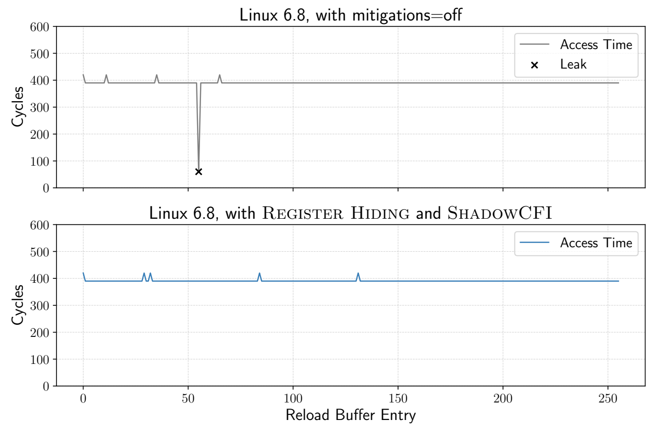

Figure 8: Results of performing an Inception attack on AMD Zen 4, leaking a single byte from a kernel module.

vulnerable to attacks like Spectre v2 [1] and Inception [4].

② **Default mitigations**: we compile Linux `6.8.0` and boot it normally. This enables default mitigations against Spectre attack, which includes AutoIBRS [15] for indirect branches, RSB stuffing for returns, and *saferet* [18] for returns. Recent reverse-engineering work revealed that AutoIBRS prevents speculative execution due to indirect branches in the kernel completely [2], while *safefret* forcibly mispredicts returns.

③ **REGISTER HIDING + SHADOWCFI**: we compile Linux `6.8.0` with our defense using REGISTER HIDING and SHADOWCFI. We instrument and protect all indirect branches, and so AutoIBRS and *saferet* are disabled. We leave RSB stuffing enabled, since according to documentation, it may help to block user-space to user-space attacks.

We consider ② and ③ to be similar in security guarantees. However, we note that ③ is better prepared for futuristic attacks, since it does not assume which predictor is used by the attacker to trigger a misprediction, and how these predictors work. For example, *saferet* assume very specific behavior of the branch predictor, including the exact predictor which is used, the behavior of the RSB and the indexing function of the BTB.

Intel. We furthermore evaluate the performance overhead of complementing IBT with SHADOWCFI on a CET-enabled machine. We conduct experiments on an Intel Core Ultra 9 285H 16-core processor (Meteor Lake). The system was equipped with 64GB of DDR5-5600 memory, running Ubuntu 22.04.4 LTS (microcode version `0x00000118`).

We compile Linux `6.8.0` with IBT and SHADOWCFI, and keep default mitigations enabled (eIBRS and RSB stuffing). While IBT limits the speculation window of indirect branch predictions (jumps and calls) to targets missing an `endbr`, SHADOWCFI furthermore clears the register state at any incorrect `endbr`-target.

8.2. Security evaluation

We evaluate the effectiveness of REGISTER HIDING and SHADOWCFI against an Inception attack [4] on our AMD Zen 4 machine. For simplicity, we insert a custom victim kernel module, with known addresses and register allocation. The attacker attempts to leak a single byte (55). The attacker first inserts the PhantomJMP and recursive PhantomCALL from user-space, and then calls the kernel module. This executes a return in kernel space, triggering transient execution of a disclosure gadget that loads a secret-dependent entry in the user's reload buffer. Back in user space, the attacker reloads all of the 256 entries in its reload buffer, measuring the latency to determine the secret.

Figure 8 shows the results on an unprotected kernel (top), and a kernel protected with REGISTER HIDING and SHADOWCFI (bottom). Clearly, the attacker can deduce the secret on the unprotected kernel. However, in the protected kernel, the disclosure gadget executes with a cleared register state, preventing secret reachability.

8.3. Performance evaluation

For performance evaluation, we disable turbo boost and configure the CPU frequency governor to `performance`, fixing the frequency range between 1.5 GHz and 3.0 GHz (AMD) and between 0.4 GHz and 2.9 GHz (Intel). Simultaneous Multithreading (SMT) remains enabled throughout all experiments.

LEBench. We first evaluate the performance using LEBench [49]. We leave the benchmark settings as is, but report the median across all iterations instead of the average, since we noticed that outliers can make the average unstable between multiple runs.

Figure 9 shows unequivocally that REGISTER HIDING + SHADOWCFI performs better than the default mitigations on Linux. Over all tests, we reduce the average performance overhead from 114.1% to 75.9%, while relying on software-only solutions without hardware support.

To further understand this performance gain, we use `perf` to sample performance counters while executing LEBench. We determine that while we commit twice as many instructions on the REGISTER HIDING + SHADOWCFI kernel during the execution on LEBench, our throughput is *more than doubled* compared to default mitigations. The latter can be mostly attributed to the branch misprediction rate: default mitigations trigger almost 10x more mispredictions than REGISTER HIDING + SHADOWCFI. This can be easily explained by the fact that the recent *saferet* mitigation forcefully mispredicts all returns in the kernel.

On our Intel CPU, SHADOWCFI shows an average LEBench overhead of 13.5% compared to only default mitigations.

Server workloads. In addition to LEBench, we benchmark nginx, httpd (Apache), memcached and redis on the

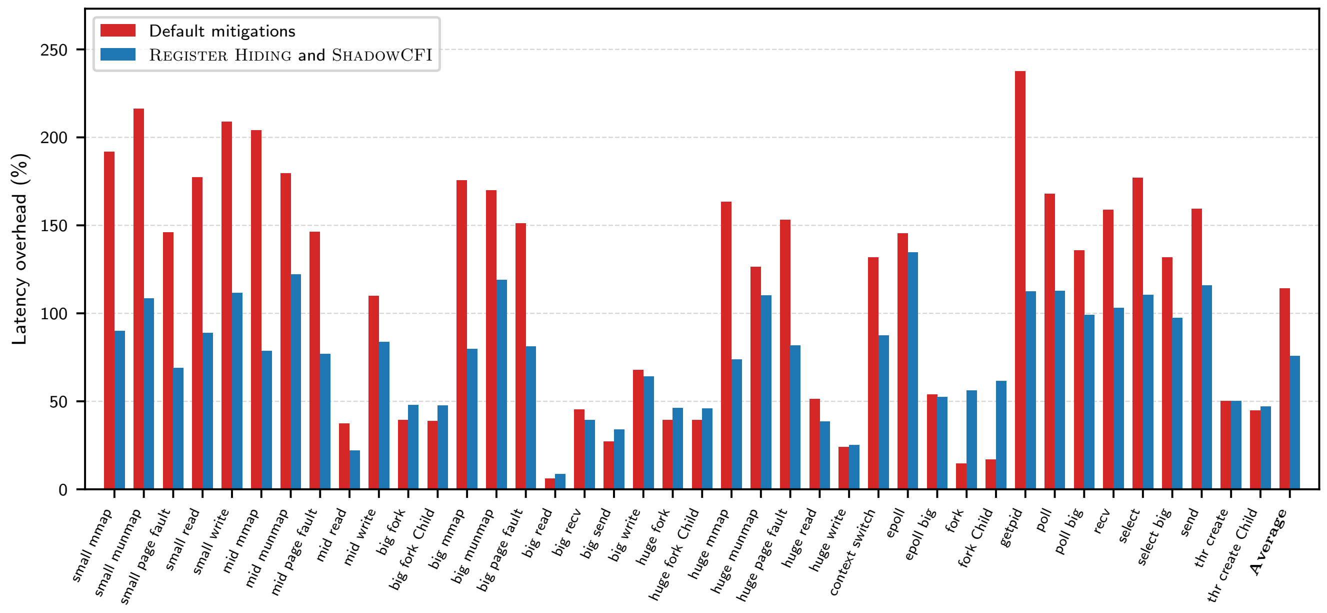

Figure 9: REGISTER HIDING and SHADOWCFI show an average performance overhead of 75.9% on LEBench, while default mitigations have an overhead of 114.1%.

REGISTER HIDING + SHADOWCFI kernel and on a kernel with default mitigations. To reduce noise, we use `numactl` to pin threads and memory of the client and server to separate NUMA nodes. We use the same parameters as previous profiling work [50], but additionally increase the number of requests to 1M for each service. As in previous work, we use `ab` to benchmark ngninx and httpd, and `redis-benchmark` for redis. To benchmark memcached, we use `memtier-benchmark` as a client [51].

Figure 10 shows the results, again proving superior performance of REGISTER HIDING and SHADOWCFI compared to default mitigations. On average, we have a 25.8% overhead for the kernel protected with REGISTER HIDING and SHADOWCFI, compared to 33.4% for the kernel with default mitigations. In particular, nginx and httpd have an overhead of 29.8% and 27.8% on our kernel, compared to 38.6% and 36.7% respectively on the kernel with default mitigations. The number of requests per second on the baseline are 29.9K and 26.1K respectively. memcached and redis have an overhead of 21.3% and 24.4%, compared to 26.9% and 31.4% on the kernel with default mitigations. The number of requests per second on the baseline are 89.4K and 81.4K, for memcached and redis respectively.

On our Intel CPU, SHADOWCFI incurs an additional average overhead of 8.9% compared to just default mitigations for the server workloads. Specifically, nginx and httpd show an overhead of 11.5% and 10.8%, respectively, while memcached and redis show overheads of 6.4% and 7%.

Alternative hidden storage. As mentioned in Section 5.4, memory could be used instead of SSE registers to implement the hidden storage. To estimate the performance overhead

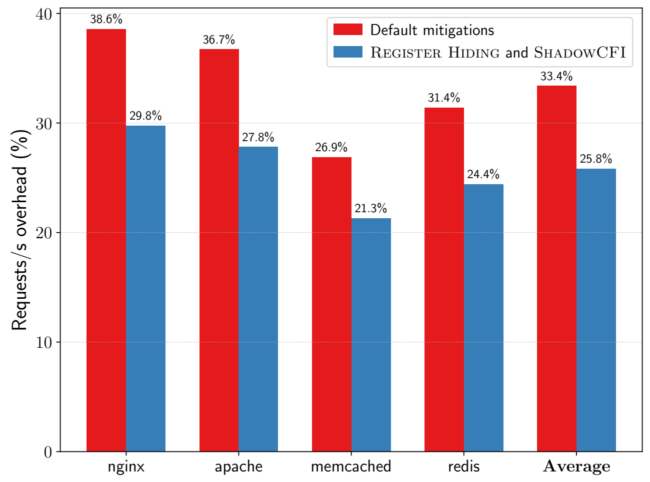

Figure 10: Performance impact of REGISTER HIDING and SHADOWCFI on server applications, compared to Linux's default mitigations.

of this alternative implementation, we compare REGISTER HIDING using memory (push/pop) and SSE registers with a micro-benchmark. Specifically, for one billion iterations, we 1) save registers in the hidden storage, 2) clear all registers, 3) execute an indirect branch, and 4) restore the registers.

Our results show that the SSE-variant takes on average 23.11 cycles per iteration for indirect jumps and indirect calls, while returns (which save and restore fewer registers) only take 8.03 cycles. The memory-variant takes 16.06 (-30.50%) and 6.49 cycles (-19.10%) respectively.

We thus believe a memory-based variant to be performant. However, implementing a verifiably-secure version of a memory-based variant may be more difficult. First, a reference to the hidden storage memory location needs to be held somewhere to accommodate *Register Restore*, which should not be used during misprediction. Furthermore, we would not be able to limit our security analysis to a well-defined set of instructions that interact with the hidden storage locations, as done in Section 6.

9. Related work

Spectre attacks have prompted extensive research into mitigation strategies that target various stages of the attack pipeline. While some propose principled ways of preventing Spectre attacks, those that have made it to real-world hardware and software have been mostly specific to attack variants and victim microarchitectures, or involved hardware changes.

Spectre mitigations. A large number of software mitigations against Spectre attacks were introduced in recent years, including barriers such as `lfence` to limit the speculation window [12]. Likewise, retpoline [13], RSB stuffing [14], jmp2ret [17] and saferet [18] forcefully trigger benign mispredictions, preventing an attacker from influencing the prediction. Other works propose hardware changes to prevent secrets from being transmitted under speculation [21], [22], [23], [24], [25], [41]. Most of these mitigations target gadget reachability or secret transmittability. In this paper, we target a new ingredient, *secret reachability*, and show how we can use this to provide a software mitigation against Spectre attacks, while being agnostic to the attack variant and underlying hardware.

Speculative control-flow integrity. Intel's Control-flow Enforcement Technology (CET) forces indirect branches (except returns) to be followed by an `endbr` instruction. While mostly meant for software attacks, Intel makes claims about the number of instructions that may execute under speculation before a missing `endbr` is detected [16]. FineIBT uses CET to provide more fine-grained control-flow integrity [36], and shows that it only leaves room for a very small speculation window. SpecCFI is a CFI mechanism designed specifically for transient execution attacks [52]. Lastly, Swivel uses register interlocking, masking the heap- and stack pointer on mispredicted paths in WebAssembly [45]. As SHADOWCFI, it does so by creating a dependency on the correct target or branch condition, with the goal of preventing memory accesses, whereas SHADOWCFI prevents access to the architectural register state.

However, both FineIBT and SpecCFI require overapproximation of potential branch targets, which are inferred using static analysis. That is, they are not fully context-sensitive. Furthermore, FineIBT and Swivel's register interlocking require CET (Intel only) and SpecCFI even requires hardware changes. In constrast, SHADOWCFI is a context-sensitive CFI, not requiring any overapproximation nor any hardware support.

Secret reachability. Previous work has accomplished limiting secret reachability for Spectre v1, using a technique called speculative load hardening or index masking [46]. By limiting the attacker's control over register values under speculation, arbitrary secrets are prevented from being reached. While Spectre v1 attacks trigger mispredictions to known targets, Spectre v2-style attacks can trigger mispredictions to arbitrary targets, increasing the complexity of limiting secret reachability significantly.

10. Conclusion

Transient execution attacks that target indirect branches remain a serious issue today. Currently deployed mitigations are microarchitecture-dependent, attack-dependent or require hardware support. In this paper, we are the first to propose to limit secret reachability, preventing disclosure gadgets from consuming desired secrets. We introduce REGISTER HIDING and SHADOWCFI, two independent techniques which together form a software-only and hardware-agnostic defense against transient execution attacks. We validate our defense using a security analysis and corresponding scanner, showing that the kernel is free of any gadget that could bypass our defense, even considering non-architectural instructions. Furthermore, REGISTER HIDING and SHADOWCFI have significantly better performance than mitigations enabled today in the Linux kernel by default. We believe this work offers immediate applicability, as well as inspiration for future defenses against the evolving landscape of transient execution attacks.

Acknowledgments

We thank our anonymous shepherd and reviewers for their valuable feedback, and Shixin Song for her help on the inductive analysis. This work was supported by the Department of Defense (DoD) through the National Defense Science & Engineering Graduate (NDSEG) Fellowship Program, and by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," *Communications of the ACM*, vol. 63, no. 7, pp. 93–101, 2020.

[2] S. Wiebing and C. Giuffrida, "Training solo: On the limitations of domain isolation against spectre-v2 attacks," in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2025, pp. 3599–3616.

[3] J. Wikner and K. Razavi, "RETBLEED: Arbitrary speculative code execution with return instructions," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3825–3842.

[4] D. Trujillo, J. Wikner, and K. Razavi, "Inception: Exposing new attack surfaces with training in transient execution," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 7303–7320.

[5] J. Wikner, D. Trujillo, and K. Razavi, “Phantom: Exploiting decoder-detectable mispredictions,” in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 49–61.

[6] E. Barberis, P. Frigo, M. Muench, H. Bos, and C. Giuffrida, “Branch history injection: On the effectiveness of hardware mitigations against cross-privilege spectre-v2 attacks,” in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 971–988.

[7] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, “Spectre returns! speculation attacks using the return stack buffer,” in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, 2018.

[8] S. Rüegge, J. Wikner, and K. Razavi, “Branch privilege injection: Compromising spectre v2 hardware mitigations by exploiting branch predictor race conditions,” USENIX Security Symposium, 2025, Aug. 2025, [Online]. Available: https://comsec.ethz.ch/wp-content/files/bprc_sec25.pdf.

[9] G. Maisuradze and C. Rossow, “ret2spec: Speculative execution using return stack buffers,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2109–2122.

[10] J. Behrens, A. Cao, C. Skeggs, A. Belay, M. F. Kaashoek, and N. Zeldovich, “Efficiently mitigating transient execution attacks using the unmapped speculation contract,” in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 1139–1154.

[11] T. H. Kim, D. Rudo, K. Zhao, Z. N. Zhao, and D. Skarlatos, “Perspective: A principled framework for pliable and secure speculation in operating systems,” in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2024, pp. 739–755.

[12] Intel Corporation, “Speculative execution side channel mitigations,” 2018, [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/speculative-execution-side-channel-mitigations.html.

[13] P. Turner, “Retpoline: a software construct for preventing branch-target-injection,” 2018, [Online]. Available: https://support.google.com/faqs/answer/7625886.

[14] Intel Corporation, “Post-barrier return stack buffer predictions / cve-2022-26373 / intel-sa-00706,” 2022, [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/post-barrier-return-stack-buffer-predictions.html.

[15] K. Phillips, “[patch 0/3] x86/speculation: Support automatic ibrs,” 2022, [Online]. Available: https://lore.kernel.org/lkml/20221104213651.141057-1-kim.phillips@amd.com/T/.

[16] Intel Corporation, “Indirect branch restricted speculation,” 2018, [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-restricted-speculation.html.

[17] A. M. Devices, “Technical guidance for mitigating branchtype confusion,” 2022, [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/resources/technical-guidance-for-mitigating-branch-type-confusion.pdf.

[18] L. K. A. Guide, “Speculative return stack overflow (srso),” 2023, [Online]. Available: https://docs.kernel.org/admin-guide/hw-vuln/srso.html.

[19] Intel Corporation, “Branch history injection and intra-mode branch target injection / cve-2022-0001, cve-2022-0002 / intel-sa-00598,” 2024, [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/branch-history-injection.html.

[20] J. Corbet, “A call to reconsider address-space isolation,” 2022, [Online]. Available: https://lwn.net/Articles/909469/.

[21] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, “Invisispec: Making speculative execution invisible in the cache hierarchy,” in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 428–441.

[22] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, “Understanding selective delay as a method for efficient secure speculative execution,” *IEEE Transactions on Computers*, vol. 69, no. 11, pp. 1584–1595, 2020.

[23] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, “Speculative taint tracking (stt) a comprehensive protection for speculatively accessed data,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 954–968.

[24] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, “Safespec: Banishing the spectre of a meltdown with leakage-free speculation,” in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019, pp. 1–6.

[25] S. Ainsworth, “Ghostminion: A strictness-ordered cache system for spectre mitigation,” in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 592–606.

[26] AMD, “Software optimization guide for the amd zen4 microarchitecture,” 2020, accessed on 05.11.2025.

[27] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, “Jump over aslr: Attacking branch predictors to bypass aslr,” in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–13.

[28] T. Zhang, K. Koltermann, and D. Evtyushkin, “Exploring branch predictors for constructing transient execution trojans,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 667–682.

[29] P. Zijlstra and T. Gleixner, “[patch v3 00/59] x86/retbleed: Call depth tracking mitigation,” 2022, [Online]. Available: https://lkml.org/lkml/2022/9/15/427.

[30] A. Milburn, K. Sun, and H. Kawakami, “You cannot always win the race: Analyzing the lfence/jmp mitigation for branch target injection,” *arXiv preprint arXiv:2203.04277*, 2022.

[31] S. Wiebing, A. de Faveri Tron, H. Bos, and C. Giuffrida, “Inspectre gadget: Inspecting the residual attack surface of cross-privilege spectre v2,” in *USENIX Security*, 2024.

[32] AMD, “Software techniques for managing speculation on amd processors,” 2022, [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/resources/software-techniques-for-managing-speculation.pdf.

[33] Intel Corporation, “Retpoline: A branch target injection mitigation,” 2022, [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-restricted-speculation.html.

[34] ——, “Indirect branch predictor delayed updates,” 2025, [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/indirect-branch-predictor-delayed-updates.html.

[35] ——, “Control-flow enforcement technology specification,” 2019, accessed on 05.12.2025.

[36] A. J. Gaidis, J. Moreira, K. Sun, A. Milburn, V. Atlidakis, and V. P. Kemerlis, “Fineibt: Fine-grain control-flow enforcement with indirect branch tracking,” in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, 2023, pp. 527–546.

[37] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, “Smotherspectre: exploiting speculative execution through port contention,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 785–800.

[38] B. Johannesmeyer, J. Koschel, K. Razavi, H. Bos, and C. Giuffrida, "Kasper: Scanning for generalized transient execution gadgets in the linux kernel." in *NDSS*, vol. 1, 2022, p. 12.

[39] A. S. Jordy Zomer, "Finding gadgets for cpu side-channels with static analysis tools," 2023, [Online]. Available: https://github.com/google/security-research/blob/master/pocs/cpus/spectre-gadgets/README.md.

[40] M. Schwarzl, C. Canella, D. Gruss, and M. Schwarz, "Specfuscator: Evaluating branch removal as a spectre mitigation," in *Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, March 1–5, 2021, Revised Selected Papers, Part I 25*. Springer, 2021, pp. 293–310.

[41] M. Hertogh, M. Wiesinger, S. Österlund, M. Muench, N. Amit, H. Bos, and C. Giuffrida, "Quarantine: Mitigating transient execution attacks with physical domain isolation," in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, 2023, pp. 207–221.

[42] E. Göktas, K. Razavi, G. Portokalidis, H. Bos, and C. Giuffrida, "Speculative probing: Hacking blind in the spectre era," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1871–1885.

[43] A. Bhattacharyya, A. Sánchez, E. M. Koruyeh, N. Abu-Ghazaleh, C. Song, and M. Payer, "SpecROP: Speculative exploitation of ROP chains," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, 2020, pp. 1–16.

[44] M. Hertogh, S. Wiebing, and C. Giuffrida, "Leaky address masking: Exploiting unmasked spectre gadgets with noncanonical address translation," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2024, pp. 158–158.

[45] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen *et al.*, "Swivel: Hardening WebAssembly against spectre," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1433–1450.

[46] F. Pizlo, "What spectre and meltdown mean for webkit," 2018, [Online]. Available: https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/.

[47] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila, "Hardware-software contracts for secure speculation," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1868–1883.

[48] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "Spectector: Principled detection of speculative information flows," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1–19.

[49] X. Ren, K. Rodrigues, L. Chen, C. Vega, M. Stumm, and D. Yuan, "An analysis of performance evolution of linux's core operations," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 554–569.

[50] M. Ugur, C. Jiang, A. Erf, T. Ahmed Khan, and B. Kasikci, "One profile fits all: Profile-guided linux kernel optimizations for data center applications," *ACM SIGOPS Operating Systems Review*, vol. 56, no. 1, pp. 26–33, 2022.

[51] RedisLabs, "memtier_benchmark: A high-throughput benchmarking tool for redis and memcached," 2023, [Online]. Available: https://github.com/RedisLabs/memtier_benchmark.

[52] E. M. Koruyeh, S. H. A. Shirazi, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Speccfi: Mitigating spectre attacks using cfi informed speculation," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 39–53.

Appendix A. Inductive analysis

In this appendix, we present our induction used to argue that our defense satisfies speculative non-interference, continuing Section 6. We have two speculative machines with states $S_1 = (R_{S_1}, H_{S_1}, hidedepth, PC)$ and $S_2 = (R_{S_2}, H_{S_2}, hidedepth, PC)$. Note that we use $hidedepth$ and PC for both machines, as they never diverge.

The operations that may be performed by the machines are shown in Operations 1, which includes mechanisms to support re-steering speculative execution. Whenever the machine speculates to an incorrect target, $misspec$ is set. Furthermore, the machine sets $PC_{shapshot}$ and $H_{snapshot}$ to the correct target and current hidden storage H, respectively. At any moment, the machine may choose to `resteer`, which resets $misspec$, and sets the PC to the correct destination stored in $PC_{snapshot}$, along with the correct hidden storage state $H_{snapshot}$. In addition, $hidedepth$ is set back to 1, which it was guaranteed to be before misprediction. For the purpose of our induction, we ignore re-steering as a possible machine action. Our goal is to use induction to show Invariant 1:

Invariant 1: Register equivalence

$$(\text{PC} \notin \textit{ReadSafe} \wedge \text{hidedepth} > 0) \Rightarrow R_{S_1} = R_{S_2}$$

To help show this invariant holds, we furthermore introduce three helper invariants. First, if $hidedepth > 1$, the hidden storage of S_1 and S_2 are equivalent as well. Second, $hidedepth$ is always greater than 1 if the PC is not *WriteSafe*, or greater than 0 if the PC is *WriteSafe*. Intuitively, this means that the number of `restore` operations performed never exceeds the number of `hide` operations executed.

Invariant 2: Hidden storage equivalence

$$\text{hidedepth} > 1 \Rightarrow H_{S_1} = H_{S_2}$$

Invariant 3: Hidedepth positivity 1

$$\text{PC} \notin \textit{WriteSafe} \Rightarrow \text{hidedepth} \geq 1$$

Invariant 4: Hidedepth positivity 2

$$\text{PC} \in \textit{WriteSafe} \Rightarrow \text{hidedepth} \geq 0$$

Initialization. We start with $hidedepth = 0$, $H_{S_1} = 0$, and $H_{S_2} = 0$. The starting PC may be any *WriteSafe* PC that constitutes an architectural instruction boundary, since `hide` is expected to execute before `branch` or `restore`.

Base step. Invariant 1 (Register equivalence) and Invariant 2 (Hidden storage equivalence) trivially hold for the initial step S, as their preconditions are unsatisfied. Since $hidedepth = 0$ and $PC \in WriteSafe$, Invariants 3 and 4 (Hidedepth positivity) hold.

Inductive step. We assume that all Invariants hold for S_1 and S_2, and wish to argue that they hold for the next states $S'_k = (R'_{S_k}, H'_{S_k}, hidedepth', PC')$ as well, where $k = 0, 1$. The speculative machines execute PC, which corresponds to an operation $OP = B[PC]$:

Operations 1 Machine operations

```
procedure $\text{read}_H(i, j)$
    $R[i] \leftarrow H[j]$
    $PC = nextPC$
end procedure

procedure $\text{write}_H(i)$
    $H[i] \leftarrow ?$                          ▷ Either from H or R
    $PC = nextPC$
end procedure

procedure branch($target, spec\_target$)
    if $misspec == 0 \wedge target \neq spec\_target$ then
        for $i = 1$ to $n$ do
            $H_{\textsf{snapshot}}[i] \leftarrow H[i]$
        end for
        $PC_{snapshot} = target$
        $misspec = 1$
    end if
    $PC = spec\_target$
end procedure

procedure hide($target, spec\_target$)
    $H[t] \leftarrow target$
    for $i = 1$ to $n$ do
        $H[i] \leftarrow R[i]$
        $R[i] \leftarrow 0$
    end for
    $hidedepth \leftarrow hidedepth + 1$
    $PC = nextPC$
end procedure

procedure restore($target$)
    if $H[t] == target$ then
        for $i = 1$ to $n$ do
            $R[i] \leftarrow H[i]$
        end for
        $hidedepth \leftarrow hidedepth - 1$
    else
        for $i = 1$ to $n$ do
            $R[i] \leftarrow 0$
        end for
    end if
    $PC = nextPC$
end procedure

procedure $\text{nop}_H$
    ...                ▷ May change $R$ independently of H
    $PC = nextPC$
end procedure

procedure resteer
    for $i = 1$ to $n$ do
        $H[i] \leftarrow H_{\textsf{snapshot}}[i]$
    end for
    $hidedepth = 1$          ▷ Will execute restore next
    $misspec = 0$
    $PC = PC_{snapshot}$
end procedure
```

Hidedepth-preserving:

- read_H. According to Property 1, PC is *ReadSafe*, and PC' will be *ReadSafe*. Thus Invariant 1 (Register equivalence) will trivially hold for S'_k, since its precondition is not satisfied. Likewise, since $\texttt{read}_H$ does not alter H_{S_k} or $hidedepth$ (i.e., $H'_{S_k} = H_{S_k}$ and $hidedepth' = hidedepth$), Invariant 2 (Hidden storage equivalence) and Invariant 3 and 4 (Hidedepth positivity) still hold for S'_k.
- write_H. Since `write` does not alter R_{S_k} nor $hidedepth$ (i.e., $R'_{S_k} = R_{S_k}$ and $hidedepth' = hidedepth$), Invariant 1 (Register equivalence) and Invariants 3 and 4 (Hidedepth positivity) still hold for S'_k. If $hidedepth = hidedept' > 1$, then $R_{S_1} = R_{S_2}$ (Invariant 1) and $H_{S_1} = H_{S_2}$ (Invariant 2), and since the write to H'_{S_k} must come from either R_{S_k} or H_{S_k}, Invariant 2 (Hidden storage equivalence) still holds for S'_k. If $hidedepth = hidedepth' \not> 1$, Invariant 2 trivially holds, since its precondition is not satisfied.
- nop_H . This operation updates R'_{S_1} and R'_{S_2} identically, independently of H_{S_k}. Thus, Invariant 1 (Register equivalence) holds for S'_k. Likewise, nop_H does not alter H'_{S_k} or $hidedepth'$, and thus Invariant 2 (Hidden storage equivalence) and Invariants 3 and 4 (Hidedepth positivity) still hold for S'_k.
- `branch`. This operation does not update R'_{S_k}, H'_{S_k} or $hidedepth'$, and thus Invariants 1 and 2 still hold for S'_k. Likewise, since $PC \notin WriteSafe$, $hidedepth' = hidedepth \geq 1$, and Invariants 3 and 4 will hold independently of which class PC' falls in.

Hidedepth-modifying:

- `hide`. Unconditionally, $R' = 0$, and thus Invariant 1 (Register equivalence) holds for S'. If:
  1) $hidedepth > 0$ (and thus $hidedepth' > 1$), then $R_1 = R_2$ (Invariant 1), and since `hide` sets $H' = R$, Invariant 2 (Hidden storage equivalence) is satisfied for S'_k.
  2) $hidedepth = 0$ (and thus $hidedepth' = 1$), Invariant 2 trivially holds for S'_k due to an unsatisfied precondition.

  Invariant 3 and Invariant 4 hold for S'_k, since $hidedepth \geq 0$, and $hidedepth' = hidedepth + 1 \geq 1$.
- `restore`. There are two cases:
  1) $H[t] \neq target$. Since $R'_{S_k} = 0$, Invariant 1 (Register equivalence) holds for S'_k.
  2) $H[t] = target$. There exists again two cases ($hidedepth \geq 1$ due to Invariant 3):
     a) $hidedepth > 1$. Since $H_{S_1} = H_{S_2}$ (Invariant 2), $R'_{S_1} = R'_{S_2}$ after restoration, and Invariant 1 (Register equivalence) holds for S'_k.
     b) $hidedepth = 1$. Invariant 1 (Register equivalence) trivially holds as since its precondition is not satisfied ($hidedepth' = 0$).

  Invariant 2 holds for S'_k as H'_{S_k} is unaltered from H_{S_k}. Since PC' will be *WriteSafe*, Invariant 3 trivially holds. Likewise, since $hidedepth' = hidedepth \geq 1$ or $hidedepth' = hidedepth - 1 \geq 0$, Invariant 4 holds.

Appendix B.
Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

The paper presents a new defense against Spectre-v2 style attacks that relies on limiting secret reachability under speculation. For this, the authors combine two new techniques: (1) RegisterHiding, which saves the content of the register state in a shadow location whenever speculation happens (and wipes out the actual state), and (2) ShadowCFI, which ensures that the shadow register state can be restored only at ”architectural” jump targets. The authors implement a prototype of their defense and apply it to secure the Linux kernel, and they compare the performance overhead w.r.t. other Spectre-v2 mitigations. Moreover, they also provide a security analysis of the new defense.

B.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Addresses a Long-Known Issue
- Provides a Valuable Step Forward in an Established Field

B.3. Reasons for Acceptance

1) This paper addresses a long-known issue and provides a valuable step forward in an established field. Preventing Spectre-v2 attacks at software-level is still an open problem. The paper provides an interesting compiler-level defense against Spectre v2 based on limiting secret reachability under speculation.
2) The paper creates a new tool to enable future science. The authors provide a full implementation of the proposed mitigation and promise to make it available as open-source. This will enable developers to benefit from increased security and it will allow researchers to use it for reproducing the paper’s results and as a basis for future research on Spectre countermeasures.

B.4. Noteworthy Concerns

1) The security analysis presented in §5.4, which attempts to justify the mitigation’s security guarantees, is unconvincing due to its informal and imprecise nature. Many core concepts are fuzzy and require further explanation (trace models, machine semantics and operations). In particular, the proposed mitigation would benefit from a formal proof of security precisely characterizing its security guarantees.