Eclipse: Preventing Speculative Memory-error Abuse with Artificial Data Dependencies

Neophytos Christou
Brown University
Providence, RI, USA
neophytos_christou@brown.edu

Alexander J. Gaidis
Brown University
Providence, RI, USA
agaidis@cs.brown.edu

Vaggelis Atlidakis
Brown University
Providence, RI, USA
eatlidak@cs.brown.edu

Vasileios P. Kemerlis
Brown University
Providence, RI, USA
vpk@cs.brown.edu

Abstract

Historically, researchers have treated memory safety-based and speculative execution attacks as two separate domains. Recent work has introduced Speculative Memory-error Abuse (SMA) attacks, which combine memory corruption vulnerabilities with Spectre-like primitives. Using SMA, an attacker can leak sensitive program information and defeat a wide variety of memory-corruption mitigations, including (K)ASLR, software-based XOM, and even ARM PA, eventually carrying out an end-to-end (architecturally-visible) exploit. We present `Eclipse`: a novel protection scheme against SMA attacks. `Eclipse` works by propagating artificial data dependencies onto sensitive data, preventing the CPU from using attacker-controlled data during speculative execution. We demonstrate that `Eclipse` provides comprehensive protection against speculative-probing and PACMAN-style attacks, two prominent examples of SMA attacks that target both the x86(-64) and ARM architectures. We evaluate the performance of `Eclipse` on x86-64 and demonstrate that it introduces minimal overhead, compared to alternative hardening approaches, incurring ≈0%–9.5% slowdown on SPEC CPU 2017, up to 8.6% slowdown in real-world applications, and negligible overhead in the Linux kernel.

CCS Concepts

• **Security and privacy → Systems security**; **Operating systems security**; **Software security engineering**.

Keywords

speculative probing, memory errors, artificial data dependencies

ACM Reference Format:
Neophytos Christou, Alexander J. Gaidis, Vaggelis Atlidakis, and Vasileios P. Kemerlis. 2024. Eclipse: Preventing Speculative Memory-error Abuse with Artificial Data Dependencies. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3658644.3690201

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3690201

1 Introduction

Memory errors have been a prevalent software security problem for decades and are still amongst the most prominent software defects [38, 77, 97]. By corrupting sensitive program data, attackers can achieve *control-flow hijacking* using various code-reuse [16] techniques and eventually perform arbitrary computations (usually via means of arbitrary code execution).

More recently, the Spectre family of attacks [55] have shown that an adversary can leverage *speculative execution* [41] to access sensitive program data that is not accessible in an architecturally-visible manner by bringing it into micro-architectural buffers and subsequently leaking it using micro-architectural side channels [66].

Past research treated Spectre attacks and memory errors as two separate domains. However, recent work [37, 72, 93] has introduced a new class of attacks that we refer to as *Speculative Memory-error Abuse* (SMA) attacks, which work by combining memory corruption vulnerabilities with Spectre-like primitives to bypass both memory-corruption and Spectre mitigations. SMA attacks allow adversaries to circumvent memory-corruption mitigations and leak sensitive program information (such as a victim program's code layout [37] or ARM Pointer Authentication Codes [93]) by causing the program to *speculatively* use corrupted data, avoiding crashes that are only triggered when the offending instructions are executed architecturally. Because SMA attacks can be carried out without causing any crashes, crash-sensitive software, such as OS kernels, become an attractive target. SMA attacks are not trivial to mitigate since neither standard memory-corruption mitigations, nor mitigations for speculative execution attacks, are effective [72, 93].

In this work, we present `Eclipse`: a comprehensive mitigation against SMA attacks. The core insight behind `Eclipse` is that CPUs cannot predict *unresolved values* (e.g., values that have not yet been computed), so instructions that have unresolved data dependencies cannot be executed, even in the speculative domain (until their dependencies are resolved). `Eclipse` propagates *artificial* data dependencies to prevent instructions from operating on corrupted data during speculative execution. In antithesis to other approaches that can mitigate SMA [21, 48], `Eclipse` does not completely stop speculation and hardens *only* specific code patterns leveraged by SMA attacks, improving performance without sacrificing security. `Eclipse` focuses on SMA attacks that exploit forward-edge control-flow transfers, while backward-edge protection (against SMA) is assumed to be handled by other mitigations.

We demonstrate that `Eclipse` successfully prevents SMA attacks, such as Speculative Probing (SP) [37] and PACMAN [93], on both x86-64 and ARM. We implement `Eclipse` as a compiler-based mitigation for the x86-64 architecture that seamlessly supports hardening both kernel code and userland applications, while incurring substantially lower overhead compared to alternative approaches for mitigating SMA: ≈0%–9.5% in the SPEC 2017 CPU benchmarking suite, ≈0%–8.6% in real-world applications, and negligible overhead in the Linux kernel.

2 Background

2.1 Memory Safety

Programs written in memory- and/or type-unsafe languages, such as C and C++, are prone to memory errors [108], which allow an adversary to *corrupt* and/or *disclose* (i.e., leak) a victim program's memory contents. Traditionally, attackers would abuse memory corruption to perform *code injection* [83], where they effectively inject an instruction stream into an executable region in a program's address space and redirect control flow to it. However, with the adoption of non-executable memory [114] on contemporary platforms [70, 76], and the enforcement of the W^X memory policy [84, 94], code injection has largely been replaced by *code reuse* [16] as the de facto method for exploiting memory-corruption vulnerabilities. By corrupting control-flow "influencing" data (e.g., code pointers [57]), attackers can employ various techniques, such as {return, jump, call}-oriented programming [14, 35, 101], to *hijack* the program's control flow and execute *existing* chunks of code in the victim program's address space *out-of-context*. Apart from code reuse, *data-oriented* attacks [43, 49] allow an attacker to exploit memory corruption to perform arbitrary computations by only tampering with select program data, while *data-only* attacks [23, 78, 88, 96] target *non-control* data.

Several defense mechanisms have been developed over the years that aim to raise the bar for exploiting memory errors. *Information-hiding* schemes conceal sensitive program information, such as the addresses or contents of program regions (e.g., code or data [69, 111]), by employing coarse- [29, 86] or fine-grain [15, 28] *memory-layout randomization/diversification* [32, 58], or by preventing read access to code regions with *execute-only* memory (XOM) [87, 98]. Other approaches aim to mitigate memory-error exploitation via integrity checking. *Control-flow integrity* (CFI) [10, 17, 33] restricts control-flow transfers to only *select*, allowed program locations, while *code-pointer integrity* (CPI) [57, 59, 90] ensures that code pointers have not been tampered with prior to use. Overall, despite decades of research, memory safety [79, 80] remains an open problem [104], as information-hiding and integrity-checking schemes have been demonstrated to be bypassable [20, 30, 31, 35, 36, 81].

2.2 Transient Execution Attacks

Modern CPUs (aggressively) employ various mechanisms to avoid CPU "idling" and maximize performance [41]. Examples include: *out-of-order execution*, where the CPU re-orders the instruction stream and executes subsequent instructions before preceding instructions have retired; and *speculative execution*, where the CPU leverages various *predictors* to *speculate* on the outcome of instructions which have not yet been computed due to unresolved *data dependencies*. These optimizations can prevent executed instructions from being committed to the CPU's architectural state—i.e., the CPU "squashes" the instructions—in the event of a fault or misprediction. Instructions whose results are never committed to the CPU's architectural state are considered to only be *transiently executed*.

Even though transiently executed instructions do not modify the CPU's architectural state, they do leave behind *observable* traces in its micro-architectural state (e.g., the cache [55, 65], load ports [100], line-fill buffers [109], and store buffers [19]). These micro-architectural side effects give rise to *transient execution attacks* [18], which coerce the CPU to transiently execute bogus computations before leaking results via *micro-architectural side channels* [40, 66, 109, 112, 118]. In general, transient execution attacks can be divided into two categories, dubbed Meltdown and Spectre, based on the cause of transient computation.

Meltdown. Attacks in the Meltdown category exploit transient, out-of-order execution proceeding a faulting instruction. They have been used to leak data across privilege boundaries and break process isolation [113], separation between user and kernel space [65], and confinement via Intel SGX [107, 113]. The majority of Meltdown mitigations prevent access to data on the micro-architectural level that are inaccessible architecturally. Newer micro-architectures provide this directly in hardware [45], while existing ones require microcode updates and/or software solutions [39, 48, 105].

Spectre. The Spectre category of attacks leaks sensitive program data by *(mis)training* or *tampering* with CPU predictors to coerce the CPU to speculatively execute *attacker-chosen* instructions that access sensitive program data, which are subsequently leaked via a micro-architectural side channel. Variants of Spectre are generally categorized based on the CPU predictor they are associated with (i.e., the predictor that is mistrained or tampered with). In Spectre-PHT (variant 1) [18], an attacker poisons the *Pattern History Table (PHT)* to cause the CPU to mispredict the outcome of *conditional branches* and speculatively execute code on the mispredicted edge of the branch [54, 55]. In Spectre-BTB (variant 2) [18], an attacker mistrains the *Branch Target Buffer (BTB)* to cause a misprediction of a computed (i.e., indirect) branch and speculatively hijack the program's control flow. Other variants include Spectre-RSB (i.e., "ret2spec"), which exploits the *Return Stack Buffer* [56, 71] used for predicting targets of return instructions, and Spectre-STL (variant 4) [50], which exploits the CPU's *Store-To-Load* forwarding logic to speculatively forward stale data to a load instruction.

Researchers and CPU vendors [11, 45] have introduced both software [22] and hardware [42] defenses for many of the Spectre variants. Since the fundamental cause of all Spectre variants is speculative execution, one mitigation approach is to place *serializing* instructions (e.g., `lfence` instructions on x86) before potentially vulnerable code locations to entirely prevent their speculative execution [11, 45, 68]. CPU vendors continue to recommend the use of serializing instructions to stop all types of speculative execution attacks [11, 48], and software vendors, such as the Linux kernel [24, 51], have followed suit. Alternative mitigation approaches include hindering the attacker's ability to extract data through side channels [52, 53, 117], or preventing CPU predictors from being mistrained or tampered with [44, 46, 47].

2.3 Speculative Load Hardening

Speculative Load Hardening (SLH) is a mitigation for Spectre-PHT introduced as an LLVM compiler pass for x86-64 [67]. SLH blocks Spectre-PHT leaks by preventing the CPU from executing instructions that load from memory during speculation. To achieve this, SLH first identifies *all* memory loads which can potentially be speculatively executed due to (mis)prediction of a preceding conditional branch. Then, it instruments the given program to prevent speculation from "consuming": (1) the memory address used as an operand by the load instruction, or (2) the loaded value itself. SLH also supports hardening indirect branches preceded by conditional branches, to mitigate Bounds Check Bypass Store (BCBS) [54]. In BCBS, an attacker is assumed capable of speculatively overwriting a value used as the target of a computed branch. By making data used in computed branches "unavailable" during speculation, SLH prevents speculative execution of indirect control-flow transfers.

2.4 Speculative Memory-error Abuse

Historically, researchers have treated memory errors and speculative execution attacks as two disjoint domains, each with its own threat model and mitigations. However, recent attacks like SPEAR [72], PACMAN [93], and BlindSide [37] have demonstrated that memory corruption and speculative execution vulnerabilities can be combined to bypass memory-safety-based mitigations. Collectively, we refer to these attacks as *Speculative Memory-error Abuse* (SMA), since they follow a common pattern: even though an attacker's attempts to exploit a memory corruption vulnerability are normally hindered by memory corruption mitigations, speculative execution allows the attacker to inhibit detection (e.g., *suppress* crashes) and extract sensitive program information.

2.4.1 SMA Attacks. In this section we highlight three SMA attacks—SPEAR [72], PACMAN [93], and Blindside [37]—that span multiple architectures (i.e., x86(-64) and ARM) and bypass various memory-safety-based mitigations (e.g., (K)ASLR and ARM PA). Even though the target of each attack differs, they all follow a common pattern: data that was architecturally corrupted is used during speculative execution, bypassing existing mitigations.

SPEAR. The first set of attacks, dubbed SPEAR [72], explores how speculative execution can be leveraged to bypass conventional hardening mechanisms. SPEAR demonstrates that an attacker can bypass certain hardening schemes, such as GCC's `vtable` verification (VTV) [106], Go's array bounds checks [34], and LLVM's Stack Smashing Protection (SSP) [27]. To achieve this, the attacker (either architecturally or speculatively) overwrites data that would normally trigger one of the aforementioned mechanisms. However, these mechanisms only raise an error when corruption is detected during non-speculative execution. By causing the CPU to speculate, the adversary temporarily achieves *speculative control-flow hijacking* and is able to access sensitive data, which they can leak through a side channel before the mitigation is architecturally triggered.

PACMAN. The second attack, PACMAN [93], targets ARM CPUs that provide support for Pointer Authentication (PA) [90]: a new feature which protects the integrity of pointers using cryptographic Message Authentication Codes (MACs), referred to as *Pointer Authentication Codes (PACs)*, embedded in unused pointer bits.

An invalid PAC causes a crash during pointer authentication in non-speculative execution, but the crash is suppressed under speculation. Thus, in PACMAN, an attacker leverages a memory corruption vulnerability to overwrite a stored pointer with a pointer of their choosing, including a corresponding, randomly guessed PAC. They then cause the CPU to try authenticating the overwritten pointer under speculative execution; if they guessed an incorrect PAC the pointer authentication will fail, but the crash will be suppressed. If, however, the guessed PAC was correct, the authentication will succeed and their pointer will be speculatively dereferenced, leaving micro-architectural side effects visible, which the attacker can observe to deduce the guessed PAC was correct.

BlindSide. The third set of attacks, dubbed Blindside [37], introduces a technique called Speculative Probing (SP), which allows an attacker to combine Spectre-like primitives with a single memory corruption vulnerability to achieve speculative control-flow hijacking. In Blindside, an attacker architecturally corrupts a code pointer and uses it to construct *probing primitives* in order to *speculatively probe* a victim program's address space for sensitive information, such as locations of memory regions, or even memory contents, without causing crashes that would normally occur under non-speculative program execution (e.g., by dereferencing a pointer to an unmapped memory page). The sensitive information is subsequently leaked and leveraged by the attacker to bypass certain information-hiding-based, memory-error mitigations, such as (K)ASLR [29, 86] or XOM [87, 98], to facilitate an architectural control-flow hijacking attack. Importantly, traditional approaches that try to stop speculative control-flow hijacking of indirect branches by preventing tampering with the indirect branch predictor [44, 47] are ineffective against SP since an attacker *architecturally* corrupts the code pointer used by the indirect branch.

2.4.2 SMA Mitigations. Several generic mitigations used against speculative execution attacks could prevent SMA attacks; however, all have significant drawbacks. First, techniques for hampering cache-based side channels [52, 99, 117] can constrain an from attacker leaking sensitive information, but do not stop them entirely, as recent work has shown that other types of side channels can be used instead [13, 112]. In short, identifying and obstructing all possible side channels is challenging. Second, the speculative execution of instructions leveraged by the attacks (e.g., indirect branches, PAC authentication instructions) can be prevented by injecting serializing instructions (e.g., `lfence` [48]) or instructions that force certain data to be unavailable during speculation (e.g., SLH [21]). Unfortunately, doing so naïvely incurs large performance overheads; SMA attacks target specific code patterns, and current approaches, such as SLH, are generic and not an ideal fit for SMA without adaptation. Finally, completely eliminating memory corruption [79, 80] can eliminate SMA, but such approaches have not yet gained traction.

3 Threat Model

Adversarial Capabilities. We assume an *unprivileged* attacker aiming to carry out an SMA attack to bypass memory-error mitigations (e.g., (K)ASLR, ARM PA) and mount an end-to-end exploit. We allow the attacker to (ab)use one or more spatial [79] or temporal [80] memory errors, whenever and however they choose, to build arbitrary *write* primitives to tamper with code pointers.

```
1  ...
2  ...
3  ...
4  cmpl   $0, %rax
5  je     no_call
6  ...
7  ...
8  ...
9  callq  *%rcx
10 .no_call:
11 ...
```

(a)

```
1  mov    $0, %r12
2  mov    $-1, %r11
3  ...
4  cmpl   $0, %rax
5  je     no_call
6  cmove  %r11, %r12
7  ...
8  or     %r12, %rcx
9  callq  *%rcx
10 .no_call:
11 ...
```

(b)

Figure 1: Code snippet vulnerable to SP on x86-64 (a), and the corresponding `Eclipse`-hardened snippet (b).

Further, the attacker is able to control the outcome of conditional branches in the vulnerable program, implicitly allowing them to (mis)train the conditional branch predictor (à la Spectre-PHT). Any attempts by the attacker to alter other speculative behavior of the program (i.e., other than the outcome of conditional branches by mistraining branch predictors [12, 55]) are orthogonal to `Eclipse`. Finally, the attacker can be co-located on the same machine as the process running the vulnerable program and can observe the micro-architectural state of the CPU via a side-channel attack, such as Prime+Probe [66], Flush+Reload [118], *etc.* Overall, we assume an adversary that is on par with the state-of-the-{art, practice} in terms of SMA attacks [37, 72, 93].

Hardening Assumptions. We assume that the underlying platform enforces the W^X policy [84, 94], preventing an attacker from performing code injection [83]. Other memory-error mitigations, such as (K)ASLR [15, 29, 86], XOM [87, 98], CFI [10, 17], and CPI [57, 59], are neither precluded nor required by `Eclipse`. Similarly, mitigations against transient execution attacks [22, 42] are orthogonal to `Eclipse`.

4 Mitigation Approach

`Eclipse` provides a generic approach for mitigating SMA attacks independent of any particular architecture or SMA attack. In this section, we first provide an overview of how an attacker carries out SMA attacks, such as a (generic) SP attack and a (ARM-specific) PACMAN attack, followed by a high-level description of `Eclipse`'s *instrumentation* that thwarts such attacks. Further discussion of how `Eclipse` generalizes to other architectures and SMA attacks can be found in Section 9.

4.1 Speculative Probing

Overview. In an SP attack, an attacker *probes* a target program's address space for sensitive information. In the example that follows, the attacker's goal is to leak the address of a target program's code region. To do this, the attacker first finds a vulnerable piece of code resembling the following:

```
if (cexpr) { ... (*fptr)(); ... }
```

Here, a code pointer (i.e., `fptr`) is conditionally dereferenced and the attacker is able to: (a) corrupt memory to (architecturally) overwrite `fptr`, and (b) control the outcome of `cexpr`.

Using (a), the attacker can control where `fptr` points, and with (b) they can train the conditional branch predictor to control when the pointer is speculatively dereferenced.

Initially, `fptr` points to a valid address within the victim program's code region, which the attacker is unable to read. To begin the attack, the attacker first trains the conditional branch predictor—e.g., by repeatedly "taking" the conditional branch and architecturally dereferencing `fptr`. Next, the attacker leverages the memory corruption vulnerability to *overwrite* `fptr` with an address where they guess the program's code region resides. Simultaneously, they *flip* `cexpr` to ensure that `fptr` is only ever dereferenced speculatively, not architecturally, avoiding a crash if `fptr` targets invalid memory. Because of the (mis)trained branch predictor, the CPU speculatively dereferences `fptr` and attempts to fetch instructions from the pointed-to location.

To determine if the guessed location targeted a valid code region, the attacker uses a side channel (e.g., Prime+Probe [66]) to check for signals that indicate whether the CPU has fetched any new instructions. If no signal was detected, the guessed address most likely does not reside in an executable page, and the process is repeated, corrupting `fptr` with a new address. In a successful SP attack, the attacker eventually guesses (i.e., corrupts the pointer with), and subsequently leaks, a correct address belonging to the program's code region.

CPU State. The C code snippet above corresponds to the assembly snippet in Figure 1a, which will serve as the basis for giving a detailed description of the CPU's *register state* that is further illustrated in Figure 3a. Figure 1b presents the corresponding `Eclipse`-hardened snippet; we provide details about the instrumentation in Section 5. A similar CPU state applies to SP attacks targeting ARM; we omit this for brevity. The following description assumes the attacker has already (mis)trained the conditional branch predictor, corrupted `fptr`, and flipped `cexpr`.

First, the CPU tries to execute the compare instruction (`cmpl`) on ln. 4 in Figure 1a (① in Figure 3a). However, this instruction is *data dependent* on the value of `rax`, which we assume has not yet been resolved. As a result, the compare instruction cannot yet be executed, and the value of `rflags` now becomes unresolved as well. Next, the CPU tries to execute the conditional branch (`je`) on ln. 5 (② in Figure 3a); however, since the value of the `rflags` register is currently unresolved, the correct outcome of the branch is unknown, and the CPU *speculates* the outcome of the branch to avoid stalling. Since the attacker has (mis)trained the CPU predictor, the CPU will speculate that the indirect branch on ln. 9 should be executed (③ in Figure 3a), and will thus (speculatively) dereference the (attacker-controlled) code pointer in `rcx`. If the attacker correctly guessed an address in an executable page, the CPU will fetch the instructions from that address and continue speculatively executing them (④ in Figure 3a). Even if the executed instructions are not committed architecturally, they still leave an observable trace on the cache.

Eventually, the value of `rax` is resolved, and the CPU (non-speculatively) executes the `cmpl` instruction and resolves the value of `rflags`. Having resolved `rflags`, it then executes the conditional branch, stops speculation, and jumps to the correct target of the conditional branch: that is, `.no_call` on ln. 10.

```
1   ...
2   ...
3   cmp    w9, 0
4   b.eq   no_call
5   ...
6   ...
7   mov    x17, #0x0
8   ldr    x8, [x16]
9   ...
10  autia  x8, x17
11  blr    x8
12  no_call:
13  ...
```

(a)

```
1   mov    x20, #0x0
2   ...
3   cmp    w9, 0
4   b.eq   no_call
5   csetm  x20, eq
6   csdb
7   mov    x17, #0x0
8   ldr    x8, [x16]
9   orr    x8, x8, x20
10  autia  x8, x17
11  blr    x8
12  no_call:
13  ...
```

(b)

Figure 2: Code snippet vulnerable to PACMAN (a), and the corresponding `Eclipse`-hardened snippet (b).

4.2 PACMAN

The goal of an attacker during a PACMAN attack [93] is to overwrite a PA-signed pointer with another chosen pointer and infer the *valid* PAC for the corrupted pointer by *leaking* the result of a *speculatively executed* PAC authentication instruction [90]. To do so, they first locate a *PACMAN gadget*, such as the one illustrated in Figure 2a, which *conditionally* executes a PAC authentication instruction and subsequently *uses* its result (e.g., branches to the location pointed-to by the authenticated pointer). The `Eclipse`-hardened version of the gadget is presented in Figure 2b; we provide more details about the instrumentation in Section 5.

The PACMAN gadget starts with a conditional branch (`b.eq`) instruction (ln. 4 in Figure 2a), then loads a modifier value in `x17` (ln. 7) and a signed pointer in `x8` (i.e., the pointer along with its PAC stored in its upper bits) to be authenticated (ln. 8). Next, it performs the authentication via `autia` (ln. 10), which, if successful, will clear the PAC from the upper bits of `x8`, allowing the subsequent indirect branch (ln. 11) to dereference it. If the authentication fails, an exception will be raised.

The logic of the attack is similar to an SP attack (§4.1). The attacker first trains the conditional branch predictor to execute the authentication instruction (`autia`; ln. 10) speculatively. Subsequently, they use a memory-corruption vulnerability to overwrite the stored pointer with a value of their choice, including a (randomly-guessed) PAC. Next, they detect whether the guessed PAC was correct by observing whether the indirect branch (ln. 11) was taken (e.g., via a side channel), which only happens if the preceding authentication instruction (`blr`; ln. 10) produced a valid pointer. If the guessed PAC was incorrect, the crash is suppressed due to speculative execution.

4.3 Eclipse Overview

`Eclipse` is a *compiler-assisted* solution for preventing SMA attacks, such as SP and PACMAN. `Eclipse` first analyzes a vulnerable program to identify the instructions that can be leveraged by an attacker to carry out an SMA attack. The specific instructions differ based on the respective attack that is being carried out (e.g., indirect branches for SP or ARM Pointer Authentication instructions for PACMAN); but, regardless of the scenario, they follow a common pattern: the attacker can force the CPU to *speculatively execute* them because of a mispredicted, preceding conditional branch (à la Spectre-PHT). We call these *SMA-Capable* (*SMAC*) instructions.

After identifying the SMAC instructions, `Eclipse` hardens them with *instrumentation* that *propagates artificial data dependencies* to prevent them from operating on attacker-controlled data (e.g., corrupted pointers) during speculative execution—inspired by the instrumentation of SLH [21], which is used for hardening memory loads (during speculative execution) using predicate-guarded, branchless code. `Eclipse` is based on the observation that, even though the CPU can predict the outcome of control-flow instructions (e.g., conditional branches) and start speculative execution, *unresolved data dependencies* cannot be predicted. `Eclipse` capitalizes on this by tying the data dependency of the conditional branch to the data used by the SMAC instructions. Effectively, `Eclipse` breaks SMA by preventing the SMAC instructions from using the data (i.e., preventing ③ in Figure 3a from occurring) until the data dependency—and thus the correct outcome of the conditional branch—is resolved. `Eclipse` has three main advantages compared to alternative approaches that can mitigate SMA (§2.4.2).

First, it does not require any hardware modification and can be implemented solely in software. Second, it does not rely on completely eliminating side channels or memory-corruption vulnerabilities, but instead prevents SMAC instructions from using (potentially) corrupted data during speculative execution. Third, it prevents SMA attacks *without* completely stopping speculation—it only delays the CPU from learning the value of the data used by the SMAC instructions until the data dependency is resolved. This allows the CPU to still: (a) speculatively execute any other non-data-dependent instructions; and (b) perform (nested) speculative execution (e.g., predict the outcome the indirect branch in SP, or indirect branches using the authenticated pointer in PACMAN), keeping performance overhead low in cases where the CPU speculates correctly.

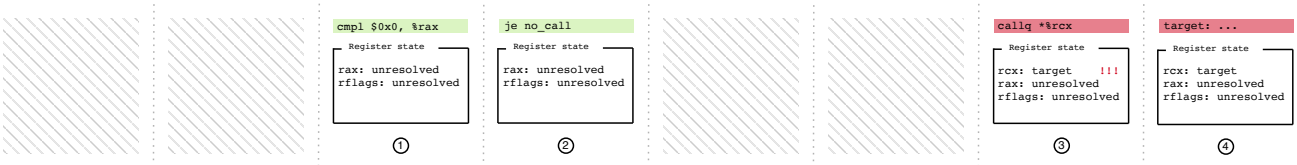
5 Design

`Eclipse` prevents SMAC instructions from using (potentially corrupted) data (e.g., code or data pointers) during speculative execution, yet improving performance by preserving other forms of speculation—e.g., nested speculation or speculative execution of non-SMAC instructions. To achieve this, `Eclipse` identifies and instruments *only* the specific code leveraged in SMA attacks and does not make use of speculation barriers.

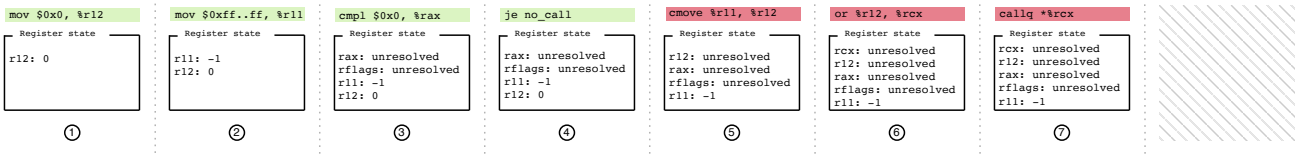
In what follows, we first describe the design of `Eclipse` in the context of SP attacks on x86(-64) before outlining how `Eclipse` can be applied to ARM and PACMAN attacks in Section 5.3.

5.1 SMAC Instruction Identification

`Eclipse` first identifies which *specific* indirect branches need to be hardened in order to prevent SP. Notably, `Eclipse` does not harden all indirect branches in the program. Instead, it locates and hardens *only* SMAC indirect branches: indirect branches that can potentially be speculatively executed as a result of the CPU mispredicting a preceding conditional branch. Figure 4 presents an example of a function analyzed by `Eclipse` that contains such a SMAC indirect branch. Highlighted instructions are those added by `Eclipse`'s instrumentation; `t` and `nt` represent taken and not-taken edges of conditional branches, respectively; arrows show control-flow between basic blocks (e.g., `.bb1` to `.bb6`).



(a) Execution state during SP on a non-hardened binary.



(b) Execution state during SP on a `Eclipse`-hardened binary.

Figure 3: Execution state during a SP attack on a {non, `Eclipse`}-hardened binary. Regular execution is denoted with and speculative execution is denoted with . Instructions that are present in both binaries are listed under the same column.

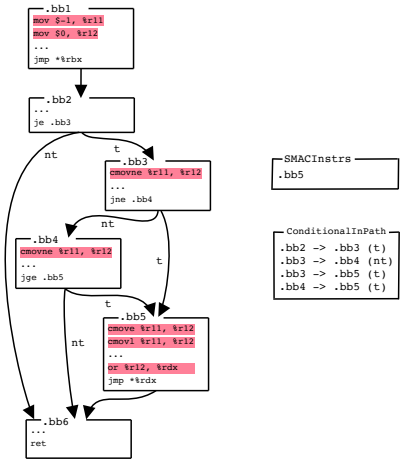


Figure 4: Control-flow graph of a `Eclipse`-instrumented function, and sets constructed by `Eclipse` analyses (§5.1). Highlighted instructions are injected by `Eclipse`.

SMAC Indirect-branch Identification. `Eclipse` identifies and collects all basic blocks within a given function into a set, called `SMACInstrs`, if they contain SMAC indirect branches. To do so, it iterates over every instruction, in each function of the analyzed program, terminating when all instructions have been analyzed. Whenever it encounters an indirect branch, `Eclipse` characterizes it as an SMAC, if it can be speculatively executed as a result of a (mis)predicted, *preceding* conditional branch (in the same function). To determine whether an indirect branch in a basic block is SMAC, `Eclipse` checks if the control-flow graph (CFG) of the function remains connected when the block containing the indirect branch is removed. This is done via a breadth-first traversal of the function's CFG, starting from the entry node and attempting to reach the exit node without visiting the block containing the indirect branch.

If a path to the exit node can be found, it means that the CFG is still connected, and the block containing the indirect branch is added to `SMACInstrs`. In Figure 4, basic blocks `.bb1` and `.bb5` contain indirect branches. When `Eclipse` analyzes the function, it does not add `.bb1` to `SMACInstrs`, since if it is removed from the CFG no possible paths will exist from the entry node (`.bb1`) to the exit node (`.bb6`). This implies that the CFG is not connected when `.bb1` is removed, and thus `.bb1` cannot be conditionally reached. However, `.bb5` is added to `SMACInstrs`, since when it is removed from the CFG there exists at least one path from the entry to exit node (e.g., `.bb1` → `.bb2` → `.bb6`).

Preceding Conditional-branch Identification. If, after the first analysis step, the `SMACInstrs` set is empty, `Eclipse` does not instrument the function. Else, it reiterates the CFG and constructs a second set, `ConditionalInPath`, containing the CFG edges of basic blocks with a conditional branch that lie in the path from the entry block to one of the blocks in `SMACInstrs` (i.e., the conditional branches whose (mis)speculation can cause speculative execution of the SMAC indirect branches). To construct this set, `Eclipse` starts from each block in `SMACInstrs` and iterates the CFG backwards, visiting each edge to a predecessor block until it reaches the entry block of the function. Whenever a block containing a conditional branch is encountered during this traversal, `Eclipse` adds the edge that was traversed to `ConditionalInPath`.

In Figure 4, `Eclipse` iterates backwards from `.bb5`—the sole member of `SMACInstrs`—adding the taken edge of `.bb2`, both edges of `.bb3`, and the taken edge of `.bb4` to `ConditionalInPath`. The not-taken edges of `.bb2` and `.bb4` are not added to the set, since they are not visited when traversing the graph.

Required Information. The only information needed by `Eclipse` to perform the aforementioned analysis is the predecessors and successors of each basic block, within a function, and the basic blocks containing indirect and conditional branches. This information can be extracted at a late compilation stage, or by statically analyzing the compiled binary itself. As a result, `Eclipse` is *agnostic* to the source language of the binary.

5.2 Instrumentation

After analyzing a given function to identify SMAC indirect branches and their preceding conditional branches, Eclipse applies its instrumentation that prevents SMAC indirect branches from speculatively dereferencing attacker-controlled pointers. This is achieved by propagating *artificial* data dependencies from the conditional branches in ConditionalInPath to the code pointers used by the indirect branches in SMACInstrs. Specifically, Eclipse injects instructions that: (a) *capture* the data dependencies causing the conditional branches to be (mis)speculated; and (b) *link* these dependencies to the code pointers dereferenced by the indirect branches.

Register Initialization. Eclipse injects two move instructions in the entry block of a vulnerable function, as seen in the highlighted code of .bb1 in Figure 4. These instructions initialize two general-purpose registers, r11 and r12, with the values −1 and 0, respectively. r12 is referred to as the *state* register and is *reserved* to propagate the data dependency from a given conditional branch to a code pointer invoked by a conditionally-reached indirect branch. Initializing the state register with 0 ensures that the behavior of the program remains unaltered during non-speculative execution (§6.1). The other register, r11, is referred to as the *poison* register, and it is used to ensure SMAC code pointers are *poisoned* when transient execution occurs (§6.2).

Capturing Data Dependencies. To capture data dependencies, Eclipse iterates each edge in the ConditionalInPath set and instruments the destination block of the edge (i.e., the successor of the block containing the conditional branch) with a *conditional move* (cmov) placed at the start of the block. The poison and state registers are used as the source and destination operands of the conditional move, respectively. Since both conditional moves and branches implicitly rely on the rflags register to determine their outcome, the injected conditional move effectively *captures* the data dependency and propagates it over to the state register. To determine the appropriate conditional code for the injected conditional move, Eclipse inspects the type of edge (i.e., taken or not-taken), as well as the conditional code of the conditional branch; if the edge is taken, Eclipse selects the *opposite* conditional code to the one in the conditional branch, else it selects the same code as the branch. This guarantees that the instrumentation does not alter the non-speculative execution of the program (§6.1).

In the example in Figure 4, Eclipse selects the opposite condition of the conditional branch in .bb2 for the conditional move injected in .bb3 (i.e., "equal" → "not equal") since the edge type is taken. Additionally, it matches the condition of the branch in .bb3 for the conditional move in .bb4 since the edge type is not-taken. Finally, Eclipse injects two conditional move instructions in .bb5, both having the opposite conditions of their respective conditional branches, since .bb5 is the destination block for two taken edges (from .bb3 and .bb4) in ConditionalInPath.

Linking Data Dependencies to Code Pointers. Lastly, Eclipse *links* the captured data dependency from the state register to the code pointers used by the SMAC indirect branches. To achieve this, Eclipse iterates each block in SMACInstrs and injects an or instruction right before the indirect branch, which uses the *state* register as the source operand and the register used in the indirect branch as the destination operand (.bb5 in Figure 4).

If the indirect branch dereferences a memory location, Eclipse *unfolds* it (i.e., adds an extra instruction which loads the code pointer from memory into a register) before linking the data dependency to the register. Combined with the *dependency-capturing* conditional moves, the *dependency-linking* or instructions effectively *propagate* the dependency on the rflags register onto the code pointers used by the SMAC indirect branches.

5.3 Applying Eclipse on ARM

Eclipse's approach generalizes beyond both SP attacks and the x86(-64) architecture, as long as the architecture provides instructions that can *capture* and *link* data dependencies to the data used by the target SMAC instructions. In what follows, we describe how Eclipse mitigates the ARM-specific PACMAN attack. (The process for hardening against SP on ARM is almost identical.) To mitigate PACMAN, Eclipse follows the process described in Section 5.1 to identify and harden SMAC instructions.

SMAC Instruction Identification. For PACMAN, the SMAC instructions that need to be identified and hardened by Eclipse are PA instructions (e.g., auti{a,b} [62], autd{a,b} [61], and ldra{a,b} [64]). To construct the SMACInstrs set, Eclipse follows the process described in Section 5.1, but now looks for these PA SMAC instructions instead of SMAC indirect branches. Similarly, to construct the ConditionalInPath set, it looks for different types of ARM conditional branches (i.e., b.cond, cbz/nz, and tbz/nz).

Instrumentation. In the same vein, to harden the identified SMAC instructions, Eclipse follows the process described in Section 5.2, but uses ARM instructions to *capture* and *link* the data dependencies. An example of the instrumentation is presented in Figure 2b and corresponds to the PACMAN-vulnerable snippet in Figure 2a. After initializing the state register (x20; ln. 1 in Figure 2b), Eclipse uses the *conditional set mask* (csetm) instruction with an appropriate condition code (ln. 6; §5.2) to *capture* the dependencies. Finally, Eclipse uses the orr instruction (ln. 10) to *link* the dependencies to the pointers used by the SMAC PA instructions.

ARM-specific Design Choices. Even though the process followed by Eclipse to mitigate SMA attacks in ARM and x86(-64) is fundamentally the same, the behavior of certain ARM instructions allow for, or require, minor adaptations. First, the conditional set mask instruction (csetm) used by Eclipse to *capture* data dependencies sets all bits of the destination register to 1 (i.e., to the value −1) when the chosen condition code is true [63], else it sets all bits to 0, so Eclipse does not need to explicitly use a poison register. Second, in cases where an identified conditional branch was either a cbz/nz or a tbz/nz, Eclipse inserts an additional cmp instruction before the branch in order to set the status register, which is implicitly read by csetm, since these conditional branches perform their branch using register arguments and do not rely on reading the status register. Lastly, according to ARM, implementations of the architecture may allow data processing instructions (such as csetm) to produce speculative values themselves [60]. When Eclipse is applied to an ARM implementation that implements this behavior, it injects an additional csdb instruction (ln. 7), which guarantees subsequent instructions do not use such speculative values.

5.4 Binary Verifier

Since Eclipse's instrumentation is introduced at compile time, we ensure that no modifications (i.e., reordering/elimination of the injected code [116]) are made to the instrumentation in subsequent compilation steps before the final binary is produced; such optimizations could render Eclipse ineffective. To do so, we designed a *binary verifier* which statically analyzes Eclipse-instrumented binaries and ensures that the instrumentation is unaltered.

The verifier first disassembles a given binary and constructs the CFG for each function. With the CFG, it reconstructs the SMACInstrs and ConditionalInPath sets (§5.1) in order to identify the necessary instrumentation points. To do so, it follows an approach similar to Eclipse: for the SMACInstrs set, it adds every block containing a SMAC instruction that can be *conditionally* reached; and for the ConditionalInPath set, it iterates the CFG backwards, starting from each block in the SMACInstrs set, and adds all edges from blocks containing conditional branches that are in the path of a SMAC instruction in the set.

Next, it verifies that each SMAC instruction in SMACInstrs is properly instrumented. For each SMAC instruction in the set, it checks that there exists a sequence of instructions starting with a *dependency-linking* instruction (e.g., or), that links the data dependency from its destination operand to the register used by the subsequent SMAC instruction. Notably, the dependency-linking instruction does not necessarily need to directly precede the SMAC instruction; for example, we found Eclipse's instrumentation against SP for x86(-64) was occasionally modified to patterns such as:

```
or      %r12, %rdx
mov     %rdx, %rax
callq   *%rax
```

These patterns are still considered valid instrumentation since the code pointer (rax) is still data dependent on the state register (r12). Finally, the verifier checks that the *dependency-linking* instruction uses the reserved state register as its source operand.

After all instructions in SMACInstrs are verified, the verifier goes through each edge in the ConditionalInPath set, verifying that they have been properly instrumented. For each edge, it checks that the destination block of the edge is instrumented with a conditional move (for x86(-64)) or conditional set mask (for ARM). Then, it checks that the conditional move or set mask uses the proper operands: the reserved state register for the destination register, and, for conditional moves, a properly initialized poison register (i.e., −1) for the source operand.

6 Security Analysis

The instrumentation introduced by Eclipse (§5.2) mitigates SMA attacks by *propagating* data dependencies to data used by SMAC instructions, aiming to prevent the latter from using the data during speculative execution. In this section, we provide a detailed security analysis of Eclipse's instrumentation. We first walk through a running example and examine how the propagated data dependencies influence the CPU's state and prevent SP while being transparent to the non-speculative execution of the program (§6.1). We also resolve an edge case that can occur during out-of-order execution, which allows an attacker to circumvent a naïvely implemented mitigation (§6.2).

Similar to Section 5, we first focus on how Eclipse mitigates SP, then we expand the analysis to PACMAN (§6.3). While we use specific examples to make the analysis easier to follow, Eclipse's protection generalizes to *all* SMA attacks.

6.1 Eclipse-hardened Code Execution

Speculative Execution. We present a concrete example of how the instrumentation introduced by Eclipse stops SP on x86(-64); this example also applies to SP on ARM without loss of generality. A code snippet of a program hardened by Eclipse is shown in Figure 1b, corresponding to the vulnerable snippet in Figure 1a. Similar to the attack example in Section 4.1, we step through program execution during the attack and describe how Eclipse prevents the CPU from speculatively dereferencing the corrupted function pointer. We also present the register state during the (speculative and non-speculative) execution of each instruction in Figure 3b.

First, in ln. 1–2, the state and poison registers (§5.2) are initialized (① and ② in Figure 3b). Next, the CPU tries to execute the compare instruction (cmpl) at ln. 4, which, if executed, will implicitly change the value of the rflags register; however, the compare instruction cannot yet be executed since the value of rax is currently unresolved (③). As a result, the value of the rflags register is now also unresolved, and the CPU speculates on the outcome of the conditional branch in ln. 5 (④). Since the branch predictor is (mis)trained, the CPU speculates that the branch is not taken.

The CPU now attempts to speculatively execute the conditional move (cmove) at ln. 6 (⑤); however, the conditional move is also data dependent on the rflags register. Thus, the (unresolved) data dependency is now *captured* by the conditional move and *propagated* onto the state register—i.e., since the conditional move cannot yet be executed, the value of the state register cannot yet be resolved. Finally, the CPU tries to execute the or instruction injected by Eclipse on ln. 8 (⑥). Since the (currently unresolved) state register is used as the source operand, this instruction cannot yet be executed either. This or instruction *links* the data dependency to the destination operand—i.e., the code pointer, rcx—which is dereferenced by the SMAC indirect branch on ln. 9. When (speculative) execution reaches this indirect branch (⑦) the unresolved code pointer cannot be speculatively dereferenced.

Effectively, Eclipse's instrumentation propagates the (unresolved) data dependency that *caused* the CPU to speculate the outcome of the conditional branch onto the (potentially corrupted) code pointer. This prevents the indirect branch from speculatively dereferencing the pointer until the data dependency is resolved, which also resolves the correct outcome of the conditional branch.

Non-speculative Execution. The instrumented code produced by Eclipse is transparent to the concrete execution of the program. The condition codes chosen by Eclipse for the injected conditional move instructions (§5.2) ensure the move is not performed during concrete execution. Consequently, the injected or instructions will *never* alter the architectural value of the code pointer.

From the example in Figure 1b, when execution reaches the conditional branch (je) on ln. 5, there are two possible outcomes, either: (a) the condition is true and the jump is taken, jumping over the instrumented code; or (b) the condition is false and the program continues executing the conditional move on ln. 6.

If the conditional move is executed, it is guaranteed that the condition was false, else the jump is taken and skips over the instructions at ln. 6–9. Since the conditional move is injected on the conditional branch's not-taken edge, and the condition code matches that of the conditional branch, the move will not be performed, and the state register will remain 0. Finally, the `or` instruction on ln. 8 will not alter the value of the code pointer, since the state register contains the value 0. Thus, the indirect branch on ln. 9 will dereference the unaltered architectural value of the code pointer in the `rcx` register.

6.2 Poisoning the Code Pointer

`Eclipse` considers an important edge case that can occur during the attack: after `rflags` is resolved, the CPU may execute the conditional move (ln. 6 in Figure 1b) and subsequent instructions *before* executing the conditional branch (ln. 5) due to *out-of-order execution* [41]. This gives a window for the attacker-controlled pointer to be dereferenced (ln. 9) before the conditional branch is executed and the speculatively executed instructions are squashed.

Out-of-order Execution Measurements. We performed an experiment demonstrating the aforementioned scenario. Namely, we constructed a program that resembles a `Eclipse`-hardened SP code pattern, but, instead of (speculatively) dereferencing a code pointer, it accesses a memory location which leaves a signal at a *distinct* cache location, as seen in the snippet below:

```
1  mov     $addr_a, %rax
2  mov     $addr_b, %rbx
3  ...
4  je      .end
5  cmove   %rbx, %rax
6  mov     (%rax), %rcx
7  .end
8  ...
9  /* Measure cache signal */
```

We set up the program so the conditional move and memory access at ln. 5–6 are *speculatively* executed. Then, we performed cache measurements to check whether `addr_a` or `addr_b` get cached.

While running this experiment, we detected cache signals at `addr_b`, indicating that the conditional move and memory access were executed. Since the `rax` register—holding the value of the accessed memory location—is data dependent to `rflags`, the instructions could not have executed during speculative execution. As such, we deduce that the instructions were executed because of out-of-order execution, as described above.

Preventing Out-of-order Dereferences. `Eclipse` prevents the value of the code pointer from being dereferenced when the aforementioned scenario occurs by ensuring that the pointer becomes *poisoned*. Under speculation, the injected conditional moves act *opposite* to their behavior in concrete execution, moving the poison value (−1) to the state register and subsequently causing the `or` instructions to alter the value of the code pointer.

Using the example in Figure 1b, we now assume that the compare instruction at ln. 4 has been architecturally executed and has resolved the value of the `rflags` register. The CPU then chooses to (incorrectly) execute the conditional move instruction at ln. 6 instead of executing (and taking) the conditional branch at ln. 5.

However, since the conditional branch should have been taken, the condition relating to both the conditional branch and the conditional move is true. As a result, the move will be performed, effectively *poisoning* the state register with the value −1 (`0xffff...ffff`). Consequently, the `or` instruction at ln. 8 will alter the value of the code pointer in `rcx`, setting it to −1. Finally, the indirect branch on ln. 9 will try to dereference address −1 instead of the address that is *architecturally* held in the pointer.

To summarize, even if the value of the code pointer is controlled by the attacker, `Eclipse` guarantees that the value is altered to −1 before it is dereferenced during out-of-order execution, preventing the attack from succeeding.

6.3 Eclipse-hardened PACMAN Analysis

`Eclipse`'s protection applies to PACMAN in a fundamentally similar fashion. The only differences are: (a) the SMAC instructions the data dependency is linked to and (b) the instructions used. An `Eclipse`-hardened PACMAN snippet corresponding to the vulnerable snippet in Figure 2a can be seen in Figure 2b.

Assuming the value of `x9` is unresolved, the `cmp` instruction (ln. 3) cannot yet execute. Because of the now-unresolved status register, the CPU speculates the outcome of the conditional branch (ln. 4). Next, the `Eclipse`-injected `csetm` instruction (ln. 5, optionally combined with the `csdb` in ln. 6; §5.3) propagates the dependency on the unresolved status register onto the state register, `x20`. As a result, the `orr` instruction (ln. 9) cannot be executed either, preventing the subsequent authentication instruction (ln. 10) from authenticating the pointer during speculative execution.

During non-speculative execution, the `csetm` will set all bits of the `x20` status register to 0, ensuring that the `orr` instruction will not modify the value of the pointer to be authenticated. Finally, the instrumentation protects against the out-of-order execution scenario (§6.2) since the `csetm` instruction will set all bits of the status register to 1 because of the matching condition code, and the `orr` instruction will alter the value of the pointer to be authenticated.

7 Implementation

We implemented `Eclipse` for the x86-64 architecture as an LLVM (v11) machine-function pass in ≈1100 lines of C++ code. The pass runs right before the register allocation phase of the compiler's pipeline. We chose to implement `Eclipse` late in the compiler optimization pipeline to ensure that subsequent optimizations modify the injected instructions as little as possible. Also, we chose to implement it before the register allocation phase, such that we could reserve a general-purpose register to be used as the state register (§5.2). For ARM, we injected `Eclipse`'s instrumentation using inline assembly for the evaluated programs.

The verifier (§5.4) was implemented using the Egalito [115] binary rewriting tool in ≈550 lines of C++ code—currently, it supports verifying binaries compiled from C code only. To ensure that any potential bugs in the implementation or logic of the LLVM pass did not transfer over to the verifier, a graduate student was provided with information regarding `Eclipse`'s instrumentation and tasked with implementing the verifier independently—i.e., the developer of the LLVM pass and the verifier were separate people.

8 Evaluation

We evaluated the performance of our Eclipse prototype on x86-64 only—our ARM implementation is not mature enough to support fully instrumenting applications. Specifically, we used a set of userland applications and the Linux kernel. For userland, we performed benchmarks using the SPEC CPU 2017 suite, as well as real-world applications. For the kernel, we performed micro-benchmarks using the LMbench suite [74] and macro-benchmarks using the Phoronix Test Suite (PTS) [89]. We also assessed Eclipse's security effectiveness on both x86-64 and ARM. For x86-64, we deployed Eclipse against an SP [37] attack on the Linux kernel; for ARM, we deployed Eclipse in the kernel and userland to defend against a PACMAN [93] and an SP attack, respectively.

x86-64 Testbed. We ran our performance experiments for x86-64 on a machine with a 16-core Intel Xeon W-2145 3.70GHz CPU and 64GB of RAM. The userland benchmarks were executed inside Docker [75] containers configured to use all available CPUs and RAM. The container(s) for the SPEC CPU 2017 benchmarks ran Void Linux [8], while the ones for the real-world benchmarks ran Alpine Linux [9]. Moreover, the benchmarked applications were linked against `musl libc` [4], built as position-independent (`-f{PIC, PIE}`, `-pie`), and optimized with `-O2`. The kernel benchmarks ran atop a Debian v11 system using Linux kernel v5.4.256. For all benchmarks, the CPU was configured to always run at 3.70GHz in the C0 C-state (i.e., fully activated) with SMT enabled and dynamic frequency and voltage scaling (DVFS, Turbo Boost) disabled, in order to reduce noise and promote reproducibility.

ARM Testbed. The ARM experiments were performed on an Apple MacBook Pro, equipped with an Apple M2 CPU and 16GB of RAM, running Darwin v23.2.0 with the XNU kernel v10002.61.3, patched [91] to allow the use of high-resolution timers in userland.

8.1 Userland Performance

We evaluated the performance overhead introduced by Eclipse on instrumented userland applications, and compared against two alternative mitigation approaches, described below. We benchmarked the C/C++ programs included in the SPEC CPU 2017 suite, as well as four real-world applications, namely Redis (v7.2.4) [6], Nginx (v1.26.0) [5], SQLite (v3.45.3) [7], and MariaDB (v10.11.7) [3]. The ratio of the total SMAC indirect branches (over all indirect branches) for each real-world application and evaluated SPEC CPU 2017 benchmark (including their dependencies) is presented in Table 2.

8.1.1 Alternative Mitigations. We compared the performance overhead of Eclipse against alternative approaches for mitigating SP attacks on x86(-64), specifically, using `lfence` and SLH.

LFENCE. One alternative approach to Eclipse is injecting serializing instructions, such as `lfence` [48], prior to SMAC indirect branches, completely stopping speculative execution. To demonstrate this, we implemented a variant of Eclipse, dubbed `Eclipse-lfence`, to compare against. To implement `Eclipse-lfence`, we modified our instrumentation (§5.2) to *only* inject an `lfence` instruction before each SMAC indirect branch in the `SMACInstrs` set, instead of artificially propagating data dependencies to the code pointers used by the branches. (Note that `Eclipse-lfence` is not an out-of-the-box approach as it relies on Eclipse to identify the SMAC indirect branches that must be hardened.)

SLH. An alternative, out-of-the-box approach to mitigating SP is Speculative Load Hardening (SLH) (§2.3) with indirect branch hardening enabled via the compile-time flag: `-x86-slh-indirect=true`. Since SLH is designed *specifically* to mitigate Spectre-PHT attacks, this option can only be enabled atop the main functionality of SLH (i.e., hardening all memory-load instructions following conditional branches). In contrast, Eclipse focuses solely on mitigating SP—and hence hardens *only* SMAC indirect branches—significantly reducing the number of instrumentation points.

8.1.2 SPEC CPU 2017. We evaluated the performance overhead of the three approaches on the C/C++ integer and floating point benchmarks in the SPEC CPU 2017 suite. Table 1 presents the average performance overhead across 5 runs of the `ref` workload, presented as percentages atop an uninstrumented baseline. (≈0% corresponds to $< 0.1\%$.) Overall, Eclipse outperforms the other two approaches in the majority of the benchmarks, incurring overheads ranging from ≈0%–9.53%. `Eclipse-lfence` incurs higher overheads compared to Eclipse overall, ranging from ≈0%–26.73%. SLH performs consistently worse than the other two approaches, incurring large overheads ranging from 2.62%–154.36%.

We observe that for `600.perlbench_s`, `619.lbm_s`, `631.deepsjeng_s`, and `638.imagick_s` the `Eclipse-lfence` approach outperforms Eclipse, with a negligible difference ($< 0.5\%$) in 3/4 cases. Eclipse injects more instructions in the binary: indirect branches are masked with an `or` instruction and conditional moves are inserted at the edges of preceding conditional branches. Further, it reserves a general-purpose register (§5.2), which increases register pressure in the program, causing more memory spills. On the other hand, `Eclipse-lfence` only inserts one instruction (i.e., `lfence`) per SMAC indirect branch. When an `lfence` is inserted in a block containing a large number of instructions, the performance penalty is larger, since it prevents speculative execution of *all* the instructions past the `lfence`. Thus, when the basic block contains a small number of instructions, the cost of stopping speculation is lower, and the extra instructions and register pressure introduced by Eclipse can dominate the performance benefits gained from not completely preventing speculation. A hybrid scheme incorporating both speculation barriers and Eclipse instrumentation could prove an interesting performance optimization. (We leave as future work the study of such a hybrid prototype.)

8.1.3 Real-world Applications. We further evaluated the performance overhead of the mitigations on four real-world applications: SQLite, Redis, Nginx, and MariaDB. The configuration parameters for applications and benchmark drivers were tuned in order to achieve maximum CPU utilization, and all benchmarked applications and drivers were run on the same host. We performed 10 iterations of each benchmark and report the averages in Table 3.

SQLite. We evaluated SQLite using the Speedtest [103] benchmark: an in-tree part of SQLite compiled into the main binary specifically designed for performance testing. To ensure that our evaluation only contained the overhead incurred by the main `sqlite` application and not the Speedtest benchmark driver, we explicitly annotated the Speedtest-specific functions with compiler attributes which instruct the compiler not to apply Eclipse instrumentation. Additionally, we configured the binary to use an in-memory database to minimize I/O and maximize CPU utilization.

Table 1: SPEC CPU 2017 performance results.

Benchmark	Eclipse	Eclipse-lfence	SLH
600.perlbench_s	4.31%	4.26%	50.82%
602.gcc_s	0.74%	0.76%	49.74%
605.mcf_s	6.52%	26.73%	58.59%
619.lbm_s	0.42%	0.35%	2.62%
620.omnetpp_s	9.05%	22.94%	33.49%
623.xalancbmk_s	8.49%	11.69%	154.36%
625.x264_s	3.85%	10.67%	26.58%
631.deepsjeng_s	0.23%	0.19%	31.49%
638.imagick_s	9.53%	≈0%	97.74%
641.leela_s	1.21%	1.23%	20.03%
644.nab_s	0.29%	0.72%	31.36%
657.xz_s	≈0%	0.13%	54.26%

Table 2: SMAC instructions ratio in userland applications.

Benchmark	# SMAC ind. branches / # Total ind. branches
SQLite	29,152/51764 (56.32%)
Redis	6137/12348 (49.70%)
Nginx	12,261/24489 (50.07%)
MariaDB	90,006/181774 (49.51%)
600.perlbench_s	553/1136 (48.68%)
602.gcc_s	9223/13433 (68.66%)
605.mcf_s	219/446 (49.10%)
619.lbm_s	183/382 (47.91%)
620.omnetpp_s	9728/20654 (47.10%)
623.xalancbmk_s	32,794/63431 (51.70%)
625.x264_s	1915/2992 (64.00%)
631.deepsjeng_s	2427/4926 (49.27%)
638.imagick_s	2886/5406 (53.39%)
641.leela_s	2463/5004 (49.22%)
644.nab_s	186/388 (47.94%)
657.xz_s	260/539 (48.24%)

The benchmark completes 8.61% slower, on average, when hardened with Eclipse, outperforming Eclipse-lfence and SLH, which incur a 12.72% and 55.11% slowdown, respectively.

Redis. We benchmarked Redis using memtier [95]: a tool designed for benchmarking NoSQL key-value databases. We configured memtier to use 3 threads with 8 clients per thread, each sending SET and GET requests for a 32-byte object in a 1:10 ratio over 1 minute. Additionally, Redis was configured to use a single I/O thread. The throughput degradation is identical for SET and GET requests and is negligible for Eclipse, while Eclipse-lfence and SLH incur 0.17% and 3.20% degradation, respectively.

Nginx. We benchmarked Nginx using wrk [2]: an HTTP benchmarking tool. We configured wrk to generate HTTP requests over 1 minute using 8 threads each making 512 simultaneous HTTP connections. We ran the benchmark three separate times, with Nginx serving 1KB, 100KB, and 1MB file sizes in each iteration. Nginx was configured to use 8, 3, and 3 threads for the 1KB, 100KB, and 1MB benchmarks, respectively, to achieve maximum CPU utilization. Eclipse has the lowest throughput degradation for the 1MB file size, at 0.42% and 3.28%, and is slightly outperformed by Eclipse-lfence for the 1KB and 100KB file sizes (1.00% vs 0.67% and 0.65% vs 0.10%, respectively). SLH has the highest degradation across all file sizes, ranging from 2.00% (1KB) up to 3.73% (100KB).

Table 3: Real-world applications performance results.

Application	Eclipse	Eclipse-lfence	SLH
SQLite	8.61%	12.72%	55.11%
Redis (GET/s)	≈0%	0.17%	3.20%
Redis (SET/s)	≈0%	0.17%	3.20%
Nginx (1KB)	1.00%	0.67%	2.00%
Nginx (100KB)	0.65%	0.10%	3.73%
Nginx (1MB)	0.36%	0.78%	3.52%
MariaDB	0.42%	1.60%	10.16%

Table 4: LMbench micro-benchmark results.

	Benchmark	Eclipse Overhead
Latency	getpid()	0.72%
	open()/close()	2.79%
	read()/write()	≈0%
	select(100 fds)	7.73%
	select(100 TCP fds)	6.04%
	stat()	2.16%
	mmap()/munmap()	2.32%
	fork()+exit()	1.10%
	fork()+exec()	0.82%
	fork()+/bin/sh	0.26%
	Install signal	≈0%
	Handle signal	0.93%
	Protection fault	0.71%
	Page fault	≈0%
	Pipe I/O	4.88%
	UNIX socket I/O	7.95%
	TCP socket I/O	2.12%
	UDP socket I/O	6.86%
	Context switch	4.93%
	File creation	1.17%
	File deletion	3.53%
Bandwidth	File I/O	0.92%
	mmap() I/O	0.97%
	Pipe I/O	≈0%
	UNIX socket I/O	3.04%
	TCP socket I/O	2.41%

MariaDB. We benchmarked MariaDB using the oltp_read_write benchmark from the sysbench [1] tool. We ran the benchmark for 5 minutes on a table containing 2-million entries, using 6 worker threads. Eclipse incurs the lowest overhead at 0.42%, outperforming both Eclipse-lfence (1.60%), and SLH (10.16%).

8.2 Kernel Performance

We also evaluated the overhead introduced by Eclipse on the Linux kernel (v5.4.256), using a set of micro-benchmarks (LMbench [74]) and macro-benchmarks (PTS [89]).

LMbench. We measured the latency overhead incurred by Eclipse for various kernel operations, such as: (a) performing system calls (e.g., read()/write(), stat(), *etc.*), (b) installing and catching signals, (c) creating processes, (d) page faults and protection faults, (e) pipe and (TCP/UDP/UNIX) socket I/O, (f) context switching, and (g) file creation and deletion. We also measured the bandwidth reduction for file I/O, (TCP/UDP) socket I/O, and pipe I/O. Results are presented in Table 4. Overall, Eclipse's latency overhead ranges from ≈0%–7.95%, and its bandwidth degradation is < 3.04%.

PTS. We evaluated the performance overhead introduced by an `Eclipse`-instrumented kernel on various real-world applications and workloads, including a web server (Nginx), a database (MariaDB), a deep-learning framework (TensorFlow), a Linux kernel build, kernel benchmarks using `perf-bench`, and benchmarks for OpenSSL and GNU `libc`. `Eclipse` incurs negligible overhead on all benchmarks (< 2%).

8.3 Security Evaluation

In addition, we assessed `Eclipse`'s effectiveness in mitigating SMA attacks on both the x86-64 and ARM architectures.

x86-64. To evaluate whether `Eclipse` successfully protects against SP attacks on x86-64, we apply our prototype to the Linux kernel and demonstrate that our instrumentation successfully blocks the original BlindSide [37] attack that de-randomizes KASLR. We created a vulnerable environment by forward-porting CVE-2017-7308 [26]—the vulnerability used by the BlindSide [37] authors to construct their proof-of-concept exploits—to kernel v5.4.256. Then, we used the exploit by Göktas et al. [37] to carry out an SP attack against the kernel to de-randomize KASLR. In short, the exploit works similar to the SP example described in Section 4.1: it leverages a heap buffer overflow to overwrite a function pointer in a socket structure, which the attacker subsequently uses to speculatively probe for the base address of the kernel image. By running the exploit on an *uninstrumented* kernel, we were able to successfully locate the base address of the kernel (i.e., we detected signals in the cache when "guessing" the kernel's base address), de-randomizing KASLR. After applying `Eclipse`'s instrumentation to the kernel, we could no longer observe any signal(s) in the cache while speculatively probing, verifying that our defense is effective.

ARM. We evaluated `Eclipse`'s effectiveness in mitigating both the (ARM-specific) PACMAN attack and SP on ARM. To evaluate `Eclipse`'s effectiveness against PACMAN, we used the original proof-of-concept exploit publicly provided by the paper authors [92]. The exploit creates a vulnerable kernel extension containing a PACMAN gadget, which is used by the exploit to carry out a PACMAN attack and leak the PAC of a kernel pointer from userland. Since the exploit targets the M1 processor, we made slight modifications to adapt it to our M2 processor (i.e., adjusting for the different cache sizes in M2) and verified that the exploit successfully leaks the PAC. After applying `Eclipse`'s mitigation on the vulnerable kernel extension, the exploit is no longer able to leak the PAC (i.e., we observe no cache signals).

Since there are no publicly available SP exploits against ARM, we developed a proof-of-concept, userland SP exploit that works similar to the SP attack described in Section 4.1 (i.e., it contains a SMAC indirect branch preceded by an attacker-controlled conditional branch). To carry out the attack, we first train the conditional branch predictor, then corrupt the pointer dereferenced by the SMAC indirect branch with the address of a function that accesses a pre-defined memory location, leaving a traceable signal in the cache. Without `Eclipse`, we are able to measure clear cache hits, indicating that the corrupted pointer is speculatively dereferenced. When applying `Eclipse`, we were unable to measure cache hits.

9 Discussion

9.1 Other SMA Attacks

While `Eclipse` is effective at mitigating SMA attacks such as SP and PACMAN on the x86(-64) and ARM architectures, its design is generic and our implementation can be adapted to other SMA attacks. For example, `Eclipse` can mitigate the attack against GCC VTV described in SPEAR [72] by introducing a data dependency between the comparison that verifies a virtual table pointer belongs to a valid set and its subsequent dereference. Additionally, `Eclipse` can be adapted to other architectures if they provide instructions that can capture and link data dependencies.

9.2 Binary Verifier

Although the binary verifier (§5.4) did not discover any cases of the compiler breaking (i.e., insecurely modifying) the instrumentation, it did uncover a bug in the x86(-64) SP implementation of `Eclipse`. Specifically, our implementation would only instrument the first indirect branch in `SMACInstrs` (§5.1), and any subsequent branches in the set would not get instrumented with an `or` instruction. The verifier was able to identify this since, during its analysis, it detected indirect branches in its re-constructed `SMACInstrs` set that were not preceded by an `or` instruction.

9.3 Limitations

Inter-procedural Hardening. `Eclipse` currently performs intra-procedural analysis and hardening. It is *complete* in terms of intra-procedural analysis, meaning it will identify all SMAC instructions within a single function. Occasionally, however, certain code patterns that span multiple functions may enable SMA attacks. For example, in SP, an attacker may be able to execute a *direct* branch to another function that contains an indirect branch, which gets speculatively executed. In current literature, inter-procedural analysis (or hardening) is considered in two cases. First, BlindSide [37] performs an inter-procedural SP gadget analysis. However, the analysis uses a conservative definition of control-dependent (i.e., SMAC) indirect branches and is thus unclear whether the results are complete inter-procedurally. Second, SLH [21] provides a mechanism for *propagating state* inter-procedurally, but it still does not perform any inter-procedural analysis to determine whether a function should be hardened; it simply hardens a function *iff* it contains memory loads. Thus, a more precise analysis is required to correctly identify inter-procedural SMAC instructions, (e.g., by following direct branches that are conditionally executed and hardening any encountered SMAC instructions). We leave this for future work.

Return Address Protection. `Eclipse` does not support hardening return instructions—current literature has not demonstrated SMA attacks that can be carried out using return instructions as SMAC instructions. Nonetheless, there are two adjacent cases worth considering. First, it may seem possible that SP can be carried out using return instructions; however, this attack variant would introduce additional complexity, potentially rendering it infeasible. Even if an attacker corrupts an address used in a conditionally-executed return instruction, they would further need to "re-corrupt" the address back to a valid address, else a crash will occur when the function (eventually) concretely returns using the corrupted address.

Second, the attack against LLVM's SSP (shown in SPEAR [72]) also appears related to return instructions. However, the goal of this attack is to *delay* the canary check and perform speculative ROP (via a corrupted return address) for one iteration, after which the process terminates. Hence, this attack cannot be used to perform SP, and we consider it orthogonal to SMA attacks. If future studies demonstrate that returns can be used to carry out SMA attacks, `Eclipse` can be extended to support hardening return instructions.

Other Side-channel Attacks. `Eclipse` is *oblivious* to the specific side channel that the attacker intends to use; because of `Eclipse`'s instrumentation, during the attack, the SMAC instruction will not use attacker-controlled data when executed, and thus no sensitive data will be accessed or leaked via a specific side channel. `Eclipse` does not mitigate any other speculative-execution-related attacks (i.e., other than SMA attacks). Hence, users interested in protecting against other side-channel attacks would need to combine `Eclipse` with additional mitigations, potentially incurring higher overheads.

10 Related Work

Adapting SLH. `Eclipse` prevents SMA attacks by introducing *artificial data dependencies*, borrowing ideas from SLH [21] and applying them to a specific (narrow-scoped) domain. Prior studies have explored extending SLH to provide stronger security guarantees, or selectively applying SLH to remedy its performance overhead. Patrignani and Guarnieri [85] show that SLH can still leak data loaded non-speculatively, and thus extend SLH to mask the inputs of all instructions that load from memory. Ultimate SLH [119] further extends SLH to apply it to variable-time arithmetic instructions. Oleksenko et al. [82] explore different instrumentations to introduce data dependencies to mitigate Spectre v1 that do not use conditional move instructions. Shivakumar et al. [102] implement *selective SLH*, an improvement over SLH that selectively hardens only values loaded into public variables. Marinaro et al. [73] investigate how the hardening introduced by SLH can be improved when taking into account properties of the underlying microarchitecture of the system. Finally, Blade [110] applies SLH only on select code patterns to improve performance, such as data-paths from sources (secrets) to sinks (cache) in cryptographic code. In antithesis, `Eclipse` focuses solely on SMA and requires identifying and hardening the code patterns leveraged by such attacks.

Intel IBT. Indirect Branch Tracking (IBT) is part of Intel's Control-flow Enforcement Technology (CET) [25], a hardware extension for Intel processors which provides control-flow transfer protection [33]. IBT provides protection for control-flow transfers by *only* allowing a forward-edge transfer if it targets an `endbr` instruction. Importantly, IBT stops control-flow transfers that do not target an `endbr`, even during speculative execution, restricting an attacker's capabilities during an SP attack. However, IBT has two main disadvantages. First, it does not stop speculative execution of *all* indirect branch targets, but only those not starting with an `endbr`. As a result, an attacker can still carry out a SP attack by probing for COP-like [35] gadgets that start with an `endbr`. Second, CET is only available for Intel processors, and thus cannot prevent other SMA attacks that target AMD or ARM processors. In contrast, `Eclipse`'s approach is generic, does not require any platform-specific features, and can be applied to many CPU architectures.

11 Conclusion

SMA allows an attacker to bypass memory-error mitigations and leak sensitive program information by combining memory corruption with speculative execution. To mitigate this threat, we introduced `Eclipse`: a *compiler-based* hardening scheme that stops SMA attacks by propagating artificial data dependencies onto select program data used during an attack. In short, `Eclipse` prevents the CPU from using of attacker-controlled data during speculative execution, providing protection while not disabling speculation entirely, resulting in low performance overhead(s). We implemented `Eclipse` as a compiler pass for x86-64 and demonstrate that our design is both performant and effective. `Eclipse` incurs ≈0%–9.5% overhead in the SPEC CPU 2017 benchmarks, < 8.6% in real-world applications, and negligible overhead in the Linux kernel. For effectiveness, we applied `Eclipse`'s instrumentation to both x86-64 and ARM and show that it mitigates SMA attacks across both the kernel and userland.

Availability

Our prototype implementation of `Eclipse` is available at: https://gitlab.com/brown-ssl/eclipse

Acknowledgments

We thank our anonymous shepherd and reviewers for their valuable feedback. This work was supported by the National Science Foundation (NSF), through award CNS-2238467. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the US government or NSF.

References

[1] 2021. sysbench. https://github.com/akopytov/sysbench.
[2] 2021. wrk – a HTTP benchmarking tool. https://github.com/wg/wrk.
[3] 2023. MariaDB. https://mariadb.com.
[4] 2023. musl libc. https://musl.libc.org/.
[5] 2023. nginx. https://nginx.org.
[6] 2023. Redis. https://redis.io.
[7] 2023. SQLite. https://www.sqlite.org.
[8] 2023. The Void (Linux) distribution. https://voidlinux.org/.
[9] 2024. Alpine Linux. https://alpinelinux.org/.
[10] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2009. Control-Flow Integrity: Principles, Implementations, and Applications. *ACM Transactions on Information and System Security (TISSEC)* 13, 1 (2009), 1–40.
[11] AMD. 2020. SOFTWARE TECHNIQUES FOR MANAGING SPECULATION ON AMD PROCESSORS. https://developer.amd.com/wp-content/resources/Managing-Speculation-on-AMD-Processors.pdf.
[12] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. 2022. Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks. In *USENIX Security Symposium (SEC)*. 971–988.
[13] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: Exploiting Speculative Execution through Port Contention. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 785–800.
[14] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. 2011. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *ACM Asia Symposium on Information, Computer and Communications Security (ASIACCS)*. 30–40.
[15] Kjell Braden, Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Stephen Crane, Michael Franz, and Per Larsen. 2016. Leakage-Resilient Layout Randomization for Mobile Devices. In *Network and Distributed System Security Symposium (NDSS)*. 21–24.
[16] Bugtraq. 1997. Getting around non-executable stack (and fix). https://seclists.org/bugtraq/1997/Aug/63.

[17] Nathan Burow, Xinping Zhang, and Mathias Payer. 2019. SoK: Shining Light on Shadow Stacks. In *IEEE Symposium on Security and Privacy (S&P)*. 985–999.

[18] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium (SEC)*. 249–266.

[19] Canella, Claudio and Genkin, Daniel and Giner, Lukas and Gruss, Daniel and Lipp, Moritz and Minkin, Marina and Moghimi, Daniel and Piessens, Frank and Schwarz, Michael and Sunar, Berk and Van Bulck, Jo, and Yarom, Yuval. 2019. Fallout: Leaking Data on Meltdown-resistant CPUs. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 769–784.

[20] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. 2015. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *USENIX Security Symposium (SEC)*. 161–176.

[21] Chandler Carruth. 2024. Speculative Load Hardening. https://llvm.org/docs/SpeculativeLoadHardening.html.

[22] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. 2022. SoK: Practical Foundations for Software Spectre Defenses. In *IEEE Symposium on Security and Privacy (S&P)*. 666–680.

[23] Shuo Chen, Jun Xu, and Emre C. Sezer. 2005. Non-Control-Data Attacks Are Realistic Threats. In *USENIX Security Symposium (SEC)*. 178–191.

[24] Jonathan Corbet. 2019. Grand Schemozzle: Spectre continues to haunt. https://lwn.net/Articles/795637/.

[25] Intel Corporation. 2019. *Control-flow Enforcement Technology Specification*.

[26] The MITRE Corporation. 2017. CVE-2017-7308.

[27] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. 1998. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *USENIX Security Symposium (SEC)*.

[28] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. 2015. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *IEEE Symposium on Security and Privacy (S&P)*. 763–780.

[29] Jake Edge. 2013. Kernel address space layout randomization. https://lwn.net/Articles/569635/.

[30] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. 2015. Missing the Point(er): On the Effectiveness of Code Pointer Integrity. In *IEEE Symposium on Security and Privacy (S&P)*. 781–796.

[31] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 901–913.

[32] Michael Franz. 2010. E Unibus Pluram: Massive-Scale Software Diversity as a Defense Mechanism. In *New Security Paradigms Workshop (NSPW)*. 7–16.

[33] Alexander J. Gaidis, Joao Moreira, Ke Sun, Alyssa Milburn, Vaggelis Atlidakis, and Vasileios P. Kemerlis. 2023. FineIBT: Fine-grain Control-flow Enforcement with Indirect Branch Tracking. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 527–546.

[34] Go. 2024. The Go Programming Language. https://go.dev/.

[35] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out Of Control: Overcoming Control-Flow Integrity. In *IEEE Symposium on Security and Privacy (S&P)*. 575–589.

[36] Enes Göktaş, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, and Herbert Bos. 2016. Undermining information hiding (and what to do about it). In *USENIX Security Symposium (SEC)*. 105–119.

[37] Enes Göktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. 2020. Speculative Probing: Hacking Blind in the Spectre Era. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1871–1885.

[38] Google Security Blog. 2021. Mitigating Memory Safety Issues in Open Source Software. https://security.googleblog.com/2021/02/mitigating-memory-safety-issues-in-open.html.

[39] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. KASLR is Dead: Long Live KASLR. In *Engineering Secure Software and Systems (ESSoS)*. 161–176.

[40] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium (SEC)*. 897–912.

[41] J.L. Hennessy and D.A. Patterson. 2017. *Computer Architecture: A Quantitative Approach*. Elsevier Science.

[42] Guangyuan Hu, Zecheng He, and Ruby B. Lee. 2021. SoK: Hardware Defenses Against Speculative Execution Attacks. In *International Symposium on Secure and Private Execution Environment Design (SEED)*. 108–120.

[43] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *IEEE Symposium on Security and Privacy (S&P)*. 969–986.

[44] Intel. 2018. Indirect Branch Restricted Speculation. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-restricted-speculation.html.

[45] Intel. 2018. Managed Runtime Speculative Execution Side Channel Mitigations. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/runtime-speculative-side-channel-mitigations.html.

[46] Intel. 2018. Retpoline: A Branch Target Injection Mitigation. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/retpoline-branch-target-injection-mitigation.html.

[47] Intel. 2018. Single Thread Indirect Branch Predictors. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/single-thread-indirect-branch-predictors.html.

[48] Intel. 2023. Intel Analysis of Speculative Execution Side Channels. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/analysis-speculative-execution-side-channels.html.

[49] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. 2018. Block Oriented Programming: Automating Data-Only Attacks. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 1868–1882.

[50] Jann Horn. 2019. speculative execution, variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528.

[51] The Linux Kernel. 2024. Spectre Side Channels. https://docs.kernel.org/admin-guide/hw-vuln/spectre.html.

[52] Khaled N Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2019. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. In *ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[53] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. 2018. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 974–987.

[54] Vladimir Kiriansky and Carl A. Waldspurger. 2018. Speculative Buffer Overflows: Attacks and Defenses. *CoRR* abs/1807.03757 (2018).

[55] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2019. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy (S&P)*. 1–19.

[56] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *USENIX Workshop on Offensive Technologies (WOOT)*.

[57] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 147–163.

[58] Per Larsen, Stefan Brunthaler, and Michael Franz. 2014. Security through Diversity: Are We There Yet? *IEEE Security & Privacy* 12, 2 (2014), 28–35.

[59] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. 2019. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. In *USENIX Security Symposium (SEC)*. 177–194.

[60] Arm Limited. 2024. ARM – Cache Speculation Side channels v2.5. https://developer.arm.com/documentation/102816/latest/.

[61] Arm Limited. 2024. Arm A64 Instruction Set Architecture – AUTDA, AUTDZA. https://developer.arm.com/documentation/ddi0596/2021-12/Base-Instructions/AUTDA--AUTDZA--Authenticate-Data-address--using-key-A-.

[62] Arm Limited. 2024. Arm A64 Instruction Set Architecture – AUTIA, AUTIA1716, AUTIASP, AUTIAZ, AUTIZA. https://developer.arm.com/documentation/ddi0596/2021-12/Base-Instructions/AUTIA--AUTIA1716--AUTIASP--AUTIAZ--AUTIZA--Authenticate-Instruction-address--using-key-A-.

[63] Arm Limited. 2024. Arm A64 Instruction Set Architecture – CSETM. https://developer.arm.com/documentation/ddi0596/2021-12/Base-Instructions/CSETM--Conditional-Set-Mask--an-alias-of-CSINV-.

[64] Arm Limited. 2024. Arm A64 Instruction Set Architecture – LDRAA, LDRAB. https://developer.arm.com/documentation/ddi0596/2021-12/Base-Instructions/LDRAA--LDRAB--Load-Register--with-pointer-authentication-.

[65] Lipp, Moritz and Schwarz, Michael and Gruss, Daniel and Prescher, Thomas and Haas, Werner and Fogh, Anders and Horn, Jann and Mangard, Stefan and Kocher, Paul and Genkin, Daniel and Yarom, Yuval and Hamburg, Mike. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium (SEC)*. 973–990.

[66] Liu, Fangfei and Yarom, Yuval and Ge, Qian and Heiser, Gernot and Lee, Ruby B. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE Symposium on Security and Privacy (S&P)*. 605–622.

[67] LLVM. 2024. The LLVM Compiler Infrastructure. https://llvm.org/.

[68] ARM Ltd. 2018. Addressing Spectre Variant 1 (CVE-2017-5753) in Software. https://developer.arm.com/documentation/102820/latest/.

[69] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P Chung, Taesoo Kim, and Wenke Lee. 2015. ASLR-Guard: Stopping Address Space Leakage for Code

Reuse Attacks. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 280–291.
[70] LWN.net. 2004. x86 NX support. https://lwn.net/Articles/87814/.
[71] Giorgi Maisuradze and Christian Rossow. 2018. ret2spec: Speculative Execution Using Return Stack Buffers. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2109–2122.
[72] A. Mambretti, A. Sandulescu, A. Sorniotti, W. Robertson, E. Kirda, and A. Kurmus. 2021. Bypassing memory safety mechanisms through speculative control flow hijacks. In *IEEE European Symposium on Security and Privacy (EuroS&P)*. 633–649.
[73] Tiziano Marinaro, Pablo Buiras, Andreas Lindner, Roberto Guanciale, and Hamed Nemati. 2023. Beyond Over-Protection: A Targeted Approach to Spectre Mitigation and Performance Optimization. *arXiv preprint arXiv:2312.09770* (2023).
[74] Larry W McVoy and Carl Staelin. 1996. lmbench: Portable Tools for Performance Analysis. In *USENIX Annual Technical Conference (ATC)*. 279–294.
[75] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014 (2014).
[76] Microsoft Docs. 2022. Data Execution Prevention. https://docs.microsoft.com/en-us/.
[77] Microsoft Security Response Center. 2019. A proactive approach to more secure code. https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/.
[78] Marius Momeu, Fabian Kilger, Christopher Roemheld, Simon Schnückel, Sergej Proskurin, Michalis Polychronakis, and Vasileios P. Kemerlis. 2024. ISLAB: Immutable Memory Management Metadata for Commodity Operating System Kernels. In *ACM ASIA Conference on Computer and Communications Security (ASIA CCS)*. 1159–1172.
[79] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. *SIGPLAN Not.* 44, 6 (2009), 245–258.
[80] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2010. CETS: Compiler-Enforced Temporal Safety for C. In *International Symposium on Memory Management (ISMM)*. 31–40.
[81] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. 2016. Poking Holes in Information Hiding. In *USENIX Security Symposium (SEC)*. 121–138.
[82] Oleksii Oleksenko, Bohdan Trach, Tobias Reiher, Mark Silberstein, and Christof Fetzer. 2018. You Shall Not Bypass: Employing data dependencies to prevent Bounds Check Bypass. *CoRR* abs/1805.08506 (2018).
[83] Aleph One. 1996. Smashing The Stack For Fun And Profit. *Phrack Magazine* 7, 49 (1996).
[84] OpenBSD. 2003. i386 W^X. https://marc.info/?l=openbsd-misc&m=105056000801065.
[85] Marco Patrignani and Marco Guarnieri. 2021. Exorcising Spectres with Secure Compilers. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 445–461.
[86] Team PaX. 2003. ASLR. https://pax.grsecurity.net/docs/aslr.txt.
[87] Marios Pomonis, Theofilos Petsios, Angelos D Keromytis, Michalis Polychronakis, and Vasileios P Kemerlis. 2017. kRˆ X: Comprehensive Kernel Protection against Just-In-Time Code Reuse. In *European Conference on Computer Systems (EuroSys)*. 420–436.
[88] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P Kemerlis, and Michalis Polychronakis. 2020. xMP: Selective Memory Protection for Kernel and User Space. In *IEEE Symposium on Security and Privacy (S&P)*. 563–577.
[89] PTS. 2024. Phoronix Test Suite. https://www.phoronix-test-suite.com/.
[90] Qualcomm Technologies Inc. 2017. Pointer Authentication on ARMv8.3. https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf.
[91] Joseph Ravichandran. 2024. PacmanPatcher. https://github.com/jprx/PacmanPatcher.
[92] Joseph Ravichandran. 2024. The PACMAN Attack. https://github.com/jprx/PacmanAttack.
[93] Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan. 2022. PACMAN: Attacking ARM Pointer Authentication with Speculative Execution. In *Annual International Symposium on Computer Architecture (ISCA)*. 685–698.
[94] redhat. 2014. New Security Enhancements in Red Hat Enterprise Linux v.3, update 3. https://static.redhat.com/legacy/f/pdf/rhel/WHP0006US_Execshield.pdf.
[95] Redis. 2023. memtier_benchmark. https://github.com/RedisLabs/memtier{_}benchmark.
[96] Roman Rogowski, Micah Morton, Forrest Li, Fabian Monrose, Kevin Z. Snow, and Michalis Polychronakis. 2017. Revisiting Browser Security in the Modern Era: New Data-Only Attacks and Defenses. In *IEEE European Symposium on Security and Privacy (EuroS&P)*. 366–381.
[97] SANS Institute. 2024. CWE/SANS TOP 25 Most Dangerous Software Errors. https://www.sans.org/top25-software-errors/.
[98] Marc Schink and Johannes Obermaier. 2019. Taking a Look into Execute-Only Memory. In *USENIX Workshop on Offensive Technologies (WOOT)*.
[99] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. 2020. ConTExT: A Generic Approach for Mitigating Spectre. In *Network and Distributed System Security Symposium (NDSS)*.
[100] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 753–768.
[101] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)*. 552–561.
[102] Basavesh Ammanaghatta Shivakumar, Jack Barnes, Gilles Barthe, Sunjay Cauligi, Chitchanok Chuengsatiansup, Daniel Genkin, Sioli O'Connell, Peter Schwabe, Rui Qi Sim, and Yuval Yarom. 2023. Spectre Declassified: Reading from the Right Place at the Wrong Time. In *IEEE Symposium on Security and Privacy (S&P)*. 1753–1770.
[103] SQLite. 2023. Database Speed Comparison. https://sqlite.org/src/file/test/speedtest1.c.
[104] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *IEEE Symposium on Security and Privacy (S&P)*. 48–62.
[105] The Linux kernel. 2024. Page Table Isolation (PTI). https://www.kernel.org/doc/html/next/x86/pti.html.
[106] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *USENIX Security Symposium (SEC)*. 941–955.
[107] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium (SEC)*. 991–1008.
[108] Victor Van der Veen, Nitish Dutt-Sharma, Lorenzo Cavallaro, and Herbert Bos. 2012. Memory Errors: The Past, the Present, and the Future. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 86–106.
[109] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-Flight Data Load. In *IEEE Symposium on Security and Privacy (S&P)*. 88–105.
[110] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kıcı, Ranjit Jhala, Dean Tullsen, and Deian Stefan. 2021. Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade. *ACM Programming Languages* 5, 49 (2021), 1–30.
[111] Zhe Wang, Chenggang Wu, Yinqian Zhang, Bowen Tang, Pen-Chung Yew, Mengyao Xie, Yuanming Lai, Yan Kang, Yueqiang Cheng, and Zhiping Shi. 2019. SafeHidden: An Efficient and Secure Information Hiding Technique Using Re-randomization. In *USENIX Security Symposium (SEC)*. 1239–1256.
[112] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. 2021. Osiris: Automated Discovery of Microarchitectural Side Channels. In *USENIX Security Symposium (SEC)*. 1–18.
[113] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. 2018. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. *Technical report* (2018).
[114] Wikipedia. 2023. NX bit. https://en.wikipedia.org/wiki/NX_bit.
[115] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. 2020. Egalito: Layout-Agnostic Binary Recompilation. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 133–147.
[116] Jianhao Xu, Kangjie Lu, Zhengjie Du, Zhu Ding, Linke Li, Qiushi Wu, Mathias Payer, and Bing Mao. 2023. Silent Bugs Matter: A Study of Compiler-Introduced Security Bugs. In *USENIX Security Symposium (SEC)*. 3655–3672.
[117] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. 2019. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy (Corrigendum). In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 428–441.
[118] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium (SEC)*. 719–732.
[119] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. 2023. Ultimate SLH: Taking Speculative Load Hardening to the Next Level. In *USENIX Security Symposium (SEC)*. 7125–7142.

Temporary page!

LaTeX was unable to guess the total number of pages correctly. As there was some unprocessed data that should have been added to the final page this extra page has been added to receive it.

If you rerun the document (without altering it) this surplus page will go away, because LaTeX now knows how many pages to expect for this document.