Ultimate SLH: Taking Speculative Load Hardening to the Next Level

Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup,
Peter Schwabe, Yuval Yarom

The University of Adelaide, Adelaide, Australia
MPI-SP, Bochum, Germany
IMDEA Software Institute, Madrid, Spain
The University of Melbourne, Melbourne, Australia
Radboud University, Nijmegen, The Netherlands

Abstract

In this paper we revisit the Spectre v1 vulnerability and software-only countermeasures. Specifically, we systematically investigate the performance penalty and security properties of multiple variants of speculative load hardening (SLH). As part of this investigation we implement the "strong SLH" variant by Patrignani and Guarnieri (CCS 2021) as a compiler extension to LLVM. We show that none of the existing variants, including strong SLH, is able to protect against all Spectre v1 attacks in practice. We do this by demonstrating, for the first time, that variable-time arithmetic instructions leak secret information even if they are executed only speculatively. We extend strong SLH to include protections also against this kind of leakage, implement the resulting full protection in LLVM, and use the SPEC2017 benchmarks to compare its performance to the existing variants of SLH and to code that uses fencing instructions to completely prevent speculative execution. We show that our proposed countermeasure offers full protection against Spectre v1 attacks at much better performance than code using fences. In fact, for several benchmarks our approach is more than twice as fast.

1 Introduction

The discovery of the Spectre attack [60] in early 2018 demonstrated that speculative execution, hitherto considered a harmless performance improvement technique, can be exploited for leaking sensitive information. Unlike many other microarchitectural attacks like Meltdown [22, 64, 91, 92, 101, 103, 105, 109, 113, 114], which were discovered concurrently and subsequent to the discovery of Spectre, these Spectre attacks—in particular so-called "Spectre v1" attacks—do not exploit a CPU bug, but a CPU *feature*. As a consequence it seems unlikely that the problems caused by Spectre will be solved by CPU microcode updates or future hardware. As Carruth phrased it in an RWC 2020 talk [25], *"Spectre "v1" is here for decades..."*. This means that at least for the foreseeable future, software handling sensitive data will need to protect against Spectre using *software* countermeasures. To understand such software countermeasures, it is useful to describe a Spectre v1 attack[1] as a four-stage process:

S1 The CPU's branch predictor mispredicts a branch and the CPU speculatively executes instructions following this mispredicted branch;

S2 during this speculative execution, secret data is (made) available in a register;

S3 still as part of speculative execution, this data is transmitted from the register onto a covert channel; and

S4 outside speculative execution—possibly by another process—the data is read from the covert channel.

As **S4** is out of control of the program under attack, countermeasures need to prevent the attack from progressing in one of the first three stages. Clearly the easiest way to prevent Spectre attacks is to prevent speculative execution to happen in the first place, i.e., to stop attacks already in **S1**. This can be accomplished by inserting serializing or speculation-blocking instructions—such as the `lfence` instruction on Intel and AMD CPUs—on the two outcomes of every branch. This countermeasure has indeed been proposed already in the original Spectre paper [60, Sec. VII], and has also been implemented in mainstream compilers. Unfortunately it comes with massive performance decline for typical software [48, 59].

As a cheaper alternative, in 2018 Carruth (following discussions with "Paul Kocher, Thomas Pornin, and several other individuals", and based on a core idea by Horn) proposed *speculative load hardening (SLH)* [24], a countermeasure that targets **S2**. This countermeasure is based on the observation that the most common way in which secret data becomes available in a register during speculative execution is through a speculative load from an unintended and possibly attacker-controlled location in memory. This is, for example, exactly

[1]Spectre v1 attacks are often referred to as "bounds check bypass", but in this paper we consider v1 in the original broader sense as any attack exploiting speculative execution following a mispredicted conditional branch.

what happens if an array-bounds check is mispredicted and data is speculatively loaded out of bounds. The idea of SLH is to maintain a predicate indicating if the execution is currently in a mispredicted branch or not. This predicate is then used to "poison" either the outputs (i.e., values) or inputs (i.e., addresses) of load instructions. Both variants are implemented in LLVM since version 8 and both variants prevent possibly sensitive data from being speculatively loaded into a register in a mispredicted branch. We will in the following refer to the variant poisoning loaded values as *LLVM-vSLH* and the one poisoning addresses in load instructions as *LLVM-aSLH*.

SLH never claimed to be a countermeasure against all Spectre v1 attacks, at least not with the broad definition we use in this paper. Specifically, Carruth [24] lists as one limitation of the approach that it *"does not defend against secret data already loaded from memory and residing in registers"*. Recent work by Patrignani and Guarnieri [82] confirms this limitation by revisiting the SLH countermeasure from a more formal point of view. They introduce a formal model capturing Spectre-v1-style leakage and show that poisoning values loaded from memory is indeed insufficient to protect against all Spectre v1 attacks. However, they also observe that poisoning *addresses* of loads has the additional effect of closing one of the most commonly used covert channels, namely address-dependent cache modifications through loads. In other words, poisoning addresses also targets **S3**. They extend this idea and use poisoning based on the misprediction predicate to also close the additional covert channels captured by their model, namely addresses of stores and branch conditions; they call this variant "strong SLH". We adopt this naming and will refer to this variant as *SSLH*.

However, also Patrignani and Guarnieri [82] leave multiple questions about SLH unanswered, in particular with regards to the application of their formal model to the real world:

- Do the differences between the different variants of SLH—LLVM-vSLH, LLVM-aSLH, and SSLH—actually matter *in practice*?
- How much larger is the performance overhead incurred by SSLH compared to LLVM-vSLH and LLVM-aSLH?
- Does any of the SLH variants indeed protect against all Spectre v1 attacks in practice. That is, does the formal model in [82] adequately capture all covert channels accessible in speculative execution?
- If there are any additional covert channels, can we extend SLH to also close these and if yes, at what cost?

Contributions. In this paper, we set out to answer these questions. We make the following contributions:

- We give a systematic overview of the different variants of SLH. We discuss the gap between the theory and practice and describe how the intricacies of the ISA affect the efficiency of the implementation of variants of SLH. We further analyze the security implications of various design and implementation choices.
- We extend the LLVM implementation of SLH to also support SSLH and evaluate the performance impact of all variants on the SPEC2017 benchmark. As expected, stronger defenses incur larger overheads, but all variants are cheaper than using the `lfence`-based countermeasure targeting **S1**.
- We present a proof-of-concept Spectre v1 gadget that is not prevented by any of the existing variants of SLH. This proof-of-concept is the first demonstration that variable-time arithmetic instructions can also be used as a covert channel to transmit sensitive data from speculatively executed code.
- We present *"ultimate SLH"* (or *USLH* for short), an extension to SSLH that also poisons inputs to variable-time arithmetic instructions. We claim that this countermeasure indeed protects against all Spectre v1 attacks and back this claim by a formal analysis and by highlighting a relation to protections against classical (i.e., non-speculative) timing attacks.
- Finally, we implement ultimate SLH in LLVM and evaluate its performance compared to other variants of SLH and `lfence`-protected code. We show that code protected with USLH is consistently faster than code protected by `lfence` and that in some benchmarks it is more than twice as fast.

Responsible disclosure. We disclosed the Spectre gadgets demonstrated in this paper to Intel, AMD, and Arm. All acknowledged the issue but did not consider that it exposes new threats in their processors and did not require embargo.

Availability of our software. The LLVM patches for implementing SSLH and USLH are available at https://github.com/0xADE1A1DE/USLH. The repository also contains some of our attack code.

Organization of the paper. Section 2 establishes the necessary background on microarchitectural attacks with a focus on transient-execution attacks and existing software countermeasures. Section 3 describes the attacker model. Section 4 explains the differences between variants of SLH and our approach to implementing SSLH. Section 5 presents our attacks. Section 6 introduces ultimate SLH as a systematic countermeasure against all Spectre v1 attacks and presents comparative benchmarks. Finally, we conclude in Section 7.

2 Background

2.1 Microarchitectural Attacks

Modern processors consist of a large number of components, collectively called the *microarchitecture*, that implement the instruction set that the processor supports. Program execution affects the state of the microarchitectural components. At the same time, the microarchitectural state affects program execution speed. Consequently, when multiple programs execute on the same processor, executing one program may affect the performance of another.

Microarchitectural attacks [39] exploit these performance effects to leak sensitive information. Specifically, by monitoring program execution speed, an attacker can determine some of the microarchitectural state and from that infer information on other programs executing on the same processor. Attacks have been demonstrated, exploiting various components, such as buses [81, 117, 122], execution ports [1, 18, 20], data caches [65, 80, 83, 124, 125], instruction and microcode caches [4, 89, 94], address translation [40, 63, 100], branch prediction [2, 3, 37, 38, 128], and other components [50, 76].

Constant-time programming. Many of the published microarchitectural attacks target cryptographic implementations [2, 3, 4, 14, 31, 40, 42, 65, 66, 76, 80, 81, 83, 84, 117, 125, 126]. Consequently, the cryptographic community developed *constant-time programming*, a programming style designed to curb microarchitectural attacks. The idea behind constant-time programming is to prevent flow of secret data into variations in microarchitectural states. In practice, this idea translates into three requirements:

1. No secret-dependent control flow;
2. No memory access to addresses that depend on secret values; and
3. No variable-time arithmetic instructions with secret-dependent arguments.

Constant-time coding is considered a de-facto standard requirement for cryptographic code. Cryptographic software and tools for developing it are often claimed to produce constant-time code [13, 15, 16, 17, 36, 54, 88] and tools for validating or enforcing constant-time coding have been developed [35, 93, 95]. The security of constant-time code has been proven [10] and attempts to relax constant-time requirements have been shown vulnerable [76, 96, 97, 126].

2.2 Speculative and Out-of-Order Execution

To improve run-time performance, modern processors employ a complex execution pipeline. The pipeline consists of two main stages. The frontend is responsible for *fetching* instructions from memory and *decoding* them, converting them to a stream of micro-operations (μops).[2] It then *issues* these μops to the execution engine. The execution engine receives the stream of issued μops and *dispatches* them to execution units. To improve performance and to exploit instruction-level parallelism, the order that the execution engine executes the μops may differ from their order in the program. Instead, the execution engine uses some variant of the Tomasulo algorithm [110] to track dependencies between μops and dispatch them to available execution engines as soon as their dependencies are satisfied. After the μops complete execution, the execution engine *retires* them to the frontend. The frontend ensures that μops retire in program order, maintaining the semantics of the machine code.

[2]The exact distinction between instructions and μops is largely irrelevant for this work and so we mostly use the terms interchangeably.

```
1 if (index < arrayLen) {
2   x = array[index];
3   y = array2[x * 4096];
4 }
```

Listing 1: Example of a Spectre v1 Gadget

When the frontend decodes a branch instruction, it often does not know what the branch destination or outcome is, e.g., because the branch condition is yet to be computed. Rather than stalling, the frontend predicts the branch outcome and proceeds to fetch, decode, and execute instructions based on the prediction. This is called speculative execution. Eventually, the execution unit executes the branch instruction and determines the real destination. In the case that the destination was correctly predicted, execution continues without interruptions. However, in the case of a misprediction, all of the μops that were incorrectly issued are *squashed*, any results computed as part of their execution are dropped, and the execution engine instructs the frontend to resume execution from the correct destination. Instructions may also be squashed when abnormal conditions, such as traps and exceptions, occur.

2.3 Transient Execution Attacks

A common consequence of speculative execution is that some μops get executed although they do not appear in the nominal program order. While these μops are eventually squashed, their *transient execution* may bypass software- and hardware-based security checks. Because squashing drops the results computed in transient execution, this was not considered a security issue. However, transiently executed μops do change the microarchitectural state and their execution can leak sensitive information [21, 60, 64]. Specifically, Spectre-type attacks exploit transient execution following a misprediction of control or data flow [5, 12, 18, 29, 56, 58, 60, 62, 68, 75, 98, 102, 107]. Conversely, Meltdown-type attacks exploit transient execution following abnormal termination of an instruction, for example, due to a trap or microcode assist [22, 64, 91, 92, 101, 103, 105, 109, 113, 114].

In this paper we focus on the Spectre attack, and in particular on Spectre v1 [60]. In this variant, the adversary exploits misprediction of a conditional branch to leak secret information. Listing 1 shows the classical case of a Spectre gadget: the conditional statement at Line 1 nominally preventing execution of the `if` body when `index` is beyond the array bound. However, if the branch mispredicts, the `if` body executes transiently, loading a value from outside the array bound and accessing `array2` at a position that depends on the loaded value. After executing the gadget, the adversary can check which offset in `array2` has been accessed, using, e.g., the Flush+Reload technique [125], and from that infer the value of `x`, which has been loaded from an arbitrary location. Due to the popularity of this example, Spectre v1 is also known as "bounds check bypass". However, security issues due to speculative execution of mispredicted branches go deeper [6, 58].

2.4 Countermeasures for Spectre v1

Execution barriers such as the x86 `lfence` instruction prevent speculation. Inserting an `lfence` at each possible outcome of conditional branches prevents Spectre v1 [48]. However, this comes at a significant performance cost [48, 59]. The performance can improve by only protecting vulnerable branches and several approaches for identifying those have been proposed [18, 53]. However, these have false negatives [59], resulting in failures to protect vulnerable branches [53].

Oleksenko et al. [78] introduce false data dependencies between arguments of leaking instructions and branch conditions to delay the instructions until after the branch is resolved. Speculative Load Hardening (SLH) [24, 82] protects against leaks by tracking the speculation state and masking values during misspeculation. We discuss SLH in more detail in Section 4. To protect against Spectre attacks from JavaScript code, browsers reduced the resolution of timers and disabled shared buffers in an effort of preventing the attacker from observing the microarchitectural state [45, 87, 116]. However subsequent works showed that attackers do not need high-resolution timers to carry out attacks [44, 98]. Additionally multiple works propose hardware-based defenses [8, 9, 55, 57, 67, 73, 99, 106, 120, 123, 127]. As these are not available in commercial processors and cannot be applied to existing hardware, these are outside the scope of this work. We refer the reader to [23] for more information about countermeasures.

Formal approaches. There exist many verification tools for checking that programs are protected against Spectre attacks. The overwhelming majority of these countermeasures and tools focus on Spectre v1; we refer to [27] for a recent overview of formal approaches. Many verification tools [11, 19, 26, 28, 32, 34, 41, 85, 86] are supported by soundness claims. Informally, soundness is stated with respect to a formal model of leakage, and a security policy based on this formal model; a typical soundness claim states that programs that pass verification satisfy the intended policy; in some cases, soundness only holds for bounded executions. Broadly speaking, these policies fall into two different categories: relative policies, requiring that speculative execution does not leak more than sequential execution, and absolute policies, requiring that speculative execution does not leak. Additionally, there are many other verification tools [43, 58, 69, 70, 74, 79, 90, 118, 119, 121] that do not aim for or are not (yet) supported by formal soundness claims. In addition to verification tools, there exist many mitigation tools that automatically transform programs so that they adhere to some intended policy; some of these tools come with a soundness proof [72, 115] whereas others do not (yet) have such proofs [51, 77, 108]. Finally, our work is most closely related to [82]. We defer a precise comparison to this work to the next sections.

3 Attacker Model

We assume a model where some data is tagged as secret. The attacker does not have direct access to secret data. The only way they can access it is by invoking some trusted code that can access this data. When the trusted code executes, it can leak some of the secret data it processes, e.g., by writing the secret data to a public variable. Additionally, the victim code may leak secret data through microarchitectural side channels, for example, by accessing a memory address that depends on secret data. We assume that the provider of the trusted code is aware of the leakage potential and accepts the level of leakage possible through nominal, non-speculative execution of the trusted code. We note that our model covers multiple real-world scenarios that enforce isolation. For example, the secret data and the trusted code could reside in a different process or virtual machine, they can be part of an SGX enclave [33], or the system can use intra-process isolation [52, 104, 112].

For side-channel leakage, we assume the typical leakage model covered by constant-time programming. That is, we assume that memory accesses leak their addresses, branches leak their outcomes, and variable-time instructions leak their arguments. This model is widely accepted for nominal execution, i.e., when the program executes in-order with no speculative execution. For transient instructions, past work assumed and demonstrated leakage of addresses from memory access [60] and of branch conditions [18, 30, 128]. In this work we further demonstrate leakage of information on the arguments of variable-time instructions executed transiently.

The attacker aims to use Spectre v1 to cause the trusted code to leak more secret data than it would leak if it were executed without speculation. For that, we assume that the attacker can cause any conditional branch in the trusted code to mispredict. We assume that the attacker cannot cause mispredictions of indirect branches and return instructions—effective countermeasures for those are available [47, 111]. We further assume that the processor is not vulnerable to Meltdown-type attacks [22, 64, 113].

4 Speculative Load Hardening

The main aim of Speculative Load Hardening (SLH) is to prevent data disclosure via microarchitectural channels during speculative execution of code. For that, SLH tracks a speculation flag whose value depends on the state of speculation. SLH then uses the speculation flag to "poison" (or "harden") sensitive values to ensure that they do not leak. For example, in the LLVM implementation of SLH, the speculation flag is 0 during nominal execution and is `0xFF...FF` while misspeculating. To poison a value, LLVM ORs it with the speculation flag, ensuring that during misspeculation the poisoned value is constant and cannot leak.

4.1 SLH Variants Implemented in LLVM

SLH in LLVM is a compiler pass that aims to protect against Spectre v1 [24]. In particular, LLVM SLH aims to protect against speculative bypass of tests such as array bound checks.

```
JC taken                JC taken
.                       CMOVC -1, %rcx
.                       .
.                       .
JMP out                 JMP out
taken:                  taken:
.                       CMOVNC -1 %rcx
.                       .
out:                    out:
```

Listing 2: Speculative state tracking in LLVM SLH

Speculation flag. To track the speculative state of the program, LLVM SLH uses a register, usually `%rcx`, as a speculation flag, setting all bits of the register to 0 during correct execution and to 1 during misspeculation. To achieve that, LLVM SLH instruments every conditional branch to include a conditional move (`CMOVcc`) in each branch, setting the speculation flag. For the condition of the conditional move, LLVM SLH uses the inverse of the branch condition for the taken branch and the branch condition for the non-taken branch. For example, the branch instruction `JC label` in the left part of Listing 2 is taken if the carry is set. When instrumented (Listing 2 right), LLVM SLH adds a `CMOVC -1, %rcx`, which sets all of the bits of `%rcx` if the carry is set, to the non-taken branch. This `CMOVC` is only expected to execute if the branch is not taken, i.e., if the carry is clear. In the nominal execution, when the branch is not taken the carry is clear, hence the value of `%rcx` does not change. However, if the branch is misspeculated, the `CMOVC` will execute speculatively even though the carry is set. Because conditional moves are not speculated, the value of `%rcx` reflects the status of misspeculation. Similarly, for the taken branch, LLVM SLH adds a `CMOVNC` conditional move instruction, that sets the speculation flag to all ones in the case of a misspeculation.

To transfer the speculation flag across function boundaries, LLVM SLH uses the high bits of the stack pointer. Valid user-space pointers in the x86-64 architecture have their 16 most significant bits all 0. Before a function call, LLVM SLH sets these bits from the speculation flag. That is, in the case of misspeculation, the most significant bits of the stack pointer are set to 1, invalidating the stack pointer. In the function prologue, LLVM SLH further adds code that checks the most significant bits of the stack pointer and sets the speculation flag accordingly. The same mechanism is used to communicate the speculation flag on function return.

Poisoning loaded value. Spectre v1 attacks exploit misspeculation to speculatively bypass data validation tests, such as array bounds checks, and leak the accessed values. LLVM protects against such bypasses by poisoning values loaded from memory during misspeculation. Conceptually, the idea is simple—when a value is loaded from memory, LLVM-vSLH ORs it with the speculation flag. This, effectively, sets the value bits to all-one during misspeculation while leaving the value unchanged during nominal execution.

Poisoning load addresses. Instead of poisoning loaded values, SLH supports an option to poison all load addresses. With this option, LLVM-aSLH poisons the values of the base and index registers of addresses that are not considered fixed (see below). This provides the protection level that SLH promises, i.e., a protection against Spectre v1, because the attacker cannot load data from arbitrary addresses.

Poisoning addresses provides some additional protection against leakage of secret values that the program has nominal access to, e.g. values in registers and those loaded from fixed addresses. Most Spectre attacks use a cache-based covert channel to communicate the leaked value to the attacker. That is, the Spectre gadget accesses a memory location that depends on the secret value in order to communicate the value. Poisoning load addresses ensures that loads in misspeculation use fixed addresses (up to the LLVM SLH definition of a fixed address; see below) thus these addresses are not data-dependent.

4.2 Strong SLH

Patrignani and Guarnieri [82] formalize variants of SLH; most notably they introduce *strong SLH* (SSLH) and provide a proof that SSLH indeed protects against all Spectre v1 attacks. The model used for this proof divides the address space into a private and a public heap. The attacker can write code that has unfettered access to the public heap. However, to access the private heap, the attacker uses a code library that is not under direct attacker control. This code library can be invoked by the attacker code and can call attacker provided subroutines.

While the attacker cannot access the private heap, the execution of the code library can leak the information it processes, either directly by writing it into the public heap, or indirectly, through address-based side channels that leak branch conditions and the addresses of memory accesses. A program is speculatively secure if any information that leaks under speculative execution also leaks under nominal execution.

In order to compare SSLH to LLVM-vSLH and LLVM-aSLH in terms of security and performance impact, we set out to implement this variant. The starting point for this implementation is LLVM-aSLH, but it turns out that in order to match all the assumptions made by the formal model of [82], the protections need to go considerably further.

Load address hardening. While both SSLH and LLVM-aSLH work by hardening addresses of loads, there is a difference in *what* loads are protected. SSLH assumes that all addresses of loads are protected, whereas LLVM-aSLH abstains from protecting "fixed" addresses. Specifically, an address is considered fixed by LLVM if both of the memory base and memory index are values known at compile time. Most notably this includes addresses that add a fixed offset to the stack pointer or to the instruction pointer. As the stack pointer may speculatively store sensitive values, we extend LLVM-aSLH to also harden those addresses in our implementation of SSLH. We do not implement hardening of addresses that add fixed offsets to the instruction pointer. We note that the security proof of Patrignani and Guarnieri [82] holds even when fixed addresses are not hardened.

Store address hardening. LLVM-aSLH does not harden addresses of store instructions. This makes sense when thinking of SSLH as a countermeasure targeting **S2**; however, as the proof of SSLH requires protection at **S3**, addresses of store instructions also require protection. We thus add this in our implementation of SSLH. Store addresses are hardened with the same logic that we also use for load addresses.

Branch hardening. As an additional covert channel that can be used to leak secrets in speculative execution, SSLH also assumes that the conditions of branches are hardened. In our implementation of SSLH we ensure that branch conditions depend on the speculation predicate.

The x86 architecture only supports a limited number of instructions for manipulating the flags. Hence, poisoning the flags, while possible, is inefficient. Instead of poisoning the condition flag used by a branch instruction, we look for the instruction that sets the flag and poison the arguments of this instruction. Specifically, if the arguments are loaded from memory, we poison the load address, just as we do for any memory access. For register arguments, we poison the register value. As with other instructions, we do not poison immediate values or fixed addresses.

A summary of the differences between LLVM-vSLH, LLVM-aSLH, and SSLH is given in Table 1; this table also includes Ultimate SLH (USLH), introduced in Section 6.

5 SLH Security

In this section we set out to answer two questions. First, do the more extensive protections of SSLH compared to LLVM-SLH (both LLVM-vSLH and LLVM-aSLH) matter in practice? Second, are the extensive protections offered by SSLH sufficient to stop all Spectre v1 attacks? We answer these questions by presenting three Spectre gadgets. The first, which exploits secret-dependent control flow (Section 5.1), is basically an adaptation of SMoTherSpectre [18] to Spectre v1. It shows that unprotected branch conditions can indeed be used as a covert channel and that hardening them in SSLH thus really matters. The second and third gadgets demonstrate the use of arithmetic instructions (Section 5.2) and repeat instructions (Section 5.3) whose execution time depends on their arguments to build covert channels. These gadgets show that even the protections implemented by SSLH are not sufficient to protect against all Spectre v1 attacks.

5.1 Exploiting Secret-Dependent Control Flow

Our first proof-of-concept shows that branches that execute speculatively can leak their condition. Consequently, poisoning the branch conditions is essential. We emphasize that SSLH does provide protections against the leak of branch condition whereas LLVM-SLH does not. We note that Spectre leakage through branch prediction has already been demonstrated [18, 30, 128].

```
1  victim(int value, int isPublic) {
2    // Branch training
3    for (volatile int i = 0; i < 200; i++);
4
5    // Safety Check
6    if (isPublic) {
7      if (value == 0) {
8        a2 = a1 | a2;
9        a3 = a2 | a3;
10       ...
11     } else {
12       a1 = crc32(a1, a1);
13       a2 = crc32(a2, a2);
14       ...
15     }
16   }
17 }
```

Listing 3: Victim function for SMoTher attack

Listing 3 shows the code of the victim. (While the example shows C code, in practice, to avoid some of the intricacies of the C compiler, we use equivalent LLVM intermediate code for this and for the other PoCs we present in this section.) To facilitate branch training, we use the technique of Röttger and Janc [98], who observe that branch prediction depends on branch history. The loop in Line 3 sets a fixed history for the authorization branch in Line 6. The attacker then invokes the function twice, each time with `value=0` and `isPublic=1`. This sets the prediction that the bodies of the `if` statements in Lines 6 and 7 should be executed.

The attacker then arranges for the victim function to be called with `isPublic=0` and a secret `value`. We note that a nominal execution with `isPublic=0` does not leak the secret value of `value`. The attacker further arranges for the `if` in Line 6 to be resolved slowly, e.g., by flushing the value of `isPublic` out of the cache. When the function executes, the branch training loop sets the branch history to the same state as in the training. Consequently, the processor mispredicts that the body of the `if` in Line 6 will be executed and proceeds

Table 1: Features of different variants of SLH: **value** is the output (value) of loads masked; **addr** is the address of load instructions masked; **ind. branch** are addresses of indirect branches masked; **cond** are conditionals used by branch instructions masked; **store** are addresses of store instructions masked; **SP+imm**: are "fixed" addresses of the form stack-pointer plus fixed offset in load/store instructions masked; **IP+imm** are "fixed" addresses of the form instruction-pointer plus fixed offset in load/store instructions masked; **rep** is the length of `rep` instructions masked; **arith** are inputs to variable-time arithmetic instructions masked.

SLH variant	value	addr	ind. branch	cond	store	SP+imm	IP+imm	rep	arith
LLVM-vSLH	✓	✗	✓	✗	✗	✗	✗	✗	✗
LLVM-aSLH	✗	✓	✓	✗	✗	✗	✗	✗	✗
SSLH	✗	✓	✓	✓	✓	✓	✗	✗	✗
USLH	✗	✓	✓	✓	✓	✓	✗	✓	✓

to speculatively execute it. The `if` is initially predicted to execute the `then` block, but because `value` is available, the `if` is evaluated quickly, and in the case that `value` is 1, execution proceeds speculatively to the `else` part of the `if` statement. Eventually, the processor evaluates `isPublic` and detects the misprediction. It then squashes all mispredicted instruction and proceeds execution along the correct path.

Attack. The attacker's aim is to distinguish whether the secret `value` is 0 or 1. To achieve that, we rely on the observation that when `value=1`, the processor speculatively executes different instructions than in the case that `value=0`. Specifically, for `value=0` we use a sequence of 48 `or` instructions, whereas for `value=1` we use a sequence of 48 `crc32` instructions.

Port contention spy. To distinguish the execution paths, we rely on port contention [20]. Specifically, the execution unit of the processor contains multiple ports, each can execute some instructions but not others. In particular, `or` uses ports 0, 1, 5, 6, whereas `crc32` uses port 1. Hyperthreads of the same execution core compete on the ports. Consequently, if both hyperthreads issue instructions for the same port, port contention will cause execution delays. Bhattacharyya et al. [18] show that speculatively executed instructions can also produce measurable delays. To exploit port contention, our spy program executes a sequence of 42 `crc32` instructions and measures the execution time of the sequence.

Synchronization. To achieve port contention, we need to ensure that the spy executes the measurement code at the same time that the victim executes the distinguishing code. For rough synchronization, we fork the spy and then the victim and migrate both to hyperthreads of the same core. However, forks are not instantaneous and migration takes time. To better synchronize the processes, we use a shared pointer chasing approach. Specifically, we create a linked list of 50 cache lines that is shared between the victim and the spy. Before forking creates new processes, we flush all of the cache lines of the linked list from the cache. Upon initialization, both processes start following the shared linked list from its head to its tail. Because the linked list is initially out of the cache, following it requires bringing all of the elements from memory. Moreover, because the processor must read a list element to determine the location of the following element, reading the elements from the memory cannot be parallelized.

The first process to follow the list has to wait for each element to be read from memory. When it follows the list, the processor caches the elements. Hence, when the second process starts following the list, it can advance much faster, until reaching the first non-cached element. From this point, both processes progress together, waiting for an element before advancing to the next. We find that after following 50 elements, both processes reach the tail of the list within 5–10 cycles of each other.

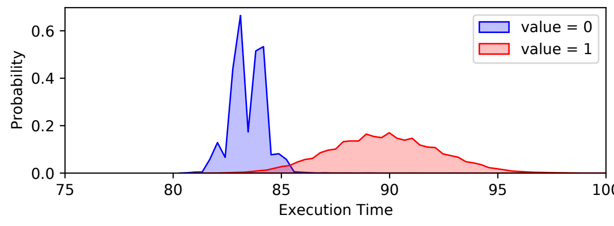

Figure 1: Spy measurement of port contention on Intel Core i7-6700K

Results. We test the code on Intel Core i7-6700K and Core i5-8265U, both running Ubuntu 20.04. We build the victim with LLVM-aSLH, the default implementation of clang 13. We collect 20,000 samples, each consisting of the average spy measurement over 100 runs of the spy and the victim. Figure 1 shows the distribution of the average measurement for the cases that the secret value is 0 and 1, when running on the Core i7-6700K processor. As we can see, when `value=1` the measurements are about 7 cycles longer than when `value=0`, and the two distributions are easily distinguishable, allowing the attacker to determine the secret value.

Vulnerability in real-world software. We implement a gadget searching tool and check if the presented secret-dependent control flow is exploitable in practice. Our tool is an LLVM backend pass that tracks the propagation of function arguments and maintaining lists of tracked variables that could potentially contain secrets. The tool searches for functions that leak information on an argument, but only under speculation. That is, the function meets the following conditions:

- The function contains a conditional branch;
- The code of one of the branch's outcome contains a second branch that depends on an argument of the function; and
- The argument does not leak through memory access or branch condition in any other part of the function.

We apply our gadget-searching tool to the latest available versions of commonly used libraries. Table 2 reports the result of the searching. Note that because we do not have any a-priori method of determining what values are secret, the gadgets we find do not necessarily leak a secret. However, we do point to locations that potentially leak under speculation.

Table 2: Number of gadgets found in common software whose two paths of a branch could potentially be distinguished by memory accesses or function calls.

	OpenSSL 1.1.1q	bash 5.1.16	libgcrypto 1.8.9	musl 1.2.3	python 3.9.14
Found gadgets	359	52	69	48	281

An example of an identified, albeit non-exploitable, gadget is the function `BN_mul_word` in OpenSSL 1.1.1q for big number multiplication (shown in Listing 4). The function accepts two arguments: a big number *a* and an integer *w*. The functions `bn_mul_words` (Line 10) and `bn_wexpand` (Line 12) are both inlined and thus are not considered function calls. Two paths (Line 8 vs. Lines 10–14) can be distinguished through whether the function `BN_zero` (Line 8) is accessed or not (e.g., by using Flush+Reload). We test this gadget with three mitigation options, namely, no mitigation, LLVM-aSLH, and SSLH. The success rates of distinguishing the two paths are 98.57%, 95.23% and 50.03% respectively. This highlights that LLVM-aSLH performs similar to no mitigation while SSLH does prevent the attack. (Note that 50% is the expected success rate for a random guess.)

```
1  int BN_mul_word(BIGNUM *a, BN_ULONG w) {
2    BN_ULONG ll;
3
4    bn_check_top(a);
5    w &= BN_MASK2;
6    if (a->top) {
7      if (w == 0)
8        BN_zero(a);
9      else {
10       ll = bn_mul_words(a->d,a->d,a->top,w);
11       if (ll) {
12         if (bn_wexpand(a,a->top+1) == NULL)
13           return 0;
14         a->d[a->top++] = ll;
15       }
16     }
17   }
18   bn_check_top(a);
19   return 1;
20 }
```

Listing 4: `BN_mul_word` in OpenSSL 1.1.1q

5.2 Exploiting Variable-Time Instructions

We now turn our attention to exploiting instructions whose execution time depends on their arguments. Passing secret information as arguments to such instructions can lead to measurable execution time differences, which leak the secret information [7, 61]. Thus it would appear that such instructions could be used to leak information from speculative execution. We emphasize that neither LLVM-SLH nor SSLH prevent this leakage.

Measuring execution time of misspeculated instructions. To extract leaked secrets from variable-time instructions, past attacks measure the execution time of some code that contain the instructions. However, this approach cannot work for measuring the execution speed of misspeculated code. Typical techniques for accurate time measurement include fence instructions that ensure that the measured code completed execution before the measurement is taken [125], but fences also terminate misspeculation. Consequently, it is impossible to use time measurements in misspeculation. At the same time, the execution speed of misspeculated code does not affect the program's execution time. Misspeculation terminates when the processor detects that it misspeculated and the timing of this detection does not depend on the execution speed of the misspeculated code.

```
victim(double value, int isPublic) {
  // Branch training
  for (volatile int i = 0; i < 200; i++);

  // Boundary Check
  if (isPublic) {
    value = sqrtsd(value);
    value = mulsd(value, value);
    ...
    value = sqrtsd(value);
    value = mulsd(value, value);
    memory_access(adrs);
  }
}
```

Listing 5: Victim function for variable-time instructions

Branch racing. Instead of directly measuring the execution time of misspeculated code, our gadget creates a race condition between the misspeculated code and the branch condition. Listing 5 shows an example of the Spectre gadget we use as a proof-of-concept. The argument `value` can hold one of two values, which we call *fast* and *slow*. Specifically, we use `65536` for *fast* and `2.34e-308` for *slow* [91].

In the misspeculated branch, the code performs a sequence of `SQRTSD` and `MULSD` instructions on `value`, which we call the *leak* sequence. This sequence is followed by a memory access (Line 12). The leak sequence is designed so that it repeatedly computes the square root of the original value of `value`. On our i7-6700K machine, executing a single block of `SQRTSD` and `MULSD` on *fast* takes 17.4 cycles on average, compared with 22.8 for *slow*. In our experiments, misprediction lasts around 240 cycles. Thus, with 11 repetition of the `SQRTSD` and `MULSD` we expect that the leak sequence will complete before the misspeculation ends when executed with the *fast* value, but not when executed with the *slow* value. Hence, if the access at Line 12 executes after the leak sequence completes, the memory access will only happen when `value` is *fast*.

Out-of-order execution. Unfortunately, ensuring that the memory access in Line 12 only executes after the leak sequence completes is not trivial. As discussed, the processor uses out-of-order execution, and will execute an instruction if all of its arguments are available and there is an available execution port. The `adrs` argument of the memory access does not depend on the computation in the leak sequence. Moreover, load instructions use ports 2 and 3, whereas the `SQRTSD` uses port 0 and `MULSD` uses ports 0 and 1. Consequently, there is no conflict between the leak sequence and the memory access, and the processor executes the memory access as soon as speculation starts.

False dependency. A naive straw-man approach is to ensure that the memory access is only executed after the leak sequence. The idea is to create a false dependency between the result of the leaky sequence and the address of the memory access. While none of the existing SLH variants is designed to protect against leaking instruction timing, the result of using this approach seems to indicate that LLVM-aSLH and SSLH protect against leakage, contradicting the analysis.

We note that the false dependency that forces the memory access to evaluate after the leak sequence also affects SLH's detection of fixed addresses. Both LLVM-aSLH and SSLH poison non-fixed addresses, including the read from `addr`. Poisoning affects the gadget in two ways. First, it creates a dependency between the branch condition and the memory access. Consequently, the memory access cannot happen before the branch condition is evaluated. This creates a race between resolving the branch and accessing the memory, which the branch is likely to win both because it is older and because poisoning needs to execute at least two more instructions: the conditional move that sets the speculation flag and the actual poisoning. Moreover, even if the memory access starts executing before the branch resolves, the location it accesses is likely to be invalid, blocking the Flush+Reload channel. We note however that poisoning the address only masks the access location not the fact that the access happens. Thus, it may be possible to create a gadget that relies on false dependency that remains exploitable in the presence of address poisoning. We leave investigating this possibility to future work. In summary, the straw-man approach cannot be directly used for measuring the execution time.

5.2.1 Time measurement with resource contention

We saw how to exploit variable-timing instructions together with a false dependency to create a covert channel for a Spectre gadget. However, due to the false dependency, SLH does not identify that the address used is constant. Hence, SLH poisons it, and "unintentionally" protects against the attack.

We now demonstrate a Spectre gadget that exploits variable-timing instructions without creating a false dependency between these instructions and the subsequent memory access. Our gadget relies on creating contention on internal resources required for scheduling μops execution. We first describe the relevant steps that the execution engine takes while running a program. We then explain how our gadget operates.

Reservation stations. Recall that the execution engine of the processor receives a stream of μops, which it executes. To exploit instruction-level parallelism, the execution engine does not execute μops in program order. Instead μops can be executed in any order that satisfies the data dependencies in the program. To track the data dependencies of a μop, the processor uses reservation stations [46], also known as scheduler entries in the Intel nomenclature [71]. Thus, μop execution consists of allocating a reservation station and other resources required for its execution. The reservation station waits until all inputs for the μop are available, at which time the scheduler queues the μop to one of the appropriate execution units.

When μops' execution takes a long time, the processor may run out of reservation stations and other resources required for their execution. When these resources are required for tracking data dependencies, as is the case with reservation stations, younger instructions cannot be safely scheduled, and their execution is stalled even if they do not depend on older instructions which are pending.

Gadget evaluation. We test the gadget in Listing 5 on an Intel Core i5-8265U, microcode 0xEA, and on an Intel Core i7-10710U, microcode 0xE8, both running Ubuntu 20.04 and both with the CPU governor set to performance. To use the gadget, we first execute the victim twice with public values, training the branch. We then flush `adrs` from the cache and execute the victim with a 'secret' `value`, which can be either *fast* or *slow*. For this attack execution we delay the evaluation of `isPublic` so that the branch in Line 6 mispredicts. Finally, when the function returns we check whether `adrs` is cached.

We collect 100,000 samples on each processors, where in each sample `value` is randomly chosen as either *fast* or *slow*.

Results. The results depend on the number of pairs of `SQRTSD` and `MULSD` instructions we use. With 40 such pairs, the memory access always executes and we observe that with a high probability, `adrs` is cached (99.9% for *slow* value). When `SQRTSD` and `MULSD` are repeated 55 times, we observe that, with a low probability, `adrs` is cached (4.3% for *fast* value). However, when the number of `SQRTSD` and `MULSD` instructions is between these values, we find that whether `adrs` is cached depends on the chosen `value`.

Specifically, for 45 repetitions of `SQRTSD` and `MULSD` we find that when `value` is *fast*, with a high probability (92.5% on the i5-8625U and 96.6% on the i7-10710U) `adrs` is cached. Conversely, when `value` is *slow*, the probability that `adrs` is cached is 5.2% and 4.5% for the i5-8625U and the i7-10710U, respectively. Moreover, building the proof-of-concept with any of the SLH variants in Section 4 does not avoid the leak.

Discussion. In the gadget in Listing 5, the memory access in Line 12 does not depend on any of the prior instructions. Moreover, load instructions use ports 2 and 3, whereas the `SQRTSD` and `MULSD` instructions use ports 0 and 1. Consequently, data dependency and execution unit availability do not explain the stall of the memory access.

We believe that the cause of the stall is resource exhaustion. The long sequence of `SQRTSD` and `MULSD` instructions consume resources required for scheduling further instructions, possibly reservation stations. As the execution of the `SQRTSD` and `MULSD` instructions completes speculatively, the processor frees the resource they consume, gradually releasing younger instructions to be scheduled. Given sufficient time, enough `SQRTSD` and `MULSD` instructions will complete execution to allow the memory access to execute. However, misspeculation only lasts until the processor computes the branch condition. Hence, we have a race between detecting the misspeculation and performing the memory access. When the number of `SQRTSD` and `MULSD` instructions is small, the memory access always wins the race. When the number of `SQRTSD` and `MULSD` instructions is sufficiently high, detecting the misspeculation always wins. However, when the number is between these extremes, the winner is determined by the rate at which the `SQRTSD` and `MULSD` instructions are executed—with *fast* value, the memory access wins and gets executed, whereas with *slow* value, the misspeculation detection wins and the memory access is not executed.

5.2.2 Leak secret bit-by-bit

We have seen that variable-time instructions can leak information through the race between the misspeculation and the branch resolving. In the following, we demonstrate how our variable-timing instructions gadget leaks a secret bit-by-bit, indicating that SSLH fails to prevent leakage in practice.

Bit-by-bit leakage. We present our gadget design in Listing 6. Since we aim at leaking a secret bit-by-bit, the function receives an additional input `bit` to indicate which bit to leak. When `isPublic=1`, the function selects *fast* or *slow* based on the value at position `bit` of `secret`, followed by a multiplication, 47 pairs of `SQRTSD` and `MULSD`, and a memory access to a secret-irrelevant address.

Gadget evaluation. We conduct our test on an Intel Core i7-10710U, microcode 0xF0 running Ubuntu 20.04 and the CPU governor set to performance. We train the branch at Line 6 by invoking the gadget twice with `isPublic=1` . We then flush the `adrs` from the cache. After that, we call the victim function with `isPublic=0` and a randomly selected `secret`. We check which value, *fast* or *slow*, is selected by reloading the flushed `adrs`. If the memory is cached, it means that the *fast* is selected, thus the tested secret bit is 1. Otherwise, the tested bit is 0. In our experiment, we compiled the gadget with no mitigation and with SSLH. We use the value 0xAB for `secret` and collect 10,000 samples.

Results. When the gadget is compiled without mitigation, the attack has the success rate of 82.2% in correctly identifying that the secret is 0xAB. When the gadget is compiled with SSLH, the success rate is 75.8%. These results confirm that variable-time instructions can bypass the SSLH protection and leaking a secret bit-by-bit is feasible.

Discussion. SSLH fails to protect the gadget in Listing 6 because the branch contains neither control flow transfer nor secret-relevant memory access. This indicates that protections against variable-timing instructions require poisoning the operands of variable-timing instructions. We describe our solution when we present USLH in Section 6.

```
1  victim(int secret, int bit, int isPublic) {
2    // Branch training
3    for (volatile int i = 0; i < 200; i++);
4
5    // Boundary Check
6    if (isPublic) {
7      uint64_t tmp =
8      ((secret >> bit) & 1) ? FAST : SLOW;
9      double tmp2 = tmp * tmp;
10     tmp2 = sqrtsd(tmp2);
11     tmp2 = mulsd(tmp2, tmp2);
12     ...
13     tmp2 = sqrtsd(tmp2);
14     tmp2 = mulsd(tmp2, tmp2);
15     memory_access(adrs);
16   }
17 }
```

Listing 6: Bit-by-bit leakage via a selection of *fast* and *slow*

5.3 Exploiting Repeat Instructions

One of the oddities of the x86 instruction set is repeat instructions. Originally added to simplify string and memory operations, these instructions perform one or more memory access and automatically increment or decrement the addresses they use, so repeated use of the instructions will perform the operation on successive addresses. Moreover, the instructions support several repeat prefixes that, when present, cause the operations to execute in a loop controlled by the `%rcx` register and possibly an additional condition on the data processed.

Repeat string operation prefix. The repeat string operation prefix specifies the number of times a string operation is to be repeated. This number is implicitly determined by the value of `%rcx` (e.g., `REP`) or `ZF` flag (e.g., `REPZ`) [49]. When a repeat instruction tests the `ZF` flag, the termination is determined by a counter register. The number of repetitions affects the amount of hardware resources consumed by repeat instructions. Under speculative execution, a younger instruction will not be speculatively executed if the required hardware resources are not available. Listing 7 presents a gadget demonstrating that repeat instructions can leak a value of the counter register.

```
1  victim(int* dst, int* src, int secret,
2                 int isPublic) {
3    // Branch training
4    for (volatile int i = 0; i < 200; i++);
5
6    // Boundary Check
7    if (isPublic) {
8      int rep = (secret == 1) ? 50 : 100;
9      memcpy(dst, src, rep);
10     memory_access(adrs);
11   }
12 }
```

Listing 7: Exploiting REP instruction

Gadget evaluation. With proper instruments, `memcpy` (Line 9 of Listing 7) could be lowered to `REP MOVSB`[3]. The repetition of `REP MOVSB` is either 50 or 100, which is determined by `secret` via a constant-time selection.

We compile the gadget with SSLH and test it on the i7-10710U, microcode 0xF0 running Ubuntu 20.04 with the CPU governor set to performance. We train the branch by executing the gadget twice with `isPublic=1`. Then we flush `adrs` from the cache and execute the victim. We delay the evaluation of the branch at Line 7 by flushing `isPublic`. During the misprediction, the repeat instruction is speculatively executed while the memory access (Line 10) may or may not be executed depending on the available hardware resources. After the execution of the victim, we reload the `adrs` and check if it is cached.

Results. We collect 100,000 samples with randomly generated `secret` (either 0 or 1). Our results show that when the secret is 0, `adrs` is *not* speculatively accessed with a probability of 99.93%. On the other hand, when the secret is 1, `adrs` is speculatively accessed with a probability of 97.85%.

Discussion. This experiment confirms that SSLH does not provide a full protection against Sprectre v1 attacks. Note that SSLH fails to protect leakages from repeat instructions because it does not consider variants of `REP MOV` instructions as memory-related operations, thus does not harden those instructions. We introduce our USLH solution in Section 6.2.

6 Ultimate Speculative Load Hardening

The attacks we presented in Section 5 demonstrate that—short of preventing speculative execution with fences—currently there are no software-based countermeasures that block all forms of Spectre v1 attacks. The presented gadgets exploiting variable-time arithmetic and `REP` instructions are somewhat specific, and are unlikely to be found in real software. Nonetheless, the risk they present is twofold. First, the gadgets show that assumptions made in prior security proofs do not hold in practice. For example, past works assume that analyzing the case of maximum misspeculation results in a worst-case leakage [41, 82]. However these ignore the impact of instruction timing on the speculation window, as demonstrated in Section 5.2.1. Second, the history of side-channel attacks shows that in many cases there are non-obvious exploits to weaknesses. For example, both Bernstein [14] and Osvik et al. [80] identify cache banks as a potential security weakness, but the first practical attack that exploits them was only published a decade later [126]. Hence, while our examples are artificial, it is impossible to preclude the presence of gadgets that exploit similar effects in real-world software.

In this section we extend our implementation of SSLH to also harden variable-time arithmetic; we call the resulting variant *ultimate SLH* (USLH). We show that USLH protects against all of the gadgets we present in Section 5. Moreover, because USLH poisons all instructions that may conflict with constant-time programming [15], we believe that USLH also protects against future variants of Spectre v1. We first describe how we implement hardening of variable-time arithmetic. We then evaluate how USLH blocks leakage from our gadgets. Last, we evaluate the performance impact of USLH.

6.1 USLH Implementation

USLH is basically SSLH with protection for variable-time instructions. We now describe how we add this protection.

Hardening repeat instructions. Repeat instructions move memory blocks from one memory address to another (i.e., for the repeat counter `%rcx`), and thus are not considered as loading a variable from a memory. As executing repeat instructions may leak the number of times they execute, we also poison `%rcx`.

Hardening floating-point instructions. For floating-point instructions, we harden SSE2, vector and X87 floating-point

[3] llc -mtriple=x86_64-linux-gnu -x86-use-fsrm-for-memcpy -mattr=+fsrm

instructions. For vector and SSE2 instructions, we poison all arguments. X87 instructions use an internal value stack for operations. Because we cannot poison the values in the internal stack, we insert an `lfence` speculation barrier in every basic block that uses X87 floating-point instructions.

6.2 Testing USLH Security

Hardening branches. Branch-condition hardening is part of our implementations of both SSLH and USLH. To demonstrate the effectiveness of the defense against our control-flow attack (Section 5.1), we compile the victim function with USLH and repeat experiment. We run tests 20,000 times and each time we take the average of 100 samples. The result, shown in Figure 2, demonstrates that the distributions of execution times for the cases of a secret value 0 and 1 are indistinguishable. This is in stark contrast with the unprotected case in Figure 1.

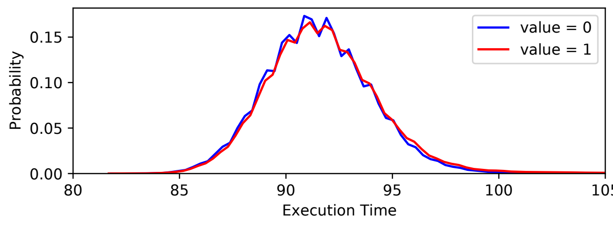

Figure 2: Mitigating V1 SMoTher Attack

Hardening variable-time instructions. We test the two gadgets that exploit variable-time instructions with USLH. USLH poisons the arguments of the floating-point instructions (and variable-time integer instructions such as `DIV64`). Therefore, during misspeculation their timings are constant and do not depend on the value of the secret.

Interestingly, for the false-dependency variant, we never observe that the memory access executes, even when we reduce the length of the leak sequence. We suspect that due to the false dependency, the dependencies of the memory access are only satisfied when the branch condition is evaluated. At this time the branch gets executed and squashes the transient execution of the memory access before the latter has the opportunity to execute.

For the resource contention variant, we observe that when the number of repetitions of the `SQRTSD` and `MULSD` instructions drops below 26, we always observe the memory access, and above that threshold we never observe the memory access. Either way, we cannot distinguish between secret values.

Protecting bit-by-bit leakage. We compile the gadget in Listing 6 with USLH, which poisons operands of variable-timing instructions. We run tests 10,000 times. Every time it identifies that the secret byte is 0x0. Table 3 shows the comparison among compiling the gadget without mitigation, with SSLH and with USLH. Even though SSLH reduces the

Table 3: Result (in %) of identifying the secret byte 0xAB

	All unset	All correct	1 bit wrong	2 bits wrong	3 bits wrong	4 bits wrong	>4 bits wrong
No Mitigation	0.80	82.22	10.47	6.60	0.49	0.09	0.05
Protected by SSLH	5.22	75.82	15.56	4.09	1.38	1.27	0.38
Protected by USLH	100.00	0.00	0.00	0.00	0.00	0.00	0.00

attack success rate, it does not prevent the leakage. USLH does protect the leakage, i.e., the attack always fails with identifying the secret to be all zero.

Hardening repeat instructions. Compiling the gadget in Listing 7 with USLH, the `%rcx` register is poisoned as shown in Listing 8. The string operations with repeat prefix are moving data from one memory address to another address and the operands of these instructions are both registers. Therefore, string operations with repeat prefix are not under the definition of load data from memory or store data to memory by LLVM-SLH and SSLH. As a result, all variants of SLH (excluding the USLH) treat repeat instructions as harmless. With USLH, the register counter (`%rcx`) is poisoned. This suffices to prevent the execution of repeat instructions and younger instructions. Specifically, when the number of repetition is not available under speculative execution, the pipeline is stalled and waits for the availability of the number of repeat. We repeat the experiment 100,000 times; `adrs` is never accessed regardless of the number of repeats.

```
CMOVGEQ     -1, %r9;update the predicate state
CMPQ         1, %rdx; constant time select
MOVL        50, %rcx
MOVL       100, %ecx
CMOVEQ    %rax, %rcx
ORQ        %r9, %rcx; poison the counter
REP;MOVSB  (%rsi), %es:(%rdi)
MOVQ        adrs, %rax
```

Listing 8: Highlighted instruction is introduced by USLH

6.3 Security Analysis

USLH prevents speculative leakage. More specifically, all leakage that occurs during speculative execution only depends on the program text, and not on secret values. As a consequence, programs output by USLH satisfy relative constant-time, an information flow policy stating that speculative execution does not leak more than sequential execution [6, 27, 41].

The proof of relative constant-time is established w.r.t. a formal semantics. Our semantics introduces leakage for variable-time arithmetic instructions, which was not considered in prior work [6, 11, 26]. Formally, the semantics is described by a transition relation $\langle C,b\rangle \xrightarrow[d]{o} \langle C',b'\rangle$. The relation says that one step execution under adversarially controlled directive d transitions from configuration $\langle C,b\rangle$ to configuration $\langle C',b'\rangle$, leading to observation o. The booleans b and b' denote if execution is misspeculating; $\top$ corresponds to misspeculative execution. The directive d is chosen by the attacker to drive execution of control-flow instructions (we assume the attacker has control over control flow) and unsafe memory instructions (we assume that the attacker has control over addresses of unsafe accesses).

The main technical lemma states that for every C, C', d such that $\langle C,\top\rangle \xrightarrow[d]{o} \langle C',\top\rangle$, the observation o only depends on the program text, and thus does not leak any information about C and C'. We provide a formal proof of our claim, in the setting of a core language, in Appendix A.

6.4 SLH Performance Overhead

In this section we report on our performance evaluation of the different variants of SLH. For this evaluation we use the SPEC2017 benchmark, compiled with clang and clang++ at optimization level O3. All experiments were run on a machine with an Intel i7-10710U CPU at microcode 0xE8 running Ubuntu 20.04. We set the performance governor to performance and we only test the performance of single-thread execution. The summary of the results is displayed in Figure 3. More details can be found in Appendix B.

As a baseline we benchmark unprotected code and as an alternative to SLH we also include code protected with the `lfence` countermeasure. This countermeasure prevents speculative execution at each branch and thus systematically prevents Spectre v1 attacks. It is thus a minimum requirement for any other Spectre v1 countermeasure to achieve better performance—we see that this is the case for all SLH variants, and by quite a margin.

Aside from benchmarking the four variants of SLH discussed in the paper, i.e., LLVM-vSLH, LLVM-aSLH, SSLH, and USLH, we also benchmark the cost of only computing the misprediction predicate, but not using it to poison any values (“Trace Only”). We run this additional benchmark to obtain a better understanding of what causes most of the slowdown in SLH: the tracing, which also requires one register, or the poisoning. We see that both contribute significantly to the slowdown, but to varying degrees in different benchmarks. This makes sense, as tracing alone is expected to be quite costly in branch-heavy code and in scenarios with high register pressure.

We see that all SLH variants incur a significant overhead, slowing down some of the benchmarks by a factor of three. However the difference between the four variants is relatively small. Unsurprisingly, USLH incurs notable additional overhead compared to SSLH only in the floating-point benchmarks. Both SSLH and USLH have a somewhat increased cost compared to the two implementations in LLVM, but this cost is not dramatic in any of the benchmarks and it is close to zero in some. The conclusion we draw from this is that applications that can afford the slowdown incurred by SLH are very likely to also tolerate the small additional cost of USLH. This will give them protection not only against the exploitation of some common Spectre v1 gadgets, but a systematic protection against all Spectre v1 attacks at a cheaper price than using `lfence`.

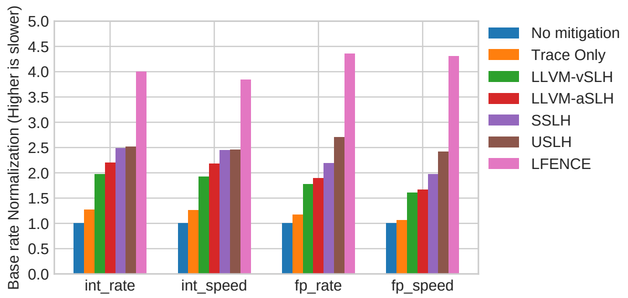

Figure 3: SPEC2017 Summary

7 Conclusion

We revisited speculative load hardening, the most promising Spectre v1 software-only countermeasure. We analyzed the differences between three different existing variants of SLH from a performance and security point of view. We presented a novel proof-of-concept attack exploiting non-constant-time arithmetic instructions in speculatively executed code. This novel attack is not prevented by any of the previously proposed variants of SLH, including the “strong SLH”. The reason is that the underlying model of “strong SLH” overlooks that variable-time arithmetic may leak in speculative execution. We showed that SLH can be extended to also protect against the novel attack and claimed that this variant of SLH indeed protects against all Spectre v1 attacks—this claim is motivated by the fact that all sources of leakage considered for constant-time in the *non-speculative* domain are eliminated in the speculative domain. This is proven in a formal model capturing all these sources of leakage.

Acknowledgments

This work has been supported by an ARC Discovery Early Career Researcher Award DE200101577; an ARC Discovery Project number DP210102670; the Blavatnik ICRC at Tel-Aviv University; the European Commission through the ERC Starting Grant 805031 (EPOQUE); by Deutsche Forschungsgemeinschaft (DFG, German research Foundation) as part of the Excellence Strategy of the German Federal and State

Governments – EXC 2092 CASA - 390781972; by the German Federal Ministry of Education and Research (BMBF) in the course of the 6GEM research hub under grant number 16KISK038; and a gift by Google.

References

[1] Onur Acıiçmez and Jean-Pierre Seifert. Cheap hardware parallelism implies cheap security. In *FDTC*, pages 80–91, 2007. 3

[2] Onur Acıiçmez, Shay Gueron, and Jean-Pierre Seifert. New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures. In *IMACC*, pages 185–203, 2007. 3

[3] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *CT-RSA*, pages 225–242, 2007. 3

[4] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *CHES*, pages 110–124, 2010. 3

[5] Ayush Agarwal, Sioli O'Connell, Jason Kim, Shaked Yehezkel, Daniel Genkin, Eyal Ronen, and Yuval Yarom. Spook.js: Attacking Chrome strict site isolation via speculative execution. In *IEEE SP*, 2022. 3

[6] Basavesh Ammanaghatta Shivakumar, Jack Barnes, Gilles Barthe, Sunjay Cauligi, Chitchanok Chuengsatiansup, Daniel Genkin, Sioli O'Connell, Peter Schwabe, Rui Qi Sim, and Yuval Yarom. Spectre declassified: Reading from the right place at the wrong time. In *IEEE SP*, 2023. 4, 12, 20

[7] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On subnormal floating point and abnormal timing. In *IEEE SP*, pages 623–639, 2015. 8

[8] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. SpecShield: Shielding speculative data from microarchitectural covert channels. In *PACT*, pages 151–164, 2019. 4

[9] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. Isolating speculative data to prevent transient execution attacks. *IEEE Comput. Archit. Lett.*, 18(2):178–181, 2019. 4

[10] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. System-level non-interference for constant-time cryptography. In *CCS*, pages 1267–1279, 2014. 3

[11] Gilles Barthe, Sunjay Cauligi, Benjamin Gregoire, Adrien Koutsos, Kevin Liao, Tiago Oliveira, Swarn Priya, Tamara Rezk, and Peter Schwabe. High-assurance cryptography in the Spectre era. In *IEEE SP*, 2021. 4, 12, 20

[12] Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Neil Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos Rozas, Adam Morrison, Frank McKeen, Fangfei Liu, Ron Gabor, Christopher W. Fletcher, Abhishek Basak, and Alaa R. Alameldeen. Speculative interference attacks: breaking invisible speculation schemes. In *ASPLOS*, pages 1046–1060, 2021. doi: 10.1145/3445814.3446708. 3

[13] Dmitry Belyavsky, Billy Bob Brumley, Jesús-Javier Chi-Domínguez, Luis Rivera-Zamarripa, and Igor Ustinov. Set it and forget it! Turnkey ECC for instant integration. In *ACSAC*, pages 760–771, 2020. 3

[14] Daniel J. Bernstein. Cache-timing attacks on AES. `https://cr.yp.to/antiforgery/cachetiming-20050414.pdf`, 2005. 3, 11

[15] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In *Latincrypt*, pages 159–176, 2012. 3, 11

[16] Daniel J. Bernstein, Chitchanok Chuengsatiansup, and Tanja Lange. Curve41417: Karatsuba revisited. In *CHES*, pages 316–334, 2014. 3

[17] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. NTRU prime: Reducing attack surface at low cost. In *SAC*, pages 235–260, 2017. 3

[18] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: Exploiting speculative execution through port contention. In *CCS*, pages 785–800, 2019. 3, 4, 6, 7

[19] Robert Brotzman, Danfeng Zhang, Mahmut Taylan Kandemir, and Gang Tan. SpecSafe: detecting cache side channels in a speculative world. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–28, 2021. 4

[20] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *IEEE SP*, pages 870–887, 2019. 3, 7

[21] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security*, pages 249–266, 2019. 3

[22] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on meltdown-resistant CPUs. In *CCS*, pages 769–784, 2019. 1, 3, 4

[23] Claudio Canella, Sai Manoj Pudukotai Dinakarrao, Daniel Gruss, and Khaled N. Khasawneh. Evolution of defenses against transient-execution attacks. In *ACM Great Lakes Symposium on VLSI*, pages 169–174, 2020. 4

[24] Chandler Carruth. Speculative load hardening. https://llvm.org/docs/SpeculativeLoadHardening.html, 2018. Accessed: 2021-12-18. 1, 2, 4, 5

[25] Chandler Carruth. Cryptographic software in a post-Spectre world. Talk at the Real World Crypto Symposium, 2020. https://chandlerc.blog/talks/2020_post_spectre_crypto/post_spectre_crypto.html Accessed: 2022-01-13. 1

[26] Sunjay Cauligi, Craig Disselkoen, Klaus von Gleissenthall, Dean M. Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. Constant-time foundations for the new Spectre era. In *PLDI*, pages 913–926, 2020. 4, 12, 20

[27] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. SoK: Practical foundations for software Spectre defenses. In *IEEE SP*, 2022. 4, 12

[28] Kevin Cheang, Cameron Rasmussen, Sanjit Seshia, and Pramod Subramanyan. A formal approach to secure speculation. In *CSF*, 2019. doi: 10.1109/CSF.2019.00027. 4

[29] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten-Hwang Lai. SgxPectre: Stealing Intel secrets from SGX enclaves via speculative execution. In *IEEE EuroS&P*, pages 142–157, 2019. 3

[30] Md Hafizul Islam Chowdhuryy, Hang Liu, and Fan Yao. BranchSpec: Information leakage attacks exploiting speculative branch instruction executions. In *ICCD*, pages 529–536, 2020. 4, 6

[31] Chitchanok Chuengsatiansup, Daniel Genkin, Yuval Yarom, and Zhiyuan Zhang. Side-channeling the Kalyna key expansion. In *CT-RSA*, 2022. 3

[32] Robert J. Colvin and Kirsten Winter. An abstract semantics of speculative execution for reasoning about security vulnerabilities. In *FM*, 2019. 4

[33] Victor Costan and Srinivas Devadas. Intel SGX explained. Cryptology ePrint Archive Report 2016/86, 2016. 4

[34] Lesly-Ann Daniel, Sebastian Bardin, and Tamara Rezk. Hunting the haunter — efficient relational symbolic execution for Spectre with Haunted RelSE. In *NDSS*, 2021. 4

[35] Sushant Dinesh, Grant Garrett-Grossman, and Christopher W. Fletcher. SynthCT: Towards portable constant-time code. In *NDSS*, 2022. 3

[36] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In *IEEE SP*, pages 1202–1219, 2019. 3

[37] Dmitry Evtyushkin, Dmitry V. Ponomarev, and Nael B. Abu-Ghazaleh. Jump over ASLR: attacking branch predictors to bypass ASLR. In *MICRO*, pages 40:1–40:13, 2016. 3

[38] Dmitry Evtyushkin, Ryan Riley, Nael B. Abu-Ghazaleh, and Dmitry Ponomarev. BranchScope: A new side-channel attack on directional branch predictor. In *ASPLOS*, pages 693–707, 2018. 3

[39] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptogr. Eng.*, 8(1):1–27, 2018. 3

[40] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *USENIX Security*, pages 955–972, 2018. 3

[41] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. Spectector: Principled detection of speculative information flows. In *IEEE SP*, pages 1–19, 2020. 4, 11, 12

[42] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games - bringing access-based cache attacks on AES to practice. In *IEEE SP*, pages 490–505, 2011. 3

[43] Shengjian Guo, Yueqi Chen, Peng Li, Yueqiang Cheng, Huibo Wang, Meng Wu, and Zhiqiang Zuo. SpecuSym: speculative symbolic execution for cache timing leak detection. In *ICSE*, pages 1235–1247, 2020. 4

[44] Noam Hadad and Jonathan Afek. Overcoming (some) Spectre browser mitigations. https://alephsecurity.com/2018/06/26/spectre-browser-query-cache/, 2018. Accessed: 2022-01-25. 4

[45] John Hazen. Mitigating speculative execution side-channel attacks in Microsoft Edge and Internet Explorer. https://blogs.windows.com/msedgedev/2018/01/03/speculative-execution-mitigations-microsoft-edge-internet-explorer/, 2018. Accessed: 2022-01-25. 4

[46] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, 5th edition, 2012. ISBN 978-0-12-383872-8. 9

[47] Intel. Retpoline: A branch target injection mitigation. https://www.intel.com/content/dam/develop/external/us/en/documents/retpoline-a-branch-target-injection-mitigation.pdf, 2018. 4

[48] Intel. Intel analysis of speculative execution side channels. https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf, 2018. Accessed: 2022-01-15. 1, 4

[49] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, April 2022. 11

[50] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gülmezoglu, Thomas Eisenbarth, and Berk Sunar. Spoiler: speculative load hazards boost Rowhammer and cache attacks. In *USENIX Security*, pages 621–637, 2019. 3

[51] Ira Ray Jenkins, Prashant Anantharaman, Rebecca Shapiro, J. Peter Brady, Sergey Bratus, and Sean W. Smith. Ghostbusting: Mitigating Spectre with intraprocess memory isolation. In *HotSos*, 2020. 4

[52] Xuancheng Jin, Xuangan Xiao, Songlin Jia, Wang Gao, Hang Zhang, Dawu Gu, Siqi Ma, Zhiyun Qian, and Juanru Li. Annotating, tracking, and protecting cryptographic secrets with CryptoMPK. In *IEEE SP*, 2022. 4

[53] Brian Johannesmeyer, Jakob Koschel, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Kasper: Scanning for generalized transient execution gadgets in the Linux kernel. In *NDSS*, 2022. 4

[54] Angshuman Karmakar, Sujoy Sinha Roy, Oscar Reparaz, Frederik Vercauteren, and Ingrid Verbauwhede. Constant-time discrete Gaussian sampling. *IEEE Trans. Computers*, 67(11):1561–1571, 2018. 3

[55] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael B. Abu-Ghazaleh. SafeSpec: Banishing the Spectre of a Meltdown with leakage-free speculation. In *DAC*, page 60, 2019. 4

[56] Vladimir Kiriansky and Carl A. Waldspurger. Speculative buffer overflows: Attacks and defenses. arXiv/1807.03757, 2018. 3

[57] Vladimir Kiriansky, Ilia A. Lebedev, Saman P. Amarasinghe, Srinivas Devadas, and Joel S. Emer. DAWG: a defense against cache timing attacks in speculative execution processors. In *MICRO*, pages 974–987, 2018. 4

[58] Ofek Kirzner and Adam Morrison. An analysis of speculative type confusion vulnerabilities in the wild. In *USENIX Security*, pages 2399–2416, 2021. 3, 4

[59] Paul Kocher. Spectre mitigations in Microsoft's C/C++ compiler. https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html, 2018. Accessed: 2022-01-25. 1, 4

[60] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE SP*, pages 1–19, 2019. 1, 3, 4

[61] David Kohlbrenner and Hovav Shacham. On the effectiveness of mitigations against floating-point timing channels. In *USENIX Security*, pages 69–81, 2017. 8

[62] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael B. Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *WOOT*, 2018. 3

[63] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. TagBleed: Breaking KASLR on the isolated kernel address space using tagged TLBs. In *IEEE EuroS&P*, pages 309–321, 2020. 3

[64] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security*, pages 973–990, 2018. 1, 3, 4

[65] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *IEEE SP*, pages 605–622, 2015. 3

[66] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography. *ACM Comput. Surv.*, 54(6):122:1–122:37, 2021. 3

[67] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. DOLMA: securing speculation with the principle of transient non-observability. In *USENIX Security*, pages 1397–1414, 2021. 4

[68] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *CCS*, pages 2109–2122, 2018. 3

[69] Andrea Mambretti, Matthias Neugschwandtner, Alessandro Sorniotti, Engin Kirda, William K. Robertson, and Anil Kurmus. Speculator: a tool to analyze speculative execution attacks and mitigations. In *ACSAC*, pages 747–761, 2019. 4

[70] Andrea Mambretti, Pasquale Convertini, Alessandro Sorniotti, Alexandra Sandulescu, Engin Kirda, and Anil Kurmus. GhostBuster: Understanding and overcoming the pitfalls of transient execution vulnerability checkers. In *SANER*, pages 307–317, 2021. 4

[71] Julius Mandelblat. Technology insight: Intel's next generation microarchitecture code name Skylake. In *Intel Developers Forum*, 2015. Available: https://en.wikichip.org/w/images/8/8f/Technology_Insight_Intel%E2%80%99s_Next_Generation_Microarchitecture_Code_Name_Skylake.pdf. 9

[72] Ross McIlroy, Jaroslav Sevcík, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. arXiv/1902.05178, 2019. 4

[73] Avi Mendelson. Secure speculative core. In *SoCC*, pages 426–431, 2019. 4

[74] Alyssa Milburn, Erik van der Kouwe, and Cristiano Giuffrida. Mitigating information leakage vulnerabilities with type-based data isolation. In *IEEE SP*, 2022. https://download.vusec.net/papers/tdi_sp22.pdf. 4

[75] Matt Miller. Analysis and mitigation of speculative store bypass (CVE-2018-3639). https://msrc-blog.microsoft.com/2018/05/21/analysis-and-mitigation-of-speculative-store-bypass-cve-2018-3639/, 2018. Accessed: 2022-01-26. 3

[76] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. MemJam: A false dependency attack against constant-time crypto implementations in SGX. In *CT-RSA*, pages 21–44, 2018. 3

[77] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean M. Tullsen, and Deian Stefan. Swivel: Hardening WebAssembly against Spectre. In *USENIX Security*, pages 1433–1450, 2021. 4

[78] Oleksii Oleksenko, Bohdan Trach, Tobias Reiher, Mark Silberstein, and Christof Fetzer. You shall not bypass: Employing data dependencies to prevent bounds check bypass. arXiv/1805.08506, 2018. 4

[79] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. SpecFuzz: Bringing Spectre-type vulnerabilities to the surface. In *USENIX Security*, pages 1481–1498, 2020. 4

[80] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *CT-RSA*, pages 1–20, 2006. 3, 11

[81] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. Lord of the ring(s): Side channel attacks on the CPU on-chip ring interconnect are practical. In *USENIX Security*, pages 645–662, 2021. 3

[82] Marco Patrignani and Marco Guarnieri. Exorcising Spectres with secure compilers. In *CCS*, pages 445–461, 2021. 2, 4, 5, 6, 11

[83] Colin Percival. Cache missing for fun and profit. In *Proceedings of BSDCan*, 2005. URL https://www.daemonology.net/papers/htt.pdf. 3

[84] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. "make sure DSA signing exponentiations really are constant-time". In *CCS*, pages 1639–1650, 2016. 3

[85] Emmanuel Pescosta, Georg Weissenbacher, and Florian Zuleger. Bounded model checking of speculative non-interference. In *ICCAD*, pages 1–9, 2021. 4

[86] Hernán Ponce de León and Johannes Kinder. Cats vs. Spectre: An axiomatic approach to modeling speculative execution attacks. arXiv/2108.13818, 2021. 4

[87] Chromium Project. Mitigating side-channel attacks. https://www.chromium.org/Home/chromium-security/ssca/, 2018. Accessed: 2022-01-25. 4

[88] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella Béguelin. EverCrypt: A fast, verified, cross-platform cryptographic provider. In *IEEE SP*, pages 983–1002, 2020. 3

[89] Ivan Puddu, Moritz Schneider, Miro Haller, and Srdjan Capkun. Frontal attack: Leaking control-flow in SGX via the CPU frontend. In *USENIX Security*, pages 663–680, 2021. 3

[90] Zhenxiao Qi, Qian Feng, Yueqiang Cheng, Mengjia Yan, Peng Li, Heng Yin, and Tao Wei. SpecTaint: Speculative taint analysis for discovering Spectre gadgets. In *NDSS*, 2021. 4

[91] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In *USENIX Security*, pages 1451–1468, 2021. 1, 3, 9

[92] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. CrossTalk: Speculative data leaks across cores are real. In *IEEE SP*, pages 1852–1867, 2021. 1, 3

[93] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *USENIX Security*, pages 431–446, 2015. 3

[94] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Jordan, Dean M. Tullsen, and Ashish Venkat. I see dead μops: Leaking secrets via Intel/AMD micro-op caches. In *ISCA*, pages 361–374, 2021. 3

[95] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. Dude, is my code constant time? In *DATE*, pages 1697–1702, 2017. 3

[96] Eyal Ronen, Kenneth G. Paterson, and Adi Shamir. Pseudo constant time implementations of TLS are only pseudo secure. In *CCS*, pages 1397–1414, 2018. 3

[97] Eyal Ronen, Robert Gillham, Daniel Genkin, Adi Shamir, David Wong, and Yuval Yarom. The 9 lives of Bleichenbacher's CAT: new cache attacks on TLS implementations. In *IEEE SP*, pages 435–452, 2019. 3

[98] Stephen Röttger and Artur Janc. A Spectre proof-of-concept for a Spectre-proof web. https://security.googleblog.com/2021/03/a-spectre-proof-of-concept-for-spectre.html, 2021. 3, 4, 6

[99] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. Efficient invisible speculative execution through selective delay and value prediction. In *ISCA*, pages 723–735, 2019. 4

[100] Stephan van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Malicious management unit: Why stopping cache attacks in software is harder than you think. In *USENIX Security*, pages 937–954, 2018. 3

[101] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: rogue in-flight data load. In *IEEE SP*, pages 88–105, 2019. 1, 3

[102] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. SGAxe: How SGX fails in practice. https://sgaxeattack.com/, 2020. 3

[103] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. CacheOut: Leaking data on Intel CPUs via cache evictions. In *IEEE SP*, pages 339–354, 2021. 1, 3

[104] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. Jenny: Securing syscalls for PKU-based memory isolation systems. In *USENIX Security*, 2022. 4

[105] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS*, pages 753–768, 2019. 1, 3

[106] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. ConTExT: A generic approach for mitigating Spectre. In *NDSS*, 2020. 4

[107] Martin Schwarzl, Pietro Borrello, Andreas Kogler, Kenton Varda, Thomas Schuster, Daniel Gruss, and Michael Schwarz. Dynamic process isolation. arXiv/2110.04751, 2021. 3

[108] Zhuojia Shen, Jie Zhou, Divya Ojha, and John Criswell. Restricting control flow during speculative execution with Venkman. arXiv/1903.10651, 2019. 4

[109] Julian Stecklina and Thomas Prescher. LazyFP: Leaking FPU register state using microarchitectural side-channels. arXiv/1806.07480, 2018. 1, 3

[110] Robert M Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development*, 11(1):25–33, 1967. 3

[111] Paul Turner. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886, 2018. Accessed: 2022-02-01. 4

[112] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: secure, efficient in-process isolation with protection keys (MPK). In *USENIX Security*, pages 1221–1238, 2019. 4

[113] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *USENIX Security*, pages 991–1008, 2018. 1, 3, 4

[114] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: hijacking transient execution through microarchitectural load value injection. In *IEEE SP*, pages 54–72, 2020. 1, 3

[115] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kici, Ranjit Jhala, Dean M. Tullsen, and Deian Stefan. Automatically eliminating speculative leaks from cryptographic code with Blade. *Proc. ACM Program. Lang.*, 5(POPL):1–30, 2021. 4

[116] Luke Wagner. Mitigations landing for new class of timing attack. https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/, 2018. Accessed: 2022-01-25. 4

[117] Junpeng Wan, Yanxiang Bi, Zhe Zhou, and Zhou Li. MeshUp: Stateless cache side-channel attack on CPU mesh. In *IEEE SP*, 2022. 3

[118] Guanhua Wang, Sudipta Chattopadhyay, Arnab Kumar Biswas, Tulika Mitra, and Abhik Roychoudhury. KLEESpectre: Detecting information leakage through speculative cache attacks via symbolic execution. *ACM Trans. Softw. Eng. Methodol.*, 29(3):14:1–14:31, 2020. 4

[119] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead defense against Spectre attacks via program analysis. *IEEE Trans. Software Eng.*, 47(11): 2504–2519, 2021. 4

[120] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch, and Baris Kasikci. NDA: Preventing speculative execution attacks at their source. In *MICRO*, pages 572–586, 2019. 4

[121] Meng Wu and Chao Wang. Abstract interpretation under speculative execution. In *PLDI*, 2019. 4

[122] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the hyper-space: High-speed covert channel attacks in the cloud. In *USENIX Security*, pages 159–173, 2012. 3

[123] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. InvisiSpec: Making speculative execution invisible in the cache hierarchy. In *MICRO*, pages 428–441, 2018. 4

[124] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher W. Fletcher, Roy H. Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *IEEE SP*, pages 888–904, 2019. 3

[125] Yuval Yarom and Katrina Falkner. Flush+Reload: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, pages 719–732, 2014. 3, 8

[126] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A timing attack on OpenSSL constant time RSA. In *CHES*, pages 346–367, 2016. 3, 11

[127] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. Speculative taint tracking (STT): a comprehensive protection for speculatively accessed data. In *MICRO*, pages 954–968, 2019. 4

[128] Tao Zhang, Kenneth Koltermann, and Dmitry Evtyushkin. Exploring branch predictors for constructing transient execution Trojans. In *ASPLOS*, pages 667–682, 2020. 3, 4, 6

A Semantic Security Proof

This section formalizes our claim that our countermeasure protects against Spectre attacks. For clarity of exposition, we consider a toy high-level language, but out results carry to realistic assembly languages.

The syntax of the programming language is given in Figure 4 where $a \in \mathcal{A}$ ranges over arrays and $x \in \mathcal{X}$ ranges over registers. We let $|a|$ denote the size of a. Moreover, we informally assume that values are either integers or booleans. Informally, the language features assignments (restricted to 3-address mode, with only variable-time operators), conditional assignments, arrays and conditionals and loops.

The operational semantics of the language is modeled in the style of [6, 11, 26] as an indexed transition relation $\langle c, \rho, \mu, b\rangle \xrightarrow[d]{o} \langle c', \rho', \mu', b'\rangle$, where o is an observation taken from the set:

$$o ::= \bullet \mid \mathsf{read}\ a, v \mid \mathsf{write}\ a, v \mid \mathsf{branch}\ b \mid \mathsf{op}\ v\ v$$

d is a directive taken from the set:

$$d ::= \mathsf{step} \mid \mathsf{force} \mid \mathsf{load}\ a, v \mid \mathsf{store}\ a, v$$

and $\langle c, \rho, \mu, b\rangle$ and $\langle c', \rho', \mu', b'\rangle$ are states consisting of a command, memories mapping variables and locations (i.e., pairs of arrays and valid indexes) to values, and b is a speculation flag tracking whether execution has entered an incorrect branch—by convention $b = \top$ if the execution is misspeculating. Figure 6 presents the rules of the speculative semantics. The rules are identical to [6], except for the rule for time-variable arithmetic instructions that leak their operands.

To model USLH, we fix a distinguished register $\tilde{b}$ used to track speculation, and not used anywhere else in the program. The definition of USLH is shown in Figure 5. The key points of the definitions are:

- USLH conditionally masks the guard of if-then-else and while statements, and updates the speculation flag immediately after entering the branch;
- USLH conditionally masks the operands of variable-time instructions;
- USLH conditionally masks the addresses of memory accesses; we assume that the 0-th entry of each array contains a default value that is never modified during execution.

In all cases the masking is done conditionally depending on the value of the speculation flag. The key correctness lemma is that leakage of transformed programs does not depend on the memory.

Lemma 1. *If* $\langle [\![c]\!], \rho, \mu, \top\rangle \xrightarrow[d]{o} \langle c', \rho', \mu', \top\rangle$ *and* $\rho(\tilde{b}) = \top$, *then o only depends on the syntax of c.*

Using this lemma, it is possible to show that transformed programs are relative constant-time (RCT), in the sense that speculative execution of transformed programs does not leak more than their sequential execution.

In order to define the notion of RCT, we define complete executions. This is done by defining the (labeled) reflexive-transitive closure $\langle c, \rho, \mu, b\rangle \overset{O}{\underset{D}{\twoheadrightarrow}} \langle c', \rho', \mu', b'\rangle$ of one-step execution, and $\langle c, \rho, \mu, b\rangle \Downarrow_D^O$ iff $\langle c, \rho, \mu, b\rangle \overset{O}{\underset{D}{\twoheadrightarrow}} \langle c', \rho', \mu', b'\rangle$, with $c' = []$ or $c' = \mathsf{fence}$ and $b = \top$. These two cases correspond to a complete execution or an execution that is interrupted due to misspeculation.

As in prior works, we do not provide a separate semantics for sequential execution. Instead, sequential execution is viewed as a special case where all adversary directives are step and the speculation flag is thus always $\perp$. We write $\langle c, \rho, \mu\rangle \Downarrow^O$ for sequential executions.

Formally, a program c is RCT iff for every executions

$$\begin{gathered}\langle c, \rho_1, \mu_1, \perp\rangle \Downarrow_D^{O_1} \\ \langle c, \rho_2, \mu_2, \perp\rangle \Downarrow_D^{O_2} \\ \langle c, \rho_1, \mu_1\rangle \Downarrow^{O_1^s} \\ \langle c, \rho_2, \mu_2\rangle \Downarrow^{O_2^s}\end{gathered}$$

we have $O_1^s = O_2^s$ implies $O_1 = O_2$.

The informal argument to prove RCT of transformed programs is as follows: first, we prove that the register $\tilde{b}$ introduced by the USLH transformation is always in sync with the speculation flag of the operational semantics, so that $\tilde{b}$ is always set to true when execution enters the wrong branch. Second, every execution can be divided into a sequential (sub-)execution, and a speculative sub-execution, which is triggered by execution entering the wrong branch. Then, thanks to the key lemma above, we know that the leakage of the speculative sub-execution does not depend on the state, and thus does not leak. This means that the leakage of the complete execution is equal to the leakage of the sequential execution plus some constant leakage that is completely determined by the syntax of the program. This suffices to conclude.

$c ::=$	$[]$	empty, do nothing	does not leak
	fence	fence	does not leak
$\mid$	$c;c$	sequence	does not leak
$\mid$	$x := op\ y\ z$	assignment	leaks operands $y,\ z$
$\mid$	$x := \tilde{b}?y : z$	cond. assignment	does not leak
$\mid$	$x := a[y]$	load from array a offset y	leaks offset y
$\mid$	$a[x] := y$	store to array a offset x	leaks offset x
$\mid$	$\mathsf{if}\ t\ \mathsf{then}\ c\ \mathsf{else}\ c$	conditional	leaks guard t
$\mid$	$\mathsf{while}\ t\ \mathsf{do}\ c$	while loop	leaks guard t

Figure 4: Language and informal leakage model

$\llbracket x := op\ y\ z \rrbracket =$	$y := \tilde{b}?0 : y;\ z := \tilde{b}?0 : z;\ x := op\ y\ z$	mask operands conditioned on $\tilde{b}$
$\llbracket a[x] := y \rrbracket =$	$x := \tilde{b} : 0?x;\ a[x] := y$	mask index conditioned on $\tilde{b}$
$\llbracket x := a[y] \rrbracket =$	$y := \tilde{b} : 0?y;\ x := a[y]$	mask index conditioned on $\tilde{b}$
$\llbracket \mathsf{if}\ t\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \rrbracket =$	$t := \tilde{b}?\bot : t;\ \mathsf{if}\ t\ \mathsf{then}\ (\tilde{b} := t?\tilde{b} : \top; \llbracket c_1 \rrbracket)\ \mathsf{else}\ (\tilde{b} := t?\top : \tilde{b}; \llbracket c_2 \rrbracket)$	mask guard conditioned on $\tilde{b}$ and update $\tilde{b}$
$\llbracket \mathsf{while}\ t\ \mathsf{do}\ c \rrbracket =$	$t := \tilde{b}?\bot : t;\ \mathsf{while}\ t\ \mathsf{do}\ (\tilde{b} := t?\tilde{b} : \top; \llbracket c \rrbracket); \tilde{b} := t?\top : \tilde{b}$	mask guard conditioned on $\tilde{b}$ and update $\tilde{b}$
$\llbracket c_1 ; c_2 \rrbracket =$	$\llbracket c_1 \rrbracket ; \llbracket c_2 \rrbracket$	

Figure 5: USLH Countermeasure

$$\frac{\rho' = \rho\{x := [\![op\ y\ z]\!]_\rho\}}{\langle x := op\ y\ z, \rho, \mu, b\rangle \xrightarrow[\mathsf{step}]{op\ \rho(y)\ \rho(z)} \langle [], \rho', \mu, b\rangle}\ [\textsc{Assign}]$$

$$\frac{}{\langle \mathsf{fence}, \rho, \mu, \bot\rangle \xrightarrow[\mathsf{step}]{\bullet} \langle [], \rho, \mu, \bot\rangle}\ [\textsc{Fen}]$$

$$\frac{[\![y]\!]_\rho \in [0, |a|) \qquad \rho' = \rho\{x := \mu[(a, [\![y]\!]_\rho)]\}}{\langle x := a[y], \rho, \mu, b\rangle \xrightarrow[\mathsf{step}]{\mathsf{read}\ a, [\![y]\!]_\rho} \langle [], \rho', \mu, b\rangle}\ [\textsc{Ld}]$$

$$\frac{[\![y]\!]_\rho \notin [0, |a|) \qquad i \in [0, |a'|) \qquad \rho' = \rho\{x := \mu[(a', i)]\}}{\langle x := a[y], \rho, \mu, \top\rangle \xrightarrow[\mathsf{load}\ a', i]{\mathsf{read}\ a, [\![y]\!]_\rho} \langle [], \rho', \mu, \top\rangle}\ [\textsc{Ld-U}]$$

$$\frac{[\![x]\!]_\rho \in [0, |a|) \qquad \mu' = \mu[(a, [\![x]\!]_\rho) := [\![y]\!]_\rho]}{\langle a[x] := y, \rho, \mu, b\rangle \xrightarrow[\mathsf{step}]{\mathsf{write}\ a, [\![x]\!]_\rho} \langle [], \rho, \mu', b\rangle}\ [\textsc{St}]$$

$$\frac{[\![x]\!]_\rho \notin [0, |a|) \qquad i \in [0, |a'|) \qquad \mu' = \mu[(a', i) := [\![y]\!]_\rho]}{\langle a[x] := y, \rho, \mu, \top\rangle \xrightarrow[\mathsf{store}\ a', i]{\mathsf{write}\ a, [\![x]\!]_\rho} \langle [], \rho, \mu', \top\rangle}\ [\textsc{St-U}]$$

$$\frac{\langle c_1, \rho, \mu, b\rangle \xrightarrow[d]{o} \langle c_1', \rho', \mu', b'\rangle}{\langle c_1; c_2, \rho, \mu, b\rangle \xrightarrow[d]{o} \langle c_1'; c_2, \rho', \mu', b'\rangle}\ [\textsc{Seq}]$$

$$\frac{\langle c_1, \rho, \mu, b\rangle \xrightarrow[d]{o} \langle [], \rho', \mu', b'\rangle}{\langle c_1; c_2, \rho, \mu, b\rangle \xrightarrow[d]{o} \langle c_2, \rho', \mu', b'\rangle}\ [\textsc{Seq-Skip}]$$

$$\frac{}{\langle \mathsf{if}\ t\ \mathsf{then}\ c_\top\ \mathsf{else}\ c_\bot, \rho, \mu, b\rangle \xrightarrow[\mathsf{step}]{\mathsf{branch}\ [\![t]\!]_\rho} \langle c_{[\![t]\!]_\rho}, \rho, \mu, b\rangle}\ [\textsc{If}]$$

$$\frac{}{\langle \mathsf{if}\ t\ \mathsf{then}\ c_\top\ \mathsf{else}\ c_\bot, \rho, \mu, b\rangle \xrightarrow[\mathsf{force}]{\mathsf{branch}\ [\![t]\!]_\rho} \langle c_{\neg[\![t]\!]_\rho}, \rho, \mu, \top\rangle}\ [\textsc{If-S}]$$

$$\frac{c_\bot = [] \qquad c_\top = c; \mathsf{while}\ t\ \mathsf{do}\ c}{\langle \mathsf{while}\ t\ \mathsf{do}\ c, \rho, \mu, b\rangle \xrightarrow[\mathsf{step}]{\mathsf{branch}\ [\![t]\!]_\rho} \langle c_{[\![t]\!]_\rho}, \rho, \mu, b\rangle}\ [\textsc{Wh}]$$

$$\frac{c_\bot = [] \qquad c_\top = c; \mathsf{while}\ t\ \mathsf{do}\ c}{\langle \mathsf{while}\ t\ \mathsf{do}\ c, \rho, \mu, b\rangle \xrightarrow[\mathsf{force}]{\mathsf{branch}\ [\![t]\!]_\rho} \langle c_{\neg[\![t]\!]_\rho}, \rho, \mu, \top\rangle}\ [\textsc{Wh-S}]$$

Figure 6: Operational semantics

B SPEC2017 Benchmark

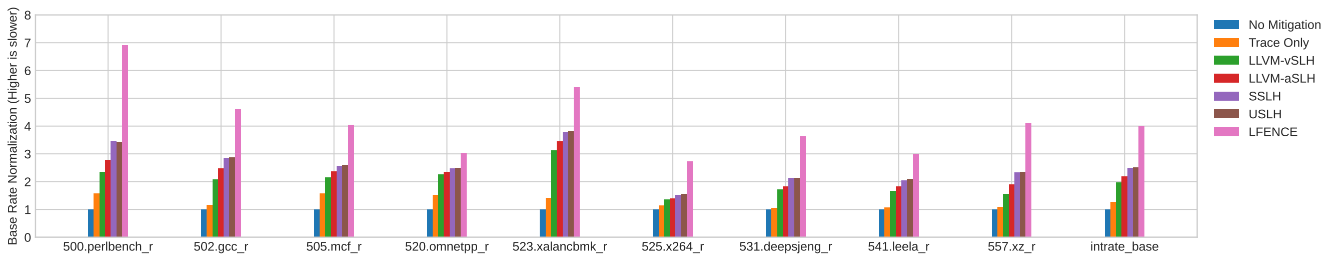

Figure 7: SPEC2017 Int(rate) Benchmark

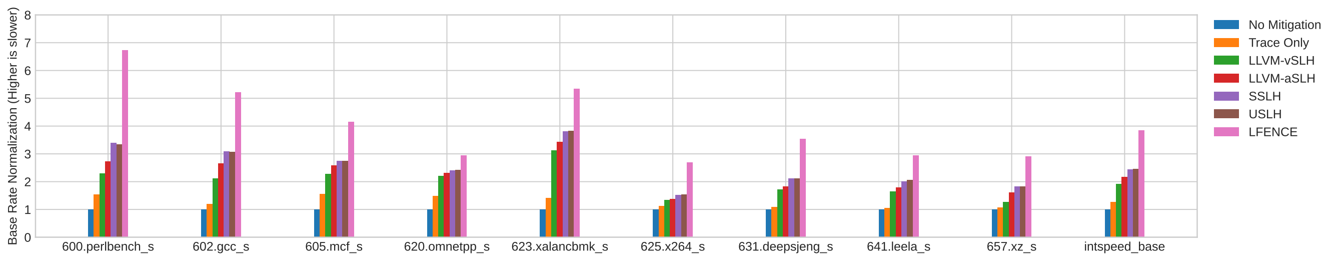

Figure 8: SPEC2017 Int(speed) Benchmark

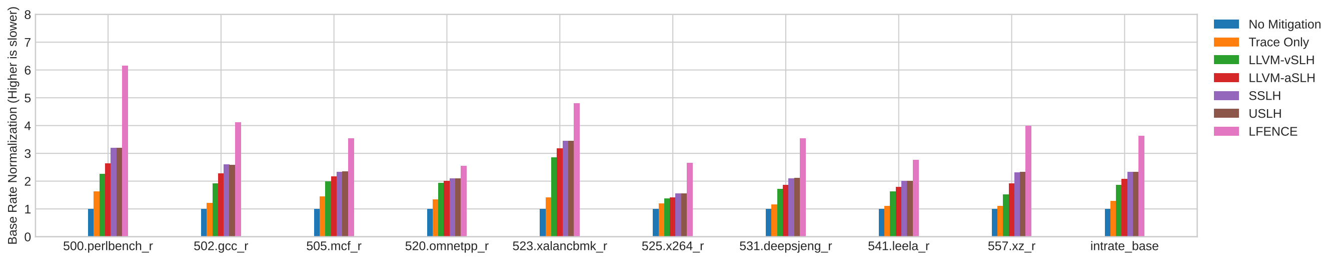

Figure 9: SPEC2017 Int 4-thread Benchmark

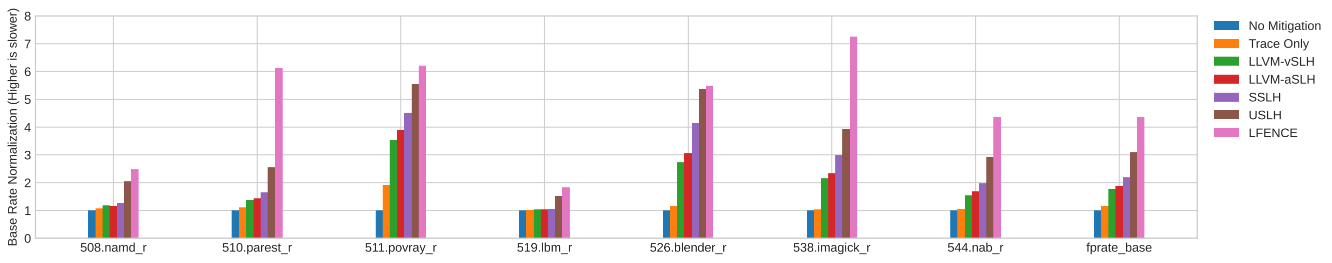

Figure 10: SPEC2017 Floating Point(rate) Benchmark

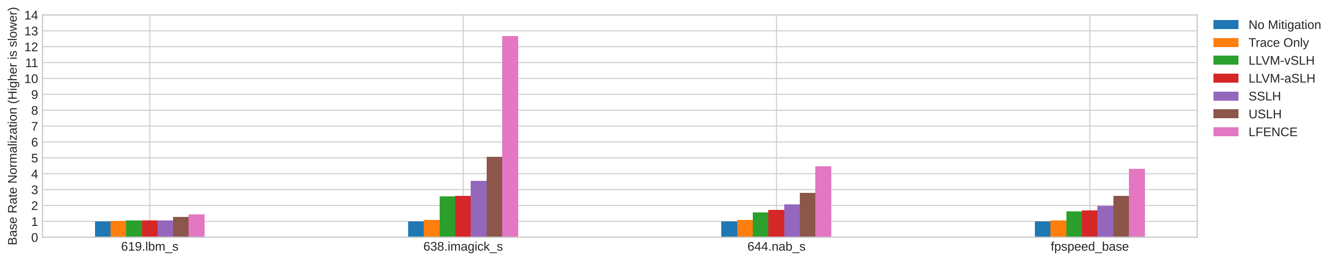

Figure 11: SPEC2017 Floating Point(speed) Benchmark

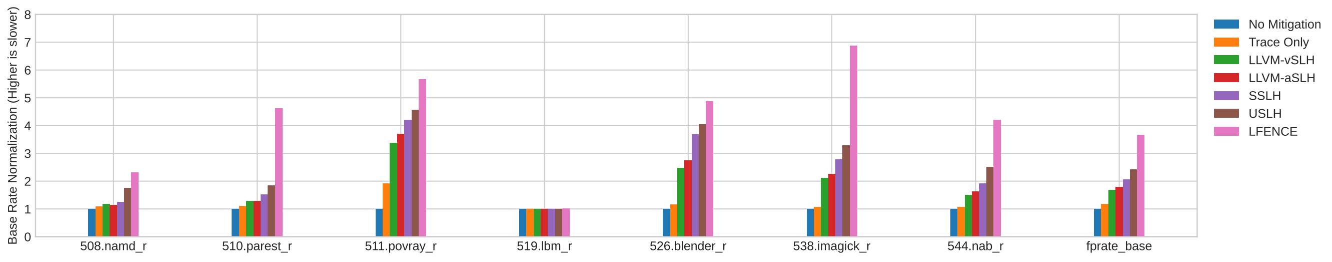

Figure 12: SPEC2017 Floating Point 4-thread Benchmark