Speculative Buffer Overflows: Attacks and Defenses

Vladimir Kiriansky
vlk@csail.mit.edu

Carl Waldspurger
carl@waldspurger.org

Abstract

Practical attacks that exploit speculative execution can leak confidential information via microarchitectural side channels. The recently-demonstrated Spectre attacks leverage speculative loads which circumvent access checks to read memory-resident secrets, transmitting them to an attacker using cache timing or other covert communication channels.

We introduce Spectre1.1, a new Spectre-v1 variant that leverages speculative *stores* to create *speculative buffer overflows*. Much like classic buffer overflows, speculative out-of-bounds stores can modify data and code pointers. Data-value attacks can bypass some Spectre-v1 mitigations, either directly or by redirecting control flow. Control-flow attacks enable arbitrary speculative code execution, which can bypass fence instructions and all other software mitigations for previous speculative-execution attacks. It is easy to construct return-oriented-programming (ROP) gadgets that can be used to build alternative attack payloads.

We also present Spectre1.2: on CPUs that do not enforce read/write protections, speculative stores can overwrite *read-only* data and code pointers to breach sandboxes.

We highlight new risks posed by these vulnerabilities, discuss possible software mitigations, and sketch microarchitectural mechanisms that could serve as hardware defenses. We have not yet evaluated the performance impact of our proposed software and hardware mitigations. We describe the salient vulnerability features and additional hypothetical attack scenarios only to the detail necessary to guide hardware and software vendors in threat analysis and mitigations. We advise users to refer to more user-friendly vendor recommendations for mitigations against speculative buffer overflows or available patches.

1 Introduction

We dub the primary new attack mechanism described in this paper Spectre1.1 (CVE-2018-3693, bounds check bypass on stores), to distinguish it from the original speculative execution attack variant 1 (CVE-2017-5753), which we refer to as Spectre1.0. We consider Spectre1.1 a minor variant in the variant 1 family, since it uses the same opening in the speculative execution window — conditional branch speculation.

1.0 Spectre1.0: Bounds Check Bypass on Loads

Allowing execution past conditional branches is the most important performance optimization employed by speculative out-of-order processors — essentially every modern high-performance CPU. Recently, multiple independent researchers have disclosed ways for attackers to leak sensitive data across trust boundaries by exploiting speculative execution [22, 35, 39]. Using speculative execution, an attacker is able to influence code in the victim's domain to access and transmit a chosen secret [22, 35].

The transmission channel in current proof-of-concept attacks uses microarchitectural cache state — a channel available to speculatively-executed instructions. Cache tag state was a previously-known channel for transmitting information in more limited scenarios — side channels (during execution of cryptographic software operating on a secret [7]), and covert channels (where a cooperating transmitter is used).

The previous Spectre1.0 attack, as well as currently deployed mitigations, target sequences like Listing 1. Since a speculative out-of-order processor may ignore the bounds check on line 1, an attacker-controlled value `x` is not constrained by `lenb`, the length of array `b`. A *secret value* addressable as `b[x]` can therefore be used to influence the index of a dependent load from array `a` into the cache.

In the simplest attack scenario, the attacker also has access to array `a`, and flushes it from the cache before executing the victim code [59]. The speculative attack leaves a footprint in the cache, and the attacker measures each cache line of `a` to determine which one has the lowest access time — inferring the secret value from the address of the fastest line. Generic mitigations against Spectre1.0 and its variants, such as restricting shared memory or reducing timer precision, have been limited to this particular ex-filtration method.

```
1     if (x < lenb)
2       return a[b[x] * 512];
```

Listing 1. Spectre1.0: Bounds Check Bypass (***on Loads***). Speculative secret access via attacker-controlled `x`, and indirect-load transmission gadget using attacker-controlled cache state for array `a`.

```
3     if (y < lenc)
4       c[y] = z;
```

Listing 2. Spectre1.1: Bounds Check Bypass (***on Stores***). Arbitrary speculative write with attacker-controlled `y`, and attacker-controlled or known value `z`.

1.1 Spectre1.1: Bounds Check Bypass on Stores

Code vulnerable to Spectre1.1 is shown in Listing 2. During speculative execution, the processor may ignore the bounds check on line 3. This provides an attacker with the full power of an arbitrary write. While this is only a *speculative* write, which leaves no architecturally-visible effects, it can still lead to information disclosure via side channels.

As a simple proof-of-concept attack, suppose `c[y]` points to the return address on the stack, and `z` contains the address of line 2 in Listing 1. During speculative execution of a function return, execution will be resteered to the transmission gadget, as previously described. Note that even if a fence instruction (e.g., `lfence` or `csdb` [47]) is added between lines 1 and 2 to mitigate against Spectre1.0, an attacker can simply adjust `z` to "jump over the fence". Return-oriented-programming (ROP) techniques can also be used to build alternative attack payloads, as described in Section 5.

In a speculative data attack, an attacker can (temporarily) overwrite data used by a subsequent Spectre1.0 gadget. Performant gadget mitigations use data-dependent truncation (e.g., `x &= (lenb-1)`) rather than fences. An attacker regains arbitrary read access by overwriting either the base of array `b` (line 2), or its length, `lenb` (line 1).

1.2 Spectre1.2: Read-only Protection Bypass

Spectre3.0, aka Meltdown [39], relies on lazy enforcement of User/Supervisor protection flags for page-table entries (PTEs). The same mechanism can also be used to bypass the Read/Write PTE flags. We introduce Spectre1.2, a minor variant of Spectre-v1 which depends on lazy PTE enforcement, similar to Spectre-v3. In a Spectre1.2 attack, speculative stores are allowed to overwrite *read-only* data, code pointers, and code metadata, including vtables, GOT/IAT, and control-flow mitigation metadata. As a result, sandboxing that depends on hardware enforcement of read-only memory is rendered ineffective.

1.3 Current Software Defenses

Currently, no effective static analysis or compiler instrumentation is available to generically detect or mitigate Spectre1.1. Manual mitigations for Spectre1.0 target only very specific cases in trusted code (e.g., in the Linux kernel), where a load is used for further indirect memory accesses.

While generic mitigations for Spectre1.0 have been productized, such as compiler analysis for C [46], they identify only a subset of vulnerable indirect-load code instances. A comprehensive compiler-based mitigation approach using speculative load hardening [9] has been proposed, but incurs a high performance cost. Generic mitigations for Spectre1.0 deployed for JavaScript, as in V8 [18] and Chakra [43], protect only bounds checks for loads.

If we must rely on software mitigations that require developers to manually reason about the necessity of mitigations, we may face decades of speculative-execution attacks. The limited success at educating software developers for the past thirty years since the 1988 public demonstration of classic buffer overflows is a cautionary guide. The silver lining is that the same coding patterns are vulnerable to speculative buffer overflows. A good first step toward preventing them would be to strengthen existing checks against stack overflows, heap overflows, integer overflows, etc.

1.4 Contributions and Organization

We make several key contributions:

- We introduce *speculative buffer overflows* — attacks based on speculative stores that break type and memory safety during speculative execution.
- We analyze salient hardware features to guide possible software and hardware mitigations.
- We present new risks posed by *impossible paths*, *ghosts*, and *halos*, and discuss possible defenses.
- We propose the *SLoth* family of microarchitectural mechanisms to defend against speculative buffer overflow attacks by reducing speculative store-to-load forwarding opportunities for attackers.
- We present a preliminary threat analysis that indicates attackers may be able to mount both local and remote confidentiality, integrity, and availability attacks.

In the next section, we provide relevant hardware and software background related to speculative execution. Section 3 presents a detailed analysis of speculative buffer overflows, including both vulnerability mechanisms and possible software mitigations. We introduce our SLoth family of hardware mitigations in Section 4. Section 5 focuses on threat analysis of payloads leading to remote confidentiality attacks and local integrity attacks. Finally, we summarize our conclusions and highlight opportunities for future work in Section 6.

2 Hardware and Software Background

We first review relevant speculative-execution performance optimizations of modern out-of-order superscalar CPUs in Section 2.1. We then describe the hardware features salient to our minor variants in Section 2.2 for Spectre1.1, and Section 2.3 for Spectre1.2. Section 2.4 discusses further hardware and software features that impact exploitation success.

2.1 Speculative Out-Of-Order Execution

Speculative-execution hardware vulnerabilities are the result of classic computer architecture optimizations from pre-Internet-era design decisions. There are three main optimizations that depend on speculative execution: branch speculation, exception speculation, and address speculation. The currently-disclosed Spectre variants 1 (bounds check bypass) and 2 (branch target injection) use branch speculation, variant 3 (rogue load) uses exception speculation, and variant 4 (speculative store bypass) is one case of address speculation.

Branch speculation takes advantage of temporal and spatial locality in program control flow, and for most programs achieves low branch misprediction rates; high-performance microarchitectures speculate through multiple branches. Exception speculation assumes that most operations, e.g., loads, do not need to trap. Address speculation is used for memory disambiguation, when loads are assumed not to conflict with earlier stores to unknown addresses. Loads are also speculated to hit L1 caches, and immediately-dependent instructions may observe *value speculation* with the value 0 (before mini-replay [22, 60]). The first two speculation types are control speculations, and all subsequent instructions are killed; for the third type, only loads and their dependent instructions need to be replayed.

Attempts to expose all three major speculation mechanisms to software — respectively, via predication, speculative loads, and advanced loads [49] — have been largely unsuccessful. Modern instruction set architectures (ISAs), such as RISC-V [55] and ARMv8 [4], are designed to assume high-performance CPUs will use speculation techniques implemented in out-of-order hardware. As a result, they avoid introducing features such as branch hints and predicated execution, and specify a relaxed memory-ordering model.

2.2 Speculative Store-to-Load Forwarding

The distinctive feature of Spectre1.1 is its dependence on a performance optimization that is usually called *store-to-load forwarding*. A *store buffer* is a microarchitectural structure that tracks stores from instruction issue until they are written back to data caches. On modern cores, such as Intel's Skylake [25], which tracks up to 56 in-flight stores, it serves a quadruple duty.

First, as for in-order cores, the store buffer serves as a write buffer to the L1 cache. Second, on out-of-order cores, speculatively-executed stores are never written back until they retire, i.e., become "senior stores". Third, the store address and the store data are executed out-of-order as separate micro-ops, which is useful when addresses are known much earlier than data. Fourth, a store buffer is used to ensure memory consistency and coherence, i.e., processors observe their own stores and stores from other SMP CPUs. Memory ordering models for most current ISAs specify that a load following a store with a matching address observes the stored value, requiring non-speculative store-to-load forwarding.

Speculative store-to-load forwarding is therefore an optimization that allows a load to execute speculatively using prior store data as soon as both the store address and data are available. The requirements are that the load size is no larger than the store size [25], and the store is the youngest at that address. The load and store physical addresses must be fully matched; address speculation techniques which use virtual addresses [5] or partial physical tags would be subject to (hypothetical) aliasing attacks.

2.3 Data TLB Speculation

Deferring the handling of data TLB page faults until a load commits is an exception-speculation mechanism used to deliver precise exceptions. Spectre3.0 (Meltdown) affects CPUs that do not nullify values on exceptions, e.g., Intel, ARM, and IBM, but not AMD. In our taxonomy, we use Spectre3.1 to refer to Spectre variant 3a, which is a low-priority vulnerability, adding to the long list of known bypasses to Kernel ASLR, which we revisit in Section 3.4.

Fortunately, an effective workaround for Spectre3.0 is to use separate user and supervisor page tables, e.g., kernel page table isolation [13]. Future Intel processors also plan to feature Rogue Data Cache Load (`RDCL_NO`) protection [29]. However, these approaches do not address Spectre1.2.

Spectre1.2: Speculative Store Read-only Overwrite We have validated this attack on both ARM and Intel x86 processors. We hope that a Rogue Data Cache *Store* protection feature can be included in future Intel processors to defend against our Spectre1.2 variant. Ideally, speculative store data should not be forwarded to dependent loads until the TLB entries have been checked to confirm write privileges. Alternatively, only the value 0 should be forwarded on a fault, which is safe as long as partial store-to-load forwarding is not allowed, as noted in Section 2.2.

2.4 Speculative Execution Window

There are two main limits for speculative attack execution — the maximum number of speculative instructions in flight, and the maximum delay of branch resolution (in both cycles and instructions). Current processors support large speculative windows. For example, the re-order buffer (ROB) on Intel's Skylake has space for 224 micro-ops, or about 200 instructions for typical code. Each SMT thread is allotted half, so an attack must complete within roughly 100 instructions.

A DRAM reference on a modern server can take 80–200 ns (60–100 ns on desktops). At a typical clock frequency of 2.5GHz, with the average instructions per cycle for systems code (IPC 1), and typical micro-ops per instruction (UPI 1.1), waiting on one DRAM reference can fill the entire window.

In addition to opening the speculative execution window, an attack is possible only until the window closes — when a branch is resolved and wrong-path instructions are flushed, or when an explicit fence is reached. Even if attackers do not have any influence over branch history, an attack opportunity is presented by sensitive branch mispredictions — when a branch is taken when it should not have been, or when a branch is not taken when it should have been.

Superscalar Execution A modern superscalar core can execute up to 8 speculative micro-ops in a given cycle (and up to 4 instructions can commit non-speculative results). For example, in the same cycle Intel's Skylake can execute up to four arithmetic instructions or up to two branches, as well as two loads and one store. For Spectre1.0, even if a bounds-checking conditional branch is resolved quickly, the few instructions needed for an attack gadget may still execute on a superscalar machine.

Non-blocking Caches Helping cores scale the "memory wall" is the most compelling reason for speculative execution, and modern out-of-order CPUs attempt to uncover independent memory requests. A *non-blocking cache* allows memory requests past predicted branches to be processed while waiting on older instructions.

The state of cache lines with outstanding cache misses is handled in a small number of Miss Status Holding Registers (MSHRs) [36]. For example, Intel's Haswell microarchitecture maintains 10 L1 MSHRs (Line Fill Buffers) for handling outstanding L1 misses [25]. Similarly, on the high-performance ARM A72 processor, 6 L1 MSHRs support up to six unique cache lines targeted by outstanding cache misses [4].

Since speculative memory requests that have missed in the L1 cache are not canceled, initiating a request and placing it in an MSHR within the speculative execution window is sufficient for a load to be cached. An attacker may simply repeat multiple re-executions in order to use values cached after previous attempts. For example, consider `a[b[i]*512]` — a typical ex-filtration gadget that uses an indirect load to form a cache side-channel transmitter. The first attempt ensures the secret value `secret=b[i]` is cached, and subsequent attempts will refer to that value to compute the indirect address and reference `a[secret*512]`.

3 Speculative Buffer Overflows

Speculative buffer overflows allow attackers to execute arbitrary untrusted code within the victim domain. To help explain the hardware mechanisms involved, we dissect our demonstration from C to assembly to RISC micro-ops (in Section 3.1), and discuss longer speculative window requirements (in Section 3.2).

We elaborate on manual mitigations in Section 3.3. Section 3.4 considers classic buffer overflow mitigations, and discusses our proposals for repurposing them to protect against speculative buffer overflows.

3.1 Spectre1.1 Assembly and Micro-ops

We have validated this attack on both ARM and Intel x86 processors[1], but we limit our exposition to x86-64 assembly.

Line 2 of the C code for Listing 3 compiles into the x86-64 assembly code on lines 5 and 6 in Listing 4. When the comparison on line 5 depends on a non-cached data value, the branch on line 6 (in Listing 4 showing the correct path)

```
1  void f(u64 x, u64 y, u64 z) {
2       if (y < lenc)
3         c[y] = z;
4  }
```

Listing 3. Spectre1.1 Vulnerable Function. On 64-bit processors, a 64-bit write must be used to overwrite code-pointers.

```
5       cmp     %rsi, lenc
6       jbe     1f ; taken
7   1:  retq
8       ... caller
```

Listing 4. Spectre1.1: Retired Instructions (x86-64). Correct path after attack.

```
1       cmp     %rsi, lenc
2       jbe     1f ; predicted not taken
3       mov     c, %rax
4       mov     %rdx,(%rax,%rsi,8)
5   1:  retq
6       ... caller
```

Listing 5. Spectre1.1: Speculated Instructions (x86-64). Speculated path before attack; RSB predicts return target correctly.

```
1       cmp     %rsi, lenc ; cache miss
2       jbe     1f         ; unresolved
3       mov     c, %rax    ; cache hit
4       mov     %rdx,(%rax,%rsi,8)
5       ; overwrites (%rsp) in store buffer
6   1:  retq  ; store-to-load forwarding
7       ... ROP gadget
```

Listing 6. Spectre1.1: Speculated Instructions (x86-64). Speculated path during attack; execution resteered.

is slow to resolve, which opens a large speculative-execution window. Listings 5 and 6 show the active speculative paths before and during an attack.

Listing 7 breaks down the last two instructions from Listing 6 into RISC micro-ops. The return instruction `retq` is internally broken into an `LDA` micro-op (line 10, Listing 7) that loads the return address, and an indirect branch to the loaded value (line 11). On some CPUs, `LDA` may additionally execute before the store address is known (line 9). When executed using the data from the speculative store (line 8) after store-to-load forwarding, the `retq` will consider the Return Stack Buffer (RSB) prediction to be incorrect, even though it is normally nearly perfect. Resteered away from the correct caller, the CPU front-end fetches the ROP gadget.

[1]Hypothesized on January 19, 2018; tested on February 11, 2018; reported on February 12, 2018. In addition to Intel and ARM, we also provided proof-of-concept code for both Spectre1.1 and Spectre1.2 to AMD, Google, IBM, and Microsoft.

```
STD %rdx            ; store data: ROP
STA (%rax,%rsi,8) ; store address: %rsp
LDA nip, (%rsp)    ; store-to-load match
JR  nip             ; resteered to ROP
```

Listing 7. Spectre1.1 in plausible RISC μops for x86-64.

3.2 Spectre1.1 Attack Preconditions

The most vulnerable branches depend on the value of previous long-latency operations, such as one or more dependent non-cached memory references, as in `array->length`.

For a Spectre1.1 code-pointer attack, the speculative window must fit not only the payload gadget(s), but also all instructions between the vulnerable conditional branch and the attacked indirect branch, typically a `ret` instruction. Since the `ret` would normally be predicted correctly, the attack *must* speculatively execute this indirect branch using corrupt data, while a prior conditional branch remains unresolved. Increasingly, indirect control transfers on x86 use a `ret`, whether as usual for function return, or for a *retpolined* indirect call/jump, since retpolines are the recommended approach for Spectre2 protection [53].

For a Spectre1.1 data attack, the speculative window must stay open until after a target Spectre1.0 sequence is reached normally. Mitigations against Spectre1.0 that use a speculation barrier (e.g., `lfence`) would be effective against a Spectre1.1 data attack. However, most deployed mitigations employ a more efficient data-dependent sequence, as discussed in Section 3.3.3. In such cases, a data attack can simply overwrite either the array base or length.

3.3 Spectre1.1 Manual Defenses

The software mitigations for preventing an out-of-bounds store for Spectre1.1 are similar to mitigations for Spectre1.0. Manual placement of these mitigations, however, requires analyzing many more potentially-vulnerable locations in order to achieve security with good performance.

3.3.1 Speculation Fences

A fence incurs a high performance penalty from stopping speculative execution, but can be added even in cases where bounds are not known. To ensure that Spectre1.0 loads are ordered after prior branches, CPU vendors have updated documentation for existing fence instructions (e.g., `lfence` on x86), and added new instructions (e.g., `csdb` on ARM). Such fences can be used to implement load-speculation barriers:

```
if (x < lenb) {
   load_barrier_nospec();
   return a[b[x]*512];
}
```

Although the x86 `lfence` instruction was originally defined architecturally as a *load fence*, Intel and AMD have clarified that it serves as a general serializing *instruction fence*. Thus, an `lfence` ensures that no later instruction will execute, even speculatively, until all prior instructions have completed [2, 27]. Other processor vendors should confirm that stores in particular, or simply all instructions, are ordered by existing or new fences, to ensure prior branches are resolved before a store:

```
if (y < lenc) {
   store_barrier_nospec();
   c[y] = z;
}
```

While such fences can be added before potentially-vulnerable stores, there is a high performance cost to unaffected paths:

```
memcpy(void* d, void* s, size_t n) {
  store_barrier_nospec();
  unsafe_memcpy(d, s, n);
}
```

3.3.2 Coarse Masking (Unsafe)

An index value can be bounded coarsely by masking it with the next power of two, as implemented in `asm.js/wasm` by the V8 JavaScript engine [18]. This may be acceptable to prevent reaching secrets with out-of-bounds loads:

```
if (x < b.size) {
   x &= b.mask; // next power of 2
   value = b.start[x];
}
```

However, depending on the layout of data in memory, this approach may fall short of protecting against out-of-bounds stores. Without accompanying changes to pad memory regions to exact powers of two (at the attendant overhead of internal memory fragmentation), vulnerable locations that break type safety may be reachable.

3.3.3 Data-dependent Exact Masking

Whenever the branch and the potentially-vulnerable store are in the same function, the most performant solution is to ensure that indices or pointers are truncated via data-dependent operations. For example, conditional masking for JavaScript loads in V8 emits code equivalent to:

```
1  if (x < b.size) {
2     // unsafe, use assembly!
3     x &= (x < b.size) ? ~0UL : 0;
4     value = b.start[x];
5  }
```

Similar sequences must be used to protect stores as well. The illustrative C code is, however, unsafe. Safe index truncation requires equivalent `asm volatile` assembly sequences. Most ISAs support conditional move instructions that compilers can emit for the ternary select operator. The mask selection (line 3) needs to use a conditional move, but the compiler may convert it to an unsafe branch instead.

However, this depends on the compiler and optimization level, as well as any profiled-guided optimizations.

Compilers may also optimize out "unnecessary" code based on assumptions about correct paths, such as those involving congruent branches or identical code. Unfortunately, speculative execution invalidates such optimizations.

The Linux kernel defines an `array_index_nospec()` macro used to safely mask an index and block speculation [38]. On x86 it succinctly uses the subtract-with-borrow `sbb` assembly instruction (while `sbb` is a vestigial low-throughput instruction, CPU vendors should offer superscalar versions for future silicon). On ARM it also includes the necessary `csdb` fence. We recommend that compiler writers provide a built-in function that safely performs the operation of truncating an index to zero on overflow.

3.3.4 Congruent Branch TOCTOU (Hypothetical)

When a mitigation check and its uses appear in separate basic blocks, placing checks safely becomes more difficult than simply strengthening existing ones. Programmers and compilers typically assume that branches testing the same conditions, as in `if` or `for` statements, behave similarly.

Impossible paths Congruent branch pairs are those where, under correct execution, either both are taken or neither are taken. Usually these are predicted using global branch history to take advantage of correlations. However, this is not guaranteed, and speculative execution may execute not just *wrong paths*, but also *impossible paths*. Invariants about buffer bases or sizes, index bounds, and loop counts will often be invalidated, e.g., due to incorrectly initialized variables in `if/else` branches. To avoid such time-of-check to time-of-use vulnerabilities, all uses must have adjacent guards with one of the two recommended mitigations above.

Ghosts Short loop trip counts (e.g., under 30 [22, 25]) are predicted perfectly by modern path-dependent branch predictors. However, attackers may prime these predictors, as well as the architectural state of stacks or heaps, via prior calls. An impossible path can influence uninitialized variables (`if` statements), and uninitialized or unconstrained values past input vector lengths (`for` loops). We refer to such pseudo-inputs as *ghosts* and *halos*, respectively.

As illustrated in Listing 8, ghosts allow arbitrary speculative reads, writes, and code execution. Ghosts can be avoided by adding a fence (line 6), or when possible, by modifying program logic. Explicit manual initialization with compiler warning assistance or automatic zero-initialization [41] would need to use flow-*insensitive* analyses to avoid optimizing out initialization.

Halos As shown in Listing 9, halos are positions beyond the expected values in array `b` which should have been validated by a gateway function. In this example, the size of array `a`

```
1   A* pa;           // uninitialized
2   if (cond)
3     pa = new A();  // skipped
4   ...
5   if (cond)
6     *pa = b;
```

Listing 8. A *ghost* write (in C++). Attacker controls `pa` (via unchecked stack contents), allowing arbitrary write.

```
1   for (i=0; i < n; i++) {
2      int pos = b[i]; // n <= i < lenb
3      a[pos] = c[i];
4   }
```

Listing 9. A *halo* read. Attacker controls `pos` (by indexing beyond the active entries of `b`), allowing arbitrary write.

may not be available to the worker function, and a fence may be too expensive to add.

Halos can be handled more efficiently by clamping iterator variables. For example, on line 2 we can ensure `i` is less than `n`, but `n` may not be the capacity of array `b`, which may contain unsanitized values; `n` depends on a slow dereference. Clamping `pos` to 0, similar to `array_index_nospec()`, should be safe, as long as this does not break any other invariants, e.g., `capacity(a[b[i]]) > len(c[i])`.

Due to impossible paths and the risk of ghosts and halos, *all* functions should be analyzed for vulnerable patterns, not only gateway functions known to process untrusted inputs.

3.4 Fortified Classic Buffer Overflow Mitigations

Several generic mitigations have been proposed to protect against classic memory-safety bugs. Mitigations against code-pointer attacks can be strengthened to protect against speculative execution attacks as well. However, data attacks remain an important concern for speculative buffer overflows.

Generic Code-Pointer Protections Robust mitigations for code-pointer attacks have been developed, based on program shepherding [32] and follow-on work [1, 33, 37, 52]. Such mitigations, including the subset productized in Microsoft's Control Flow Guard (CFG) [56], can be strengthened to perform all target validation checks without using conditional branches. Similar speculation-safe checks can augment memory integrity checkers [3] to use poisoned write pointers.

Guards implemented by conditional branches can be replaced with guards using arithmetic sequences. Such sequences can conditionally create non-canonical 64-bit virtual addresses that poison indirect transfer or write addresses. For example, by conditionally `XOR`ing either 0 or the MSB bit, an unlikely security violation will result in a general protection fault. Removing never-taken conditional branches from the instruction stream avoids pollution of global branch history and should improve the prediction accuracy of remaining branches. Although more predictable branch-predictor behavior might benefit attackers, this change may be an overall win for both security and performance.

Metadata in CFG is currently protected by read-only PTEs, which are insufficient due to Spectre1.2, and must be strengthened. The CFG reference monitor is indirectly reached via a read-only code pointer, which is an easy attack target. If indirection is desirable, compiled direct calls to a thunk routine that can be patched should be used instead. While speculative stores will succeed at overwriting read-only code, we do not expect their results to be forwarded to instruction fetch. Such store-to-*ifetch* optimization is unlikely, yet x86[2] CPU vendors should document this explicitly.

Hardware mitigations, like Intel's future CET [24, 31], should offer generic protection, provided that security checks are not evadable micro-ops that depend on control speculation, and PTE-based protections for shadow stacks are enforced during speculative execution. Yet, CET is incompatible with *retpolines* [8, 53], since they employ stack smashing as a stronger mitigation for Spectre2. Indirect Branch Restricted Speculation (IBRS) leaves open possibilities for internal interference of indirect branch targets [35].

Return Protections Simple variations of classic stack canary checks [15, 32] can be added to protect speculative return instructions: `XOR`ing into the return address the difference between a stack canary value and its expected value (secret). This mechanism has the advantage of backwards compatibility with current compiler mitigations, preserving the same stack layout and return address for backtraces. Unfortunately, canaries may be elided by compiler optimizations when writes on all architectural paths are "proven safe" [42]; such analysis is invalidated on speculative paths.

Return Stack Buffer (RSB) hardware protections [31] can be repurposed against non-sequential speculative return address overwrites. For example, hardware protections may disallow RSB mispredictions from being resolved speculatively (mini-exceptions), or may prevent speculative store forwarding to `ret` (i.e., forward only from senior stores).

ASLR Address Space Layout Randomization, which has been deployed for user processes, OS kernels, and hypervisors, is the weakest classic buffer overflow mitigation. Nevertheless, it is the only generic mitigation currently available against speculative buffer overflows, and it mitigates both code and data attacks. However, ASLR is rendered ineffective by both classic information leaks, e.g., as used in EternalBlue (CVE-2017-0144) [57], as well as side-channels against branch history [14] or MMU page-table walkers [19].

A small change to the vulnerable statement on line 3 in Listing 3, to `c[y] += z`, allows a relative overwrite of a code-pointer and sidesteps ASLR with Spectre1.1. Additionally, Spectre3.0 and Spectre3.1 (plus additional Spectre3.1.x variants) can be used to bypass Kernel ASLR (KASLR).

Memory Protection Keys In some ISAs, applications may attempt to keep sensitive data accessible only under a *protection key*, such as the Memory Protection Key (MPK) technology recently added to Intel systems [28]. On current hardware, MPK may not be enforcable due to Spectre3, but future processors with hardware mitigations should be able to prevent Spectre1.0 gadgets from accessing secrets.

A Spectre1.1 speculative code-execution gadget, however, can first disable these protections before reading a secret. Significantly more sophisticated solutions are needed to prevent classic buffer overflows [54] from accessing the MPK `wrpkru` instruction, which modifies protection keys. For speculative buffer overflows, however, modifying the architectural behaviour of `wrpkru` to internally include `lfence` should prevent misuse under speculative attacks.

4 Hardware Mitigations

In this section, we sketch plausible hardware mitigations specific to Spectre1.1. We are also designing more general microarchitectural support to protect against both known and unknown variants of Spectre, but this ongoing work is beyond the scope of this paper.

To defend against Spectre1.1, we propose the *SLoth* family of microarchitectural mitigations that constrain store-to-load forwarding. Successive design points have increasing expected performance, but also increasing hardware and software complexity: store-to-load blocking ("SLoth Bear"), lazy store-to-load forwarding ("SLoth"), and frozen store-to-load forwarding ("Arctic SLoth").

4.1 Store-to-Load Blocking

The "SLoth Bear" mitigation anticipates plausible microcode updates for existing silicon to prevent store-to-load forwarding either from speculative stores, or to speculative loads. The viability of implementing this mitigation in microcode is unknown. It affects hardware paths similar to Spectre4, for which existing microcode updates to Intel's production silicon offer a backup plan mitigation (at up to 8% cost on SPECint [26]). If store-to-load blocking is possible with minimum complexity, it would provide maximum security with a minimum trusted computing base (TCB).

This approach is likely to incur high performance overheads, as it may impact operations such as register spills and C++ member variable accesses. Nevertheless, this design point would enable a quick response for unpatched software, while software developers are educated about how to look for vulnerable code. Overall, this mitigation offers a good safety net for users who find its performance acceptable.

[2]Unlike most ISAs, on x86 self-modifying code does not require a barrier.

4.2 Lazy Store-to-Load Forwarding

The "SLoth" mitigation uses compiler-marked instructions that are candidates for forwarding. For example, compilers may allow retpolines to smash the stack, and may mark register spills and restores explicitly.

The low complexity and small TCB of this approach are attractive, with changes localized to the load-store unit. Performance is likely to be acceptable even without software changes; with compiler co-design, it can achieve optimal performance.

4.3 Frozen Store-to-Load Forwarding

If error-prone software mitigations are the only practical alternative solution to this class of speculative execution attacks, a higher-performance hardware design may be justifiable, despite its complexity.

The "Arctic SLoth" mitigation employs dynamic detection of pairs of stores and loads that are candidates for forwarding. A simpler variant can track the load instructions that have previously required store-to-load forwarding on correct paths, while accepting data from any store.

This requires a stronger hardware address speculation mechanism, similar to high-performance Alpha processors [11], which may increase the complexity, power, and area of current CPUs. Full physical address tags for load and store instructions would be required to securely track white-listed pairs of previously-committed instructions.

5 Speculative Attack Payloads

Speculative buffer overflows allow arbitrary speculative code execution within the victim domain. Yet, these are short code fragments, limited to roughly a hundred instructions, and have short-lived ephemeral effects. In this section, we discuss hypothetical payloads that attackers can deploy to escape the weak sandbox of out-of-order execution.

The speculative attacks in this section are based on our hypothetical threat model analysis for SLoth. Our *preliminary* threat analysis indicates that attackers may be able to mount *both local and remote* confidentiality, integrity, and availability attacks. We advise software developers to broaden the scope of vulnerable software analysis, and system builders to design generic defense-in-depth mitigations.

5.1 Threat Model

We assume most systems that process untrusted inputs are at risk from both local and remote attackers. High-value systems that use or maintain secret information (user credentials, private keys, etc.) are the primary concern.

At highest risk are systems that execute untrusted code, including virtual machines, containers, and sandboxed web browser environments. Prior threat analysis of microarchitectural in-filtration and side-channel ex-filtration limited the threat surface to local information disclosure. Remote confidentiality attack targets may also include login, database, and web servers, SSL-terminating firewalls, etc.

We focus our discussion on Spectre1.x, where speculative window in-filtration is possible based solely on untrusted inputs. We assume victims process attacker requests and may respond to them. (Spectre2 also allowed in-filtration into instances that do not communicate with the attacker, a threat only if the attacker and victim share a core.)

5.2 Speculative Attack Ingredients

A speculative attack combines several ingredients:

- *vulnerable code* – reachable by unprivileged attackers.
- *vulnerable data* – untrusted input, used to trigger an out-of-bound access.
- *sensitive data* – known and addressable chosen secrets.
- *speculative payload data* – passed addresses (sensitive data and/or channel parameters).
- *speculative payload code* – present executable gadgets.

An attacker must be able to reach code susceptible to a hardware speculation vulnerability that will not be resolved quickly. The speculative payload parameters and code must also be under attacker control.

Exposed Hardware Vulnerability The vulnerable code may be affected by its use of a speculative read or write. In addition, reads or writes may use addresses that differ from the intended addresses (shadows or aliases):

- *write* – Spectre1.1 (the focus of this paper).
- *read* – an out-of-bound function pointer read [22].
- *shadow* – same virtual address used in different address spaces.
- *alias* – partial physical-address matching, within or across address spaces.

An out-of-bounds or uninitialized read is an instance of Spectre1.0, while an out-of-bounds or uninitialized write is an instance of Spectre1.1. Shadow and alias address speculation vulnerabilities are instances of Spectre4.

We generally assume a Spectre1.1 vulnerability leverages existing code, and may further use ROP gadgets, stack pivoting, etc. For completeness, a plausible Spectre1.0 out-of-bounds function pointer read [22] also allows arbitrary code execution, in addition to being treated simply as a form of out-of-bounds load [35]. Vtables for ghost objects (Section 3.3.4) can be exploited in a similar manner; i.e., a Spectre1.0.1 sub-minor variant in the current taxonomy. SLoth does not protect against Spectre1.0.

5.3 Payload Code – Confidentiality Attacks

There are generally two types of side channels where an unintended shared medium is used for covert or unintended communication. In a *stateful* channel, receivers use footprint timing (Section 5.5); in a *stateless* channel, receivers rely on throughput timing (Section 5.6).

Secret Access The attacker selects a secret bit or byte to transmit using an attacker-controlled parameter, such as `x` in Listing 1. The secret may be accessed via a simple absolute address, such as `b[x]`. Similar attacks may involve executing more flexible code sequences to locate a secret by traversing live application pointers. An attacker may first need to disable any memory protection keys (Section 3.4).

Secret Transmission Depending on the ex-filtration channel, transmission includes any necessary bit or byte extraction and shifting, e.g., `(b[x] & 0x1) << 9`. Data flow from the extracted secret is then directed to an instruction with a generalized data dependence: an address-dependent memory access, a control-dependent instruction selection, or a data-dependent variable-latency operation.

A generalized attack schema can be composed of one or more stages [34], whose output impacts only microarchitectural state. Attacks may also be composed by combining multiple invocations of payload stages, where later stages use microarchitectural state as input, instead of secrets.

5.4 Receiver (Non-Speculative Code)

Receiving a secret is generally within the attacker domain, and not speculative. Timing an access in a sandboxed or virtualized environment, however, may be subject to coarsened timer precision, slowing down a local attack.

Amplifying Timer Precision A software mitigation deployed in web browsers is to reduce timer precision [44]. For example, Chrome coarsened `performance.now()` to use 100 µs granularity, in order to prevent accurate measurement of events at time scales that are orders of magnitude smaller, such as a ~100 ns cache miss to DRAM.

However, this only slows down attacks, without preventing them. Listing 10 illustrates a simple amplification of a timing attack, by requesting multiple cache lines that correspond to each measured secret bit. This approach consumes a larger cache footprint as the amplification factor grows, which may induce evictions. Nevertheless, the huge timing difference between accessing numerous mostly-resident *vs.* non-resident lines still provides a strong signal.

As a small pessimization designed to reduce memory-level parallelism, we also carry a dependence through all accesses — the value of `zero` is always 0, since the contents of array `a` are zero-filled prior to ex-filtration.

```
    zero = 0;
    t0 = performance.now();
    for (i = 0; i < 1024; i++)
        zero += a[i][1+zero];
    t1 = performance.now();
```

Listing 10. Cache Timing Amplification (Hypothetical)

5.5 Footprint Timing Side Channels

A stateful channel allows time-sharing between transmission and reception, e.g., the attacker can observe the footprint after the victim code executes. Although the most commonly demonstrated attack mechanism uses timing of cache-line presence state, various microarchitectural resources can be exploited[17], including:

- Cache memory state – cache lines, cache sets, replacement metadata [34], fill buffers, write-back buffers, prefetchers.
- Branch predictor state – branch target buffer, branch history table, branch history register, RSB.
- Address translation state – TLB, PTE cache.

The easiest and most well-studied side channel uses the particular cache state of individual cache lines. For example, in flush+reload [59] the victim and attacker share a cache line. In prime+probe [40], the attacker no longer needs memory shared with the victim, and instead can use any congruent cache lines to check if any of them has been evicted from a shared LLC cache set. Additional variants include evict+reload [40] and flush+flush [21]. Cache timing attacks have even been demonstrated in JavaScript [45], including the ability to bypass ASLR [19].

All footprint attacks can be prevented by carefully partitioning microarchitectural state. For example, DAWG [34] proposes hardware mitigations that securely partition all microarchitectural memory structures (set-associative caches, TLBs, PTE caches, etc.) to protect against both non-speculative and speculative footprint attacks. A remote cache-timing *reflection attack* is also outlined in [34].

5.6 Throughput Timing Side Channels

Throughput or contention timing requires transmission and reception to be concurrent. Traditional side-channel attacks use contention on shared resources as an indicator. In speculative execution, the victim may additionally interfere with its own instructions that are executed non-speculatively. Examples from classic side-channel attacks measure contention between the victim and attacker for diverse shared resources:

- Cache resources – slice, bank.
- OoO execution resources – execution ports, variable-latency ALUs, banked register file, load buffer, store buffer, reorder buffer, branch order buffer, reservation station, physical register files, free lists, etc.
- System resources – DRAM, QPI.

5.6.1 Reflected Throughput

We generalize this class of channels as measuring victim system throughput after any temporary microarchitectural state is influenced by secret data. An attacker simply measures performance, such as the number of executed macro-operations.

In a traditional SMT attack, the measured thread may be under attacker control. This method requires SMT sharing, and is viable for privilege escalation within a single OS. It is not feasible for cross-VM attacks in most cloud settings, which typically avoid scheduling separate VMs onto hardware threads associated with the same core. However, some cloud providers do offer low-cost burstable "micro" instances that may allow such SMT sharing.

It is also possible to measure the influence of speculative execution on an SMT peer within the attacker domain. This approach is plausible against public cloud instances which share SMT cores only within a trusted domain, and requires detection of how connections are mapped to processing threads.

Finally, the self-interference of the victim thread can simply be measured. This method is the most general, and would be effective for both sandbox escapes or remote attacks, as it requires only connection persistence.

5.6.2 Local and Remote Channel Modulation

Any busy resource can be used as an indicator of speculative execution behavior. Effects that persist even after speculative instruction cancellation are the easiest to measure. We consider several diverse microarchitectural behaviors that hypothetically may be influenced by a speculative attacker payload on some systems.

MSHR Modulation Since speculative memory requests are not canceled, and each core has a limited number of MSHRs (Section 2.4), modulating the memory level parallelism available to non-speculative execution could be measured by its impact on throughput.

Variable-Latency ALU Modulation Some data-dependent instructions, such as square-root and division operations that are not fully pipelined, may exhibit variable latency that can affect peer threads. Moreover, the throughput of non-speculative execution will also be impacted when these instructions are non-cancellable, or if the instruction window does not prioritize the oldest instruction [60].

AVX2 DVFS Modulation Speculative instruction selection over Intel's AVX2 instructions could be used to modulate the reliability and power-saving features of the power control unit. For example, if AVX2 instructions are used they may result in a lower maximum TurboBoost frequency [23].

RDRAND Contention Modulation An attacker could modulate the throughput of Intel's high-quality `rdrand` random number generator, which is used non-speculatively by some SSL implementations for handshakes [16]. This could be prevented by having `rdrand` perform an internal `lfence`.

This random sample from across the instruction manual illustrates that modulation opportunities are pervasive. As seen with defenses against classic buffer overflows, detecting all bad behaviors will be harder than having SLoth prevent attackers from taking over speculative execution.

5.7 Integrity Attacks

In addition to confidentiality attacks, cache eviction can be used for integrity attacks, both indirectly and directly. Practical integrity breaches may simply follow a confidentiality breach in which an attacker steals access credentials.

Hypothetically [10, 50], speculative execution can also be used to mount a RowHammer [30, 48] integrity attack, such as by leveraging indirect load gadgets as a tool for cache evictions [20], or by generalizing network attacks [51]. Since the attacker is operating within the victim domain, non-speculative mitigations [6] are not effective. SLoth prevents execution of such attacks via SPECTRE1.1.

6 Conclusions

We have explored new speculative-execution attacks and defenses, focusing primarily on the use of speculative stores to create speculative buffer overflows, which we refer to as SPECTRE1.1. The ability to perform arbitrary speculative writes presents significant new risks, including arbitrary speculative execution. Unfortunately, this enables both local and remote attacks, even when SPECTRE1.0 gadgets are not present. It also allows attackers to bypass recommended software mitigations for previous speculative-execution attacks.

Speculative execution of wrong or *impossible* paths creates speculative bug class *doppelgängers* to the known classes of pernicious bugs breaking memory and type safety [12, 58]. Given the heightened public awareness due to Spectre and related attacks, there is higher consumer and business acceptance of previously unthinkable performance overheads for security protections. We hope this opportunity will be used to raise the bar for strong generic mitigations against both speculative and classic buffer overflows, as we have outlined for both software and hardware (in Section 3.4).

We also believe SPECTRE1.1 speculative buffer overflows are completely addressable by hardware (in Section 4). Rather than adding to the classic buffer overflow patch burden, future systems should be able to close this attack vector completely, with good performance.

We are confident that future secure hardware and software will be able to retain the performance benefits of speculative-execution processors. We hope to make additional progress in this direction, as we continue to explore more general microarchitectural support and software co-design to protect against both existing and future Spectre variants. In the short term, there may be a few rough patches (to be applied).

Acknowledgments

We are grateful to Joel Emer for his early feedback on this work. Thanks to Matt Miller for his thorough technical review and helpful discussions. Jason Brandt, Martin Dixon, David Kaplan, and Paul Kocher also provided valuable feedback and suggestions.

Thanks to Intel for their partial sponsorship of this research conducted in February 2018.

Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS '05)*. ACM, New York, NY, USA, 340–353. https://doi.org/10.1145/1102120.1102165

[2] Advanced Micro Devices, Inc. 2018. Software Techniques for Managing Speculation on AMD Processors, Revision 1.24.18. (2018). https://developer.amd.com/wp-content/resources/Managing-Speculation-on-AMD-Processors.pdf [Online; accessed 09-June-2018].

[3] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. 2008. Preventing Memory Error Exploits with WIT. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (SP '08)*. IEEE Computer Society, Washington, DC, USA, 263–277. https://doi.org/10.1109/SP.2008.30

[4] ARM. 2015. ARM Cortex-A72 MPCore Processor Technical Reference Manual. (2015).

[5] Darrell Boggs, Aravindh Baktha, Jason Hawkins, Deborah Marr, J. Alan Miller, Patrice Roussel, Ronak Singhal, Bret Toll, and K.S. Venkatraman. 2004. The Microarchitecture of the Intel Pentium 4 Processor on 90nm Technology. *Intel Technology Journal* 8, 1 (2004).

[6] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. CAn't Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 117–130. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/brasser

[7] David Brumley and Dan Boneh. 2005. Remote timing attacks are practical. *Computer Networks* (2005).

[8] Chandler Carruth. 2018. Introduce the "retpoline" x86 mitigation technique for variant #2 of the speculative execution vulnerabilities. http://lists.llvm.org/pipermail/llvm-commits/Week-of-Mon-20180101/513630.html. (January 2018).

[9] Chandler Carruth. 2018. Speculative Load Hardening: A Spectre Variant #1 Mitigation Technique. (2018). https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html

[10] Christopher Celio and Jose Renau. 2018. Securing High-performance RISC-V Processors from Time Speculation. https://riscv.org/2018/05/risc-v-workshop-in-barcelona-proceedings/. (May 2018).

[11] George Z. Chrysos and Joel S. Emer. 1998. Memory Dependence Prediction Using Store Sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA '98)*. Washington, DC, USA, 142–153. https://doi.org/10.1145/279358.279378

[12] Common Weakness Enumeration. 2018. Weaknesses in Software Written in C++. (2018). https://cwe.mitre.org/data/definitions/659.html [Online; accessed 09-June-2018].

[13] Jonathan Corbet. 2017. KAISER: hiding the kernel from user space. https://lwn.net/Articles/738975/. (November 2017).

[14] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking Branch Predictors to Bypass ASLR. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. IEEE Press, Piscataway, NJ, USA, Article 40, 13 pages. http://dl.acm.org/citation.cfm?id=3195638.3195686

[15] Mike Frantzen and Mike Shuey. 2001. StackGhost: Hardware Facilitated Stack Protection. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10 (SSYM'01)*. USENIX Association, Berkeley, CA, USA, Article 5. http://dl.acm.org/citation.cfm?id=1251327.1251332

[16] Alan O. Freier, Philip Karlton, and Paul C. Kocher. 2011. The Secure Sockets Layer (SSL) Protocol Version 3.0. *RFC* 6101 (2011), 1–67. https://doi.org/10.17487/RFC6101

[17] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2018. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering* 8, 1 (2018), 1–27.

[18] Google. 2018. V8 JavaScript Engine. https://v8project.blogspot.com/2018/02/v8-release-65.html. (Feb 2018).

[19] Ben Gras and Kaveh Razavi. 2017. ASLR on the Line: Practical Cache Attacks on the MMU. *NDSS* (2017).

[20] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2016. Rowhammer.js: A remote software-induced fault attack in JavaScript. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 300–321.

[21] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 279–299.

[22] Jann Horn. 2018. Reading privileged memory with a side-channel. https://googleprojectzero.blogspot.com/2018/01/. (January 2018).

[23] Intel. 2014. Intel Xeon Processor E5 v3 Family Uncore Performance Monitoring. https://www-ssl.intel.com/content/www/us/en/processors/xeon/xeon-e5-v3-uncore-performance-monitoring.html. (2014).

[24] Intel. 2017. Control-flow Enforcement Technology Preview. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf. (2017).

[25] Intel. 2017. Intel 64 and IA-32 Architectures Optimization Reference Manual. http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html. (2017). [Online; accessed 11-February-2018].

[26] Intel. 2018. Addressing New Research for Side-Channel Analysis: Details and Mitigation Information for Variant 4. (May 2018). https://newsroom.intel.com/editorials/addressing-new-research-for-side-channel-analysis/ [Online; accessed 09-June-2018].

[27] Intel. 2018. Analyzing Potential Bounds Check Bypass Vulnerabilities. (July 2018). https://software.intel.com/en-us/side-channel-security-support/ [Online; to appear 10-July-2018].

[28] Intel. 2018. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3: System Programming Guide. (May 2018). https://www.intel.com/sdm/

[29] Intel. 2018. Speculative Execution Side Channel Mitigations. Revision 2.0. https://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf. (May 2018).

[30] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ISCA*. IEEE Press.

[31] Vladimir Kiriansky. 2003. *Secure Execution Environment via Program Shepherding*. M.Eng. Thesis. Massachusetts Institute of Technology, Cambridge, MA. http://groups.csail.mit.edu/commit/papers/03/vlk-MEthesis.pdf

[32] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. 2002. Secure Execution via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium*. USENIX Association, Berkeley, CA, USA, 191–206. http://dl.acm.org/citation.cfm?id=647253.720293

[33] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. 2003. *Execution Model Enforcement Via Program Shepherding*. Technical Report MIT/LCS Technical Memo LCS-TM-638. Massachusetts Institute of Technology, Cambridge, MA. http://groups.csail.mit.edu/commit/papers/03/RIO-security-TM-638.pdf

[34] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. 2018. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. Cryptology ePrint Archive, Report 2018/418. (May 2018). https://eprint.iacr.org/2018/418

[35] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *ArXiv e-prints* (Jan. 2018). arXiv:1801.01203

[36] David Kroft. 1981. Lockup-free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of the 8th Annual Symposium on Computer Architecture (ISCA '81)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 81–87. http://dl.acm.org/citation.cfm?id=800052.801868

[37] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-pointer Integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, Berkeley, CA, USA, 147–163. http://dl.acm.org/citation.cfm?id=2685048.2685061

[38] Linux Kernel. 2018. Speculation Documentation. (2018). https://www.kernel.org/doc/Documentation/speculation.txt [Online; accessed 09-June-2018].

[39] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *ArXiv e-prints* (Jan. 2018). arXiv:1801.01207

[40] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *Security and Privacy*. IEEE.

[41] Kangjie Lu, Marie-Therese Walter, David Pfaff, Stefan Nürnberger, Wenke Lee, and Michael Backes. 2017. Unleashing use-before-initialization vulnerabilities in the Linux kernel using targeted stack spraying *(NDSS'17)*.

[42] Microsoft. 2013. Software defense: mitigating stack corruption vulnerabilties. https://blogs.technet.microsoft.com/srd/2013/10/02/software-defense-mitigating-stack-corruption-vulnerabilties/. (2013).

[43] Microsoft. 2018. Add JIT mitigations for Spectre. (February 2018). https://github.com/Microsoft/ChakraCore/commit/08b82b8d33e9b36c0d6628b856f280234c87ba13

[44] Mozilla. 2018. MDN web docs — performance.now(). (2018). https://developer.mozilla.org/en-US/docs/Web/API/Performance/now [Online; accessed 09-June-2018].

[45] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. 2015. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and Their Implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 1406–1418. https://doi.org/10.1145/2810103.2813708

[46] Andrew Pardoe. 2018. Spectre mitigations in MSVC. https://blogs.msdn.microsoft.com/vcblog/2018/01/15/spectre-mitigations-in-msvc/. (January 2018).

[47] Richard Grisenthwaite. 2018. Cache Speculation Side-channels. https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability. (January 2018).

[48] Mark Seaborn and Thomas Dullien. 2015. Exploiting the DRAM RowHammer bug to gain kernel privileges. http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html. (Mar 2015). [Online; accessed 19-February-2018].

[49] Stephen Shankland. 2005. Itanium: A cautionary tale. http://news.cnet.com/Itanium-A-cautionary-tale/2100-1006_3-5984747.html. (Dec 2005). [Online; accessed 15-January-2018].

[50] Stelios Sidiroglou-Douskos. 2018. SpecHammer = Spectre1.1 + RowHammer. Private communication. (February 2018).

[51] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Throwhammer: Rowhammer Attacks over the Network and Defenses. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/atc18/presentation/tatar

[52] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-edge Control-flow Integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Conference on Security Symposium (SEC'14)*. USENIX Association, Berkeley, CA, USA, 941–955. http://dl.acm.org/citation.cfm?id=2671225.2671285

[53] Paul Turner. 2018. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886. (January 2018).

[54] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, and Peter Druschel. 2018. ERIM: Secure and Efficient In-process Isolation with Memory Protection Keys. *CoRR* abs/1801.06822 (2018). arXiv:1801.06822 http://arxiv.org/abs/1801.06822

[55] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović. 2014. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0*. Technical Report UCB/EECS-2014-54. EECS Department, University of California, Berkeley. http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html

[56] David Weston and Matt Miller. 2016. Windows 10 Mitigation Improvements. https://www.blackhat.com/docs/us-16/materials/us-16-Weston-Windows-10-Mitigation-Improvements.pdf. (2016).

[57] Wikipedia. 2018. EternalBlue. (2018). https://en.wikipedia.org/wiki/EternalBlue [Online; accessed 09-June-2018].

[58] Wikipedia. 2018. Memory safety — Types of memory errors. (2018). https://en.wikipedia.org/wiki/Memory_safety#Types_of_memory_errors [Online; accessed 09-June-2018].

[59] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*.

[60] K.C. Yeager. 1996. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro* (1996). https://doi.org/10.1109/40.491460