On the Spectre and Meltdown Processor Security Vulnerabilities

Mark D. Hill[1], Jon Masters[2], Parthasarathy Ranganathan[3], Paul Turner[3], and John Hennessy[3,4]

[1] University of Wisconsin-Madison, [2] Red Hat, [3] Alphabet/Google, [4] Stanford University

For IEEE Micro Special Issue on Hot Chips 2018

January 30, 2019

Abstract

This paper first reviews the Spectre and Meltdown processor security vulnerabilities that were revealed during January-October 2018 and that allow the extraction of protected information from billions of processors in systems large and small. It then discusses short-term mitigation actions and speculates on the longer-term implications to computer software and hardware. The paper expands from a keynote/panel by the authors at IEEE Hot Chips 2018.

Introduction

Security and privacy are more important than ever. Substantial personal, business, and governmental information is stored online. Systems are interconnected across the world. State actors and cybercriminals are using increasingly sophisticated methods. The attack surface is constantly growing due to ever more complex software, complex hardware, executing untrusted downloaded code, and cloud cotenancy.

While many attacks continue to focus on software vulnerabilities, the increased targeting of hardware vulnerabilities--such as Spectre variants [G0, Spec] (including Meltdown [Melt]) described herein--are of great concern because they can be widely available (e.g., across operating systems) and because hardware may take months, or even years, to replace or patch, even after a fix is found. Moreover, Spectre variants cast concern regarding the half-century-old industry definition of computer hardware correctness. Further, strategic industry goals, including accelerators and vector instructions, necessary for scaling performance, are likely to provide new exploitable scope.

As early as IBM System/360 in 1964, computer systems have defined hardware correctness in terms of *timing-independent* functional behavior complying with a particular Instruction Set Architecture specification, ISA or architecture for short [H+P]. Software is written or compiled to the architecture specification, enabling it to run correctly on many different implementations of the architecture. While implementations may differ significantly, they are all capable of executing software written to the ISA specification.

Implementers of an architecture use techniques--called microarchitecture--to create optimized processors for the market, allowing for many trade-offs appropriate to the intended uses. Design

teams may target speed, low cost, low power, silicon area, or some combination thereof [H&P]. Important performance techniques include instruction (or micro-op) level parallelism (ILP) with speculation and substantial use of caches.

To maximize ILP, modern out-of-order processor cores commonly allow for many instructions to be scheduled onto the hardware in parallel; e.g. Intel's Skylake CPUs can issue four instructions every clock and have up to 224 instructions in overlapped execution. However, to satisfy this the CPU is required to predict the outcome of branches so that it can choose a path to speculate against. Correct speculation is made architecturally visible ("retired"), while incorrect speculation is discarded ("aborted"). Spectre variants exploit such speculation, but we conjecture that fixing Spectre by naively eliminating such speculation would make it hard to deliver viable, high-performance products.

Spectre variants are a form of "side-channel" attack [SaSh] in which microarchitectural state, formerly intended to be isolated from retired execution, becomes observable at an architectural level by an attacker program sharing resources with the victim. This state can include secrets loaded into shared architectural state including data referenced via speculation prior to completing access, validity, or bounds checks. The resources through which such secrets are extracted may take the form of a shared cache hierarchy, but also include other shared structures, such as Translation Lookaside Buffers (TLBs). Known microprocessor side channels are at least a dozen years old [WL06].

Side-channels are not a new concept. An early example occurred in a 1970s password cracking attack against TENEX OS software. The OS would perform password validation by comparing a user-supplied string against a stored password. A vulnerability existed because the OS would return a failure as soon as there was a mismatch between the current user supplied character and the stored password. Exploitation relied upon cleverly placing the user-supplied string on a page boundary, and setting an architectural "trap to user" bit on the next page. This allowed the attacker to effectively guess a password, one letter at a time, by measuring page faults.

The US Department of Defense was specifically concerned about side channels in its 1983-85 "Orange Book." [Orange]. Until Spectre variants, many--especially computer architects outside of the computer security and cryptography communities--considered side channels at most a modest concern for the private sector. There are also forms of side-channel attacks, such as TLBleed and BranchScope, that rely on microarchitectural leakage, but are different from Spectre attacks, and usually expose information at significantly lower rates or in a more restricted manner.

Spectre variants rely upon abusing the speculative nature of modern microprocessors. When speculating, a processor dispatches instructions prior to resolving control flow dependencies, such as branches. Exceptions occurring during speculation cannot be handled until we know the instruction is not speculative; this happens at instruction retirement. Some processors may also speculate beyond an exception-causing instruction, such as an access violation, and the Meltdown variants result from this deferred handling of potential exceptions as well as the

associated permission check itself. All widely-used processors implement speculation on top of Out-of-Order execution, which allows a processor to dynamically schedule instructions based upon data dependencies, rather than literal program order. Instructions are allocated into an internal processor structure known as a ReOrder Buffer (ROB) and tracked while in-flight, only "retiring" and becoming architecturally visible in program order. When misspeculation occurs, the ROB is cleared, and the processor restarts on the correct path.

To exploit Spectre, a malicious attacker causes intentional misspeculation of instructions. These instructions, which have been called "transient" [SysTrans], will be aborted prior to retirement and thus do not have an architectural impact. However, they will cause a change to internal processor microarchitectural state that can be observed using a side-channel, such as a shared cache. Cache access times depend upon whether data is present in a particular location, and this can be used to determine data that might have been brought into the cache as a result of speculative execution of instructions.

The typical structure of an attack has three steps:

1. Train some microarchitectural state, e.g.:
    a. Clear 256 blocks from the cache at address X, so that we can sense which block has been moved into the cache.
    b. Prime the branch predictor to predict a given branch "not taken," so that we can coax the processor to execute code that will eventually misspeculate.

2. Save secret in the microarchitecture, e.g., cache state:
    a. Branch-equal r0, r1, elsewhere ; predicted "not taken"
    ; but will actually branch to "elsewhere" this time
    b. Load-byte r2, (r1) ; load secret at "protected address" r1, load will abort
    ; when instruction aborts, protection exception ignored
    c. Load-word r3, (X + r2) ; this instruction also aborts, but it causes a cache fill
    ; the block loaded in cache depends on value of r2
    ; r2 contains the secret from protected address r1!
    ; The value loaded into r3 is never used.

3. Extract secret from the microarchitecture, e.g., cache state:
    a. Time accesses to 256 cache blocks at X; the block that hits (and is faster) corresponds to the value of Memory at address r1!

This basic format is followed by the Spectre variants, and most other microarchitectural attacks described here as well. Each targets a different underlying hardware vulnerability, and the impact varies significantly. Some vulnerabilities can be exploited by interpreted code running within a normally secure "sandbox", some can be exploited across virtual machine boundaries, and some allow attacks from applications on the OS kernel, or against other applications. In some cases, even new extensions with encrypted memory (e.g., within an SGX enclave) may be vulnerable, as these attacks allow the extraction of state from microarchitectural structures

previously believed to be sufficiently isolated to contain plain-text values. Practical application of these attacks often use results from the composition of one or more “gadgets.” Gadgets, a term from return-oriented programming, are code snippets within an existing program that are useful to the attack being performed.

The next sections present the initial Spectre variants v1-v3 announced in January 2018, examples of vulnerable gadgets, and initial mitigation strategies. We then provide a table of variants publically known as of October 2018 and opine on longer-term implications. Unfortunately, no simple, single hardware or software change solves all issues without a significant performance impact.

Spectre Variant 1: Bounds Check Bypass

Spectre-v1 follows the template above to enable user code (e.g., downloaded Javascript) to use speculation and extract protected information elsewhere in the user’s address space (e.g., Chrome browser state) at an incredible rate (e.g., 500 Kbytes/second). Here is simplified C code:

```
if (untrusted_offset < array_length) {
  val = private_memory[untrusted_offset]
  x = accessible_memory[(val & 1)*cache_line_size]
}
```

A conditional expression (`untrusted_offset < array_length`) is the subject of possible speculative execution in which the hardware will speculate that the `untrusted_offset` lies within the array bounds prior to determining (resolving) whether this is indeed the case. Within the speculatively executed block, the offset is used to index into a `private_memory` region which may cause a speculative read beyond the bounds of that memory. A subsequent secondary access touches a predictable memory location having a dependency upon the value of the `private_memory`. An attacker uses cache side-channel analysis to determine the cache line affected by the dependent load and is able to reconstruct the value of a data item in `private_memory`.

The example demonstrates a gadget code that may execute speculatively following a predicted conditional expression. While the example has all attack code in one place, in practice there could be significant code distance between the component parts, making it difficult to find Spectre-v1 gadgets through simple code inspection. Various tooling has been created to aid in automating this process such as “`smatch`”, used by some Linux kernel developers to scan for gadgets in order to aid manual mitigation, while others rely upon automated compiler solutions. Notable is LLVM’s Speculative Load Hardening, which introduces data-dependencies against the control-flow itself; restricting the potential divergence and exploitability of speculative execution.

Spectre-v1 is of great concern because many modern high-performance microprocessors are susceptible to it. These include processors made by Intel, AMD, IBM, Arm (and their licensees), and others. A particular challenge for Spectre-v1--and for many Spectre variants--is that hardware mitigation is difficult because managed languages and browsers have software protection boundaries (e.g., sandboxes for Javascript) that are not revealed to hardware, much less manifest as hardware boundaries. Moreover, Spectre variants currently invalidate the assumption that the language and runtime can protect secrets in the same address space. While it may be possible to add contextual information into future ISAs and extensions to existing architectures, implementing such mitigations in hardware will take some time and perhaps add inherent performance penalties. A number of different approaches have been proposed and are under debate.

Mitigating Spectre-v1 in software requires that we either prevent speculation beyond the bounds check operation, or that we ensure any subsequent array index operation is “clamped” such that under speculation it always falls within the bounds of the array. The process of determining which way the bounds check branch will go is known as “resolving” the branch, and the time taken for this to occur is a function of the time taken to load the desired offset and array limit. Initial mitigations on some architectures focused on forcing the processor to wait until it had resolved the branch by inserting an architecture-specific context serializing instruction following the bounds check. In the case of Intel and AMD x86 architectures, the “lfence” instruction was used for this purpose, while IBM POWER added a new millicoded instruction from their nop encoding space having a similar effect. Other architectures began with the clamping approach. For example, Arm exploited the fact that existing designs do not speculate through a conditional select (CSEL) to form a data dependency. This necessitated a new barrier (CSDB) to future revisions of the architecture in order to avoid future speculation through the CSEL.

While the initial serializing load mitigations were effective, they had a significant performance impact in some workloads. As a result, the software community is generally transitioning to a speculative index clamping solution in which the processor is allowed to speculate beyond a bounds check, but any array access is performed through a macro that will cause the index to be clamped between 0 and the size of the array (e.g., with logical AND). Contemporary processors generally do not yet contain sophisticated enough value predictors to attempt to speculate upon the logical operation.

Spectre Variant 3: Rogue Load from Kernel Space (a.k.a., Meltdown)

We next discuss Spectre-v3 (a.k.a., Meltdown), despite the enumeration, as it closely resembles Spectre-v1. Example V3 C code:

```
if (mispredicted_branch) { // We'd fault if we actually retired below.
  val = kernel_memory[untrusted_offset]
  x = user_memory[(val & 1)*64]
}
```

In the quest for more performance, speculation may occur even past potential exceptions in transient instructions. Spectre-v3 (Meltdown) is an example of this, using the same basic attack structure as Spectre-v1, with the conditional being a user/kernel permission check that may or may not trap. Vulnerable processor designs will identify an exception generating instruction during speculation, tagging its entry in the ReOrder Buffer (ROB). They will then continue to speculate, only handling the potential exception if the instructions actually retire and become the architecturally visible state. Implicit in this design is the incorrect assumption that speculation is a "black box" that cannot be observed by attacker code.

In the above example, an `untrusted_offset` in `kernel_memory` is supplied by the attacker and speculatively loaded into the variable `val`. The processor tags the ROB entry for this load instruction with an exception bit and continues speculation (since the exception is not yet known to be as a result of any retired instruction). It then performs a second load from `user_memory` that depends upon the value of the secret data `val` from `kernel_memory`. This second load has a measurable impact upon the state of the shared cache that can be exploited later by an attacker to determine the value of the secret data. This is where the boundary between microarchitectural and architectural state is broken and exfiltration can occur.

A number of processor designs are vulnerable to one of several Meltdown attacks [SysTrans], including most recent x86-64 implementations from Intel, some aggressive implementations from Apple, Arm (and their licensees), and IBM as well. Meltdown can be avoided if all subsequent dependent loads are always serialized against the permission checks they depend on.

Exploitation in practice on vulnerable designs is typically tied to the L1 cache. Its latency (and access accelerating features such as additional data tagging) is sufficiently low that there is an opportunity for further OOO parallelism when evaluating the permission check above asynchronously. As a consequence, Meltdown can also be avoided in some cases by keeping secrets out of the L1 data cache. This has created a broader interest in secret "scrubbing" in software.

Software mitigation of Meltdown involves one of two possible paths:

1. Preventing vulnerable processors from having valid address translations for privileged (kernel) memory when running in an unprivileged (user) state. This is known as "Page Table Isolation" (PTI). It is the approach used on Intel x86-64 and Arm processors.
2. Preventing the Level 1 data cache from containing secret data that could be loaded by malicious user code. This is achieved through flushing the L1 data cache on return from the OS or Hypervisor into application code. It is the approach used on IBM POWER processors.

Hardware fixes will address the root cause of Meltdown by reworking the handling of permission checks. As an example, Intel announced at Hot Chips 2018 that their upcoming Cascade Lake

[CLX] product would advertise an “RDCL_NO” (Rogue Data Cache Load NO) feature indicating the part is not vulnerable to Meltdown and does not require software to apply mitigations.

Spectre Variant 2: Branch Target Injection

Unfortunately, other Spectre variants exploit microarchitecture features beyond branch and trap prediction used above for Spectre-v1 and Meltdown. Instructions--such as returns and indirect jumps--transfer control flow (“jump”) to one of many different locations with the expected next instruction location predicted with a Branch Target Buffer (BTB). Here is C-like code for Spectre-v2 that exploits branch target (address) (mis-)prediction:

```
4148c0:  object->Foo();
…
9812ab:  val = private_memory[untrusted_large_offset]
           x = accessible_memory[(val & 1)*64]
```

The attacker trains the BTB for the jump at `4148c0` to predict next instruction address to be `9812ab` where the attacker has identified existing code that can serve as a malicious gadget that then speculatively executes the pattern from the introduction: load a secret, use bit(s) from the secret in an address to a second load that perturbs the cache, and end speculation. Due to sharing of functional units, it may even be possible to mistrain or influence the BTB state from a different SMT thread on the same core, or even across host/guest boundaries. This exploitation occurs, in part, because BTB entries are shared instead of isolated via address space and virtual machine identifiers. In many contemporary processors, only a limited number of virtual address bits are used to disambiguate between two different branch addresses, increasing conflict likelihood.

The mitigation of Spectre-v2 involves either limiting the amount of speculation performed using the indirect branch predictor or replacing indirect branches with a safe alternative. The short deadline prior to public disclosure resulted in many initial software updates leveraging newly defined hardware control interfaces added through firmware, microcode, or millicode intended to limit speculation. We examine these hardware approaches first and then explore the software approach, called a “return trampoline” or “retpoline.”

Modern microprocessors are built with some (limited) ability for in-field update, referred to as patching. The precise mechanisms differ from one architecture (and microarchitecture) to the next, but typically involve a combination of “chicken bits” (previously undisclosed configuration knobs used to restrict various capabilities within the processor post-release), microcode extensions (additions that allow certain operations to trigger a “microcode assist” that can modify behavior), and firmware that provides certain platform control interfaces or pre-OS configuration. There can be a significant performance impact for some applications from using microcode or firmware interfaces.

Intel and AMD x86 microarchitectures initially added a new microcoded processor speculation control interface (SPEC_CTRL) through which the Operating System can limit indirect branch speculation (IBRS - Indirect Branch Restrict Speculation), or flush the predictor state (IBPB - Indirect Branch Predictor Barrier). IBRS is typically used across privilege boundaries to prevent indirect branch speculation during system calls into the Operating System, while IBPB is used to invalidate the predictor state when switching from one process into another (with an optimization limiting this to those switches where the target is not already dumpable/debuggable by the first). The SPEC_CTRL interface is implemented as an MSR (Model Specific Register) in microcode, which introduces a performance impact for some applications due to the serializing nature of using MSRs on x86.

Other architectures implement similar interfaces. For example, Arm extends ATF (Arm Trusted Firmware) to perform indirect predictor invalidation (or other appropriate platform-specific mitigation) on kernel entry or process context switch. IBM implement millicode assists for their processors that extend existing nop instructions to achieve similar results. Across all architectures, the presence of Simultaneous Multithreading (SMT) may necessitate additional controls due to the tight coupling of thread resources within a single physical core. Intel x86 processors add STIBP (Single Threaded Indirect Branch Predictors) that can be used by an Operating System to inhibit indirect branch speculation within one thread while another is active on a core.

An alternative approach to mitigating Spectre-v2 in pure software comes from the realization that function returns are merely special case variants of branches. In the case of a return, however, the processor does not (normally) use the indirect branch prediction hardware. In fact, since they are strongly predictable, a return specific predictor such as a Return Stack Buffer (RSB) is often implemented. Thus, it is possible to convert indirect function calls into manufactured returns through careful and deliberate manipulation of the local stack. A traditional x86 indirect branch may look like:

```
jmp *%r11
```

The target of this branch is placed into the processor General Purpose Register `r11`, and then the `jmp` instruction is used to cause a jump to this location. The indirect branch can be converted into an x86 implementation of a retpoline such as this:

```
 call set_up_target;
capture_spec:  // next two instrs never execute, except speculatively
  pause;
  jmp capture_spec;
set_up_target:
  mov %r11, (%rsp);
  ret;
```

In the above code, `set_up_target` overwrites its own return address with the desired location of the indirect branch and then performs a return which causes the function to be executed but does not use the indirect predictor. Immediately following the actual call to `set_up_target` is a harmless code sequence intended to capture any speculated execution into an infinite loop. When present, structures such as the RSB will guarantee speculation is captured by the originally recorded return target (*capture_spec*).

A special exception exists on Intel Skylake and later processors (known as Skylake+). Like many other contemporary processors, these include an RSB (Return Stack Buffer) that attempts to keep track of function calls and returns, predicting the location of a matching call for a return. The RSB shadows the actual function call stack, and it is of a small, finite size. There are many cases in which it can underflow, especially with deep call stacks. Skylake+ processors will begin to speculate from the indirect predictor BTB in the case of an RSB underflow. To prevent this occurrence, it is necessary to also add a benign RSB "stuffing" code sequence to certain code paths within Operating System kernels, such as those transitioning from one privilege level to another. This ensures that the RSB never effectively empties. As a side effect of this, the processor is also mitigated against various attacks that target the RSB, such as SpectreRSB.

Hardware mitigations for Spectre-v2 are expected as well, including in the Intel Cascade Lake release [CLX]. One step forward is for hardware implementers to isolate predictors from different SMT hyperthreads and host/guest via identifiers (such as address space and virtual machine IDs), partitioning, or flushing at a performance cost to be determined.

Spectre Reference Table

Below is a reference table of Spectre variants publically known as of October 2018 with text that is necessarily denser than the rest of this paper. To dig deeper, we also recommend a recently-developed promising taxonomy [SysTrans]. We also call out vulnerable isolation boundaries beyond the local control flow which may be violated by each variant under "scope".

Variant Name and Gist	The gist of Mitigation Strategies
V1 (Bounds Check Bypass). Mistrained conditional branch predictor used to violate program semantics by speculatively accessing data beyond an array limit. **Scope:** user-to-kernel, process-to-process, sandboxes	Either enforce instruction stream serialization with respect to later loads (e.g. through a "lfence" on x86) or introduce dependencies on the retired control flow (LLVM calls this "speculative load hardening"). **Implementation**: software
V1.1 (Bounds Check Bypass Store). Similar to variant 1 but applies to stores, allowing e.g. speculative buffer overflow/stack overflow with re-steering of returns. **Scope**: sandboxes	Careful auditing for potentially risky stores, aided by automated tools (smatch, etc.) or compiler lift (e.g. LLVM speculative load hardening, MSVC). Enforce instruction stream serialization or use clamping. **Implementation**: software
V1.2 (Read-only Protection Bypass). Hardware may implement lazy enforcement of page table protections allowing speculative writes to read-only data. **Scope**: sandboxes	Extension of Bounds Check Bypass Store. Relying on read-only memory protections against e.g function pointer overwrite is not sufficient. It is necessary to protect against potential overwrites into RO memory. **Implementation**: software
V2 (Branch Target Injection). Mistrained indirect branch predictor Branch Target Buffer (BTB) to speculatively execute attacker-controlled "gadgets". **Scope:** user-to-	Limit ability to train the branch predictor and/or to use branch predictor information based on data from different security domains (kernel vs. user, guest vs. host, different tasks) **Implementation**:

kernel, user-to-user, virtual machines, sandboxes	software with perf cost, future hardware
V3 (Rogue Data Cache Load, aka "Meltdown"). User load that speculatively accesses kernel space. See [Melt]. **Scope:** user-to-kernel	Exploitation requires both a valid address translation as well as (typically) data present in the L1 data cache. Either separate address space between privileged and unprivileged execution states, and/or ensure data is not present in the cache and cannot be loaded by an attacker. On some architectures, implement Page Table Isolation (PTI) between user/kernel, on others use an L1D flush. **Implementation**: software with perf cost, future hardware
V3a (Rogue System Register Read). Speculative reads to normally inaccessible system registers may be used to infer information, such as page table base address used to point to all active page tables **Scope:** user-to-kernel, virtual machines	In some cases, updated microcode (etc.) can be used to make such reads serializing and not execute speculatively. In other cases, it may not be possible to prevent certain information leakage - such as the location in memory of page table base address. **Implementation**: hardware change
V4 (Speculative Store Bypass). Speculative reads may proceed prior to determining whether a conflicting store exists in the store buffer (memory disambiguation), **Scope:** sandboxes	Disabling speculative store buffer bypassing (aka "memory disambiguation") either globally, or on a per-application basis, is one mitigation path. Another is aggressive use of process-level isolation (separating contexts of execution), but this is difficult for some cases. Linux eBPF and Java runtimes are examples where a per-process control to disable speculative bypassing of the store buffer is typically employed. **Implementation**: software with perf cost, future hardware
LazyFPU save/restore. Processor implementation may be optimized to avoid saving Floating Point Unit (and vector) context when switching tasks until the new task performs an FPU operation. Vulnerable hardware still allows speculative reads of the disabled FPU state. **Scope:** user-to-user, guest-to-host-process	Disabling lazy save/restore of Floating Point Unit state. In many cases, this actually improves performance on contemporary processors, particularly those which have hardware assisted save/restore FPU instructions. **Implementation**: software now, future hardware
SpectreRSB/ret2spec. Return Stack Buffer manipulated in order to divert the speculative execution of a function return into an attacker-determine leak gadget [arXiv:1807.07940]. **Scope:** sandboxes	RSB "stuffing" is employed to ensure the RSB is filled with a benign delay gadget. This RSB stuffing approach is also used as part of the mitigation for Spectre-v2 on some processors (e.g. Intel Skylake+) wherein an underfill in the RSB causes speculation from the BTB. Thus, it is preferable to reuse the existing mitigation. **Implementation**: software now, future hardware
NetSpectre. Similar to Spectre but performed over a network using a combination of a leak gadget (used to alter microarchitectural state) and transmission gadget (used to transmit this altered state across a network). **Scope:** sandboxes (without explicitly running code!), kernel, remotely exploitable	Mitigation is similar to Spectre-v1, however, the impacted code is potentially very significant. As a result, other solutions at the network layer may be employed, or the impact of leakage may be reduced through careful application of rekeying during transactions. Very sensitive deployments may choose to recompile significant portions of applications using speculative load hardening techniques e.g. as found in LLVM. **Implementation:** software
L1TF (L1 Terminal Fault, "Foreshadow" - SGX). Speculative loads to virtual addresses translated by Page Table Entries (PTEs) with "present" bit not set may result in the processor forwarding the incorrect physical address to the L1 data cache (L1D), allowing reads of attacker-controlled addresses if in the cache. **Scope:** virtual machines	L1TF requires that data be present in the L1 Data cache of impacted Intel processors and that it is possible to construct a vulnerable page table entry. For the "bare metal" use case of an OS on hardware, it is possible to protect against malicious applications by ensuring that all "not present" OS PTEs are masked such that the address is outside of populated physical memory. For virtual machines, it is necessary to employ an L1D cache flush via microcode assist on VM entry. **Implementation:** software now, future hardware
PortSmash Precisely crafted instruction sequences based upon known latencies and contention at the port interfaces to processor execution units can be leveraged to infer the behavior of a sibling SMT thread sharing resources within a core. **Scope:** inference about execution on sibling SMT thread(s)	Currently proposed mitigations involve careful scheduling of application and virtual machine code such that they are not coresident on the same core as potentially malicious code. In some cases, it may be necessary to disable SMT (known as "Hyperthreading" in Intel's implementation) through OS or firmware interfaces provided by a given platform. **Implementation**: software

The Spectre of Spectre

As the table above shows, many Spectre variants have been announced after the original three, including store buffer bypassing and virtual machine address translation. There have also been ongoing discussions of exploiting cache coherence, memory bank conflicts, functional unit timing, and even GPU execution.

A troubling example is NetSpectre [NetS], which can force a victim machine to inadvertently leak information using packets crafted to trigger Spectre-v1 attack(s), *despite not directly executing any code on the victim machine*. NetSpectre currently achieves a very low bandwidth (< 1 bit/minute), but it is a recent existence proof with the significant implication that local user access is not fundamental for enabling microarchitectural attacks.

The real “spectre” of Spectre is that we don’t currently know Spectre limits. It is hard to mitigate known Spectre variants and harder to deal with the unknown variants. Moreover, there is a real danger of “Spectre fatigue” where it will be hard to enthusiastically mobilize for the n^{th} variant.

Where do we go from here?

Of course, security is an end-to-end property with the weakest link determining the overall vulnerability. With Spectre, the hardware becomes a weak link, one that undermines the security of all software. These vulnerabilities will not be solved by hardware and software engineers working in isolation. Software engineers will need to have some notion of how processors behave, an understanding of caches, memory management, speculation, and out-of-order execution. Of course, we cannot expect that software programmers will have deep expertise in these areas, but we must communicate more as we define the contract between software and hardware. No more “us” and “them,” as has sometimes been prevalent in the space of hardware and software engineering; we are in the same boat!

Second, Open Source specifications (ISA) and implementations can help. Security benefits from “many eyeballs,” cleaner design with smaller attack surfaces, and from designs that the security researchers can use to collaborate on solutions that are not specific to one commercial processor vendor. Many security researchers are using RISC-V (https://riscv.org/) already, and it is likely to become the de facto reference used in many future developments. Unfortunately, open architectures and implementations won’t magically solve our security problems as they too can contain unintentional or intentional flaws. Opening up commercial microprocessor microcode/designs--at least to a core of trusted security experts--is extremely difficult but arguably necessary.

Third, our concept of hardware must change. Addressing Meltdown and Spectre-v2 in future hardware is relatively straightforward. Addressing Spectre-v1 and v4 (SSB) may be possible through novel approaches (such as register tagging/tainting).

More generally, Spectre variants have shown the need to move beyond just delivering performance--as markets and program committees have often valued--to more deeply consider security. Of course, a fundamental adjustment in the focus on security AND performance will

require work how to quantify tradeoffs: What percentage of performance overhead is acceptable to eliminate Spectre, and when? When is security just paramount?

In the short and medium term, hardware implementers should seek microarchitectural remedies. Companies must focus first on quickly-implementable fixes/mitigations to known variants. They may wish to consider more “chicken bits” to turn on/off various structures to respond rapidly against currently unknown Spectre variants as they are revealed.

Academics should seek broader solutions in what we expect will be a groundswell of future papers. Ruby Lee in her MICRO 2018 keynote advocated: (a) no (even speculative) access without authorization, (b) no observable micro-architecture effects, and (c) no interference through shared resources unless made indistinguishable through randomization.

Other ideas for moving forward include:

- Better specification of security expectation, e.g., among VMs and with cloud co-tenency.
- Informing hardware of all software boundaries,
- Logically or physically isolating speculative state to make some “poisoning” infeasible,
- Further gating, or even undoing, microarchitectural changes after mis-speculation,
- Interfaces for flushing hardware state to increase the scope of software mitigation,
- Randomizing timing or indexing to reduce side-channel bandwidth, and
- Bifurcating in time (mode) and space (different cores) to support speed and safety.

Moreover, security research benefits from the robust interplay of “attack” and “defend” papers (with coordinated disclosure) whereas, in the past, architecture venues have had too few attack papers and security venues perhaps too many.

Greater use of formal methods will also be required as simulation and testing alone don’t usually withstand a determined adversary against a large, complex attack surface. At best, these methods--e.g., from information flow theory--can safeguard all processors; at a minimum, they may be able to provide a template for slower, “safe”, processors or modes used in the support of aggressive OOO parallelism.

In a larger sense, Spectre variants have exposed a flaw in how we have defined hardware correctness since 1964. The timing-independent functional behavior of a computer--let’s call it Architecture 1.0--is not sufficient to stop information extraction via Spectre-like timing channels, both known and unknown. We have discussed ways to improve microarchitectures vis-a-vis Spectre, but can or will this ever end? Must we manage it like crime, which is essentially how software security flaws are handled?

We challenge the computer science community to develop *Architecture 2.0* wherein all correct-by-Architecture-2.0 implementations must provide both software compatibility (like Architecture 1.0) and prevent Spectre-like timing channel information exfiltration (unlike Architecture 1.0).

While this may be hard--and should begin with a better understanding of threat models and timing channels--it is important for the next decades of our cyber world.

Beyond Spectre attacks

The Spectre and Meltdown attacks just point to one instance of the need to consider security as a first-class design constraint at a system level across hardware and software boundaries. For example, another security vulnerability recently in the news is around managing the root of trust [Titan, Azure] opens many questions: How do we know our computing equipment is not being spoofed? How can we trust the boot chain on our devices? How should we pre-emptively design to protect against not-yet-known vulnerabilities? Another opportunity is around building secure, performant enclaves. More generally, as evidenced by the DARPA SSITH program (www.darpa.mil/program/system-security-integration-through-hardware-and-firmware), our relatively young field needs to develop frameworks and abstractions to categorize vulnerabilities and systematically develop hardware security architectures and associated design tools to protect our future computing systems. Perhaps this can be done by developing/using formalisms that anchor other fields successfully.

Acknowledgements

We thank the numerous professionals who commented on this paper. We regret that the short paper format precluded more elaborate discussion on some of the nuances. Hill was supported by Google during his sabbatical and at Wisconsin by NSF CCF-1617824, NSF CNS-1815656, and John P. Morgridge Endowed Chair.

References

[G0] J. Horn, “Reading privileged memory with a side-channel,” Project Zero, vol. 3, 2018. URL: https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html

[Spec] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre attacks: Exploiting speculative execution,” arXiv preprint arXiv:1801.01203, 2018.

[Melt] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, et al., “Meltdown: Reading kernel memory from user space,” in 27th USENIX Security Symposium (USENIX Security 18), USENIX Association, 2018.

[H+P] J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach. Elsevier, 2019.

[SaSh] J. H. Saltzer and M. D. Schroeder, “The protection of information in computer systems,” Proceedings of the IEEE, vol. 63, no. 9, pp. 1278– 1308, 1975.

[WL06] Z. Wang and R. B. Lee, “Covert and Side Channels due to Processor Architecture,” Annual Computer Security Applications Conference (ACSAC), 2006.

[Orange] U.S. Department of Defense, “Trusted Computer System Evaluation Criteria,” tech. rep., U.S. Department of Defense publication 5200.28- STD, 1985. A.k.a, “Orange Book.”

[SysTrans] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, “A systematic evaluation of transient execution attacks and defenses,” arXiv preprint arXiv:1811.05441v1, 2018.

[CLX] S. Kottapalli and A. Kumar, “Next generation intel xeon(r) scalable processor: Cascade lake,” HotChips, 2018.

[NetS] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss, “Netspectre: Read arbitrary memory over network,” arXiv preprint arXiv:1807.10535, 2018.

[Titan] D. Rizzo, S. Johnson, J. McCune, R. Ho, and P. Ranganathan, “Titan: Google’s root-of-trust security,” IEEE Hot Chips, 2018.

[Azure] D. Stiles, “The hardware security platform behind azure sphere,” IEEE Hot Chips, 2018.

Mark D. Hill is John P. Morgridge Professor and Gene M. Amdahl Professor of Computer Sciences at the University of Wisconsin-Madison. His research interests include parallel-computer system design, memory system design, and computer simulation. Hill has a PhD in computer science from the University of California, Berkeley. He is a fellow of IEEE and the ACM, as well as Chair of the Computer Community Consortium. Email: markhill@cs.wisc.edu

Jon Masters is a Computer Architect at Red Hat, where he was tech lead for mitigation efforts against Meltdown and Spectre. Jon has worked closely with high performance microprocessor design teams for years on emerging alternative server platforms, and also currently leads the CCIX software working group helping to define high performance cache coherent interconnects for workload acceleration. Jon has been a Linux developer for 22 years, since beginning college at the age of 13, and has authored a number of books on Linux technology. He lives in Cambridge, MA, and enjoys running marathons and hiking in his spare time.

Parthasarathy Ranganathan is a distinguished engineer at Google where he is the area tech lead for platforms hardware and datacenters. Prior to this, he was a HP Fellow and Chief Technologist at Hewlett Packard Labs. Partha's research interests include systems, architecture, and energy efficiency. He has a PhD in computer engineering from Rice University. He is a fellow of IEEE and the ACM. Email: partha.ranganathan@google.com

Paul Turner is a principal engineer at Google where he is the technical lead for CPU scheduling and security. Paul's research interests include systems, concurrency, architecture, virtual machines, and security. He has a Bachelors in Pure Mathematics and Computer Science from the University of Waterloo. Email: pjt@google.com

John L. Hennessy is a Professor of Electrical Engineering and Computer Science at Stanford University and the Director of Knight-Hennessy Scholars, a graduate level scholarship program for future world leaders. He also is Chairman of the Board of Alphabet (the parent of Google). Formerly the tenth President of Stanford University, he also co-founded MIPS Computer Systems and Atheros

Communications. He was awarded the 2012 IEEE Medal of Honor and the 2017 ACM A.M. Turing Prize (jointly with David Patterson).