DOI:10.1145/3399742

Spectre Attacks: Exploiting Speculative Execution

By Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom

Abstract

Modern processors use branch prediction and speculative execution to maximize performance. For example, if the destination of a branch depends on a memory value that is in the process of being read, CPUs will try to guess the destination and attempt to execute ahead. When the memory value finally arrives, the CPU either discards or commits the speculative computation. Speculative logic is unfaithful in how it executes, can access the victim's memory and registers, and can perform operations with measurable side effects.

Spectre attacks involve inducing a victim to speculatively perform operations that would not occur during correct program execution and which leak the victim's confidential information via a side channel to the adversary. This paper describes practical attacks that combine methodology from side-channel attacks, fault attacks, and return-oriented programming that can read arbitrary memory from the victim's process. More broadly, the paper shows that speculative execution implementations violate the security assumptions underpinning numerous software security mechanisms, such as operating system process separation, containerization, just-in-time (JIT) compilation, and countermeasures to cache timing and side-channel attacks. These attacks represent a serious threat to actual systems because vulnerable speculative execution capabilities are found in microprocessors from Intel, AMD, and ARM that are used in billions of devices.

Although makeshift processor-specific countermeasures are possible in some cases, sound solutions will require fixes to processor designs as well as updates to instruction set architectures (ISAs) to give hardware architects and software developers a common understanding as to what computation state CPU implementations are (and are not) permitted to leak.

1. INTRODUCTION

Computations performed by physical devices often leave observable side effects beyond the computation's nominal outputs. Side-channel attacks focus on exploiting these side effects to extract otherwise-unavailable secret information. Since their introduction in the late 90s,[14] various physical effects such as power consumption have been leveraged to extract cryptographic keys as well as other secrets.[13]

External side-channel measurements can be used to extract secret information from complex devices such as PCs and mobile phones. However, because these devices often execute code from a potentially unknown origin, they face additional threats in the form of software-based attacks, which do not require external measurement equipment. Although some attacks exploit software logic errors, other software attacks leverage hardware properties to infer sensitive information. Attacks of the latter type include microarchitectural attacks exploiting cache timing[3, 6, 17] and branch prediction history.[1] Software-based techniques have also been used to induce computation errors, such as fault attacks that alter physical memory[11] or internal CPU values.[25]

Several microarchitectural design techniques have facilitated the increase in processor speed over the past decades. One such advancement is speculative execution, which is widely used to increase performance and involves having the CPU guess likely future execution directions and prematurely execute instructions on these paths. More specifically, consider an example where the program's control flow depends on an uncached value located in external physical memory. As this memory is much slower than the CPU, it often takes several hundred clock cycles before the value becomes known. Rather than wasting these cycles by idling, the CPU attempts to guess the direction of control flow, saves a checkpoint of its register state, and proceeds to speculatively execute the program on the guessed path. When the value eventually arrives from memory, the CPU checks the correctness of its initial guess. If the guess was wrong, the CPU discards the incorrect speculative execution by reverting the register state back to the stored checkpoint, resulting in performance comparable to idling. However, if the guess was correct, the speculative execution results are committed, yielding a significant performance gain as useful work was accomplished during the delay.

From a security perspective, speculative execution involves executing a program in possibly incorrect ways. However, because CPUs are designed to maintain functional correctness by reverting the results of incorrect speculative executions to their prior states, these errors were previously assumed to be safe.

In this paper, we analyze the security implications of such incorrect speculative execution. We present a class of microarchitectural attacks which we call *Spectre attacks*. At a high level, Spectre attacks trick the processor into speculatively executing instruction sequences that should not have been executed under correct program execution. As the effects of these instructions on the nominal CPU state are eventually reverted, we call them *transient instructions*. Transient

The original version of this paper appeared in *Proceedings of the 40th IEEE Symposium on Security and Privacy* (May 2019).

instructions can, however, have observable effects that convey information. By influencing which transient instructions are speculatively executed, we are able to leak information from within the victim's memory address space.

Spectre attacks can be applied to leak information across a broad range of security domains. In this paper, we describe several implementations and variations, such as attacks that extract information from other processes and from kernel memory, and that violate sandboxes enforced by programming languages.

At a high level, Spectre attacks violate memory isolation boundaries by combining speculative execution with data exfiltration via microarchitectural covert channels. More specifically, to mount a Spectre attack, an attacker starts by locating or introducing a sequence of instructions within the process address space which, when executed, acts as a covert channel transmitter that leaks the victim's memory or register contents. The attacker then tricks the CPU into speculatively and erroneously executing this instruction sequence, thereby leaking the victim's information over the covert channel. Finally, the attacker retrieves the victim's information over the covert channel. Although the changes to the nominal CPU state resulting from this erroneous speculative execution are eventually reverted, previously leaked information or changes to other microarchitectural states of the CPU, for example, cache contents, can survive nominal state reversion.

The above description of Spectre attacks is general and needs to be concretely instantiated with a way to induce erroneous speculative execution as well as with a microarchitectural covert channel. Although many choices are possible for the covert channel component, the implementations described in this work use cache-based covert channels,[24] that is, Flush+Reload[29] and Evict+Reload.[5, 15]

The underlying vulnerability arises from the composition of widely used microarchitectural features, rather than an implementation error in a single component. We have verified the vulnerability in all processors tested that implement speculative execution, such as multiple designs from Intel, AMD, and ARM. This contrasts with a related issue, Meltdown,[16] which exploits a vulnerability specific to many Intel and a few ARM processors, which allows user-mode instructions to infer the contents of kernel memory.

Following the practice of responsible disclosure, we participated in an embargo of the results. This process was unusually complex due to the large number of stakeholders and affected products.

2. BACKGROUND

In this section, we introduce some of the microarchitectural components of modern high-speed processors as well as several attack techniques.

2.1. Speculative execution

Often, the processor does not know the future instruction stream of a program. For example, this occurs when out-of-order execution reaches a conditional branch instruction whose direction depends on preceding instructions whose execution is not completed yet. In such cases, the processor can preserve its current register state, make a prediction as to the path that the program will follow, and *speculatively* execute instructions along the path. If the prediction turns out to be correct, the results of the speculative execution are committed (i.e., saved), yielding a performance advantage over idling during the wait. Otherwise, when the processor determines that it followed the wrong path, it abandons the work it performed speculatively by reverting its register state and resuming along the correct path.

We refer to instructions which are performed erroneously (i.e., as the result of a misprediction), but may leave microarchitectural traces, as *transient instructions*. Although the speculative execution maintains the architectural state of the program as if execution followed the correct path, microarchitectural elements may be in a different (but valid) state than before the transient execution.

Speculative execution on modern CPUs can run several hundred instructions ahead.

2.2. Branch prediction

During speculative execution, the processor makes guesses as to the likely outcome of branch instructions. Better predictions improve performance by increasing the number of speculatively executed operations that can be successfully committed.

Branch predictors of modern processors can have multiple prediction mechanisms for direct and indirect branches. Indirect branch instructions can jump to arbitrary target addresses computed at runtime, such as instructions that jump to an address in a register, memory location, or on the stack (e.g., "`jmp eax`" on x86). Return instructions are a type of indirect branch, and modern CPUs often include additional mechanisms for predicting return addresses.

For conditional branches, recording the target address is not necessary for predicting the outcome of the branch, because the destination is typically encoded in the instruction although the condition is determined at runtime. To improve predictions, the processor maintains a record of branch outcomes, both for recent direct and indirect branches.

2.3. The memory hierarchy

To bridge the speed gap between the faster processor and the slower memory, processors use a hierarchy of successively smaller but faster caches. The caches divide the memory into fixed size chunks called *lines*, with typical line sizes being 64 or 128 bytes. When the processor needs data from memory, it first checks if the *L1* cache contains a copy. In the case of a *cache hit*, that is, the data is found in the cache, the data is retrieved from the L1 cache and used. Otherwise, in the case of a *cache miss*, the procedure is repeated to attempt to retrieve the data from the next cache levels, and finally the external memory. Once a read is completed, the data is typically stored in the cache (and a previously cached value is evicted to make room) in case it is needed again in the near future.

2.4. Microarchitectural side-channel attacks

The microarchitectural components discussed above improve the processor performance by predicting future program behavior. To that aim, they maintain state that depends on past program behavior and assume that future behavior is similar to or related to past behavior.

When multiple programs execute on the same hardware, either concurrently or via time-sharing, changes in the microarchitectural state caused by the behavior of one program may affect other programs. This, in turn, may result in unintended information leaks from one program to another.

Initial microarchitectural side-channel attacks exploited timing variability[14] and leakage through the L1 data cache[27] to extract keys from cryptographic primitives. Over the years, channels have been demonstrated over multiple microarchitectural components, such as lower level caches[10, 17] and branch history.[1]

In this work, we use the Flush+Reload technique,[6, 29] and its variant Evict+Reload.[5] Using these techniques, the attacker begins by evicting a cache line from the cache that is shared with the victim. After the victim executes for a while, the attacker measures the time it takes to perform a memory read at the address corresponding to the evicted cache line. If the victim accessed the monitored cache line, the data will be in the cache, and the access will be fast. Otherwise, if the victim has not accessed the line, the read will be slow. Hence, by measuring the access time, the attacker learns whether the victim accessed the monitored cache line between the eviction and probing steps.

The main difference between the two techniques is the mechanism used for evicting the monitored cache line from the cache. In the Flush+Reload technique, the attacker uses a dedicated machine instruction, for example, x86's `clflush`, to evict the line. Using Evict+Reload, eviction is achieved by forcing contention on the cache set that stores the line. Due to the limited size of the cache, reading several other memory locations that map to the same cache set can cause the processor to discard (evict) the desired line.

3. ATTACK OVERVIEW

Spectre attacks induce a victim to speculatively perform operations that would not occur during strictly serialized in-order processing of the program's instructions, and that leak victim's confidential information via a covert channel to the adversary.

In most cases, the attack begins with a setup phase, where the adversary performs operations that mistrain the processor so that it will later make an exploitably erroneous speculative prediction. In addition, the setup phase may include steps that help induce speculative execution, such as manipulating the cache state to remove data that the processor will need to determine the actual control flow. During the setup phase, the adversary can also prepare the covert channel that will be used for extracting the victim's information, for example, by performing the flush or evict part of a Flush+Reload or Evict+Reload attack.

During the second phase, the processor speculatively executes instruction(s) that transfer confidential information from the victim context into a microarchitectural covert channel. This may be triggered by having the attacker request that the victim performs an action, for example, via an API call. In other cases, the attacker may leverage the speculative (mis-)execution of its own code to obtain sensitive information from the same process. For example, attack code, which is sandboxed by an interpreter, a just-in-time compiler, or a "safe" language, may wish to read memory it is not supposed to access. Although speculative execution can potentially expose sensitive data via a broad range of covert channels, the examples given cause speculative execution to first read a memory value at an attacker-chosen address and then perform a memory operation that modifies the cache state in a way that exposes the value.

For the final phase, the sensitive data is recovered. For Spectre attacks using Flush+Reload or Evict+Reload, the recovery process consists of timing the access to memory addresses in the cache lines being monitored.

Spectre attacks only assume that speculatively executed instructions can read from memory that the victim process could access normally, for example, without triggering a page fault or exception. Hence, Spectre is orthogonal to Meltdown,[16] which exploits scenarios where some CPUs allow out-of-order execution of user instructions to read kernel memory. Consequently, even if a processor prevents speculative execution of instructions in user processes from accessing kernel memory, Spectre attacks still work.

4. VARIANT 1: EXPLOITING CONDITIONAL BRANCH MISPREDICTION

In this section, we demonstrate how conditional branch misprediction can be exploited by an attacker to read arbitrary memory from another context, for example, another process.

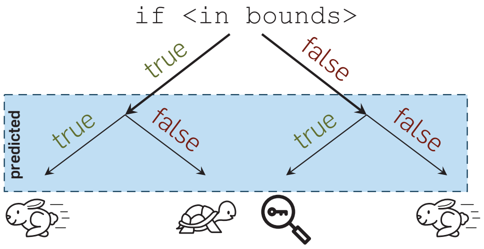
Consider the case where the code here is part of a function (e.g., a system call or a library) receiving an unsigned integer `x` from an untrusted source. The process running the code has access to an array of unsigned bytes: `array1` of size `array1_size`, and a second byte array `array2` of size 1 MB.

```
if (x < array1_size)
   y = array2[array1[x] * 4096];
```

The code fragment begins with a bounds check on `x`. This check is essential for security because it prevents the processor from reading sensitive memory outside of `array1`. Otherwise, an out-of-bounds input `x` could trigger an exception or could cause the processor to access sensitive memory by supplying `x` = (address of a secret byte to read) – (base address of `array1`).

Figure 1 illustrates the four cases of the bounds check in combination with speculative execution. Before the result of the bounds check is known, the CPU speculatively executes code following the condition by predicting the most likely outcome of the comparison. There are many reasons why the result of a bounds check may not be immediately known, for example, a cache miss preceding or during the bounds check, congestion of a required execution unit, complex arithmetic dependencies, or nested speculative execution.

Figure 1. Before the correct outcome of the bounds check is known, the branch predictor continues with the most likely branch target, leading to an overall execution speed-up if the outcome was correctly predicted. However, if the bounds check is incorrectly predicted as true, an attacker can leak secret information in certain scenarios.



However, as illustrated, a correct prediction of the condition in these cases leads to faster overall execution.

Unfortunately, during speculative execution, the conditional branch for the bounds check can follow the incorrect path. In this example, suppose an adversary causes the code to run such that:

- the value of `x` is maliciously chosen (out-of-bounds), such that `array1[x]` resolves to a secret byte k somewhere in the victim's memory;
- `array1_size` and `array2` are uncached, but k is cached; and
- previous operations received values of `x` that were valid, leading the branch predictor to assume the `if` will likely be true.

This cache configuration can occur naturally or can be created by an adversary, for example, by causing eviction of `array1_size` and `array2` and then having the kernel use the secret key in a legitimate operation.

When the compiled code above runs, the processor begins by comparing the malicious value of `x` against `array1_size`. Reading `array1_size` results in a cache miss, and the processor faces a substantial delay until its value is available from DRAM. In the meantime, the branch predictor assumes the `if` will be true, then speculative execution adds `x` to the base address of `array1` and requests the data at the resulting address from the memory subsystem. This read is a cache hit, and quickly returns the value of the secret byte k. Speculative execution continues, using k to compute the address of `array2 [k*4096]`, and sending a request to read this address from memory (resulting in a cache miss). At some point after the read from `array2` is initiated, the processor realizes that its speculative execution was erroneous and rewinds its register state. However, the speculative read from `array2` affects the cache state in an address-specific manner, where the address depends on k.

To complete the attack, the adversary measures which location in `array2` was brought into the cache, for example, via Flush+Reload or Prime+Probe. This reveals the value of k, because the victim's speculative execution cached `array2 [k*4096]`, causing `array2[i*4096]` to read quickly for $i = k$, but slowly for all other $k \in 0.255$. Alternatively, by using Evict+Time, the adversary can immediately call the target function again with an in-bounds value `x'` and measure how long this second call takes. If `array1[x']` equals k, then the location accessed in `array2` is in the cache, and the operation will tend to be faster. (The multiplication by 4096 simplifies the attack by ensuring that each potential value of k maps to a different memory page, avoiding effects due to intra-page prefetching.)

Many different scenarios can lead to exploitable leaks using this variant. For example, instead of performing a bounds check, the mispredicted conditional branch(es) could be checking a previously computed safety result or an object type. Similarly, the code that is speculatively executed can take other forms, such as leaking a comparison result into a fixed memory location or may be spread over a much larger number of instructions. The cache status described above is also more restrictive than may be required. For example, in some scenarios, the attack works even if `array1_size` is cached, for example, if branch prediction results are applied during speculative execution even if the values involved in the comparison are known. As a result, mitigation efforts are likely to be ineffective if targeted narrowly to a specific code pattern or scenario (see Sections 6 and 7).

4.1. Experimental results

We performed experiments on multiple Intel x86 processor architectures (Ivy Bridge, Haswell, Broadwell, Skylake, and Kaby Lake) and AMD Ryzen. The Spectre vulnerability was observed on all these CPUs, and we observed that speculative execution can run hundreds of instructions ahead. Similar results were observed on both 32- and 64-bit modes, and under both Linux and Windows. Some processors based on the ARM architecture also support speculative execution, and our initial testing confirmed that ARM Cortex-A57 and Cortex-A53 and Qualcomm Kyro 280 CPUs.

4.2. Example implementation in C

Proof-of-concept code in C for x86 processors is found in the full paper or is available from https://gist.github.com/anonymous/99a72c9c1003f8ae0707b4927ec1bd8a. This unoptimized implementation can read around 10KB/s on an i7-4650U with a low (<0.01%) error rate.

4.3. Example implementation in JavaScript

We developed a proof-of-concept in JavaScript and tested it in Google Chrome version 62.0.3202, which allows a Website to read private memory from the process in which it runs. The code is illustrated in Listing 1.

On branch-predictor mistraining passes, `index` is set (via bit operations) to an in-range value. On the final iteration, `index` is set to an out-of-bounds address into `simpleByteArray`. We used a variable `localJunk` to ensure that operations are not optimized out. The "`|0`" operation converts the value to a 32-bit integer, acting as an optimization hint to the JavaScript interpreter. Like other optimized JavaScript engines, V8 performs just-in-time compilation to convert JavaScript into machine language. Dummy operations were placed in the code surrounding Listing 1 to make

```
if (index < simpleByteArray.length) {
  index = simpleByteArray[index | 0];
  index = (((index * 4096)|0) & (32*1024*1024-1))|0;
  localJunk ^= probeTable[index|0]|0;
}
```

Listing 1. Exploiting speculative execution via JavaScript.

```
cmpl r15,[rbp-0xe0]             ; Compare index (r15) against simpleByteArray.length
jnc 0x24dd099bb870              ; If index >= length, branch to instruction after movq below
REX.W leaq rsi,[r12+rdx*1]      ; Set rsi = r12 + rdx = addr of first byte in simpleByteArray
movzxbl rsi,[rsi+r15*1]         ; Read byte from address rsi+r15 (= base address + index)
shll rsi,12                     ; Multiply rsi by 4096 by shifting left 12 bits
andl rsi,0x1ffffff              ; AND reassures JIT that next operation is in-bounds
movzxbl rsi,[rsi+r8*1]          ; Read from probeTable
xorl rsi,rdi                    ; XOR the read result onto localJunk
REX.W movq rdi,rsi              ; Copy localJunk into rdi
```

Listing 2. Disassembly of JavaScript example from Listing 1.

`simpleByteArray.length` be stored in local memory so that it can be removed from the cache during the attack. See Listing 2 for the resulting disassembly output from D8.

As the `clflush` instruction is not accessible from JavaScript, we use cache eviction instead,[19] that is, we access other memory locations in a way such that the target memory locations are evicted afterward. The leaked results are conveyed via the cache status of `probeTable[`n`*4096]` for $n \in 0.255$, so the attacker has to evict these 256 cache lines. The length parameter (`simpleByteArray.length` in the JavaScript code and `[ebp-0xe0]` in the disassembly) needs to be evicted as well.

JavaScript does not provide access to the `rdtscp` instruction, and Chrome intentionally degrades the accuracy of its high-resolution timer to dissuade timing attacks using `performance.now()`. However, the Web Workers feature of HTML5 makes it simple to create a separate thread that repeatedly decrements a value in a shared memory location.[22] This approach yields a high-resolution timer that provides sufficient resolution.

4.4. Example implementation exploiting eBPF

As a third example of exploiting conditional branches, we developed a reliable proof-of-concept which leaks kernel memory from an unmodified Linux kernel without patches against Spectre by abusing the extended BPF (eBPF) interface. eBPF is a Linux kernel interface based on the Berkeley Packet Filter (BPF)[18] that can be used for a variety of purposes, such as filtering packets based on their contents. eBPF permits unprivileged users to trigger the interpretation or JIT compilation and subsequent execution of user-supplied, kernel-verified eBPF bytecode in the context of the kernel. The basic concept of the attack is similar to the concept of the attack against JavaScript.

In this attack, we use the eBPF code only for the speculatively executed code. We use native code in user space to acquire the covert channel information. This is a difference to the JavaScript example above, where both functions are implemented in the scripted language. To speculatively access secret-dependent locations in user-space memory, we perform speculative out-of-bounds memory accesses to an array in kernel memory, with an index large enough that the user-space memory is accessed instead.

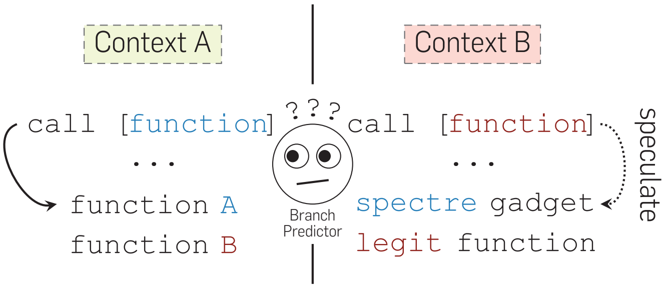
See the full paper for additional details.

5. VARIANT 2: POISONING INDIRECT BRANCHES

In this section, we demonstrate how indirect branches can be poisoned by an attacker and the resulting misprediction of indirect branches can be exploited. If the determination of the destination address of an indirect branch is delayed, due to a cache miss, speculative execution will often continue at a location predicted from previous code execution.

In Spectre variant 2, the adversary mistrains the branch predictor with malicious destinations, such that speculative execution continues at a location chosen by the adversary. This is illustrated in Figure 2, where the branch predictor is (mis-)trained in one context and applies the prediction in a different context. More specifically, the adversary can misdirect speculative execution to locations that would never occur during a legitimate program execution. This is an extremely powerful means for attackers, for example, enabling exposure of victim's memory even in the absence of an exploitable conditional branch misprediction leveraged in Section 4.

Figure 2. The branch predictor is (mis-)trained in the attacker-controlled context A. In context B, the branch predictor makes its prediction on the basis of training data from context A, leading to speculative execution at an attacker-chosen address which corresponds to the location of the Spectre gadget in the victim's address space.



For a simple example attack, we consider an attacker seeking to read a victim's memory, who has control over two registers when an indirect branch occurs. This commonly occurs in real-world binaries because functions manipulating externally received data routinely make function calls although registers contain values that an attacker controls. Often these values are ignored by the called function and instead they are simply pushed onto the stack in the function prologue and restored in the function epilogue.

The attacker also needs to locate a "Spectre gadget," that is, a code fragment whose speculative execution will transfer the victim's sensitive information into a covert channel. For this example, a simple and effective gadget would be formed by two instructions (which do not necessarily need to be adjacent) where the first adds (or XORs, subtracts, etc.) the memory location addressed by an attacker-controlled register R1 onto an attacker-controlled register R2, followed by any instruction that accesses memory at the address in R2. In this case, the gadget provides the attacker control (via R1) over which address to leak and control (via R2) over how the leaked memory maps to an address which is read by the second instruction. On the CPUs we tested, the gadget must reside in memory executable by the victim for the CPU to perform speculative execution. However, with several megabytes of shared libraries mapped into most processes,[5] an attacker has ample space to search for gadgets without even having to search in the victim's own code.

The choice of gadget depends on what state is known or controlled by the adversary, where the information sought by the adversary resides (e.g., registers, stack, memory, etc.), the adversary's ability to control speculative execution, what instruction sequences are available to form gadgets, and what channels can leak information from speculative operations. For example, a cryptographic function that returns a secret value in a register may become exploitable if the attacker can simply induce speculative execution at an instruction that brings memory from the address specified in the register into the cache. Likewise, although the example above assumes that the attacker controls two registers, the attacker's control over a single register, value on the stack, or memory value is sufficient for some gadgets.

In many ways, exploitation is similar to return-oriented programming (ROP),[23] except that the correctly written software is vulnerable, gadgets are limited in their duration but need not terminate cleanly (because the CPU will eventually recognize the speculative error), and gadgets must exfiltrate data via side channels rather than explicitly. Still, speculative execution can perform complex sequences of instructions, such as reading from the stack, performing arithmetic, branching (including multiple times), and reading memory.

The full paper includes details about branch predictor behavior and mistraining techniques for a range of processors, as well as attack implementations targeting a Microsoft Windows application and the KVM hypervisor.

6. VARIATIONS

So far, we have demonstrated attacks that leverage changes in the state of the cache that occur during speculative execution. Future processors (or existing processors with different microcode) may behave differently, for example, if measures are taken to prevent speculatively executed code from modifying the cache state. In this section, we examine potential variants and conclude that virtually any observable effect of speculatively executed code can potentially lead to leaks of sensitive information. Although the following techniques are not needed for the processors we tested, it is essential to understand potential variations when designing or evaluating mitigations.

Spectre variant 4. Spectre variant 4 uses speculation in the store-to-load forwarding logic.[7] The processor speculates that a load does not depend on the previous store. The exploitation mechanics are similar to variant 1 and 2 that we discussed in detail in this paper.

Evict+Time. The Evict+Time attack[20] works by measuring the timing of operations that depend on the state of the cache. This technique can be adapted to use Spectre as follows. Consider the code:

```
if (false but mispredicts as true)
   read array1[R1]
read [R2]
```

Suppose register R1 contains a secret value. If the speculatively executed memory read of `array1[R1]` is a cache hit, then nothing will go on the memory bus, and the read from `[R2]` will initiate quickly. If the read of `array1[R1]` is a cache miss, then the second read may take longer, resulting in different timing for the victim thread. In addition, other components in the system that can access memory (such as other processors) may be able to sense the presence of activity on the memory bus or other effects of the memory read. We note that this attack can work even if speculative execution does not modify the contents of the cache. All that is required is that the state of the cache affects the timing of speculatively executed code or some other property that ultimately becomes visible to the attacker.

Instruction timing. Spectre vulnerabilities do not necessarily need to involve caches. Instructions whose timing depends on the values of the operands may leak information on the operands. In the following example, the multiplier is occupied by the speculative execution of `multiply R1, R2`. The timing of when the multiplier becomes available for `multiply R3, R4` (either for out-of-order execution or after the misprediction is recognized) could be affected by the timing of the first multiplication, revealing information about `R1` and `R2`.

```
if (false but mispredicts as true)
   multiply R1, R2
multiply R3, R4
```

Contention on the register file. Suppose the CPU has a register file with a finite number of registers available for storing checkpoints for speculative execution. In the following example, if `condition on R1` in the second "if" is true, then an extra speculative execution checkpoint will be created than if `condition on R1` is false. If an adversary can detect this checkpoint, if speculative execution of code in hyperthreads is reduced due to a shortage of storage, this reveals information about `R1`.

```
if (false but mispredicts as true)
   if (condition on R1)
      if (condition)
```

Variations on speculative execution. Even code that contains no conditional branches can potentially be at risk. For example, consider the case where an attacker wishes to determine whether `R1` contains an attacker-chosen value *X* or some other value. The ability to make such determinations is sufficient to break some cryptographic implementations. The attacker mistrains the branch predictor such that after an interrupt occurs, the interrupt return mispredicts to an instruction that reads memory `[R1]`. The attacker then chooses *X* to correspond to a memory address suitable for Flush+Reload, revealing whether `R1` = *X*. Although the `iret` instruction is serializing on Intel CPUs, other processors may apply branch predictions.

Leveraging arbitrary observable effects. Virtually any observable effect of speculatively executed code can be leveraged to create the covert channel that leaks sensitive information. For example, consider a processor that has been designed so that speculative reads cannot modify the cache. When the code here runs, the speculative lookup in `array2` still occurs, and its timing will be affected by the cache state entering speculative execution. This timing in turn can affect the depth and timing of subsequent speculative operations. Thus, by manipulating the state of the cache prior to speculative execution, an adversary can potentially leverage virtually any observable effect from speculative execution.

```
if (x < array1_size){
  y = array2[array1[x] * 4096];
  // do something detectable when
  // speculatively executed
}
```

The final observable operation could involve virtually any side channel or covert channel, such as contention for resources (buses, arithmetic units, etc.) and conventional side-channel emanations (such as electromagnetic radiation or power consumption).

A more general form of this would be:

```
if (x < array1_size) {
  y = array1[x];
  // something using y that is observable
  // when speculatively executed
}
```

7. MITIGATION OPTIONS

Several countermeasures for Spectre attacks have been proposed. Each addresses one or more of the features that the attack relies upon. We now discuss these countermeasures and their applicability, effectiveness, and cost.

7.1. Preventing speculative execution

Speculative execution is required for Spectre attacks. Ensuring that instructions are executed only when the control flow leading to them is ascertained would prevent speculative execution and, with it, Spectre attacks. Although effective as a countermeasure, this would cause a significant degradation in the performance of the processor.

Although current processors do not appear to have methods that allow software to disable speculative execution, such modes could be added in future processors, or potentially be introduced via microcode changes. Still, this solution is unlikely to provide an immediate fix to the problem.

Alternatively, the software could be modified to use *serializing* or *speculation blocking* instructions that ensure that instructions following them are not executed speculatively. For x86, CPU vendors recommend the use of the `lfence` instruction.[9] The safest approach to protect conditional branches would be to add such an instruction on the two outcomes of every conditional branch, but this amounts to disabling branch prediction and would dramatically reduce performance. An improved approach is to use static analysis[9] to reduce the number of speculation blocking instructions required, as many code paths do not have the potential to read and leak out-of-bounds memory. In contrast, Microsoft's C compiler MSVC takes an approach of defaulting to unprotected code unless the static analyzer detects a known bad code pattern but, as a result, misses many vulnerable code patterns.[12]

The approach requires that all potentially vulnerable software is instrumented. Hence, for protection, updated software binaries and libraries are required. This could be an issue for legacy software. In addition, this approach is primarily focused on variant 1, and does not address all variants.

7.2. Preventing access to secret data

Other countermeasures can prevent speculatively executed code from accessing secret data. One such measure, used by the Google Chrome Web browser, is to execute each Website in a separate process.[26] Because Spectre attacks only leverage the victim's permissions, an attack such as the one we performed using JavaScript (cf. Section IV-C) would not be able to access data from the processes assigned to other Websites.

WebKit employs two strategies for limiting access to secret data by speculatively executed code.[21] The first strategy replaces array bounds checking with index masking. Instead of checking that an array index is within the bounds of the array, WebKit applies a bit mask to the index, ensuring that it is not much bigger than the array size. Although masking may result in access outside the bounds of the array, this limits the distance of the bounds violation, preventing the attacker from accessing arbitrary memory. The second strategy protects access to pointers by xoring them with a pseudo-random *poison* value. An adversary who does not know the poison value cannot use a poisoned pointer (although various cache attacks could leak the poison value), and the poison value ensures that mispredictions on the branch instructions used for type checks will result in pointers associated with the type being used for another type.

7.3. Preventing data from entering covert channels

Future processors could potentially track whether data was fetched as the result of a speculative operation and, if so, prevent that data from being used in subsequent operations that might leak it. However, current processors do not generally have this capability.

7.4. Limiting data extraction from covert channels

To exfiltrate information from transient instructions, Spectre attacks use a covert communication channel. Multiple approaches have been suggested for mitigating such channels (cf. Ge et al.[4]). A common approach is to degrade timers, which may decrease attack performance, but does not guarantee that attacks are not possible.

7.5. Preventing branch poisoning

To prevent indirect branch poisoning, Intel and AMD extended the ISA with mechanisms for limiting adversaries' ability to influence indirect branch speculation.[2, 8] The performance impact varies from a few percent to a factor of 4 or more, depending on which countermeasures are employed, how comprehensively they are applied (e.g., limited use in the kernel vs. full protection for all processes), and the efficiency of the hardware and microcode implementations.

Google suggests an alternative mechanism for preventing indirect branch poisoning called *retpolines*.[28] A retpoline is a code sequence that replaces indirect branches with return instructions. The construct further contains code that makes sure that the return instruction is predicted to a benign endless loop through the return stack buffer, although the actual target destination is reached by pushing it on the stack and returning to it, that is, using the `ret` instruction. When return instructions can be predicted by other means, the method may be impractical. Intel issued microcode updates for some processors, which fall back to the BTB for the prediction, to disable this fallback mechanism.[9]

8. CONCLUSION

A fundamental assumption underpinning software security techniques is that the processor will faithfully execute program instructions, such as its safety checks. This paper presents Spectre attacks, which leverage the fact that speculative execution violates this assumption. The techniques we demonstrate are practical, do not require any software vulnerabilities, and allow adversaries to read private memory and register contents from other processes and security contexts.

Software security fundamentally depends on having a clear common understanding between hardware and software developers as to what information CPU implementations are (and are not) permitted to expose from computations. As a result, although the countermeasures described in the previous section may help limit practical exploits in the short term, they are only stopgap measures as there is typically formal architectural assurance as to whether any specific code construction is safe across today's processors—much less future designs. As a result, we believe that long-term solutions will require fundamentally changing instruction set architectures.

More broadly, there are trade-offs between security and performance. The vulnerabilities in this paper, as well as many others, arise from a long-standing focus in the technology industry on maximizing performance. As a result, processors, compilers, device drivers, operating systems, and numerous other critical components have evolved compounding layers of complex optimizations that introduce security risks. As the costs of insecurity rise, these design choices need to be revisited. In many cases, alternative implementations optimized for security will be required.

Acknowledgments

Several authors of this paper found Spectre independently, ultimately leading to this collaboration. We thank Mark Brand from Google Project Zero for contributing ideas. We thank Intel for their professional handling of this issue through communicating a clear timeline and connecting all involved researchers. We thank ARM for technical discussions on aspects of this issue. We thank Qualcomm and other vendors for their fast response upon disclosing the issue. Finally, we want to thank our reviewers for their valuable comments.

Daniel Gruss, Moritz Lipp, Stefan Mangard, and Michael Schwarz were supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402).

Daniel Genkin was supported by NSF awards #1514261 and #1652259, financial assistance award 70NANB15H328 from the U.S. Department of Commerce, National Institute of Standards and Technology, the 2017–2018 Rothschild Postdoctoral Fellowship, and the Defense Advanced Research Project Agency (DARPA) under Contract #FA8650-16-C-7622.

References

1. Acıiçmez, O., Koç, Ç.K., Seifert, J.-P. Predicting Secret Keys Via Branch Prediction. In: *CT-RSA*, 2007.
2. Advanced Micro Devices, Inc. Software Techniques for Managing Speculation on AMD Processors, 2018. [Online]. http://developer:amd:com/wordpress/media/2013/12/Managing-Speculation-on-AMD-Processors:pdf
3. Bernstein, D.J. Cache-Timing Attacks on AES. 2005. [Online]. http://cr:yp:to/antiforgery/cachetiming-20050414:pdf
4. Ge, Q., Yarom, Y., Cock, D., Heiser, G. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptogr. Eng. 1*, 8 (2018), 1–27.
5. Gruss, D., Spreitzer, R., Mangard, S. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security Symposium*, 2015.
6. Gullasch, D., Bangerter, E., Krenn, S. Cache games—Bringing access-based cache attacks on AES to practice. In *S&P*, 2011.
7. Horn, J. Speculative execution, variant 4: Speculative store bypass, 2018. [Online]. https://bugs:chromium:org/p/project-zero/issues/detail?id=1528
8. Intel Corp. Speculative Execution Side Channel Mitigations, Jan. 2018. [Online]. https://software:intel:com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations:pdf
9. Intel Corp. Intel Analysis of Speculative Execution Side Channels, Jan. 2018. [Online]. https://newsroom:intel:com/wpcontent/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels:pdf
10. Irazoqui Apecechea, G., Eisenbarth, T., Sunar, B. S$A: A shared cache attack that works across cores and defies VM sandboxing—and its application to AES. In *S&P*, 2015.

11. Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J.H., Lee, D., Wilkerson, C., Lai, K., Mutlu, O. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ISCA*, 2014.
12. Kocher, P. Spectre mitigations in Microsoft's C/C++ compiler, 2018. [Online]. https://www:paulkocher:com/doc/MicrosoftCompilerSpectreMitigation:html
13. Kocher, P., Jaffe, J., Jun, B. Differential power analysis. In *CRYPTO*, 1999.
14. Kocher, P.C. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *CRYPTO*, 1996.
15. Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., Mangard, S. ARMageddon: Cache attacks on mobile devices. In *USENIX Security Symposium*, 2016.
16. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium (to appear)*, 2018.
17. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B. Last-level cache side-channel attacks are practical. In *S&P*, 2015.
18. McCanne, S., Jacobson, V. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter*, 1993.
19. Oren, Y., Kemerlis, V.P., Sethumadhavan, S., Keromytis, A.D. The spy in the sandbox: Practical cache attacks in JavaScript and their implications. In *CCS*, 2015.
20. Osvik, D.A., Shamir, A., Tromer, E. Cache attacks and countermeasures: The case of AES. In *CT-RSA*, 2006.
21. Pizlo, F. What spectre and meltdown mean for WebKit, Jan. 2018. [Online]. https://webkit:org/blog/8048/what-spectreand-meltdown-mean-for-webkit/
22. Schwarz, M., Maurice, C., Gruss, D., Mangard, S. Fantastic timers and where to find them: High-resolution microarchitectural attacks in JavaScript. In *Financial Cryptography*, 2017.
23. Shacham, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS*, 2007.
24. Sibert, O., Porras, P.A., Lindell, R. The Intel 80×86 processor architecture: Pitfalls for secure systems. In *S&P*, 1995.
25. Tang, A., Sethumadhavan, S., Stolfo, S. CLKSCREW: Exposing the perils of security-oblivious energy management. In *USENIX Security Symposium*, 2017.
26. The Chromium Projects. Site Isolation. [Online]. http://www:chromium:org/Home/chromiumsecurity/site-isolation
27. Tsunoo, Y., Saito, T., Suzaki, T., Shigeri, M., Miyauchi, H. Cryptanalysis of DES implemented on computers with cache. In *CHES*, 2003.
28. Turner, P. Retpoline: A software construct for preventing branch-target-injection. [Online]. https://support:google:com/faqs/answer/7625886
29. Yarom, Y., Falkner, K. Flush + reload: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium*, 2014.

Paul Kocher (https://www.paulkocher.com) (paul@paulkocher.com), Independent.

Jann Horn (jannh@google.com), Google Project Zero.

Anders Fogh (Anders_fogh@hotmail.com), G DATA Advanced Analytics.

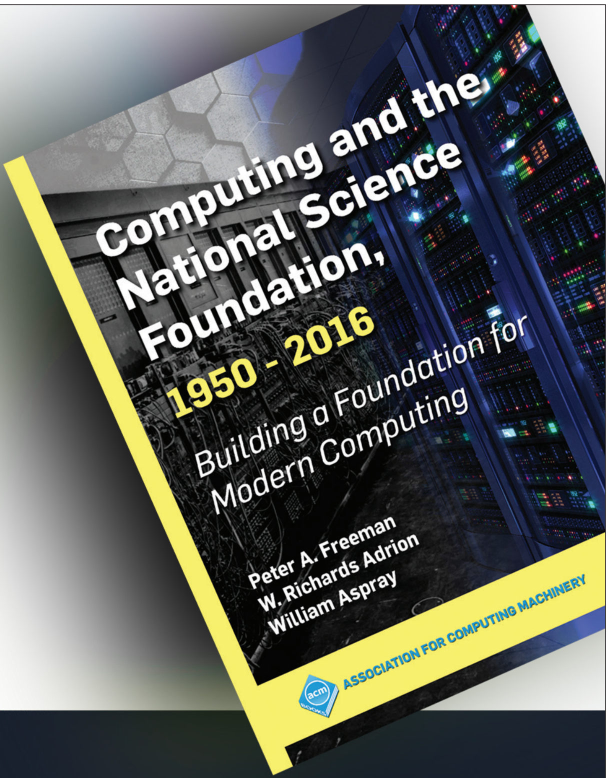
Daniel Genkin (genkin@umich.edu), University of Michigan.

Daniel Gruss, Moritz Lipp, and Michael Schwarz ({daniel.gruss, moritz.lipp, michael.schwarz}@iaik.tugraz.at), Graz University of Technology.

Werner Haas and Thomas Prescher ({werner.haas, thomas.prescher}@cyberus-technology.de), Cyberus Technology.

Mike Hamburg (mhamburg@rambus.com), Rambus, Cryptography Research Division.

Yuval Yarom (yval@cs.adelaide.edu.au), University of Adelaide and Data61.


Computing and the National Science Foundation, 1950 - 2016
Building a Foundation for Modern Computing
Peter A. Freeman
W. Richards Adrion
William Aspray
ASSOCIATION FOR COMPUTING MACHINERY