Project Zero (/)

> list pages

Reading privileged memory with a side-channel

(/2018-JAN-03) Jann Horn

(/2018/01/READING-PRIVILEGED-MEMORY-WITH-SIDE.HTML)

Posted by Jann Horn, Project Zero

We have discovered that CPU data cache timing can be abused to efficiently leak information out of mis-speculated execution, leading to (at worst) arbitrary virtual memory read vulnerabilities across local security boundaries in various contexts.

Variants of this issue are known to affect many modern processors, including certain processors by Intel, AMD and ARM. For a few Intel and AMD CPU models, we have exploits that work against real software. We reported this issue to Intel, AMD and ARM on 2017-06-01 [1].

So far, there are three known variants of the issue:

- Variant 1: bounds check bypass (CVE-2017-5753)
- Variant 2: branch target injection (CVE-2017-5715)
- Variant 3: rogue data cache load (CVE-2017-5754)

Before the issues described here were publicly disclosed, Daniel Gruss, Moritz Lipp, Yuval Yarom, Paul Kocher, Daniel Genkin, Michael Schwarz, Mike Hamburg, Stefan Mangard, Thomas Prescher and Werner Haas also reported them; their [writeups/blogposts/paper drafts] are at:

- Spectre (https://spectreattack.com/spectre.pdf) (variants 1 and 2)
- Meltdown (https://meltdownattack.com/meltdown.pdf) (variant 3)

During the course of our research, we developed the following proofs of concept (PoCs):

1. A PoC that demonstrates the basic principles behind variant 1 in userspace on the tested Intel Haswell Xeon CPU, the AMD FX CPU, the AMD PRO CPU and an ARM Cortex A57 [2]. This PoC only tests for the ability to read data inside mis-speculated execution within the same process, without crossing any privilege boundaries.
2. A PoC for variant 1 that, when running with normal user privileges under a modern Linux kernel with a distro-standard config, can perform arbitrary reads in a 4GiB range [3] in kernel virtual memory on the Intel Haswell Xeon CPU. If the kernel's BPF JIT is enabled (non-default configuration), it also works on the AMD PRO CPU. On the Intel Haswell Xeon CPU, kernel virtual memory can be read at a rate of around 2000 bytes per second after around 4 seconds of startup time. [4]
3. A PoC for variant 2 that, when running with root privileges inside a KVM guest created using virt-manager on the Intel Haswell Xeon CPU, with a specific (now outdated) version of Debian's distro kernel [5] running on the host, can read host kernel memory at a rate of around 1500 bytes/second, with room for optimization. Before the attack can be performed, some initialization has to be performed that takes roughly between 10 and 30 minutes for a machine with 64GiB of RAM; the needed time should scale roughly linearly with the amount of host RAM. (If 2MB hugepages are available to the guest, the initialization should be much faster, but that hasn't been tested.)
4. A PoC for variant 3 that, when running with normal user privileges, can read kernel memory on the Intel Haswell Xeon CPU under some precondition. We believe that this precondition is that the targeted kernel memory is present in the L1D cache.

For interesting resources around this topic, look down into the "Literature" section.

A warning regarding explanations about processor internals in this blogpost: This blogpost contains a lot of speculation about hardware internals based on observed behavior, which might not necessarily correspond to what processors are actually doing.

We have some ideas on possible mitigations and provided some of those ideas to the processor vendors; however, we believe that the processor vendors are in a much better position than we are to design and evaluate mitigations, and we expect them to be the source of authoritative guidance.

The PoC code and the writeups that we sent to the CPU vendors are available here: https://bugs.chromium.org/p/project-zero/issues/detail?id=1272 (https://bugs.chromium.org/p/project-zero/issues/detail?id=1272) .

Tested Processors

- Intel(R) Xeon(R) CPU E5-1650 v3 @ 3.50GHz (called "Intel Haswell Xeon CPU" in the rest of this document)
- AMD FX(tm)-8320 Eight-Core Processor (called "AMD FX CPU" in the rest of this document)
- AMD PRO A8-9600 R7, 10 COMPUTE CORES 4C+6G (called "AMD PRO CPU" in the rest of this document)
- An ARM Cortex A57 core of a Google Nexus 5x phone [6] (called "ARM Cortex A57" in the rest of this document)

Glossary

retire: An instruction retires when its results, e.g. register writes and memory writes, are committed and made visible to the rest of the system. Instructions can be executed out of order, but must always retire in order.

logical processor core: A logical processor core is what the operating system sees as a processor core. With hyperthreading enabled, the number of logical cores is a multiple of the number of physical cores.

cached/uncached data: In this blogpost, "uncached" data is data that is only present in main memory, not in any of the cache levels of the CPU. Loading uncached data will typically take over 100 cycles of CPU time.

speculative execution: A processor can execute past a branch without knowing whether it will be taken or where its target is, therefore executing instructions before it is known whether they should be executed. If this speculation turns out to have been incorrect, the CPU can discard the resulting state without architectural effects and continue execution on the correct execution path. Instructions do not retire before it is known that they are on the correct execution path.

mis-speculation window: The time window during which the CPU speculatively executes the wrong code and has not yet detected that mis-speculation has occurred.

Variant 1: Bounds check bypass

This section explains the common theory behind all three variants and the theory behind our PoC for variant 1 that, when running in userspace under a Debian distro kernel, can perform arbitrary reads in a 4GiB region of kernel memory in at least the following configurations:

- Intel Haswell Xeon CPU, eBPF JIT is off (default state)
- Intel Haswell Xeon CPU, eBPF JIT is on (non-default state)
- AMD PRO CPU, eBPF JIT is on (non-default state)

The state of the eBPF JIT can be toggled using the net.core.bpf_jit_enable sysctl.

Theoretical explanation

The Intel Optimization Reference Manual (https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf) says the following regarding Sandy Bridge (and later microarchitectural revisions) in section 2.3.2.3 ("Branch Prediction"):

> Branch prediction predicts the branch target and enables the processor to begin executing instructions long before the branch true execution path is known.

In section 2.3.5.2 ("L1 DCache"):

> Loads can:
> [...]
> - Be carried out speculatively, before preceding branches are resolved.
> - Take cache misses out of order and in an overlapped manner.

Intel's Software Developer's Manual [7] states in Volume 3A, section 11.7 ("Implicit Caching (Pentium 4, Intel Xeon, and P6 family processors"):

> Implicit caching occurs when a memory element is made potentially cacheable, although the element may never have been accessed in the normal von Neumann sequence. Implicit caching occurs on the P6 and more recent processor families due to aggressive prefetching, branch prediction, and TLB miss handling. Implicit caching is an extension of the behavior of existing Intel386, Intel486, and Pentium processor systems, since software running on these processor families also has not been able to deterministically predict the behavior of instruction prefetch.

Consider the code sample below. If `arr1->length` is uncached, the processor can speculatively load data from `arr1->data[untrusted_offset_from_caller]`. This is an out-of-bounds read. That should not matter because the processor will effectively roll back the execution state when the branch has executed; none of the speculatively executed instructions will retire (e.g. cause registers etc. to be affected).

```
struct array {
 unsigned long length;
 unsigned char data[];
};
struct array *arr1 = ...;
unsigned long untrusted_offset_from_caller = ...;
if (untrusted_offset_from_caller < arr1->length) {
 unsigned char value = arr1->data[untrusted_offset_from_caller];
 ...
}
```

However, in the following code sample, there's an issue. If `arr1->length`, `arr2->data[0x200]` and `arr2->data[0x300]` are not cached, but all other accessed data is, and the branch conditions are predicted as true, the processor can do the following speculatively before `arr1->length` has been loaded and the execution is re-steered:

- load `value = arr1->data[untrusted_offset_from_caller]`
- start a load from a data-dependent offset in `arr2->data`, loading the corresponding cache line into the L1 cache

```
struct array {
 unsigned long length;
```

```
 unsigned char data[];
};
struct array *arr1 = ...; /* small array */
struct array *arr2 = ...; /* array of size 0x400 */
/* >0x400 (OUT OF BOUNDS!) */
unsigned long untrusted_offset_from_caller = ...;
if (untrusted_offset_from_caller < arr1->length) {
 unsigned char value = arr1->data[untrusted_offset_from_caller];
 unsigned long index2 = ((value&1)*0x100)+0x200;
 if (index2 < arr2->length) {
   unsigned char value2 = arr2->data[index2];
 }
}
```

After the execution has been returned to the non-speculative path because the processor has noticed that `untrusted_offset_from_caller` is bigger than `arr1->length`, the cache line containing `arr2->data[index2]` stays in the L1 cache. By measuring the time required to load `arr2->data[0x200]` and `arr2->data[0x300]`, an attacker can then determine whether the value of `index2` during speculative execution was 0x200 or 0x300 - which discloses whether `arr1->data[untrusted_offset_from_caller]&1` is 0 or 1.

To be able to actually use this behavior for an attack, an attacker needs to be able to cause the execution of such a vulnerable code pattern in the targeted context with an out-of-bounds index. For this, the vulnerable code pattern must either be present in existing code, or there must be an interpreter or JIT engine that can be used to generate the vulnerable code pattern. So far, we have not actually identified any existing, exploitable instances of the vulnerable code pattern; the PoC for leaking kernel memory using variant 1 uses the eBPF interpreter or the eBPF JIT engine, which are built into the kernel and accessible to normal users.

A minor variant of this could be to instead use an out-of-bounds read to a function pointer to gain control of execution in the mis-speculated path. We did not investigate this variant further.

Attacking the kernel

This section describes in more detail how variant 1 can be used to leak Linux kernel memory using the eBPF bytecode interpreter and JIT engine. While there are many interesting potential targets for variant 1 attacks, we chose to attack the Linux in-kernel eBPF JIT/interpreter because it provides more control to the attacker than most other JITs.

The Linux kernel supports eBPF since version 3.18. Unprivileged userspace code can supply bytecode to the kernel that is verified by the kernel and then:

- either interpreted by an in-kernel bytecode interpreter
- or translated to native machine code that also runs in kernel context using a JIT engine (which translates individual bytecode instructions without performing any further optimizations)

Execution of the bytecode can be triggered by attaching the eBPF bytecode to a socket as a filter and then sending data through the other end of the socket.

Whether the JIT engine is enabled depends on a run-time configuration setting - but at least on the tested Intel processor, the attack works independent of that setting.

Unlike classic BPF, eBPF has data types like data arrays and function pointer arrays into which eBPF bytecode can index. Therefore, it is possible to create the code pattern described above in the kernel using eBPF bytecode.

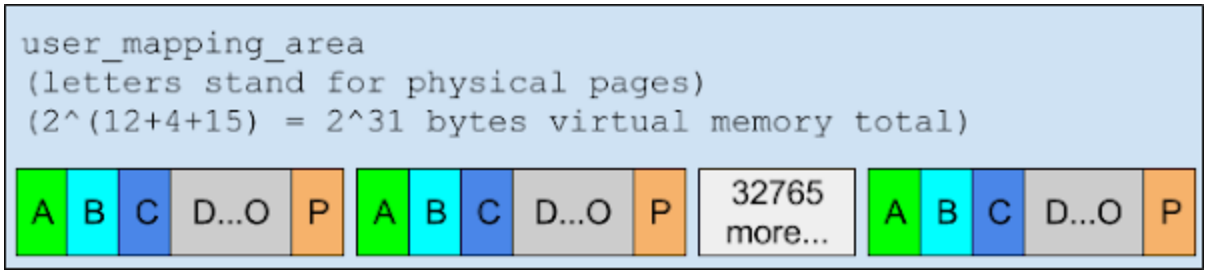
eBPF's data arrays are less efficient than its function pointer arrays, so the attack will use the latter where possible.

Both machines on which this was tested have no SMAP, and the PoC relies on that (but it shouldn't be a precondition in principle).

Additionally, at least on the Intel machine on which this was tested, bouncing modified cache lines between cores is slow, apparently because the MESI protocol is used for cache coherence [8]. Changing the reference counter of an eBPF array on one physical CPU core causes the cache line containing the reference counter to be bounced over to that CPU core, making reads of the reference counter on all other CPU cores slow until the changed reference counter has been written back to memory. Because the length and the reference counter of an eBPF array are stored in the same cache line, this also means that changing the reference counter on one physical CPU core causes reads of the eBPF array's length to be slow on other physical CPU cores (intentional false sharing).

The attack uses two eBPF programs. The first one tail-calls through a page-aligned eBPF function pointer array `prog_map` at a configurable index. In simplified terms, this program is used to determine the address of `prog_map` by guessing the offset from `prog_map` to a userspace address and tail-calling through `prog_map` at the guessed offsets. To cause the branch prediction to predict that the offset is below the length of `prog_map`, tail calls to an in-bounds index are performed in between. To increase the mis-speculation window, the cache line containing the length of `prog_map` is bounced to another core. To test whether an offset guess was successful, it can be tested whether the userspace address has been loaded into the cache.

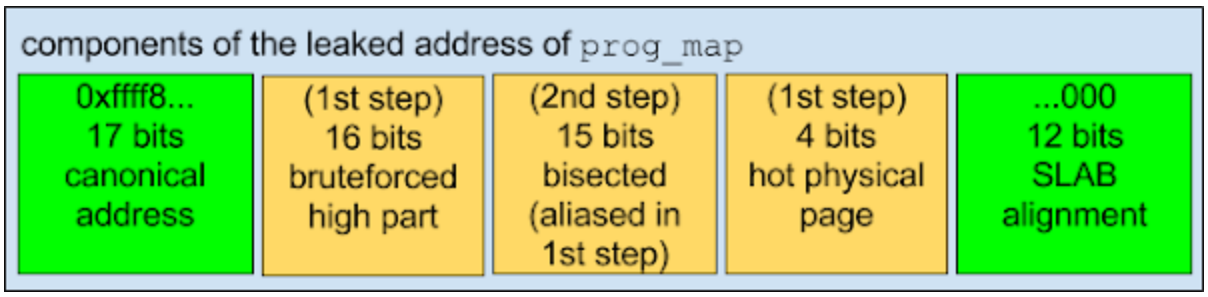
Because such straightforward brute-force guessing of the address would be slow, the following optimization is used: 2^{15} adjacent userspace memory mappings [9], each consisting of 2^4 pages, are created at the userspace address `user_mapping_area`, covering a total area of 2^{31} bytes. Each mapping maps the same physical pages, and all mappings are present in the pagetables.



(/images/oldblog/blogger.googleusercontent.com/img/b/R29vZ2xl/AVvXsEidyTz_s624_image3.png.1.png)

This permits the attack to be carried out in steps of 2^{31} bytes. For each step, after causing an out-of-bounds access through `prog_map`, only one cache line each from the first 2^4 pages of `user_mapping_area` have to be tested for cached memory. Because the L3 cache is physically indexed, any access to a virtual address mapping a physical page will cause all other virtual addresses mapping the same physical page to become cached as well.

When this attack finds a hit—a cached memory location—the upper 33 bits of the kernel address are known (because they can be derived from the address guess at which the hit occurred), and the low 16 bits of the address are also known (from the offset inside `user_mapping_area` at which the hit was found). The remaining part of the address of `user_mapping_area` is the middle.



(/images/oldblog/blogger.googleusercontent.com/img/b/R29vZ2xl/AVvXsEgGWqa_s624_image5.png)

The remaining bits in the middle can be determined by bisecting the remaining address space: Map two physical pages to adjacent ranges of virtual addresses, each virtual address range the size of half of the remaining search space, then determine the remaining address bit-wise.

At this point, a second eBPF program can be used to actually leak data. In pseudocode, this program looks as follows:

```
uint64_t bitmask = <runtime-configurable>;
uint64_t bitshift_selector = <runtime-configurable>;
uint64_t prog_array_base_offset = <runtime-configurable>;
uint64_t secret_data_offset = <runtime-configurable>;
// index will be bounds-checked by the runtime,
```

```
// but the bounds check will be bypassed speculatively
uint64_t secret_data = bpf_map_read(array=victim_array, index=secret_data_offset);
// select a single bit, move it to a specific position, and add the base offset
uint64_t progmap_index = (((secret_data & bitmask) >> bitshift_selector) << 7) +
prog_array_base_offset;
bpf_tail_call(prog_map, progmap_index);
```

This program reads 8-byte-aligned 64-bit values from an eBPF data array "`victim_map`" at a runtime-configurable offset and bitmasks and bit-shifts the value so that one bit is mapped to one of two values that are 2^7 bytes apart (sufficient to not land in the same or adjacent cache lines when used as an array index). Finally it adds a 64-bit offset, then uses the resulting value as an offset into `prog_map` for a tail call.

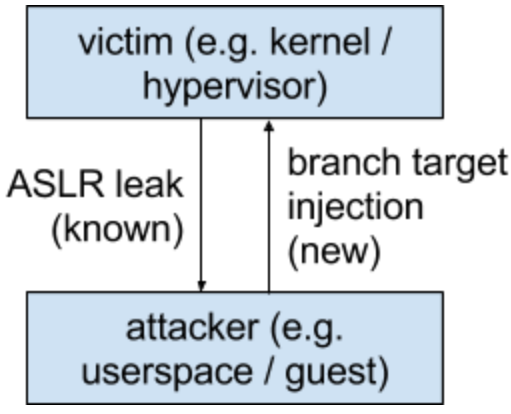
This program can then be used to leak memory by repeatedly calling the eBPF program with an out-of-bounds offset into `victim_map` that specifies the data to leak and an out-of-bounds offset into `prog_map` that causes `prog_map + offset` to point to a userspace memory area. Misleading the branch prediction and bouncing the cache lines works the same way as for the first eBPF program, except that now, the cache line holding the length of `victim_map` must also be bounced to another core.

Variant 2: Branch target injection

This section describes the theory behind our PoC for variant 2 that, when running with root privileges inside a KVM guest created using virt-manager on the Intel Haswell Xeon CPU, with a specific version of Debian's distro kernel running on the host, can read host kernel memory at a rate of around 1500 bytes/second.

Basics

Prior research (see the Literature section at the end) has shown that it is possible for code in separate security contexts to influence each other's branch prediction. So far, this has only been used to infer information about where code is located (in other words, to create interference from the victim to the attacker); however, the basic hypothesis of this attack variant is that it can also be used to redirect execution of code in the victim context (in other words, to create interference from the attacker to the victim; the other way around).



(/images/oldblog/blogger.googleusercontent.com/img/b/R29vZ2xl/AVvXsEgaupA_s278_image2.png.2.png)

The basic idea for the attack is to target victim code that contains an indirect branch whose target address is loaded from memory and flush the cache line containing the target address out to main memory. Then, when the CPU reaches the indirect branch, it won't know the true destination of the jump, and it won't be able to calculate the true destination until it has finished loading the cache line back into the CPU, which takes a few hundred cycles. Therefore, there is a time window of typically over 100 cycles in which the CPU will speculatively execute instructions based on branch prediction.

Haswell branch prediction internals

Some of the internals of the branch prediction implemented by Intel's processors have already been published; however, getting this attack to work properly required significant further experimentation to determine additional details.

This section focuses on the branch prediction internals that were experimentally derived from the Intel Haswell Xeon CPU.

Haswell seems to have multiple branch prediction mechanisms that work very differently:

- A generic branch predictor that can only store one target per source address; used for all kinds of jumps, like absolute jumps, relative jumps and so on.
- A specialized indirect call predictor that can store multiple targets per source address; used for indirect calls.
- (There is also a specialized return predictor, according to Intel's optimization manual, but we haven't analyzed that in detail yet. If this predictor could be used to reliably dump out some of the call stack through which a VM was entered, that would be very interesting.)

Generic predictor

The generic branch predictor, as documented in prior research, only uses the lower 31 bits of the address of the last byte of the source instruction for its prediction. If, for example, a branch target buffer (BTB) entry exists for a jump from 0x4141.0004.1000 to 0x4141.0004.5123, the generic predictor will also use it to predict a jump from 0x4242.0004.1000. When the higher bits of the source address differ like this, the higher bits of the predicted destination change together with it—in this case, the predicted destination address will be 0x4242.0004.5123—so apparently this predictor doesn't store the full, absolute destination address.

Before the lower 31 bits of the source address are used to look up a BTB entry, they are folded together using XOR. Specifically, the following bits are folded together:

bit A	bit B
0x40.0000	0x2000

0x80.0000	0x4000
0x100.0000	0x8000
0x200.0000	0x1.0000
0x400.0000	0x2.0000
0x800.0000	0x4.0000
0x2000.0000	0x10.0000
0x4000.0000	0x20.0000

In other words, if a source address is XORed with both numbers in a row of this table, the branch predictor will not be able to distinguish the resulting address from the original source address when performing a lookup. For example, the branch predictor is able to distinguish source addresses 0x100.0000 and 0x180.0000, and it can also distinguish source addresses 0x100.0000 and 0x180.8000, but it can't distinguish source addresses 0x100.0000 and 0x140.2000 or source addresses 0x100.0000 and 0x180.4000. In the following, this will be referred to as aliased source addresses.

When an aliased source address is used, the branch predictor will still predict the same target as for the unaliased source address. This indicates that the branch predictor stores a truncated absolute destination address, but that hasn't been verified.

Based on observed maximum forward and backward jump distances for different source addresses, the low 32-bit half of the target address could be stored as an absolute 32-bit value with an additional bit that specifies whether the jump from source to target crosses a 2^{32} boundary; if the jump crosses such a boundary, bit 31 of the source address determines whether the high half of the instruction pointer should increment or decrement.

Indirect call predictor

The inputs of the BTB lookup for this mechanism seem to be:

- The low 12 bits of the address of the source instruction (we are not sure whether it's the address of the first or the last byte) or a subset of them.
- The branch history buffer state.

If the indirect call predictor can't resolve a branch, it is resolved by the generic predictor instead. Intel's optimization manual hints at this behavior: "Indirect Calls and Jumps. These may either be predicted as having a monotonic target or as having targets that vary in accordance with recent program behavior."

The branch history buffer (BHB) stores information about the last 29 taken branches - basically a fingerprint of recent control flow - and is used to allow better prediction of indirect calls that can have multiple targets.

The update function of the BHB works as follows (in pseudocode; `src` is the address of the last byte of the source instruction, `dst` is the 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);
}
```

Some of the bits of the BHB state seem to be folded together further using XOR when used for a BTB access, but the precise folding function hasn't been understood yet.

The BHB is interesting for two reasons. First, knowledge about its approximate behavior is required in order to be able to accurately cause collisions in the indirect call predictor. But it also permits dumping out the BHB state at any repeatable program state at which the attacker can execute code - for example, when attacking a hypervisor, directly after a hypercall. The dumped BHB state can then be used to fingerprint the hypervisor or, if the attacker has access to the hypervisor binary, to determine the low 20 bits of the hypervisor load address (in the case of KVM: the low 20 bits of the load address of kvm-intel.ko).

Reverse-Engineering Branch Predictor Internals

This subsection describes how we reverse-engineered the internals of the Haswell branch predictor. Some of this is written down from memory, since we didn't keep a detailed record of what we were doing.

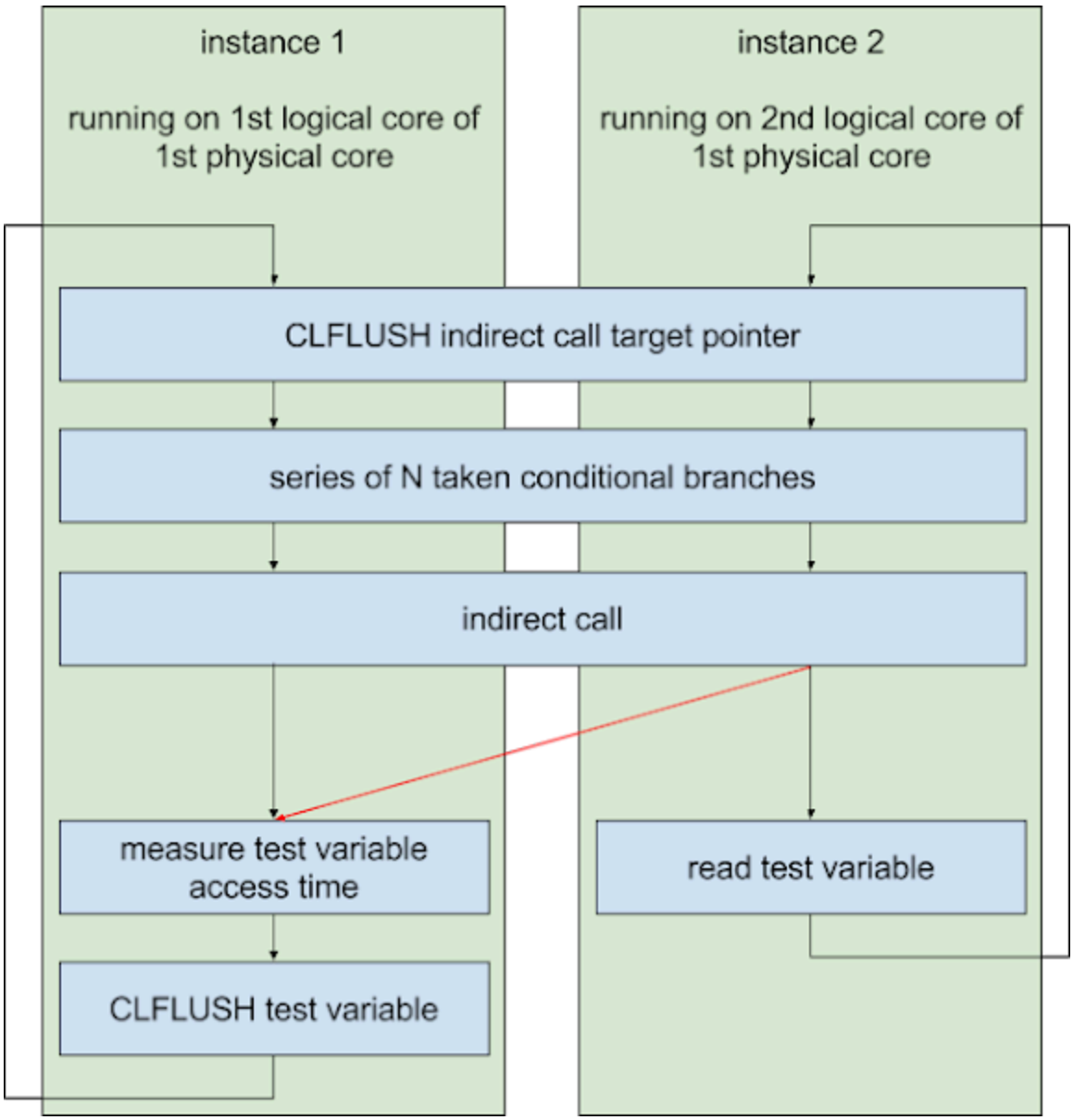
We initially attempted to perform BTB injections into the kernel using the generic predictor, using the knowledge from prior research that the generic predictor only looks at the lower half of the source address and that only a partial target address is stored. This kind of worked - however, the injection success rate was very low, below 1%. (This is the method we used in our preliminary PoCs for method 2 against modified hypervisors running on Haswell.)

We decided to write a userspace test case to be able to more easily test branch predictor behavior in different situations.

Based on the assumption that branch predictor state is shared between hyperthreads [10], we wrote a program of which two instances are each pinned to one of the two logical processors running on a specific physical core, where one instance attempts to perform branch injections while the other measures how often branch injections are successful. Both instances were executed with ASLR disabled and had the same code at the same addresses. The injecting process performed indirect calls to a function that

accesses a (per-process) test variable; the measuring process performed indirect calls to a function that tests, based on timing, whether the per-process test variable is cached, and then evicts it using CLFLUSH. Both indirect calls were performed through the same callsite. Before each indirect call, the function pointer stored in memory was flushed out to main memory using CLFLUSH to widen the speculation time window. Additionally, because of the reference to "recent program behavior" in Intel's optimization manual, a bunch of conditional branches that are always taken were inserted in front of the indirect call.

In this test, the injection success rate was above 99%, giving us a base setup for future experiments.



(/images/oldblog/blogger.googleusercontent.com/img/b/R29vZ2xl/AVvXsEjb4j9_s640_image7.png)

We then tried to figure out the details of the prediction scheme. We assumed that the prediction scheme uses a global branch history buffer of some kind.

To determine the duration for which branch information stays in the history buffer, a conditional branch that is only taken in one of the two program instances was inserted in front of the series of always-taken

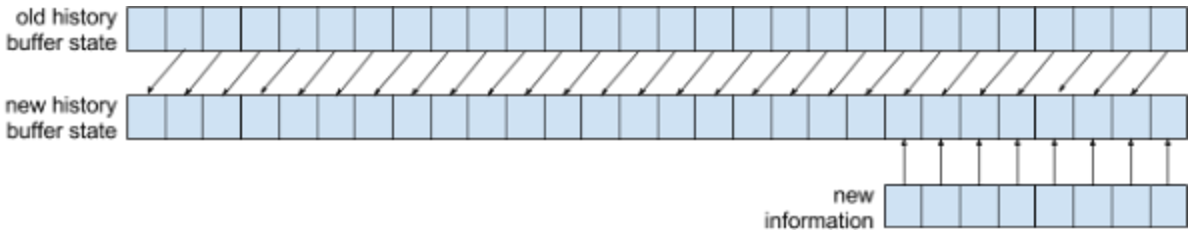
conditional jumps, then the number of always-taken conditional jumps (N) was varied. The result was that for N=25, the processor was able to distinguish the branches (misprediction rate under 1%), but for N=26, it failed to do so (misprediction rate over 99%).
Therefore, the branch history buffer had to be able to store information about at least the last 26 branches.

The code in one of the two program instances was then moved around in memory. This revealed that only the lower 20 bits of the source and target addresses have an influence on the branch history buffer.

Testing with different types of branches in the two program instances revealed that static jumps, taken conditional jumps, calls and returns influence the branch history buffer the same way; non-taken conditional jumps don't influence it; the address of the last byte of the source instruction is the one that counts; IRETQ doesn't influence the history buffer state (which is useful for testing because it permits creating program flow that is invisible to the history buffer).

Moving the last conditional branch before the indirect call around in memory multiple times revealed that the branch history buffer contents can be used to distinguish many different locations of that last conditional branch instruction. This suggests that the history buffer doesn't store a list of small history values; instead, it seems to be a larger buffer in which history data is mixed together.

However, a history buffer needs to "forget" about past branches after a certain number of new branches have been taken in order to be useful for branch prediction. Therefore, when new data is mixed into the history buffer, this can not cause information in bits that are already present in the history buffer to propagate downwards - and given that, upwards combination of information probably wouldn't be very useful either. Given that branch prediction also must be very fast, we concluded that it is likely that the update function of the history buffer left-shifts the old history buffer, then XORs in the new state (see diagram).



(/images/oldblog/blogger.googleusercontent.com/img/b/R29vZ2xl/AVvXsEh8AkJ_s624_image6.png)

If this assumption is correct, then the history buffer contains a lot of information about the most recent branches, but only contains as many bits of information as are shifted per history buffer update about the last branch about which it contains any data. Therefore, we tested whether flipping different bits in the source and target addresses of a jump followed by 32 always-taken jumps with static source and target allows the branch prediction to disambiguate an indirect call. [11]

With 32 static jumps in between, no bit flips seemed to have an influence, so we decreased the number of static jumps until a difference was observable. The result with 28 always-taken jumps in between was that bits 0x1 and 0x2 of the target and bits 0x40 and 0x80 of the source had such an influence; but flipping both 0x1 in the target and 0x40 in the source or 0x2 in the target and 0x80 in the source did not permit disambiguation. This shows that the per-insertion shift of the history buffer is 2 bits and shows which data

is stored in the least significant bits of the history buffer. We then repeated this with decreased amounts of fixed jumps after the bit-flipped jump to determine which information is stored in the remaining bits.

Reading host memory from a KVM guest

Locating the host kernel

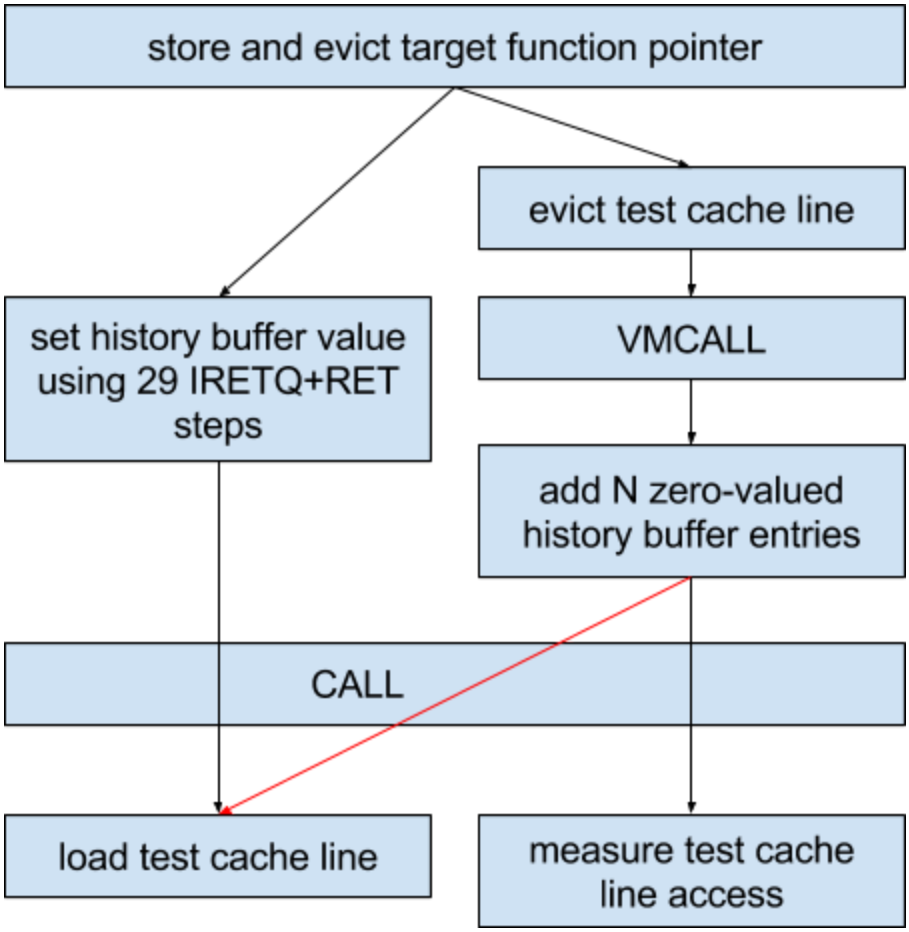
Our PoC locates the host kernel in several steps. The information that is determined and necessary for the next steps of the attack consists of:

- lower 20 bits of the address of kvm-intel.ko
- full address of kvm.ko
- full address of vmlinux

Looking back, this is unnecessarily complicated, but it nicely demonstrates the various techniques an attacker can use. A simpler way would be to first determine the address of vmlinux, then bisect the addresses of kvm.ko and kvm-intel.ko.

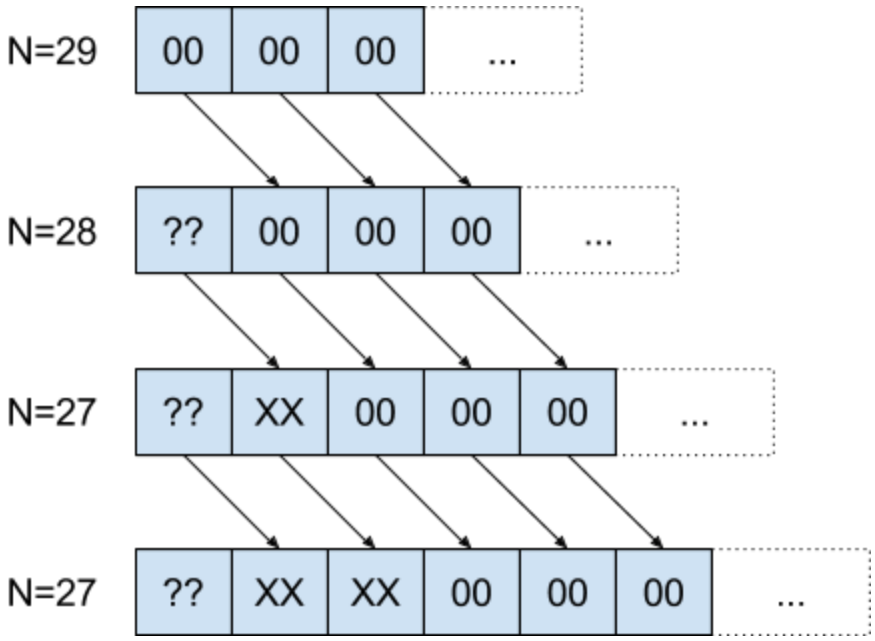
In the first step, the address of kvm-intel.ko is leaked. For this purpose, the branch history buffer state after guest entry is dumped out. Then, for every possible value of bits 12..19 of the load address of kvm-intel.ko, the expected lowest 16 bits of the history buffer are computed based on the load address guess and the known offsets of the last 8 branches before guest entry, and the results are compared against the lowest 16 bits of the leaked history buffer state.

The branch history buffer state is leaked in steps of 2 bits by measuring misprediction rates of an indirect call with two targets. One way the indirect call is reached is from a vmcall instruction followed by a series of N branches whose relevant source and target address bits are all zeroes. The second way the indirect call is reached is from a series of controlled branches in userspace that can be used to write arbitrary values into the branch history buffer.
Misprediction rates are measured as in the section "Reverse-Engineering Branch Predictor Internals", using one call target that loads a cache line and another one that checks whether the same cache line has been loaded.



(/images/oldblog/blogger.googleusercontent.com/img/b/R29vZ2xl/AVvXsEg--Ny_s462_image4.png.1.png)

With N=29, mispredictions will occur at a high rate if the controlled branch history buffer value is zero because all history buffer state from the hypercall has been erased. With N=28, mispredictions will occur if the controlled branch history buffer value is one of 0<<(28*2), 1<<(28*2), 2<<(28*2), 3<<(28*2) - by testing all four possibilities, it can be detected which one is right. Then, for decreasing values of N, the four possibilities are {0|1|2|3}<<(28*2) | (history_buffer_for(N+1) >> 2). By repeating this for decreasing values for N, the branch history buffer value for N=0 can be determined.



(/images/oldblog/blogger.googleusercontent.com/img/b/R29vZ2xl/AVvXsEhlU9B_s457_image1.png.2.png)

At this point, the low 20 bits of kvm-intel.ko are known; the next step is to roughly locate kvm.ko.
For this, the generic branch predictor is used, using data inserted into the BTB by an indirect call from kvm.ko to kvm-intel.ko that happens on every hypercall; this means that the source address of the indirect call has to be leaked out of the BTB.

kvm.ko will probably be located somewhere in the range from `0xffffffffc0000000` to `0xffffffffc4000000`, with page alignment (0x1000). This means that the first four entries in the table in the section "Generic Predictor" apply; there will be 2^4-1=15 aliasing addresses for the correct one. But that is also an advantage: It cuts down the search space from 0x4000 to 0x4000/2^4=1024.

To find the right address for the source or one of its aliasing addresses, code that loads data through a specific register is placed at all possible call targets (the leaked low 20 bits of kvm-intel.ko plus the in-module offset of the call target plus a multiple of 2^{20}) and indirect calls are placed at all possible call sources. Then, alternatingly, hypercalls are performed and indirect calls are performed through the different possible non-aliasing call sources, with randomized history buffer state that prevents the specialized prediction from working. After this step, there are 2^{16} remaining possibilities for the load address of kvm.ko.

Next, the load address of vmlinux can be determined in a similar way, using an indirect call from vmlinux to kvm.ko. Luckily, none of the bits which are randomized in the load address of vmlinux are folded together, so unlike when locating kvm.ko, the result will directly be unique. vmlinux has an alignment of 2MiB and a randomization range of 1GiB, so there are still only 512 possible addresses.
Because (as far as we know) a simple hypercall won't actually cause indirect calls from vmlinux to kvm.ko, we instead use port I/O from the status register of an emulated serial port, which is present in the default configuration of a virtual machine created with virt-manager.

The only remaining piece of information is which one of the 16 aliasing load addresses of kvm.ko is actually correct. Because the source address of an indirect call to kvm.ko is known, this can be solved using bisection: Place code at the various possible targets that, depending on which instance of the code is speculatively executed, loads one of two cache lines, and measure which one of the cache lines gets loaded.

Identifying cache sets

The PoC assumes that the VM does not have access to hugepages.To discover eviction sets for all L3 cache sets with a specific alignment relative to a 4KiB page boundary, the PoC first allocates 25600 pages of memory. Then, in a loop, it selects random subsets of all remaining unsorted pages such that the expected number of sets for which an eviction set is contained in the subset is 1, reduces each subset down to an eviction set by repeatedly accessing its cache lines and testing whether the cache lines are always cached (in which case they're probably not part of an eviction set) and attempts to use the new eviction set to evict all remaining unsorted cache lines to determine whether they are in the same cache set [12].

Locating the host-virtual address of a guest page

Because this attack uses a FLUSH+RELOAD approach for leaking data, it needs to know the host-kernel-virtual address of one guest page. Alternative approaches such as PRIME+PROBE should work without that requirement.

The basic idea for this step of the attack is to use a branch target injection attack against the hypervisor to load an attacker-controlled address and test whether that caused the guest-owned page to be loaded. For this, a gadget that simply loads from the memory location specified by R8 can be used - R8-R11 still contain guest-controlled values when the first indirect call after a guest exit is reached on this kernel build.

We expected that an attacker would need to either know which eviction set has to be used at this point or brute-force it simultaneously; however, experimentally, using random eviction sets works, too. Our theory is that the observed behavior is actually the result of L1D and L2 evictions, which might be sufficient to permit a few instructions worth of speculative execution.

The host kernel maps (nearly?) all physical memory in the physmap area, including memory assigned to KVM guests. However, the location of the physmap is randomized (with a 1GiB alignment), in an area of size 128PiB. Therefore, directly bruteforcing the host-virtual address of a guest page would take a long time. It is not necessarily impossible; as a ballpark estimate, it should be possible within a day or so, maybe less, assuming 12000 successful injections per second and 30 guest pages that are tested in parallel; but not as impressive as doing it in a few minutes.

To optimize this, the problem can be split up: First, brute-force the physical address using a gadget that can load from physical addresses, then brute-force the base address of the physmap region. Because the physical address can usually be assumed to be far below 128PiB, it can be brute-forced more efficiently, and brute-forcing the base address of the physmap region afterwards is also easier because then address guesses with 1GiB alignment can be used.

To brute-force the physical address, the following gadget can be used:

```
ffffffff810a9def:       4c 89 c0                mov    rax,r8
ffffffff810a9df2:       4d 63 f9                movsxd r15,r9d
ffffffff810a9df5:       4e 8b 04 fd c0 b3 a6    mov    r8,QWORD PTR [r15*8-0x7e594c40]
ffffffff810a9dfc:       81
ffffffff810a9dfd:       4a 8d 3c 00             lea    rdi,[rax+r8*1]
ffffffff810a9e01:       4d 8b a4 00 f8 00 00    mov    r12,QWORD PTR [r8+rax*1+0xf8]
ffffffff810a9e08:       00
```

This gadget permits loading an 8-byte-aligned value from the area around the kernel text section by setting R9 appropriately, which in particular permits loading `page_offset_base`, the start address of the physmap. Then, the value that was originally in R8 - the physical address guess minus 0xf8 - is added to the result of the previous load, 0xfa is added to it, and the result is dereferenced.

Cache set selection

To select the correct L3 eviction set, the attack from the following section is essentially executed with different eviction sets until it works.

Leaking data

At this point, it would normally be necessary to locate gadgets in the host kernel code that can be used to actually leak data by reading from an attacker-controlled location, shifting and masking the result appropriately and then using the result of that as offset to an attacker-controlled address for a load. But piecing gadgets together and figuring out which ones work in a speculation context seems annoying. So instead, we decided to use the eBPF interpreter, which is built into the host kernel - while there is no legitimate way to invoke it from inside a VM, the presence of the code in the host kernel's text section is sufficient to make it usable for the attack, just like with ordinary ROP gadgets.

The eBPF interpreter entry point has the following function signature:

```
static unsigned int __bpf_prog_run(void *ctx, const struct bpf_insn *insn)
```

The second parameter is a pointer to an array of statically pre-verified eBPF instructions to be executed - which means that `__bpf_prog_run()` will not perform any type checks or bounds checks. The first parameter is simply stored as part of the initial emulated register state, so its value doesn't matter.

The eBPF interpreter provides, among other things:

- multiple emulated 64-bit registers
- 64-bit immediate writes to emulated registers
- memory reads from addresses stored in emulated registers
- bitwise operations (including bit shifts) and arithmetic operations

To call the interpreter entry point, a gadget that gives RSI and RIP control given R8-R11 control and controlled data at a known memory location is necessary. The following gadget provides this functionality:

```
ffffffff81514edd:       4c 89 ce                mov    rsi,r9
ffffffff81514ee0:       41 ff 90 b0 00 00 00    call   QWORD PTR [r8+0xb0]
```

Now, by pointing R8 and R9 at the mapping of a guest-owned page in the physmap, it is possible to speculatively execute arbitrary unvalidated eBPF bytecode in the host kernel. Then, relatively straightforward bytecode can be used to leak data into the cache.

Variant 3: Rogue data cache load

Basically, read Anders Fogh's blogpost: https://cyber.wtf/2017/07/28/negative-result-reading-kernel-memory-from-user-mode/ (https://cyber.wtf/2017/07/28/negative-result-reading-kernel-memory-from-user-mode/)

In summary, an attack using this variant of the issue attempts to read kernel memory from userspace without misdirecting the control flow of kernel code. This works by using the code pattern that was used for the previous variants, but in userspace. The underlying idea is that the permission check for accessing an address might not be on the critical path for reading data from memory to a register, where the permission check could have significant performance impact. Instead, the memory read could make the result of the read available to following instructions immediately and only perform the permission check asynchronously, setting a flag in the reorder buffer that causes an exception to be raised if the permission check fails.

We do have a few additions to make to Anders Fogh's blogpost:

> "Imagine the following instruction executed in usermode
> `mov rax,[somekernelmodeaddress]`
> It will cause an interrupt when retired, [...]"

It is also possible to already execute that instruction behind a high-latency mispredicted branch to avoid taking a page fault. This might also widen the speculation window by increasing the delay between the read from a kernel address and delivery of the associated exception.

> "First, I call a syscall that touches this memory. Second, I use the prefetcht0 instruction to improve my odds of having the address loaded in L1."

When we used prefetch instructions after doing a syscall, the attack stopped working for us, and we have no clue why. Perhaps the CPU somehow stores whether access was denied on the last access and prevents the attack from working if that is the case?

> "Fortunately I did not get a slow read suggesting that Intel null's the result when the access is not allowed."

That (read from kernel address returns all-zeroes) seems to happen for memory that is not sufficiently cached but for which pagetable entries are present, at least after repeated read attempts. For unmapped memory, the kernel address read does not return a result at all.

Ideas for further research

We believe that our research provides many remaining research topics that we have not yet investigated, and we encourage other public researchers to look into these.
This section contains an even higher amount of speculation than the rest of this blogpost - it contains untested ideas that might well be useless.

Leaking without data cache timing

It would be interesting to explore whether there are microarchitectural attacks other than measuring data cache timing that can be used for exfiltrating data out of speculative execution.

Other microarchitectures

Our research was relatively Haswell-centric so far. It would be interesting to see details e.g. on how the branch prediction of other modern processors works and how well it can be attacked.

Other JIT engines

We developed a successful variant 1 attack against the JIT engine built into the Linux kernel. It would be interesting to see whether attacks against more advanced JIT engines with less control over the system are also practical - in particular, JavaScript engines.

More efficient scanning for host-virtual addresses and cache sets

In variant 2, while scanning for the host-virtual address of a guest-owned page, it might make sense to attempt to determine its L3 cache set first. This could be done by performing L3 evictions using an eviction pattern through the physmap, then testing whether the eviction affected the guest-owned page.

The same might work for cache sets - use an L1D+L2 eviction set to evict the function pointer in the host kernel context, use a gadget in the kernel to evict an L3 set using physical addresses, then use that to identify which cache sets guest lines belong to until a guest-owned eviction set has been constructed.

Dumping the complete BTB state

Given that the generic BTB seems to only be able to distinguish 2^{31-8} or fewer source addresses, it seems feasible to dump out the complete BTB state generated by e.g. a hypercall in a timeframe around the order of a few hours. (Scan for jump sources, then for every discovered jump source, bisect the jump target.) This could potentially be used to identify the locations of functions in the host kernel even if the host kernel is custom-built.

The source address aliasing would reduce the usefulness somewhat, but because target addresses don't suffer from that, it might be possible to correlate (source,target) pairs from machines with different KASLR offsets and reduce the number of candidate addresses based on KASLR being additive while aliasing is bitwise.

This could then potentially allow an attacker to make guesses about the host kernel version or the compiler used to build it based on jump offsets or distances between functions.

Variant 2: Leaking with more efficient gadgets

If sufficiently efficient gadgets are used for variant 2, it might not be necessary to evict host kernel function pointers from the L3 cache at all; it might be sufficient to only evict them from L1D and L2.

Various speedups

In particular the variant 2 PoC is still a bit slow. This is probably partly because:

- It only leaks one bit at a time; leaking more bits at a time should be doable.
- It heavily uses IRETQ for hiding control flow from the processor.

It would be interesting to see what data leak rate can be achieved using variant 2.

Leaking or injection through the return predictor

If the return predictor also doesn't lose its state on a privilege level change, it might be useful for either locating the host kernel from inside a VM (in which case bisection could be used to very quickly discover the full address of the host kernel) or injecting return targets (in particular if the return address is stored in a cache line that can be flushed out by the attacker and isn't reloaded before the return instruction).

However, we have not performed any experiments with the return predictor that yielded conclusive results so far.

Leaking data out of the indirect call predictor

We have attempted to leak target information out of the indirect call predictor, but haven't been able to make it work.

Vendor statements

The following statement were provided to us regarding this issue from the vendors to whom Project Zero disclosed this vulnerability:

Intel

Intel is committed to improving the overall security of computer systems. The methods described here rely on common properties of modern microprocessors. Thus, susceptibility to these methods is not limited to Intel processors, nor does it mean that a processor is working outside its intended functional specification. Intel is working closely with our ecosystem partners, as well as with other silicon vendors whose processors are affected, to design and distribute both software and hardware mitigations for these methods.

For more information and links to useful resources, visit:

https://security-center.intel.com/advisory.aspx?intelid=INTEL-SA-00088&languageid=en-fr (https://security-center.intel.com/advisory.aspx?intelid=INTEL-SA-00088&languageid=en-fr)
http://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf (http://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf)

AMD

AMD provided the following link: http://www.amd.com/en/corporate/speculative-execution (http://www.amd.com/en/corporate/speculative-execution)

ARM

Arm recognises that the speculation functionality of many modern high-performance processors, despite working as intended, can be used in conjunction with the timing of cache operations to leak some information as described in this blog. Correspondingly, Arm has developed software mitigations that we recommend be deployed.

Specific details regarding the affected processors and mitigations can be found at this website: https://developer.arm.com/support/security-update (https://developer.arm.com/support/security-update)

Arm has included a detailed technical whitepaper as well as links to information from some of Arm's architecture partners regarding their specific implementations and mitigations.

Literature

Note that some of these documents - in particular Intel's documentation - change over time, so quotes from and references to it may not reflect the latest version of Intel's documentation.

- https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf (https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf) : Intel's optimization manual has many interesting pieces of optimization advice that hint at relevant microarchitectural behavior; for example:

  - "Placing data immediately following an indirect branch can cause a performance problem. If the data consists of all zeros, it looks like a long stream of ADDs to memory destinations and this can cause resource conflicts and slow down branch recovery. Also, data immediately following indirect branches may appear as branches to the branch predication [sic] hardware, which can branch off to execute other data pages. This can lead to subsequent self-modifying code problems."
  - "Loads can:[...]Be carried out speculatively, before preceding branches are resolved."
  - "Software should avoid writing to a code page in the same 1-KByte subpage that is being executed or fetching code in the same 2-KByte subpage of that is being written. In addition, sharing a page containing directly or speculatively executed code with another processor as a data page can trigger an SMC condition that causes the entire pipeline of the machine and the trace cache to be cleared. This is due to the self-modifying code condition."
  - "if mapped as WB or WT, there is a potential for speculative processor reads to bring the data into the caches"
  - "Failure to map the region as WC may allow the line to be speculatively read into the processor caches (via the wrong path of a mispredicted branch)."

- https://software.intel.com/en-us/articles/intel-sdm (https://software.intel.com/en-us/articles/intel-sdm) : Intel's Software Developer Manuals
- http://www.agner.org/optimize/microarchitecture.pdf (http://www.agner.org/optimize/microarchitecture.pdf) : Agner Fog's documentation of reverse-engineered processor behavior and relevant theory was very helpful for this research.
- http://www.cs.binghamton.edu/~dima/micro16.pdf (http://www.cs.binghamton.edu/~dima/micro16.pdf) and https://github.com/felixwilhelm/mario_baslr (https://github.com/felixwilhelm/mario_baslr) : Prior research by Dmitry Evtyushkin, Dmitry Ponomarev and Nael Abu-Ghazaleh on abusing branch target buffer behavior to leak addresses that we used as a starting point for analyzing the branch prediction of Haswell processors. Felix Wilhelm's research based on this provided the basic idea behind variant 2.
- https://arxiv.org/pdf/1507.06955.pdf (https://arxiv.org/pdf/1507.06955.pdf) : The rowhammer.js research by Daniel Gruss, Clémentine Maurice and Stefan Mangard contains information about L3 cache eviction patterns that we reused in the KVM PoC to evict a function pointer.
- https://xania.org/201602/bpu-part-one (https://xania.org/201602/bpu-part-one) : Matt Godbolt blogged about reverse-engineering the structure of the branch predictor on Intel processors.
- https://www.sophia.re/thesis.pdf (https://www.sophia.re/thesis.pdf) : Sophia D'Antoine wrote a thesis that shows that opcode scheduling can theoretically be used to transmit data between hyperthreads.
- https://gruss.cc/files/kaiser.pdf (https://gruss.cc/files/kaiser.pdf) : Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard wrote a paper on mitigating microarchitectural issues caused by pagetable sharing between userspace and the kernel.
- https://www.jilp.org/ (https://www.jilp.org/) : This journal contains many articles on branch prediction.
- http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/ (http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/) : This blogpost by Henry Wong investigates the L3 cache replacement policy used by Intel's Ivy Bridge architecture.

References

[1] This initial report did not contain any information about variant 3. We had discussed whether direct reads from kernel memory could work, but thought that it was unlikely. We later tested and reported variant 3 prior to the publication of Anders Fogh's work at https://cyber.wtf/2017/07/28/negative-result-reading-kernel-memory-from-user-mode/ (https://cyber.wtf/2017/07/28/negative-result-reading-kernel-memory-from-user-mode/) .

[2] The precise model names are listed in the section "Tested Processors". The code for reproducing this is in the writeup_files.tar archive in our bugtracker, in the folders userland_test_x86 and userland_test_aarch64.

[3] The attacker-controlled offset used to perform an out-of-bounds access on an array by this PoC is a 32-bit value, limiting the accessible addresses to a 4GiB window in the kernel heap area.

[4] This PoC won't work on CPUs with SMAP support; however, that is not a fundamental limitation.

[5] linux-image-4.9.0-3-amd64 at version 4.9.30-2+deb9u2 (available at http://snapshot.debian.org/archive/debian/20170701T224614Z/pool/main/l/linux/linux-image-4.9.0-3-amd64_4.9.30-2%2Bdeb9u2_amd64.deb (http://snapshot.debian.org/archive/debian/20170701T224614Z/pool/main/l/linux/linux-image-4.9.0-3-amd64_4.9.30-2%2Bdeb9u2_amd64.deb) , sha256 5f950b26aa7746d75ecb8508cc7dab19b3381c9451ee044cd2edfd6f5efff1f8, signed via Release.gpg (http://snapshot.debian.org/archive/debian-security/20170701T114538Z/dists/stretch/updates/Release.gpg) , Release (http://snapshot.debian.org/archive/debian-security/20170701T114538Z/dists/stretch/updates/Release) , Packages.xz (http://snapshot.debian.org/archive/debian-security/20170701T114538Z/dists/stretch/updates/main/binary-amd64/Packages.xz)); that was the current distro kernel version when I set up the machine. It is very unlikely that the PoC works with other kernel versions without changes; it contains a number of hardcoded addresses/offsets.

[6] The phone was running an Android build from May 2017.

[7] https://software.intel.com/en-us/articles/intel-sdm (https://software.intel.com/en-us/articles/intel-sdm)

[8] https://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads (https://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads) , section "background"

[9] More than 2^{15} mappings would be more efficient, but the kernel places a hard cap of 2^{16} on the number of VMAs that a process can have.

[10] Intel's optimization manual states that "In the first implementation of HT Technology, the physical execution resources are shared and the architecture state is duplicated for each logical processor", so it would be plausible for predictor state to be shared. While predictor state could be tagged by logical core, that would likely reduce performance for multithreaded processes, so it doesn't seem likely.

[11] In case the history buffer was a bit bigger than we had measured, we added some margin - in particular because we had seen slightly different history buffer lengths in different experiments, and because 26 isn't a very round number.

[12] The basic idea comes from http://palms.ee.princeton.edu/system/files/SP_vfinal.pdf (http://palms.ee.princeton.edu/system/files/SP_vfinal.pdf) , section IV, although the authors of that paper still used hugepages.