Assignment III – Meltdown & Spectre

Due by Week 7 (October 28, 2024) at 23:59

Learning Objectives

The goal of this assignment is to fundamentally understand what transient execution attacks are, what causes instructions to become transiently executed, and how an attacker can abuse these to compromise data confidentiality. While Meltdown is largely fixed in modern CPUs (e.g., in Intel's 9th generation and later), it is one of the seminal works in this field and its mitigations caused a performance setback in the industry [7]. Spectre and its variants, however, largely required software mitigations, and they remain a target of active research.

In the first week, we will give you an insight into the basic workings of the Meltdown and Spectre v1 attacks. In the second week, we will explore a more recent attack called Retbleed [10] which applies Branch Target Injection (BTI) while bypassing mitigations.

Overview

In the early 2018, Spectre [5] and Meltdown [8] were publicly disclosed. The two are subclasses of what is called *transient execution attacks* [1], which are attacks exploiting instructions that reach the CPU backend and execute, but will not get committed to the architectural state (i.e., *retire*). Because the results of these instructions are discarded, we say that such instructions execute *transiently.* The effectivity of any transient attack depends on the ability of an attacker to cause a delay long enough for the transient execution to trigger an observable side effect.

The root cause of Meltdown is that out-of-order execution allows data to be forwarded from the L1d cache (level 1 data cache) asynchronously while the protection bits of its respective page are still being verified. This leads to a race condition, where data from the L1d is transiently used in operations despite page protection bits prohibiting it. Meltdown exploits the User/Supervisor protection bit, which indicates whether a page is accessible from user mode. The result is that privileged memory can be used transiently by a user-mode program — essentially breaking the isolation between the user and the OS.

The root cause of Spectre is that the branch predictor of the CPU frontend provides an incorrect prediction, causing transient execution of the wrong control-flow path. The mispredicted control flow is eventually corrected when the dependencies necessary for the correct prediction are resolved. This procedure is initiated by a microarchitectural resteer signal [11]. We discuss two variants of Spectre in this assignment: Spectre Variant 1 (v1, Spectre-PHT) and Variant 2 (v2, Spectre-BTI). While the underlying principle is the same, Spectre v1 aims for conditional direct branches (the misspeculation happens on the decision "taken"/"not taken"), while Spectre v2 targets indirect branches, misspeculating on the target address of the branch.

The root cause of Retbleed is an attack against a Spectre v2 mitigation called retpoline. To avoid indirect branch poisoning without disabling indirect transient branches altogether[1], the Linux kernel replaced all indirect branches with return (`ret`) instructions. While also an indirect jump, an additional buffer called the Return Stack Buffer (RSB) takes priority over the Branch Target Buffer (BTB) for returns, mitigating the risks of out-of-place training. Retbleed breaks this mitigation by causing the RSB to overflow, forcing the CPU to fall back to the BTB.

[1]Why is this solution not feasible?

Exfiltrating the secret. Because the attacker cannot read the secret directly[2], they instead use the secret in a memory access, consequently leaving a secret-dependent trace in the cache. By loading just a single byte of the secret and using it as an index into the probe array, there is only 1 out of 256 possible entries of the probe array that can have been transiently accessed. Multiplying the secret by 4096 before using it ensures that each possible access falls into a separate page, which avoids prefetching. Hence, the attacker can exfiltrate the transiently accessed secret using a Flush+Reload procedure:

1. Flush all entries of the probe array `buff`.
2. Trigger the transient execution.
3. Reload and time all entries of `buff`

The result of this procedure may lead to the results shown in Figure 1. The entry that is hot after the transient execution represents the accessed secret.

The last two instructions of Listing 1 and 2 use the transiently accessed secret in a memory access to a *probe array*.

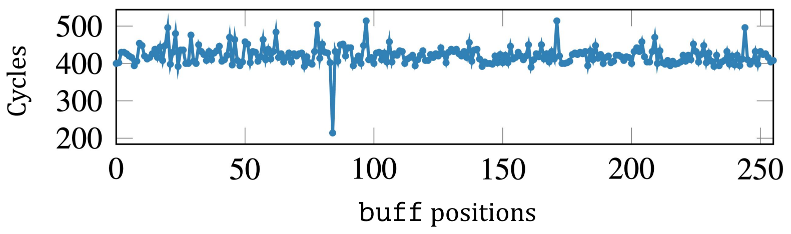

Figure 1: Time measurements of probe array with *hot* array entry at the lowest number of cycles.

Week 1 – Meltdown and Spectre v1

In this first part of the assignment, you will leak a secret from the L1d using a transient, illegal access. While you could leak the secret directly, you will not be able to make any use of it — once the CPU detects that the access was illegal, it will raise a Page Fault (#PF) and Linux will signal a Segmentation Fault (SIGSEGV) to your user process. You will therefore explore a set of possible suppression mechanisms for the #PF, allowing you to probe the reload buffer and infer the secret.

The first step is to use the naive method of handling the #PF by registering a SIGSEGV handler (**Task 1**). You will then use Intel's Transactional Synchronization Extensions (TSX) which provides an isolated memory transaction that never raises a #PF (**Task 2**). The tasks can be carried out in any preferred order: TSX is not necessarily harder than SEGV, nor does it depend on it in any particular way. Lastly, you will use speculative execution to avoid a #PF like when using TSX, but without relying on specific ISA extensions (**Task 3**). Namely, you will use Spectre to enter a transient window before you access the secret through Meltdown.

Your victim. The assignment ships a kernel module `WOM` for creating "Write-only Memory", which loads some data into privileged memory. WOM has been pre-installed on all our machines. You can write data into it by writing to `/dev/wom`. You can also ask the kernel to access the data and return a pointer to it via the `ioctl` system call. The source code of it is provided for reference in the assignment template. **Your task is to leak the data from privileged memory with your Meltdown exploit.**

[2] See Appendix A.

Task 1: Meltdown with suppressed SEGV handling

To prevent illegal memory accesses from crashing your program, you can register a signal handler for SEGV. To do this, it may help to read the manual pages for `sigaction(2)` and `setjmp(3)`.

Task 2: Meltdown within a TSX

Instead of handling the #PF, would it not be better not to trigger it in the first place? Intel Skylake-like processors, like the one you will use, support the RTM interface to TSX, which includes the `XBEGIN` and `XEND` instructions that indicate the start and end of a transaction, respectively. If a #PF occurs within the transaction, it will not be handled, but execution will continue from the label you specified by the `XBEGIN` instruction. More information about TSX is available in the Intel Software Developer Manual (SDM), Volume 1, Chapter 16 [4].

Task 3: Meltdown within a conditional branch misprediction

Intel TSX is often disabled on systems for security reasons. In this part, we will use a conditional branch misprediction (just like in Spectre V1) to suppress the #PF of the Meltdown kernel memory access. Instead of suppressing the #PF using TSX, trigger the secret load in a transient window caused by a mispredicted conditional branch. Modern branch predictors are quite smart, and the Skylake branch predictor may consider the outcome of the last 29 taken branches when predicting the next outcome [5].

Week 2 – Retbleed

In this week, you will leak secret kernel memory using Retbleed [10]. This assignment is designed to give you a hands-on experience with the Retbleed attack without requiring you to reverse-engineer the kernel to find vulnerable components yourself. Instead, we will provide you with an "oracle" module that contains a leaking gadget and provides you with some information to debug your exploit.

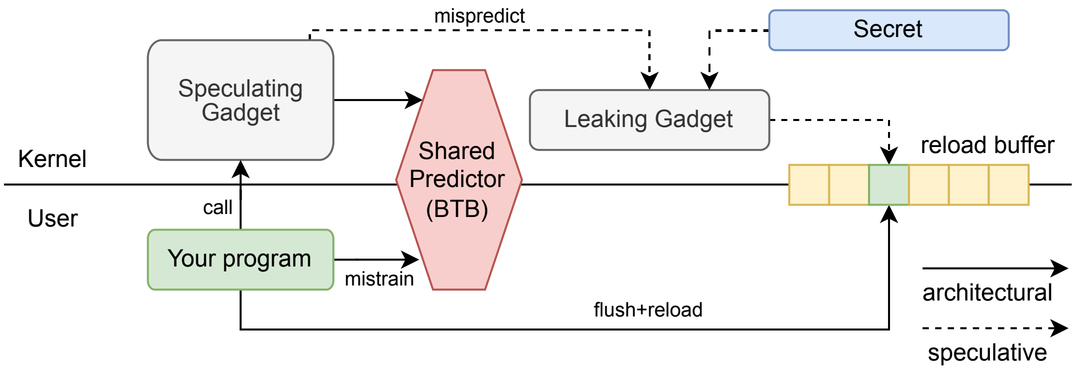

Figure 2: Assignment overview. You will create a BTB collision to cause speculative execution of a leaking gadget, which will leave traces in the reload buffer.

In Figure 2, we provide an overview of the assignment, which we explain in more detail next.

Preparing the reload buffer (Task 4). To enable a linear access pattern, you should allocate a contiguous memory area large enough to act as a Flush+Reload-buffer for a full byte value. Once you have set up the memory, you should find the details of your attack target, i.e., the jump target and return addresses. You can test your understanding and testing setup by calling the leaking function architecturally.

Create a BTB collision to leak the secret (Task 5). Once you have analyzed the parameters, you should be able to create BTB collisions that steer a function execution to your speculative path. As discussed earlier, Retbleed is only possible if you create a BTB collision through a hash collision *and* the RSB overflows. You will attack a special type of function: a recursive call. This simplifies the attack as the same function both overflows the RSB and creates a branch history that is hashed from a single call instruction. Your task is to create a function call that causes a hashing collision and trains the BTB in a way that causes a subsequent (honest) call to transiently jump to a function that reads from a secret and performs a secret-dependent access into the reload buffer. At last, if you have set up everything correctly, you should be able to leak the secret in a speculative execution path.

Task 4: Preparing the reload buffer

To create a working reload buffer, you need to allocate a large contiguous memory area. This is required to ensure that the offsets into the reload buffer are identical in both the kernel and user space (see Figure 3). To help you, our kernel module will throw an error if you try to initialize the secret with a non-hugepage reload buffer.

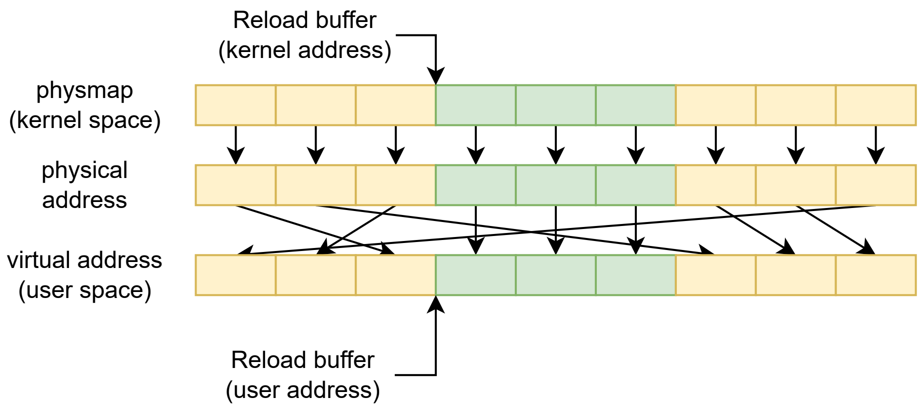

Figure 3: Physical memory layout of the kernel. The kernel maps all physical memory into the `physmap` linearly. By mapping a hugepage, you can ensure that the same linear mapping exists in user space, albeit with a different base address (green mapping).

Our exploitable module provides a disclosure gadget as shown in Listing 1, where both `my_reload_buffer` and `my_secret_address` are pointers under your control.

```c
void disclose(volatile uint8_t* my_reload_buffer, uint8_t* my_secret_address) {
    uint8_t value = *my_secret_address;
    *my_reload_buffer[value << 12];
}
```

Listing 1: The disclosure gadget as provided by our exploitable module.

The leaking gadget normally resides at a randomized address due to Kernel Address Space Layout Randomization (KASLR). In the Retbleed paper, KASLR first needs to be broken, but we spare you with this task and give you the gadget address directly. However, you cannot directly call the gadget[3]. Instead, you must cause another function in the module, which you can call directly, to mispredict and jump to the gadget speculatively.

[3]We give you a way to do this to aid in your debugging, but you will not receive points when you submit a solution using this method.

Hint: When you have reached this point, you can try to call the leakage function directly to test if your reload-buffer arrangement works. If you fail at this stage, you *will never* be able to leak speculatively. Read the `retbleed_oracle.h` file for more information.

Task 5: Create a BTB collision

After solving Task 4, you can start actually causing collisions in the Branch Target Buffer (BTB). To understand how, Horn et al. [3] provide a nice write-up on the BTB and BHB (Branch History Buffer). We extracted the key information for you in Appendix E, but give it a read!

As depicted in Figure 4, the BTB entries are determined from the page-offset of the branch source instruction and a history of prior calls.

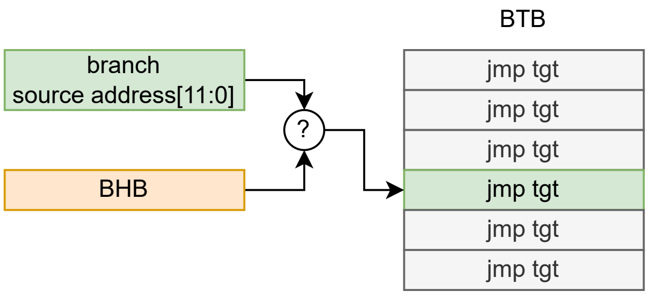

Figure 4: Addressing the BTB. The BTB is a cache that stores the target address of a branch instruction. By causing a collision and overloading higher-priority predictors, we can steer execution based on the BTB entry.

The Retbleed paper goes to great lengths to construct leakage channels by chaining jumps that cause BTB collisions. For this assignment, we target an edge case that makes the attack considerably easier. By providing a relatively deep recursive function (as depicted in Figure 5), it is sufficient for you to also create a recursive function that shares the same lower bits of the function and call addresses of the speculation gadget.

We are very clear about this: **You do *not* have to reverse-engineer the BHB function for this assignment. To begin with, the easiest collision is the one where all the bits included in the hash calculation match.**

The required addresses can be obtained from the kernel module using the function we provide to help you. For more clarity, we describe the meaning of the obtained values in Listing 2.

```
void speculation_gadget(
  uint8_t* reload_buffer /* you control this */,
  uint8_t* secret_ptr    /* you control this */,
  int repeat             /* = 29 */) {
 if (repeat > 0) {
  // The speculation call address returns the address of the call instruction.
  speculation_gadget(reload_buffer, secret_ptr, repeat - 1);
 }
 asm ("ret");   // The speculation return address
}
```

Listing 2: The speculation gadget as implemented in our exploitable module.

Note that this is a simplified depiction. To increase your chances of leaking, we stall execution of the speculation gadget by flushing the stack and inserting sufficient `NOPs` to ensure your speculation window is large enough.

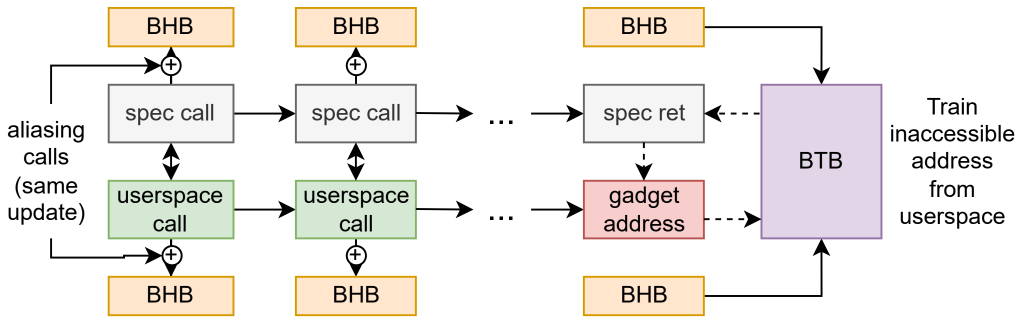
BHB
spec call
userspace call
aliasing calls (same update)
spec ret
gadget address
BTB
Train inaccessible address from userspace
...

Figure 5: Call chain. The chain of function calls that you need to create a BTB collision. Note the relation of calls that you need to obtain to ensure that the BHB matches.

Once you have created the BTB collision, you should be able to leak the secret. As you have noticed, to predict the gadget address in the kernel, you must call the kernel address from user space. This will cause a segfault, but still train the BTB. To avoid this crash, you should handle segmentation faults in your program. You should already have done this for the previous tasks, and you may use any facility that you have seen there — for example, a simple segfault handler.

Note that in our speculation gadget, we recurse. This already fills the RSB (16 entries on Intel Coffee Lake), causing an overflow and a fallback to the BTB.

Hint: To create non-predictable call chains reliably, you can push the call targets to the stack and use `ret` to go through the queue. You also may find ROP-style programming guides helpful.

Hint: If you want to start with a fixed address, you may set a corresponding flag in the code to disable randomization.

Deliverables for Code Submission

- **(Week 1)** The `grade.sh` script should run each variant 10 times and show that:
  1. All variants finish with a median time of less than 1s.
  2. All variants have an accuracy over 95%.
- **(Week 2)** The `grade.sh` script should successfully run your exploit at least 10 out of 12 times.
- Include a plot in `/figures/meltdown.pdf` that shows the accuracy over time to leak the secret for the SEGV, TSX, and Spectre variants. The purpose is to show the interplay between side-channel bandwidth and accuracy for the different variants.
- No build with `make` or `grade.sh` will result in a failing grade.

Grading Scheme

The grading is based on the following guidelines:

- 4.5 → Task 1 + Task 2
- 5.0 → Task 1 + Task 2 + Task 3
- 6.0 → Task 1 + Task 2 + Task 3 + Task 4 + Task 5

Presentation Submission

You must present this assignment in a 5-minute presentation on 29 October 2024. For this, you should prepare slides that detail the assignment and your steps taken to solve it. Upload it to Moodle at the "Assignment X | Presentation Submission" form. All team members should be able to present the slides and answer questions about the assignment, the lecture, and the papers we presented in the lecture.

Appendices

A. Meltdown

To efficiently handle OS interactions like system calls and interrupts, kernel memory is commonly mapped in the address space of a user process. The kernel memory is marked inaccessible from the process by clearing the `_PAGE_USER` bit in its respective PTEs. Due to the deferred check of this bit, however, Meltdown attacks transiently access this memory. This allows for leaking secrets like browser data, passwords, SSH keys, or anything in physical memory that can be brought into L1d. After the check, the incorrectly executed instructions are rolled back. However, the instructions still impose side effects on the microarchitecture. For example, transient secret-dependent memory operations leave observable traces in the caches despite being rolled back.

Directly accessing kernel memory from user mode results in a Page Fault (#PF), which will signal a Segmentation Fault (SIGSEGV) to your user process, typically causing it to exit with an error status. In this assignment, you will explore three different methods to suppress the #PF from the illegal memory access so that you can infer what memory was illegally accessed.

Listing 1 shows what a Meltdown snippet may look like. On line 3, the secret is accessed, which triggers a #PF. However, lines 4 and 5 will still transiently execute. This allows the attacker to encode the secret in the probe array before the instructions are rolled back.

```
1     ; rcx = kernel/secret address
2     ; rbx = probe array
3     movzx al, byte ptr [rcx]    ; secret access causing a #PF
4     shl rax, 12                 ; transiently executed: left-shift %rax by 12
5     mov rbx, qword [rbx + rax]  ; transiently executed: secret-dependent access in probe array
```

Listing 1: Meltdown. The load on line 3 will raise a fault when loading the secret.

B. Spectre v1

Spectre Variant 1 exploits conditional branches, so the dependency necessary to resolve is the branch condition. This condition may in turn depend on a slow memory operation, causing a long-running transient execution window of the wrong control-flow path upon a misprediction. Specifically, the condition used in many Spectre V1 PoCs is a bounds check, where a mispredicted bounds check may result in out-of-bounds memory accesses, allowing an attacker to leak secrets.

```
1 ; rax = out-of-bounds index ; rbx = points to buf size
2 ; rsi = victim array ; rdi = probe array
3 cmp rax, [rbx]
4 jg out_of_bounds       ; mispredicted as non-taken
5 movzx dl, [rsi + rax]  ; transiently executed, %dl is LSB of %rdi
6 shl rdx, 12            ; transiently executed
7 mov rdx, [rdx + rdi] . ; transiently executed: secret-dependent access in probe array %rdi
```

Listing 2: Spectre Variant 1 (V1).

Listing 2 demonstrates a possible snippet vulnerable to Spectre Variant 1. On line 4, the conditional branch is mispredicted as non-taken, resulting in fall-though behavior to execute lines 5–7 transiently. If `rax` provides an index that is not within the valid range, line 5 will load an arbitrary (potentially secret) value from memory. Similar to Listing 1, the secret value is encoded in the probe array (`rdi`) so that the attacker can infer its value. Once the memory load of the pointer in `rbx` is resolved, the frontend resteers to the correct path.

C. Retbleed

We discuss Spectre v1 in the first part of the assignment, which exploits an internal structure called the Pattern History Table (PHT). Simply put, when reaching a conditional branch, the CPU will predict the outcome based on the history of a few previous branches in a binary fashion: Taken or Not Taken. However, indirect branches are not binary, but allow access to any address in memory. Listing 3 shows the differences between a direct and indirect branch.

```
; Note: Intel syntax (mnemonic destination, source)
; rax = function pointer ; rbx = pointer table ; rcx = upper bound
cmp rax, rcx  ; check rax bounds
jl .Lerror    ; Either take or don't (binary prediction)

mov [rbx], rax      ; store any value to the table
; Do something else
call QWORD PTR[rbx] ; indirect call to the table - where are we going?
```

Listing 3: Comparing direct and indirect branches.

Because indirect branches happen frequently, x86 CPUs have a special structure called the Branch Target Buffer (BTB) to predict the target of indirect branches. The BTB is indexed by a hash of the history of prior branches, and the target address is stored in the BTB. The assumption is the same as with Spectre v1: If the code took a similar path as before, it's likely that it will continue to follow the same path. Now, being able to change either the BTB or creating a collision of address hashes allows an attacker to steer the transient execution to a controlled gadget. The term for this is *Branch Target Injection (BTI)* [6].

The kernel then sought to mitigate the issue of indirect branching by using a special type of branch: `ret`. Remember that `ret` is essentially a `pop rip` (which does not exist) and would normally be emulated with `pop rcx; jmp *rcx` - so it's an indirect branch as well. However, with function calls being used frequently, a special additional stack buffer was introduced earlier, the Return Stack Buffer (RSB) [9]. The idea is that `calls` and `rets` would additionally push and pop into the return stack buffer, respectively — a buffer that does not get polluted by data, as the normal stack does. Because the RSB is checked before the BTB, it prevents indirect call poisoning. At least, this was the rationale behind a retpoline as shown in Listing 4.

```
; Code aimed to replace: jmp rax

; retpoline code
call .Lload_label          ; push .Lcapture_speculation to the RSB, continue to .Lload_label
.Lcapture_speculation:
pause                      ; efficient lfence, essentially: Drain the memory queue
jmp .Lcapture_speculation ; Loop: Transient execution gets stuck here
.Lload_label:
mov [rsp], rax             ; Save the target address to the stack (overwrite the return address)
ret                        ; Do an indirect jump. If we were speculating, the RSB entry
                           ; created at the first line would be used and we would return
                           ; to .Lcapture_speculation, which forces speculation to stop.
```

Listing 4: Retpoline implementation.

Of course, there is an issue with this: The RSB structure is limited in size. If the RSB overflows, the CPU will fall back to the BTB, as shown in Figure 6 [2]. Abusing this overflow is Retbleed in a nutshell.

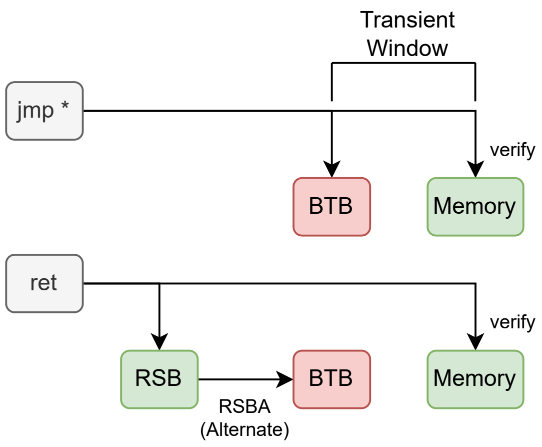

Figure 6: RSB-to-BTB Fallback. The RSB is usually a good predictor for indirect branches and takes priority over the BTB for returns. However, when the RSB overflows, the CPU will fall back to the BTB, enabling BTI attacks.

D. Kernel-to-user accesses

For security reasons, the kernel will not directly access user-space addresses. This is due to a security feature called Supervisor Mode Access Prevention (SMAP). By default (and historically), a higher privilege levels have full access to lower privilege levels. As a result, exploits would sometimes aim to trick the kernel into accessing attacker-controller user-space memory rather than safe kernel memory. With SMAP, accesses will fail if the kernel tries to access user-space memory. Only in few, explicitly allowed cases, the kernel will set a flag to disable SMAP.

Physmap. Because the SMAP flag is set in `CR4.AC` and serializes memory operations, we cannot leak the secret directly to user space - not even speculatively. However, we can achieve similar behavior by exploiting another kernel facility: the `physmap`. For ease of maintenance, the kernel always maps all physical memory linearly into the `physmap` at a constant address — randomized at boot time through KASLR. Assuming you know the address of the physmap in the kernel, you could steer the reload buffer to this address and leak the secret from this linear mapping. Because the memory allocator may provide non-contiguous pages, the easiest way to ensure that the same linear mapping exists in user space (so you can flush and time) is to allocate a hugepage as we show in Figure 3.

E. BTB and BHB

Horn et al. [3] provide the following function as pseudocode for their reverse-engineered history function on Intel Haswell CPUs:

```c
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);
}
```

Note the following:

- The BHB state depends *both* on the destination and the source address.
- There is a limited number of bits that are used to update the state. Namely, bit 19 of the branch source is the highest bit used.
- The BHB holds 29 entries (2 bits per jump, 58 bits of state).

Note as well that the Current Privilege Level (CPL) is not part of this calculation, and the kernel bits (kernel memory starts at `0xffff800000000000` for 4-level pagetables on amd64[4]) are also absent. As a result, given the same address with a given alignment (*hint hint*), the BTB will predict the same target address between kernel and user space. This function may have changed with more recent CPUs and differs between vendors.

References

[1] Claudio Canella et al. "A Systematic Evaluation of Transient Execution Attacks and Defenses." In: *USENIX Security Symposium.* 2019, pp. 249–266.

[2] Andrew Cooper. "Re: [PATCH 0/7] IBRS patch series". In: (). URL: \url{https://lkml.org/lkml/2018/1/4/724}.

[3] J Horn. "Reading privileged memory with a side-channel". In: (). URL: \url{https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html}.

[4] *Intel® 64 and IA-32 Architectures Software Developer Manuals.* URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html (visited on 05/21/2023).

[5] Paul Kocher et al. "Spectre attacks: Exploiting speculative execution". In: *Communications of the ACM* 63.7 (2020), pp. 93–101.

[6] Paul Kocher et al. "Spectre attacks: Exploiting speculative execution". In: *Communications of the ACM* 63.7 (2020), pp. 93–101.

[7] Michael Larabel. "Linux 5.0 Spectre/Meltdown Benchmarks". In: (). URL: \url{https://www.phoronix.com/review/linux50-spectre-meltdown}.

[8] Moritz Lipp et al. "Meltdown". In: *arXiv preprint arXiv:1801.01207* (2018).

[9] Design-Scalable Microarchitecture. "First the Tick, Now the Tock: Intel® Microarchitecture (Nehalem)". In: ().

[10] Johannes Wikner and Kaveh Razavi. "Retbleed: Arbitrary Speculative Code Execution with Return Instructions". In: *USENIX Security.* Intel Bounty Reward, CSAW Europe finalist. Aug. 2022. URL: Paper=https://comsec.ethz.ch/wp-content/files/retbleed_sec22.pdfURL=https://comsec.ethz.ch/research/microarch/retbleed.

[11] Johannes Wikner, Daniël Trujillo, and Kaveh Razavi. "Phantom: Exploiting Decoder-detectable Mispredictions". In: *MICRO.* Oct. 2023. URL: Paper=https://comsec.ethz.ch/wp-content/files/phantom_micro23.pdfURL=https://comsec.ethz.ch/research/microarch/inception.

[4] https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt