BPF and Spectre: Mitigating transient execution attacks

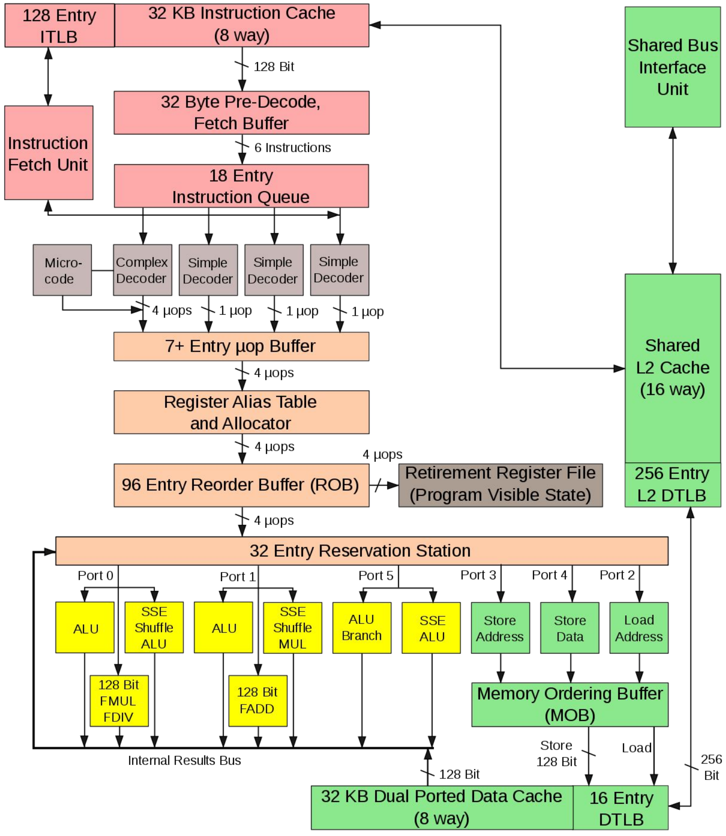
Daniel Borkmann (Isovalent, co-maintainer BPF)

Microarchitecture

µarch is the way a given ISA like x86 is implemented

- ➔ Can vary due to different optimization goals or technology shifts
- ➔ µarchitectural concepts include:

Branch prediction
Out-of-order execution
Speculative execution



Intel Core 2 Architecture

Microarchitecture

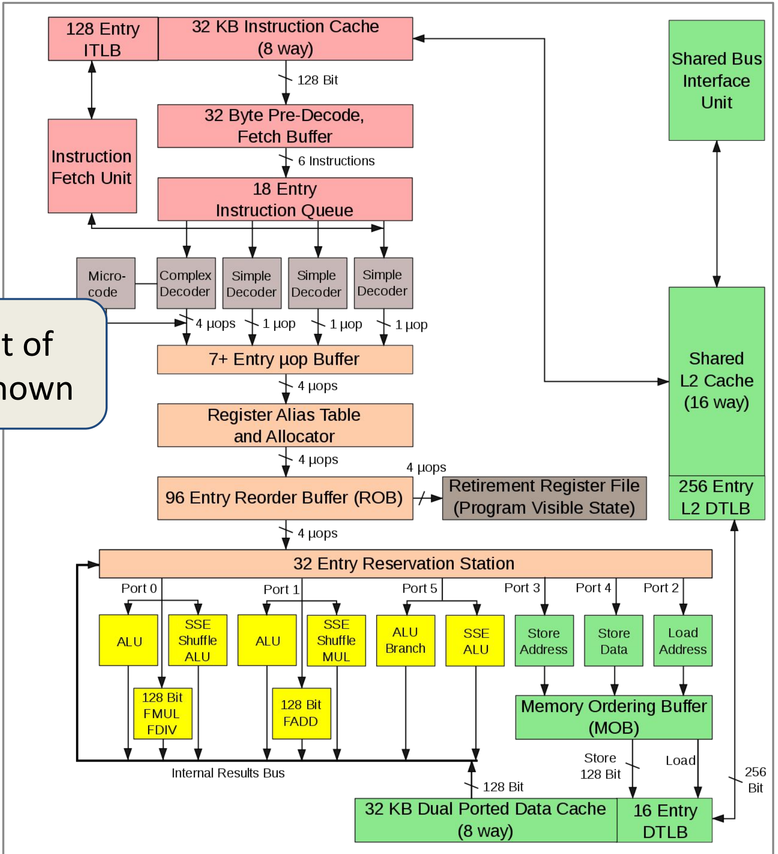
µarch is the way a given ISA like x86 is implemented

➔ Can vary due to different optimization goals or technology shifts

➔ µarchitectural concepts include:

Branch prediction
Out-of-order execution
Speculative execution

Predicts outcome and target of branches before they are known



Intel Core 2 Architecture

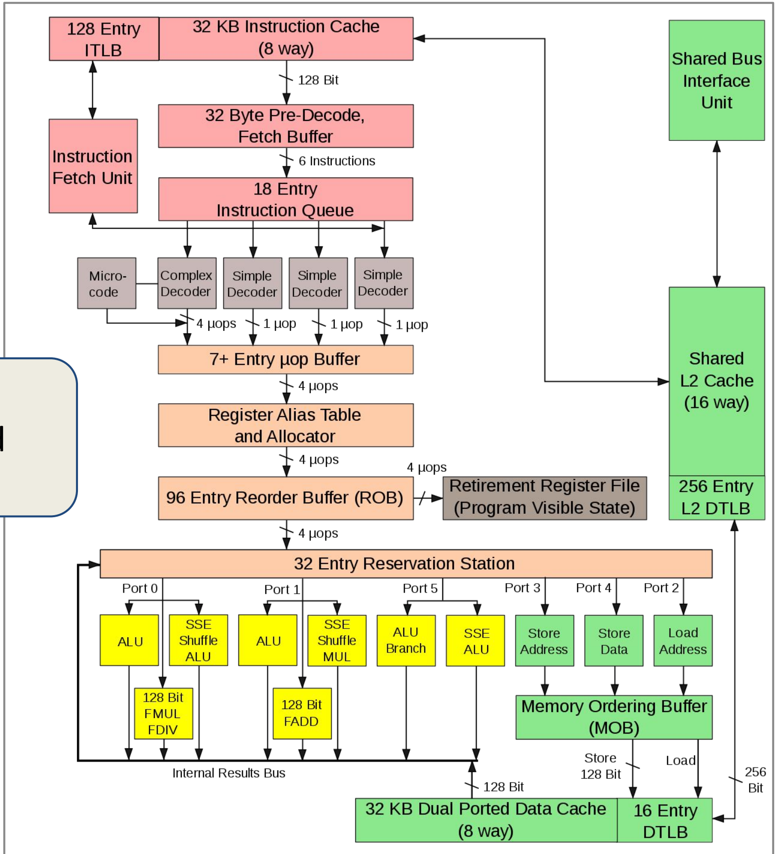
Microarchitecture

µarch is the way a given ISA like x86 is implemented

- ➔ Can vary due to different optimization goals or technology shifts
- ➔ µarchitectural concepts include:

Branch prediction
Out-of-order execution
Speculative execution

Avoids pipeline stalls due to waiting on data being fetched from memory



Intel Core 2 Architecture

Microarchitecture

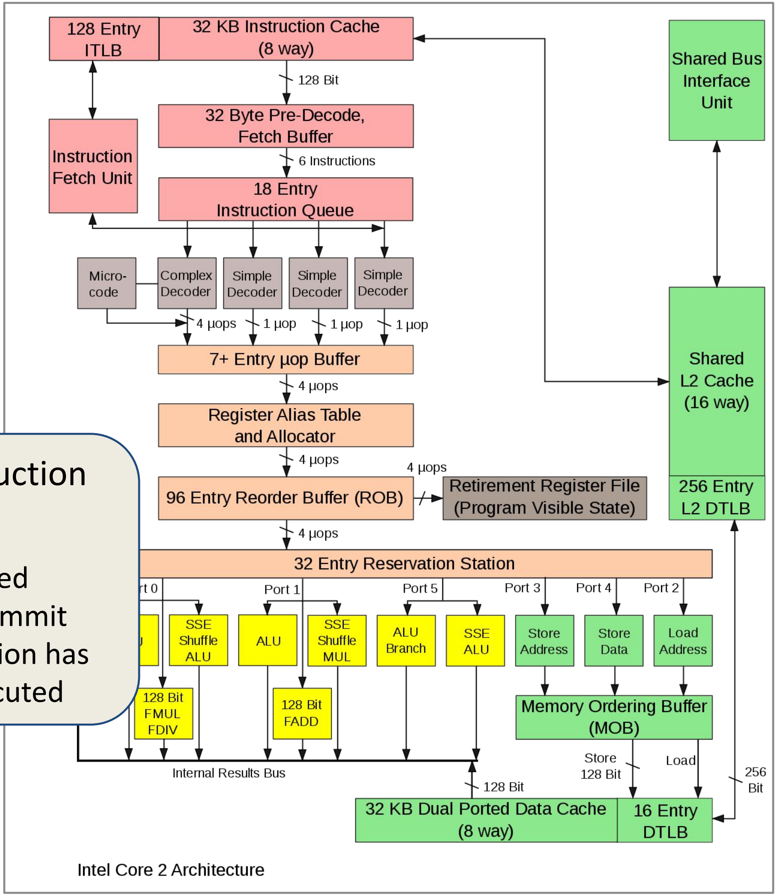
µarch is the way a given ISA like x86 is implemented

- ➔ Can vary due to different optimization goals or technology shifts
- ➔ µarchitectural concepts include:

Branch prediction
Out-of-order execution
Speculative execution

Continues execution of instruction with predicted outcome.

- ➔ **If prediction true:** predicted execution is allowed to commit
- ➔ **If prediction false:** execution has to be unrolled and re-executed



Microarchitecture

µarch is the way a given ISA like x86 is implemented

➔ Can vary due to different optimization goals or technology shifts

➔ µarchitectural concepts include:

Branch prediction
Out-of-order execution
Speculative execution

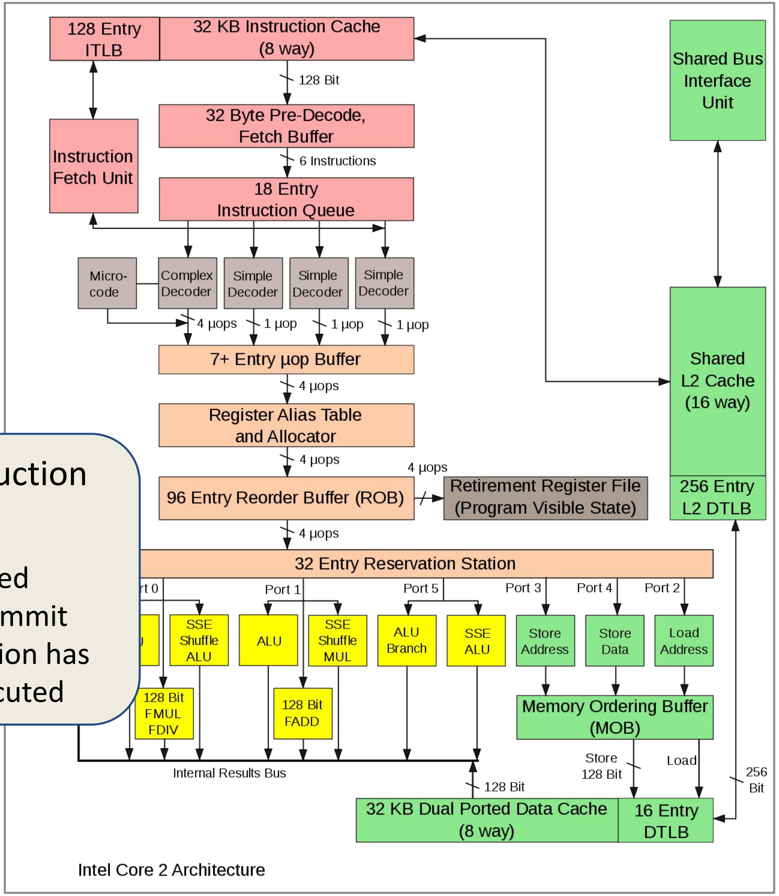
Continues execution of instruction with predicted outcome.

➔ **If prediction true:** predicted execution is allowed to commit
➔ **If prediction false:** execution has to be unrolled and re-executed

"Transient instructions"

Rollback on misspeculation:

➔ Old register states preserved → restored
➔ Memory writes are buffered → discarded
➔ Cache modifications → **not restored**



Microarchitecture

µarch is the way a given ISA like x86 is implemented

➔ Can vary due to different optimization goals or technology shifts

➔ µarchitectural concepts include:

Branch prediction
Out-of-order execution
Speculative execution

Continues execution of instruction with predicted outcome.

➔ **If prediction true:** predicted execution is allowed to commit

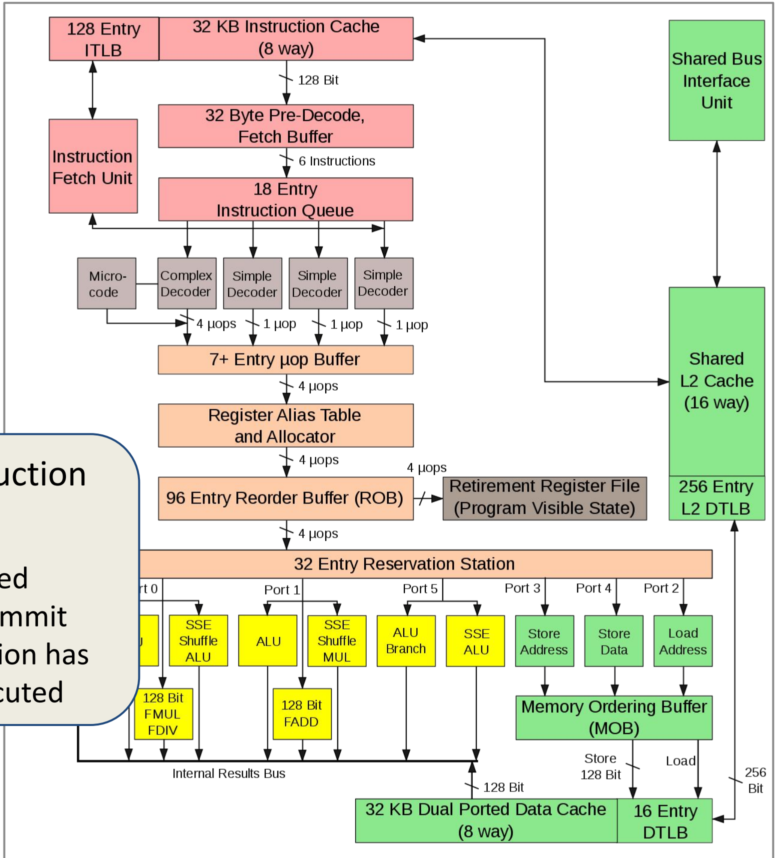
If prediction false: execution has to be unrolled and re-executed

"Transient instructions"

Rollback on misspeculation:

➔ Old register states preserved → restored
➔ Memory writes are buffered → discarded
➔ Cache modifications → **not restored**

observable side-effect!



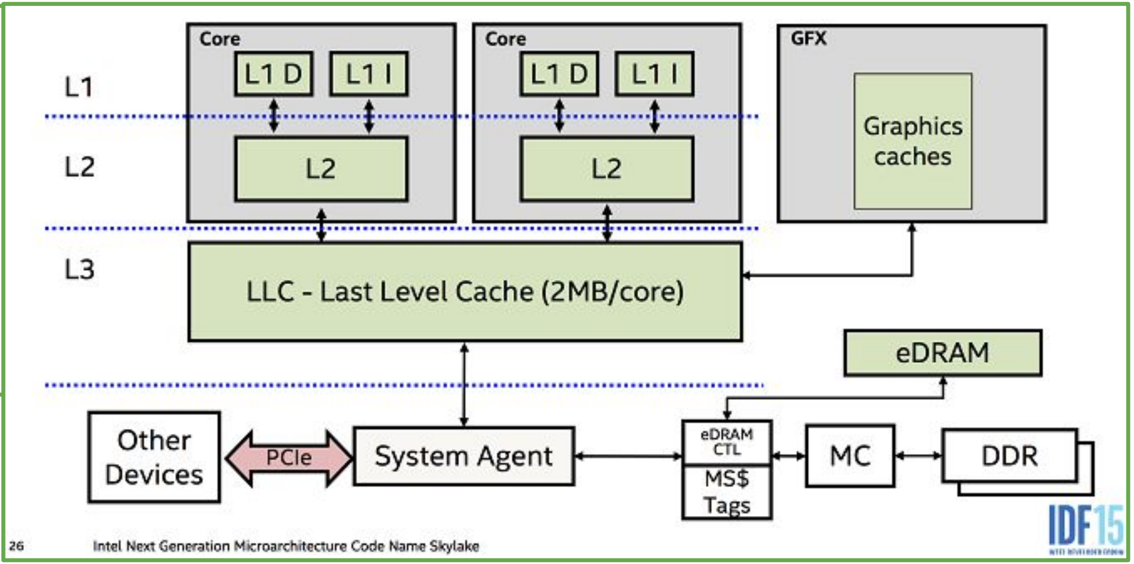
Intel Core 2 Architecture

Microarchitecture

Covertly leaking data from transient instructions: **caches as side-channels**

Short primer on caches:

- ➔ Provide faster access to frequently used data (the closer to the core, the less time required to load data)
- ➔ Difference in access time can be measured by software
- ➔ Possible to determine whether an address is cached



Microarchitecture

Covertly leaking data from transient instructions: **caches as side-channels**

Short primer on caches:

L1 data cache: 32KB, fastest latency: **4 cycles**

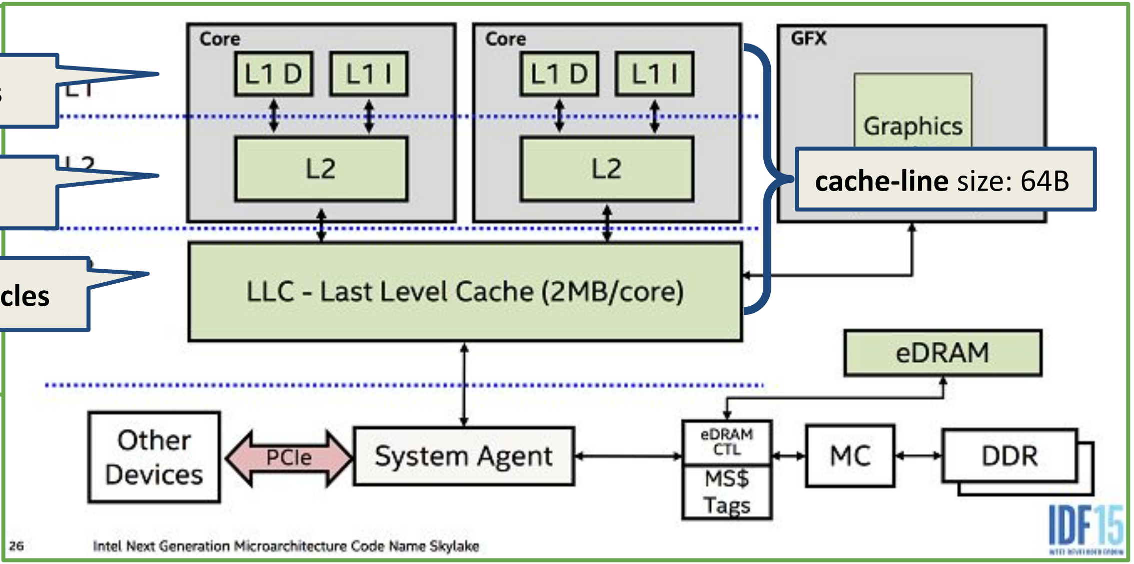
used data (the closer to the core, the

L2 cache: 256KB, fastest latency: **12 cycles**

Difference in access time can be

L3 cache: 8MB, shared, fastest latency: **38 cycles**

address is cached



Microarchitecture

Covertly leaking data from transient instructions: **caches as side-channels**

Short primer on caches:

used data (the closer to the core, the

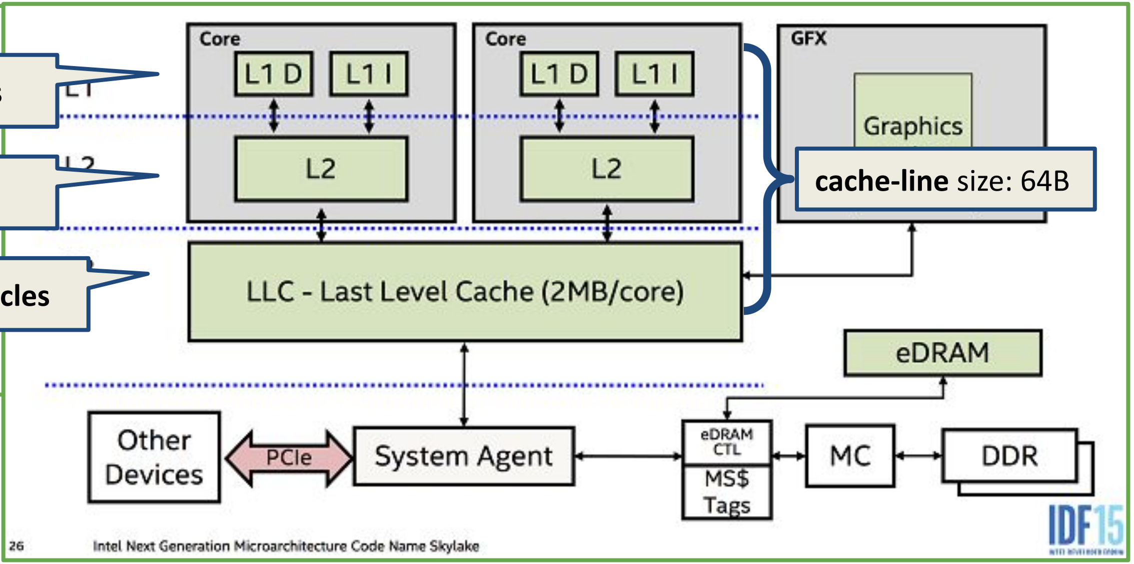
Difference in access time can be

address is cached

L1 data cache: 32KB, fastest latency: **4 cycles**

L2 cache: 256KB, fastest latency: **12 cycles**

L3 cache: 8MB, shared, fastest latency: **38 cycles**



cache-line size: 64B

1 Bit signal for side-channel:

```
time = rdtsc();
mem_access(&data[0x100]);
delta = rdtsc() - time;
```

time delta *low*: **in cache**
time delta *high*: **not in cache**

(can then be compared to *known* 'in cache'/'not in cache' timings)

Microarchitecture

Covertly leaking **example via BPF** (principal is same for different Spectre attacks):

'Leaker' BPF prog:

```
u8  value = *(u8 *)ptr;
u32 index = (((value >> bit) & 1) * 0x100) + 0x200;
mem_access(&map_value[index]);
```

Non-speculative domain: points to e.g. BPF map value
Under speculation: points to attacker controlled address

Examples shown later on how this can be triggered.

Microarchitecture

Covertly leaking **example via BPF** (principal is same for different Spectre attacks):

'Leaker' BPF prog:

```
u8  value = *(u8 *)ptr;
u32 index = (((value >> bit) & 1) * 0x100) + 0x200;
mem_access(&map_value[index]);
```

Shift to bit-position to **extract individual bits**

Microarchitecture

Covertly leaking **example via BPF** (principal is same for different Spectre attacks):

'Leaker' BPF prog:

```
u8  value = *(u8 *)ptr;
u32 index = (((value >> bit) & 1) * 0x100) + 0x200;
mem_access(&map_value[index]);
```

Resulting index becomes **either**:

➔ **0** * 0x100 + 0x200 = 0x200

➔ **1** * 0x100 + 0x200 = 0x300

Microarchitecture

Covertly leaking **example via BPF** (principal is same for different Spectre attacks):

'Leaker' BPF prog:

```
u8  value = *(u8 *)ptr;
u32 index = (((value >> bit) & 1) * 0x100) + 0x200;
mem_access(&map_value[index]);
```

Access address at valid BPF map:

➔ map_value[0x200]

➔ map_value[0x300]

Microarchitecture

Covertly leaking **example via BPF** (principal is same for different Spectre attacks):

'Leaker' BPF prog:

```
u8  value = *(u8 *)ptr;
u32 index = (((value >> bit) & 1) * 0x100) + 0x200;
mem_access(&map_value[index]);
```

Access address at valid BPF map:

➔ map_value[0x200]

➔ map_value[0x300]

'Reader' BPF prog:

```
time = ktime_get_ns();
mem_access(&map_value[index]);
delta = ktime_get_ns() - time;
// store delta in different BPF map
```

Microarchitecture

Covertly leaking **example via BPF** (principal is same for different Spectre attacks):

'Leaker' BPF prog:

```
u8  value = *(u8 *)ptr;
u32 index = (((value >> bit) & 1) * 0x100) + 0x200;
mem_access(&map_value[index]);
```

Access address at valid BPF map:

- ➔ map_value[0x200]
- ➔ map_value[0x300]

value (8 bits)

0	?	?	?	?	?	?	?

'Reader' BPF prog:

```
time = ktime_get_ns();
mem_access(&map_value[index]);
delta = ktime_get_ns() - time;
// store delta in different BPF map
```

Example timing:

- ➔ map_value[0x200]: *'in cache'*
- ➔ map_value[0x300]: *'not in cache'*

Microarchitecture

Covertly leaking **example via BPF** (principal is same for different Spectre attacks):

'Leaker' BPF prog:

```
u8  value = *(u8 *)ptr;
u32 index = (((value >> bit) & 1) * 0x100) + 0x200;
mem_access(&map_value[index]);
```

Access address at valid BPF map:

➔ map_value[0x200]
➔ map_value[0x300]

`value` (8 bits)

0	0	?	?	?	?	?	?

'Reader' BPF prog:

```
time = ktime_get_ns();
mem_access(&map_value[index]);
delta = ktime_get_ns() - time;
// store delta in different BPF map
```

Example timing:

➔ map_value[0x200]: *'in cache'*
➔ map_value[0x300]: *'not in cache'*

Microarchitecture

Covertly leaking **example via BPF** (principal is same for different Spectre attacks):

‘Leaker’ BPF prog:

```
u8  value = *(u8 *)ptr;
u32 index = (((value >> bit) & 1) * 0x100) + 0x200;
mem_access(&map_value[index]);
```

Access address at valid BPF map:

- ➔ map_value[0x200]
- ➔ map_value[0x300]

`value` (8 bits)

0	0	1	?	?	?	?	?

‘Reader’ BPF prog:

```
time = ktime_get_ns();
mem_access(&map_value[index]);
delta = ktime_get_ns() - time;
// store delta in different BPF map
```

Example timing:

- ➔ map_value[0x200]: *‘not in cache’*
- ➔ map_value[0x300]: *‘in cache’*

(between runs needs to bounce cachelines, so both slots are ‘not in cache’)

Microarchitecture & Spectre

Generally any runtime affected, not just BPF, given these are **hardware bugs**

➔ Not triggered by software bugs whatsoever
➔ Execution without speculation is safe

Spectre: injecting misspeculation to then covertly leak data via side-channel

➔ Different attacks to trigger misspeculation

Microarchitecture & Spectre

Generally any runtime affected, not just BPF, given these are **hardware bugs**

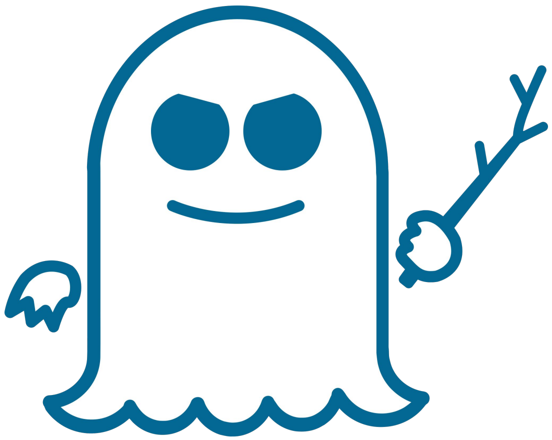
- ➔ Not triggered by software bugs whatsoever
- ➔ Execution without speculation is safe

Spectre: injecting misspeculation to then covertly leak data via side-channel

- ➔ Different attacks to trigger misspeculation

Example **attacks and mitigations** shown for **BPF runtime**

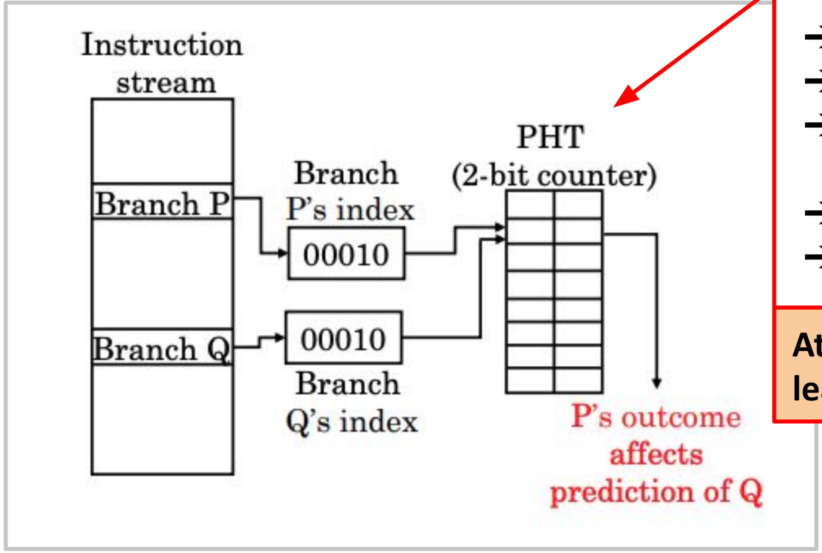
- ➔ **Disclaimer:** not able to cover every aspect due to time limit
- ➔ Focus on Spectre v1/v2/v4
- ➔ Relation to process capabilities



BPF & Spectre v1

Bounds Check Bypass to gain memory out-of-bounds access under speculation

➔ CPU reduces perf penalty by predicting outcome of branches



(Image from https://arxiv.org/pdf/1804.00261.pdf)

- ➔ Typically implemented as Pattern History Table (PHT)
- ➔ Predicts outcome of branch taken/not taken
- ➔ Expected to be sometimes wrong (example: induction variable in loop)
- ➔ PC indexed via partial virtual address (e.g. lower bits)
- ➔ Subject to aliasing/interference, see P versus Q

Attack injects misspeculation to array bounds check. Then leaks data when out-of-bounds.

```
if (x < *array_max) {
    val = array[x];
    // leak val (as shown earlier)
}
```

array_max not in cache
(condition mispredicted to be in-bounds)

BPF & Spectre v1

Example attack in BPF, 1: load slowly-loaded value and turn into constant

```
// r0 = pointer to a map array entry
// r7 = pointer to a map array entry
r4 = r10
r4 += -1
*(u8 *)(r4 -511) = 0
r2 = *(u64 *)(r7 +8)
r3 = *(u64 *)(r0 +4608)
r3 &= 1
r3 &= 2
r3 -= 511
if r2 != r3 goto pc+7
r4 += r2
r4 = *(u8 *)(r4 +0)
// leak r4
```

Inits stack slot to 0

Loads OOB address

Loads uncached bound and turns it into a slow constant

Mispredicts condition and loads OOB address

Without CPU speculation: Reads back stack slot

BPF & Spectre v1

Example attack in BPF, 1: load slowly-loaded value and turn into constant

```
// r0 = pointer to a map array entry
// r7 = pointer to a map array entry
r4 = r10
r4 += -1
*(u8 *)(r4 -511) = 0
r2 = *(u64 *)(r7 +8)
r3 = *(u64 *)(r0 +4608)
r3 &= 1
r3 &= 2
r3 -= 511
if r2 != r3 goto pc+7
r4 += r2
r4 = *(u8 *)(r4 +0)
// leak r4
```

Inits stack slot to 0

Loads OOB address

Loads uncached bound and turns it into a slow constant

Mispredicts condition and loads OOB address

Nothing to speculate anymore! (r11 is hidden auxiliary register)

Mitigated by BPF verifier:

```
// r0 = pointer to a map array entry
// r7 = pointer to a map array entry
r4 = r10
r4 += -1
*(u8 *)(r4 -511) = 0
r2 = *(u64 *)(r7 +8)
r3 = *(u64 *)(r0 +4608)
r3 &= 1
r3 &= 2
r3 -= 511
if r2 != r3 goto pc+7
w11 = 511
r4 -= r11
r4 = *(u8 *)(r4 +0)
// leak r4
```

observed that r3 is constant!

rewrites pointer arithmetic

BPF & Spectre v1

Two mitigation approaches performed by BPF verifier

- ➔ **Eliminate speculation** if possible by rewrite with constants
- ➔ Safely **redirect speculation** to be within array bounds

What if offset is not known?

```
// r2 = unknown but in [0,32]
r4 += r2
r4 = *(u8 *)(r4 +0)
// leak r4
```

Redirected speculation:

```
// r2 = unknown but in [0,32]
w11 = 32
r11 -= r2
r11 |= r2
r11 = -r11
r11 s>>= 63
r11 &= r2
r4 += r11
r4 = *(u8 *)(r4 +0)
// leak r4
```

BPF & Spectre v1

Two mitigation approaches performed by BPF verifier

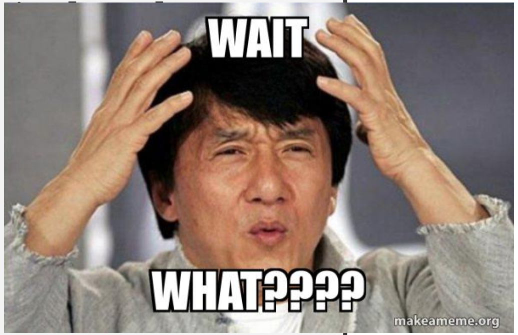
➔ **Eliminate speculation** if possible by rewrite with constants
➔ Safely **redirect speculation** to be within array bounds

What if offset is not known?

```
// r2 = unknown but in [0,32]
r4 += r2
r4 = *(u8 *)(r4 +0)
// leak r4
```

Redirected speculation:

```
// r2 = unknown but
w11 = 32
r11 -= r2
r11 |= r2
r11 = -r11
r11 s>>= 63
r11 &= r2
r4 += r11
r4 = *(u8 *)(r4 +0)
// leak r4
```



BPF & Spectre v1

Two mitigation approaches performed by BPF verifier

- ➔ **Eliminate speculation** if possible by rewrite with constants
- ➔ Safely **redirect speculation** to be within array bounds

offset is “in-bounds”

Redirected speculation:

```
// r2 = unknown but in [0,32]
w11 = 32
r11 -= r2
r11 |= r2
r11 = -r11
r11 s>>= 63
r11 &= r2
r4 += r11
r4 = *(u8 *)(r4 +0)
// leak r4
```

Example	**r2 speculation: 31 (0x1F)** **max value: 32 (0x20)**	**r2 speculation: 34 (0x22)** **max value: 32 (0x20)**
w11 = 32	0000000000000020	
r11 -= r2	0000000000000001	
r11 |= r2	000000000000001f	
r11 = -r11	ffffffffffffffe1	
r11 s>>= 63	ffffffffffffffff	
r11 &= r2	000000000000001f	
r4 += r11	➔ **r4 += 31**	

BPF & Spectre v1

Two mitigation approaches performed by BPF verifier

- ➔ **Eliminate speculation** if possible by rewrite with constants
- ➔ Safely **redirect speculation** to be within array bounds

offset is "in-bounds"

Redirected speculation:

```
// r2 = unknown but in [0,32]
w11 = 32
r11 -= r2
r11 |= r2
r11 = -r11
r11 s>>= 63
r11 &= r2
r4 += r11
r4 = *(u8 *)(r4 +0)
// leak r4
```

Example	**r2 speculation: 31 (0x1F)** **max value: 32 (0x20)**	**r2 speculation: 34 (0x22)** **max value: 32 (0x20)**
`w11 = 32`	0000000000000020	
`r11 -= r2`	0000000000000001	
`r11 |= r2`	000000000000001f	
`r11 = -r11`	ffffffffffffffe1	
`r11 s>>= 63`	ffffffffffffffff	
`r11 &= r2`	000000000000001f	
`r4 += r11`	➔ r4 += 31	

BPF & Spectre v1

Two mitigation approaches performed by BPF verifier

- ➔ **Eliminate speculation** if possible by rewrite with constants
- ➔ Safely **redirect speculation** to be within array bounds

offset is “in-bounds”

offset is “out-of-bounds”

Redirected speculation:

```
// r2 = unknown but in [0,32]
w11 = 32
r11 -= r2
r11 |= r2
r11 = -r11
r11 s>>= 63
r11 &= r2
r4 += r11
r4 = *(u8 *)(r4 +0)
// leak r4
```

Example	**r2 speculation: 31 (0x1F)** **max value: 32 (0x20)**	**r2 speculation: 34 (0x22)** **max value: 32 (0x20)**
w11 = 32	0000000000000020	0000000000000020
r11 -= r2	0000000000000001	fffffffffffffffe
r11 |= r2	000000000000001f	fffffffffffffffe
r11 = -r11	ffffffffffffffe1	0000000000000002
r11 s>>= 63	ffffffffffffffff	0000000000000000
r11 &= r2	000000000000001f	0000000000000000
r4 += r11	➔ **r4 += 31**	➔ **r4 += 0**

BPF & Spectre v1

Two mitigation approaches performed by BPF verifier

➔ **Eliminate speculation** if possible by rewrite with constants

➔ Safely **redirect speculation** to be within array bounds

Redirected speculation:

```
// r2 = unknown but in [0,32]
w11 = 32
r11 -= r2
r11 |= r2
r11 = -r11
r11 s>>= 63
r11 &= r2
r4 += r11
r4 = *(u8 *)(r4 +0)
// leak r4
```

offset is “in-bounds”

offset is “out-of-bounds”

Example	**r2 speculation: 31 (0x1F)** **max value: 32 (0x20)**	**r2 speculation: 34 (0x22)** **max value: 32 (0x20)**
w11 = 32	0000000000000020	0000000000000020
r11 -= r2	0000000000000001	fffffffffffffffe
r11 |= r2	000000000000001f	fffffffffffffffe
r11 = -r11	ffffffffffffffe1	0000000000000002
r11 s>>= 63	ffffffffffffffff	0000000000000000
r11 &= r2	000000000000001f	0000000000000000
r4 += r11	➔ **r4 += 31**	➔ **r4 += 0**

BPF & Spectre v1

Two mitigation approaches performed by BPF verifier

- ➔ **Eliminate speculation** if possible by rewrite with constants
- ➔ Safely **redirect speculation** to be within array bounds

Redirected speculation:

```
// r2 = unknown but in [0,32]
w11 = 32
r11 -= r2
r11 |= r2
r11 = -r11
r11 s>>= 63
r11 &= r2
r4 += r11
r4 = *(u8 *)(r4 +0)
// leak r4
```

offset is "in-bounds"

offset is "out-of-bounds"

Example	**r2 speculation: 31 (0x1F)** **max value: 32 (0x20)**	**r2 speculation: 34 (0x22)** **max value: 32 (0x20)**
w11 = 32	0000000000000020	0000000000000020
r11 -= r2	0000000000000001	fffffffffffffffe
r11 |= r2	000000000000001f	fffffffffffffffe
r11 = -r11	ffffffffffffffe1	0000000000000002
r11 s>>= 63	ffffffffffffffff	0000000000000000
r11 &= r2		0000000000000000
r4 += r11	➔ **r4 += 31**	➔ **r4 += 0**

Speculation is "redirected" branchless to be "in-bounds"

BPF & Spectre v1

Two mitigation approaches performed by BPF verifier

- ➔ **Eliminate speculation** if possible by rewrite with constants
- ➔ Safely **redirect speculation** to be within array bounds

What if offset is not known?

```
// r2 = unknown but in [0,32]
r4 += r2
r4 = *(u8 *)(r4 +0)
// leak r4
```

Steps done by BPF verifier:

- ➔ Observes pointer move, derives max register offset/limit
- ➔ Spawns a new verification path to simulate program under truncation (**r4 += 0** case)
- ➔ Rewrites pointer arithmetic with masking

BPF & Spectre v1

Example attack in BPF, 2: pointer type confusion under speculation

Can BPF verifier conclude that this is safe?

```
// r0 = pointer to a map array entry
// r6 = pointer to readable stack slot
// r9 = scalar controlled by attacker

1: r0 = *(u64 *)(r0)
2: if r0 != 0x0 goto line 4
3: r6 = r9
4: if r0 != 0x1 goto line 6
5: r9 = *(u8 *)(r6)
6: // leak r9
```

Mutually exclusive paths

BPF & Spectre v1

Example attack in BPF, 2: pointer type confusion under speculation

Can BPF verifier conclude that this is safe?

```
// r0 = pointer to a map array entry
// r6 = pointer to readable stack slot
// r9 = scalar controlled by attacker

1: r0 = *(u64 *)(r0)
2: if r0 != 0x0 goto line 4
3: r6 = r9
4: if r0 != 0x1 goto line 6
5: r9 = *(u8 *)(r6)
6: // leak r9
```

cache-miss

No! Under misspeculation this can be executed:

```
// ...
// r6 = pointer to readable stack slot
// r9 = scalar controlled by attacker

1: ...
2: ...
3: r6 = r9
4: ...
5: r9 = *(u8 *)(r6)
6: // leak r9
```

BPF & Spectre v1

Example attack in BPF, 2: pointer type confusion under speculation

Can BPF verifier conclude that this is safe?

```
// r0 = pointer to a map array entry
// r6 = pointer to readable stack slot
// r9 = scalar controlled by attacker

1: r0 = *(u64 *)(r0)
2: if r0 != 0x0 goto line 4
3: r6 = r9
4: if r0 != 0x1 goto line 6
5: r9 = *(u8 *)(r6)
6: // leak r9
```

cache-miss

No! Under misspeculation this can be executed:

```
// ...
// r6 = pointer to readable stack slot
// r9 = scalar controlled by attacker

1: ...
2: ...
3: r6 = r9
4: ...
5: r9 = *(u8 *)(r6)
6: // leak r9
```

See earlier 'P versus Q' aliasing/interference:
Attacker trains branch predictor from user space at 'colliding' indices in PHT, both as: not taken

BPF & Spectre v1

Mitigation approach performed by BPF verifier

➔ **Verify 'impossible' paths for safety** that can be reached from speculation

No! Under misspeculation this can be executed:

```
// ...
// r6 = pointer to readable stack slot
// r9 = scalar controlled by attacker

1: ...
2: ...
3: r6 = r9
4: ...
5: r9 = *(u8 *)(r6)
6: // leak r9
```

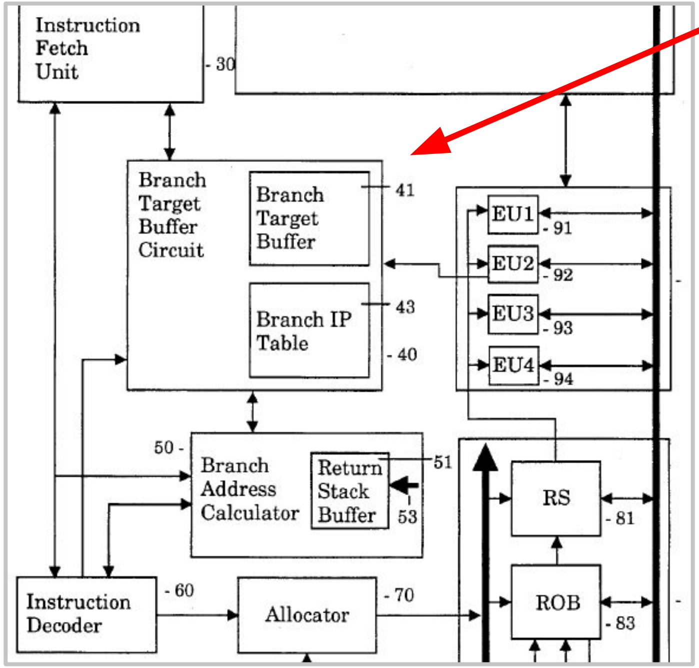
Steps done by BPF verifier:

- ➔ Spawns a new verification path to simulate unreachable paths from non-speculative domain
- ➔ Verifier ensures that program paths from speculative domain do not prune non-speculative ones
- ➔ Rejects program when e.g. type confusion observed

```
304: (71) r9 = *(u8 *)(r6 +0)
R6 invalid mem access 'inv'
processed 303 insns (limit 1000000) max_states_per_insn 0
===========================
bpf_stuff: prog load: Permission denied
```

BPF & Spectre v2

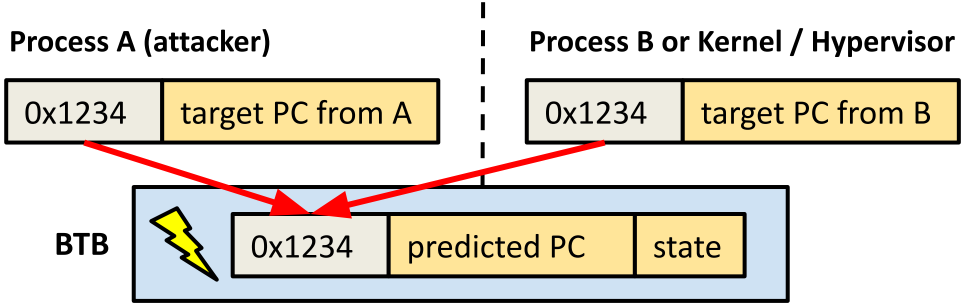
Branch Target Buffer (BTB) reduces perf penalty by predicting path of branches



(Image from original Intel patent)

- ➔ Predicts address of next instruction fetch before it is actually computed by the execution unit
- ➔ Look up on current PC to gather predicted target PC
- ➔ Expected to be sometimes wrong
- ➔ PC indexed via partial virtual address (e.g. lower bits)

Attack injects misspeculation to controlled addresses across security domains. Jump to 'gadget' code for leaking data.



BPF & Spectre v2

How is BPF affected? Everything that is having indirect calls.

➔ **Example 1:** Indirect calls inside helpers or first entry into the BPF program itself

```
BPF_CALL_4(bpf_map_update_elem, struct bpf_map *, map, void *, key,
           void *, value, u64, flags)
{
     return map->ops->map_update_elem(map, key, value, flags);
}
```

Dispatches into underlying BPF map implementation, e.g. array, hash, LRU, LPM, ...

(Not covered in this talk, see appendix.)

➔ **Example 2:** BPF tail calls used in BPF code

```
static inline int parse_eth_proto(struct __sk_buff *skb, __u16 proto)
{
     bpf_tail_call(skb, &jmp_table, proto);
     return 0;
}
```

Based on dynamic target index for BPF tail call map, it continues execution on target prog

BPF & Spectre v2

BPF tail calls: How do they work internally? Think of execv(3) ...

Interpreter

```
// R1: pointer to ctx
// R2: pointer to array (tail call map)
// R3: index

if (unlikely(index >= array->map.max_entries))
      goto next_insn;
if (unlikely(tail_call_cnt >= MAX_TAIL_CALLS))
      goto next_insn;
tail_call_cnt++;

prog = READ_ONCE(array->ptrs[index]);
if (!prog)
      goto next_insn;

insn = prog->insnsi;
goto next_insn;
```

JITed

```
33:   cmp    %edx,0x24(%rsi)
36:   jbe    0x63
38:   mov    0x24(%rbp),%eax
3e:   cmp    $0x20,%eax    ; 0x20: MAX_TAIL_CALLS
41:   ja     0x63
43:   add    $0x1,%eax
46:   mov    %eax,0x24(%rbp)
4c:   mov    0x90(%rsi,%rdx,8),%rax    ; get prog
54:   test   %rax,%rax
57:   je     0x63
59:   mov    0x28(%rax),%rax
5d:   add    $0x25,%rax          ; offset to entry
61:   jmpq   *%rax               ; indirect jump
63:   // fallthrough path
```

Subject to misspeculation!

BPF & Spectre v2

JIT mitigation, part 1: retpoline (return trampoline) to trap speculation in loop

`JITed, unprotected`

```
[...]
4c:   mov     0x90(%rsi,%rdx,8),%rax    ; get prog
54:   test    %rax,%rax
57:   je      0x63
59:   mov     0x28(%rax),%rax
5d:   add     $0x25,%rax             ; offset to entry
61:   jmpq    *%rax                  ; indirect jump
63:   // fallthrough path
```

`JITed, w/ mitigation`

```
[...]
4c:   mov     0x90(%rsi,%rdx,8),%rax    ; get prog
54:   test    %rax,%rax
57:   je      0x72
59:   mov     0x28(%rax),%rax
5d:   add     $0x25,%rax             ; offset to entry
61:   callq   0x6d                   ; 61-71: retpoline
66:   pause
68:   lfence
6b:   jmp     0x66
6d:   mov     %rax,(%rsp)
71:   retq
72:   // fallthrough path
```

Modifies return stack to force "return" to target.

Capturing CPU speculation in loop.

pause: to relinquish pipeline resources
lfence: as speculation barrier
i.e. both stop CPU from wasting power/time

BPF & Spectre v2

JIT mitigation/optimization, part 2: remove possibility to speculate via direct call

JITed, w/ retpoline

```
[...]
4c:   mov    0x90(%rsi,%rdx,8),%rax    ; get prog
54:   test   %rax,%rax
57:   je     0x72
59:   mov    0x28(%rax),%rax
5d:   add    $0x25,%rax              ; offset to entry
61:   callq  0x6d                    ; 61-71: retpoline
66:   pause
68:   lfence
6b:   jmp    0x66
6d:   mov    %rax,(%rsp)
71:   retq
72:   // fallthrough path
```

JITed, w/ direct call: no prog in map

```
[...]
4c:   nopl   0x0(%rax,%rax,1)
51:   // fallthrough path
```

Plain **nop** to force fall-through.

BPF & Spectre v2

JIT mitigation/optimization, part 2: remove possibility to speculate via direct call

`JITed, w/ direct call: no prog in map`

```
[...]
4c:   nopl   0x0(%rax,%rax,1)
51:   // fallthrough path
```

`JITed, w/ direct call: prog is in map`

```
[...]
4c:   jmpq   0xffffffffffb09f55
51:   // fallthrough path
```

Plain **nop** to force fall-through.

Plain **direct** jump to target.

Rewrite transition via **text_poke_bp()**.

- ➔ Possible if **map & key is constant**, that is, not dynamic & same from different paths
- ➔ Update on map triggers image update
- ➔ **Transitions:** nop→jmp (**insertion**), jmp→nop (**deletion**), jmp→jmp (**update**)
- ➔ Otherwise if preconditions not satisfied: emission of retpoline

BPF & Spectre v2

libbpf: small helper for BPF program authors called **bpf_tail_call_static()**

```
static inline void bpf_tail_call_static(void *ctx, const void *map, const __u32 slot)
{
	if (!__builtin_constant_p(slot))
		__bpf_unreachable(); // force compilation error if it gets built-in

	asm volatile("r1 = %[ctx]\n\t"
		     "r2 = %[map]\n\t"
		     "r3 = %[slot]\n\t"
		     "call 12"
		     :: [ctx]"r"(ctx), [map]"r"(map), [slot]"i"(slot)
		     : "r0", "r1", "r2", "r3", "r4", "r5");
}
```

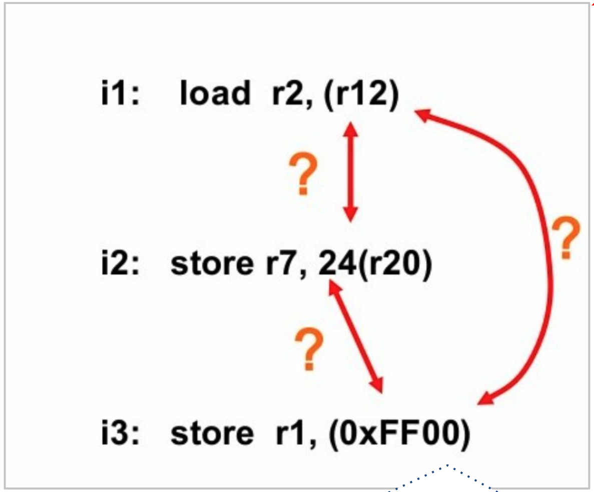
Given map & slot does not change, allows for direct jmp/nop transition in JIT.

➔ Performance studies ([here]() & [here]()): cost of one tail call drops more than half

BPF & Spectre v4

Memory disambiguator: memory dependence speculation

➔ Given OOO instruction execution, it predicts whether load depends on earlier store



(Do all these point to the same memory location?)

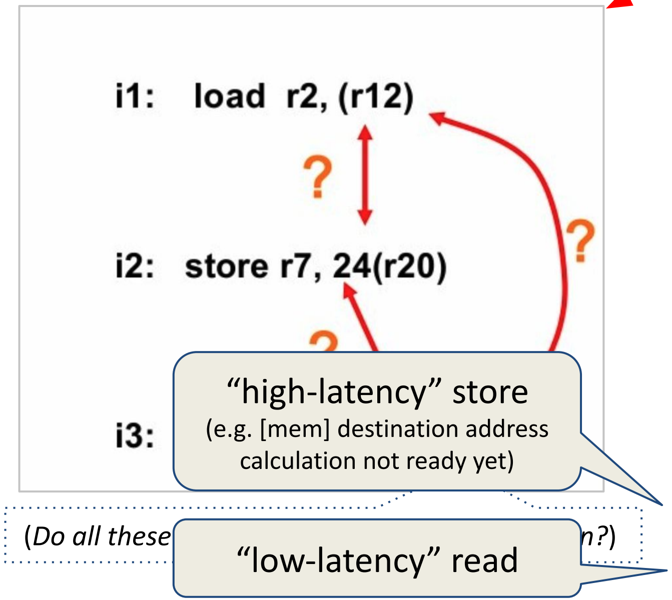
- ➔ Ambiguous dependency also forces “sequentiality”
- ➔ To increase CPU’s instruction level parallelism, it needs disambiguation mechanisms that are either safe or recoverable (from speculation)
- ➔ Dependency prediction expected to be sometimes wrong

Attack (Speculative Store Bypass) triggers misspeculation so that memory load executes ahead of dependant older store. A ‘gadget’ code can read stale data and utilize it for leaking.

BPF & Spectre v4

Memory disambiguator: memory dependence speculation

➔ Given OOO instruction execution, it predicts whether load depends on earlier store



- ➔ Ambiguous dependency also forces “sequentiality”
- ➔ To increase CPU’s instruction level parallelism, it needs disambiguation mechanisms that are either safe or recoverable (from speculation)
- ➔ Dependency prediction expected to be sometimes wrong

Attack (Speculative Store Bypass) triggers misspeculation so that memory load executes ahead of dependant older store. A ‘gadget’ code can read stale data and utilize it for leaking.

1:	store pointer_A to [mem]
N:	store pointer_B to [mem]
N+1:	load from [mem]

1:	store pointer_A to [mem]
N+1:	load from [mem]
N:	store pointer_B to [mem]

(dependency misspeculation → reordering)

BPF & Spectre v4

Example attack in BPF: crafting ‘fast’ versus ‘slow’ registers

```
// r2 = scalar controlled by attacker
// r7 = pointer to map value

*(u64 *)(r10 -16) = r2
r9 = r10
// train predictor to break r9/r10 dependency
r1 = map[id:4]
r2 = r7
r3 = 0
r4 = 4
call bpf_ringbuf_output#194288
*(u64 *)(r10 -16) = r7
r2 = *(u64 *)(r9 -16)
r3 = *(u8 *)(r2 +0)
// leak r3
```

Inits stack slot to OOB address

r9 == r10, both pointing to stack

Dummy call to helper

Spills valid map value pointer to stack

Misspeculates r9/r10 dependency, loads stale OOB address

Without CPU speculation: Reads out map value (r7)

BPF & Spectre v4

Example attack in BPF: crafting ‘fast’ versus ‘slow’ registers

```
// r2 = scalar controlled by attacker
// r7 = pointer to map value

*(u64 *)(r10 -16) = r2
r9 = r10
// train predictor to break r9/r10 dependency
r1 = map[id:4]
r2 = r7
r3 = 0
r4 = 4
call bpf_ringbuf_output#194288
*(u64 *)(r10 -16) = r7
r2 = *(u64 *)(r9 -16)
r3 = *(u8 *)(r2 +0)
// leak r3
```

Inits stack slot to OOB address

r9 == r10, both pointing to stack

Dummy call to helper

Spills valid map value pointer to stack

Misspeculates r9/r10 dependency, loads stale OOB address

Without CPU speculation: Reads out map value (r7)

But where do we speculate?

BPF & Spectre v4

Example attack in BPF: crafting ‘fast’ versus ‘slow’ registers

bpf_ringbuf_output() helper code:

- ➔ Internally **pushes & pops** register **r10 to stack** (due to calling convention)
- ➔ While **r9** stays in CPU **hardware register**
- ➔ Given the push/pop latency, value of r10 not yet known

```
r1 = map[id:4]
r2 = r7
r3 = 0
r4 = 4
call bpf_ringbuf_output#194288
*(u64 *)(r10 -16) = r7
r2 = *(u64 *)(r9 -16)
r3 = *(u8 *)(r2 +0)
// leak r3
```

Dummy call to helper

Spills valid map value pointer to stack

Misspeculates r9/r10 dependency, loads stale OOB address

Without CPU speculation: Reads out map value (r7)

But where do we speculate?

BPF & Spectre v4

Mitigation: emission of **lfence** instruction by BPF verifier as speculation barrier

Mitigated version:

```
...
*(u64 *)(r10 -16) = r7
nospec
r2 = *(u64 *)(r9 -16)
r3 = *(u8 *)(r2 +0)
// leak r3
```

Steps done by BPF verifier:

- ➔ Observes pointer spill/fills to BPF stack
- ➔ Observes ‘first-use’ of BPF stack slots (data or pointers)
- ➔ Inserts nospec BPF instruction after store
- ➔ JIT backends like x86 translate to lfence
- ➔ Now subsequent load cannot overtake anymore

Relation to Process Capabilities

Privileged BPF (CAP_BPF & CAP_PERFMON), e.g. used for tracing:

- ➔ Programs have v2 mitigations enabled as aligned with rest of kernel
- ➔ Performance impact low given retpoline-avoidance optimizations
- ➔ Generally little practical impact for vast majority of BPF projects

Unprivileged BPF (no CAPs) if available/enabled[1], e.g. reuseport programs:

- ➔ Programs have all v1/v2/v4 mitigations transparently enabled
- ➔ Performance impact low-medium depending on v2/v4 mitigations involved

1: See also /proc/sys/kernel/unprivileged_bpf_disabled or BPF_UNPRIV_DEFAULT_OFF kernel config

tlr;dr Summary

BPF runtime transparently applies Spectre v1/v2/v4 mitigations

➔ Mitigations like masking harden the code also for non-Spectre attacks
➔ They are applied in addition to the mitigations enforced by the kernel

tlr;dr Summary

BPF runtime transparently applies Spectre v1/v2/v4 mitigations

- ➔ Mitigations like masking harden the code also for non-Spectre attacks
- ➔ They are applied in addition to the mitigations enforced by the kernel

BPF verifier performing deeper static analysis than compilers:

- ➔ Spawns program path analysis also under speculative execution

tlr;dr Summary

BPF runtime transparently applies Spectre v1/v2/v4 mitigations

➔ Mitigations like masking harden the code also for non-Spectre attacks
➔ They are applied in addition to the mitigations enforced by the kernel

BPF verifier performing deeper static analysis than compilers:

➔ Spawns program path analysis also under speculative execution

BPF verifier also eliminates speculation possibilities for v1/v2 where possible:

➔ Pointer ALU rewrites with constant offsets instead of register-based offsets
➔ Transforms indirect jumps into direct jumps where retpolines can be avoided

tlr;dr Summary

BPF runtime transparently applies Spectre v1/v2/v4 mitigations

- ➔ Mitigations like masking harden the code also for non-Spectre attacks
- ➔ They are applied in addition to the mitigations enforced by the kernel

BPF verifier performing deeper static analysis than compilers:

- ➔ Spawns program path analysis also under speculative execution

BPF verifier also eliminates speculation possibilities for v1/v2 where possible:

- ➔ Pointer ALU rewrites with constant offsets instead of register-based offsets
- ➔ Transforms indirect jumps into direct jumps where retpolines can be avoided

BPF verifier applies mitigations for v4 only when necessary:

- ➔ Pointer spill/fill to BPF stack (e.g. under register pressure from LLVM side)
- ➔ Initial BPF stack usage to prevent read of prior stack data

Thank you!

Jann Horn (Google, Project Zero)

Piotr Krysiuk (Symantec, Threat Hunter Team)

Benedict Schlüter (Ruhr University Bochum)

Adam Morrison (Tel Aviv University)

John Fastabend (Isovalent)

Alexei Starovoitov (Facebook)

… and whole BPF and netdev community!

(Appendix: extract of BPF-related commits for more details on mitigation work)

Appendix: Spectre v1 & BPF work (extract)

b2157399cc98 (“bpf: prevent out-of-bounds speculation”)

be95a845cc44 (“bpf: avoid false sharing of map refcount with max_entries”)

c93552c443eb (“bpf: properly enforce index mask to prevent out-of-bounds speculation”)

979d63d50c0c (“bpf: prevent out of bounds speculation on pointer arithmetic”)

d3bd7413e0ca (“bpf: fix sanitation of alu op with pointer / scalar type from different paths”)

9d5564ddcf2a (“bpf: fix inner map masking to prevent oob under speculation”)

3612af783cf5 (“bpf: fix sanitation rewrite in case of non-pointers”)

f232326f6966 (“bpf: Prohibit alu ops for pointer types not defining ptr_limit”)

10d2bb2e6b1d (“bpf: Fix off-by-one for area size in creating mask to left”)

7fedb63a8307 (“bpf: Tighten speculative pointer arithmetic mask”)

b9b34ddbe207 (“bpf: Fix masking negation logic upon negative dst register”)

801c6058d14a (“bpf: Fix leakage of uninitialized bpf stack under speculation”)

bb01a1bba579 (“bpf: Fix mask direction swap upon off reg sign change”)

Appendix: Spectre v1 & BPF work (extract /2)

a7036191277f (“bpf: No need to simulate speculative domain for immediates”)

fe9a5ca7e370 (“bpf: Do not mark insn as seen under speculative path verification”)

9183671af6db (“bpf: Fix leakage under speculation on mispredicted branches”)

e042aa532c84 (“bpf: Fix pointer arithmetic mask tightening under state pruning”)

Appendix: Spectre v2 & BPF work (extract)

290af86629b2 (“bpf: introduce BPF_JIT_ALWAYS_ON config”)
a493a87f38cf (“bpf, x64: implement retpoline for tail call”)
ce02ef06fcf7 (“x86, retpolines: Raise limit for generating indirect calls from switch-case”)
a9d57ef15cbe (“x86/retpolines: Disable switch jump tables when retpolines are enabled”)
09772d92cd5a (“bpf: avoid retpoline for lookup/update/delete calls on maps”)
81c22041d9f1 (“bpf, x86, arm64: Enable jit by default when not built as always-on”)
da765a2f5993 (“bpf: Add poke dependency tracking for prog array maps”)
d2e4c1e6c294 (“bpf: Constant map key tracking for prog array pokes”)
428d5df1fa4f (“bpf, x86: Emit patchable direct jump as tail call”)
cc52d9140aa9 (“bpf: Fix record_func_key to perform backtracking on r3”)
75ccbef6369e (“bpf: Introduce BPF dispatcher”)
7e6897f95935 (“bpf, xdp: Start using the BPF dispatcher for XDP”)
0e9f6841f664 (“bpf, libbpf: Add bpf_tail_call_static helper for bpf programs”)

Appendix: Spectre v4 & BPF work (extract)

af86ca4e3088 (“bpf: Prevent memory disambiguation attack”)

f5e81d111750 (“bpf: Introduce BPF nospec instruction for mitigating Spectre v4”)

2039f26f3aca (“bpf: Fix leakage due to insufficient speculative store bypass mitigation”)