BPF and Spectre: Mitigating transient execution attacks

Benedict Schlüter (Ruhr University Bochum)
Daniel Borkmann (Isovalent, co-maintainer BPF)
Piotr Krysiuk (Symantec, Threat Hunter Team)

PriSC / POPL 2022

BPF and Spectre

What is BPF?

BPF Verifier

Speculative Execution

Side-Channels

BPF, Spectre & Mitigations

Q&A

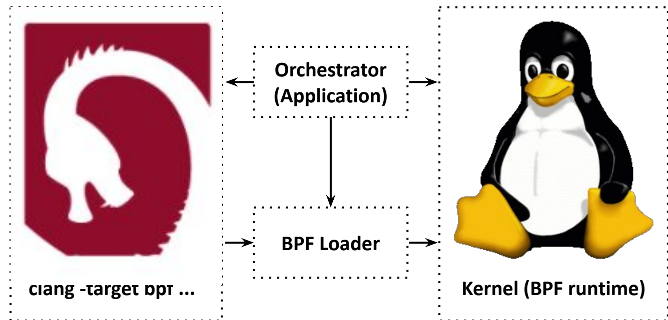
What is BPF?

Framework to extend the OS kernel

- ➔ BPF as a general purpose engine with minimal instruction set
- ➔ Allows for running programs in kernel to customize its behavior
- ➔ Without changing kernel's source, w/o need for reboot, w/o crashing

Use Cases

- ➔ Networking
  - ◆ Denial-of-service protection
  - ◆ Load-balancing, gateways, firewalling
  - ◆ Reduction of attack surface
  - ◆ Customization of host stack (e.g. TCP, K8s, ...)
- ➔ Security observability
- ➔ Security enforcement
- ➔ Kernel tracing and profiling



[https://ebpf.io/what-is-ebpf]

What is BPF?

BPF as new type of software

	Execution model	User defined	Compilation	Security	Failure mode	Resource access
User	task	yes	any	user based	abort	syscall, fault
Kernel	task	no	static	none	panic	direct
BPF	event	yes	JIT, CO-RE	verified, JIT	error message	restricted helpers

[Gregg, BPF Internals, LISA’21]

What is BPF?

BPF program state model

Off-CPU

event fires

On-CPU

program ended

Enabled

BPF

preempt

attach

helpers

Loaded

Sleeping

Kernel

page fault

(restricted state)

spinlock

Spinning

[Gregg, BPF Internals, LISA'21]

BPF Verifier

Static code analyzer walking in-kernel copy of BPF program instructions

➔ **Ensuring program termination**

- DFS traversal to check program is a DAG
- Preventing *un*bounded loops
- Preventing out-of-bounds or malformed jumps

BPF Verifier

Static code analyzer walking in-kernel copy of BPF program instructions

- ➔ **Ensuring program termination**
  - ◆ DFS traversal to check program is a DAG
  - ◆ Preventing *un*bounded loops
  - ◆ Preventing out-of-bounds or malformed jumps
- ➔ **Ensuring memory safety**
  - ◆ Preventing out-of-bounds memory access
  - ◆ Preventing use-after-free bugs and object leaks
  - ◆ Also mitigating vulnerabilities in the underlying hardware (Spectre)

BPF Verifier

Static code analyzer walking in-kernel copy of BPF program instructions

- **Ensuring program termination**
  - DFS traversal to check program is a DAG
  - Preventing *un*bounded loops
  - Preventing out-of-bounds or malformed jumps
- **Ensuring memory safety**
  - Preventing out-of-bounds memory access
  - Preventing use-after-free bugs and object leaks
  - Also mitigating vulnerabilities in the underlying hardware (Spectre)
- **Ensuring type safety**
  - Preventing type confusion bugs
  - BPF Type Format (BTF) for access to (kernel's) aggregate types

BPF Verifier

Static code analyzer walking in-kernel copy of BPF program instructions

- **Ensuring program termination**
  - DFS traversal to check program is a DAG
  - Preventing *un*bounded loops
  - Preventing out-of-bounds or malformed jumps
- **Ensuring memory safety**
  - Preventing out-of-bounds memory access
  - Preventing use-after-free bugs and object leaks
  - Also mitigating vulnerabilities in the underlying hardware (Spectre)
- **Ensuring type safety**
  - Preventing type confusion bugs
  - BPF Type Format (BTF) for access to (kernel's) aggregate types
- **Preventing hardware exceptions (division by zero)**
  - For unknown scalars, instructions rewritten to follow aarch64 spec

BPF Verifier

Works by simulating execution of *all* paths of the program

- ➔ **Follows control flow graph**
  - For each instruction computes set of possible states (BPF register set & stack)
  - Performs safety checks (e.g. memory access) depending on current instruction
  - Register spill/fill tracking for program's private BPF stack

BPF Verifier

Works by simulating execution of *all* paths of the program

- **Follows control flow graph**
  - For each instruction computes set of possible states (BPF register set & stack)
  - Performs safety checks (e.g. memory access) depending on current instruction
  - Register spill/fill tracking for program's private BPF stack
- **Back-edges in control flow graph**
  - Bounded loops by brute-force simulating all iterations up to a limit

BPF Verifier

Works by simulating execution of *all* paths of the program

- **Follows control flow graph**
  - For each instruction computes set of possible states (BPF register set & stack)
  - Performs safety checks (e.g. memory access) depending on current instruction
  - Register spill/fill tracking for program’s private BPF stack
- **Back-edges in control flow graph**
  - Bounded loops by brute-force simulating all iterations up to a limit
- **Dealing with potentially large number of states**
  - Path pruning logic compares current state vs prior states
    - Current path “equivalent” to prior paths with safe exit?
  - Function-by-function verification for state reduction
  - On-demand scalar precision (back-)tracking for state reduction
  - Terminates with rejection upon surpassing “complexity” threshold

BPF Verifier

BPF register state tracking

BPF reg		
	type	u32
	id	u32
	off	s32
	var_off	tnum
	s64min	s64
	s64max	s64
	u64min	u64
	u64max	u64
	s32min	s32
	s32max	s32
	u32min	u32
	u32max	u32
	...	

type: uninit, scalar, ptr_to_* types. Types can be composable, e.g. or'ed with ptr_maybe_null.

id: Identifier for state propagation (e.g. learned bits from conditions)

off: Fixed part of pointer offset (pointer types only).

var_off:

tnum		
	value	u64
	mask	u64

Represents knowledge of actual value for scalars (known and unknown bits).

s64min … u32max: Determined signed and unsigned 64 and 32-bit (sub-register) bounds.

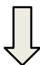
Coupled to the var_off tnum, holding a lower and upper bound of the unknown value.

Used to determine if any memory access using this register will result in a bad access.

BPF Verifier

Short primer on BPF tristate numbers (tnums)

tnum		
tnum	**value**	u64
	mask	u64



for each bit position P

P.value	P.mask	P.state
0	0	0
1	0	1
0	1	X
1	1	NaN

Example, 4 bit tnum:

010X ⟶ v = 0100, m = 0001
010X represents S { 0100, 0101 } ⟶ { **4, 5** }

XXXX ⟶ v = 0000, m = 1111
XXXX represents S { 0000, 0001, ..., 1111 } ⟶ { **0, 1, ..., 15** }

tnum and 64/32 bit min/max bounds relation:

Both needed, verifier propagates & refines knowledge between them.

Example state:

R ⟶ { 64 bit bounds [1, 0x77fffffff],
32 bit bounds [0, 0x7ffffff],
tnum v = 0, m = 0x77fffffff }

BPF Verifier

Short primer on BPF tristate numbers (tnums)

tnum	value	u64
	mask	u64

```
def tnum_add(tnum P, tnum Q):

    u64 sv := P.v + Q.v
    u64 sm := P.m + Q.m
    u64 Σ := sv + sm
    u64 χ := Σ ⊕ sv
    u64 η := χ | P.m | Q.m
    tnum R := tnum(sv & ~η, η)
    return R
```

Example, 4 bit tnum addition:

10X0 → v = 1000, m = 0010 → { **8, 10** }
\+ 10X1 → v = 1001, m = 0010 → { **9, 11** }
= 10XX1 → v = 10001, m = 00110 → { **17, 19, 21**, **23** }

[Vishwanathan et al., Sound, Precise, and Fast Abstract Interpretation with Tristate Numbers]

BPF Verifier

Short primer on BPF tristate numbers (tnums)

tnum	value	u64
	mask	u64

```
def  our_mul(tnum P, tnum Q):

  ACC_v := tnum(P.v * Q.v, 0)
  ACC_m := tnum(0, 0)

  while P.value or P.mask:
    # LSB of tnum P is a certain 1
    if (P.v[0] == 1) and (P.m[0] == 0):
      ACC_m := tnum_add(ACC_m, tnum(0, Q.m))
    # LSB of tnum P is uncertain
    else if (P.m[0] == 1):
      ACC_m := tnum_add(ACC_m, tnum(0, Q.v|Q.m))
    # Note: no case for LSB is certain 0
    P := tnum_rshift(P, 1)
    Q := tnum_lshift(Q, 1)

  tnum R := tnum_add(ACC_v, ACC_m)
  return R
```

Example, 4 bit tnum multiplication:

X01 ⟶ v = 001, m = 100 ⟶ { **1, 5** }
* X10 ⟶ v = 010, m = 100 ⟶ { **2, 6** }
= XXX10 ⟶ v = 00010, m = 11100 ⟶ { **2, 6, 10, 14, 18, 22, 26, 30** }

[Vishwanathan et al., Sound, Precise, and Fast Abstract Interpretation with Tristate Numbers]

BPF Verifier

Toy example

```
struct {
  uint8_t index;
  int32_t value;
  int32_t array[256];
} s;

s.array[s.index] = -s.value;
```

BPF Verifier

Toy example

```
struct {
  uint8_t index;
  int32_t value;
  int32_t array[256];
} s;

s.array[s.index] = -s.value;
```

BPF bytecode

```
; r0 points to s
r1 = *(u8*)(r0 + offsetof(s, index))
r2 = *(u32*)(r0 + offsetof(s, value))
r0 += offsetof(s, array)
r1 *= sizeof(int32_t)
r0 += r1
r2 = -r2
*(u32*)(r0) = r2
```

BPF Verifier

BPF bytecode:

```
; r0 points to s
```

; bpf_reg_state[]:

```
; r0 map_value, off=0, vs=1032
```

BPF Verifier

BPF bytecode:

```
; r0 points to s

r1 = *(u8*)(r0 + offsetof(s, index))
```

; bpf_reg_state[]:

```
; r0 map_value, off=0, vs=1032

; r1 umax_value=255,
     var_off=(0x0; 0xff)
```

BPF Verifier

BPF bytecode:

```
; r0 points to s

r1 = *(u8*)(r0 + offsetof(s, index))

r2 = *(u32*)(r0 + offsetof(s, value))
```

; bpf_reg_state[]:

```
; r0 map_value, off=0, vs=1032

; r1 umax_value=255,
     var_off=(0x0; 0xff)
; r2 umax_value=4294967295,
     var_off=(0x0; 0xffffffff)
```

BPF Verifier

BPF bytecode:

```
; r0 points to s

r1 = *(u8*)(r0 + offsetof(s, index))

r2 = *(u32*)(r0 + offsetof(s, value))

r0 += offsetof(s, array)
```

; bpf_reg_state[]:

```
; r0 map_value, off=0, vs=1032

; r1 umax_value=255,
     var_off=(0x0; 0xff)
; r2 umax_value=4294967295,
     var_off=(0x0; 0xffffffff)
; r0 map_value, off=8, vs=1032
```

BPF Verifier

BPF bytecode:

```
; r0 points to s

r1 = *(u8*)(r0 + offsetof(s, index))

r2 = *(u32*)(r0 + offsetof(s, value))

r0 += offsetof(s, array)
r1 *= sizeof(int32_t)
```

; bpf_reg_state[]:

```
; r0 map_value, off=0, vs=1032

; r1 umax_value=255,
     var_off=(0x0; 0xff)
; r2 umax_value=4294967295,
     var_off=(0x0; 0xffffffff)
; r0 map_value, off=8, vs=1032
; r1 umax_value=1020,
     var_off=(0x0; 0x3fc)




; r1 ∈ {0, 4, 8, …, 1020}
```

BPF Verifier

BPF bytecode:

```
; r0 points to s

r1 = *(u8*)(r0 + offsetof(s, index))

r2 = *(u32*)(r0 + offsetof(s, value))

r0 += offsetof(s, array)
r1 *= sizeof(int32_t)

r0 += r1
```

; bpf_reg_state[]:

```
; r0 map_value, off=0, vs=1032

; r1 umax_value=255,
     var_off=(0x0; 0xff)
; r2 umax_value=4294967295,
     var_off=(0x0; 0xffffffff)
; r0 map_value, off=8, vs=1032
; r1 umax_value=1020,
     var_off=(0x0; 0x3fc)
; r0 map_value, off=8, vs=1032,
     umax_value=1020,
     var_off=(0x0; 0x3fc)
```

BPF Verifier

BPF bytecode:

```
; r0 points to s

r1 = *(u8*)(r0 + offsetof(s, index))

r2 = *(u32*)(r0 + offsetof(s, value))

r0 += offsetof(s, array)
r1 *= sizeof(int32_t)

r0 += r1



r2 = -r2
```

; bpf_reg_state[]:

```
; r0 map_value, off=0, vs=1032

; r1 umax_value=255,
     var_off=(0x0; 0xff)
; r2 umax_value=4294967295,
     var_off=(0x0; 0xffffffff)
; r0 map_value, off=8, vs=1032
; r1 umax_value=1020,
     var_off=(0x0; 0x3fc)
; r0 map_value, off=8, vs=1032,
     umax_value=1020,
     var_off=(0x0; 0x3fc)
; r2 [NO CONSTRAINTS]

; simplifies BPF verifier
```

BPF Verifier

BPF bytecode:

```
; r0 points to s

r1 = *(u8*)(r0 + offsetof(s, index))

r2 = *(u32*)(r0 + offsetof(s, value))

r0 += offsetof(s, array)
r1 *= sizeof(int32_t)

r0 += r1



r2 = -r2
*(u32*)(r0) = r2
```

; bpf_reg_state[]:

```
; r0 map_value, off=0, vs=1032

; r1 umax_value=255,
     var_off=(0x0; 0xff)
; r2 umax_value=4294967295,
     var_off=(0x0; 0xffffffff)
; r0 map_value, off=8, vs=1032
; r1 umax_value=1020,
     var_off=(0x0; 0x3fc)
; r0 map_value, off=8, vs=1032,
     umax_value=1020,
     var_off=(0x0; 0x3fc)
; r2 [NO CONSTRAINTS]

; safe for all simulated r0 values
```

BPF Verifier

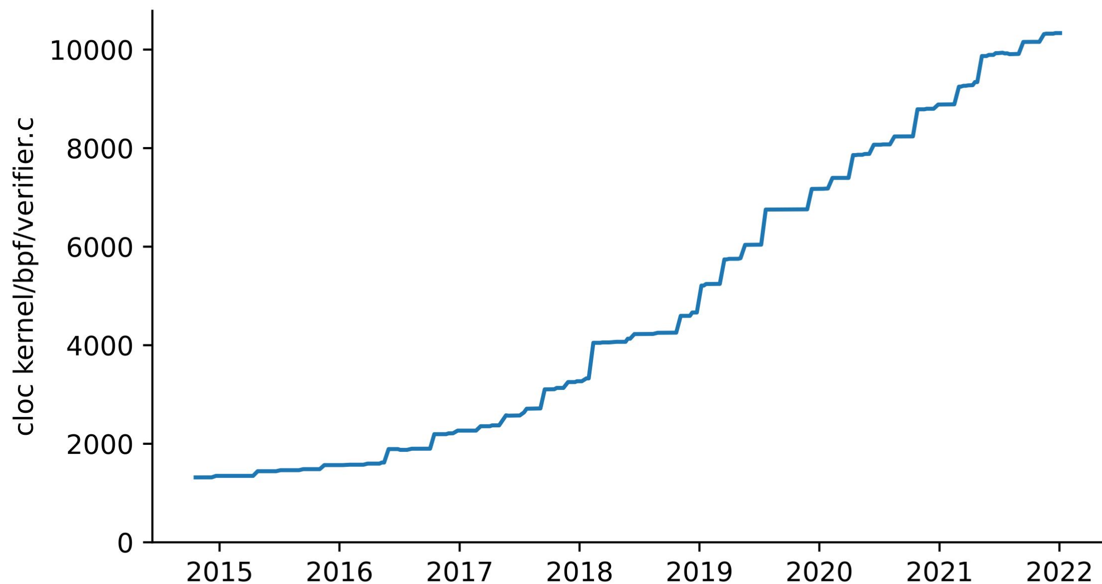
Challenges

- ➔ **Attractive target for exploitation when exposed to non-root**
  - ◆ Growing verifier complexity
  - ◆ Programmability can be abused to bypass mitigations once in OS kernel
- ➔ **Reasoning about verifier correctness is non-trivial**
  - ◆ Especially Spectre mitigations
  - ◆ Only partial formal verification (e.g. tnums, JITs)
- ➔ **Occasions where valid programs get rejected**
  - ◆ LLVM vs verifier "disconnect" to understand optimizations
  - ◆ Restrictions when tracking state
- ➔ **"Stable ABI" for BPF program types (with some limitations)**
  - ◆ BPF programs in production should not break upon OS kernel upgrade
- ➔ **Performance vs security considerations**
  - ◆ Verification of complex programs must be efficient to be practical
  - ◆ Mitigations must be practical as performance of programs crucial

BPF Verifier

Challenges (cont)

➔ Allowing both 32-bit and 64-bit operations in BPF programs contributes to complexity
➔ Under active development to support new BPF features



Speculative Execution

“Iron Law” of processor performance

$$\frac{time}{program} = \frac{instructions}{program} \times \frac{cycles}{instruction} \times \frac{time}{cycle}$$

program source code

compiler technology

instruction set architecture (ISA)

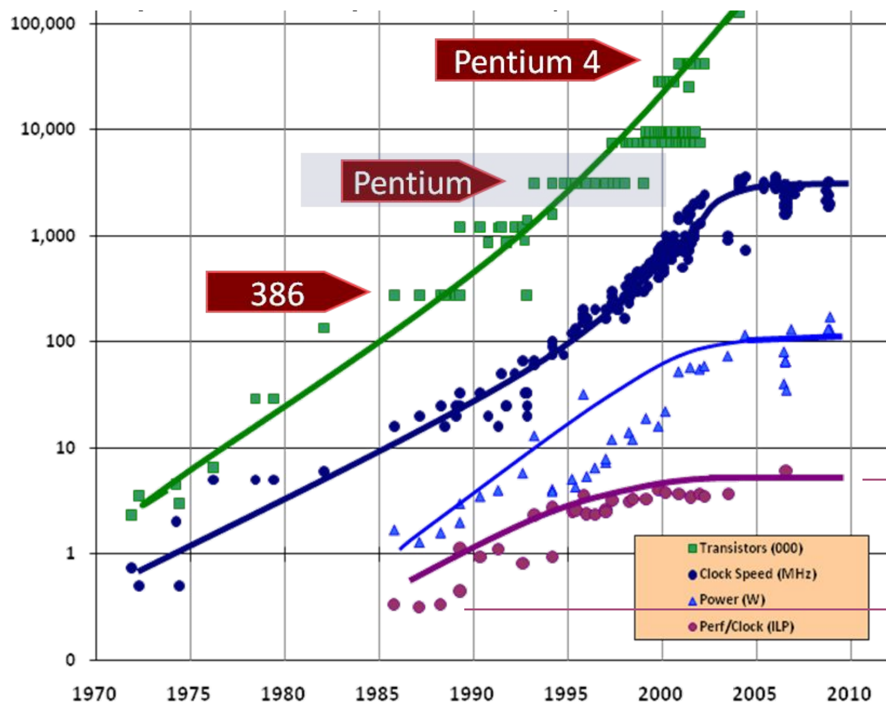
microarchitecture

fabrication technology

[Krste Asanovic, CS152, 2009]

Speculative Execution

Microarchitecture optimizations contributed significantly to performance gains



Contribution from microarchitectural improvements

[Sutter, The Free Lunch Is Over, 2009]

Speculative Execution

Microarchitecture optimizations example

- ➔ Exploit Instruction-Level Parallelism by executing independent instructions in parallel
- ➔ Or even out-of-order based on input data and hardware resources availability

Modern hardware implements variants of Tomasulo's algorithm (1967)

- ➔ Allows for multiple in-flight instructions
- ➔ Instruction dependencies tracked using “data flow” graph

```
r4 = r1 op r2
r5 = r2 op r3
r6 = [r5]                ; memory load can be slow
r7 = r6 op r4            ; waits for r6
r8 = r3 op r4            ; could execute in the meantime
```

- ➔ Executes instructions once dependencies are ready, perhaps out-of-order
- ➔ Commits results in program order to maintain illusion of sequential execution

Speculative Execution

Parallel execution challenges relevant to our work

- ➔ Control dependencies
- ➔ Ambiguous memory dependencies

Control dependencies

- ➔ Conditional and indirect branch instructions occur frequently in typical programs
- ➔ However branch outcomes are predictable with high accuracy
- ➔ Rollback on misprediction
- ➔ Universally exploited to significantly increase gains from parallel execution

Ambiguous memory dependencies

- ➔ Load depends on preceding store only when accessing the same memory location
- ➔ Resolved after memory addresses become available
- ➔ However indirect addressing is very common and may delay disambiguation

Speculative Execution

Speculative memory disambiguation

- ➔ Ambiguous memory dependency example

```
r11 = [r10]          ; memory load can be slow
[r11] = r12          ; waits for r11
r4 = r1 op r2
r5 = r2 op r3
r6 = [r5]            ; execute in the meantime?
```

- ➔ Speculatively proceed with load assuming r5 != r11
- ➔ Rollback, including dependencies, if wrong

Memory disambiguation on Intel microprocessors

- ➔ Speculation techniques described in “Intel® 64 and IA-32 Architectures Optimization Reference Manual”
- ➔ Testing indicates that speculation is enabled via predictor (no official documentation)

Speculative Execution

Rollback on misspredition is limited to architectural (visible) state

➔ Not practical to extend to microarchitectural state, e.g. predictors that depend on past behaviour

How to abuse: Speculative Store Bypass (SSB)

1. Train memory disambiguator
2. Speculatively execute unsafe load
3. Modify microarchitectural state under speculation
4. Extract information via side channel

Speculative Execution

Disable speculation at the cost of CPU performance?

➔ Hardware vendors provide mechanism
➔ Software developers to make the choice

Speculative Execution

Speculative Store Bypass (SSB) mitigations

- ➔ `lfence` instruction (x86)
  - ◆ Stops younger instructions from executing until all store addresses are resolved
- ➔ CPU configuration registers

Speculative Execution

Common patterns in speculative-execution vulnerabilities

- ➔ Hardware relies on probabilistic methods to break dependencies
  - ◆ Required to maximise performance
- ➔ Microarchitectural state affected at least to track prediction accuracy
  - ◆ Not fixable
- ➔ Side-channel to extract microarchitectural state
  - ◆ Variety of high-resolution times available

Speculative Execution

Hardware vendors continue to innovate to extract parallelism

➔ “Security analysis of AMD predictive store forwarding” (AMD, March 2021)

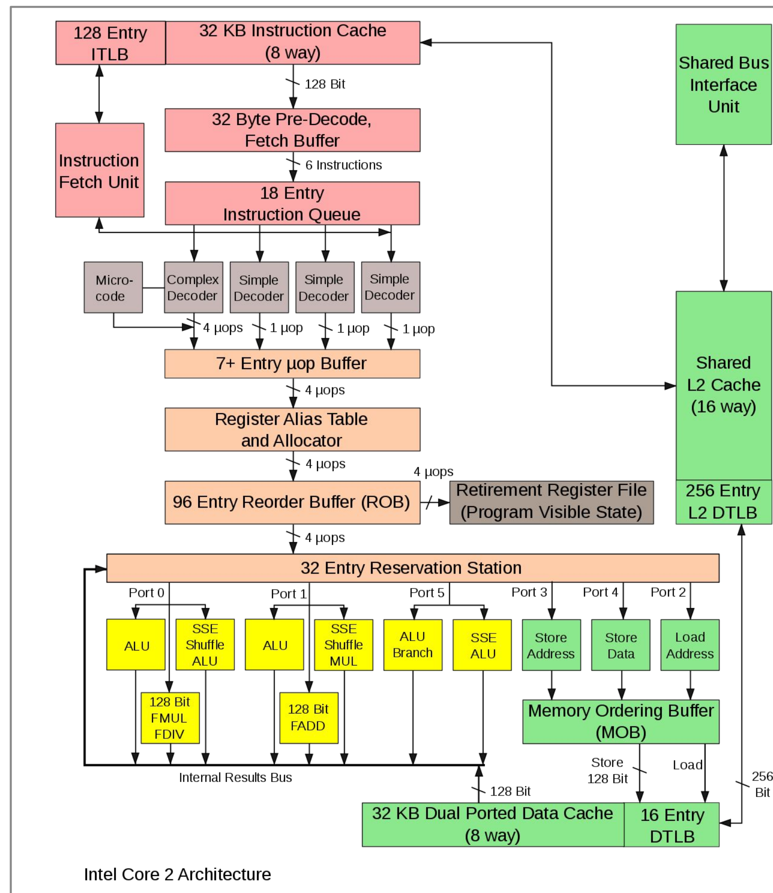
Software needs to accommodate

Side-Channels

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



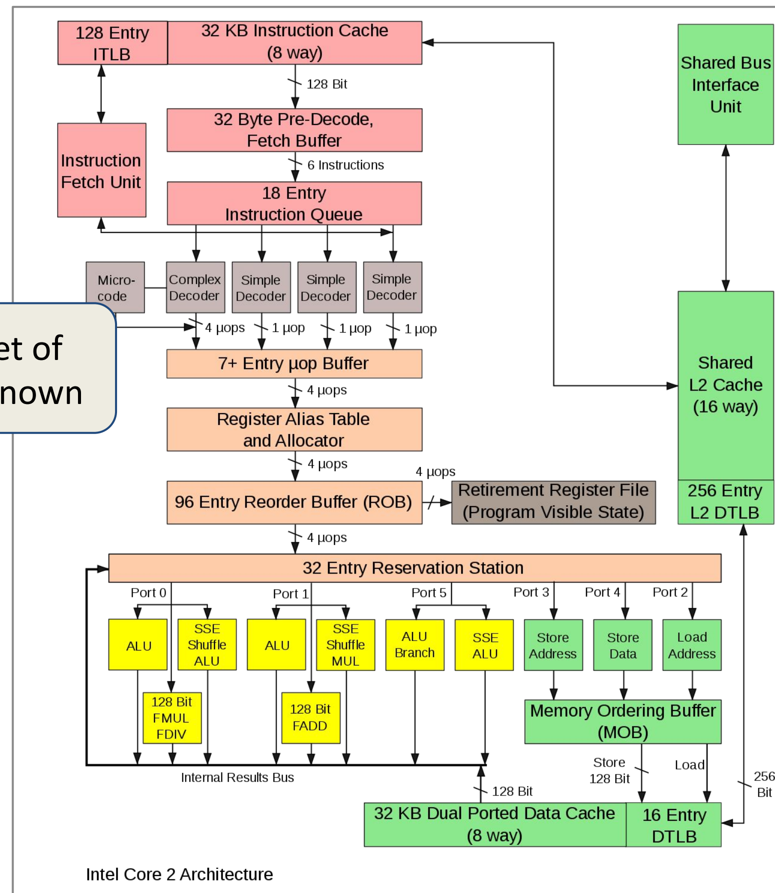
Side-Channels

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



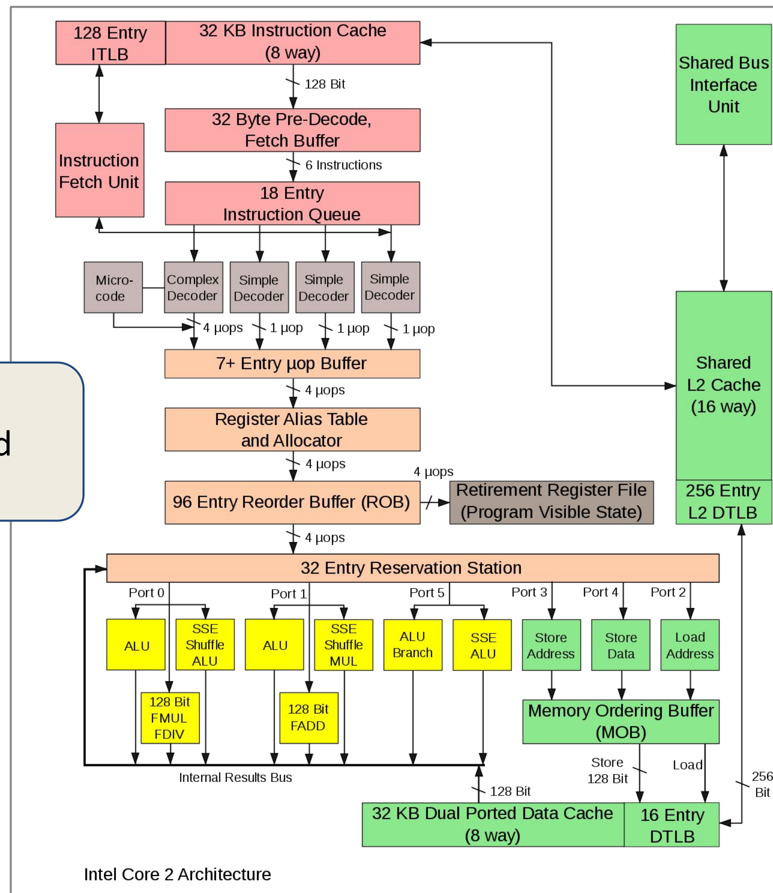
Side-Channels

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

Side-Channels

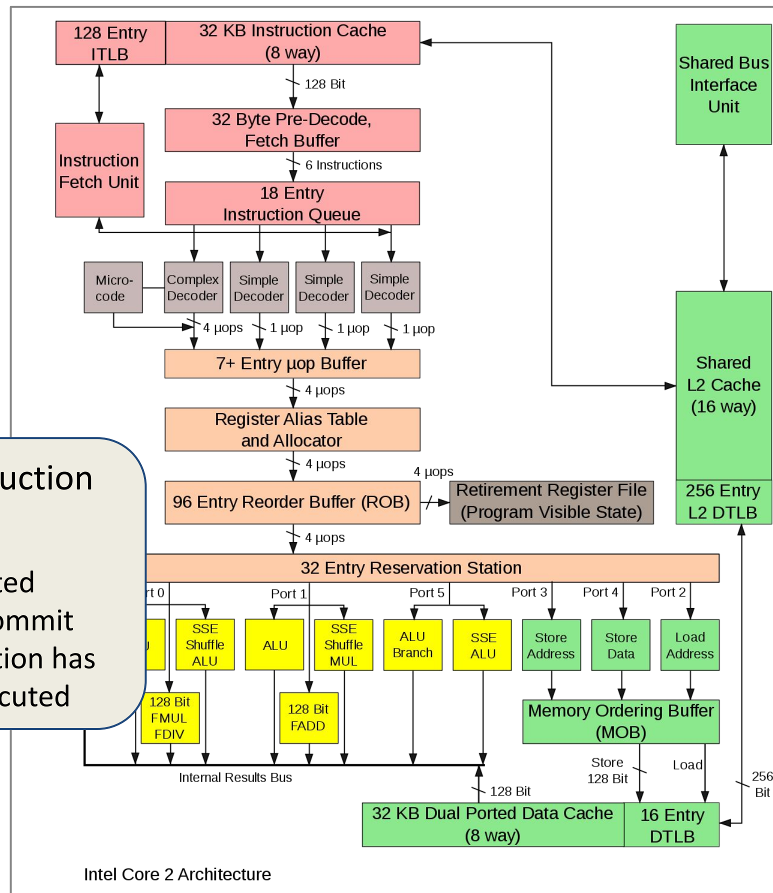
µarch is the way a given ISA like x86 is implemented

- ➔ 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



Side-Channels

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.

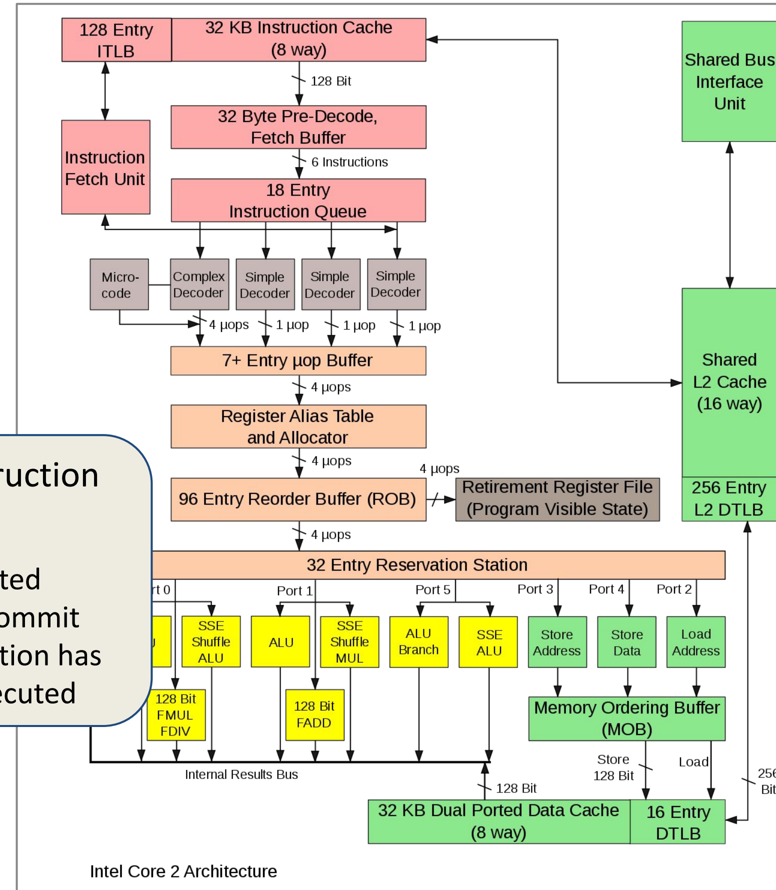
➔ **If prediction true:** predicted execution is allowed to commit

➔ **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**



Side-Channels

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.

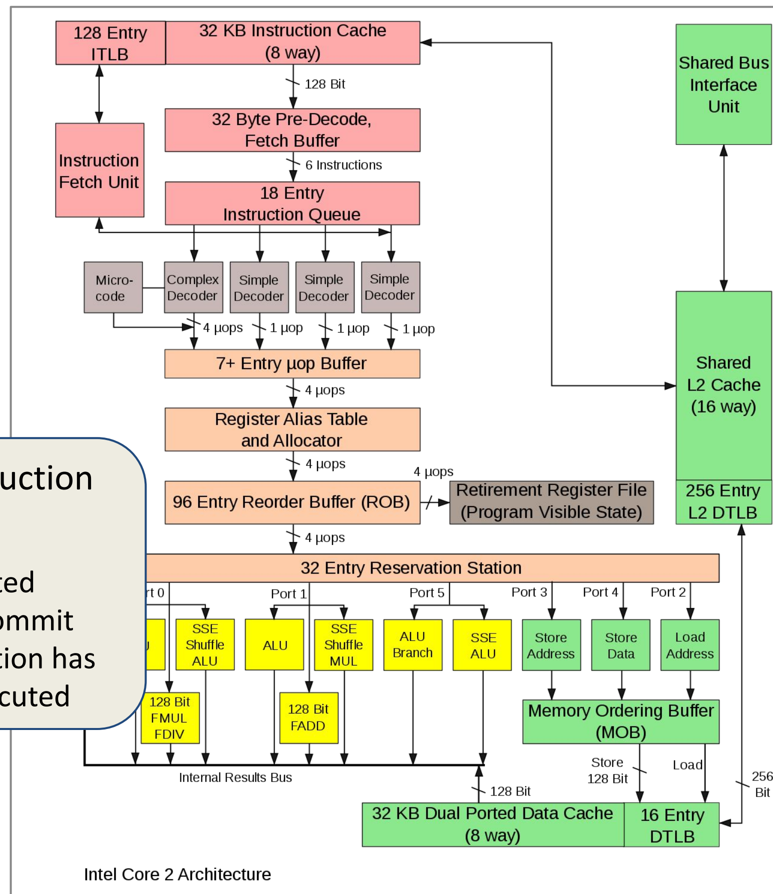
- ➔ **If prediction true:** predicted execution is allowed to commit
- ➔ **If prediction false:** execution has to be unrolled and re-executed

“Transient instructions”

Rollback on misspeculation:

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

observable side-effect!

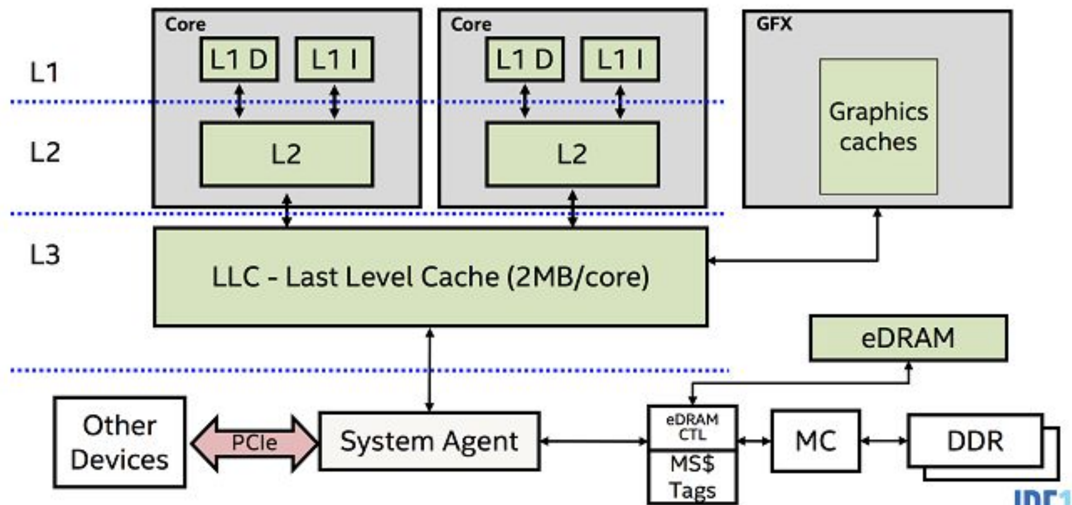


Side-Channels

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

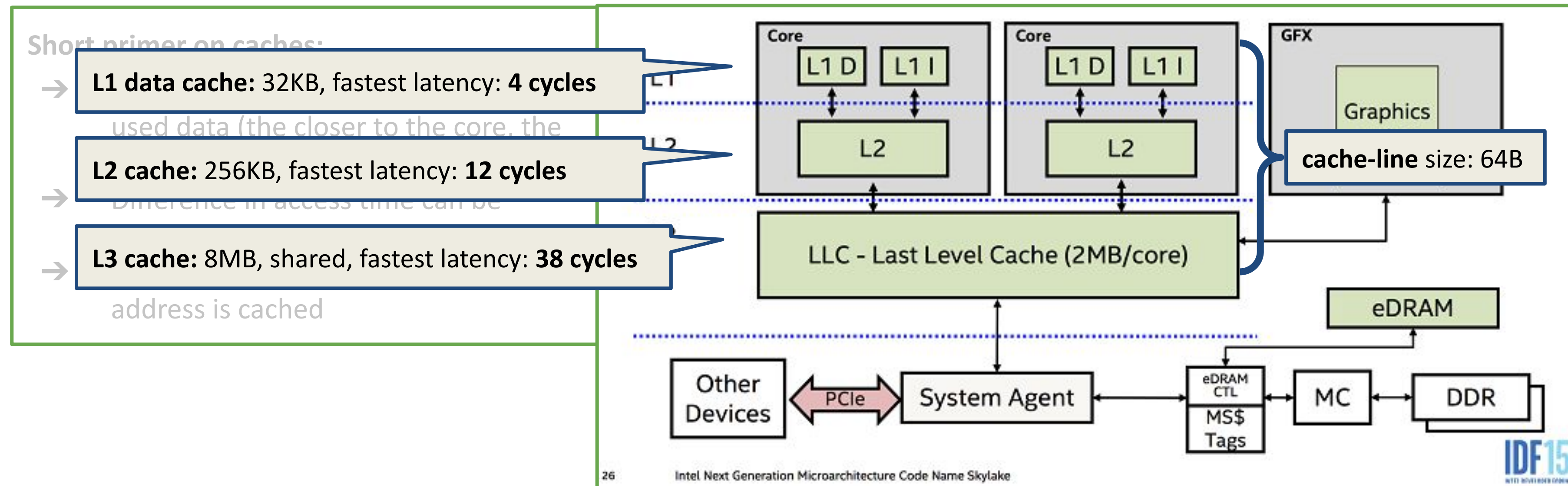
Short primer on caches:

- ➔ 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



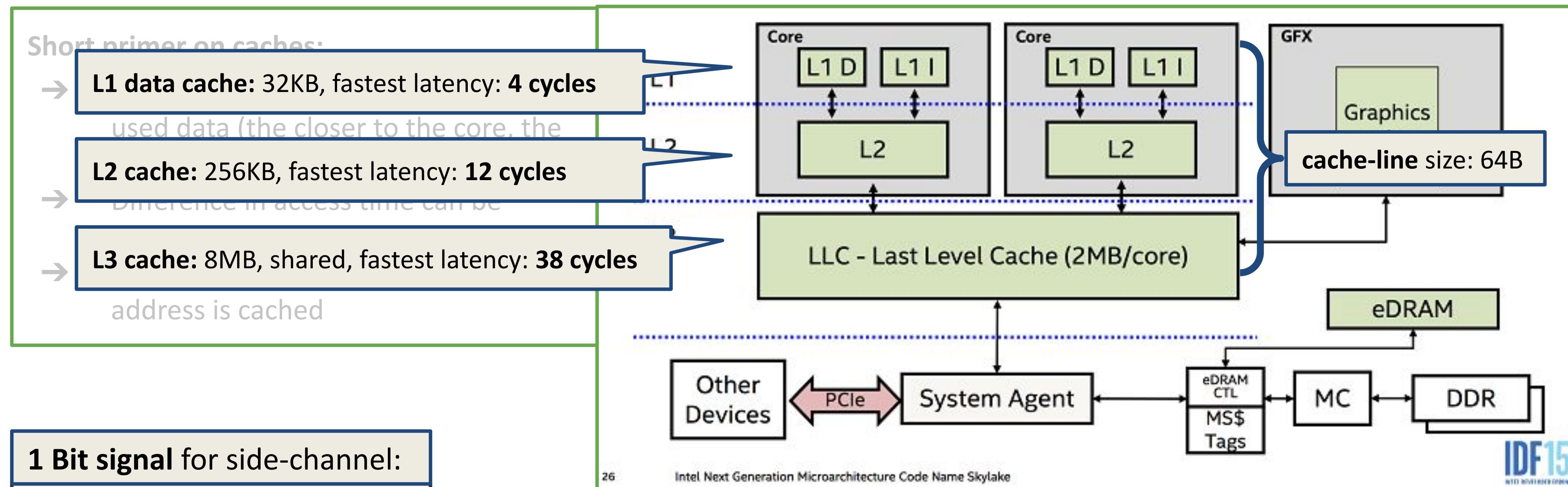
Side-Channels

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



Side-Channels

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



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)

Side-Channels

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.

Side-Channels

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**

Side-Channels

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

Side-Channels

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]

Side-Channels

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:

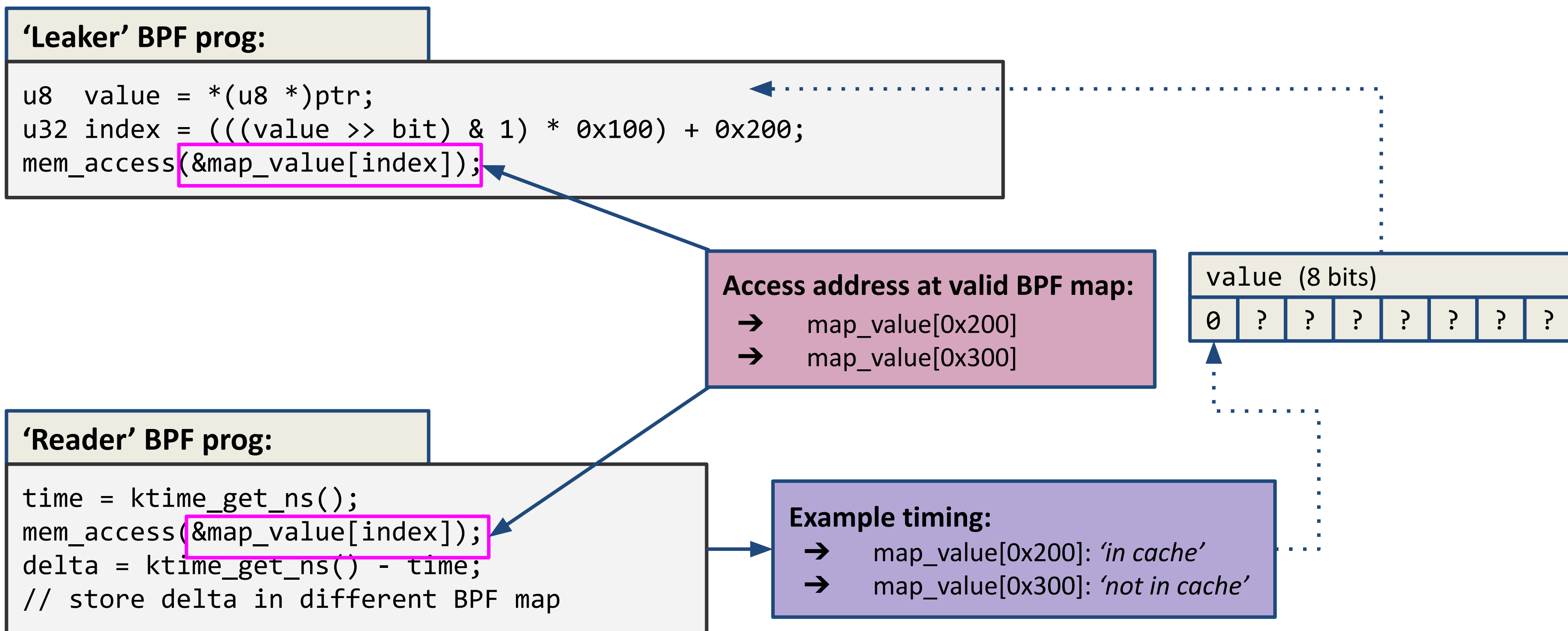
- ➔ map_value[0x200]
- ➔ map_value[0x300]

'Reader' BPF prog:

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

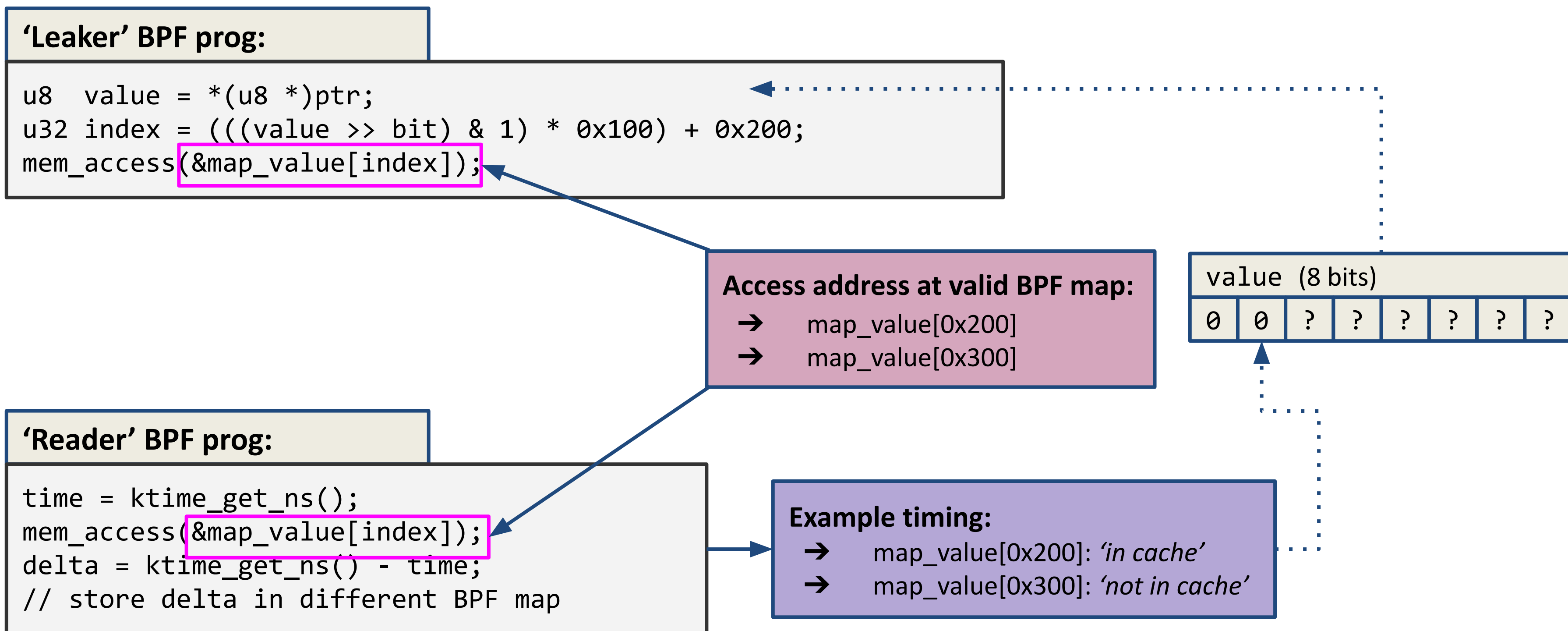
Side-Channels

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



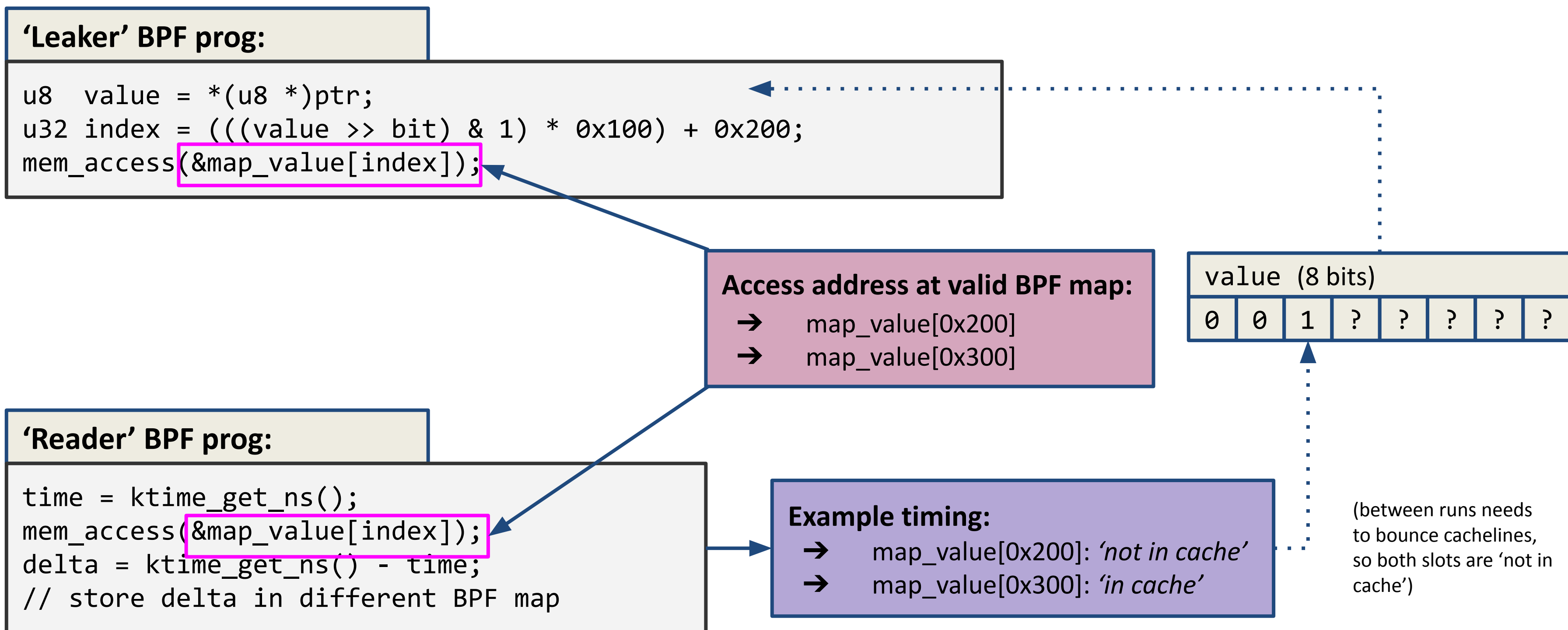
Side-Channels

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



Side-Channels

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



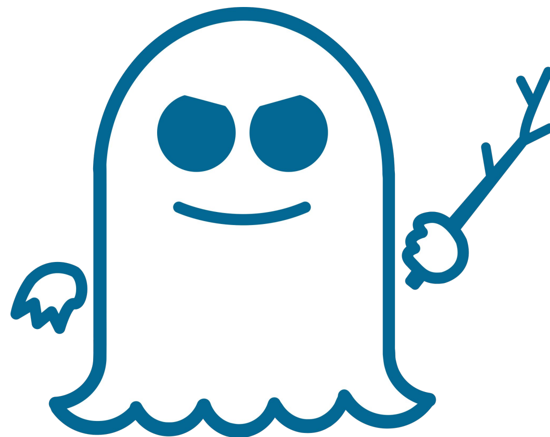
Microarchitecture & Spectre

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**

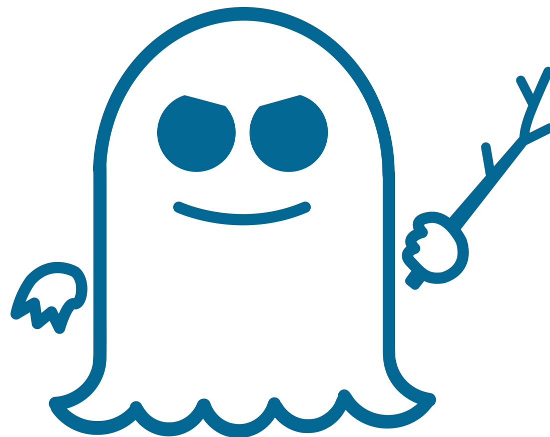
- ➔ Not triggered by software bugs whatsoever
- ➔ Execution without speculation is safe

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

- ➔ Different attacks to trigger misspeculation

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

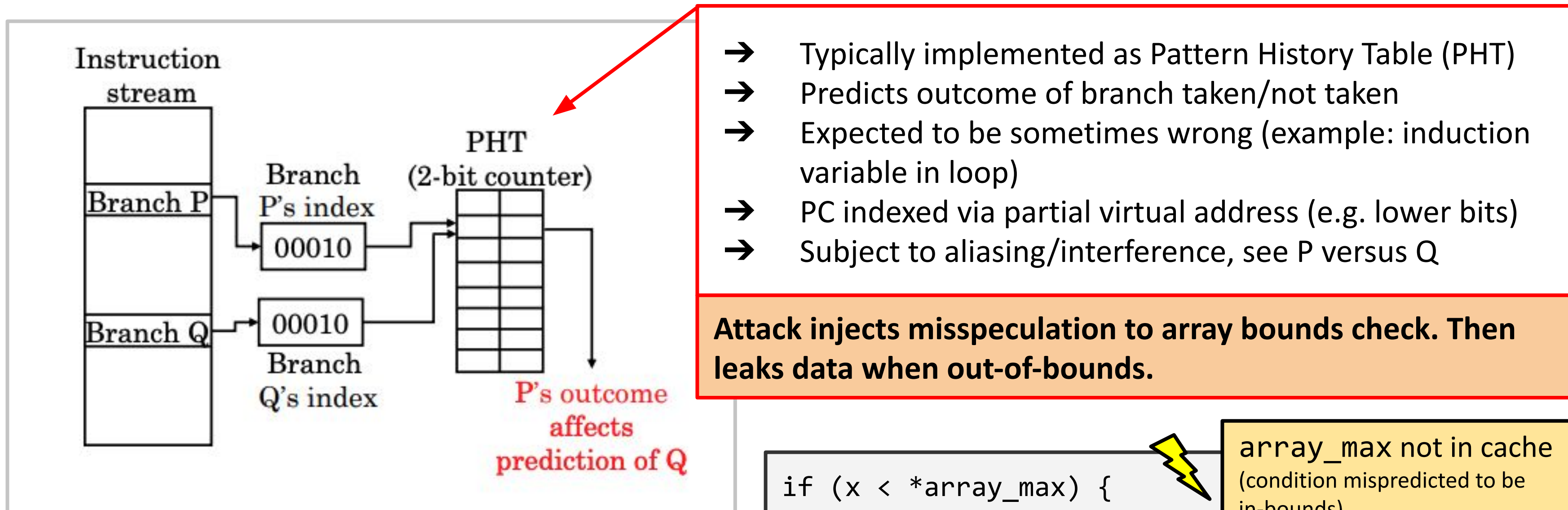
- ➔ **Disclaimer:** not able to cover every aspect due to time limit
- ➔ Focus on Spectre v1/v2/v4
- ➔ Relation to process capabilities



BPF & Spectre v1

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

- ➔ CPU reduces perf penalty by predicting outcome of branches



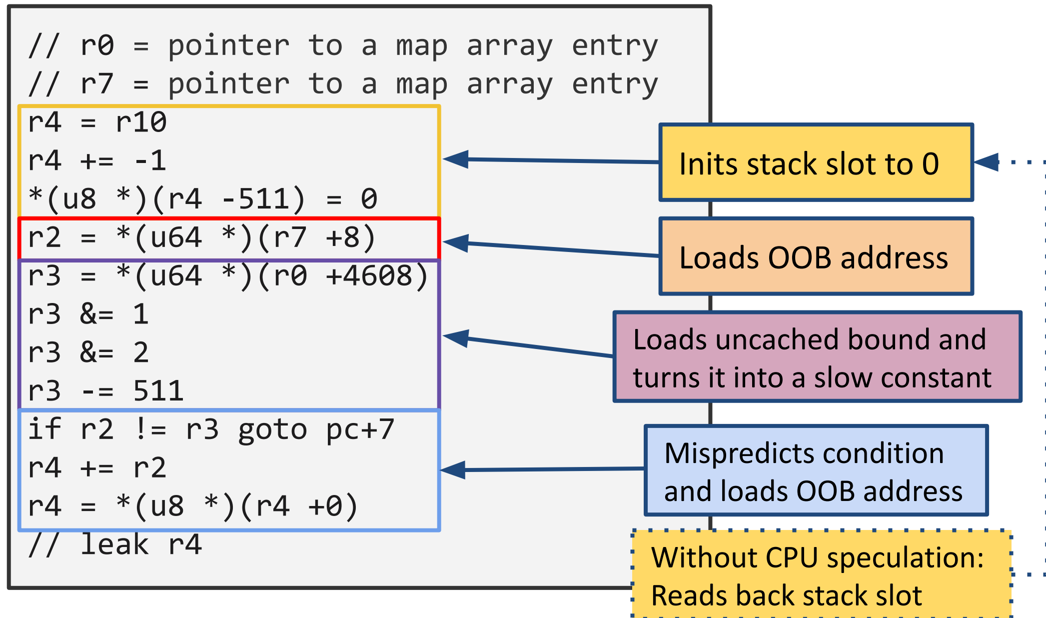
(Image from https://arxiv.org/pdf/1804.00261.pdf)

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

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

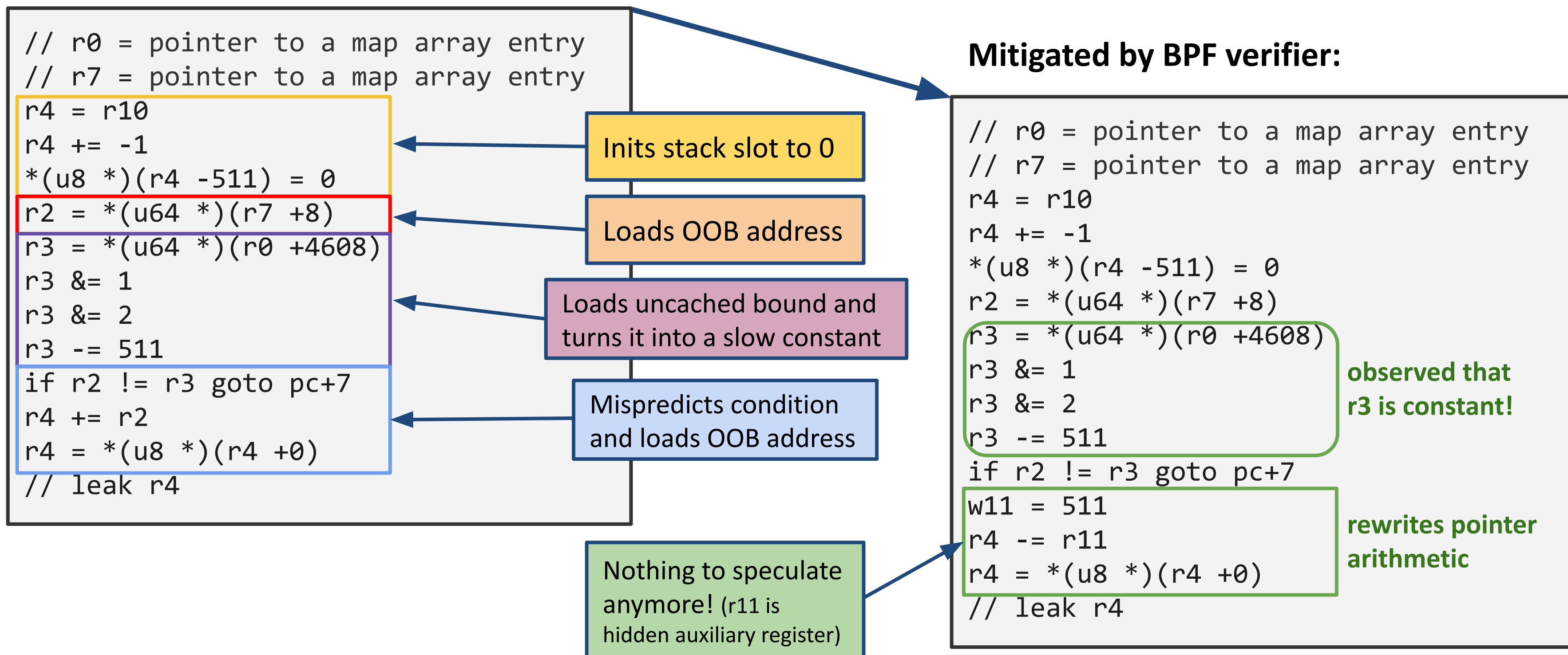
BPF & Spectre v1

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



BPF & Spectre v1

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



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

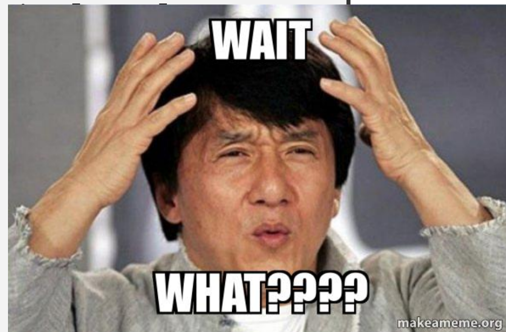
➔ **Eliminate speculation** if possible by rewrite with constants
➔ Safely **redirect speculation** to be within array bounds

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

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
```

Example	offset is "in-bounds" r2 speculation: 31 (0x1F) max value: 32 (0x20)	offset is "out-of-bounds" 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
```

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:

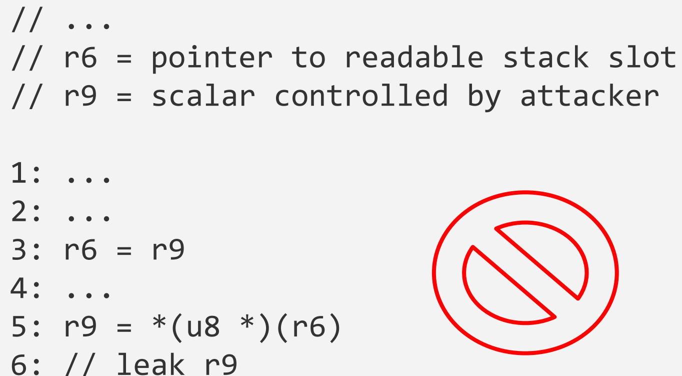
```
// ...
// r6 = pointer to readable stack slot
// r9 = scalar controlled by attacker

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



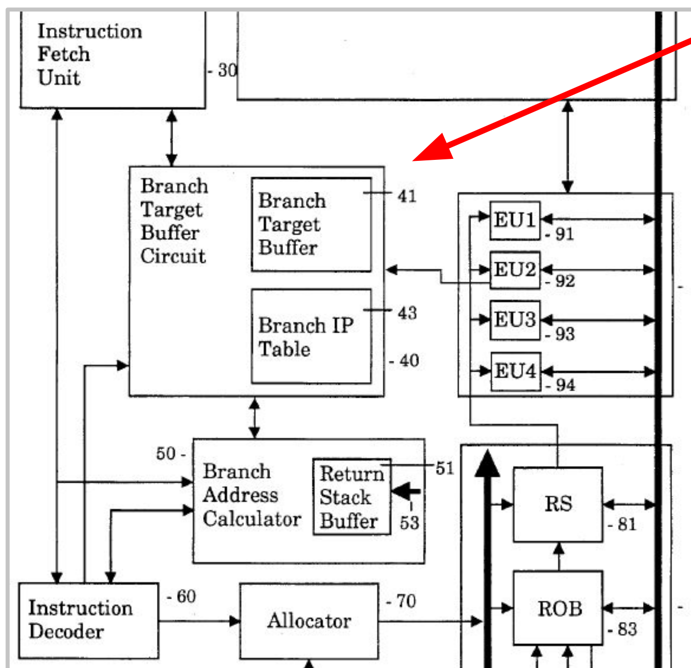
Steps done by BPF verifier:

- ➔ 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)

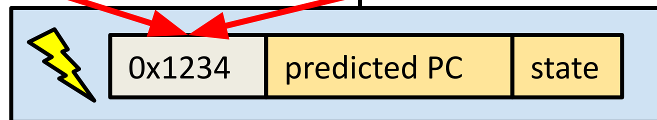
- ➔ Predicts address of next instruction fetch before it is actually computed by the execution unit
- ➔ Look up on current PC to gather predicted target PC
- ➔ Expected to be sometimes wrong
- ➔ PC indexed via partial virtual address (e.g. lower bits)

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

Process A (attacker)

Process B or Kernel / Hypervisor

BTB



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;
```

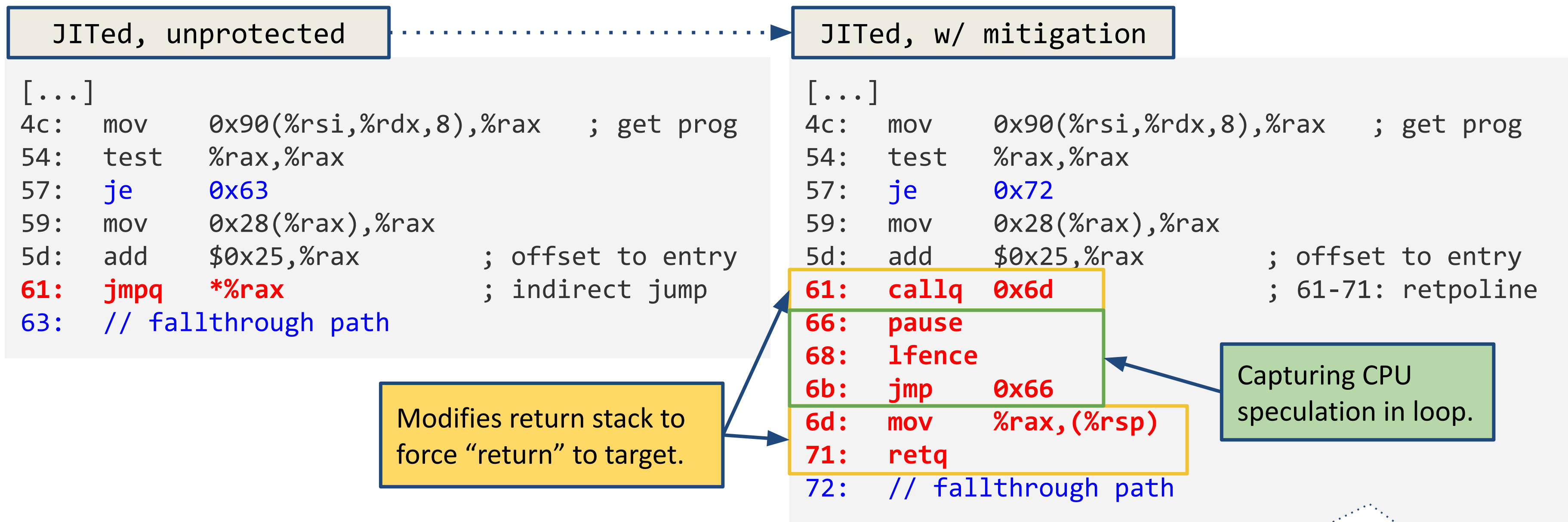
JITed

```
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



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
```

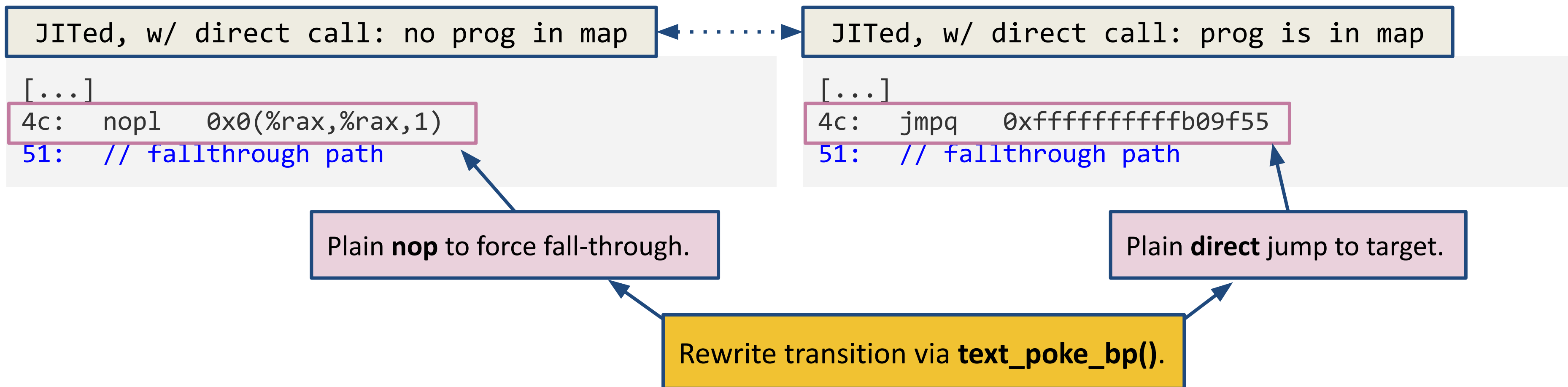
`JITed, w/ direct call: no prog in map`

```
[...]
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



- ➔ 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");
}
```

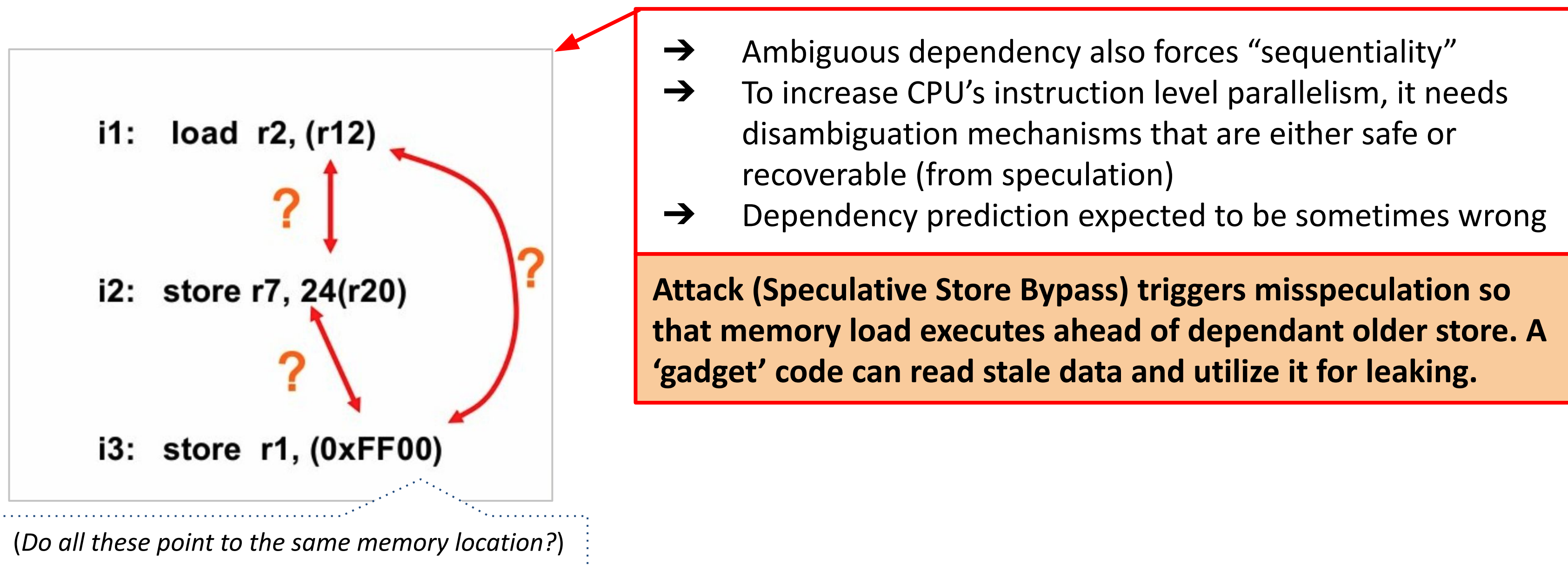
Given map & slot does not change, allows for direct jmp/nop transition in JIT.

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

BPF & Spectre v4

Memory disambiguator: memory dependence speculation

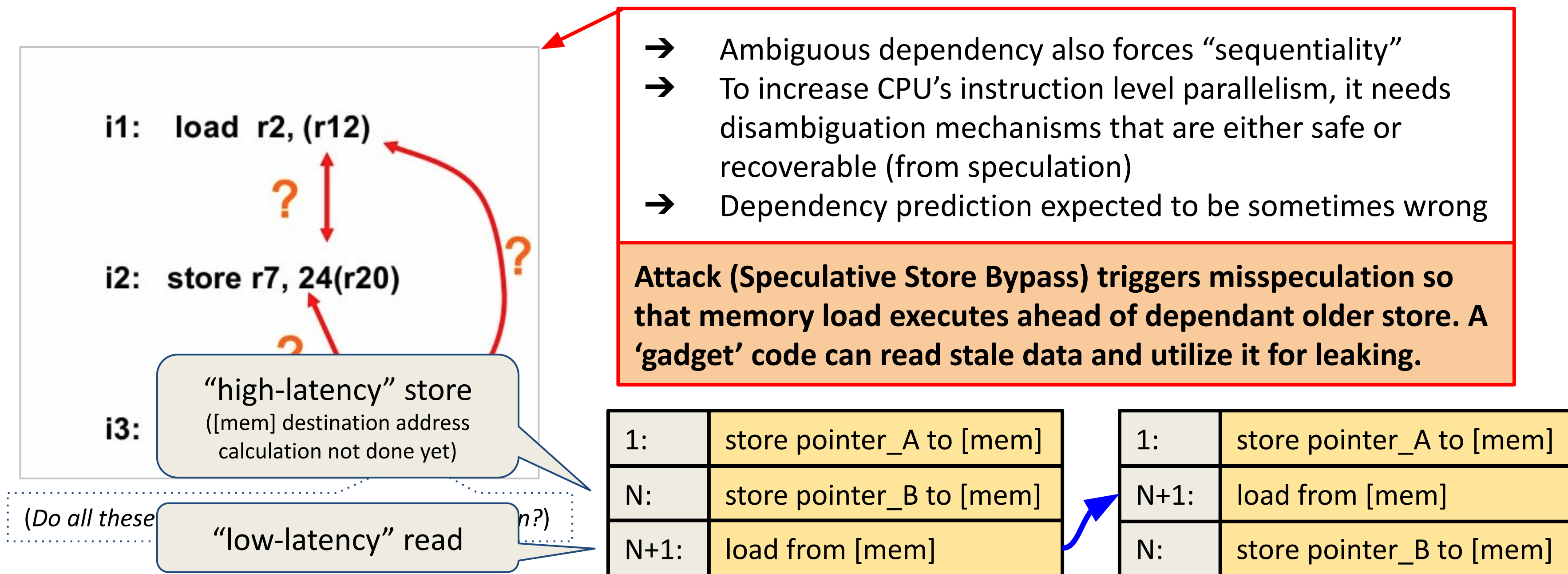
➔ Given OOO instruction execution, it predicts whether load depends on earlier store



BPF & Spectre v4

Memory disambiguator: memory dependence speculation

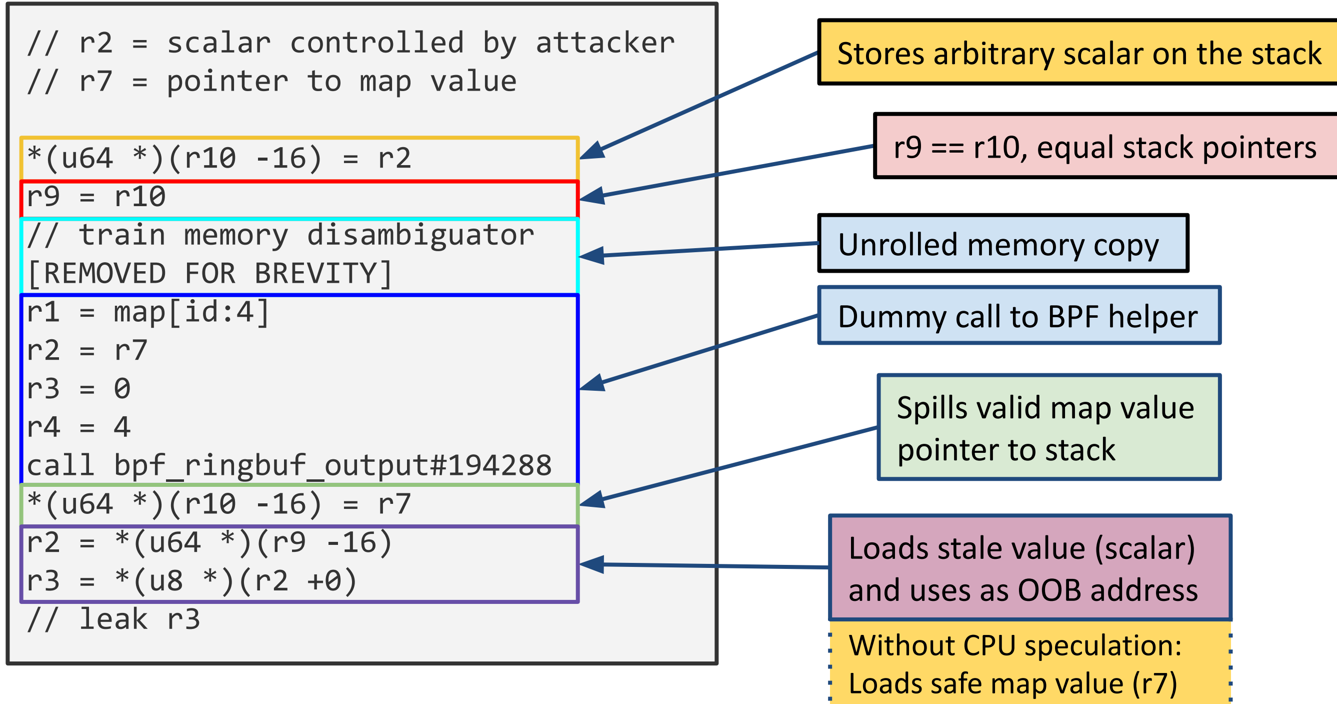
➔ Given OOO instruction execution, it predicts whether load depends on earlier store



(dependency misspeculation → unsafe reordering)

BPF & Spectre v4

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



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 memory disambiguator
[REMOVED FOR BREVITY]
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
```

Stores arbitrary scalar on the stack

r9 == r10, equal stack pointers

Unrolled memory copy

Dummy call to BPF helper

Spills valid map value pointer to stack

Loads stale value (scalar) and uses as OOB address

Without CPU speculation:
Loads safe map value (r7)

BPF & Spectre v4

Example attack in BPF: crafting 'fast' versus 'slow' registers

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

*(u64 *)(r10 -16) = r2
r9 = r10
// train memory disambiguator
[REMOVED FOR BREVITY]
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
```

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 pop latency, value of **r10** not available immediately on return

Hardware executes speculative loads while store waits for r10

Dummy call to BPF helper

Spills valid map value pointer to stack

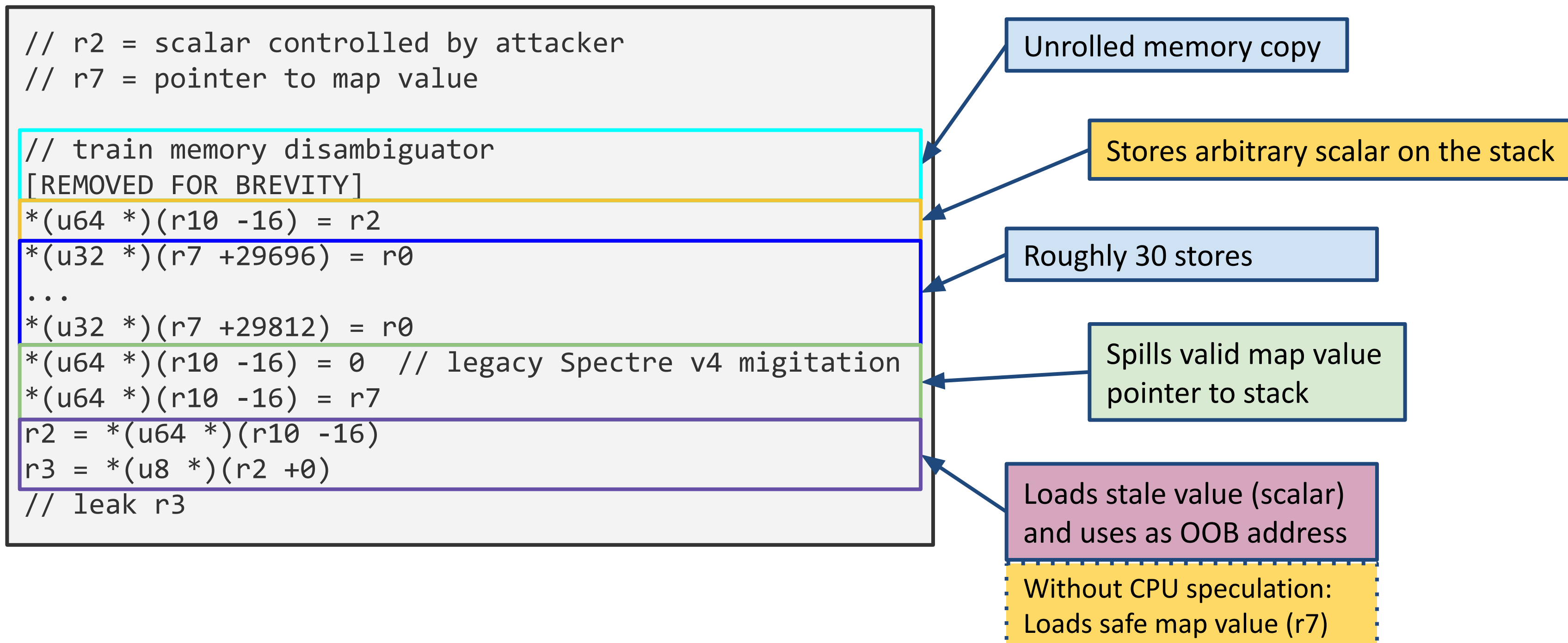
Loads stale value (scalar) and uses as OOB address

Without CPU speculation: Loads safe map value (r7)

But why we speculate?

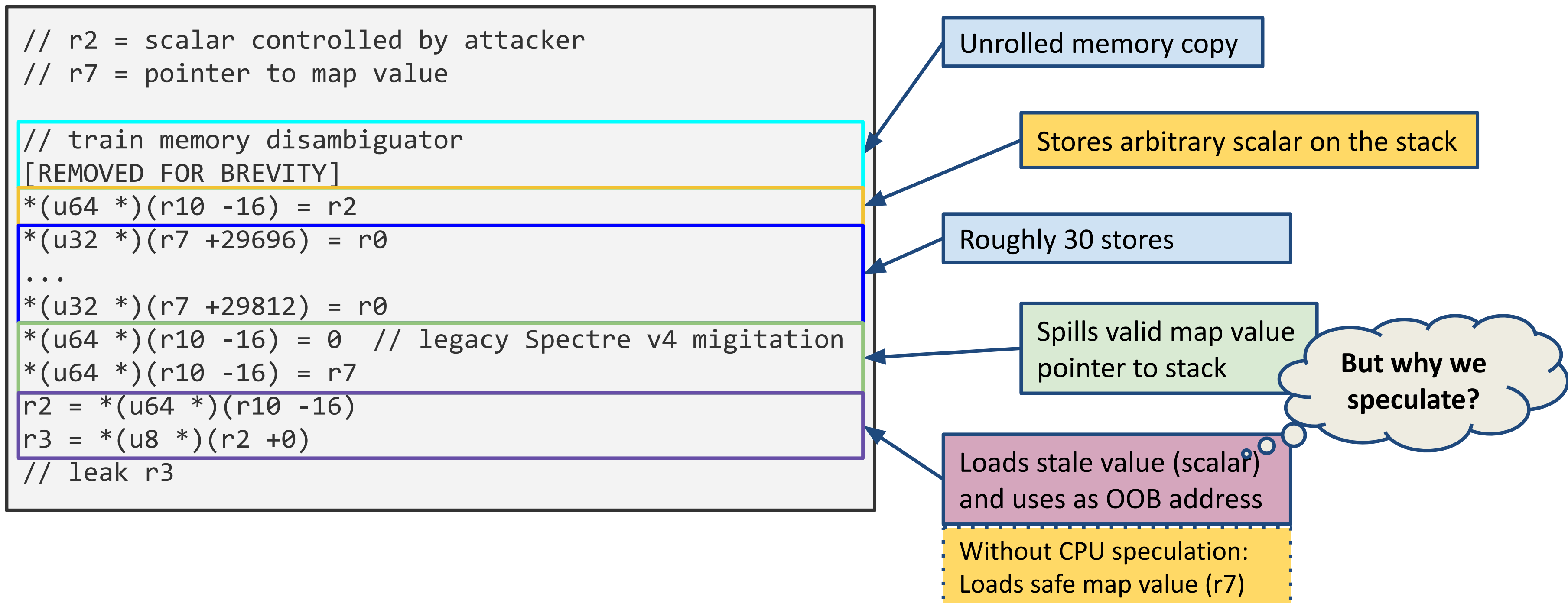
BPF & Spectre v4

Example attack in BPF: saturating “Store Address” ports



BPF & Spectre v4

Example attack in BPF: saturating "Store Address" ports



BPF & Spectre v4

Example attack in BPF: saturating “Store Address” ports

- ➔ How hardware executes loads and stores?

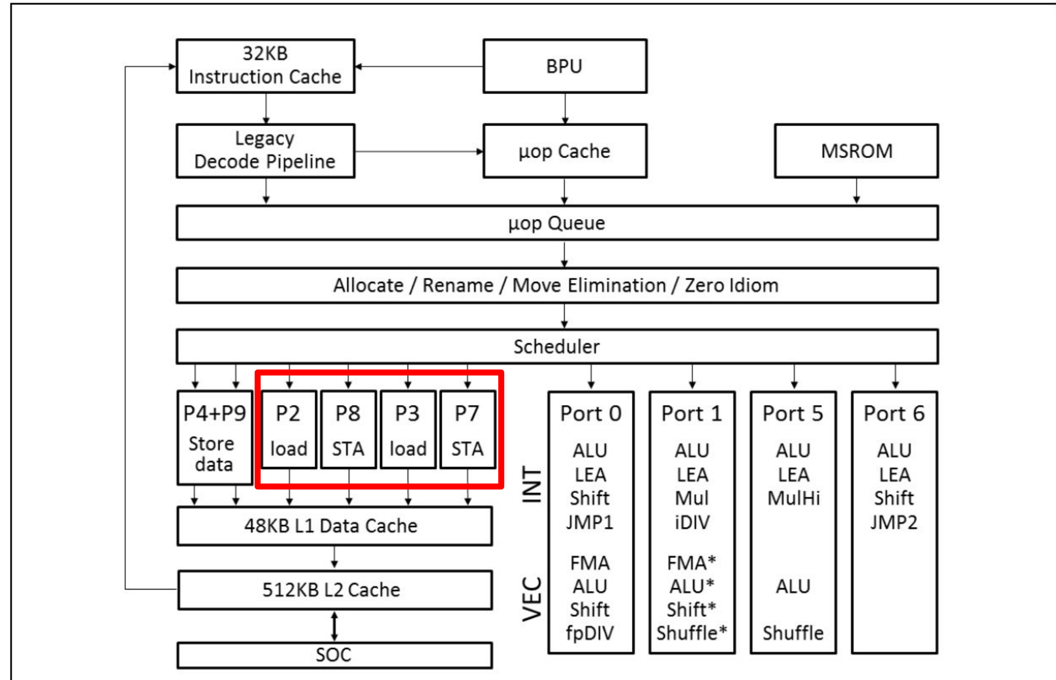


Figure 2-1. Processor Core Pipeline Functionality of the Ice Lake Client Microarchitecture[1]

[Intel, 2021]

BPF & Spectre v4

Example attack in BPF: saturating "Store Address" ports

➔ Dedicated (separate) ports to execute loads and compute store addresses

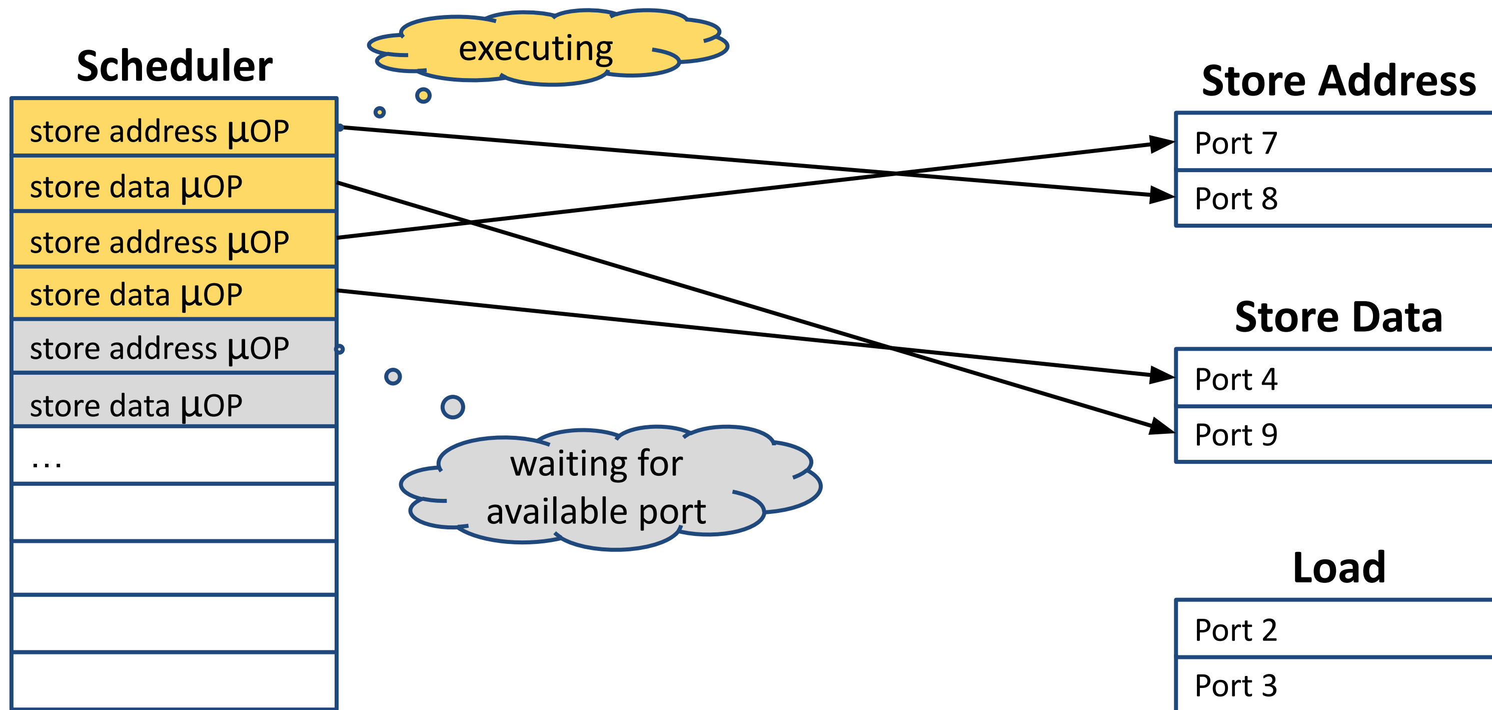
Table 2-1. Dispatch Port and Execution Stacks of the Ice Lake Client Microarchitecture

Port 0	Port 1[1]	Port 2	Port 3	Port 4	Port 5	Port 6	Port 7	Port 8	Port 9
INT ALU LEA INT Shift Jump1	INT ALU LEA INT Mul INT Div	Load	Load	Store Data	INT ALU LEA INT MUL Hi	INT ALU LEA INT Shift Jump2	Store Address	Store Address	Store Data
FMA Vec ALU Vec Shift FP Div	FMA* Vec ALU* Vec Shift* Vec Shuffle*				Vec ALU Vec Shuffle				

[Intel, 2021]

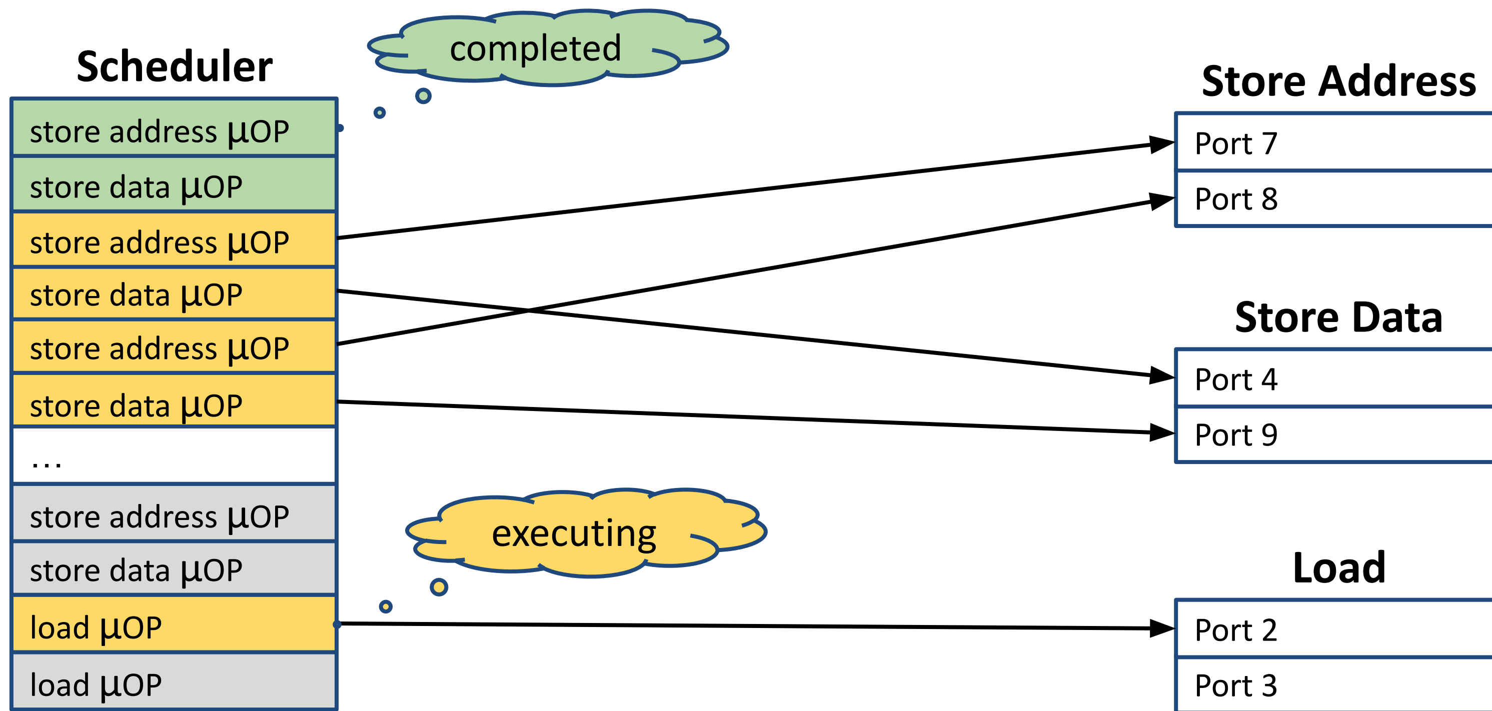
BPF & Spectre v4

Example attack in BPF: saturating "Store Address" ports



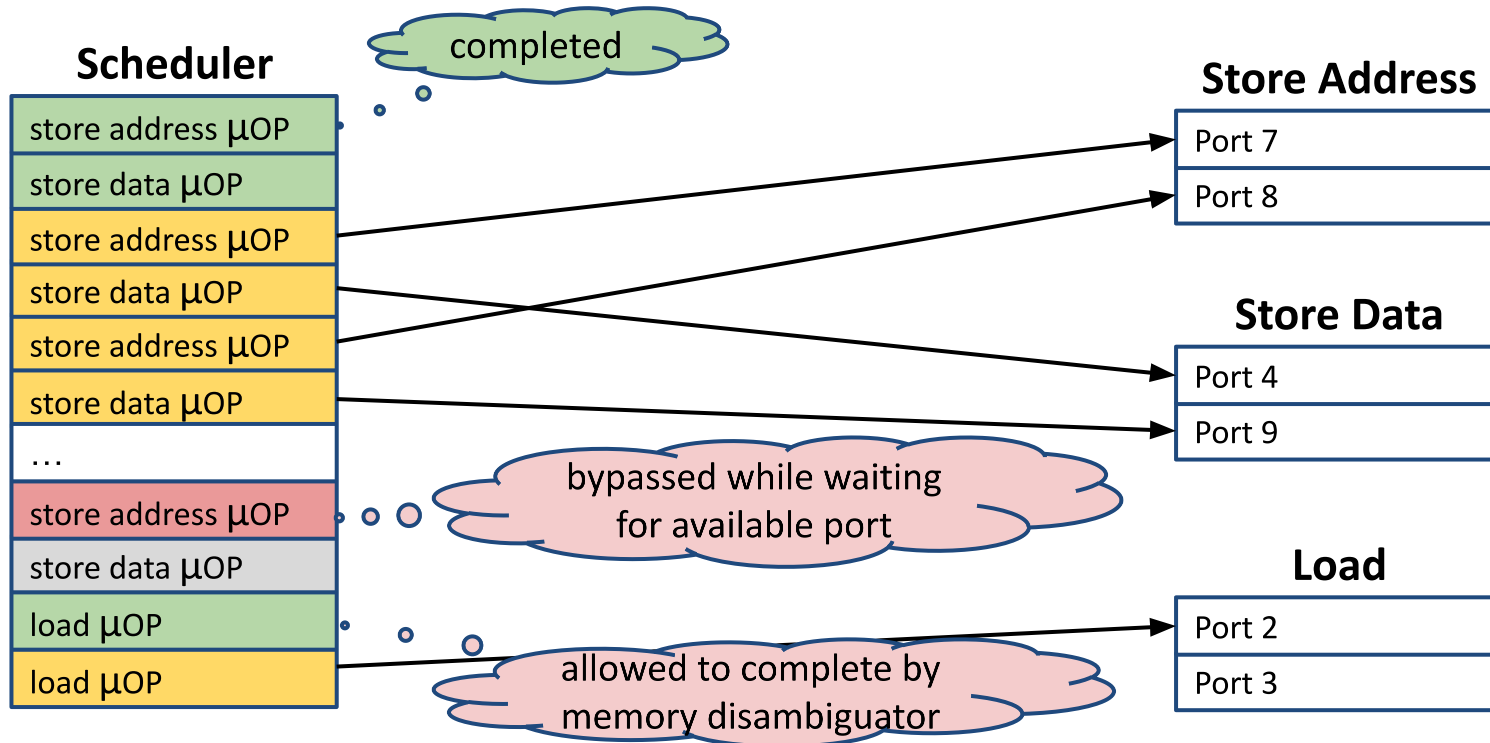
BPF & Spectre v4

Example attack in BPF: saturating “Store Address” ports



BPF & Spectre v4

Example attack in BPF: saturating “Store Address” ports



BPF & Spectre v4

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

Mitigated version:

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

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: **Unprivileged BPF is off by default**, see also /proc/sys/kernel/unprivileged_bpf_disabled and 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

Future Work

Core BPF is not perfect in terms of verifiability

- ➔ Only few constructs have been formally verified so far
  - ◆ However, research around BPF from academic community increasing
- ➔ Some operations, i.e. division on tnums, are not range tracked

Recently published work improves the multiplication range tracking [1]

- ➔ tnum multiplication now maintains more precise information
- ➔ Addition, subtraction, multiplication algorithm was formally proved

Document/formalize the current verification procedure

- ➔ Not much documentation how the verifier operates internally except for the source
- ➔ Challenging to get an overview of the verification structure from C code

Thank you!

Jann Horn (Google, Project Zero)

Adam Morrison (Tel Aviv University)

John Fastabend (Isovalent)

Alexei Starovoitov (Facebook)

… and whole BPF, netdev &
security research community!

(Appendix #1: Extract of BPF-related commits for more details on mitigation work,
Appendix #2: Extract of academic research related to BPF)

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”)

Appendix: Research related to BPF (extract)

"Sound, Precise, and Fast Abstract Interpretation with Tristate Numbers", Vishwanathan et al.

"Eliminating bugs in BPF JITs using automated formal verification", Nelson et al.

"A proof-carrying approach to building correct and flexible BPF verifiers", Nelson et al.

"Automatically optimizing BPF programs using program synthesis", Xu et al.

"Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions", Gershuni et al.

"An Analysis of Speculative Type Confusion Vulnerabilities in the Wild", Kirzner et al.