Microarchitectural Attacks and Defenses

EE282 Lecture 15

Nicholas Mosier

June 4, 2025

Key Tools for System Architects

- Pipelining
- Parallelism
- Out-of-order execution
- Prediction (or speculation)
- Locality & caching
- Indirection
- Amortization
- Redundancy
- Specialization
- Focus on the common case for efficiency
- Focus on the uncommon case for security

- **Isolation**

Lect. 18's focus: microarchitectural security of speculative, out-of-order processors

Lect. 1–17's main focus

Adapted from Lecture 1, slide 41

Outline

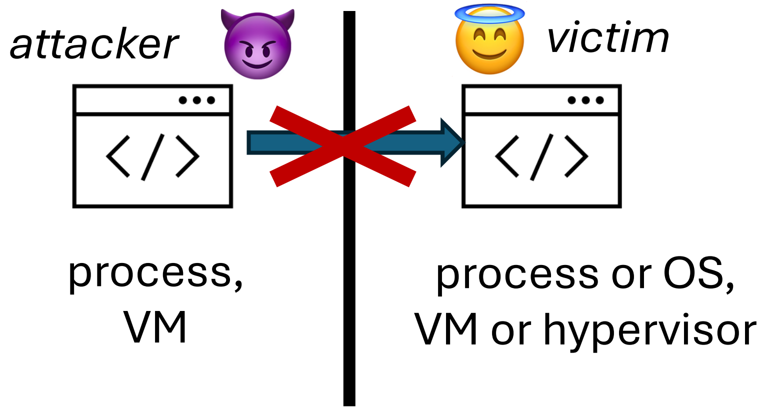
- Isolation
- Non-speculative side channel attacks and defenses
- Transient execution attacks and defenses overview
- Meltdown attacks and defenses
- Spectre attacks and defenses

Outline

- Isolation
- Non-speculative side channel attacks and defenses
- Transient execution attacks and defenses overview
- Meltdown attacks and defenses
- Spectre attacks and defenses

Isolation

- What does it mean for two entities to be **isolated**?
  - *Entity A (attacker) cannot observe or influence Entity B's(victim's) private/privileged state*
- What entities do we want to isolate?
  - Process vs. OS or other process
  - VM vs. hypervisor or other VM
  - Sandboxed program vs. sandbox runtime or other sandboxed program
  - *Any other ideas?*
- What resources/state do we want to isolate?
  - Memory
  - Register file (including MSRs)

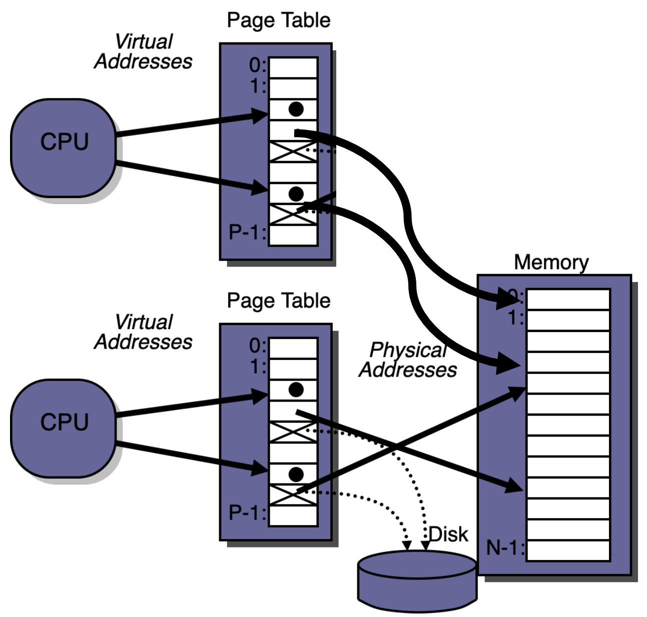


isolation:
attacker cannot observe or influence victim's private state

Architectural Isolation

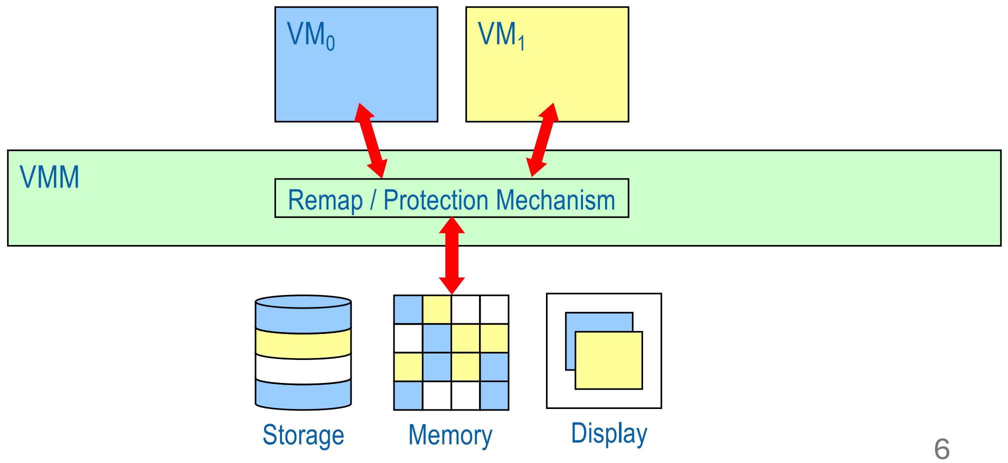
- **Architectural isolation**: two entites cannot directly access (read/write) each other's architectural state
  - Modern systems are generally designed to enforce architectural isolation
  - Approaches:
    - indirection (e.g., virtualization)
    - resource partitioning
    - privilege checks

Address space isolation via virtual memory



Lecture 15, slide 20

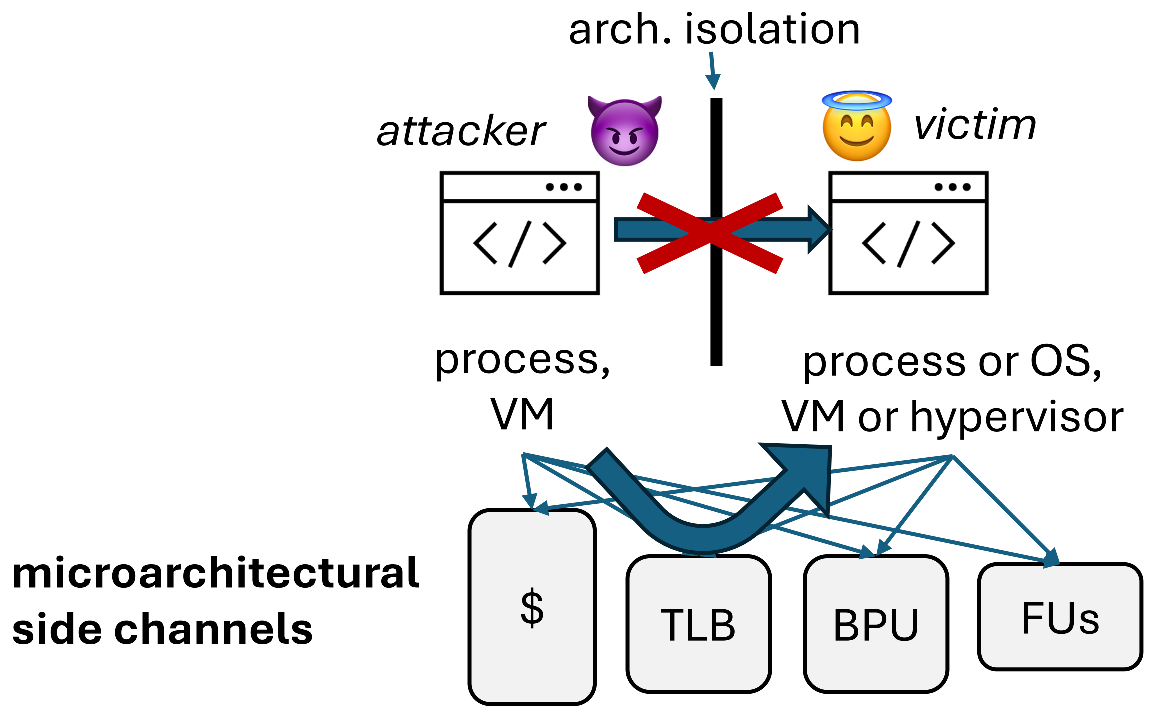
VM isolation via resource partitioning



Lecture 14, slide 16

No microarchitectural isolation

- **Microarchitectural isolation**: two entites cannot observe or influence each other's **microarchitectural** state
  - Modern systems generally **do not** enforce microarchitectural isolation
  - Non-architectural resources are rarely partitioned
  - E.g., the OS and user processes share the same caches, branch predictors, etc.



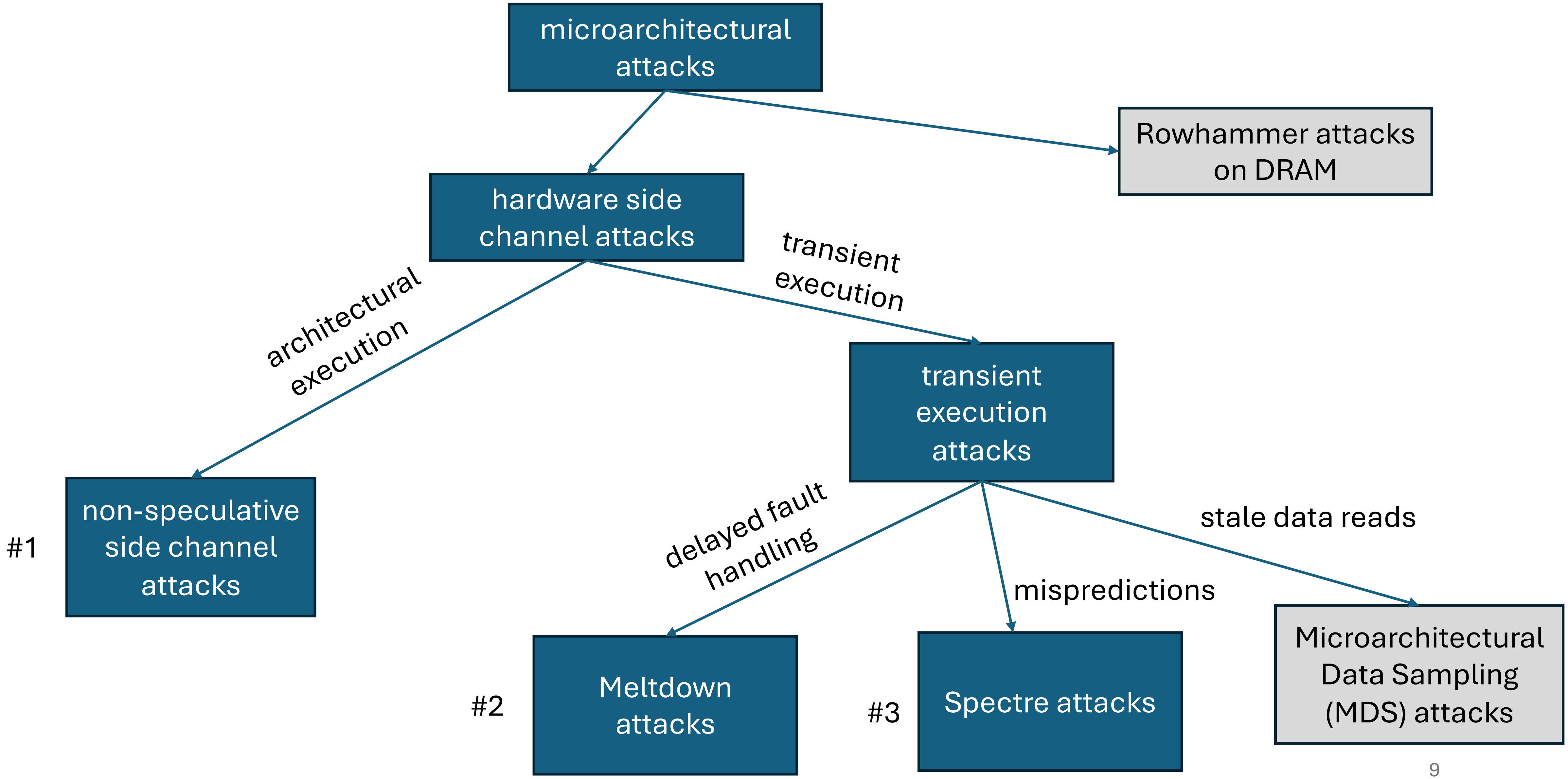
Root problem:

- victim's usage of microarchitectural resources visible to observer
- victim's usage of microarchitectural resources depends on private architectural state

Themes

- Architectural isolation ≠ microarchitectural isolation
- Architectural semantics vs. microarchitectural semantics
- Performance optimizations maintain architectural isolation for correctness but break microarhitectural isolation

Microarchitectural Attacks: Overview

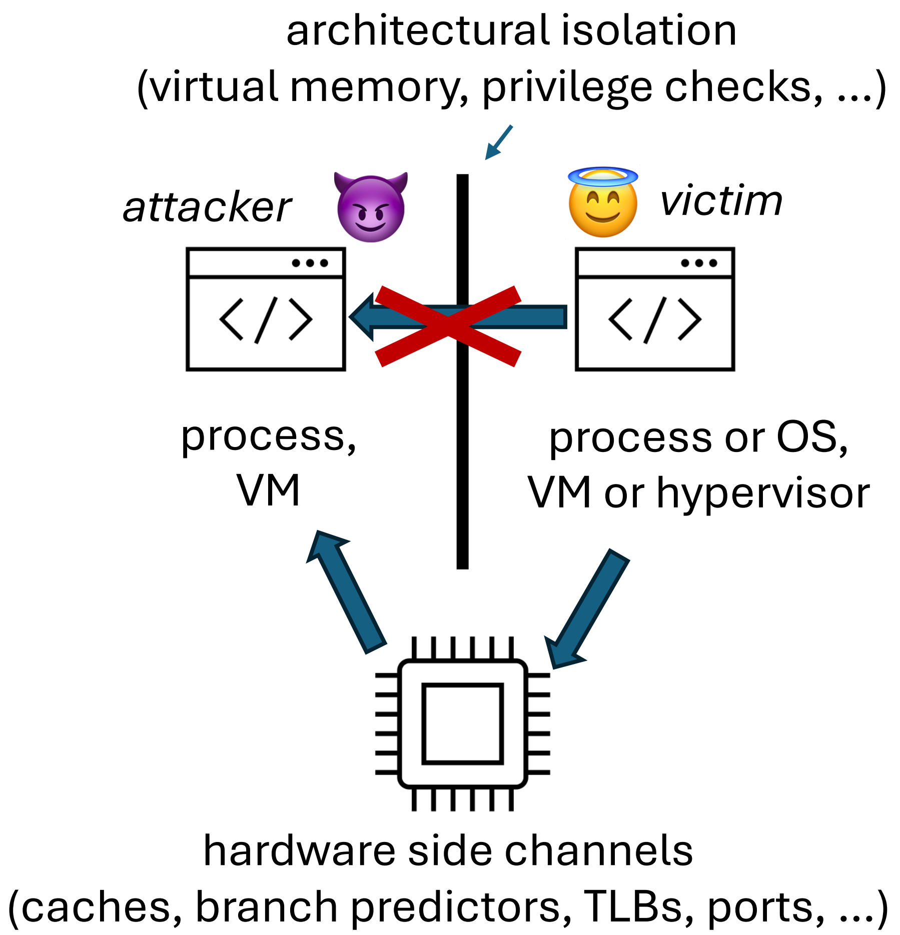

microarchitectural attacks
Rowhammer attacks on DRAM
hardware side channel attacks
architectural execution
transient execution
transient execution attacks
#1
non-speculative side channel attacks
delayed fault handling
mispredictions
stale data reads
#2
Meltdown attacks
#3
Spectre attacks
Microarchitectural Data Sampling (MDS) attacks

Outline

- Isolation
- Non-speculative side channel attacks and defenses
- Transient execution attacks and defenses overview
- Meltdown attacks and defenses
- Spectre attacks and defenses

Hardware Side Channels

- **Hardware side channels**: a shared hardware resource that the victim modulates as a function of its architectural state
  - The attacker then infers the victim's architectural state by observing the victim's channel modulation
- Data **leaks** if it is exposed via a hardware side channel



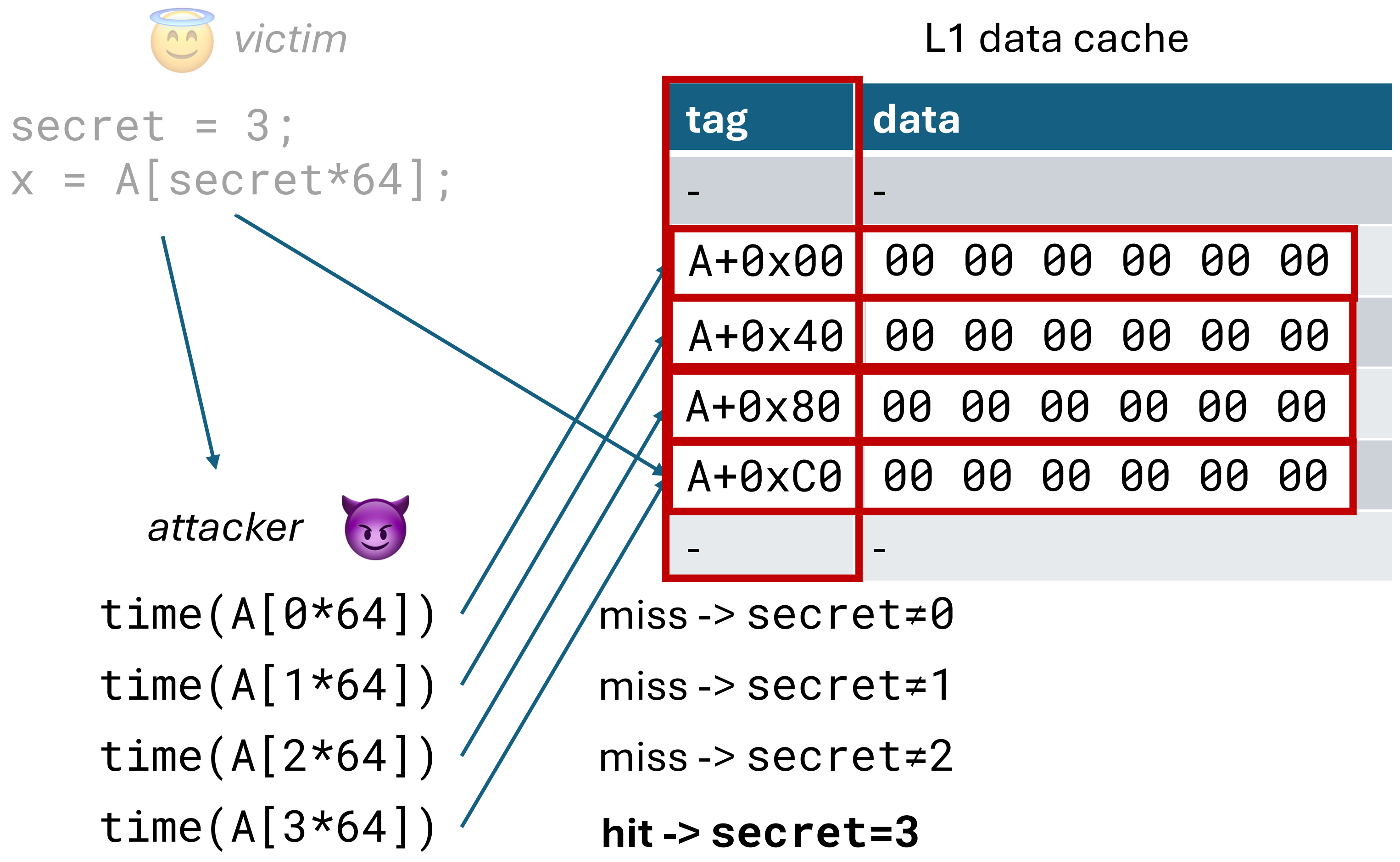
Microarchitectural Side Channel Example: data caches

What *specific* shared microarchitectural state in the cache does the victim modulate and the attacker observe?

(assume `A[ ]` is located in shared memory, but `secret` is private)

Example of a **Flush+Reload** attack [Yarom+ USENIX'14]

victim: read miss for address `A+0xC0`
victim: cache line at `A+0xC0`
context switch
attacker: read A+0x00 -> miss
attacker: read A+0x40 -> miss
attacker: read A+0x80 -> miss
attacker: read A+0xC0 -> hit!



```
secret = 3;
x = A[secret*64];
```

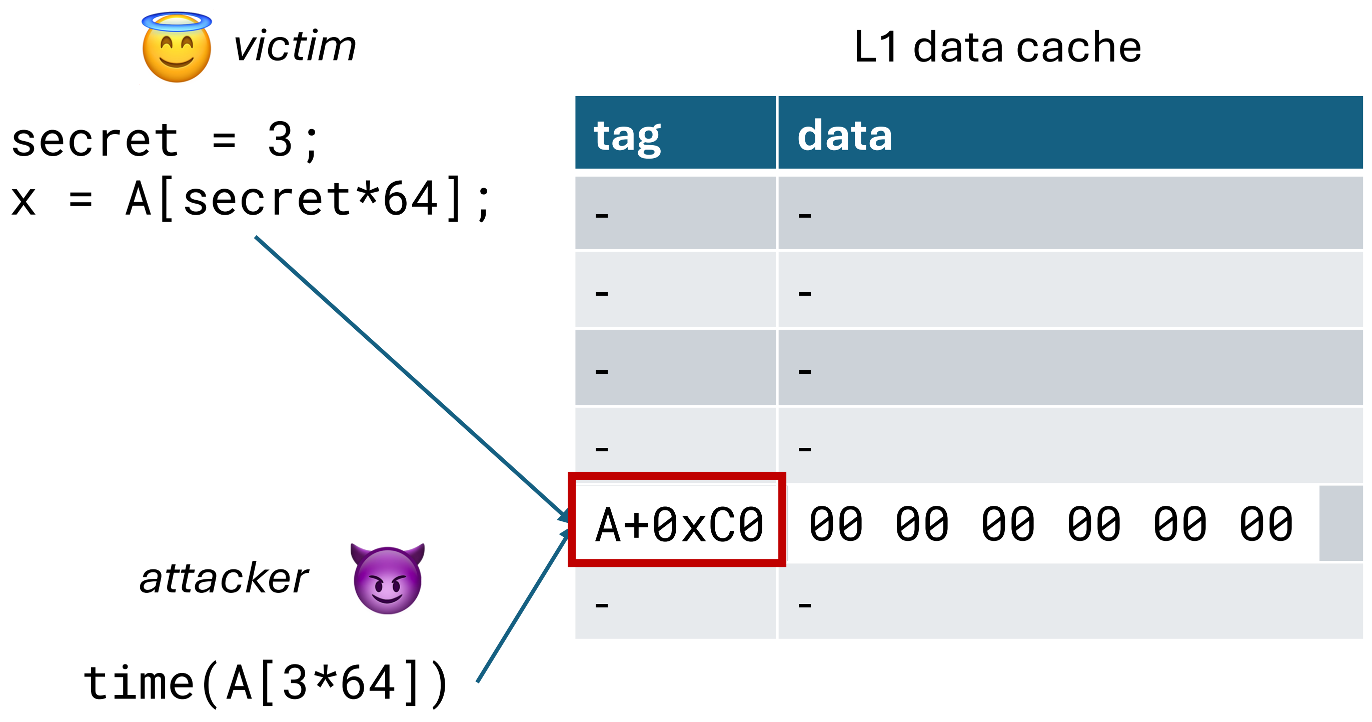
tag	data
-	-
A+0x00	00 00 00 00 00 00
A+0x40	00 00 00 00 00 00
A+0x80	00 00 00 00 00 00
A+0xC0	00 00 00 00 00 00
-	-

`time(A[0*64])`	miss -> `secret≠0`
`time(A[1*64])`	miss -> `secret≠1`
`time(A[2*64])`	miss -> `secret≠2`
`time(A[3*64])`	**hit -> `secret=3`**

victim's secret leaks to the attacker via the cache

Hardware Side Channel Example: data caches

- **Root problem**: attacker can microarchitecturally read victim virtual address tags
    - **data-flow** from victim to attacker
- No easy solutions
    - Augment tag with process ID?
        - Not helpful when attacker + victim may share physical memory...
    - Flush cache on context switch?
        - High overhead



```
secret = 3;
x = A[secret*64];
```

```
time(A[3*64])
```

tag	data
-	-
-	-
-	-
-	-
A+0xC0	00 00 00 00 00 00
-	-

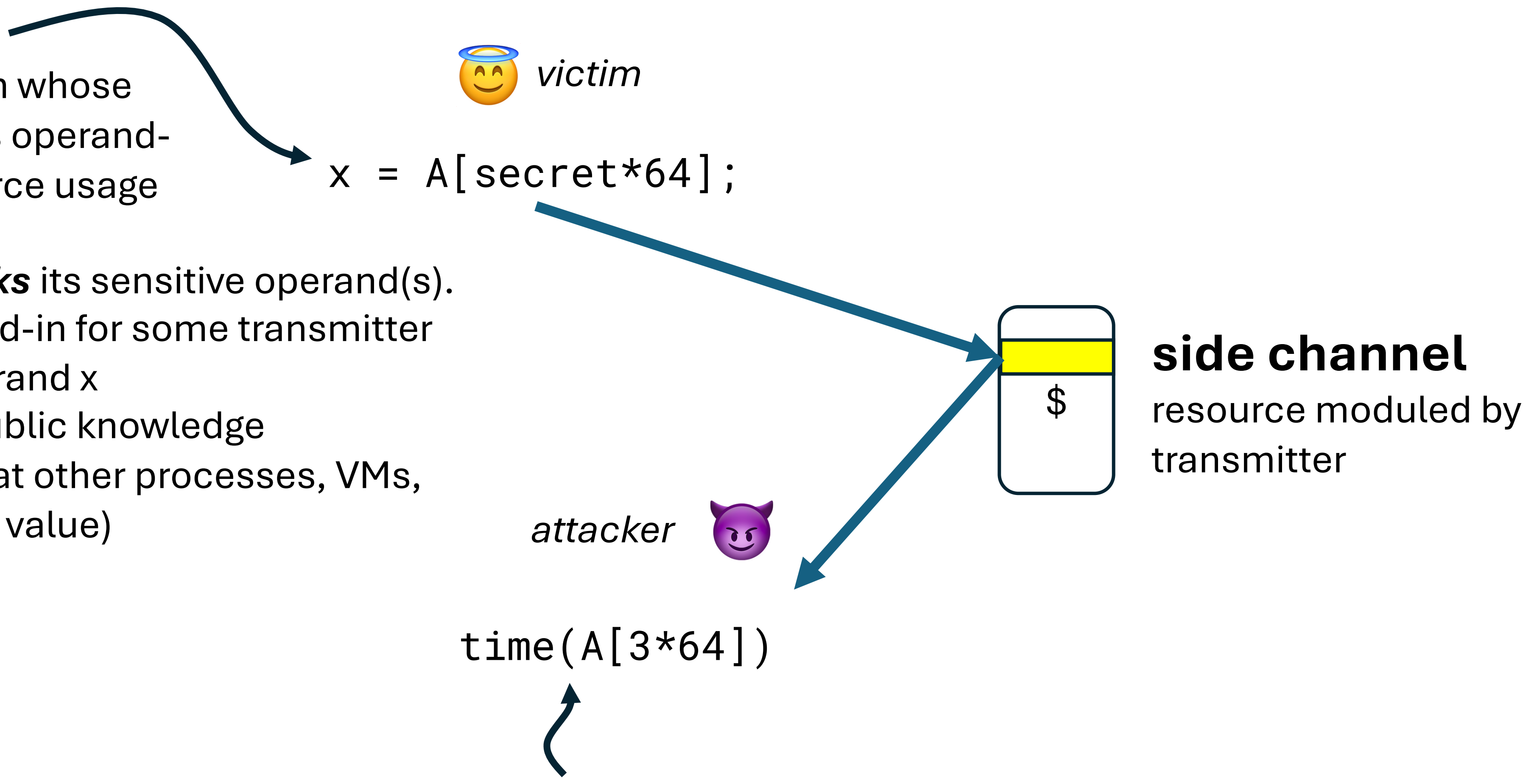
More Hardware Side Channels

- TLBs (tag bits expose victim memory addresses)
- Branch predictors (predictor state exposes victim instruction addresses / control-flow path)
- Port contention (exposes victim instruction types)
- Execution time (variable-time instructions, like division)
- Any other ideas?

Side Channel Terminology

transmitter
unsafe instruction whose execution creates operand-dependent resource usage

- A transmitter ***leaks*** its sensitive operand(s).
- "leak(x)" is a stand-in for some transmitter that leaks its operand x
- **Leaked data** = public knowledge (must assume that other processes, VMs, etc. may know its value)



side channel
resource moduled by transmitter

receiver
observer of side channel

Transmitters on Modern Microarchitectures

- Loads and stores: leak their address operand
  ```
  ld rd, [rs1] // leaks rs1
  st rd, [rs1] // leaks rs1
  ```
- Conditional branches: leak their condition/target operand
  ```
  bnz rs1, <label> // partially leaks rs1
  jalr rs1         // leaks rs1
  ```
- Variable-latency operations (floating-point, division, zero-skip multiplier, …): leak a function of their inputs based on which paths are taken
  ```
  fadd fd, fs1, fs2 // partially leaks fs1, fs2
  div rd, rs1, rs2  // partially leaks rs1, rs2
  ```

Side Channel Attacks: 2 Real World Examples

Side channel attack on AES
(Daniel J Bernstein, 2005)

```
SoftAesBlock
softaes_block_encrypt(
  const SoftAesBlock block,
  const SoftAesBlock rk)
{
  uint8_t ix0[4];
  /* ... */
  out.w0 = LUT0[ix0[0]];
  out.w1 = LUT0[ix0[1]];
  out.w2 = LUT0[ix0[2]];
  out.w3 = LUT0[ix0[3]];
  /* ... */
}
```

secret

loads (transmitters)
leak secrets in ix0
via the cache

softaes_block_encrypt leaks
the AES secret key to attacker

libsodium

src/libsodium/crypto_core/softaes/softaes.c

Side channel attack on RSA square-and-multiply

secret

```
int BN_mod_exp_mont(u8 *p, /* ... */) {
  /* ... */
  for (;;) {
    if (BN_is_bit_set(p, wstart) == 0) {
      /* square */
      continue;
    }
    /* square and multiply */
    if (!bn_mul_mont_fixed_top(
          r, r, val[wvalue >> 1], mont, ctx))
      goto err;
  }
}
```

conditional branch (transmitter)
leaks each bit of p (via i-cache,
branch predictor, execution time)

BN_mod_exp_mont leaks the
RSA private key to attacker

openSSL LIBRARY

crypto/bn/bn_exp.c

Techniques for Mitigating Side Channel Attacks

- Hardware architects have adopted two hardware-software codesign techniques to avoid side-channel leakage of secrets
- **Approach #1 (new ISA):** replace leaky software primitives with secure hardware primitives
- **Approach #2 (leakage contracts):** expose a side-channel leakage contract to software
- Notably absent: disabling leaky optimizations

Side-Channel Mitigtion Technique #1: new ISA

Approach #1 (new ISA): implement side-channel-vulnerable primitives in hardware

Example: x86 ISA extensions for performing AES encryption/decryption in hardware (*Intel AESNI*, *AMD AES*)

vulnerable: software, secret-dependent loads

```
softaes_block_encrypt(block, rk)
{
  uint8_t ix0[4];
  /* ... */
  out.w0 = LUT0[ix0[0]];
  out.w1 = LUT0[ix0[1]];
  out.w2 = LUT0[ix0[2]];
  out.w3 = LUT0[ix0[3]];
  /* ... */
}
```

secure: hardware, no secret-dependent loads

```
_mm_aesenc(block, rk);
```

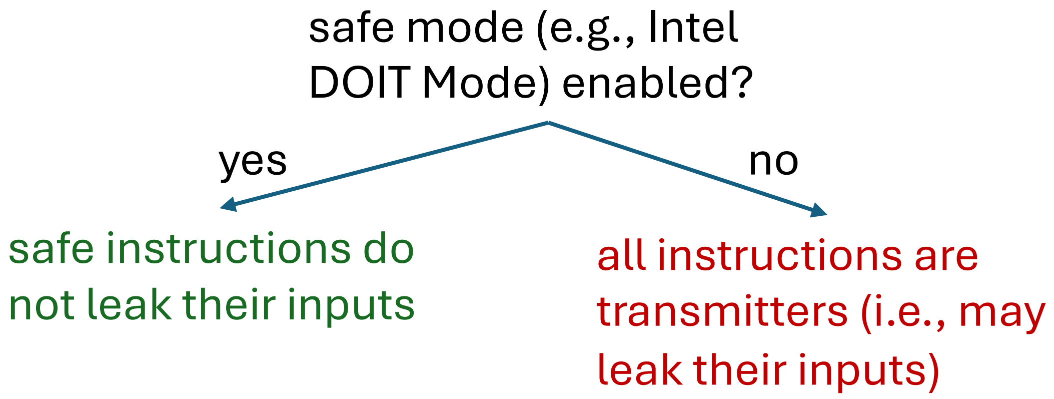
- specialized solution, applied to specific algorithms
- hardware implementation is guaranteed to use constant time/resources

Side-Channel Mitigtion Technique #2: leakage contracts

Approach #2 (leakage contracts): expose a side-channel leakage contract to software

Examples: Intel DOIT, Arm DIT, RISC-V ZKT

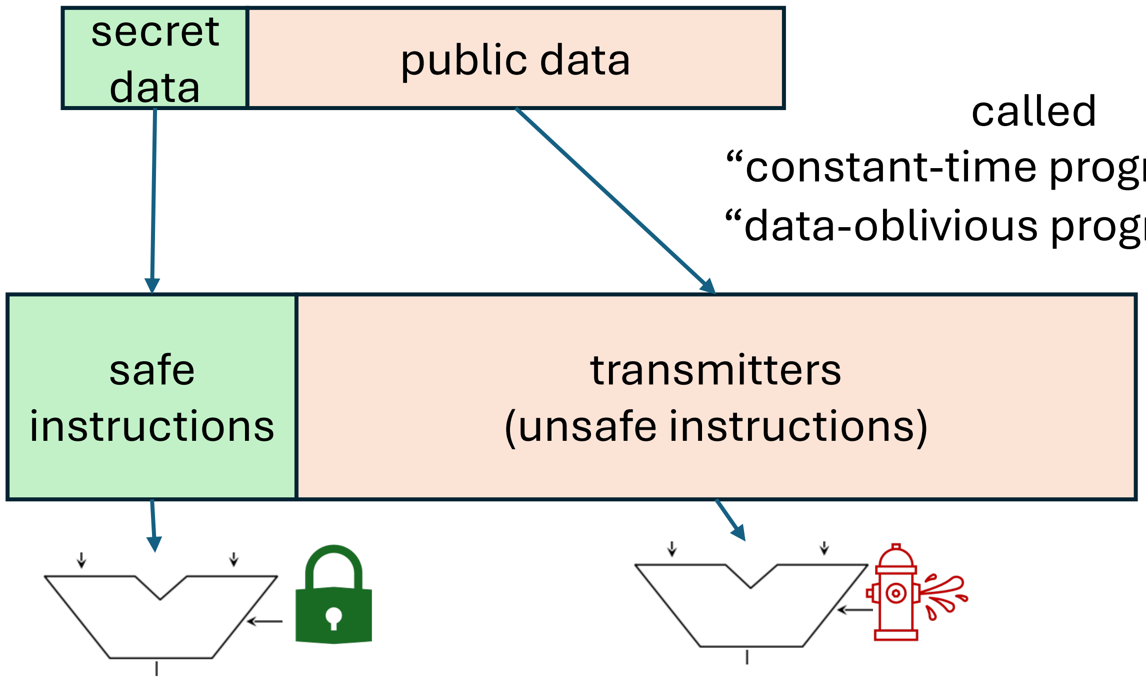
- Introduces new “safe” processor mode defining “safe” instructions
    - all other instructions are transmitters
- Hardware promises to not leak inputs to safe instructions via side channels
- Software promises to pass all secret data to safe instructions only
    - called *constant-time programming* or *data-oblivious programming*


safe mode (e.g., Intel DOIT Mode) enabled?
yes
no
safe instructions do not leak their inputs
all instructions are transmitters (i.e., may leak their inputs)

Side-Channel Mitigtion Technique #2: leakage contracts

<u>safe mode (e.g., Intel DOIT Mode) enabled</u>

victim (e.g., crypto code)



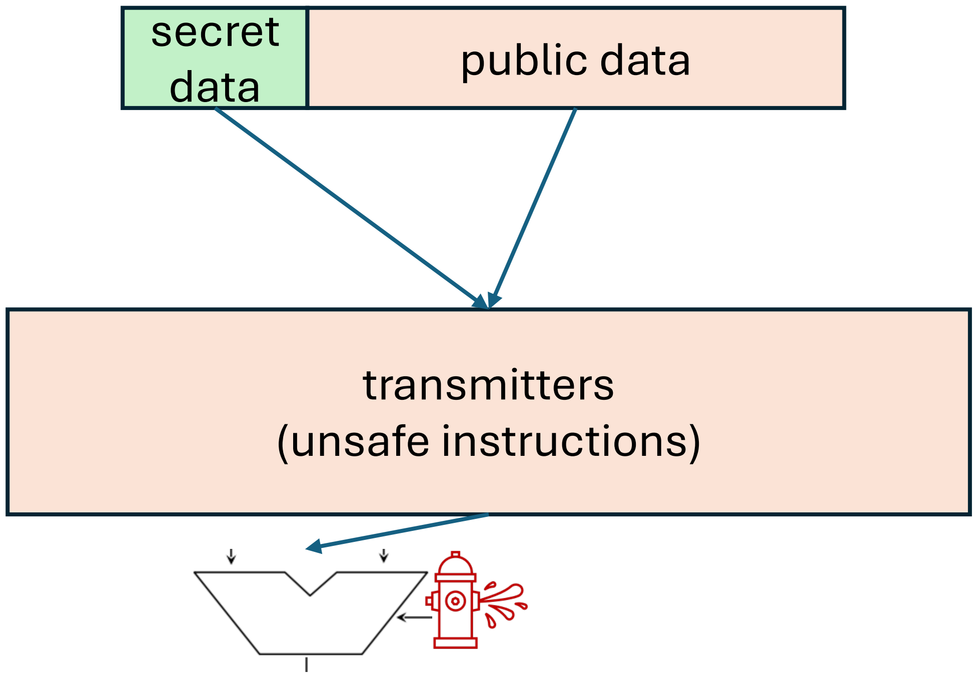
called
“constant-time programming”
“data-oblivious programming”

safe (data-independent)
execution + resource usage

unsafe (data-dependent)
execution + resource usage

<u>safe mode disabled (default)</u>

victim (e.g., crypto code)



unsafe (data-dependent)
execution + resource usage

Challenges of Mitigating Side Channels

“New ISA” approach

- not general: implemented on a per-algorithm/application basis
- not sustainable: cannot introduce new ISA extension for every leaky algorithm

“Leakage contract” approach

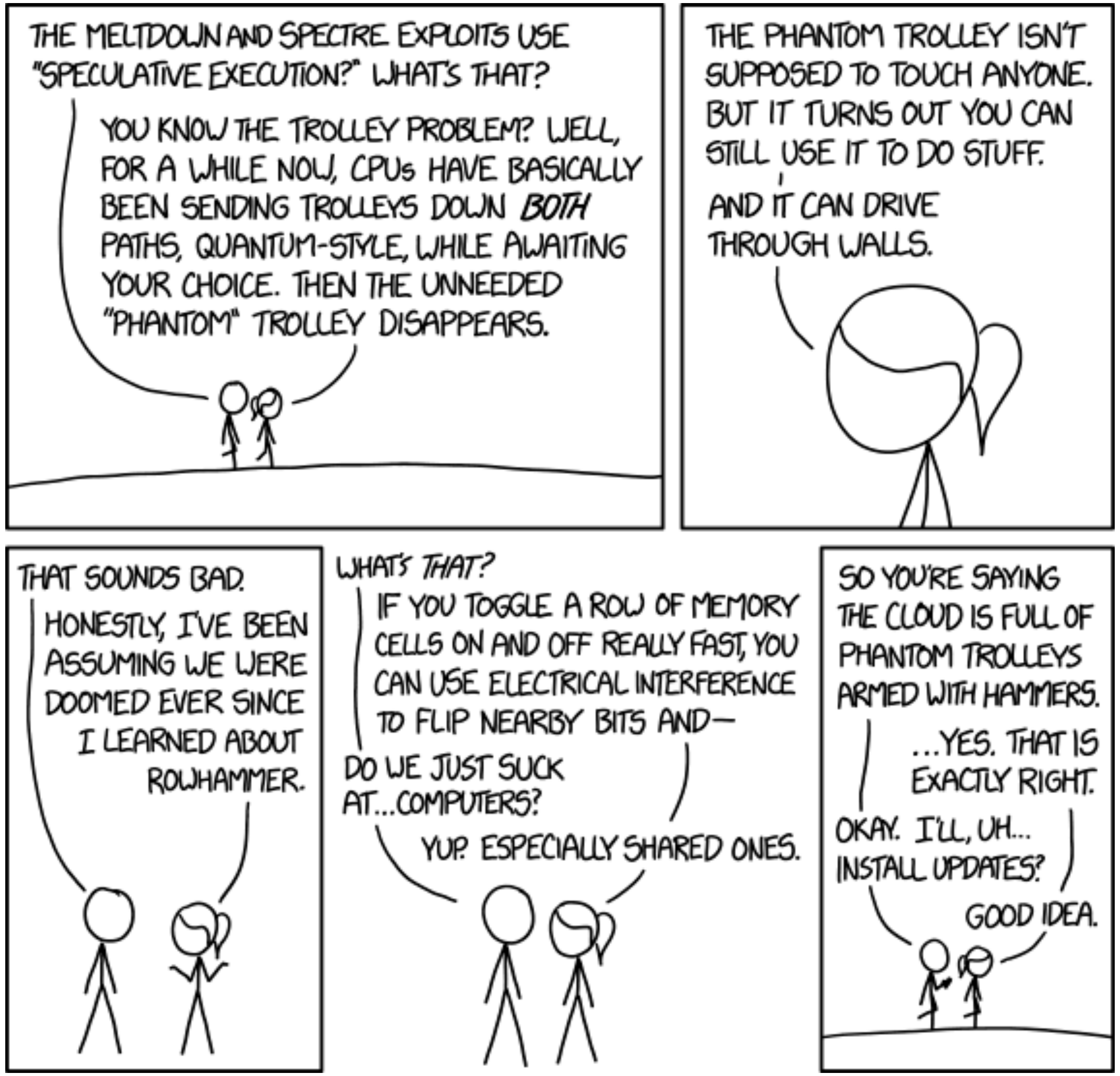
- declares transmitters, but does not eliminate side channels
- loads, stores, branches, variable-time ops still labeled as unsafe transmitters
- Intel **discourages** use of DOIT (safe) mode outside cryptographic applications

Recap of Side Channel Attacks

- Attacker indirectly reads victim’s shared microarchitectural state to leak victim’s architectural state
- Comprehensive defenses against non-speculative side channel attacks require cooperation of both hardware and software

Outline

- Isolation
- Non-speculative side channel attacks and defenses
- Transient execution attacks and defenses overview
- Meltdown attacks and defenses
- Spectre attacks and defenses


THE MELTDOWN AND SPECTRE EXPLOITS USE "SPECULATIVE EXECUTION?" WHAT'S THAT?
YOU KNOW THE TROLLEY PROBLEM? WELL, FOR A WHILE NOW, CPUs HAVE BASICALLY BEEN SENDING TROLLEYS DOWN BOTH PATHS, QUANTUM-STYLE, WHILE AWAITING YOUR CHOICE. THEN THE UNNEEDED "PHANTOM" TROLLEY DISAPPEARS.
THE PHANTOM TROLLEY ISN'T SUPPOSED TO TOUCH ANYONE. BUT IT TURNS OUT YOU CAN STILL USE IT TO DO STUFF.
AND IT CAN DRIVE THROUGH WALLS.
THAT SOUNDS BAD.
HONESTLY, I'VE BEEN ASSUMING WE WERE DOOMED EVER SINCE I LEARNED ABOUT ROWHAMMER.
WHAT'S THAT?
IF YOU TOGGLE A ROW OF MEMORY CELLS ON AND OFF REALLY FAST, YOU CAN USE ELECTRICAL INTERFERENCE TO FLIP NEARBY BITS AND—
DO WE JUST SUCK AT...COMPUTERS?
YUP. ESPECIALLY SHARED ONES.
SO YOU'RE SAYING THE CLOUD IS FULL OF PHANTOM TROLLEYS ARMED WITH HAMMERS.
...YES. THAT IS EXACTLY RIGHT.
OKAY. I'LL, UH... INSTALL UPDATES?
GOOD IDEA.

(Beware: some factual inaccuracies)

https://xkcd.com/1938/

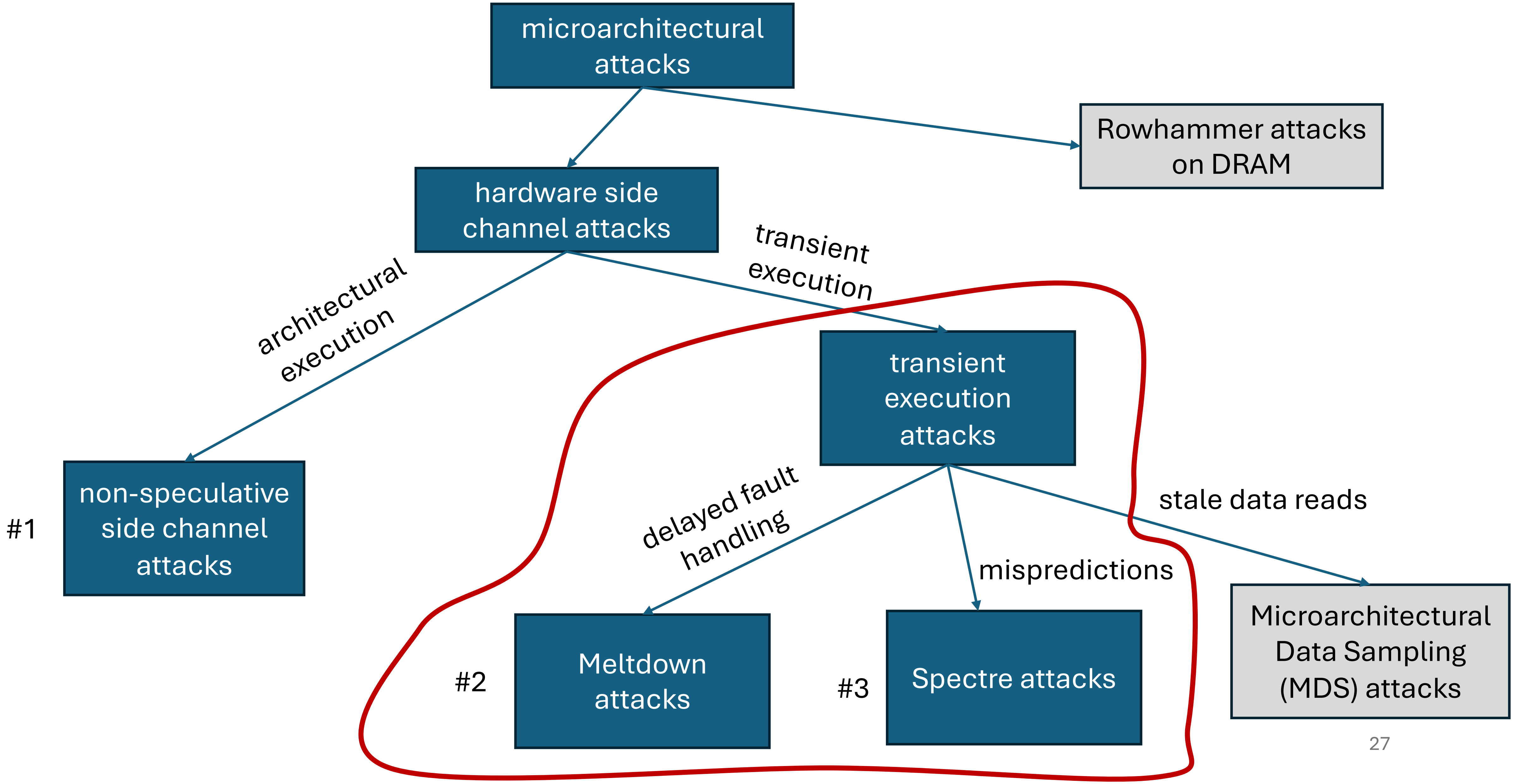
Reminder: processor speculation

- Prediction (control-flow, data-flow):
  - Increase ILP by breaking dependencies
- Delayed exception handling
  - Ensure precise exceptions

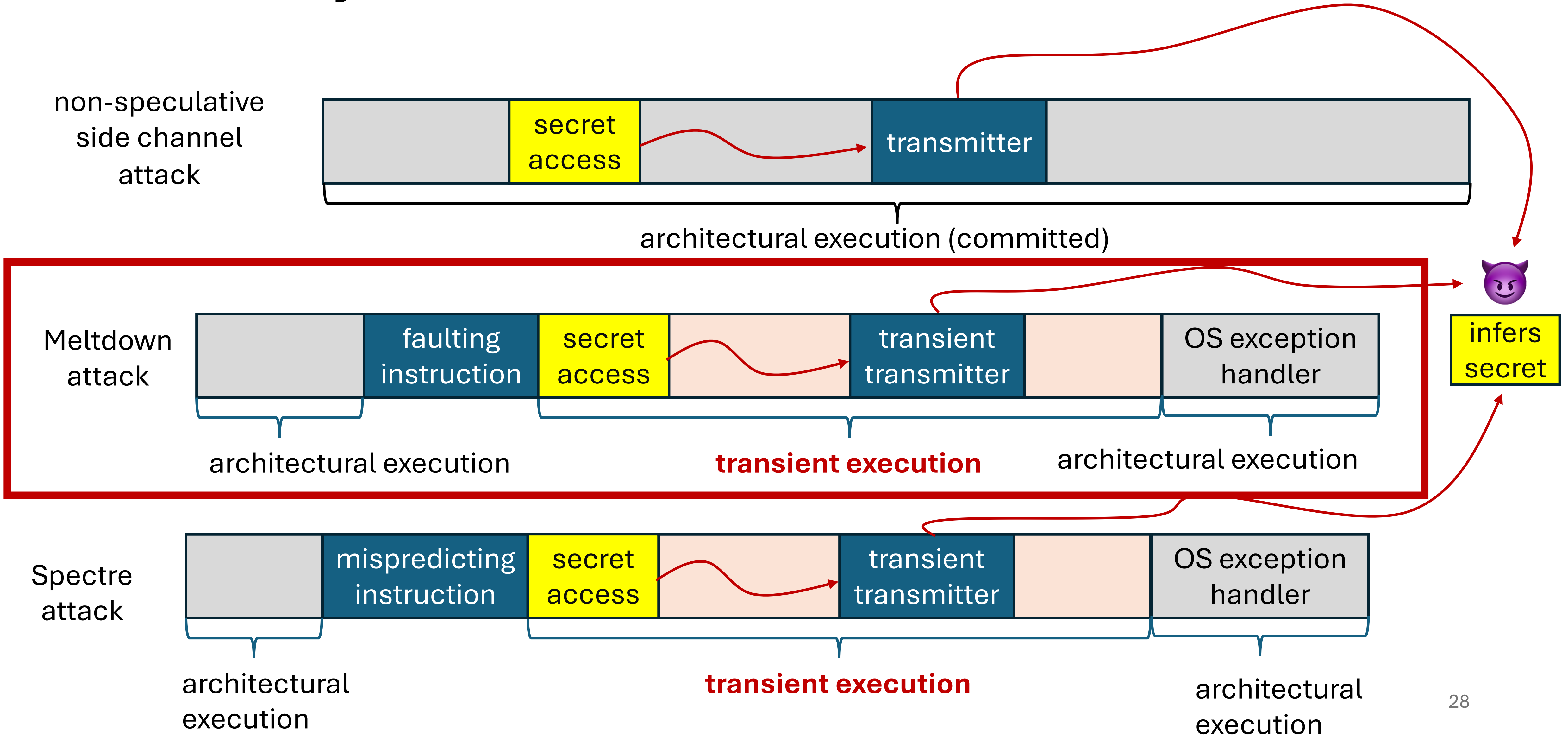
Definition — **Transient execution**: the execution of instructions that are not architecturally committed, due to a prior misprediction or exception.

Definition — **Architectural execution**: the execution of instructions that are architecturally committed.

Microarchitectural Attacks: Overview


microarchitectural attacks
Rowhammer attacks on DRAM
hardware side channel attacks
architectural execution
transient execution
transient execution attacks
#1
non-speculative side channel attacks
delayed fault handling
mispredictions
stale data reads
#2
Meltdown attacks
#3
Spectre attacks
Microarchitectural Data Sampling (MDS) attacks

Anatomy of Transient Execution Attacks



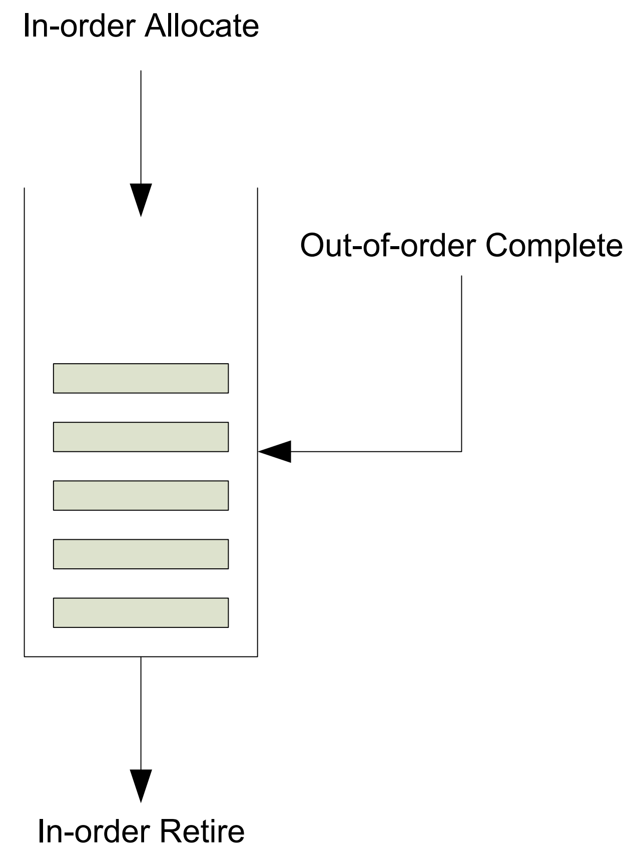
Outline

- Isolation
- Non-speculative side channel attacks and defenses
- Transient execution attacks and defenses overview
- Meltdown attacks and defenses
- Spectre attacks and defenses

Recall: precise exception handling
(Lect. 7, slides 85–87)

- **Precise exceptions**: when exception occurs in instruction `I`…
  - Instruction `I` has not modified the architectural state
  - All older instructions have fully completed
  - No younger instruction has modified the architectural state
- **Enforced via reorder buffer (ROB)**:
  - FIFO for instruction tracking
  - 1 entry per instruction, in program order
    - PC, register/memory address, new value, ready, **exception**
  - In general, ROB used for **undoing transient execution**

ROB algorithm



ROB entry

PC	Dst Value	Dst Address	Ready	Exception Info	Type

Delayed Exception Handling

- Way to implement precise exception handling using the ROB
- Only handle exceptions for instruction at the head of the ROB
  - Allow younger instructions to execute
  - **Faulting instruction may speculatively write back a "bad" result, and younger instructions may transiently compute on it**

```
1000: ld r1, [good_vaddr]
1004: ld r2, [bad_vaddr]
1008: add r3, r2, 1
```

execute I1 (#PF)
execute I2
execute I0
commit I0
squash I1, I2
jump to OS page fault handler

ID	PC	Dst Value	Dst Address	Ready	Exception Info	Type
→ I0	1000	0	r1	1	(none)	NOP
→ ~~I1~~	~~1004~~	~~???~~	~~r2~~	~~1~~	~~#PF: page fault~~	~~LD~~
~~I2~~	~~1008~~	~~???~~	~~r3~~	~~1~~	~~(none)~~	~~ADD~~

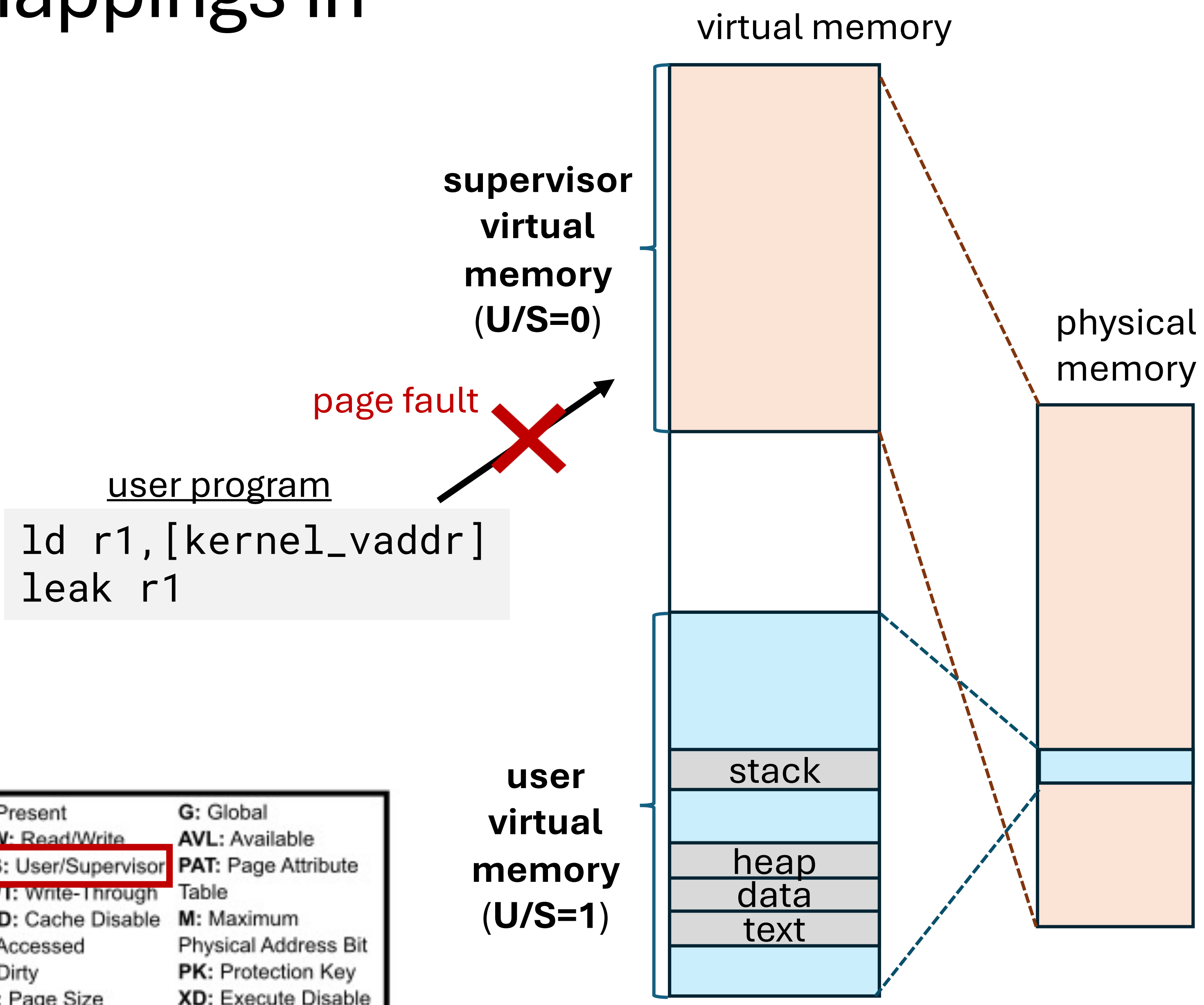
Why would we want to do this?

Why would we *not* want to do this?

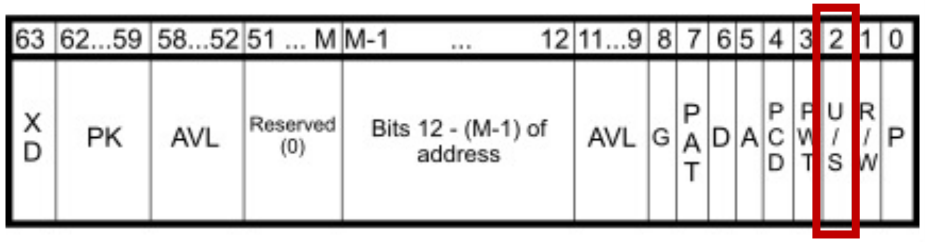
pollev....

Background: kernel mappings in user page tables

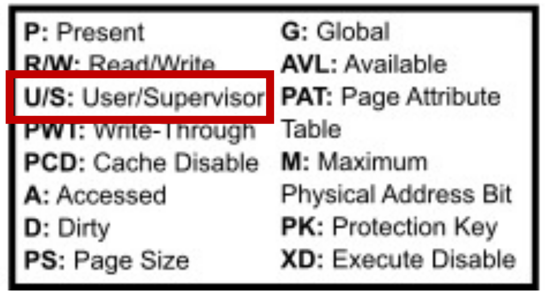
- To avoid switching page tables on every user → OS transition, the OS maps kernel virtual pages into user page tables
- U/S bit: whether page is accessible from userspace (0=no, 1=yes)
- Page fault on user access to kernel page with U/S=0, ensuring architectural isolation



```
ld r1,[kernel_vaddr]
leak r1
```



63	62...59	58...52	51 ... M	M-1 ... 12	11...9	8	7	6	5	4	3	2	1	0
XD	PK	AVL	Reserved (0)	Bits 12 - (M-1) of address	AVL	G	PAT	D	A	PCD	PWT	U/S	R/W	P



P: Present	**G:** Global
R/W: Read/Write	**AVL:** Available
U/S: User/Supervisor	**PAT:** Page Attribute Table
PWT: Write-Through	**M:** Maximum Physical Address Bit
PCD: Cache Disable	**PK:** Protection Key
A: Accessed	**XD:** Execute Disable
D: Dirty	
PS: Page Size	

https://wiki.osdev.org/Paging

x86-64 page-table entry

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

Meltdown-US

[Lipp+ arXiv’18]

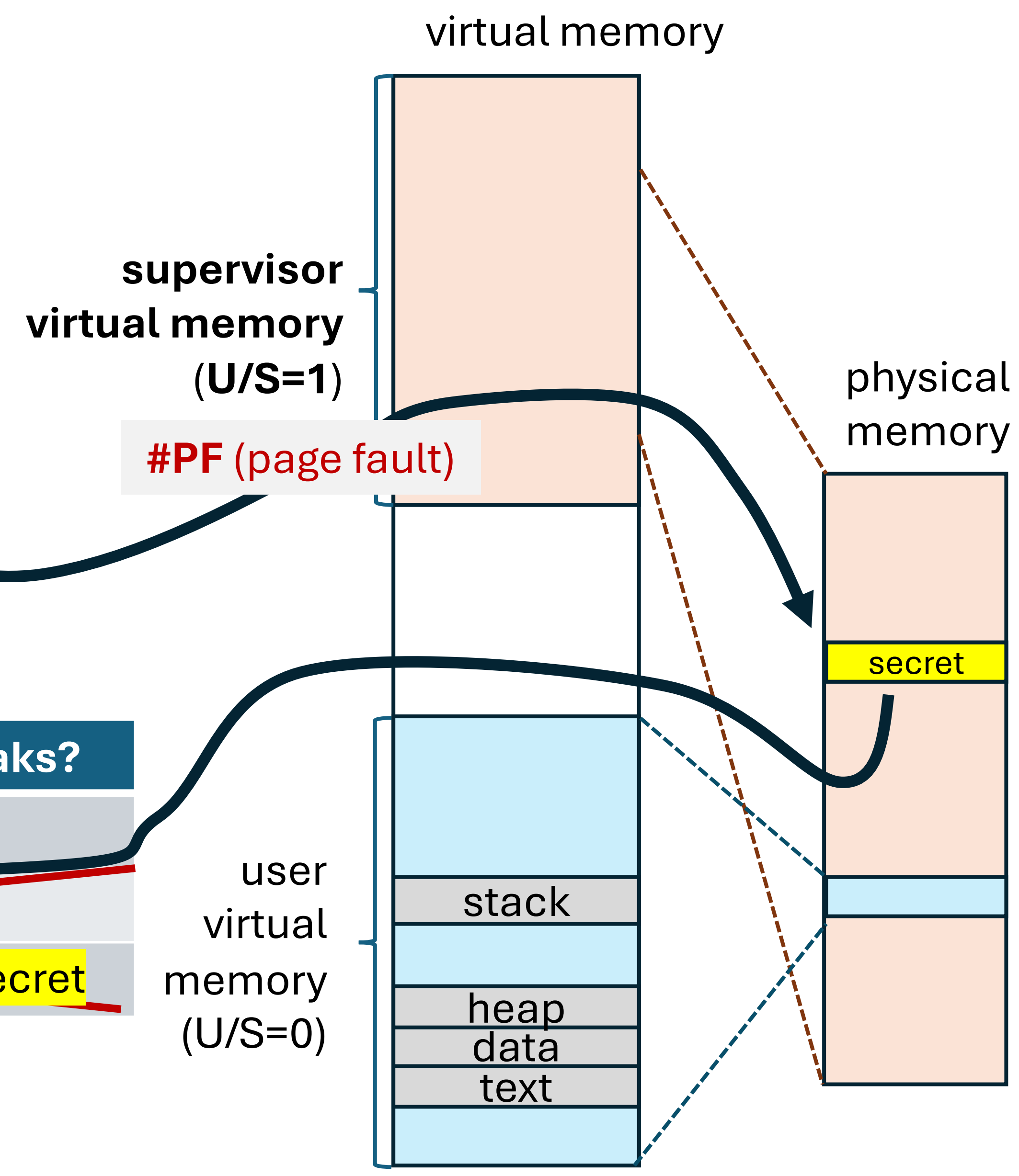
What happens *microarchitecturally* when a user loads from a supervisor page?

user program

```
...
ld r1,[kernel_vaddr]
leak r1
```

PC	Dst Value	Dst Address	Ready	Exception	Type	Leaks?
1000	...	...	1	(none)	...	
~~1004~~	~~secret~~	~~r1~~	~~1~~		~~ld~~	
~~1008~~	~~...~~	~~...~~	~~1~~	~~(none)~~	~~leak~~	~~secret~~

squashed (transient execution), but too late: attacker learned secret 😈

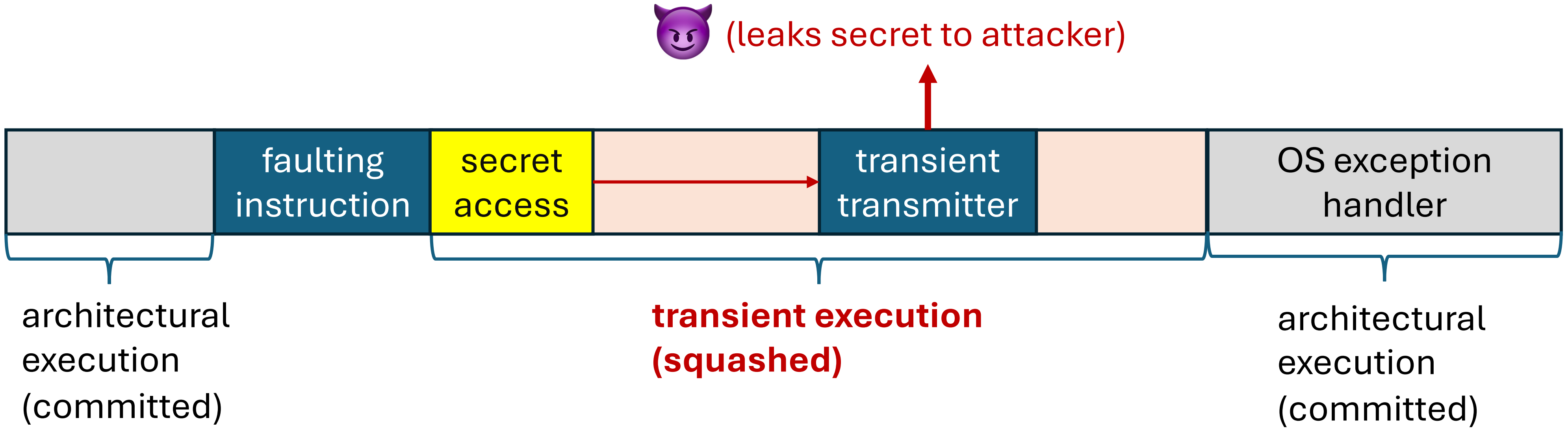


Meltdown-US allows an unprivileged attacker to leak **all physical memory** (via a side channel attack).

Anatomy of Meltdown attacks

Meltdown attacks involve:

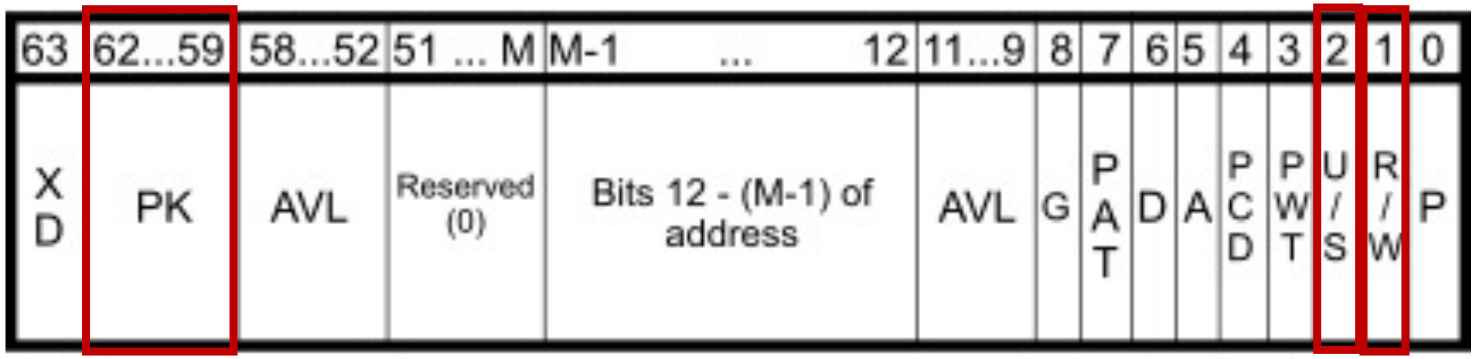
- a faulting instruction,
- a dependent transient transmitter that leaks private/privileged data to an attacker



(leaks secret to attacker)
faulting instruction
secret access
transient transmitter
OS exception handler
architectural execution (committed)
transient execution (squashed)
architectural execution (committed)

Other Meltdown variants exploiting page faults
[Canella+ USENIX'19]

- Categorized based on which page table entry protection bits are transiently ignored/bypassed
- **Meltdown-US**: transiently ignores U/S=0, allowing user to transiently read+leak supervisor pages
- **Meltdown-RW**: transiently ignores R/W=0 (read-only), allowing user to transiently write to read-only pages (writes never commit)
- **Meltdown-PK**: transiently ignores protection key (PK), allowing user to transiently read+leak pages for which read access has been disabled

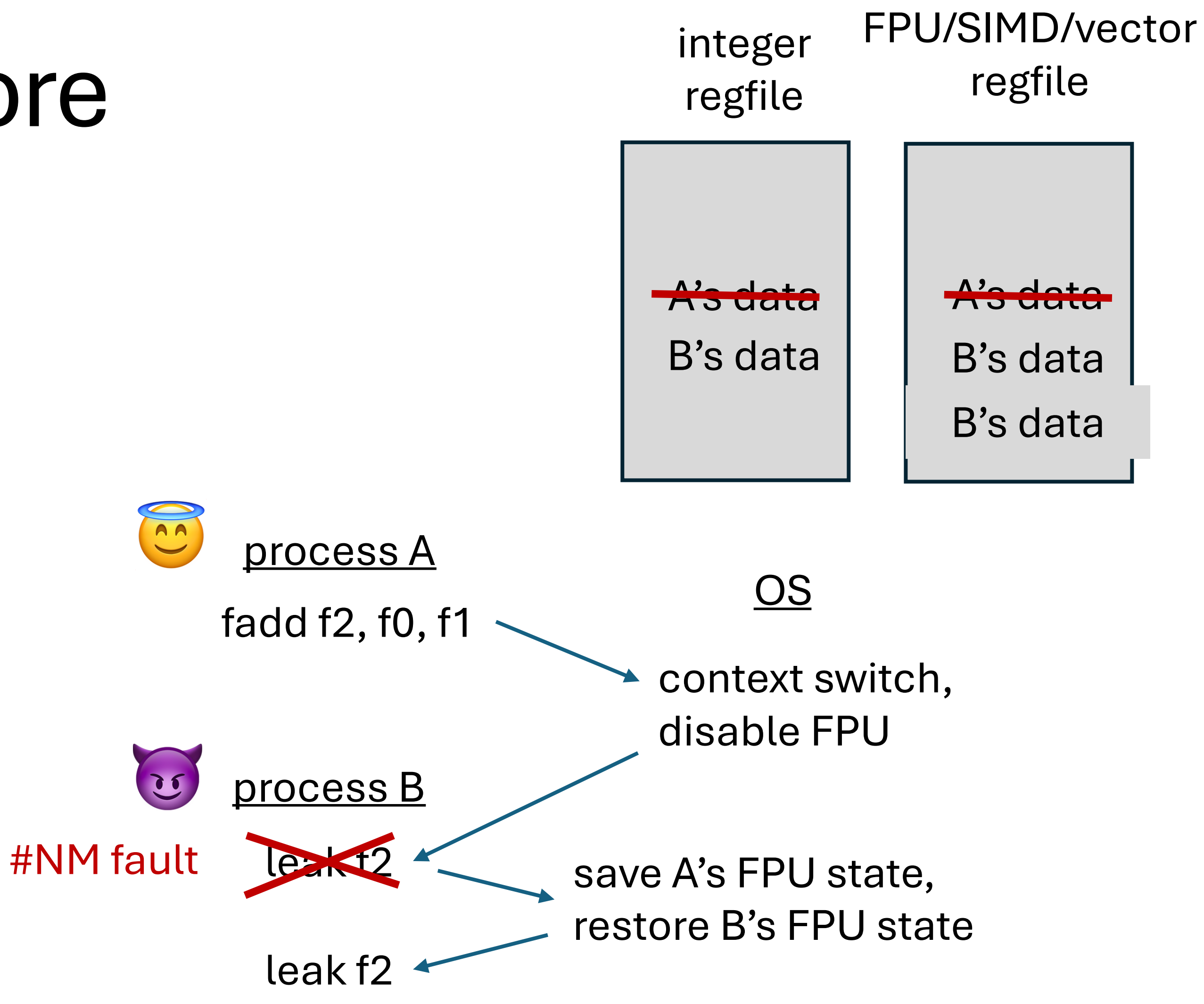


63	62...59	58...52	51 ... M	M-1 ... 12	11...9	8	7	6	5	4	3	2	1	0
XD	PK	AVL	Reserved (0)	Bits 12 - (M-1) of address	AVL	G	PAT	D	A	PCD	PWT	U/S	R/W	P

P: Present	**G:** Global
R/W: Read/Write	**AVL:** Available
U/S: User/Supervisor	**PAT:** Page Attribute Table
PWT: Write-Through	**M:** Maximum Physical Address Bit
PCD: Cache Disable	**PK:** Protection Key
A: Accessed	**XD:** Execute Disable
D: Dirty	
PS: Page Size	

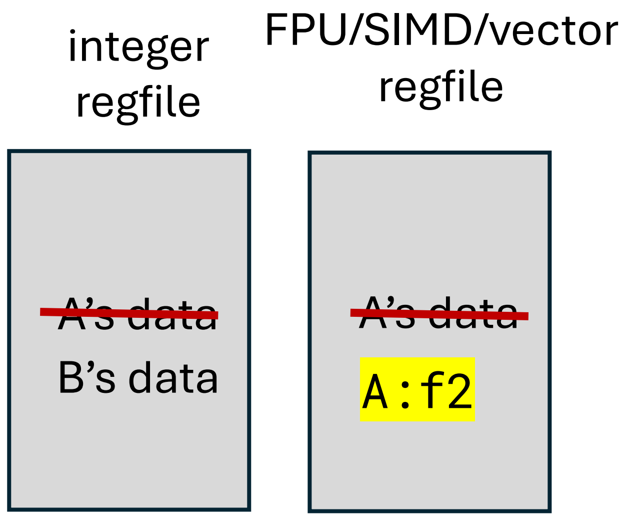
Background: lazy floating point restore

- To avoid saving/restoring entire FPU/SIMD/vector state on every context switch, the OS may mark the FPU/SIMD/vector unit as *unavailable* and only restore upon its first use
  - Executing a FPU/SIMD/vector instruction triggers a "device not available" (#NM) fault
  - Traps into OS, which saves the current state (belonging to other process) and restores current process' FPU state

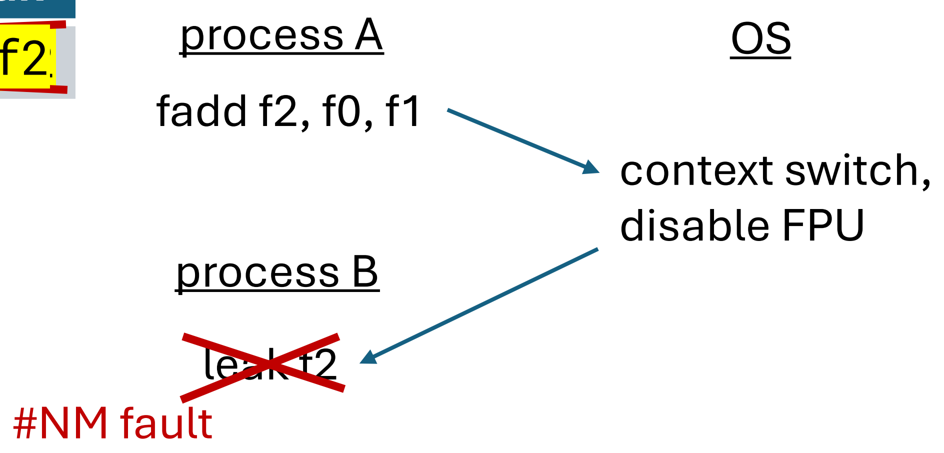


Meltdown-NM

What happens *microarchitecturally* when a process B tries to leak process A's `f2`?



PC	Dst	Src	Ready	Exception	Type	Leak
~~1000~~			~~1~~	~~#NM fault~~	~~leak~~	~~A:f2~~



squashed (transient execution), but too late: attacker learned secret

Meltdown Defenses

- Software defenses:
  - Meltdown-US: kernel page table isolation (KPTI) — avoid mapping kernel pages into user page tables
  - Meltdown-NM: eager FP state restore
  - Other Meltdown variants: impossible to mitigate in software
- Hardware defenses:
  - Update pipeline logic to not forward/writeback “bad” data produce by a faulting instruction
  - Makes it impossible to leak unauthorized data via side channels, while still using delayed fault handling

Meltdown takeaways

- Meltdown attacks “melt down” architectural isolation
- Enforcing architectural isolation lazily is dangerous
- Privilege checks should be enforced at execution, not at retirement.
- In theory, Meltdown is a solved problem.
- In practice, some Meltdown variants remain unfixed in recent CPUs.
    - E.g., Meltdown-GP: Rogue System Register Read

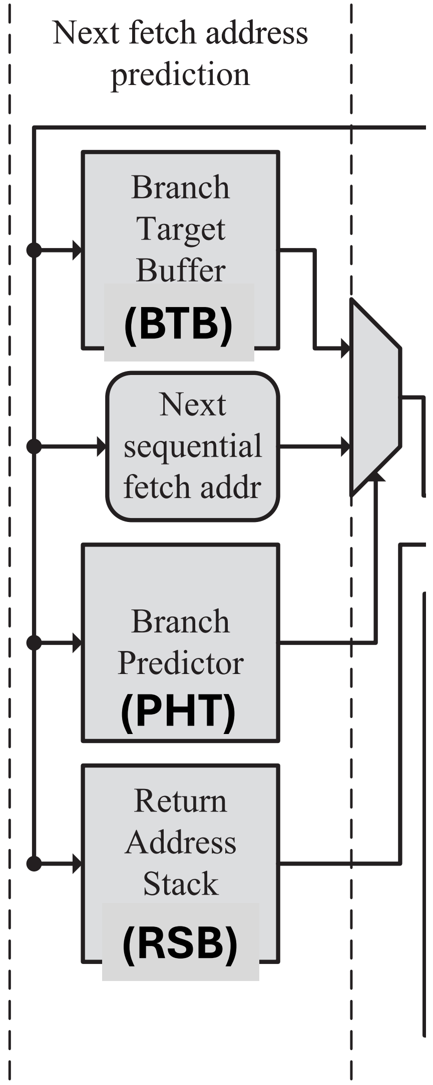
Outline

- Isolation
- Non-speculative side channel attacks and defenses
- Transient execution attacks and defenses overview
- Meltdown attacks and defenses
- Spectre attacks and defenses

Reminder: branch prediction

- **Branch predictor (PHT)**: predicts the direction of *conditional branches* (T/NT), updated based on past branch outcomes
  - Key property: if a branch is almost always taken (not-taken), it will be predicted taken (not-taken)
- **Branch target buffer (BTB)**: predicts the target of (taken) branches, updated based on past branch targets
- **Return address stack (RSB)**: predicts the target of *returns*, updated by calls

What happens on branch **mis**predictions?


Next fetch address prediction
Branch Target Buffer (BTB)
Next sequential fetch addr
Branch Predictor (PHT)
Return Address Stack (RSB)

Lecture 7, slide 78

Branch mispredictions

1. Predict next PC: nPC=1004
2. Speculatively fetch+execute instructions at nPC=1004,1008
3. Detect branch misprediction once bge executes
4. Squash instructions at 1004,1008
5. Redirect nPC to architecturally correct control-flow path

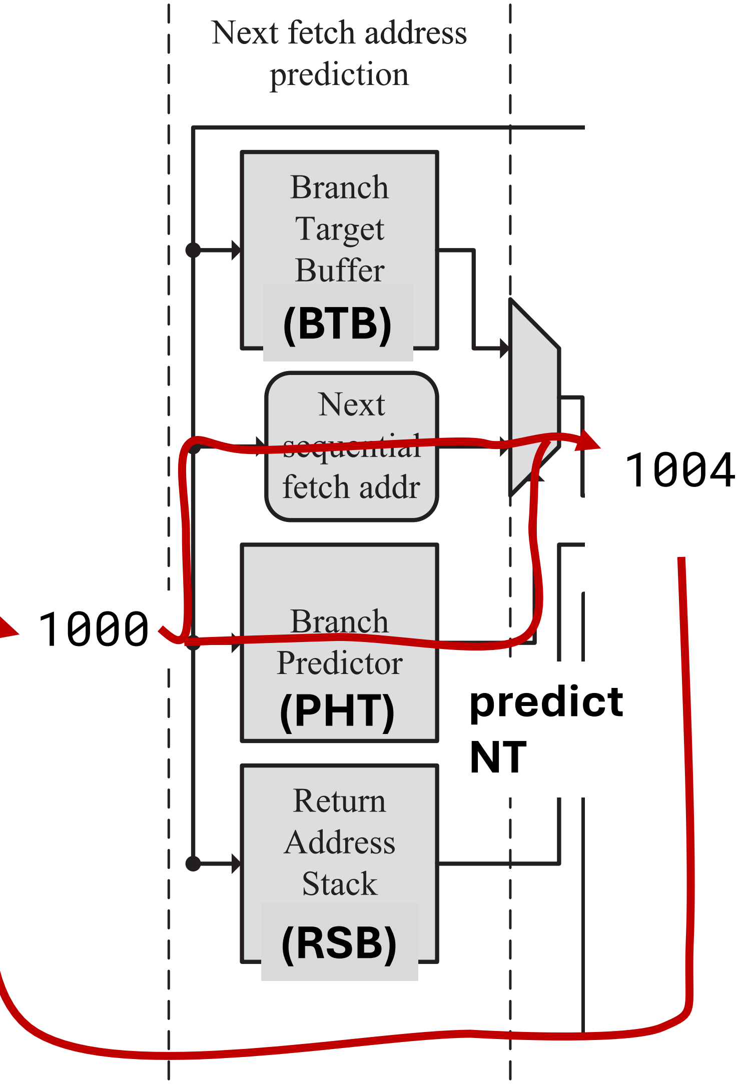
<u>source</u>

```
if (idx < 16) {
  x = A[idx];
  leak(x);
}
```

<u>assembly</u>

```
1000: bge r1,16,skip
1004: ld r3,[A+r1]
1008: leak r3
100C: ...
```

PC	Dst	Src	Ready	Mispredict?	Leak?
1000	-	16	1	yes	


Next fetch address prediction
Branch Target Buffer (BTB)
Next sequential fetch addr
Branch Predictor (PHT)
Return Address Stack (RSB)
1000
1004
predict NT

😈 A[16] transiently leaked via side channel…

Spectre [Kocher+ S&P'19]

victim memory:

```
int A[16] = {0, 0, 0, ...};
int secret = 42;
```

😇 *victim* (kernel, hypervisor, other process, etc.)

branch misprediction

```
if (idx < 16) {
  ~~x = A[idx];~~   // access out-of-bounds secret
  ~~leak(x);~~      // leak secret to attacker
}
```

attacker 😈 (process, VM)

train the branch predictor to predict not-taken (index in-bounds)

```
victim(idx = 0)
victim(idx = 0)
victim(idx = 0)
```

branch predictor predicts not-taken

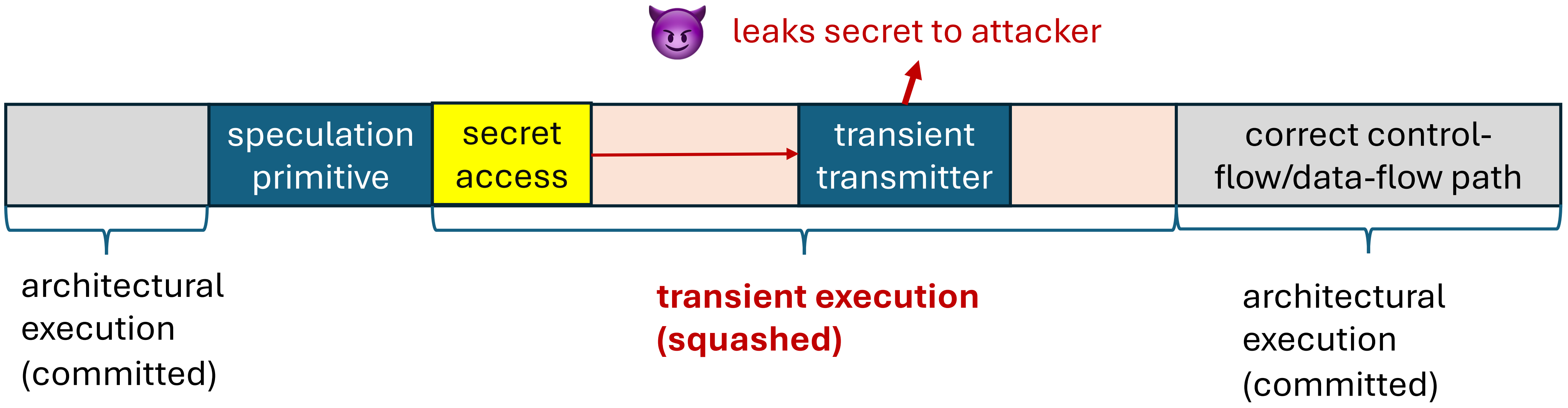
```
victim(idx =16)
```

Spectre attacks exploit mispredictions to coerce a victim into transiently leaking arbitrary victim architectural state to an attacker via transmitters.

Anatomy of Spectre attacks

Spectre attacks involve:

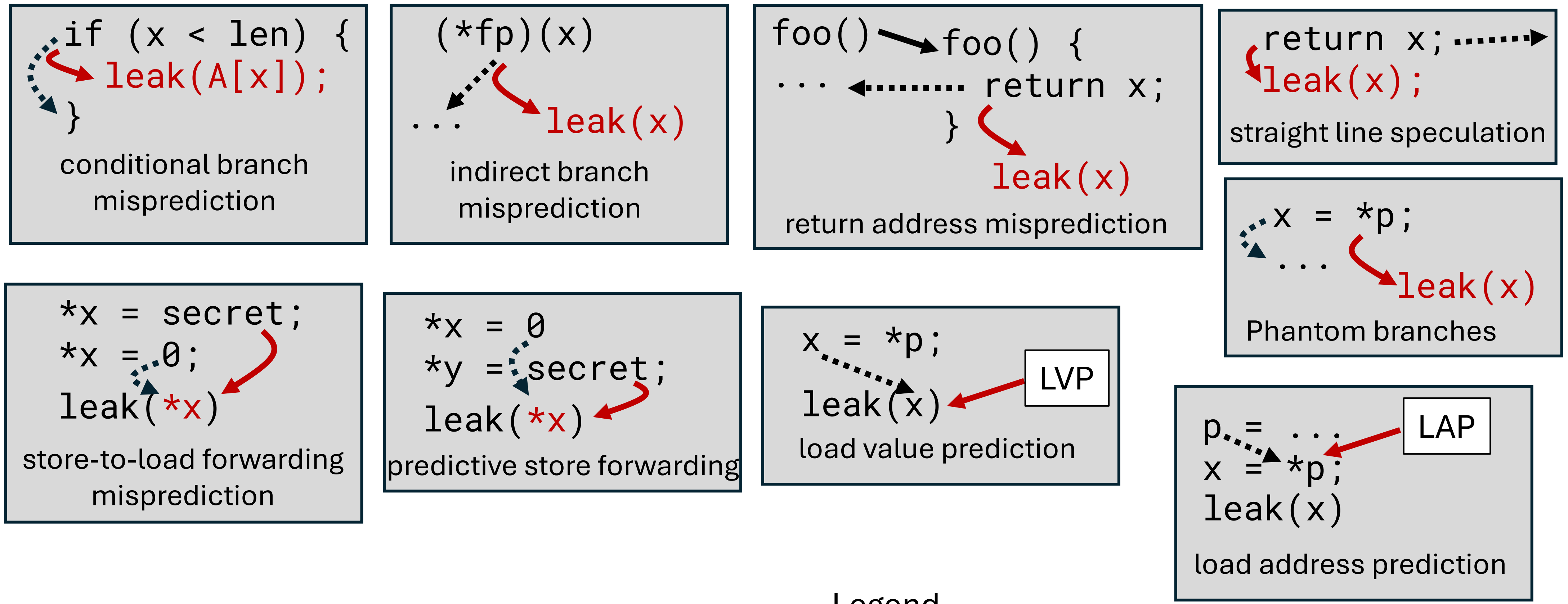
- a mispredicting instruction, called a **speculation primitive**
- a dependent transient transmitter that leaks secret data to an attacker



leaks secret to attacker
speculation primitive
secret access
transient transmitter
correct control-flow/data-flow path
architectural execution (committed)
transient execution (squashed)
architectural execution (committed)

Speculation Primitives on Modern Hardware

- Spectre attacks can exploit *any* speculation primitive



```
if (x < len) {
  leak(A[x]);
}
```
conditional branch misprediction

```
(*fp)(x)
...        leak(x)
```
indirect branch misprediction

```
foo()  foo() {
...      return x;
       }
         leak(x)
```
return address misprediction

```
return x;
leak(x);
```
straight line speculation

```
x = *p;
...    leak(x)
```
Phantom branches

```
*x = secret;
*x = 0;
leak(*x)
```
store-to-load forwarding misprediction

```
*x = 0
*y = secret;
leak(*x)
```
predictive store forwarding

```
x = *p;
leak(x)      LVP
```
load value prediction

```
p = ...    LAP
x = *p;
leak(x)
```
load address prediction

PHT = pattern history table
RSB = return stack buffer (i.e., RAS)

Legend

architectural control-/data-flow path

mispredicted control-/data-flow path

Software-only Spectre Defenses

- Mitigate subset of Spectre leakage using existing hardware primitives
- Little to no coordination between software and hardware
- Non-comprehensive: cannot protect against all speculation primitives
- Three flavors of approaches:
  - **Speculation fences**: e.g., x86 LFENCE
  - **Code transformations**: e.g., Speculative Load Hardening
  - **Transient control-flow and data-flow restrictions**: e.g., Intel CET-based approaches
- The most comprehensive defenses combine multiple approaches

Speculation fences

- Many ISAs provide a speculation fence instruction (e.g., x86's LFENCE)
- Semantics: no younger instruction can issue until speculation fence retires
- Can defend against:
  - conditional branch misprediction
  - straight-line speculation
  - data-flow mispredictions
- *Cannot* defend against:
  - indirect branch prediction
  - Phantom branches
  - return address prediction
- Limitations:
  - Extreme overhead (>600%): prevents out-of-order execution

Example: securing classic Spectre attack using speculation fence

```
if (idx < len) {
   SPEC_FENCE();
   x = A[idx];
   leak(x);
}
```

blocks transient execution

Speculative Load Hardening
(example code transformation)

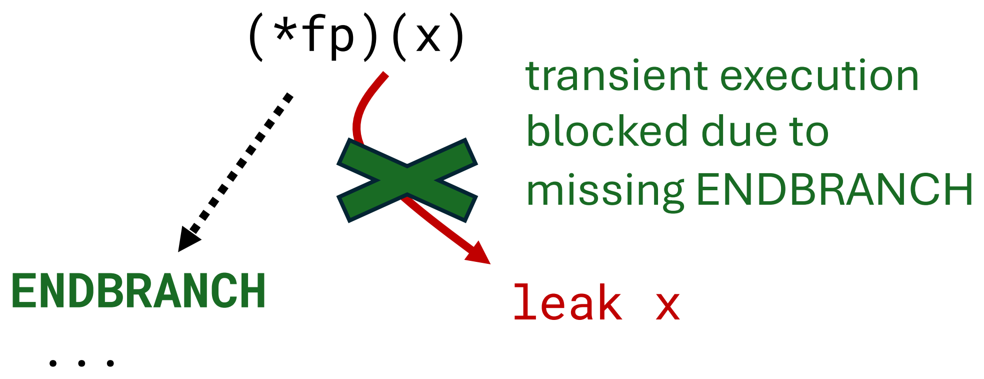
- Idea:
  - detect conditional branch mispredictions
  - mask off data that may transiently leak following misprediction
- Limitations:
  - only works with conditional branch prediction
  - high overhead (>100%)

```
if (idx < len) {
  mispred = (idx >= len) ? 0 : -1;
  x = A[idx];
  x &= mispred;
  leak(x);
}
```

x=0 on cond. branch misprediction

Transient Control-Flow Restrictions using Intel CET

- Intel Control-Flow Enforcement Technology (CET)
- Offers architectural and transient control-flow integrity (CFI)
- Modifies semantics for indirect branches:
  - Indirect branches can only jump targets marked with ENDBRANCH instructions
- Limitations:
  - Restricts but does not prevent all Spectre leaks due to indirect branch misprediction
  - Does not protect against other Spectre variants
- Advantages: useful building block for stronger software Spectre defenses



Serberus

[Mosier+ S&P'24]

- Combines all three techniques to obtain the most comprehensive software-only defense to date
- *Transient control-flow and data-flow restrictions*: **enables Intel CET, PSFD**
- *Code transformation*: **function-private stacks, register cleaning**
- *Speculation fence insertion* as fallback when needed

software defense	**PHT** conditional branch pred	**BTB** indirect branch pred	**RSB** return prediction	**STL** store-to-load fwding pred	**PSF** predictive store fwding
Intel LFENCE	⌫				
UltimateSLH [Zhang+ USENIX'23]	●				
Blade [Vassena+ POPL'21]	●				
selSLH [Shivakumar+ S&P'23]	●				
Misspec. types [Shivakumar+ S&P'23]	●				
retpoline		⌫			
IPRED_DIS		⌫			
SSBD				⌫	⌫
PSFD					⌫
Securest SOTA	●	⌫		⌫	⌫
Serberus [Mosier+ S&P'24]	●	●	●	●	⌫

Legend

⌫ disable speculation

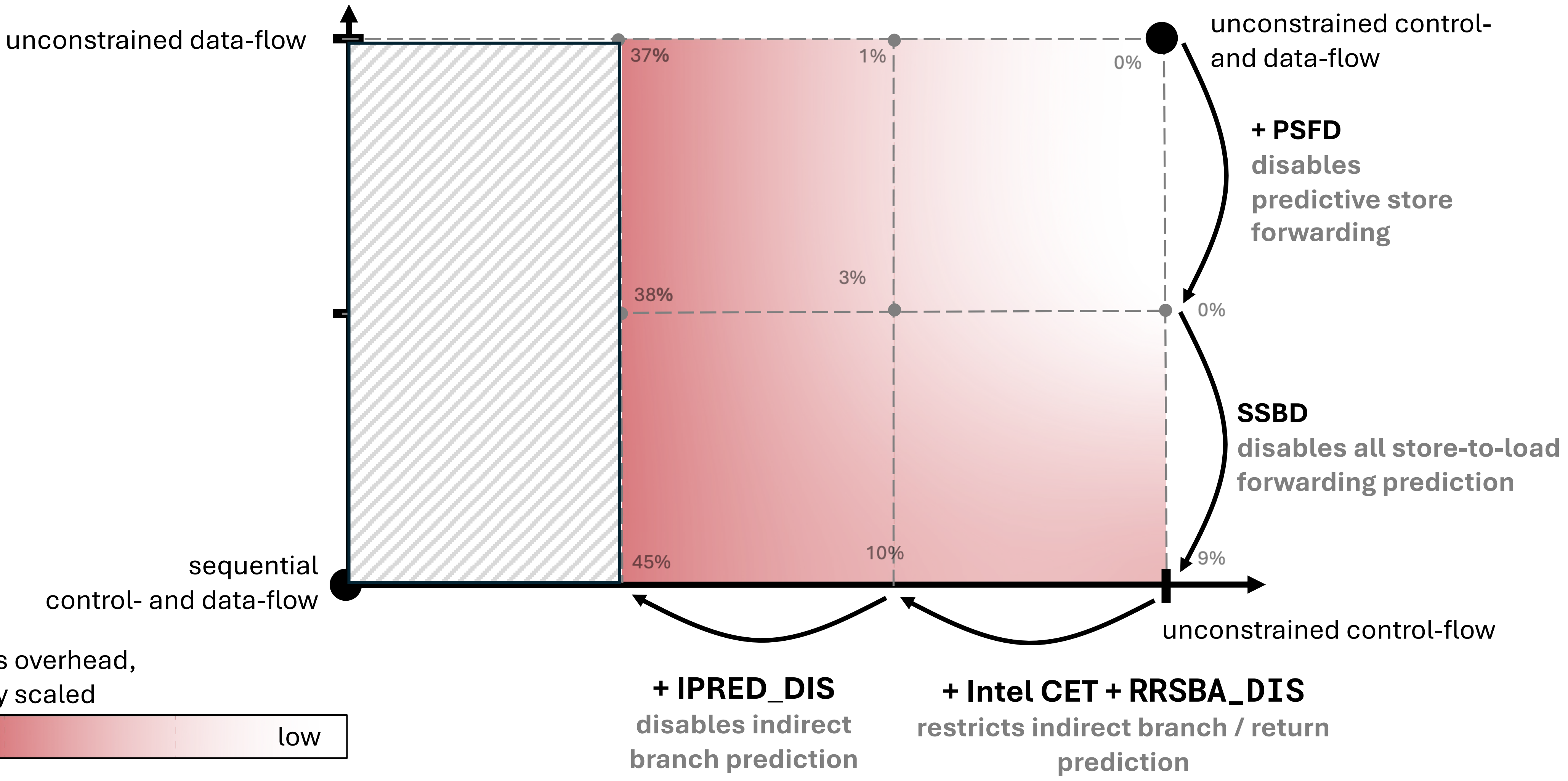
● secure speculation

Hardware-Based Spectre Defenses

- Many hardware Spectre defenses have been proposed, but none have been adopted
- Generation 1 (2018):
  - First acaedmic defenses blocked Spectre leakage via **cache side channels only** → non-comprehensive
  - Industry-deployed speculation controls only disable / heuristically restrict a subset of speculation primitives → non-comprehensive
- Generation 2 (2019–2021):
  - Follow-up academic defenses block Spectre leakage via all side channels and transmitters, but only for a subset of secret architectural state → semi-comprehensive
  - Mostly hardware-only
- Generation 3 (2021–):
  - Latest Spectre defenses can protect all secret data → comprehensive
  - Mostly hardware-software codesigned
  - Trade-off between extensive programmer input vs. fully programmer transparent

Hardware Speculation Controls

Hardware speculation controls deployed by Intel and AMD restrict/disable a subset of speculation primitives (non-comprehensive), at a high performance cost.

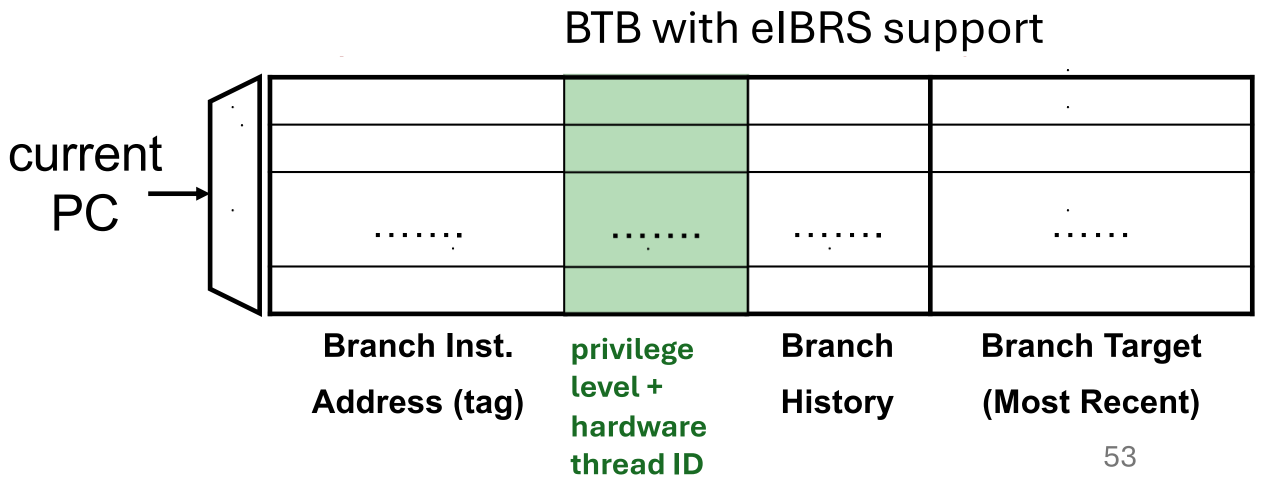

unconstrained data-flow
unconstrained control- and data-flow
37%
1%
0%
+ PSFD
disables predictive store forwarding
38%
3%
0%
SSBD
disables all store-to-load forwarding prediction
45%
10%
9%
sequential control- and data-flow
unconstrained control-flow
+ IPRED_DIS
disables indirect branch prediction
+ Intel CET + RRSBA_DIS
restricts indirect branch / return prediction

Legend:
color indicates overhead, logarithmically scaled

Note: overheads measured on SPEC2017 intspeed.

Other Incomplete Spectre Mitigations by Vendors

- Focus primarily on mitigating cross-domain Spectre-BTB attacks
- **Intel Enhanced Indirect Branch Restricted Speculation** (eIBRS)
    - Partial Spectre-BTB mitigation
    - Prevents user from poisoning kernel's or sibling SMT thread's BTB entries
- **Indirect Branch Prediction Barrier** (IBPB)
    - Partial Spectre-BTB mitigation
    - Clears BTB
    - High penalty


BTB with eIBRS support
current PC
....
....
....
......
Branch Inst. Address (tag)
privilege level + hardware thread ID
Branch History
Branch Target (Most Recent)

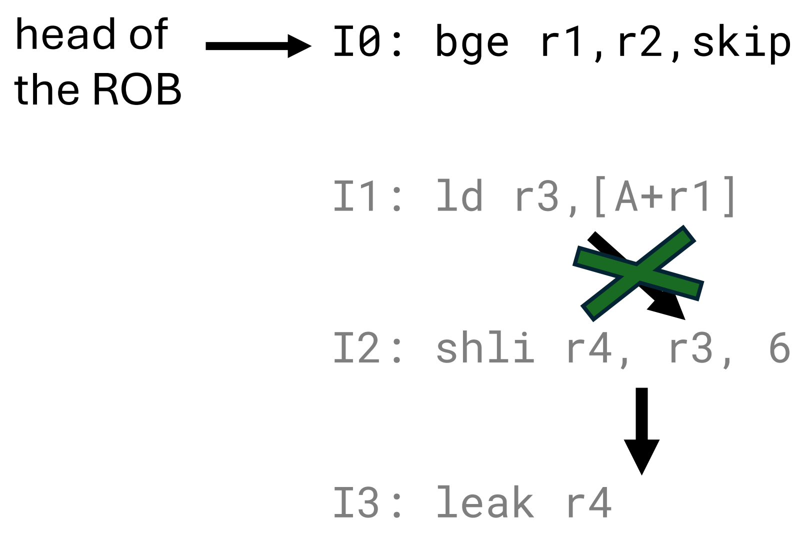
NDA [Weisse+ MICRO’19]

- Semi-comprehensive hardware-only defense
  - Protects **secrets in memory** from leaking transiently via all transmitters and via all speculation primitives
- Approach: prevent all loads from writing back until they are ready to commit
- Limitations: high overhead

source

```
if (idx < len) {
  x = A[idx];
  y = x * 64;
  leak(y);
}
```

assembly



```
I0: bge r1,r2,skip
I1: ld r3,[A+r1]
I2: shli r4, r3, 6
I3: leak r4
```

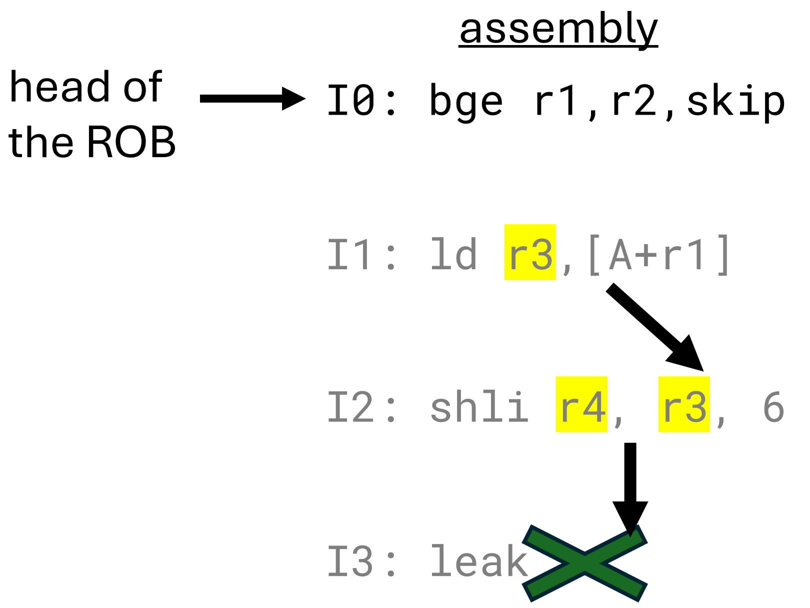
I2 and I3 cannot speculatively execute until I1 retires.

Speculative Taint Tracking [Yu+ MICRO'19]

- Semi-comprehensive hardware-only defense:
  - Protects secrets in memory from leaking transiently via all transmitters and via all speculation primitives
- **Youngest root of taint (YRoT)**: youngest speculative load that each instruction depends on
- Do not issue transmitters until their YRoT commits
- Key advantage: allows safe instructions that depend on loads to speculatively execute
- Limitation: does not protect secrets in registers

source

```
if (idx < len) {
  x = A[idx];
  x = x * 64;
  leak(x);
}
```

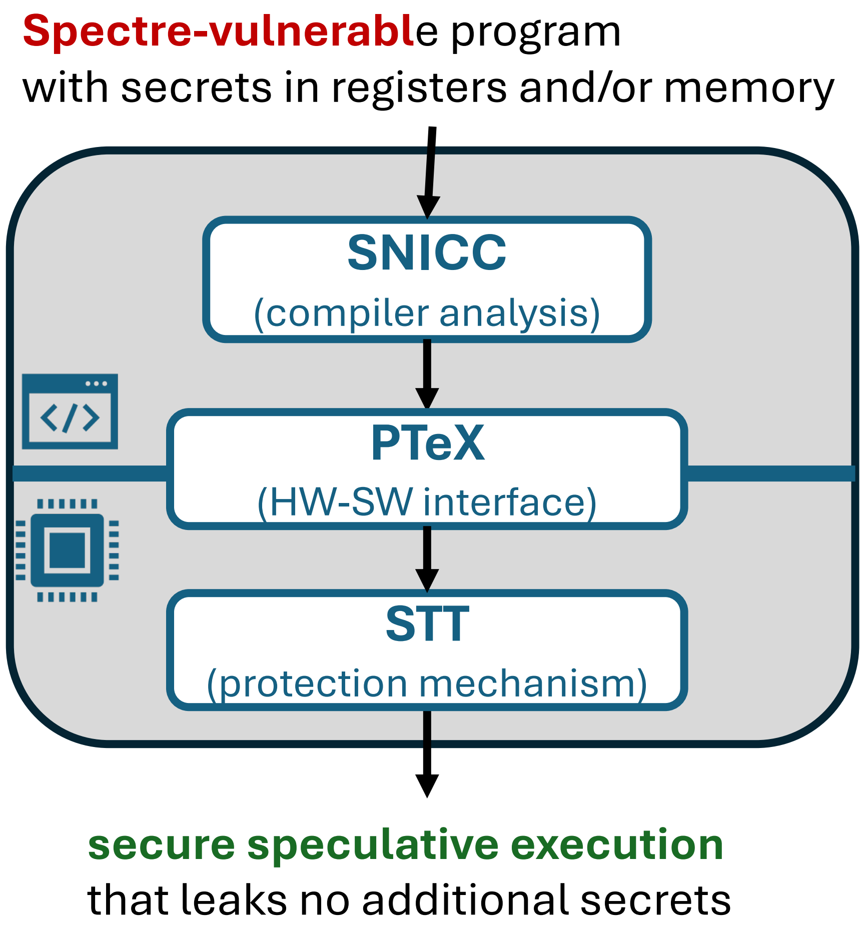


```
I0: bge r1,r2,skip

I1: ld r3,[A+r1]

I2: shli r4, r3, 6

I3: leak
```

I3 cannot speculatively execute until I1 retires.

Mieros (ongoing research)

- Hardware-software co-design that extends STT to protect secrets in both **registers and memory**
- **Key optimization**: allows data to leak speculatively if it leaks non-speculatively
- Compiler analysis identifies non-speculatively leaked data
- New UNPROT instruction communicates to HW which data non-speculatively leaks
- Hardware uses Speculative Taint Tracking to block the issue of transmitters that depend on speculative UNPROT instructions

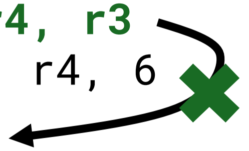

Spectre-vulnerable program
with secrets in registers and/or memory
SNICC
(compiler analysis)
PTeX
(HW-SW interface)
STT
(protection mechanism)
secure speculative execution
that leaks no additional secrets

source

```
if (idx < len) {
  x = A[idx];
  y = unprot(x);
  y = y * 64;
  leak(y);
}
```

assembly

```
1000: bge r1,r2,skip
1004: ld r3,[A+r1]
1008: unprot r4, r3
100C: shli r5, r4, 6
1010: leak r5
```



cannot issue transmitter until unprot commits

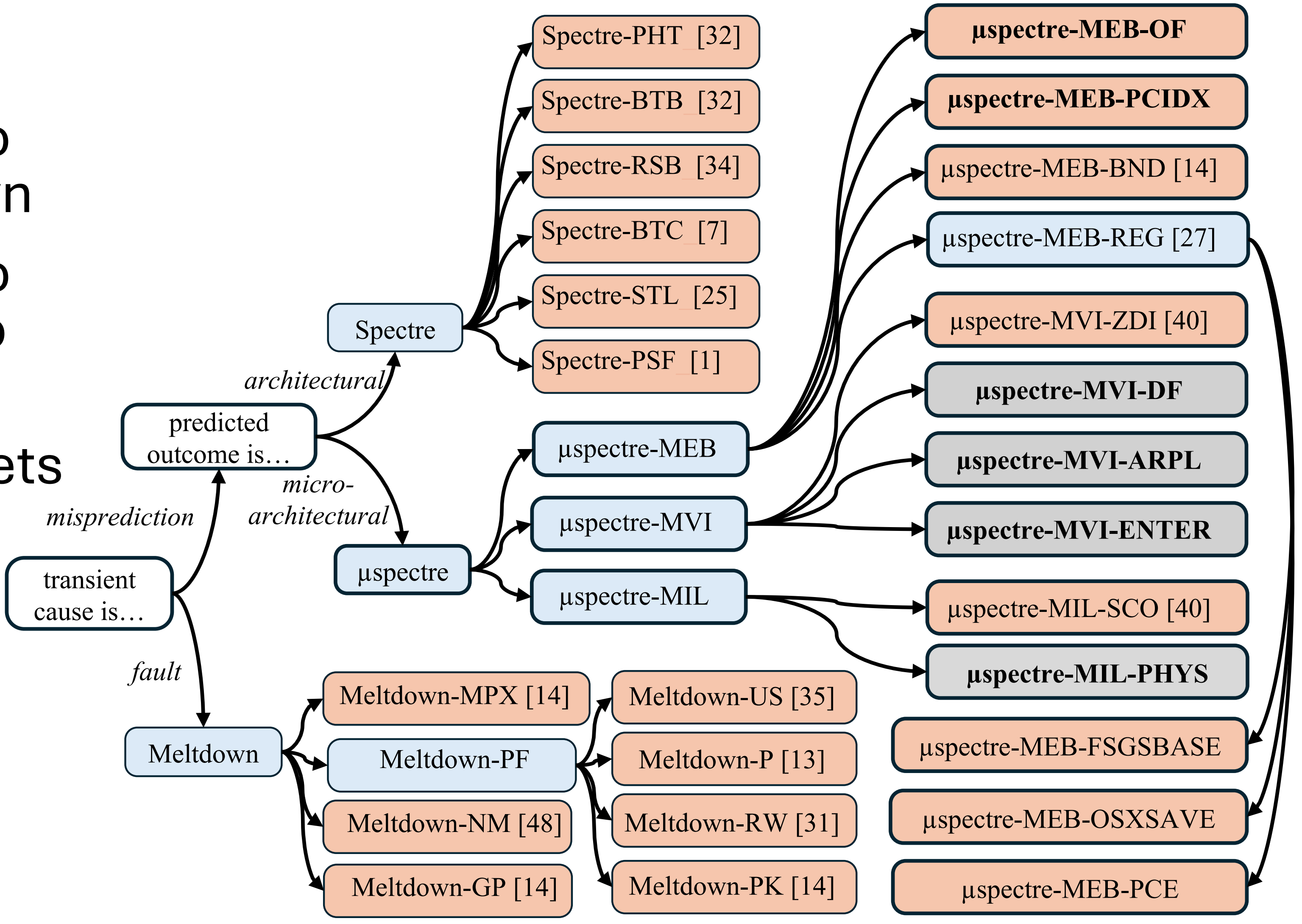
Spectre Takeaways

- Spectre attacks exploit **mispredictions** to transiently leak secrets via side channels
- Why is it so hard to mitigate?
    - Prediction is critical to performance
    - But mispredictions can leak secrets
    - How to know what is secret?

Putting it all together

- Transient execution due to faults give rise to Meltdown
- Transient execution due to mispredictions give rise to Spectre
- Both transiently leak secrets via side channels
- Meltdown generally has simple hardware fixes
- Spectre requires complex defenses across the hardware-software stack

Transient execution attack classification tree



Putting it all together: Spectre defense tradeoffs

Axis	Categories	Trade-offs		
		Comprehensiveness	HW/SW complexity	Overhead
Design type	• software • **hardware** • **hardware-software**	• non-comprehensive • depends • depends	• none/high • high/none • moderate/moderate	• high • moderate • low
Side channel coverage	• **all side channels** • specific side channels	• comprehensive • non-comprehensive	• low/low • high/high	• high • low
Speculation primitive coverage	• **all speculation primitives** • specific speculation primitives	• comprehensive • non-comprehensive	• low/impossible • moderate / moderate	• high • low
Scope of protected arch. state	• **all architecturally confidential state** • specific architectural state	• comprehensive • non-comprehensive	• moderate–high / high • low–moderate / low–moderate	• high • low–moderate

Other microarchitectural security topics

- DRAM: RowHammer
- Non microarchitectural side channels like power, etc.
- Non-speculative leaky optimizations introducing new side channels: cache compression, register file compression, silent stores, …

Questions?