School of Computer Science & Engineering

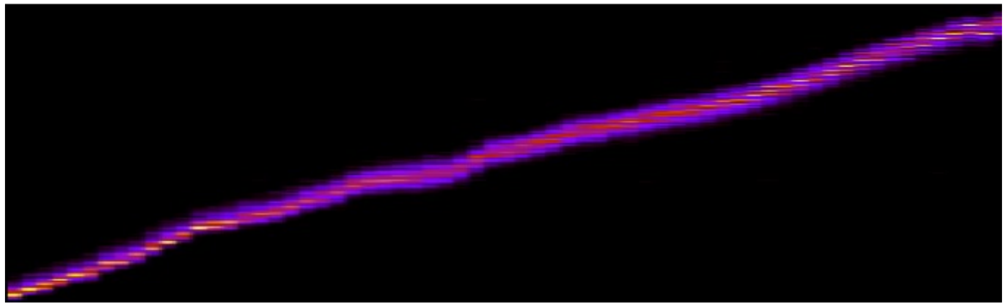
COMP9242 Advanced Operating Systems

2021 T2 Week 09 Part 1

Security: Secure Operating Systems & Information Leakage

@GernotHeiser

Spectre/Meltdown material courtesy of
Yuval Yarom (UAde) & Daniel Genkin (UMI)



Secure Operating Systems

Principles

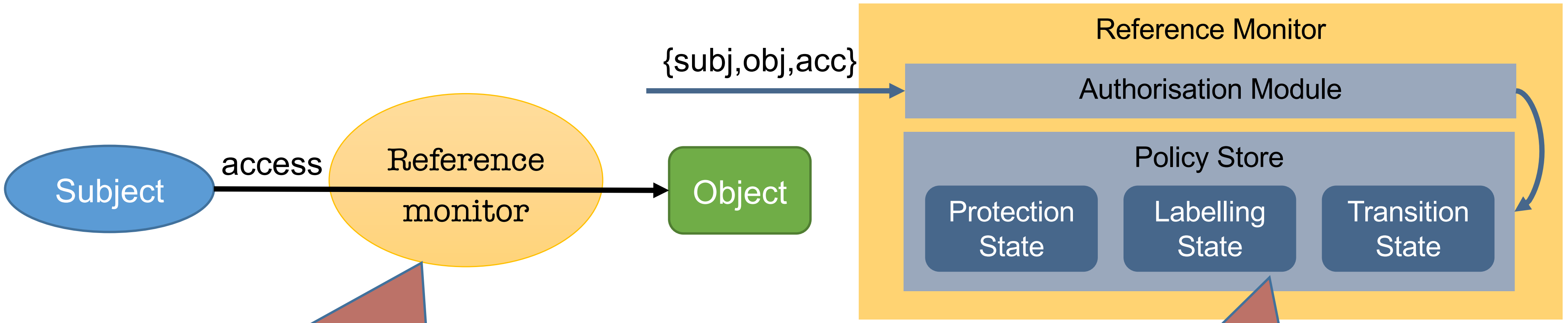
What is a Secure OS?

An OS is *secure* if it enforces the system's *security policy*.

Secure Operating Systems

Secure OS: [Jaeger: OS Security]

Access enforcement satisfies the *reference monitor* concept



Enforces *mandatory protection:*
- non-bypassable
- tamperproof
- verifiable

Permission: relation over labels

Policy examples

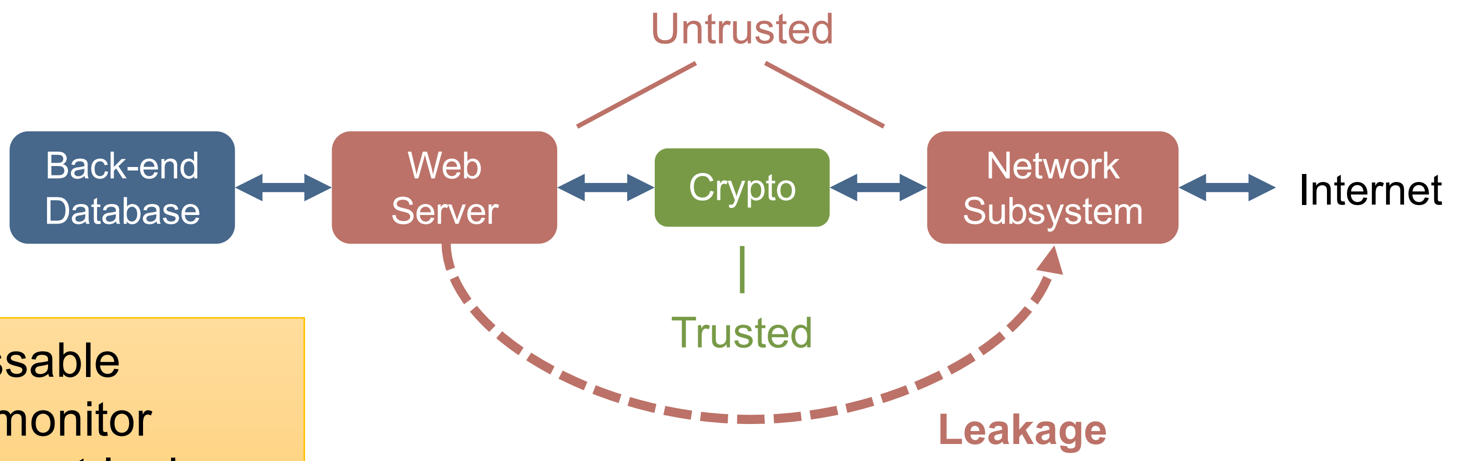
Hierarchical (e.g. Bell-LaPadula):
Directed information flow
- no read up
- no write down

Isolation (e.g. VMs in public cloud):
- no read access to other VM
- no write access to other VM

Integrity:
- untrusted code must not write outside its space

Confinement:
- untrusted code must not leak secrets
- write access only to specific buffers

Confinement Example



Non-bypassable reference monitor should prevent leakage

Challenges:

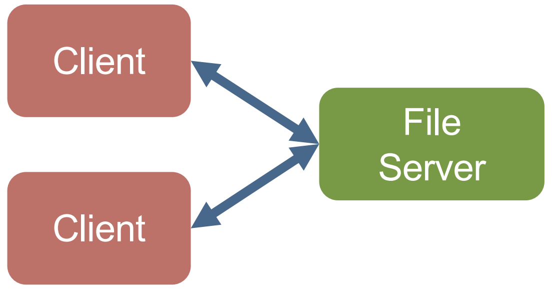
- Prevent changes to security state – mandatory enforcement
- Covert channels

How Secure Are Traditional OSes?

E.g. Unix/Linux:

- Complete Mediation?
  - `ioctl()/fcntl()` modify state without write permission
  - no authorisation for some resources (eg. network)
- Tamperproof?
  - protection system discretionary – users can change protection state
  - `/proc` etc: user processes modify kernel state
  - many privilege-escalation exploits
- Verifiable?
  - informal specification of functionality and security
  - huge TCB (kernel and all `root` processes)

Resource Management



Server holds meta-data, caches, etc
- allocated on behalf of clients
- where from?

Common memory pool is insecure!
- Denial-of-service attacks
- Covert channels

Solutions:
1. Static partitioning
2. Resource-donation scheme

Timing Channels

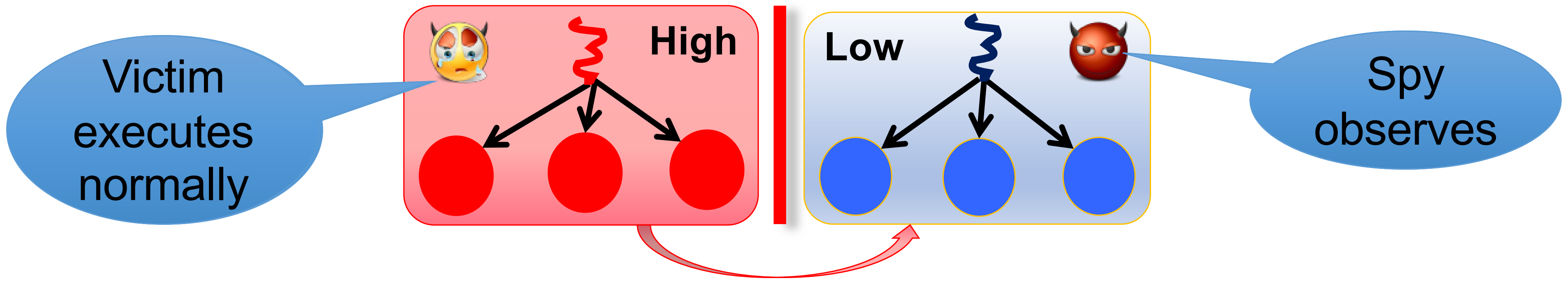
Principles

Refresh: Timing Channels

Information leakage through timing of events

- Typically by observing response latencies or own execution speed

Covert channel: Information flow that bypasses the security policy



Side channel: Covert channel exploitable without insider help

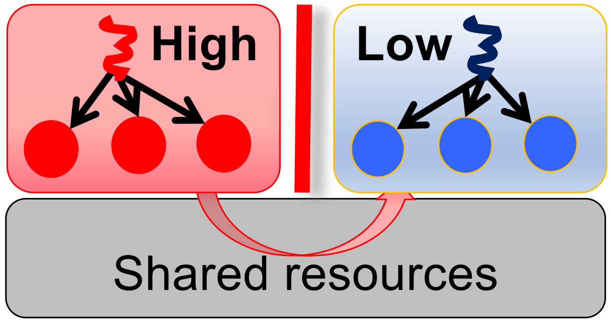
Causes of Timing Channels

Algorithmic

```
if (secret) {
   short_operation(...);
} else {
   long_operation(...);
}
```

Resource Contention

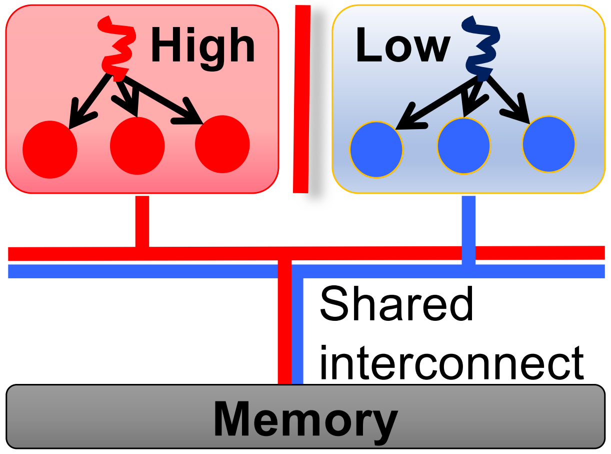
- Software resources
  - OS abstractions
  - buffer cache…
- Hardware resources
  - caches etc
  - not visible at ISA (HW-SW contract)



Affect execution speed

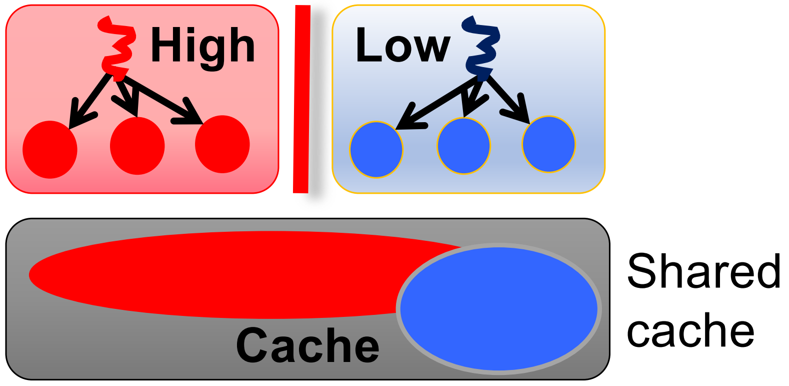
Micro-architectural timing channels

Shared Hardware: Stateless Interconnect



H/W is *bandwidth-limited*
- Interference during concurrent access
- Generally reveals no data or addresses
- Must encode info into access patterns
- *Only usable as covert channel, not side channel*

Shared Hardware: Stateful Resources



Can be any state-holding microarchitectural feature:

- CPU caches
- branch predictor
- pre-fetcher state machines

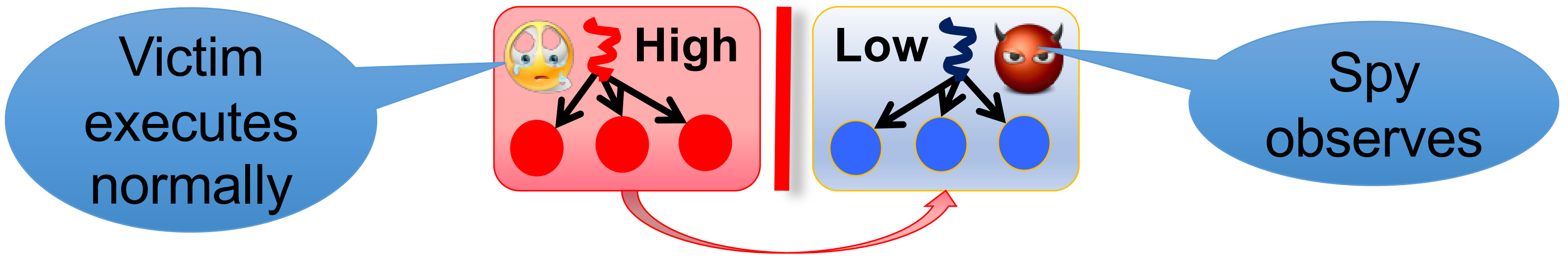
H/W is *capacity-limited*

- Interference during
  - concurrent access
  - time-shared access
- Collisions reveal addresses
- *Usable as side channel*

Timing Channels

Example: LLC Side Channel

Methodology: Prime and Probe



1. Fill cache with own data

2. Touch cache lines

Input Signal

3. Traverse cache, measure execution time

Output Signal

Challenge: Slow LLC Access Times

- L1 (32 KiB) probe:
  - 64 sets * 8 ways * 4 cycles = 2,048 cycles
- Small last-level cache (6 MiB):
  - 8,192 sets * 12 ways * ~30 cycles = ~3,000,000 cycles

Probing entire LLC is too slow, but single set is fast

Example: Look for `square` code in square-and-multiply exponentiation of GnuPG

- Approach:
  - *Probe one or a few cache sets at a time*
  - *Find "interesting" sets ("eviction set") by looking for patterns*

Searching for square Code

Modular reduction: *$r = b^e \bmod m$*

```
long_int r (long_int b, m, e) {
  res = 1;
  for (i = n-1; i >= 0; i--) {
    if (e[i]) {
      r = mod (r * b, m);
    }
  }
  return res;
}
```

i^{th} bit of e

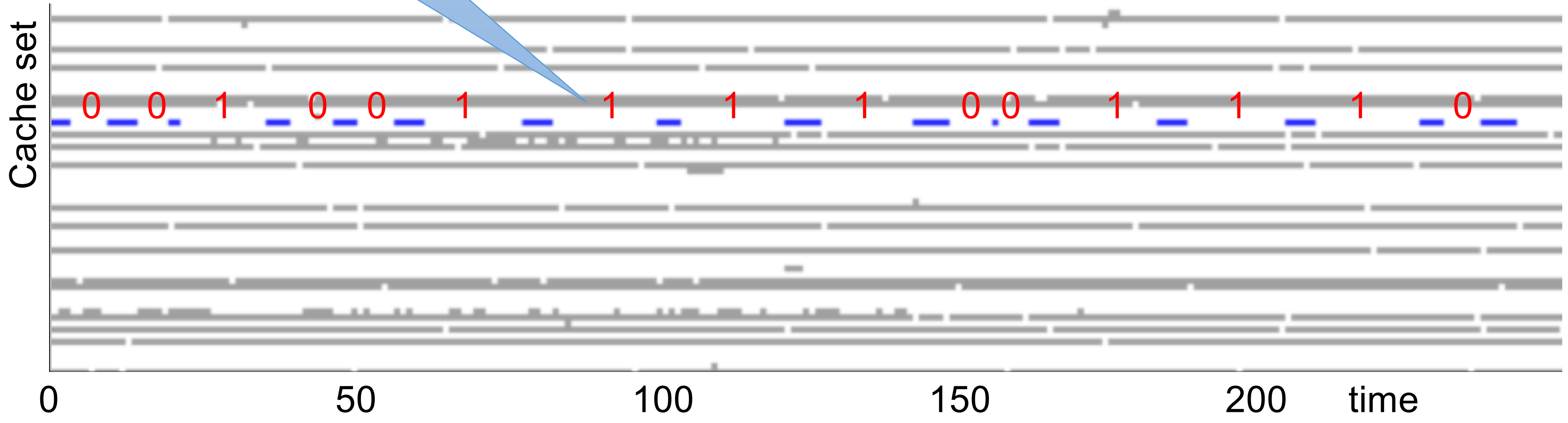
Long computation if bit is set

Expected pattern: Bursts of activity separated by longer or shorter intervals indicating modular reduction operation

Searching for `square` Code

Can read out bits of exponent! [Liu S&P'15]

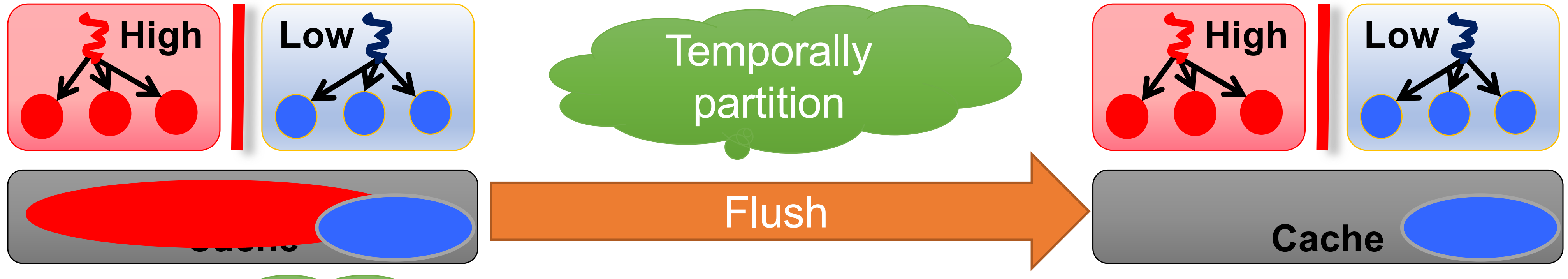
Expected pattern: Bursts of activity separated by longer or shorter intervals indicating modular reduction operation

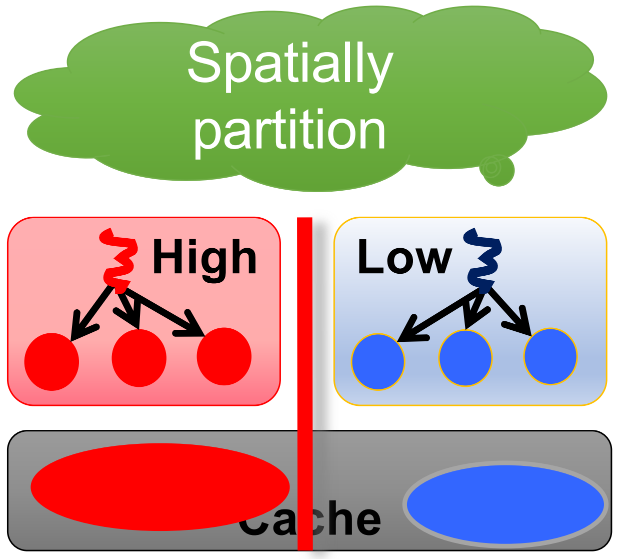


Timing Channels

Evaluating Hardware

Timing-Channel Prevention: Partition HW





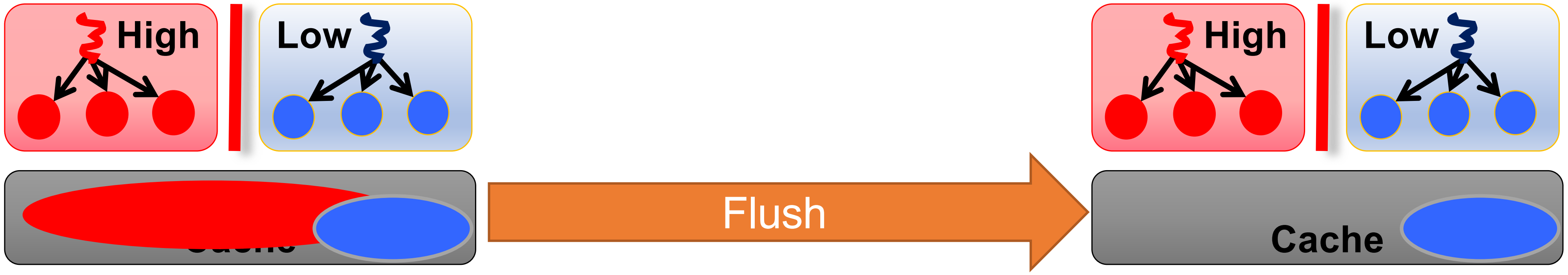
Need both!

Cannot spatially partition on-core caches (L1, TLB, branch predictor, pre-fetchers)
- virtually-indexed
- OS cannot control

Flushing useless with concurrent access:
- HW threads (SMT)
- Multicore

No partitioning support for interconnects!

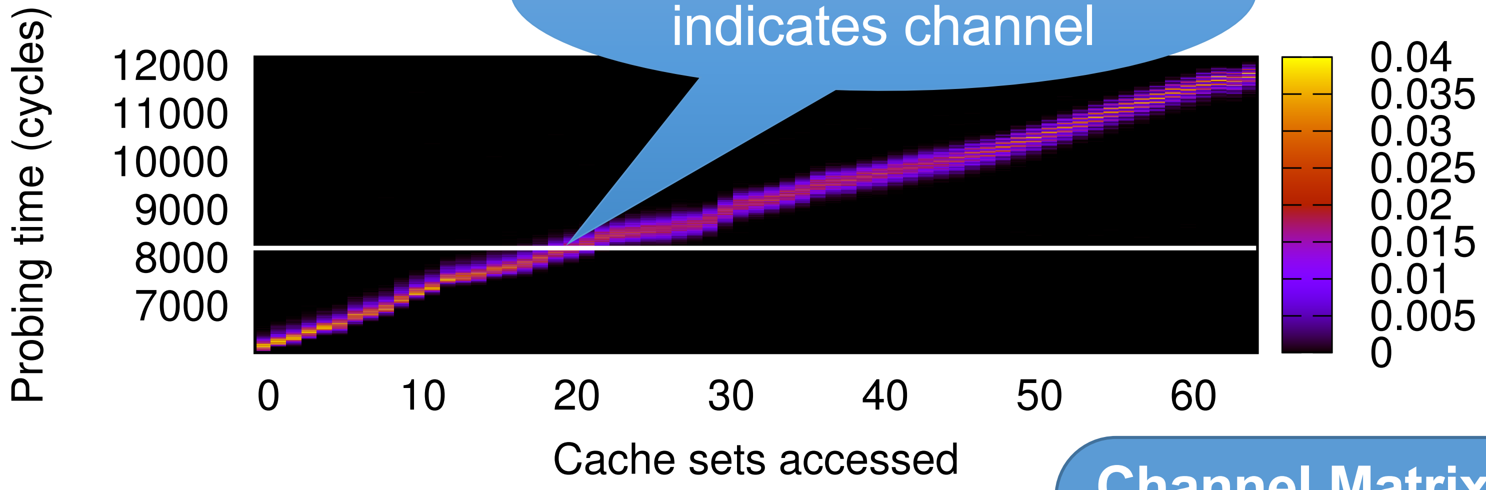
Evaluating Intra-Core Channels



Methodology:

- Flush all caches on context switch
  - using all flush ops provided by HW
- Run prime&probe *covert channel* attack

Methodology: Channel Matrix



Horizontal variation indicates channel

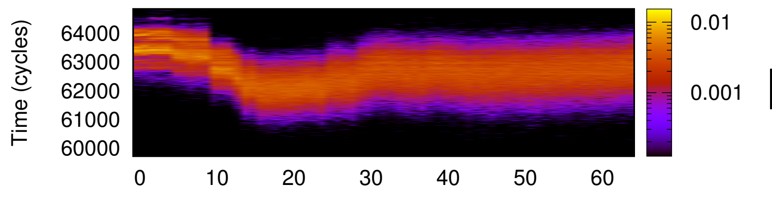
Raw I-cache channel
Intel Sandy Bridge

Channel Matrix:

- Conditional probability of observing time *t*, given input *n*.
- Represented as heat map: bright = high probability.

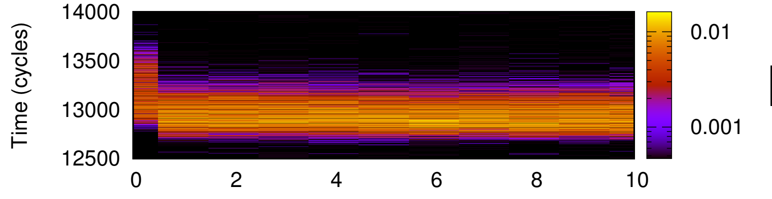
I-Cache Channel With Full State Flush

CHANNEL!



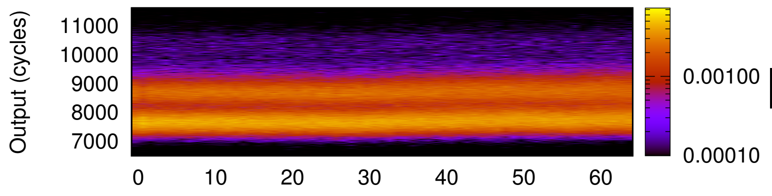
Intel Sandy Bridge

CHANNEL!



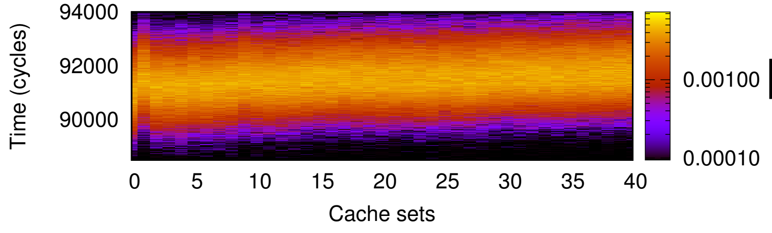
Intel Haswell

No evidence of channel



Intel Skylake

SMALL CHANNEL!

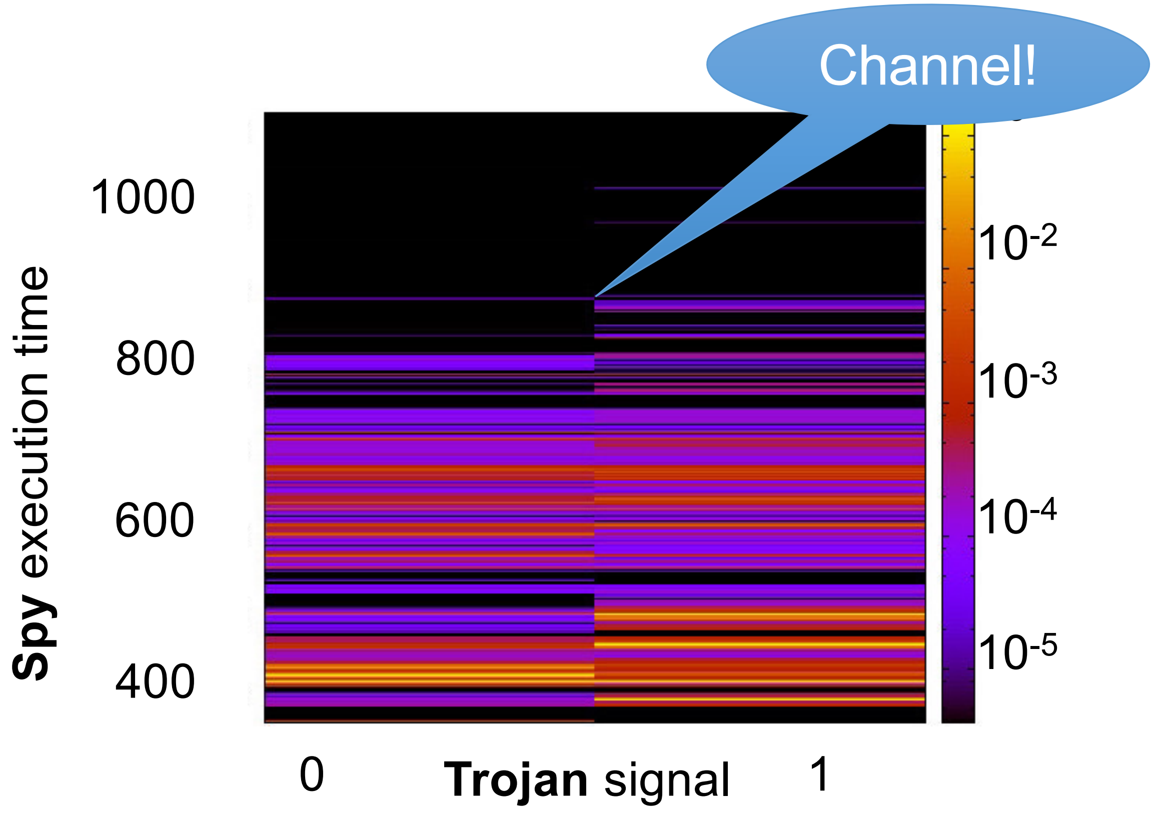


HiSilicon A53

HiSilicon A53 Branch History Buffer

Branch history buffer (BHB)

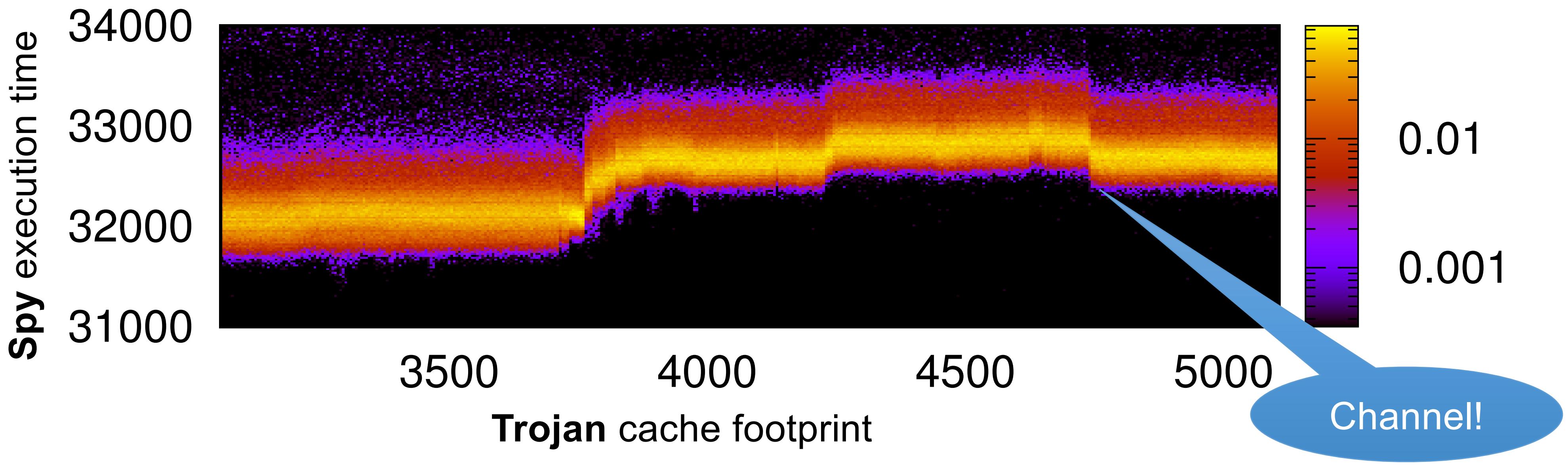
- Prediction of branch taken
- One-bit channel
- All reset operations applied



Intel Haswell Branch Target Buffer

Branch target buffer

- Prediction of branch destination
- All reset operations applied



Result Summary: Measured Capacities

Channel	Sandy Bridge		Haswell		Skylake		ARM A9		ARM A53	
	raw	flush	raw	flush	raw	flush	raw	flush	raw	flush
L1 D-cache	4.0	0.04	4.7	0.43	3.3	0.18	5.0	0.11	2.8	0.15
L1 I-cache	3.7	0.85	0.46	0.36	0.37	0.18	4.0	1.0	4.5	0.5
TLB	3.2	0.47	3.2	0.18	2.5	0.11	0.33	0.16	3.4	0.14
BTB	2.0	1.7	4.1	1.6	1.8	1.9	1.1	0.07	1.3	0.64
BHB	1.0	1.0	1.0	1.0	1.0	1.0	1.0	0.01	1.0	0.5

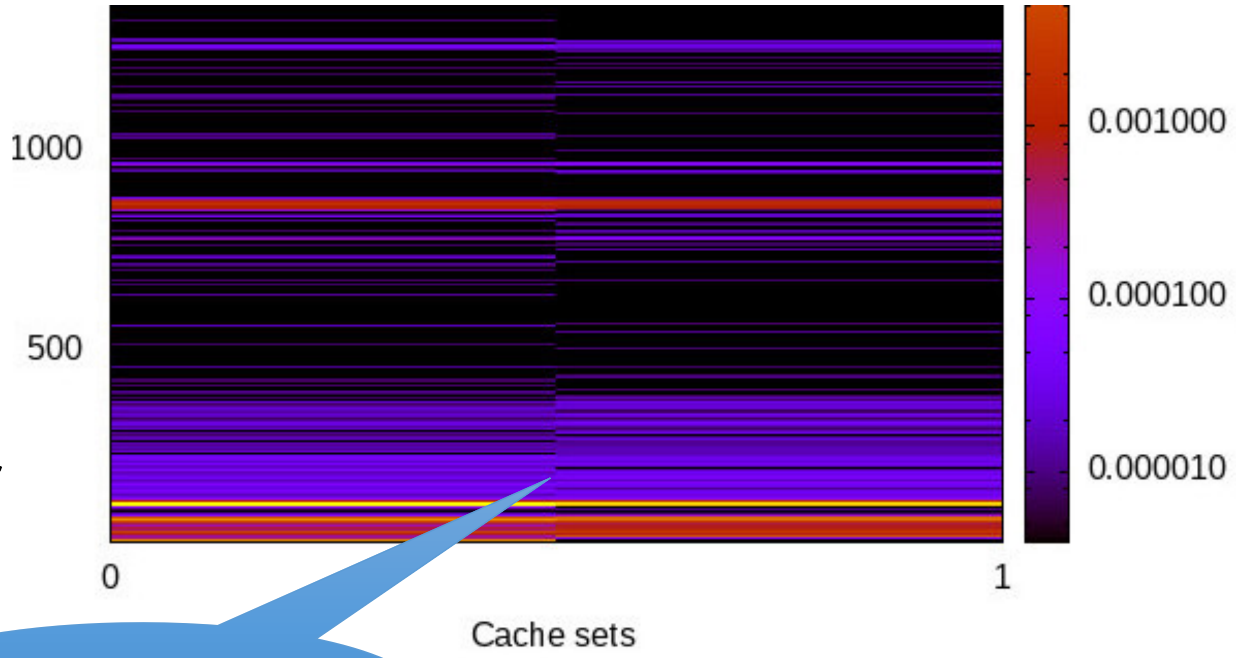
Residual channels

Uncloseable channel on each processor studied!

Intel Spectre Defences

Intel added *indirect branch control* (IBC) feature, which closes most channels, but…

Intel Skylake
Branch history buffer

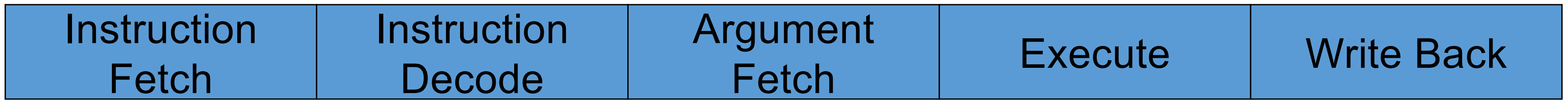


Small channel!

Speculating Desaster

Instruction Pipelining

- Nominally, the processor executes instructions one after the other
- Instruction execution consists of multiple steps
  - Each uses a different unit



Instruction Pipelining

- Nominally, the processor executes instructions sequentially
- Instruction execution consists of multiple steps
  - Each uses a different unit
- Pipelining concurrently instruction execution

```
c = a / b;
d = c + 5;
```

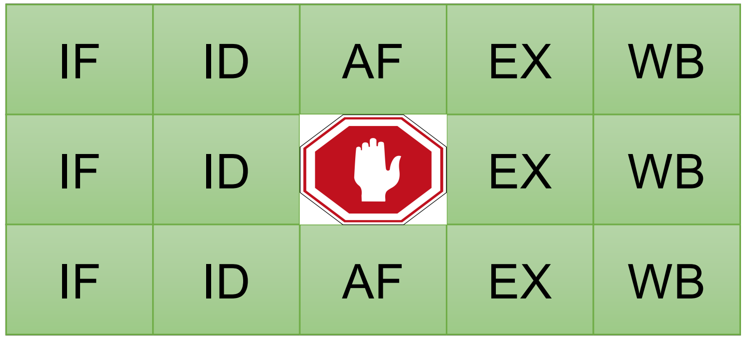
Problem: Dependencies

Inst fetch	Inst decode	Arg fetch	Execute	Write back
Inst fetch	Inst decode	Arg fetch	Execute	Write back
Inst fetch	Inst decode	Arg fetch	Execute	Write back
Inst fetch	Inst decode	Arg fetch	Execute	Write back
Inst fetch	Inst decode	Arg fetch	Execute	Write back

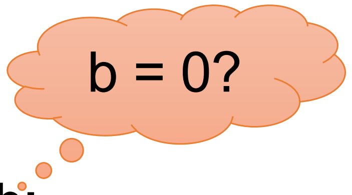
```
mulq $m0
add  %rax,$A[0]
mov  8*2($np),%rax
lea  32($tp),$tp
adc  \$0,%rdx
mov  %rdx,$A[1]
mulq $m1
add  %rax,$N[0]
mov  8($a,$j),%rax
adc  \$0,%rdx
add  $A[0],$N[0]
adc  \$0,%rdx
mov  $N[0],-24($tp)
mov  %rdx,$N[1]
mulq $m0
add  %rax,$A[1]
mov  8*1($np),%rax
adc  \$0,%rdx
mov  %rdx,$A[0]
mulq $m1
add  %rax,$N[1]
mov  ($a,$j),%rax
mov  8($a,$j),%rax
adc  \$0,%rdx
```

Out-of-Order Execution

- Execute instructions when data is available



Completed instructions wait in *reorder buffer* until all previous ones *retired*

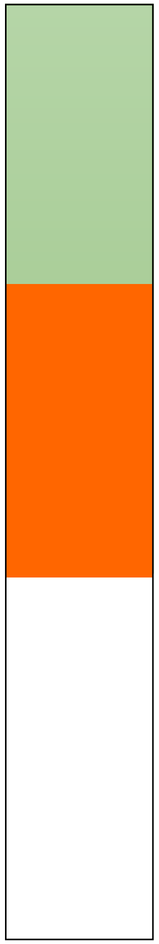


c = a / b;

d = c + 5;

e = f + g;

Out-of-order is speculative!



Out-of-Order Execution

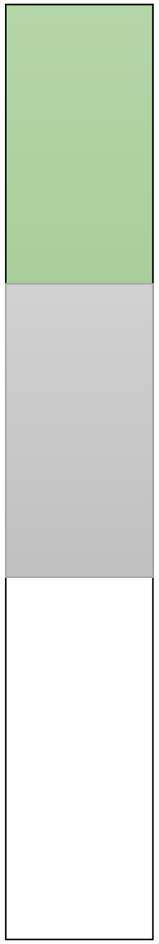
- Abandon instructions if never executed in program order

IF	ID	AF	EX	WB
IF	ID	AF	EX	WB
IF	ID	AF	EX	WB

b == 0!

```
c = a / b;
d = c + 5;
e = f + g;
```

Also useful for branches



Speculative Execution and Branches

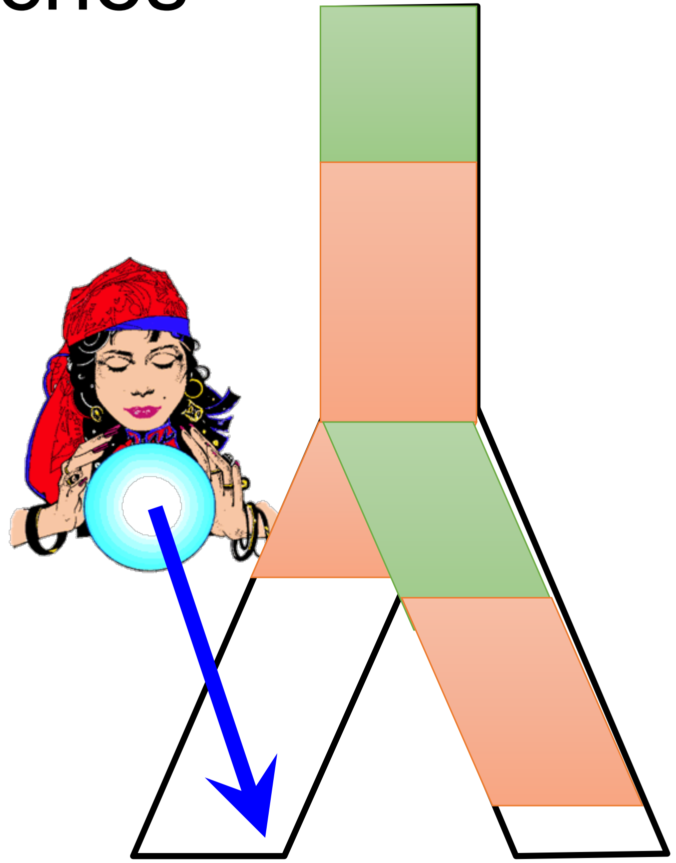
When execution reaches a branch:

- Predict outcome of branch
- Proceed (speculatively!) along predicted branch

Correct prediction: All good

Mis-prediction: Abandon and resume

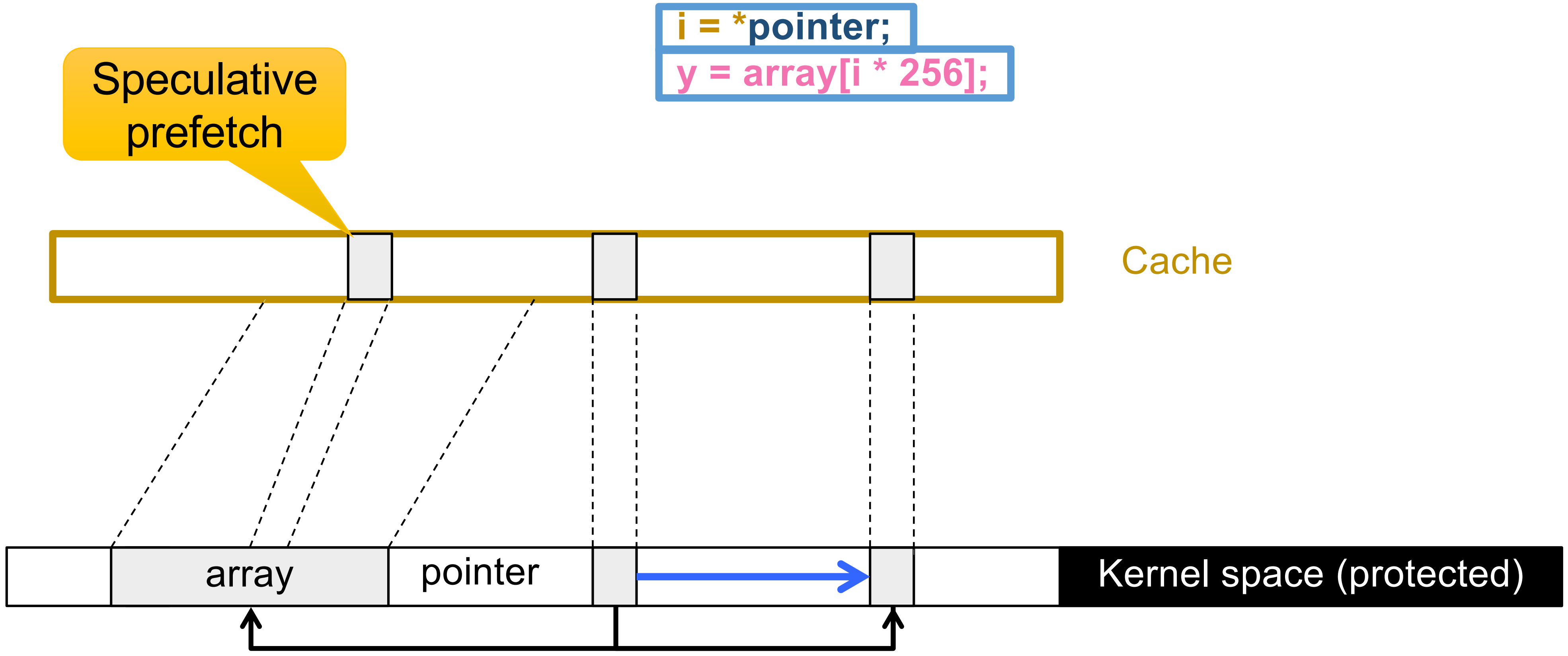
Minor problem: Speculation pollutes cache!



Speculating Desaster

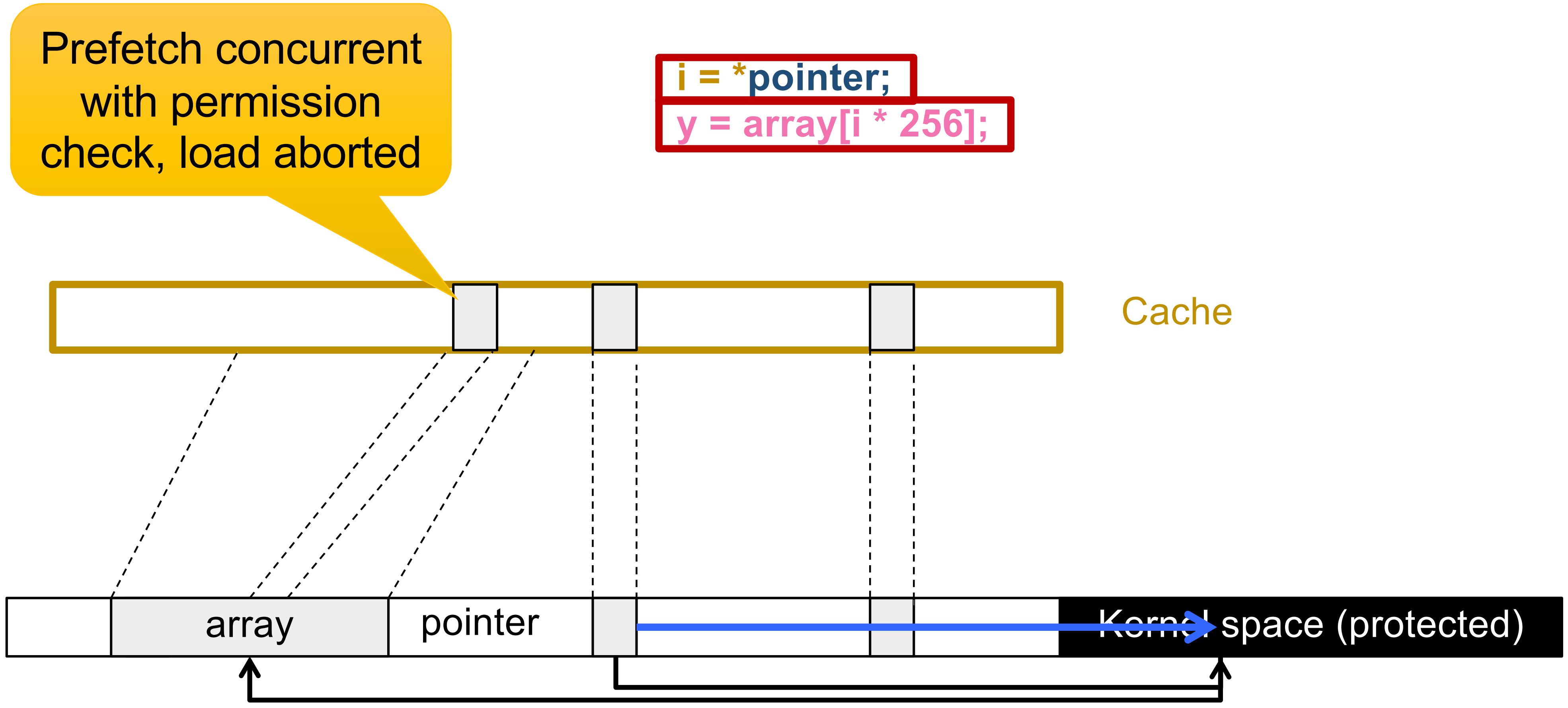
Meltdown

Meltdown: Speculative Load

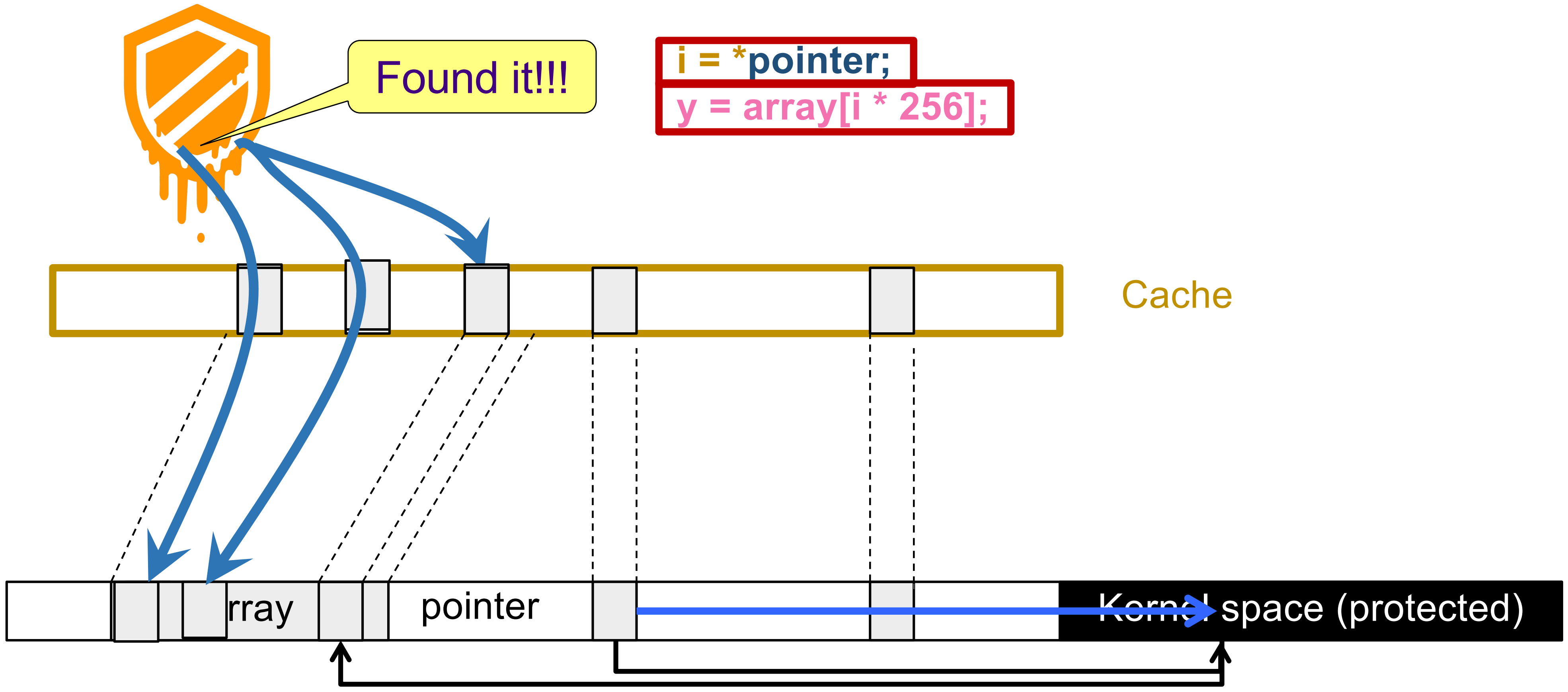


Meltdown: Speculative Loads

```
i = *pointer;
y = array[i * 256];
```

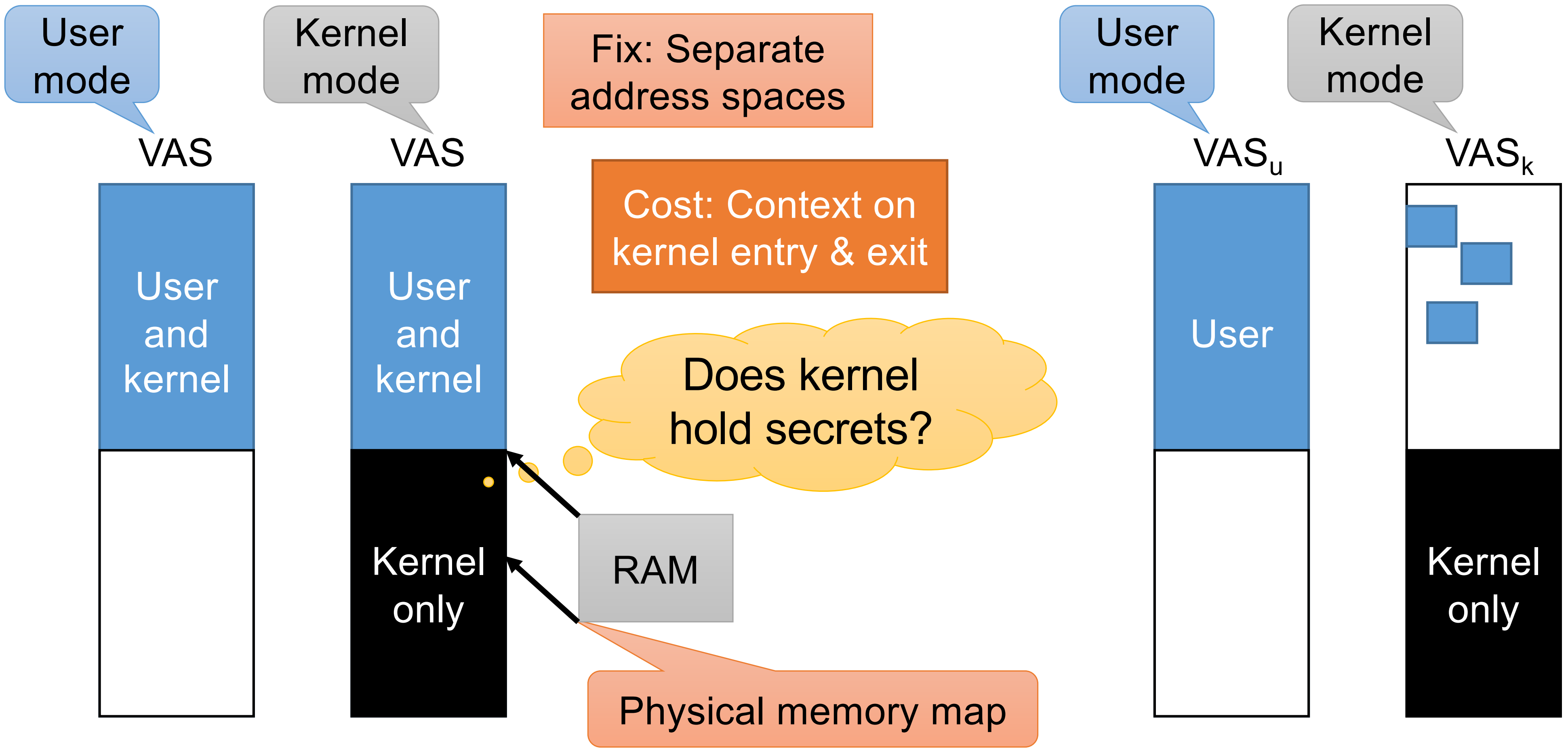


Meltdown: Cache-Channel to Read

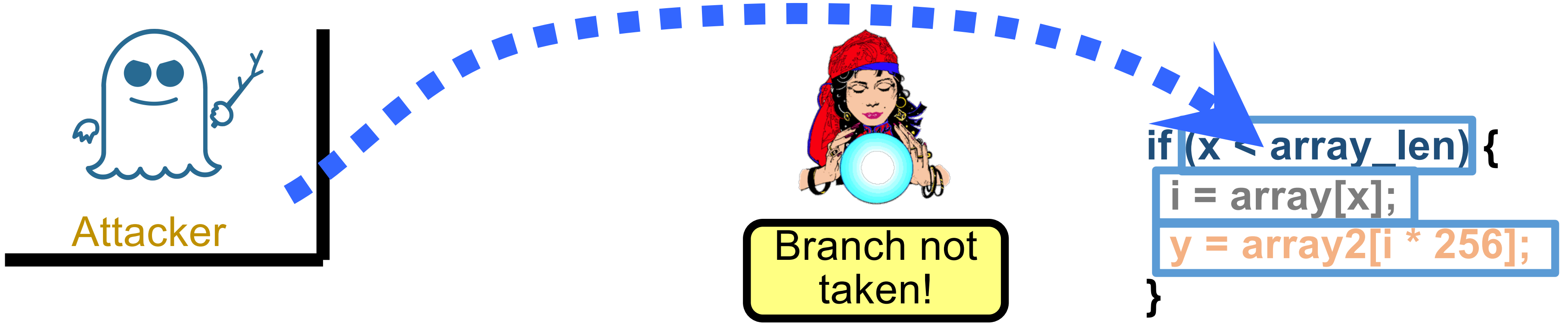


```
i = *pointer;
y = array[i * 256];
```

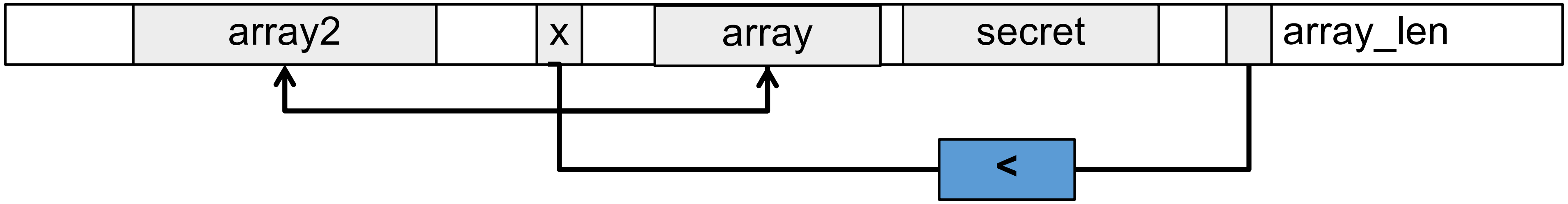
Meltdown: Full Kernel Memory Disclosure



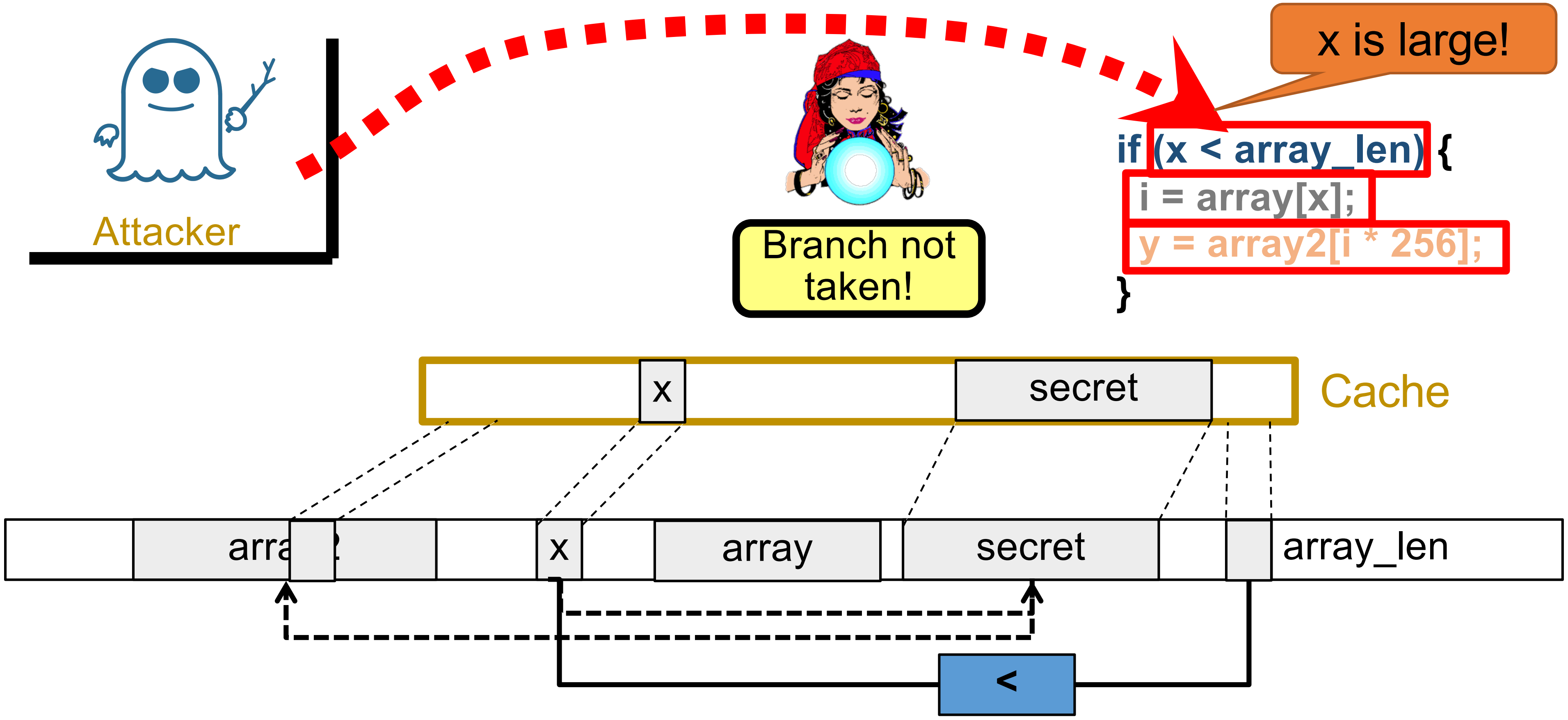
Spectre: Branch Prediction (Variant 1)



```
if (x < array_len) {
  i = array[x];
  y = array2[i * 256];
}
```

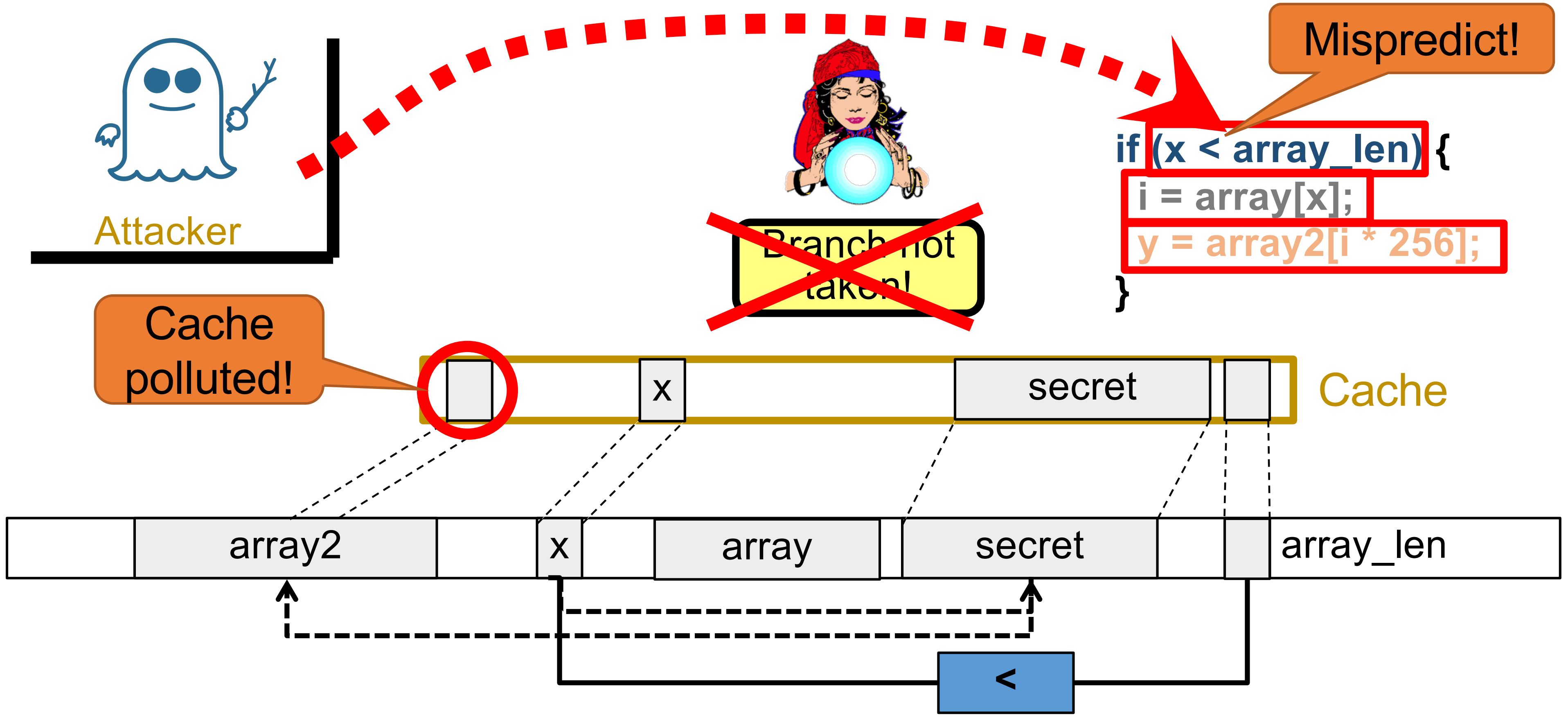


Spectre: Branch Prediction (Variant 1)



```
if (x < array_len) {
  i = array[x];
  y = array2[i * 256];
}
```

Spectre: Branch Prediction (Variant 1)



Reaction

consistent with spec, i.e. ISA

Steve Smith, Corporate vice president, Intel

“The processor is, in fact, operating as it is designed,” Smith said. “And in every case, it's been this side-channel approach that the researchers used to gain information even while the processor is executing normally its intended functions.”

Inevitable conclusion:

- This ISA is an insufficient contract for building secure systems
- **We need a new hardware-software contract!**