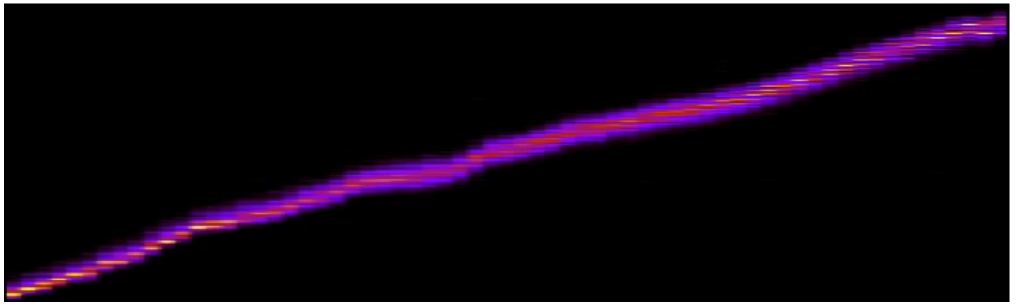


School of Computer Science & Engineering

COMP9242 Advanced Operating Systems

2026 T3 Week 08 Part 1

Security: Secure Operating Systems & Information Leakage



Gernot Heiser

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

Copyright Notice

These slides are distributed under the
Creative Commons Attribution 4.0 International (CC BY 4.0) License

- You are free:
  - to share—to copy, distribute and transmit the work
  - to remix—to adapt the work
- under the following conditions:
  - **Attribution:** You must attribute the work (but not in any way that suggests that the author endorses you or your use of the work) as follows:

    "Courtesy of Gernot Heiser, UNSW Sydney"

The complete license text can be found at
http://creativecommons.org/licenses/by/4.0/legalcode

© Gernot Heiser 2019–25 – CC BY 4.0

Today's Lecture

- Mandatory Access-Control Policies
- Secure Operating Systems
- Timing Channels
- Speculation Attacks (Meltdown/Spectre)

© Gernot Heiser 2019–25 – CC BY 4.0

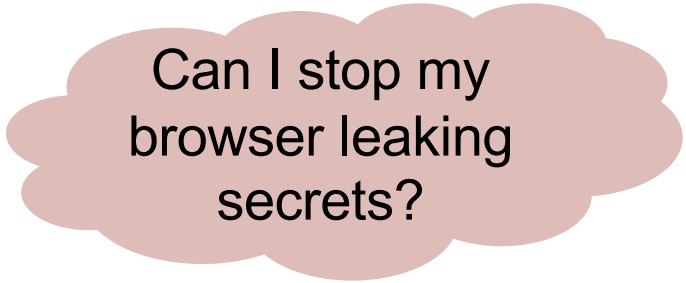
Mandatory AC Policies

© Gernot Heiser 2019–25 – CC BY 4.0

Mandatory vs Discretionary Access Control

Discretionary Access Control (DAC):

- Users can make access control decisions
  - Delegate their access to other users etc.



Mandatory Access Control (MAC):

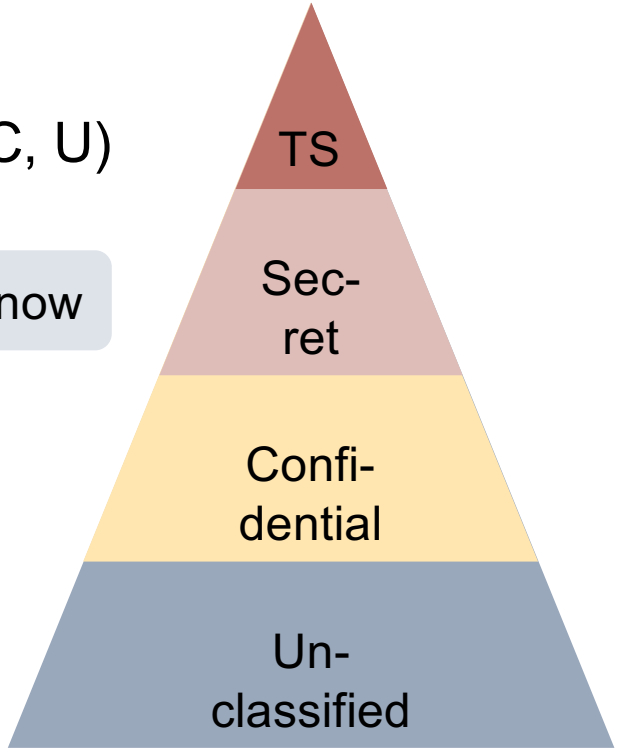
- System enforces administrator-defined policy
- Users can only make access control decisions subject to mandatory policy
- Can limit damage caused by untrusted applications
- Traditionally used in national security environments

© Gernot Heiser 2019–25 – CC BY 4.0

MAC: Bell & LaPadula (BLP) Model [1966]

- MAC Policy/Mechanism
  - Formalises national security classifications
- Every object assigned a **classification** (eg TS, S, C, U)
  - orthogonal security **compartments**
- Classifications ordered in a **lattice**
  - e.g. TS > S > C > U
- Every subject assigned a **clearance**
  - Highest classification it's allowed to learn

Need-to-know



Labelled security:

- Subjects and objects are *labelled*
- Permitted accesses: relation over labels
  `allow(subject.label, object.label, operation)`

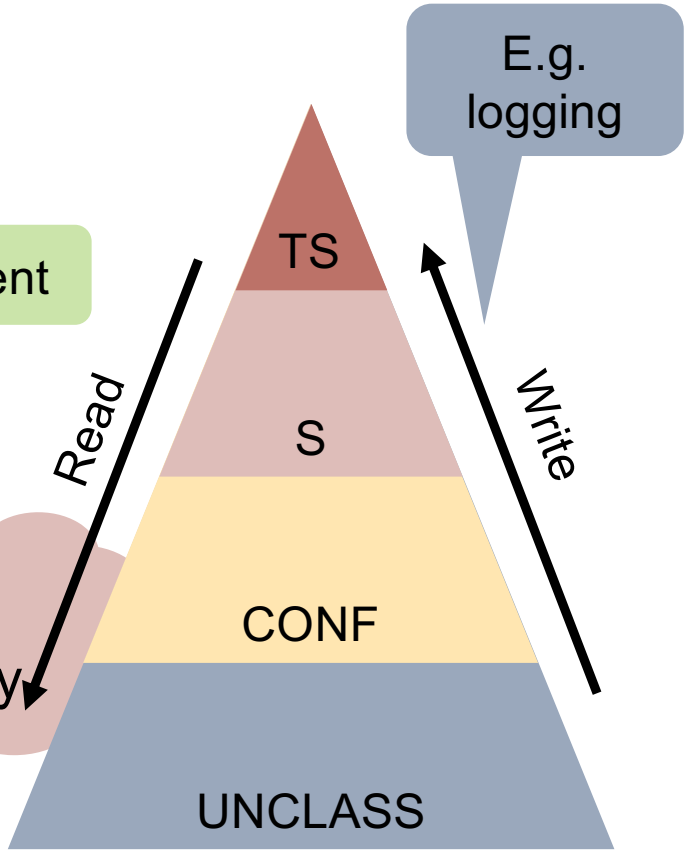
© Gernot Heiser 2019–25 – CC BY 4.0

BLP: Rules

- **Simple Security Property** (“no read up”):
  - **s** can read **o** iff `clearance(s) >= class(o)`
  - S-cleared subject can read U,C,S but not TS
  - standard confidentiality
- **★-Property** (“no write down”):
  - **s** can write **o** iff `clearance(s) <= class(o)`
  - S-cleared subject can write TS,S, but not C,U
  - to prevent accidental or malicious leakage of data to lower levels
- In practice need exceptions
  - allow *trusted* entity to write down
  - *“de-classify/downgrade”* – e.g. encryption

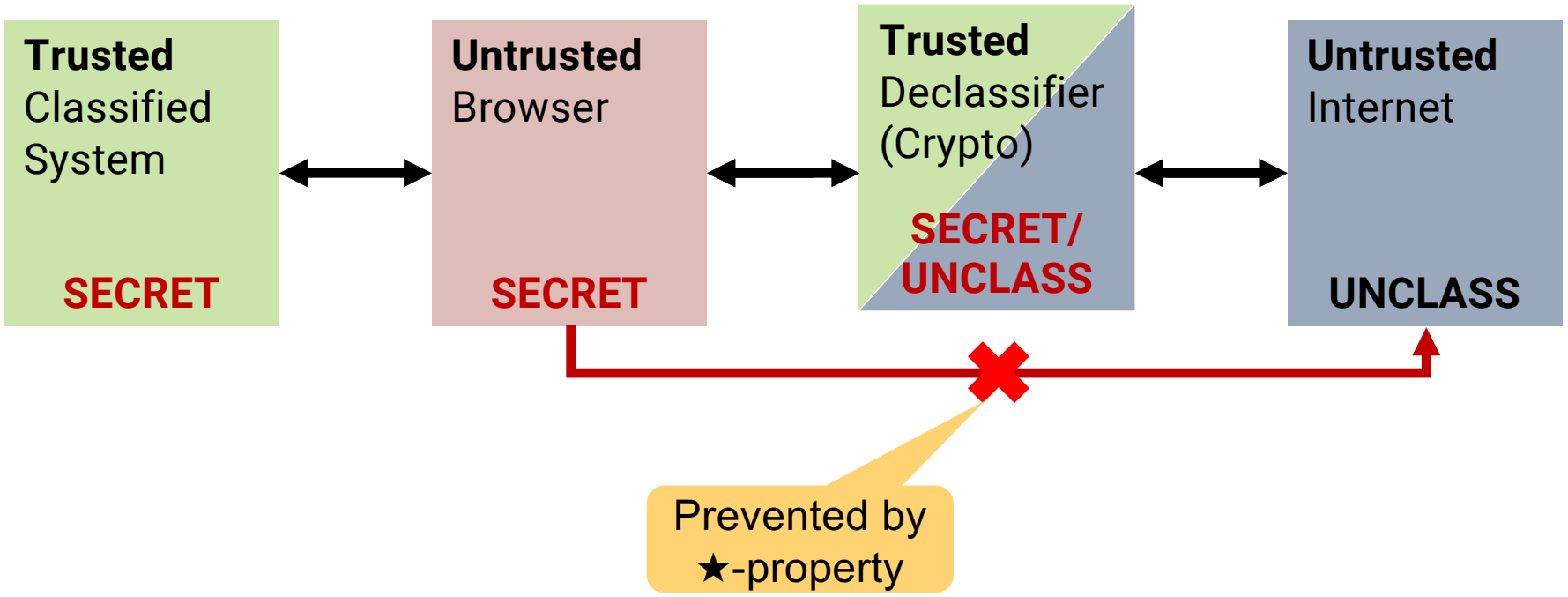
Confinement

Trend to over-classify



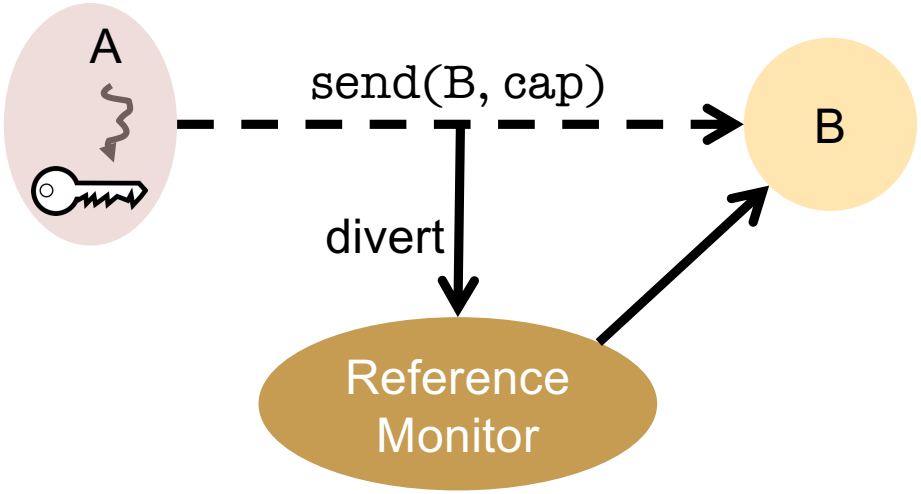
© Gernot Heiser 2019–25 – CC BY 4.0

Web Browser Example



© Gernot Heiser 2019–25 – CC BY 4.0

MAC With Caps: Reference Monitor

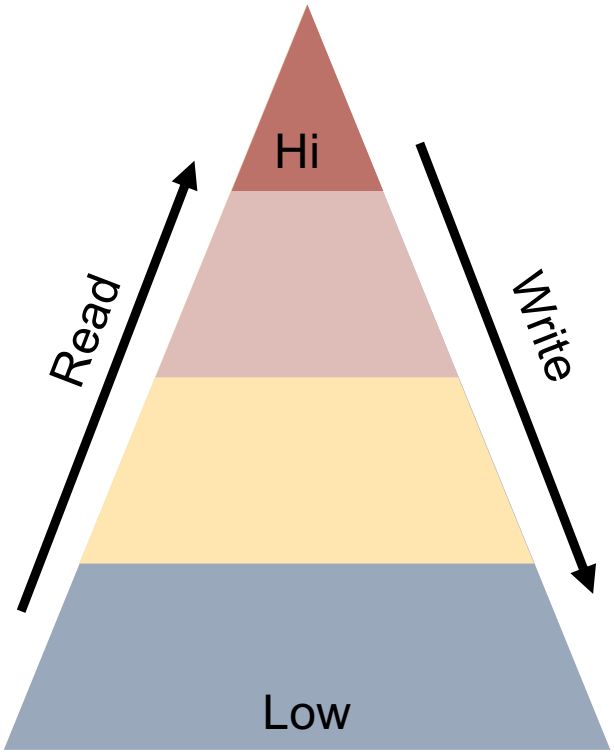



```
interpose_transfer(cap) {
   if (A.clear > B.clear) {
      c = mint(cap, -r);
      send(B,c);
   } else if (A.clear < B.clear) {
      c = mint(cap, -w);
      send(B,c);
   } else {
      send(B,cap);
   }
}
```


© Gernot Heiser 2019–25 – CC BY 4.0

MAC: Biba Integrity Model

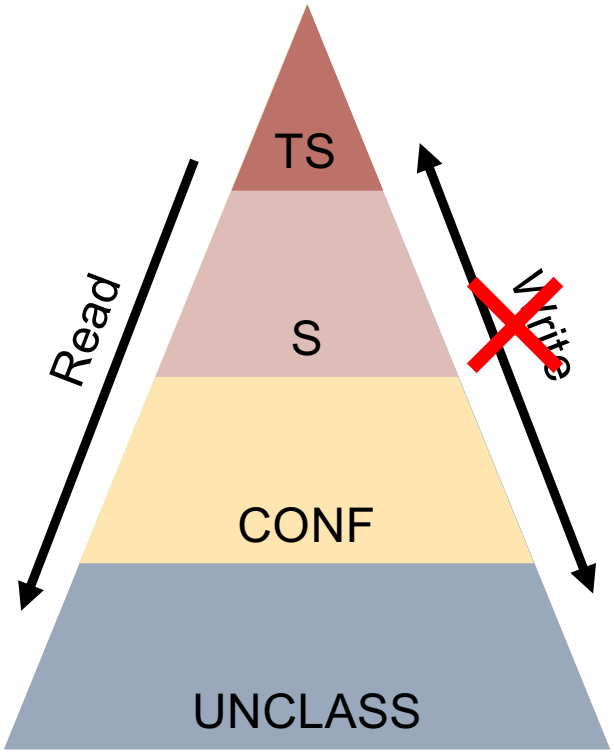
- Bell-LaPadula enforces **confidentiality**
- **Biba:** Its dual, enforces **integrity**
- Objects now carry **integrity** classification
- Subjects labelled by **lowest** level of data each subject is allowed to learn
- BLP order is inverted:
  - **s** can read **o** iff `clearance(s) <= class(o)`
  - **s** can write **o** iff `clearance(s) >= class(o)`



© Gernot Heiser 2019–25 – CC BY 4.0

Confidentiality + Integrity

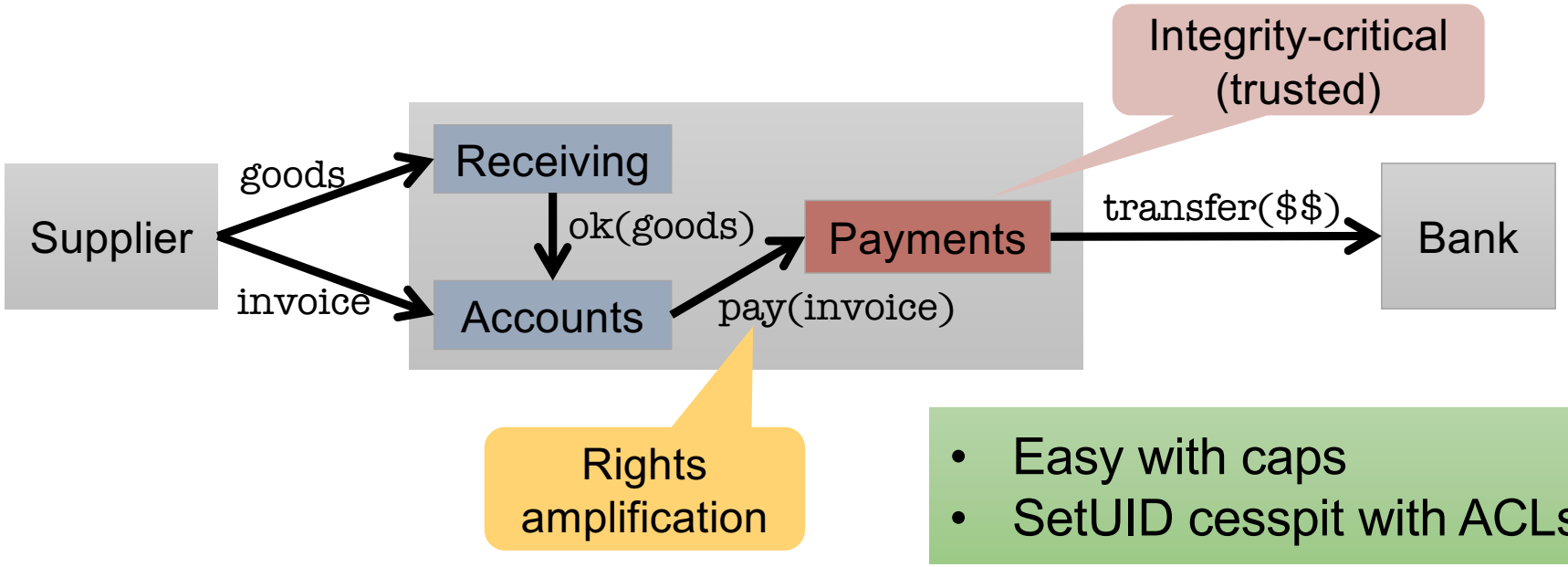
- BLP+Biba allows no information flow across classes
- Practicality requires weakening
  - Assume high-classified subject to treat low-integrity info responsibly
  - Allow read-down
  - Independent labelling for confidentiality and integrity
- **Strong *-Property** (“matching writes only”):
  - **s** can write **o** iff `clearance(s) = class(o)`
  - Eg for logging, high reads low data and logs



© Gernot Heiser 2019–25 – CC BY 4.0

Clark & Wilson Model

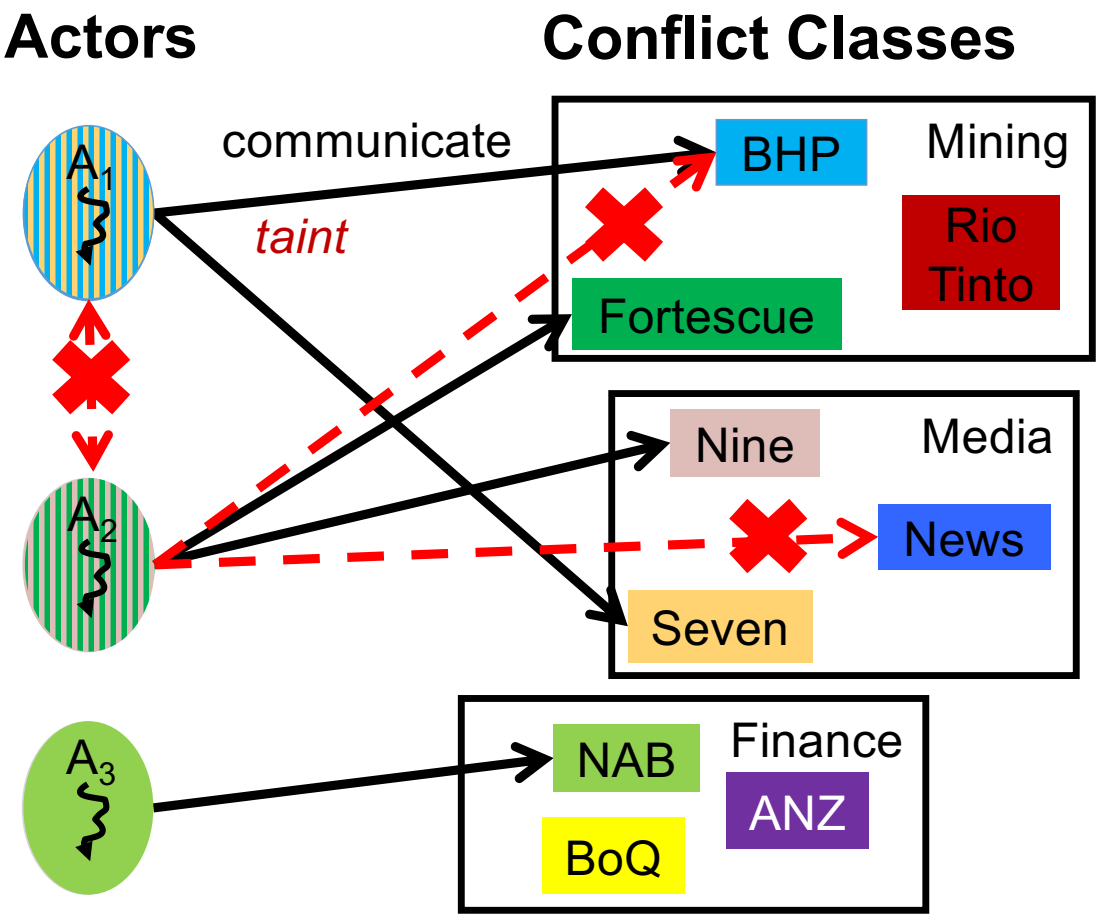
- In commercial settings integrity is more important than confidentiality
- Restrict possible operations to *well-formed transactions*
  - eg payment issued only after goods and invoice received



- Easy with caps
- SetUID cesspit with ACLs

© Gernot Heiser 2019–25 – CC BY 4.0

Chinese Wall (aka Brewer & Nash) Model



Dynamic policy:

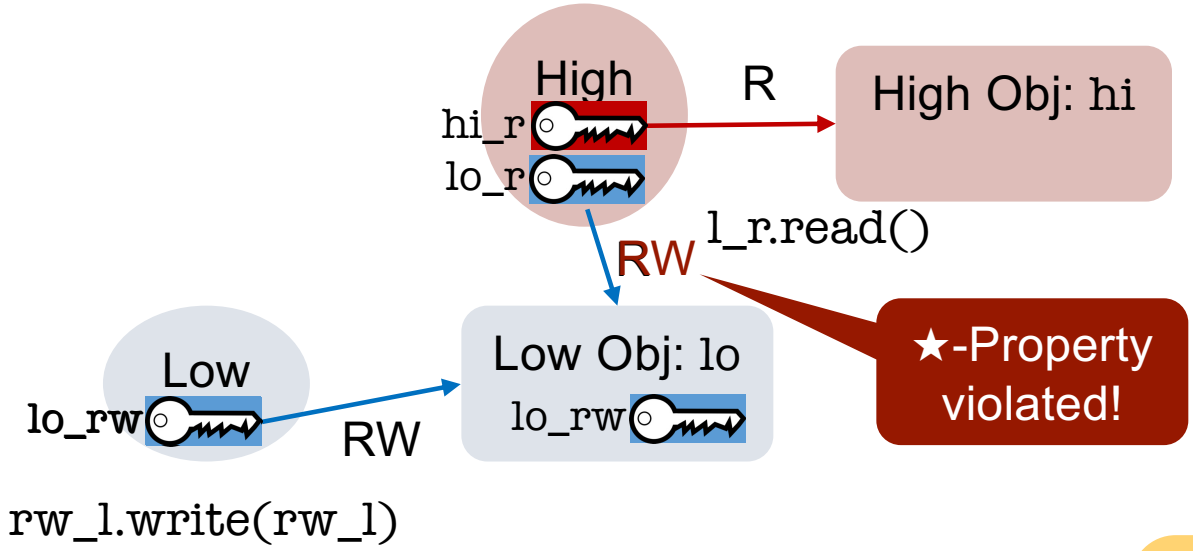
- Communication *taints* actor
- tainted actor cannot communicate with other entity in same conflict class

Conflict-of-interest prevention

- Law practices
- Consultancies

© Gernot Heiser 2019–25 – CC BY 4.0

Boebert's Attack on Capability Machines



Takeaway:

- Need mechanism to limit cap propagation: *take-grant model*
- seL4 *grant* right on endpoints

"On the inability of an unmodified capability machine to enforce the ★-property" [Boebert'84]

Works where caps are indistinguishable from data (HW & sparse caps)

© Gernot Heiser 2019–25 – CC BY 4.0

Decidability

Safety: Given initial *safe state s*, system will never reach *unsafe state s'*

Decidability: AC system is decidable if safety can aways be computationally determined

Equivalent to halting problem [Harrison, Ruzzo, Ullman '75]

- Most capability systems are decidable
- Unclear for many common ACL systems

© Gernot Heiser 2019–25 – CC BY 4.0

Summary: AC Principles

- ACLs and Capabilities:
  - Capabilities tend to better support least privilege
  - But ACLs can be better for auditing
- MAC good for global security requirements
- Not all mechanisms can enforce all policies
  - e.g. ★-property with sparse or HW capabilities
- AC systems should be decidable so we can reason about security

© Gernot Heiser 2019–25 – CC BY 4.0

Secure Operating Systems

Principles

© Gernot Heiser 2019–25 – CC BY 4.0

What is a Secure OS?

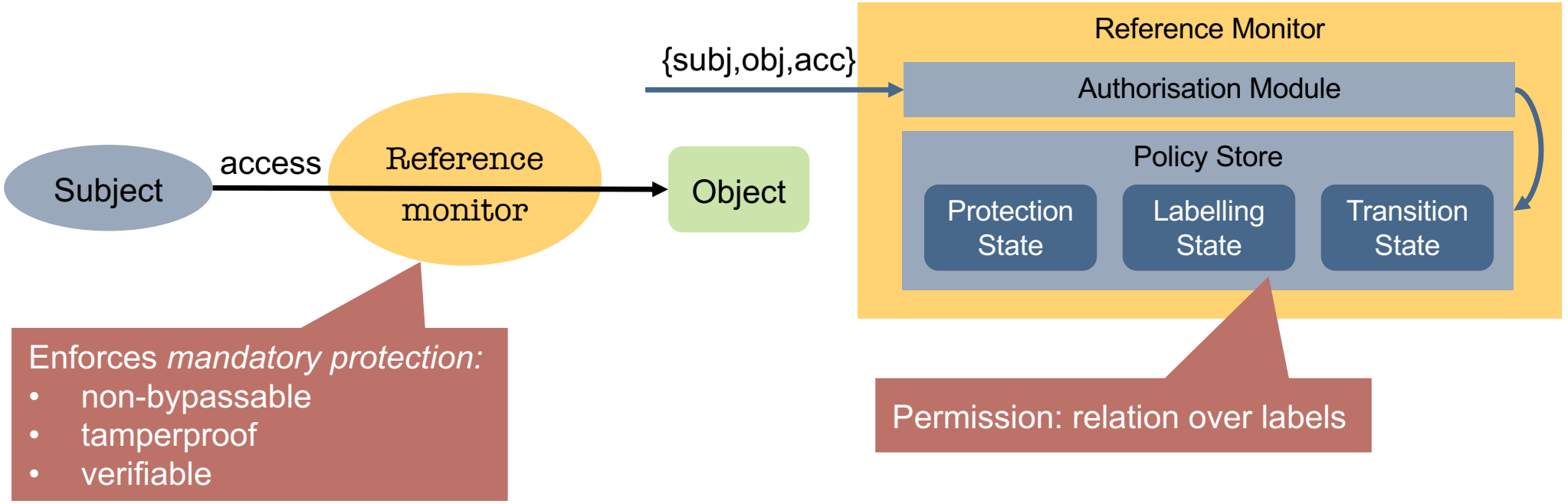
An OS is *secure* if it enforces the system's *security policy*.

© Gernot Heiser 2019–25 – CC BY 4.0

Secure Operating Systems

Secure OS: [Jaeger: OS Security]

Access enforcement satisfies the *reference monitor* concept



© Gernot Heiser 2019–25 – CC BY 4.0

Policy examples

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

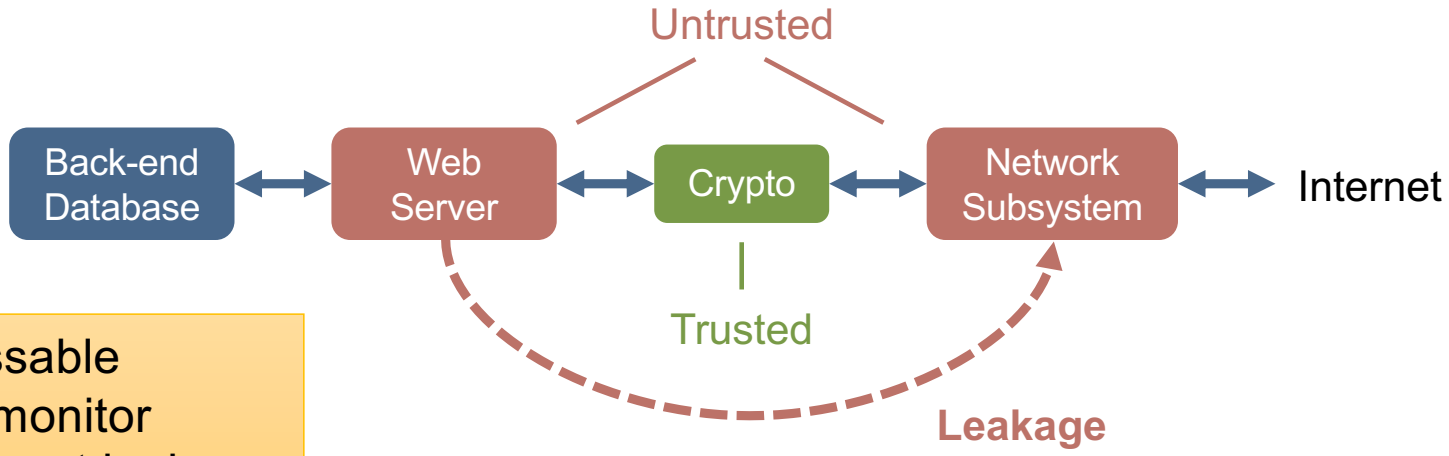
Isolation (e.g. VMs in public cloud):
- no read access to other VM
- no write access to other VM

Integrity:
- untrusted code must not write outside its space

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

© Gernot Heiser 2019–25 – CC BY 4.0

Confinement Example



Non-bypassable reference monitor should prevent leakage

Challenges:

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

© Gernot Heiser 2019–25 – CC BY 4.0

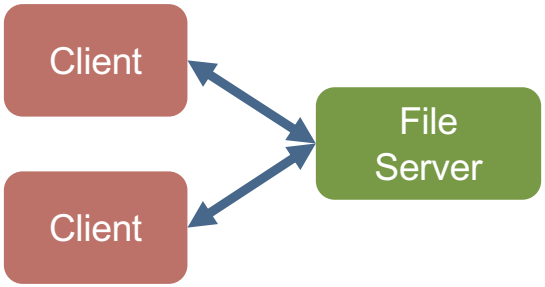
How Secure Are Traditional OSes?

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)

© Gernot Heiser 2019–25 – CC BY 4.0

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

© Gernot Heiser 2019–25 – CC BY 4.0

Timing Channels

Principles

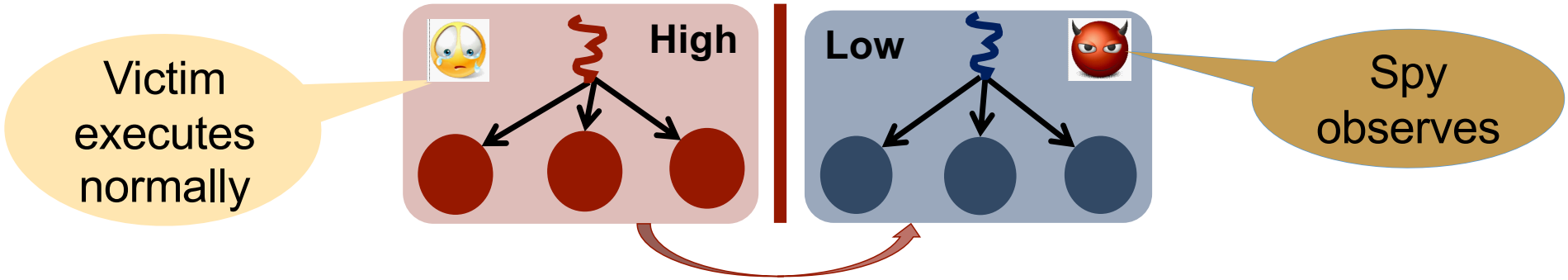
© Gernot Heiser 2019–25 – CC BY 4.0

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



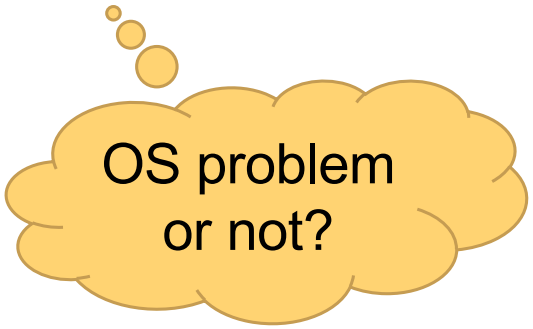
Side channel: Covert channel exploitable without insider help

© Gernot Heiser 2019–25 – CC BY 4.0

Causes of Timing Channels

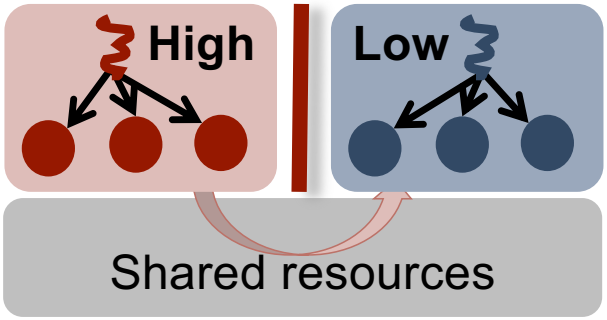
Algorithmic

```
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)

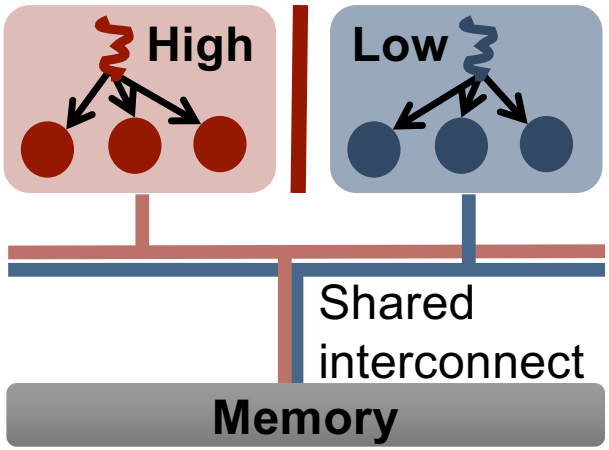


Affect execution speed

Micro-architectural timing channels

© Gernot Heiser 2019–25 – CC BY 4.0

Shared Hardware: Stateless Interconnect

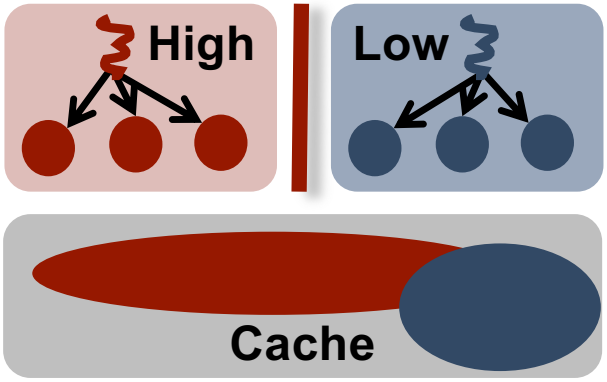


H/W is *bandwidth-limited*

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

© Gernot Heiser 2019–25 – CC BY 4.0

Shared Hardware: Stateful Resources



Shared cache

H/W is *capacity-limited*

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

Can be any state-holding microarchitectural feature:

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

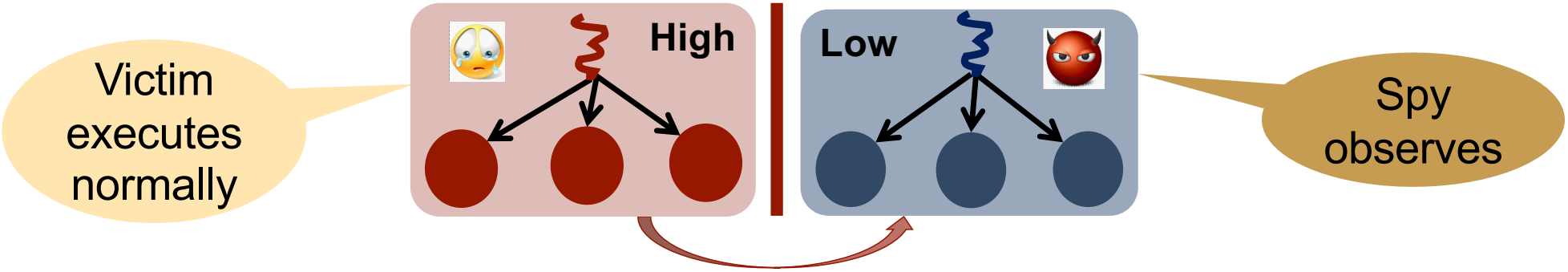
© Gernot Heiser 2019–25 – CC BY 4.0

Timing Channels

Example: LLC Side Channel

© Gernot Heiser 2019–25 – CC BY 4.0

Methodology: Prime and Probe



1. Fill cache with own data

2. Touch cache lines

Input Signal

3. Traverse cache, measure execution time

Output Signal

© Gernot Heiser 2019–25 – CC BY 4.0

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

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

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

© Gernot Heiser 2019–25 – CC BY 4.0

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

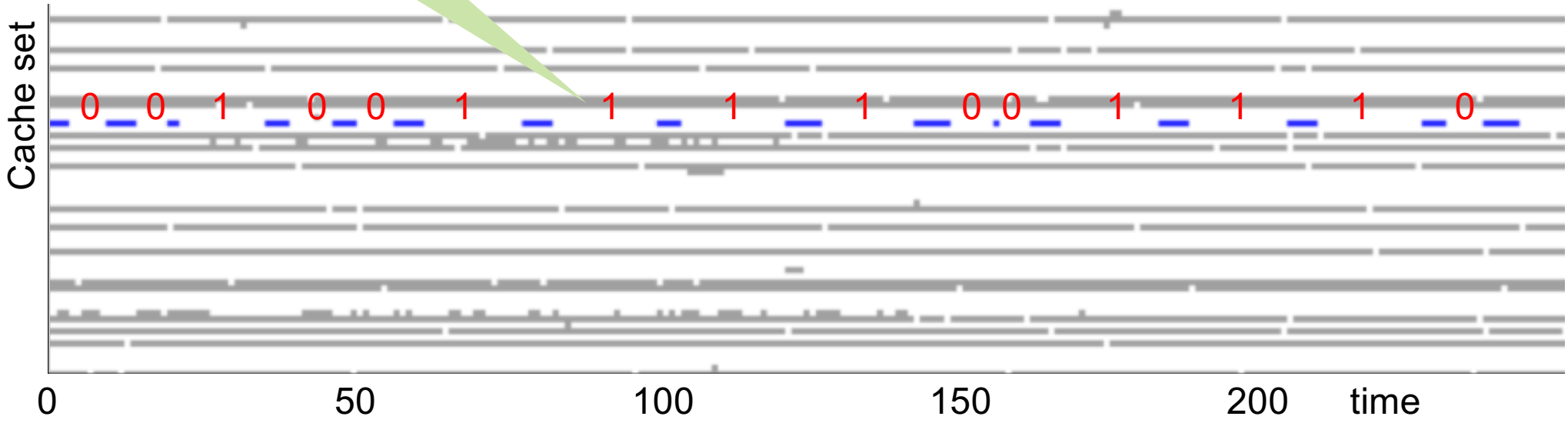
Expected pattern: Bursts of activity separated by longer or shorter intervals indicating modular reduction operation

© Gernot Heiser 2019–25 – CC BY 4.0

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



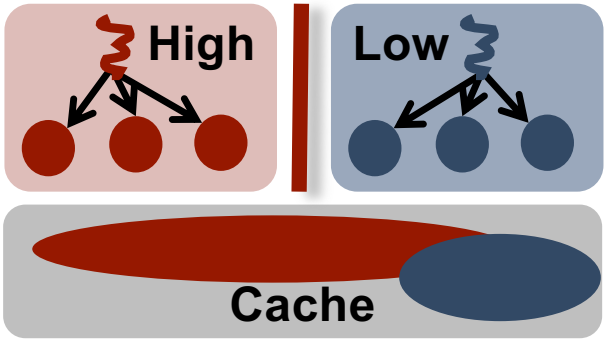
© Gernot Heiser 2019–25 – CC BY 4.0

Timing Channels

Evaluating Hardware

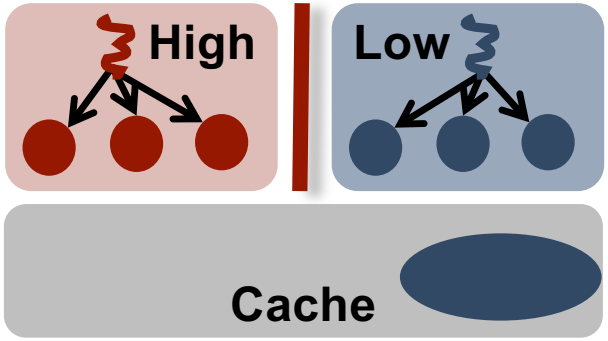
© Gernot Heiser 2019–25 – CC BY 4.0

Timing-Channel Prevention: Partition HW

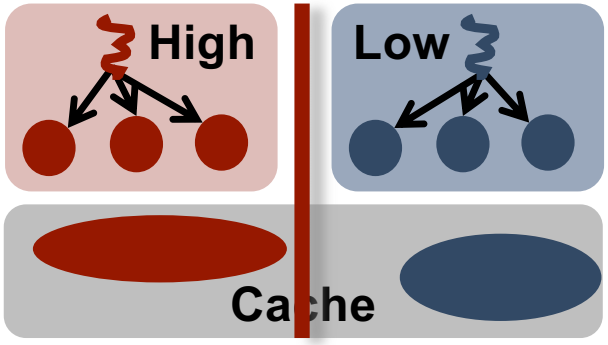


Temporally partition

Flush



Spatially partition



Need both!

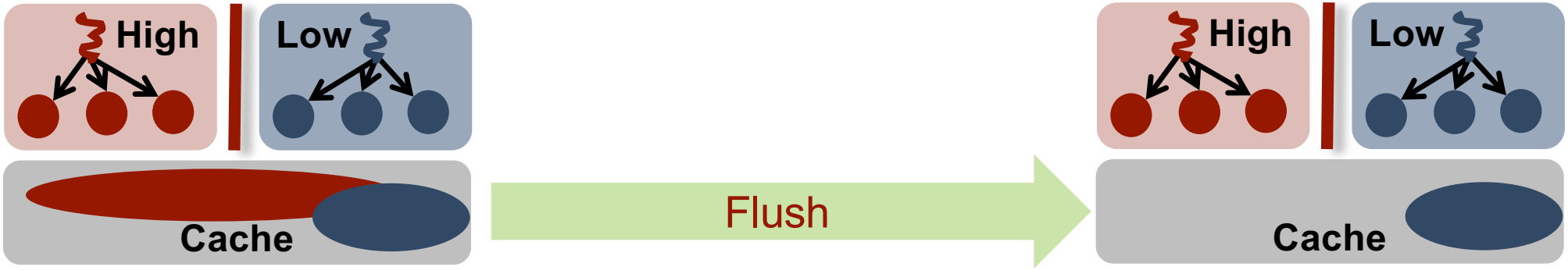
Flushing useless with concurrent access:
- HW threads (SMT)
- Multicore

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

No partitioning support for interconnects!

© Gernot Heiser 2019–25 – CC BY 4.0

Evaluating Intra-Core Channels

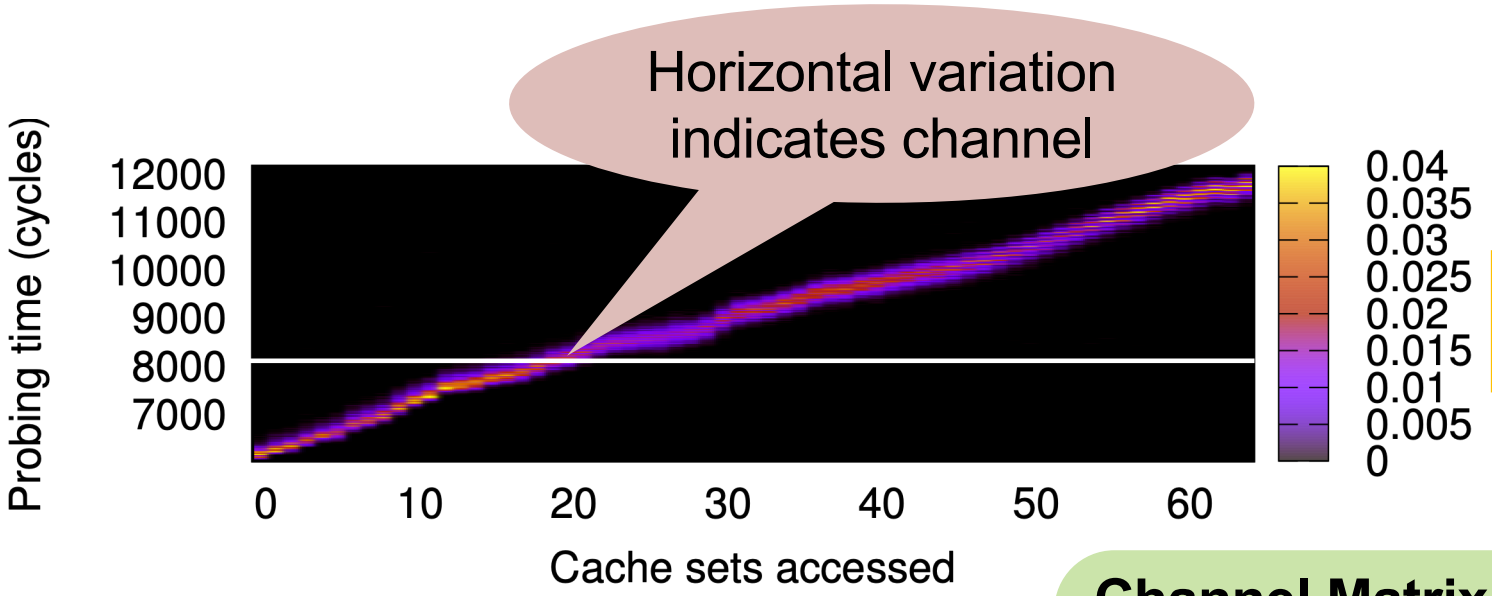


Methodology:

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

© Gernot Heiser 2019–25 – CC BY 4.0

Methodology: Channel Matrix



Raw I-cache channel
Intel Sandy Bridge

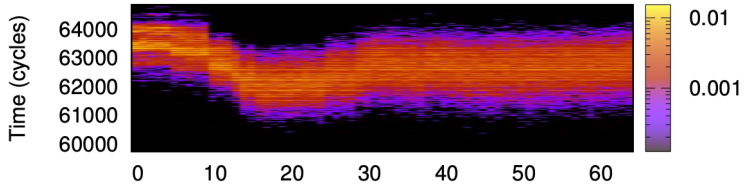
Channel Matrix:

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

© Gernot Heiser 2019–25 – CC BY 4.0

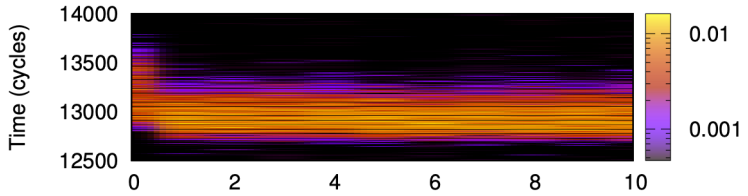
I-Cache Channel With Full State Flush

CHANNEL!



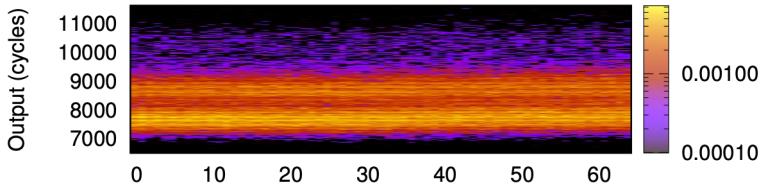
Intel Sandy Bridge

CHANNEL!



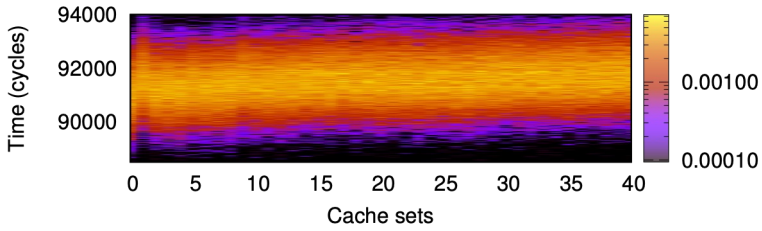
Intel Haswell

No evidence of channel



Intel Skylake

SMALL CHANNEL!



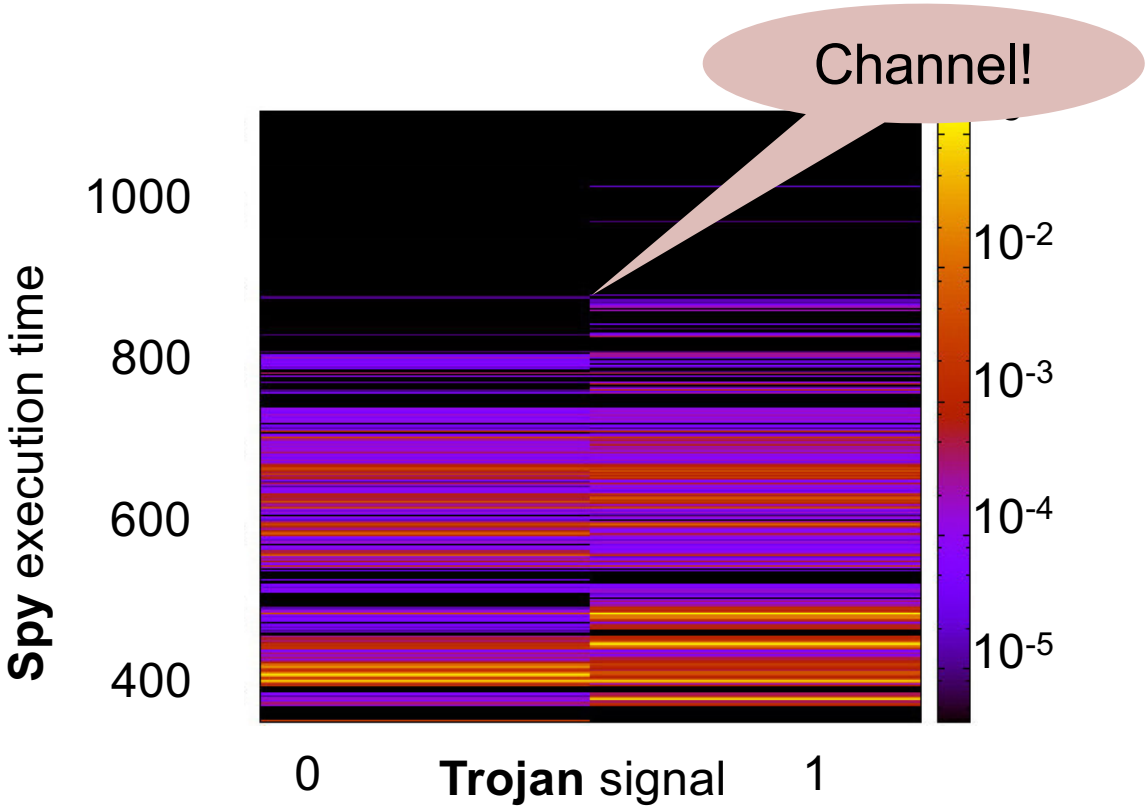
HiSilicon A53

© Gernot Heiser 2019–25 – CC BY 4.0

HiSilicon A53 Branch History Buffer

Branch history buffer (BHB)

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

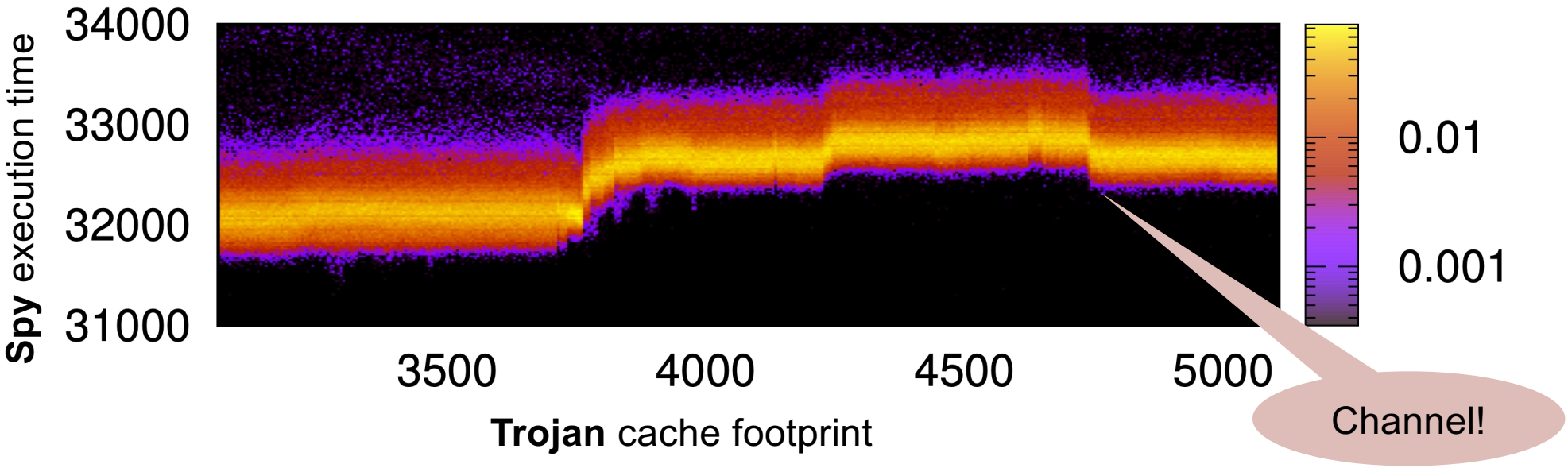


© Gernot Heiser 2019–25 – CC BY 4.0

Intel Haswell Branch Target Buffer

Branch target buffer

- Prediction of branch destination
- All reset operations applied



© Gernot Heiser 2019–25 – CC BY 4.0

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



Uncloseable channel on each processor studied!

© Gernot Heiser 2019–25 – CC BY 4.0

Speculating Desaster

© Gernot Heiser 2019–25 – CC BY 4.0

Instruction Pipelining

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



© Gernot Heiser 2019–25 – CC BY 4.0

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,%rdD
```


© Gernot Heiser 2019–25 – CC BY 4.0

Out-of-Order Execution

- Execute instructions when data is available

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

b = 0?

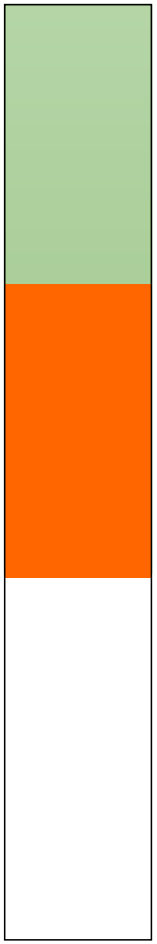
c = a / b;

d = c + 5;

e = f + g;

Out-of-order is speculative!

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



© Gernot Heiser 2019–25 – CC BY 4.0

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



© Gernot Heiser 2019–25 – CC BY 4.0

Speculative Execution and Branches

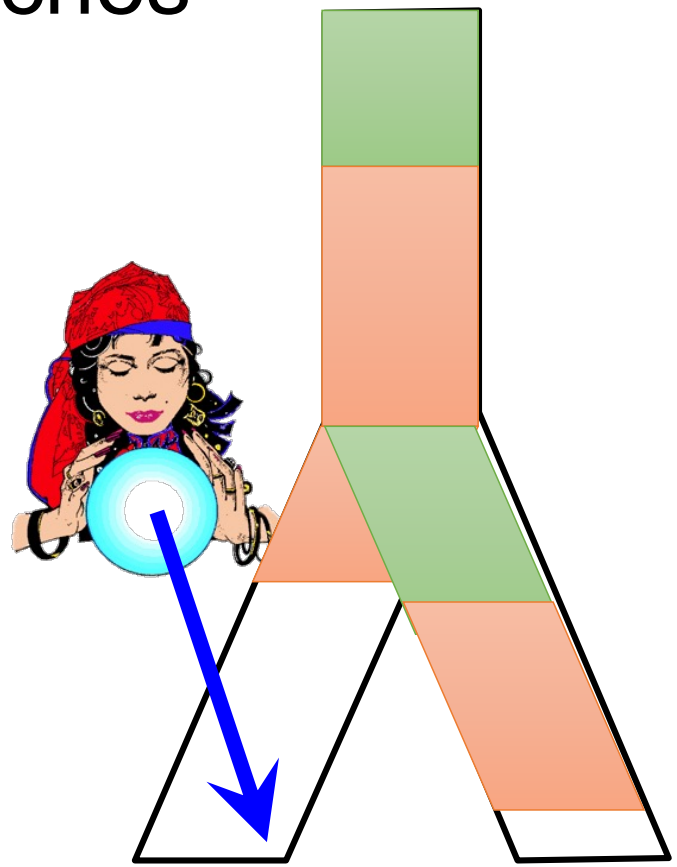
When execution reaches a branch:

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

Correct prediction: All good

Mis-prediction: Abandon and resume

Minor problem: Speculation pollutes cache!



© Gernot Heiser 2019–25 – CC BY 4.0

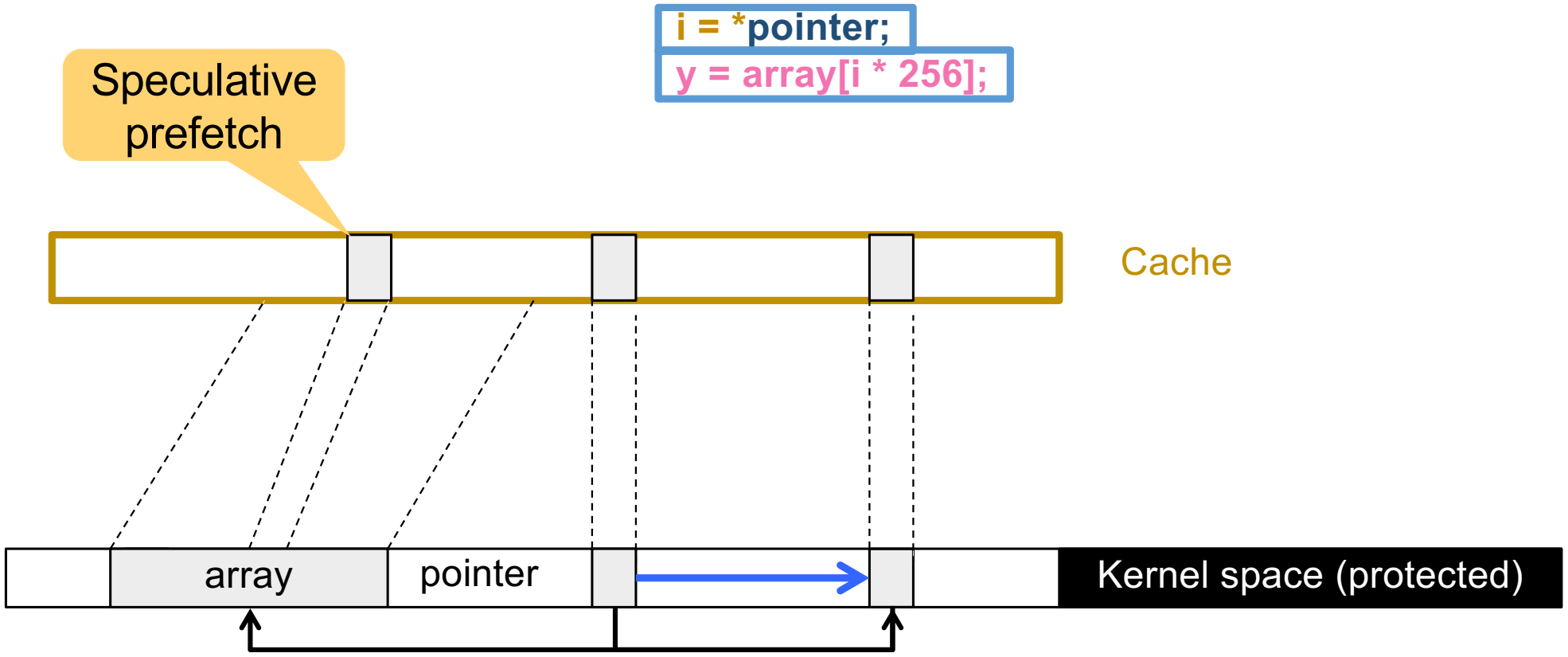
Speculating Desaster

Meltdown



© Gernot Heiser 2019–25 – CC BY 4.0

Meltdown: Speculative Load

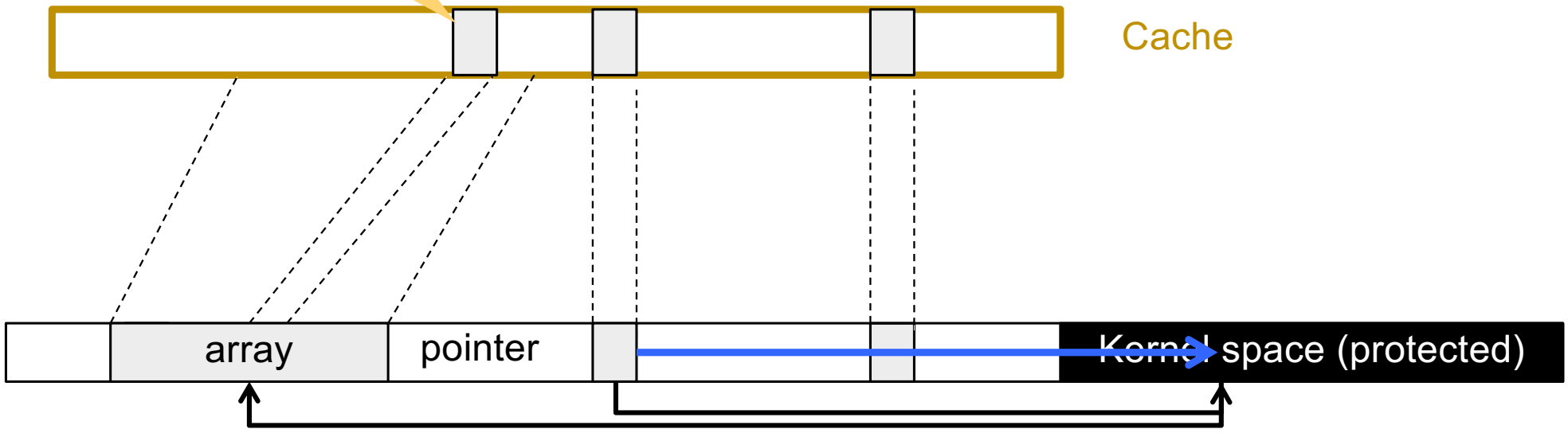


© Gernot Heiser 2019–25 – CC BY 4.0

Meltdown: Speculative Loads

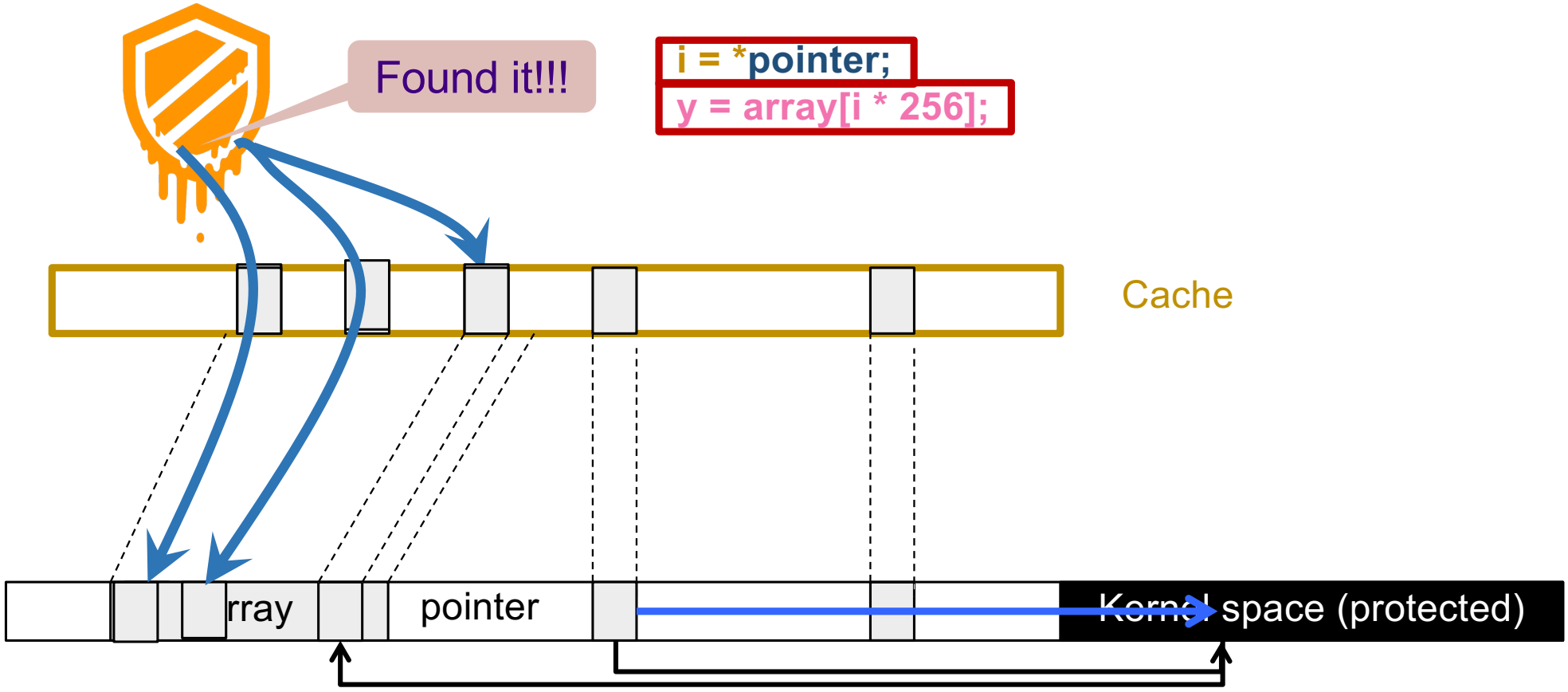
Prefetch concurrent with permission check, load aborted

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



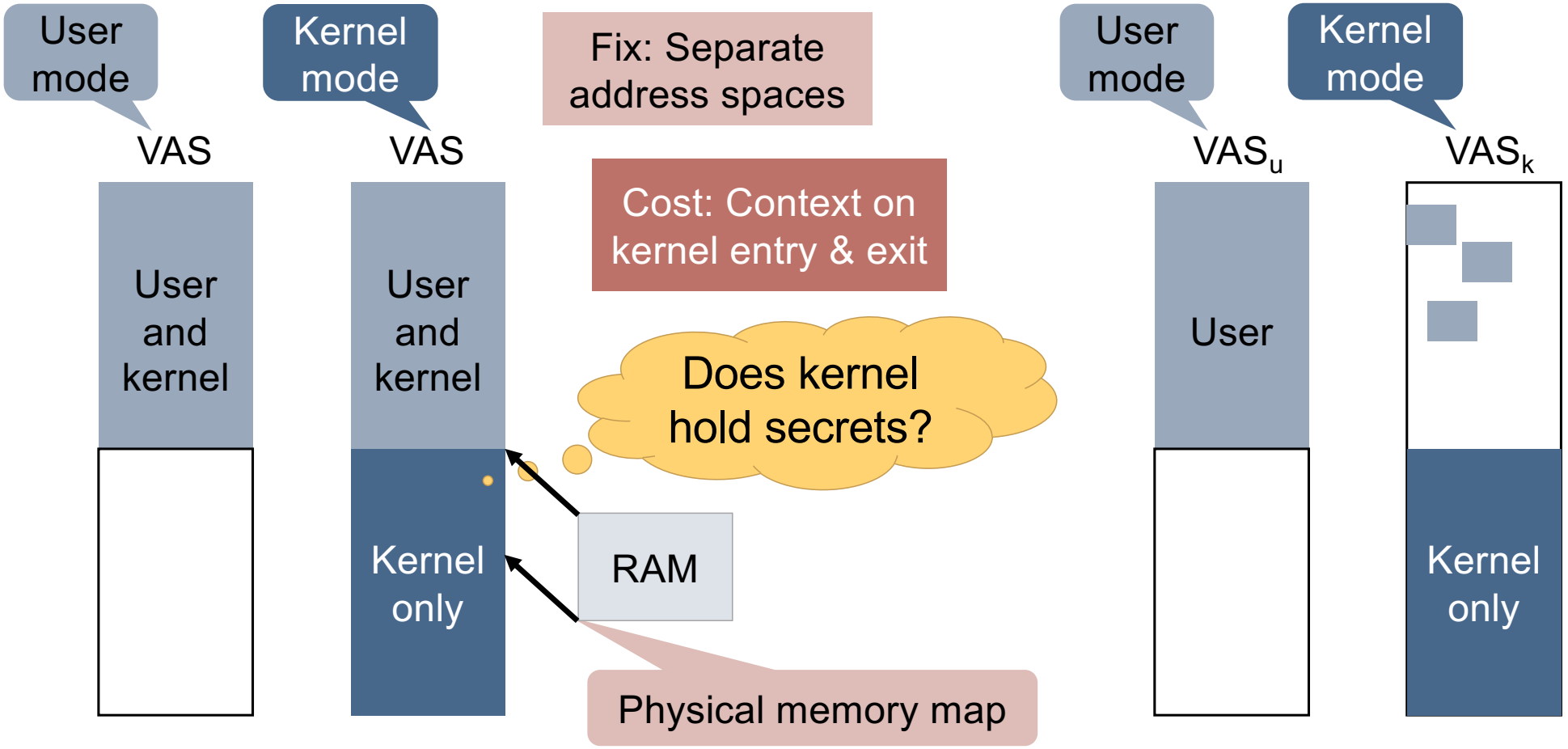
© Gernot Heiser 2019–25 – CC BY 4.0

Meltdown: Cache-Channel to Read



© Gernot Heiser 2019–25 – CC BY 4.0

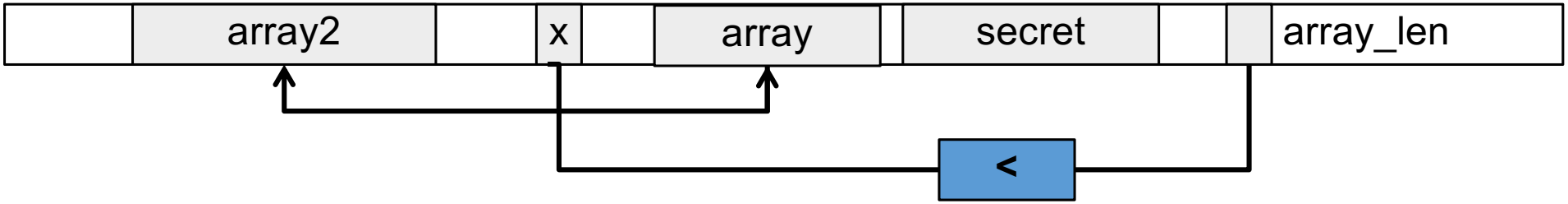
Meltdown: Full Kernel Memory Disclosure



© Gernot Heiser 2019–25 – CC BY 4.0

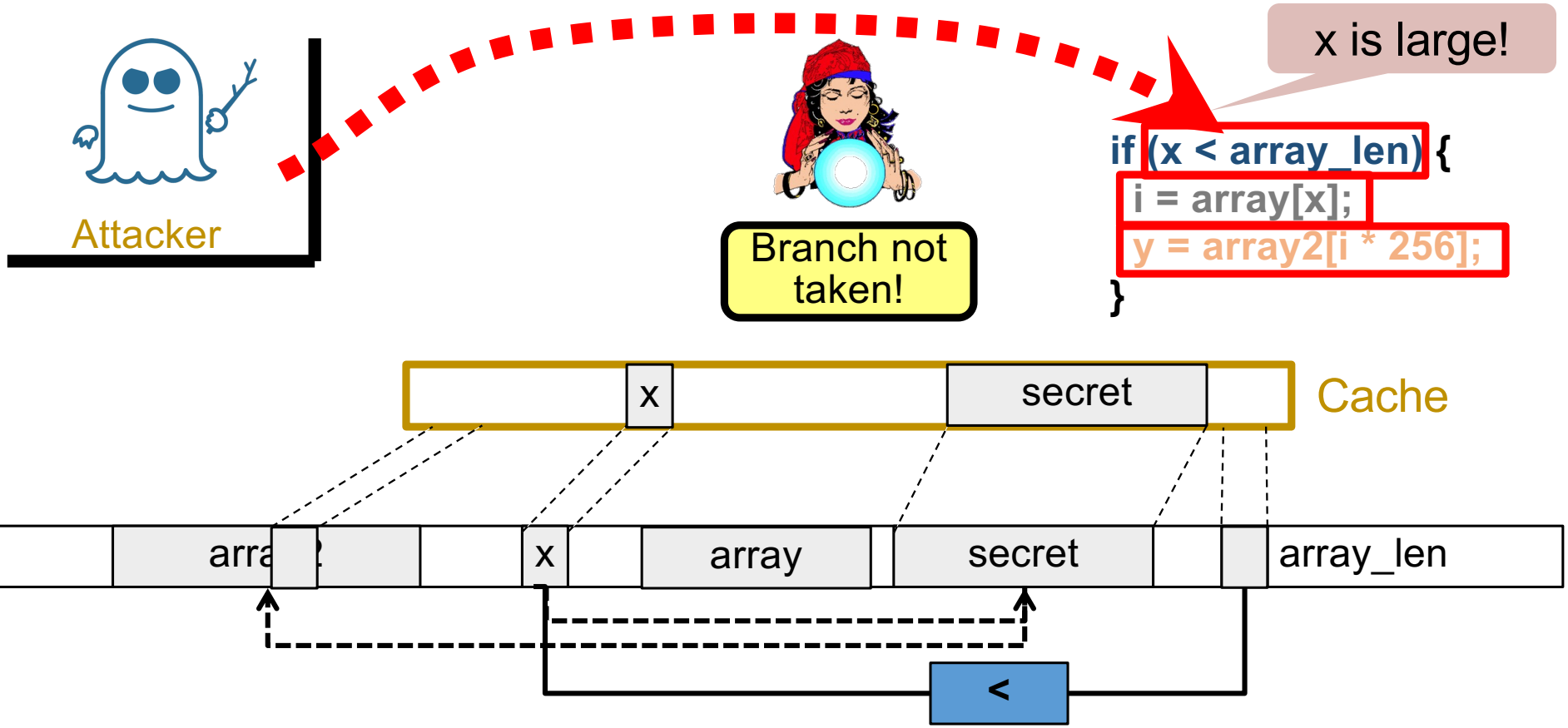
Spectre: Branch Prediction (Variant 1)





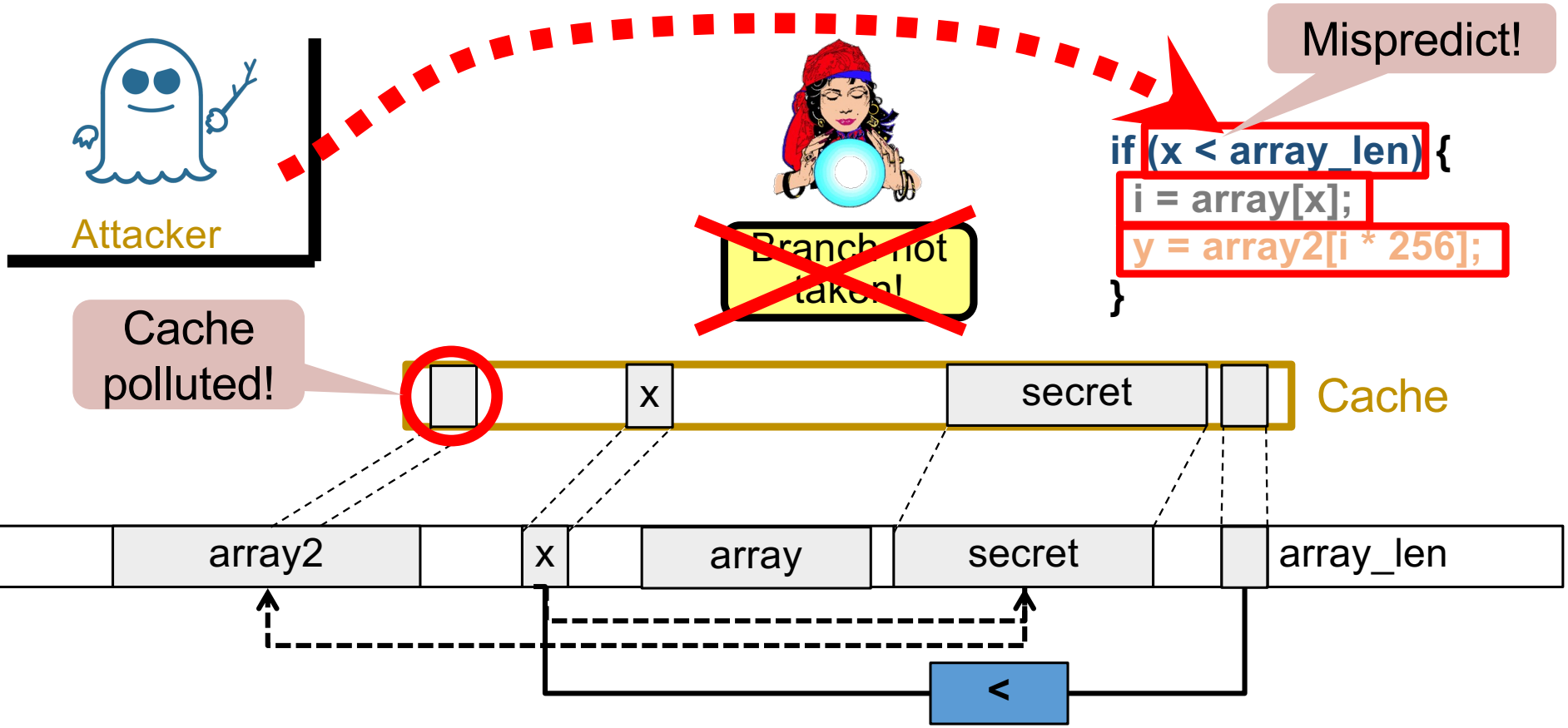
© Gernot Heiser 2019–25 – CC BY 4.0

Spectre: Branch Prediction (Variant 1)



© Gernot Heiser 2019–25 – CC BY 4.0

Spectre: Branch Prediction (Variant 1)



© Gernot Heiser 2019–25 – CC BY 4.0

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:

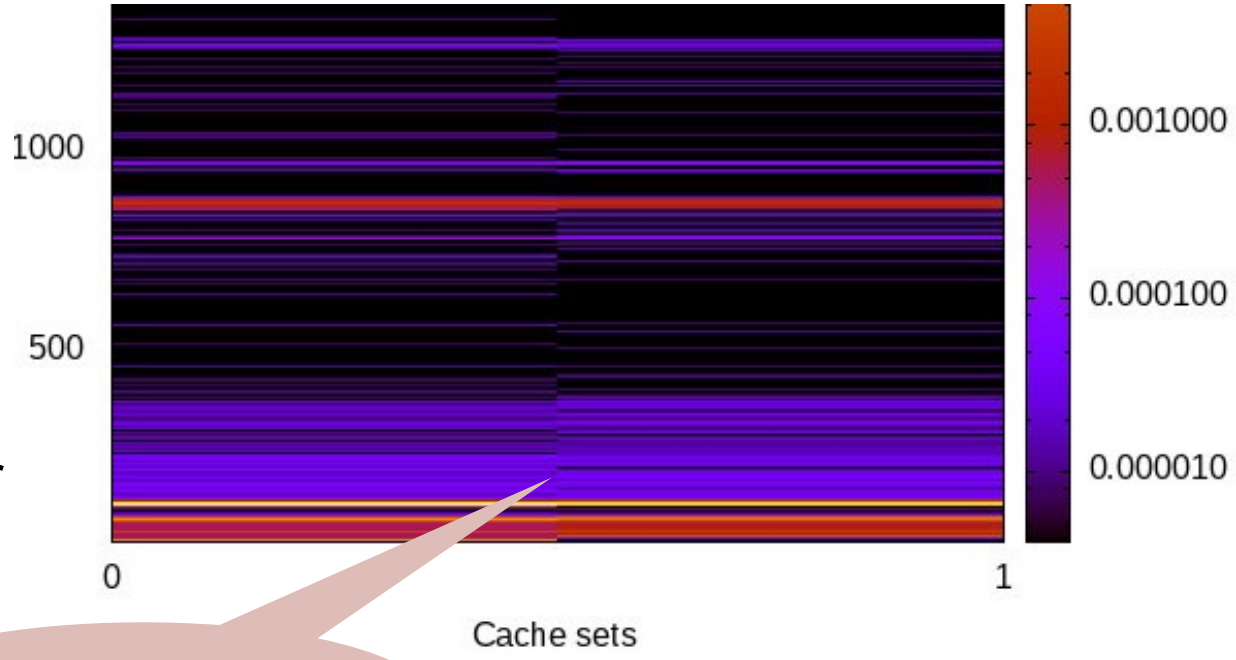
- This ISA is an insufficient contract for building secure systems
- **We need a new hardware-software contract!**

© Gernot Heiser 2019–25 – CC BY 4.0

Intel Spectre Defences

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

Intel Skylake
Branch history buffer



Small channel!

© Gernot Heiser 2019–25 – CC BY 4.0

Reminders

- Arista Advanced Operating Systems Prize for top performer in this course
- OS Hall of Fame for straight HDs in OS, AOS, OS Thesis
- Honours theses at Trustworthy Systems
  https://trustworthy.systems/students/theses
- **John Lions CS Honours Award for thesis in OS**
  https://www.scholarships.unsw.edu.au/scholarships/id/1993/7299
  Deadline: 30 November for T1/27!

© Gernot Heiser 2019–25 – CC BY 4.0