Advanced
Computer Architecture

—

Part III: Hardware Security
Microarchitectural Side-Channel Attacks

Paolo Ienne
<paolo.ienne@epfl.ch>

Why Hardware Security?

- **Software complexity**
  - OSes and hypervisors are too complex to be trusted to be bug free
  - Who can trust OSes and hypervisors?! **Secure processor architectures**
- **Microarchitectural side-channel** attacks
  - Sharing with other users gives them the ability to discover our secrets
    - Shared caches, shared processors (branch predictors, pipelines, etc.)
- Physical monitoring attacks and **physical side-channel attacks**
  - Users cannot physically protect their computing hardware
    - Hardware is often in the cloud
    - Hardware is embedded and remote (Internet-of-Things, IoT)

Outline of the Next Three Lectures

1. **Microarchitectural Side-Channel Attacks** (this lecture)
   - A set of **extremely powerful attacks** which intimately depend on the microarchitectural features of our processor (= 1st part of CS-470)
2. **Trusted Execution Environments**
   - First attempts to develop **architectural features to mitigate some of the most severe threats** to isolation and confidentiality
3. **Physical Side-Channel Attacks**
   - Possibly the most elusive class of attacks
   - **So far** of moderate concern for general purpose computing but **extremely critical for embedded applications** (e.g., smart cards) and devices not physically located with the owner/user (e.g., IoT)

Outline of This Lecture

1. Basic Definitions
2. Attacks on Memory to Compromise Integrity (Rowhammer)
3. Covert Channels and Side-Channel Attacks
4. Attacks on Timing to Break Isolation and Confidentiality (Timing Side-Channel Attacks)
5. Attacks on Memory to Break Isolation and Confidentiality (Cache Side-Channel Attacks)
6. Combined Attacks to Break Isolation and Confidentiality (Meltdown)
7. Combined Attacks to Break Isolation and Confidentiality (Spectre)

1

Basic Definitions

Threat Model

Specification of the threats that a system is protected against

- **Trusted Computing Base**: what is the set of trusted hardware and software components
- **Security properties**: what the trusted computing base is supposed to guarantee
- **Attacker assumptions**: what a potential attacker is assumed capable of
- **Potential vulnerabilities**: what an attacker might be able to gain

Classic Security Properties

- **Confidentiality**
  - → prevent the disclosure of secret information
- **Integrity**:
  - → prevent the modification of protected information
- **Availability**
  - → guarantee the availability of services and systems

We will also speak of **isolation**, that is the possibility to prevent any interaction between users and processes, often used to guarantee confidentiality and integrity

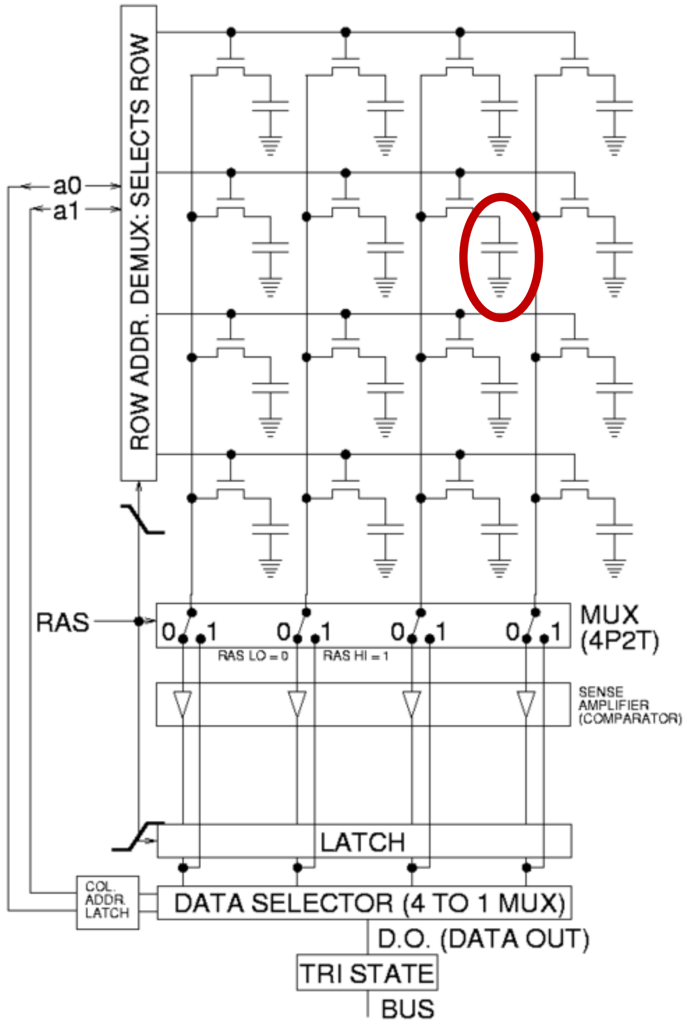
2

Attacks on Memory to Compromise Integrity

(Rowhammer)

Dynamic Random-Access Memory

- DRAMs are the densest (and thus cheapest) form of random-access semiconductor memory
- DRAMs store **information as charge in small capacitors** part of the memory cell
- First patented in 1968 by Robert Dennard, scaled amazingly over decades and was somehow an important ingredient of the progress of computing systems
- Charge **leaks off** the capacitor due to parasitic resistances → every DRAM cell needs a **periodic refresh** (e.g., every ~60 ms) lest it forgets information

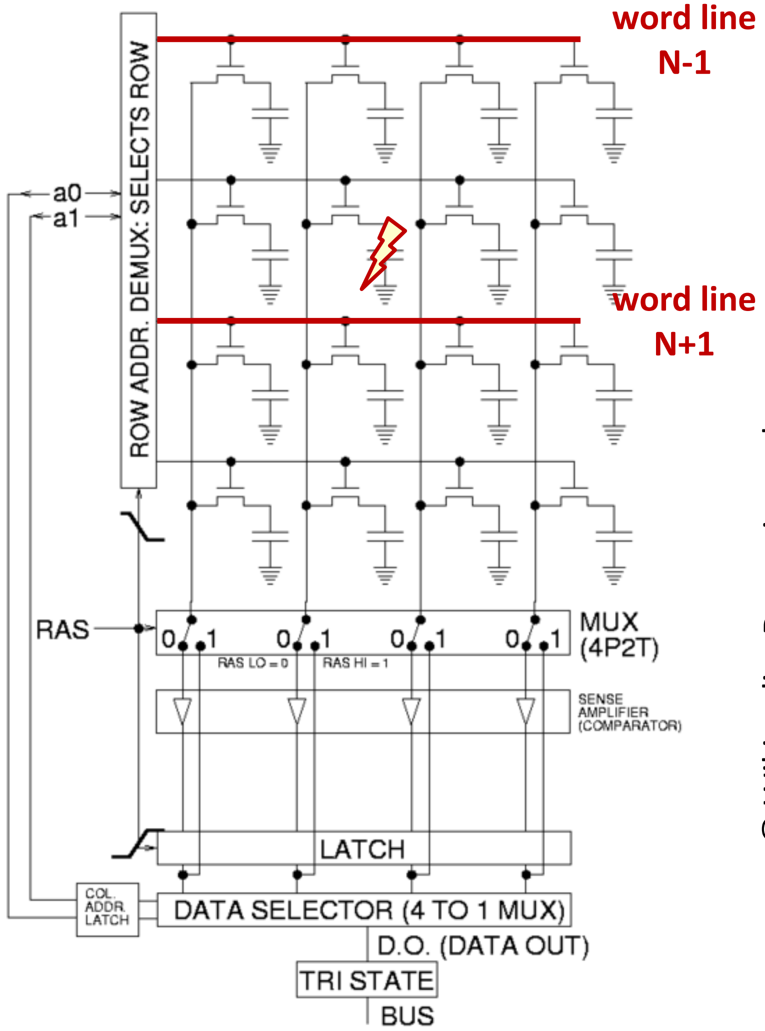


© Wikipedia, Dynamic random-access memory

Apparently Only a Reliability Issue

- To increase density (i.e., reduce cost) memory cells have become incredibly small (→ **small storage capacitance, smaller noise margin**) and word lines got extremely close to each other (→ **larger crosstalk capacitive coupling**)
- Frequent **activation of word lines** neighbouring particular cells between refreshes may **flip the cell states** due to various forms of capacitive coupling
- **Disturbance errors** have been a known design issue of DRAMs since ever, but failure in commercial DDR3 chips was demonstrated in 2014

Rowhammer



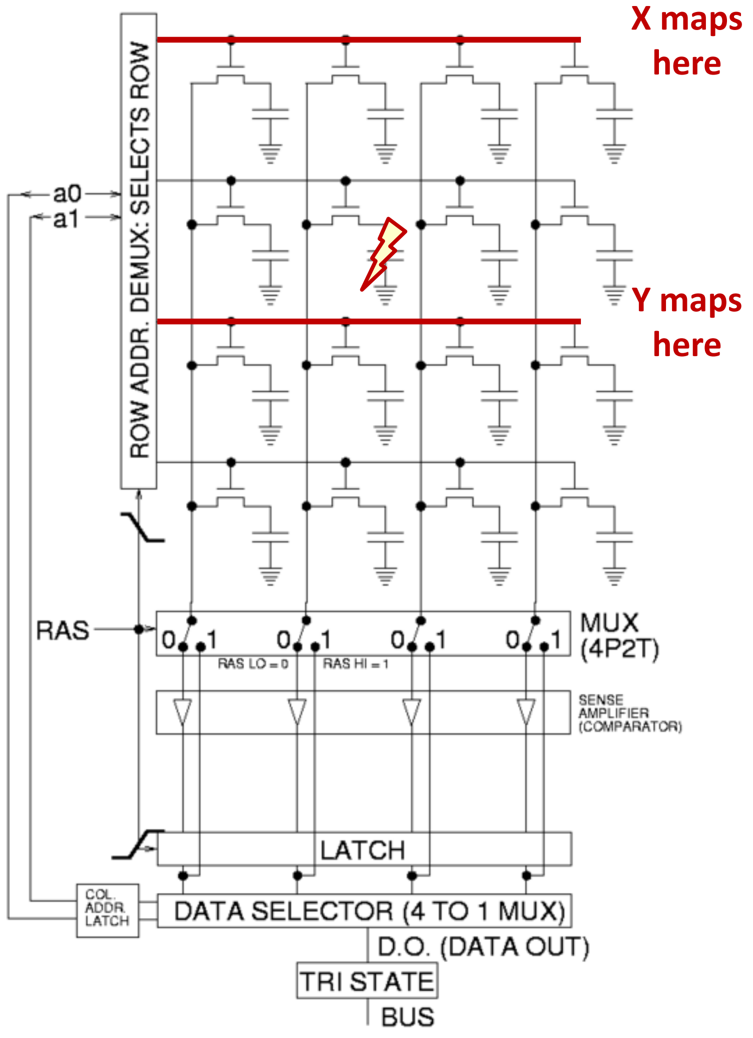
© Wikipedia, Dynamic random-access memory

A Remarkably Simple Code

Rowhammer

```
code1a:
  mov (X), %eax   // read from address X
  mov (Y), %ebx   // read from address Y
  clflush (X)     // flush cache for address X
  clflush (Y)     // flush cache for address Y
  mfence
  jmp code1a
```

- “mov” instructions activate neighbouring rows
- “clflush” unprivileged x86 instructions flush the cash from the values of X and Y (so that future accesses are misses) and “mfence” roughly waits for the flush
- Repeat as quickly as possible



© Wikipedia, Dynamic random-access memory

An Opportunity for Attacks

- Rowhammer effectively **violates memory protection** (“if I can read, I can also write”) which is a key ingredient to privilege separation across processes
- By accessing locations in neighbouring rows one could gain unrestricted memory access and privilege escalation
  - Allocate large chunks of memory, try many addresses, **learn weak cells**
  - Release memory to the OS
  - Repeatedly map a file with RW permissions to **fill memory with page table entries (PTEs)**
  - Use Rowhammer to **flip (semirandomly) a bit in one of these PTEs**; it will now point to the wrong physical page
  - Chances are that this physical page contains PTEs too, so now **accessing that particular mapping of the file (RW) actually modifies the PTEs**, not the file
  - Attacker can arbitrarily change PTEs and **memory protection is gone**
- Not that simple in practice, tons of difficulties, but people managed to make it work!

Mitigations

- Error Correcting Codes (ECC) may fail to detect multiple flips
- Shortening the refresh intervals mitigates but does not solve problems; implemented in firmware by some vendors
- Hard or impossible to avoid altogether without **changes in the DRAMs**
- Increase electrical noise margins (costly!)
- Count inside the DRAM the number of row activations within a time window, **identify potential victims, and refresh**
  - Introduced in DDR4 products but not in the JEDEC standard

An Aside on DRAMs: Data Remanence

- A completely different problem with storing data on capacitors: cells may leak information quickly in the worst case but very many do not leak much in typical conditions
- Lowering significantly the device temperature (e.g., use spray refrigerants) makes **most cells retain charge for long time** (seconds to minutes)
- **Coldboot** attacks:
  - Cool a working DRAM device
  - Switch off
  - Move the device to another computer or reboot a malicious OS
  - Read content (passwords, secret keys, etc.)

3

Covert Channels and Side-Channel Attacks

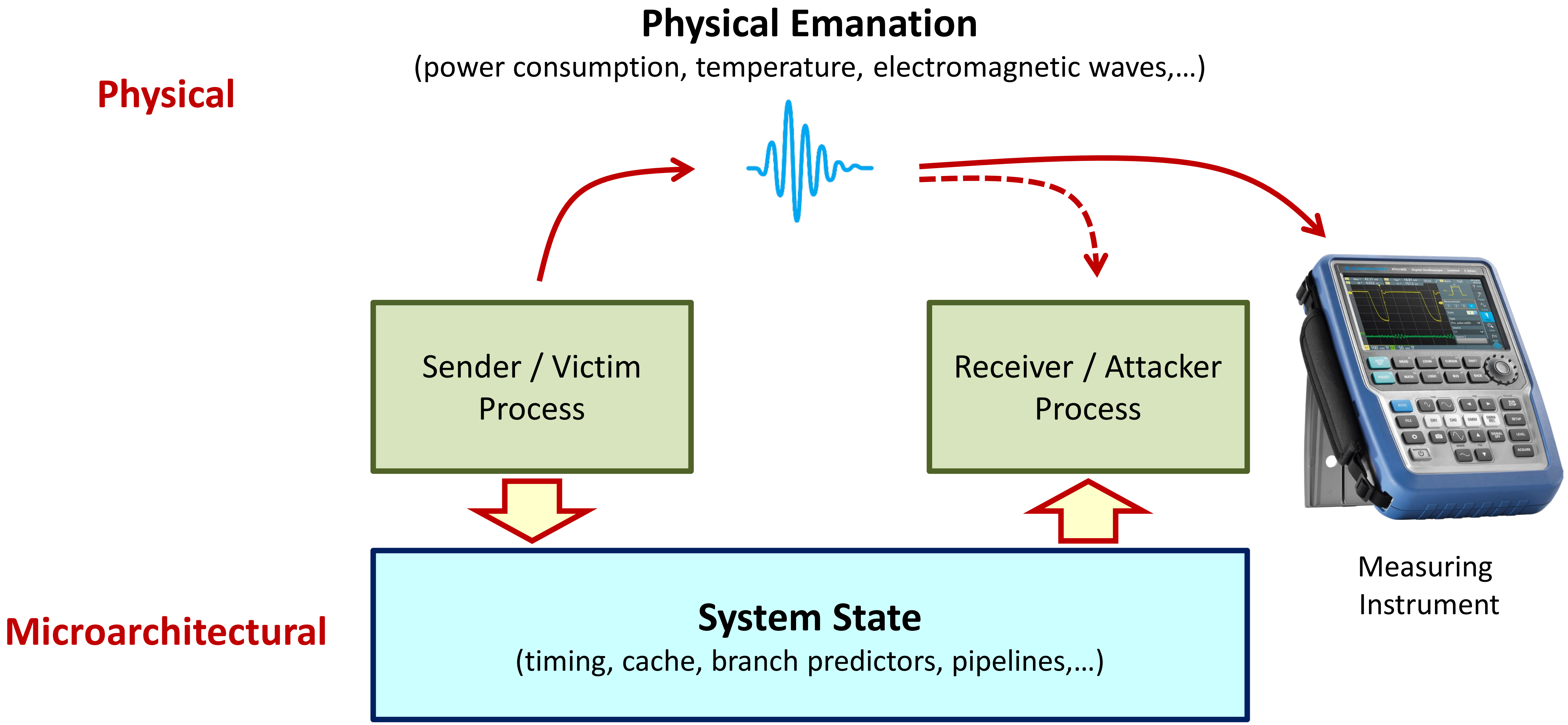
Covert Channels

- "A covert channel is an **intentional communication** between a sender and a receiver **via a medium not designed to be a communication channel**" (Szefer, 2019)
- If we isolate a critical process inside a virtual machine (or an *enclave*, see later), a covert channel may allow a rogue programme inside of the isolated process (a **Trojan horse**) to leak a secret to some malicious receiver without anyone to notice (no conventional communication channel visible)

Side Channels Attacks

- Attacks where the sender is the unsuspecting **victim** of the attack, who is **unknowingly transmitting information through a covert channel**, and the receiver is the attacker
- Sending (or leaking) information is a **side effect of the normal operation of the victim**, either because of the hardware implementation of the system or because of the software implementation of the victim—or both

Covert- and Side-Channels


Physical Emanation
(power consumption, temperature, electromagnetic waves,…)
Physical
Sender / Victim Process
Receiver / Attacker Process
Measuring Instrument
Microarchitectural
System State
(timing, cache, branch predictors, pipelines,…)

Covert- and Side-Channels

- Microarchitectural
  - Based on the **existence of microarchitectural state**, that is state not (normally) visible to the programmer—because architectural state is known and thus, apart from bugs, inherently protected!
  - Based on the **sharing of hardware components** featuring such microarchitectural state
  - Physical replication and isolation may solve the problem
- Physical
  - Based on the **physical nature** of the computing system
  - Potentially more difficult to fight, but also harder to exploit

4

Attacks on Timing to Break Isolation and Confidentiality

(Timing Side-Channel Attacks)

Execution Time Reveals Something on Data

Compare

```
bool insecureStringCompare(const void *a, const void *b, size_t length) {
  const char *ca = a, *cb = b;
  for (size_t i = 0; i < length; i++)
    if (ca[i] != cb[i])
      return false;
  return true;
}
```

Return as soon as a difference is found

with

```
bool constantTimeStringCompare(const void *a, const void *b, size_t length) {
  const char *ca = a, *cb = b;
  bool result = true;
  for (size_t i = 0; i < length; i++)
    result &= ca[i] == cb[i];
  return result;
}
```

Record the difference and return always after checking the entire string

Blinding through Constant Time

- Not always easy:
  - May need to **fight compiler optimizations**
    - Time is typically made constant by **provably unnecessary computation**
  - Variability may arise from **microarchitectural phenomena**
    - Data-dependent instruction latency
    - Virtual memory and caches
    - Instruction scheduling
    - …
- In a sense, most if not all of the attacks discussed in the following slides are ultimately **timing attacks** of specific nature

5

Attacks on Memory to Break Isolation and Confidentiality

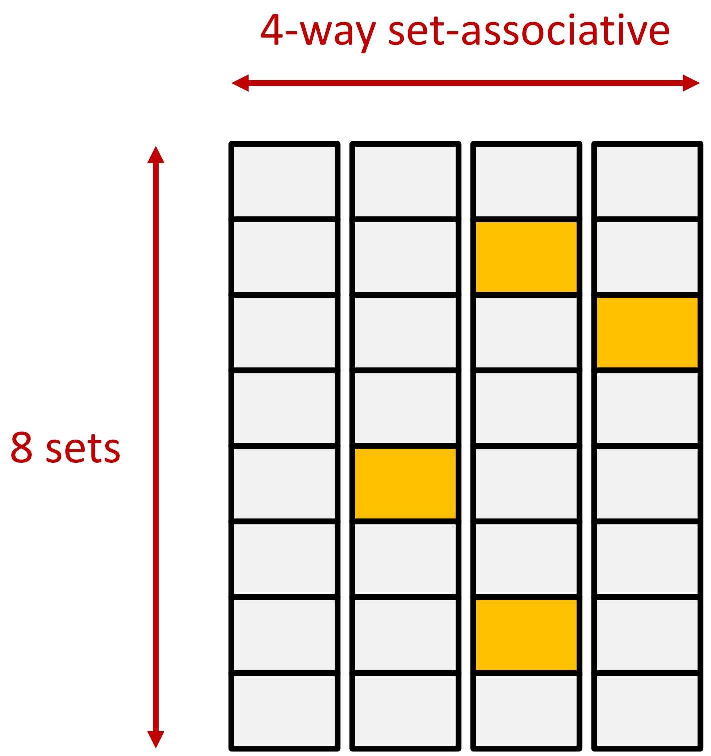
(Cache Side-Channel Attacks)

Cache Side-Channel Attacks

- Oldest and perhaps most powerful example of **microarchitectural side-channel** (cache shared but not architecturally visible)
  - Evoked since 1992 but first fully demonstrated in 2005
- Attacker can **differentiate hits and misses** using some high-resolution **timing** measurement (e.g., processor cycles)
- Victim memory accesses (= **where** the victim loads or stores) reveal secrets
  - E.g., D$ accesses to an AES sbox() depend on the secret key
  - E.g., I$ accesses to different RSA functions depend on the secret key
- Attacker can **run victim code**
  - E.g., write to a file into an encrypted volume, send packets through a VPN interface

Evict+Time

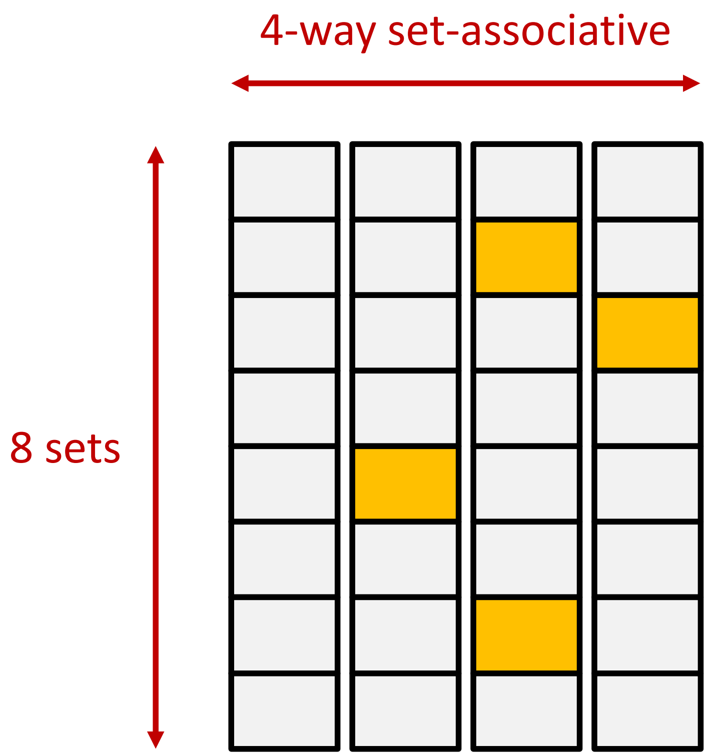
Does the victim access location X (in set Y)?



1. Run the victim

Evict+Time

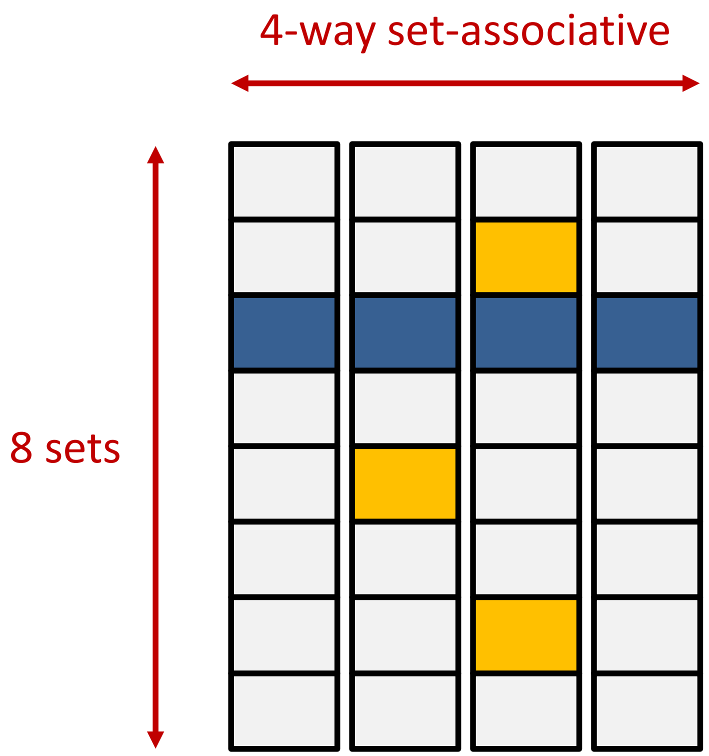
Does the victim access location X (in set Y)?



1. Run the victim
2. Run the victim and time it
   - fast because data are in cache

Evict+Time

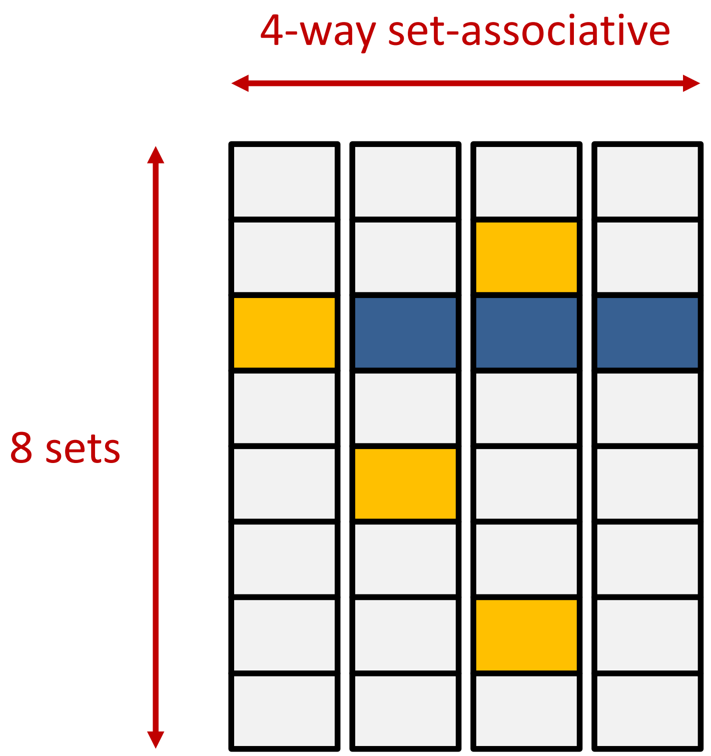
Does the victim access location X (in set Y)?



1. Run the victim
2. Run the victim and time it
   - fast because data are in cache
3. Evict content from set Y ***(evict)***
   - by replacing with attacker content
   - making sure to pollute all ways

Evict+Time

Does the victim access location X (in set Y)?



1. Run the victim
2. Run the victim and time it
   - fast because data are in cache
3. Evict content from set Y ***(evict)***
   - by replacing with attacker content
   - making sure to pollute all ways
4. Run the victim and time it ***(time)***
   - **if step 4 takes longer than 2, the victim accessed something in set Y**

Evict+Time

- Problem
  - Relies on measuring the **precise execution time of the victim**
  - Repeats the **same execution**, so no variability in the executed code
  - Yet, timing may be affected by **environmental issues**
    - System call
    - Branch prediction
    - Instruction scheduling
  - It is small noise but may be comparable to the quantity being measured
  - **Repeat many times**

Evict+Time

- Results affected by noise, but noise can usually be controlled by **repeating the experiment** a sufficient number of times

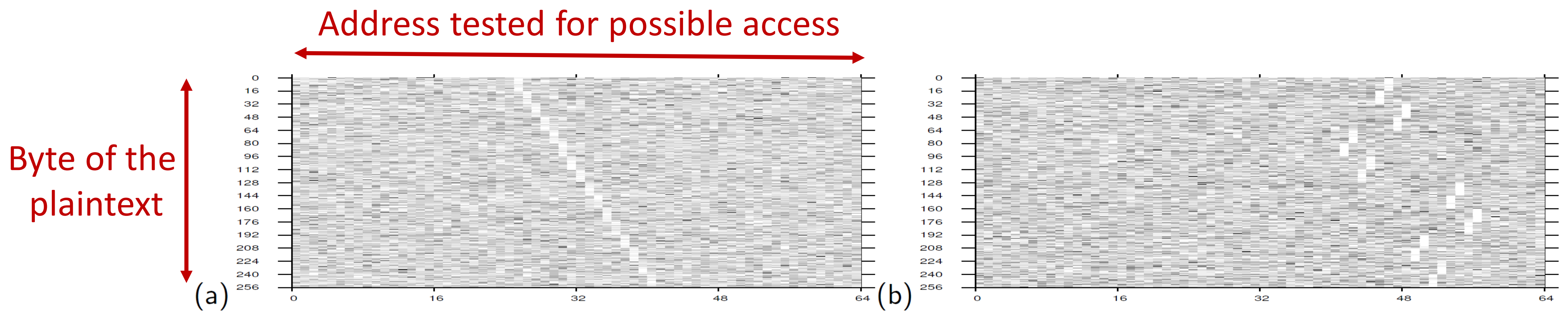
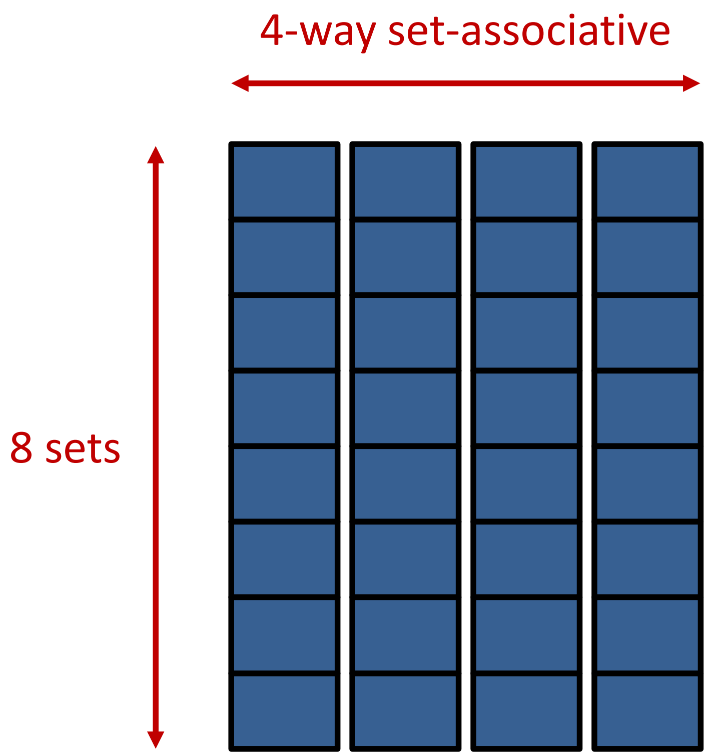


Fig. 4. Timings (lighter is slower) in Evict+Time measurements on a 2GHz Athlon 64, after 10,000 samples, attacking a procedure that executes an encryption using OpenSSL 0.9.8. The horizontal axis is the evicted cache set (i.e., $\langle y \rangle$ plus an offset due to the table's location) and the vertical axis is p_0 (left) or p_5 (right). The patterns of bright areas reveal high nibble values of 0 and 5 for the corresponding key byte values.

Source: Osvik et al., CT-RSA'06

Prime+Probe

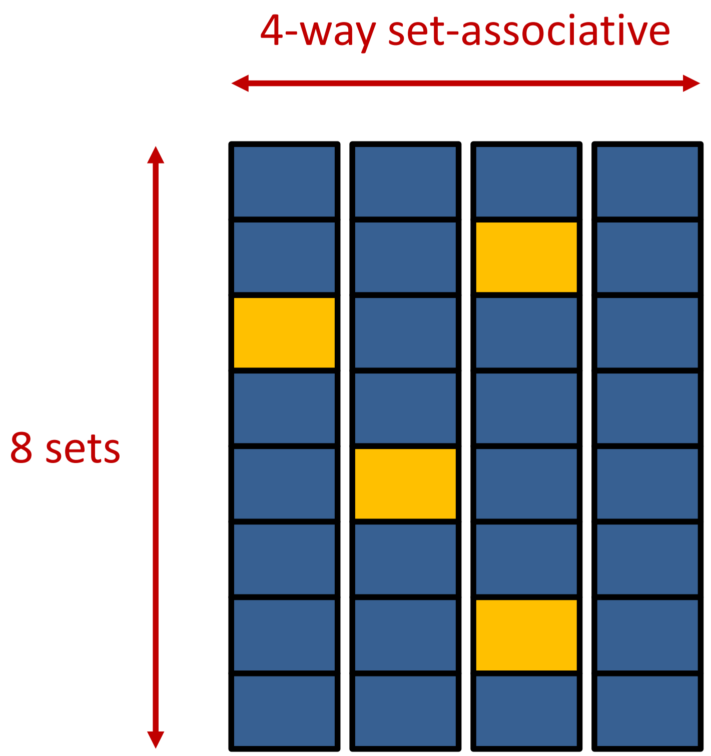
What location (set) does the victim access?



1. Fill all sets with attacker content ***(prime)***
2. Read all pieces of data for all sets and time each set
   - fast because data are in cache

Prime+Probe

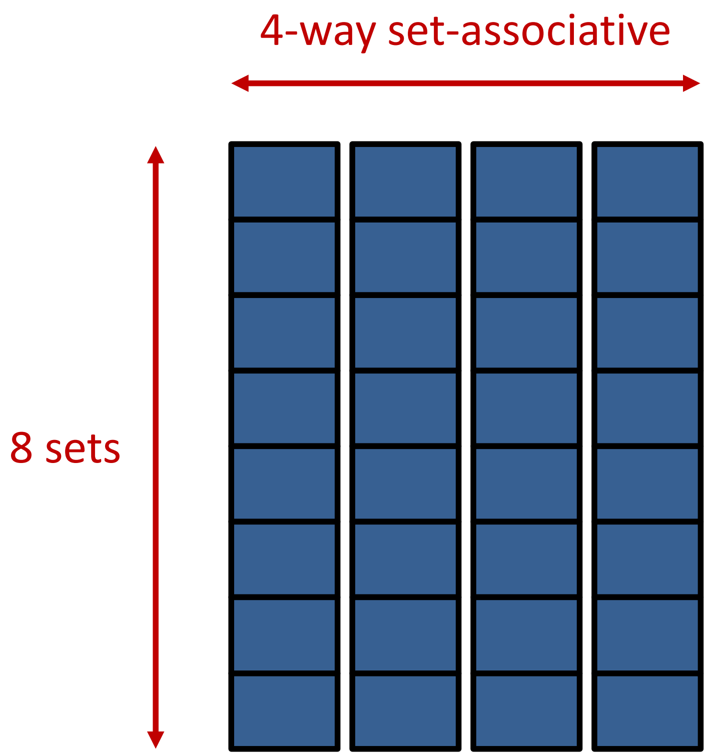
What location (set) does the victim access?



1. Fill all sets with attacker content ***(prime)***
2. Read all pieces of data for all sets and time each set
   - fast because data are in cache
3. Run the victim

Prime+Probe

What location (set) does the victim access?



1. Fill all sets with attacker content ***(prime)***
2. Read all pieces of data for all sets and time each set
   - fast because data are in cache
3. Run the victim
4. Read all pieces of data for all sets and time each set ***(probe)***
   - **if step 4 takes longer than 2 for set Y, the victim accessed something in set Y**

Prime+Probe

- **Key advantage**: now the attacker times their own code and not the victim's code, arguably allowing **better control of measurement noise**

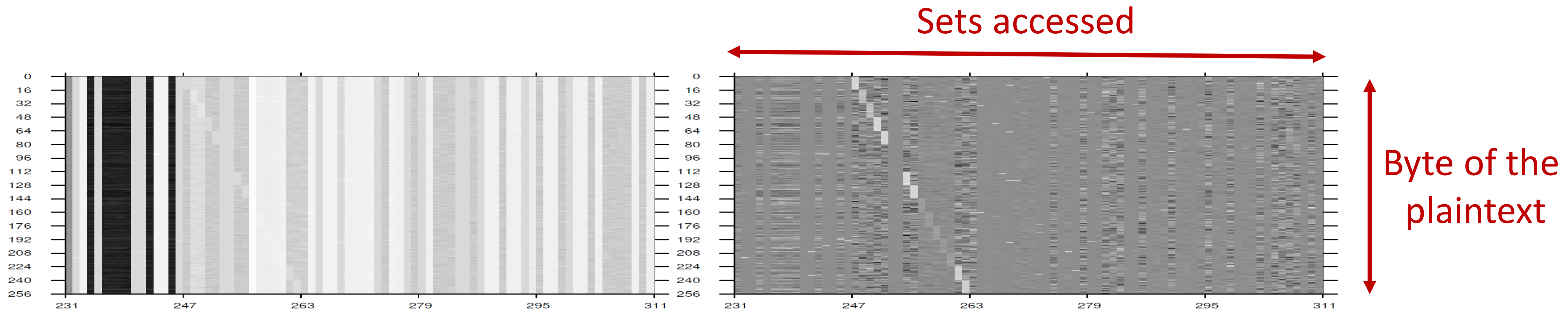


Fig. 5. Prime+Probe attack using 30,000 encryption calls on a 2GHz Athlon 64, attacking Linux 2.6.11 `dm-crypt`. The horizontal axis is the evicted cache set (i.e., $\langle y \rangle$ plus an offset due to the table's location) and the vertical axis is p_0. Left: raw timings (lighter is slower). Right: after subtraction of the average timing of the cache set. The bright diagonal reveals the high nibble of $p_0 = 0x00$.

Source: Osvik et al., CT-RSA'06

Candidate Scores

- Many attacks to **cryptographic algorithms** involve trying multiple plaintexts and/or key hypotheses and distinguishing between most likely and least likely over many attempts

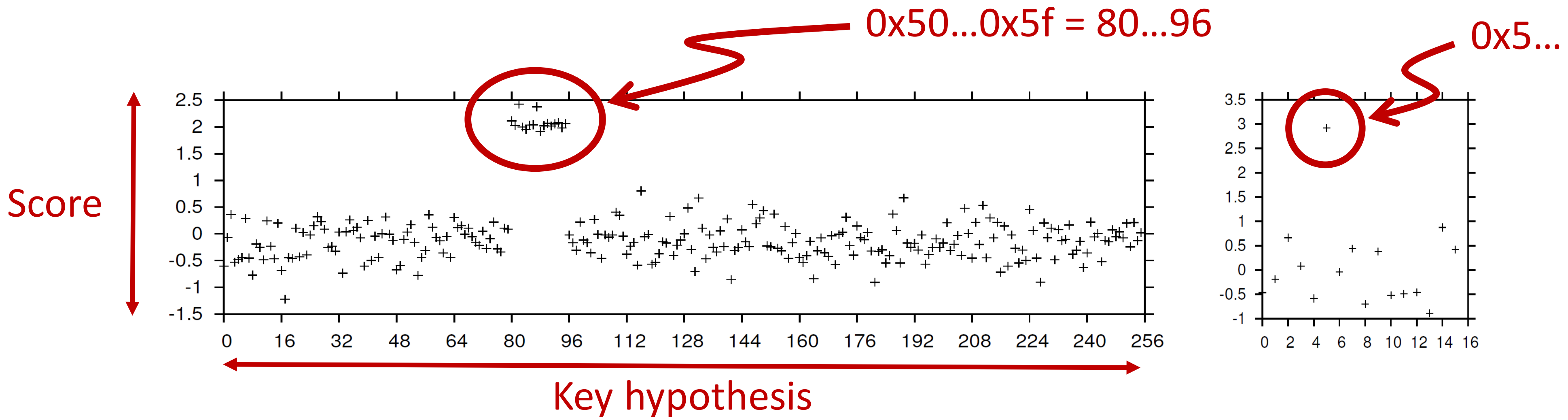
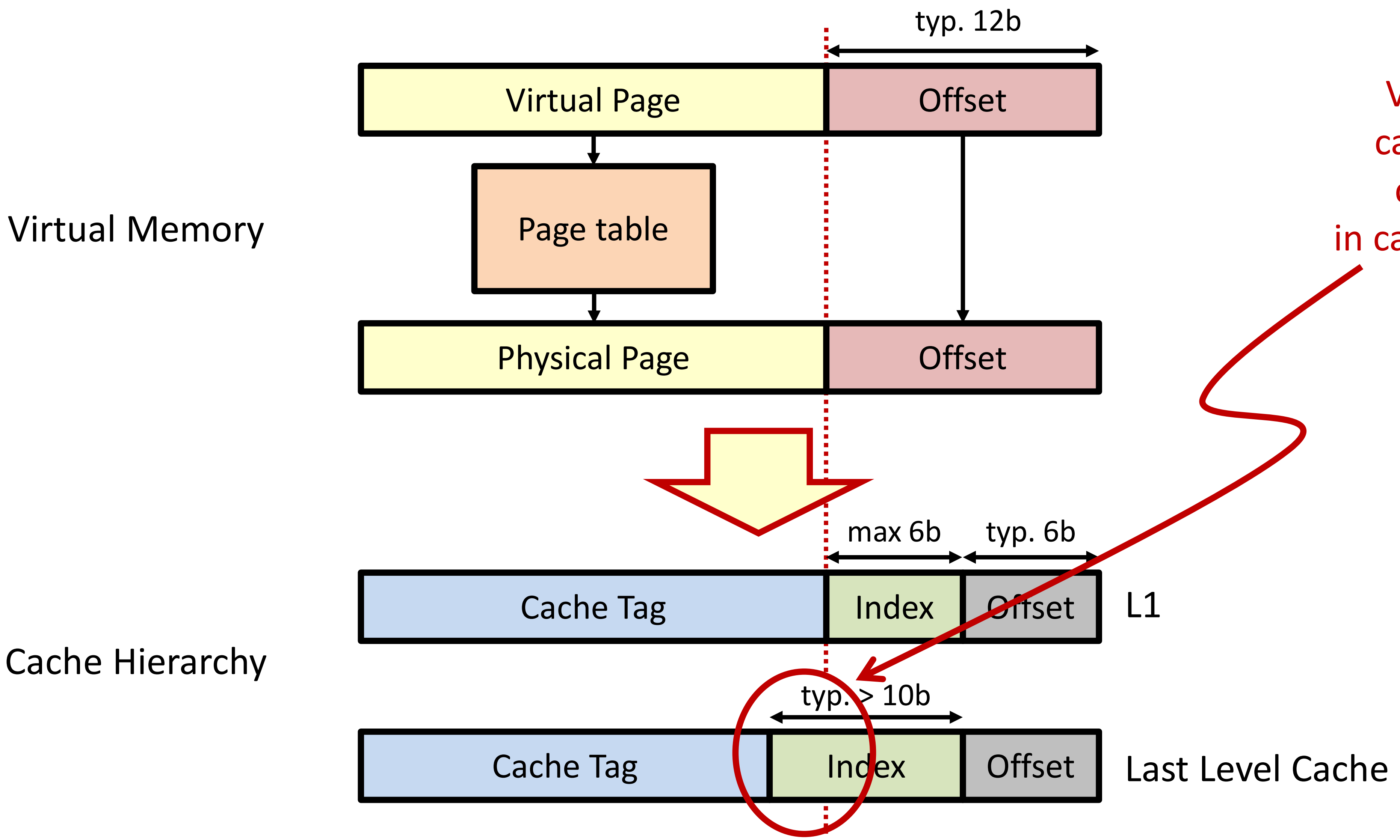


Fig. 2. Candidate scores for a synchronous attack using Prime+Probe measurements, analyzing a `dm-crypt` encrypted filesystem on Linux 2.6.11 running on an Athlon 64, after analysis of 30,000 (left) or 800 (right) triggered encryptions. The horizontal axis is $\tilde{k_5} = p_5 \oplus y$ (left) or $\langle \tilde{k_5} \rangle$ (right) and the vertical axis is the average measurement score over the samples fulfilling $y = p_5 \oplus \tilde{k_5}$ (in units of clock cycles). The high nibble of $k_5 = \text{0x50}$ is easily gleaned.

Source: Osvik et al., CT-RSA'06

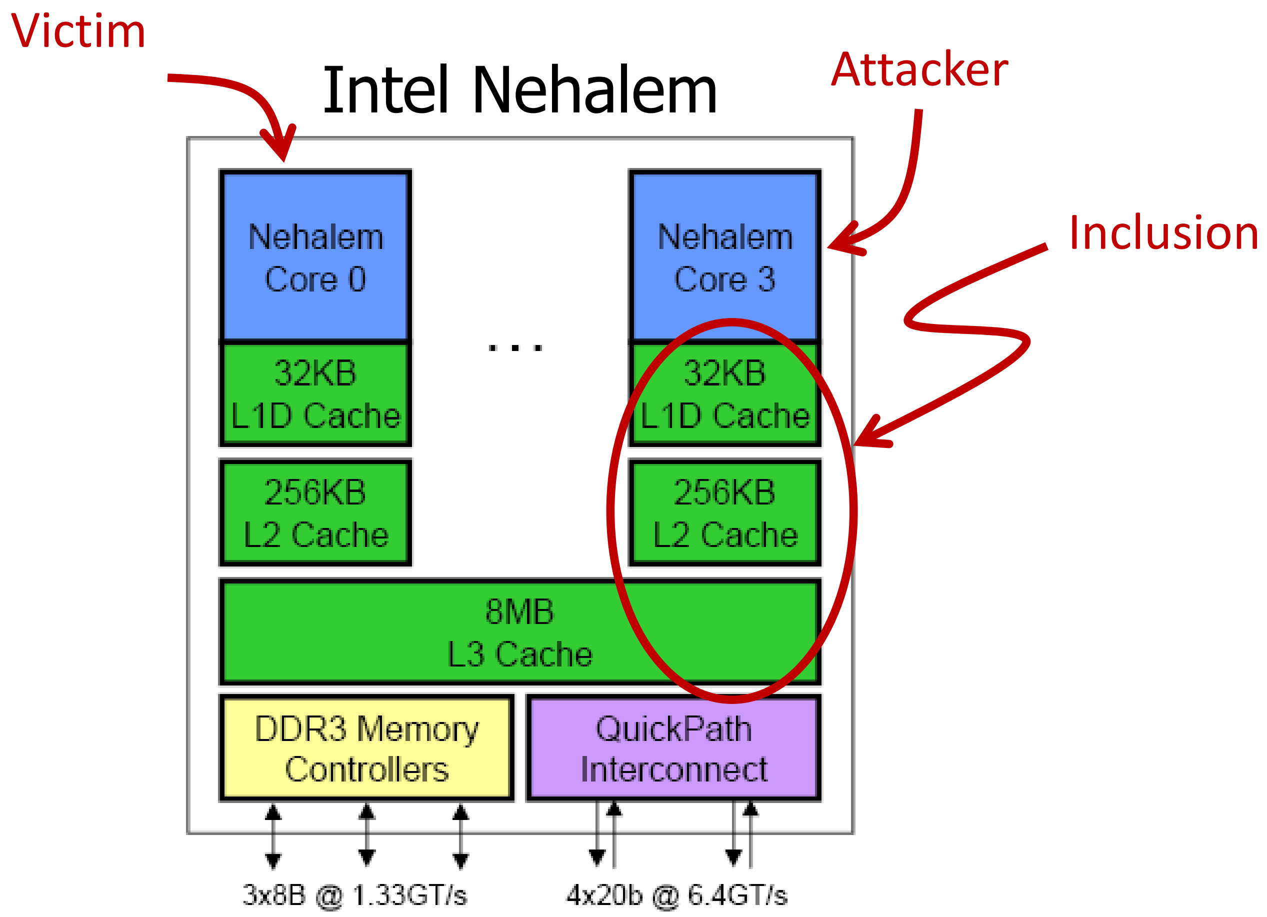
What about Large Caches?



Virtual addresses cannot control the cache set to use in caches larger than L1

Why an Attacker Cares about LLC?

- The attacker and the victim may be on **different cores**
- Also more index bits (= more sets) lead to **better resolution** (e.g., 1/2048 vs. 1/64)

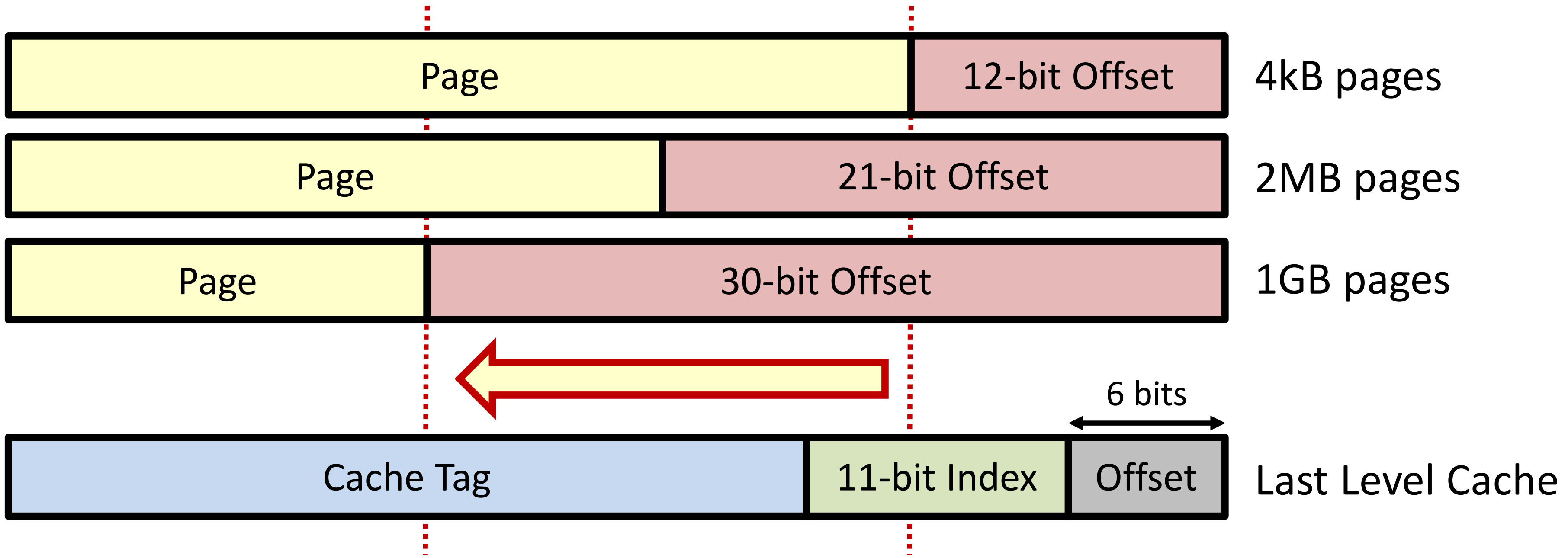


- **Inclusion property**
  - Whatever is in L1D is certainly in L2; whatever is in L2 is certainly in L3, etc.
  - Useful to maintain coherence
- In particular
  - If the attacker evicts something in L1D, it is evicted also in L2 and L3

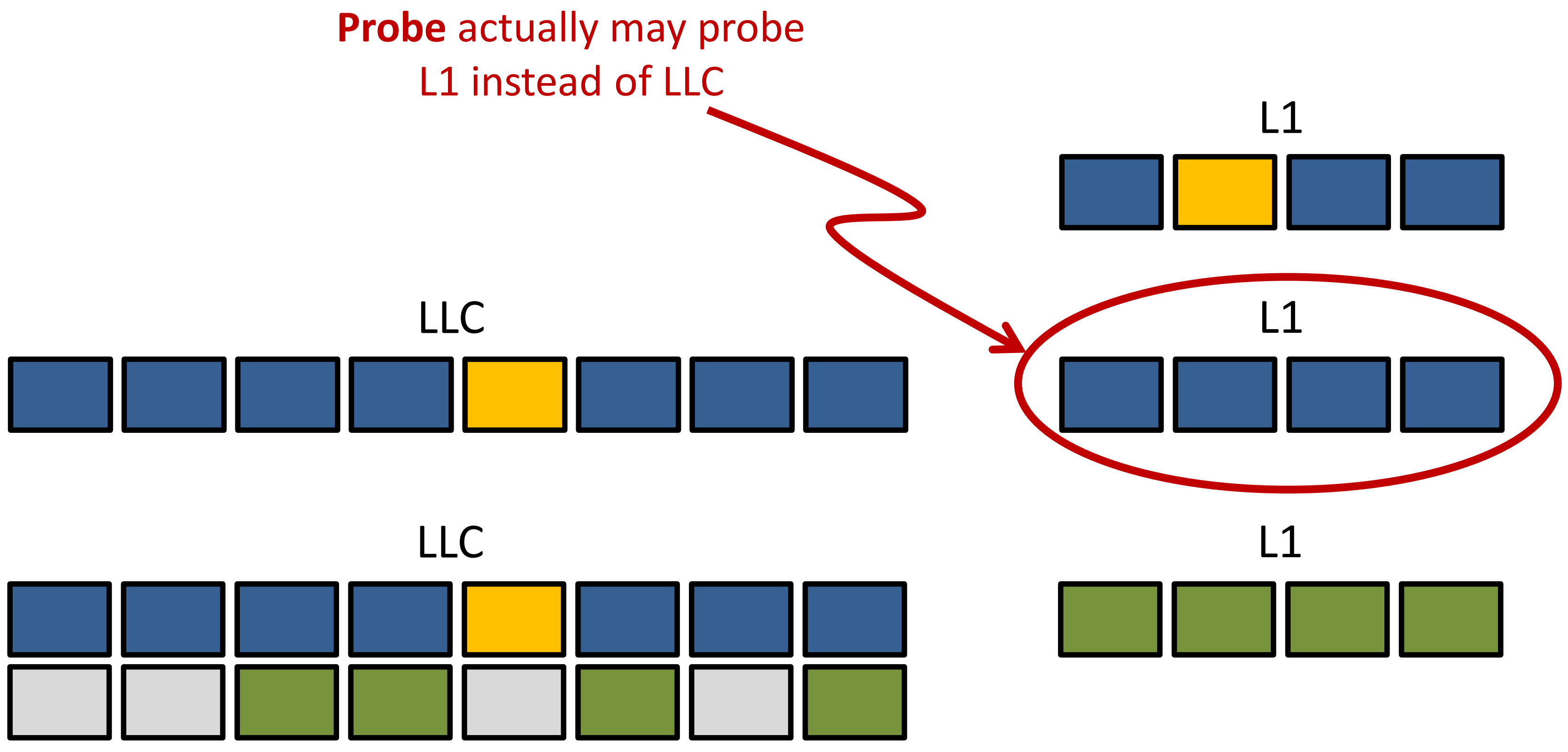
Source: http://realworldtech.com/, © RWT 2008

Huge Virtual Pages

- Takes advantage of the hardware support for large pages
  - Large pages → More memory waste but **more efficient management** (e.g., less TLB misses) in applications with large data footprint
- Attacker needs **administrator rights** to set large pages
  - Not a problem, because they have them in their guest OS



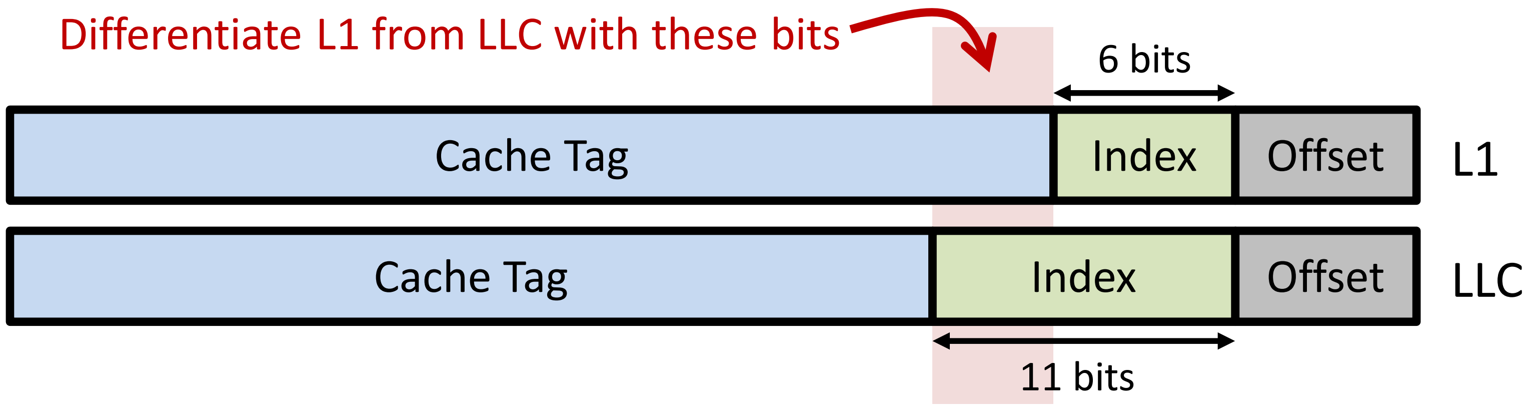
Prime+Probe and Multiple Levels



1. Fill set ***(prime)***
2. Run the victim
3. Find misses ***(probe)***

1. Fill set ***(prime)***
2. Run the victim
3. Find misses ***(probe)***

1. Fill set in LLC ***(prime)***
2. **Fill set (= evict) in L1 *(reprime)***
   - **Do not evict from LLC!**
3. Run the victim
4. Find misses ***(probe)***

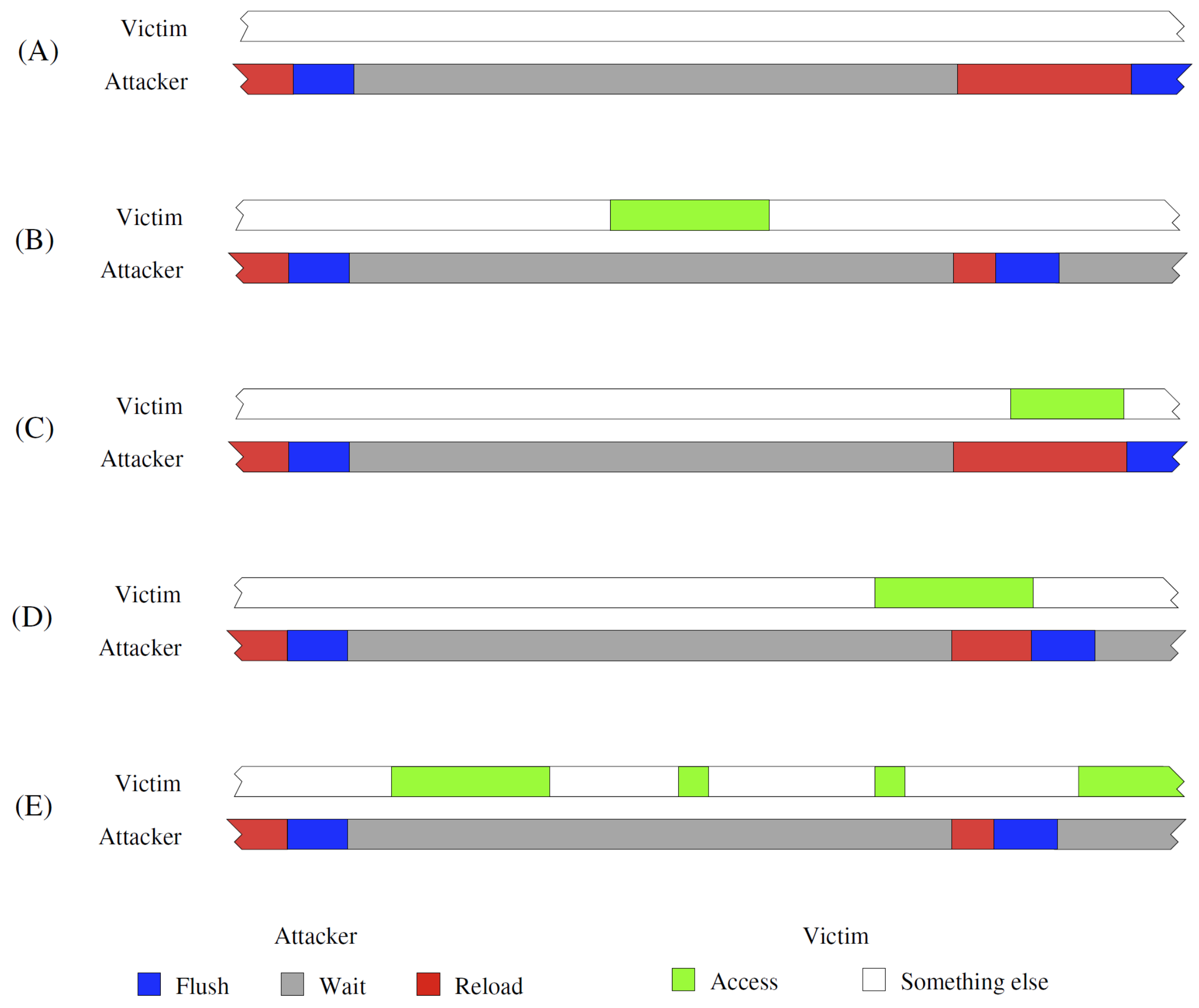


Reprime with elements in the same set in L1 but in a different set in LLC

Asynchronous Attacker and Victim?

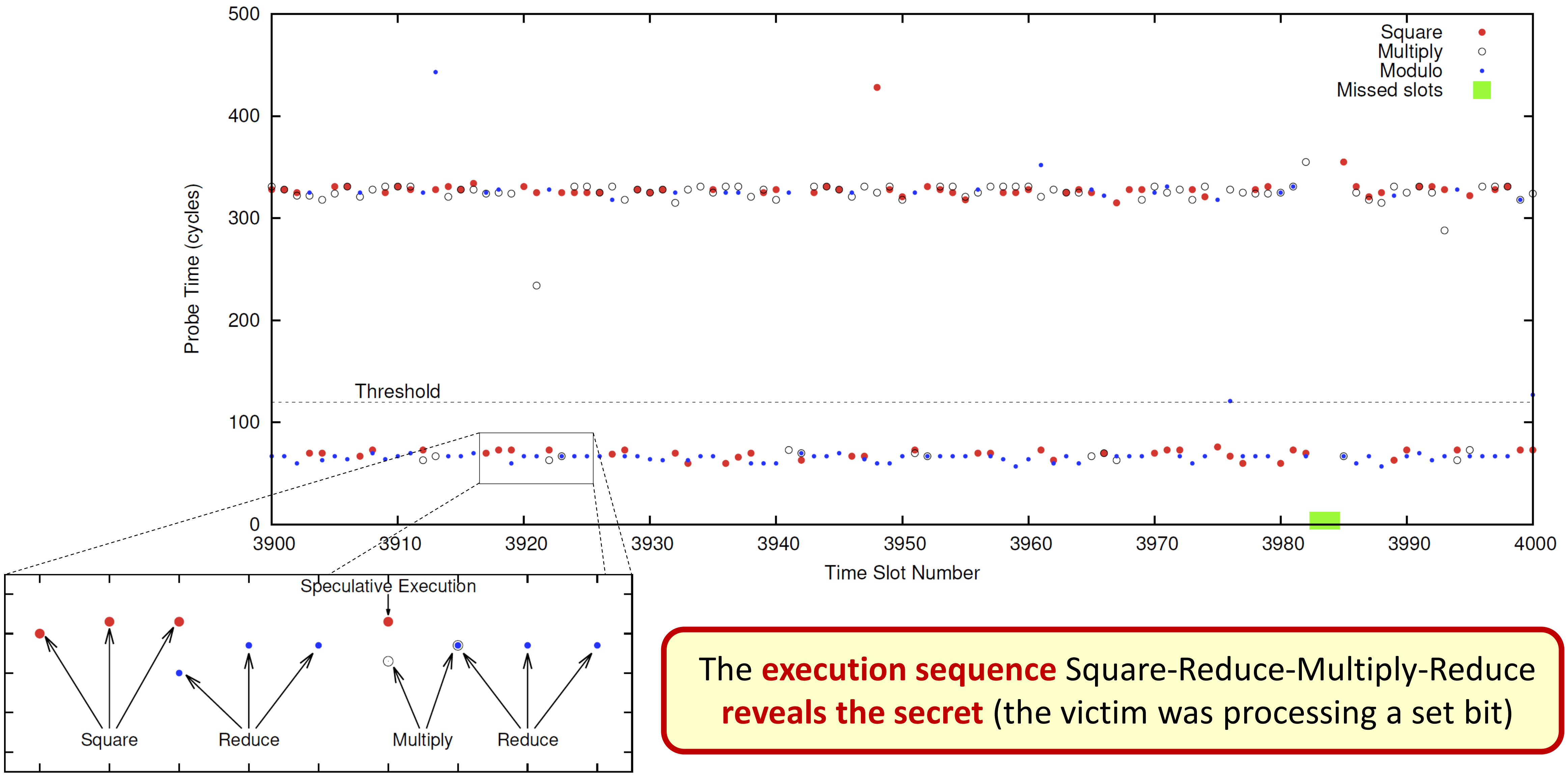
The attacker runs in a **virtual machine** and the victim in **another one**, so no synchronization possible

- The example is here a **Flush+Reload** attack, similar to Prime+Probe but uses the *clflush* instruction of x86 to evict a specific cache line and **depends on virtual machine page deduplication** (if two users load the same executable or libraries, only one is kept in memory)
  - Attacker and victim use different virtual addresses in different virtual machines, but the **physical address is the same**
- **Tracks accesses to code** to infer the internal state of the victim



Source: Yarom and Falkner, USENIX Security '14

Asynchronous Attacker and Victim?



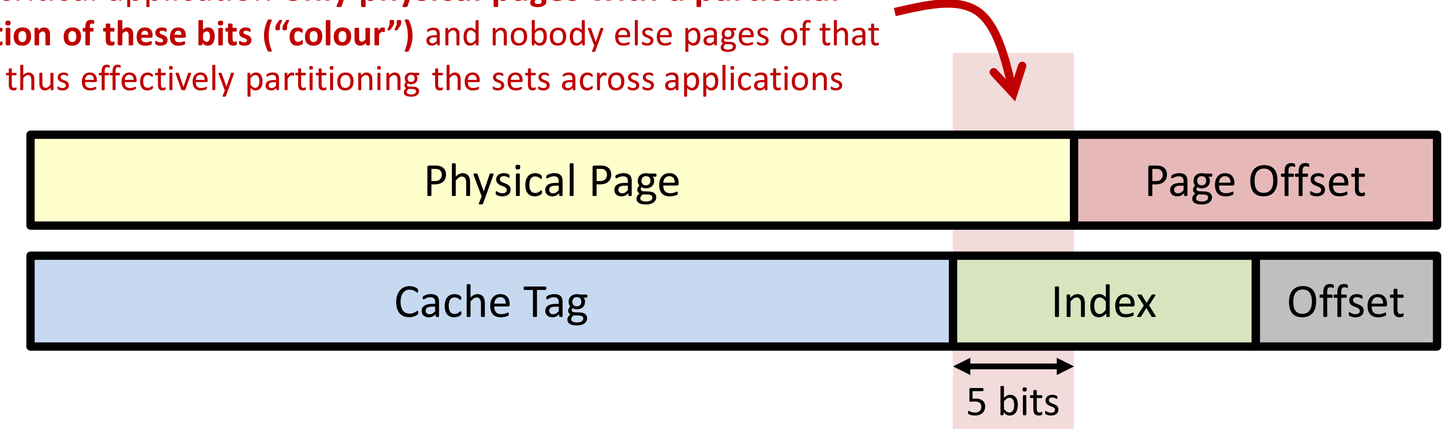
The **execution sequence** Square-Reduce-Multiply-Reduce **reveals the secret** (the victim was processing a set bit)

Source: Yarom and Falkner, USENIX Security '14

Possible Mitigations

- Huge amount of research
  - Hardware solutions
    - Various forms of partitioning and randomization
    - Practically **none is truly effective and efficient**, and thus viable for general use
  - Generic software solutions
    - **Clumsy, difficult to generalize, costly, not always effective**
    - Cache colouring → OS uses physical page allocation to reserve sets for some processes
  - **Application-specific solutions**
    - Possibly the safest option

Give a critical application **only physical pages with a particular combination of these bits ("colour")** and nobody else pages of that colour, thus effectively partitioning the sets across applications



6

Combined Attacks to Break Isolation and Confidentiality

(Meltdown)

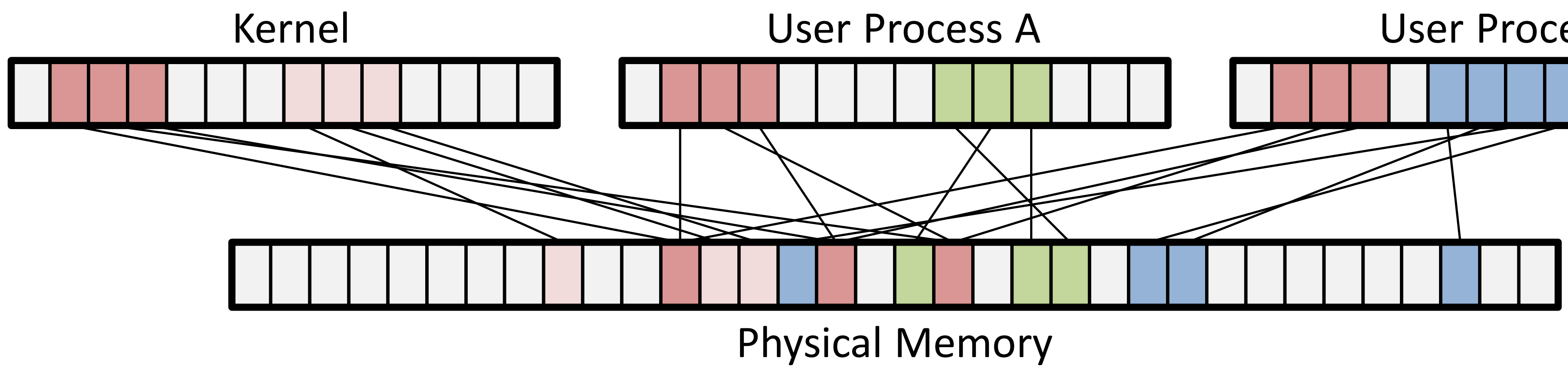
Meltdown

- Catastrophic attack making it possible to **read all memory of a process** (including **protected kernel data**)
- By product of the way some microarchitectural features are implemented (e.g., AMD x86 implementations are **per chance** resistant to Meltdown)
- Exploits **race condition** between **memory access** and **protection checks**
  - Ultimately exploits the microarchitectural nature of caches (something is left in the cache upon exception because **the cache is not part of the architectural state**)

The attacker executes a **forbidden access** and **speculatively uses the result** to obtain **nonarchitectural side-effects** that reveal the secrets **before the forbidden access is squashed**

Mapping Kernel Pages in User Space

- Most OSes map physical kernel memory pages into every user's virtual memory space



- Minimizes the cost of some exceptions (e.g., fast interrupt handling, less TLB flushes)
- Of course, **access is protected**—can be read only in kernel mode
- But everyone can **address them**!

Meltdown

execute a ① **forbidden access**
and ② **speculatively use the result**
with ③ **nonarchitectural side-effects**
that reveal the secrets
before the forbidden access is squashed

We try to read anything we want, **provided that it is mapped in our virtual addressing space** (but the value will be removed from the ROB and an exception thrown)

Attacker:

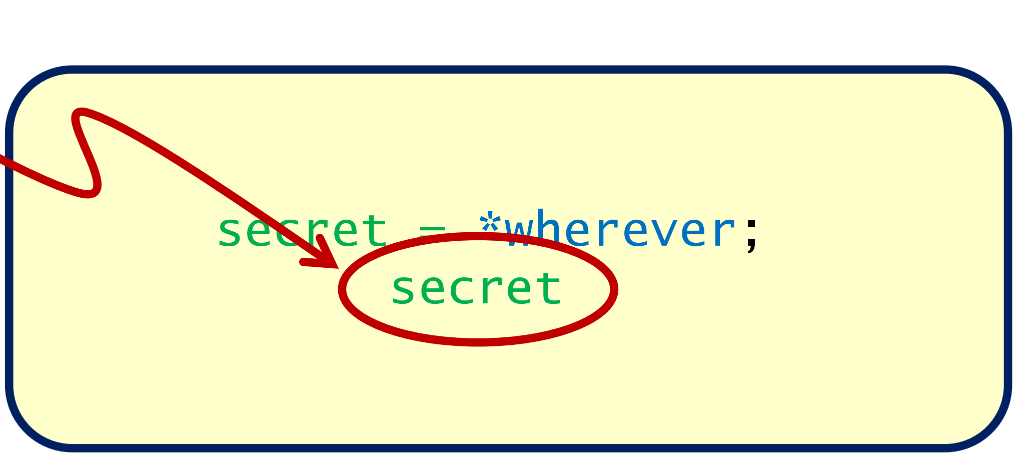
```
*wherever;
```

Meltdown

execute a ① **forbidden access**
and ② **speculatively use the result**
with ③ **nonarchitectural side-effects**
that reveal the secrets
before the forbidden access is squashed

Before the exception is thrown something else will be executed

Attacker:



```
secret = *wherever;
secret
```

Meltdown

execute a ① **forbidden access**
and ② **speculatively use the result**
with ③ **nonarchitectural side-effects**
that reveal the secrets
before the forbidden access is squashed

Attacker:

```
secret = *wherever;
array[secret * 4096];
```

If we use the abusively loaded value (secret) for a **legitimate** memory access, **trace of it will remain in the cache**

Meltdown

Make sure that a secret the attacker cannot read leaves a trace before it is cancelled

execute a ① **forbidden access**
and ② **speculatively use the result**
with ③ **nonarchitectural side-effects**
that reveal the secrets
before the forbidden access is squashed

Perform a Prime+Probe cache attack to learn the secret

Attacker:

```
secret = *wherever;
array[secret * 4096];
```

Renamed register which will never be committed

The ROB View

secret

wherever

```
0x1000000c: lw  $r3, 0($r5)
0x10000010: sll $r3, $r3, 12
0x10000014: add $r3, $r3, $r6
0x10000018: lw  $r4, 0($r3)
```

array[]

Excpt.	PC	Tag	Register	Address	Value
0					
0					
0	0x1000 0004		**$r9**		0x627f ba5a
0	0x1000 0008	**FP3**	**$f3**		**???**
1	0x1000 000c		**$r3**		**0x1234**
0	0x1000 0010		**$r3**		0x123 4000
0	0x1000 0014		**$r3**		0xf123 4000
0	0x1000 0018	**MEM3**	**$r4**		**???**
0					

head

tail

The protection violation has been discovered and is set to raise soon an exception…

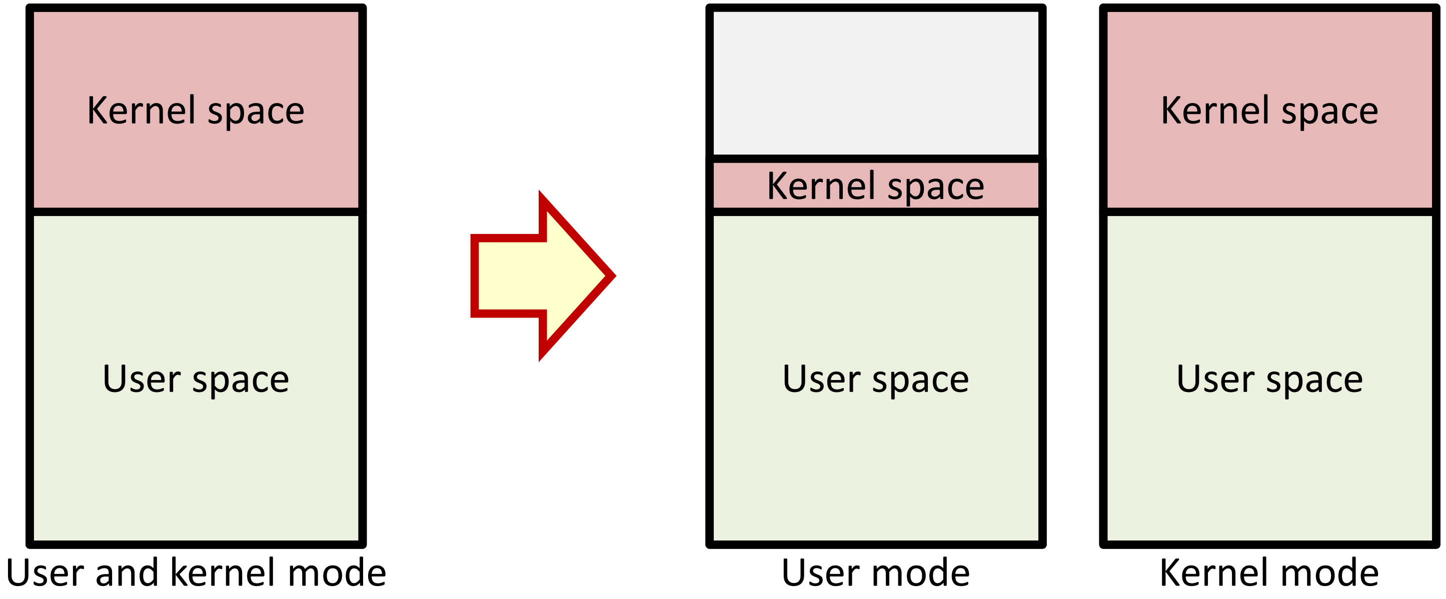
…yet the secret value is in the ROB and has been used already to affect the cache state

Does It Affect All Processors?

Processors	Affected?
Intel x86	Most processors since 1995
AMD x86	**None**
ARM	Cortex-A75
Apple ARM	Most processors
IBM POWER	POWER8 and POWER9
VIA x86	Most processors

Possible Mitigations

- The obvious proper solution is to **change the processor design**
  - Test privilege level **before** making the result of a speculative access available
  - Per chance, AMD does this already
- The other line of mitigation is to **better isolate user space and kernel** space memory
  - In Linux, **Kernel page-table isolation (KPTI)**; similar in other OSs
  - Performance penalty in Linux around 5-10%, up to 30%



7

Combined Attacks to Break Isolation and Confidentiality (Spectre)

Spectre

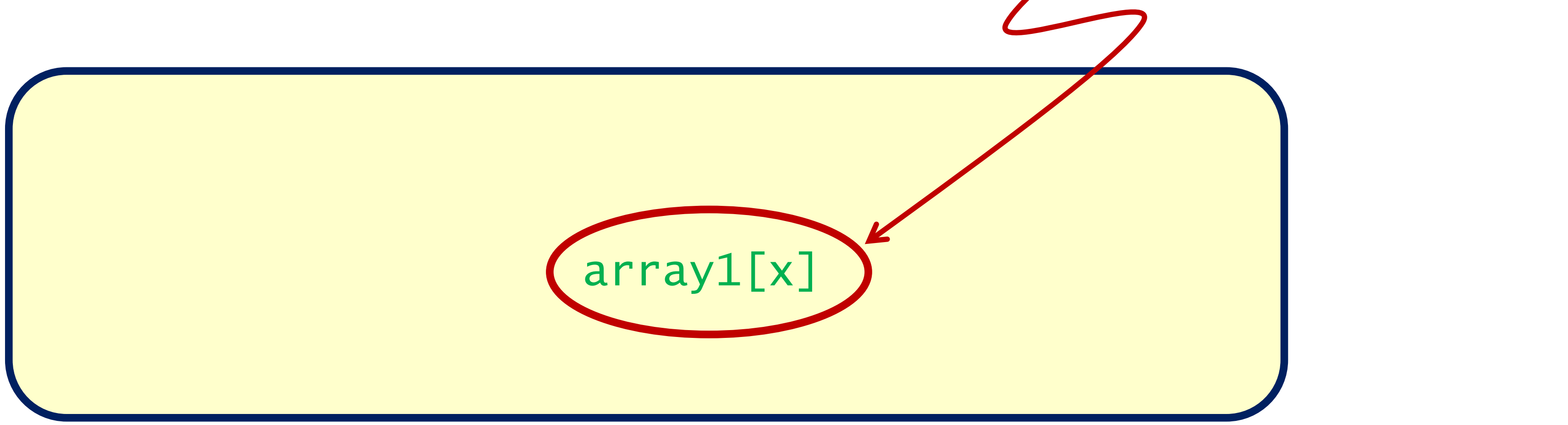
- Another catastrophic attack making it possible to **read all memory**
- Addresses another shared resource: **branch predictors**
  - For simplicity, branch predictors are not thread specific (see also Simultaneous Multithreading lecture)
- Exploits side effects of **(mispredicted) speculative execution**
  - Mispeculation does not affect the architectural state (of course!)…
  - …but it may **affect microarchitectural structures** (e.g., caches)

Get the victim to **speculatively** execute **leaky** code
whose **nonarchitectural side-effects** reveal the secrets

Spectre

② **speculatively** execute
① **leaky** code
with ③ **nonarchitectural side-effects**
that reveal the secrets

Victim:


With an appropriate value for x we can **read anything we want**
array1[x]

Spectre

If we can get the processor
to **mispredict the condition**,
the access will be speculatively performed
(but the **value will be removed from the ROB**)

② **speculatively** execute
① **leaky** code
with ③ **nonarchitectural side-effects**
that reveal the secrets

Victim:

```
if (x < array1_size)
                array1[x]
```

Spectre

② **speculatively** execute
① **leaky** code
with ③ **nonarchitectural side-effects**
that reveal the secrets

Victim:

```
if (x < array1_size)
    y = array2[array1[x] * 4096];
```

If we use the speculatively loaded value (array1[x]) for a memory access, **trace of it will remain in the cache**

Spectre

Force the victim to mispeculate

② **speculatively** execute
① **leaky** code
with ③ **nonarchitectural side-effects**
that reveal the secrets

Perform a Prime+Probe cache attack to learn the secret

Victim:

```
if (x < array1_size)
   y = array2[array1[x] * 4096];
```

Possible Mitigations

- **Hardware**
  - Disable speculative execution
  - Separate branch predictors per process/thread
- **General software** approaches
  - Run only an application per processor
- **Partial and application-specific software** approaches
  - Add serialization instructions between branches and loads
  - Make it impossible through JavaScript in browsers

"As [Spectre] is not easy to fix, it will haunt us for quite some time."

https://meltdownattack.com/

Conclusions

- Large **catalogue of powerful primitive attacks** exploiting microarchitectural state
- Real attacks are a **composition of primitives** (A → B → C…)

Matryoshka Dolls

- Fairly **difficult to fight** them comprehensively, without hardware support, and without a serious loss of performance

References

General

- J. Szefer, Principles of Secure Processor Architecture Design, Synthesis Lectures on Computer Architecture, Morgan & Claypool, 2019

RowHammer

- M. Seaborn and T. Dullien, Exploiting the DRAM RowHammer Bug to Gain Kernel Privileges, Black Hat USA, August 2015

Cache Attacks

- D. A. Osvik, A. Shamir, and E. Tromer, *Cache Attacks and Countermeasures: the Case of AES*, CT-RSA, February 2006
- Y. Yarom and K. Falkner, *FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack*, USENIX Security, August 2014
- G. Irazoqui, Th. Eisenbarth, and B. Sunar, *S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing—and its Application to AES*, IEEE S&P, May 2015

Meltdown and Spectre

- M. Lipp, M. Schwarz, D. Gruss, Th. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, Meltdown: Reading Kernel Memory from User Space, USENIX Security, August 2018
- P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, Th. Prescher, M. Schwarz, and Y. Yarom, *Spectre Attacks: Exploiting Speculative Execution*, IEEE S&P, May 2019