WS2018/19

Advanced Internet Security

Hardware-Software Co-Attacks

Adrian Dabrowski
Christian Kudera
Martina Lindorfer
Georg Merzdovnik
Michael Pucher

Slides are partially based on "Drammer: Flip Feng Shui goes mobile" by Victor van der Veen at the 2017 U'Smile Android Symposium

News from the Field

The Register®
Biting the hand that feeds IT

DATA CENTRE SOFTWARE **SECURITY** DEVOPS BUSINESS PERSONAL TECH SCIENCE

Security

PortSmash attack blasts hole in Intel's Hyper-Threading CPUs, leaves with secret crypto keys

Malware already on machines can exploit SMT using side-channel techniques to snatch private info

By Thomas Claburn in San Francisco 2 Nov 2018 at 20:18 25 SHARE ▼

Resources Industry Voice Data Strategy Spotlight
the INQUIRER
Newsletter Follow us
Search here...

News Artificial Intelligence Internet of Things Open Source Hardware Software Security All sections

Trending: OnePlus 6T IBM buys Red Hat Huawei P20 Pro Facebook breach iPhone XS

Chips

Intel Skylake and Kaby Lake CPUs vulnerable to Portsmash side-channel attack

Intel can't seem to catch a break from security woes

Roland Moore-Colyer
@RolandM_C

ZDNet
CENTRAL EUROPE MIDDLE EAST SCANDINAVIA AFRICA UK

MUST READ: Apple Mac mini (2018) review: The little Mac that

ars TECHNICA
BIZ & IT TECH SCIENCE POLICY CARS GAMING & CULTURE

SMT IN THE CROSS HAIRS —

Intel CPUs fall to new hyperthreading exploit that pilfers crypto keys

Side-channel leak in Skylake and Kaby Lake chips probably affects AMD CPUs, too.

DAN GOODIN - 11/3/2018, 12:00 AM

Intel CPUs impacted by new PortSmash side-channel vulnerability

Vulnerability confirmed on Skylake and Kaby Lake CPU series. Researchers suspect AMD processors are also impacted.

By Catalin Cimpanu for Zero Day | November 2, 2018 -- 12:19 GMT (12:19 GMT) | Topic: Security

have **discovered yet**
aby Lake generation

aneous Multithreading
data from the CPU or the
eously on a CPU core

Nvidia might be killing off the GeForce GTX 1080 TI

Overview

- Side Channels - an Introduction
- Cache
  - Flush+Reload Attack
  - & More Cache Attacks
- µArchitectural Attacks
  - The ISA is a lie
  - Meltdown
  - Spectre
- Rowhammer
  - Exploitation Steps and Variants
  - Attack Targets
  - Defenses


YOU GET A SIDE CHANNEL
AND YOU GET A SIDE CHANNEL
EVERYONE GETS A SIDE CHANNEL
imgflip.com

Sources: http://www.careersatpizzahut.co.uk/drivers/
http://time.com/5412993/pentagon-mail-quarantined-after-traces-of-deadly-poison-found-on-envelopes/

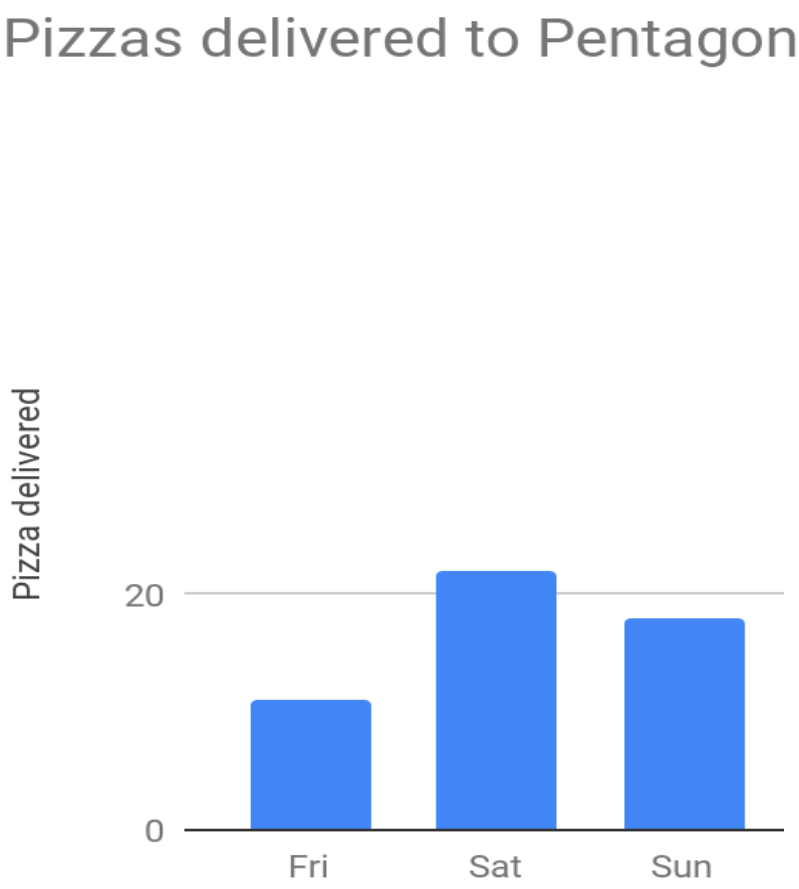
No one knows, when the invasion is going to start

Your career plans changed and you are now a pizza delivery guy in the DC area...

No one knows when the invasion...

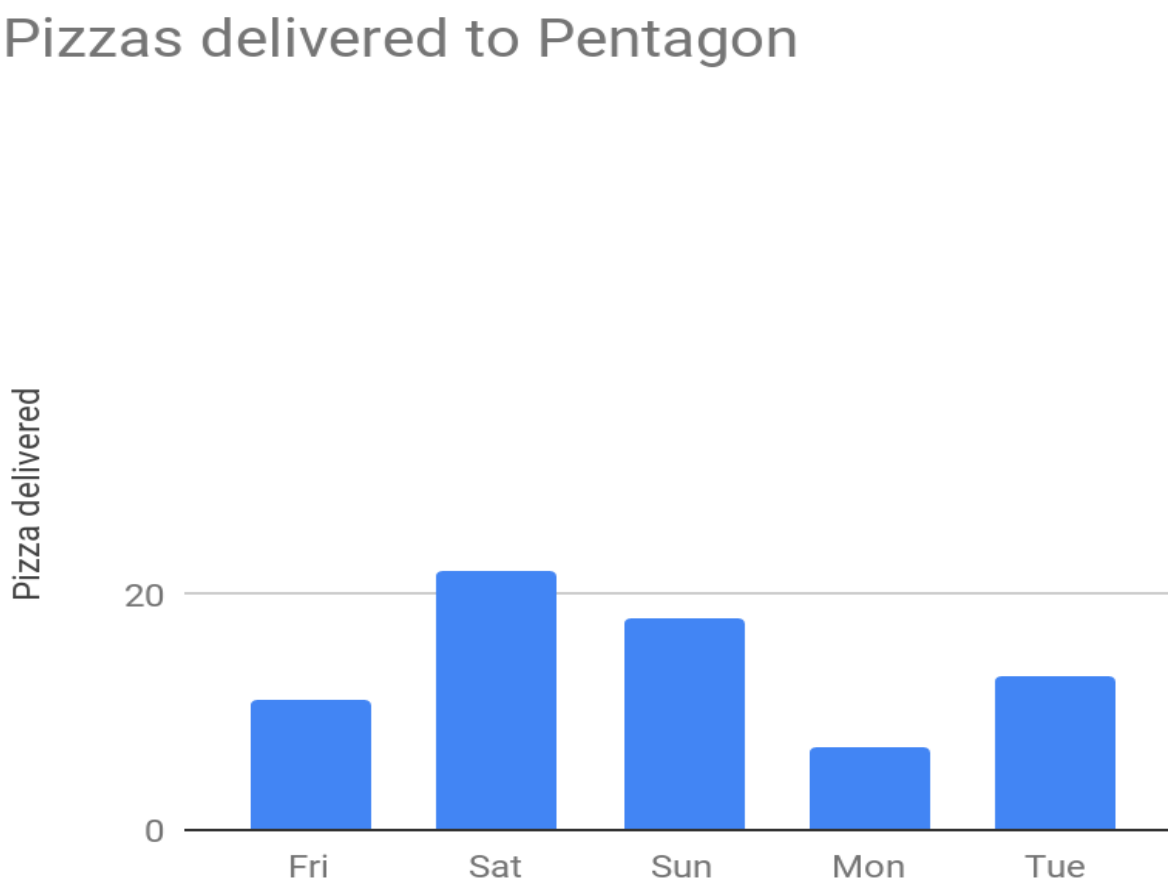
Status: USA has been threatened. Rumors of an impending invasion spread throughout Washington DC.

No one knows when the invasion...

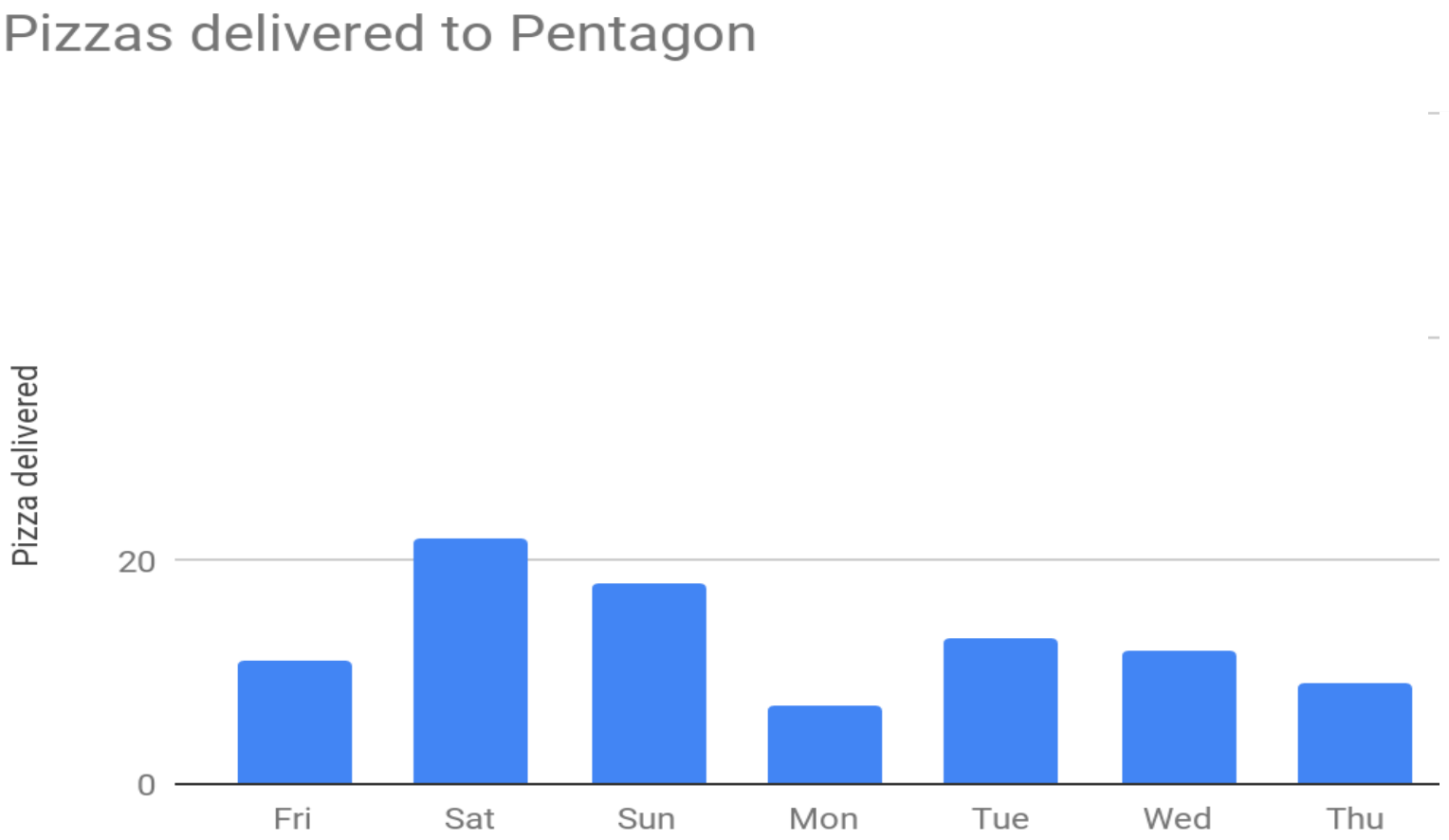


.. .but you have to continue doing your job

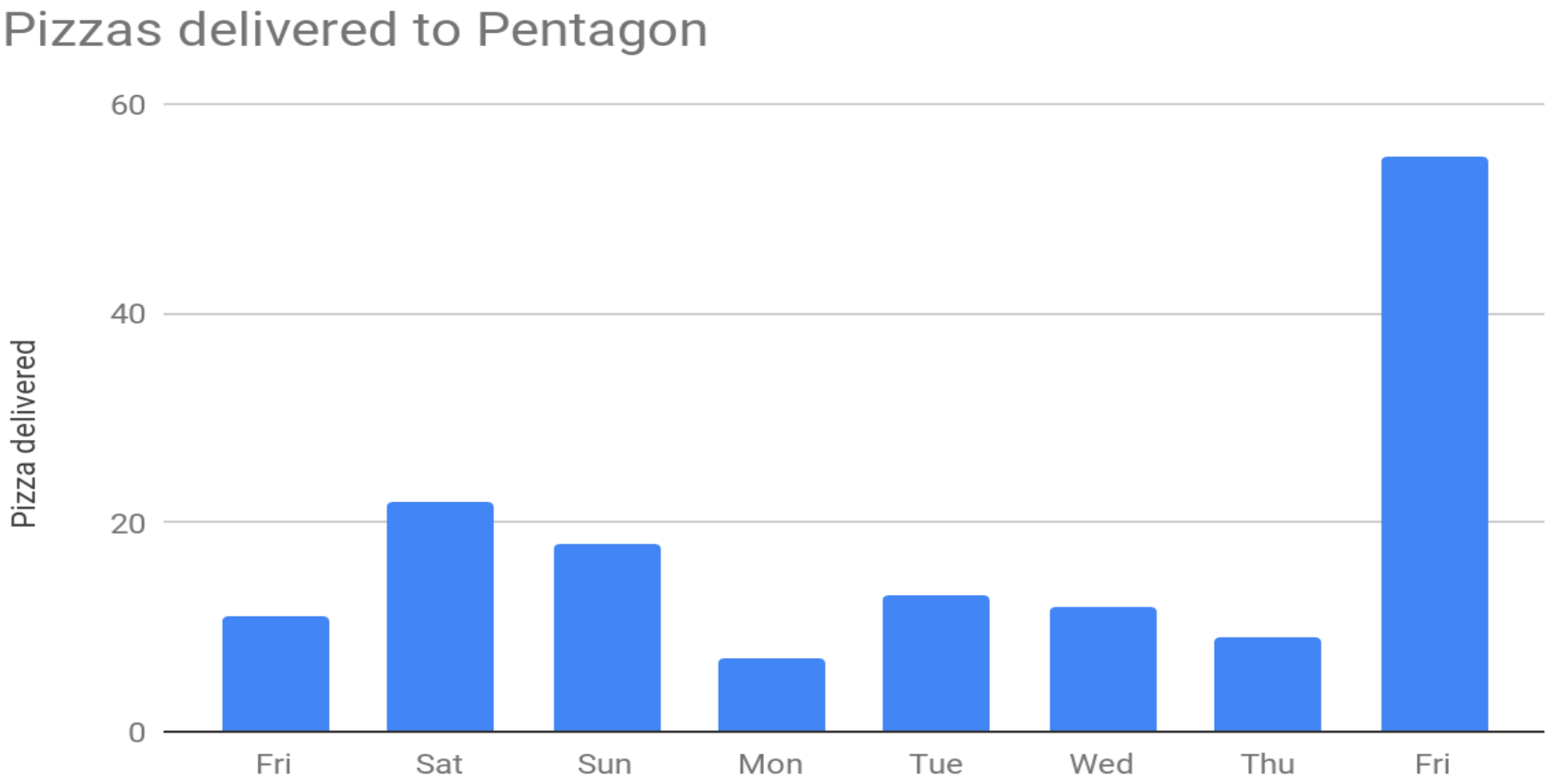
No one knows when the invasion...



No one knows when the invasion...



No one knows when the invasion...



Side Channels

- unintended (intentional → "covert channel")
- leaks information about the secret internal state
- typ. a weakness of the implementation not the algorithm
- Examples:
  - timing, heat, electro-magnetic radiation, power usage, acoustic, induced faults, optical, cache

Timing Side Channel Example

```
bool check_password(char *passwd) {
    for (int i=0; i<pass_len; i++) {
        if (passwd[i] != stored_passwd[i])
            return false;
        }
    return true;
}
```

Runtime is dependent on input!

Timing Side Channel

passwd[]	stored_passwd[]	for-exec → time
a	test	1x
b	test	1x
a	test	1x
d	test	1x
e	test	1x
f	test	1x
g	test	1x
….	….	….
s	test	1x
t	test	**2x**
ta	test	2x
tb	test	2x
tc	test	2x
td	test	2x
te	test	**3x**
…..		

```
bool check_password(char *passwd) {
    for (int i=0; i<pass_len; i++) {
        if (passwd[i] != stored_passwd[i])
            return false;
        }
    return true;
}
```

Note: Fast CPUs do not provide protection.
Faster CPUs have higher resolution timers

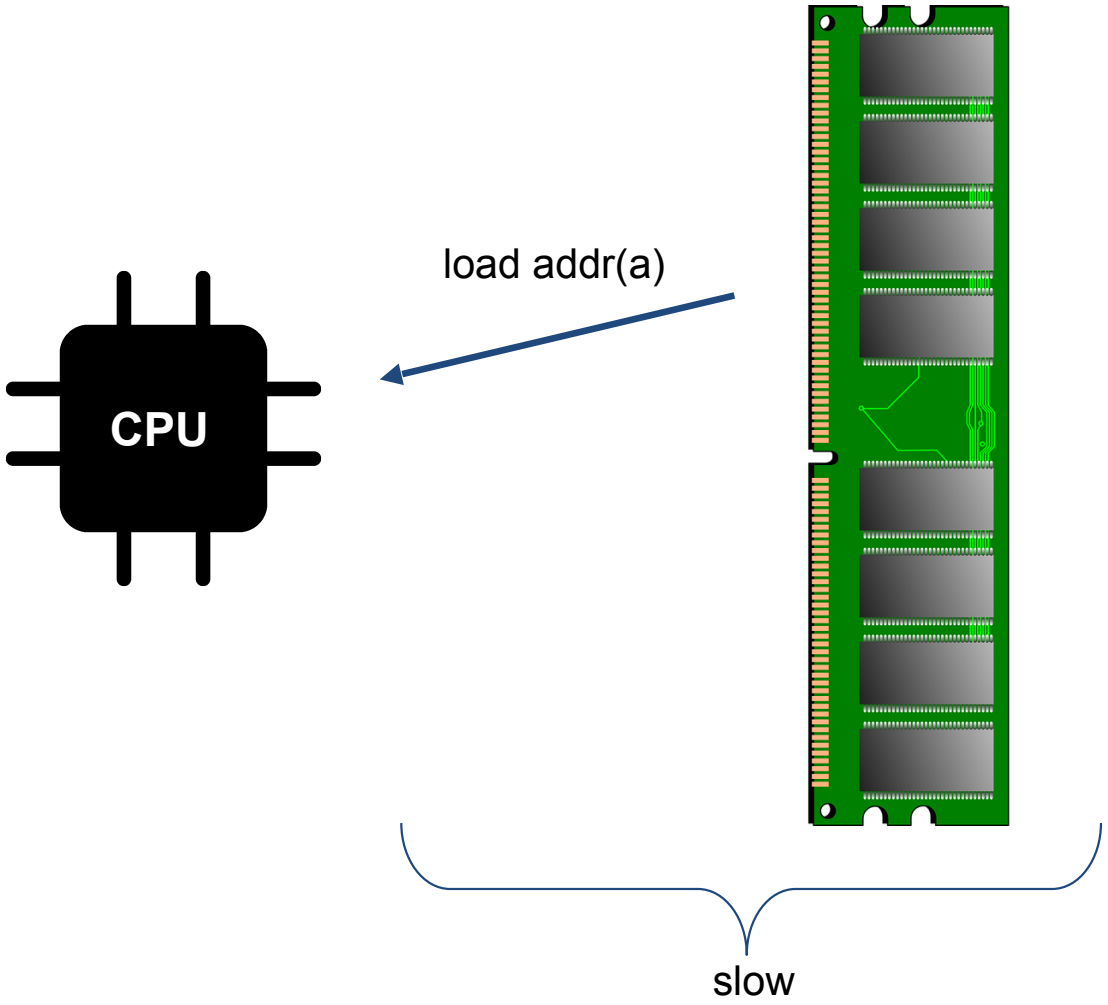
Over the network:
Repeat multiple (1000x) times, to cancel out latency and jitter.

The Cache

and RAM

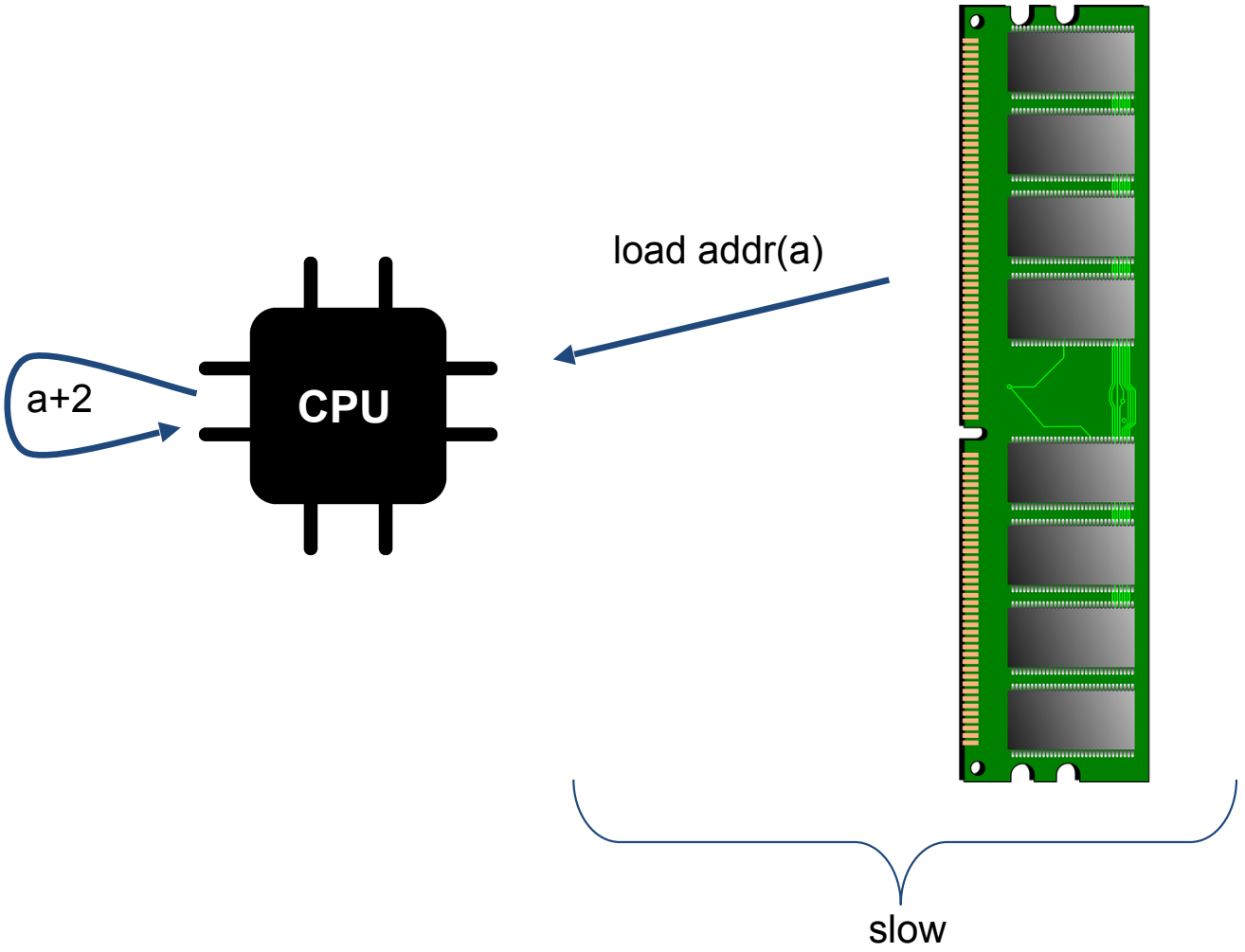
No Cache

a=a+2;



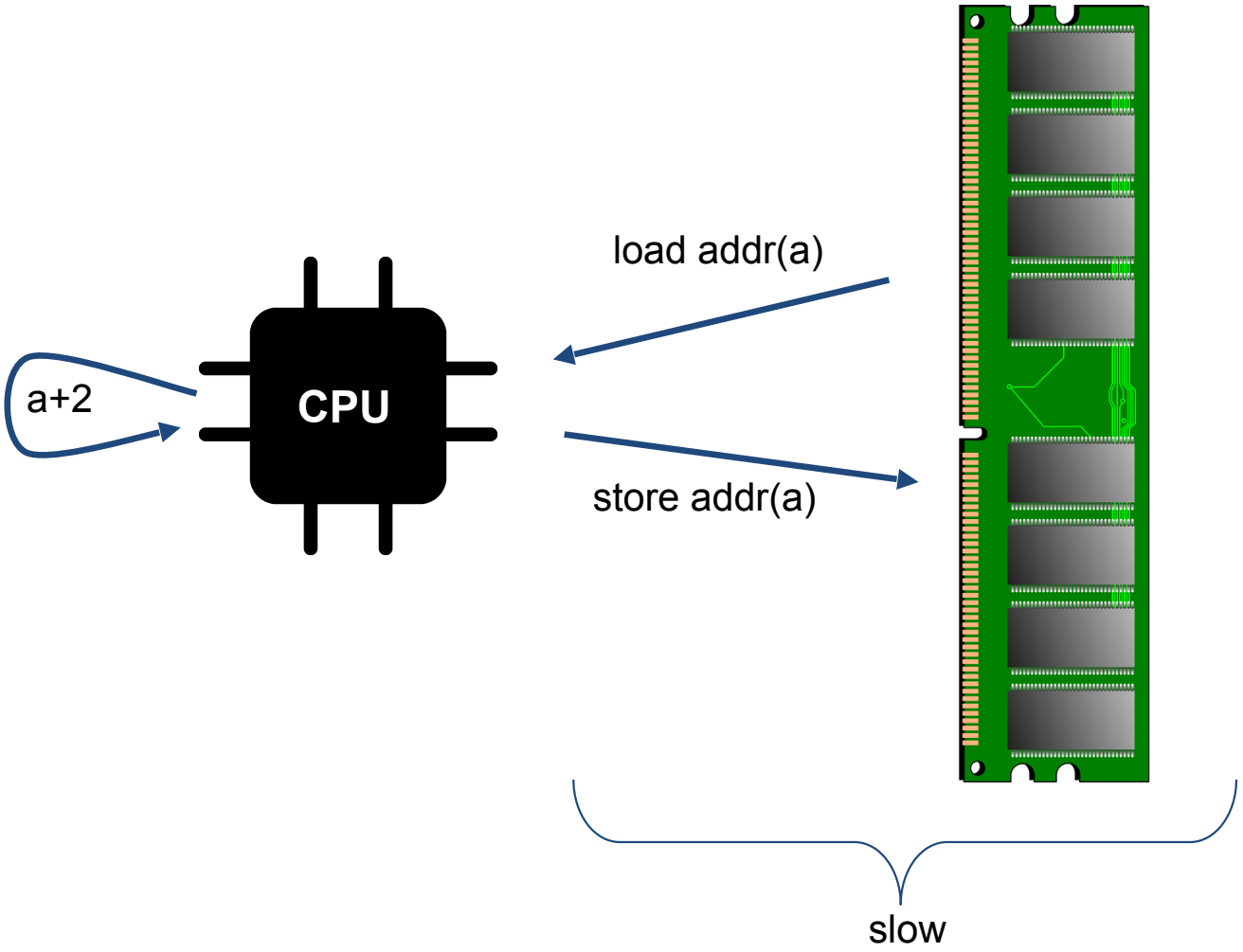
No Cache

a=a+2;



No Cache

a=a+2;



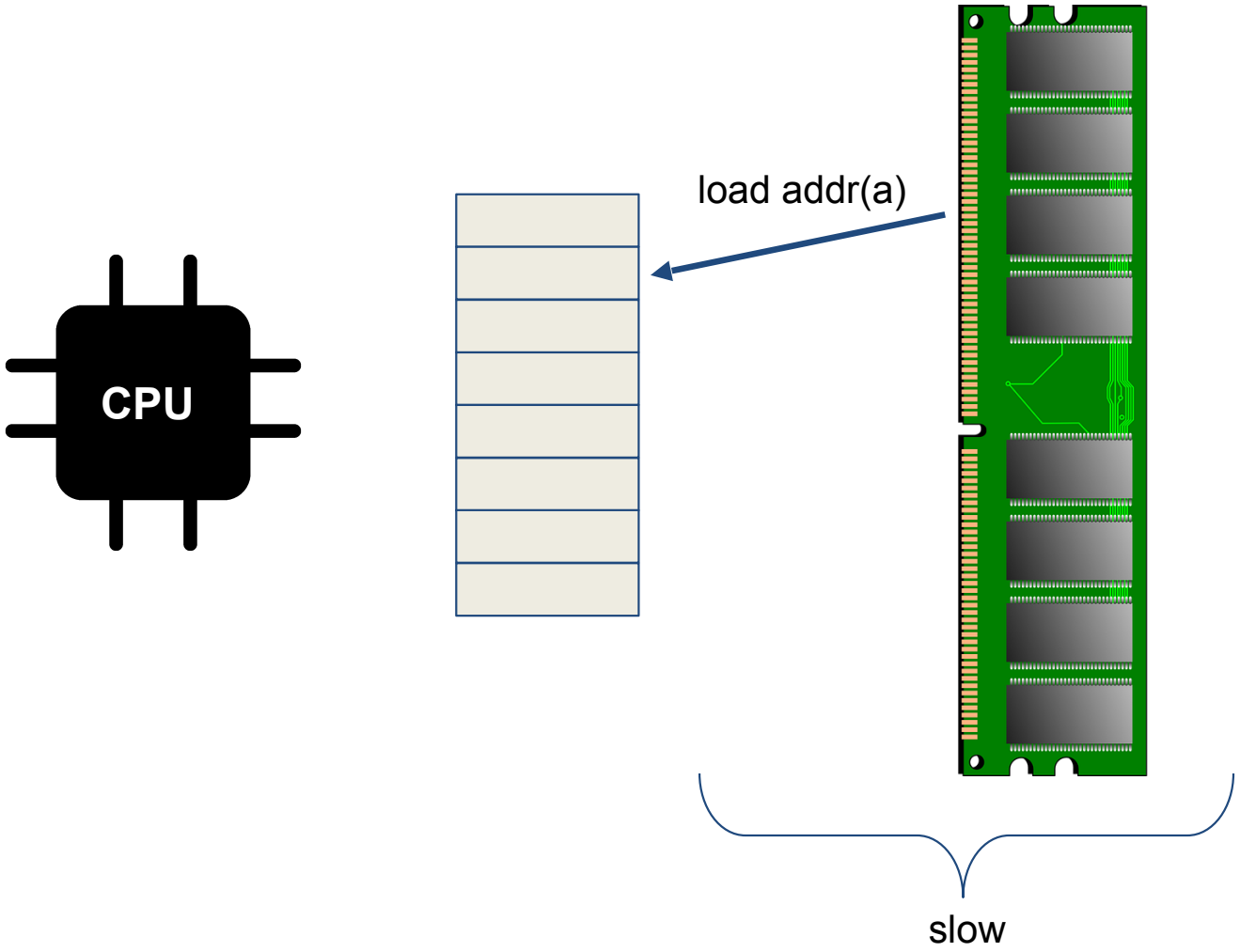
Cache

- a fast memory
- temporarily storing values from (slower) RAM
- more expensive than RAM

- Cache miss: a requested value is not stored and needs to be fetched from RAM
- Cache hit: the requested value is stored

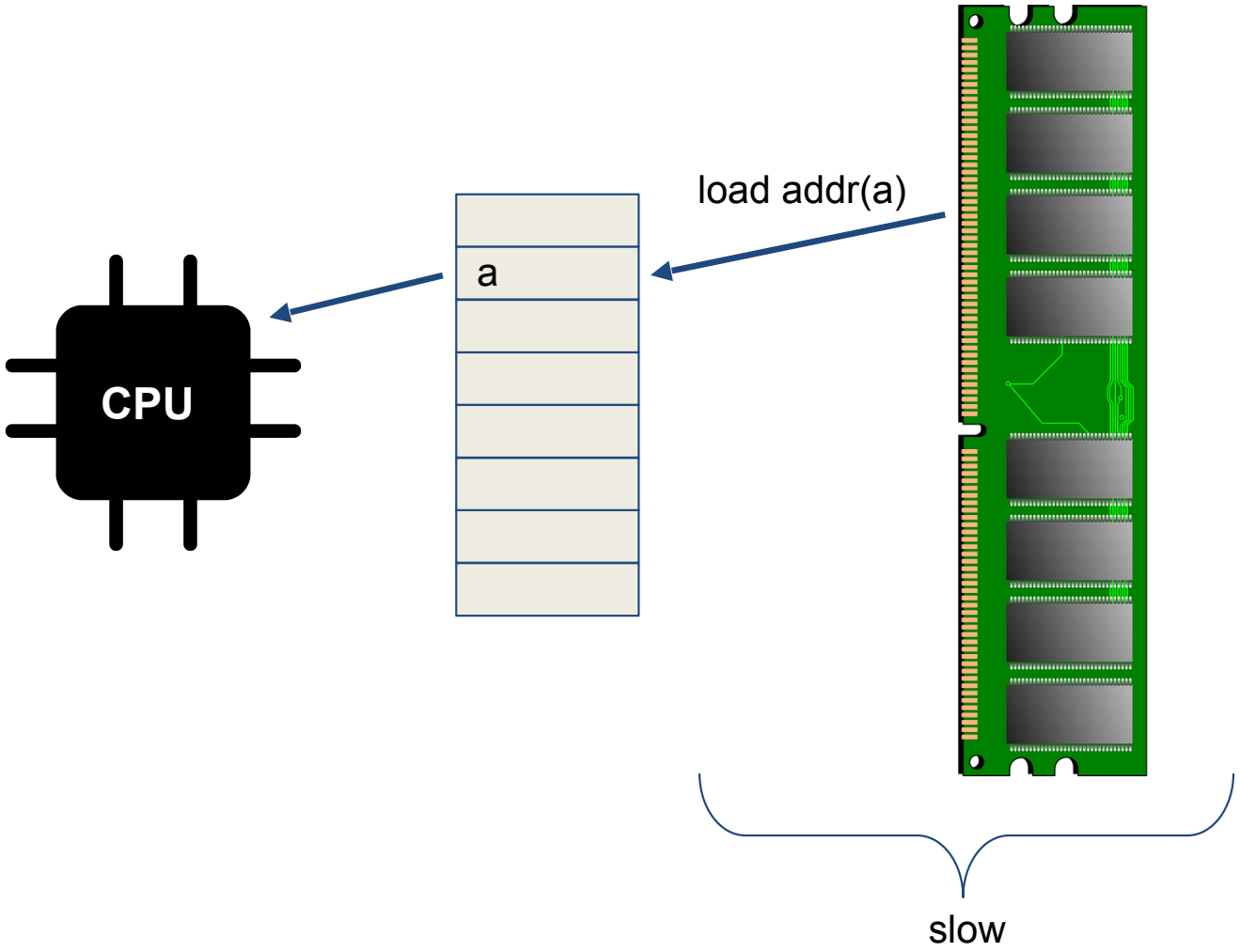
Cache miss

a=a+2;

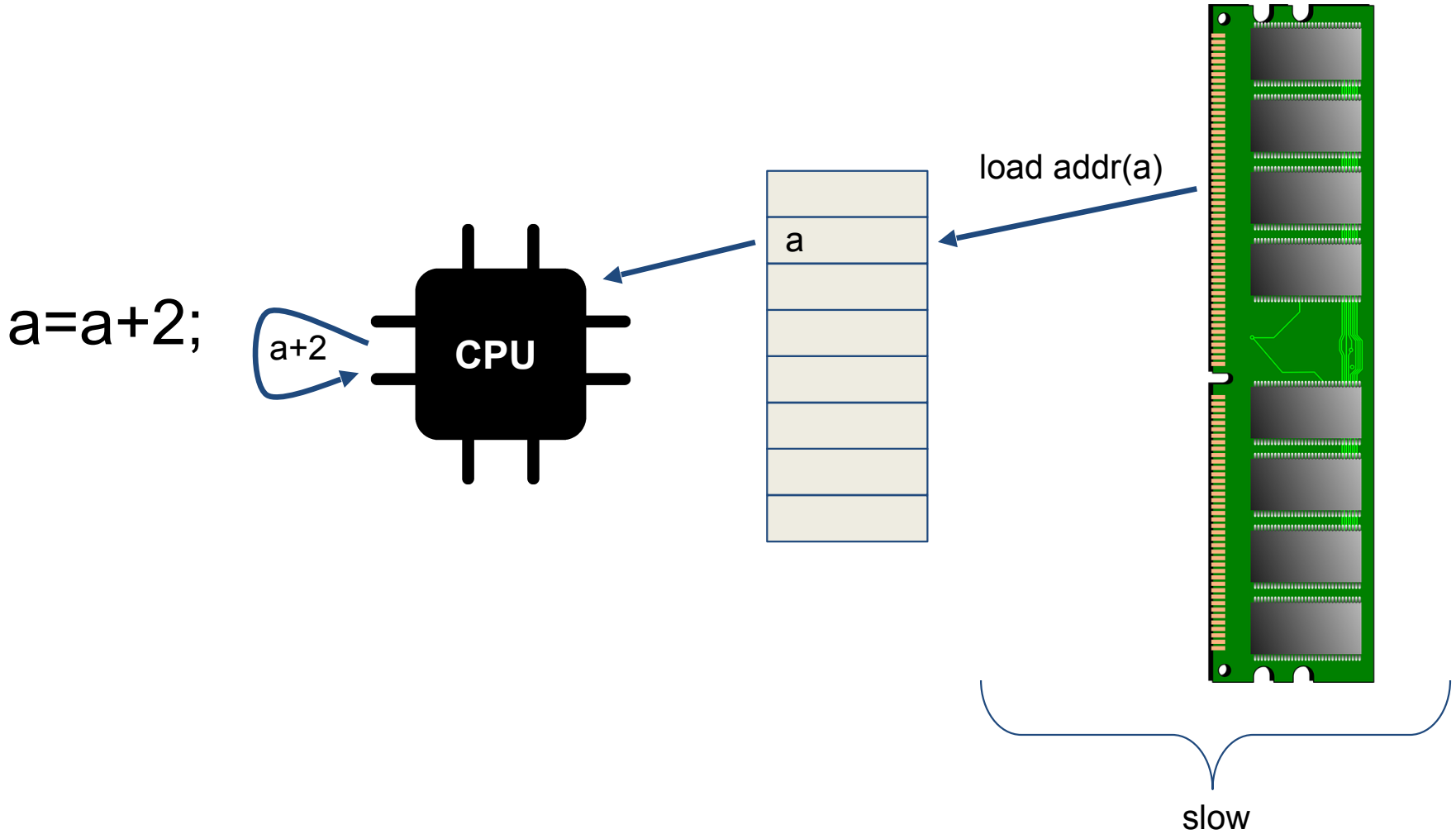


Cache miss

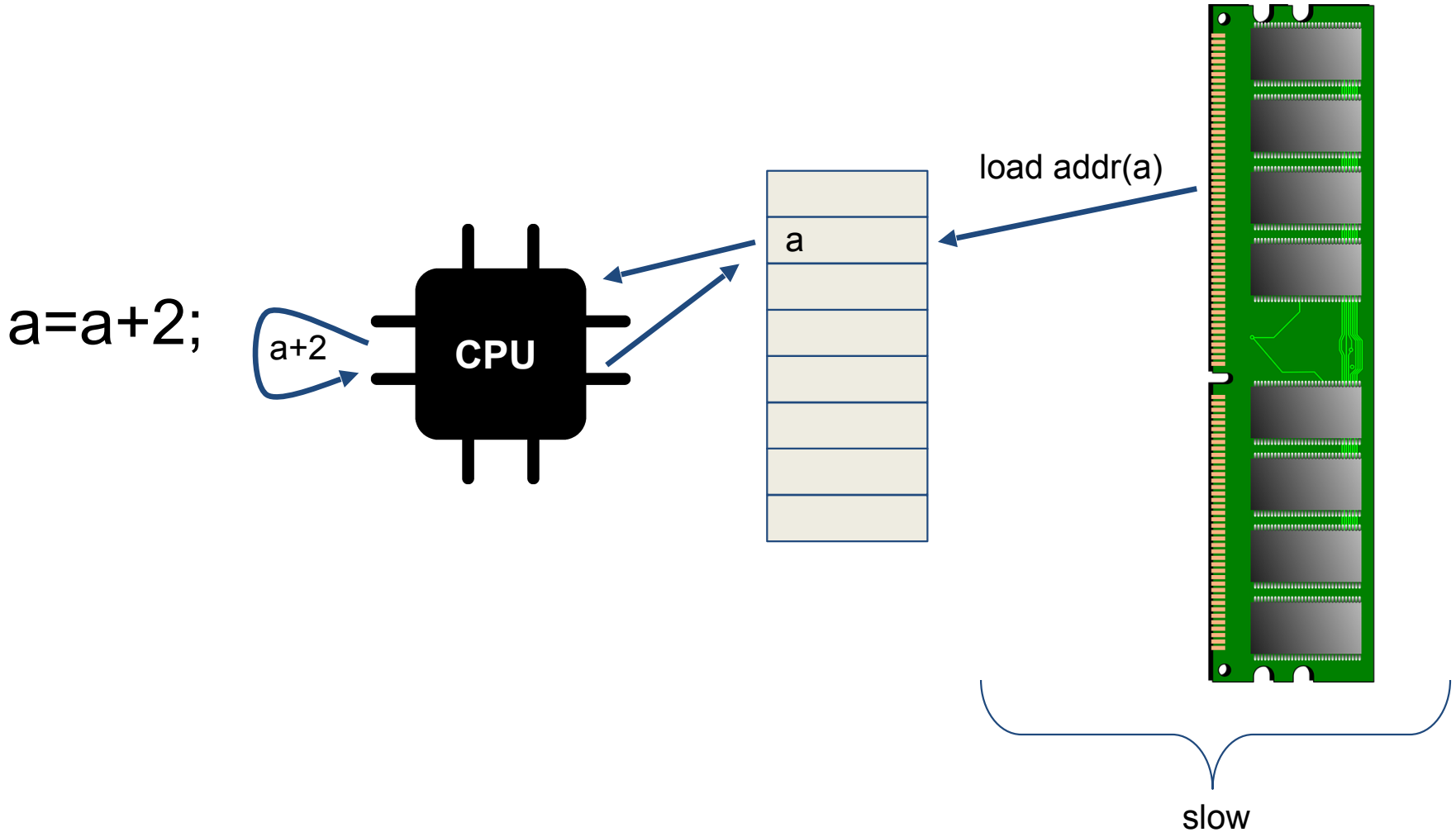
a=a+2;



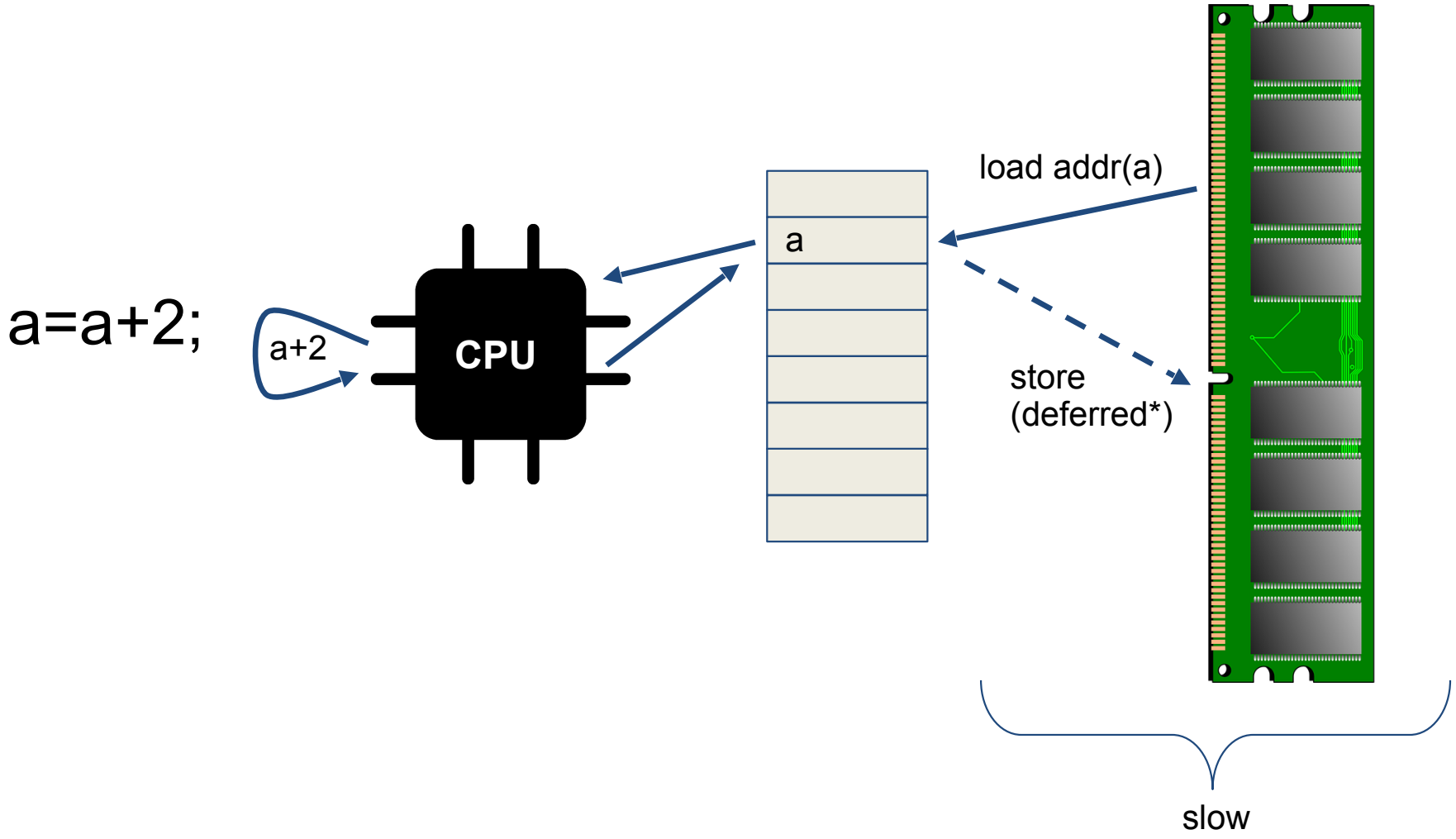
Cache miss



Cache miss



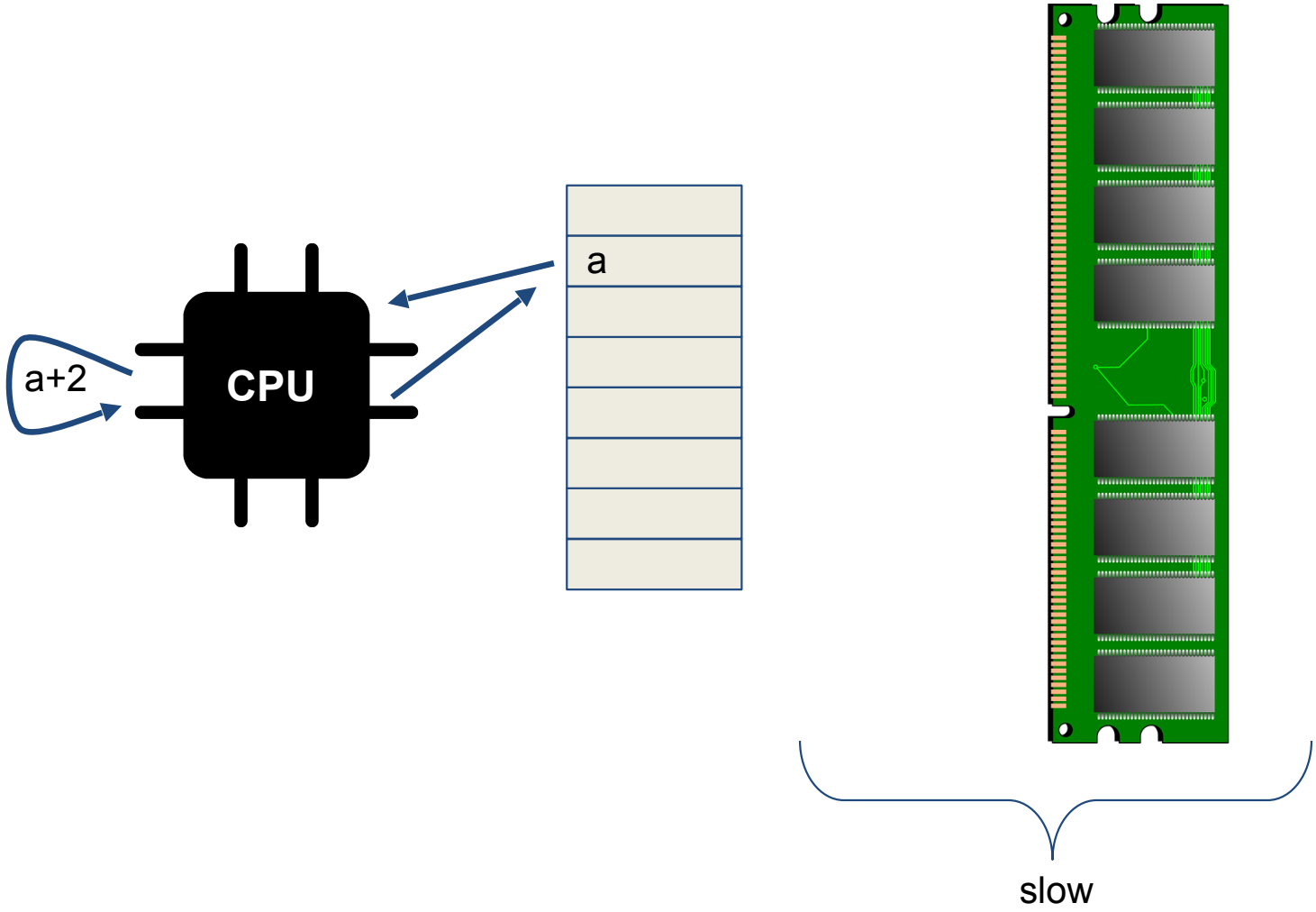
Cache miss



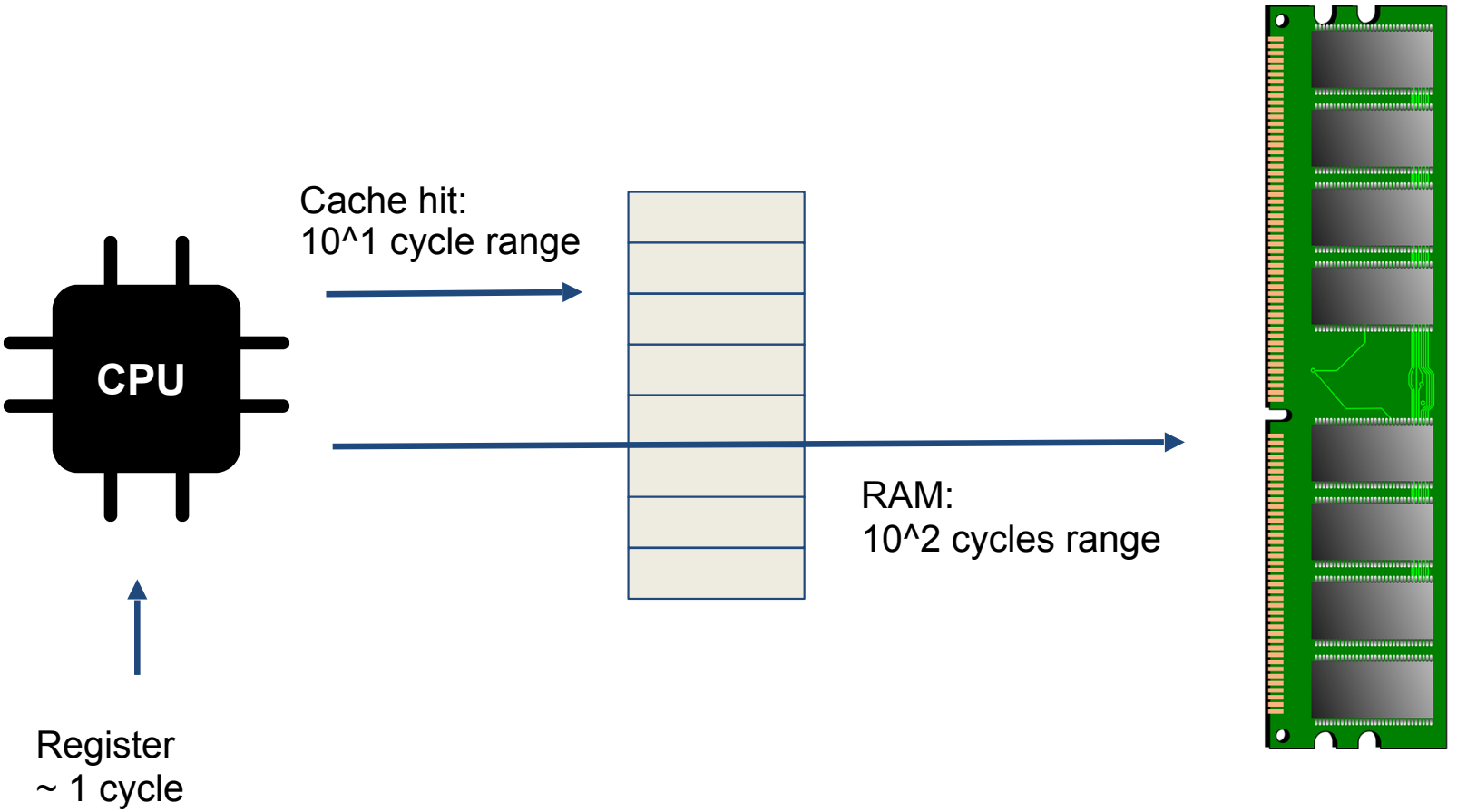
*) with write-back cache

Cache hit

a=a+2;



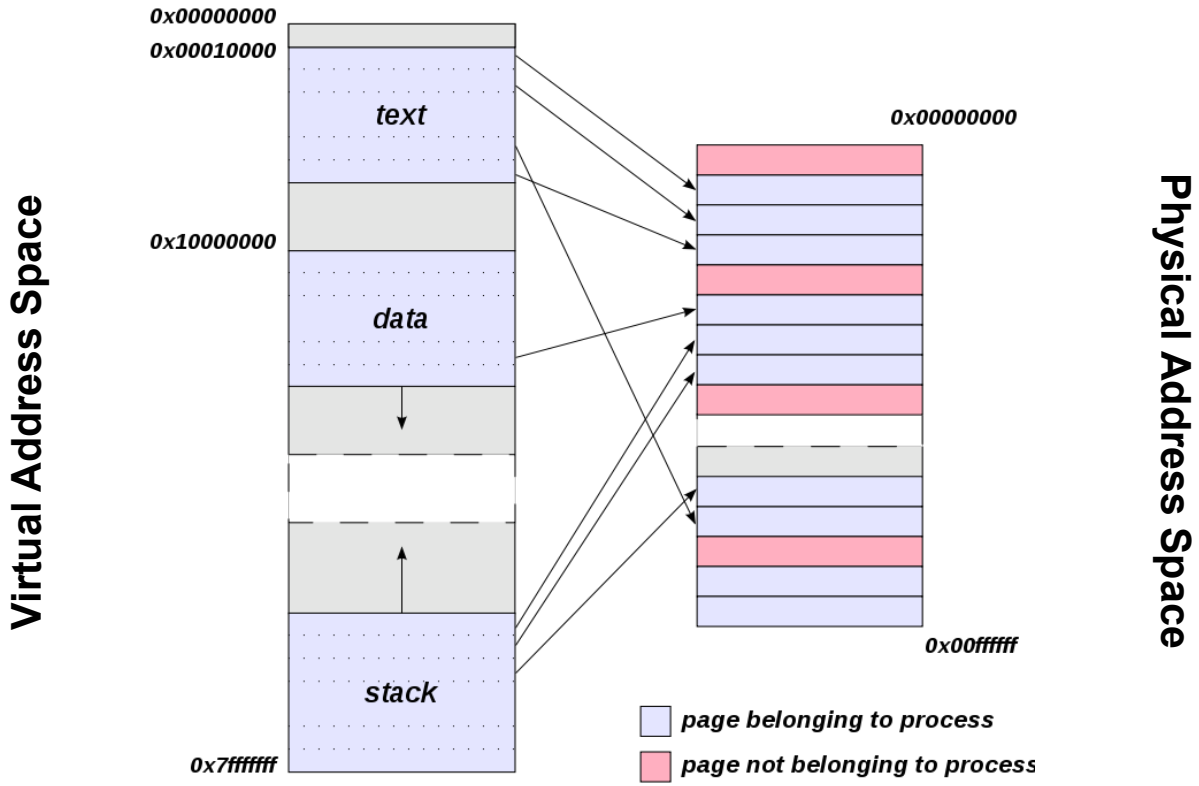
Timing (Intel, typ)



Source: https://en.wikipedia.org/wiki/Page_table

Virtual Memory Layout Basics

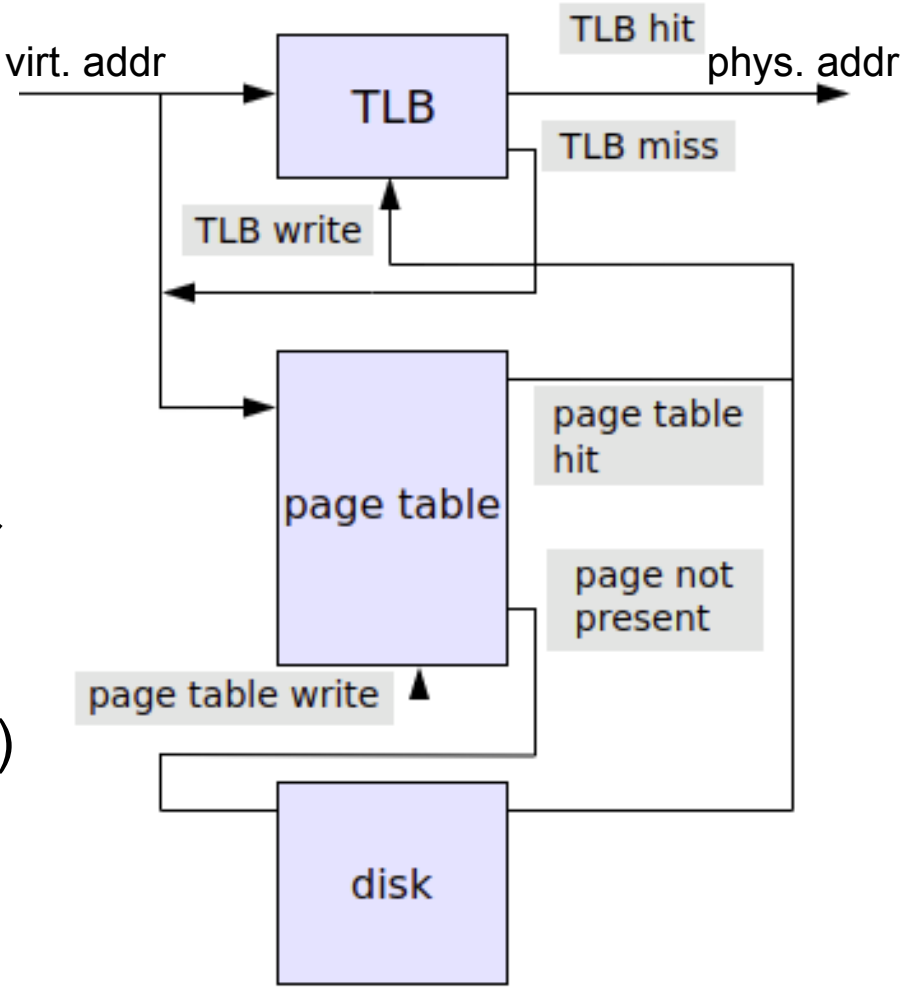
- HW uses physical memory addresses, SW works with virtual ones
- Each process works with a virtually contiguous chunk of memory, but may be spread across different parts of physical memory



Source: https://en.wikipedia.org/wiki/Page_table

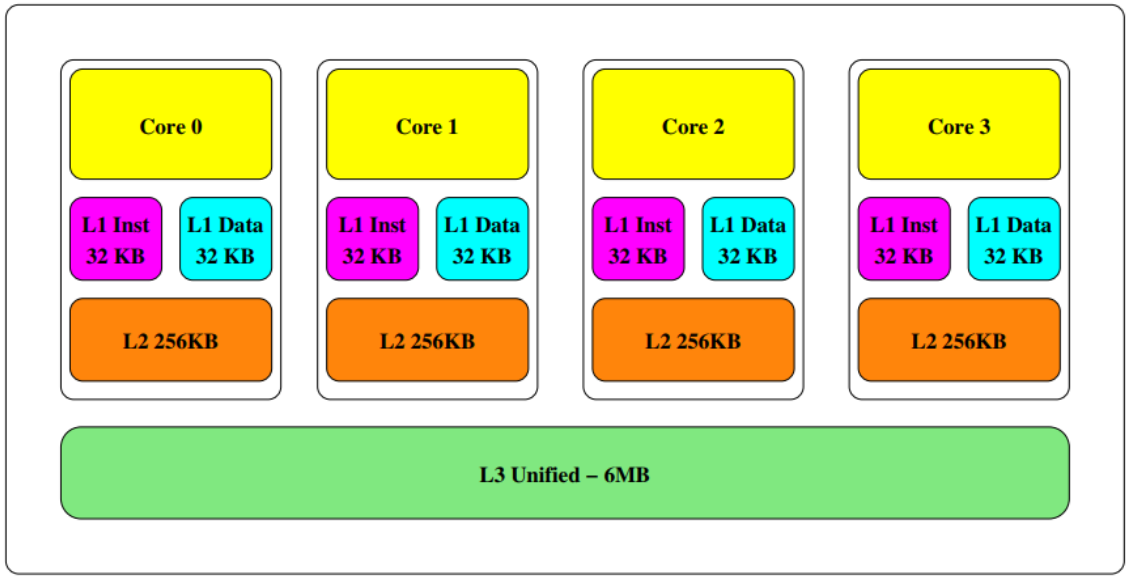
Virtual Memory Basics

- Virtual memory manages a “private” address space for each process
- Mapping is managed by the OS via Page Tables (PT) and used by the CPU’s MMU to lookup virtual addresses to physical addresses.
- PT includes flags about permissions & status
- PT are nested, lookups are cached in the Translation Lookaside Buffer (TLB)
- Switching PT is expensive! e.g., all caches incl TLB invalidated



Real-World Cache

- Multiple levels of cache
- some are shared between cores
- CPU & Memory Management Unit (MMU) transparently handles cache
  - but offers some cache-related opcodes: e.g., **`clflush`**
- Example: Core i5-3470



Flush+Reload - Attack (2013)

- Observations:
  - Cache is a shared resource between all processes
  - On Intel, L3 cache is shared among all CPU cores
  - Memory is cached-in when needed
  - Memory is shared between processes/virtual machines
    - shared libraries
    - memory deduplication
  - **`clflush`** opcode removes an address from all caches
- An attacker can measure load times and determine if a particular memory has been recently used.

Source: Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. 2013

Flush+Reload - Attack (2013)

- AES encryption and decryption depends on raising to the power of the (private-)key exponent.
- Exponentiation is often implemented as "Square-and-Multiply"
- Multiplication is dependent on key bits
- We can measure if the multiplication has been loaded into the cache

```
1  function exponent(b, e, m)
2  begin
3    x ← 1
4    for i ← |e| − 1 downto 0 do
5      x ← x^2
6      x ← x mod m
7      if (e_i = 1) then
8        x ← xb
9        x ← x mod m
10     endif
11   done
12   return x
13 end
```

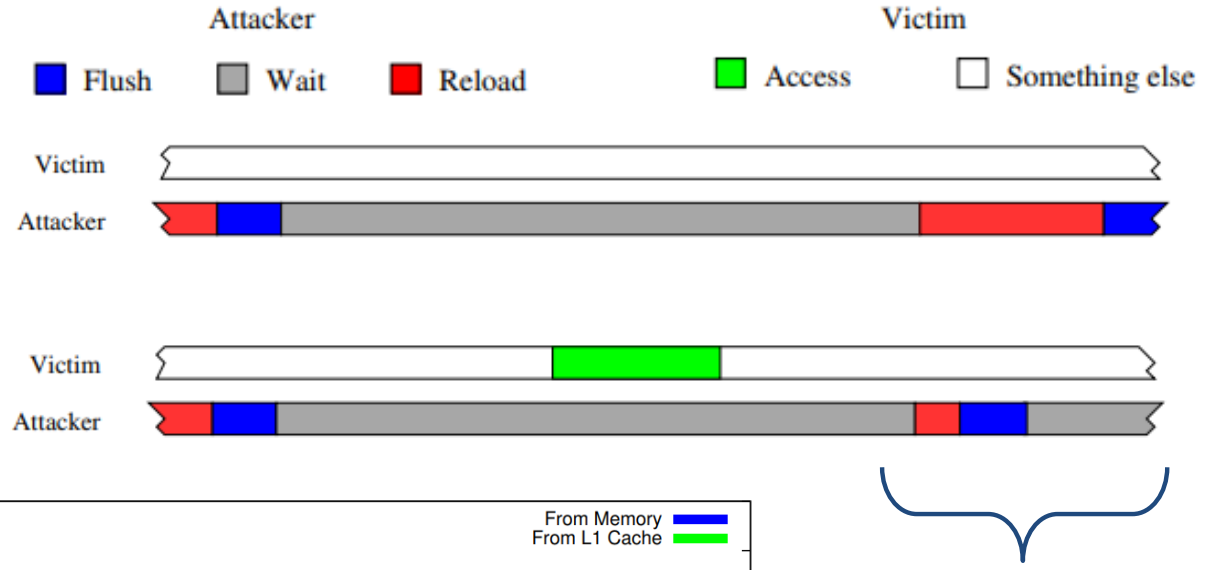
Source: Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. 2013

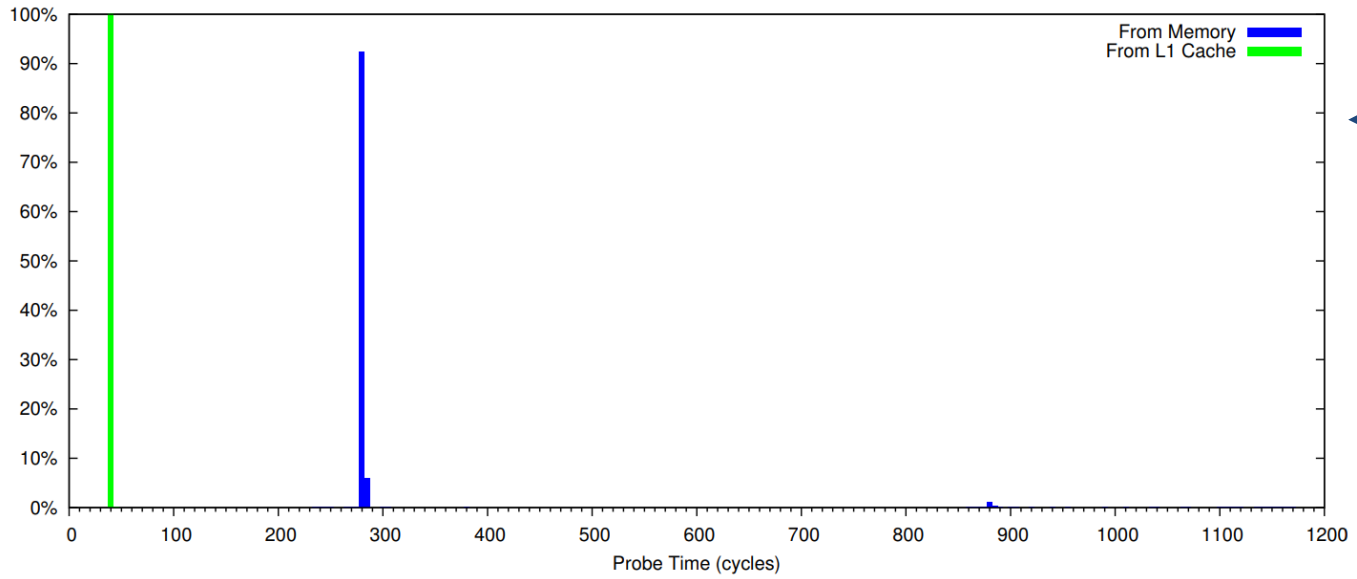
Flush+Reload - Attack (2013)

“clean” states

No access by victim

Access by victim





load times differ

Monitor if the victim has accessed a particular memory address

Source: Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. 2013

Flush+Reload - Attack (2013)

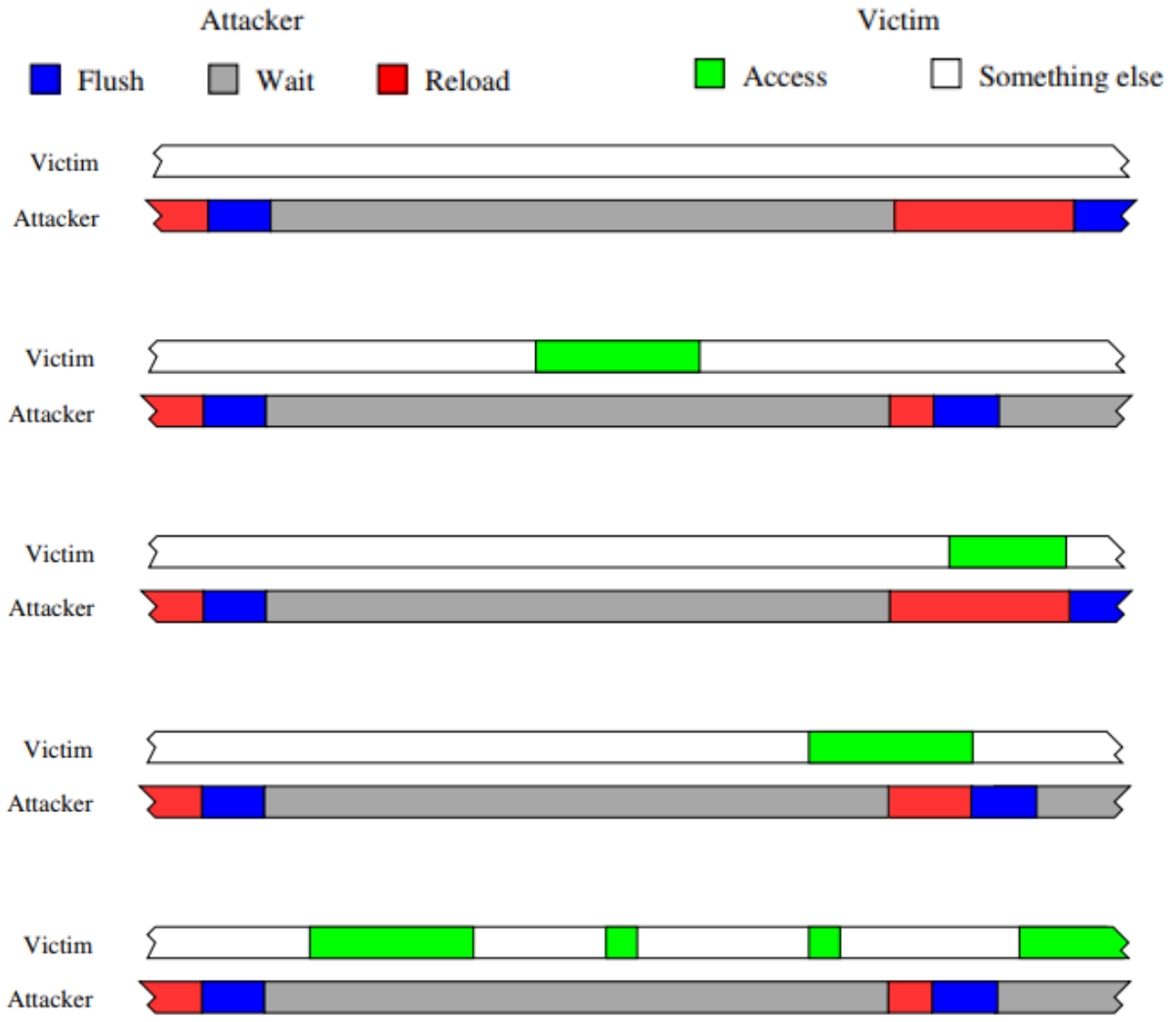
No access by victim

Access by victim

Victim access overlap

Partial overlap

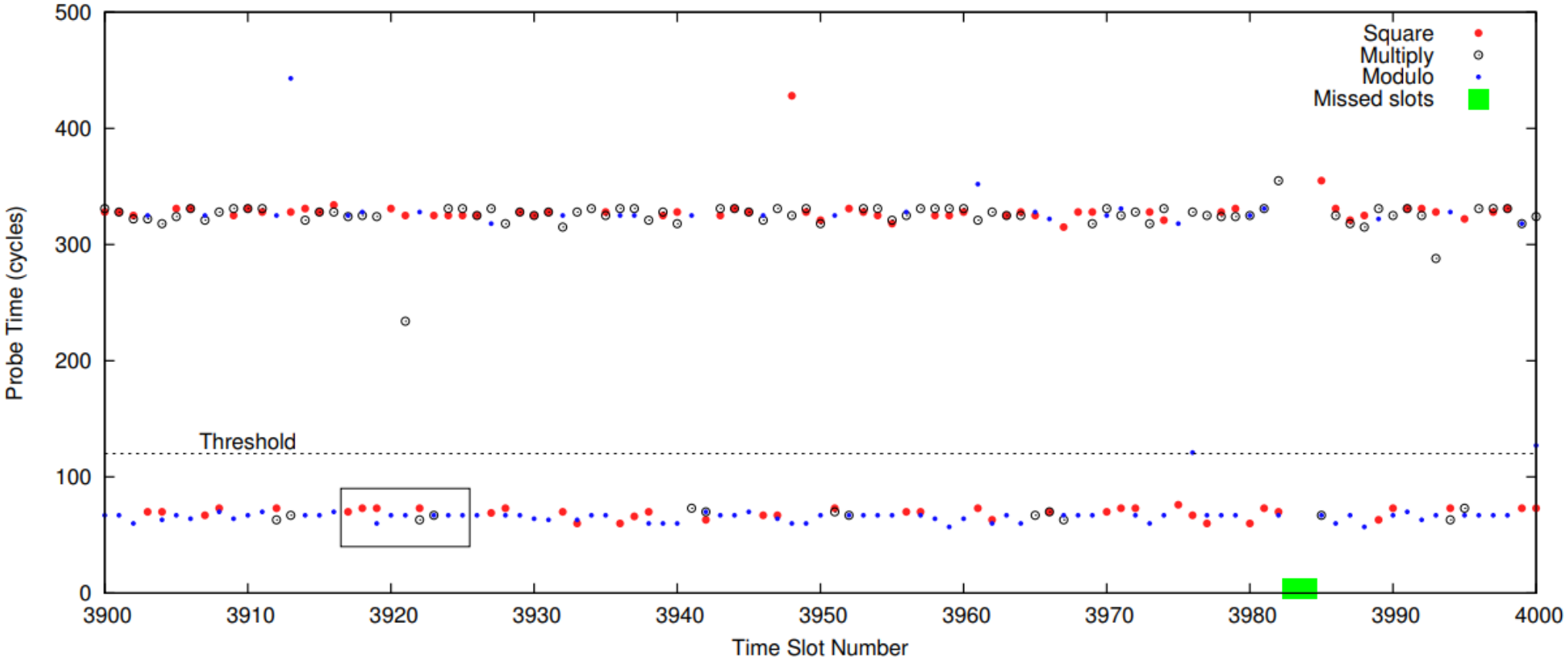
Multiple victim accesses



Source: Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. 2013

Monitoring Square, Multiply and Mod

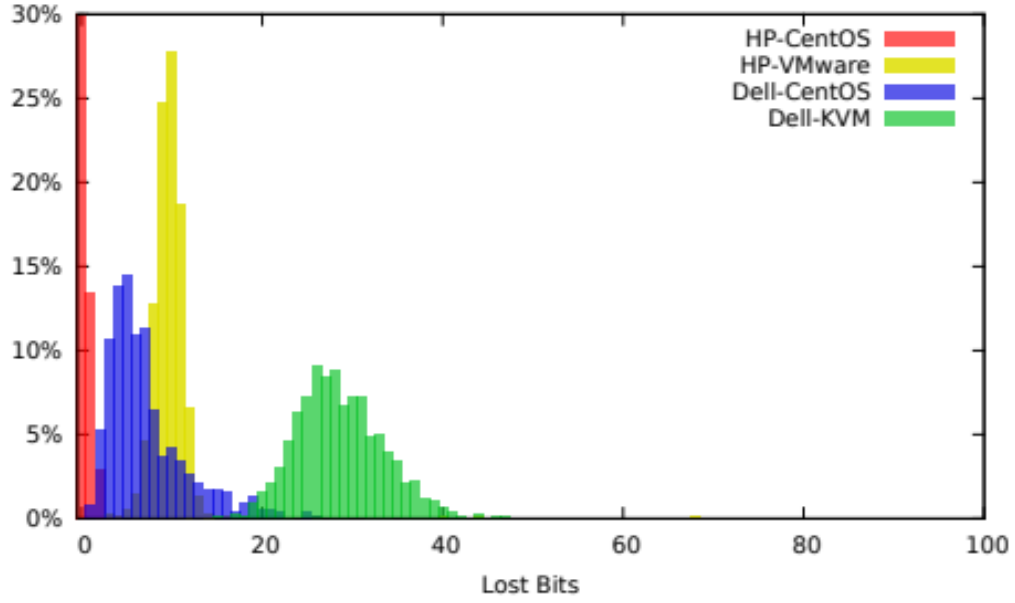
reveals actions in every single loop traversal



Source: Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. 2013

Flush+Reload - Attack (2013)

- reconstruction of AES key bits. POC with GnuPG 1.4.13
- low bit errors (after some opt.)



- Mitigate by always multiplying

```
function exponent(b, e, m)
begin
  x ← 1
  for i ← |e| − 1 downto 0 do
    x ← x^2
    x ← x mod m
    x' ← xb
    x' ← x' mod m
    if (e_i = 1) then
      x = x'
    endif
  done
  return x
end
```

Source: Anders Fogh. Cache side channel attacks: CPU Design as a security problem (2016)

L3 cache in depth

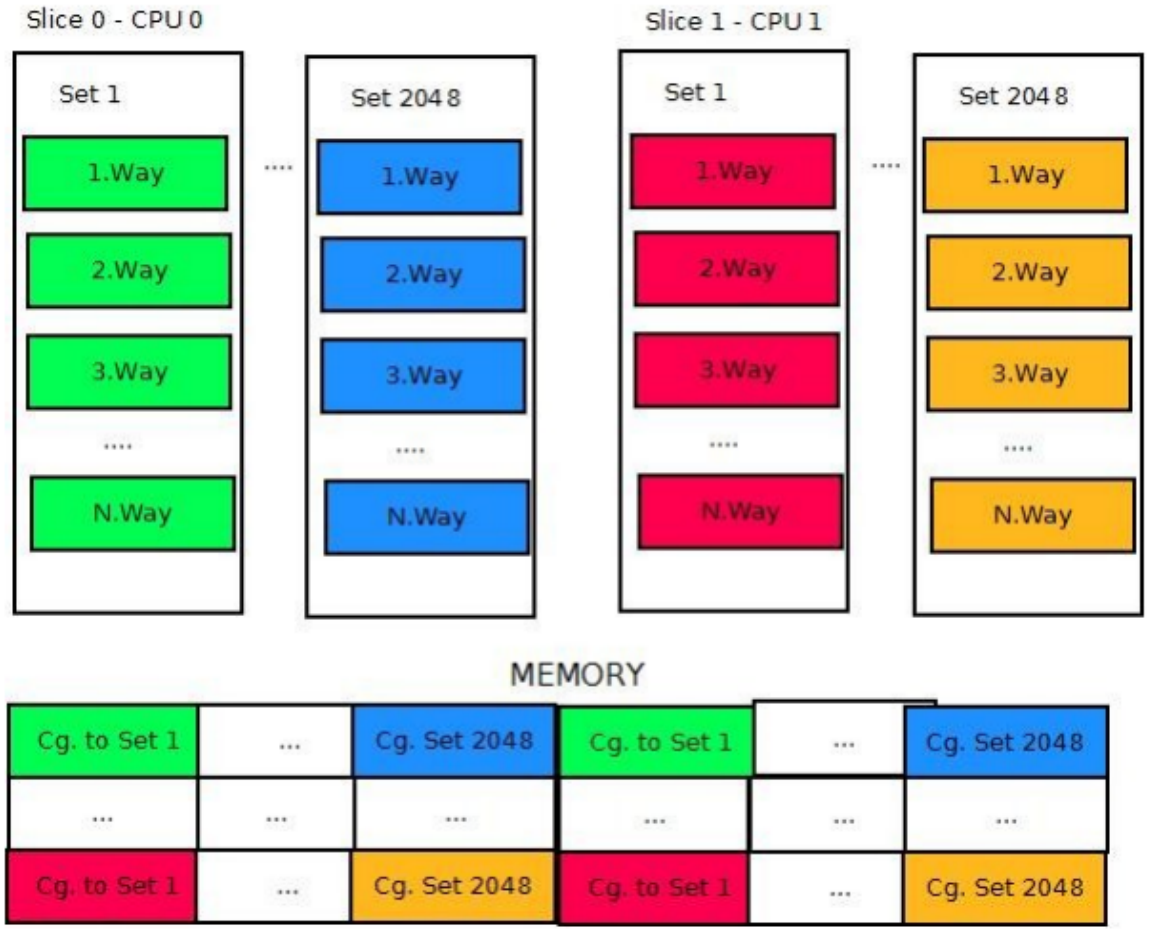
- It takes a lot of infrastructure to keep track of the position of each byte of memory in the cache
- A mapping function based on the phys. address determines where a “cache line” (typ. 64 bytes) is stored in cache.
- N-Way Set associative cache
  - Any cache line belongs to a so called cache set. Which is determined by the address.
  - There are 2048 cache sets per slice
  - Each set can store N (typically 12-20) cache lines, depending on total cache size. Each storage position is called a „way“

Source: Anders Fogh. Cache side channel attacks: CPU Design as a security problem (2016)

L3 cache in depth

If we know the mapping function, we can predict which address is loaded in which set

Addresses that map to the same cache set are “congruent”



Source: Anders Fogh. Cache side channel attacks: CPU Design as a security problem (2016)

Cache operations

- **Prime**: Place known addresses in the cache
- **Evict**: Access memory until a given address is no longer cached
- **Flush**: Remove a given address using clflush instruction

Note: you don't know what is in the cache, but you can test if something particular is in there.

Source: Anders Fogh. Cache side channel attacks: CPU Design as a security problem (2016)

Cache Attacks: Flush+Reload

- Flush + Reload
  a. flush address from cache
  b. wait for victim
  c. reload and time access to shared address
     - if c. was fast, the victim used the address

- needs shared memory
- live analysis
- if no flush opcode is available, overload the cache with other addresses

Source: Anders Fogh. Cache side channel attacks: CPU Design as a security problem (2016)

Cache Attacks: Prime+Probe

- Prime + Probe
  a. Prime a cache set to contain known attacker addresses
  b. Wait for victim activity
  c. time access to addresses from step a.
     - if slow (cach miss), victim has used memory congruent with cache set from step a.

- no shared memory needed
- live analysis
- need to know cache address mapping
- works in Javascript

Source: Anders Fogh. Cache side channel attacks: CPU Design as a security problem (2016)

Cache Attacks: Evict+Time

- Evict + Time
  a. Execute a function to prime cache
  b. Time the function
  c. Evict a cache set
  d. Time the function
     - if b was faster, then function used memory congruent to the cache set in c.

- no shared memory needed
- post-analysis
- works in Javascript

Cache Attacks: Flush+Flush

- Idea: Flushing an address from cache is slower when the address is actually in cache
  - needs to be written back

- also needs a high resolution timer

Cache attacks are noisy

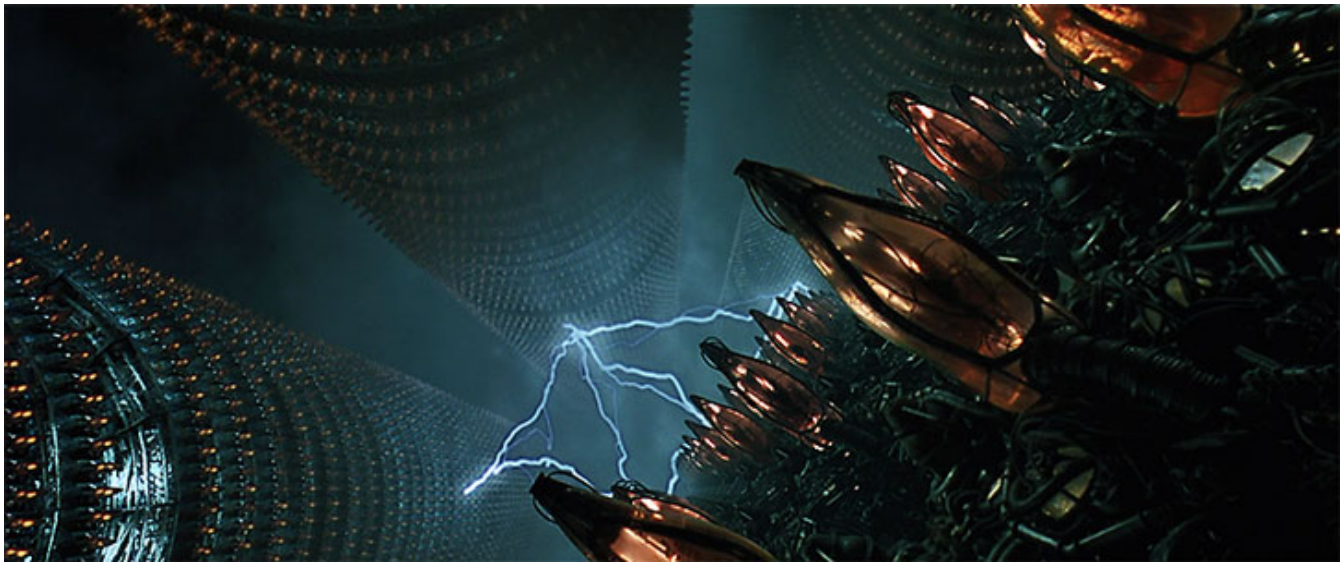
- other code running
  - also uses cache lines / cache sets
  - use other shared subsystems
- Operating System
- Interrupts
- Hardware prefetcher
- Speculative execution

→ Repeat multiple times

µArchitectural Attacks

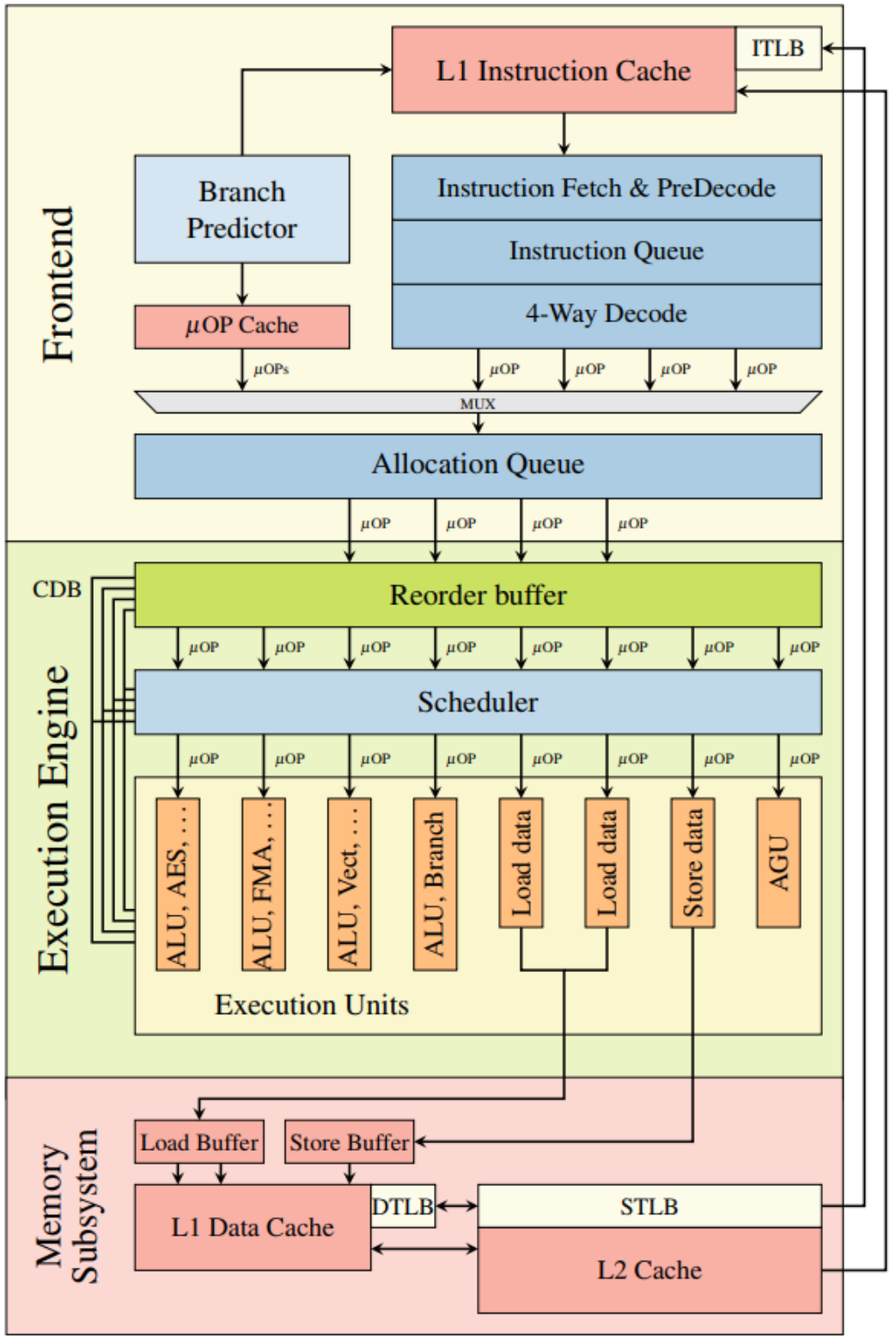
Architecture vs. µArchitecture

- The Intel architecture is a ‘lie’…
  - a convenient facade
  - x86 instruction sequences are simulated to match the i386 execution model
  - it is a story told to you to make your life easier and to not worry about the details



Reordering and Parallelism

- CISC (Complex Instruction Set Computers) instructions are often composed of multiple smaller steps
- Often-used opcodes have their own hardware implementation
  - Seldom-used and complex instructions are emulated in microcode
- In a classic CPU, only one instruction is active at a time. All other hardware functions are idle.
- What, if we could reorder instructions in a way that idle hardware can be utilized?



Source: Moritz Lipp , Michael Schwarz , Daniel Gruss , Thomas Prescher , Werner Haas , Anders Fogh , Jann Horn , Stefan Mangard , Paul Kocher , Daniel Genkin, Yuval Yarom , Mike Hamburg. Meltdown: Reading Kernel Memory from User Space

- CPU-OPs are translated into µOPs
- Reordered in a way to maximize hardware utilization of execution units.
- Results of transient instructions are "retired" (written back) in order
- Dependant results, use virtual registers
- Unknown values, are assumed
  - if wrong, then they are discarded without retiring
- Conditional branch targets are assumed (predicted)

Speculative Execution

- Fills the hardware execution units with work
- Tries executing things, even if they turn out wrong afterwards
  - Branch prediction uses run-time statistics
  - Best case: results are ready before actually “needed”
- Is ahead of the architectural execution
- Results become “visible” when retired (in-order)
- Intel & co is able to transparently hide CPU internals from programmers, provide unified opcode architecture

Paper: Moritz Lipp , Michael Schwarz , Daniel Gruss , Thomas Prescher , Werner Haas , Anders Fogh , Jann Horn , Stefan Mangard , Paul Kocher , Daniel Genkin, Yuval Yarom , Mike Hamburg. “Meltdown: Reading Kernel Memory from User Space”

Meltdown

- 1995: “The Intel 80x86 Processor Architecture: Pitfalls for Secure Systems” at Usenix
- July 2017: Anders Fogh posted blog post about speculative operations fetching data with invalid permissions
- ~Aug 2017: Jann Horn (Project Zero) first to report to Intel
- Aug-December: independently discovered by multiple groups (e.g. TU Graz)
- Premature release Jan 3rd 2018

Meltdown observations

- Memory permission is checked when the operation is retired, not when executed
  - i.e., it is fully executed on the μarchitectural level
  - including fetching the memory contents into the cache

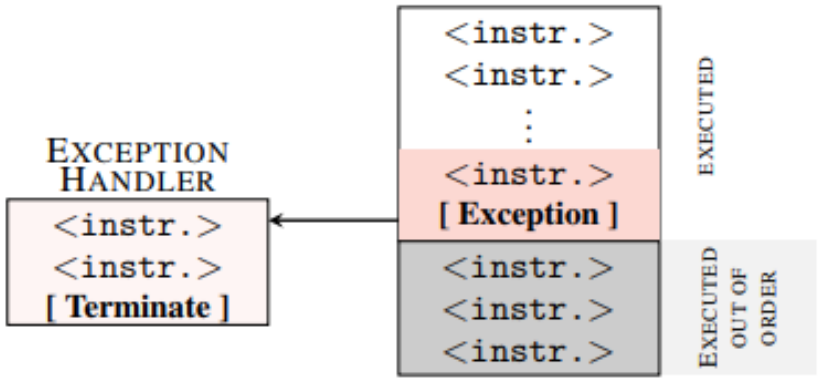
- Attack

  Have the CPU fetch a value on the μArchitectural level
  - before retirement, divert the execution “unexpectedly” -- so the branch prediction doesn’t optimize the execution

Meltdown

1. raise an exception [line 4]
   - read a forbidden memory location
2. access another (valid) memory location (probe array), based on the forbidden value [line 7]
   - only executed speculatively, code never “officially” reached architecturally
   - state of cache is changed: the specific cell from the probe array is loaded
3. in the exception handler, use flush+reload to check which memory location was accessed in 2.
   - read every cell from probe array and time it

```
1 ; rcx = kernel address, rbx = probe array
2 xor rax, rax
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```



Meltdown Walkthrough

CPU execution →

speculative execution →

```
1 ; rcx = kernel address, rbx = probe array
2 xor rax, rax
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

Meltdown Walkthrough

speculative execution reads byte from *rcx → al

CPU execution →

speculative execution →

```
1 ; rcx = kernel address, rbx = probe array
2 xor rax, rax
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

Meltdown Walkthrough

speculative execution generates array offset
rax ← al*4096

CPU execution →
speculative execution →

```
1 ; rcx = kernel address, rbx = probe array
2 xor rax, rax
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

Meltdown Walkthrough

speculative execution reads probe array rbx[al*4096]

- the specific cell from the probe array is loaded into the cache

CPU execution →

speculative execution →

```
1 ; rcx = kernel address, rbx = probe array
2 xor rax, rax
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

Meltdown Walkthrough

- CPU execution encounters exception: read from kernel memory is not allowed

CPU execution →

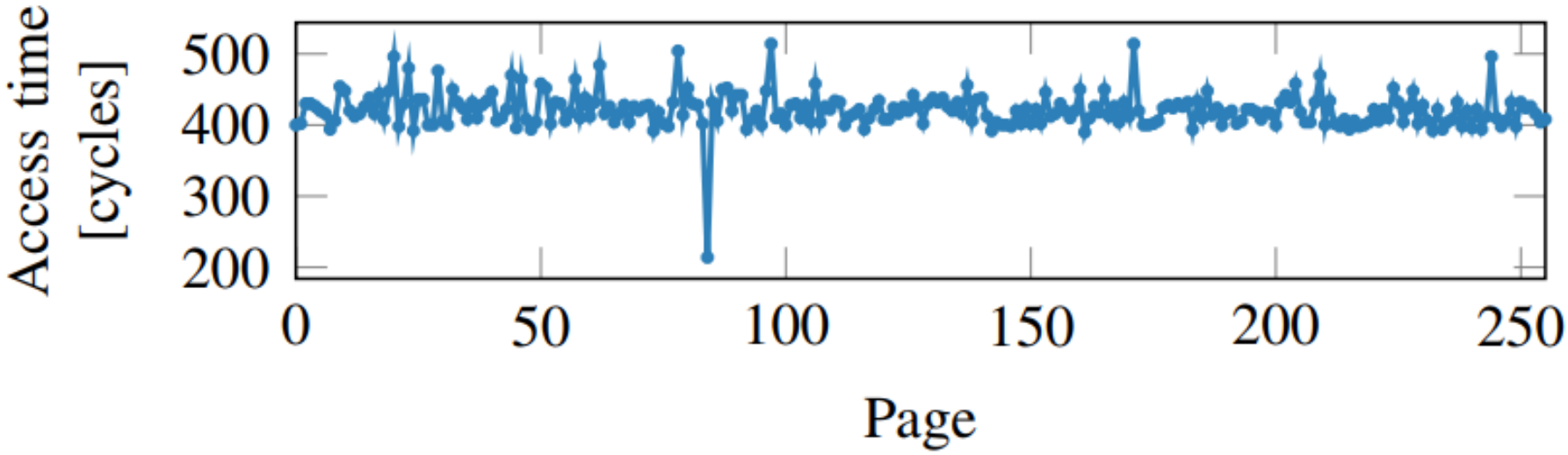
```
1 ; rcx = kernel address, rbx = probe array
2 xor rax, rax
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

speculative execution → .....

However, the cache state changed. It now contains the cache line for rbx[al<<12]

Source: Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg. "Meltdown: Reading Kernel Memory from User Space"

Meltdown Example (probeArray)



256 entries (pages), accessed depending on a byte value read from kernel memory → changes cache state

Reading back all probeArray[] entries reveals timing differences

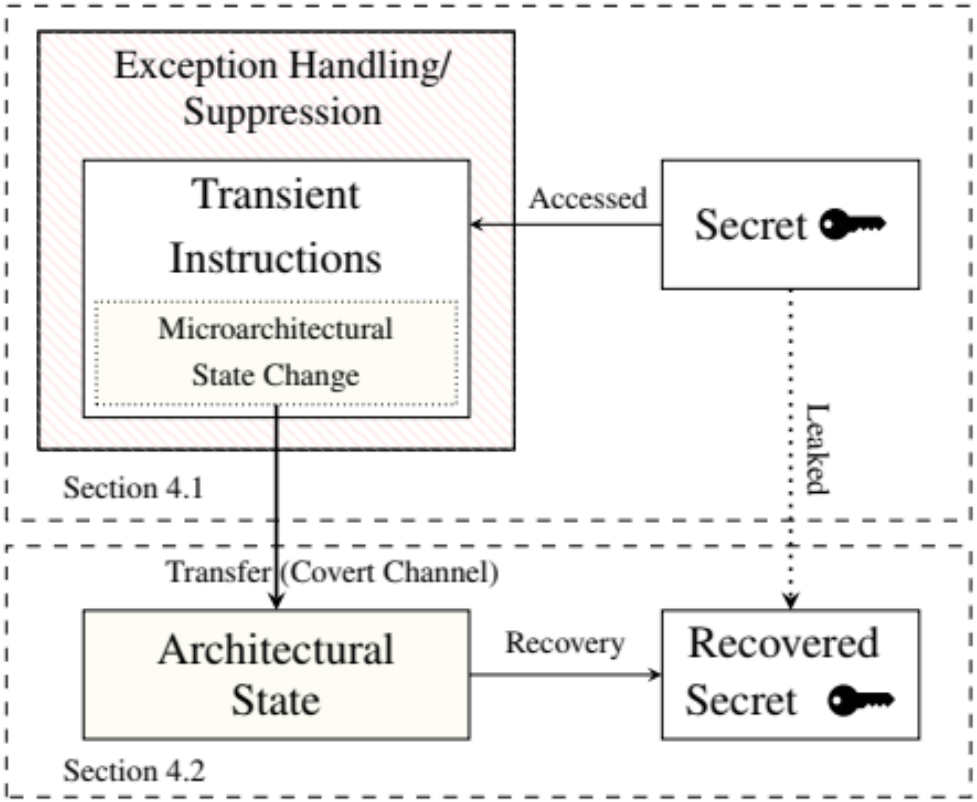
Leaking memory up to ~500 KB/s

Source: Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg. "Meltdown: Reading Kernel Memory from User Space"

Meltdown Leakage via Cache State

Even though the architectural state does not change, the μarchitectural state changes.

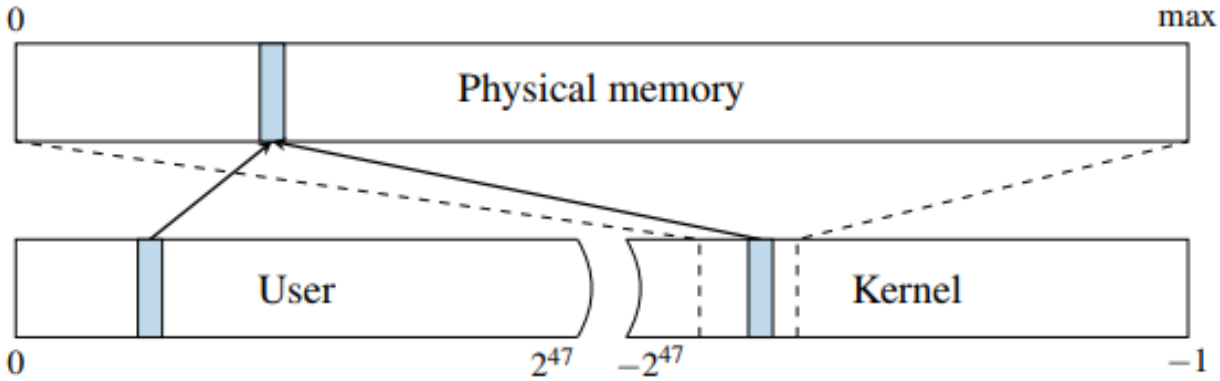
These changes (change of cache) are than recovered on an architectural level.



Source: Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg. "Meltdown: Reading Kernel Memory from User Space"

Reading Kernel Memory == Full Memory

- The kernel is mapped into every process virtual memory address space at an fixed address
- The kernel mapps the full physical memory within its "private" space.
  - Allows the kernel to access virtual memory from other processes
  - Remember: switching Page Tables is expensive

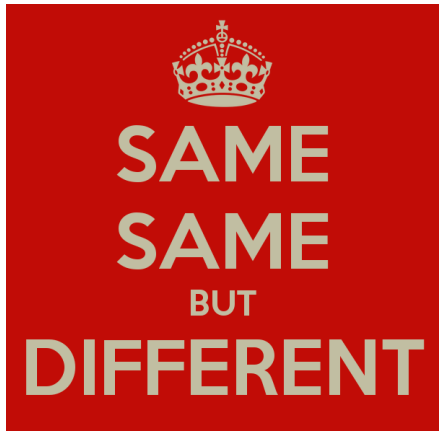


Meltdown Defenses
(KAISER, KPTI)

- Don’t have the full kernel and the full physical memory mapped into every virtual memory space
  - Kernel Page Table Isolation (KPTI)
  - Only a “small” facade-kernel is mapped
  - Allows execution transfer between user space and kernel
- Needs to remap memory everytime control is transferred between user mode and kernel
  - but changing page tables is expensive!
  - both page table sets needs to be held in sync

Spectre

- Builds on branch prediction, not exceptions
  - 1) train the predictor for a specific code branch
  - 2) at the end, let the predictor execute an illegal access
- Target: not accessing system memory
  but memory of the own process that it does not want to share
- exploitable also via Javascript
- breaks all kinds of sandboxes
- Multiple variants



Spectre (Basic Variant)

Javascript Example: some junk added, to disable JIT optimizations

```
1 if (index < simpleByteArray.length) {
2   index = simpleByteArray[index | 0];
3   index = (((index * 4096)|0) & (32*1024*1024-1))|0;
4   localJunk ^= probeTable[index|0]|0;
5 }
```

let index be 1, 2, 3, 4, 5, 6, 7, 8, 20000 (out of bound)

- first 8 calls, train the predictor to jump into the if statement -- will execute it speculatively
- 9th call access probeTable with illegal obtained value μArchitecturally.

Use cache timing side channel to find out, which.

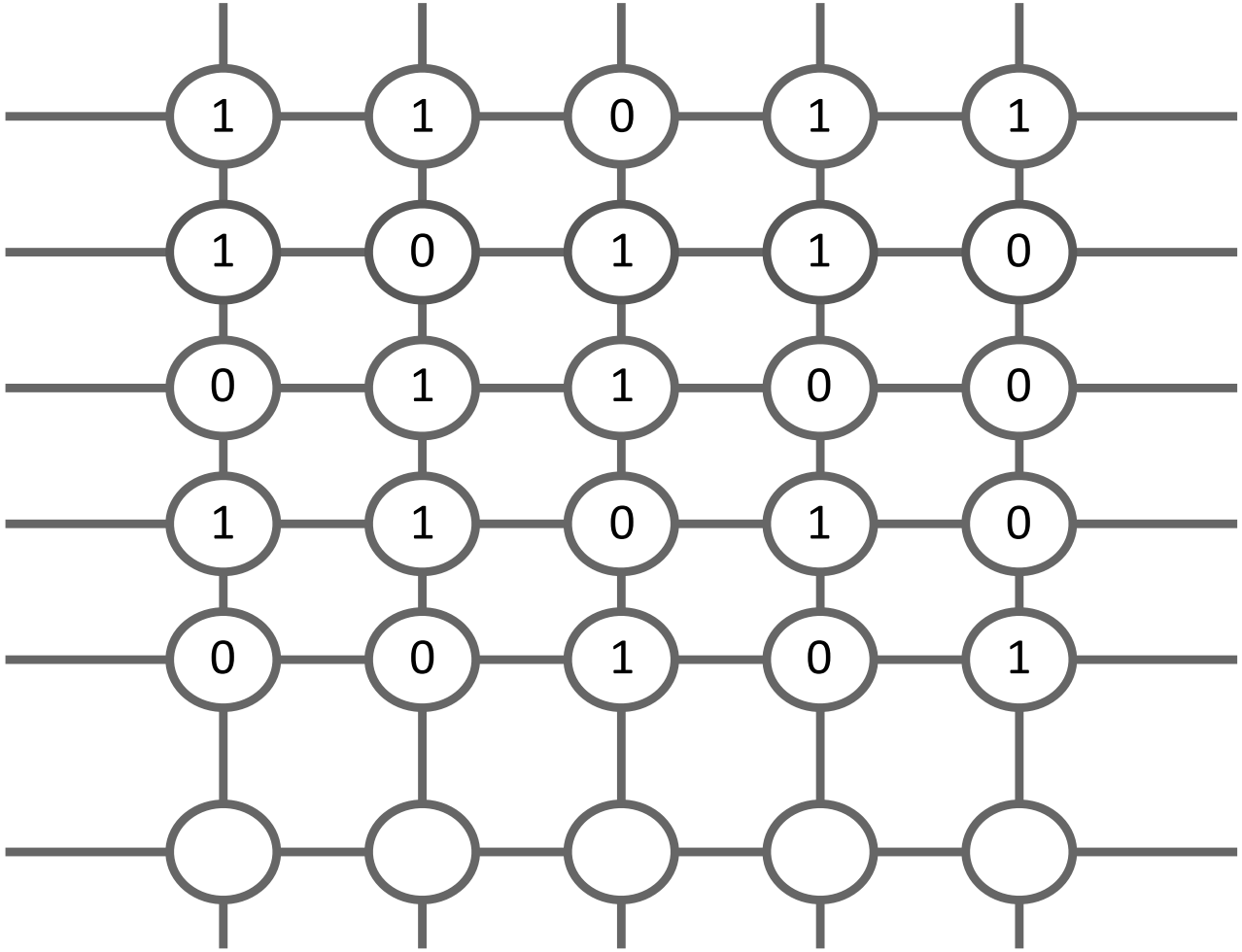
Rowhammer

Rowhammer?

"It's like breaking into an apartment by repeatedly slamming a neighbor's door until the vibrations open the door you were after." -- Vice Motherboard

- Originally thought to be just a reliability issue
- Hardware bug that allows attacker to exploit a system *without* relying on any software vulnerability
- Disturbance error in DRAM chips
- Modify memory *without* accessing it
- Widespread issue:
  ~85% of DDR3 [Kim et. al ICSA 2014]

DRAM Basics: Array of Memory Cells (Capacitors)

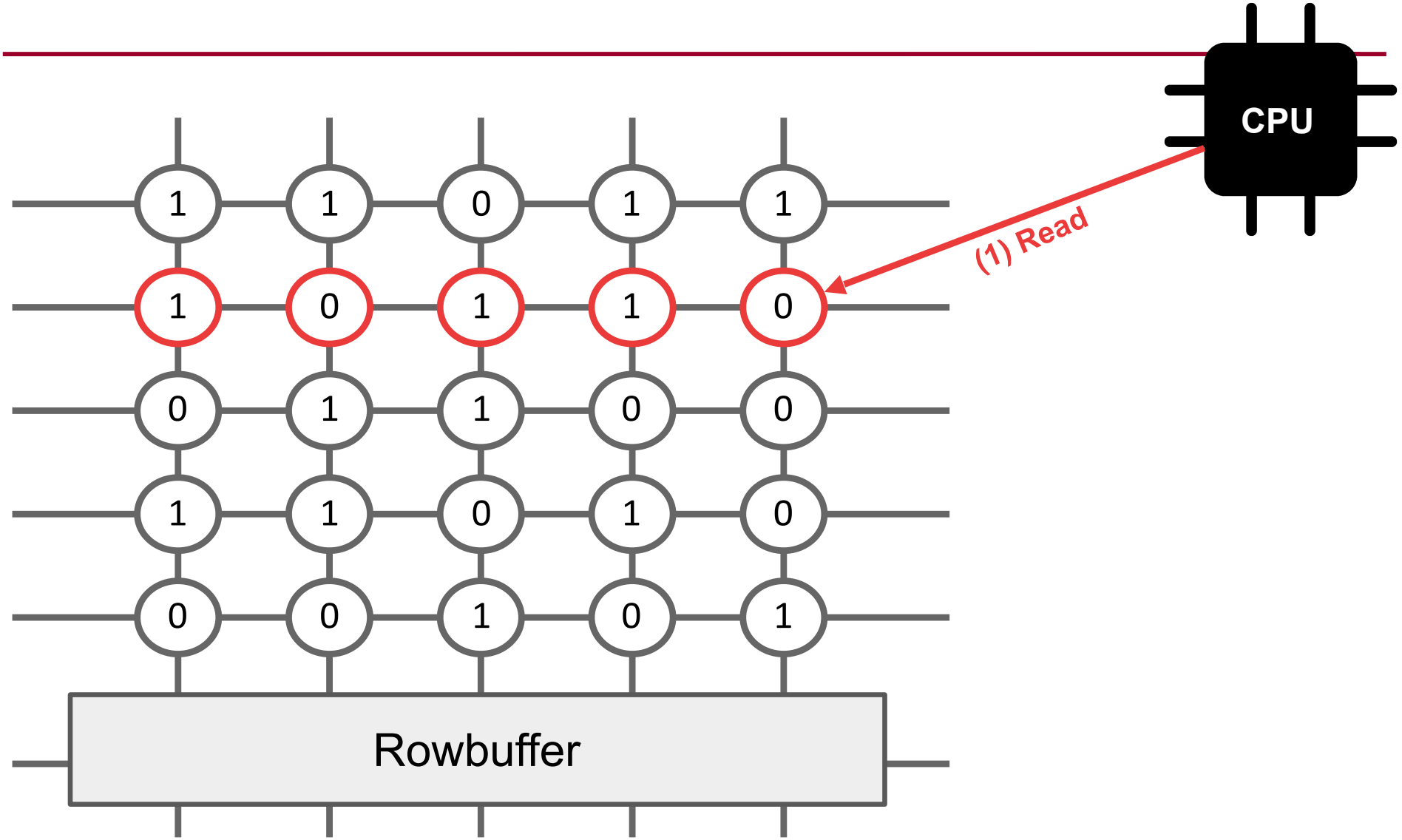


DRAM Basics: Data Storage

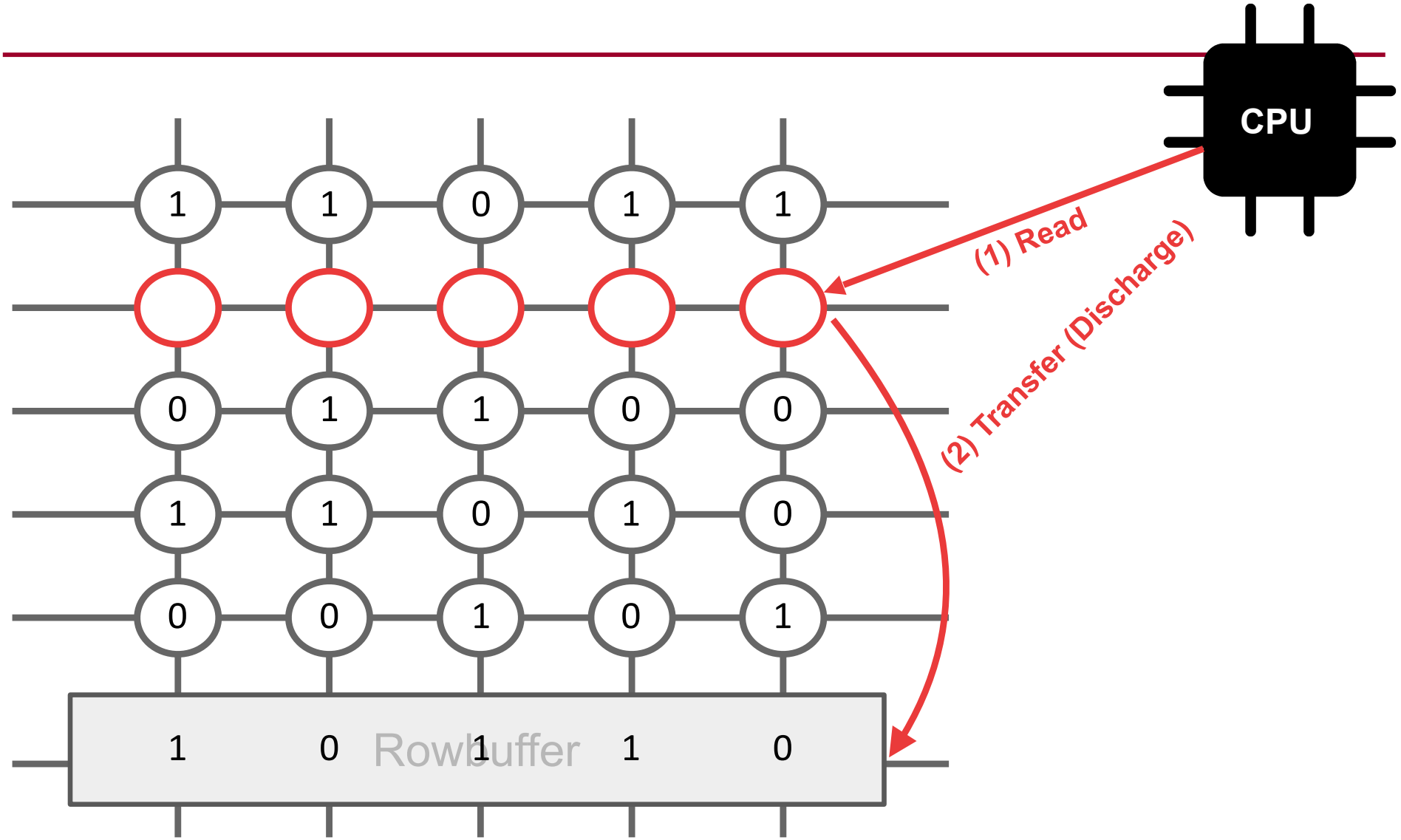
- Memory cell is charged to represent 1, otherwise 0
  (or the other way around)
- Memory cells leak charge and lose their state over time
  → Need to be periodically refreshed (e.g., every 64ms)
- Every access (activation) leaks charge to adjacent cells
  → If enough charge leaks, bits might flip from 1 > 0 or 0 > 1

Rowhammer = Race against the memory's refresh interval

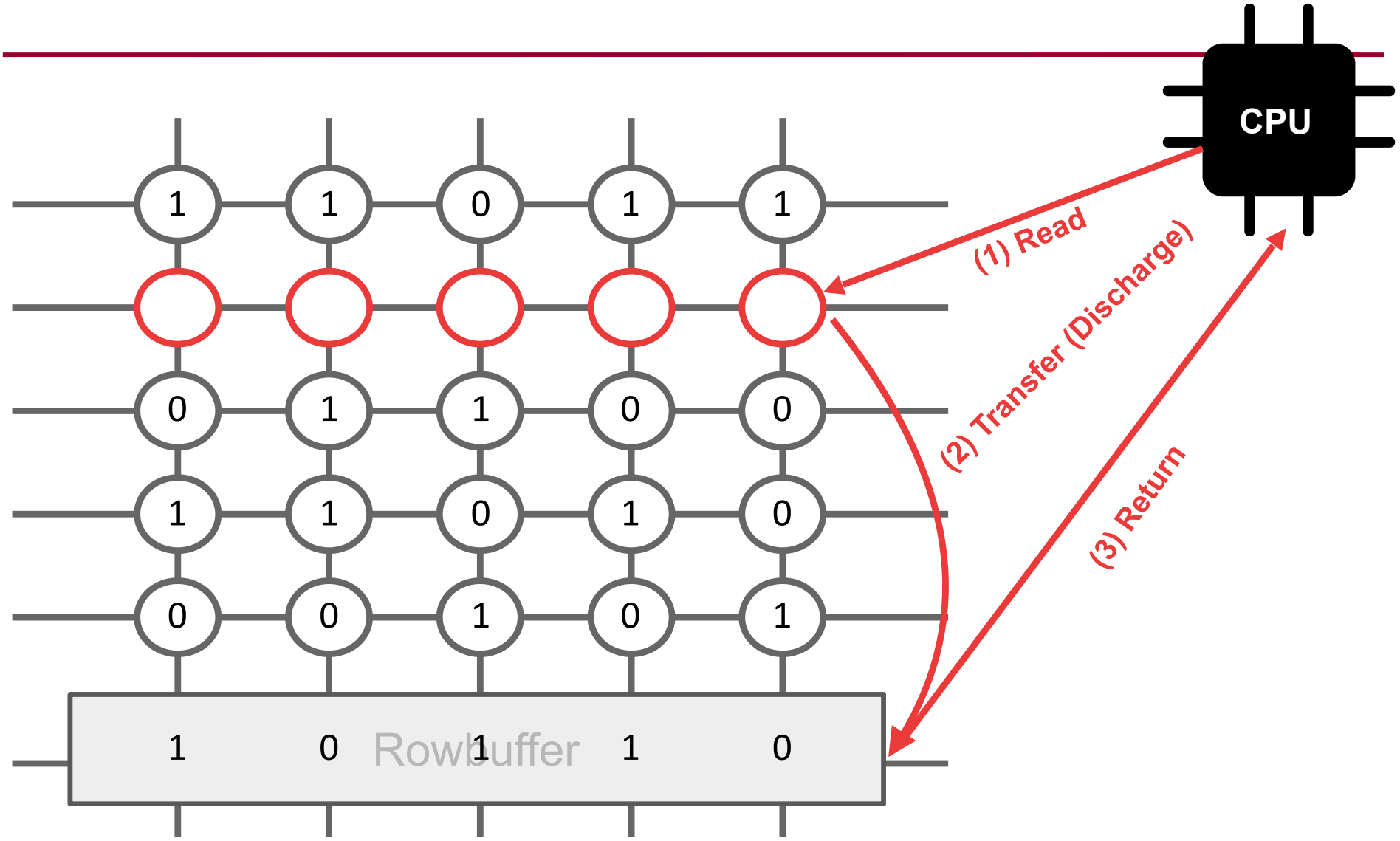
DRAM Basics: Memory Access



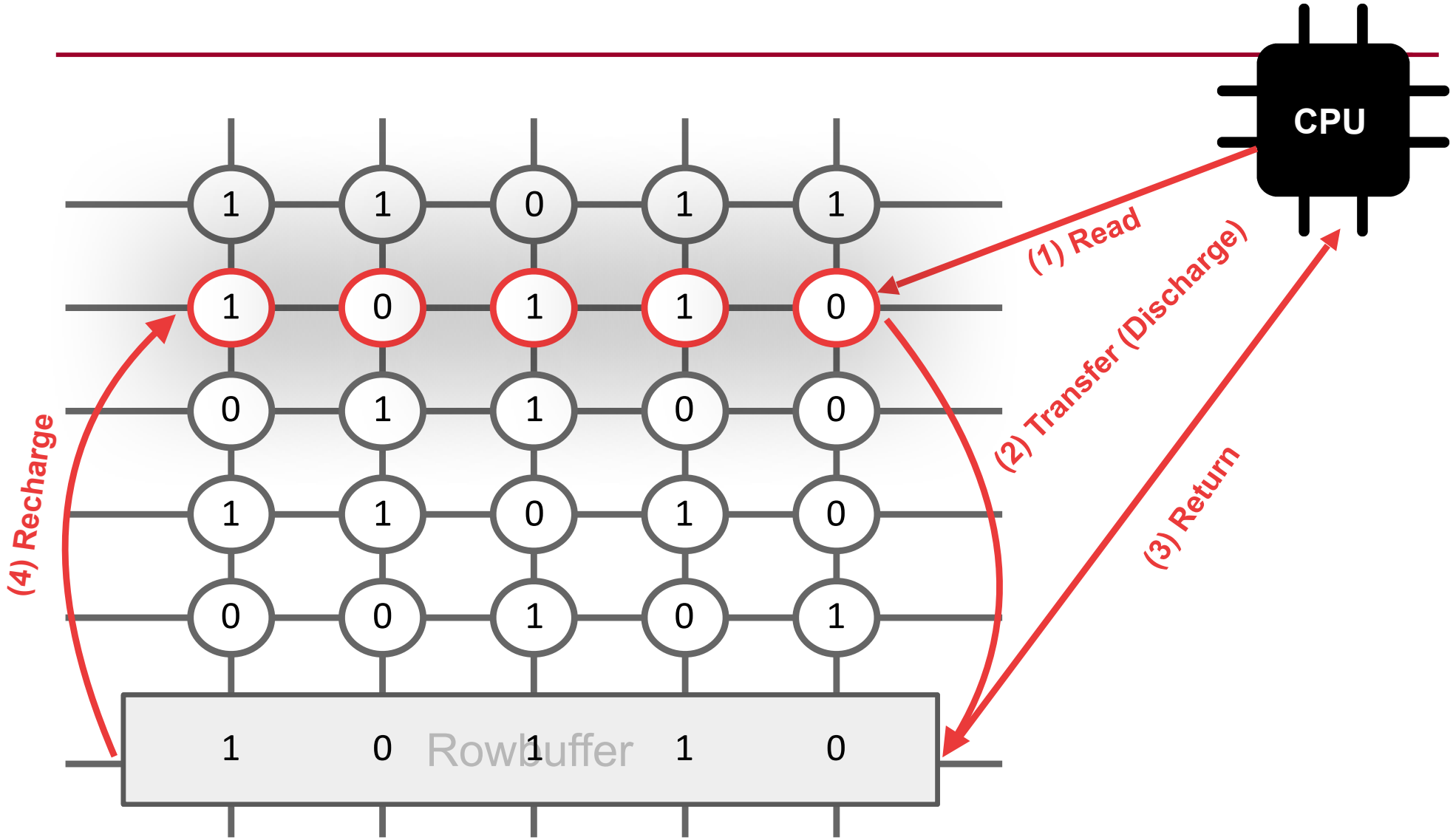
DRAM Basics: Memory Access



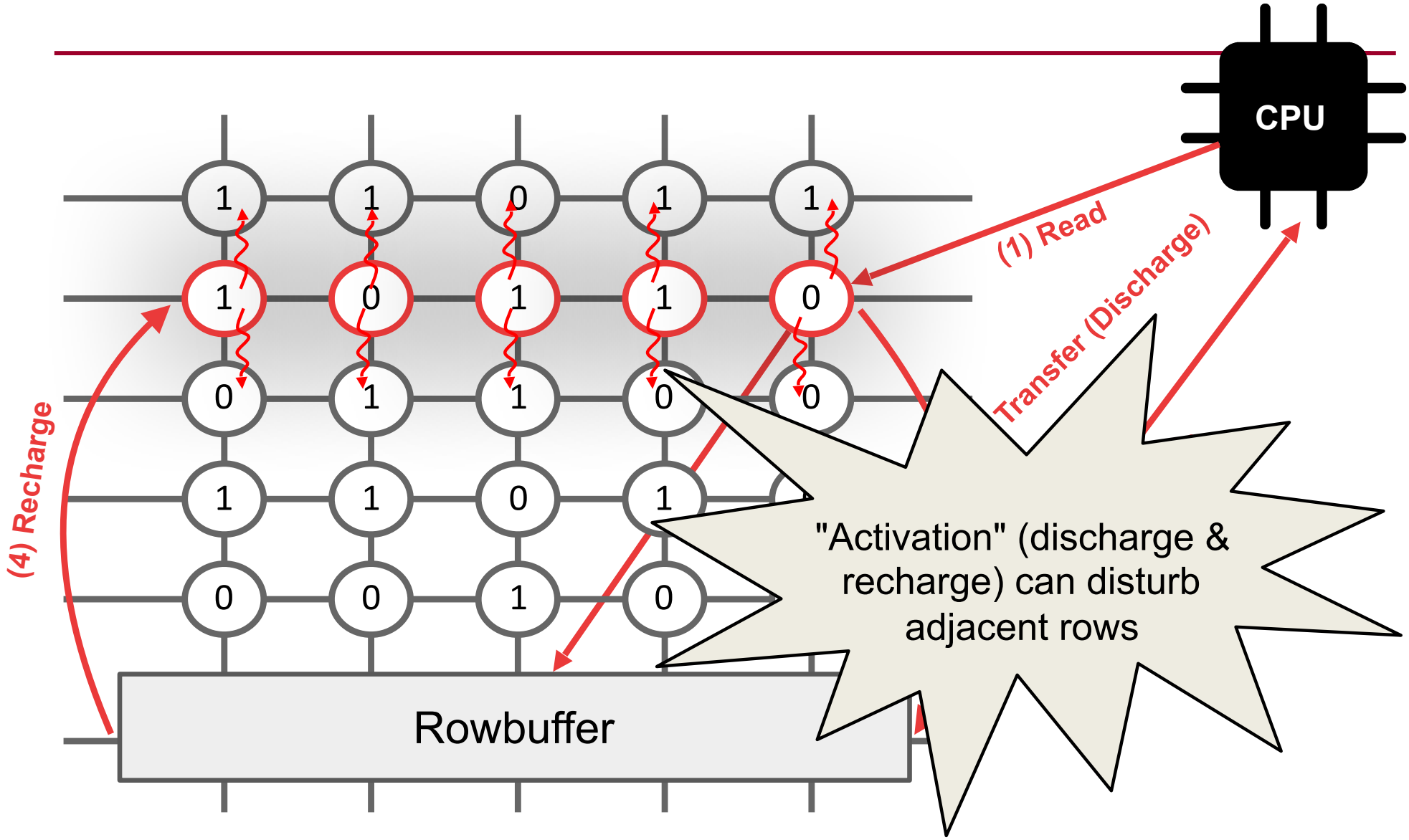
DRAM Basics: Memory Access



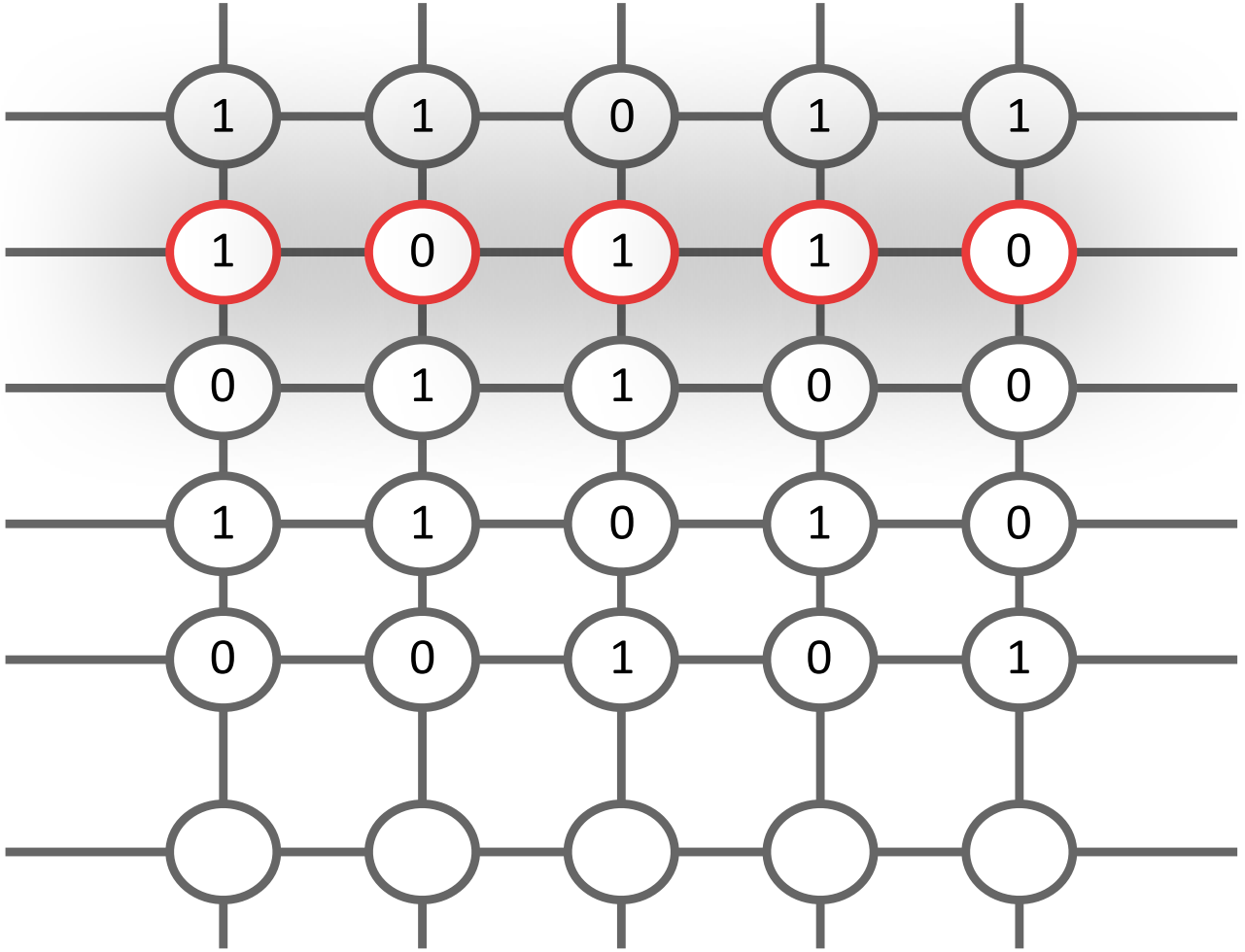
DRAM Basics: Memory Access



DRAM Basics: Memory Access



Rowhammer Attack Principle

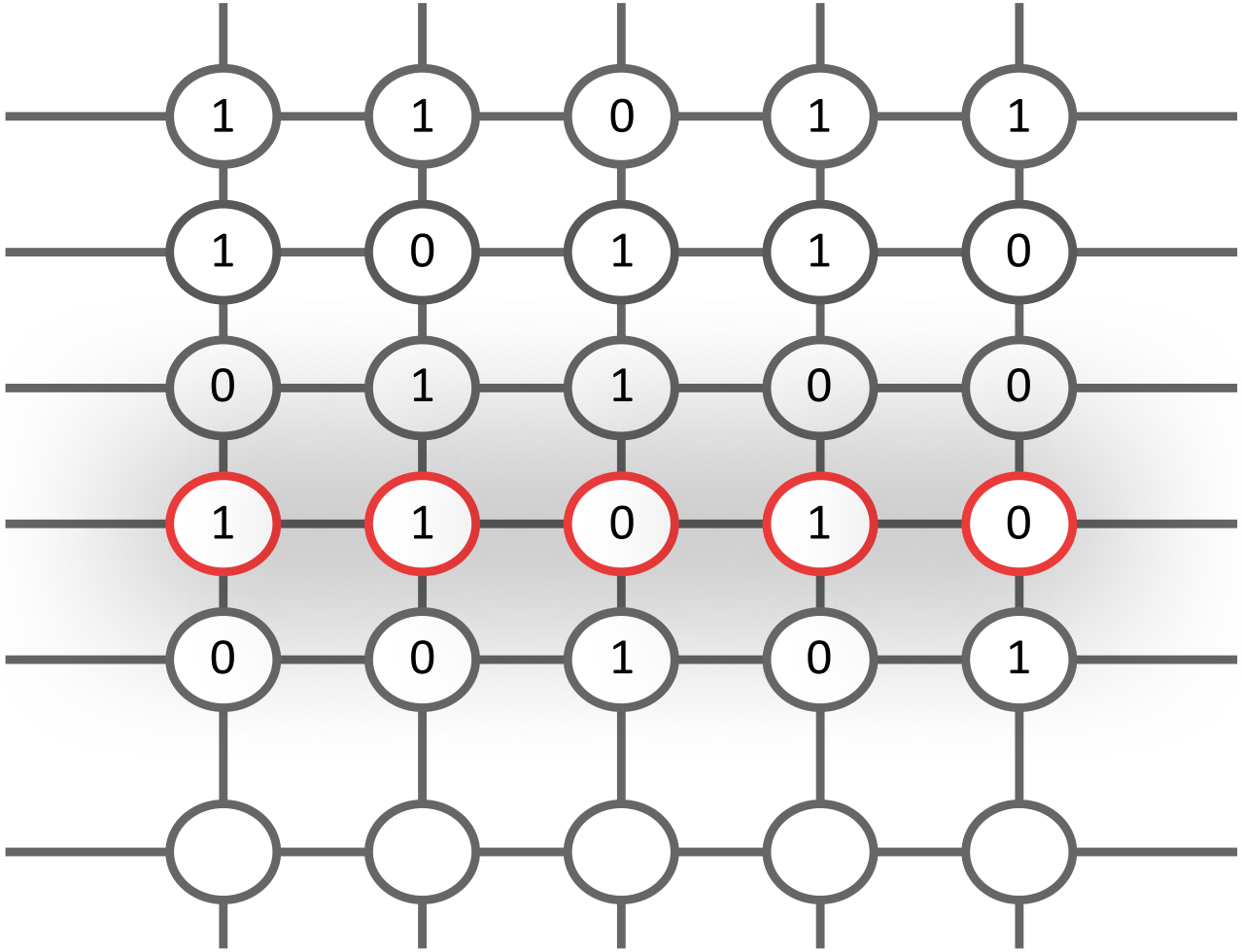


Aggressor Row A

Victim Row

Aggressor Row B

Rowhammer Attack Principle

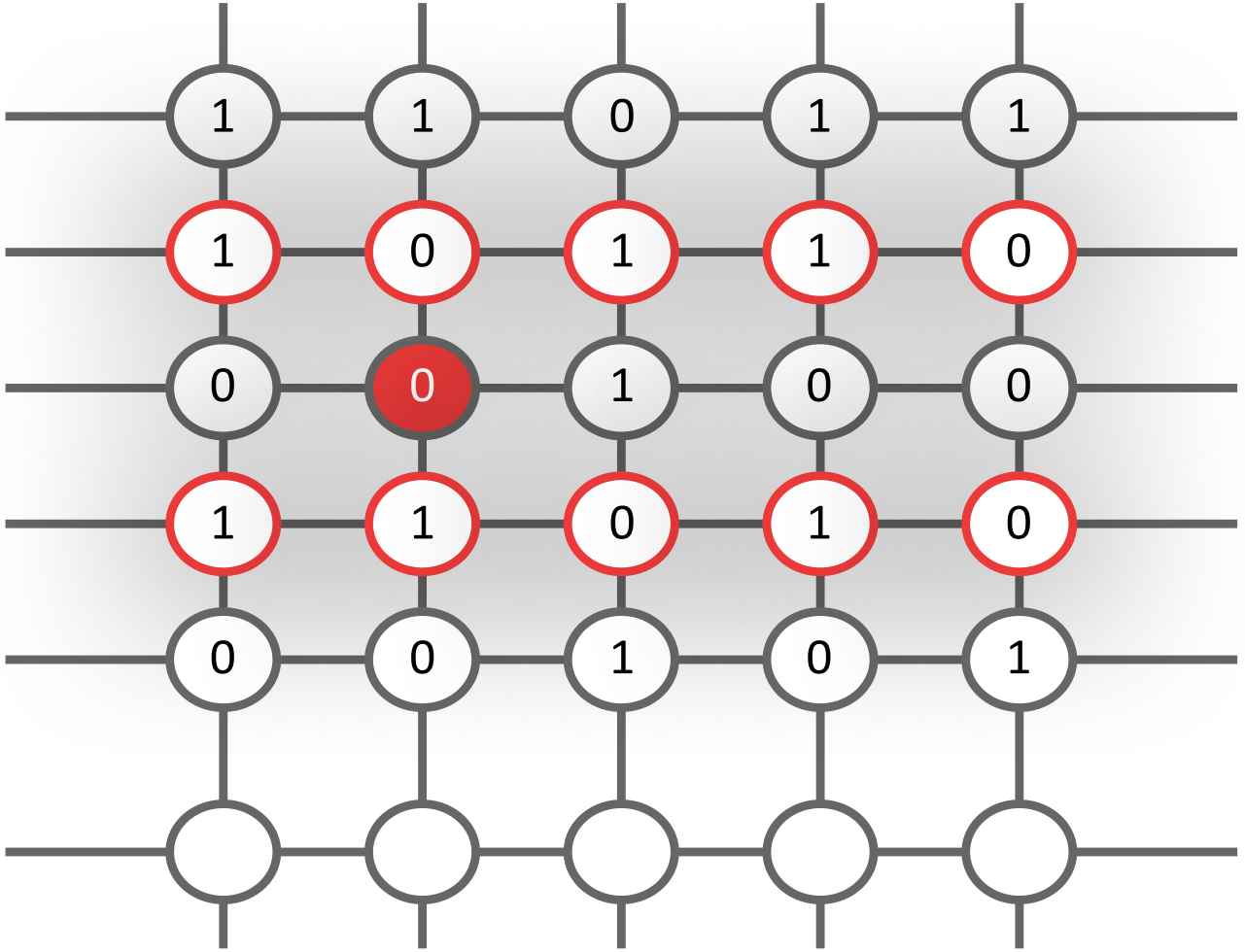


Aggressor Row A

Victim Row

Aggressor Row B

Rowhammer Attack Principle



Aggressor Row A

Victim Row

Aggressor Row B

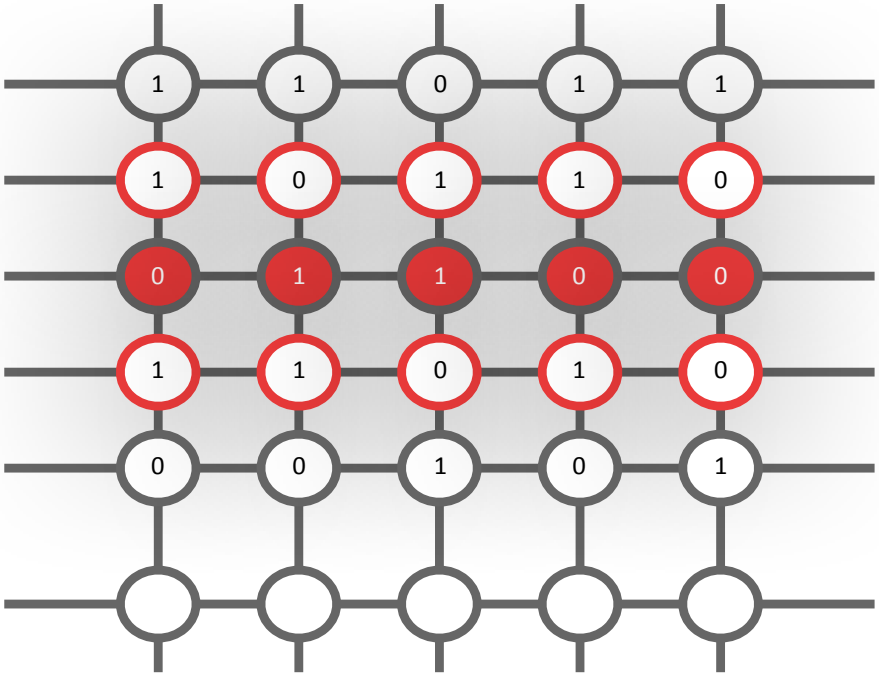
Double- vs. Single-sided Hammering

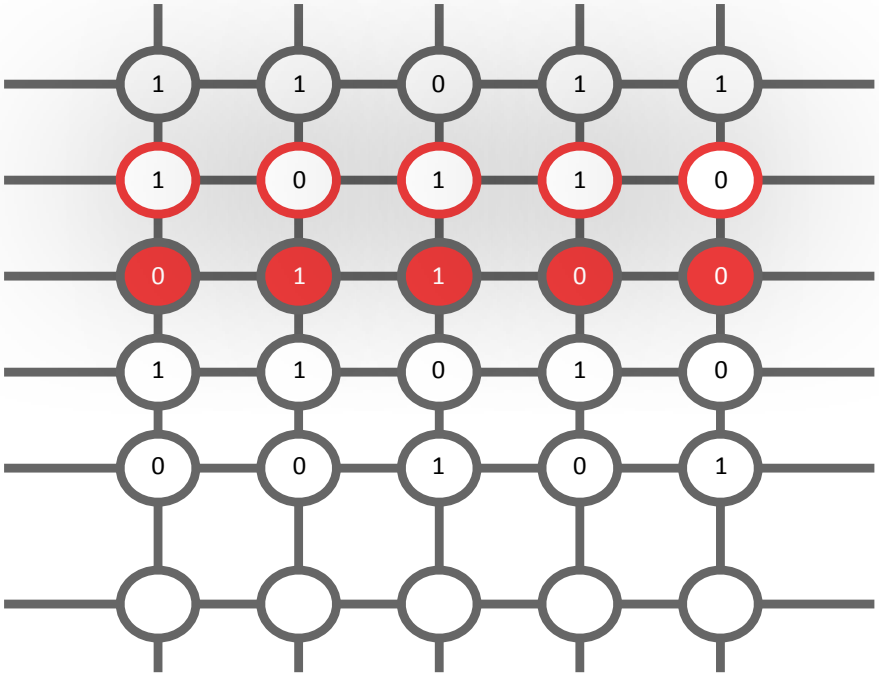
Double-sided Hammering

- Two aggressor rows
- One above & one below the victim row

Single-sided Hammering

- Only one aggressor row
- Either above or below the victim row
- Less efficient than double-sided





Rowhammer Attack Stages

Not every memory cell/row is vulnerable, but bit flips are largely reproducible: once a bit flips we can likely flip it again

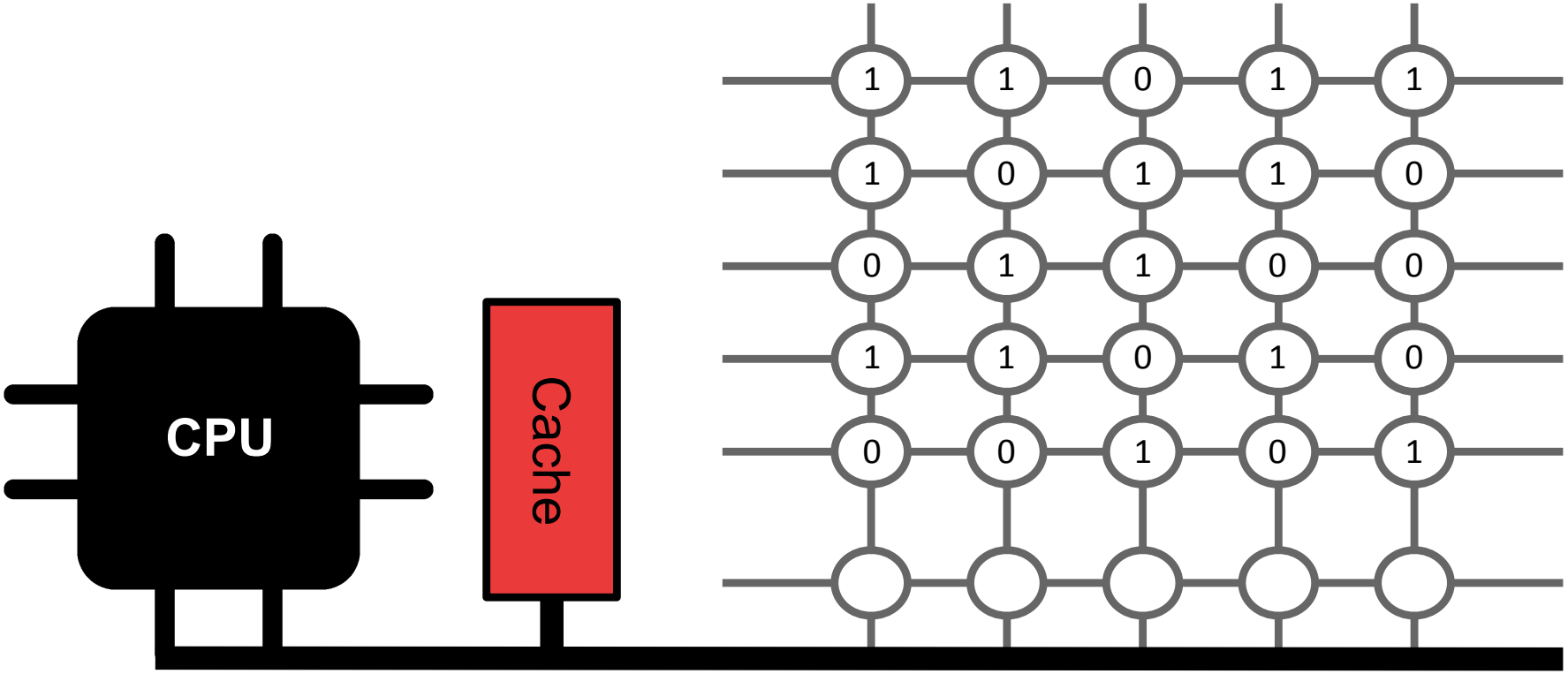
- **Stage 1: Reconnaissance (aka Memory Templating)**
  Scan memory for vulnerable locations

- **Stage 2: Land sensitive data (aka Memory Massaging)**
  Trick the victim (e.g., the OS) to place security-sensitive information in vulnerable location

- **Stage 3: Attack (aka Hammertime)**
  Reproduce the bit flip to modify the targeted data structure

Rowhammer Attack in Practice

- DRAM accesses are slow
- Repeated memory accesses are serviced by the cache instead



Basic Rowhammer Loop

```
loop:
mov (A), %eax   // Read from address A
mov (B), %ebx   // Read from address B
clflush (A)     // Flush cache for address A
clflush (B)     // Flush cache for address B
jmp loop
```

- First observed by Kim et al. in 2014 [ICSA 2014]
- First exploit by Google Project Zero in 2015 [BH USA 2015] https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html

Rowhammer Attack Primitives

1. **Circumventing the cache**
   → Attacker needs to directly access the physical memory each time

2. **Determining physical address of aggressor & victim rows**
   → Attacker needs to find adjacent rows in physical memory

3. **Memory massaging**
   → Attacker needs to control the physical location of the target data

Circumventing the Cache (1/2)

Strategy 1: Evict memory from cache after each access

- Explicitly remove data from cache
  - Cache flush instruction (`clflush`) or system call
  - Like Flush+Reload

- Build a cache eviction set
  - Implicitly removes data from the cache by reading more data until the cache is full
  - Like Prime+Probe

Circumventing the Cache (2/2)

Strategy 2: Use uncached memory

- Non-temporal memory access instructions
  - Tell the CPU there's no need to cache anything because data is only accessed once
  - e.g., `MOVNTI, MOVNTDQA`

- Direct Memory Access (DMA)
  - Sometimes caches are detrimental when offload tasks to dedicated hardware (devices for graphics/audio/…)
  - Simple devices might not be able to translate virtual to physical address mappings (i.e., must also provide contiguous memory)
  - On Android provided through ION memory allocator

Finding the Aggressor & Victim Rows

- Every process only sees virtual address space
- How to select the addresses to read from, i.e., the aggressor rows surrounding a victim row?

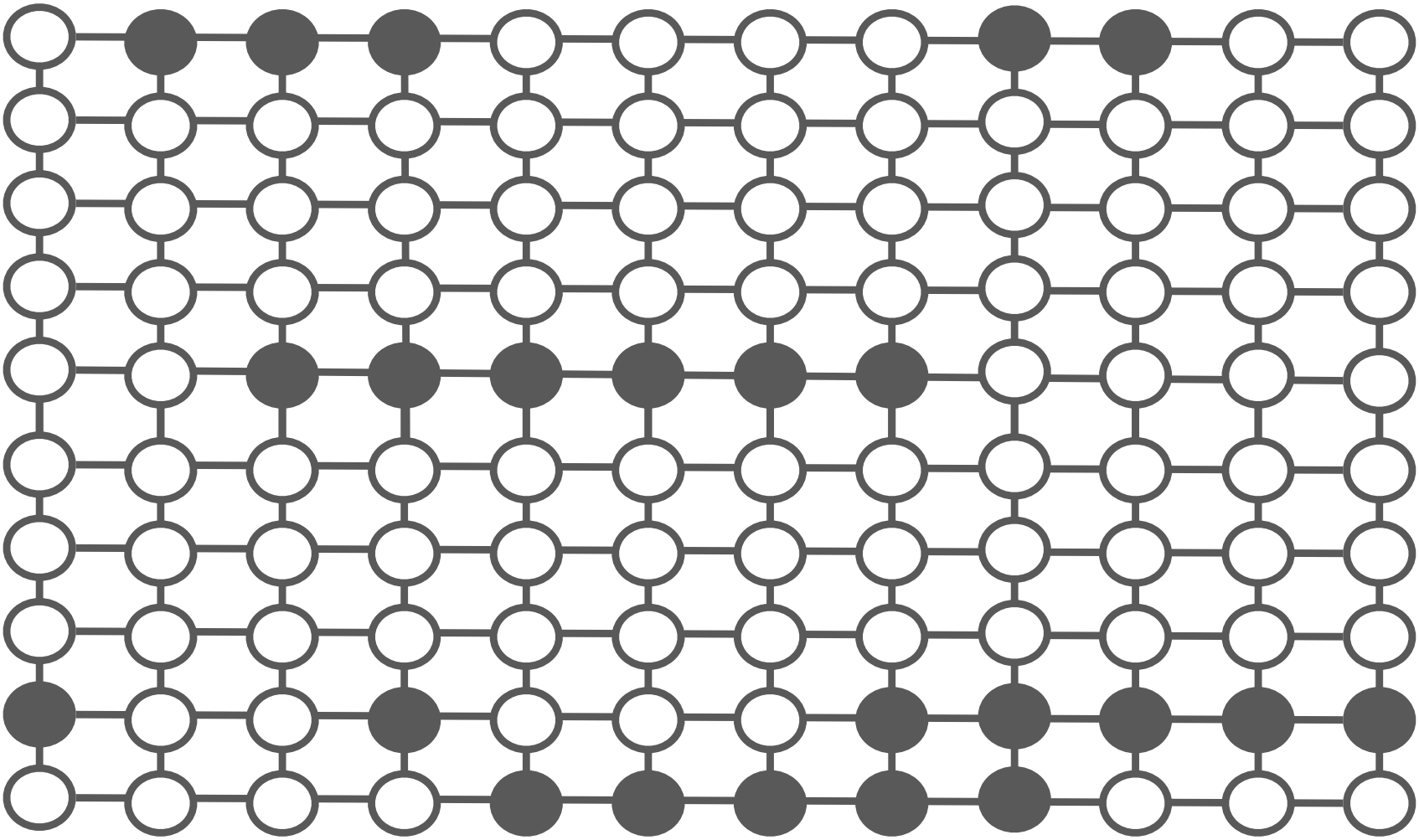
- Use `/proc/self/pagemap`
  - Special file that stores translation of virtual to physical addresses
  - Newer Linux versions disable user space access as a countermeasure

- Use specialized memory
  - Huge Pages: Linux provides 2MB of contiguous memory
  - Again, DMA to the rescue: Android ION provides flag to request physically contiguous memory

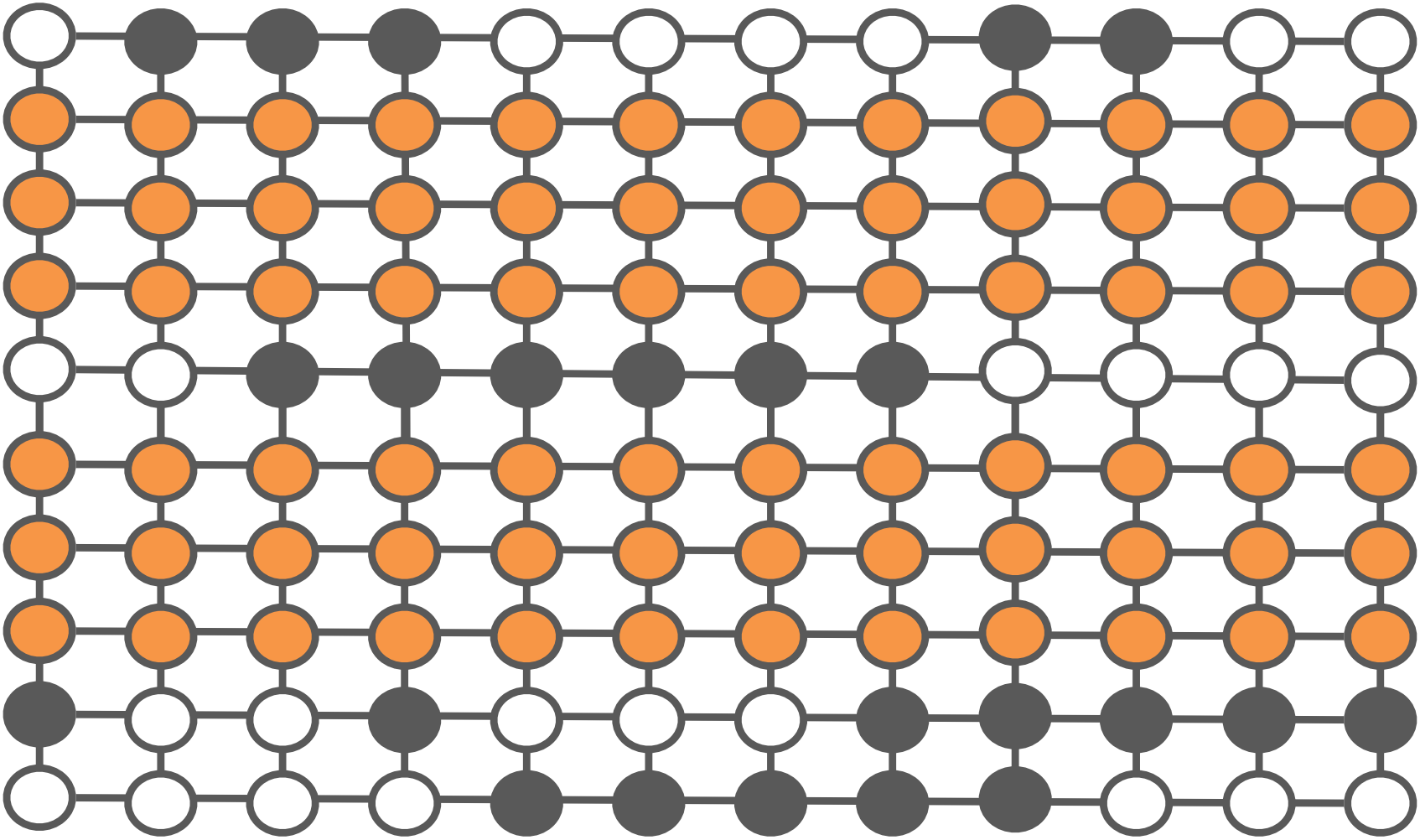
Landing Sensitive Data ("Flip Feng Shui")

- Probabilistic exploits
  - Spray memory with data you want to attack, e.g., page tables, and hope for the best (original Project Zero attack)
- Deterministic exploits
  - Rely on special memory management features, e.g., memory deduplication, MMU paravirtualization
  - Alternative: exploit predictable behavior of memory allocators
  - They are optimized for performance and to minimize memory fragmentation
  - Attacker can force the OS to place sensitive data in a vulnerable location by targeted allocations and deallocations

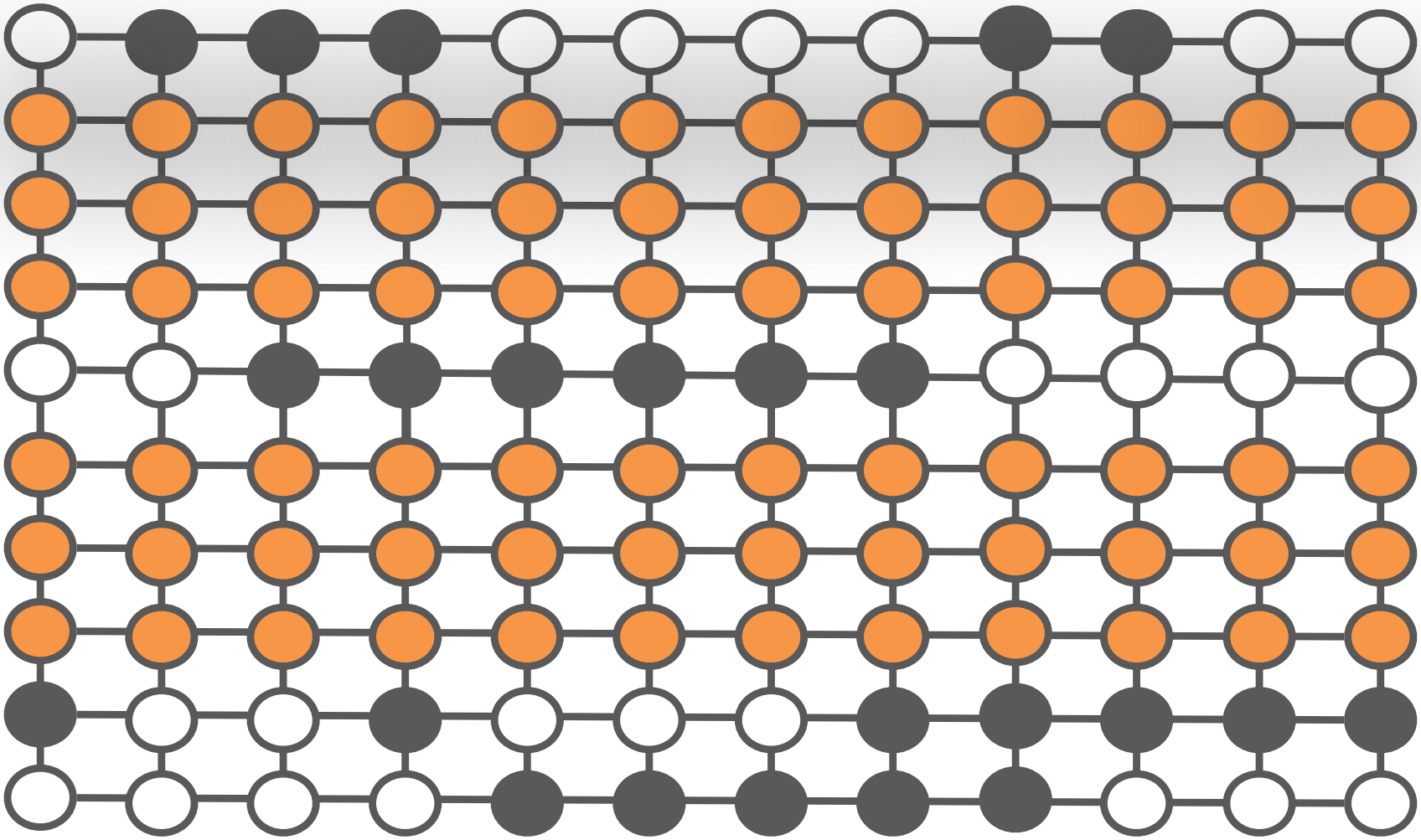
Deterministic Attack: Free and Used Memory Before the Attack



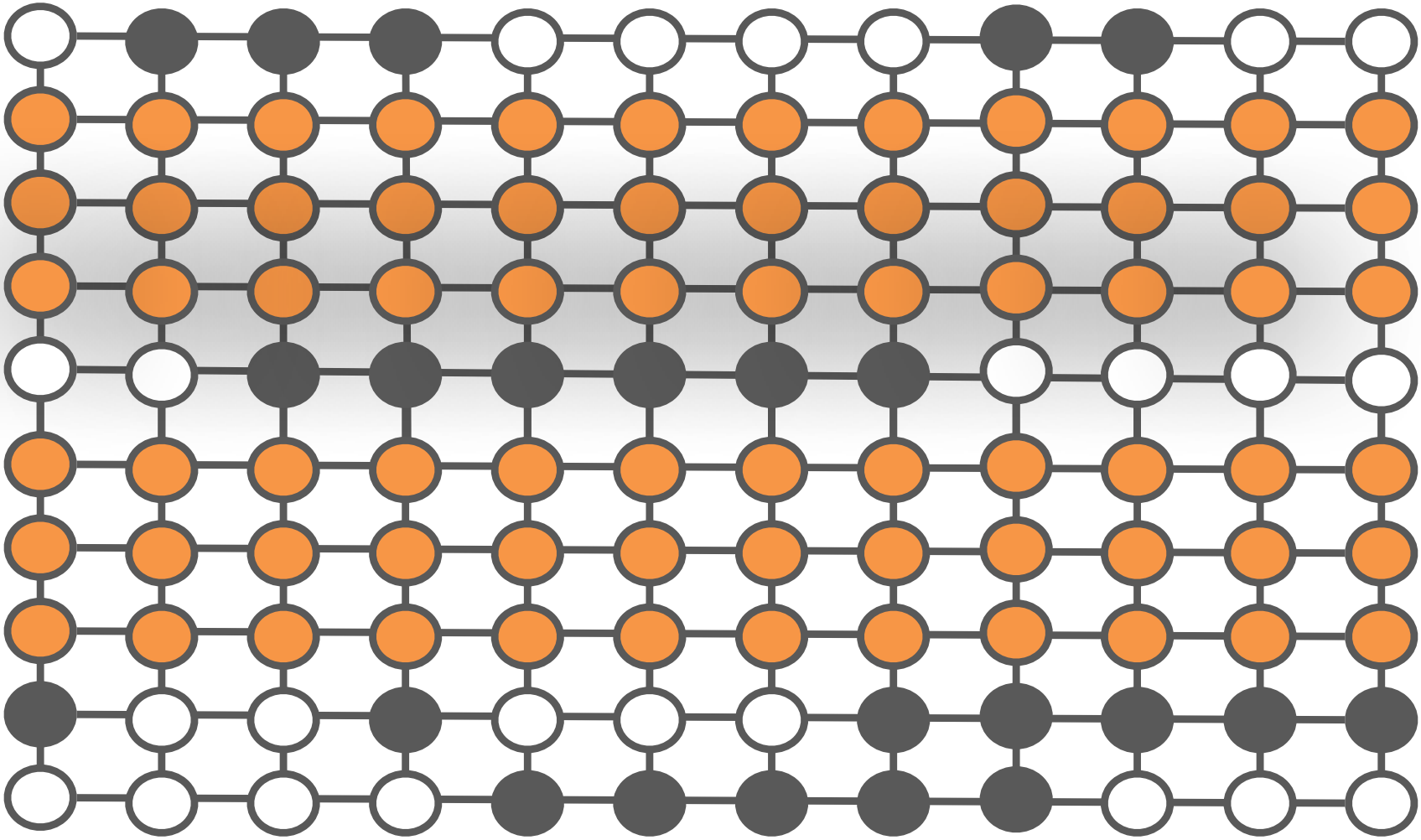
Deterministic Attack:
(1) Allocate Memory for Templating



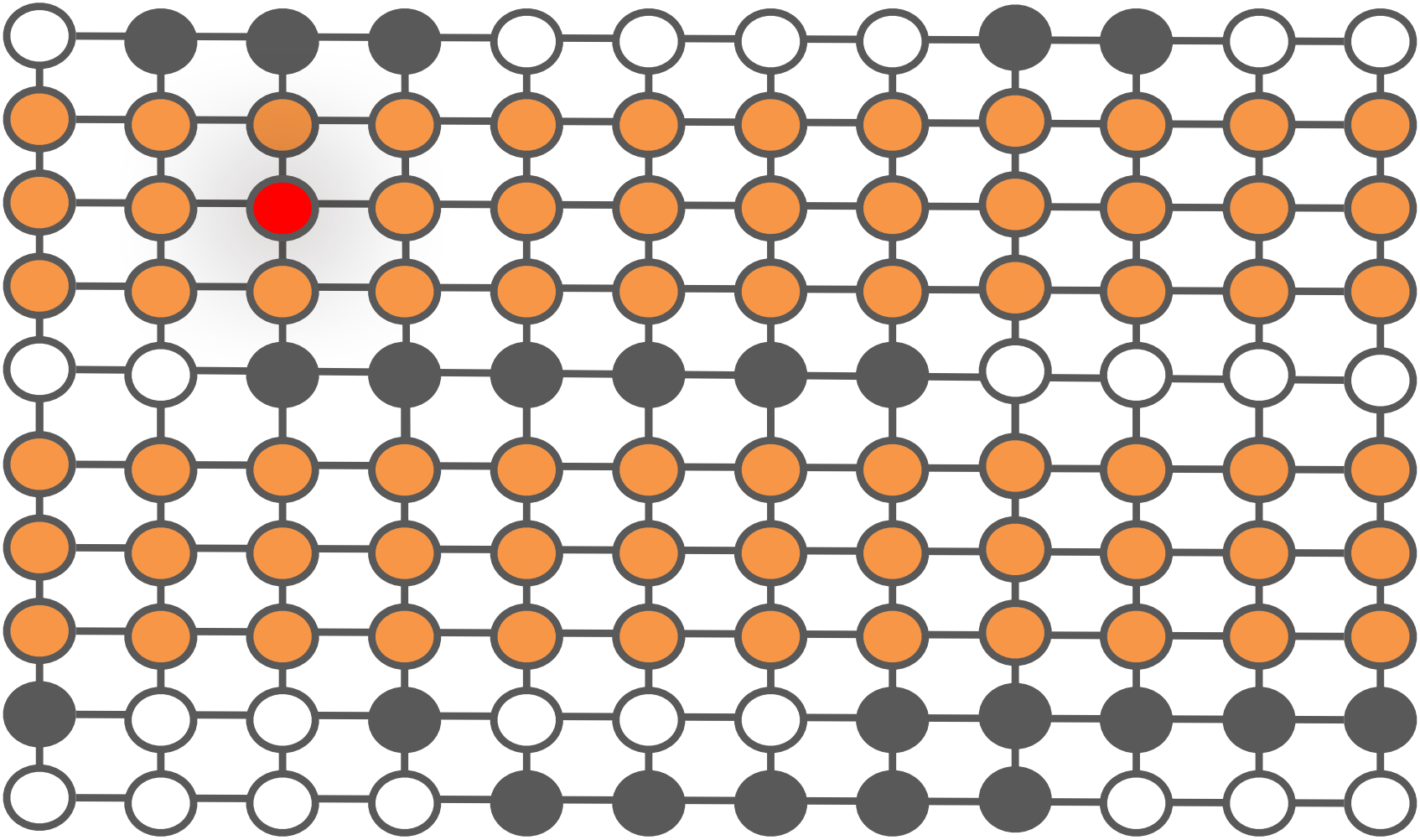
Deterministic Attack:
(2) Scan Memory for Bit Flips



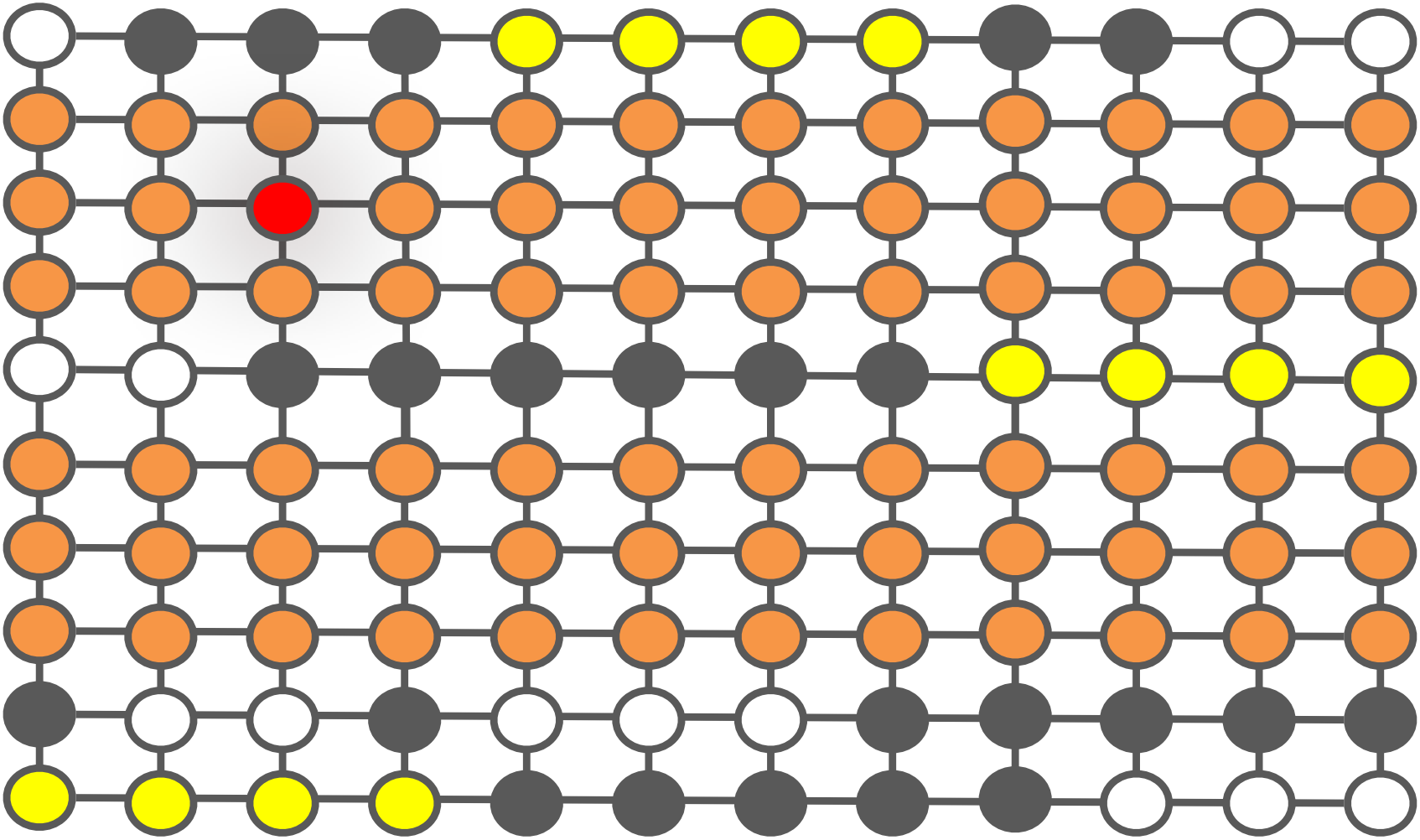
Deterministic Attack:
(2) Scan Memory for Bit Flips



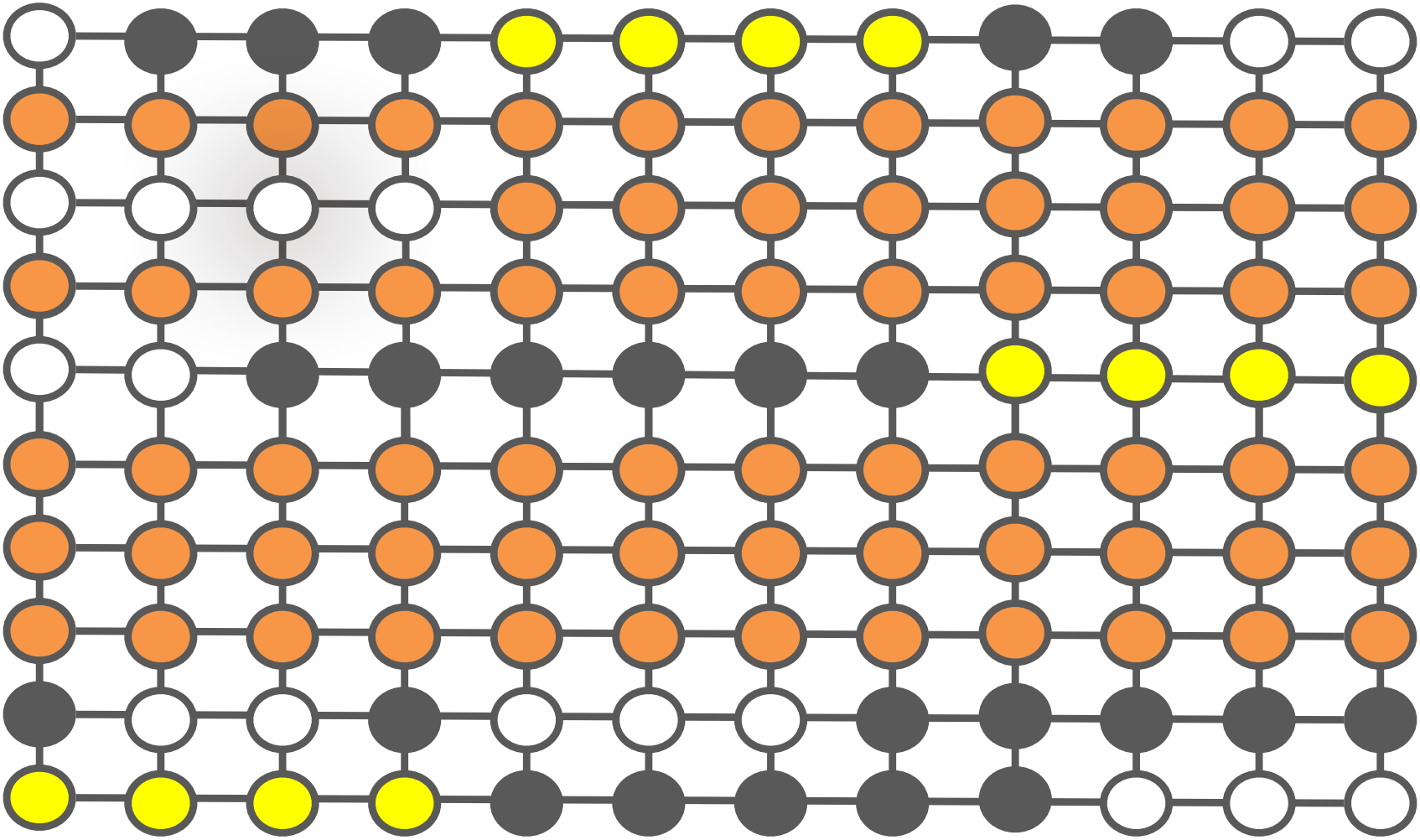
Deterministic Attack:
(2) Scan Memory for Bit Flips (Success!)



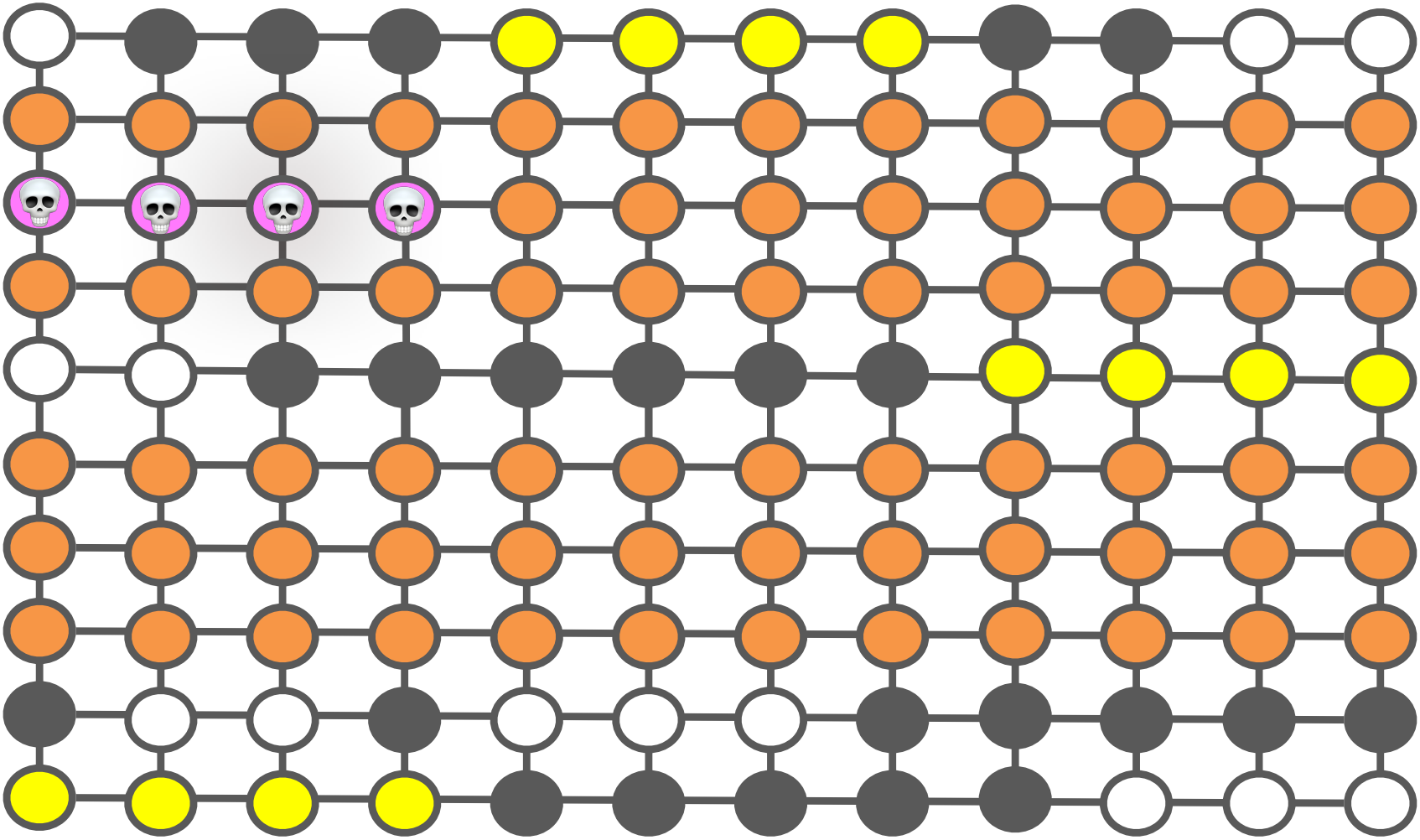
Deterministic Attack:
(3) Fill up Remaining Memory



Deterministic Attack:
(4) Release Vulnerable Memory



Deterministic Attack:
(5) Trigger Kernel to Allocate Data



What could possibly go wrong?

Attack Targets

- Privilege escalation
  - Bit flips in page table entries (PTEs) can give the attacking process R/W access to its own page table
    → R/W access to physical memory
  - Gain root privileges from user space by modifying credential structures (e.g., `struct cred`)
  - Break out of the browser sandbox
  - Flip instructions in a program to bypass authentication (e.g., `sudo, sshd`)

More Attack Targets

- Break cryptographic keys
  - Public RSA key from `.ssh/authorized keys`
  - Flipping a bit changes the public/private key pair
    `ssh-rsa AAAAB3NzaC1yc2EAAAADAQABl8h0VfRbC7naVs…`
    `ssh-rsa AAAAB4NzaC1yc2EAAAADAQABl8h0VfRbC7naVs...`
  - New public key is easy to factorize
    (i.e., attacker can derive the private key)

- Domain names
  - Ubuntu repository for apt-get upgrade
  - `security.ubuntu.com` → `security.ubunvu.com`
  - Packages are signed, also need to flip bit in the GPG keychain (`/etc/apt/trusted.gpg`)

Demonstrated Attacks ...

- … in the browser: flipping bits from JavaScript
  - Dedup Est Machina [S&P 2016], Rowhammer.js [DIMVA 2016]
- … in the cloud: flipping bits in another virtual machine (VM)
  - Flip Feng Shui [USENIX Sec 2016], Cloud Flops [USENIX Sec 2016]
- … on phones: flipping bits from an app (without any permissions)
  - Drammer [CCS 2016]
- … on phones via the browser: flipping bits through the GPU
  - Grand Pwning Unit aka GLitch [S&P 2018]
- … over the network: flipping bits remotely
  - ThrowHammer [USENIX ATC 2018], NetHammer [arXiv 2018]

Rowhammer Tests & Source

Tools to test whether your device is vulnerable …
(at your own risk)

- Desktop (x86-64)
  https://github.com/google/rowhammer-test

- Mobile Devices (ARM)
  https://github.com/vusec/drammer

Defenses: ideally in Hardware

- First response: increase the memory refresh rate
  - e.g., 64ms → 32ms
  - Costs performance and battery
  - Not effective against all types of attacks
- Since LPDDR4: Target Row Refresh (TRR)
  - Preemptively refresh rows that exceed a maximum activation count
  - Needs to be supported by the memory (controller)
- Error-correcting code (ECC) memory
  - Not designed to defend against Rowhammer
  - Raises an alarm if bit flips reach a certain threshold
  - Only defends against single bit flips in a row

Defenses in Software

Replacing hardware is not always feasible, what about legacy devices?

- Disabling cache flush instructions
  - e.g., in the browser sandbox
  - But there are numerous other ways to circumvent the cache
- Disabling DMA APIs
  - Google disabled contiguous memory through Android ION in response to Drammer
  - Still vulnerable to updated attacks (RAMpage [DIMVA 2018])

Defenses in Software (ongoing Research Topic)

- Memory separation and isolation
  - Blacklist vulnerable rows by disabling them at boot (B-CATT [arXiv 2016])
  - Strict separation of physical memory into security domains, e.g., kernel and user space (C-CATT [USENIX Sec 2017])
  - Guard rows between memory of different processes and security domains (GuardION [DIMVA 2018], ZebRAM [OSDI 2018])

- Hardware performance counters
  - e.g., Intel Performance Counter Monitor (PMU)
  - On Linux accessible through `perf` for system profiling
  - Detect suspicious peaks in the number of cache misses
  - Locality of memory accesses (ANVIL [ASPLOS 2016])

Conclusion

- Software security relies on hardware security
- … but can we trust the hardware?

- Side-channels and rowhammer show how this trust-relationship is broken
  - Even if we could build the perfect bug-free software…
  - … it can still be compromised through side-channels and bit flips

- Hard to defend against these types of attacks in software

Advertisement

- Praktika, Bachelor & Master theses on applied security & privacy topics
- Also looking for potential PhD students
- Topics include….

- malware analysis, large-scale mobile app analysis for privacy leaks, and rowhammering

martina@iseclab.org

- infrastructure, radio networks, LTE, Wi-Fi, power grids, internet protocols, secure execution environments

atrox@iseclab.org

Questions?

References

[Kim et. al ICSA 2014] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. *Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors*. International Symposium on Computer Architecture (ISCA), 2014.

[Project Zero, BH USA 2015] M. Seaborn and T. Dullien. *Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges*. Black Hat USA, 2015.

[ANVIL ASPLOS 2016] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin. *ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks*. ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2016.

[Dedup Est Machina S&P 2016] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida. *Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector*. IEEE Symposium on Security and Privacy (S&P), 2016.

[Rowhammer.js DIMVA 2016] D. Gruss, C. Maurice, and S. Mangard. *Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript*. Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), 2016.

[Flip Feng Shui USENIX Sec 2016] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos. *Flip Feng Shui: Hammering a Needle in the Software Stack*. USENIX Security Symposium, 2016.

References

[Cloud Flops USENIX Sec 2016] Y. Xiao, X. Zhang, Y. Zhang, and M.-R. Teodorescu. *One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation*. USENIX Security Symposium, 2016.

[Drammer CCS 2016] V. van der Veen, V., Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, C. Giuffrida. *Drammer: Deterministic Rowhammer Attacks on Mobile Platforms*. ACM Conference on Computer and Communications Security (CCS), 2016.

[B-CATT arXiv 2016] F. Brasser, L. Davi, D. Gens, C. Liebchen, A.R. Sadeghi. *CAn't Touch This: Practical and Generic Software-only Defenses Against Rowhammer Attacks*. arXiv:1611.08396, 2016.

[C-CATT USENIX Sec 2017] F. Brasser, L. Davi, D. Gens, C. Liebchen, A.R. Sadeghi. *CAn't Touch This: Practical and Generic Software-only Defenses Against Rowhammer Attacks*. USENIX Security Symposium, 2017.

[GLitch S&P 2018] P. Frigo, C. Giuffrida, H. Bos, K. Razavi. *Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU*. IEEE Symposium on Security and Privacy (S&P), 2018.

[GuardION & RAMpage DIMVA 2018] V. van der Veen, M. Lindorfer, Y. Fratantonio, H.P. Pil-Lai, G. Vigna, C. Kruegel, H. Bos, *GuardION: Practical mitigation of DMA-based Rowhammer attacks on ARM*. Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), 2018.

References

[ThrowHammer USENIX ATC 2018] A. Tatar, R. Konoth, E. Athanasopoulos, E. Giuffrida, H. Bos, K. Razavi. *Throwhammer: Rowhammer Attacks over the Network and Defenses*. USENIX Annual Technical Conference (ATC), 2018.

[NetHammer arXiv 2018] M. Lipp, M. Tadesse Aga, M. Schwarz, D. Gruss, C. Maurice, L. Raab, L. Lamster. *Nethammer: Inducing Rowhammer Faults through Network Requests*. arXiv:1805.04956, 2018.

[ZebRAM OSDI 2018] R. Konoth, M. Oliverio, A. Tatar, D. Andriesse, H. Bos, C. Giuffrida, K. Razavi. *ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks*. USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2018.

Yuval Yarom and Katrina Falkner. *FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack*. 2013

Anders Fogh. *Cache side channel attacks: CPU Design as a security problem*. Presentation 2016

Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg. *Meltdown: Reading Kernel Memory from User Space*. 2018