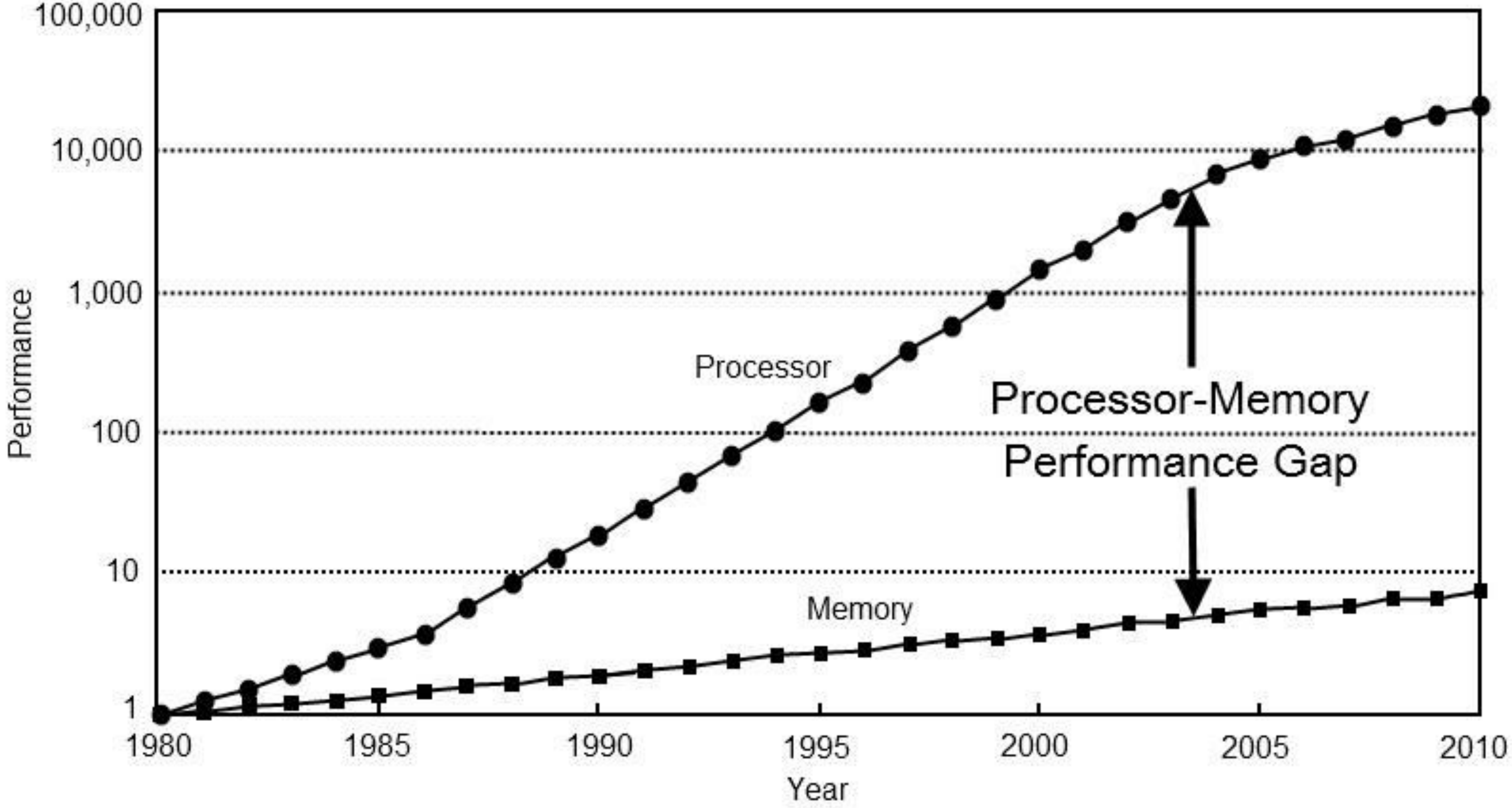
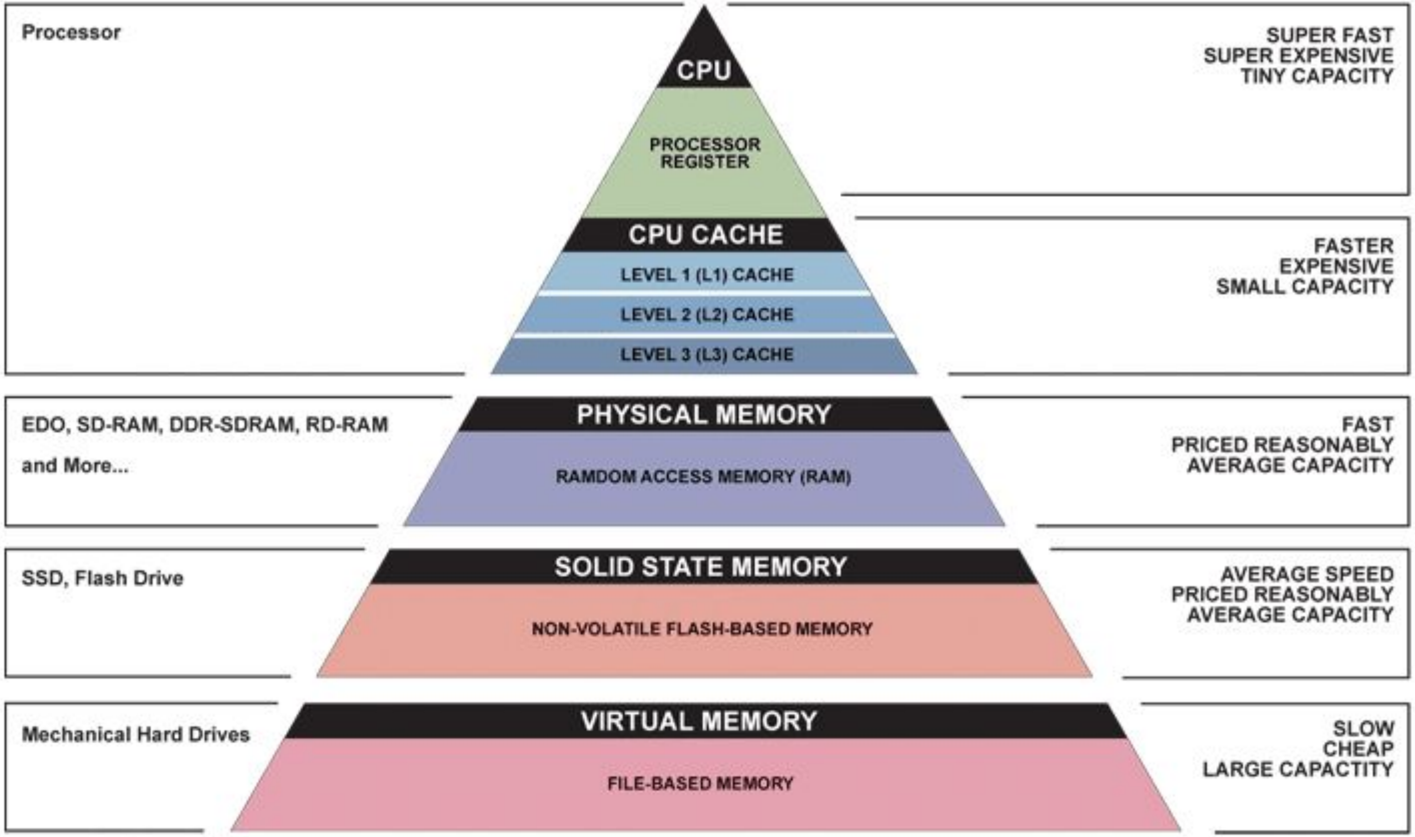
CASS: Exercise session 7

Caches and Microarchitectural Timing Attacks


100,000
10,000
1,000
100
10
1
Performance
Processor
Memory
Processor-Memory
Performance Gap
1980
1985
1990
1995
2000
2005
2010
Year



Processor

CPU

PROCESSOR REGISTER

SUPER FAST
SUPER EXPENSIVE
TINY CAPACITY

CPU CACHE

LEVEL 1 (L1) CACHE

LEVEL 2 (L2) CACHE

LEVEL 3 (L3) CACHE

FASTER
EXPENSIVE
SMALL CAPACITY

EDO, SD-RAM, DDR-SDRAM, RD-RAM

and More...

PHYSICAL MEMORY

RAMDOM ACCESS MEMORY (RAM)

FAST
PRICED REASONABLY
AVERAGE CAPACITY

SSD, Flash Drive

SOLID STATE MEMORY

NON-VOLATILE FLASH-BASED MEMORY

AVERAGE SPEED
PRICED REASONABLY
AVERAGE CAPACITY

Mechanical Hard Drives

VIRTUAL MEMORY

FILE-BASED MEMORY

SLOW
CHEAP
LARGE CAPACTITY

▲ Simplified Computer Memory Hierarchy
Illustration: Ryan J. Leng

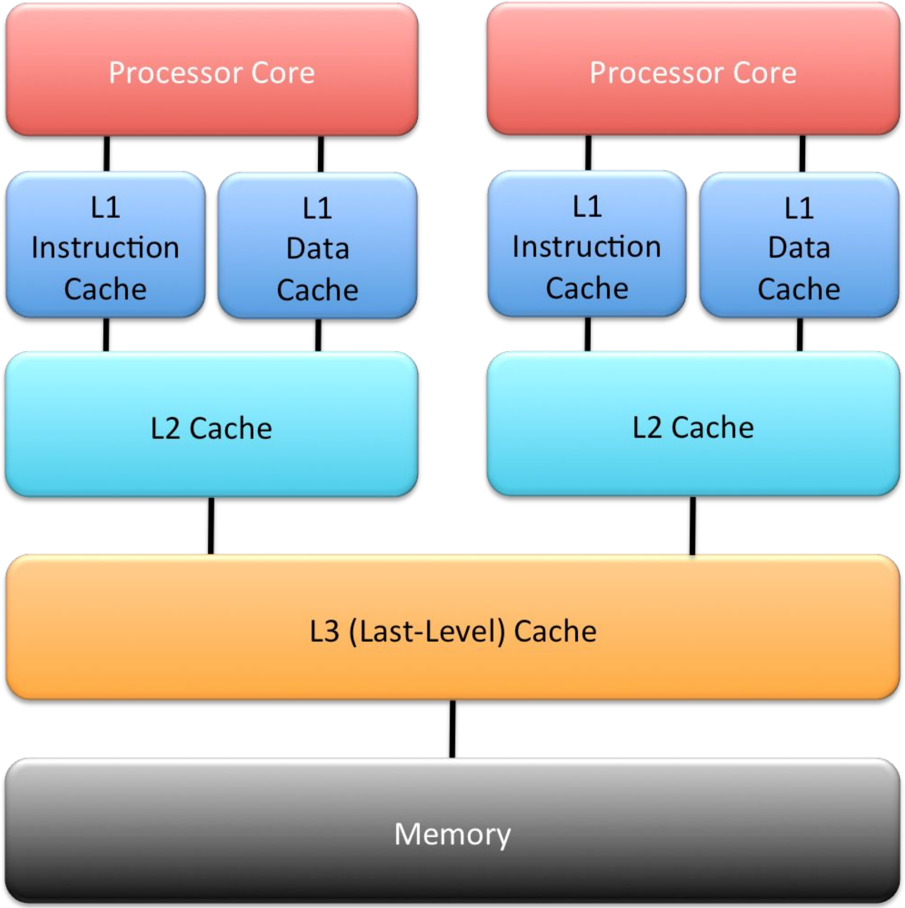


Image source: https://microkerneldude.wordpress.com/2015/04/27/how-to-steal-encryption-keys-your-cloud-is-not-as-secure-as-you-may-think/

Caches are small.

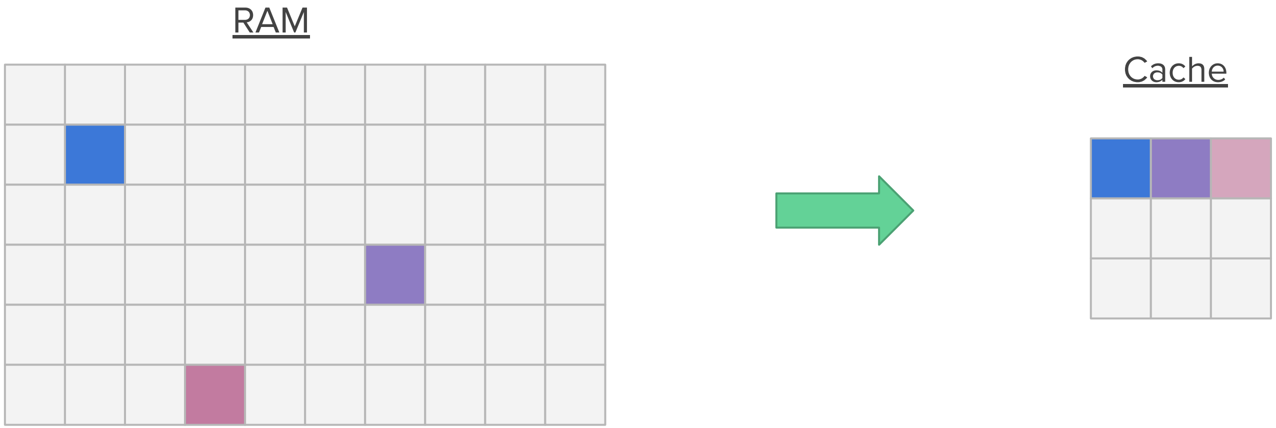
How can they make a difference?

“Locality principle”

Programs access a relatively **small portion** of **address space** at a time

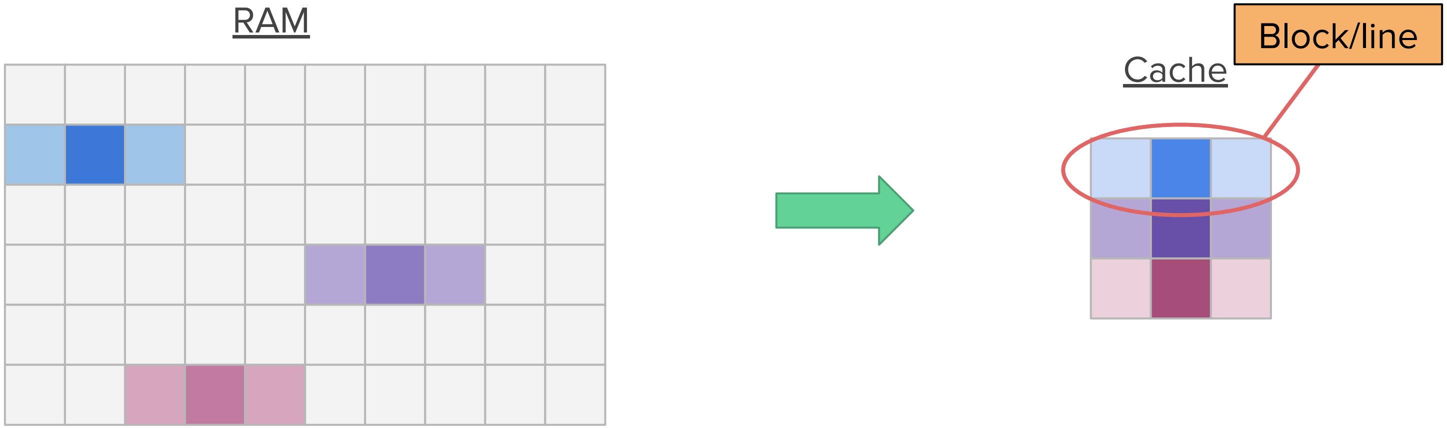
Temporal locality

- Memory location accessed?
  - Same location *likely* accessed again soon!
  - Add it to cache!



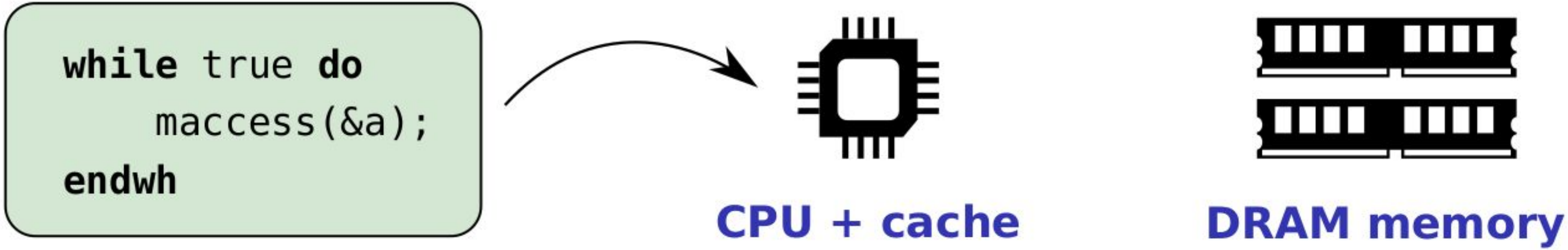
Spatial locality

- Memory location accessed?
  - Nearby locations *likely* accessed soon!
- Transfer **block** to cache
  - Cache line = block = #bytes transferred at once



CPU cache operation

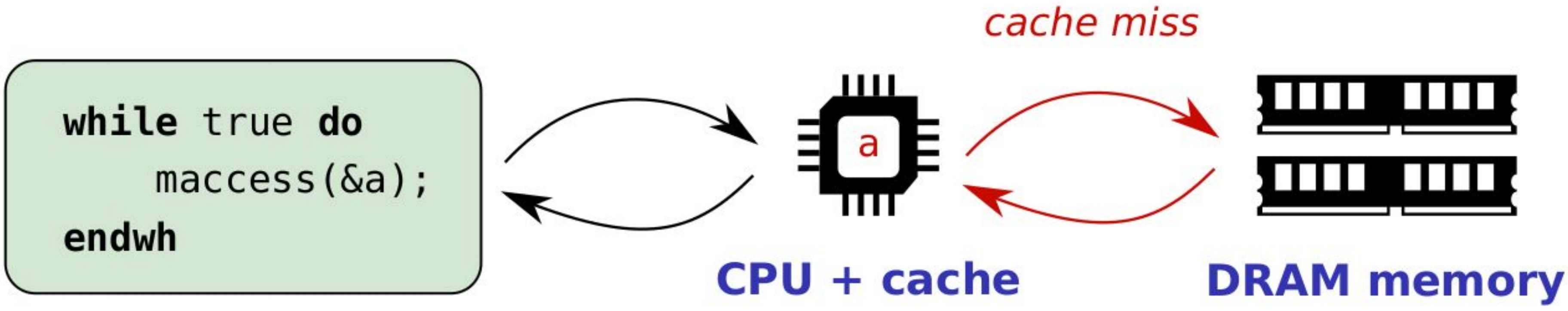
Cache principle: CPU speed ≫ DRAM latency → *cache code/data*



```
while true do
    maccess(&a);
endwh
```

CPU cache operation

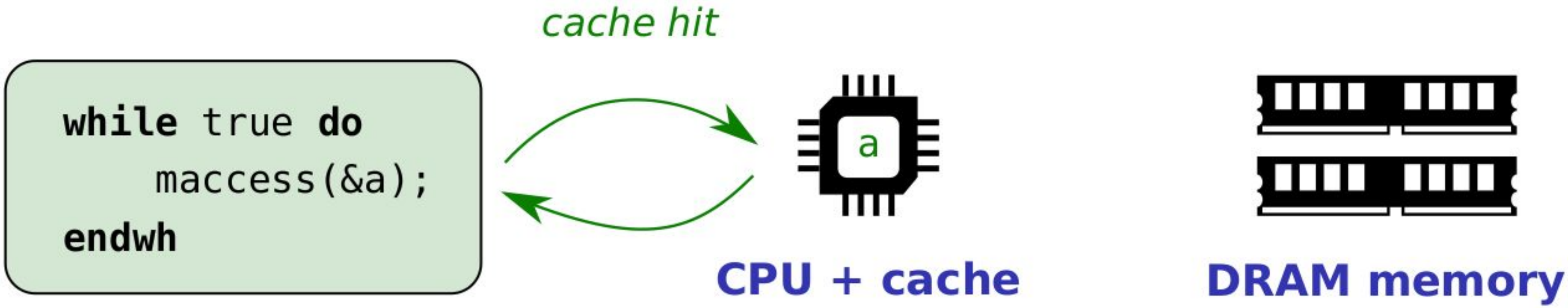
Cache miss: Request data from (slow) DRAM upon first use



```
while true do
    maccess(&a);
endwh
```

CPU cache operation

Cache hit: No DRAM access required for subsequent uses



```
while true do
    maccess(&a);
endwh
```

cache hit

a

CPU + cache

DRAM memory

Cache performance

miss rate = (1 – hit rate)

Conclusion:

Small amount of memory, used in smart way

Can we abuse timing differences as an attacker to infer secret information? :-)

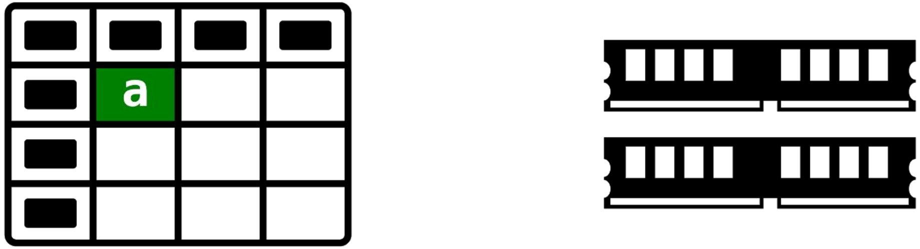
CPUs offer **instructions to *flush* the cache**!

Flush+Reload: Cache timing attacks on shared memory

```
if secret do
    maccess(&a);
else
    maccess(&b);
endif
```

```
flush(&a);
start_timer
   maccess(&a);
end_timer
```

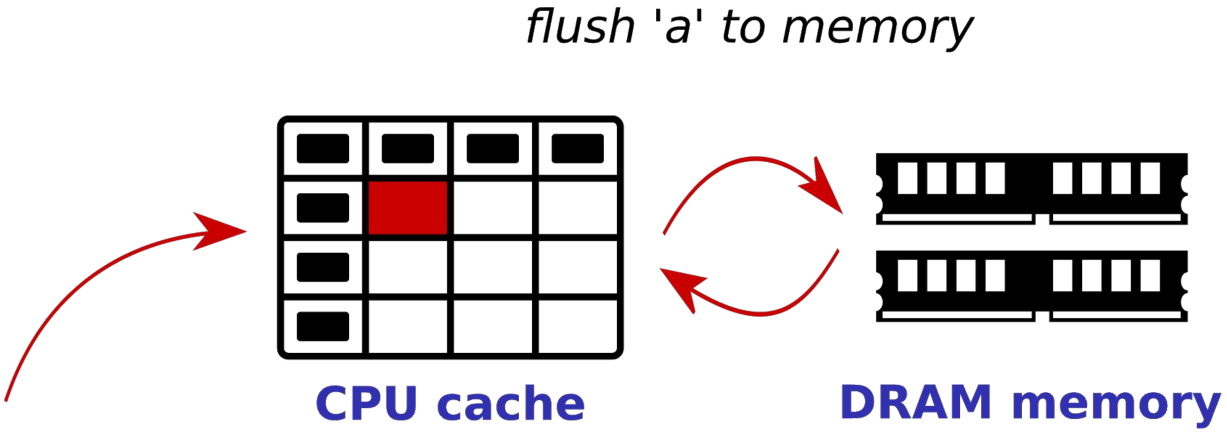
'a' is accessible to attacker



Flush+Reload: Cache timing attacks on shared memory

```
if secret do
    maccess(&a);
else
    maccess(&b);
endif
```

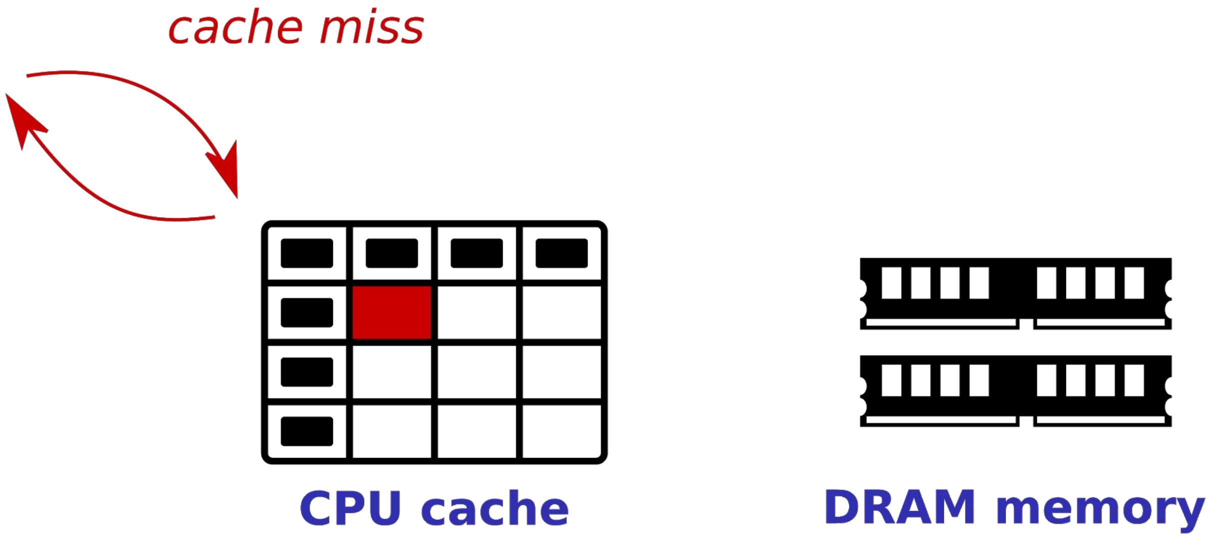
```
flush(&a);
start_timer
    maccess(&a);
end_timer
```



Flush+Reload: Cache timing attacks on shared memory

```
if secret do
    maccess(&a);
else
    maccess(&b);
endif
```

```
flush(&a);
start_timer
   maccess(&a);
end_timer
```



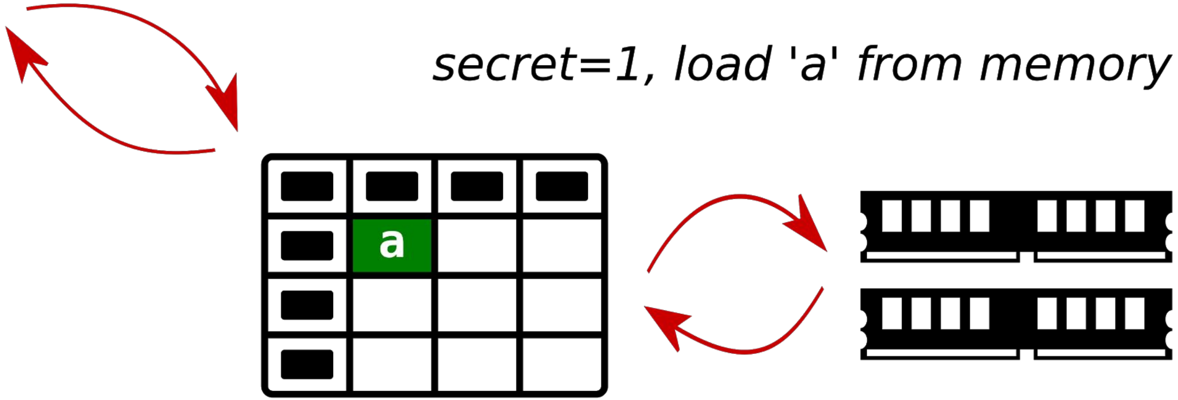
Flush+Reload: Cache timing attacks on shared memory

```
if secret do
    maccess(&a);
else
    maccess(&b);
endif
```

```
flush(&a);
start_timer
   maccess(&a);
end_timer
```

cache miss

secret=1, load 'a' from memory



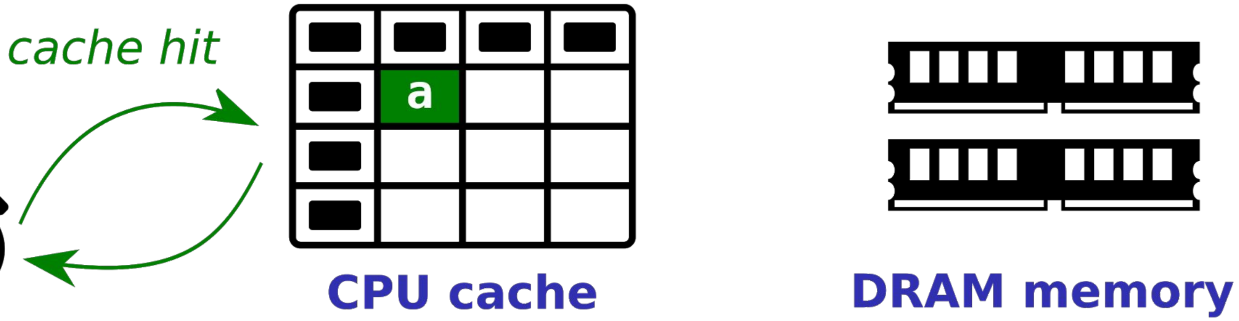
CPU cache

DRAM memory

Flush+Reload: Cache timing attacks on shared memory

```
if secret do
    maccess(&a);
else
    maccess(&b);
endif
```

```
flush(&a);
start_timer
   maccess(&a);
end_timer
```

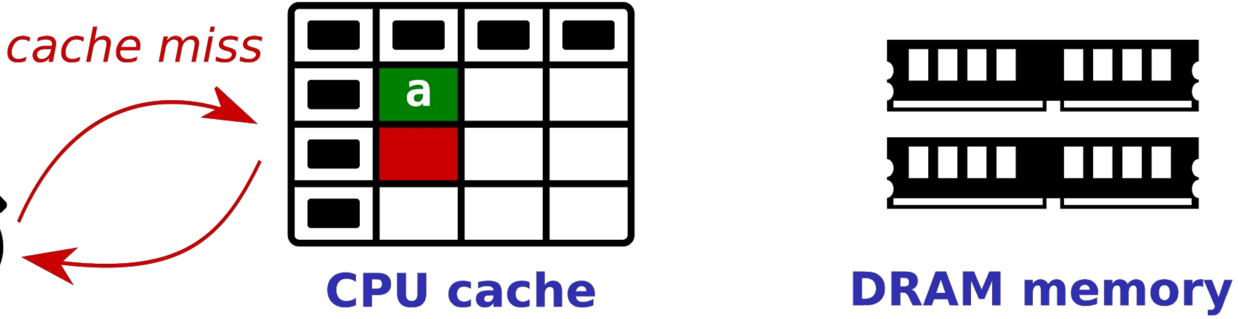


fast access(&a) → secret=1

Flush+Reload: Cache timing attacks on shared memory

```
if secret do
    maccess(&a);
else
    maccess(&b);
endif
```

```
flush(&a);
start_timer
   maccess(&b);
end_timer
```

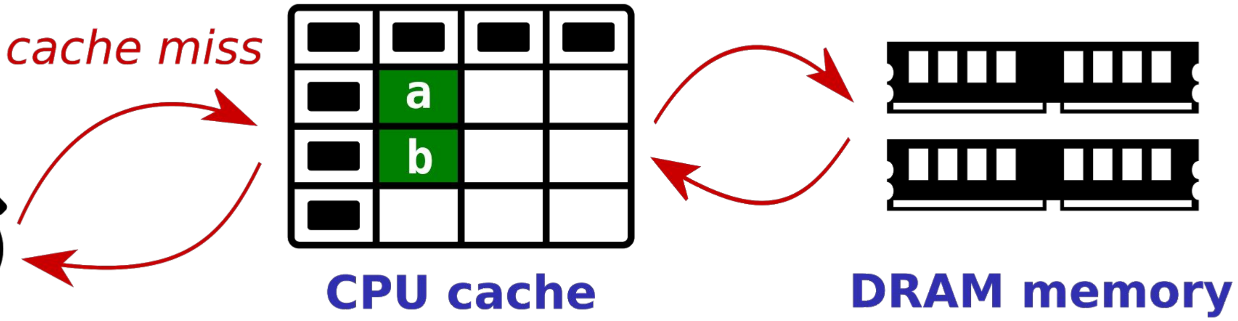


Flush+Reload: Cache timing attacks on shared memory

```
if secret do
    maccess(&a);
else
    maccess(&b);
endif
```

```
flush(&a);
start_timer
    maccess(&b);
end_timer
```

load 'b' from memory



slow access(&b) → secret=1

Flush+Reload Limitations

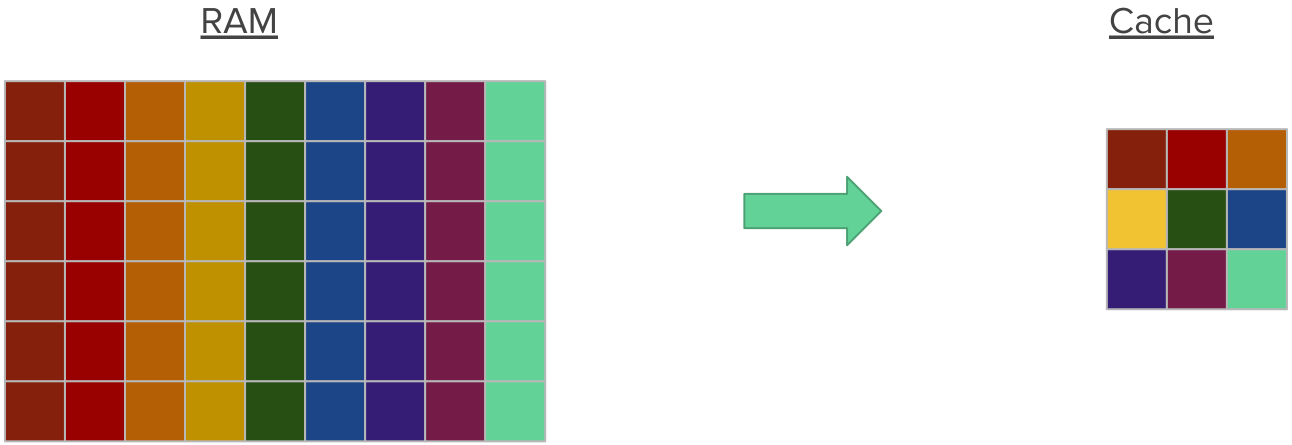
- Very **reliable and easy** attack
- But requires **shared memory** between victim and attacker apps
  - otherwise attacker cannot *flush* victim cache lines

→ **Generic** attack: does *not require knowledge* of internal cache organization, but only applicable in *limited scenarios* (when victim and attacker share memory)

Where in the cache should we store the contents of a memory location?

Direct mapping: concept

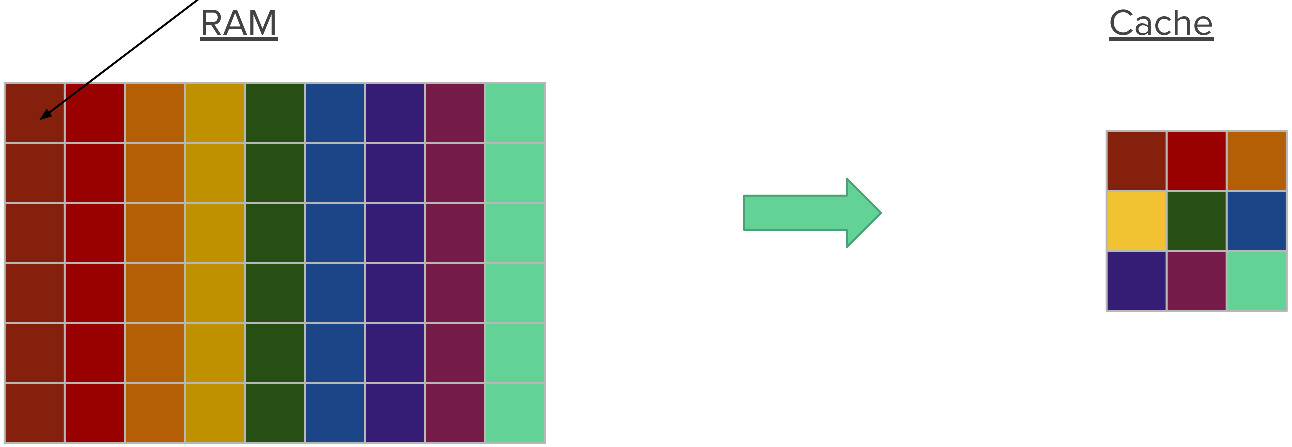
Notice: memory block index % 9 = cache block index



Direct mapping: concept

size of a block: 2^n bytes
=>
Address in same block share all but last n bits

Each cache block has multiple addresses!



size of a block: 2^n bytes => Address in same block share all but last n bits

Why?

Take n = 3

=> 8 memory (byte)-addresses per block

Notice #1: shared bits = *memory block index* in binary!!

Notice #2: *last n bits* are in index within the block

Block index	Memory byte-address range	Binary range
0	[0, 7]	[**00**000 - **00**111]
1	[8, 15]	[**01**000 - **01**111]
2	[16, 23]	[**10**000 - **10**111]
...	...	...

size of a block: 2^n bytes => Address in same block share all but last n bits

Remember
memory block index % cache size (in blocks) = cache block index

Why?
Take n = 3
=> 8 memory (byte)-addresses per block

Notice #1: shared bits = *memory block index* in binary!!
Notice #2: *last n bits* are in index within the block

Block index	Memory byte-address range	Binary range
0	[0, 7]	[**00**000 - **00**111]
1	[8, 15]	[**01**000 - **01**111]
2	[16, 23]	[**10**000 - **10**111]
...	...	...

binary(0xabc) =

$$\underset{= 10}{\{1010\}}\underset{= 23}{\{10111\}}\underset{= 4}{\{100\}}$$

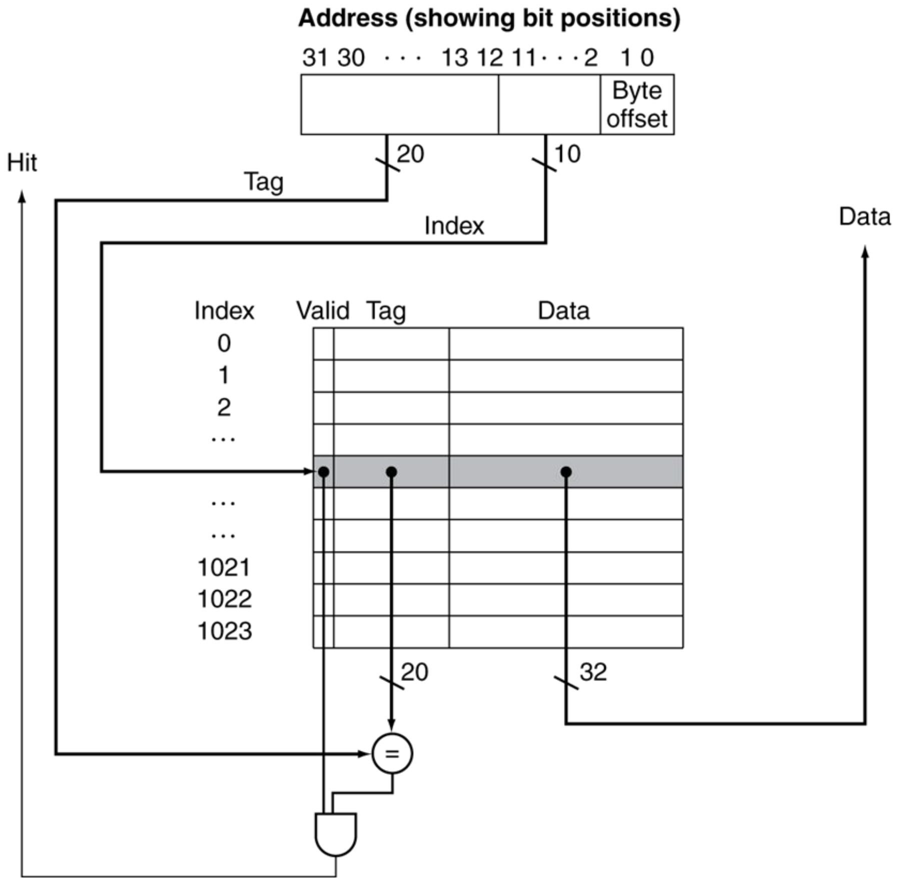
Assume:
Cache size: 32 blocks (2^5)
Block size: 8 bytes (2^3)

Index	Valid	Tag	Data
0	0	???	???
...	...	...	
10111 (23)	1	1010	8 bytes (0xab8 - 0xabf)
...	...	...	
11111 (31)	???	???	???

<u>Notice:</u>

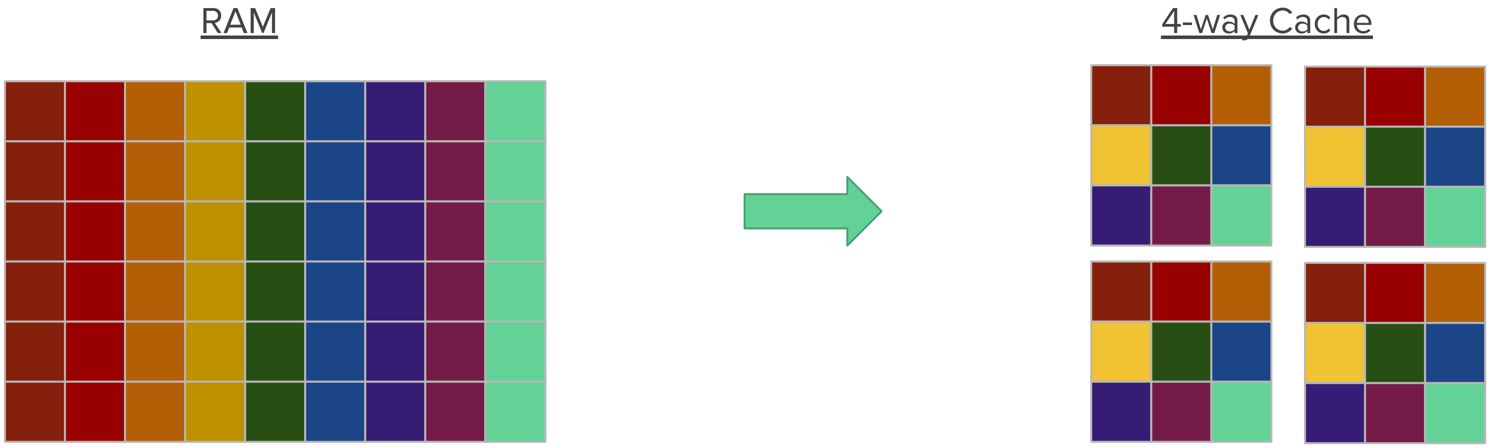
Index determines where to look

Tag determines hit or miss



Set-associativity: concept

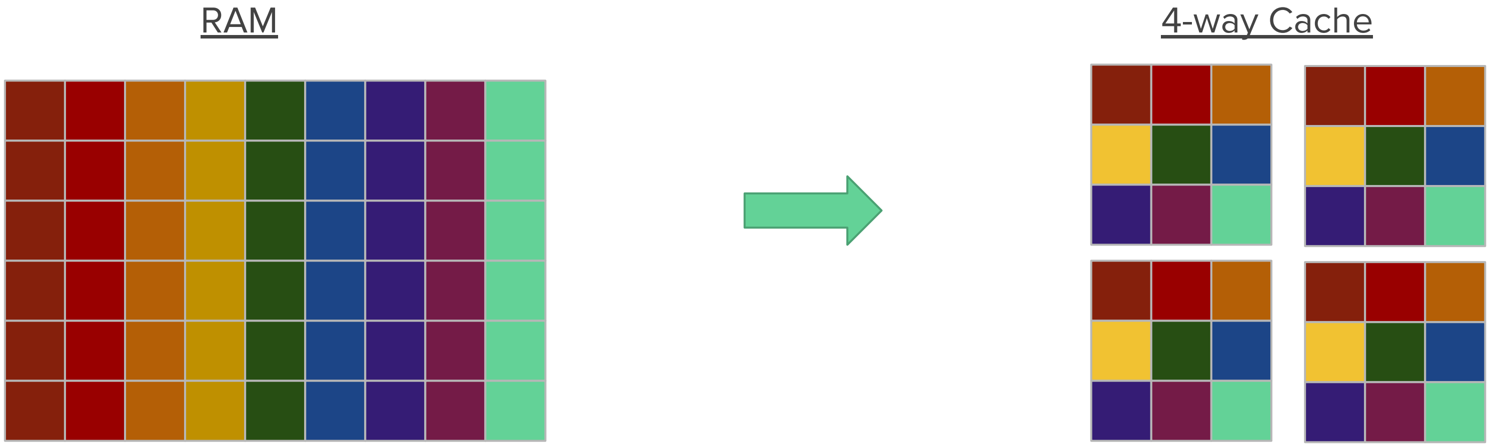
Notice: each memory block can be located in *n* different **ways**



Set-associativity: concept

Notice
n = 1: direct mapping!

Notice: each memory block can be located in *n* different **ways**



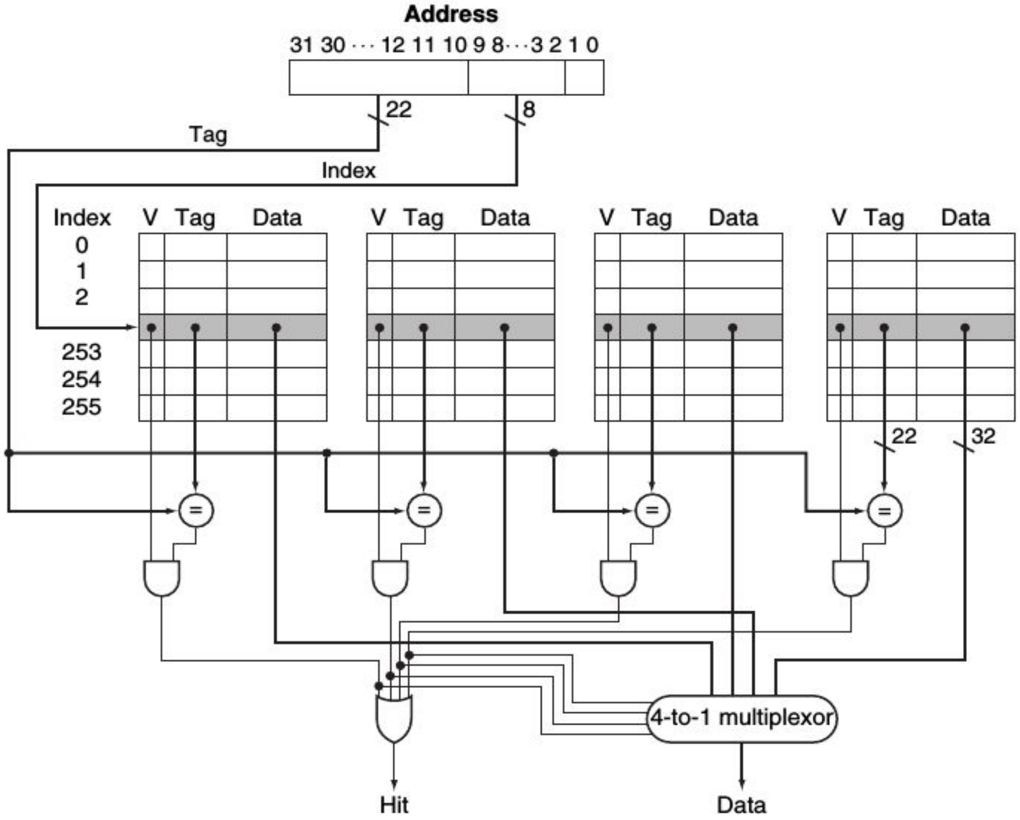
Notice:

Index determines where to look

BUT: multiple (#ways) possibilities

Check all:

Tag determines hit or miss



Fully associative cache

- n-way set associative cache
  - n = #memory-blocks

- Best performance possible
  - only miss when a new unique address is accessed
  - any repeated request is a hit!

E.g., a cache with 8 blocks:

Set	V	Tag	Data	V	Tag	Data	V	Tag	Data	V	Tag	Data	V	Tag	Data	V	Tag	Data	V	Tag	Data	V	Tag	Data
0																								

Fully associative cache

Basically, searching the entire cache without any indexing – sequentially super slow!

Solution: search in different ways in parallel.
Need **n** comparators for any associative cache.

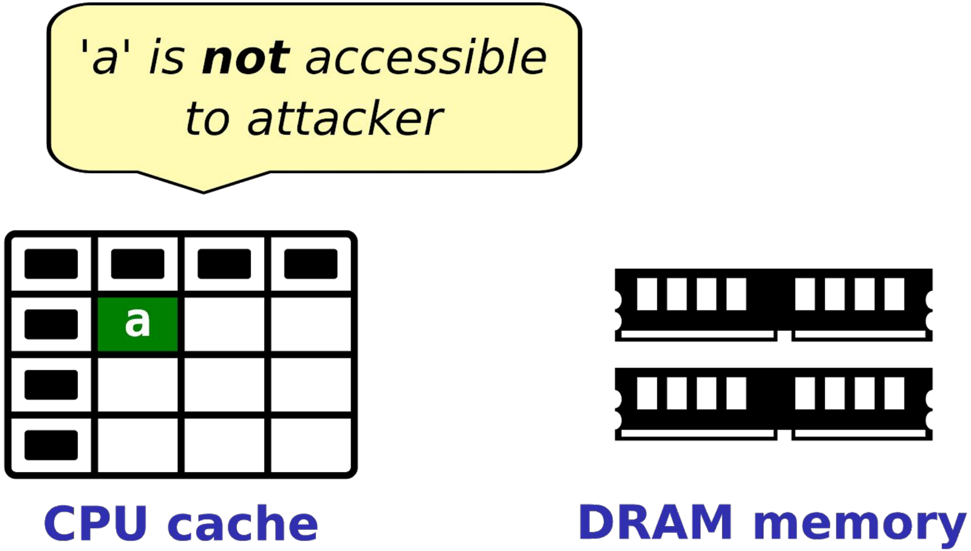
Set	V	Tag	Data	V	Tag	Data	V	Tag	Data	V	Tag	Data	V	Tag	Data	V	Tag	Data	V	Tag	Data	V	Tag	Data
0																								

Can we leverage detailed knowledge of mapping schemes and cache collisions to mount more advanced cache timing attack?

Prime+Probe: Cache timing attacks across protection domains

```
if secret do
    maccess(&a);
else
    maccess(&b);
endif
```

```
maccess(&c);
start_timer
   maccess(&c);
end_timer
```



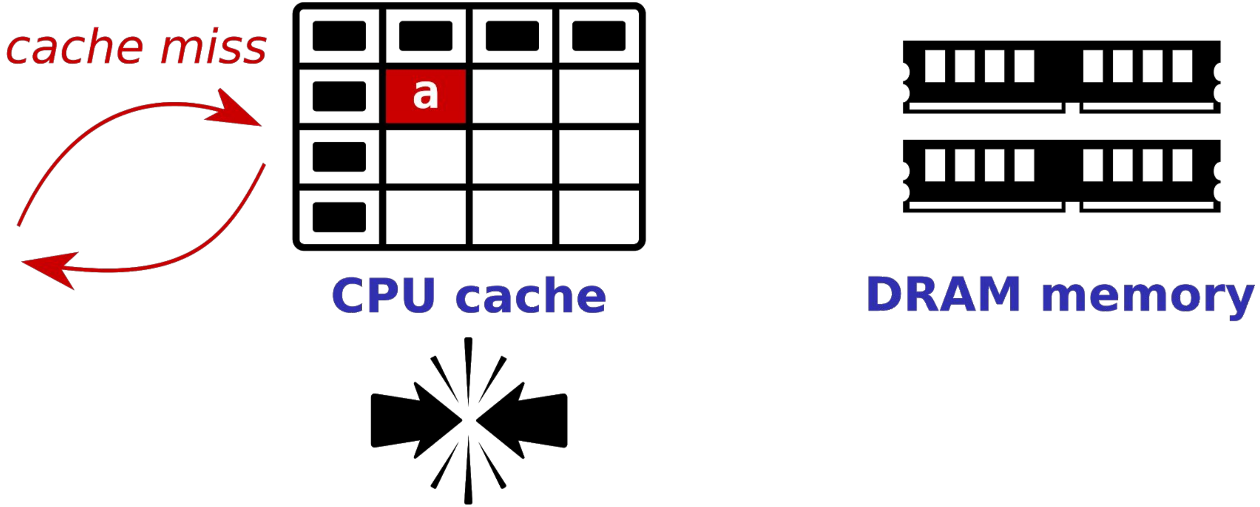
'a' is **not** accessible to attacker

Prime+Probe: Cache timing attacks across protection domains

```
if secret do
    maccess(&a);
else
    maccess(&b);
endif
```

```
maccess(&c);
start_timer
   maccess(&c);
end_timer
```

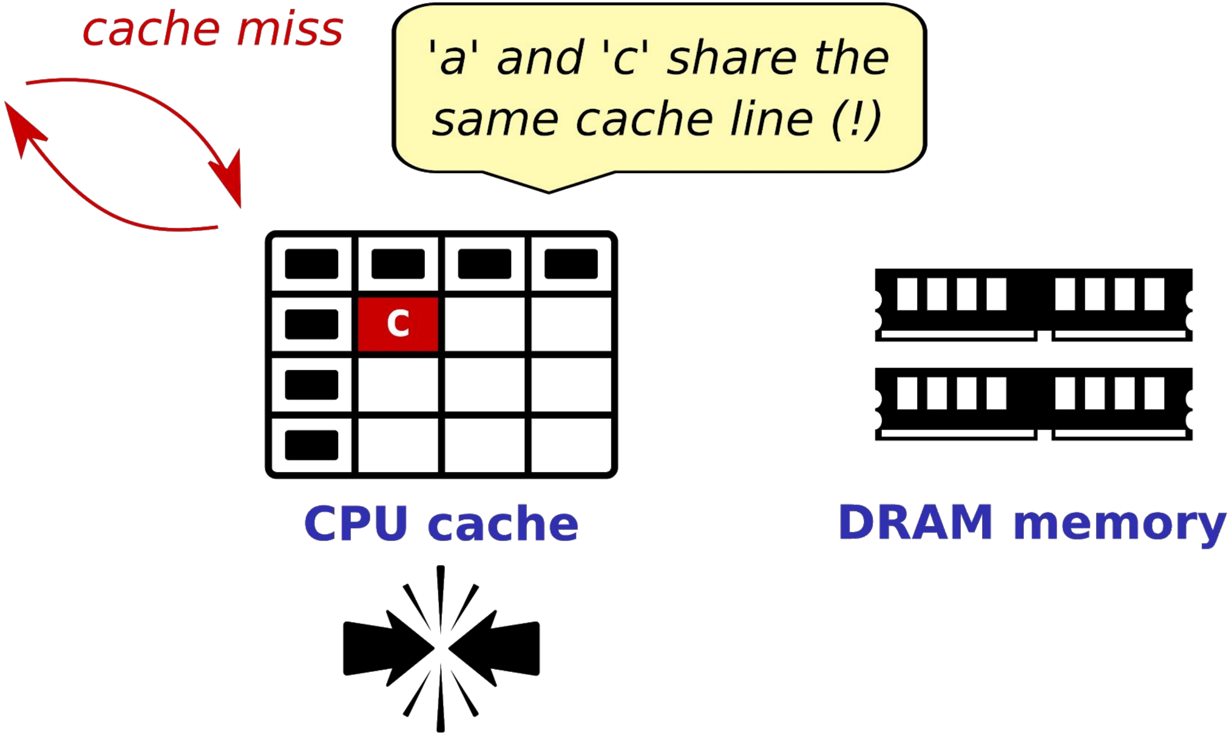
'a' and 'c' share the same cache line (!)



Prime+Probe: Cache timing attacks across protection domains

```
if secret do
    maccess(&a);
else
    maccess(&b);
endif
```

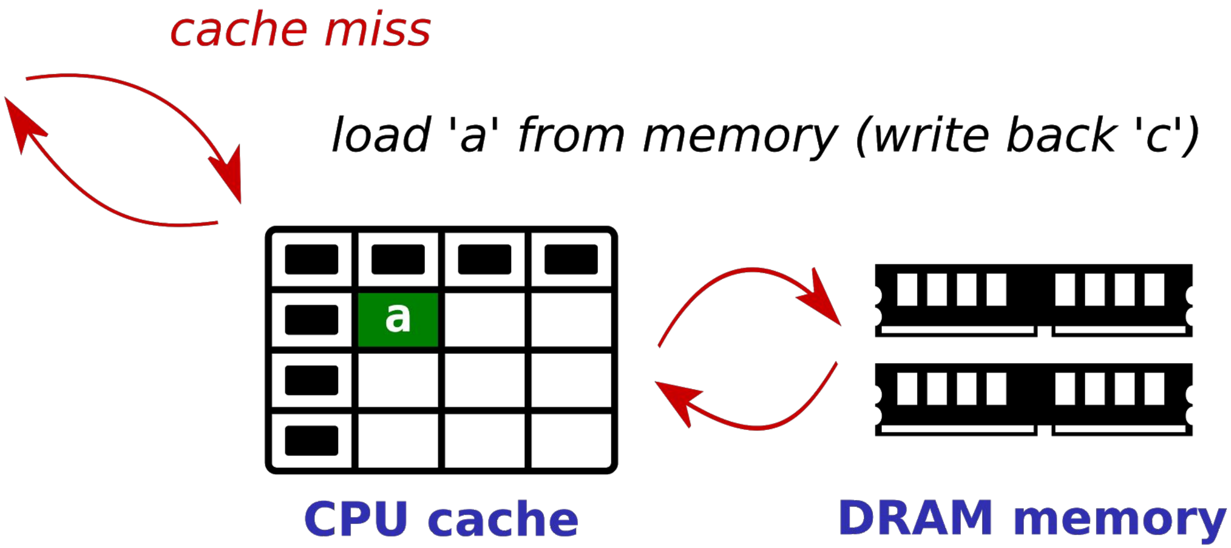
```
flush(&c);
start_timer
   maccess(&c);
end_timer
```



Prime+Probe: Cache timing attacks across protection domains

```
if secret do
    maccess(&a);
else
    maccess(&b);
endif
```

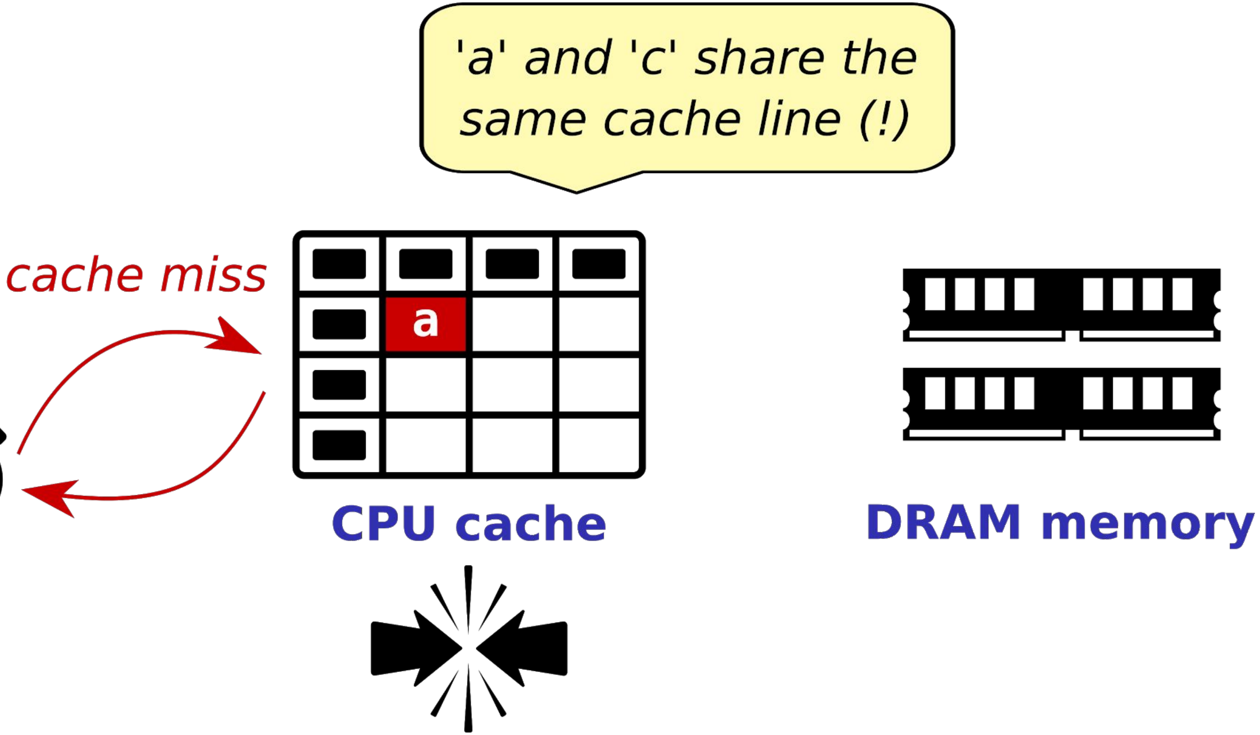
```
flush(&c);
start_timer
   maccess(&c);
end_timer
```



Prime+Probe: Cache timing attacks across protection domains

```
if secret do
    maccess(&a);
else
    maccess(&b);
endif
```

```
flush(&c);
start_timer
    maccess(&c);
end_timer
```



'a' and 'c' share the same cache line (!)

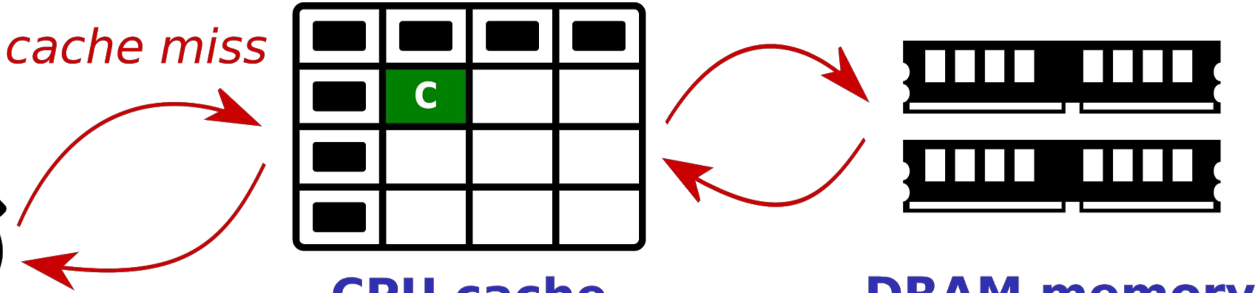
Prime+Probe: Cache timing attacks across protection domains

```
if secret do
    maccess(&a);
else
    maccess(&b);
endif
```

```
flush(&c);
start_timer
    maccess(&c);
end_timer
```

load 'c' from memory (write back 'a')

cache miss



CPU cache

DRAM memory

slow access(&c) → secret=1

Appendix

Example 1: direct mapping 1-word blocks

Word address references:
22, 26, 22, 27, 3, 27, 53, 55

010110 → miss
011010 → miss
010110 → hit
011011 → miss
000011 → miss
011011 → miss
110101 → miss
110111 → miss

Index	V	Tag	Data
000	0		
001	0		
010	0		
011	0		
100	0		
101	0		
110	1	010	22
111	0		

Index	V	Tag	Data
000	0		
001	0		
010	1	011	26
011	0		
100	0		
101	0		
110	1	010	22
111	0		

Index	V	Tag	Data
000	0		
001	0		
010	1	011	26
011	1	011	27
100	0		
101	0		
110	1	010	22
111	0		

Index	V	Tag	Data
000	0		
001	0		
010	1	011	26
011	1	000	3
100	0		
101	0		
110	1	010	22
111	0		

Index	V	Tag	Data
000	0		
001	0		
010	1	011	26
011	1	011	27
100	0		
101	0		
110	1	010	22
111	0		

Index	V	Tag	Data
000	0		
001	0		
010	1	011	26
011	1	011	27
100	0		
101	0	110	53
110	1	010	22
111	0		

Index	V	Tag	Data
000	0		
001	0		
010	1	011	26
011	1	011	27
100	0		
101	0	110	53
110	1	010	22
111	1	110	55

Example 2: direct mapping 4-word blocks

Word address references:
22, 26, 22, 27, 3, 27, 53, 55

010110 → 101 → miss
011010 → 110 → miss
010110 → 101 → hit
011011 → 110 → hit
000011 → 000 → miss
011011 → 110 → hit
110101 → 110 → miss
110111 → 110 → hit

			Data			
Index	V	Tag	W1	W2	W3	W4
000	1					
001	0					
010	0					
011	0					
100	0					
101	1	010	20	21	22	23
110	0					
111	0					

			Data			
Index	V	Tag	W1	W2	W3	W4
000	0					
001	0					
010	0					
011	0					
100	0					
101	1	010	20	21	22	23
110	1	011	24	25	26	27
111	0					

			Data			
Index	V	Tag	W1	W2	W3	W4
000	1	000	0	1	2	3
001	0					
010	0					
011	0					
100	0					
101	1	010	20	21	22	23
110	1	011	24	25	26	27
111	0					

			Data			
Index	V	Tag	W1	W2	W3	W4
000	1	000	0	1	2	3
001	0					
010	0					
011	0					
100	0					
101	1	010	20	21	22	23
110	1	110	52	53	54	55
111	0					

Example 2: direct mapping 4-word blocks

Word address references:
22, 26, 22, 27, 3, 27, 53, 55

010110 → 101 → miss
011010 → 110 → miss
010110 → 101 → hit
011011 → 110 → hit
000011 → 000 → miss
011011 → 110 → hit
110101 → 110 → miss
110111 → 110 → hit

Miss rate decreased. Why?

			Data			
Index	V	Tag	W1	W2	W3	W4
000	1					
001	0					
010	0					
011	0					
100	0					
101	1	010	20	21	22	23
110	0					
111	0					

			Data			
Index	V	Tag	W1	W2	W3	W4
000	0					
001	0					
010	0					
011	0					
100	0					
101	1	010	20	21	22	23
110	1	011	24	25	26	27
111	0					

			Data			
Index	V	Tag	W1	W2	W3	W4
000	1	000	0	1	2	3
001	0					
010	0					
011	0					
100	0					
101	1	010	20	21	22	23
110	1	011	24	25	26	27
111	0					

			Data			
Index	V	Tag	W1	W2	W3	W4
000	1	000	0	1	2	3
001	0					
010	0					
011	0					
100	0					
101	1	010	20	21	22	23
110	1	110	52	53	54	55
111	0					

Example 2: direct mapping 4-word blocks

Word address references:
22, 26, 22, 27, 3, 27, 53, 55

010110 → 101 → miss
011010 → 110 → miss
010110 → 101 → hit
011011 → 110 → hit
000011 → 000 → miss
011011 → 110 → hit
110101 → 110 → miss
110111 → 110 → hit

Miss rate decreased. Why?

But miss penalty increased. Why?

			Data			
Index	V	Tag	W1	W2	W3	W4
000	1					
001	0					
010	0					
011	0					
100	0					
101	1	010	20	21	22	23
110	0					
111	0					

			Data			
Index	V	Tag	W1	W2	W3	W4
000	0					
001	0					
010	0					
011	0					
100	0					
101	1	010	20	21	22	23
110	1	011	24	25	26	27
111	0					

			Data			
Index	V	Tag	W1	W2	W3	W4
000	1	000	0	1	2	3
001	0					
010	0					
011	0					
100	0					
101	1	010	20	21	22	23
110	1	011	24	25	26	27
111	0					

			Data			
Index	V	Tag	W1	W2	W3	W4
000	1	000	0	1	2	3
001	0					
010	0					
011	0					
100	0					
101	1	010	20	21	22	23
110	1	110	52	53	54	55
111	0					

Example 3: 2-way set-associative 1-word blocks

Word address references:
22, 26, 22, 27, 3, 27, 53, 55

010110 → miss
011010 → miss
010110 → hit
011011 → miss
000011 → miss
011011 → hit
110101 → miss
110111 → miss

Set	V	Tag0	Data0	Tag1	Data1
00	0				
01	0				
10	1	0101	22		
11	0				

Set	V	Tag0	Data0	Tag1	Data1
00	0				
01	0				
10	1	0101	22	0110	26
11	0				

Set	V	Tag0	Data0	Tag1	Data1
00	0				
01	0				
10	1	0101	22	0110	26
11	1	0110	27		

Set	V	Tag0	Data0	Tag1	Data1
00	0				
01	0				
10	1	0101	22	0110	26
11	1	0110	27	0000	3

Set	V	Tag0	Data0	Tag1	Data1
00	0				
01	1	1101	53		
10	1	0101	22	0110	26
11	1	0110	27	0000	3

Set	V	Tag0	Data0	Tag1	Data1
00	0				
01	1	1101	53		
10	1	0101	22	0110	26
11	1	0110	27	1101	55

Replacement rule:
Least **R**ecently **U**sed

Example 3: 2-way set-associative 1-word blocks

Word address references:
22, 26, 22, 27, 3, 27, 53, 55

010110 → miss
011010 → miss
010110 → hit
011011 → miss
000011 → miss
011011 → hit
110101 → miss
110111 → miss

Set	V	Tag0	Data0	Tag1	Data1
00	0				
01	0				
10	1	0101	22		
11	0				

Set	V	Tag0	Data0	Tag1	Data1
00	0				
01	0				
10	1	0101	22	0110	26
11	0				

Set	V	Tag0	Data0	Tag1	Data1
00	0				
01	0				
10	1	0101	22	0110	26
11	1	0110	27		

Set	V	Tag0	Data0	Tag1	Data1
00	0				
01	0				
10	1	0101	22	0110	26
11	1	0110	27	0000	3

Set	V	Tag0	Data0	Tag1	Data1
00	0				
01	1	1101	53		
10	1	0101	22	0110	26
11	1	0110	27	0000	3

Set	V	Tag0	Data0	Tag1	Data1
00	0				
01	1	1101	53		
10	1	0101	22	0110	26
11	1	0110	27	1101	55

Miss rate decreased. Why?

Trade-off?

Example 3: 2-way set-associative 1-word blocks

Word address references:
22, 26, 22, 27, 3, 27, 53, 55

010110 → miss
011010 → miss
010110 → hit
011011 → miss
000011 → miss
011011 → hit
110101 → miss
110111 → miss

Set	V	Tag0	Data0	Tag1	Data1
00	0				
01	0				
10	1	0101	22		
11	0				

Set	V	Tag0	Data0	Tag1	Data1
00	0				
01	0				
10	1	0101	22	0110	26
11	0				

Set	V	Tag0	Data0	Tag1	Data1
00	0				
01	0				
10	1	0101	22	0110	26
11	1	0110	27		

Set	V	Tag0	Data0	Tag1	Data1
00	0				
01	0				
10	1	0101	22	0110	26
11	1	0110	27	0000	3

Set	V	Tag0	Data0	Tag1	Data1
00	0				
01	1	1101	53		
10	1	0101	22	0110	26
11	1	0110	27	0000	3

Set	V	Tag0	Data0	Tag1	Data1
00	0				
01	1	1101	53		
10	1	0101	22	0110	26
11	1	0110	27	1101	55

Miss rate decreased. Why?

Hit time increased! We search longer