

# System Security - Attack and Defense for Binaries



CS 4390/5390, Spring 2026

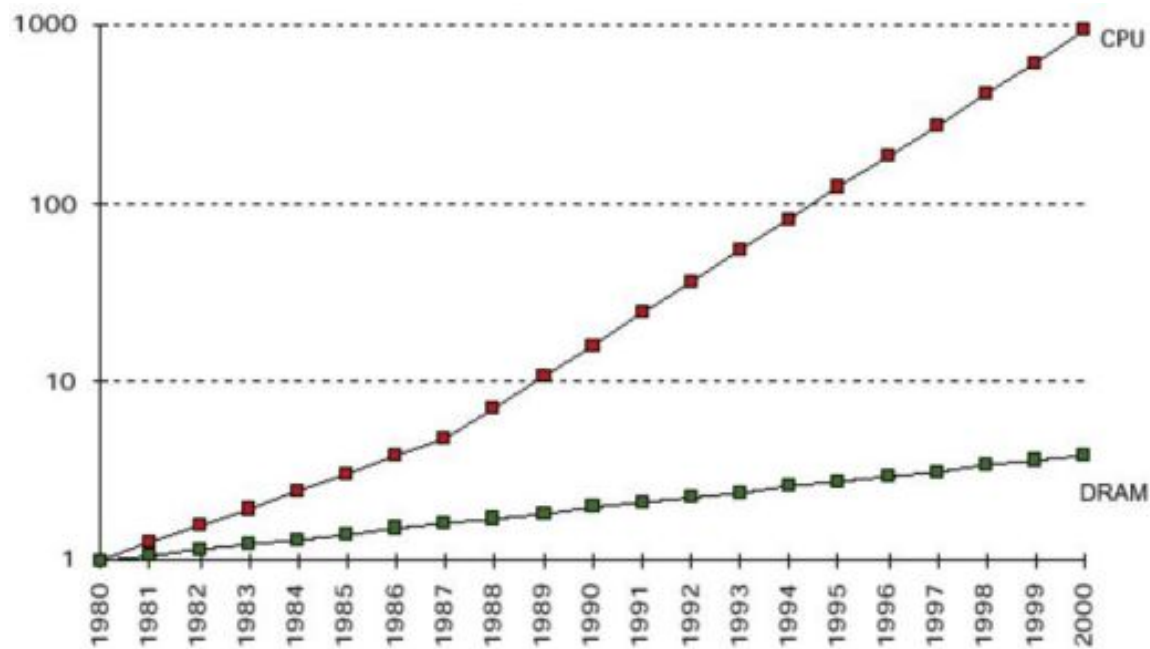
Instructor: MD Armanuzzaman (*Arman*)

# Cache side channel attack

## Meltdown

## Spectre

# Speed Gap Between CPU and DRAM



# Memory Hierarchy

*A tradeoff between Speed,  
Cost and Capacity*

*Ideally one would desire an indefinitely large memory capacity such that any particular ... word would be immediately available. ... We are ... forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible.*

**A. W. Burks, H. H. Goldstine, and  
J. von Neumann**

*Preliminary Discussion of the Logical Design of an  
Electronic Computing Instrument, 1946*

# CPU Cache

A cache is a small amount of fast, expensive memory (SRAM). The cache goes between the CPU and the main memory (DRAM).

It keeps a copy of the most frequently used data from the main memory.

All levels of caches are integrated onto the processor chip.

# Access Time

|           |                       | Access Time in 2012              |
|-----------|-----------------------|----------------------------------|
| Cache     | <u>Static RAM</u>     | <u>0.5 - 2.5 ns</u>              |
| Memory    | <u>Dynamic RAM</u>    | <u>50- 70 ns</u>                 |
| Secondary | <u>Flash</u>          | <u>5,000 - 50,000 ns</u>         |
|           | <u>Magnetic disks</u> | <u>5,000,000 - 20,000,000 ns</u> |

# Cache Hits and Misses

A cache hit occurs if the cache contains the data that we're looking for.

A cache miss occurs if the cache does not contain the requested data.

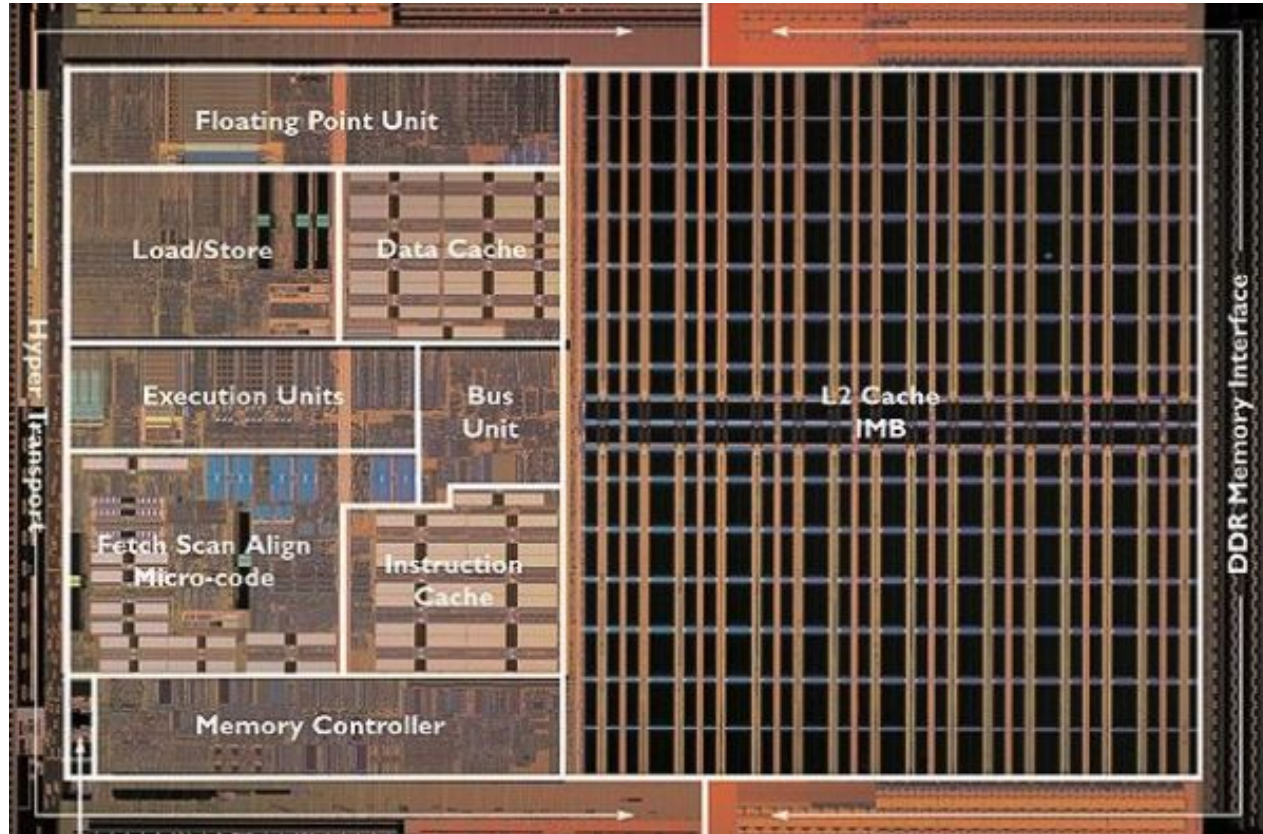
# Cache Hierarchy

L1 Cache is closest to the CPU. Usually divided in Code and Data cache

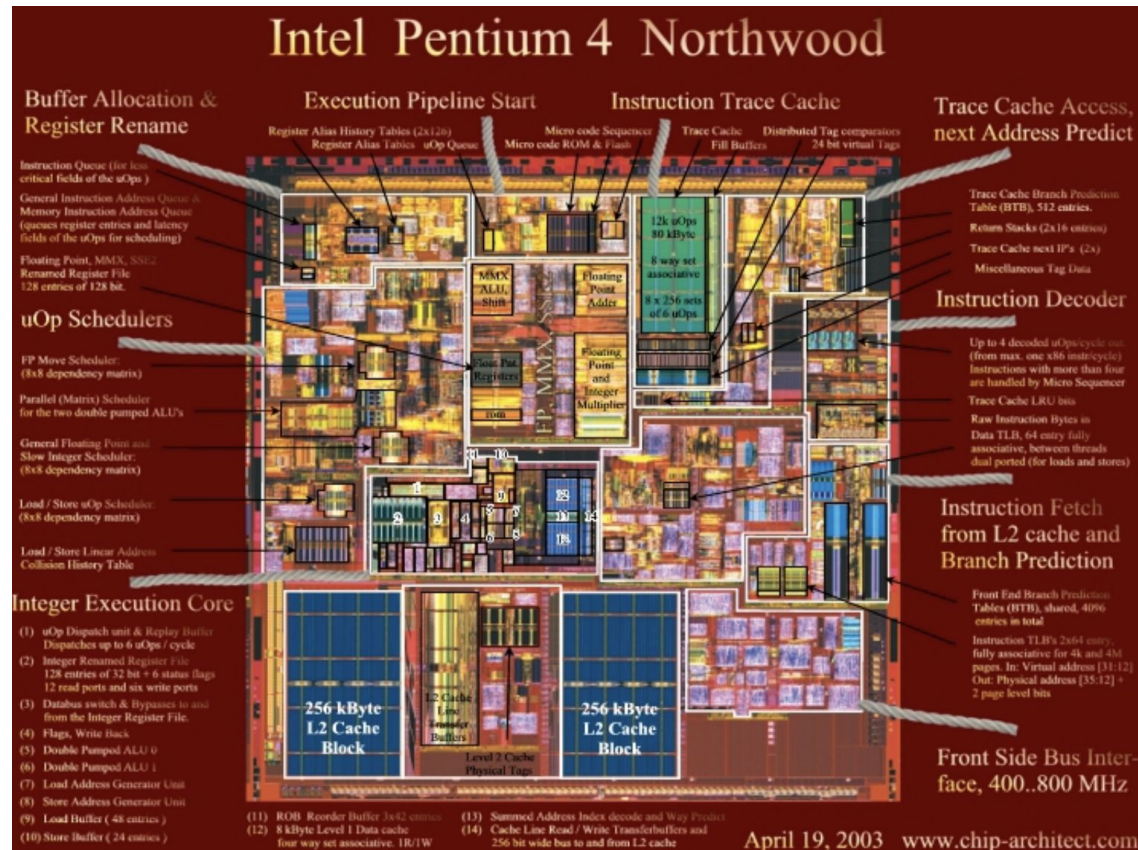
L2 and L3 cache are usually unified.

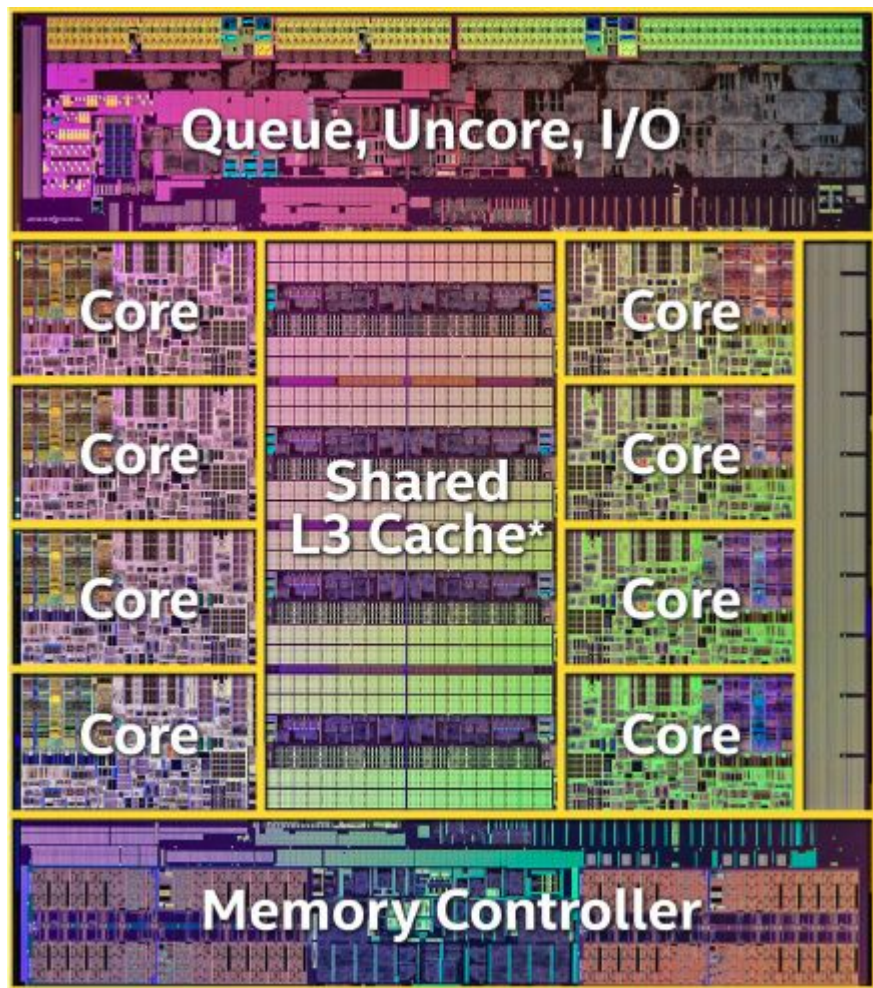


# Cache Hierarchy



# Cache Hierarchy





# Cache Line/Block

The minimum unit of information that can be either present or not present in a cache.

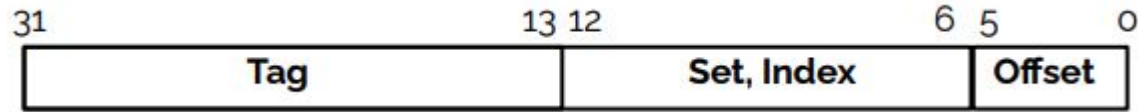
64 bytes in modern Intel and ARM CPU



# n-Way Set-Associative Cache

Any given block/line in the main memory may be cached in any of the  $n$  cache lines in one **cache set**.

# n-Way Set-Associative Cache



32KB 4-way set-associative data cache, 64 bytes per line

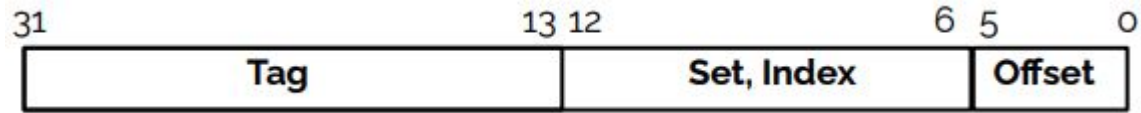
Number of sets

= Cache Size / (Number of ways \* Line size)

=  $32 * 1024 / (4 * 64)$

= 12

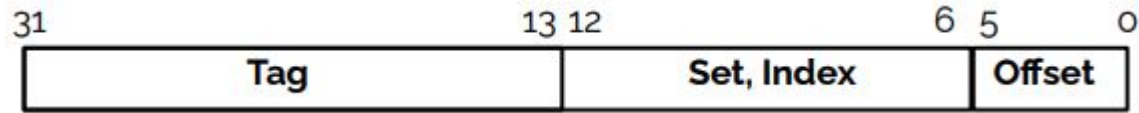
# n-Way Set-Associative Cache



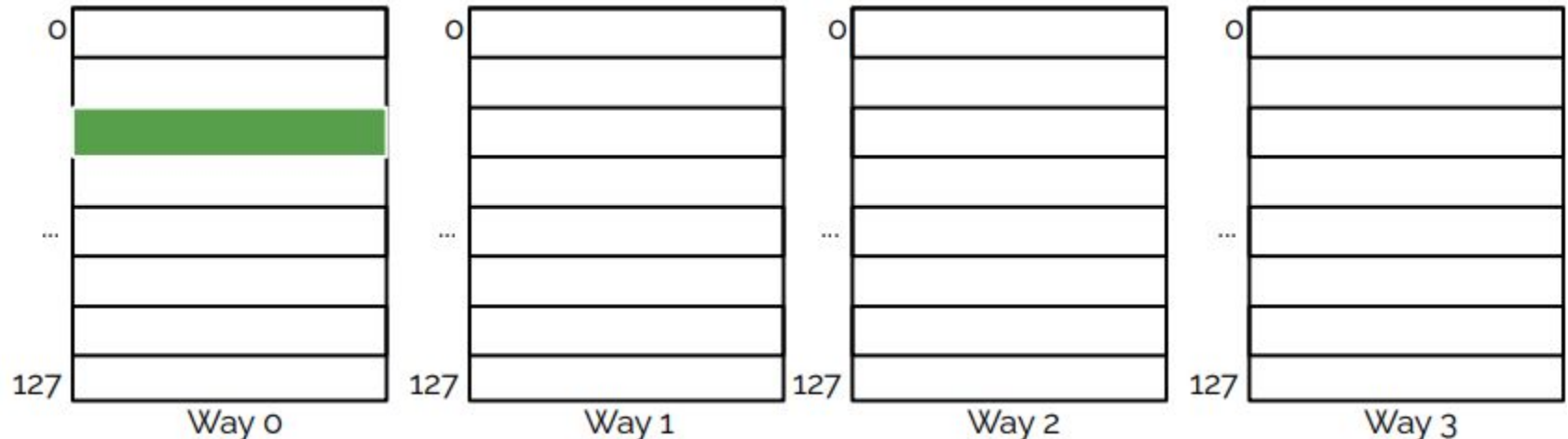
32KB 4-way set-associative data cache, 64 bytes per line



# n-Way Set-Associative Cache

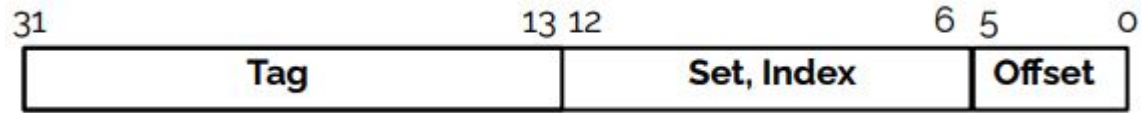


32KB 4-way set-associative data cache, 64 bytes per line

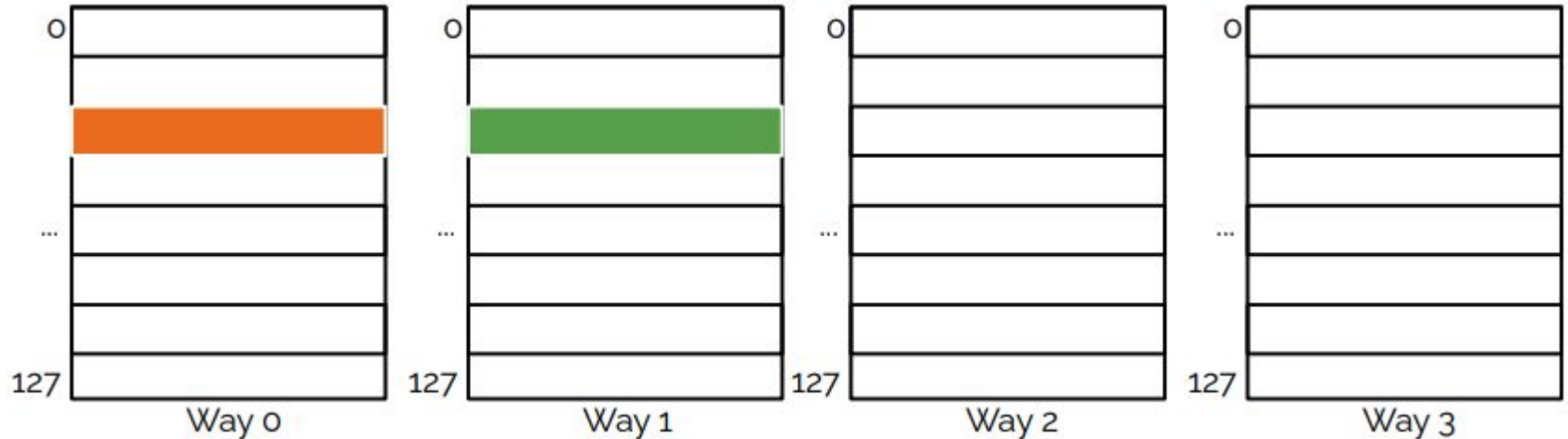




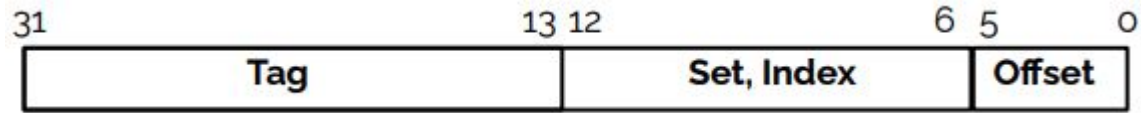
# n-Way Set-Associative Cache



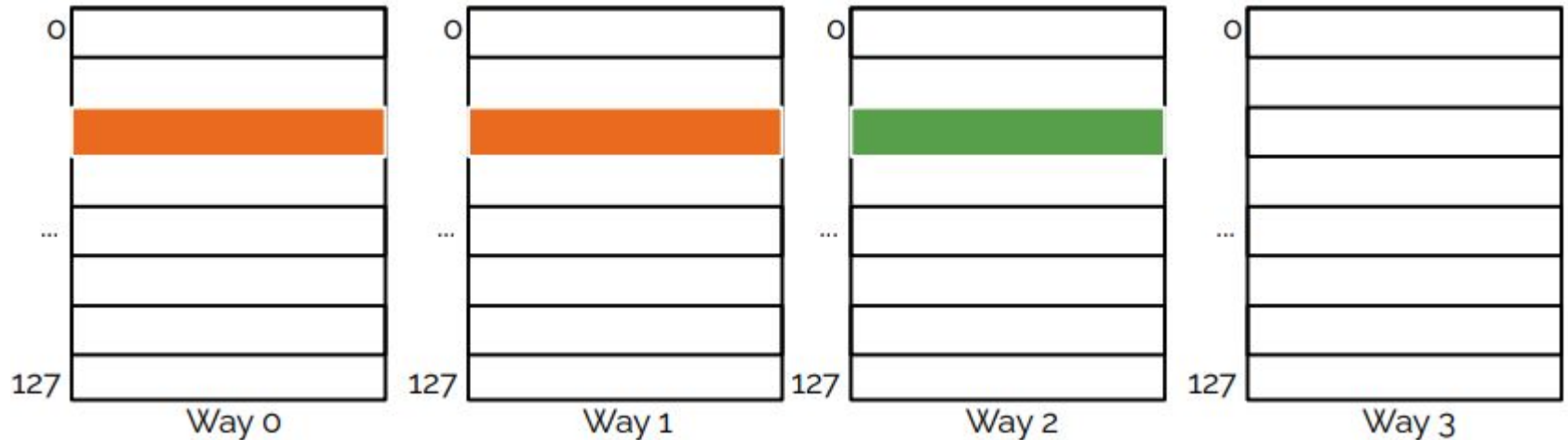
32KB 4-way set-associative data cache, 64 bytes per line



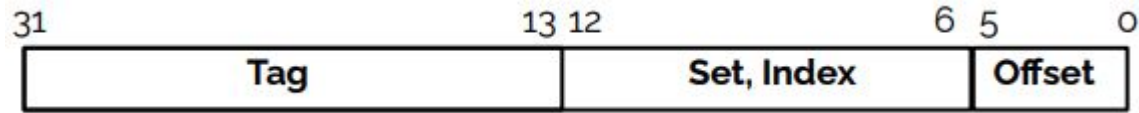
# n-Way Set-Associative Cache



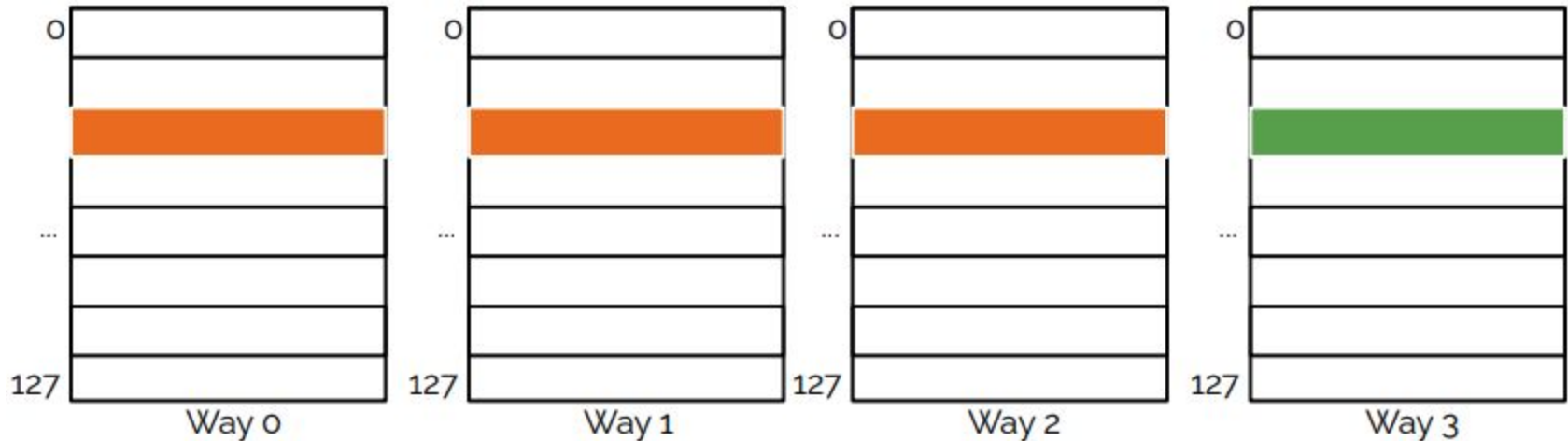
32KB 4-way set-associative data cache, 64 bytes per line



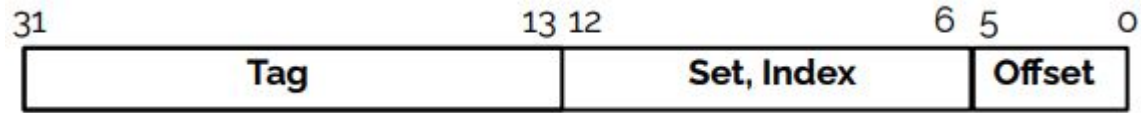
# n-Way Set-Associative Cache



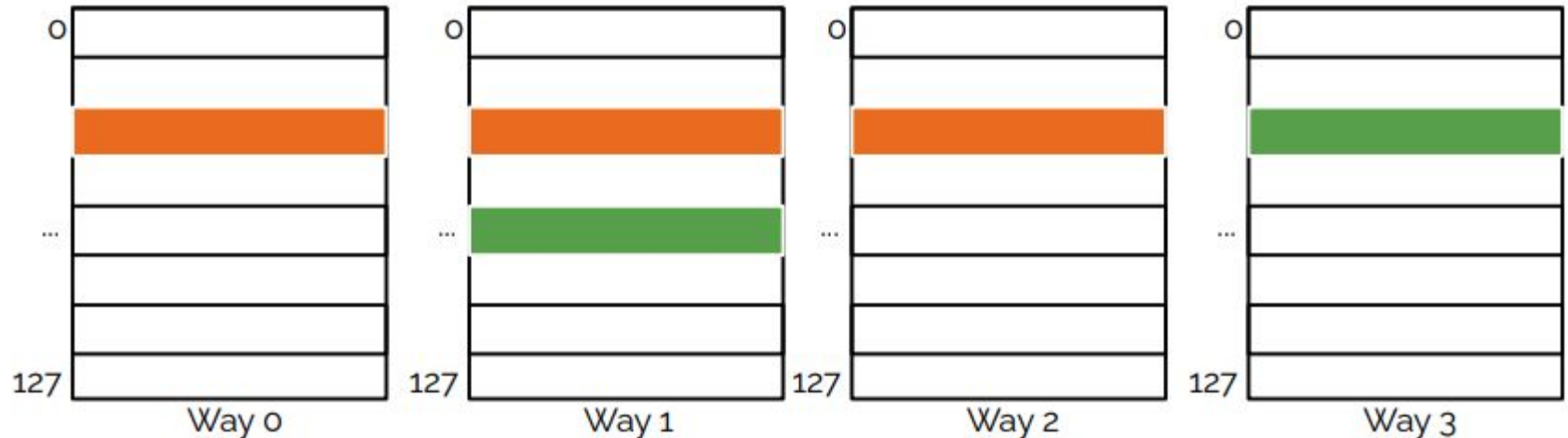
32KB 4-way set-associative data cache, 64 bytes per line



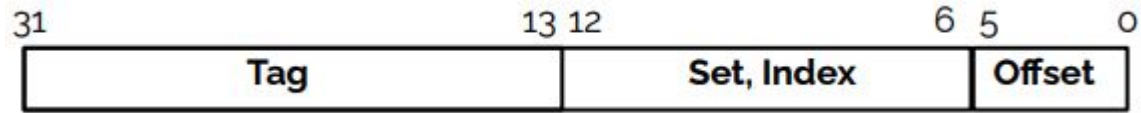
# n-Way Set-Associative Cache



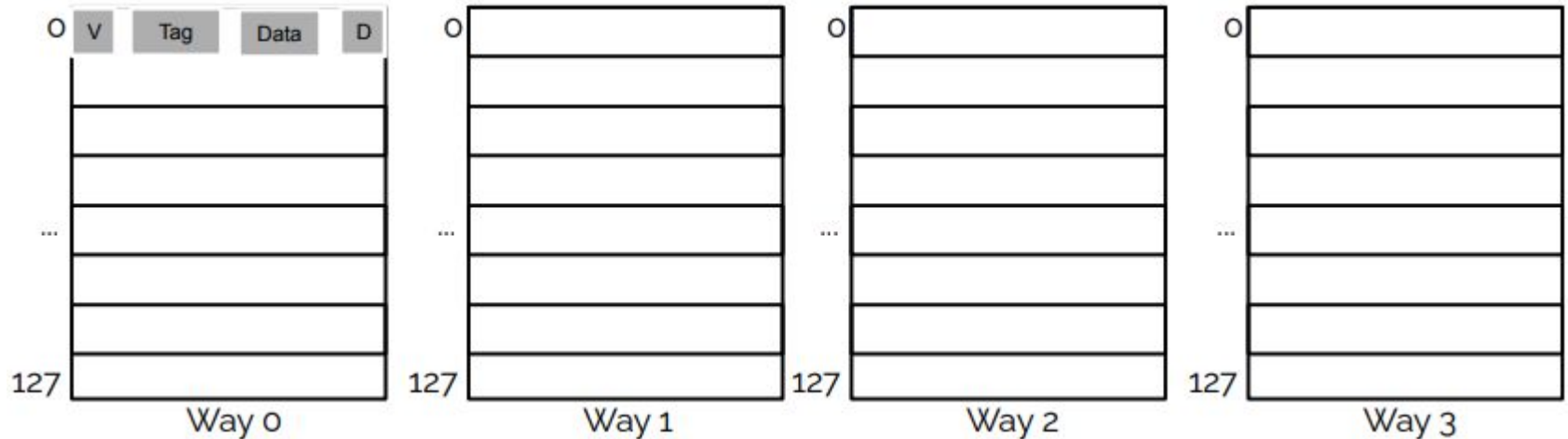
32KB 4-way set-associative data cache, 64 bytes per line



# n-Way Set-Associative Cache



32KB 4-way set-associative data cache, 64 bytes per line



# Congruent Addresses

Each memory address maps to one of these cache sets.

Memory addresses that map to the same cache set are called **congruent**.

Congruent addresses compete for cache lines within the same set, where replacement policy needs to decide which line will be replaced.

# Replacement Algorithm

Least recently used (LRU)

First in first out (FIFO)

Least frequently used (LFU)

Random

# Cache Side-Channel Attacks

Cache side-channel attacks utilize time differences between a cache hit and a cache miss to infer whether specific code/data has been accessed.

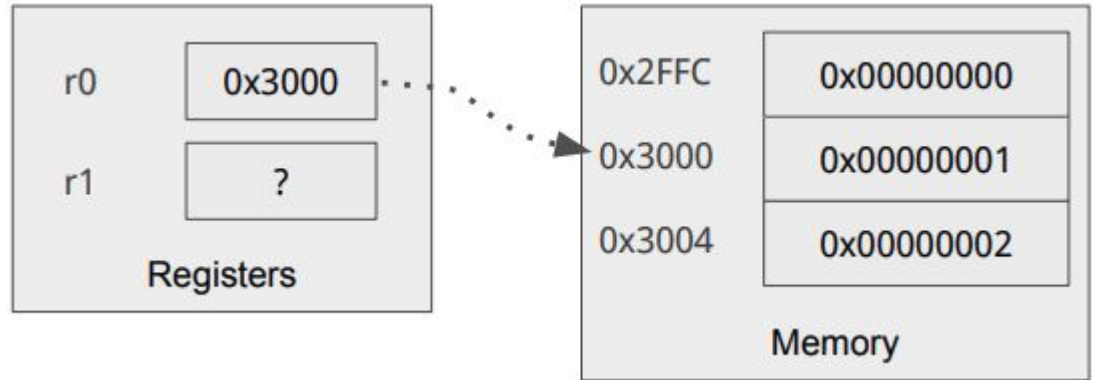


# Cache Side-Channel Attacks

; Assume r0 = 0x3000

; Load a word:

LDR r1, [r0]

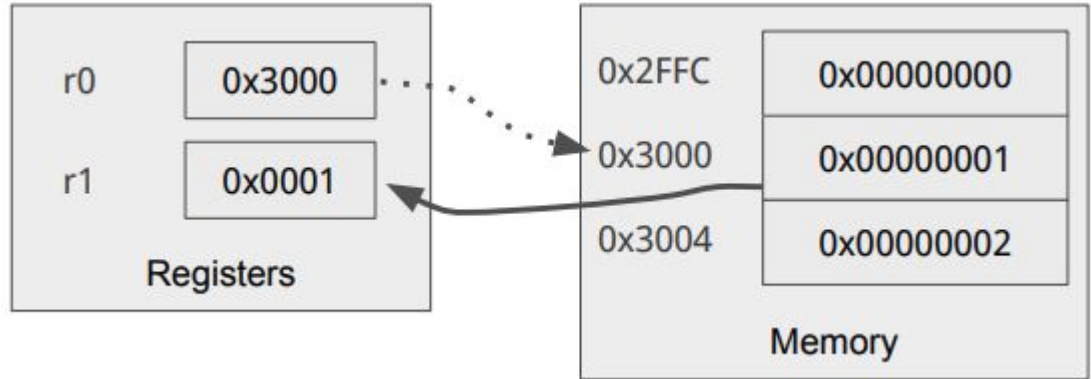


# Cache Side-Channel Attacks

; Assume r0 = 0x3000

; Load a word:

LDR r1, [r0]

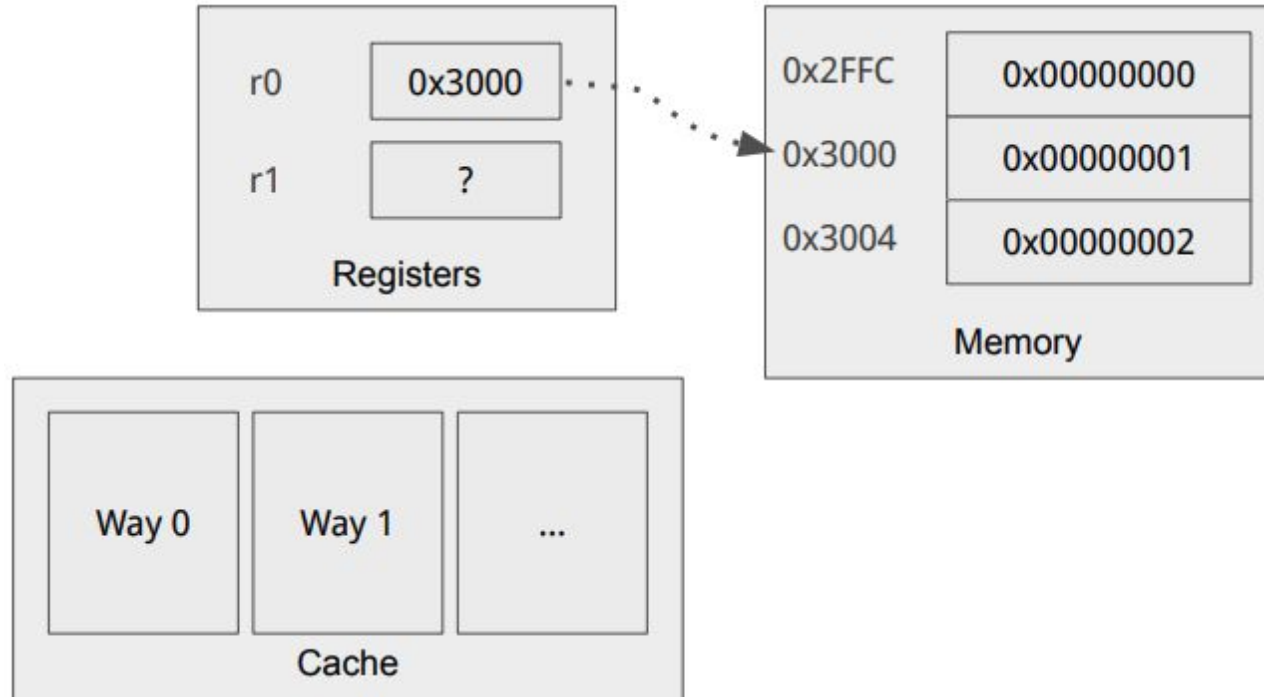


# Cache Side-Channel Attacks

; Assume r0 = 0x3000

; Load a word:

LDR r1, [r0]

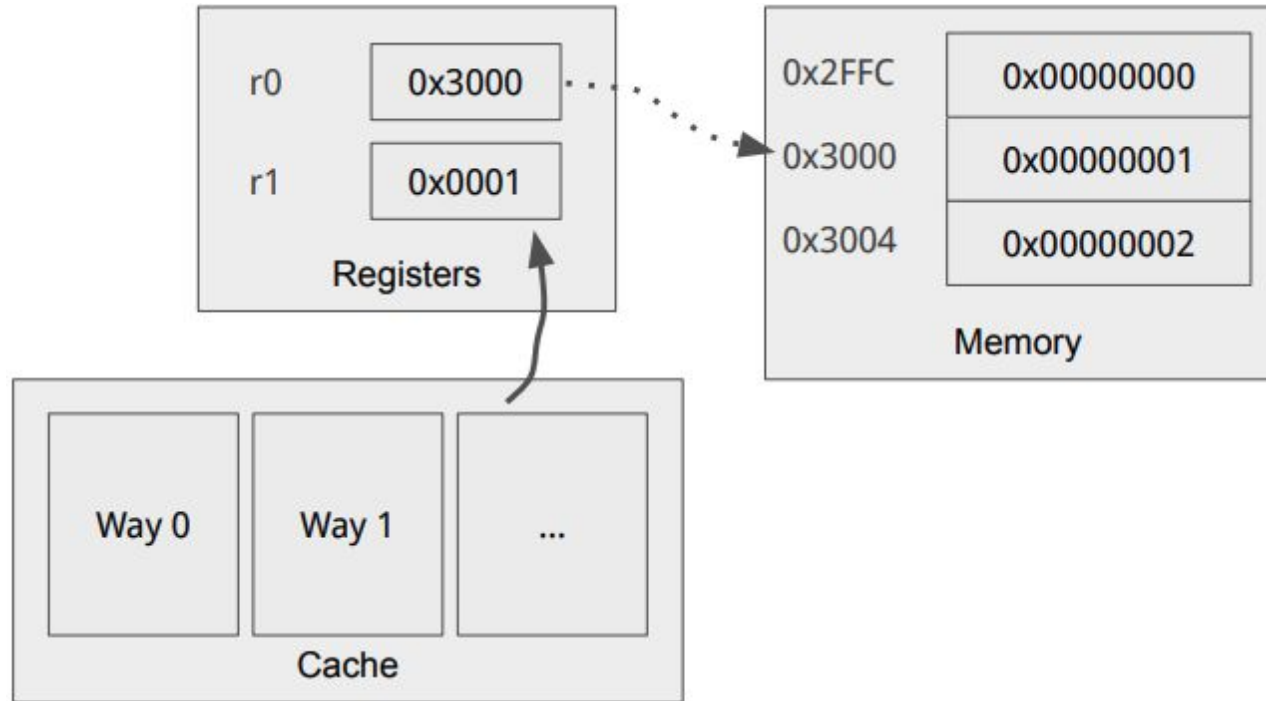


# Cache Side-Channel Attacks

; Assume r0 = 0x3000

; Load a word:

LDR r1, [r0]

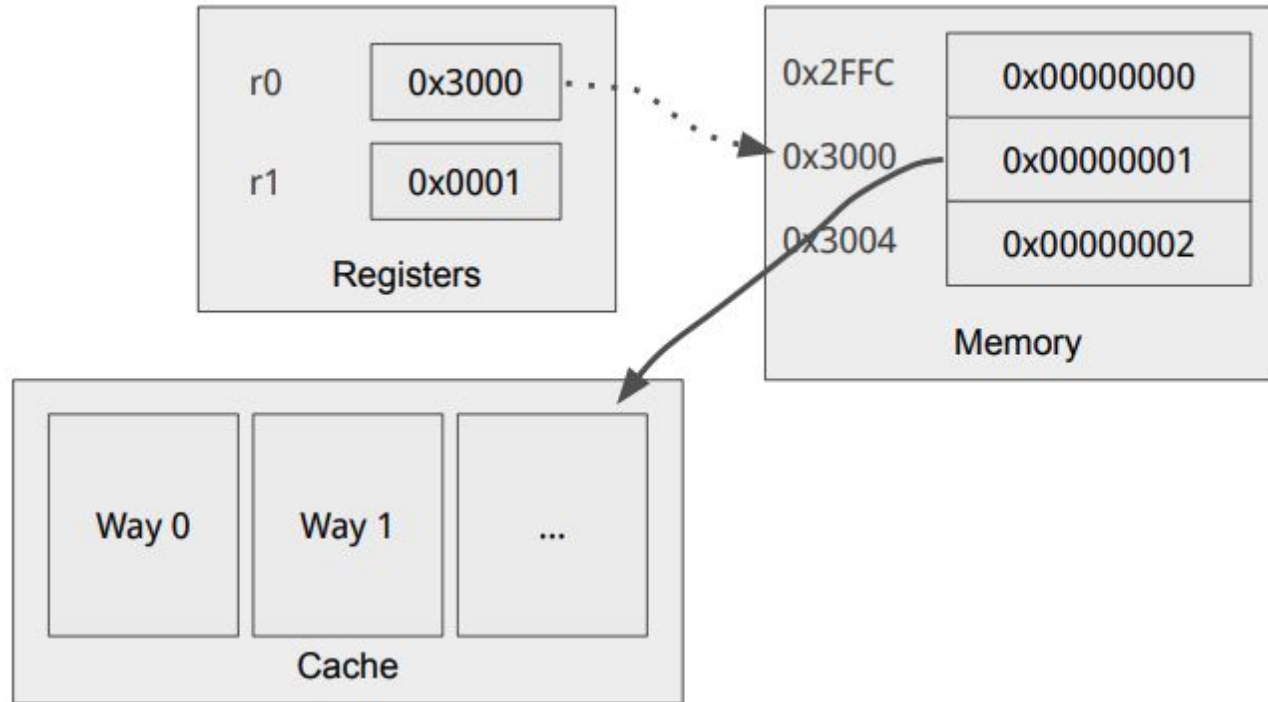


# Cache Side-Channel Attacks

; Assume r0 = 0x3000

; Load a word:

LDR r1, [r0]

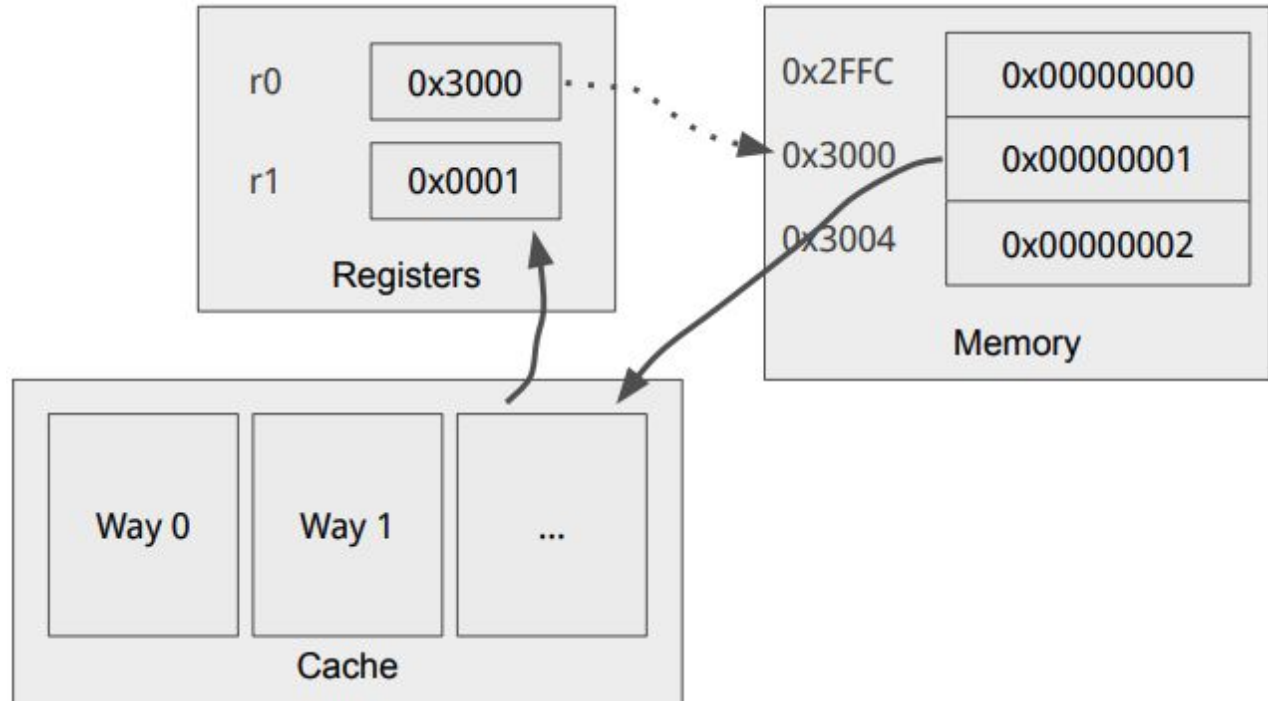


# Cache Side-Channel Attacks

; Assume r0 = 0x3000

; Load a word:

LDR r1, [r0]



# Cache Side-Channel Attacks

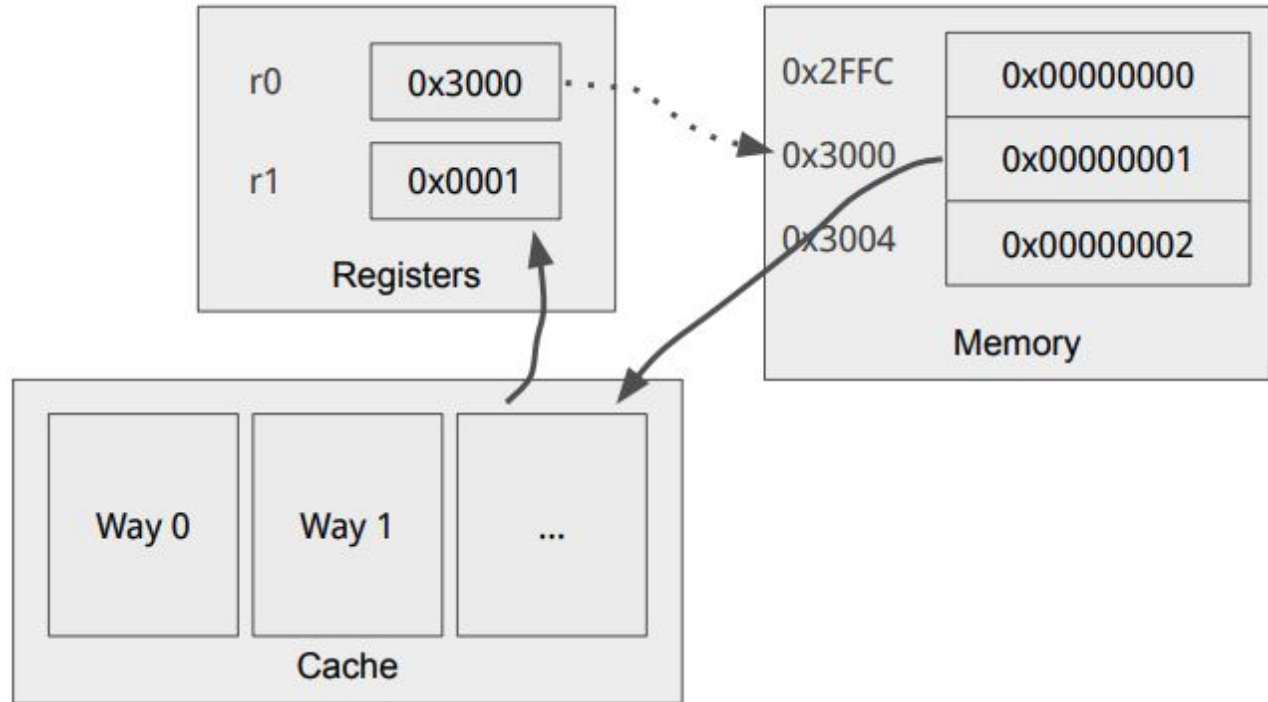
; Assume  $r0 = 0x3000$

; Load a word:

*;Get current time  $t1$*

LDR  $r1, [r0]$

*;Get current time  $t2$ ;  $t2 - t1$*



# Attack Primitives

Evict+Time

Prime+Probe

Flush+Flush

Flush+Reload

Evict+Reload



### 2.4.1 *Evict+Time*

In 2005 Percival [66] and Osvik et al. [63] proposed more fine-grained exploitations of memory accesses to the CPU cache. In particular, Osvik et al. formalized two concepts, namely *Evict+Time* and *Prime+Probe* that we will discuss in this and the following section. The basic idea is to determine which specific cache sets have been accessed by a victim program.

---

**Algorithm 1** *Evict+Time*

---

- 1: Measure execution time of victim program.
  - 2: Evict a specific cache set.
  - 3: Measure execution time of victim program again.
- 

The basic approach, outlined in Algorithm 1, is to determine which cache set is used during the victim's computations. At first, the execution time of the victim program is measured. In the second step, a specific cache set is evicted before the program is measured a second time in the third step. By means of the timing difference between the two measurements, one can deduce how much the specific cache set is used while the victim's program is running.

Osvik et al. [63] and Tromer et al. [81] demonstrated with *Evict+Time* a powerful type of attack against **AES** on OpenSSL implementations that requires neither knowledge of the plaintext nor the ciphertext.

# Prime+Probe

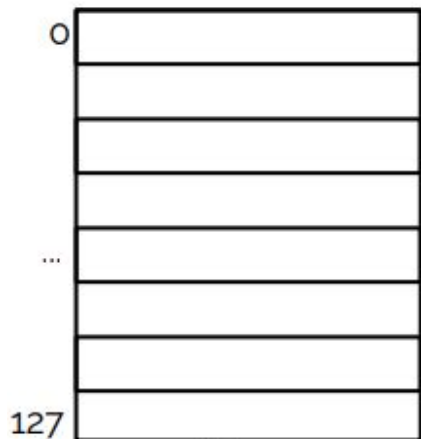
Step 1 Prime: Attacker occupies a set



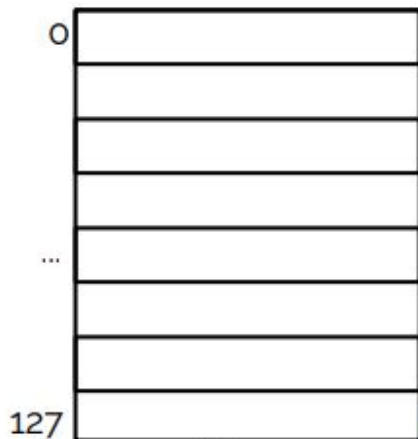
Attacker Address Space



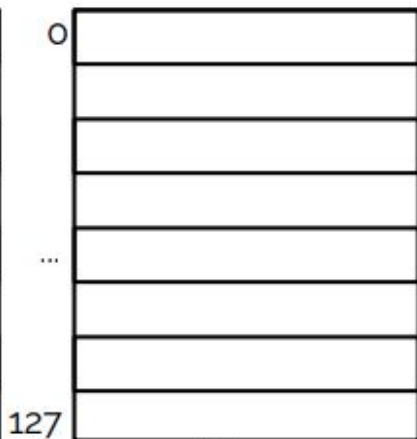
Victim Address Space



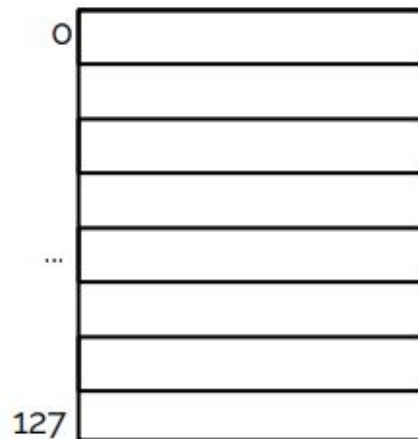
Way 0



Way 1



Way 2



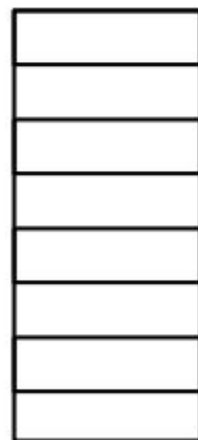
Way 3

# Prime+Probe

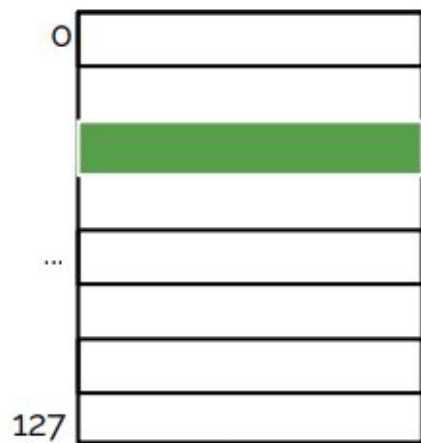
Step 1 Prime: Attacker occupies a set



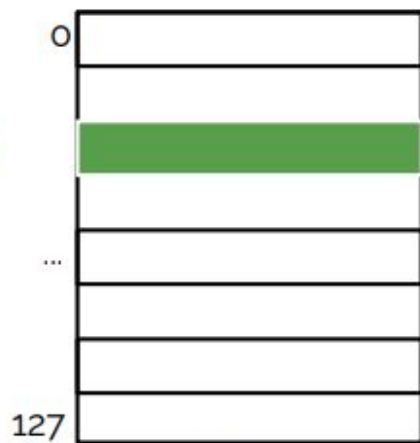
Attacker Address Space



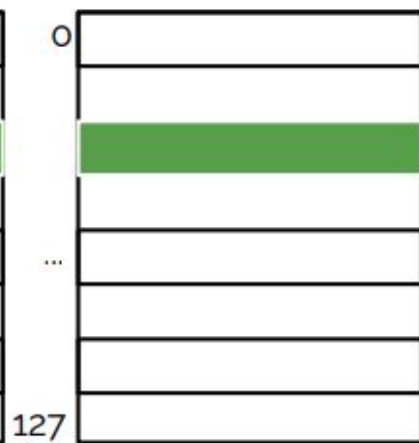
Victim Address Space



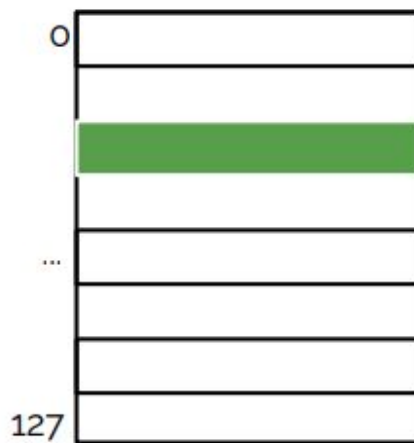
Way 0



Way 1



Way 2



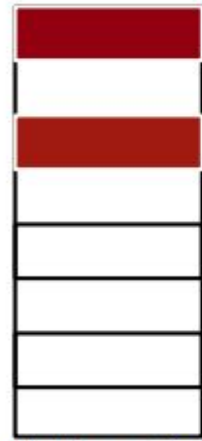
Way 3

# Prime+Probe

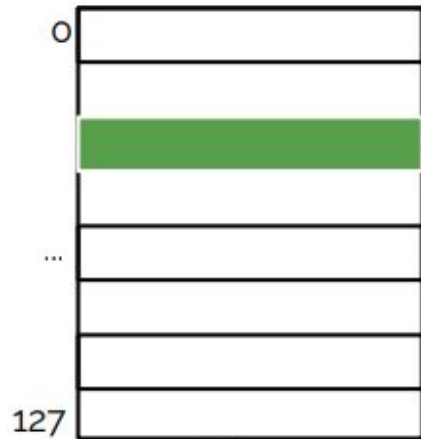
Step 2: Victim runs



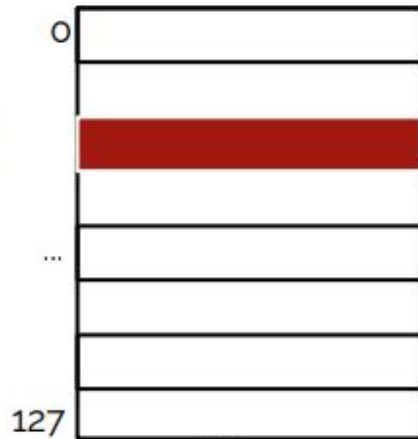
Attacker Address Space



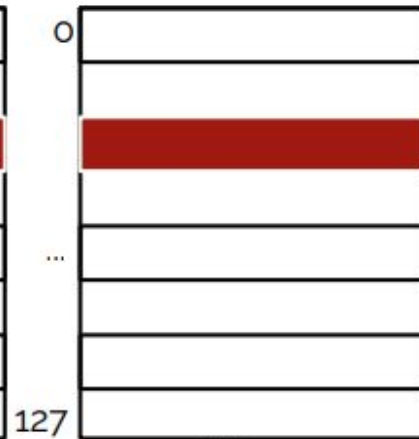
Victim Address Space



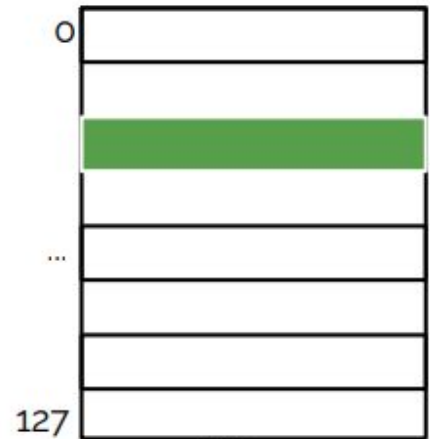
Way 0



Way 1



Way 2



Way 3

# Flush+Reload

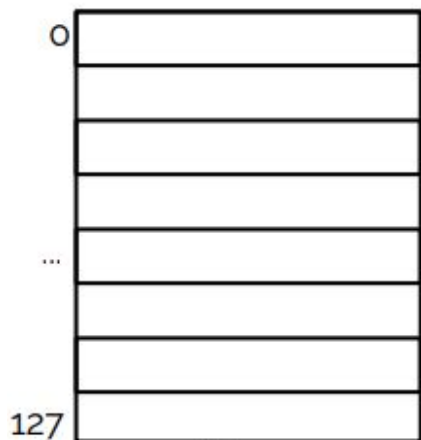
Step 1 Flush: Attacker flushes this memory block out of cache



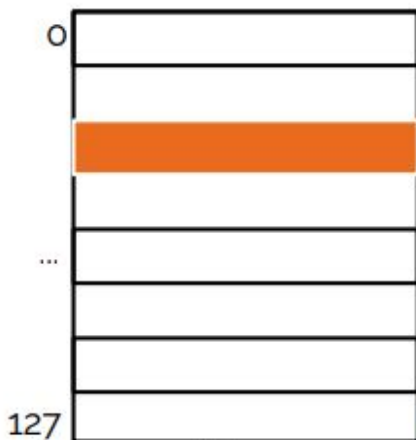
Attacker Address Space



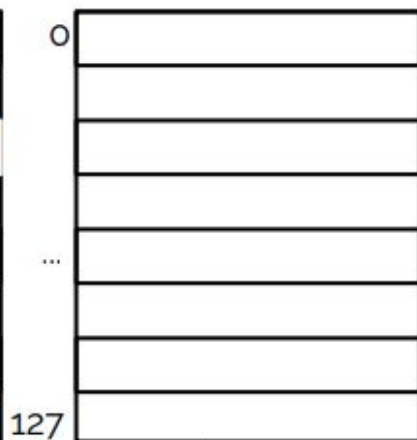
Victim Address Space



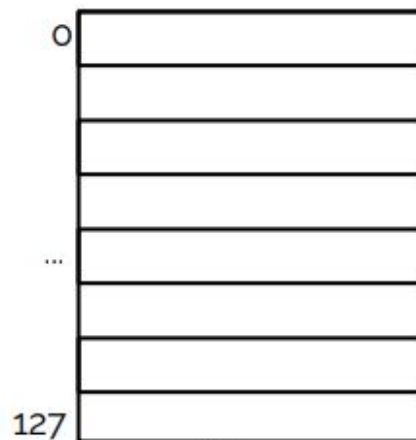
Way 0



Way 1



Way 2



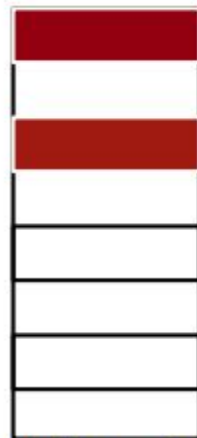
Way 3

# Prime+Probe

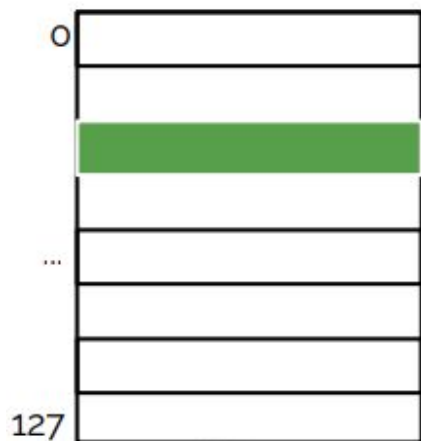
Step 3 Probe: Attacker accesses memory again and measures the time



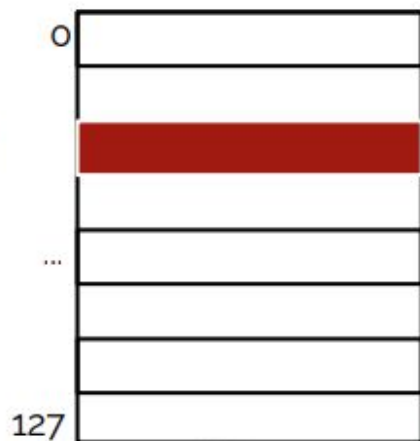
Attacker Address Space



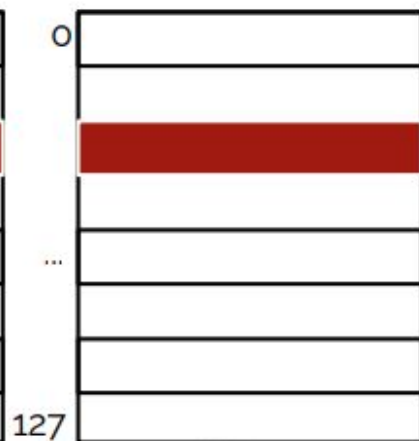
Victim Address Space



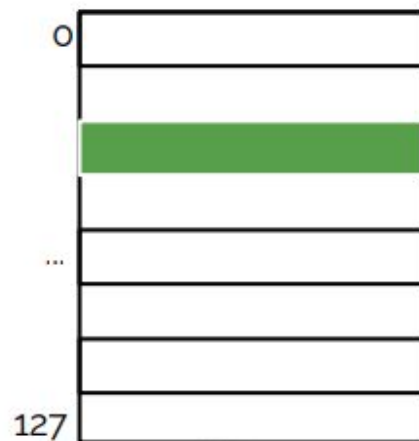
Way 0



Way 1



Way 2



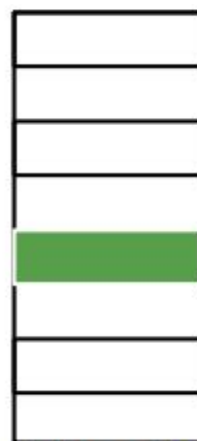
Way 3

# Flush+Reload

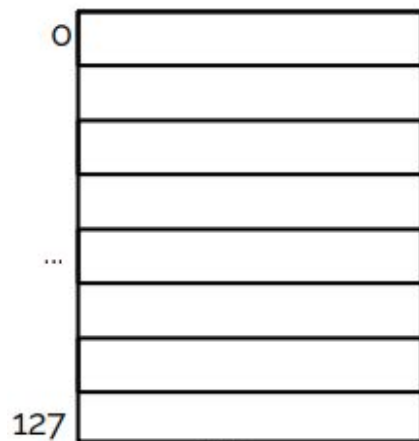
A memory block is cached



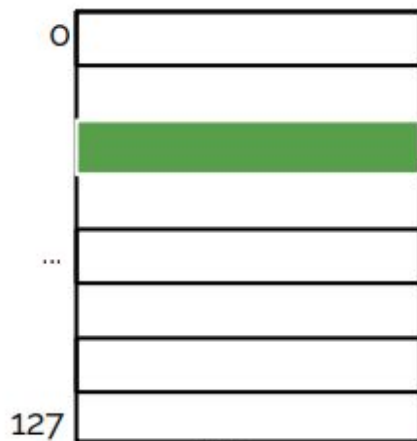
Attacker Address Space



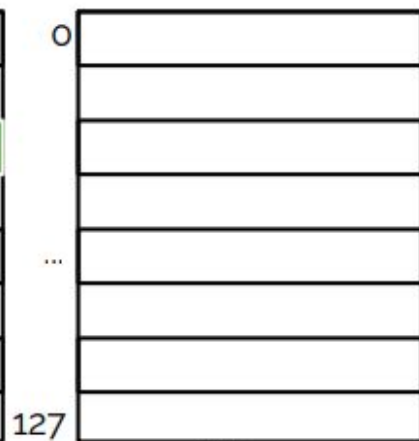
Victim Address Space



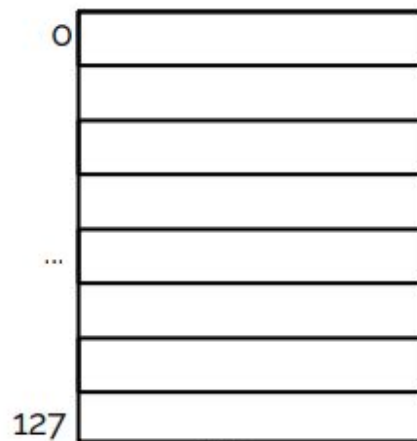
Way 0



Way 1



Way 2



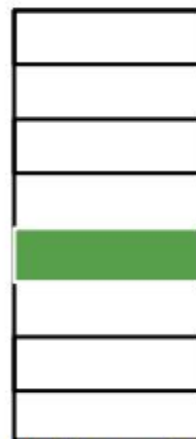
Way 3

# Flush+Reload

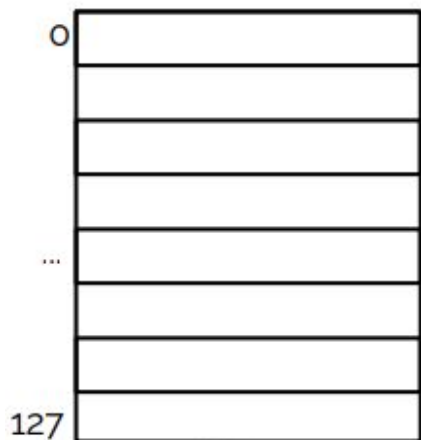
Step 2 Reload: Victim may / may not access that block again



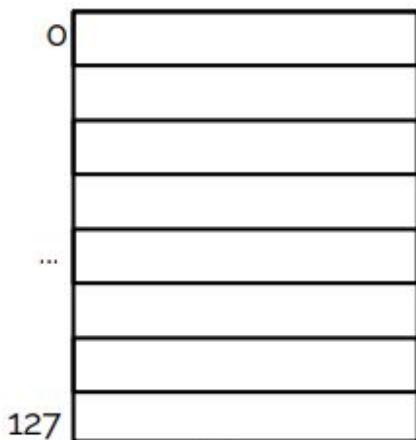
Attacker Address Space



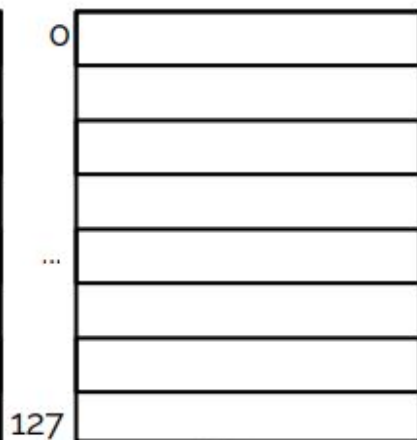
Victim Address Space



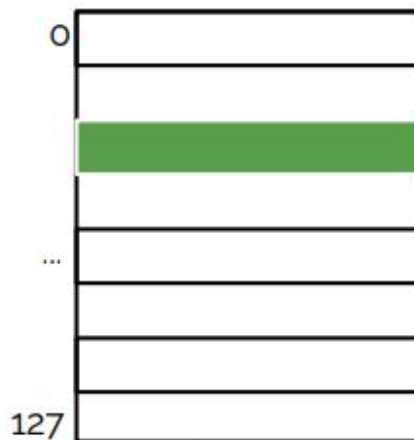
Way 0



Way 1



Way 2



Way 3

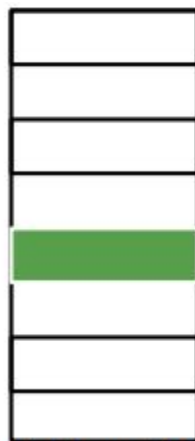


# Flush+Reload

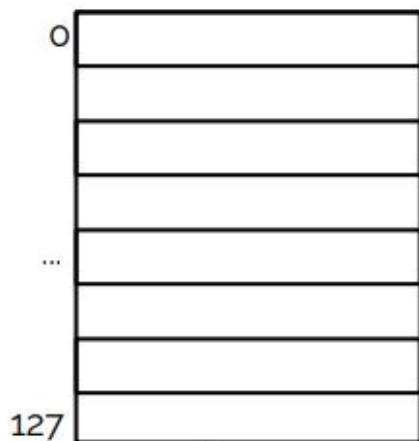
Step 3 Probe: Attacker accesses that block again and measure



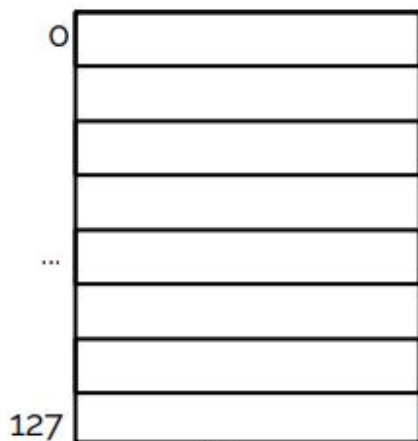
Attacker Address Space



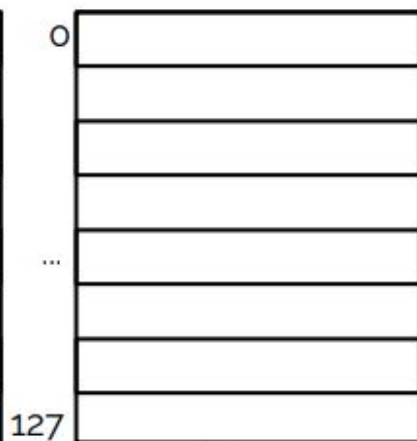
Victim Address Space



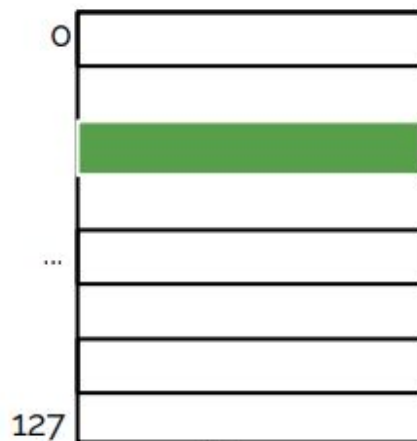
Way 0



Way 1



Way 2



Way 3

# Cachetime.c

## from SEED labs

```
uint8_t array[10*4096];

int main(int argc, const char **argv) {
    int junk=0;
    register uint64_t time1, time2;
    volatile uint8_t *addr;
    int i;

    // Initialize the array
    for(i=0; i<10; i++) array[i*4096]=1;

    // FLUSH the array from the CPU cache
    for(i=0; i<10; i++) _mm_clflush(&array[i*4096]);

    // Access some of the array items
    array[3*4096] = 200;
    array[7*4096] = 200;

    for(i=0; i<10; i++) {
        addr = &array[i*4096];
        time1 = __rdtscp(&junk);
        junk = *addr;
        time2 = __rdtscp(&junk) - time1;
        printf("Access time for array[%d*4096]: %d CPU cycles\n",i, (int)time2);
    }
    return 0;
}
```

# Flush\_reload.c from SEED labs

```
gcc -march=native cachetime.c
```

```
[11/23/20]seed@VM:~$ lscpu
Architecture:          i686
CPU op-mode(s):        32-bit
Byte Order:            Little Endian
CPU(s):                2
On-line CPU(s) list:   0,1
Thread(s) per core:    1
Core(s) per socket:    2
Socket(s):             1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                126
Model name:            Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz
Stepping:              5
CPU MHz:              1497.600
BogoMIPS:              2995.20
Hypervisor vendor:     KVM
Virtualization type:   full
L1d cache:            48K
L1i cache:            32K
L2 cache:             512K
L3 cache:             8192K
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
cmov pat pse36 clflush mmx fxsr sse sse2 ht nx rdtscp constant_tsc xtopology non
```

