# CSE 410/510 Special Topics: Software Security

Instructor: Dr. Ziming Zhao

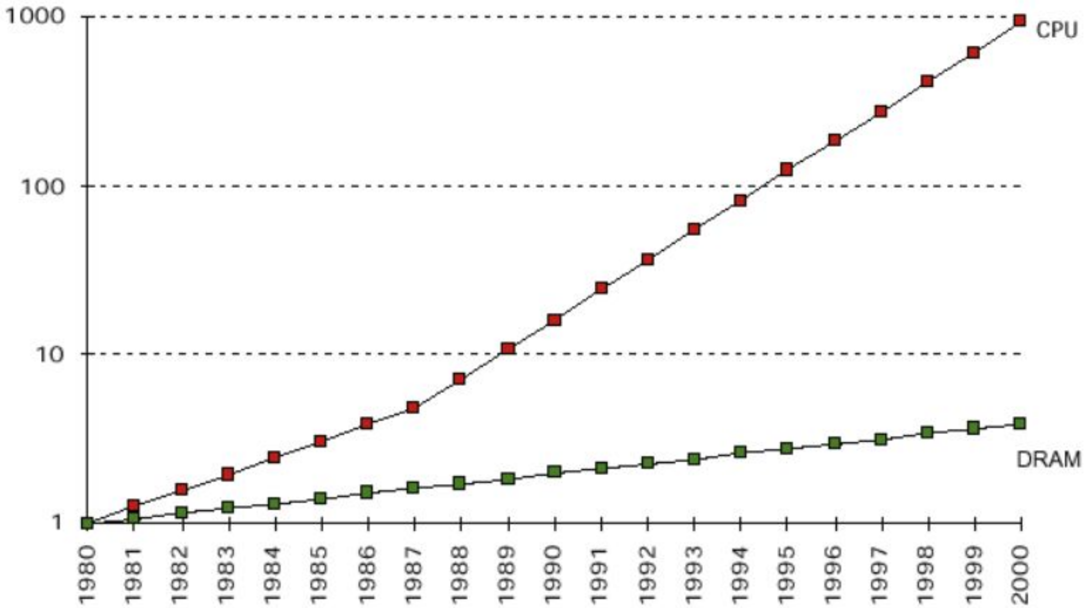
Location: Norton 218

Time: Monday, 5:00 PM - 7:50 PM

# Today’s Agenda

1. Cache side channel attack
2. Meltdown
3. 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* | Static RAM | 0.5 - 2.5 ns |
| *Memory* | Dynamic RAM | 50- 70 ns |
| *Secondary* | Flash | 5,000 - 50,000 ns |
| | Magnetic disks | 5,000,000 - 20,000,000 ns |

# Cache Hits and Misses

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

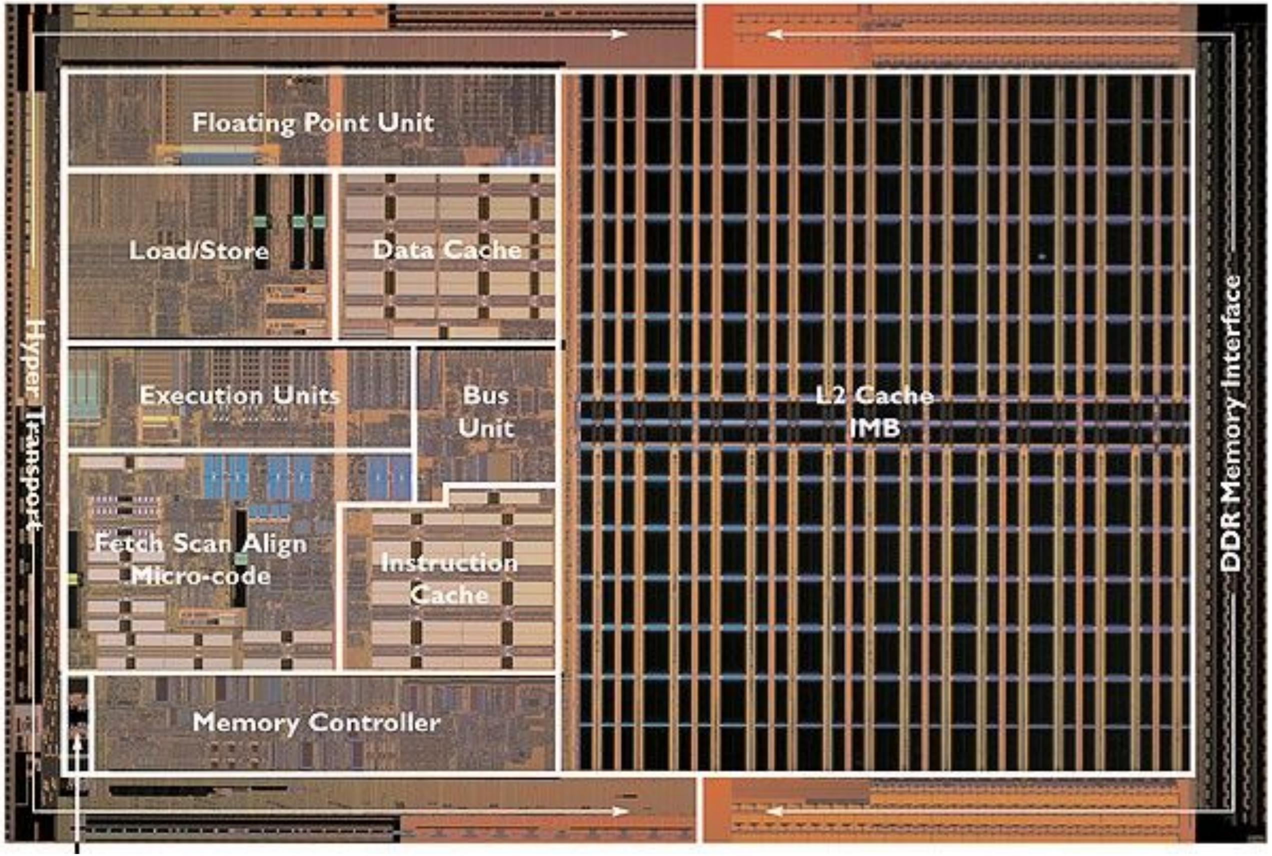
A cache miss occurs if the cache does not contain the requested data.

# Cache Hierarchy

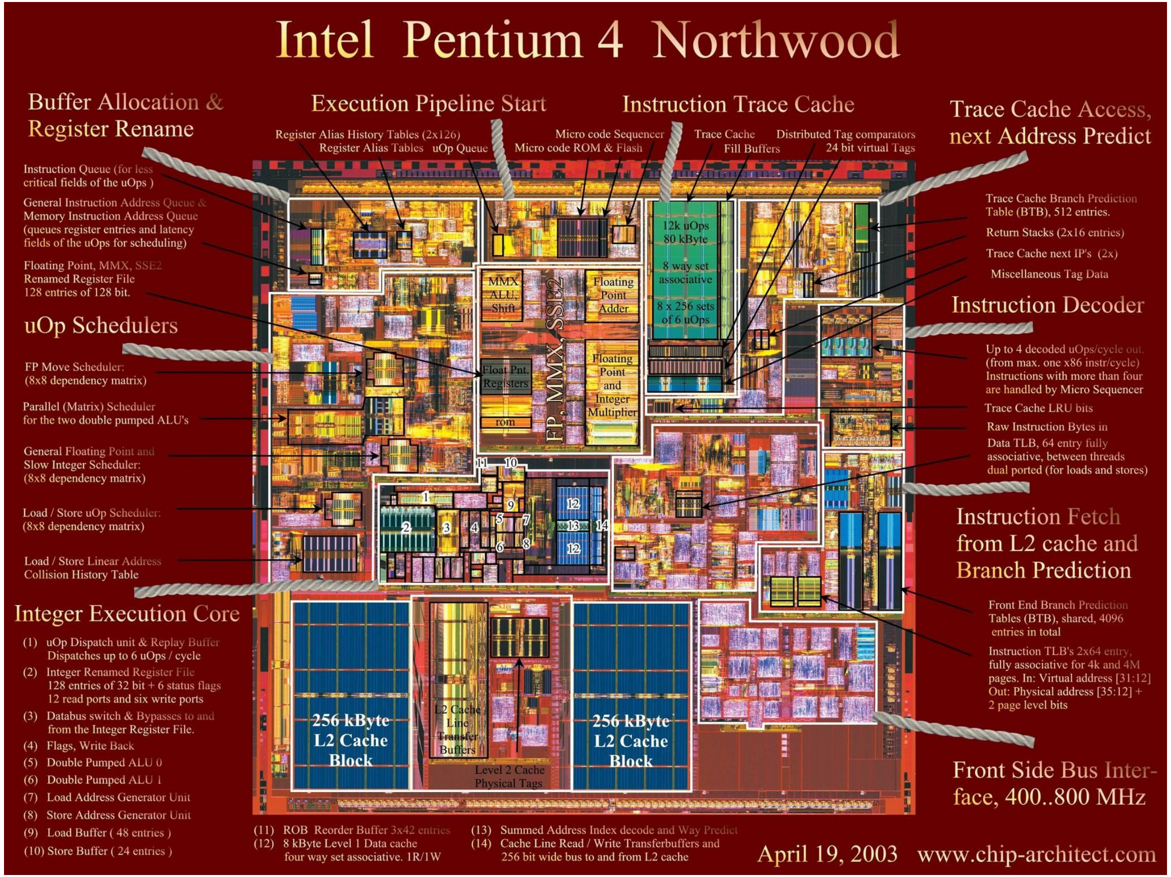
L1 Cache is closest to the CPU. Usually divided in Code and Data cache

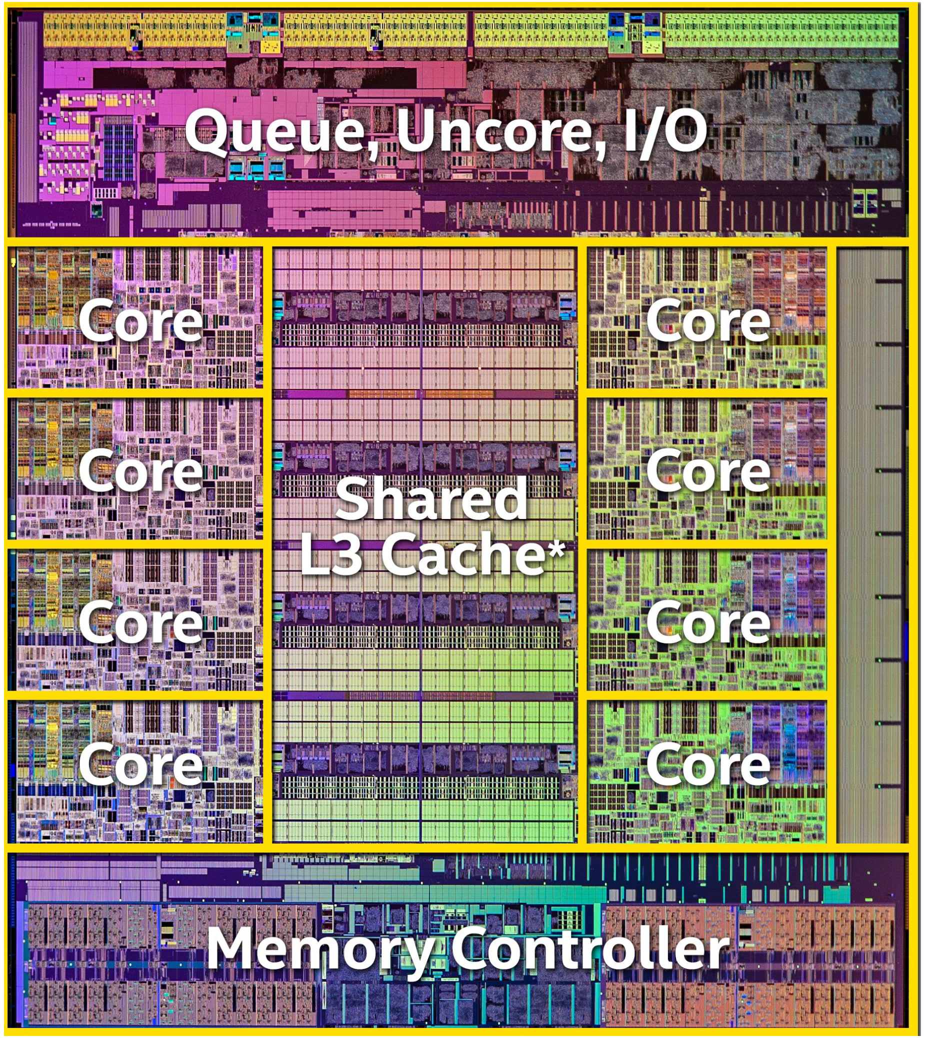
L2 and L3 cache are usually unified.

# Cache Hierarchy



# Cache Hierarchy




Queue, Uncore, I/O
Core
Core
Core
Core
Shared
L3 Cache*
Core
Core
Core
Core
Memory Controller

# Cache Line/Block

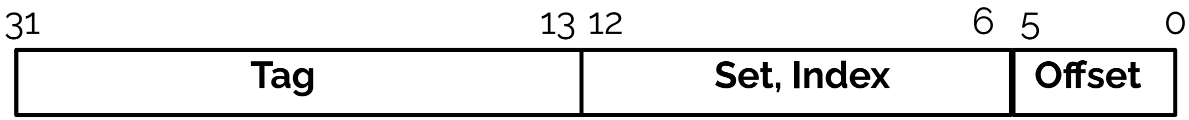
The minimum unit of information that can be either present or not present in a cache.

64 bytes in modern Intel and ARM CPUs

# *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

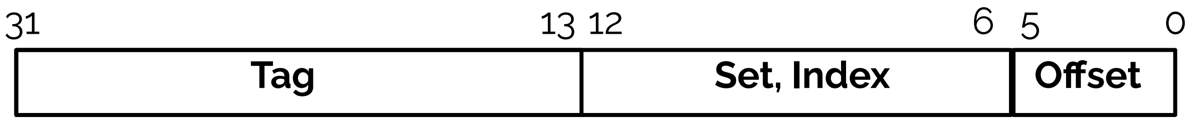
Number of sets

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

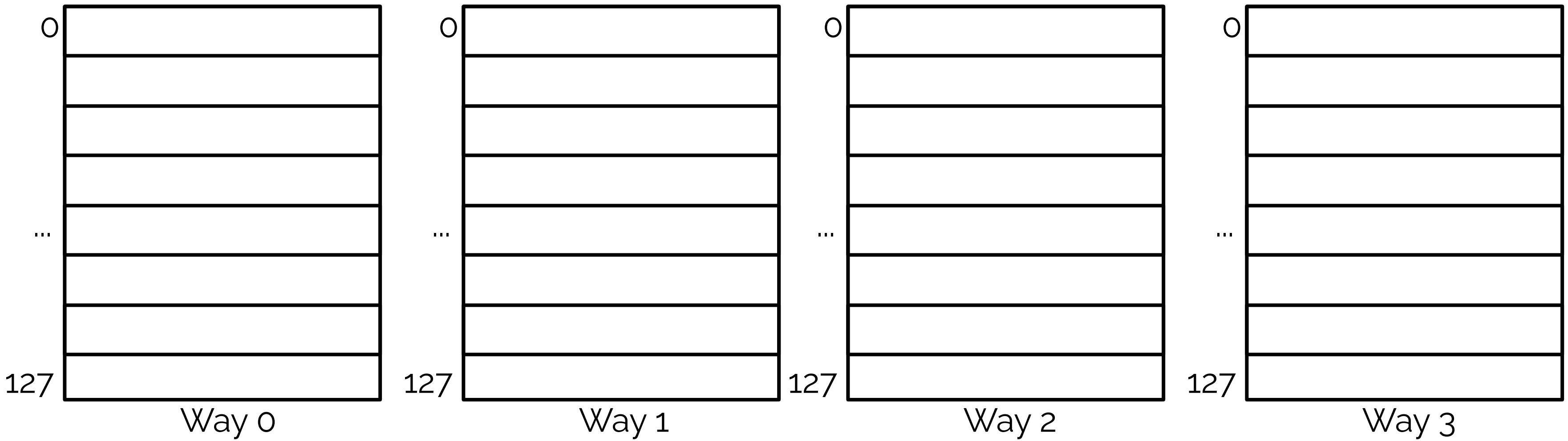
= 32 * 1024 / (4 * 64)

= 128

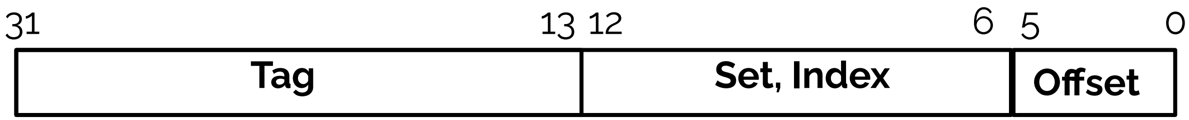
# *n*-Way Set-Associative Cache



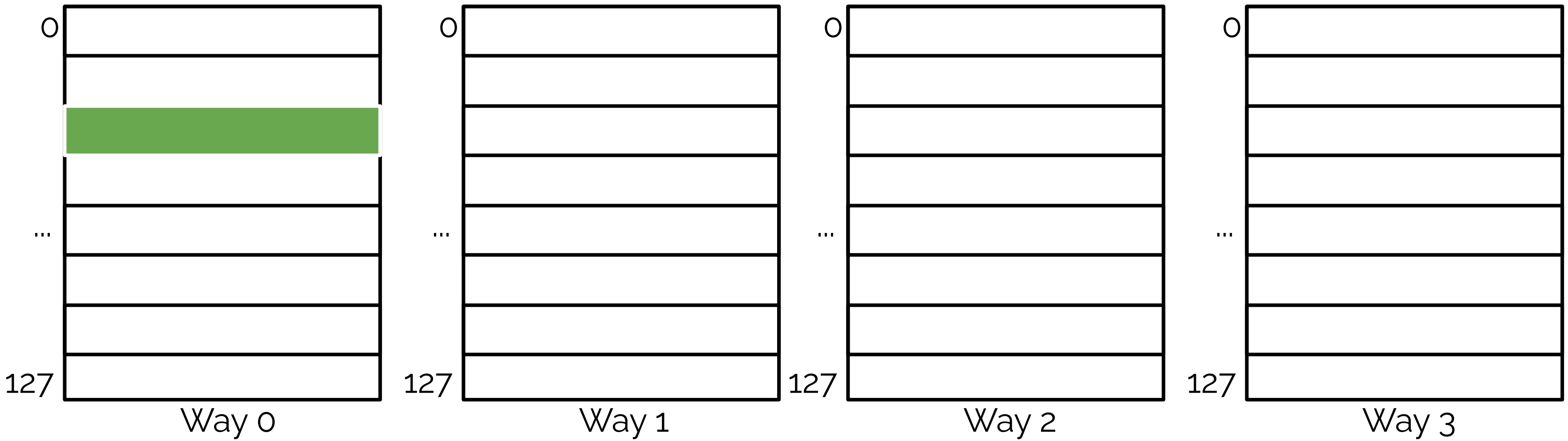

32KB 4-way set-associative data cache, 64 bytes per line



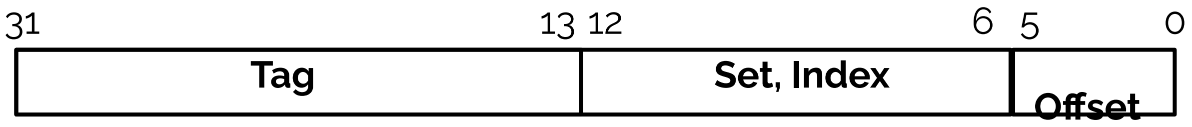
# *n*-Way Set-Associative Cache



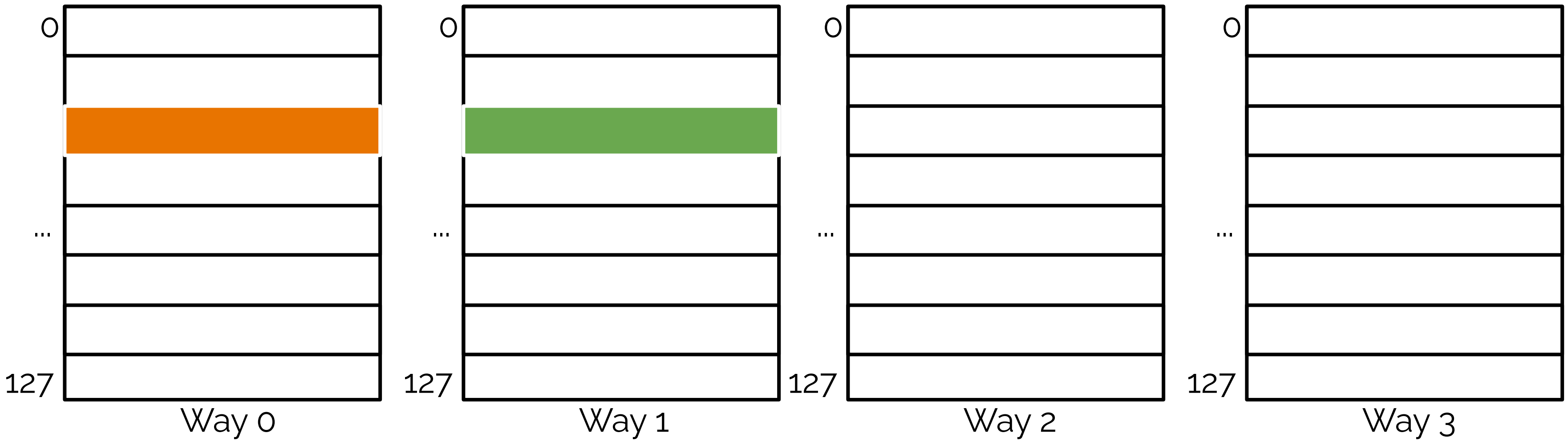

32KB 4-way set-associative data cache, 64 bytes per line



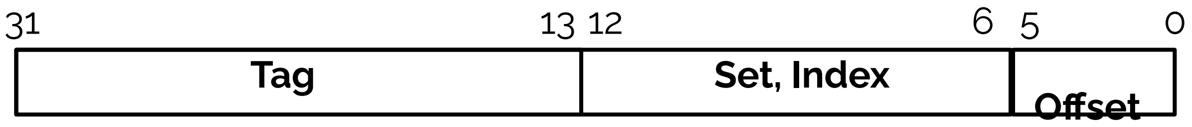
# *n*-Way Set-Associative Cache



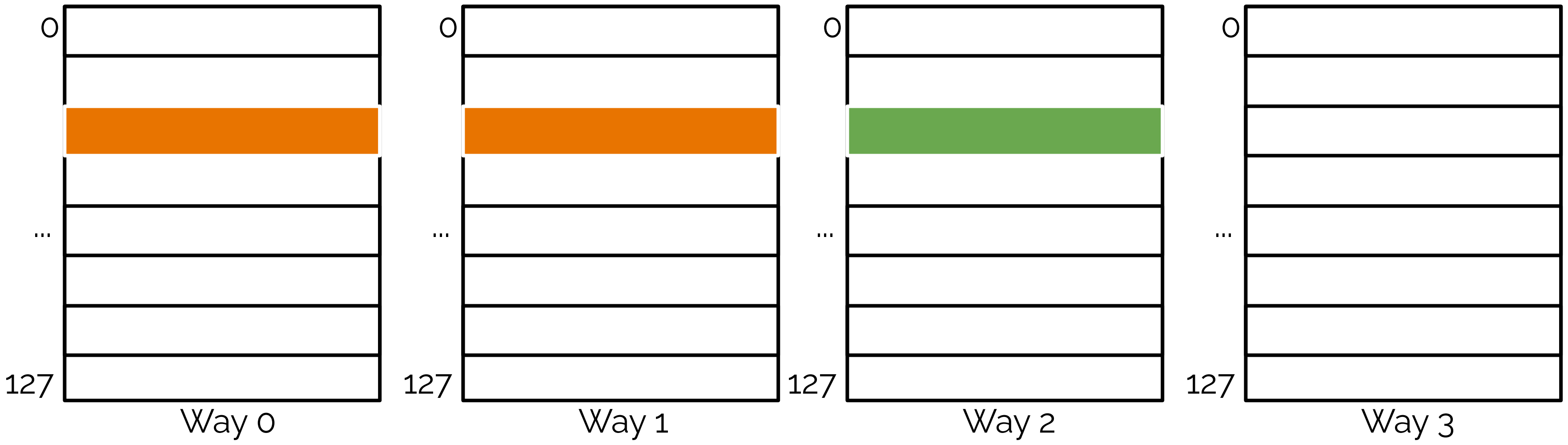

32KB 4-way set-associative data cache, 64 bytes per line



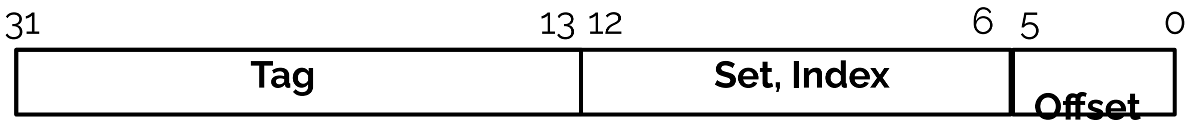
# *n*-Way Set-Associative Cache



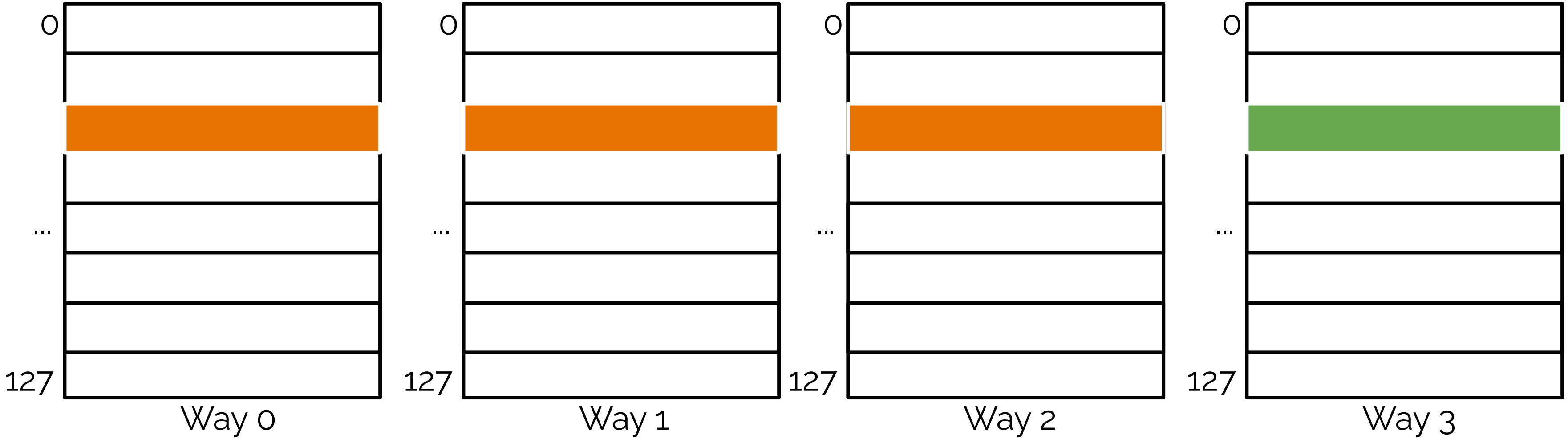

32KB 4-way set-associative data cache, 64 bytes per line



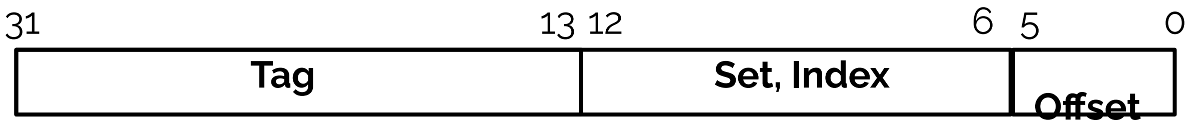
# *n*-Way Set-Associative Cache



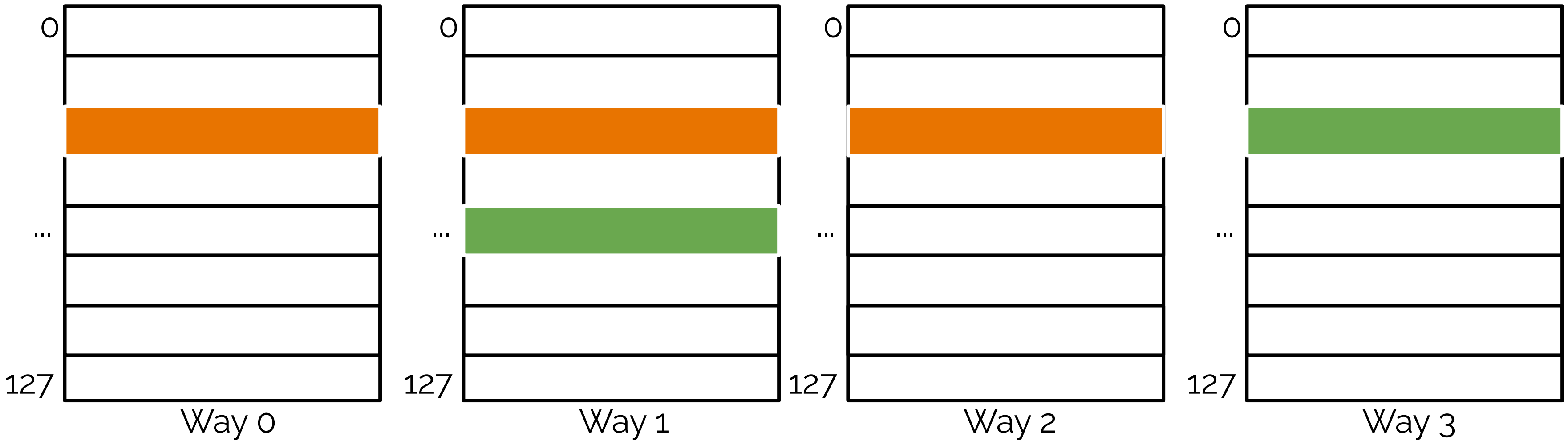

32KB 4-way set-associative data cache, 64 bytes per line



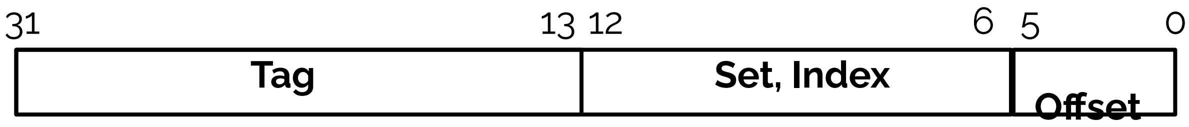
# *n*-Way Set-Associative Cache



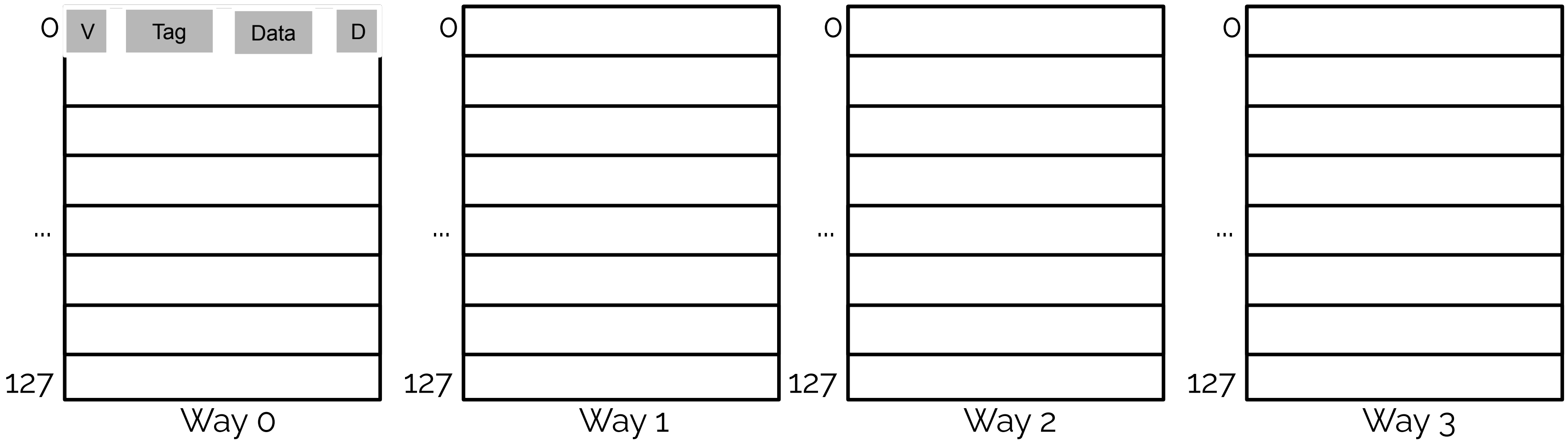

32KB 4-way set-associative data cache, 64 bytes per line



# Cache Line/Block Content




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

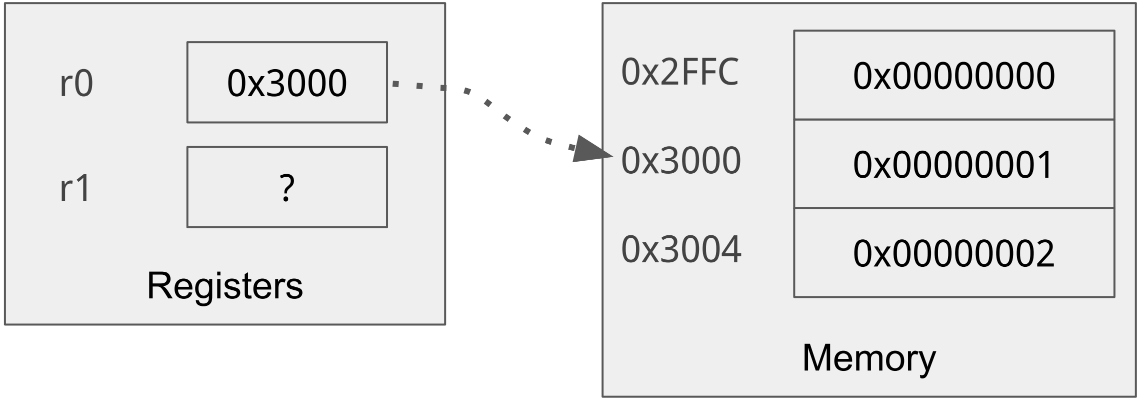
Cache side-channel attacks utilize time differences between a cache hit and a cache miss to infer whether specific code/data has been accessed.

# Cache Side-Channel Attack

; Assume r0 = 0x3000

; Load a word:

LDR r1, [r0]

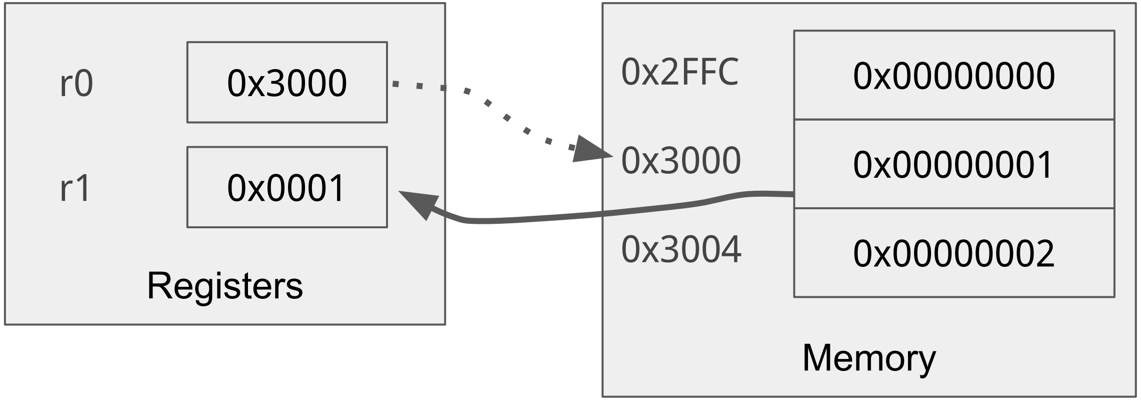


# Cache Side-Channel Attack

; Assume r0 = 0x3000

; Load a word:

LDR r1, [r0]

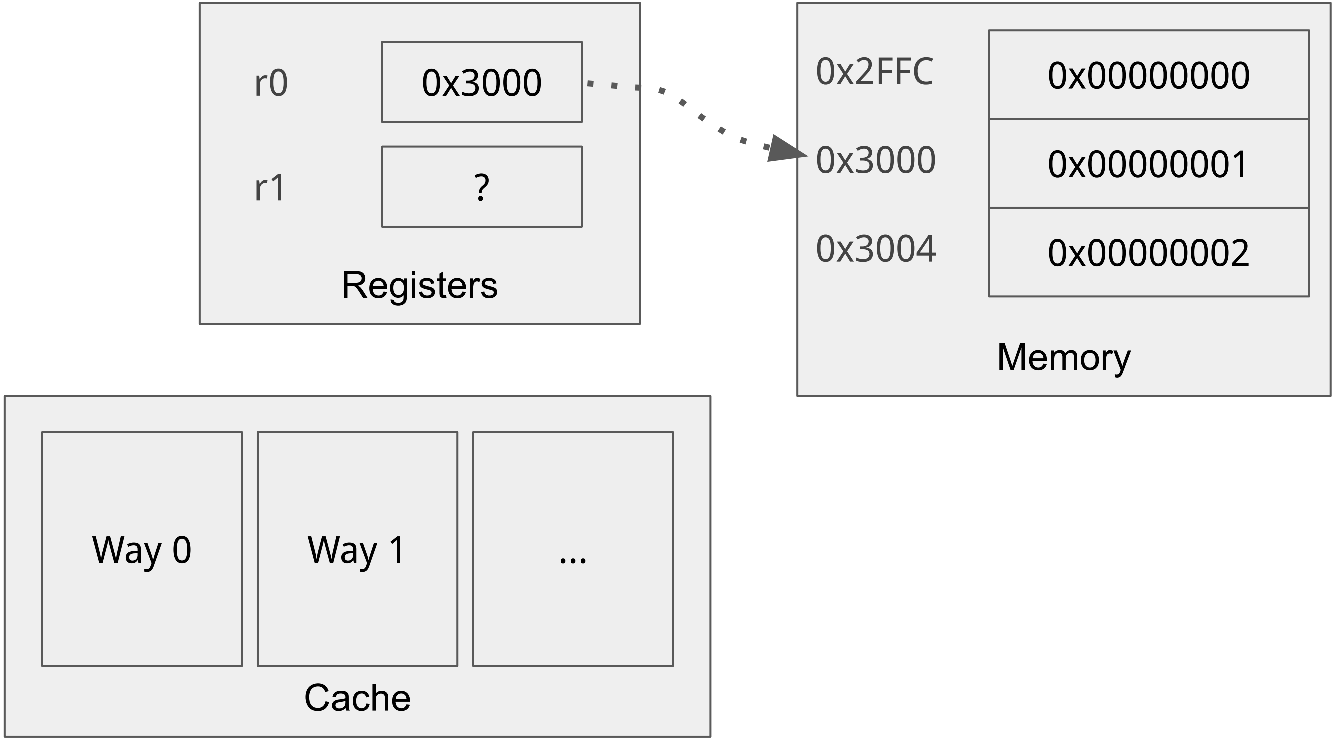


# Cache Side-Channel Attack

; Assume r0 = 0x3000

; Load a word:

```
LDR r1, [r0]
```

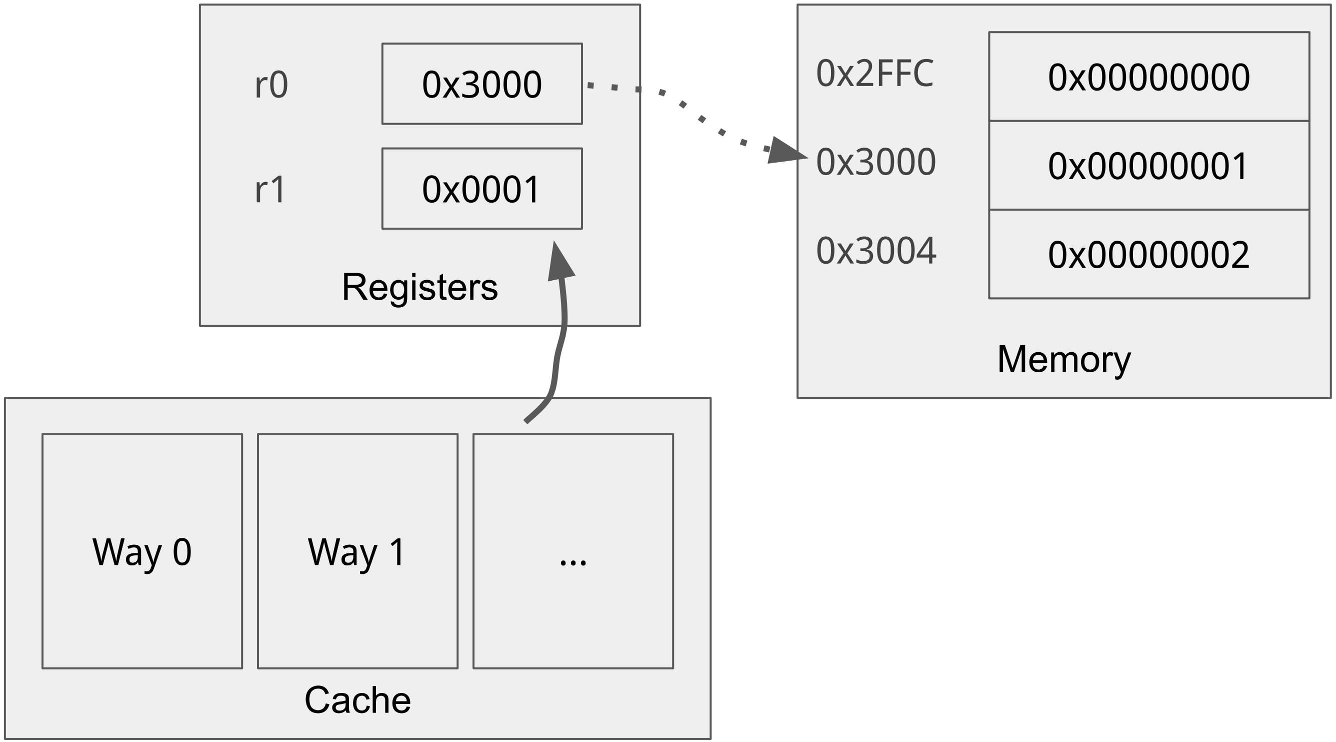


# Cache Side-Channel Attack

```
; Assume r0 = 0x3000

; Load a word:

LDR r1, [r0]
```

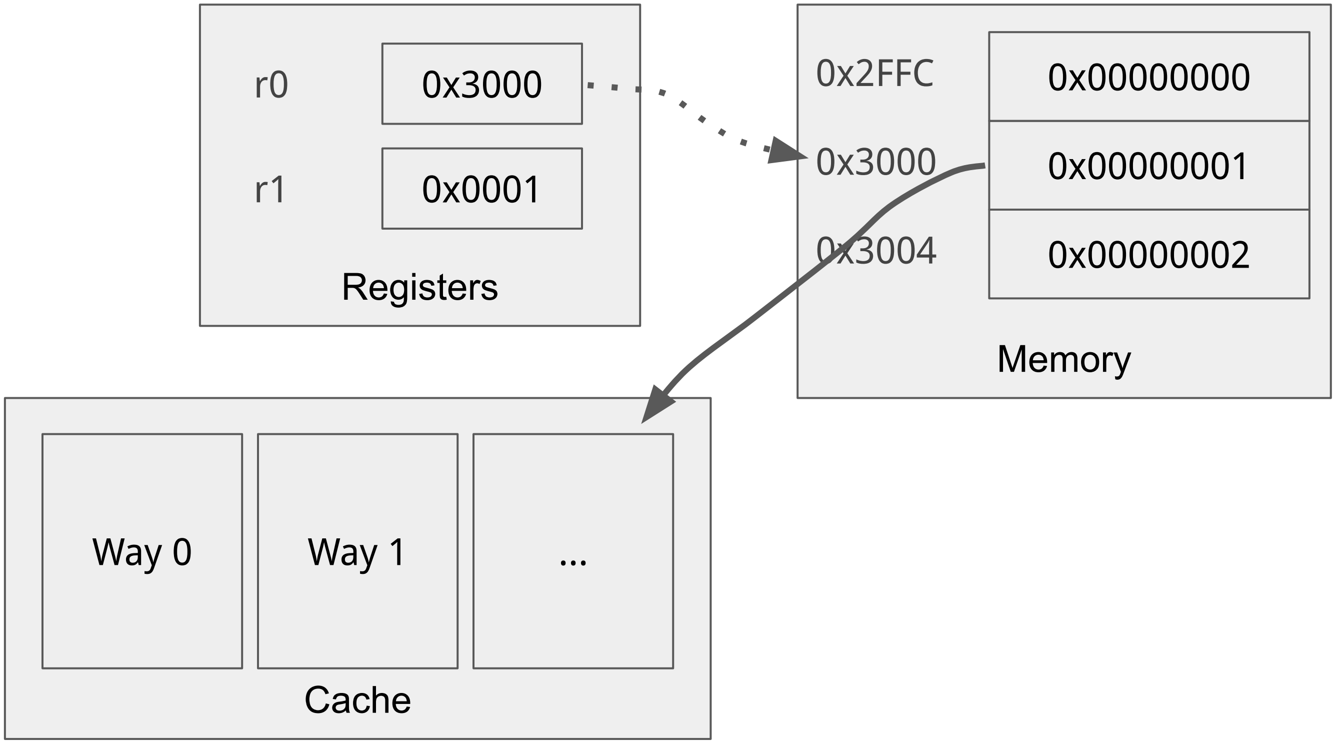


# Cache Side-Channel Attack

; Assume r0 = 0x3000

; Load a word:

LDR r1, [r0]

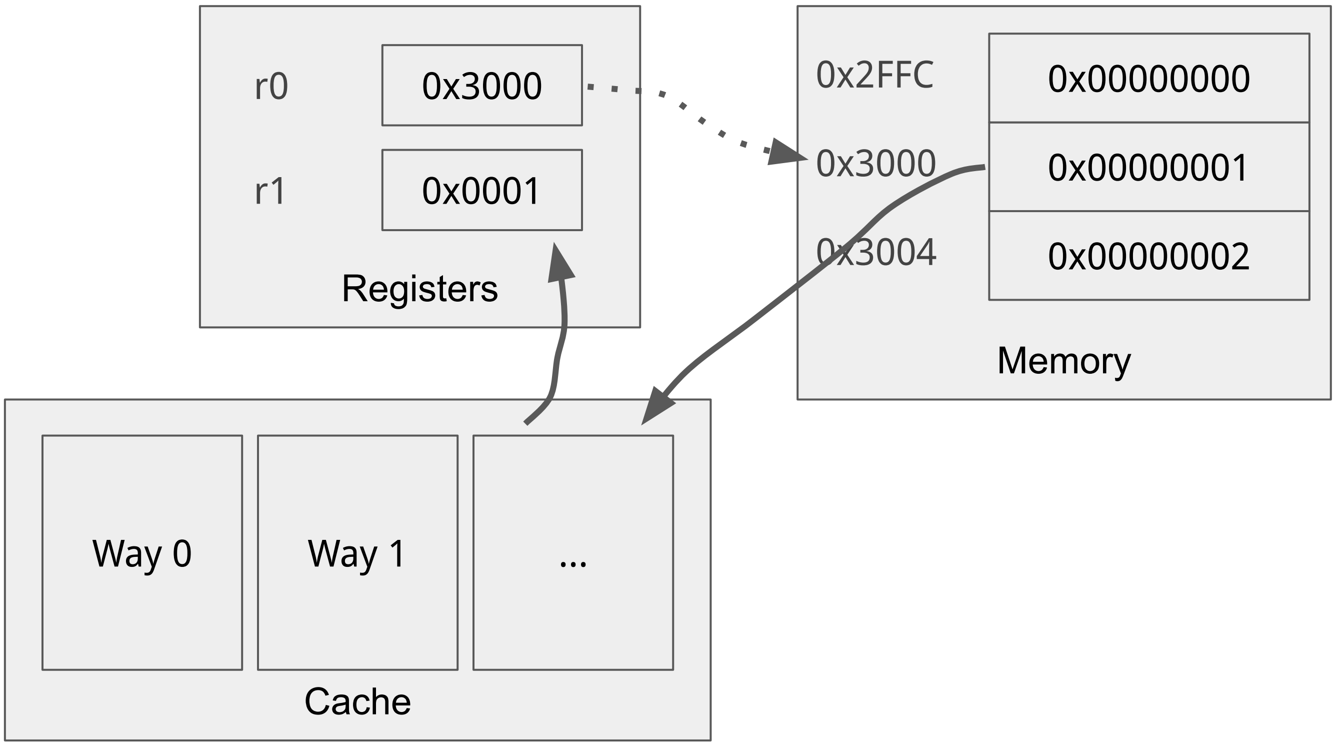


# Cache Side-Channel Attack

; Assume r0 = 0x3000

; Load a word:

LDR r1, [r0]



# Cache Side-Channel Attack

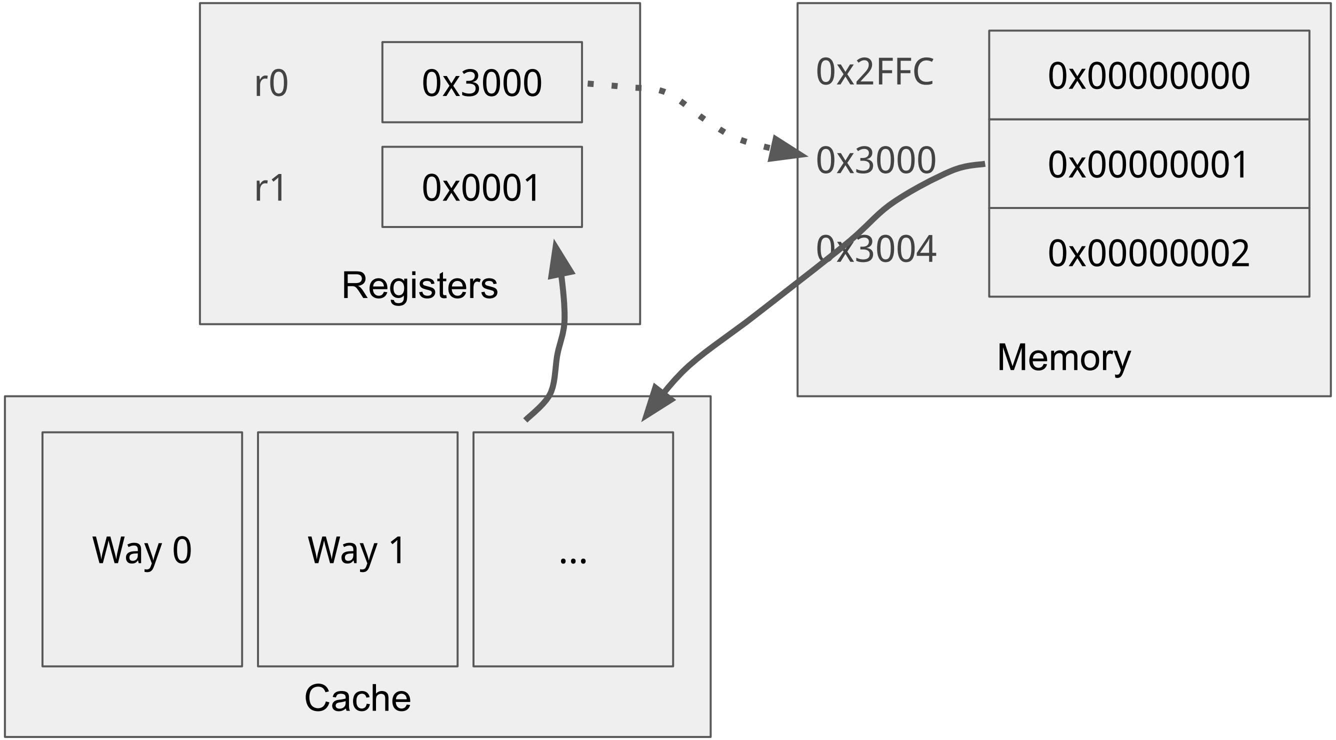
; Assume r0 = 0x3000

; 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.

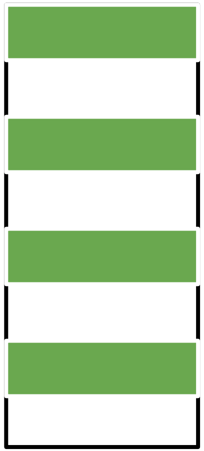
---

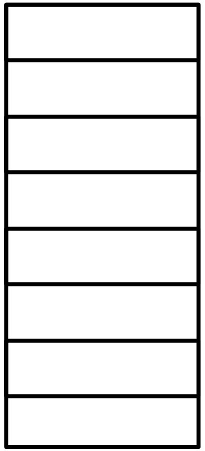
The basic approach, outlined in Algorithm 1, is to determine which cache set is used during the victim's computations. At first, the execution time of the victim program is measured. In the second step, a specific cache set is evicted before the program is measured a second time in the third step. By 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.

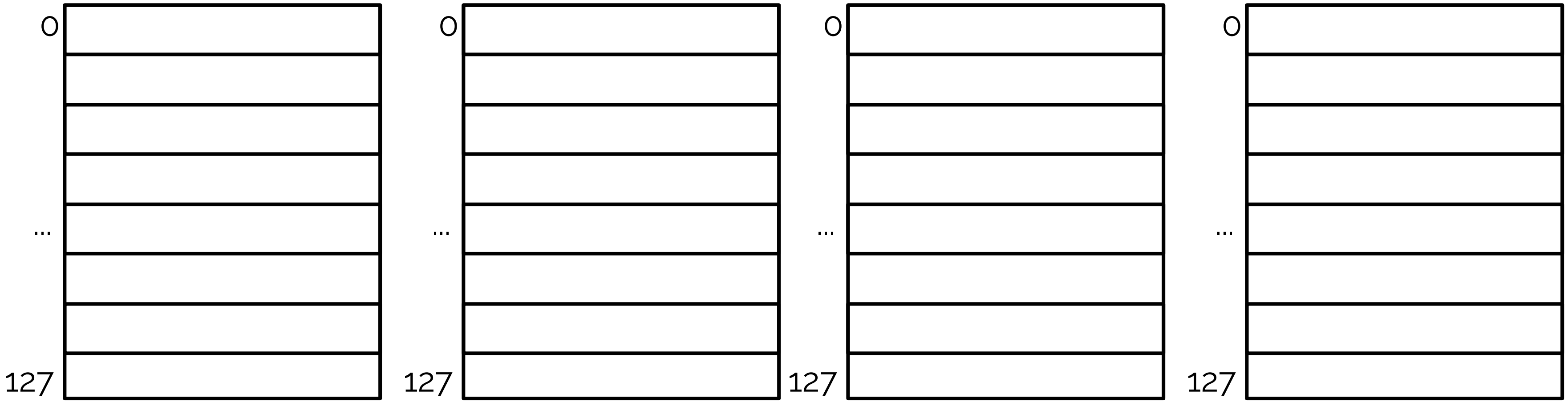
Osvik et al. [63] and Tromer et al. [81] demonstrated with *Evict+Time* a powerful type of attack against AES on OpenSSL implementations that requires neither knowledge of the plaintext nor the ciphertext.

# Prime+Probe

Step 1 Prime: Attacker occupies a set

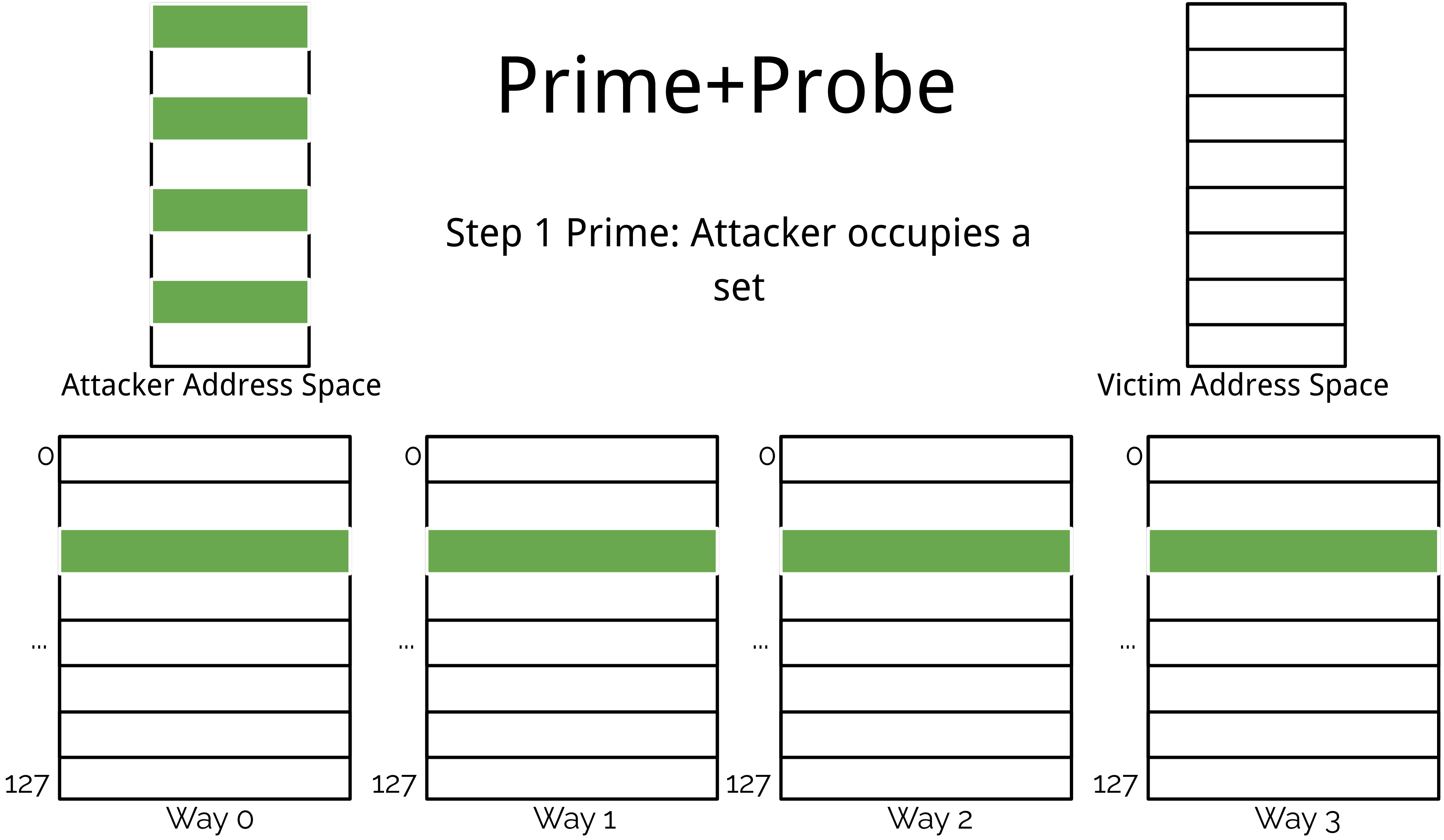


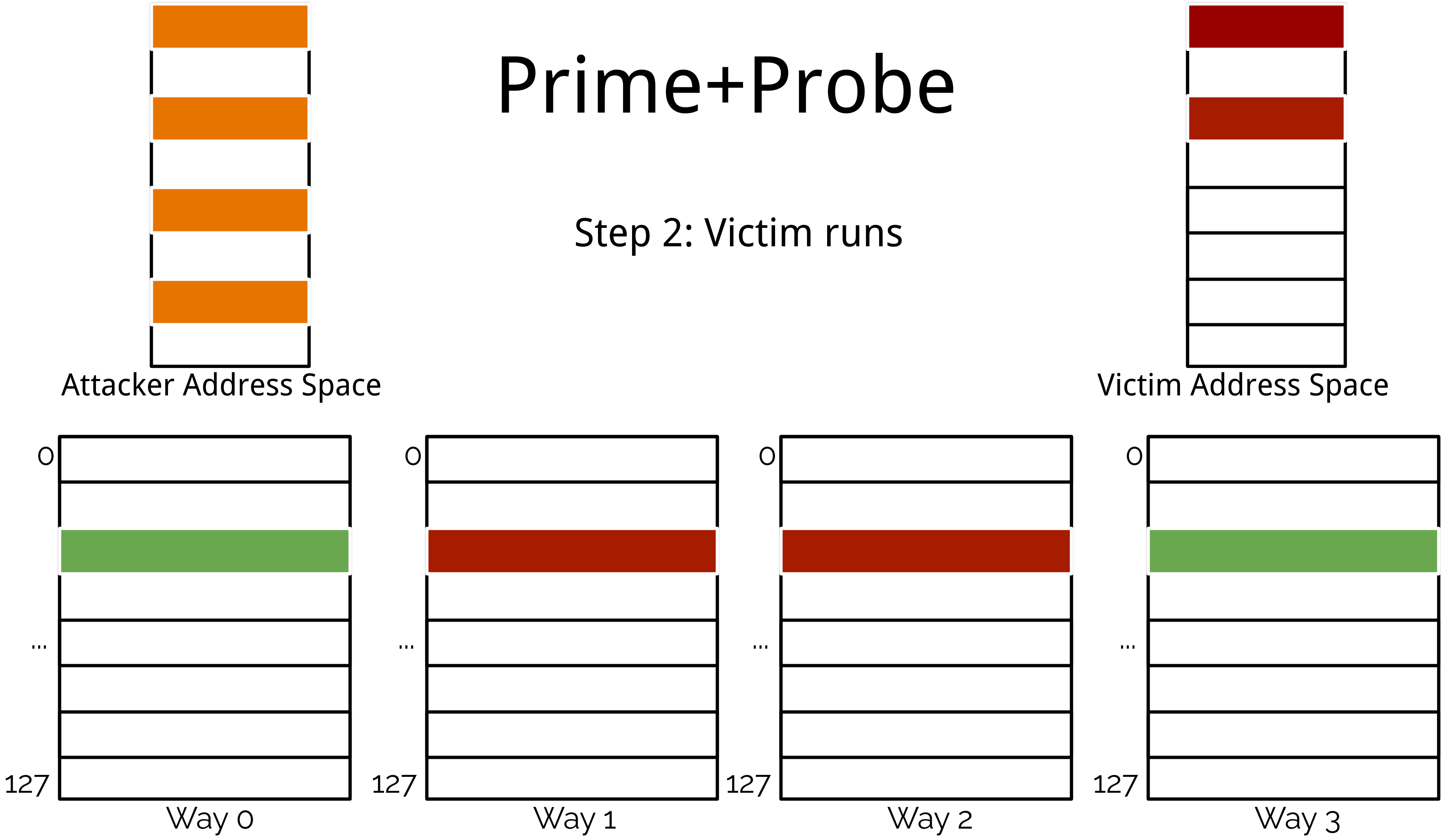
# Prime+Probe

Step 1 Prime: Attacker occupies a set

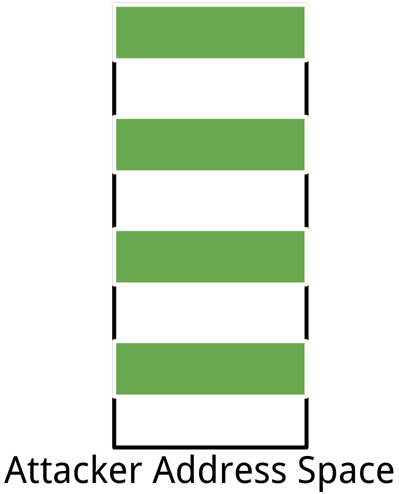


# Prime+Probe

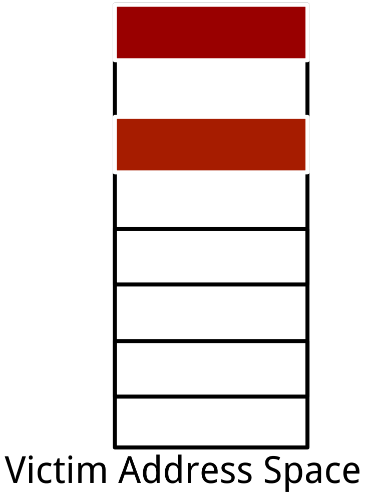
Step 2: Victim runs

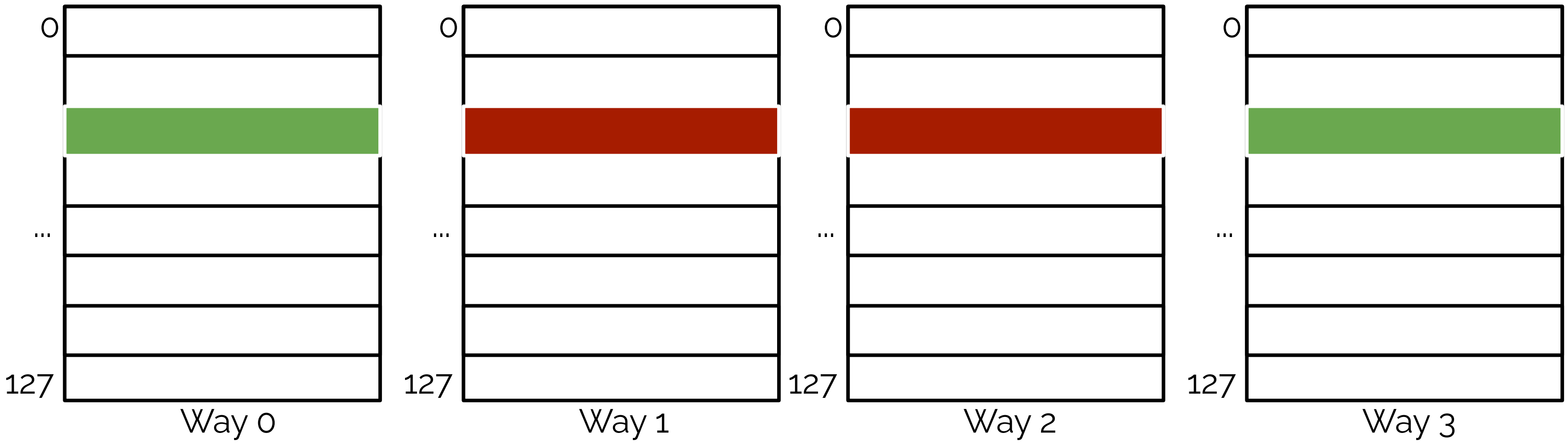


# Prime+Probe



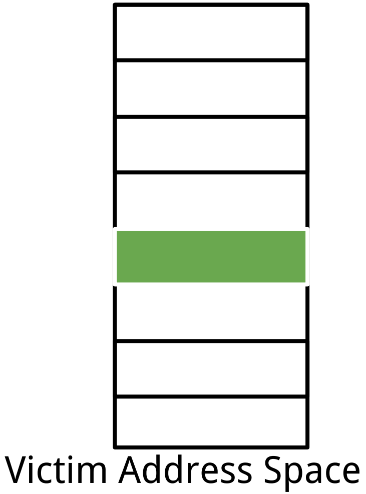

Step 3 Probe: Attacker accesses memory again and measures the time

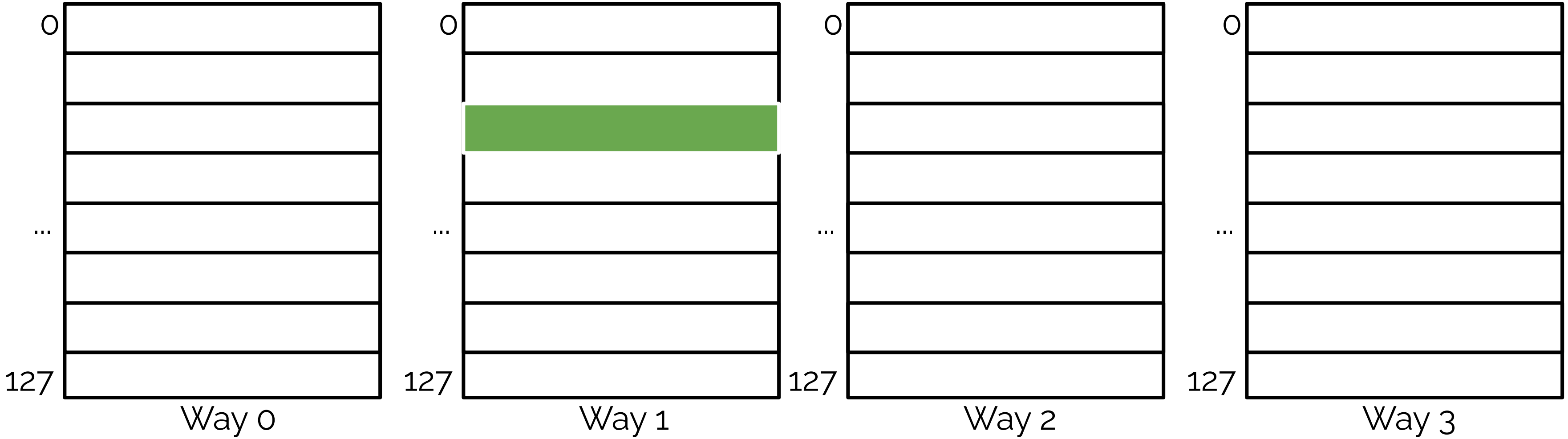


# Flush+Reload

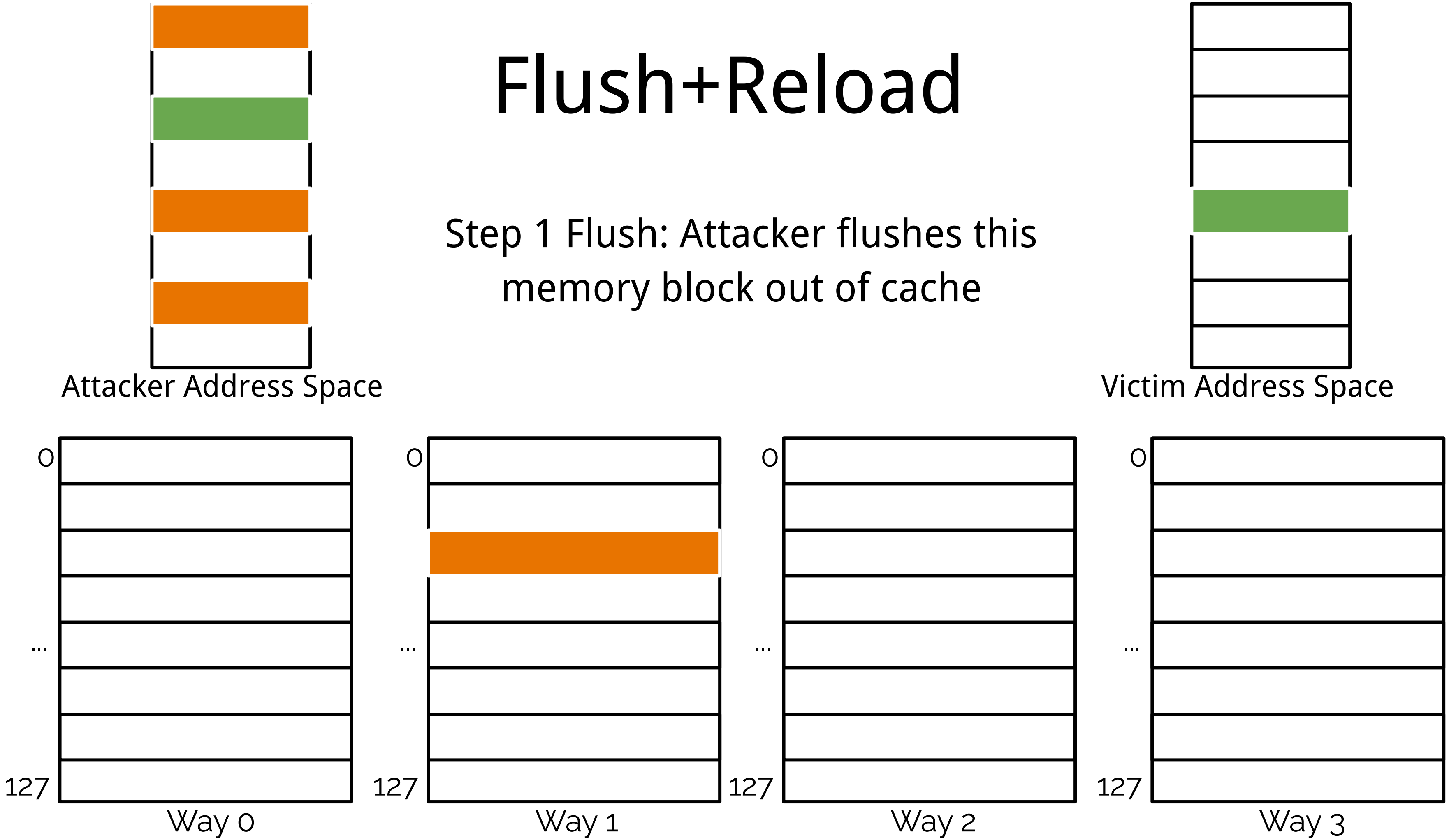
A memory block is cached

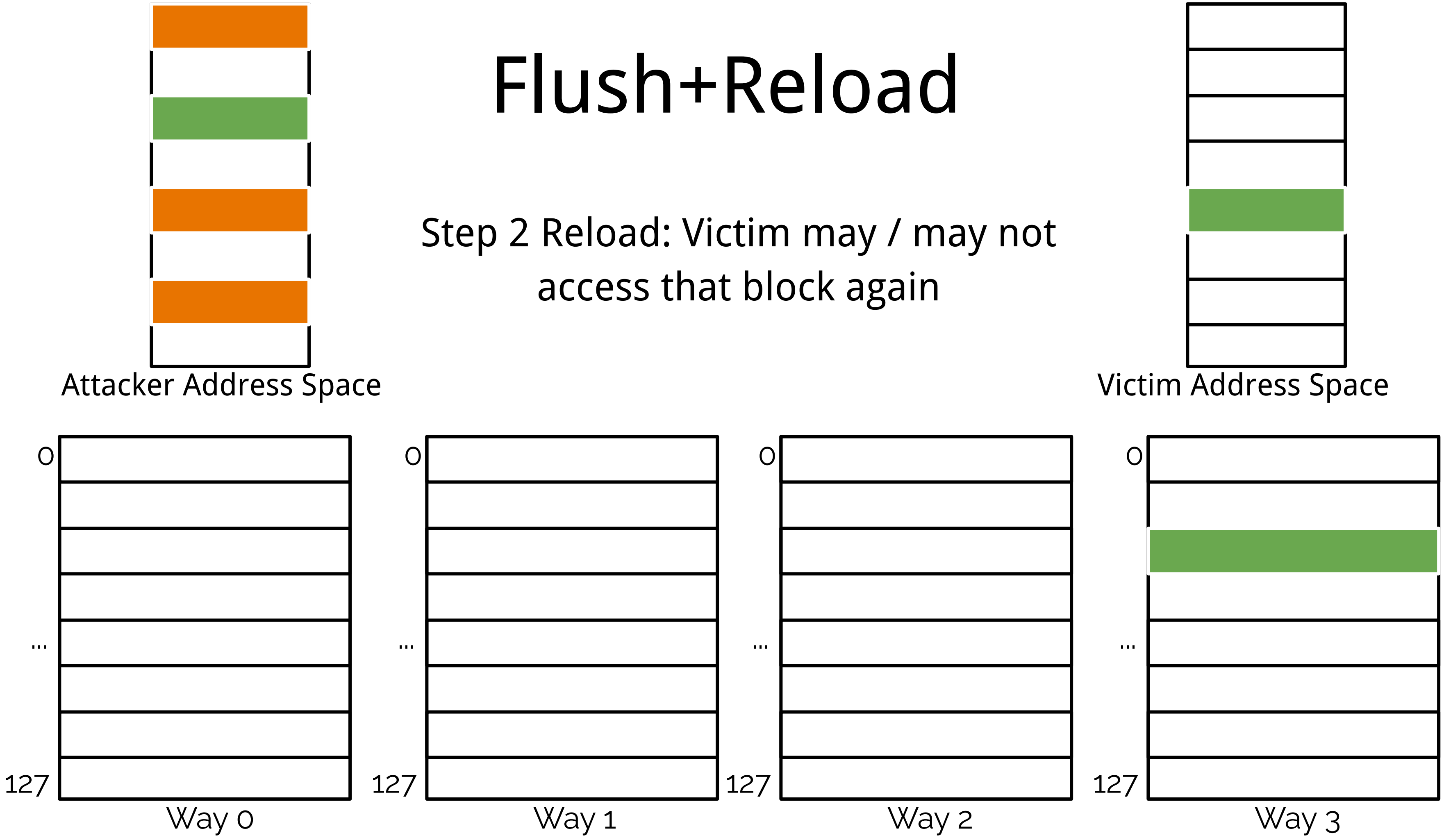
# Flush+Reload

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



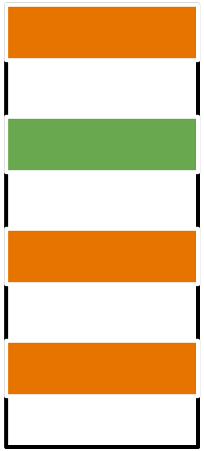
# Flush+Reload

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

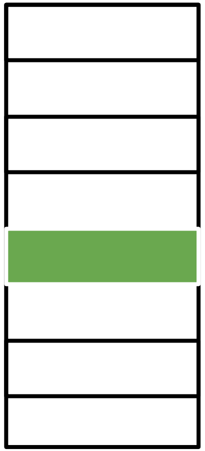


# Flush+Reload

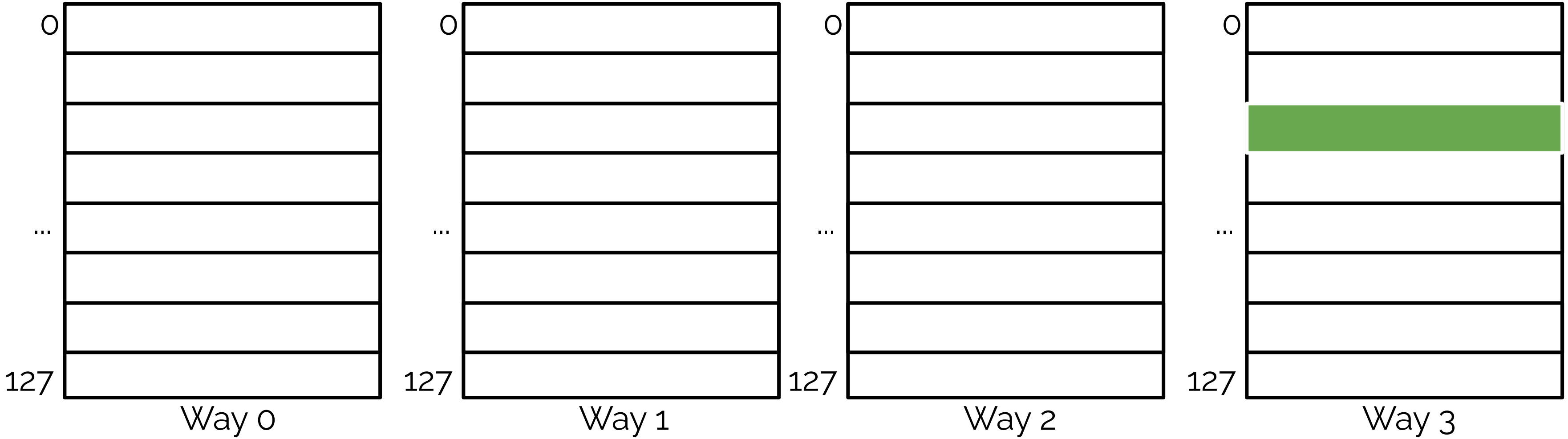
Step 3 Probe: Attacker accesses that block again and measure



Attacker Address Space



Victim Address Space



# 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[2*4096] = 200;
  array[8*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

```
Terminal
[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
```

# Meltdown and Spectre

https://meltdownattack.com/

https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5754

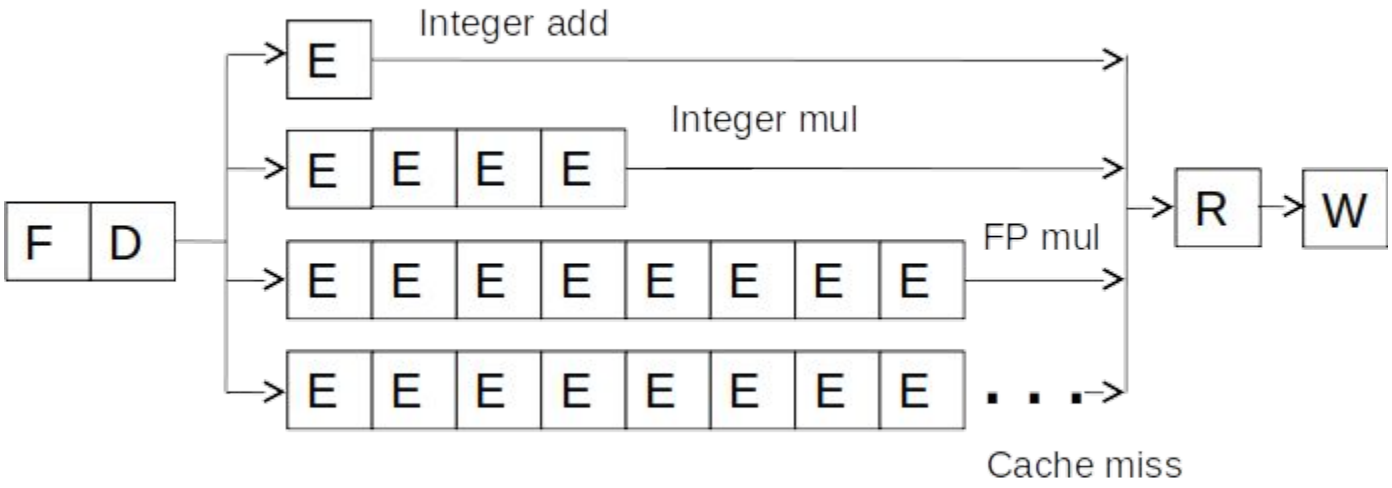
# Meltdown Basics

Meltdown allows attackers to read arbitrary physical memory (including kernel memory) from an unprivileged user process

Meltdown uses ***out of order instruction execution*** to leak data via a processor covert channel (cache lines)

Meltdown was patched (in Linux) with KAISER/KPTI

# An In-order Pipeline




Problem: A true data dependency stalls dispatch of younger instructions into functional (execution) units

Dispatch: Act of sending an instruction to a functional unit

# Can We Do Better?

What do the following two pieces of code have in common (with respect to execution in the previous design)?

```
IMUL  R3 ← R1, R2
ADD   R3 ← R3, R1
ADD   R1 ← R6, R7
IMUL  R5 ← R6, R8
ADD   R7 ← R3, R5
```

```
LD    R3 ← R1 (0)
ADD   R3 ← R3, R1
ADD   R1 ← R6, R7
IMUL  R5 ← R6, R8
ADD   R7 ← R3, R5
```

Answer: First ADD stalls the whole pipeline!
ADD cannot dispatch because its source registers unavailable
Later independent instructions cannot get executed

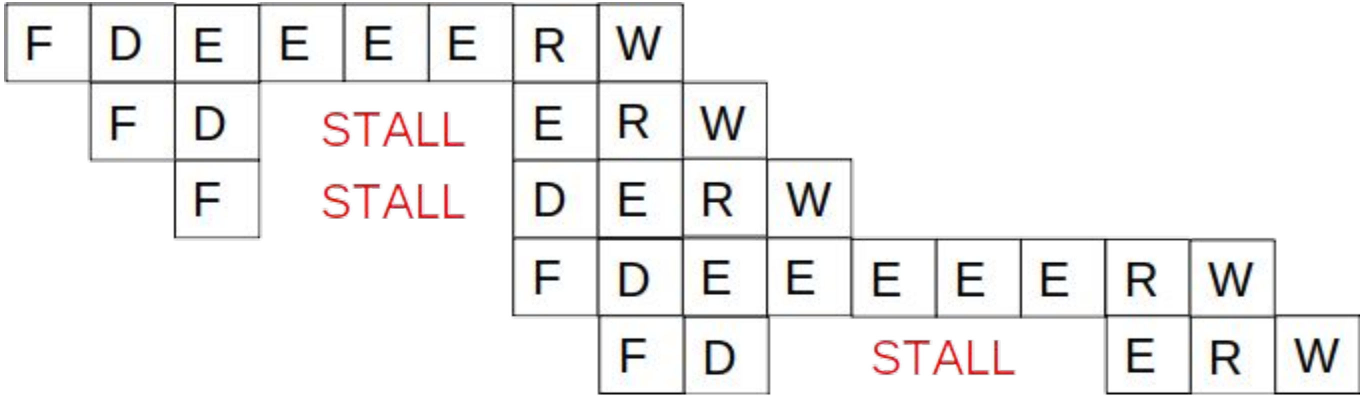
# Out-of-Order Execution (Dynamic Instruction Scheduling)

Idea: Move the dependent instructions out of the way of independent ones; Rest areas for dependent instructions: Reservation stations

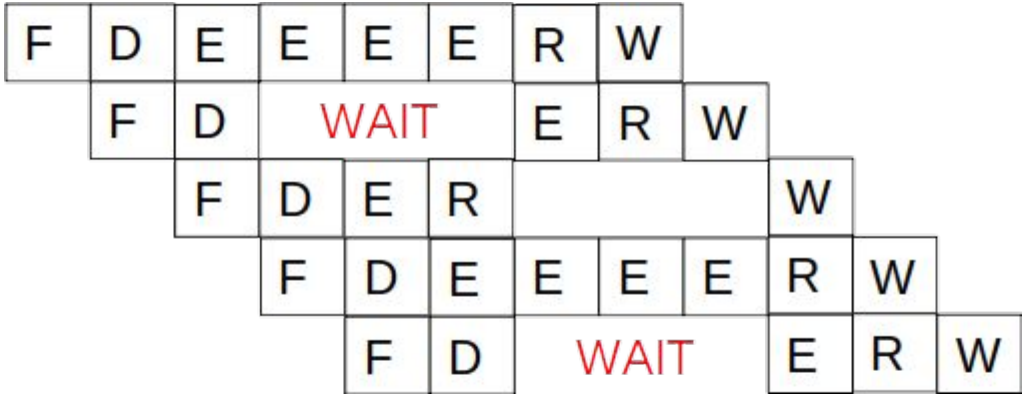
Monitor the source “values” of each instruction in the resting area. When all source “values” of an instruction are available, “fire” (i.e. dispatch) the instruction. Instructions dispatched in dataflow (not control-flow) order

Benefit: Latency tolerance: Allows independent instructions to execute and complete in the presence of a long latency operation

# In-order vs. Out-of-order Dispatch




IMUL R3 ← R1, R2
ADD R3 ← R3, R1
ADD R1 ← R6, R7
IMUL R5 ← R6, R8
ADD R7 ← R3, R5



```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/vmalloc.h>
#include <linux/version.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <linux/uaccess.h>

static char secret[8] = {'S','E','E','D','L','a','b','s'};
static struct proc_dir_entry *secret_entry;
static char* secret_buffer;

static int test_proc_open(struct inode *inode, struct file *file)
{
#if LINUX_VERSION_CODE <= KERNEL_VERSION(4,0,0)
   return single_open(file, NULL, PDE(inode)->data);
#else
   return single_open(file, NULL, PDE_DATA(inode));
#endif
}

static ssize_t read_proc(struct file *filp, char *buffer,
                         size_t length, loff_t *offset)
{
   memcpy(secret_buffer, &secret, 8);
   return 8;
}

static const struct file_operations test_proc_fops =
{
   .owner = THIS_MODULE,
   .open = test_proc_open,
   .read = read_proc,
   .llseek = seq_lseek,
   .release = single_release,
};

static __init int test_proc_init(void)
{
   // write message in kernel message buffer
   printk("secret data address:%p\n", &secret);

   secret_buffer = (char*)vmalloc(8);

   // create data entry in /proc
   secret_entry = proc_create_data("secret_data",
                  0444, NULL, &test_proc_fops, NULL);
   if (secret_entry) return 0;

   return -ENOMEM;
}

static __exit void test_proc_cleanup(void)
{
   remove_proc_entry("secret_data", NULL);
}

module_init(test_proc_init);
module_exit(test_proc_cleanup);[12/02/20]seed@VM:~/Meltdown_Attack$
```

# Speculative Execution

The processor can preserve its current register state, make a prediction as to the path that the program will follow, and speculatively execute instructions along the path.

If the prediction turns out to be correct, the results of the speculative execution are committed (i.e., saved), yielding a performance advantage over idling during the wait.

Otherwise, when the processor determines that it followed the wrong path, it abandons the work it performed speculatively by reverting its register state and resuming along the correct path.

# Speculative Execution

Speculative execution on modern CPUs can run several hundred instructions ahead.

Speculative execution is an optimization technique where a computer system performs some task that may not be needed. Work is done before it is known whether it is actually needed, so as to prevent a delay that would have to be incurred by doing the work after it is known that it is needed.

# Branch Prediction

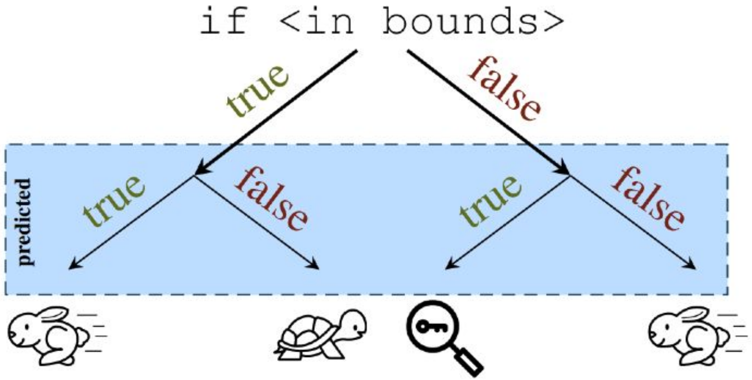
During speculative execution, the processor makes guesses as to the likely outcome of branch instructions.

The branch predictors of modern Intel processors, e.g., Haswell Xeon processors, have multiple prediction mechanisms for direct and indirect branches.

# Spectre V1

Conditional branch misprediction

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



# Spectre V2

Indirect branches can be poisoned by an attacker and the resulting misprediction of indirect branches can be exploited to read arbitrary memory from another context.

# Spectre vs. Meltdown

Meltdown does not use branch prediction. Instead, it relies on the observation that when an instruction causes a trap, following instructions are executed out-of-order before being terminated.

Second, Meltdown exploits a vulnerability specific to many Intel and some ARM processors which allows certain speculatively executed instructions to bypass memory protection.

Meltdown accesses kernel memory from user space. This access causes a trap, but before the trap is issued, the instructions that follow the access leak the contents of the accessed memory through a cache covert channel.