

# **CSE 410/518 Special Topics: Software Security**

Instructor: Dr. Ziming Zhao

# **Heap-based Buffer Overflow**

# Heap Overflow

- Buffer overflows are basically the same on the heap as they are on the stack
- Heap cookies/canaries aren't a thing
  - No 'return' addresses to protect
- In the real world, lots of cool and complex things like objects/structs end up on the heap
  - Anything that handles the data you just corrupted is now viable attack surface in the application
- It's common to put function pointers in structs which generally are malloc'd on the heap

# code/heapoverflow1

```
void fly()
{
    printf("Flying ...\n");
}

typedef struct airplane
{
    void (*pfun());
    char name[20];
} airplane;
```

```
int main()
{
    printf("fly() at %p; print_flag() at %p\n", fly,
    print_flag);

    struct airplane *p1 = malloc(sizeof(airplane));
    printf("Airplane 1 is at %p\n", p1);

    struct airplane *p2 = malloc(sizeof(airplane));
    printf("Airplane 2 is at %p\n", p2);

    p1->pfun = fly;
    p2->pfun = fly;

    fgets(p2->name, 10, stdin);
    fgets(p1->name, 50, stdin);

    p1->pfun();
    p2->pfun();

    free(p1);
    free(p2);
    return 0;
}
```

# code/heapoverflow1

```
void fly()
{
    printf("Flying ...\n");
}

typedef struct airplane
{
    void (*pfun)();
    char name[20];
} airplane;
```

```
int main()
{
    printf("fly() at %p; print_flag() at %p\n", fly,
    print_flag);

    struct airplane *p1 = malloc(sizeof(airplane));
    printf("Airplane 1 is at %p\n", p1);

    struct airplane *p2 = malloc(sizeof(airplane));
    printf("Airplane 2 is at %p\n", p2);

    p1->pfun = fly;
    p2->pfun = fly;

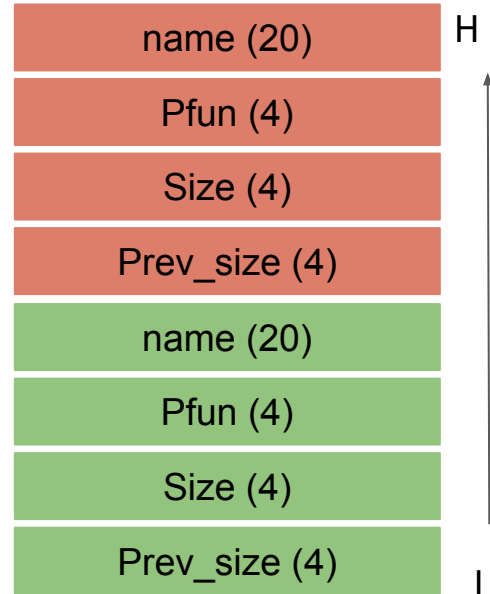
    fgets(p2->name, 10, stdin);
    fgets(p1->name, 50, stdin);

    p1->pfun();
    p2->pfun();

    free(p1);
    free(p2);
    return 0;
}
```

Airplane 2

Airplane 1



# code/heapoverflow1

```
void secret()
{
    printf("The secret is bla bla...\n");
}

void fly()
{
    printf("Flying ...\n");
}

typedef struct airplane
{
    void (*pfun)();
    char name[20];
} airplane;
```

```
int main()
{
    printf("fly() at %p; secret() at %p\n", fly, secret);

    struct airplane *p1 = malloc(sizeof(airplane));
    printf("Airplane 1 is at %p\n", p1);

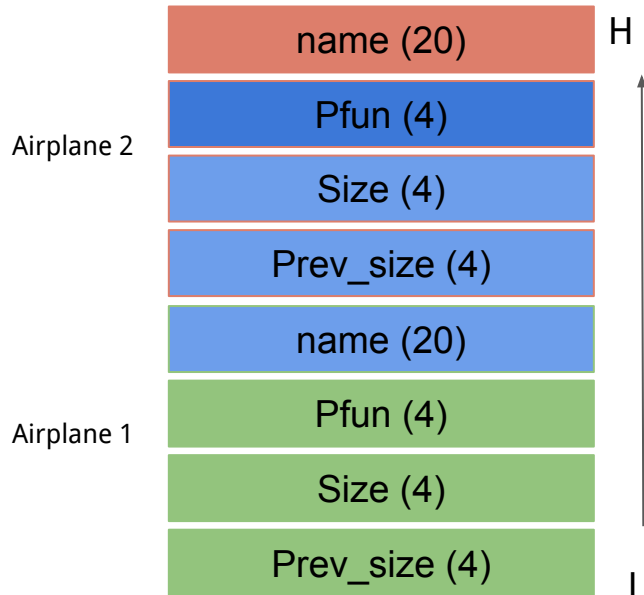
    struct airplane *p2 = malloc(sizeof(airplane));
    printf("Airplane 2 is at %p\n", p2);

    p1->pfun = fly;
    p2->pfun = fly;

    fgets(p2->name, 10, stdin);
    fgets(p1->name, 50, stdin);

    p1->pfun();
    p2->pfun();

    free(p1);
    free(p2);
    return 0;
}
```



Exploit looks like

```
python -c "print 'a\n' + 'a'*28 + '\x4d\x62\x55\x56'" | ./heapoverflow32
```

## **Use after free (UAF)**

A class of vulnerability where data on the heap is freed, but a leftover reference or 'dangling pointer' is used by the code as if the data were still valid.

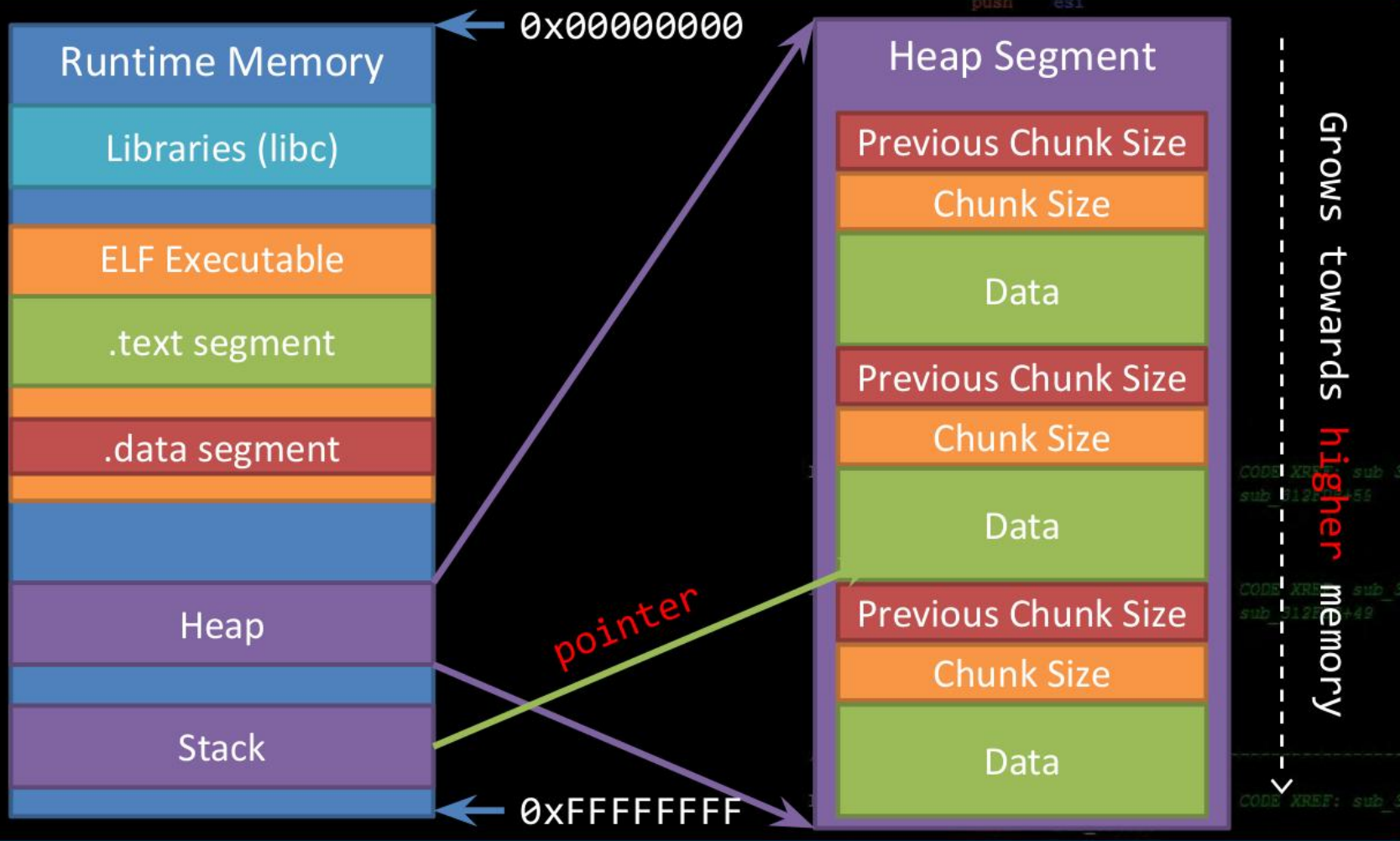
Most popular in Web Browsers, complex programs



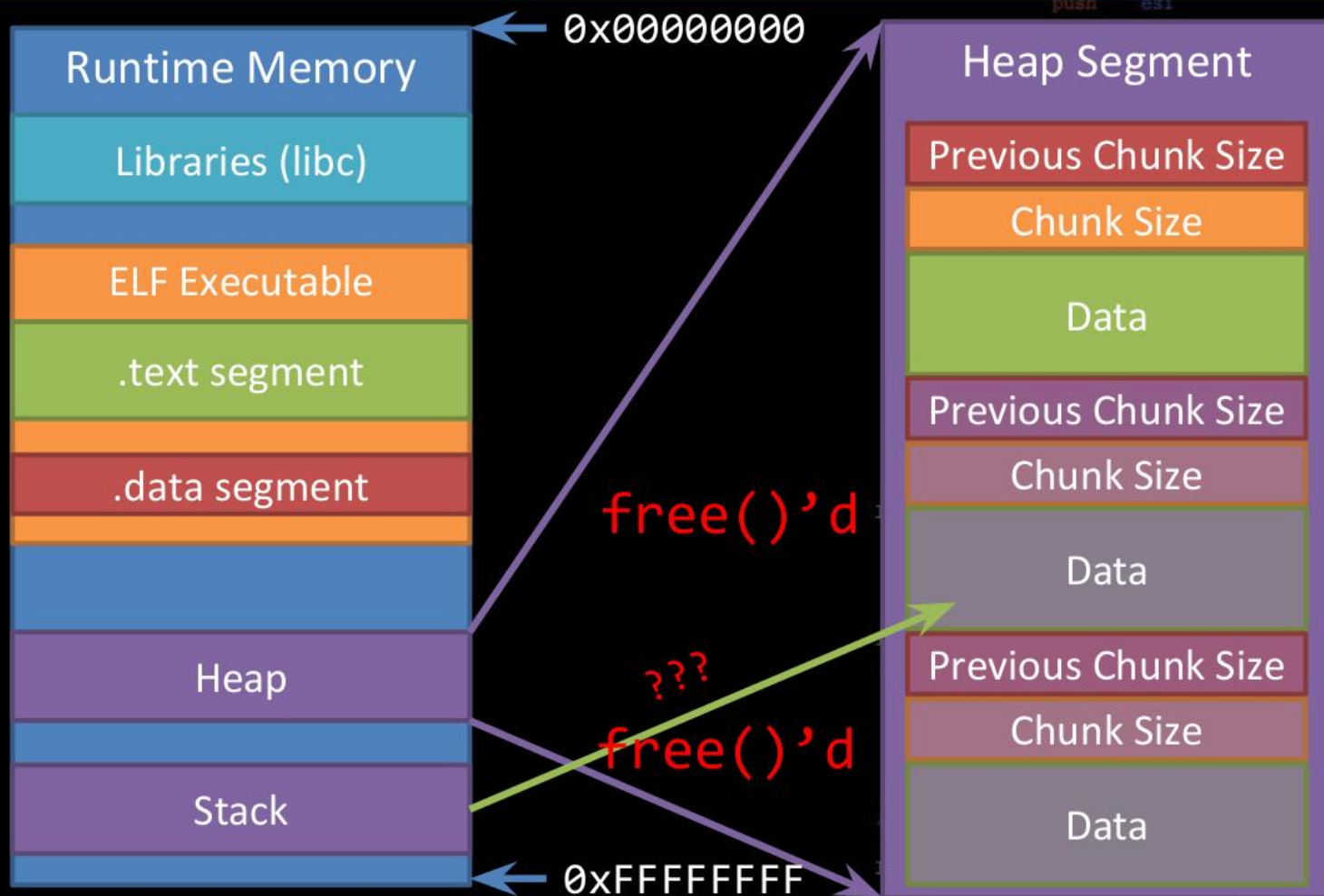
## The CWE Top 25

Below is a list of the weaknesses in the 2022 CWE Top 25, including the overall score of each. The KEV Count (CVEs) shows the number of CVE-2020/CVE-2021 Records from the CISA KEV list that were mapped to the given weakness.

Rank	ID	Name	Score	KEV Count (CVEs)	Rank Change vs. 2021
1	<a href="#">CWE-787</a>	Out-of-bounds Write	64.20	62	0
2	<a href="#">CWE-79</a>	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	45.97	2	0
3	<a href="#">CWE-89</a>	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	22.11	7	+3 ▲
4	<a href="#">CWE-20</a>	Improper Input Validation	20.63	20	0
5	<a href="#">CWE-125</a>	Out-of-bounds Read	17.67	1	-2 ▼
6	<a href="#">CWE-78</a>	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	17.53	32	-1 ▼
7	<a href="#">CWE-416</a>	Use After Free	15.50	28	0
8	<a href="#">CWE-22</a>	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	14.08	19	0
9	<a href="#">CWE-352</a>	Cross-Site Request Forgery (CSRF)	11.53	1	0
10	<a href="#">CWE-434</a>	Unrestricted Upload of File with Dangerous Type	9.56	6	0
11	<a href="#">CWE-476</a>	NULL Pointer Dereference	7.15	0	+4 ▲
12	<a href="#">CWE-502</a>	Deserialization of Untrusted Data	6.68	7	+1 ▲
13	<a href="#">CWE-190</a>	Integer Overflow or Wraparound	6.53	2	-1 ▼
14	<a href="#">CWE-287</a>	Improper Authentication	6.35	4	0
15	<a href="#">CWE-798</a>	Use of Hard-coded Credentials	5.66	0	+1 ▲
16	<a href="#">CWE-862</a>	Missing Authorization	5.53	1	+2 ▲
17	<a href="#">CWE-77</a>	Improper Neutralization of Special Elements used in a Command ('Command Injection')	5.42	5	+8 ▲
18	<a href="#">CWE-306</a>	Missing Authentication for Critical Function	5.15	6	-7 ▼
19	<a href="#">CWE-119</a>	Improper Restriction of Operations within the Bounds of a Memory Buffer	4.85	6	-2 ▼
20	<a href="#">CWE-276</a>	Incorrect Default Permissions	4.84	0	-1 ▼
21	<a href="#">CWE-918</a>	Server-Side Request Forgery (SSRF)	4.27	8	+3 ▲
22	<a href="#">CWE-362</a>	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	3.57	6	+11 ▲
23	<a href="#">CWE-400</a>	Uncontrolled Resource Consumption	3.56	2	+4 ▲
24	<a href="#">CWE-611</a>	Improper Restriction of XML External Entity Reference	3.38	0	-1 ▼
25	<a href="#">CWE-94</a>	Improper Control of Generation of Code ('Code Injection')	3.32	4	+3 ▲



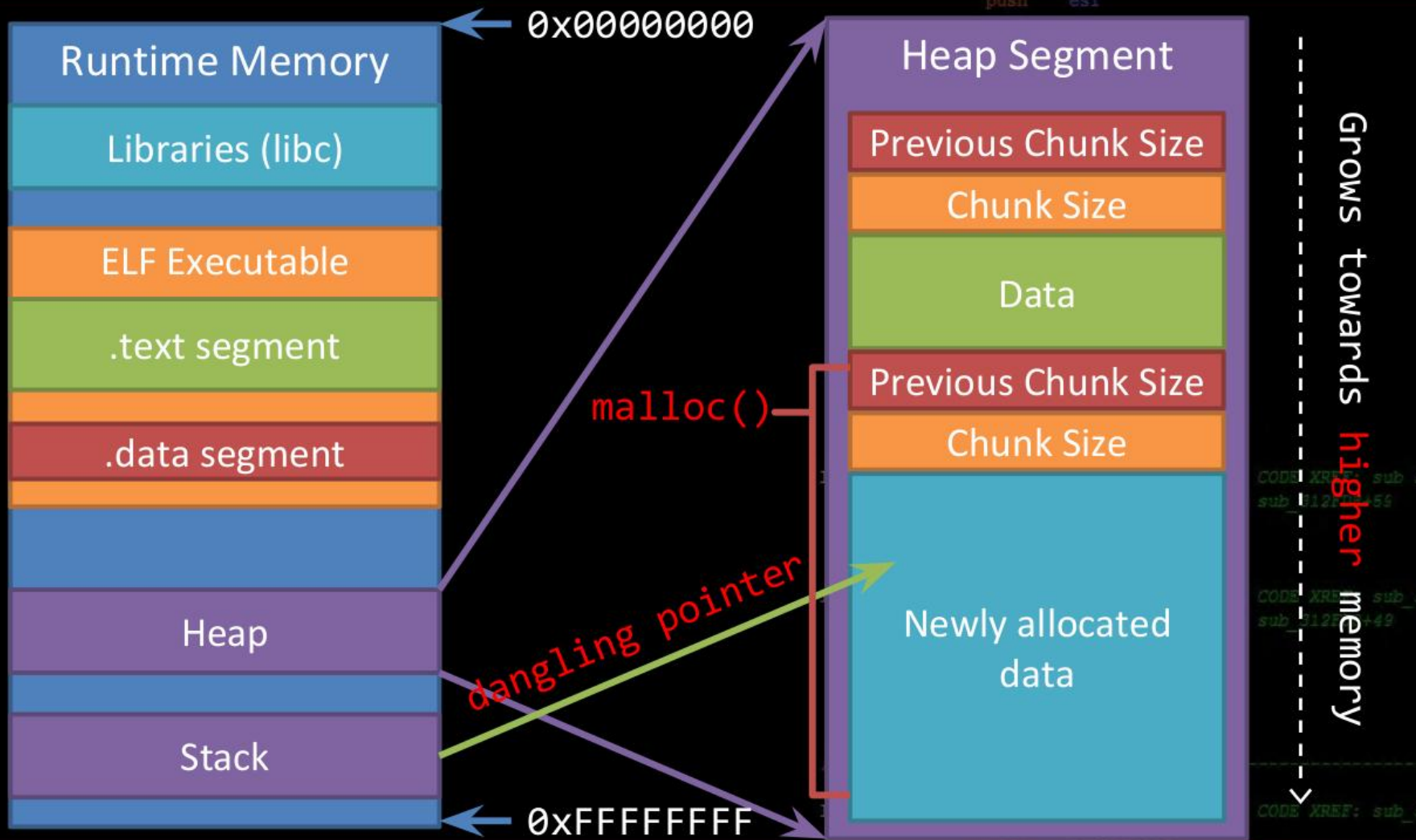
Grows towards **higher** memory



# Dangling Pointer

## Dangling Pointer

- A left over pointer in your code that references free'd data and is prone to be re-used
- As the memory it's pointing at was freed, there's no guarantees on what data is there now
- Also known as stale pointer, wild pointer



# Exploit UAF

To exploit a UAF, you usually have to allocate a different type of object over the one you just freed

# code/heapoverflow2

```
void fly()
{
    printf("Flying ...\n");
}

typedef struct airplane
{
    void (*pfun)();
    char name[20];
} airplane;

typedef struct car
{
    int volume;
    char name[20];
} car;
```

```
int main()
{ printf("fly() at %p; print_flag() at %u\n", fly, (unsigned int)print_flag);

    struct airplane *p = malloc(sizeof(airplane));
    printf("Airplane is at %p\n", p);
    p->pfun = fly;
    p->pfun();
    free(p);

    struct car *p1 = malloc(sizeof(car));
    printf("Car is at %p\n", p1);

    int volume;
    printf("What is the volume of the car?\n");
    scanf("%u", &volume);
    p1->volume = volume;

    p->pfun();
    free(p);
    return 0;
}
```

# Dlmalloc (using glibc 2.3 as an example)

```
struct malloc_chunk
{
    INTERNAL_SIZE_T prev_size; /* Size of previous chunk (if free). */
    INTERNAL_SIZE_T size; /* Size in bytes, including overhead. */

    struct malloc_chunk* fd; /* double links -- used only if free. */
    struct malloc_chunk* bk;

};

typedef struct malloc_chunk* mchunkptr;
```

**Mem** is the pointer returned by malloc() call, while **chunk pointer** is what malloc considers the start of the chunk.

The whole heap is bounded from top by a **wilderness** chunk. In the beginning, this is the only chunk existing and malloc first makes allocated chunks by splitting the wilderness chunk.

glibc 2.3 allows for many heaps arranged into several **arenas**—one arena for each thread

– From the book “Buffer Overflow Attacks: Detect, Exploit, Prevent” Syngree



# Consolidating chunks when free()-d

When a previously allocated chunk is free()-d, it can be either consolidated with previous (backward consolidation) and/or follow (forward consolidation) chunks, if they are free.

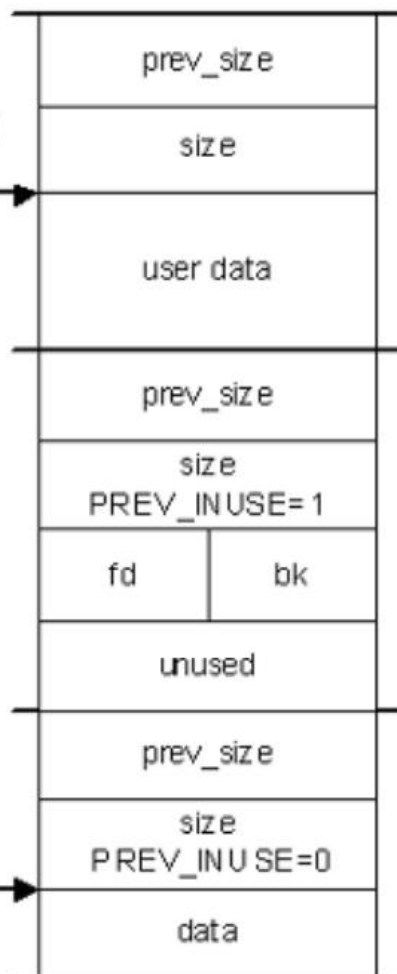
This ensures that there are no two adjacent free chunks in memory. The resulting chunk is then placed in a ***bin***, which is a ***doubly linked list of free chunks of a certain size***.

There is a set of bins for chunks of different sizes:

- 64 bins of size 8 ■ 32 bins of size 64 ■ 16 bins of size 512
- 8 bins of size 4096 ■ 4 bins of size 32768 ■ 2 bins of size 262144
- 1 bin of size what's left

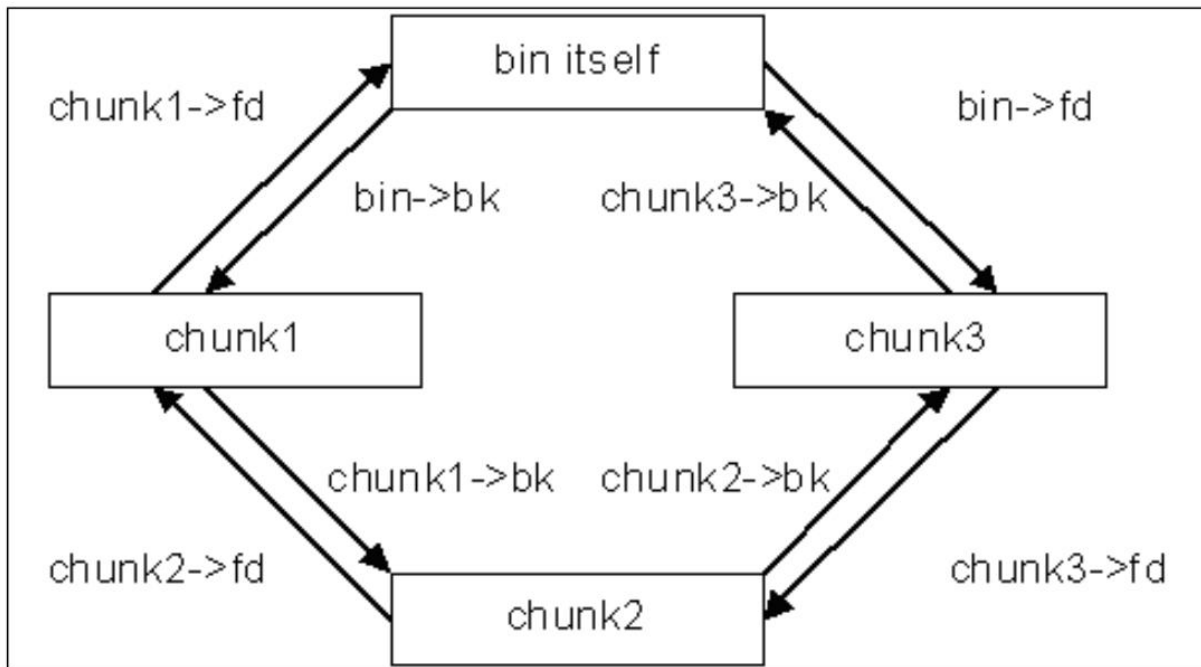
chunk A,  
being freed

A →



chunk A will be  
forward consolidated  
with B

# Example Bin with Three Free Chunks



FD and BK are pointers to “next” and “previous” chunks inside a linked list of a bin, ***not adjacent physical chunks***.

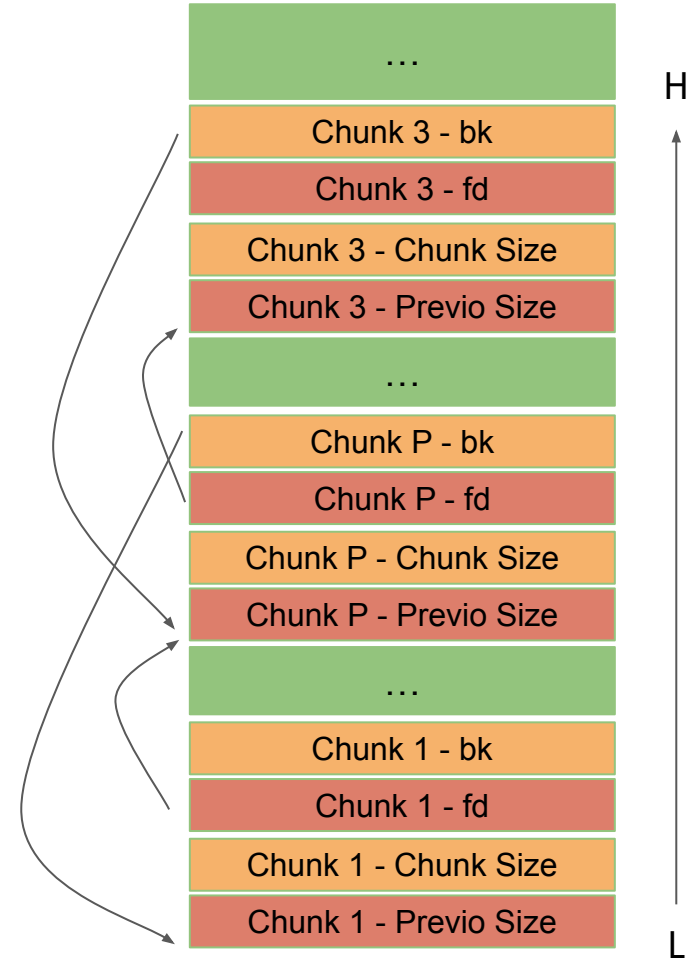
Pointers to chunks, physically next to and previous to this one in memory, can be obtained from current chunks by using **size** and **prev\_size** offsets.

# Pointers to physically next to and previous chunk

```
/* Ptr to next physical malloc_chunk. */  
#define next_chunk(p) ((mchunkptr)((char*)(p) + ((p)->size & ~PREV_INUSE) ))  
  
/* Ptr to previous physical malloc_chunk */  
#define prev_chunk(p) ((mchunkptr)((char*)(p) - ((p)->prev_size) ))
```

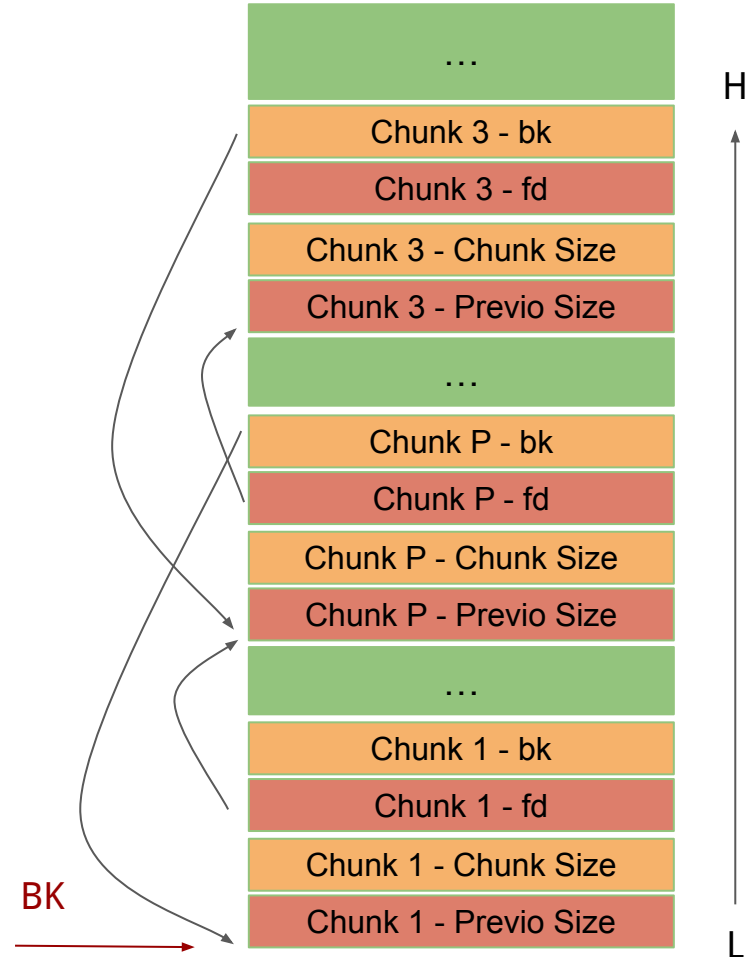
# Unlink() a Free Chunk P from the Bin

```
#define unlink( P, BK, FD ) {  
    BK = P->bk;  
    FD = P->fd;  
    FD->bk = BK;  
    BK->fd = FD;  
}
```



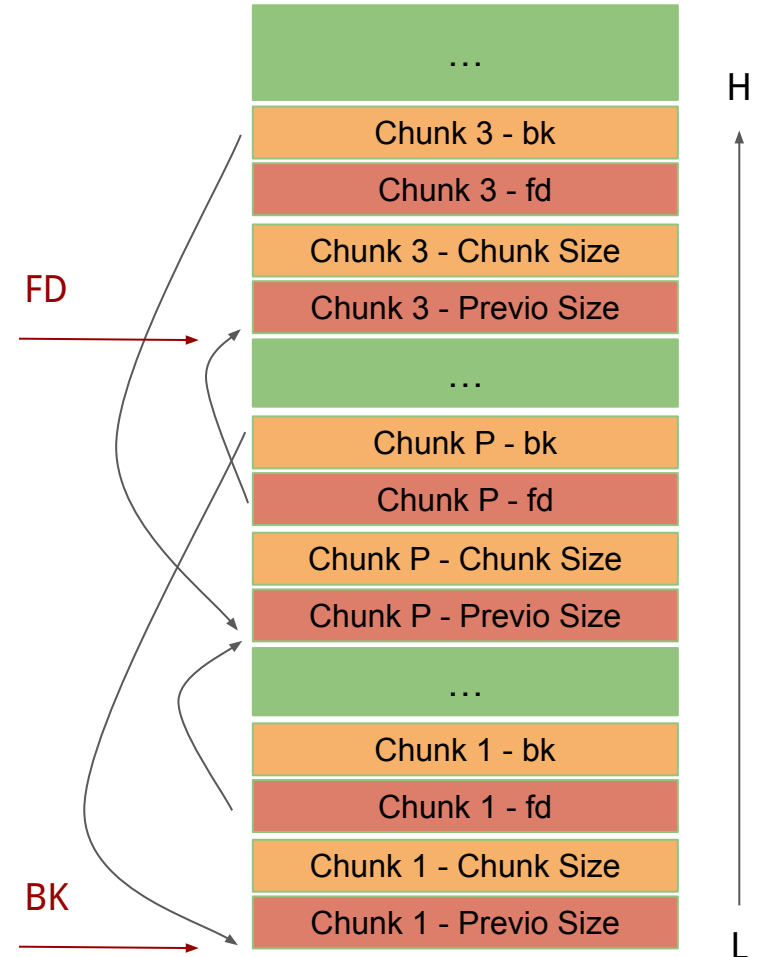
# Unlink() a Free Chunk P from the Bin

```
#define unlink( P, BK, FD ) {  
    BK = P->bk;  
    FD = P->fd;  
    FD->bk = BK;  
    BK->fd = FD;  
}
```



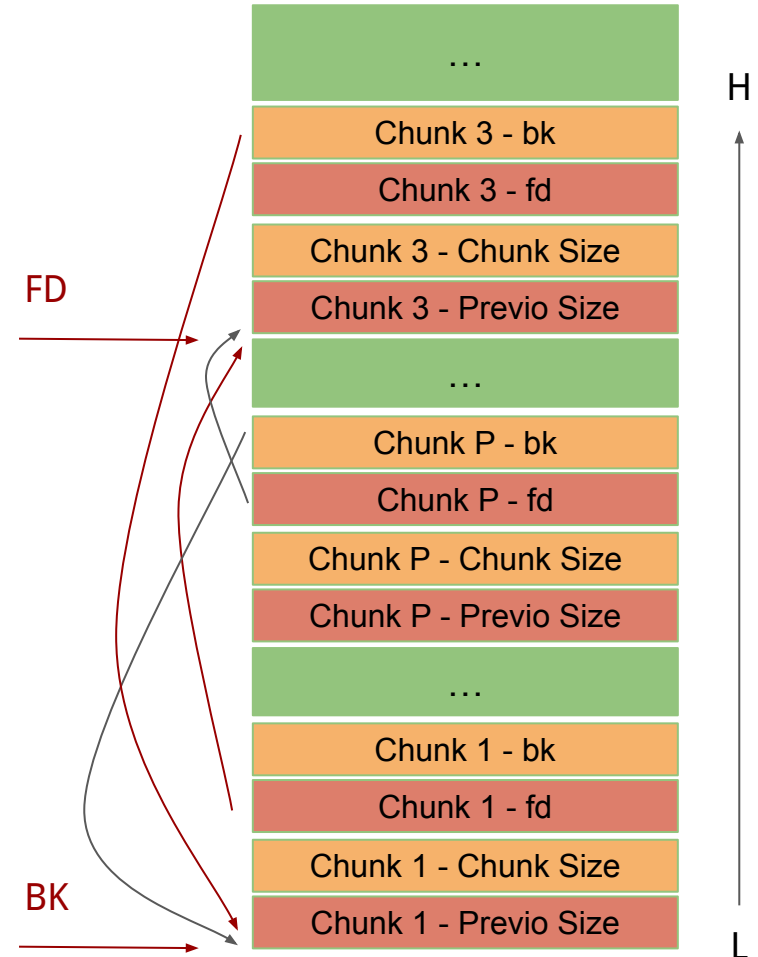
# Unlink() a Free Chunk P from the Bin

```
#define unlink( P, BK, FD ) {  
    BK = P->bk;  
    FD = P->fd;  
    FD->bk = BK;  
    BK->fd = FD;  
}
```



# Unlink() a Free Chunk P from the Bin

```
#define unlink( P, BK, FD ) {  
    BK = P->bk;  
    FD = P->fd;  
    FD->bk = BK;  
    BK->fd = FD;  
}
```



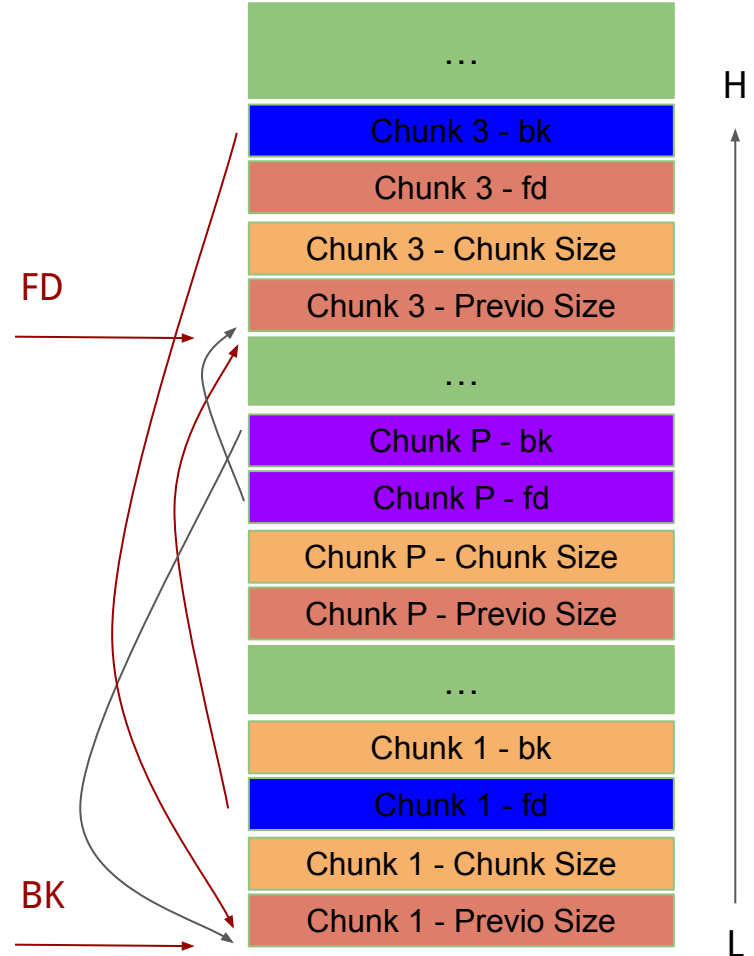


# Unlink() from an Attacker's Point of View

```
*(P->fd+12) = P->bk;  
// 4 bytes for size, 4 bytes for prev_size and 4 bytes for fd  
  
*(P->bk+8) = P->fd;  
// 4 bytes for size, 4 bytes for prev_size
```

Arbitrary write attack?

If an attacker is able to overwrite these two pointers and force the call to unlink(), he can overwrite any memory location.



# The free() Algorithm

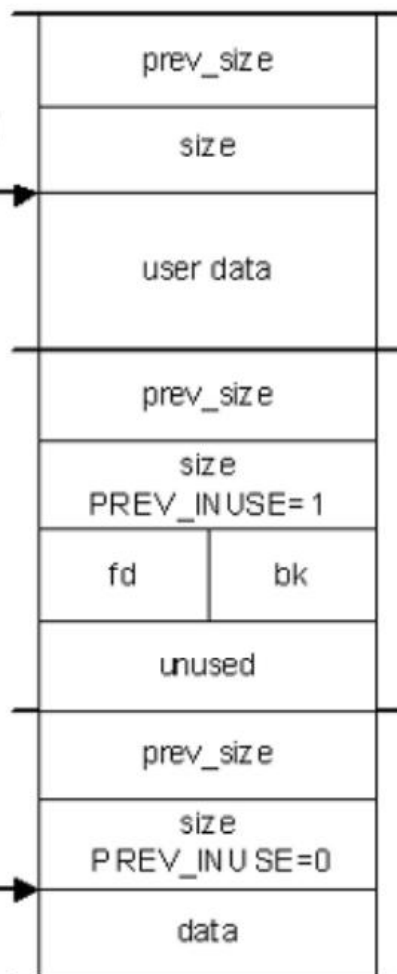
- `free(0)` has no effect.
- If the chunk was allocated via `mmap`, it is released via `munmap()`. Only large chunks are MMAP-ped, and we are not interested in these.
- If a returned chunk borders the current high end of memory (wilderness chunk), it is consolidated into the wilderness chunk, and if the total unused topmost memory exceeds the trim threshold, `malloc_trim()` is called.
- Other chunks are consolidated as they arrive, and placed in corresponding bins.

# The free() Algorithm - last case

- If no adjacent chunks are free, then the freed chunk is simply linked into corresponding with bin via frontlink().
- If the chunk next in memory to the freed one is free and if this next chunk borders on wilderness, then both are consolidated with the wilderness chunk.
- If not, and the previous or next chunk in memory is free and they are not part of a most recently split chunk (this splitting is part of malloc() behavior and is not significant to us here), they are taken off their bins via unlink(). Then they are merged (through forward or backward consolidation) with the chunk being freed and placed into a new bin according to the resulting size using frontlink(). If any of them are part of the most recently split chunk, they are merged with this chunk and kept out of bins. This last bit is used to make certain operations faster.

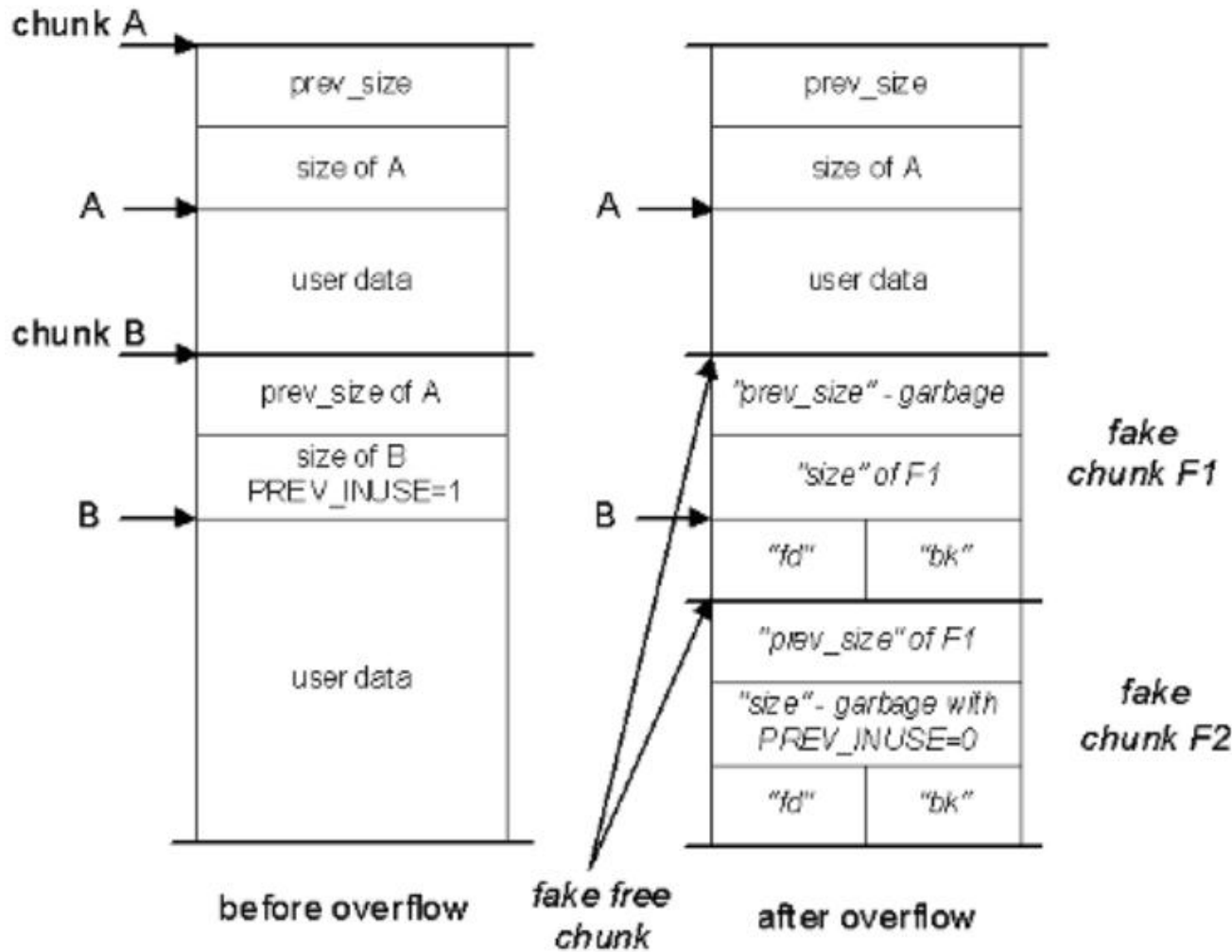
chunk A,  
being freed

A →



chunk A will be  
forward consolidated  
with B

lower addresses



1. Overwrite A and B
2. Create a fake chunk F1 and F2, so that when free(A), unlink(F1) is also called.
3. F1->FD has the address we want to overwrite and F1->BK has the data we want to overwrite

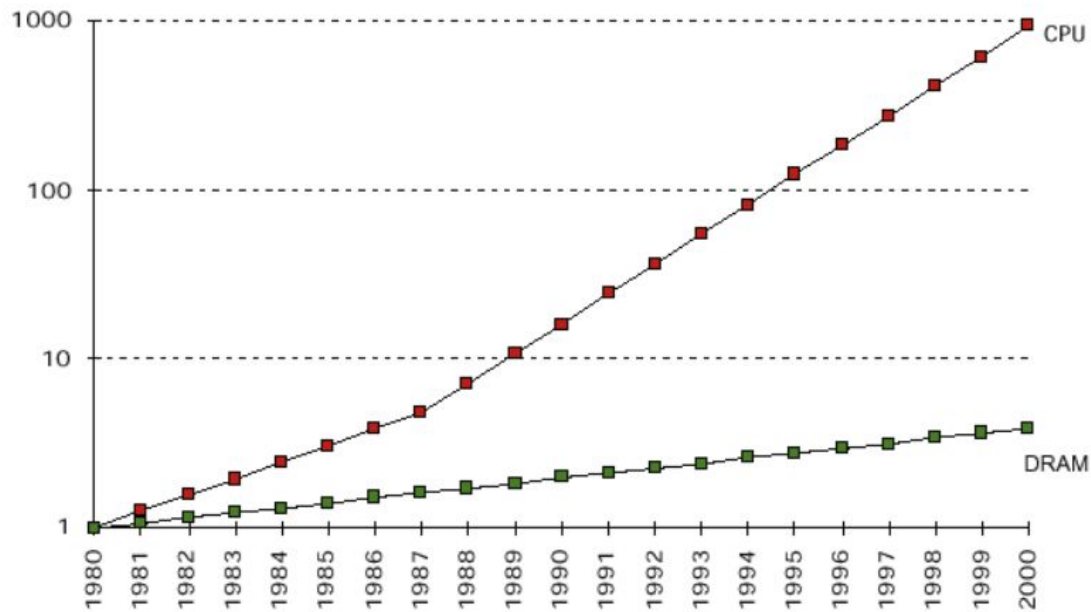
# **CSE 410/518 Special Topics: Software Security**

Instructor: Dr. Ziming Zhao

# 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

<i>Cache</i>	<u>Static RAM</u>	<u>0.5 - 2.5 ns</u>
<i>Memory</i>	<u>Dynamic RAM</u>	<u>50- 70 ns</u>
<i>Secondary</i>	<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.

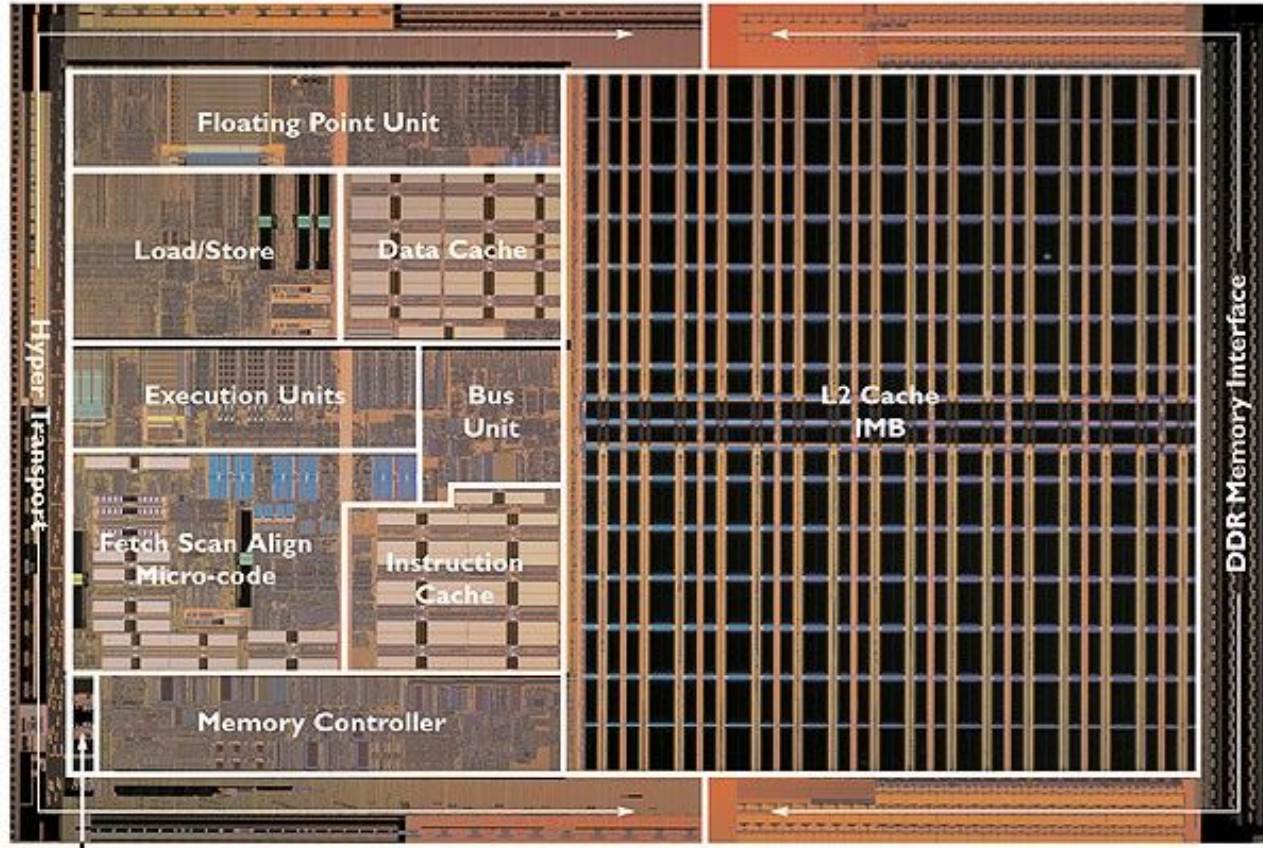
A cache miss occurs if the cache does not contain the requested data.

# Cache Hierarchy

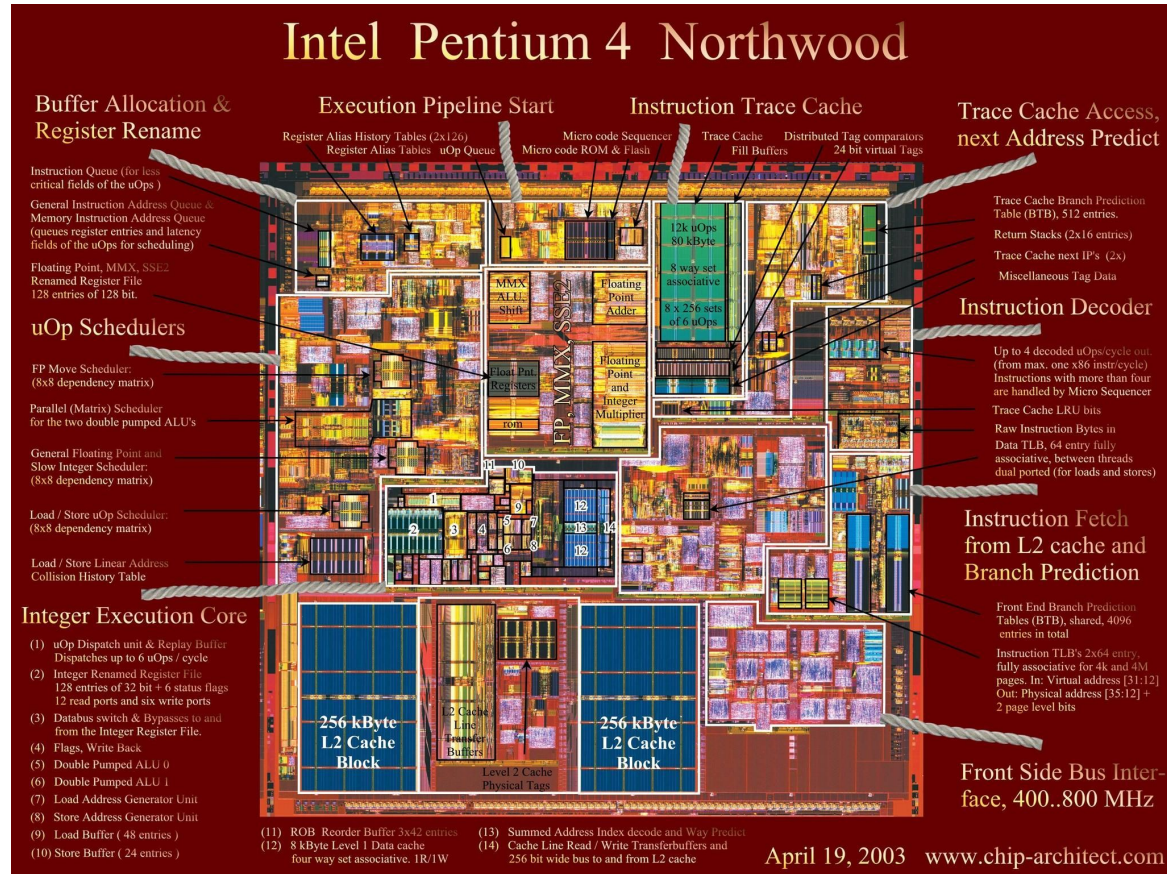
L1 Cache is closest to the CPU. Usually divided in Code and Data cache

L2 and L3 cache are usually unified.

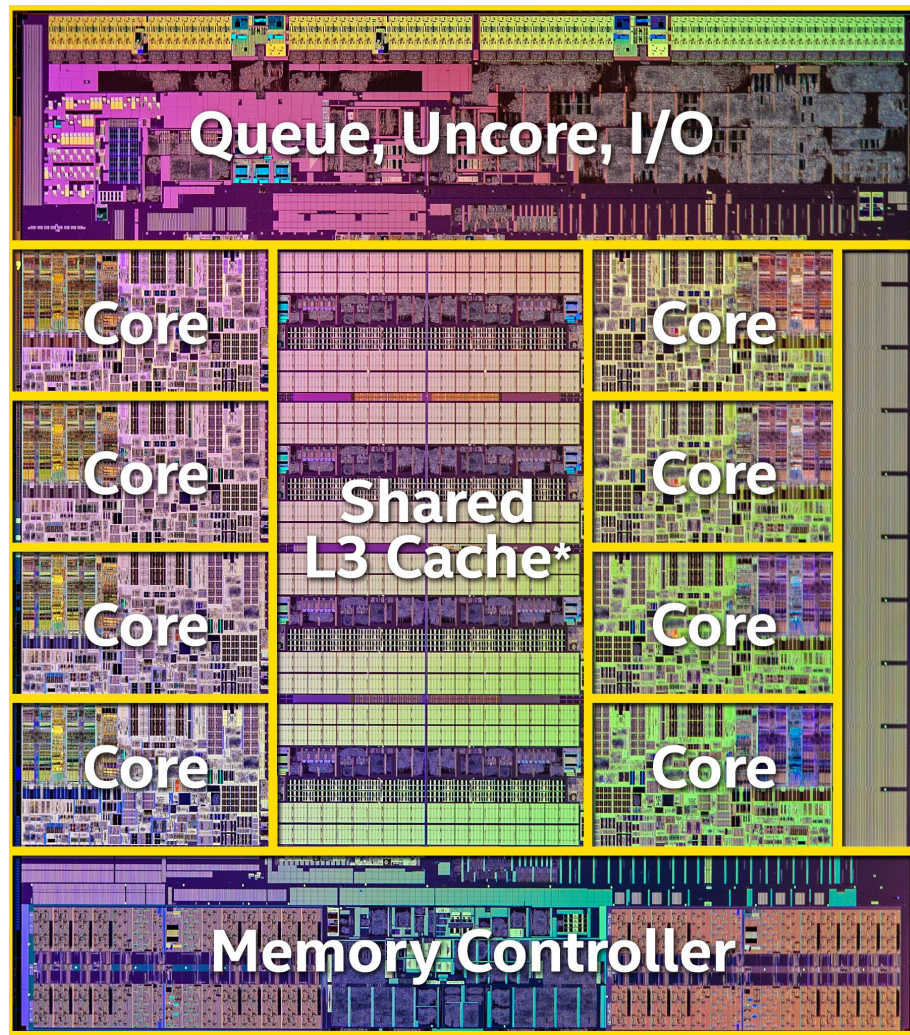
# Cache Hierarchy



# Cache Hierarchy









# Cache Line/Block

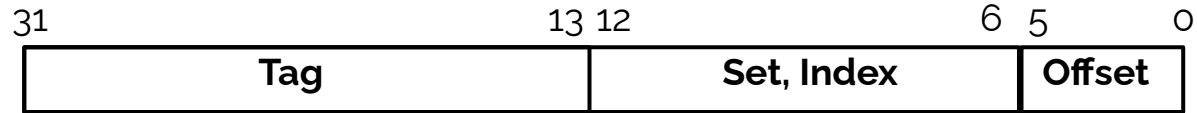
The minimum unit of information that can be either present or not present in a cache.

64 bytes in modern Intel and ARM 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)

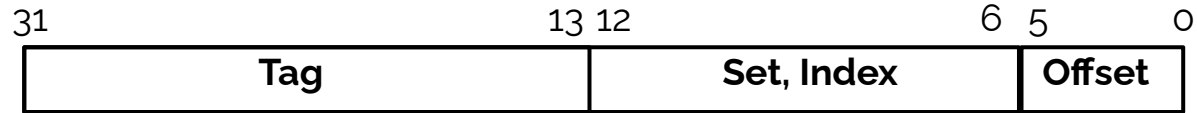
=  $32 * 1024 / (4 * 64)$

= 128

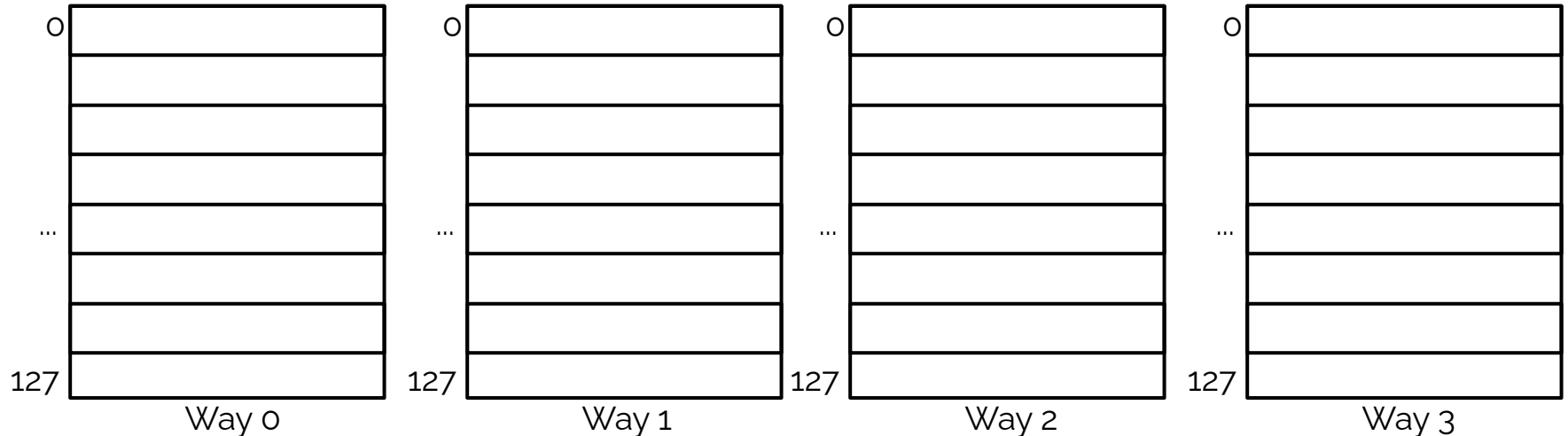
<https://www.geeksforgeeks.org/virtually-indexed-physically-tagged-vipt-cache/>

PIPT VIVT

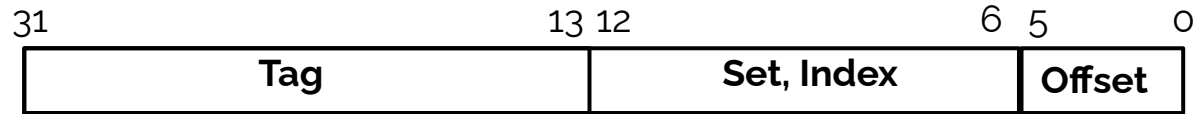
# *n*-Way Set-Associative Cache



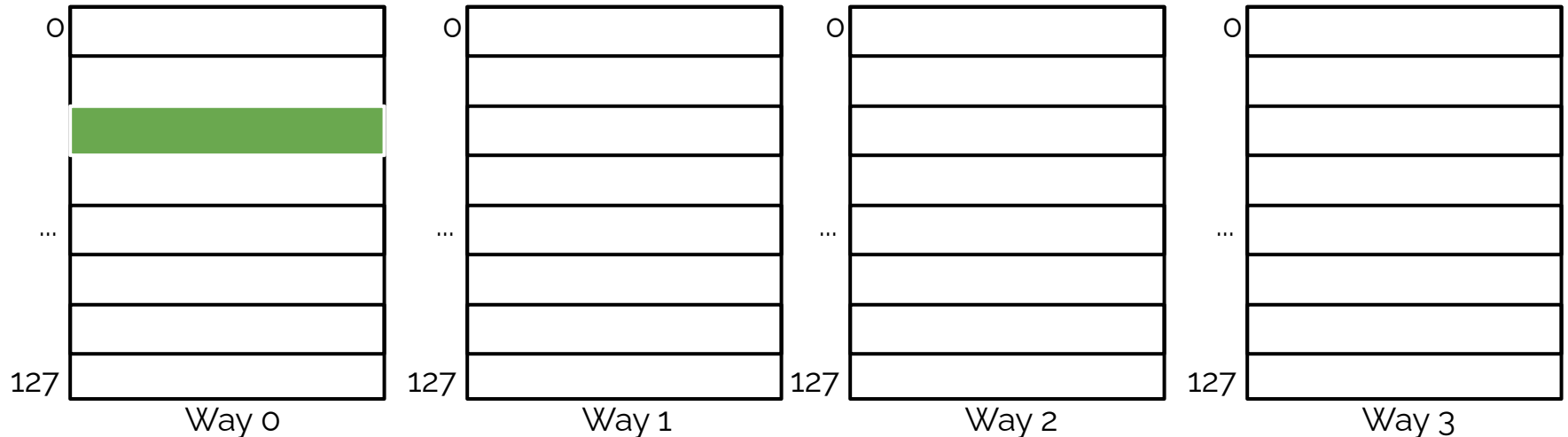
32KB 4-way set-associative data cache, 64 bytes per line



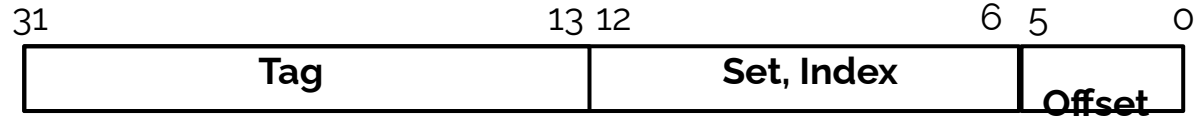
# *n*-Way Set-Associative Cache



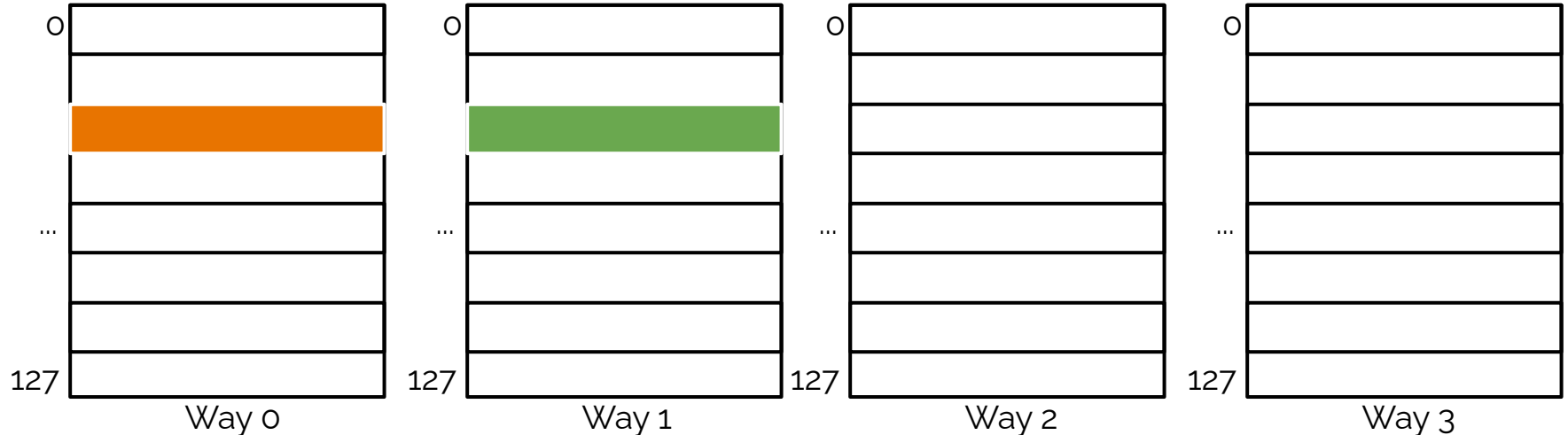
32KB 4-way set-associative data cache, 64 bytes per line



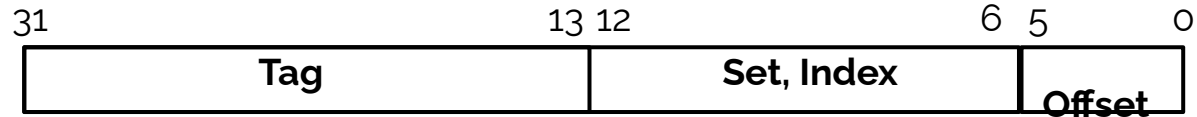
# *n*-Way Set-Associative Cache



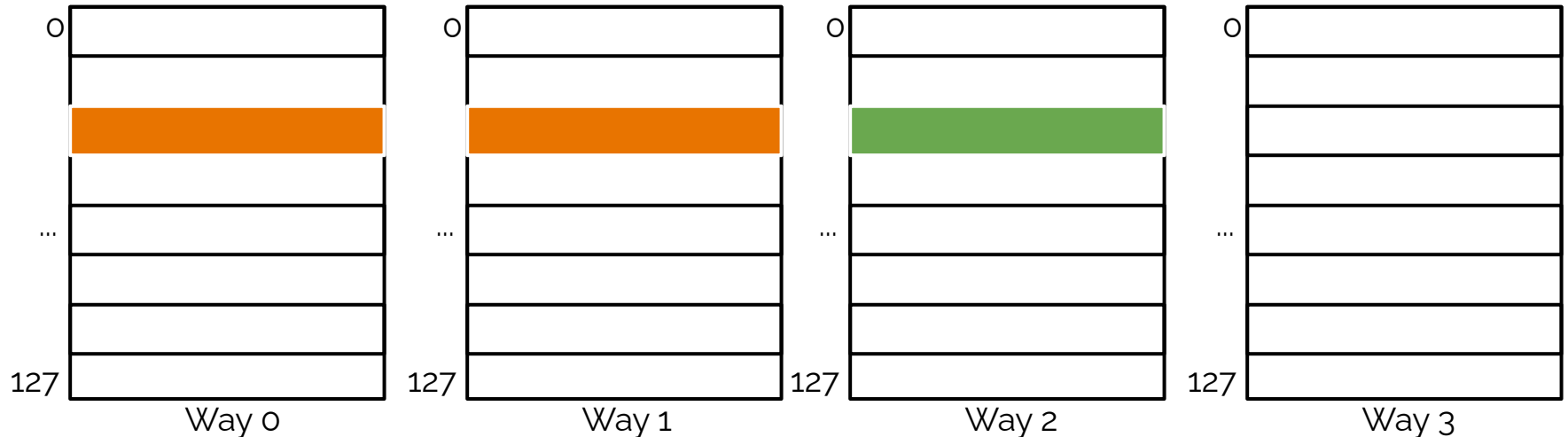
32KB 4-way set-associative data cache, 64 bytes per line



# *n*-Way Set-Associative Cache

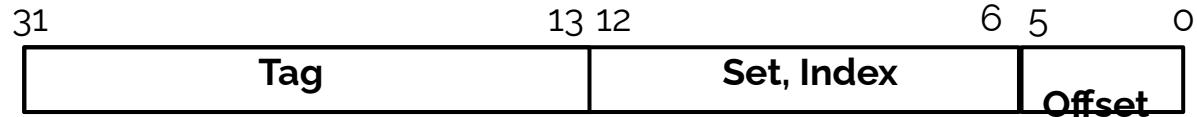


32KB 4-way set-associative data cache, 64 bytes per line

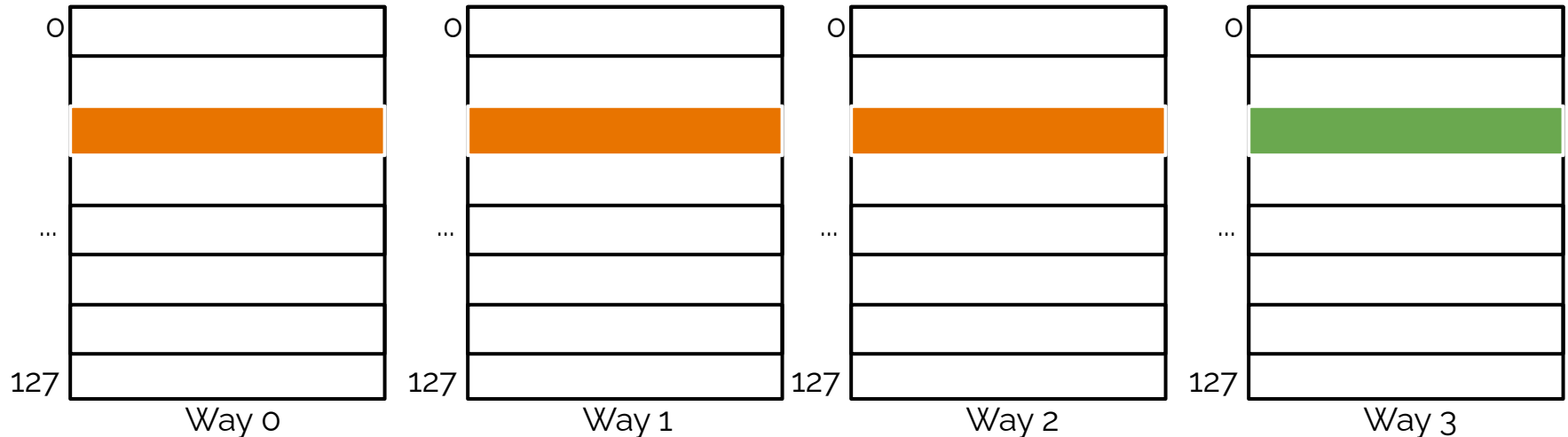




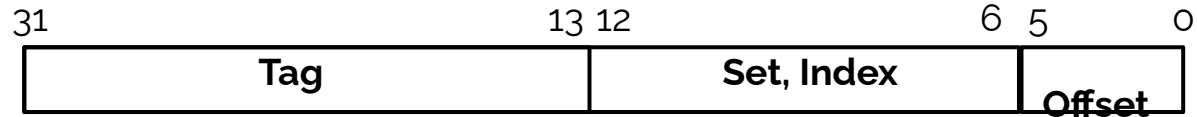
# *n*-Way Set-Associative Cache



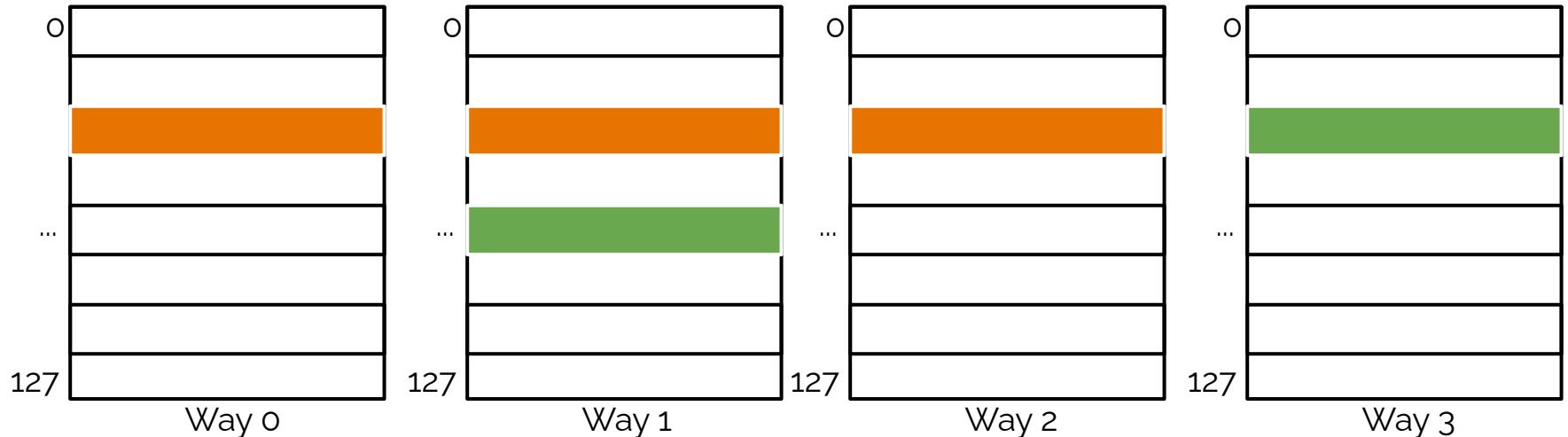
32KB 4-way set-associative data cache, 64 bytes per line



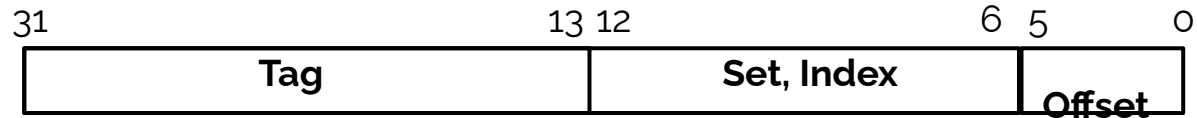
# *n*-Way Set-Associative Cache



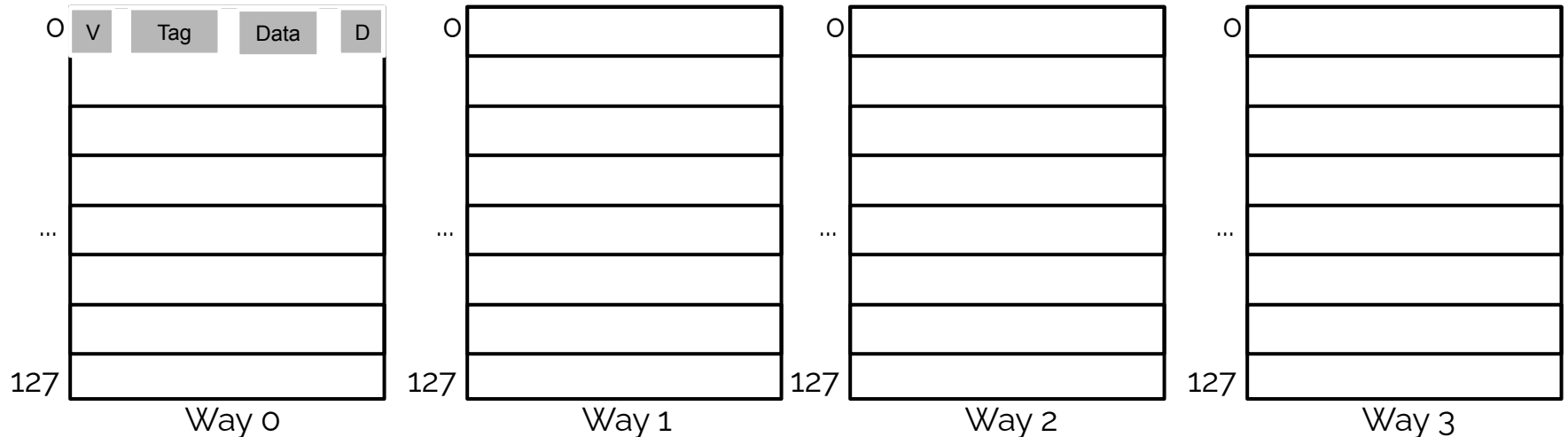
32KB 4-way set-associative data cache, 64 bytes per line



# 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

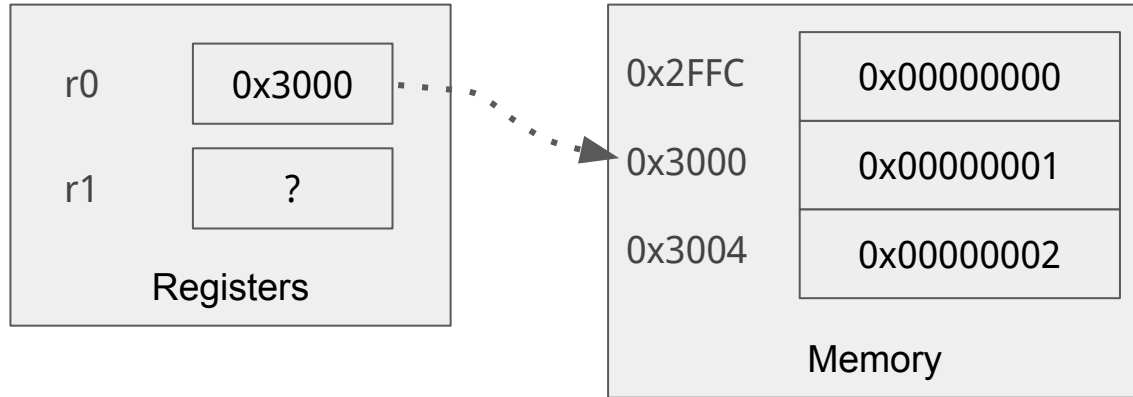
Cache side-channel attacks utilize time differences between a cache hit and a cache miss to infer whether specific code/data has been accessed.

# Cache Side-Channel Attack

; Assume r0 = 0x3000

; Load a word:

LDR r1, [r0]

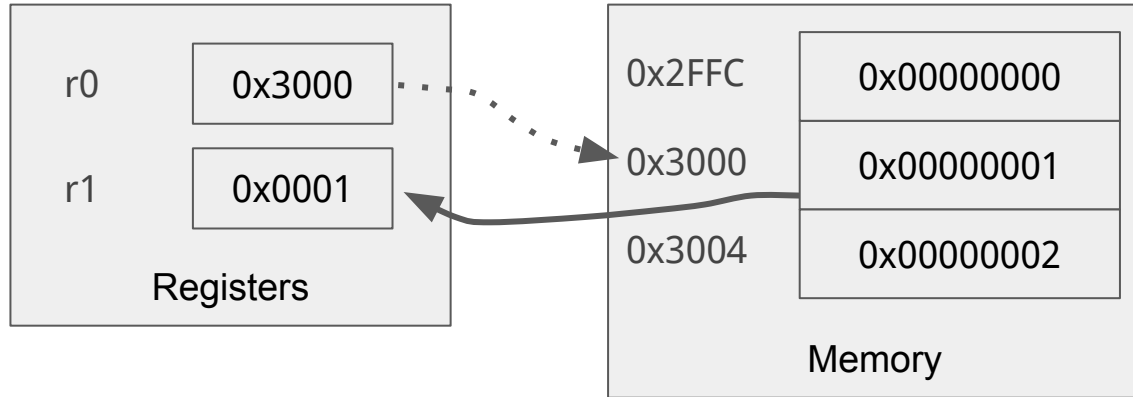


# Cache Side-Channel Attack

; Assume r0 = 0x3000

; Load a word:

LDR r1, [r0]



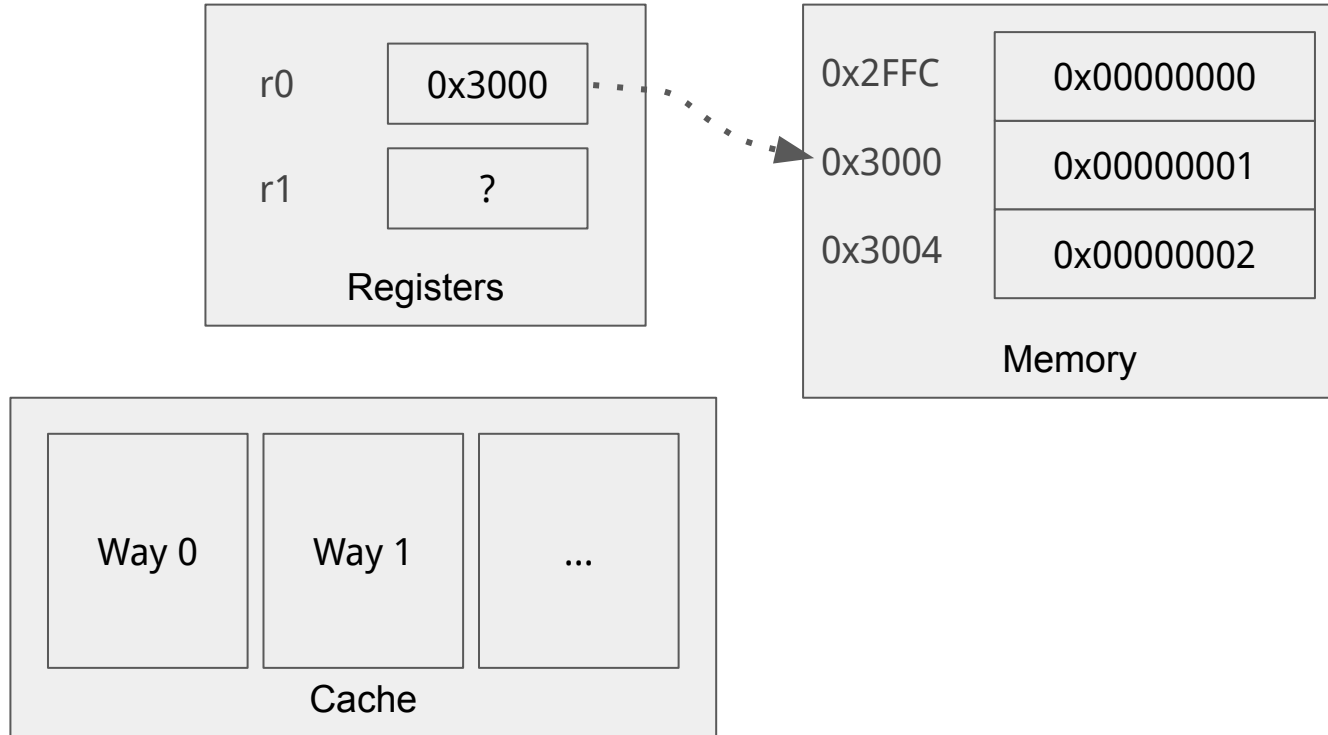


# Cache Side-Channel Attack

; Assume r0 = 0x3000

; Load a word:

LDR r1, [r0]

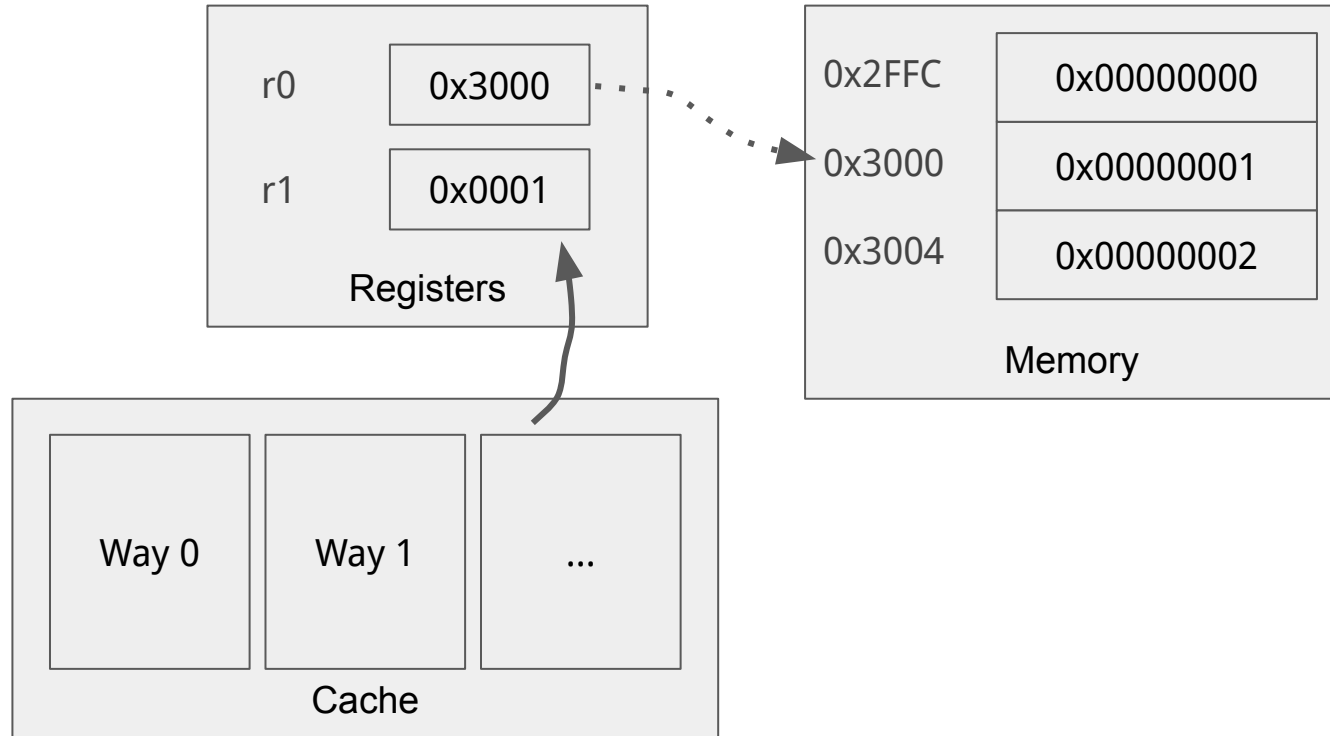


# Cache Side-Channel Attack

; Assume r0 = 0x3000

; Load a word:

LDR r1, [r0]

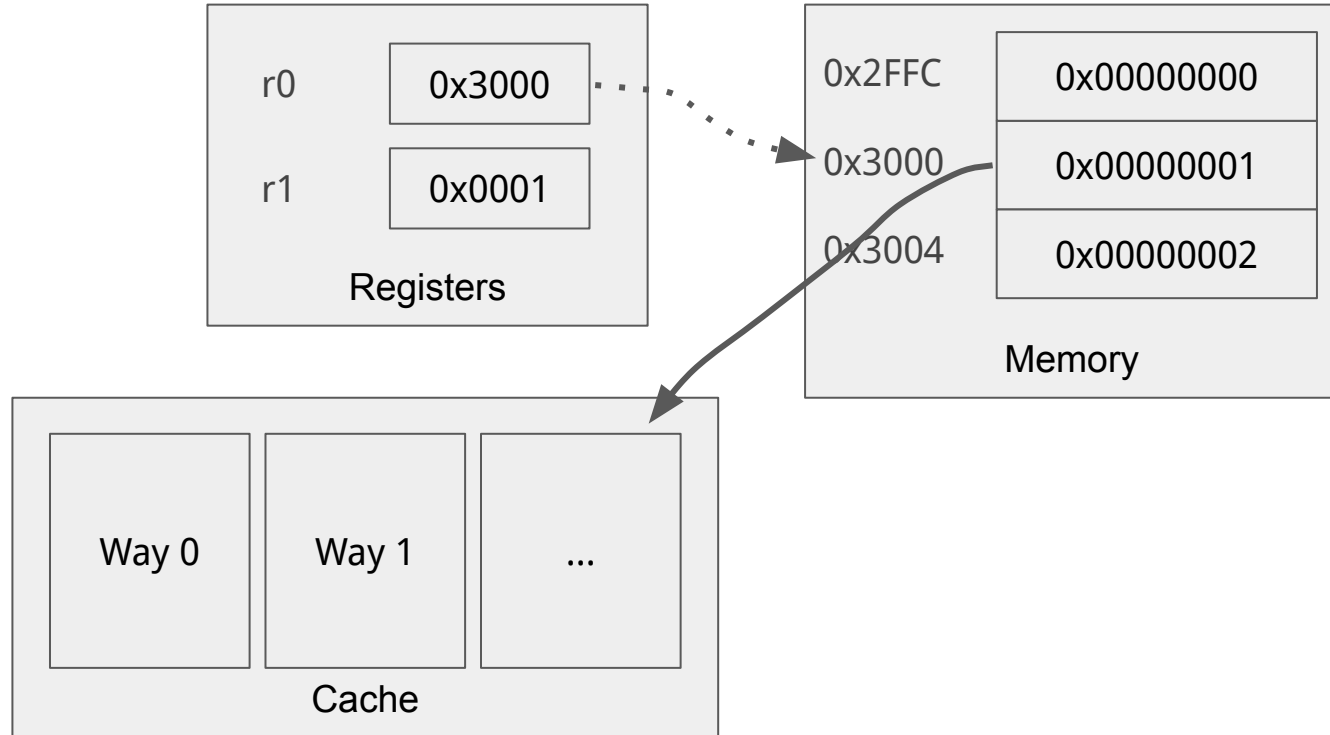


# Cache Side-Channel Attack

; Assume r0 = 0x3000

; Load a word:

LDR r1, [r0]

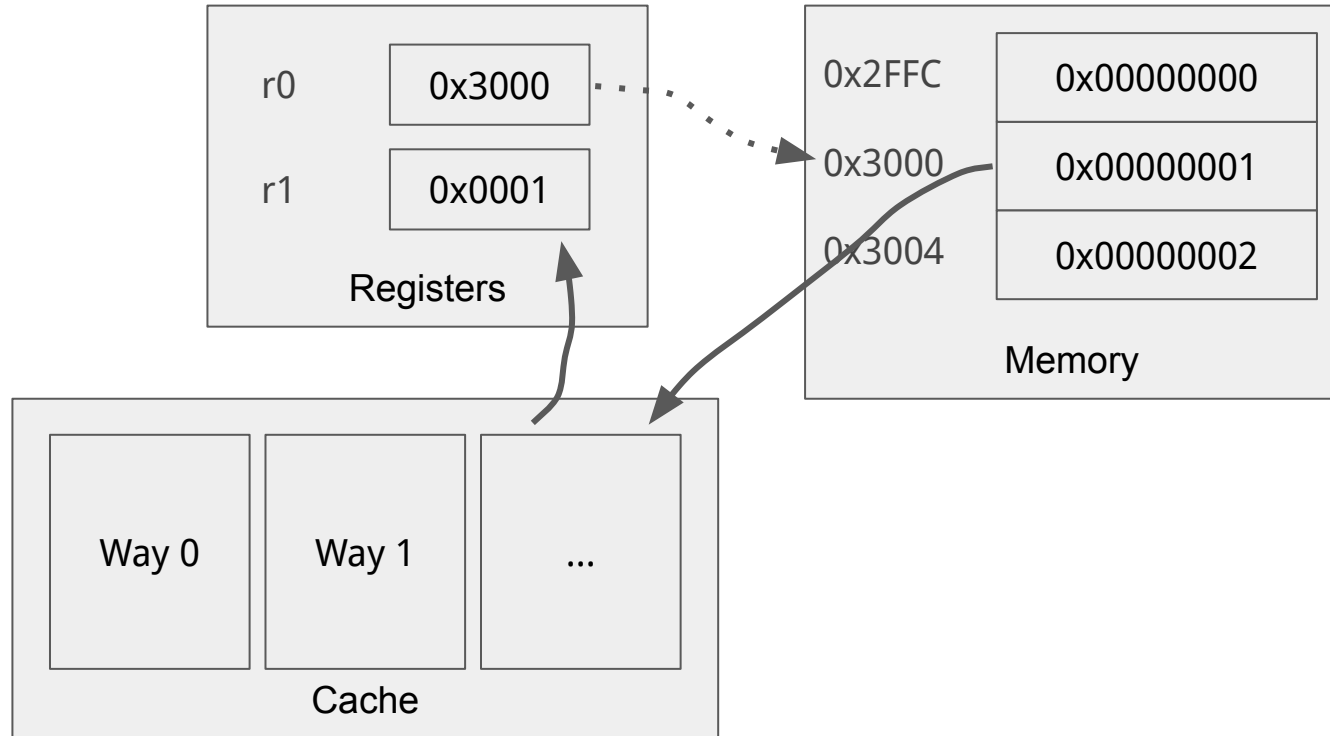


# Cache Side-Channel Attack

; Assume r0 = 0x3000

; Load a word:

LDR r1, [r0]



# Cache Side-Channel Attack

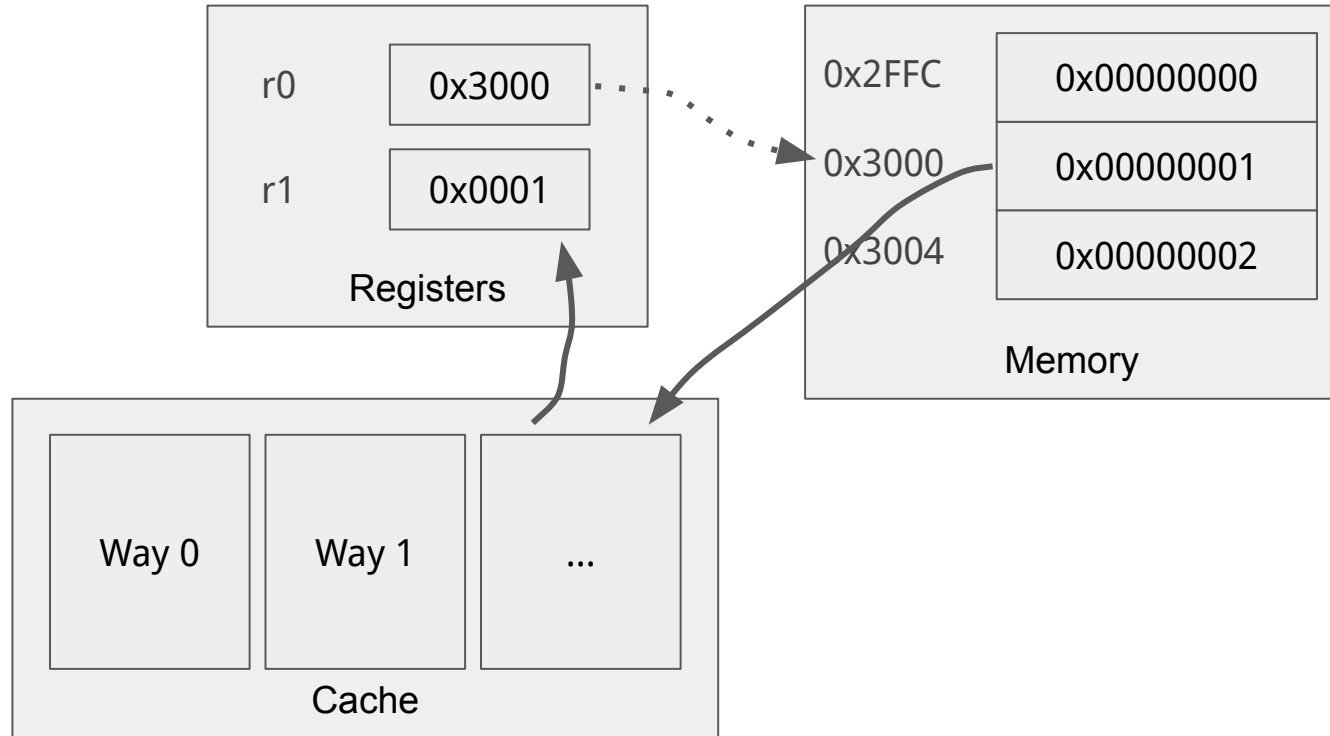
; Assume r0 = 0x3000

; 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.

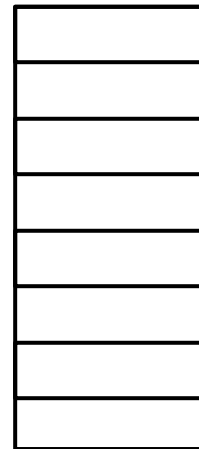
Osvik et al. [63] and Tromer et al. [81] demonstrated with *Evict+Time* a powerful type of attack against AES on OpenSSL implementations that requires neither knowledge of the plaintext nor the ciphertext.

# Prime+Probe

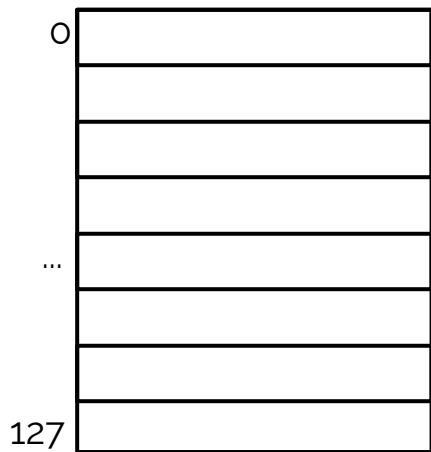
Step 1 Prime: Attacker occupies a set



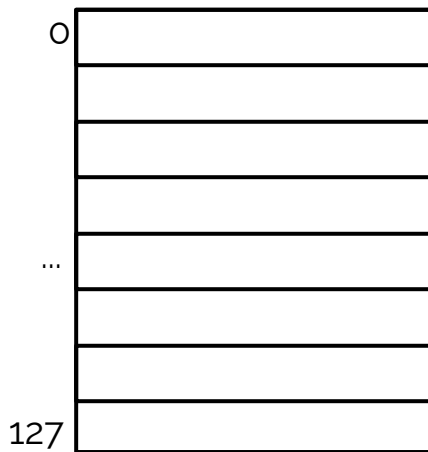
Attacker Address Space



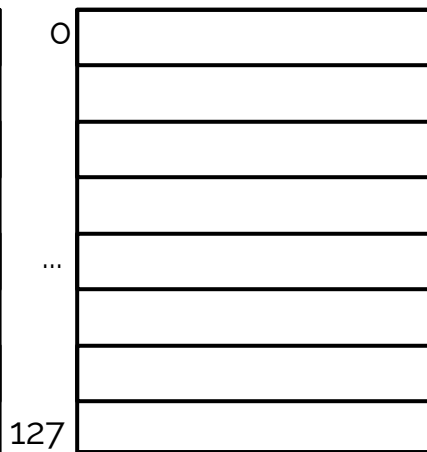
Victim Address Space



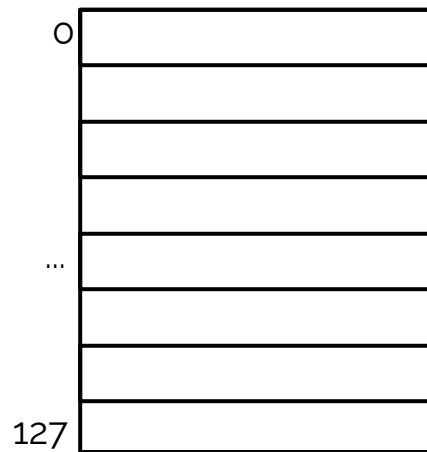
Way 0



Way 1



Way 2



Way 3

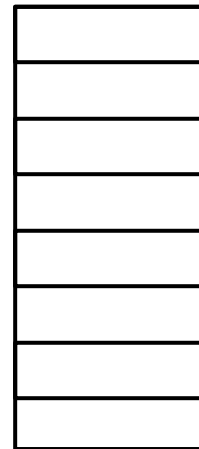


# Prime+Probe

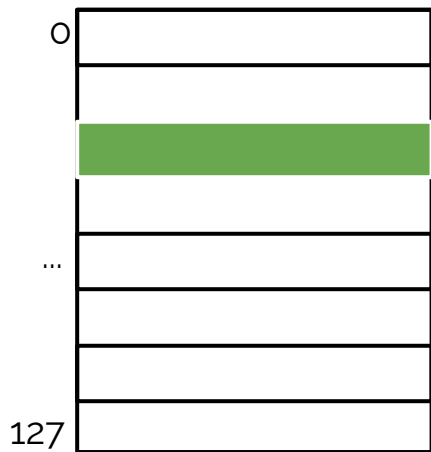
Step 1 Prime: Attacker occupies a set



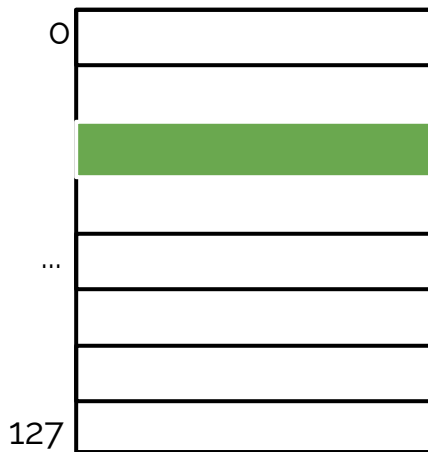
Attacker Address Space



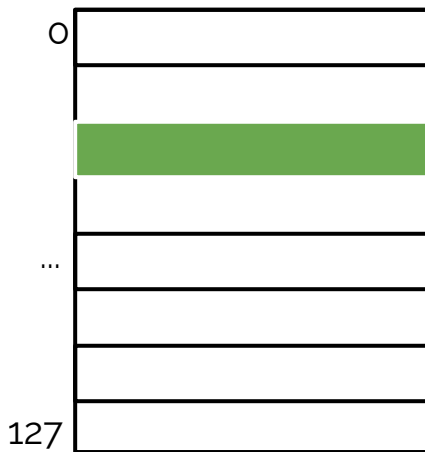
Victim Address Space



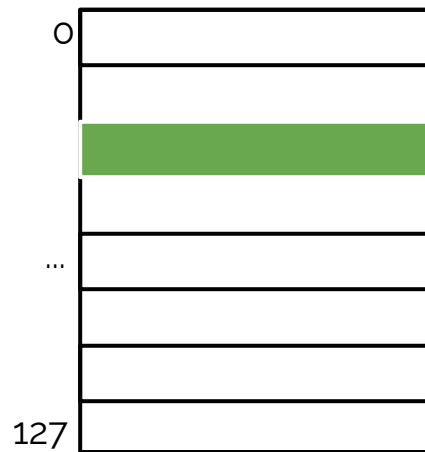
Way 0



Way 1



Way 2



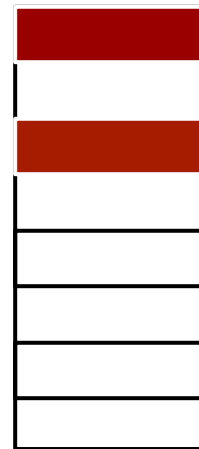
Way 3

# Prime+Probe

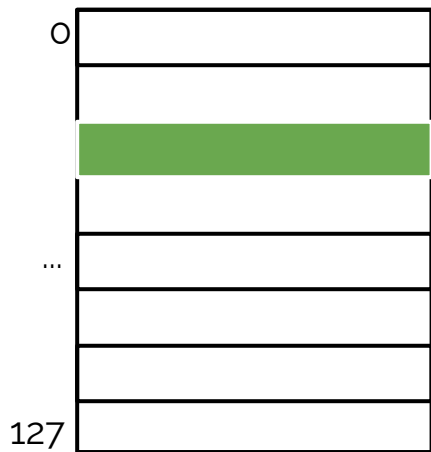
Step 2: Victim runs



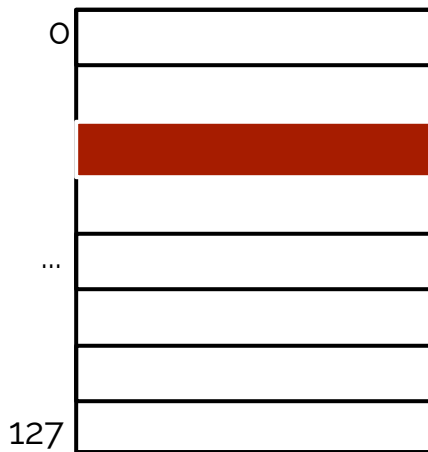
Attacker Address Space



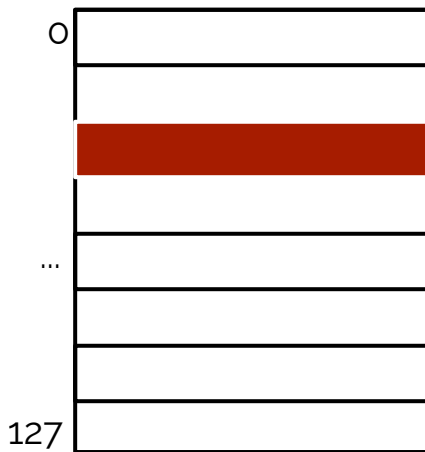
Victim Address Space



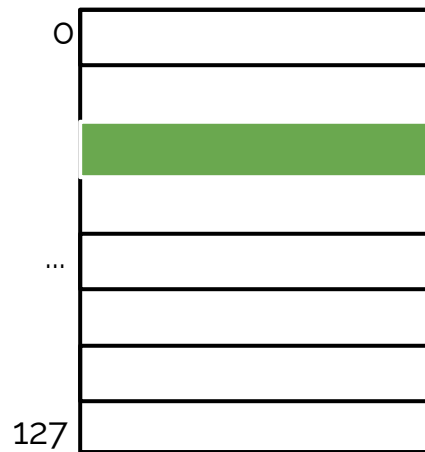
Way 0



Way 1



Way 2



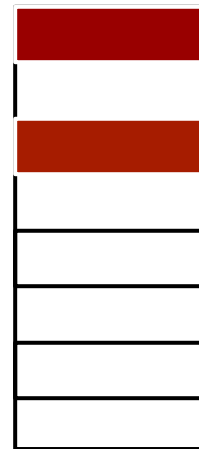
Way 3

# Prime+Probe

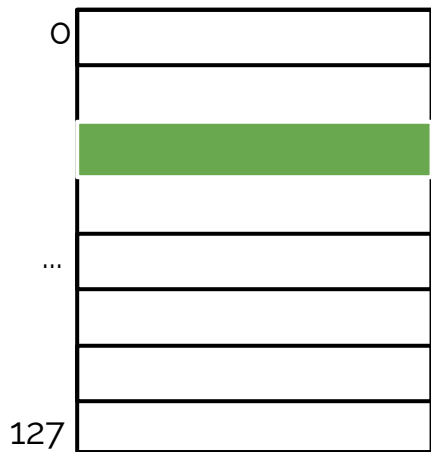
Step 3 Probe: Attacker accesses  
memory again and measures the  
time



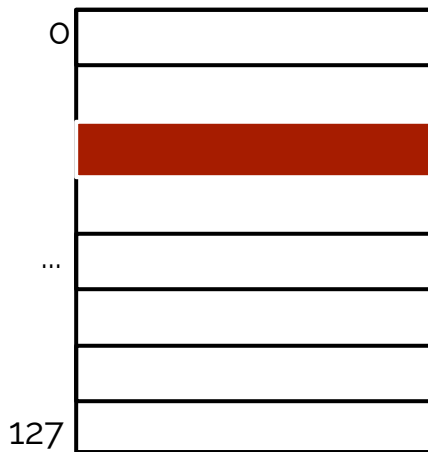
Attacker Address Space



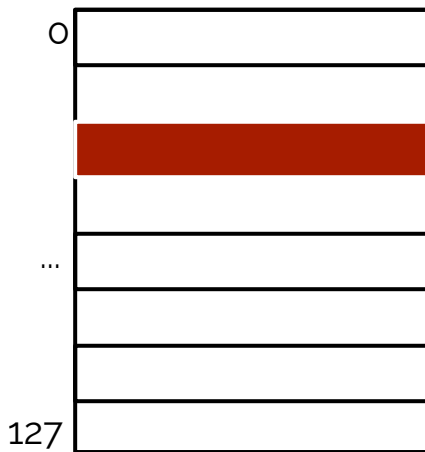
Victim Address Space



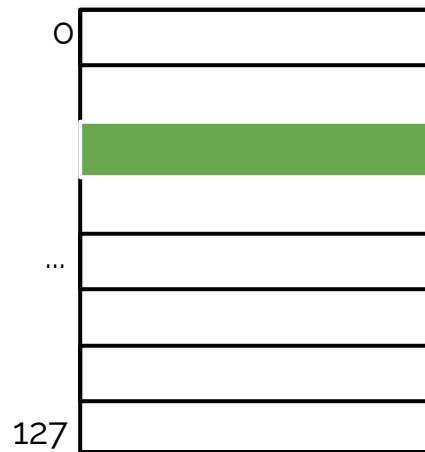
Way 0



Way 1



Way 2



Way 3

# Flush+Reload

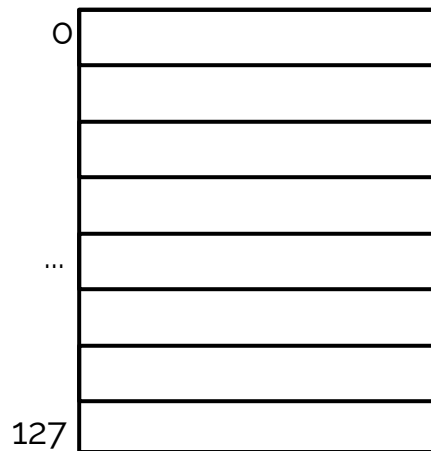
A memory block is cached



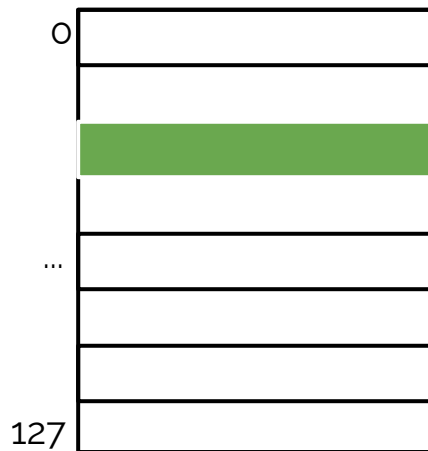
Attacker Address Space



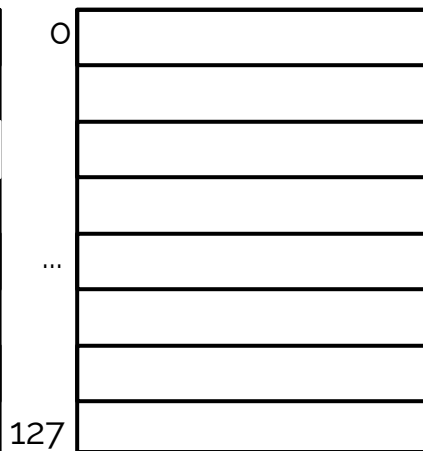
Victim Address Space



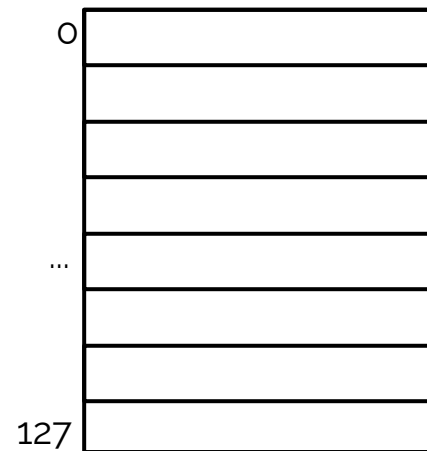
Way 0



Way 1



Way 2



Way 3

# Flush+Reload

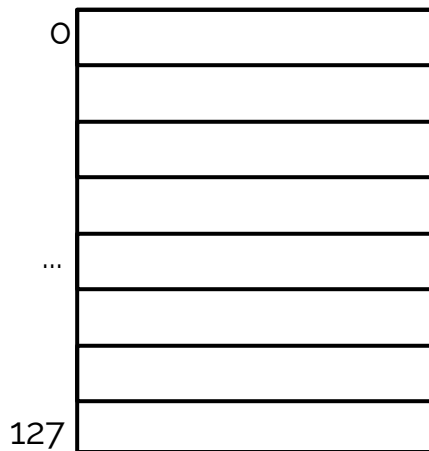
Step 1 Flush: Attacker flushes this  
memory block out of cache



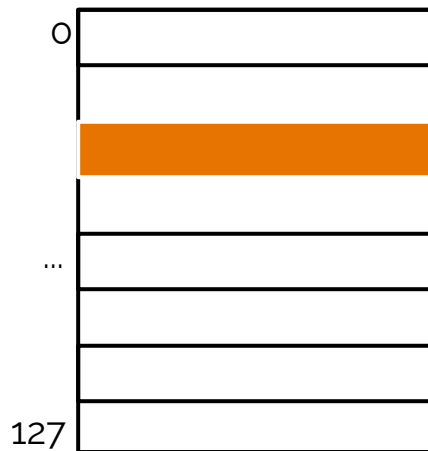
Attacker Address Space



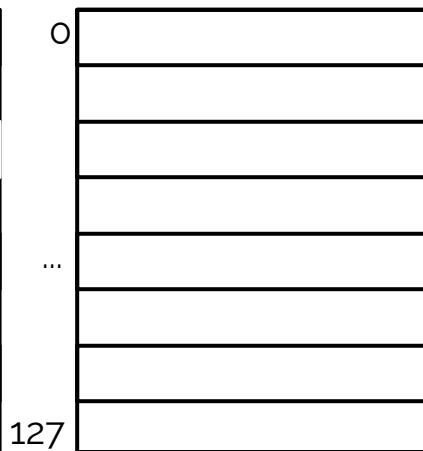
Victim Address Space



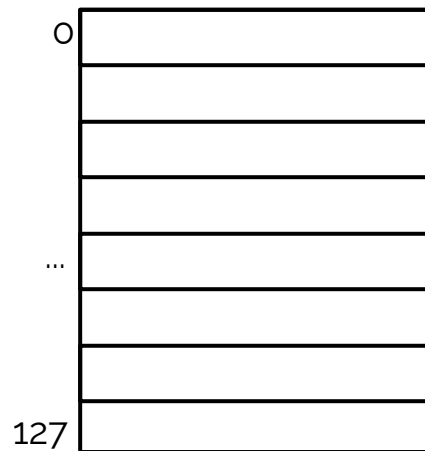
Way 0



Way 1



Way 2



Way 3

# Flush+Reload

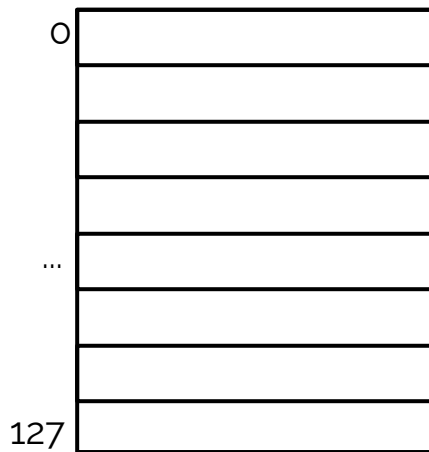
Step 2 Reload: Victim may / may not  
access that block again



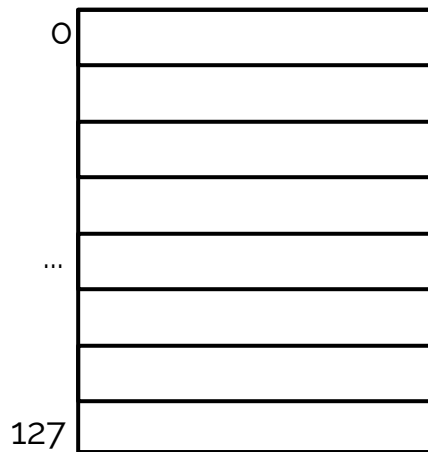
Attacker Address Space



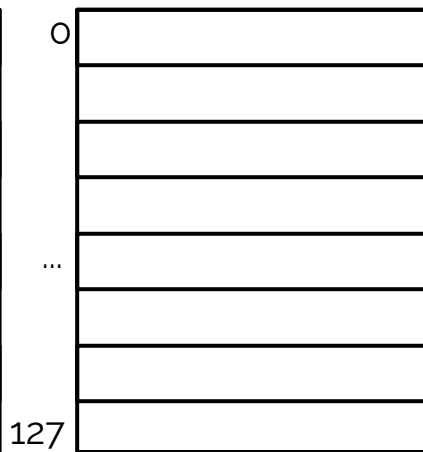
Victim Address Space



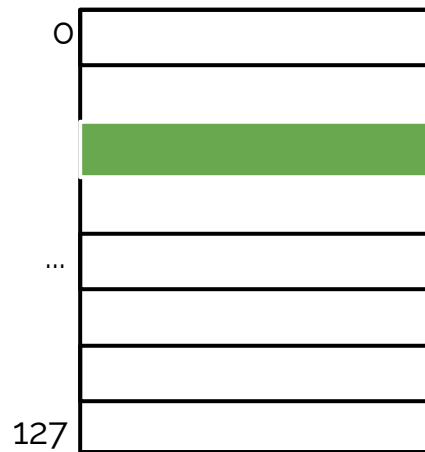
Way 0



Way 1



Way 2



Way 3

# Flush+Reload

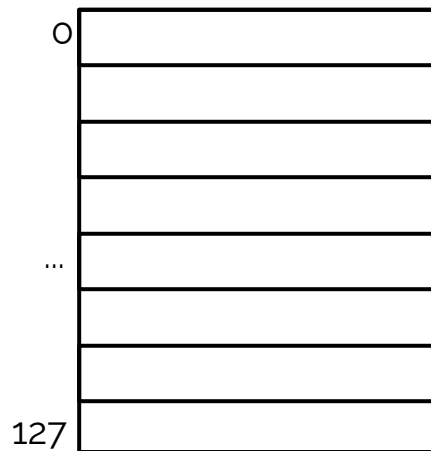
Step 3 Probe: Attacker accesses that  
block again and measure



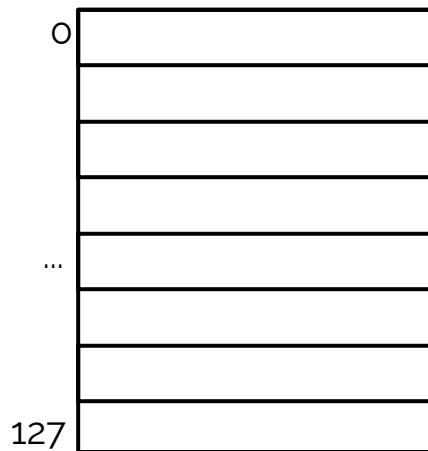
Attacker Address Space



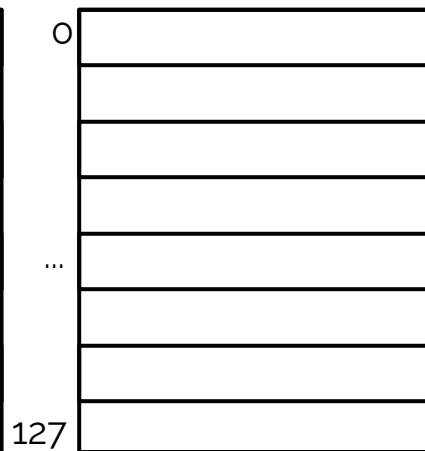
Victim Address Space



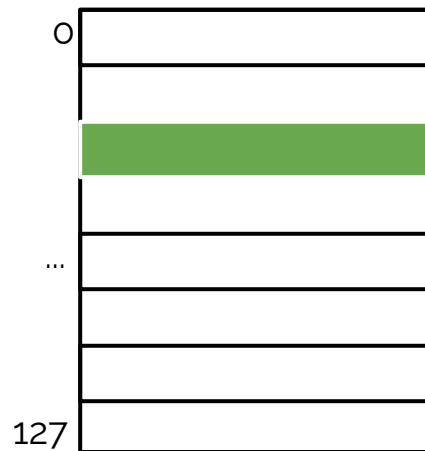
Way 0



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[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
```

[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>

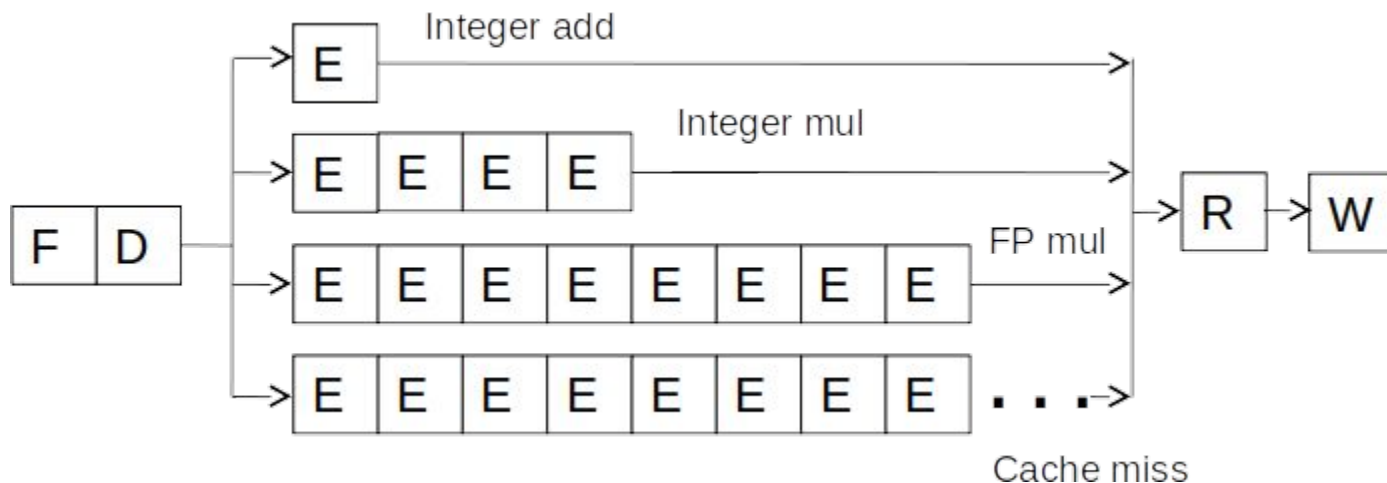
# Meltdown Basics

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

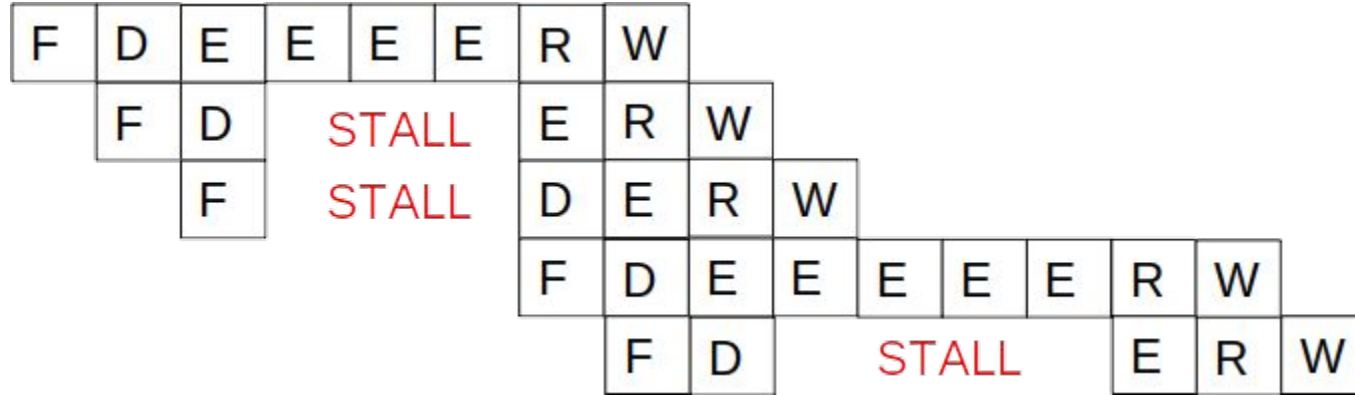
# Out-of-Order Execution (Dynamic Instruction Scheduling)

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



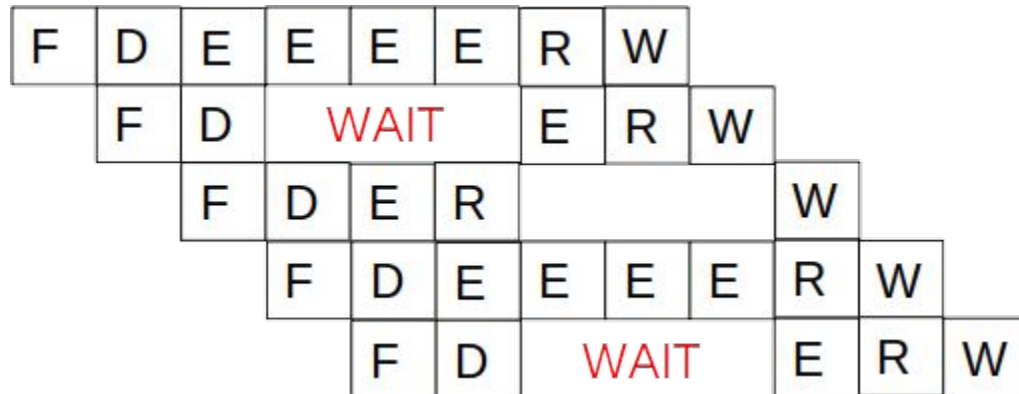
IMUL R3  $\leftarrow$  R1, R2

ADD R3  $\leftarrow$  R3, R1

ADD R1  $\leftarrow$  R6, R7

IMUL R5  $\leftarrow$  R6, R8

ADD R7  $\leftarrow$  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

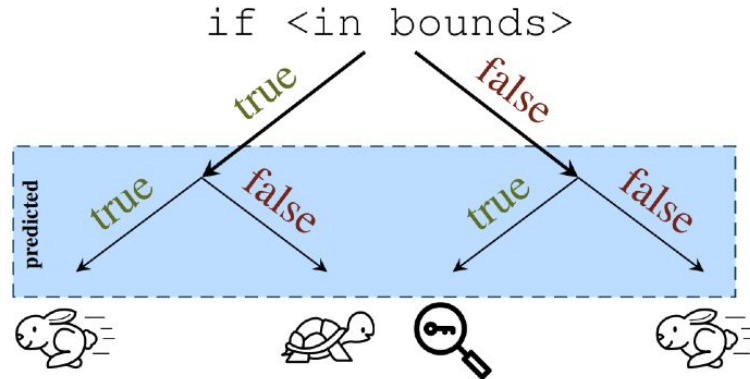
During speculative execution, the processor makes guesses as to the likely outcome of branch instructions.

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.