Cache Attacks on Various CPU Architectures

Jinbum Park
Security Team
Samsung Research
jinb-park.github.io

Introduction

What is Cache Attack? Why it matters?

- Cache Attack allows unprivileged attackers to see sensitive-data by monitoring CPU cache.
- Cache Attack is the most powerful and practical exploitation method.
- Cache Attack is used to exploit recent micro-arch vulnerabilities (meltdown, spectre, …).

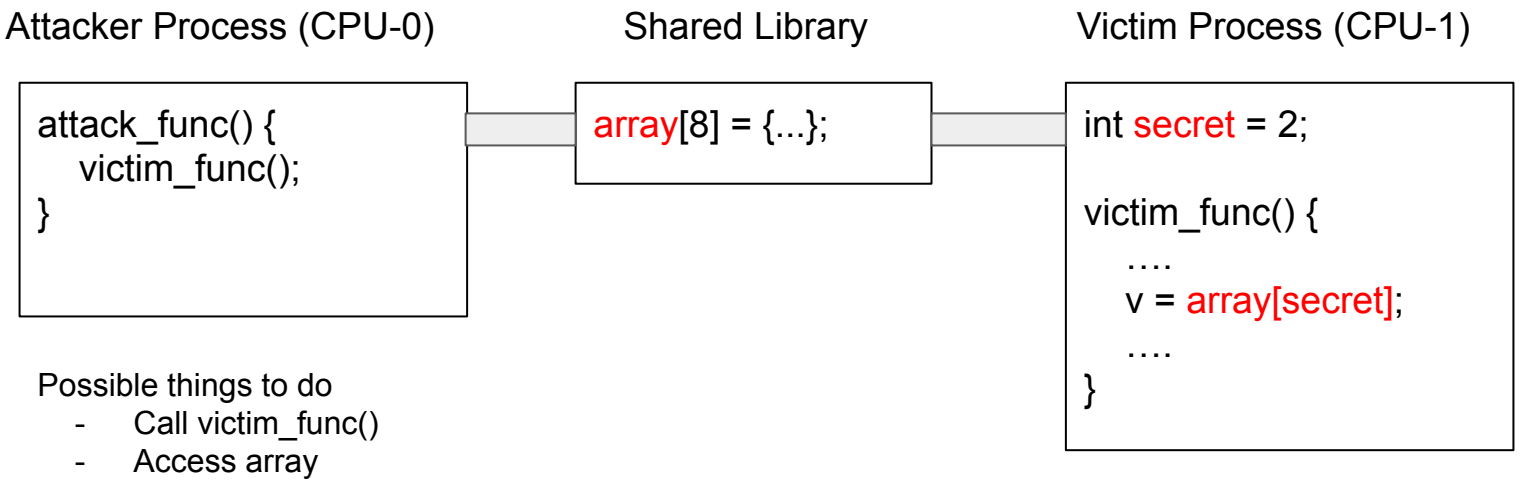
Problem & Goal

- Problem
  - Cache Attacks have been well-studied, but only for Intel CPU.
  - But, way of cache attacks vary depends on CPU type. (ARM, Intel)

- Goal
  - Look at various CPU cache designs and how the different designs affect way of cache attacks. Specifically, focus on comparing Intel and ARM.

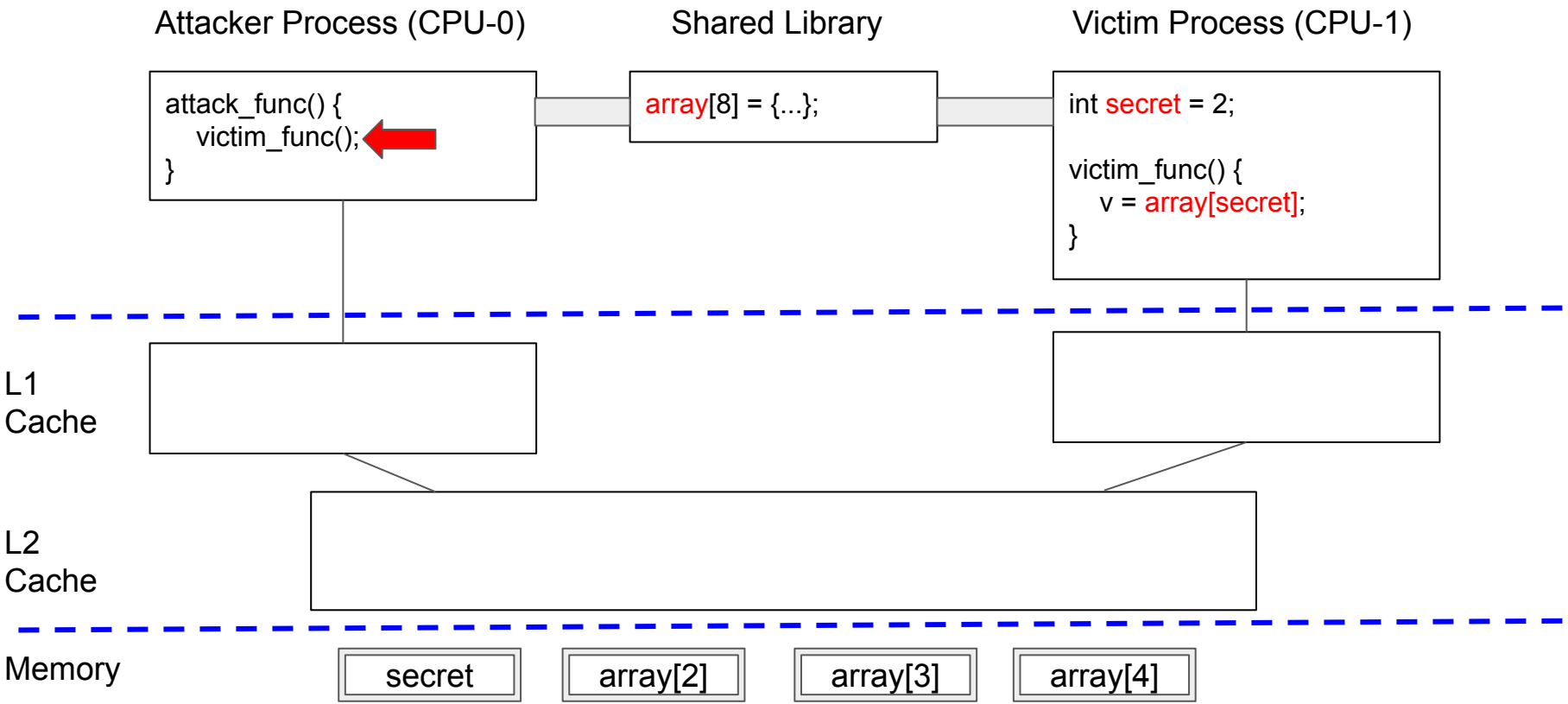
Simple example of cache attack

Cache attack example

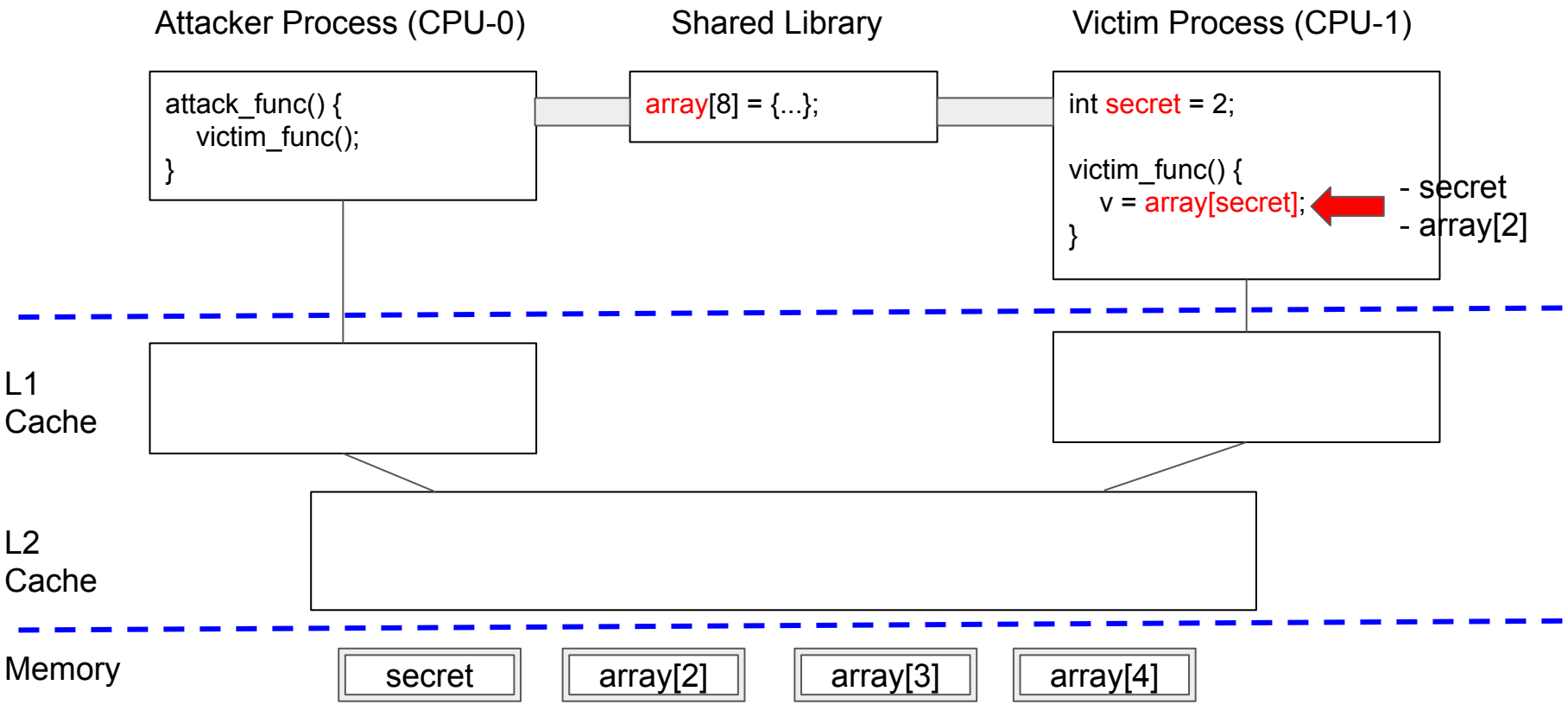


How can attackers leak secret in this environment, by cache attack?

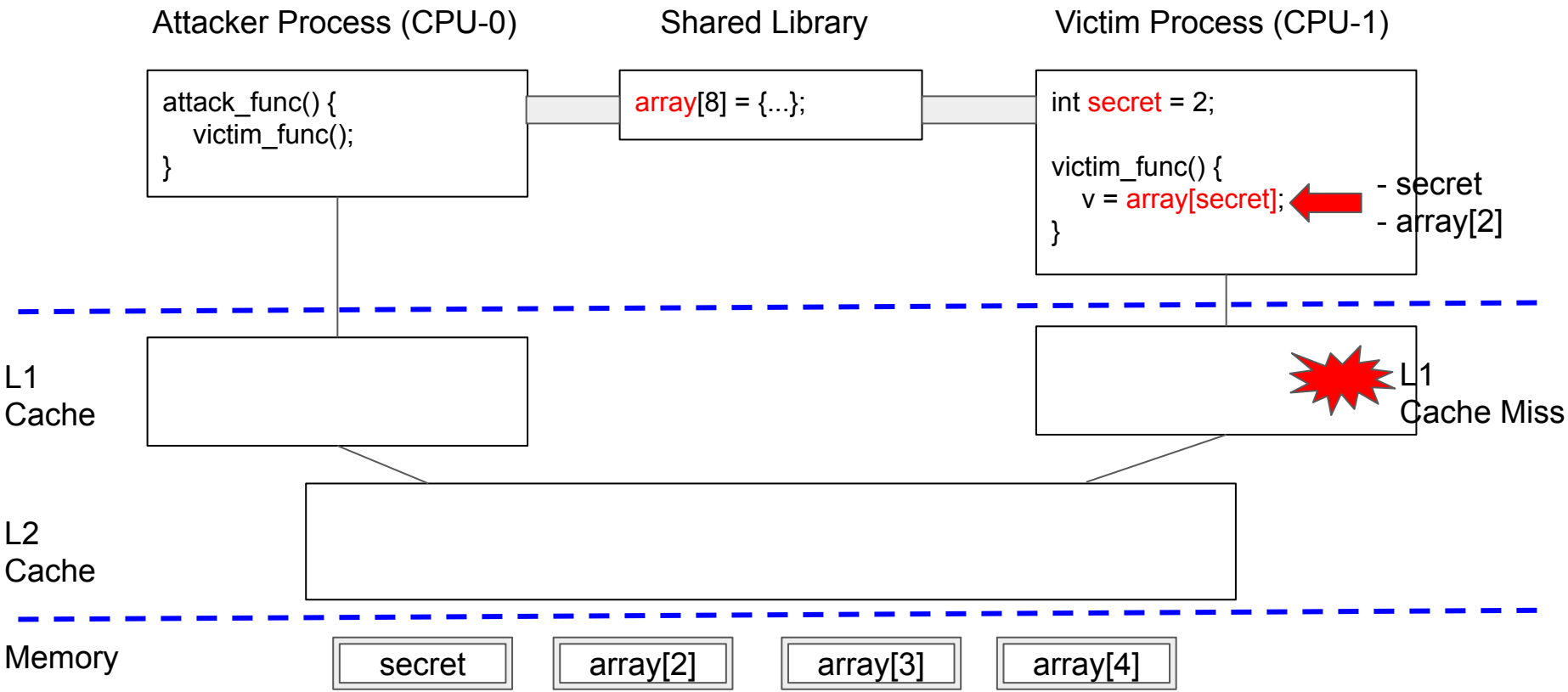
Cache attack example: Warm-up



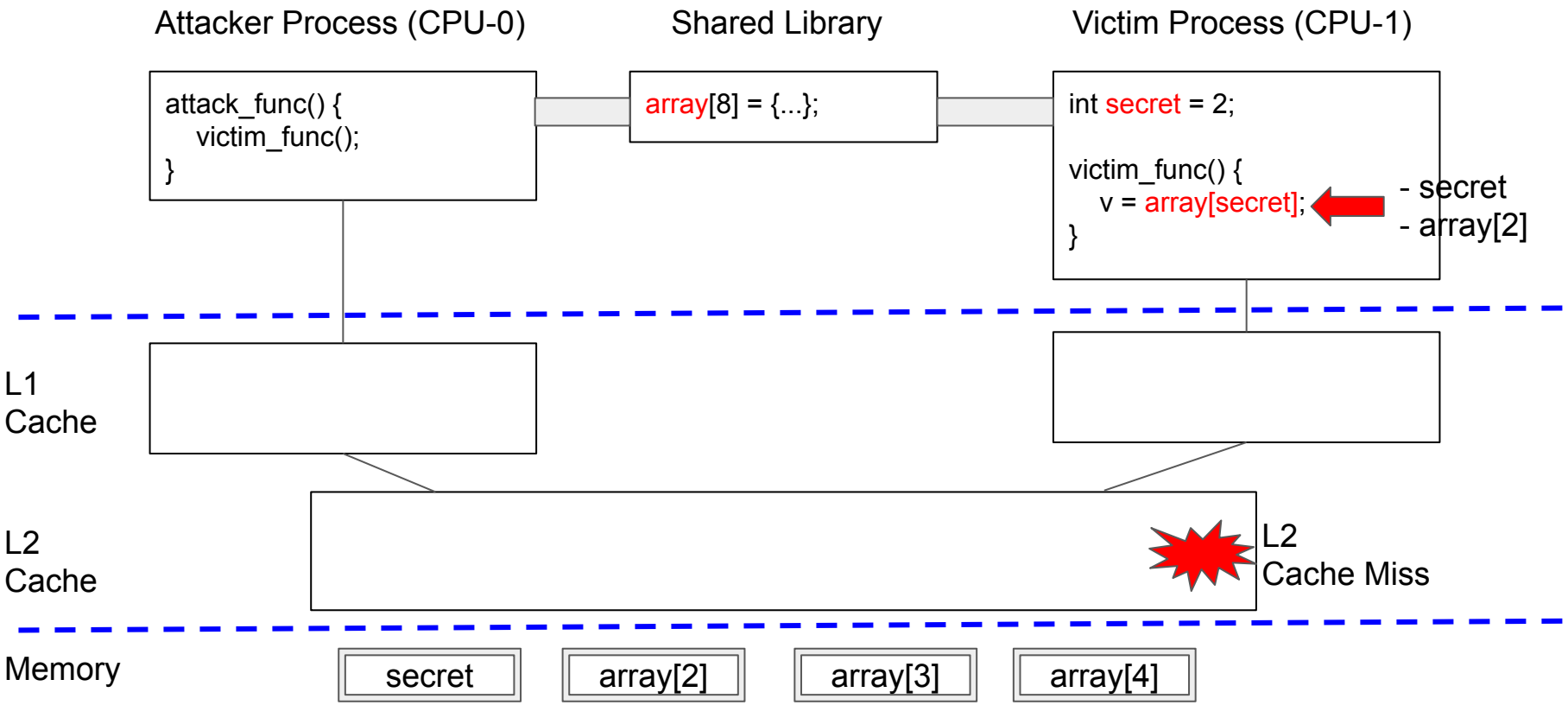
Cache attack example: Warm-up (Cont)



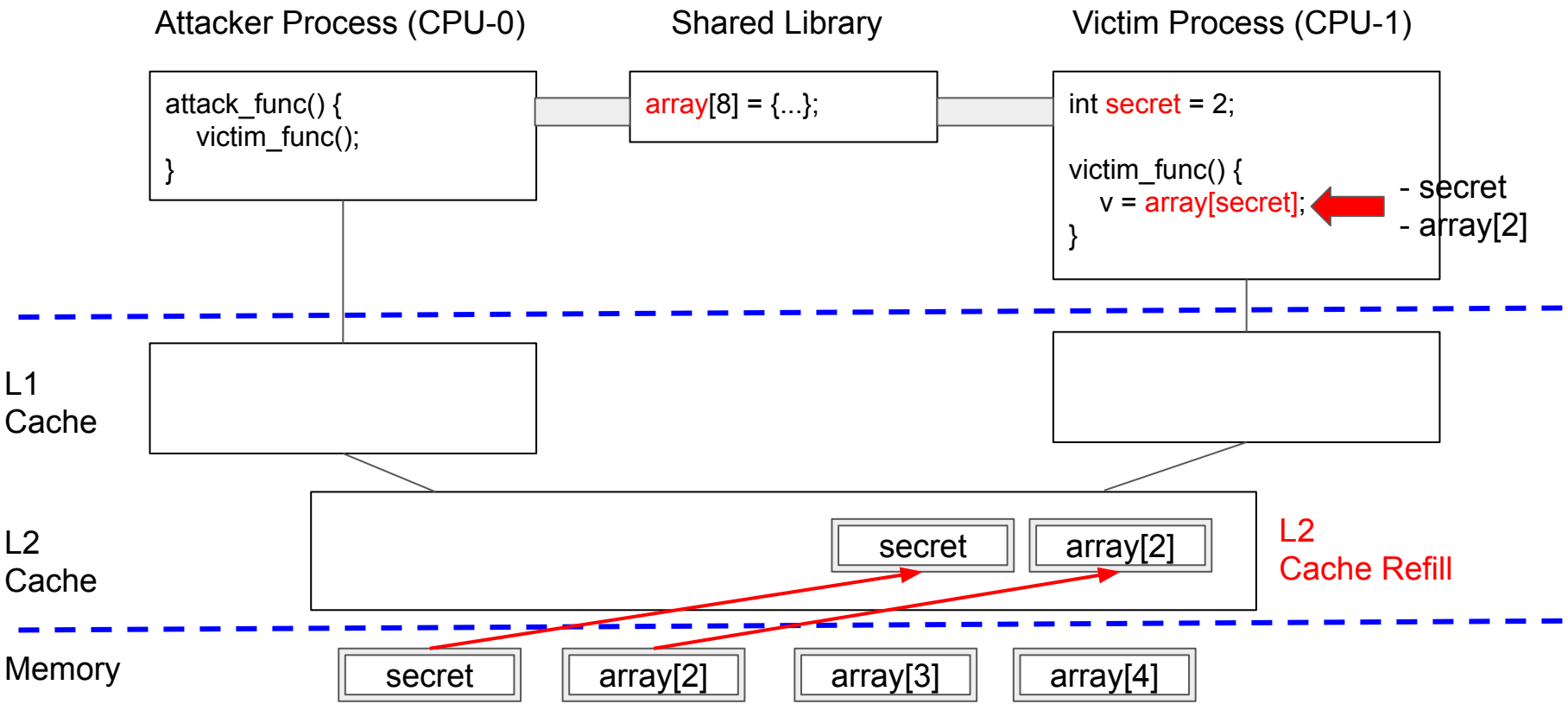
Cache attack example: Warm-up (Cont)



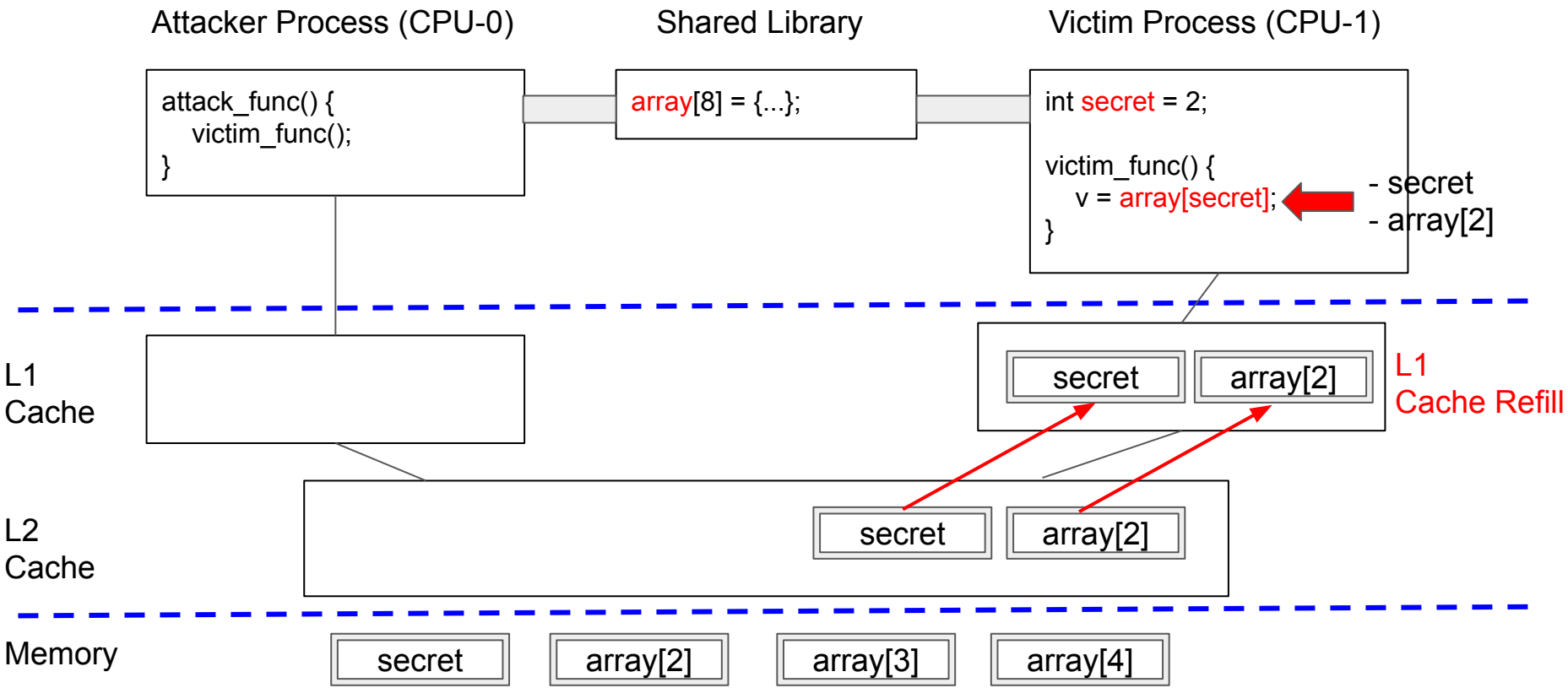
Cache attack example: Warm-up (Cont)



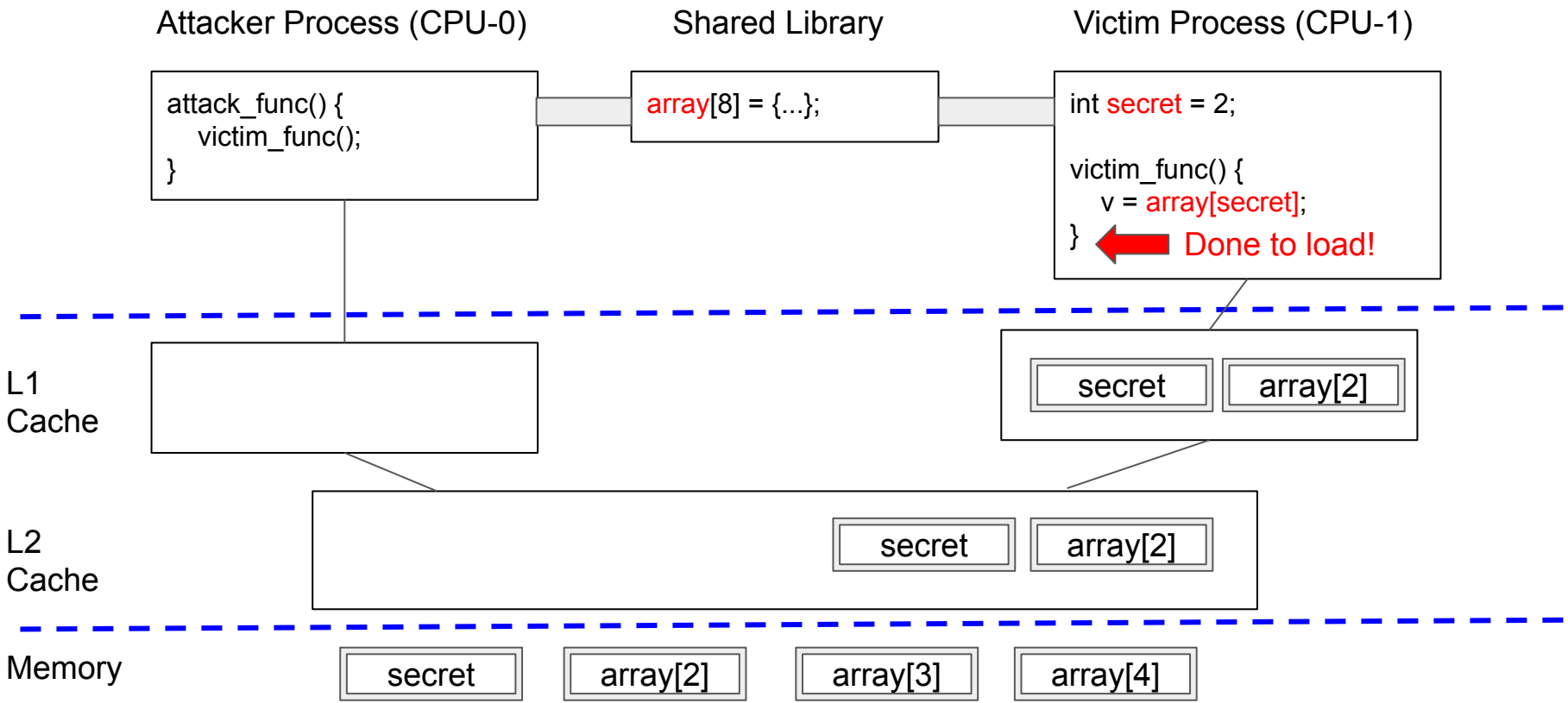
Cache attack example: Warm-up (Cont)



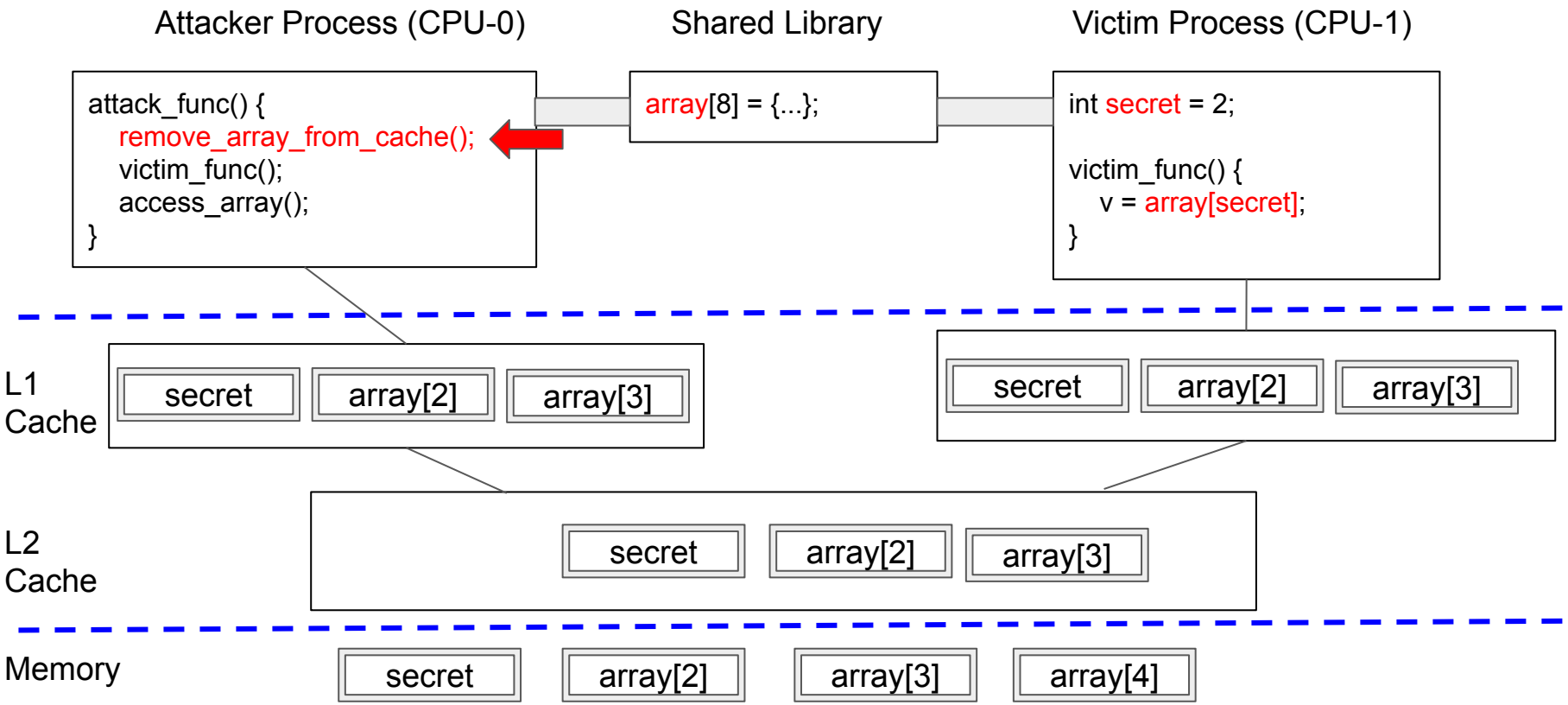
Cache attack example: Warm-up (Cont)



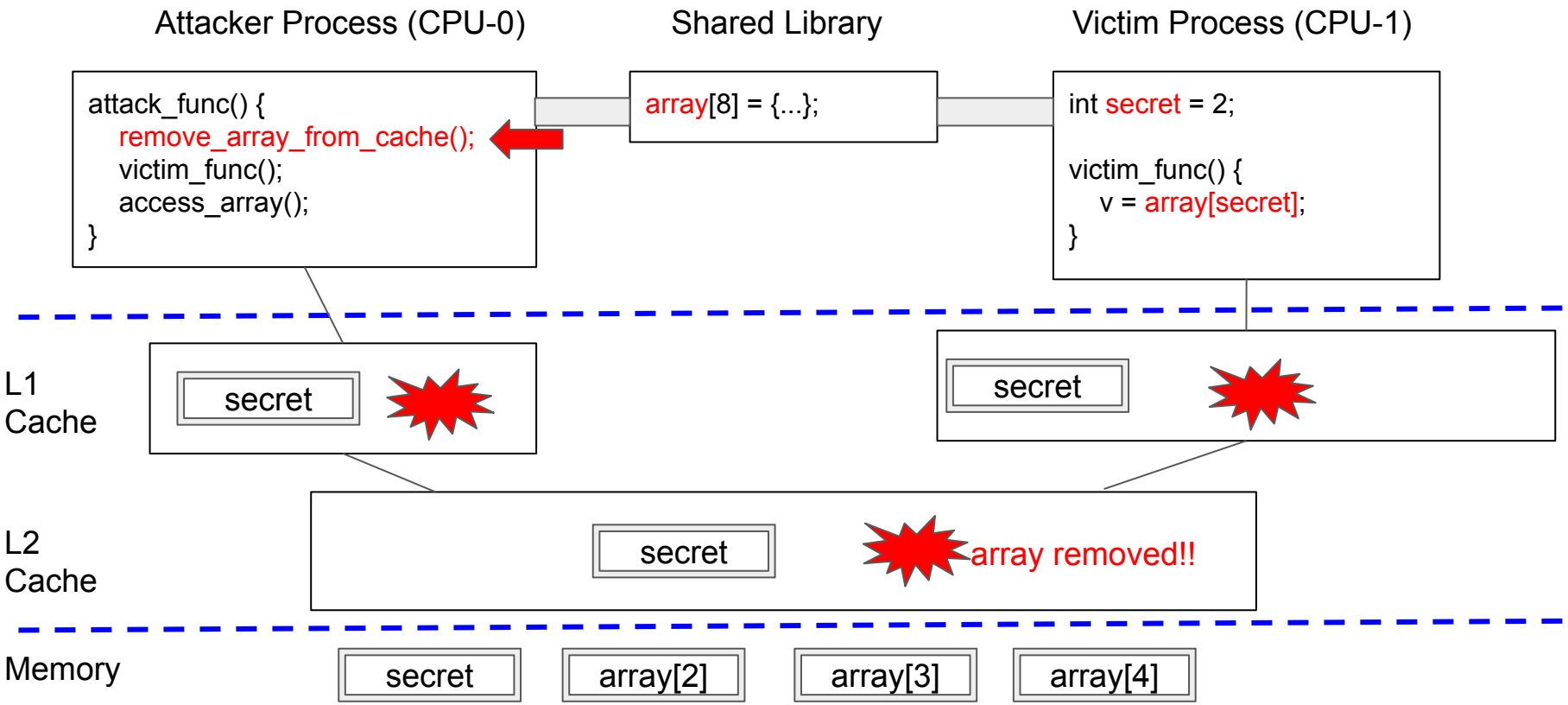
Cache attack example: Warm-up (Cont)



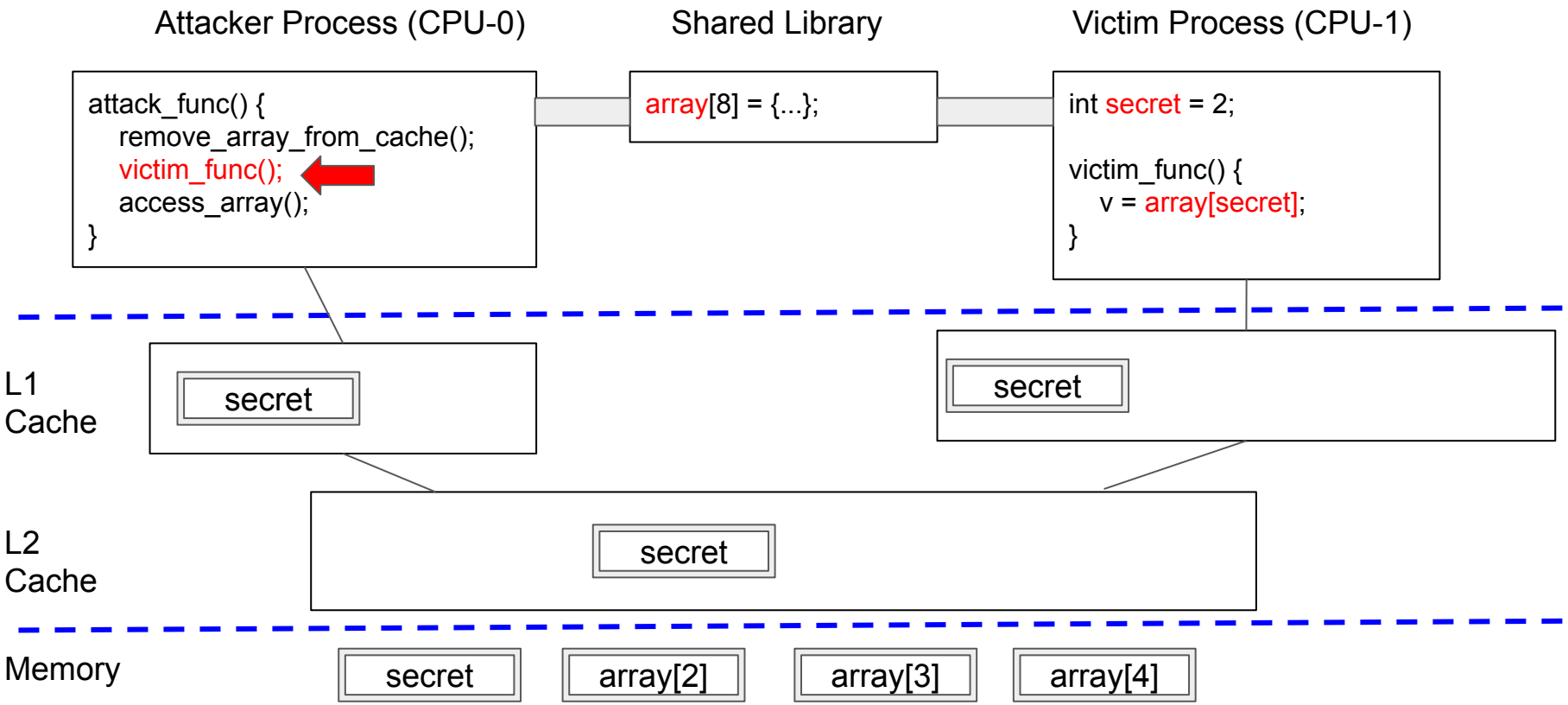
Cache attack example: Attack



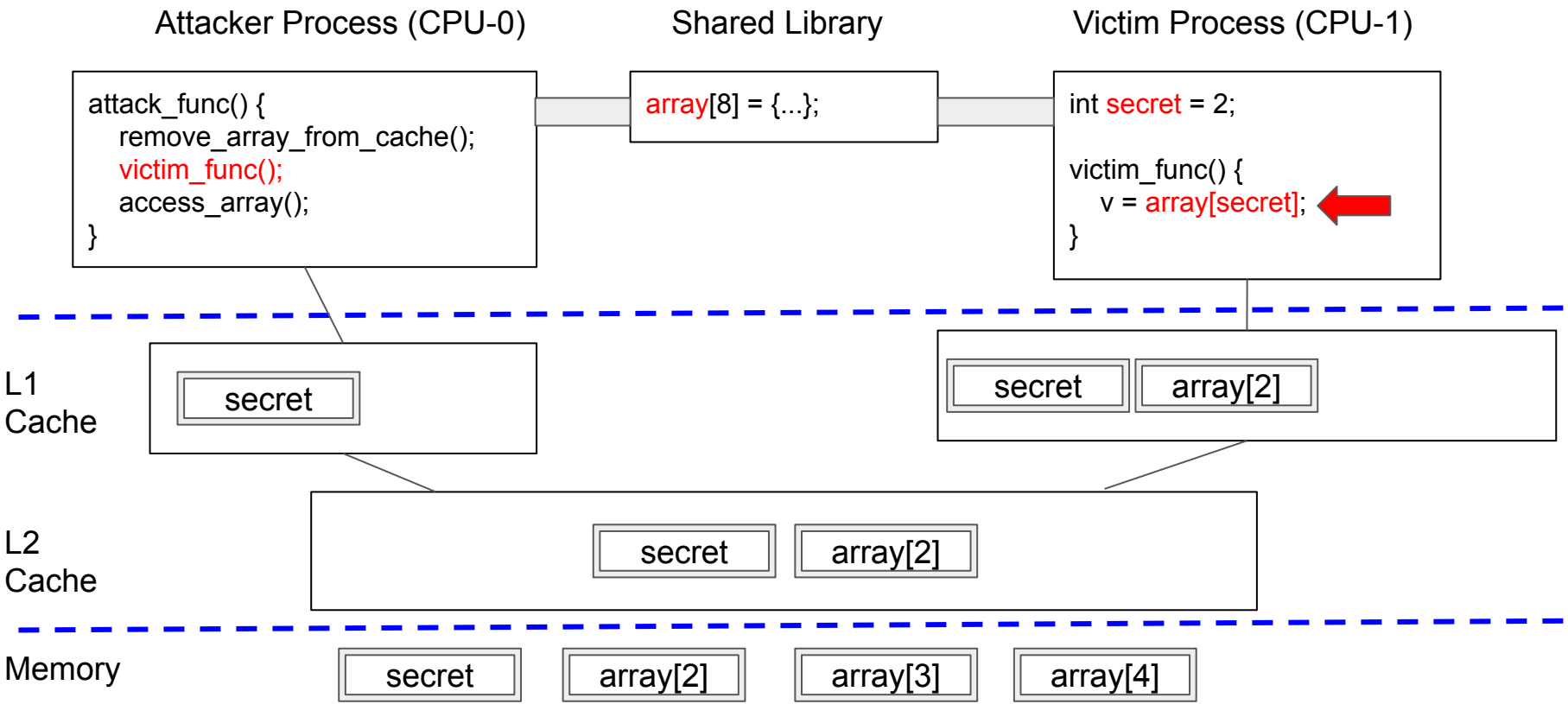
Cache attack example: Attack



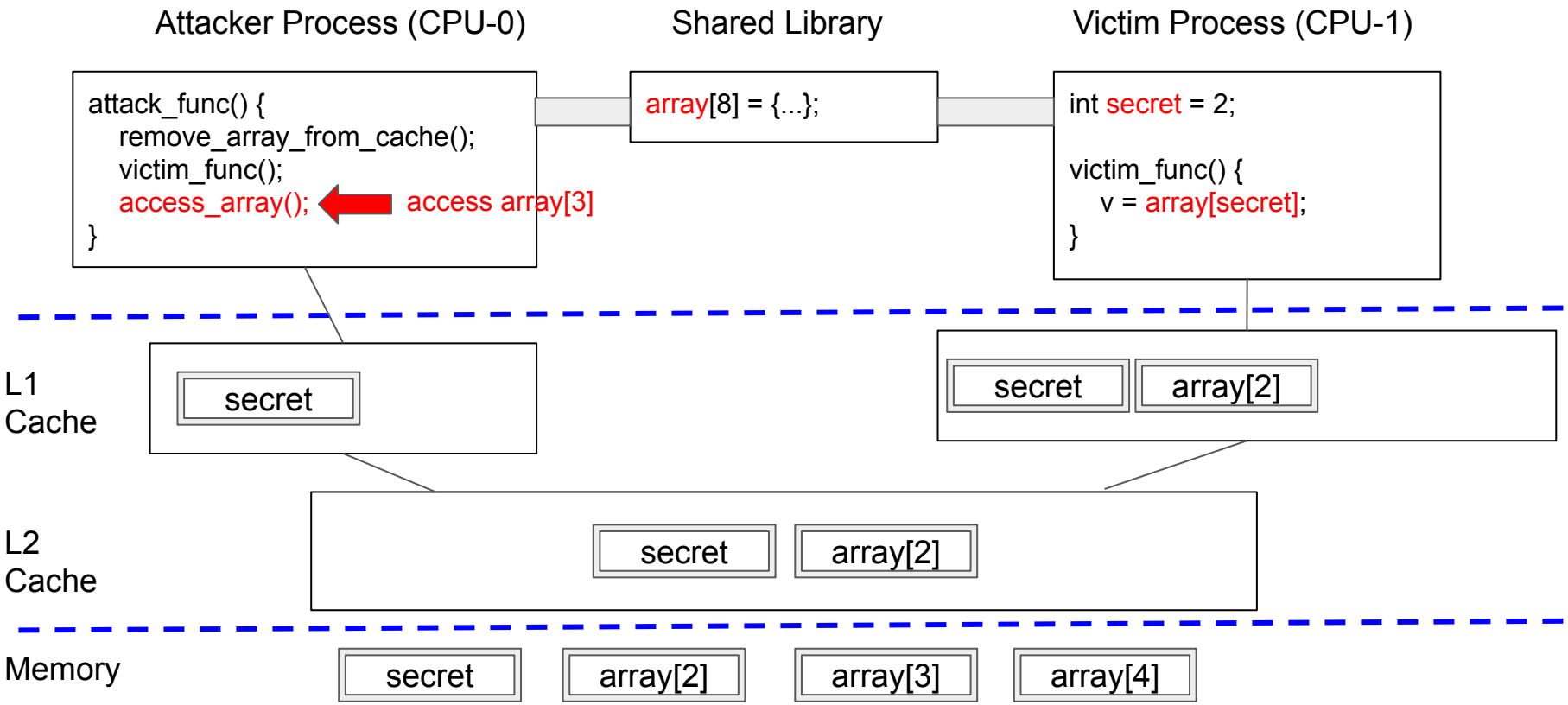
Cache attack example: Attack (Cont)



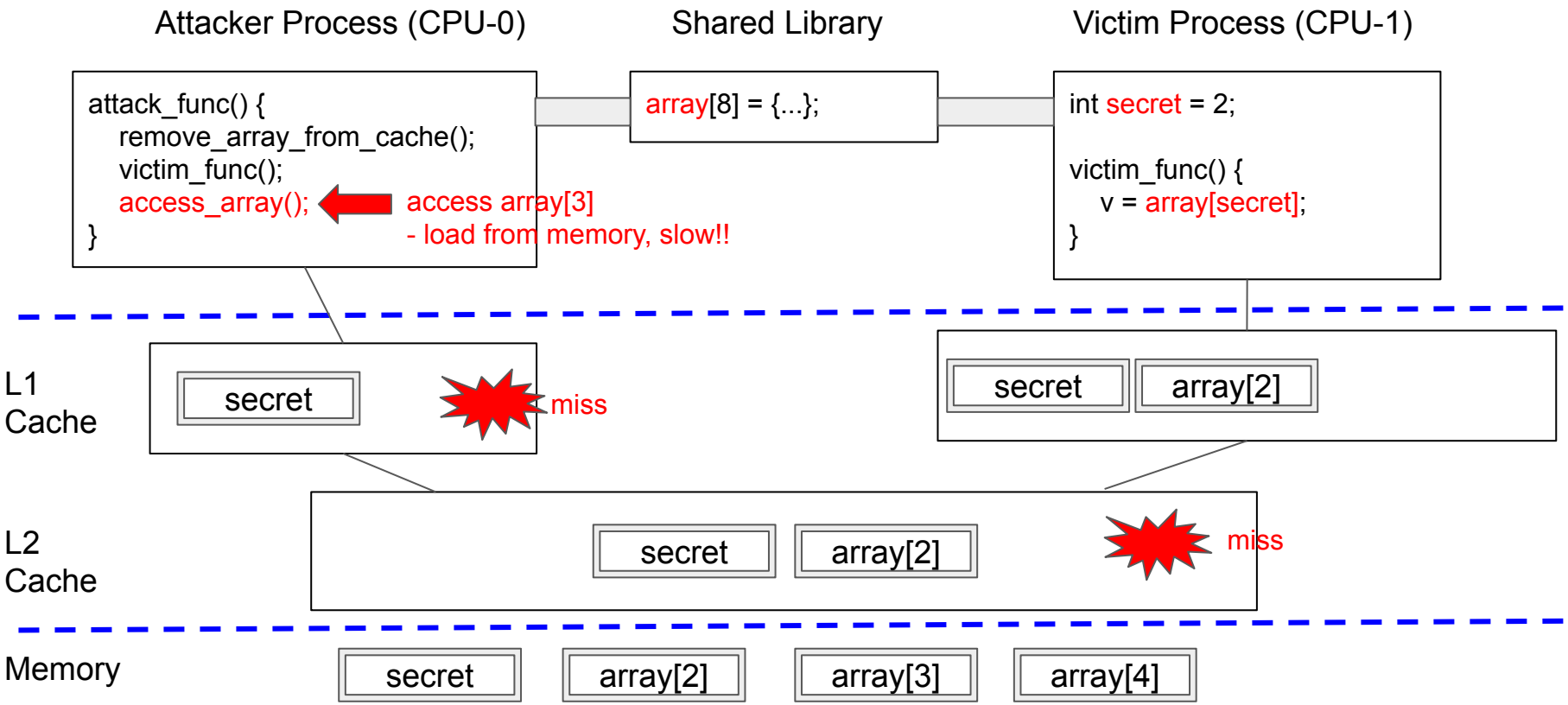
Cache attack example: Attack (Cont)



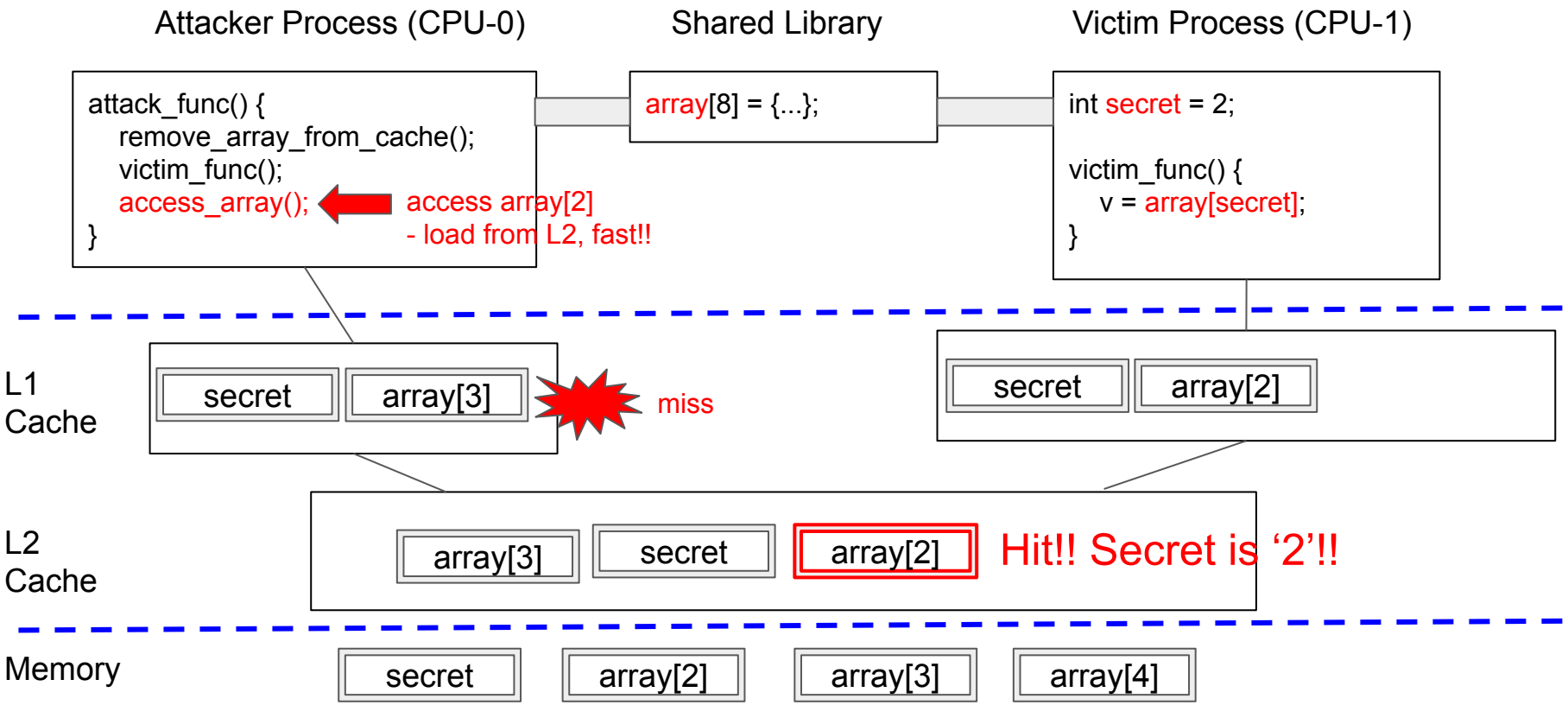
Cache attack example: Attack (Cont)



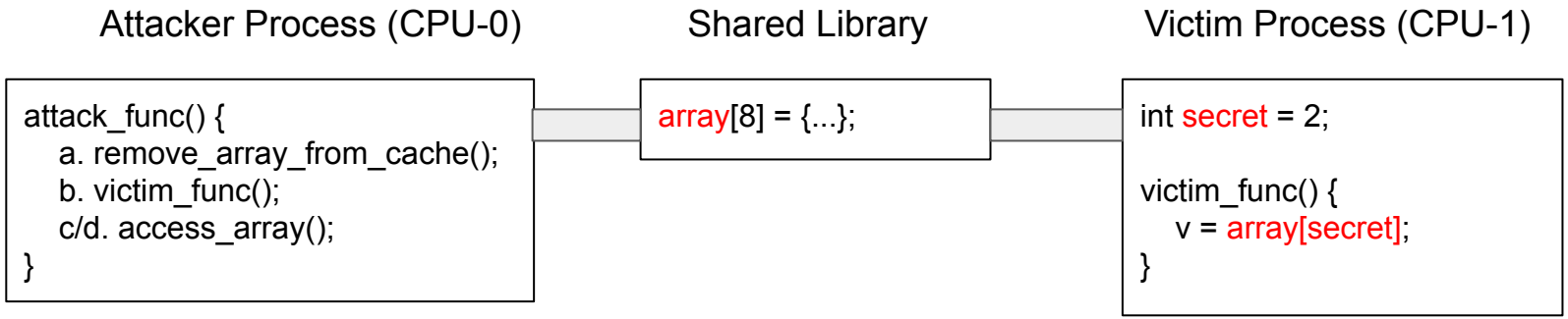
Cache attack example: Attack (Cont)



Cache attack example: Attack (Cont)



Cache attack example: Summary



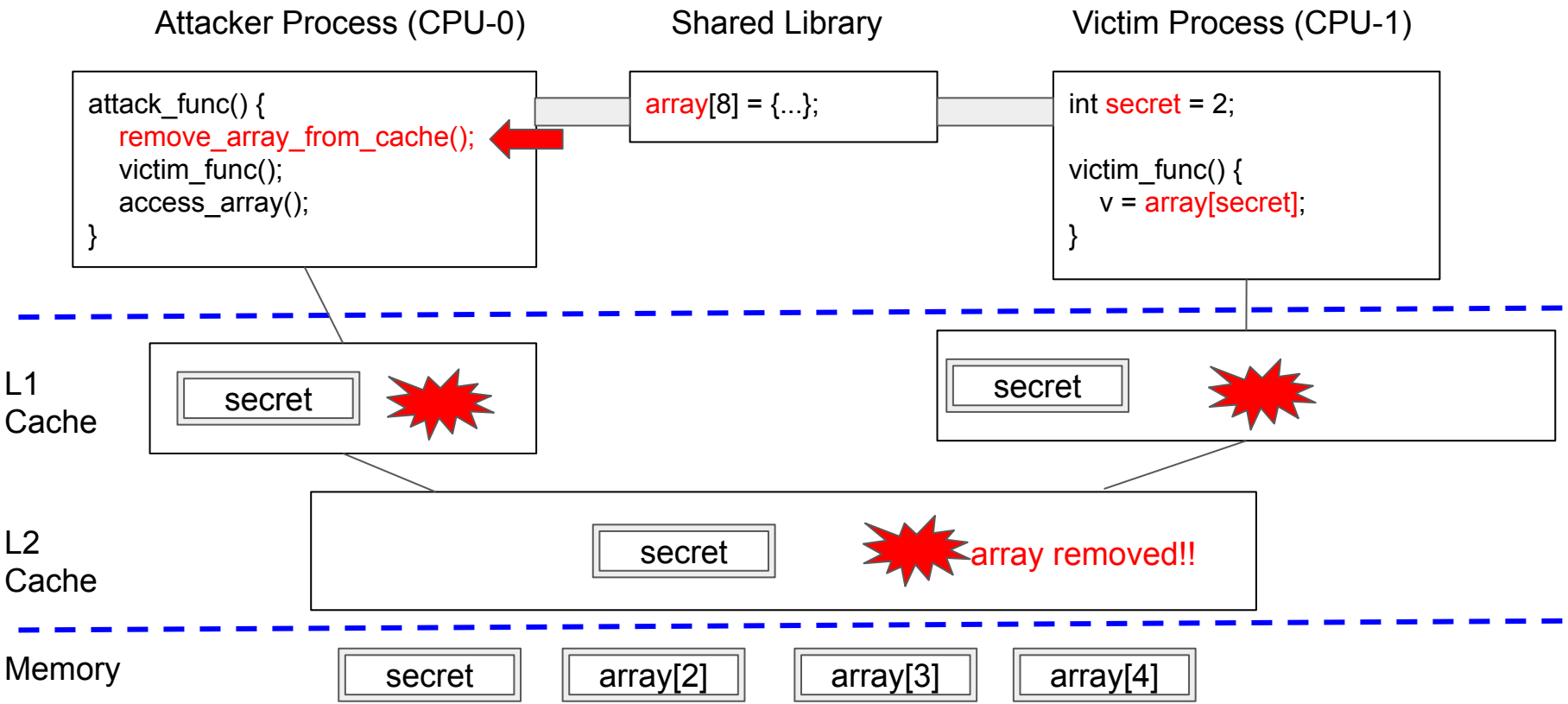
- **Cache attack phases**
  a. Remove secret-related data (array[0]~array[7]) from all level of cache.
  b. Make Victim do secret-related accesses (by invoking victim_func())
  c. Access all secret-relevant data and measure the time each data takes.
  d. Secret would be an item which takes the shortest time, because CPU cache would make a timing difference between secret-related data.

Challenge-1:

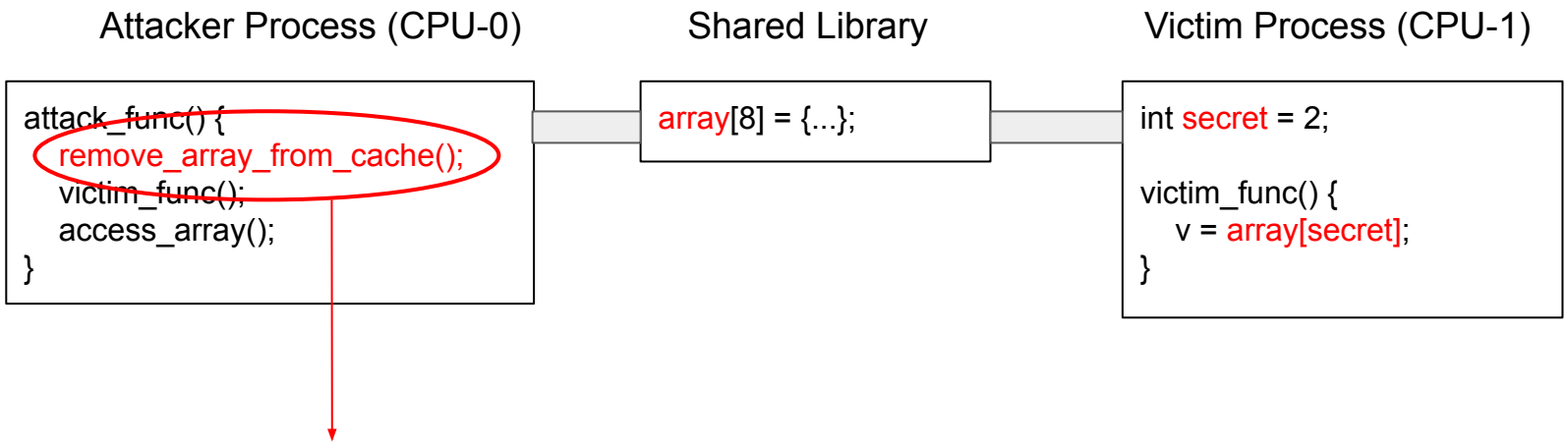
Remove secret-related data from all level of cache

Revisit

Would it be easily achieved in both Intel and ARM CPU?
Intel -> Yes, ARM -> No



CLFLUSH (Intel)



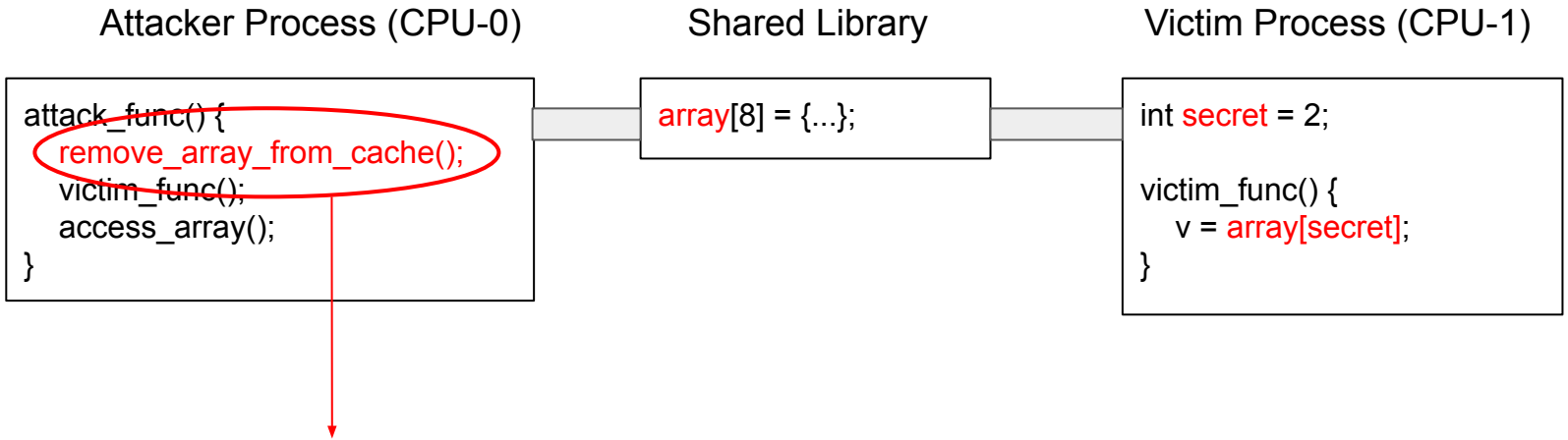

```
CLFLUSH &array[0];
CLFLUSH &array[1];
CLFLUSH &array[2];
….
….
CLFLUSH &array[7];
```

CLFLUSH (Intel) takes an address and
remove the address from all of levels of cache.

CLFLUSH (Intel) can be issued by both Kernel and User.
So user-level attackers easily can remove all cache using this!

From FLUSH+RELOAD (USENIX Security 2014)

DC IVAC (ARM)




```
DC IVAC &array[0];
DC IVAC &array[1];
DC IVAC &array[2];
….
….
DC IVAC &array[7];
```

DC IVAC (ARM) takes an address and remove the address from all of levels of cache.

DC IVAC (ARM) can be issued only by Kernel.
So user-level attackers can’t remove cache using this!

Solution-1:

EVICT+RELOAD

EVICT+RELOAD

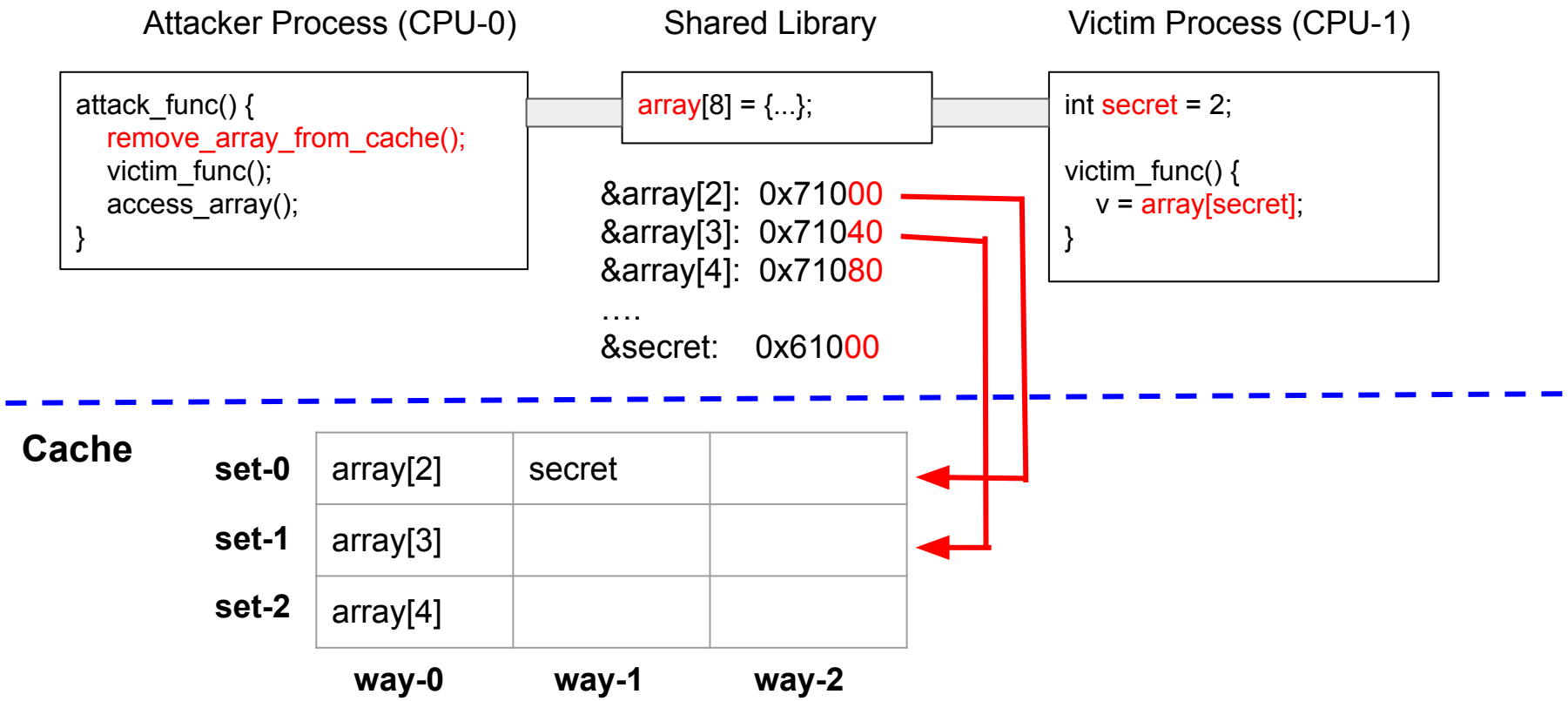
- EVICT+RELOAD is a cache attack method, which has been published as part of ARMageddon (USENIX Security 2016).

- EVICT+RELOAD makes it possible for user-level attackers to launch the aforementioned attack.

- EVICT+RELOAD is not a technique dedicated to ARM. The key concept in it can be applied to all kinds of CPU.

- You might feel like EVICT+RELOAD is very similar to Spraying Techniques in modern exploits.

EVICT+RELOAD: Background (set-associative cache)



EVICT+RELOAD: Eviction

Attacker Process (CPU-0)

```
attack_func() {
    remove_array_from_cache();
    victim_func();
    access_array();
}
```

Shared Library

```
array[8] = {...};
```

```
&array[2]:  0x71000
&array[3]:  0x71040
&array[4]:  0x71080
....
&secret:    0x61000
```

Victim Process (CPU-1)

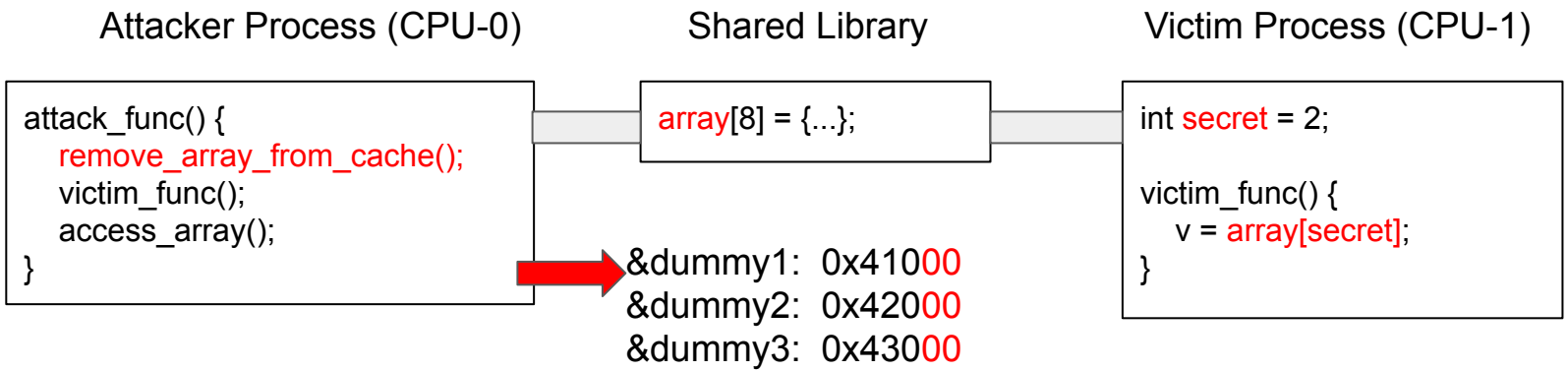
```
int secret = 2;

victim_func() {
    v = array[secret];
}
```

Cache

	way-0	way-1	way-2
set-0	**array[2]**	secret	
set-1	array[3]		
set-2	array[4]		

Spraying set-0 with dummy data!

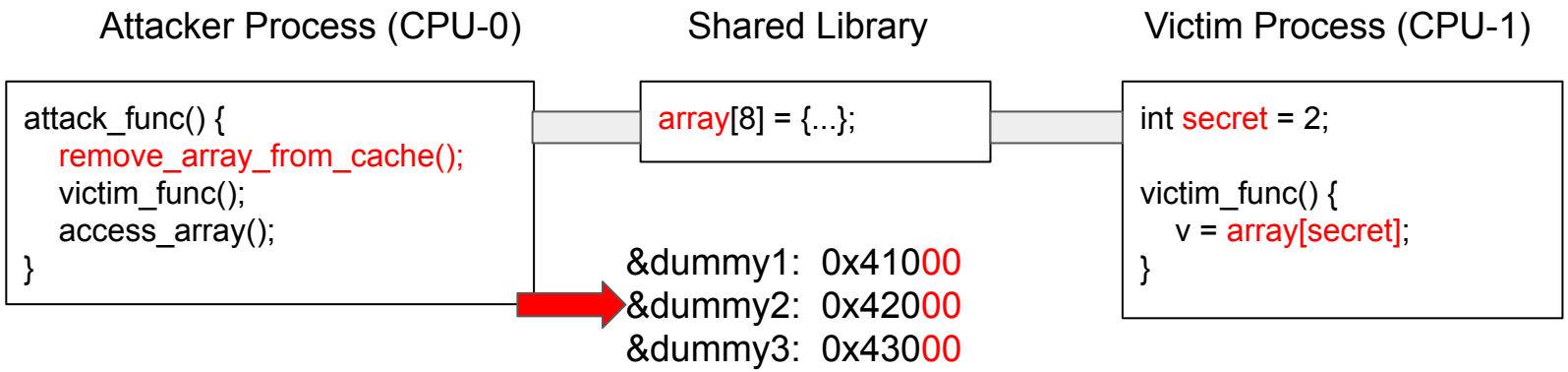
EVICT+RELOAD: Eviction (Cont)



Cache

	way-0	way-1	way-2
set-0	**array[2]**	secret	**dummy1**
set-1	array[3]		
set-2	array[4]		

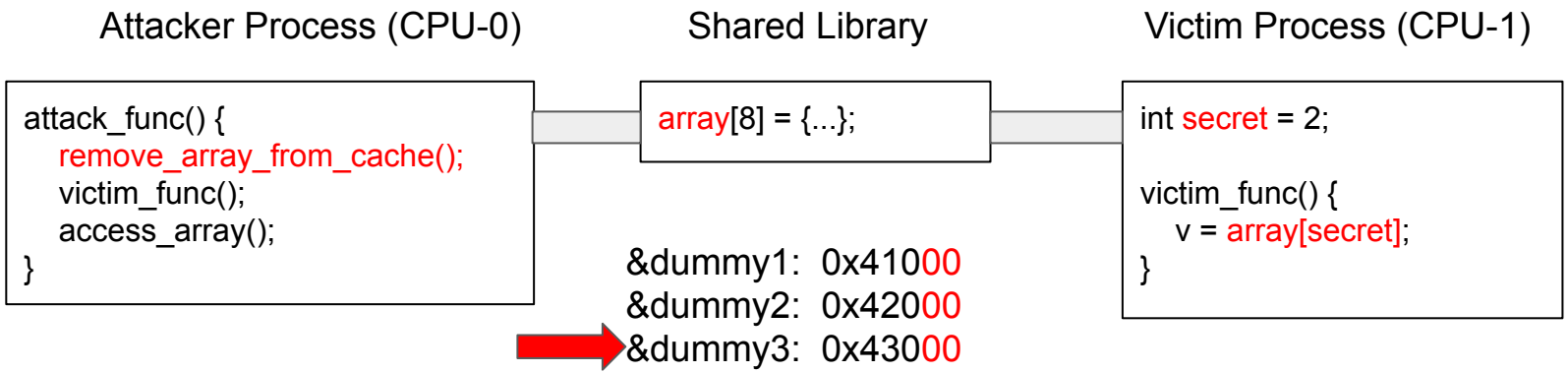
EVICT+RELOAD: Eviction (Cont)



Cache

	way-0	way-1	way-2
set-0	**array[2]**	**dummy2**	**dummy1**
set-1	array[3]		
set-2	array[4]		

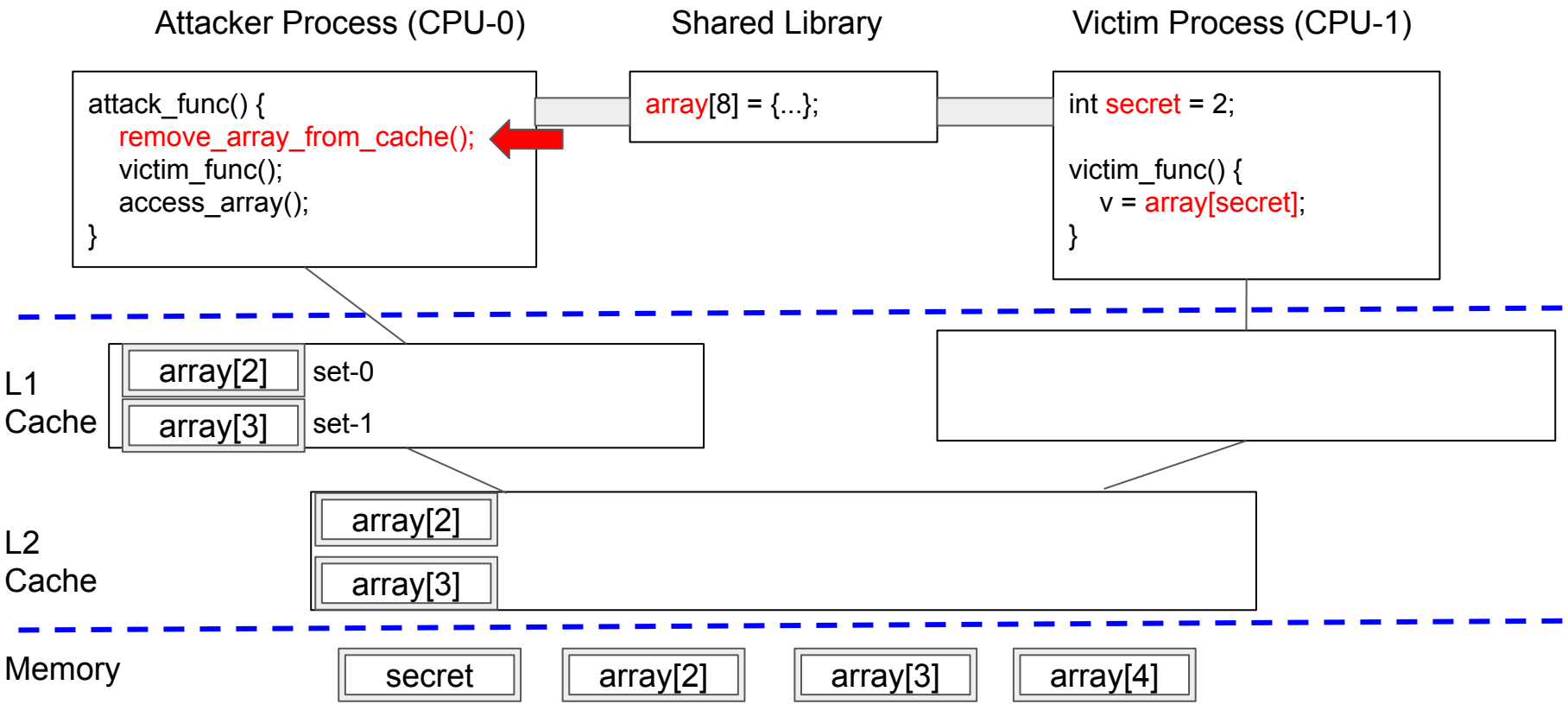
EVICT+RELOAD: Eviction (Cont)



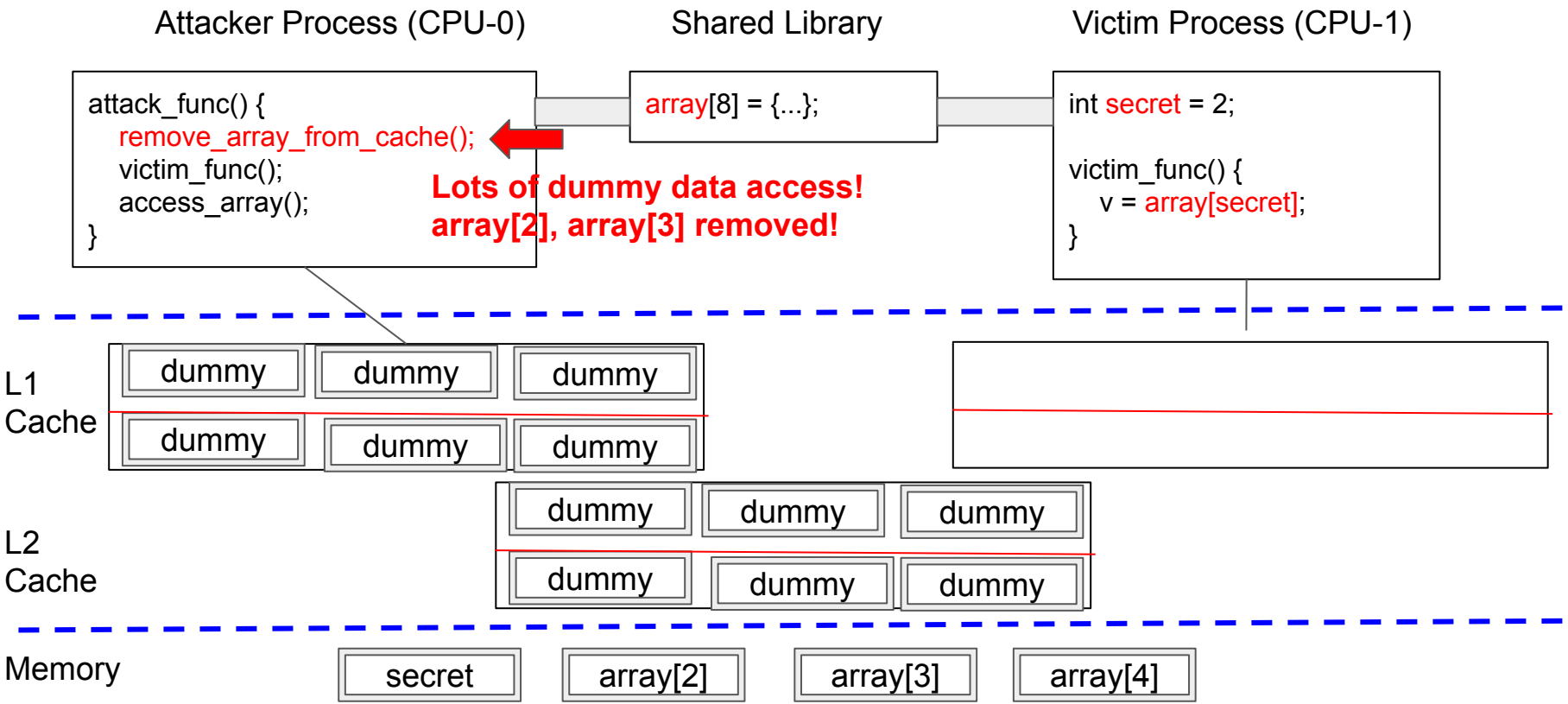
Cache

set-0	**dummy3**	**dummy2**	**dummy1**
set-1	array[3]		
set-2	array[4]		
	way-0	**way-1**	**way-2**

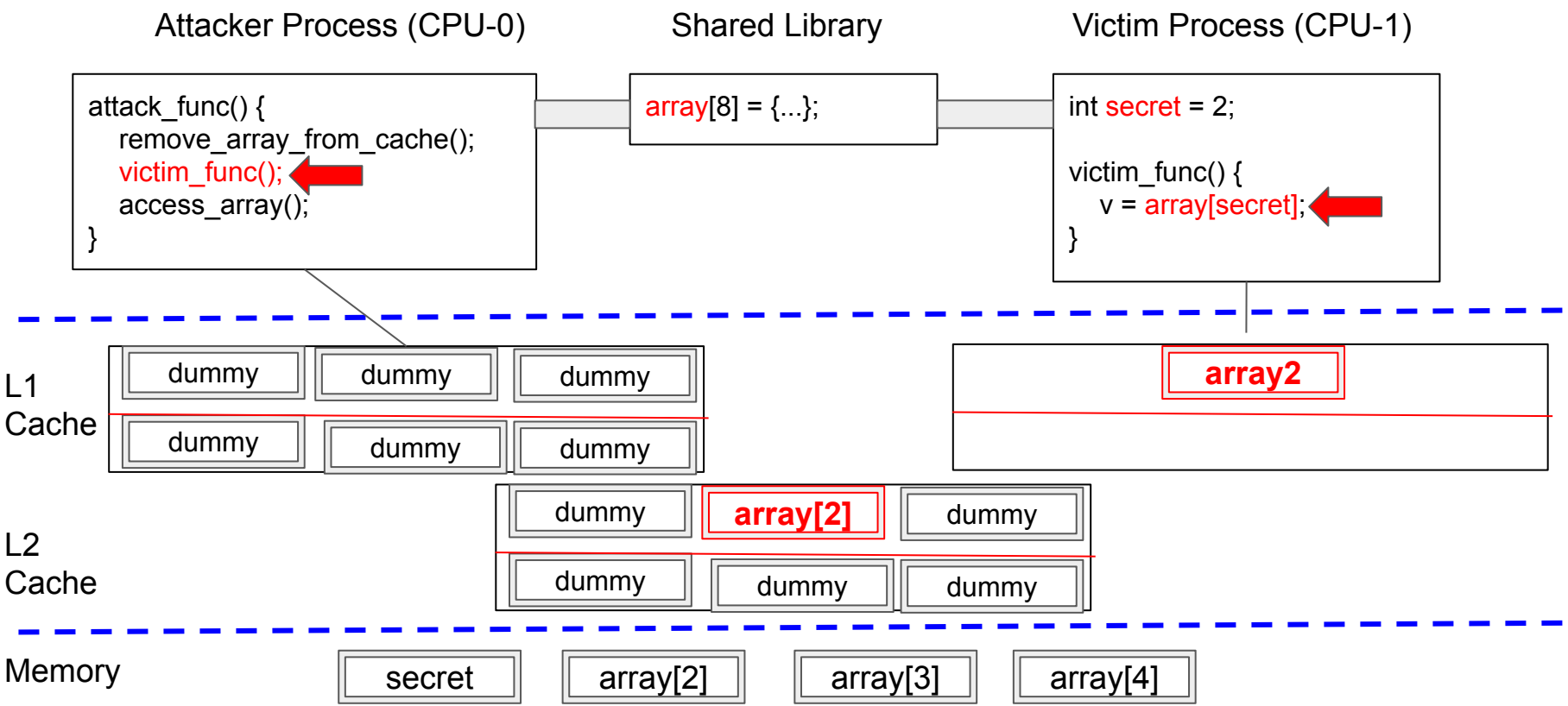
EVICT+RELOAD: Revisit Attack



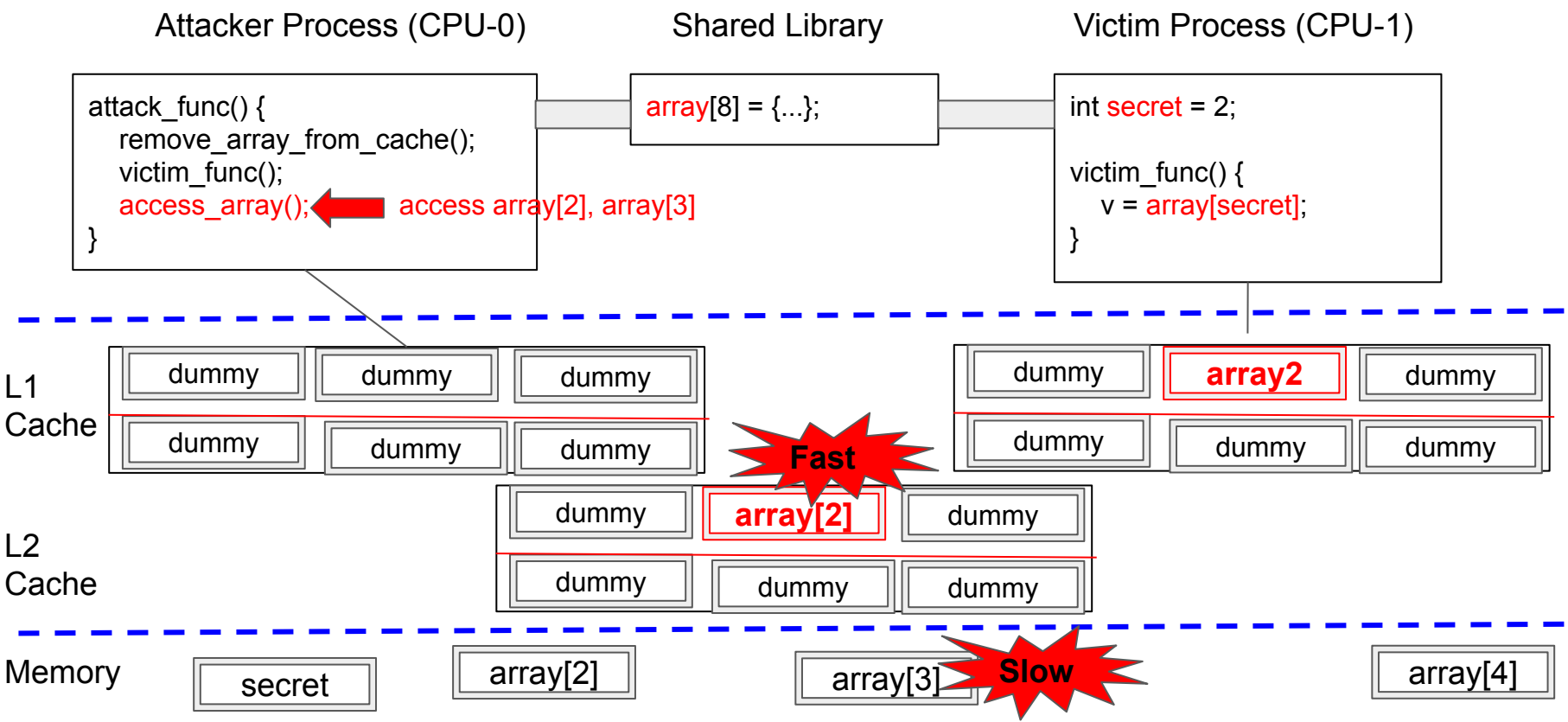
EVICT+RELOAD: Revisit Attack (Cont)



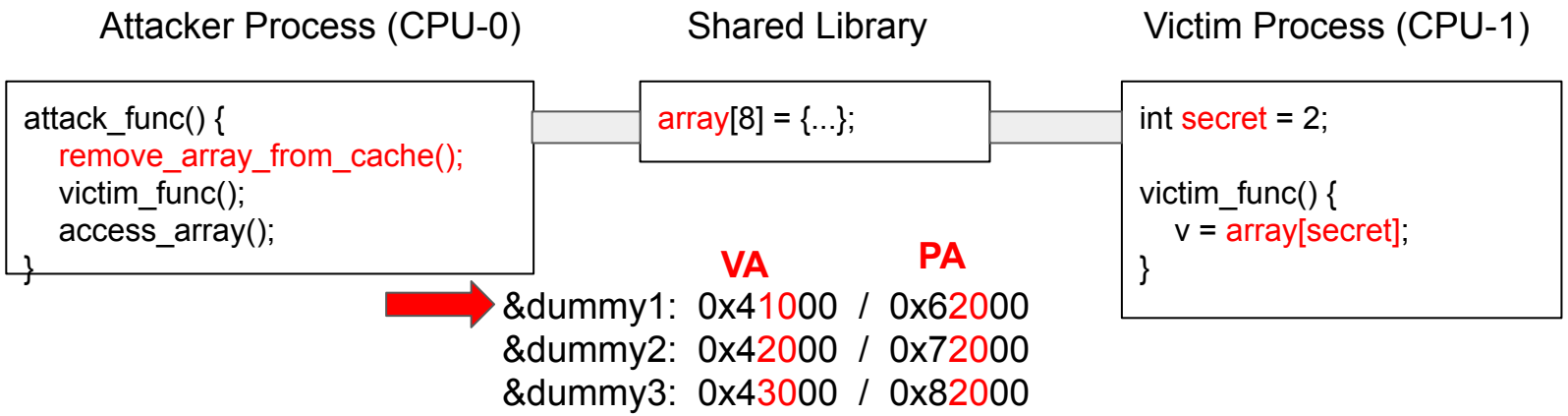
EVICT+RELOAD: Revisit Attack (Cont)



EVICT+RELOAD: Revisit Attack (Cont)



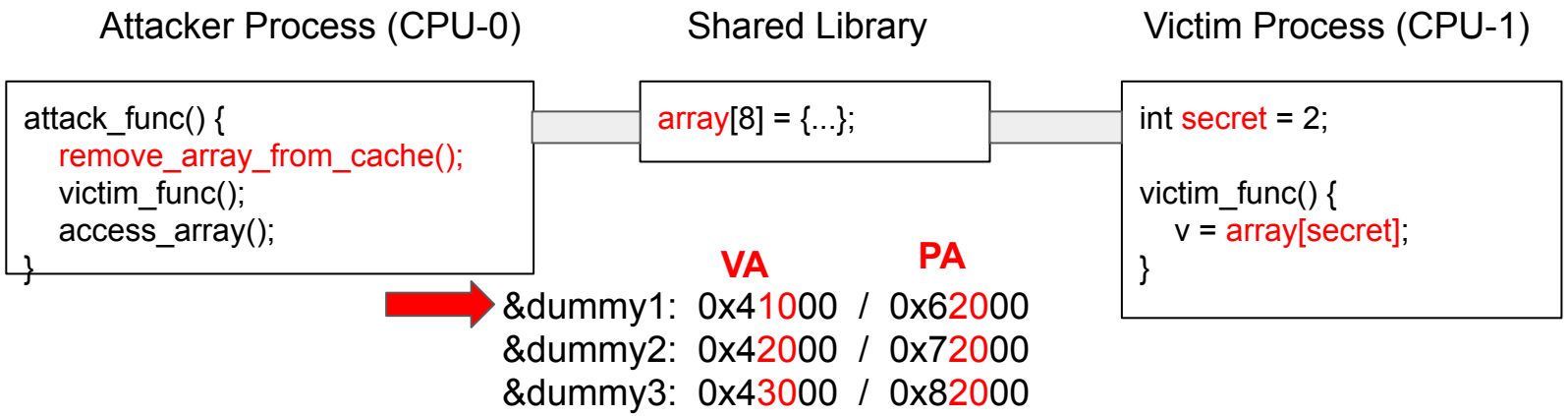
EVICT+RELOAD: Challenge about physical address



Cache

set-0	**array[2]**		
set-1	array[3]		
set-2	array[4]		
	way-0	**way-1**	**way-2**

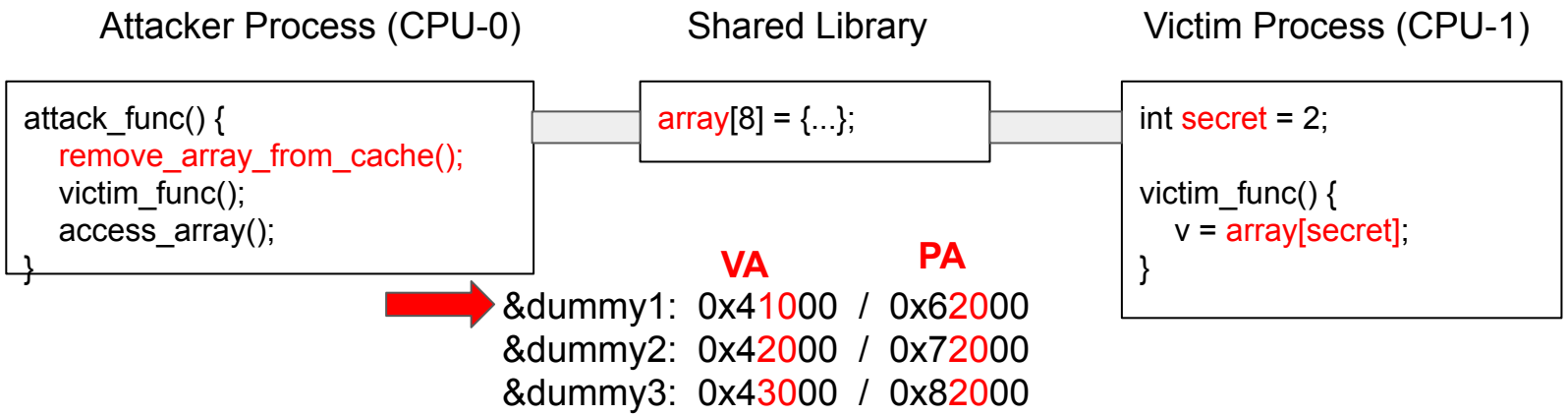
EVICT+RELOAD: Challenge about physical address



Cache

set-0	**array[2]**		
set-1	array[3]	**dummy1**	
set-2	array[4]		
	way-0	**way-1**	**way-2**

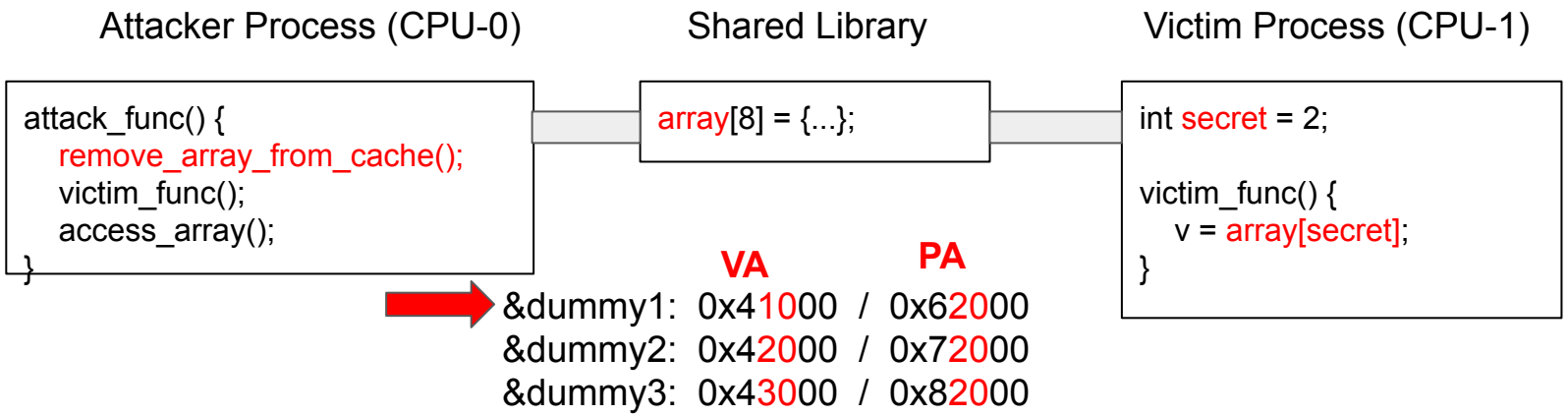
EVICT+RELOAD: Challenge about physical address



Cache

set-0	**array[2]**		
set-1	array[3]	**dummy1**	**dummy2**
set-2	array[4]		
	way-0	**way-1**	**way-2**

EVICT+RELOAD: Challenge about physical address



Cache

	way-0	way-1	way-2
set-0	**array[2]**		
set-1	**dummy3**	**dummy1**	**dummy2**
set-2	array[4]		

EVICT+RELOAD: Challenge about physical address

Attacker Process (CPU-0)

```
attack_func() {
    remove_array_from_cache();
    victim_func();
    access_array();
}
```

Shared Library

```
array[8] = {...};
```

Victim Process (CPU-1)

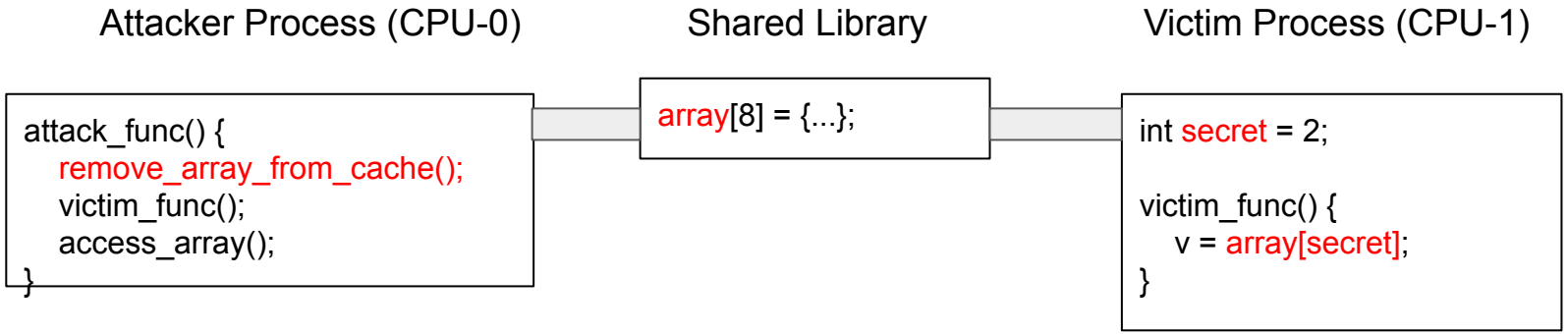
```
int secret = 2;

victim_func() {
    v = array[secret];
}
```

Huge amount of random dummy data!!!

Cache

	way-0	way-1	way-2
set-0	**dummy111**	**dummy712**	**dummy1011**
set-1	**dummy3**	**dummy1**	**dummy2**
set-2	array[4]		

EVICT+RELOAD: Challenge about physical address



	VA	PA
&dummy1:	0x41000	0x62000
&dummy2:	0x42000	0x72000
&dummy3:	0x43000	0x80000

Possible by timing difference!!

From [Theory and Practice of Finding Eviction Sets]

Can we check dummy1 and dummy2 place on same cache set?

Likewise, Can we check dummy2 and dummy3 place on different cache set?

EVICT+RELOAD: Note

- Ideally, EVICT+RELOAD works well for ARM CPU.

- But in reality, EVICT+RELOAD is much harder attack than you think.

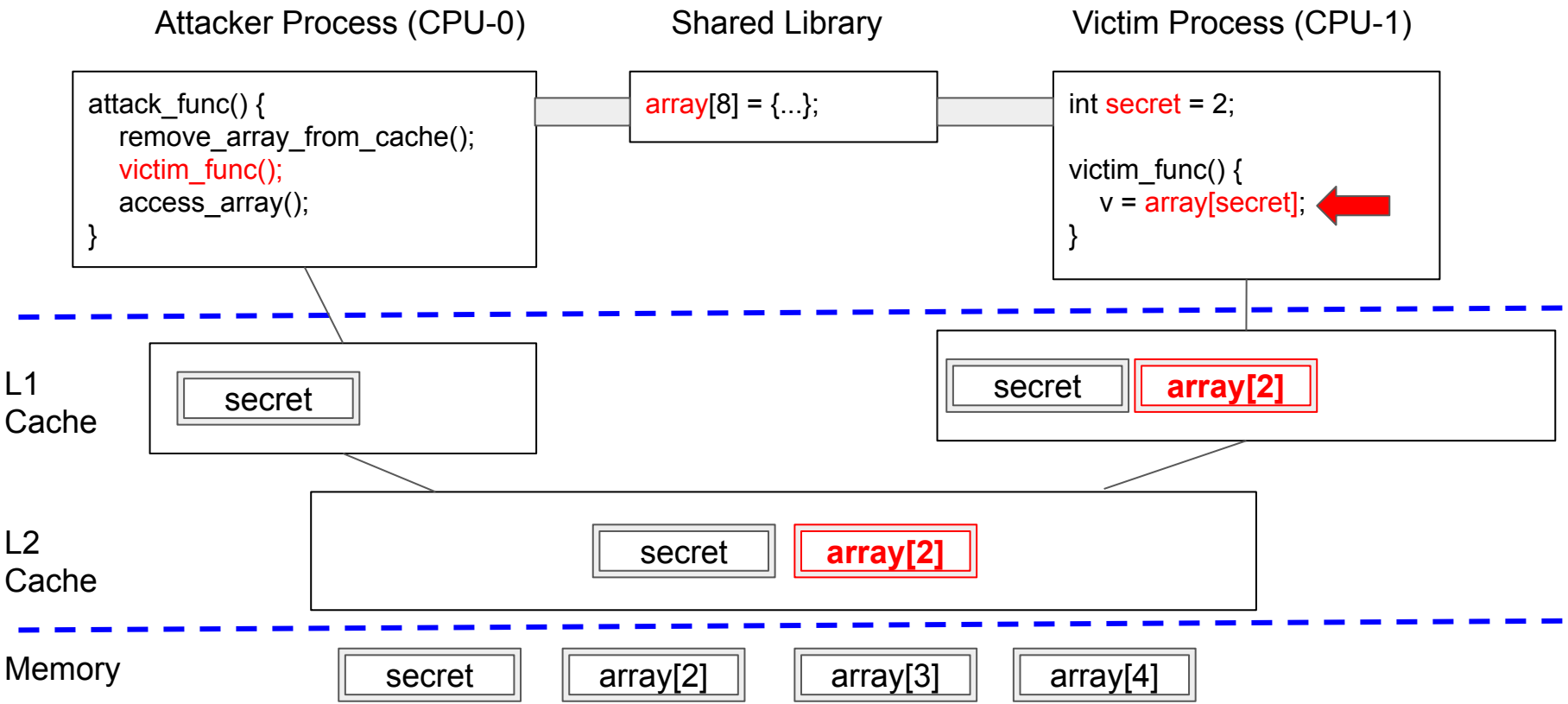
- To understand it deeper, recommended to see
  - ARMageddon (USENIX Security 2016)
  - Theory and Practice of Finding Eviction Sets (IEEE S&P 2019)

Challenge-2:

Cache Inclusion

Revisit

Can array[2] live in both L1 and L2 cache?
Intel -> Yes, ARM -> No ⇒ due to Cache Inclusion



Cache Inclusion

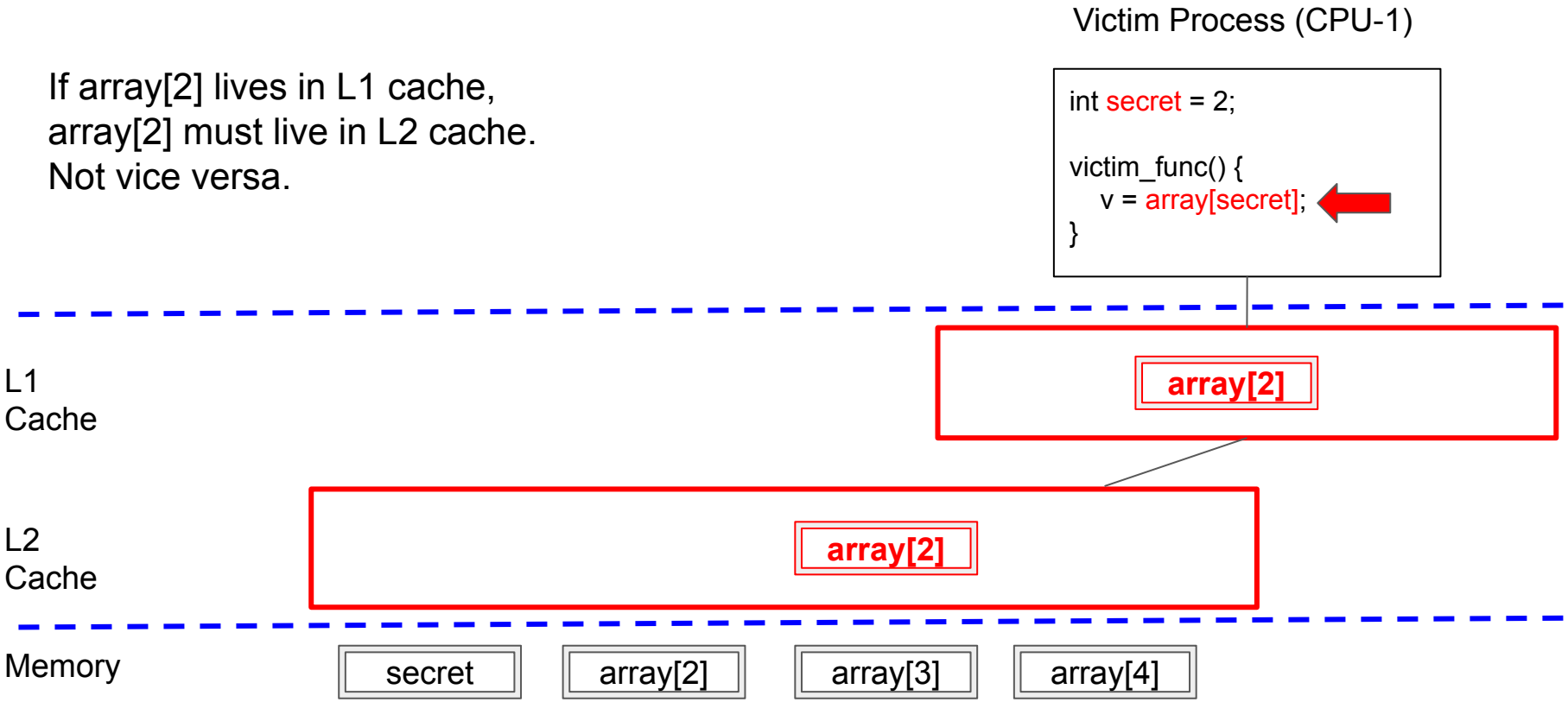
- Cache Inclusion means about how a data laid on various levels of cache.

- 3 Cache Inclusion types
    - Inclusive Cache ⇒ Intel (L1 and L2 cache)
        - Exceptionally, L3 cache of Intel typically employs Exclusive Cache
    - Exclusive Cache ⇒ ARM, AMD
    - Non-Inclusive Cache ⇒ ARM

- In ARM, policy for cache inclusiveness depends on micro-architecture. (i.e., Cortex-AXX)

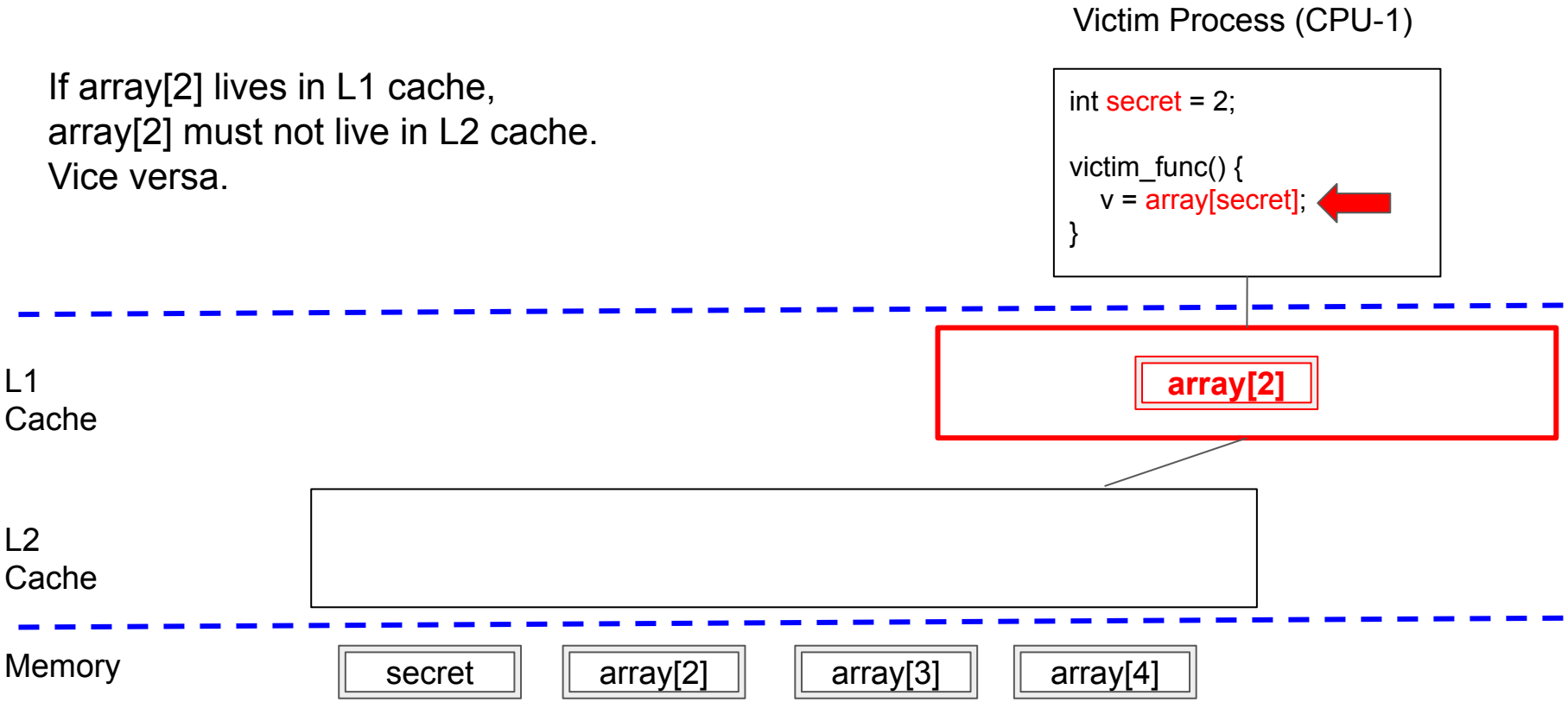
Inclusive Cache

If array[2] lives in L1 cache,
array[2] must live in L2 cache.
Not vice versa.



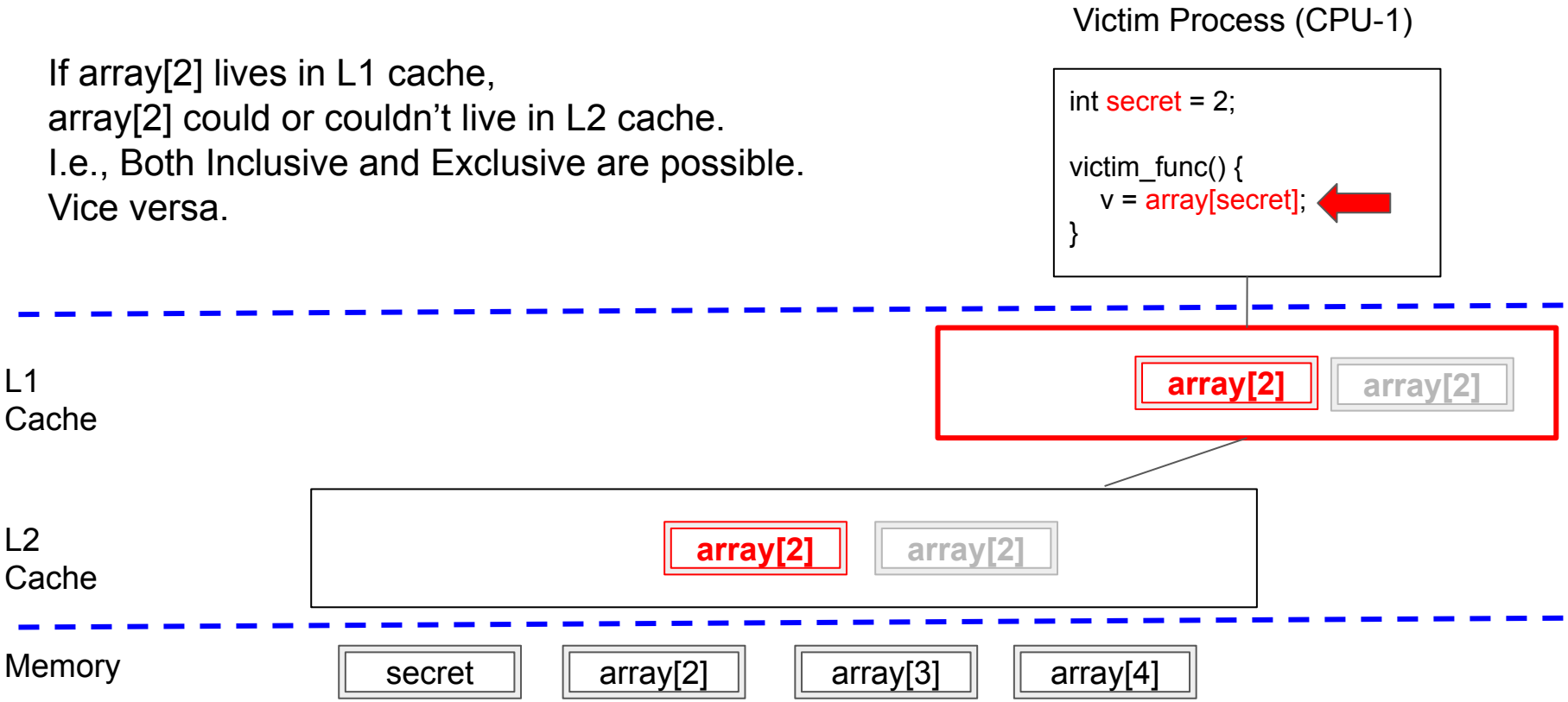
Exclusive Cache

If array[2] lives in L1 cache,
array[2] must not live in L2 cache.
Vice versa.

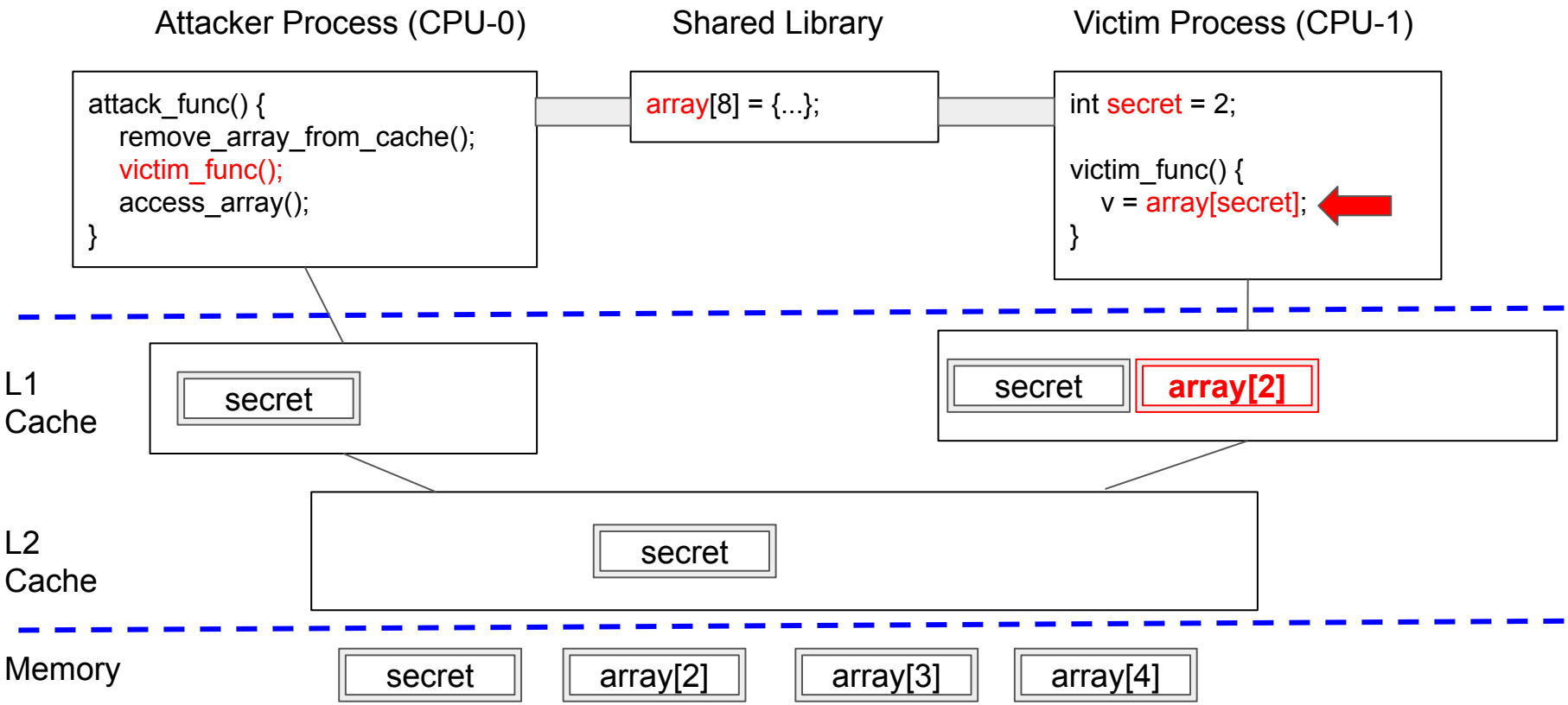


Non-Inclusive Cache

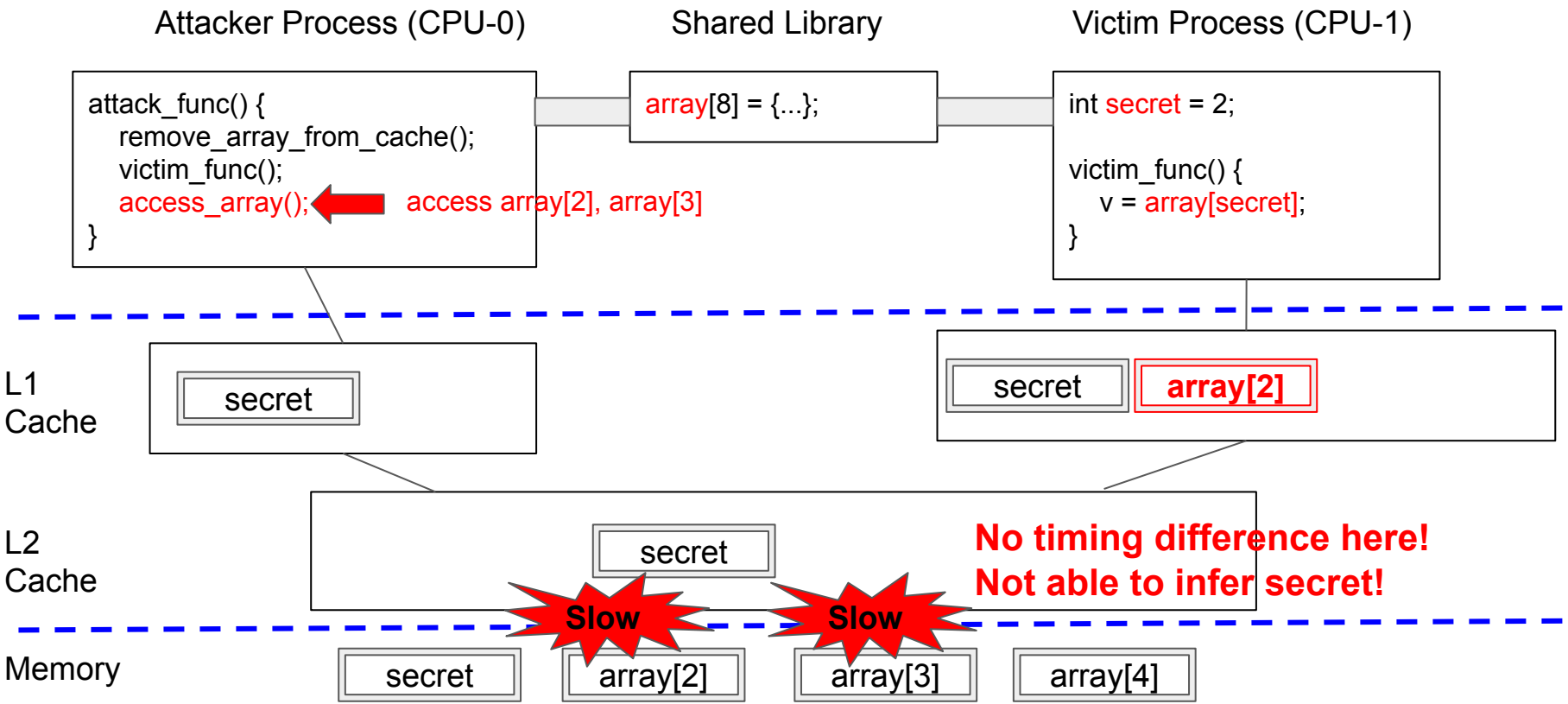
If array[2] lives in L1 cache,
array[2] could or couldn't live in L2 cache.
I.e., Both Inclusive and Exclusive are possible.
Vice versa.



Problem in Exclusive/Non-Inclusive Cache



Problem in Exclusive/Non-Inclusive Cache (Cont)



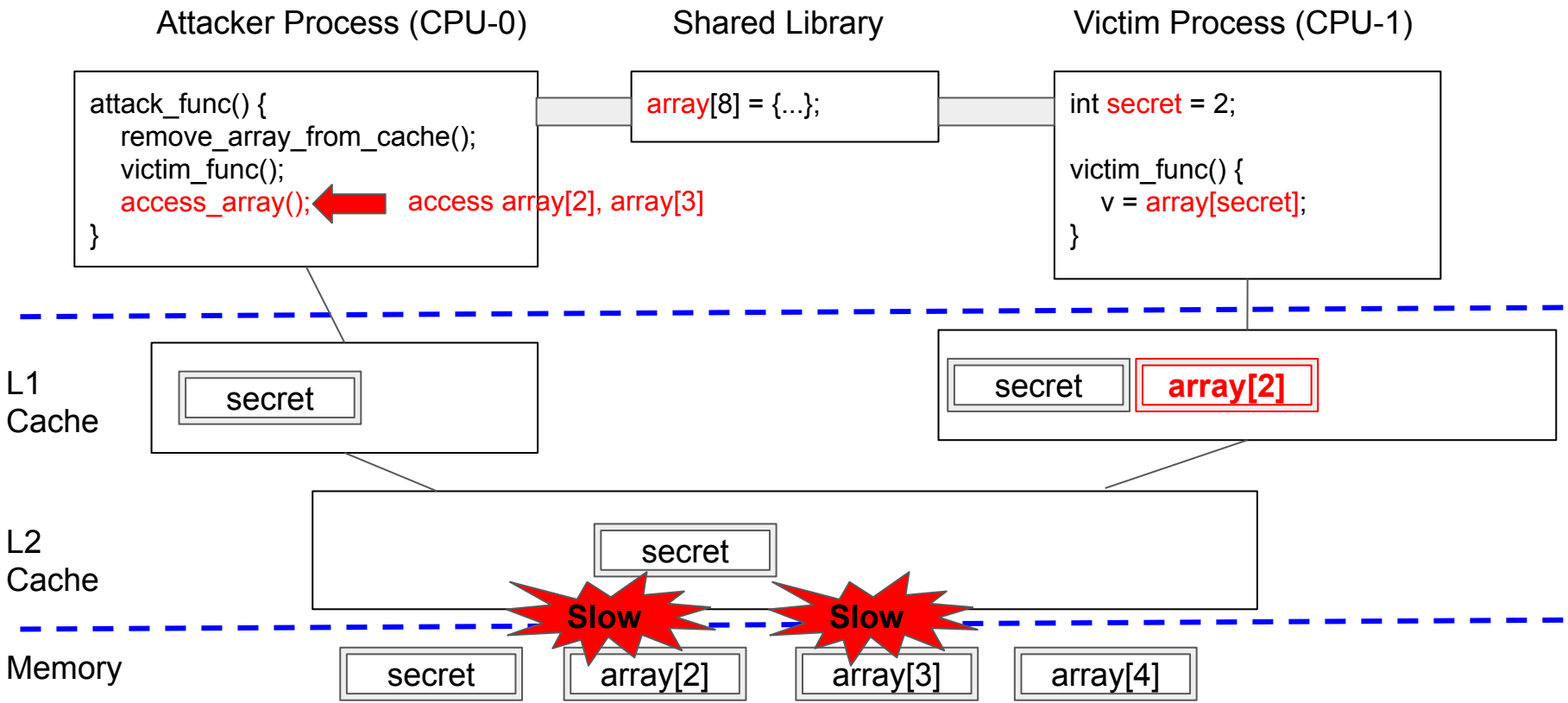
Solution-2:

2-1: Cache Coherency Protocol

2-2: Simultaneous Multithreading (SMT)

Revisit

Would array[2] loaded from Memory in reality?
NO! Due to Cache Coherency Protocol!



Cache Coherency Protocol

- Cache coherency protocol works for synchronization between all levels of cache as well as memory.

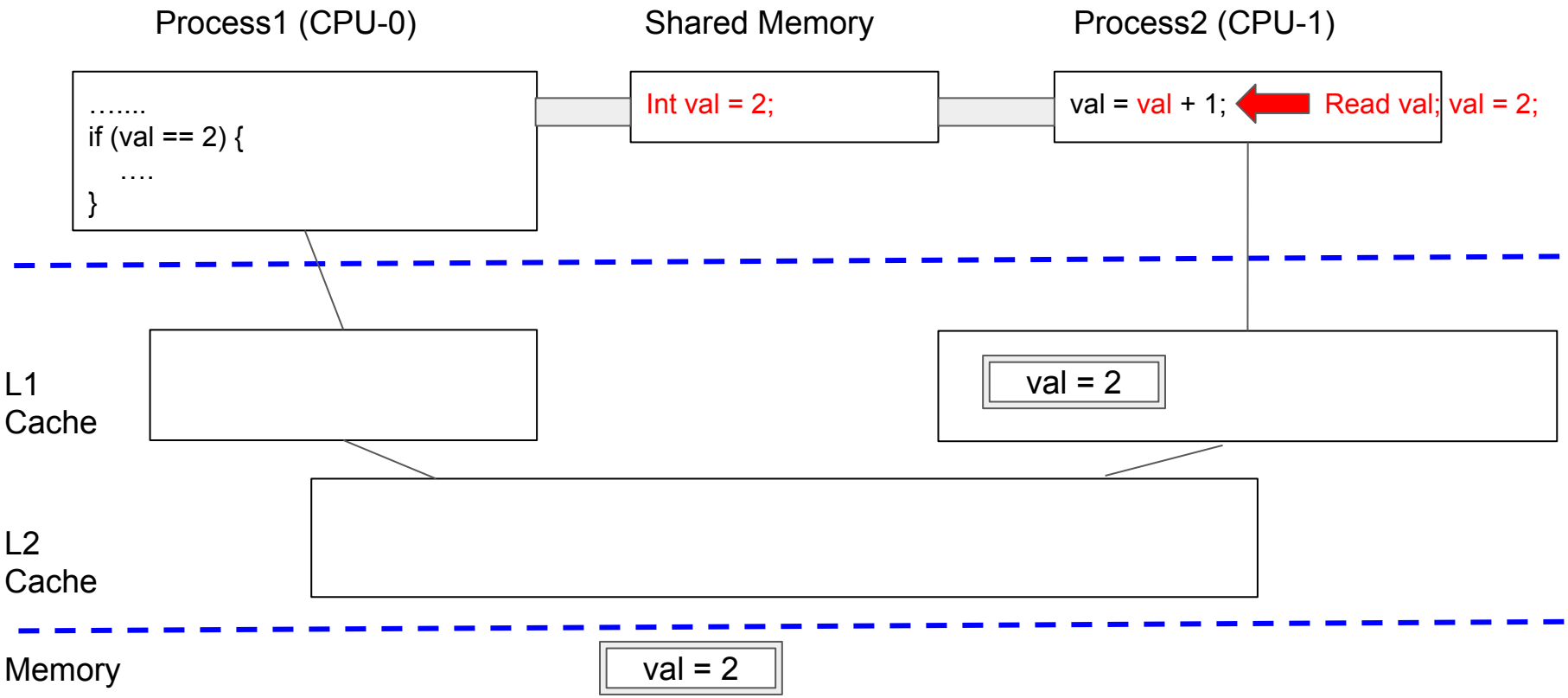
- Two different kinds of cache coherency protocol

    - Snooping-based (MESI, MOESI, MESFI, …)

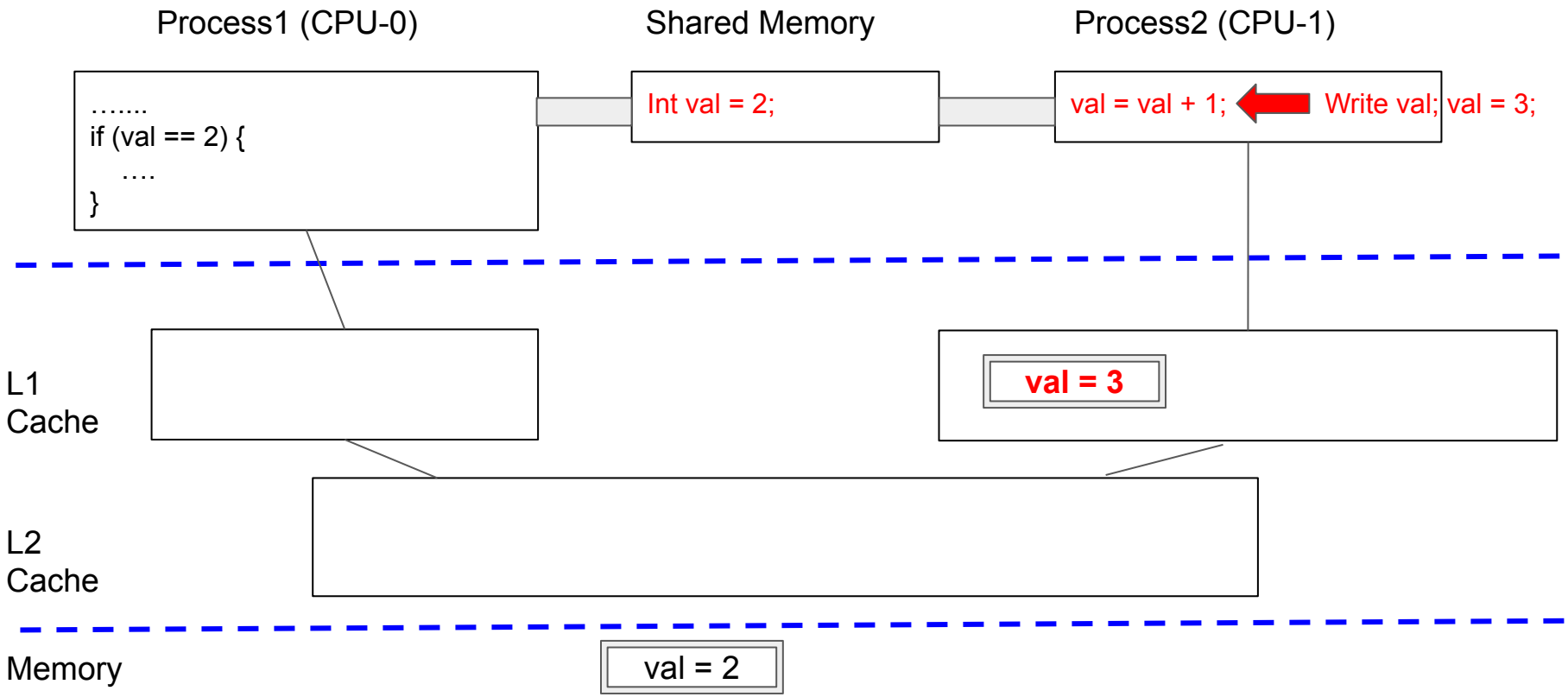
    - Directory-based

- The principles of them is exactly same. Just the way of implementation is different.

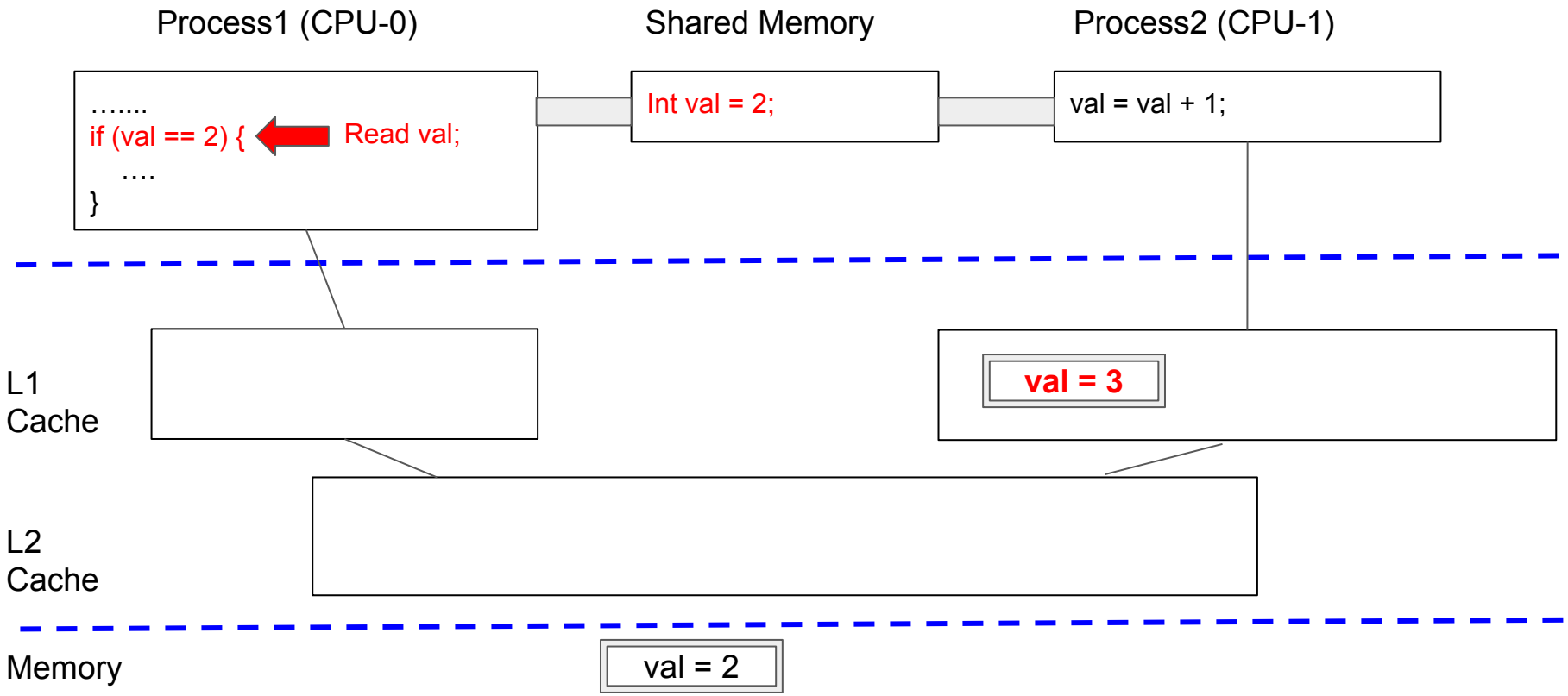
Problem without Cache Coherence



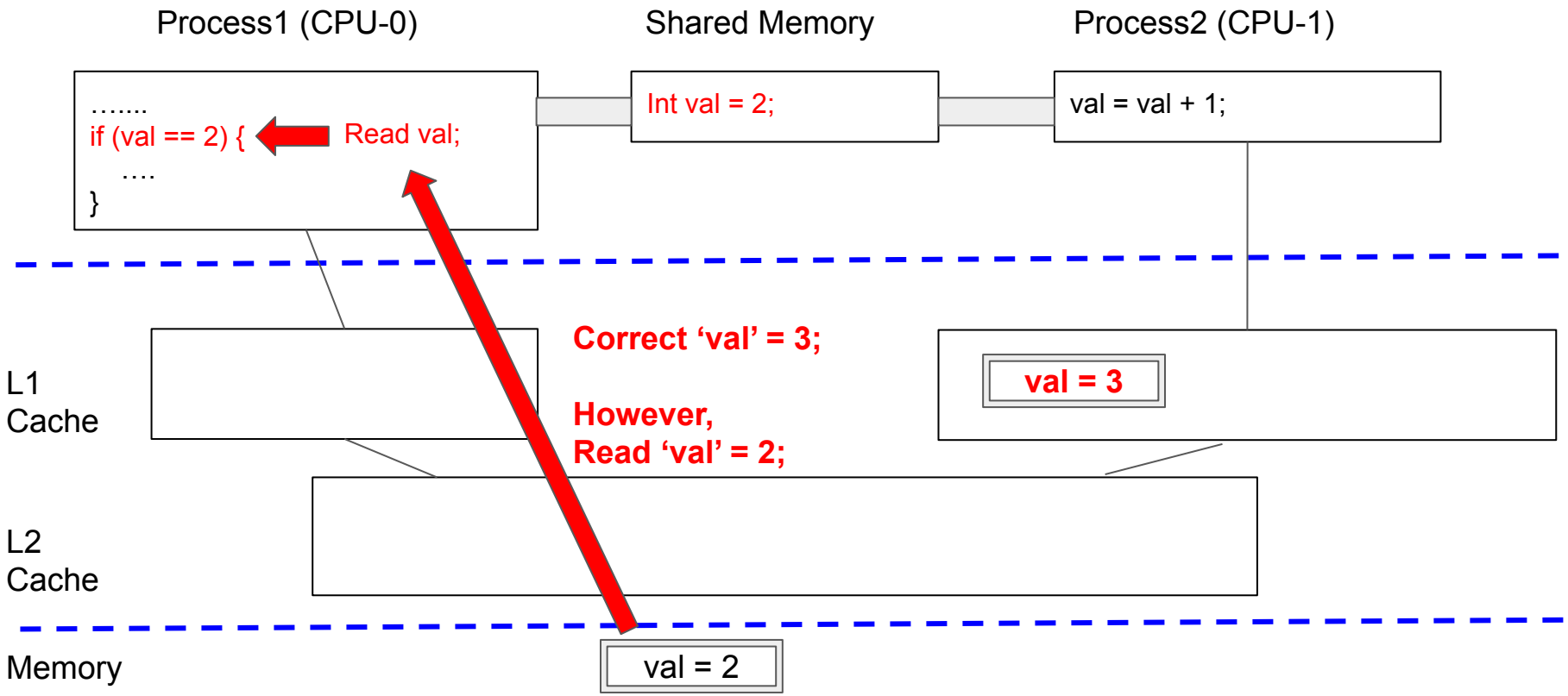
Problem without Cache Coherence (Cont)



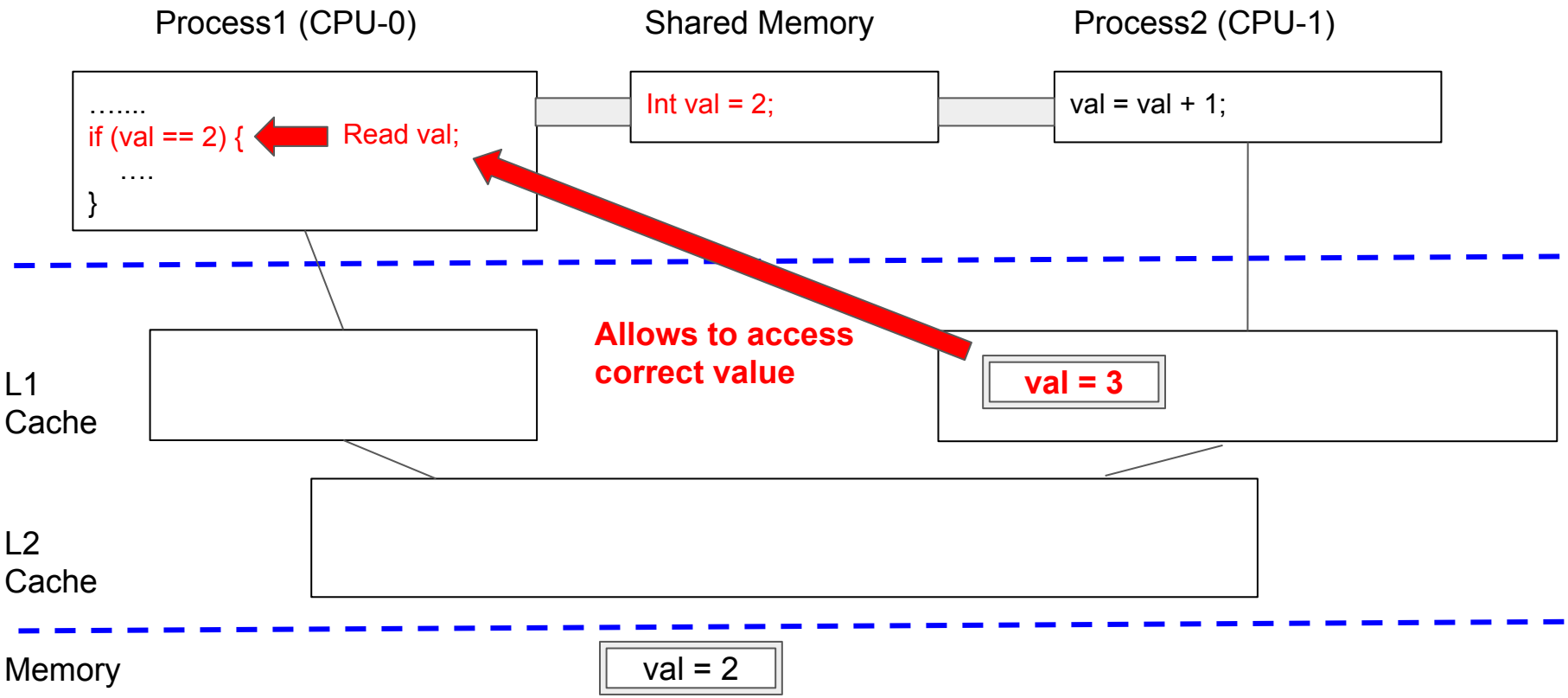
Problem without Cache Coherence (Cont)



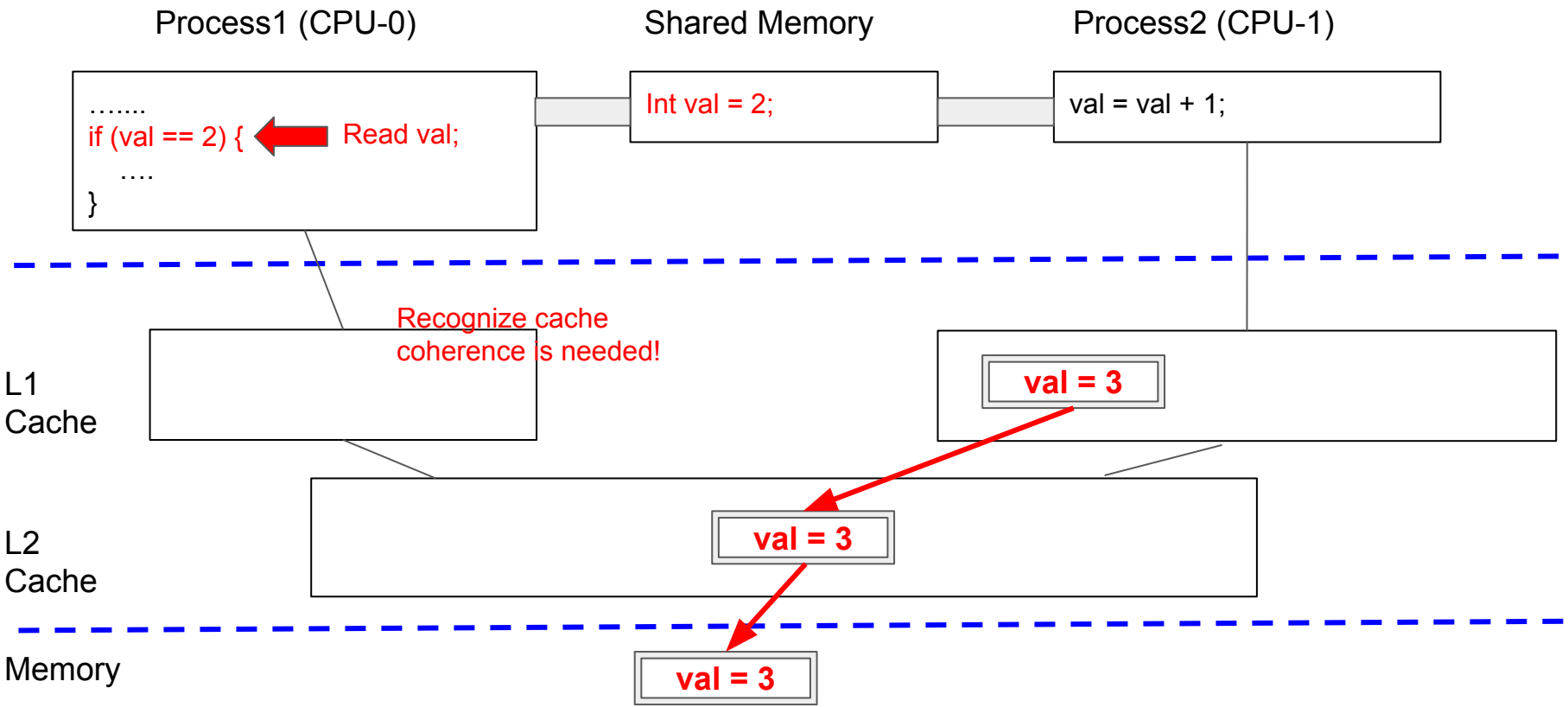
Problem without Cache Coherence (Cont)



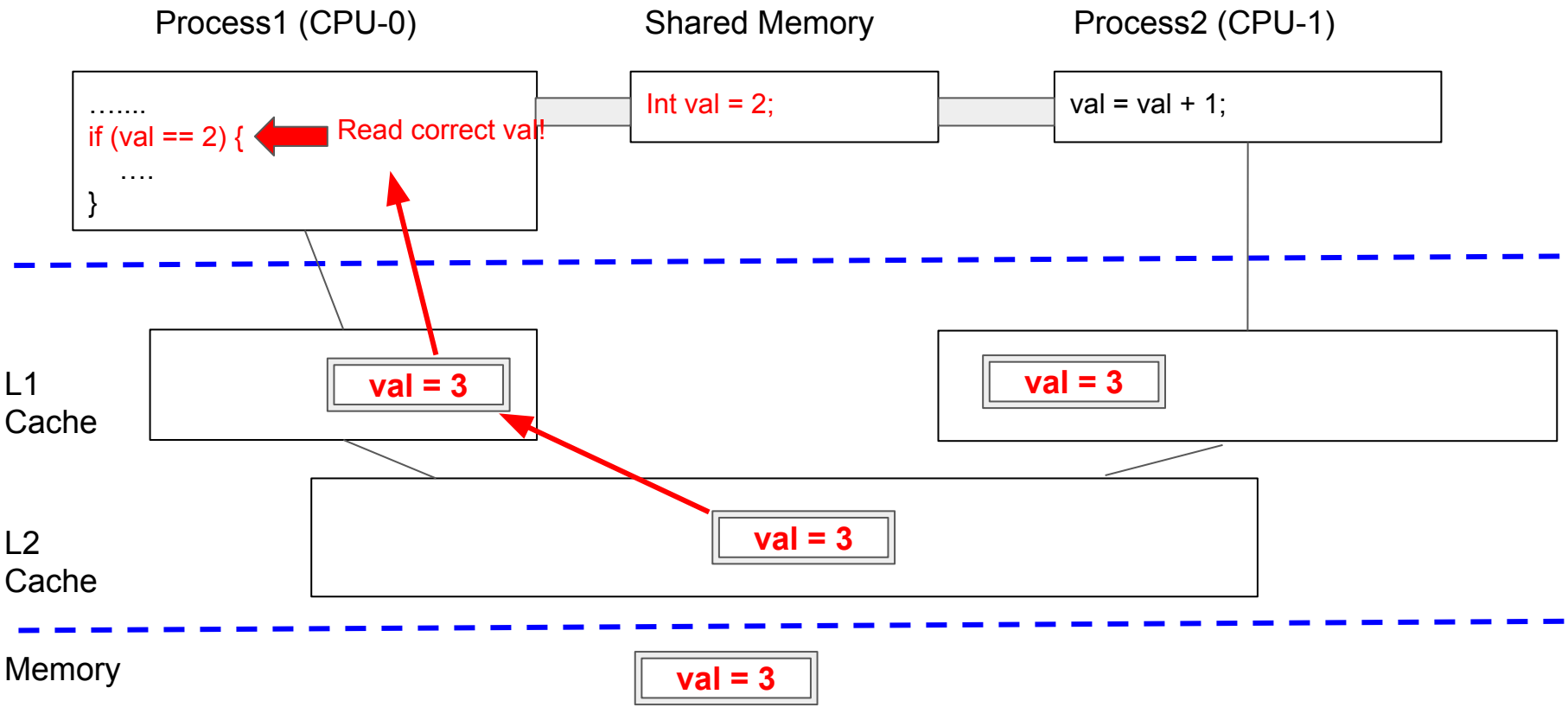
What Cache Coherence does



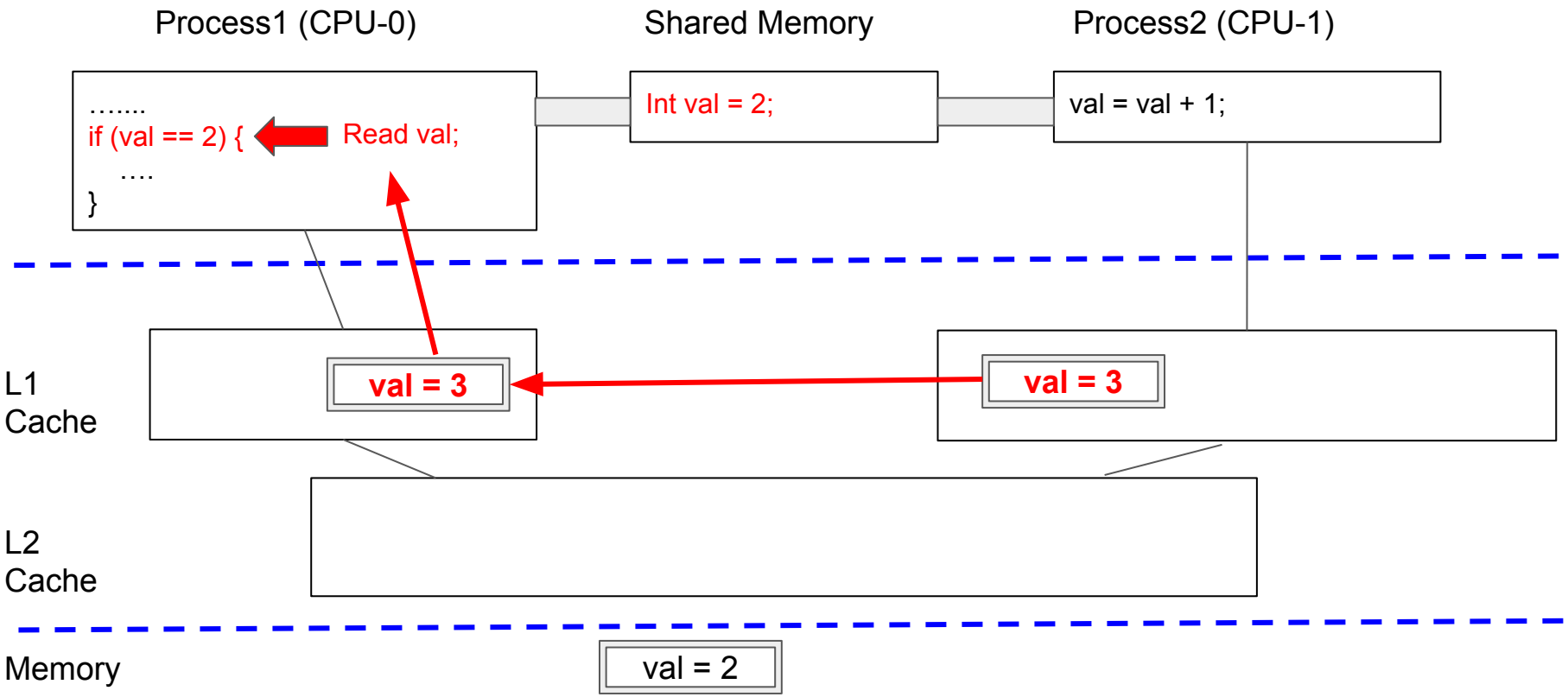
What Cache Coherence does (Option-1)



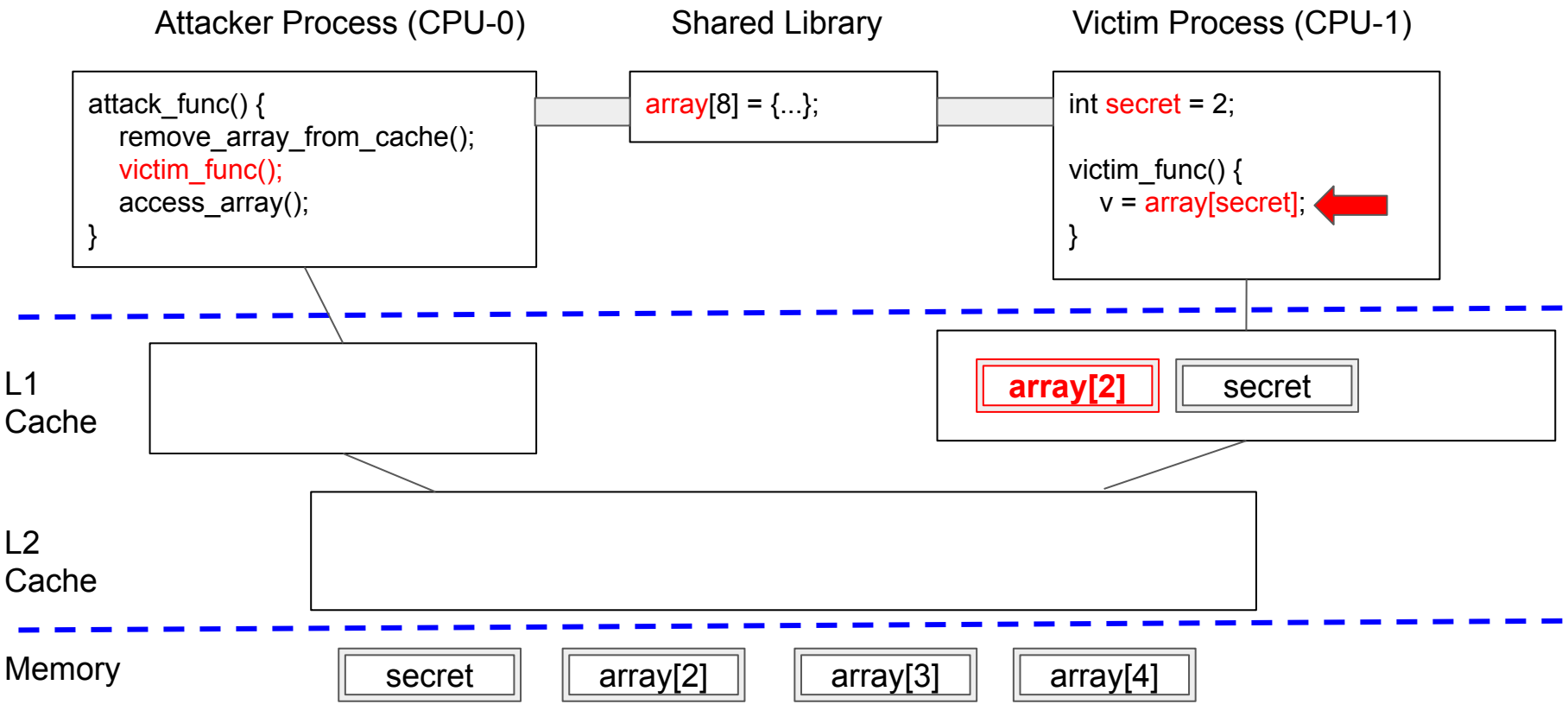
What Cache Coherence does (Option-1) (Cont)



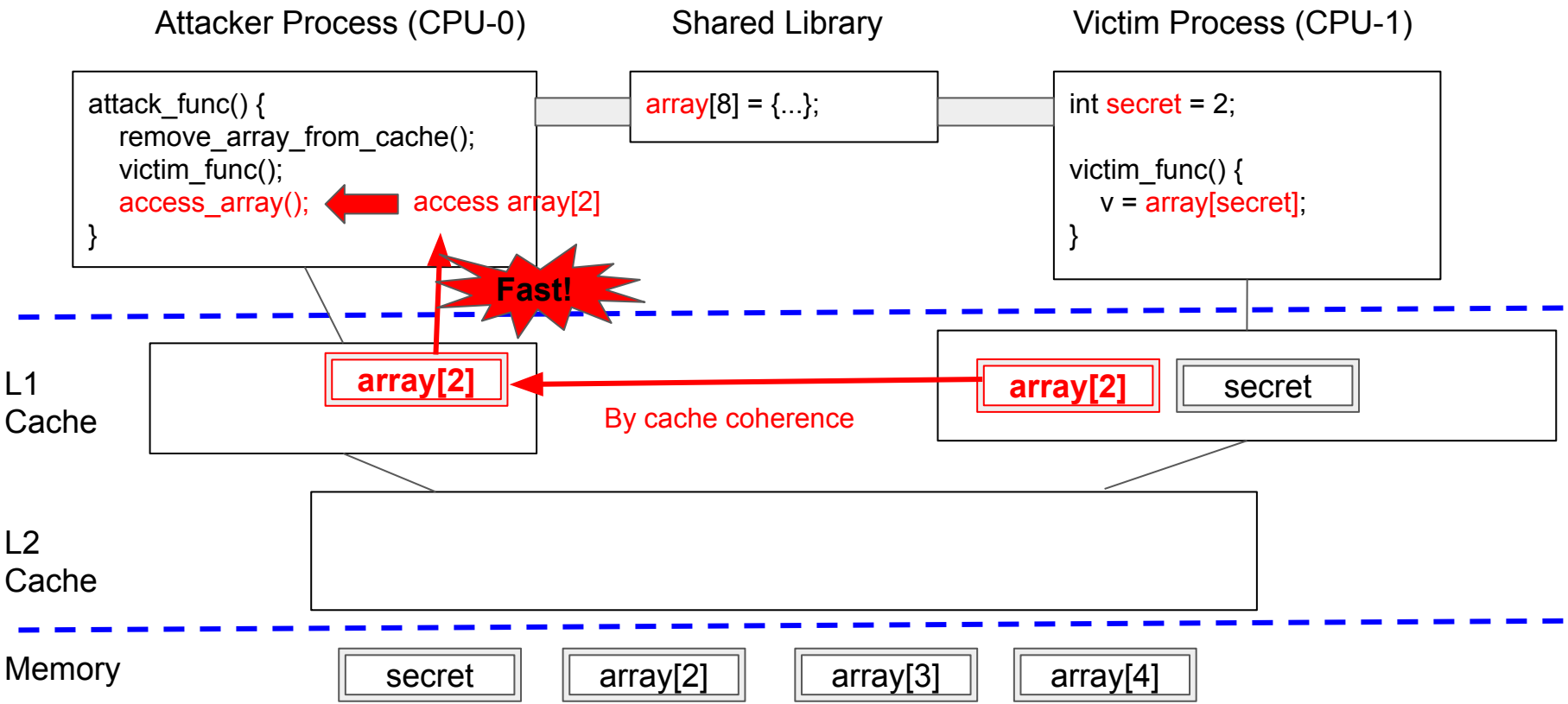
What Cache Coherence does (Option-2)



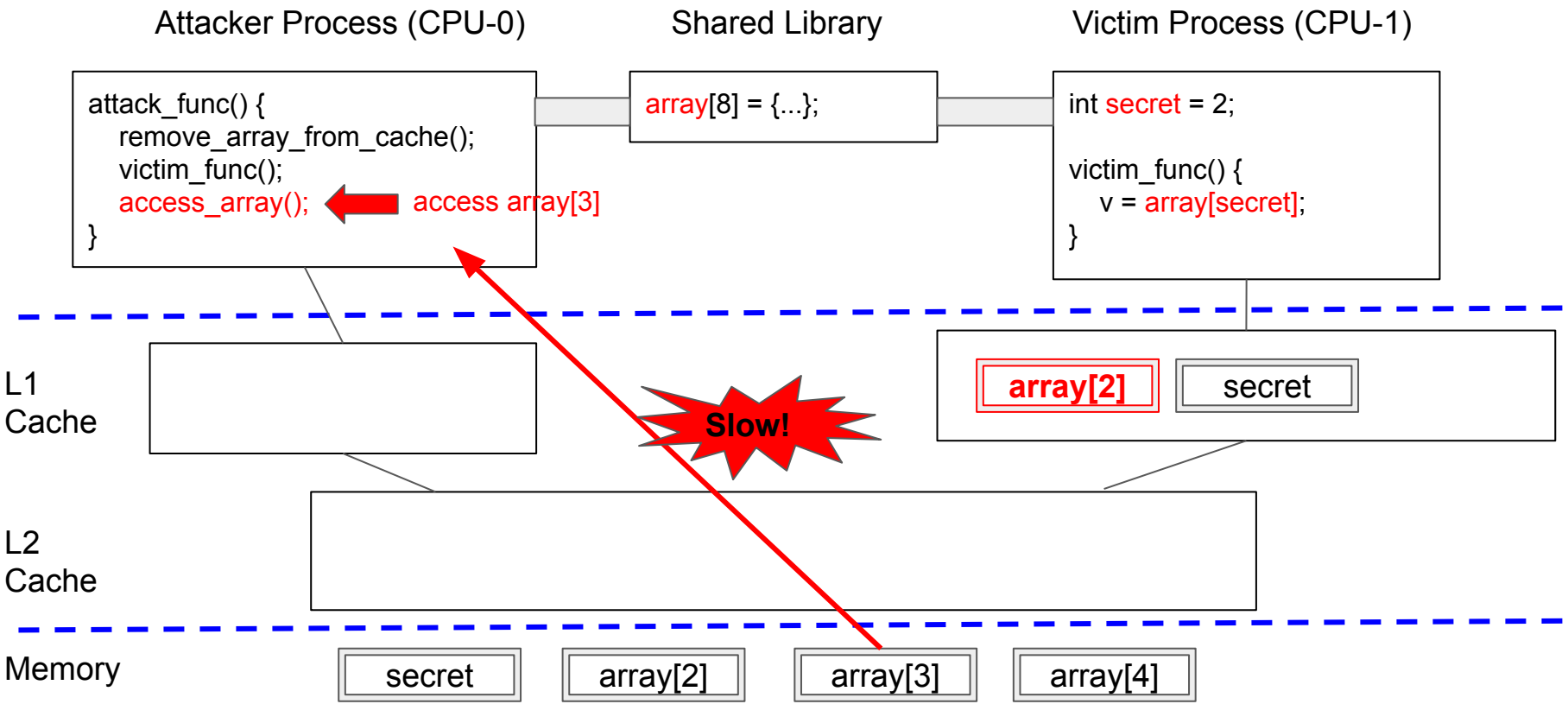
Cache Coherence: Revisit Attack



Cache Coherence: Revisit Attack (Cont)



Cache Coherence: Revisit Attack (Cont)



Note

- Cross Processor Cache Attacks (ASIACCS 2016) demonstrated this attack scenario on AMD.
- SmokeBomb (MobiSys 2020) demonstrated this attack scenario on ARM.
- Even worse in Intel CPU, Snoop attack recently demonstrated leaking L1 cache data by exploiting snooping-based cache coherency protocol like what Meltdown/Foreshadow did.

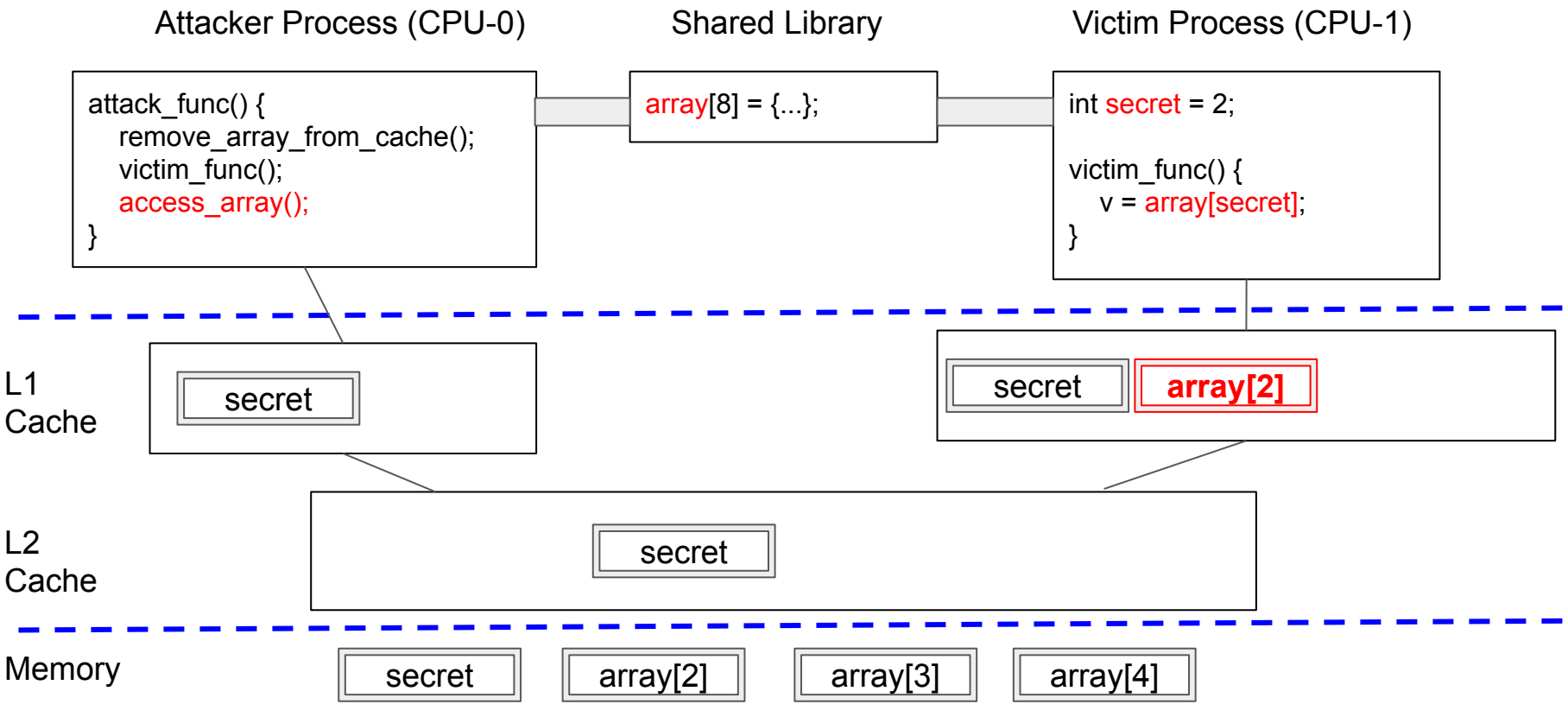
Solution-2:

2-1: Cache Coherency Protocol

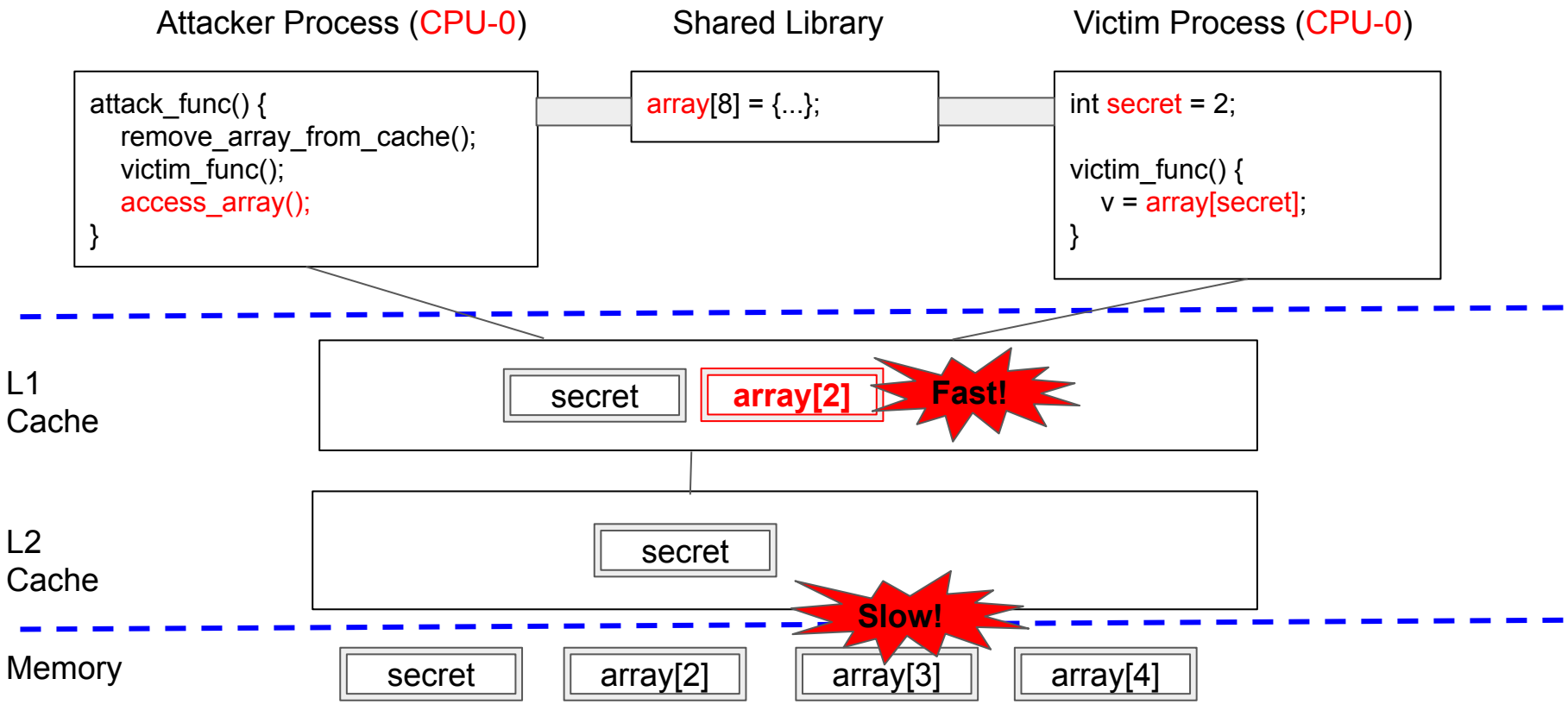
2-2: Simultaneous Multithreading (SMT)

Revisit

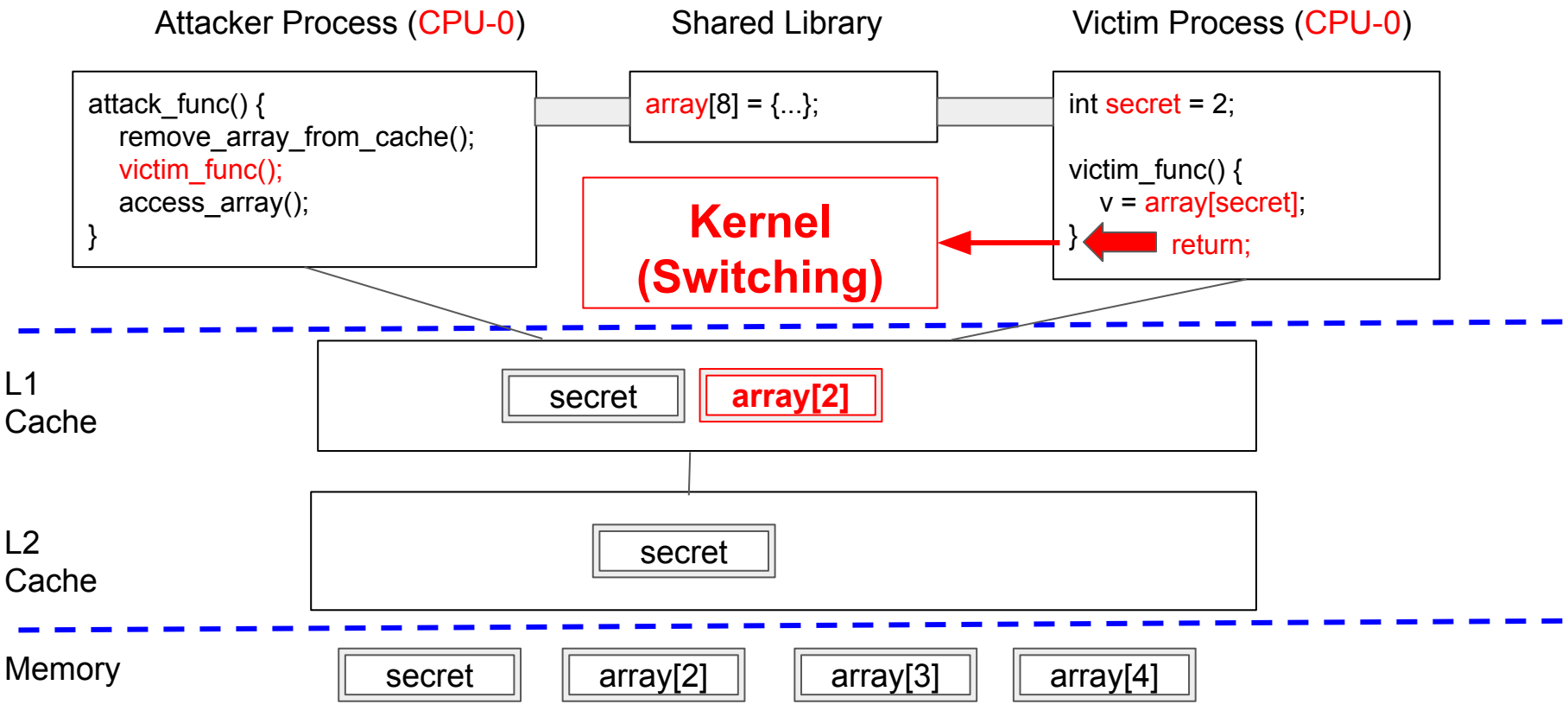
Remove the assumption
that two processes are running on different core.



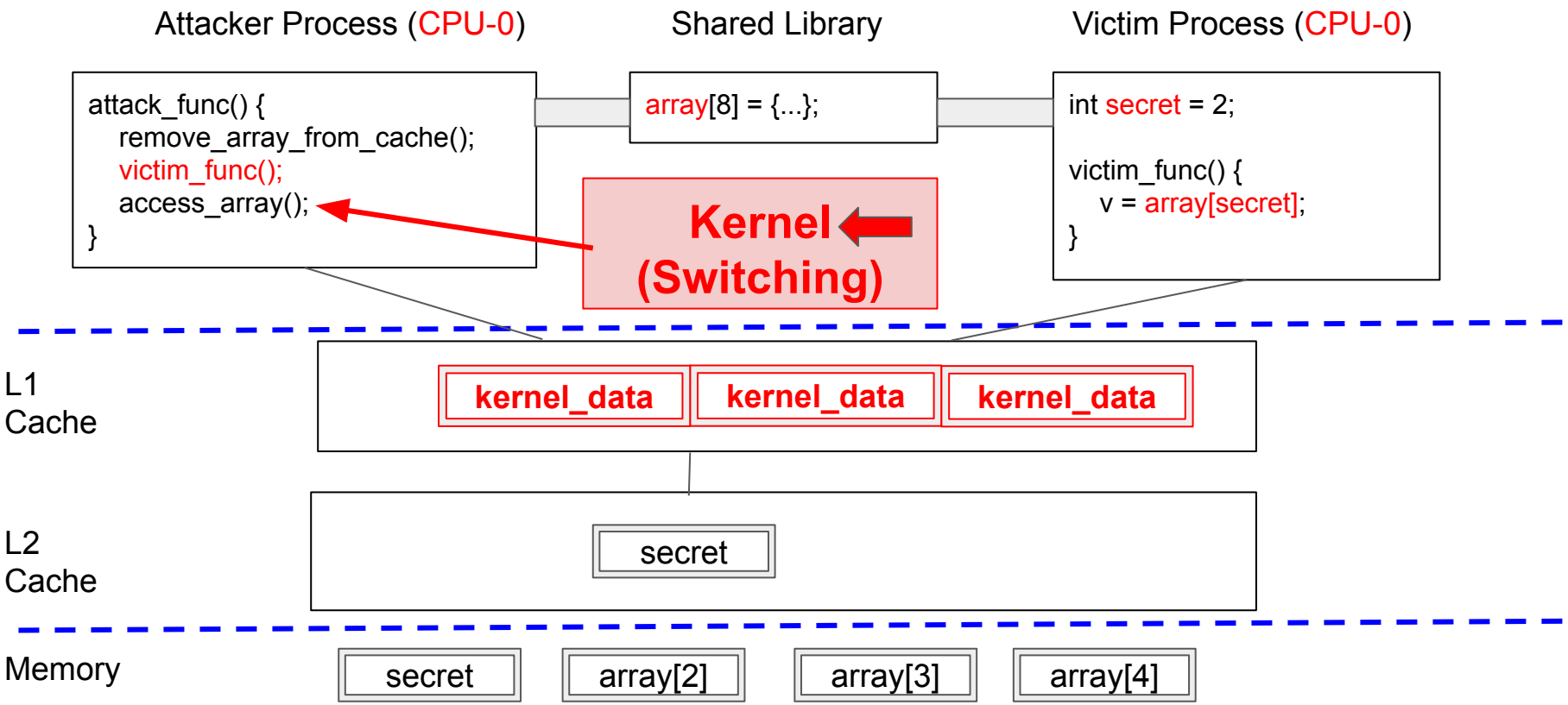
Same Core Scenario (Ideal)



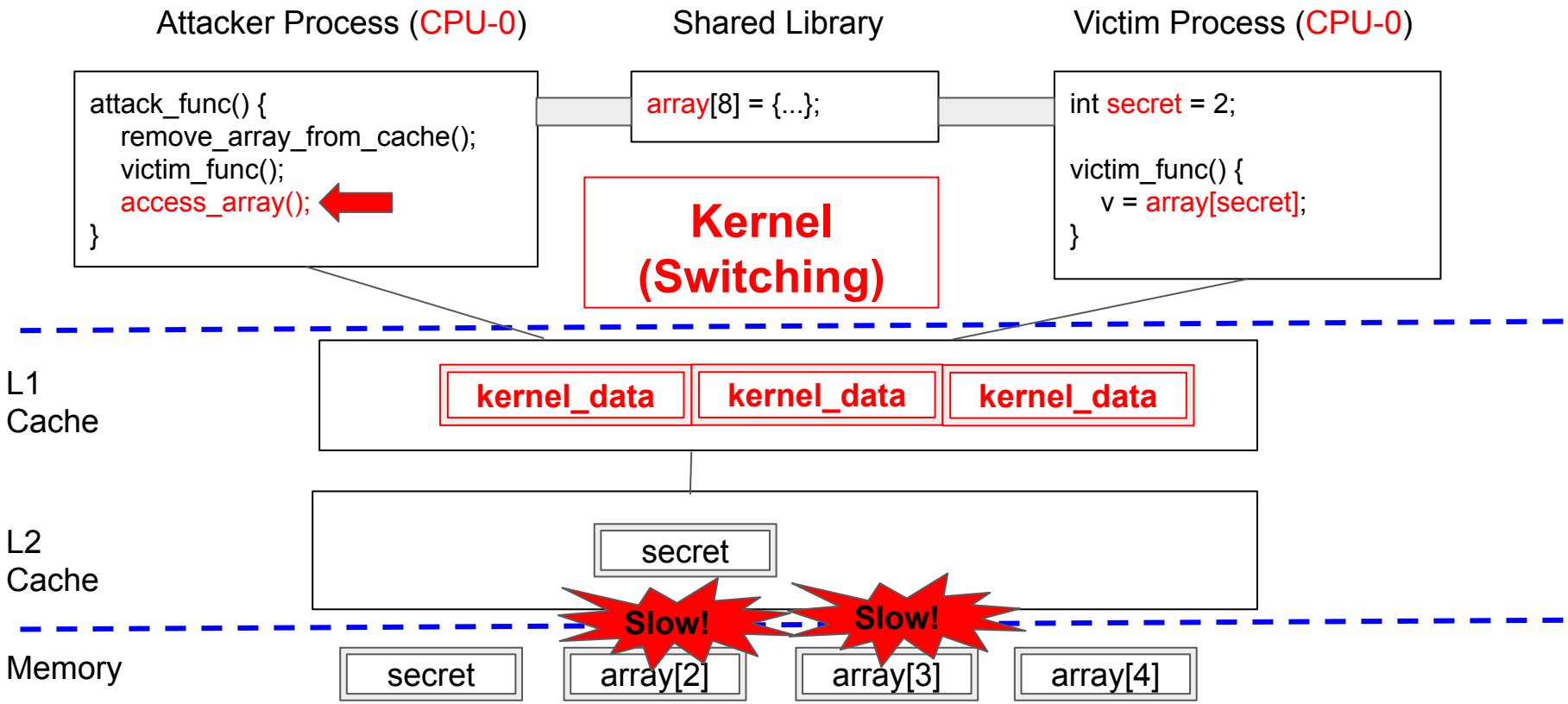
Same Core Scenario (Reality)



Same Core Scenario (Reality) (Cont)



Same Core Scenario (Reality) (Cont)



Solution: SMT (Simultaneous Multithreading)

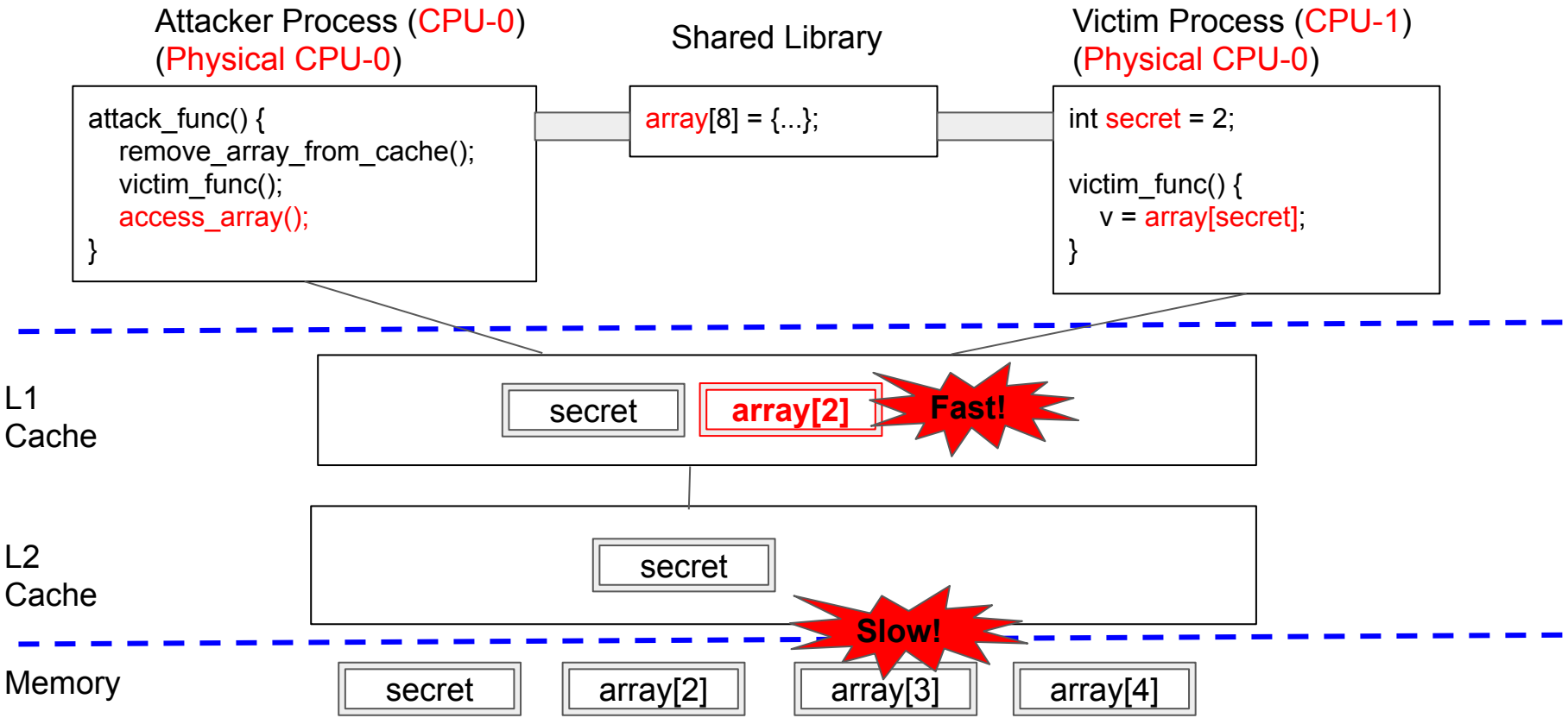
- SMT is also known as Hyperthreading.

- SMT splits one physical core to two logical core.
  In other words, they seems to have two physical core but actually one physical core there.

- Intel => SMT / ARM => No SMT
  It makes attacks easier on Intel CPUs.

- 8 Cores for ARM means 8 physical cores.
  8 Cores for Intel means 4 physical cores and 8 logical cores.

Revisit Attack with SMT



Note

- Many of modern micro-arch attacks are relying on SMT for successful exploitations.
  e.g., MDS (Fallout, RIDL, ZombieLoad), LVI (Load Value Injection)

- Without SMT, many of micro-arch attacks won't work.

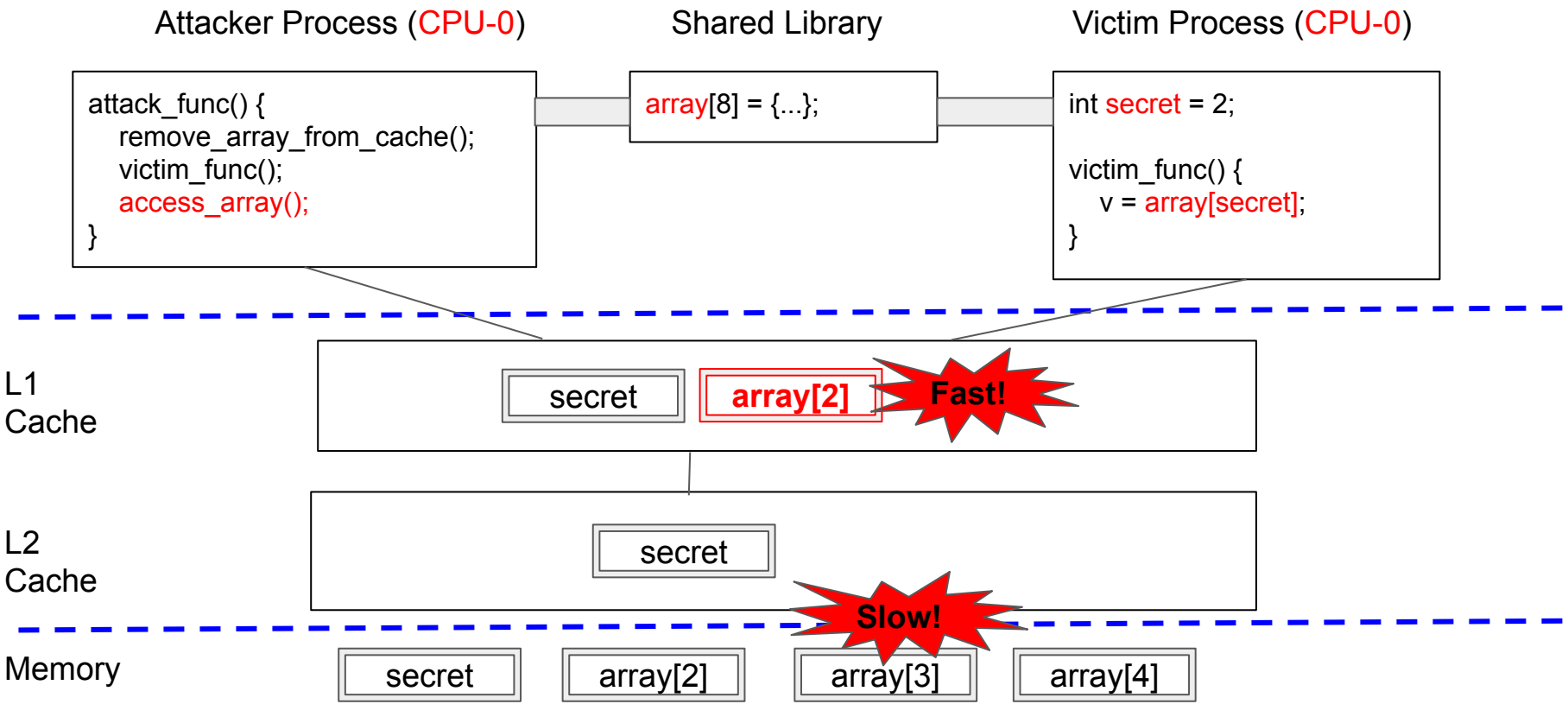
- SMT is a key reason of why researchers and attackers tend to focus on Intel CPU.

Challenge-3:

Cache Replacement Policy

Revisit

Would Same Core Scenario be working even without SMT?
=> Depends on Cache Replacement Policy!



Cache Replacement Policy

Process (CPU-0)

```
attack_func() {
    data1 = 1;
    data2 = 2;
    data3 = 3;
    data4 = 4;
}
```

All data1 ~ data4 are stored on set-0.

Data4 is going to be newly loaded on cache set-0.

Cache

set-0	**data1**	**data2**	**data3**
set-1			
set-2			
	way-0	**way-1**	**way-2**

How can we determine data to go away?

=> Cache Replacement Policy

Cache Replacement Policy (Cont)

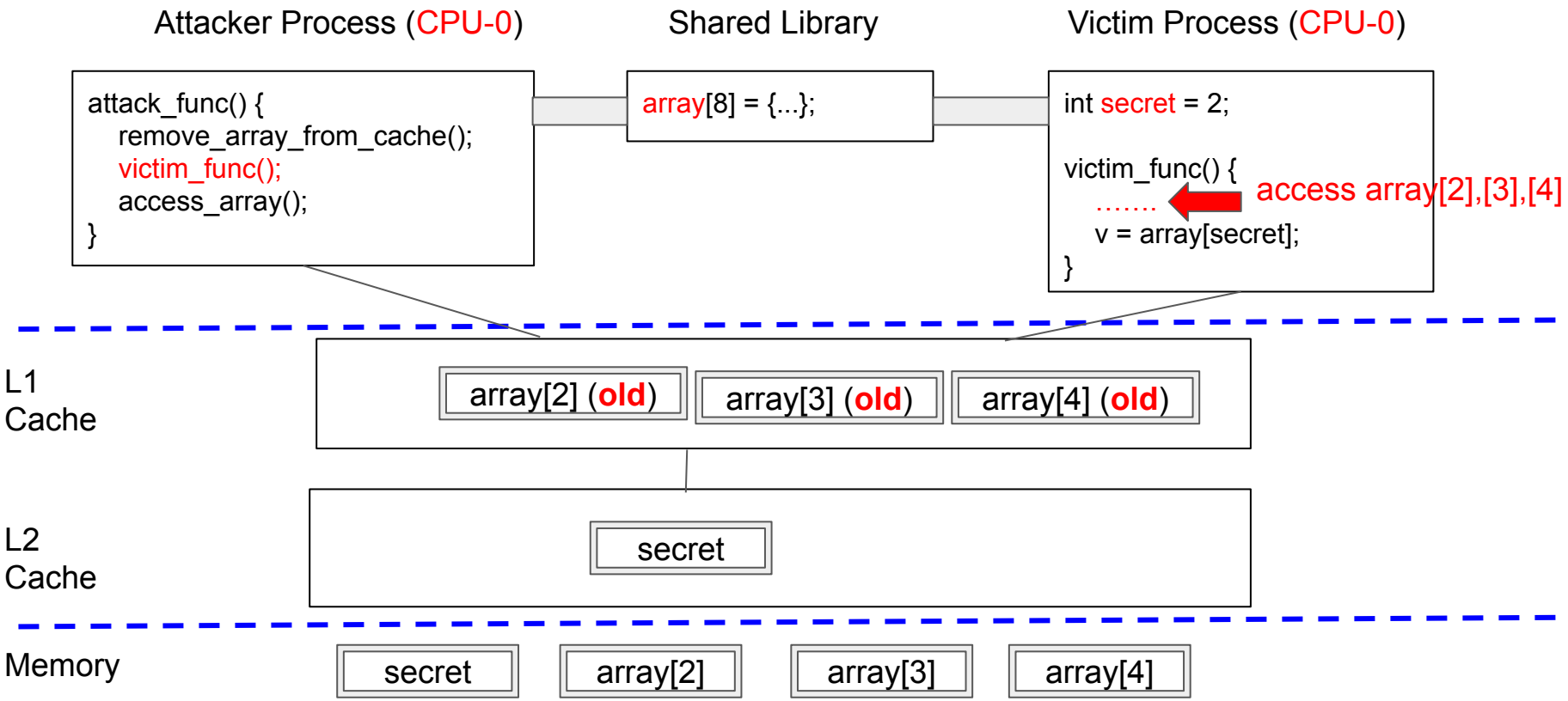
- Cache Replacement Policy is a strategy of how to determine which data is going to be evicted if a cache set is full.

- Pseudo-Random Policy
  - Randomly choose data to be evicted.

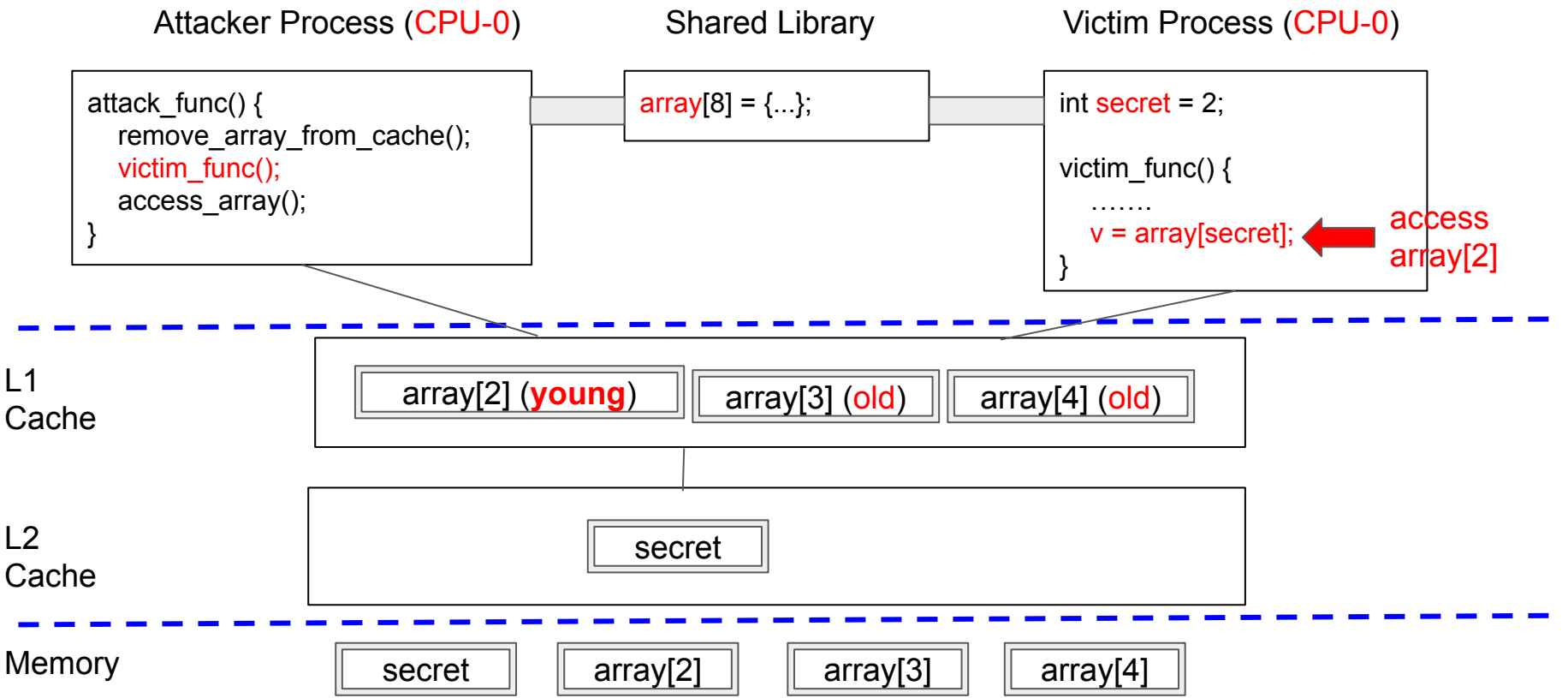
- Least Recently Used (LRU)
  - Choose the oldest data in cache. Recently used data most likely remains in cache.

- Intel => LRU
  ARM => Most of it use Pseudo-Random, A few of it use LRU.

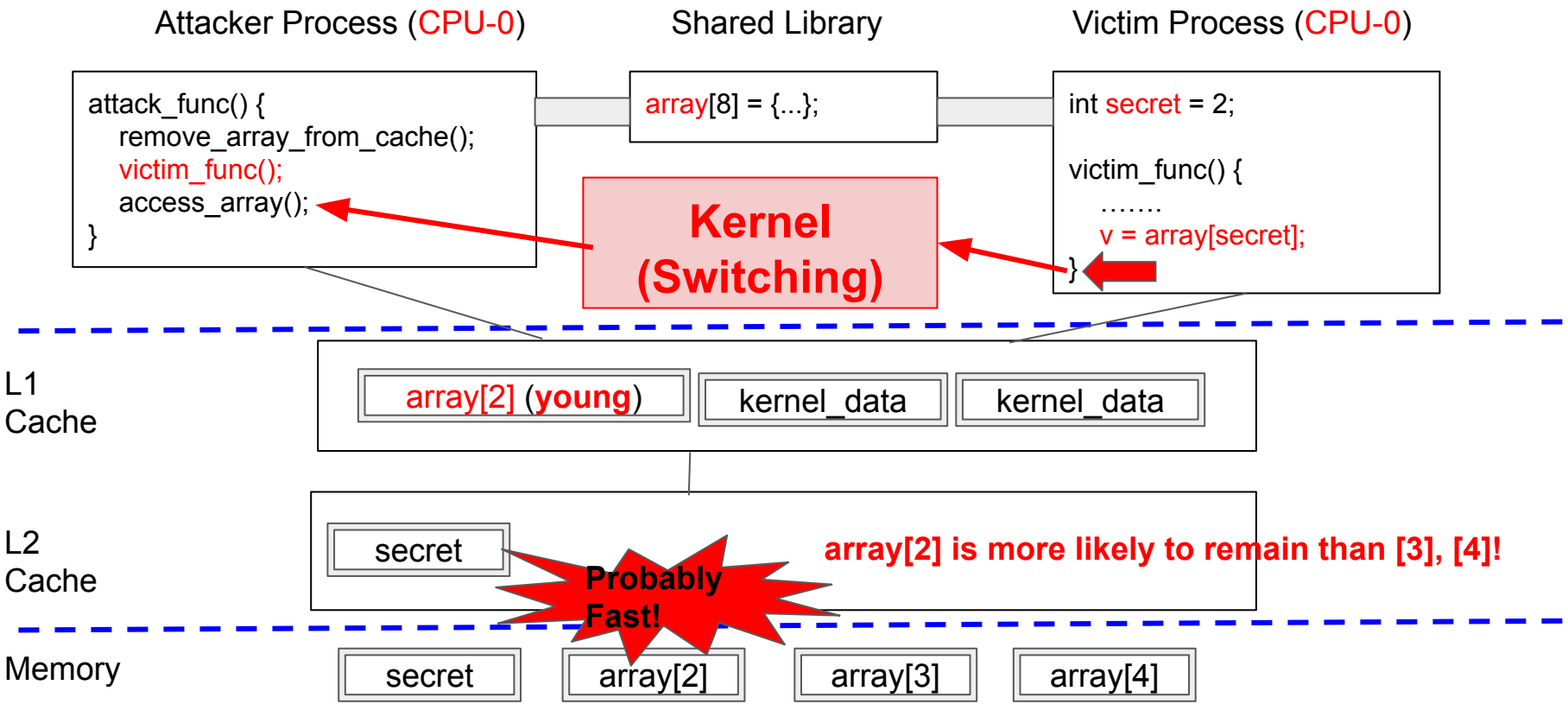
Revisit Attack with LRU policy



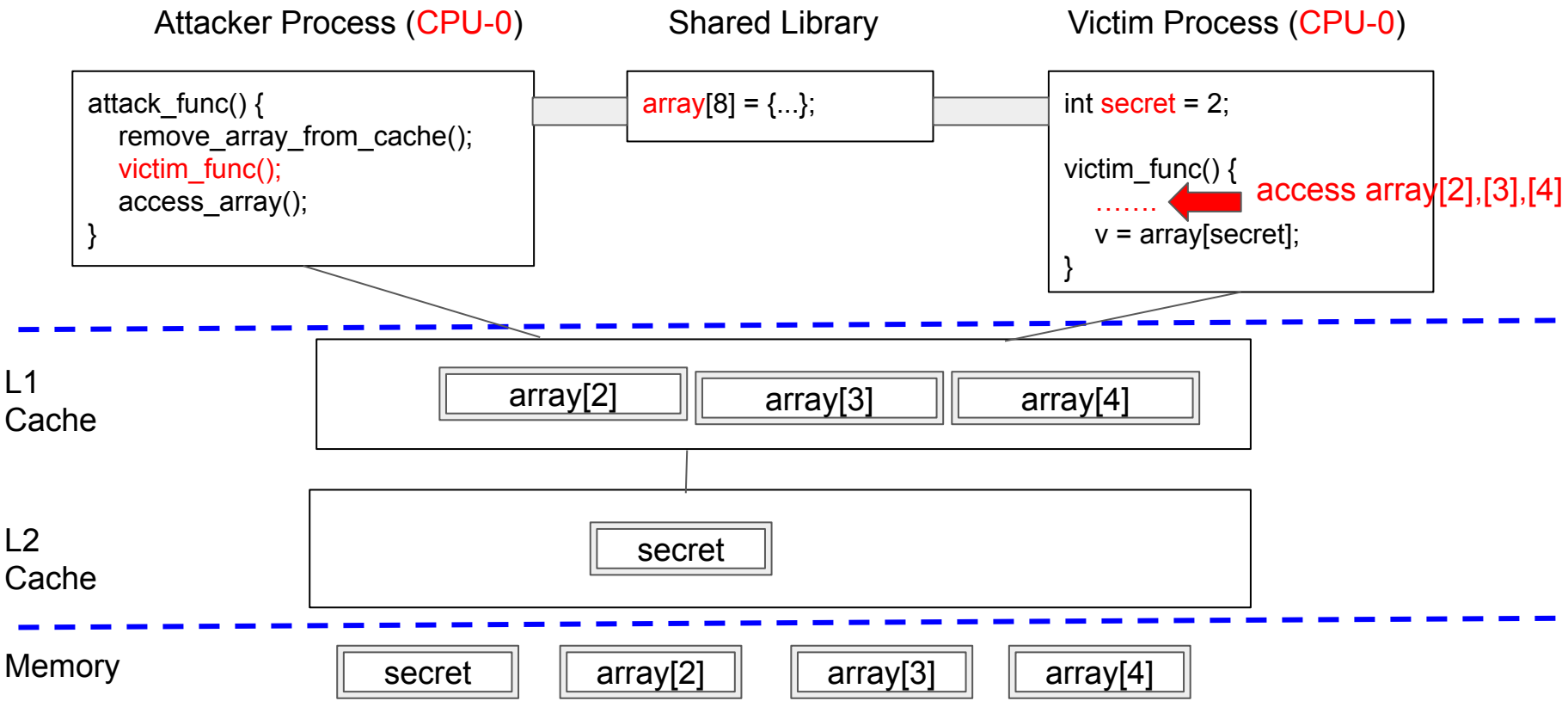
Revisit Attack with LRU policy (Cont)



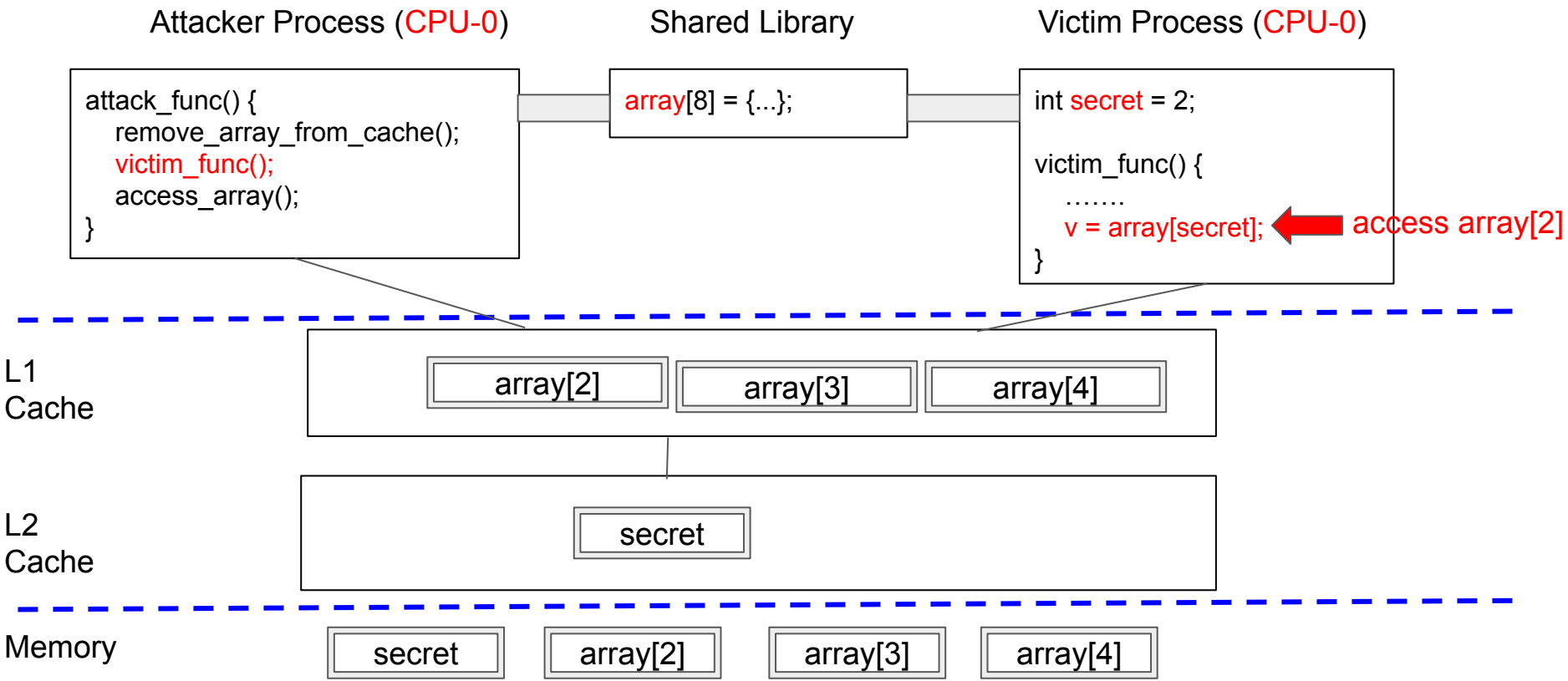
Revisit Attack with LRU policy (Cont)



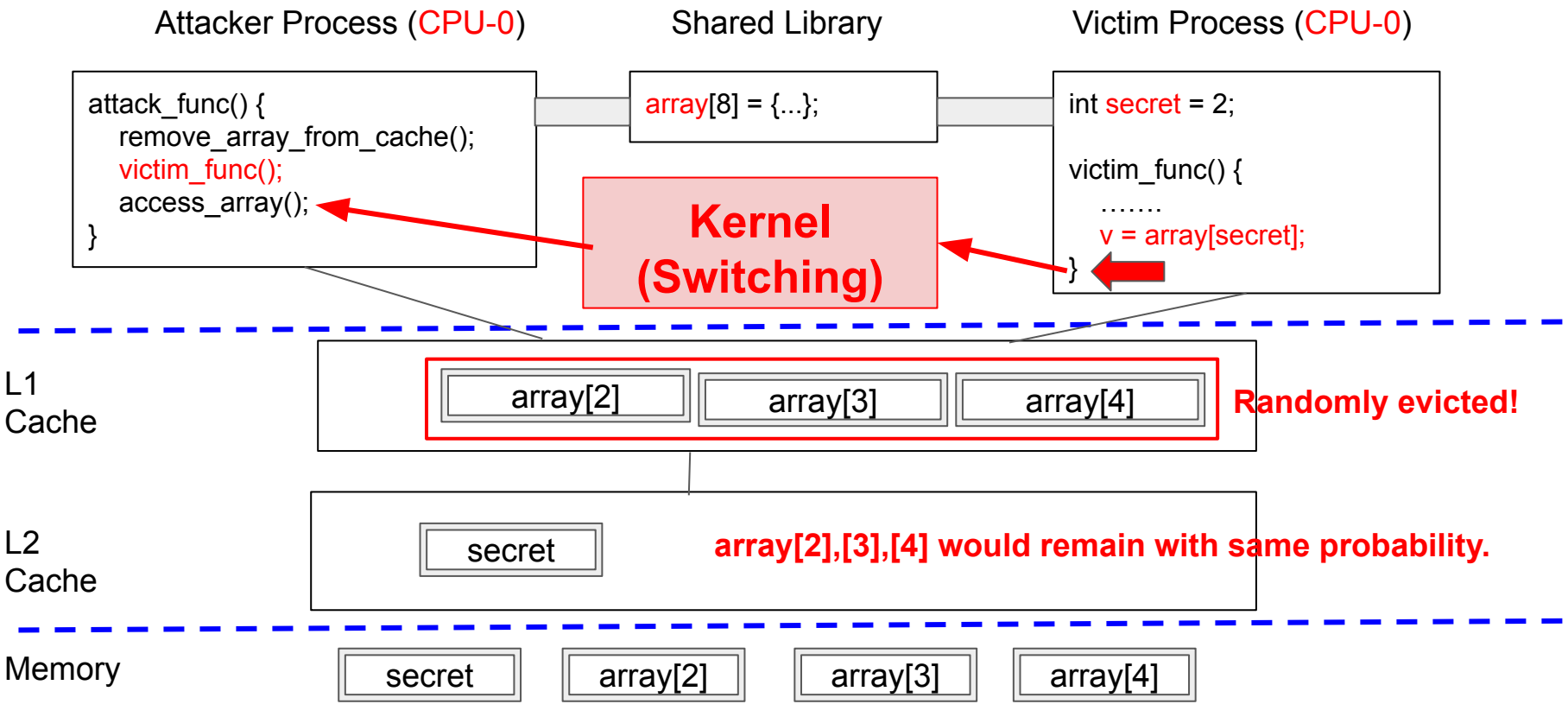
Revisit Attack with Random policy



Revisit Attack with Random policy (Cont)

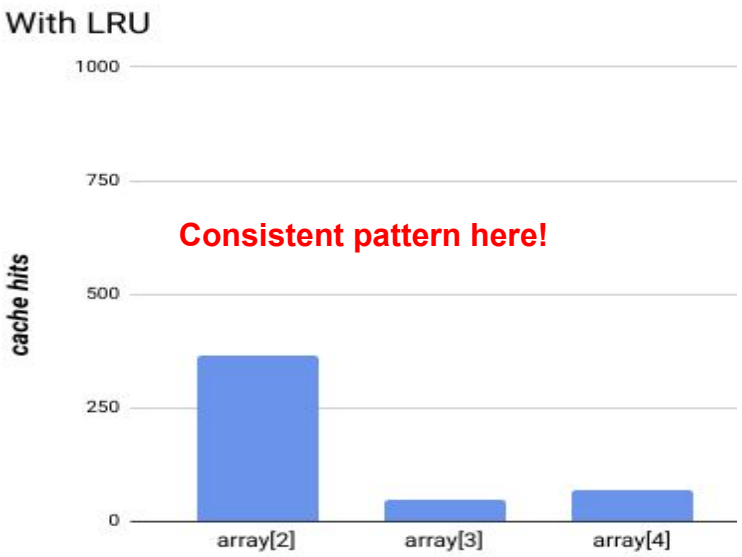


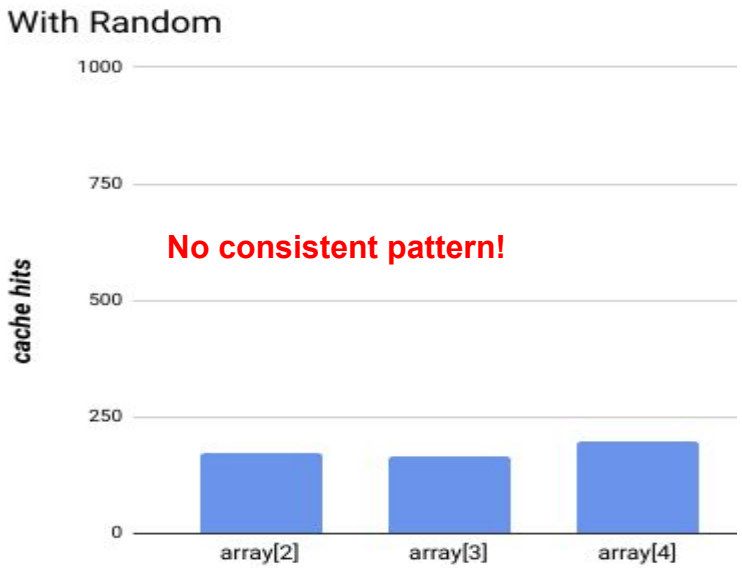
Revisit Attack with Random policy (Cont)



Graph: LRU vs Random

- Assume that attackers tried the attack 1000 times for each replacement policy.





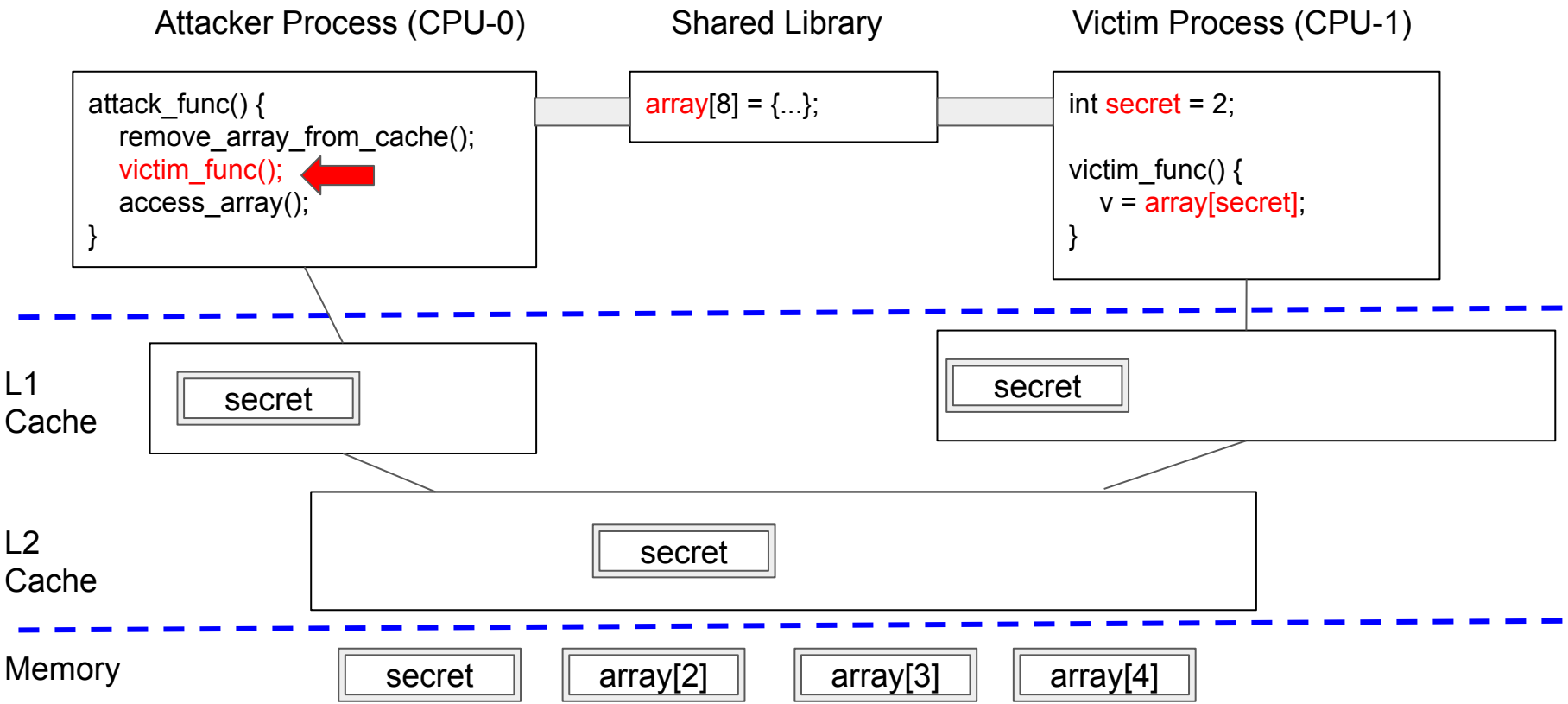
RELOAD+REFRESH

- Cache attacks using EVICT+RELOAD makes many number of cache misses which can be an abnormal behavior.

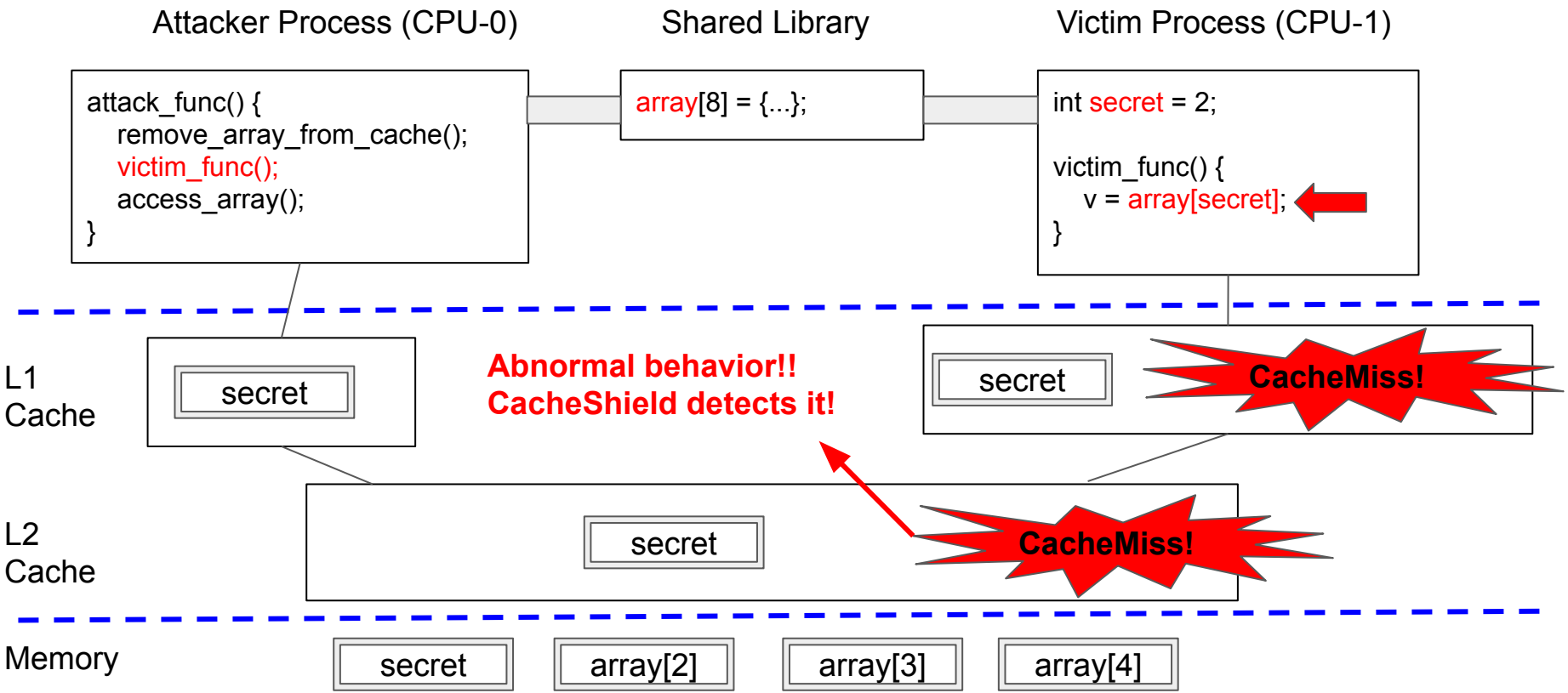
- So defense solution could detect an attempt of cache attacks by abnormal detection based on the number of cache misses or how much time it takes.
  => CacheShield (CODASPY 2018)

- How can attackers bypass the defense? ⇒ RELOAD+REFRESH (USENIX Security 2020) exploited LRU policy to bypass defense solutions against cache attacks.

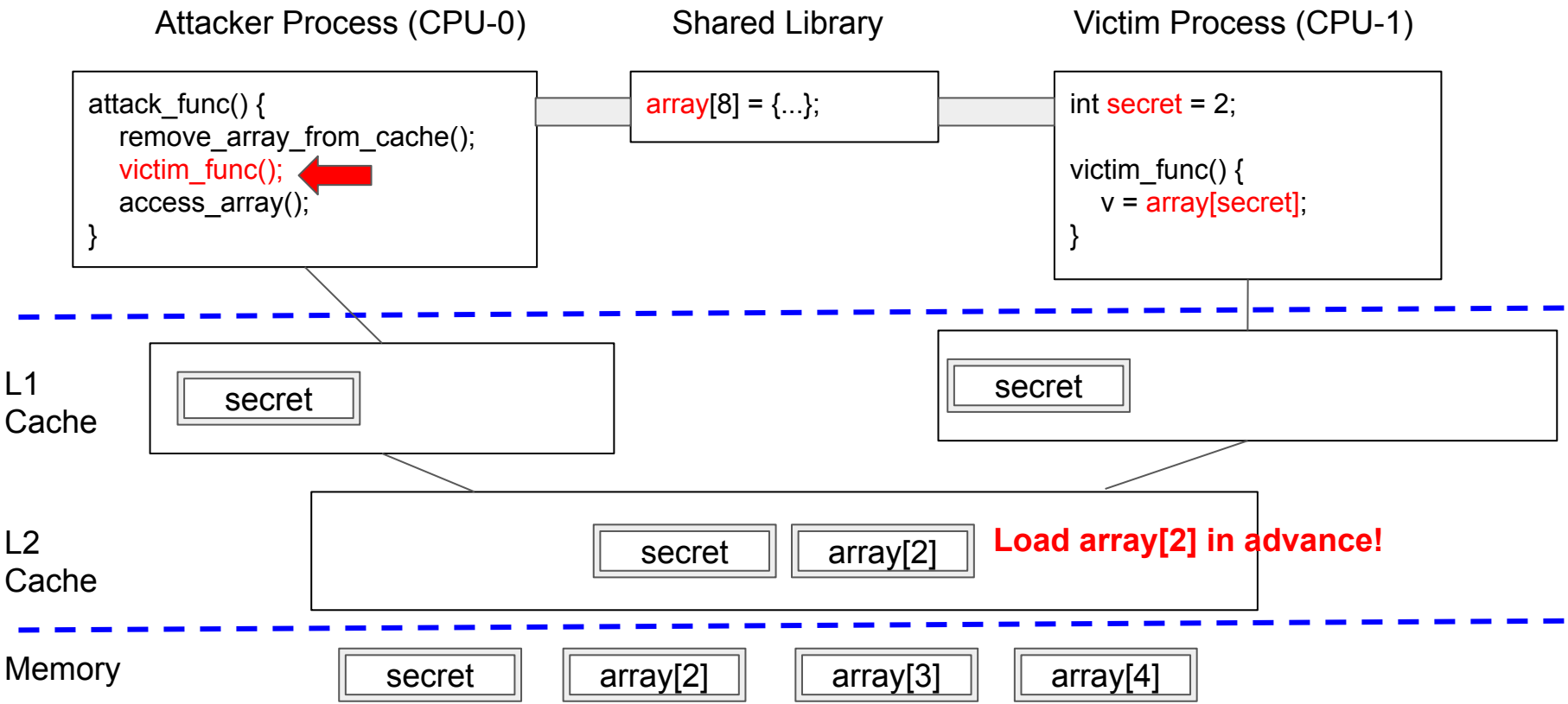
Revisit attack



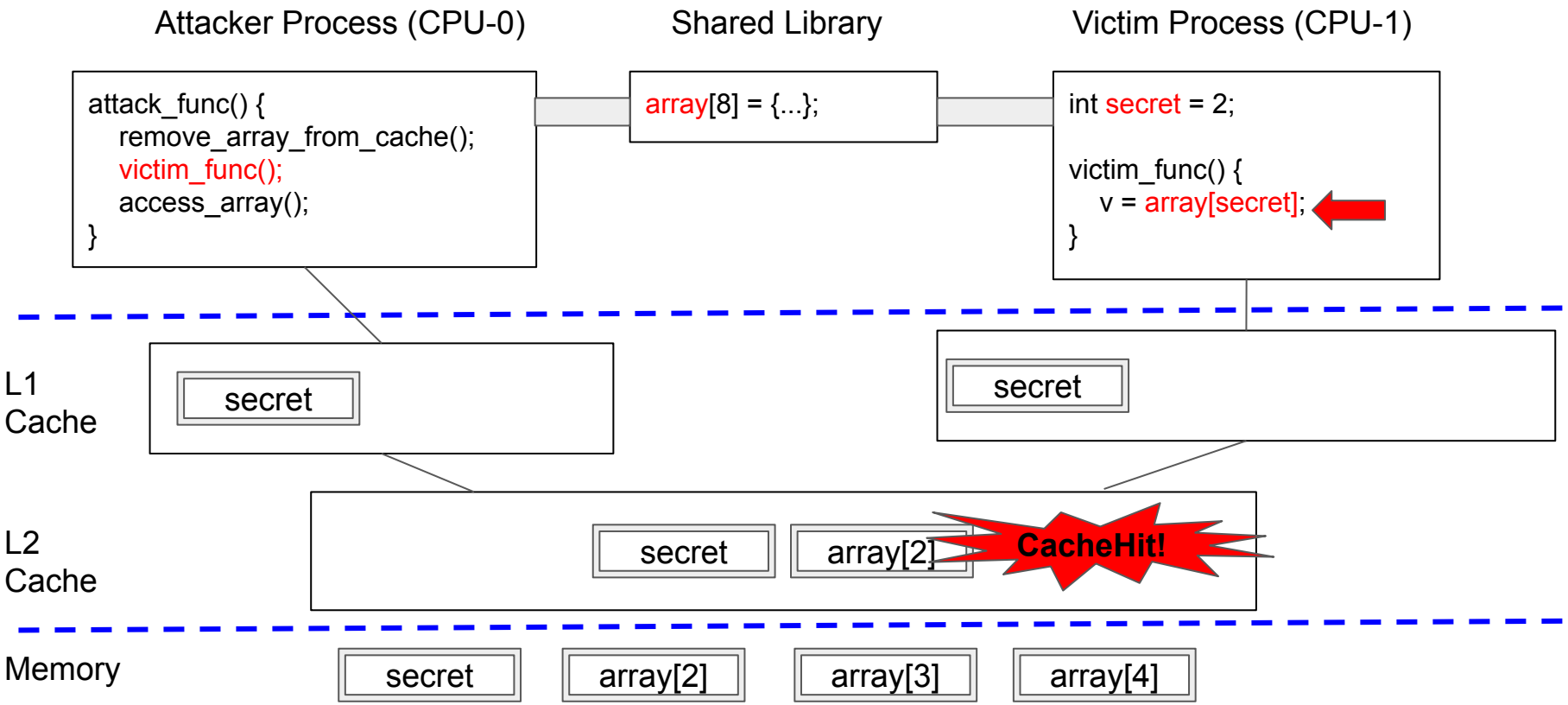
Revisit attack: Cache miss happens at all time!



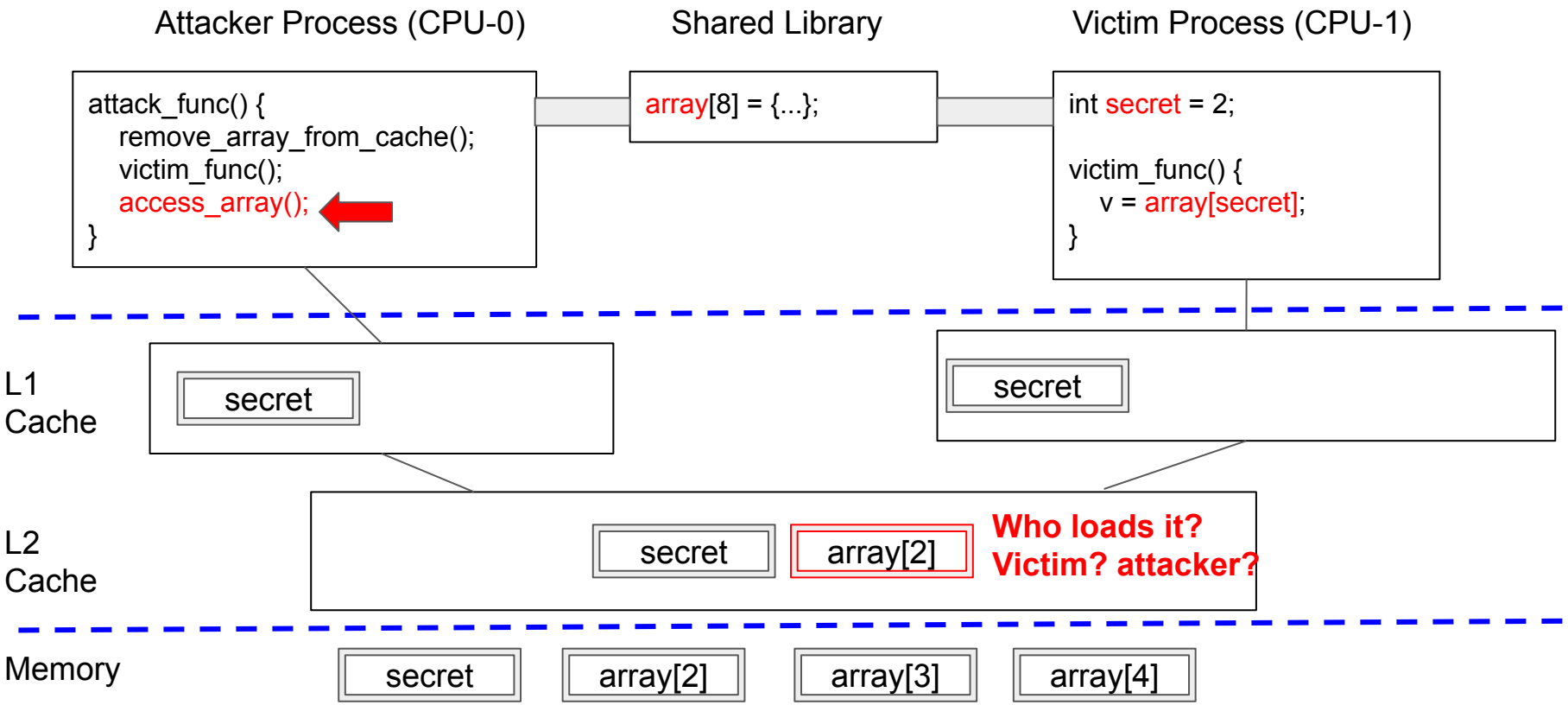
How can we bypass CacheShiled?



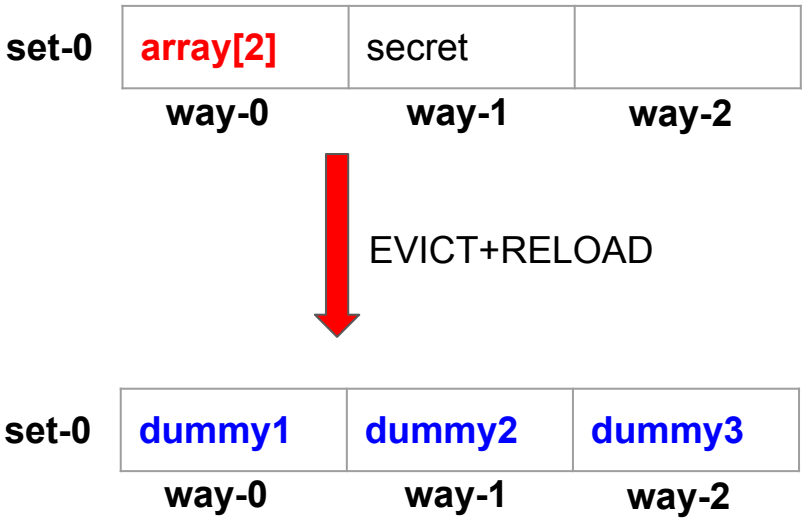
How can we bypass CacheShiled?

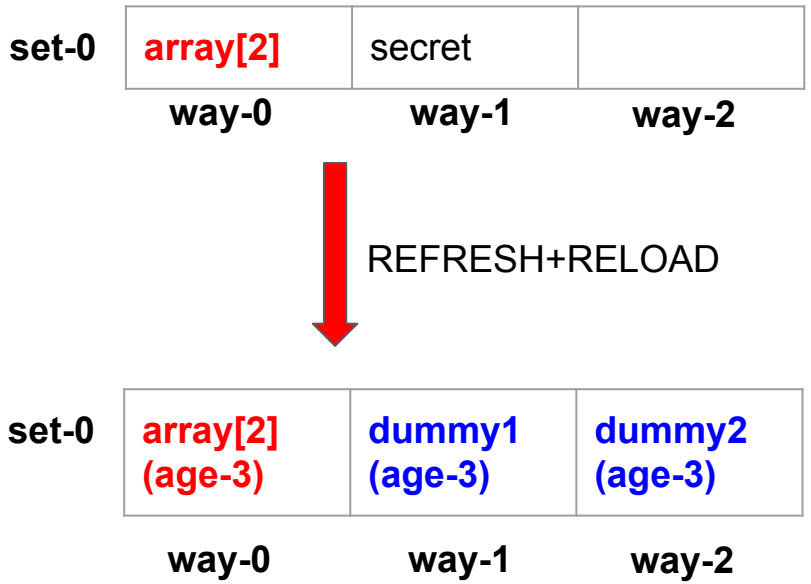


How can we distinguish victim's access from ours?



RELOAD+REFRESH





RELOAD+REFRESH (Cont)

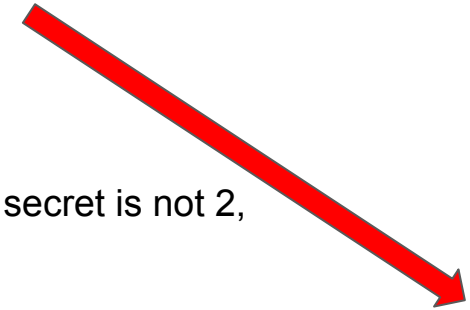
Victim Process (CPU-1)

```
int secret = 2;

victim_func() {
    v = array[secret];
}
```

If secret is 2,

	way-0	way-1	way-2
set-0	array[2] (age-1)	dummy1 (age-5)	dummy2 (age-5)

If secret is not 2,



	way-0	way-1	way-2
set-0	array[2] (age-5)	data1 (age-1)	dummy2 (age-5)

RELOAD+REFRESH (Cont)

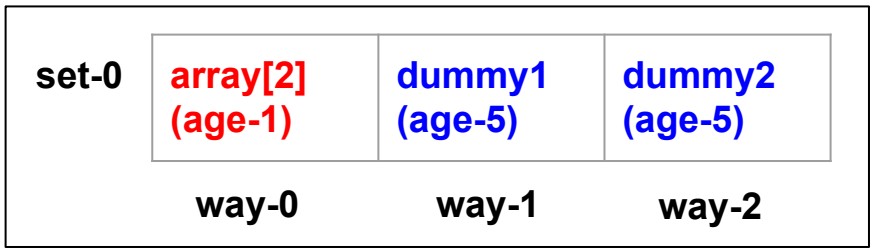
Attacker Process (CPU-0)

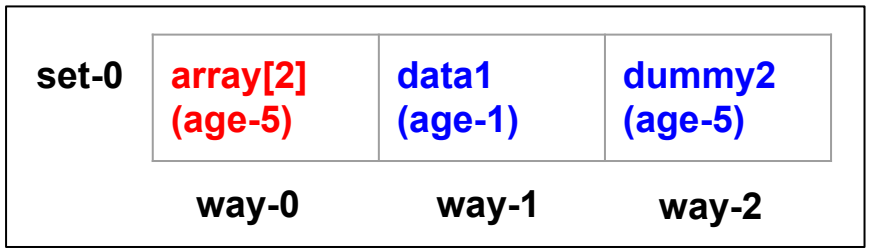
```
attack_func() {
    remove_array_from_cache();
    victim_func();
    access_array();
}
```

dummy3
dummy2

If secret is 2,
(victim did access)

If secret is not 2, (victim didn't access)





RELOAD+REFRESH (Cont)

Attacker Process (CPU-0)

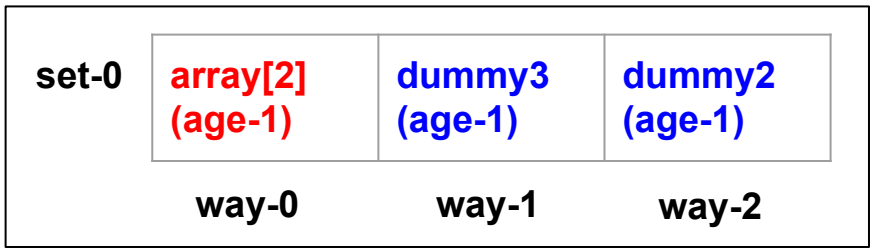
```
attack_func() {
    remove_array_from_cache();
    victim_func();
    access_array();
}
```

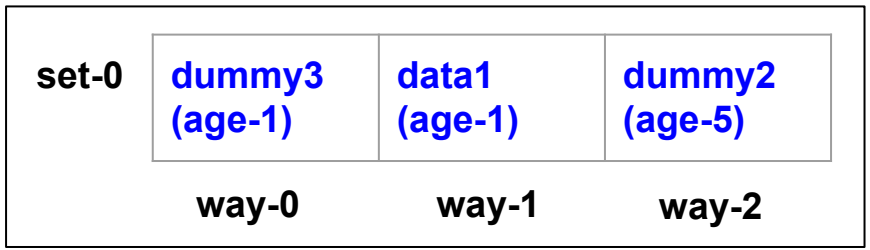
dummy3
dummy2

Access dummy3 and dummy2

If secret is 2,
(victim did access)

If secret is not 2, (victim didn't access)





RELOAD+REFRESH (Cont)

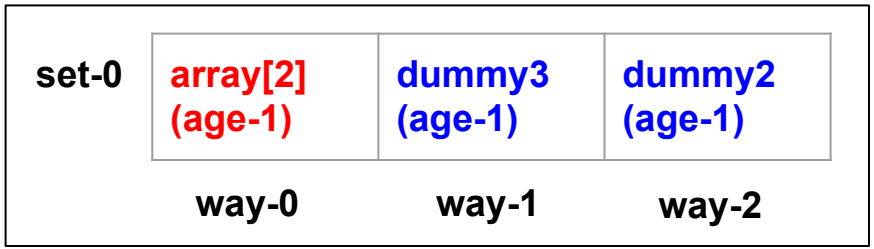
Attacker Process (CPU-0)

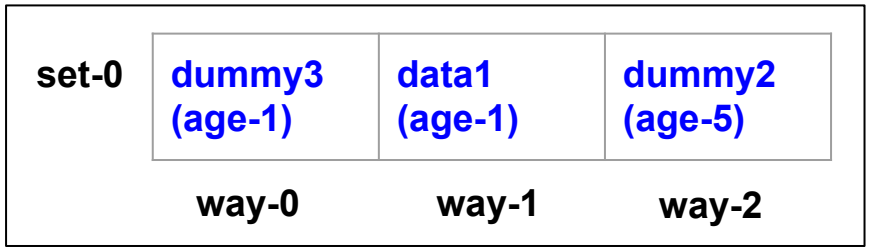
```
attack_func() {
    remove_array_from_cache();
    victim_func();
    access_array();
}
```

array[2] ← Reload array[2].
If cache hit → secret is 2.
If cache miss → secret is not 2.

If secret is 2,
(victim did access)

If secret is not 2, (victim didn't access)





Note

- SmokeBomb (MobiSys 2020) demonstrated this attack scenario on ARM.
- RELOAD+REFRESH (USENIX Security 2020) exploited LRU policy to circumvent defense solutions against cache attacks.
- CacheShield (CODASPY 2018) detects attempts of cache attacks by monitoring cache misses in victim side.

End