Attack of the Knights:
Side-Channel Attack on Non-Uniform Cache Architecture

Farabi Mahmud, Sungkeun Kim, Harpreet Singh Chawla, EJ Kim, Chia-Che Tsai, Abdullah Muzahid
Texas A&M University, College Station, TX

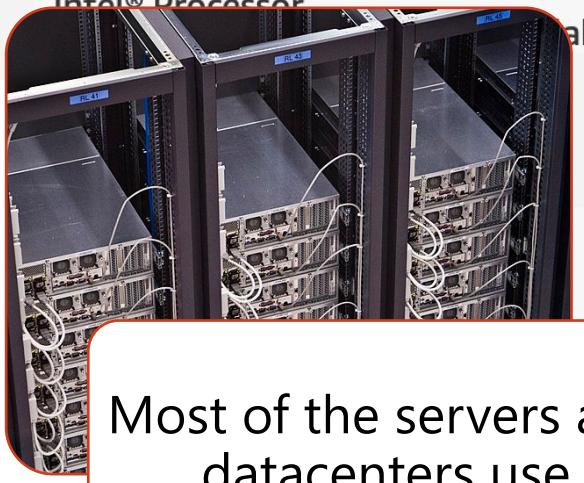
Non-Uniform Cache Architecture is Everywhere!

AWS Instance Type	M6i/M6id	M5zn	M5n	M5	T3 (Burstable)	T2 (Burstable)
Intel® Processor	3rd Gen Intel Xeon Scalable	2nd Gen Intel® Xeon® Scalable Processors	2nd Gen Intel® Xeon® Scalable Processors	Intel® Xeon® Platinum 8175M Processors	Intel® Xeon® Scalable Processors	Intel® Xeon® Processors

AWS Instance Type	DL1	VT1	P4	G4	P3
Intel® Processor	2nd Gen Intel®	2nd Gen Intel®	2nd Gen Intel®	Intel® Xeon®	Intel® Xeon®

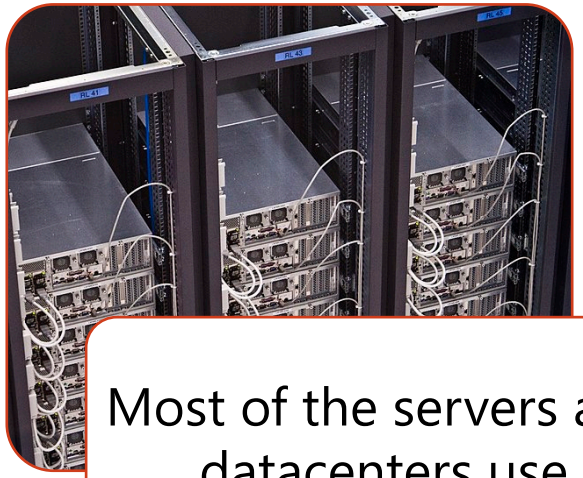
AWS Instance Type	R6i/R6id	X2idn/X2iedn	X2iezn	R5b	R5n	R5/R5d	X1e/X1	Z1d
Intel® Processor	3rd Gen Intel® Xeon®	3rd Gen Intel® Xeon® Scalable	2nd Gen Intel® Xeon®	2nd Gen Xeon®	2nd Gen Intel® Xeon®	Intel® Xeon® Platinum	Intel® Xeon® E7 8880 v3	Intel® Xeon® Platinum

AWS Instance Type	C6i/C6id	C5	C5n
Intel® Processor	3rd Gen Intel® Xeon® Scalable Processors	2nd Gen Intel® Xeon® Scalable Processors	Intel® Xeon® Platinum 8124M Processors



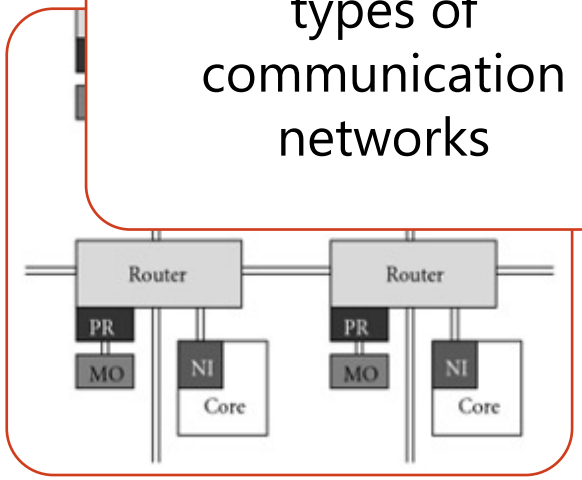
Most of the servers and datacenters use multicore processors

Non-Uniform Cache Architecture is Everywhere!



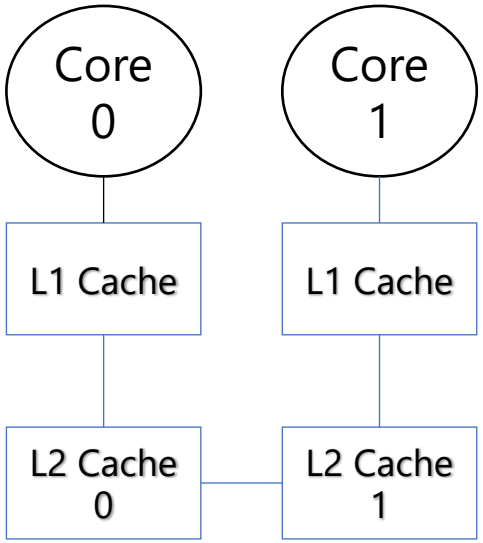
Most of the servers and datacenters use multicore processors

Multiple cores are connected via different types of communication networks



Exploiting Secret dependent communication can be fun (and $$$)

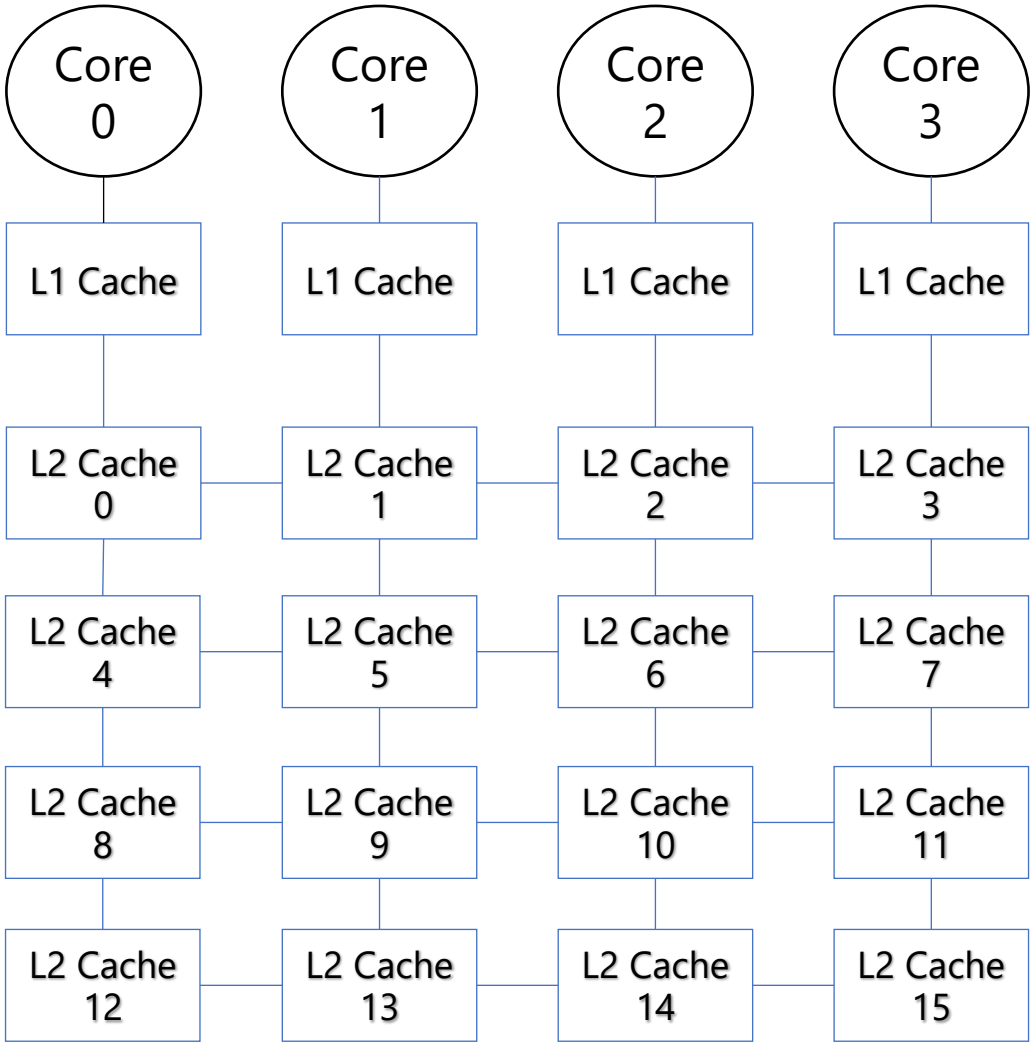
NUCA Architecture



Every Core has -

- Private Cache
- Shared LLC (L2 Cache)

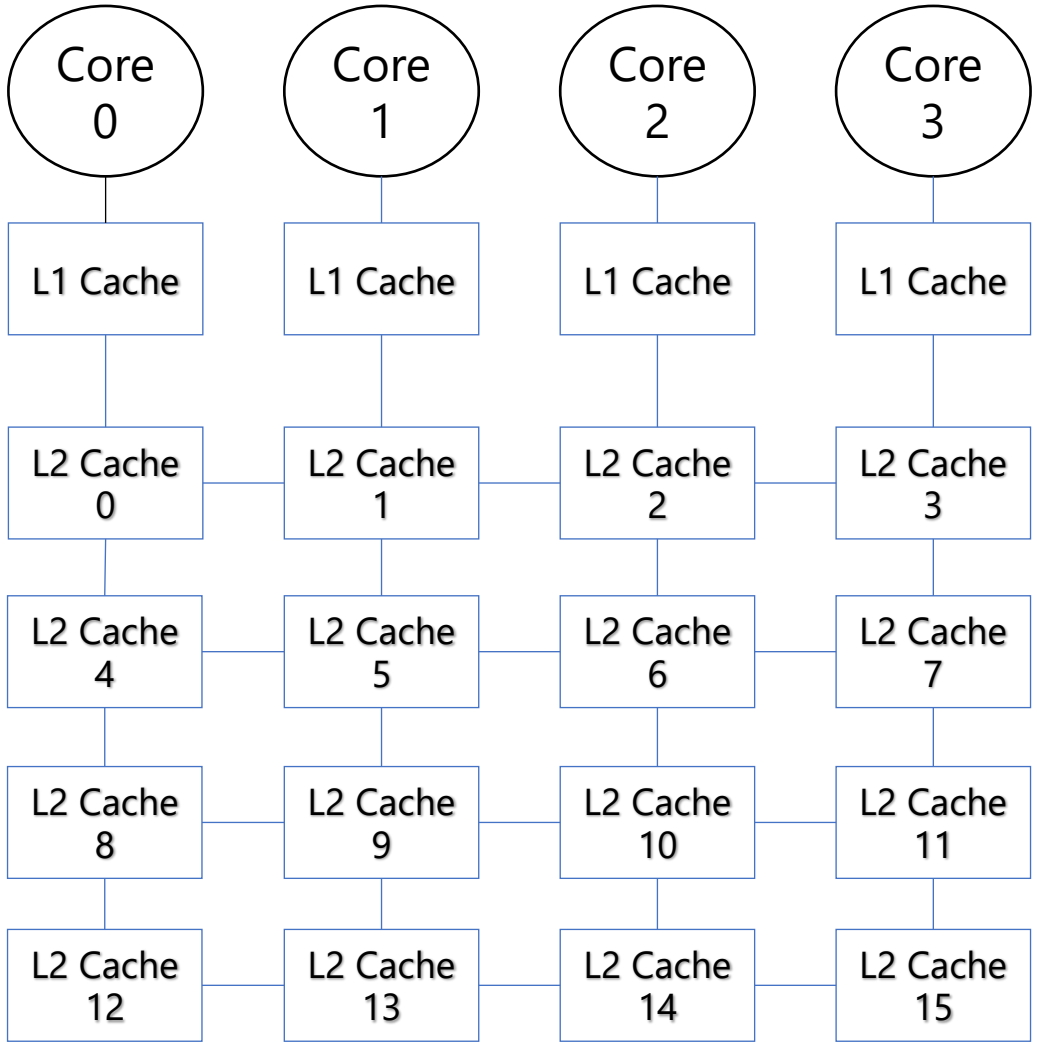
NUCA Architecture



Every Core has -

- Private Cache
- Shared LLC (L2 Cache)

NUCA Architecture



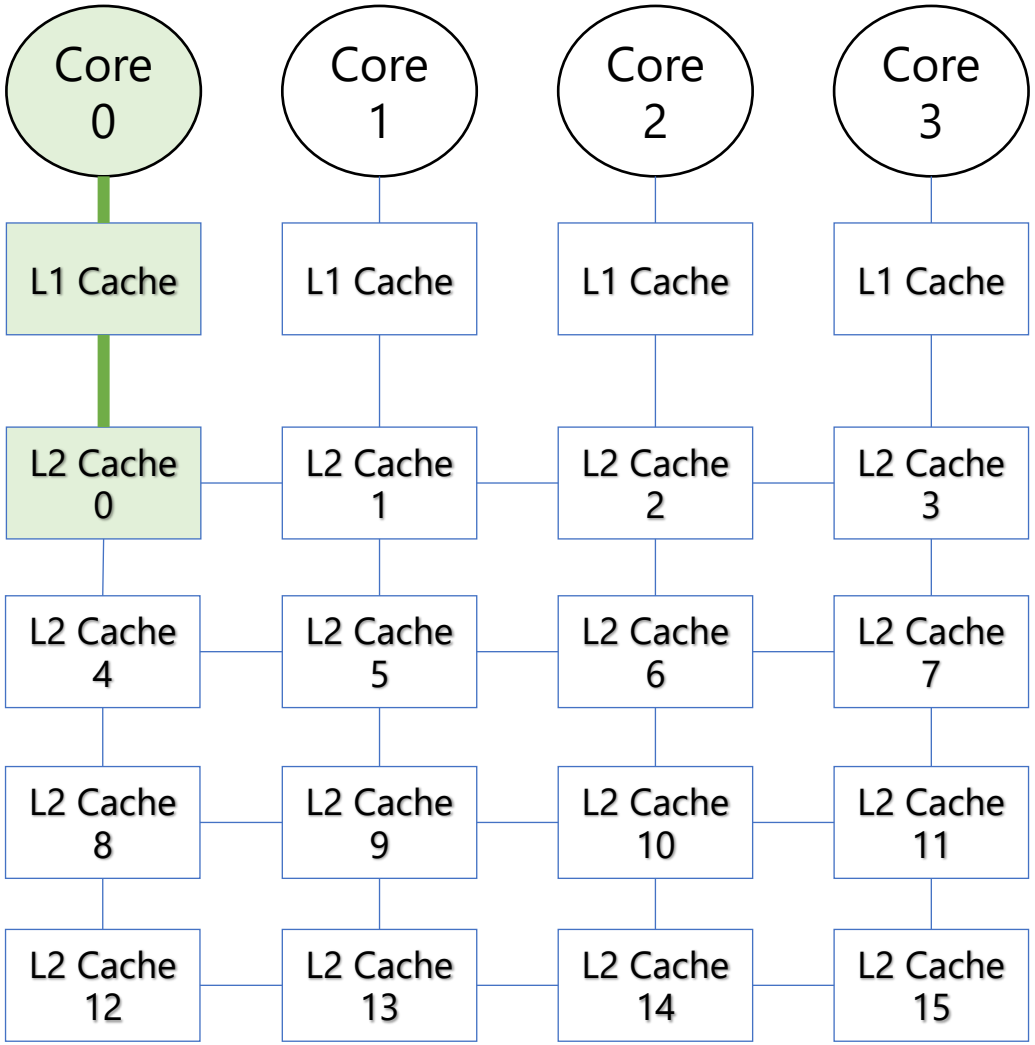
Every Core has -

- Private Cache
- Shared LLC (L2 Cache)

4 x 4 configuration

- Connected via 2D mesh network
- Data can be available at one or multiple L2 Cache locations

NUCA Architecture



Every Core has -

- Private Cache
- Shared LLC (L2 Cache)

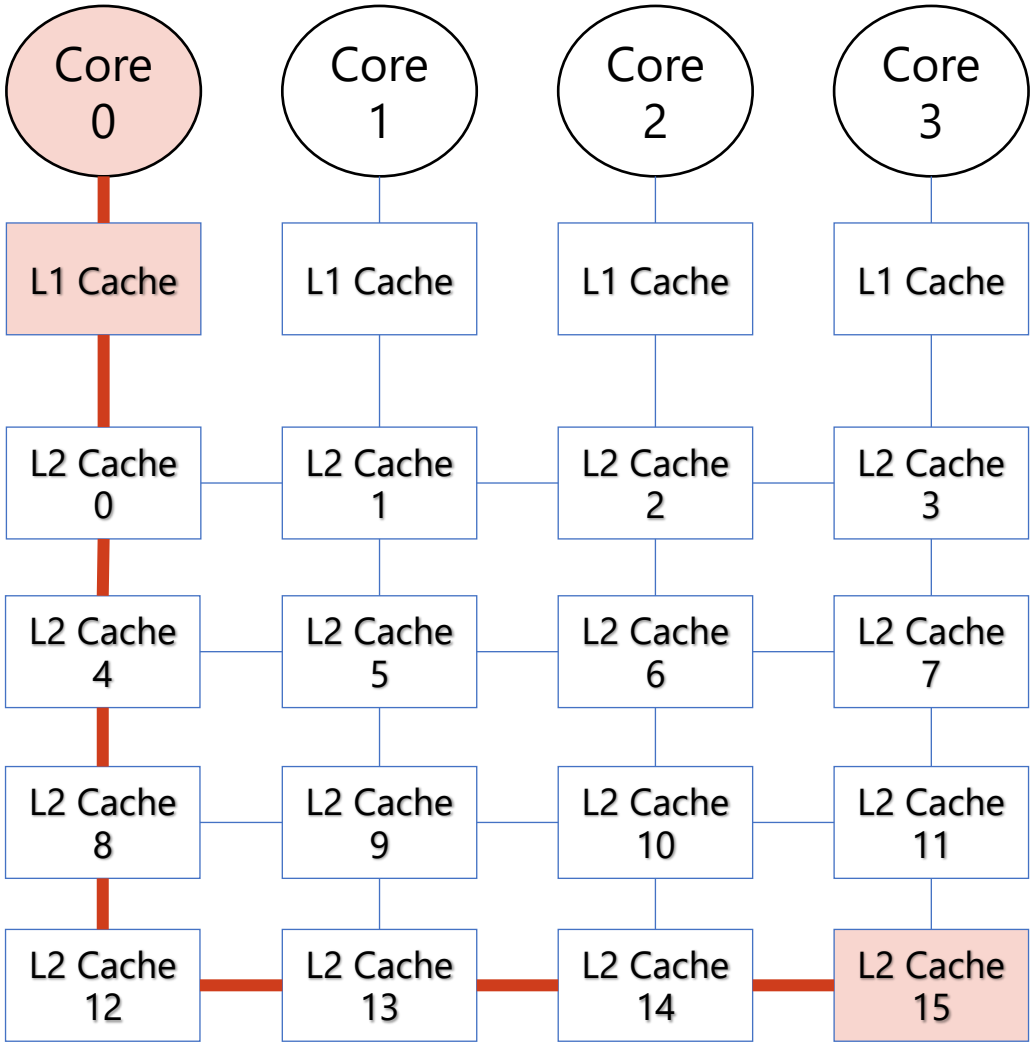
4 x 4 configuration

- Connected via 2D mesh network
- Data can be available at one or multiple L2 Cache locations

LLC Hit Timing is dependent on LLC Location -

- Nearer L2 Cache accesses are faster
- Farther L2 Cache accesses are slower

NUCA Architecture



Every Core has -

- Private Cache
- Shared LLC (L2 Cache)

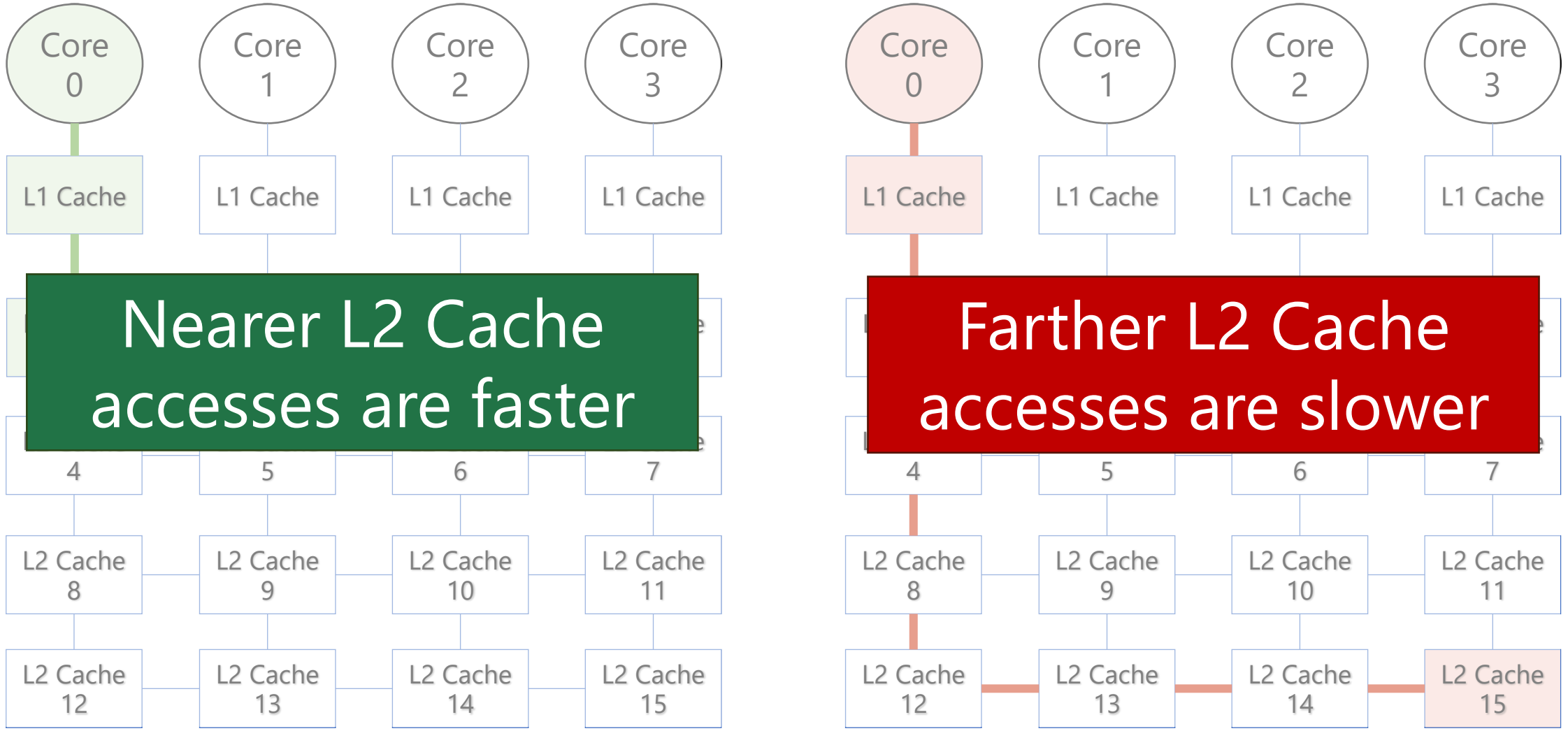
4 x 4 configuration

- Connected via 2D mesh network
- Data can be available at one or multiple L2 Cache locations

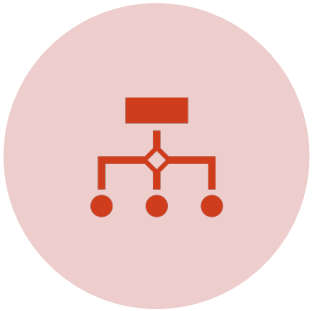
LLC Hit Timing is dependent on LLC Location -

- Nearer L2 Cache accesses are faster
- Farther L2 Cache accesses are slower

NUCA Architecture



Cache Side-Channel Attacks on NUCA Architecture



DON’T MESH AROUND
[USENIX’22]

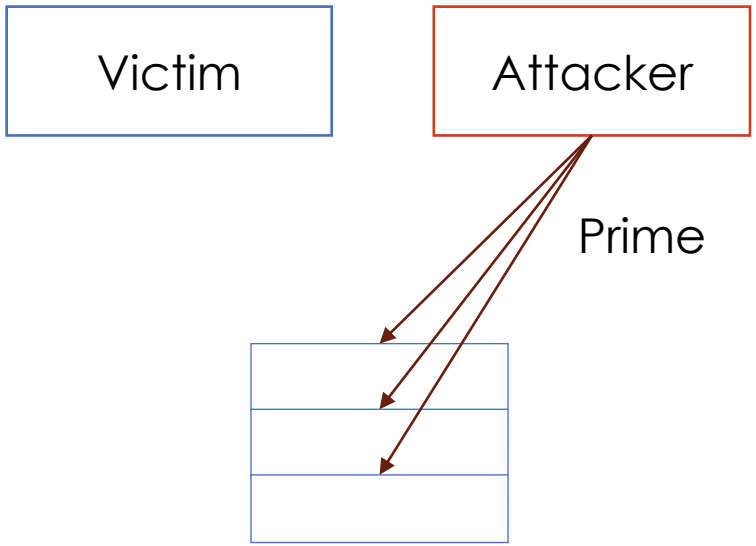
MESHUP
[S&P’22]

LORD OF THE RING(S)
[USENIX’21]

ADVERSARIAL PREFETCH
[S&P’21]

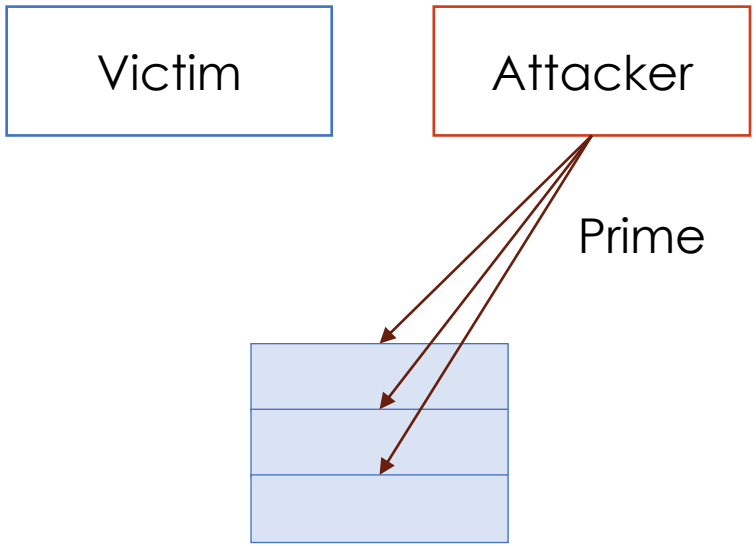
Existing Cache Side-Channel Attacks

Prime + Probe
[Crypto'06]



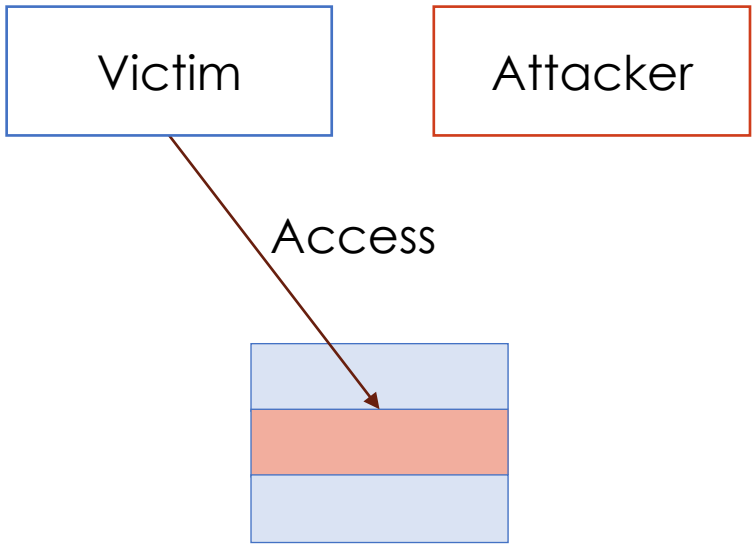
Existing Cache Side-Channel Attacks

Prime + Probe
[Crypto'06]



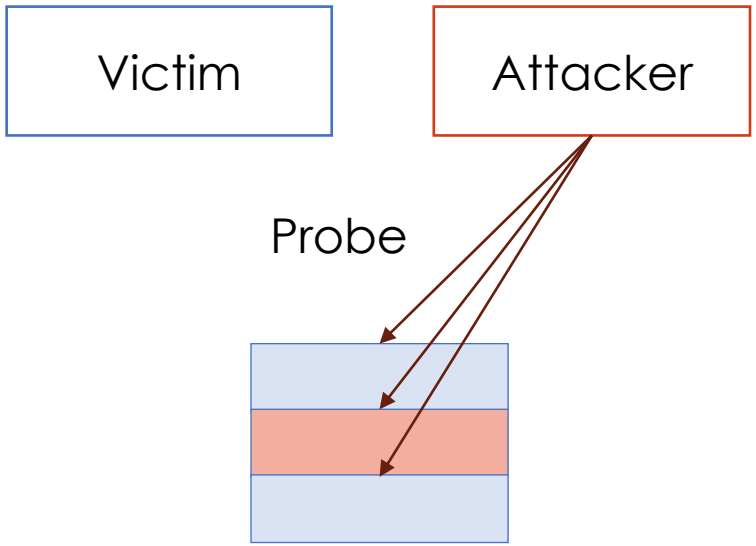
Existing Cache Side-Channel Attacks

Prime + Probe
[Crypto'06]



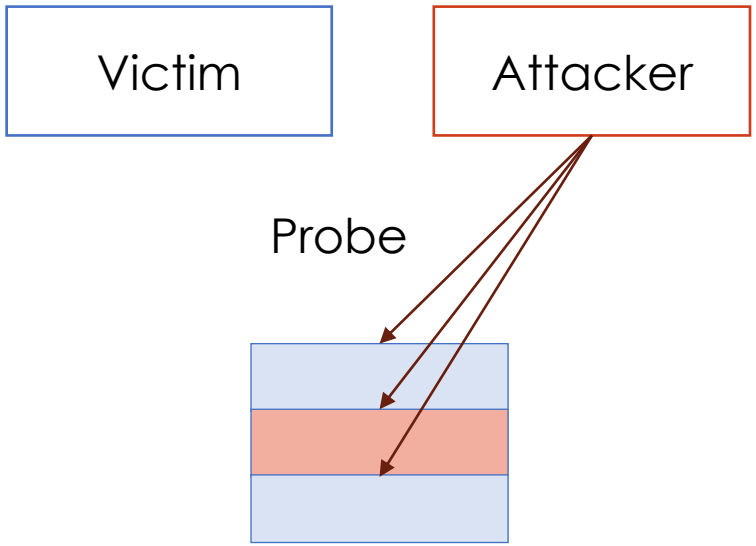
Existing Cache Side-Channel Attacks

Prime + Probe
[Crypto'06]

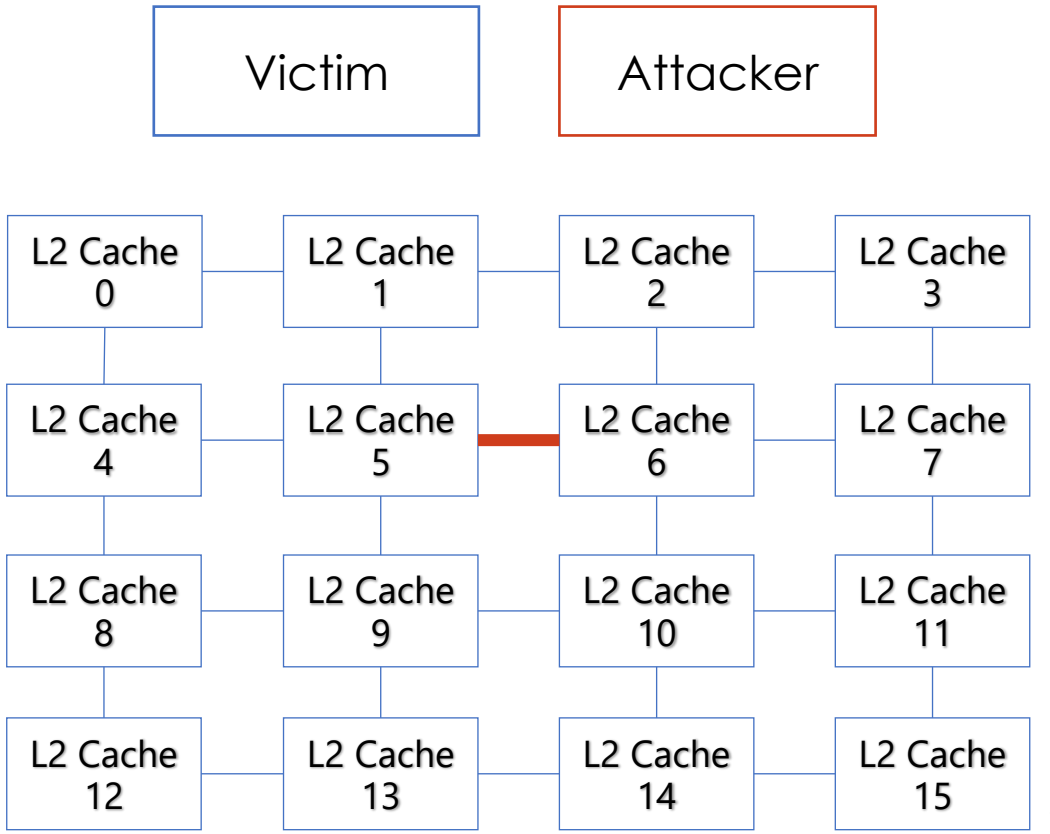


Existing Cache Side-Channel Attacks

Prime + Probe
[Crypto'06]

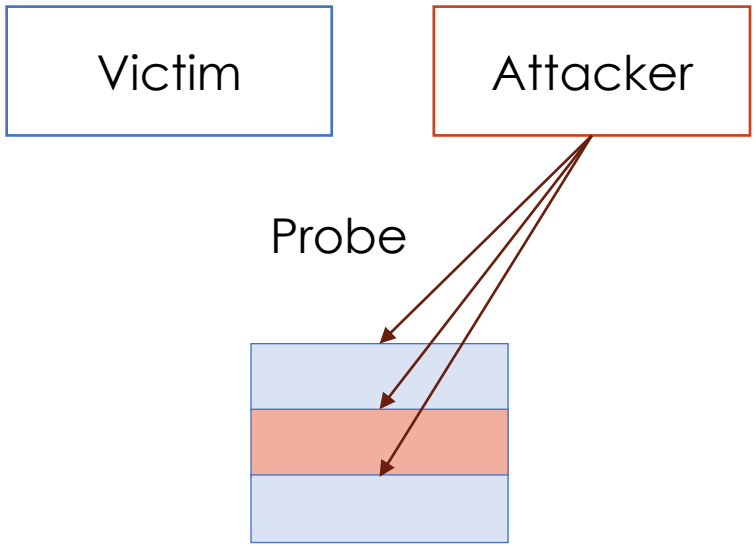


Don't Mesh Around
[USENIX'22]

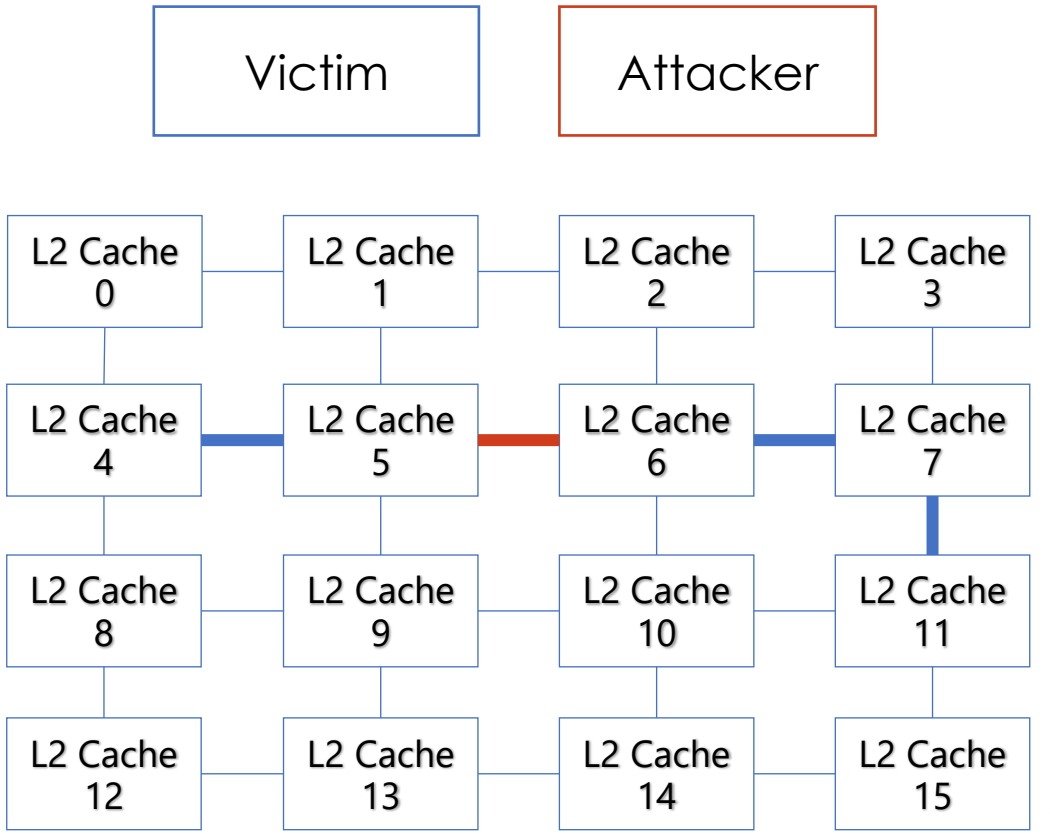


Existing Cache Side-Channel Attacks

Prime + Probe
[Crypto'06]

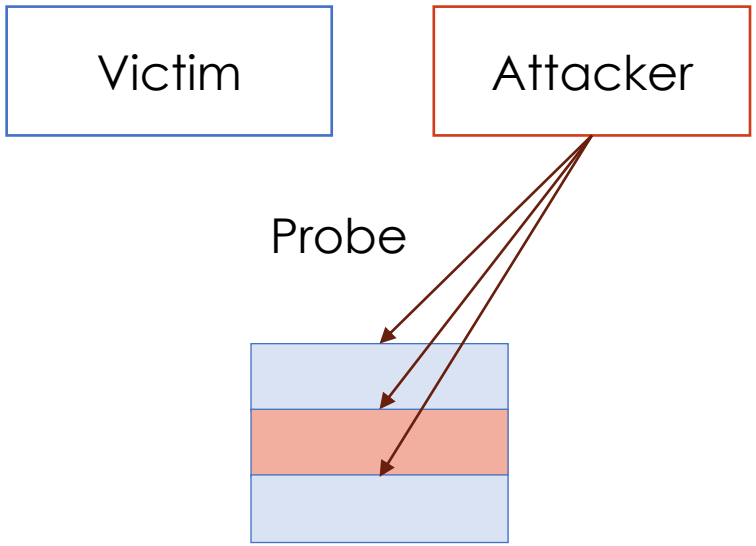


Don't Mesh Around
[USENIX'22]

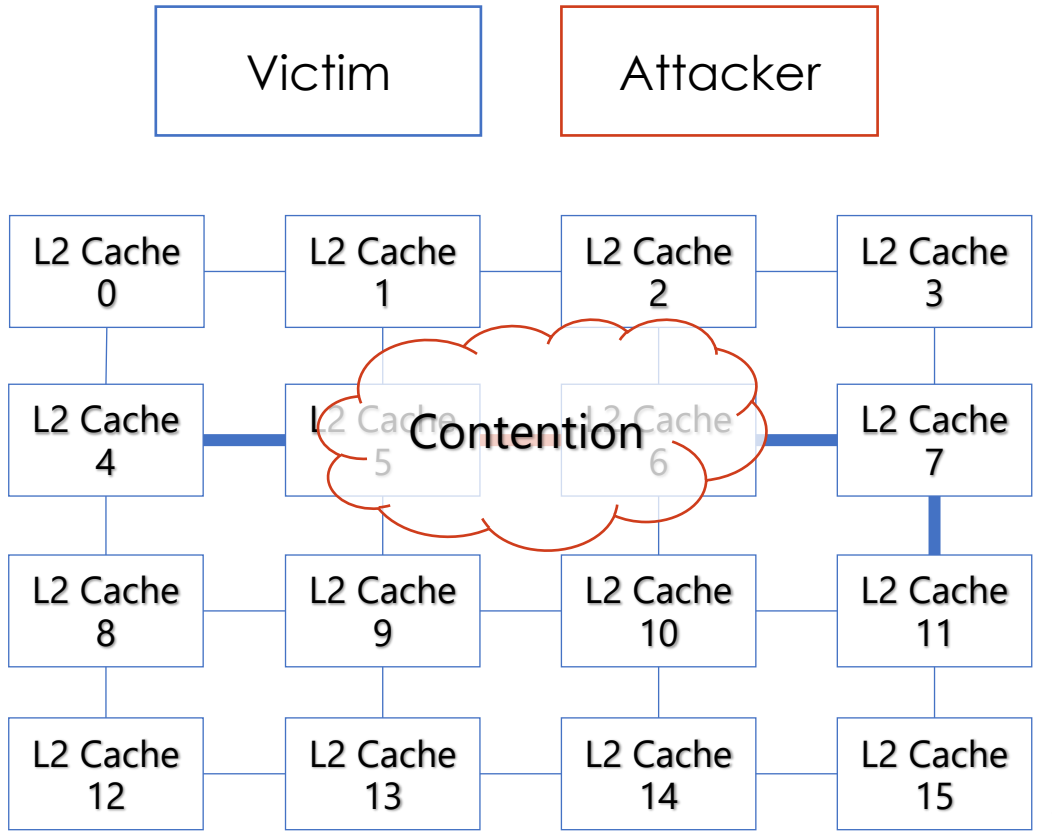


Existing Cache Side-Channel Attacks

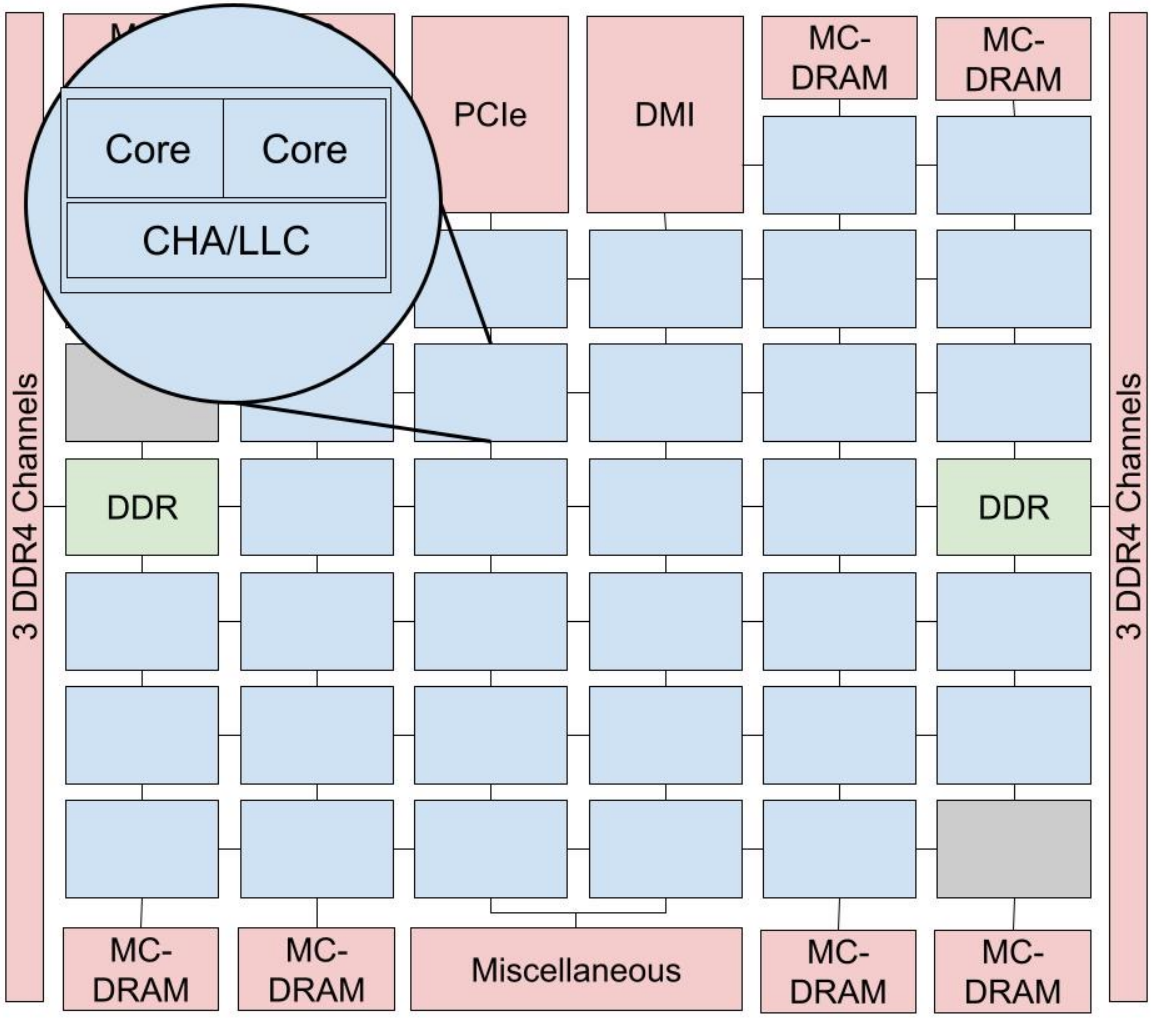
Prime + Probe
[Crypto'06]



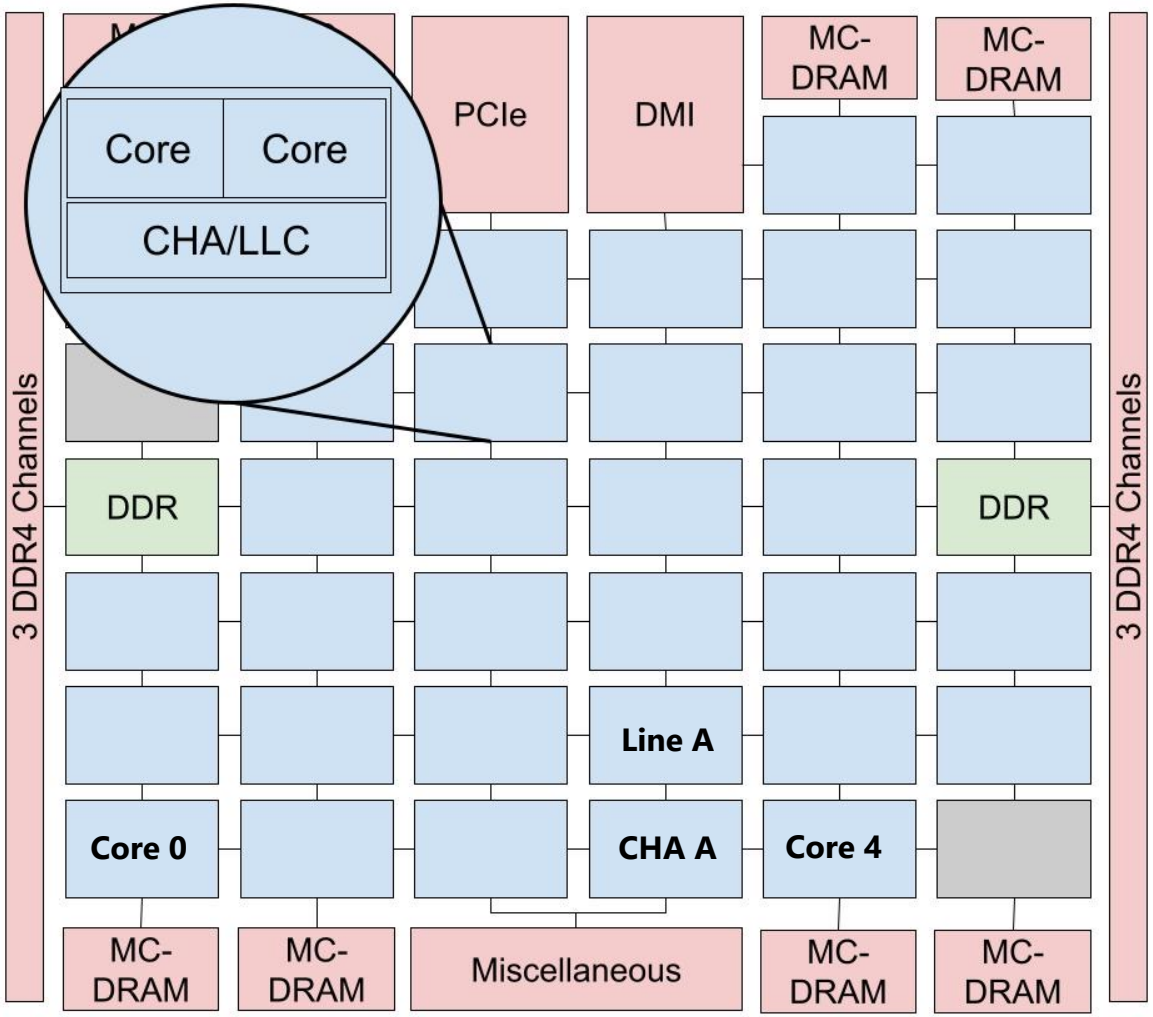
Don't Mesh Around
[USENIX'22]



Target Architecture – Intel Xeon Phi



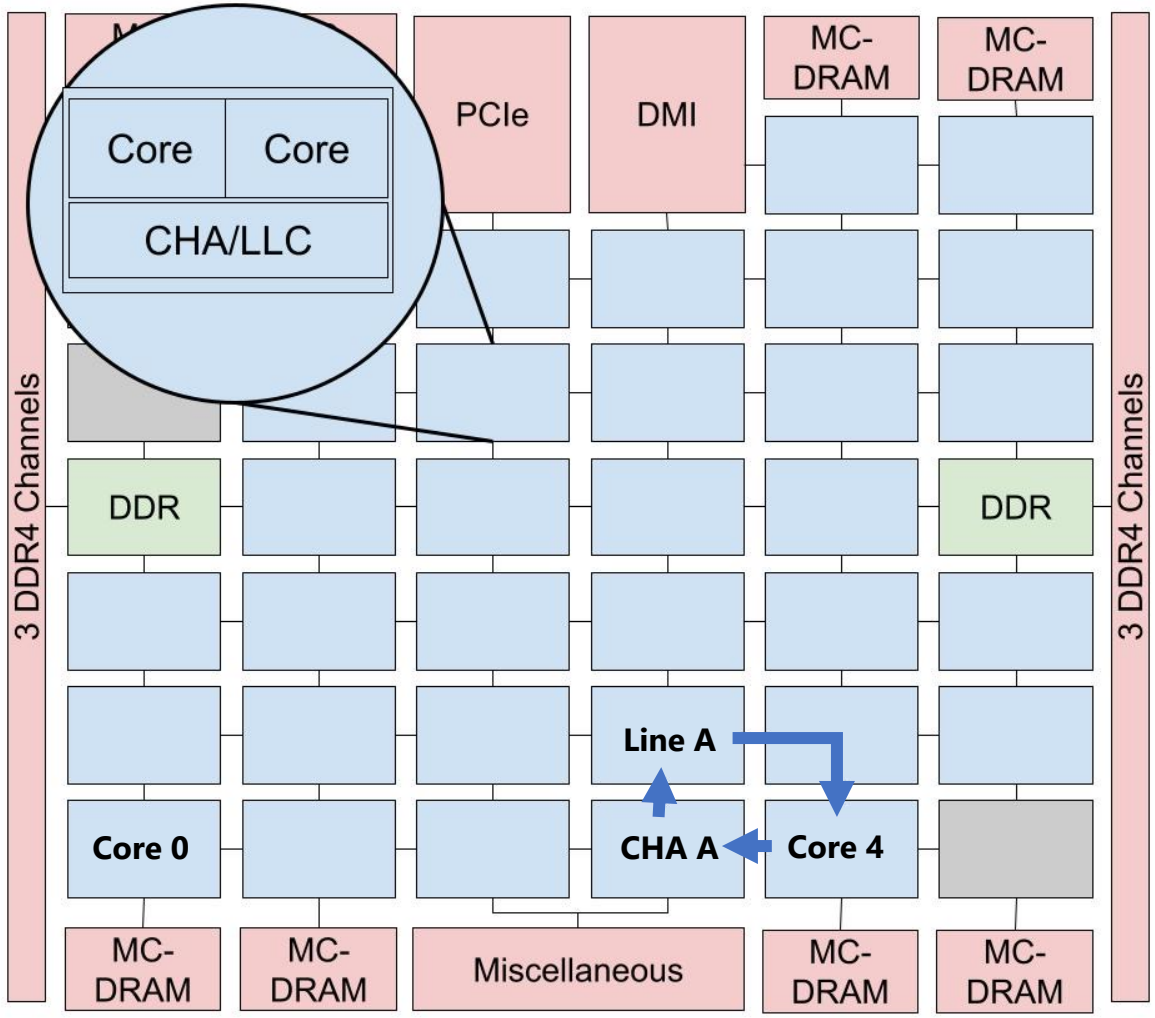
Target Architecture – Intel Xeon Phi



CHA = Caching and Homing Agent

- Part of the Tag Directory
- Tracks Locations

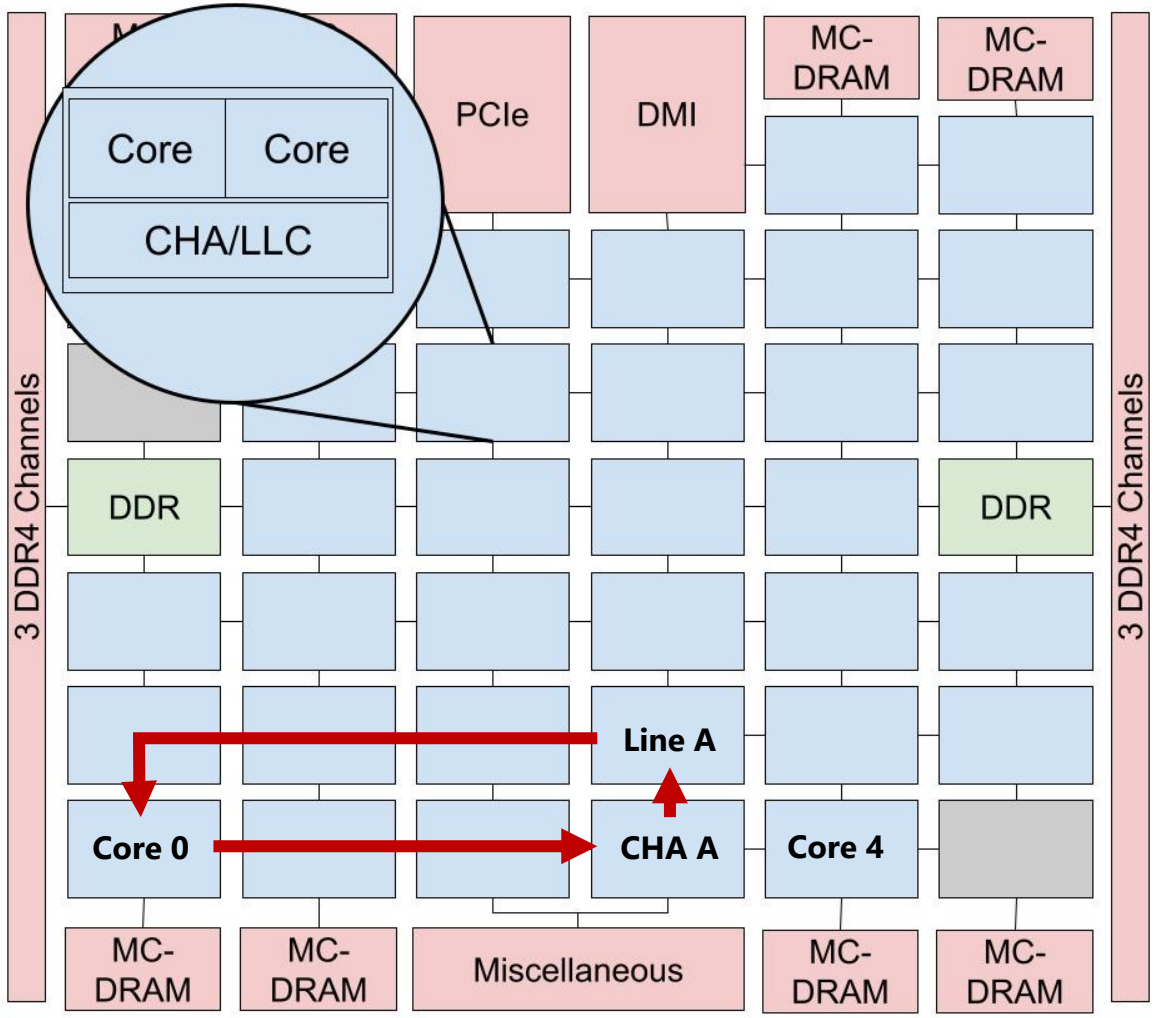
Target Architecture – Intel Xeon Phi



CHA = Caching and Homing Agent
- Part of the Tag Directory
- Tracks Locations

Core 4 requests Line A
- Request Goes to CHA A
- CHA Forwards Request to LLC hosting Line A
- LLC hosting Line A sends Response to Core 4

Target Architecture – Intel Xeon Phi

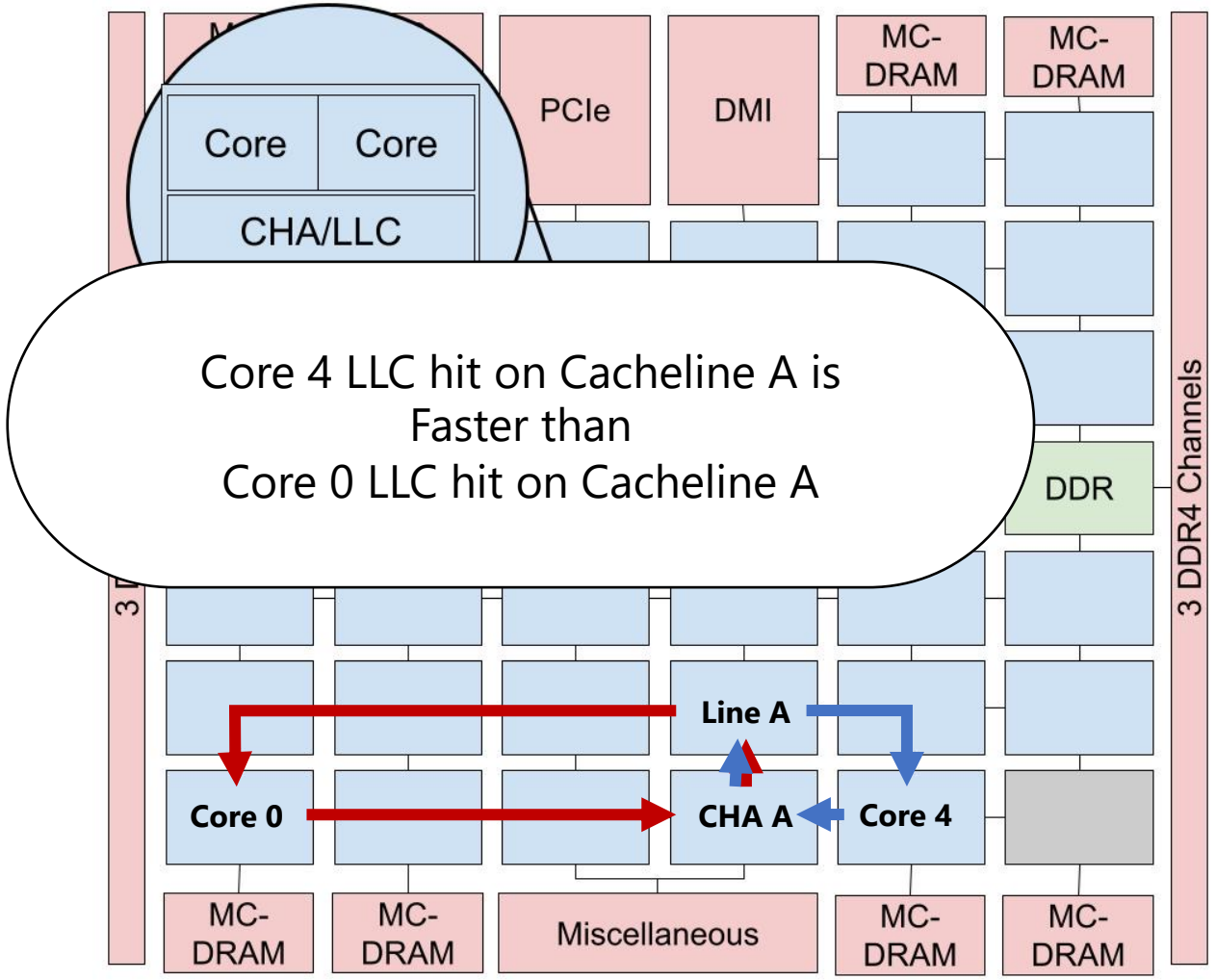


CHA = Caching and Homing Agent
- Part of the Tag Directory
- Tracks Locations

Core 4 requests Line A
- Request Goes to CHA A
- CHA Forwards Request to LLC hosting Line A
- LLC hosting Line A sends Response to Core 4

Core 0 requests Line A
- Request Goes to CHA A
- CHA Forwards Request to LLC hosting Line A
- LLC Hosting Line A sends Response to Core 0

Target Architecture – Intel Xeon Phi

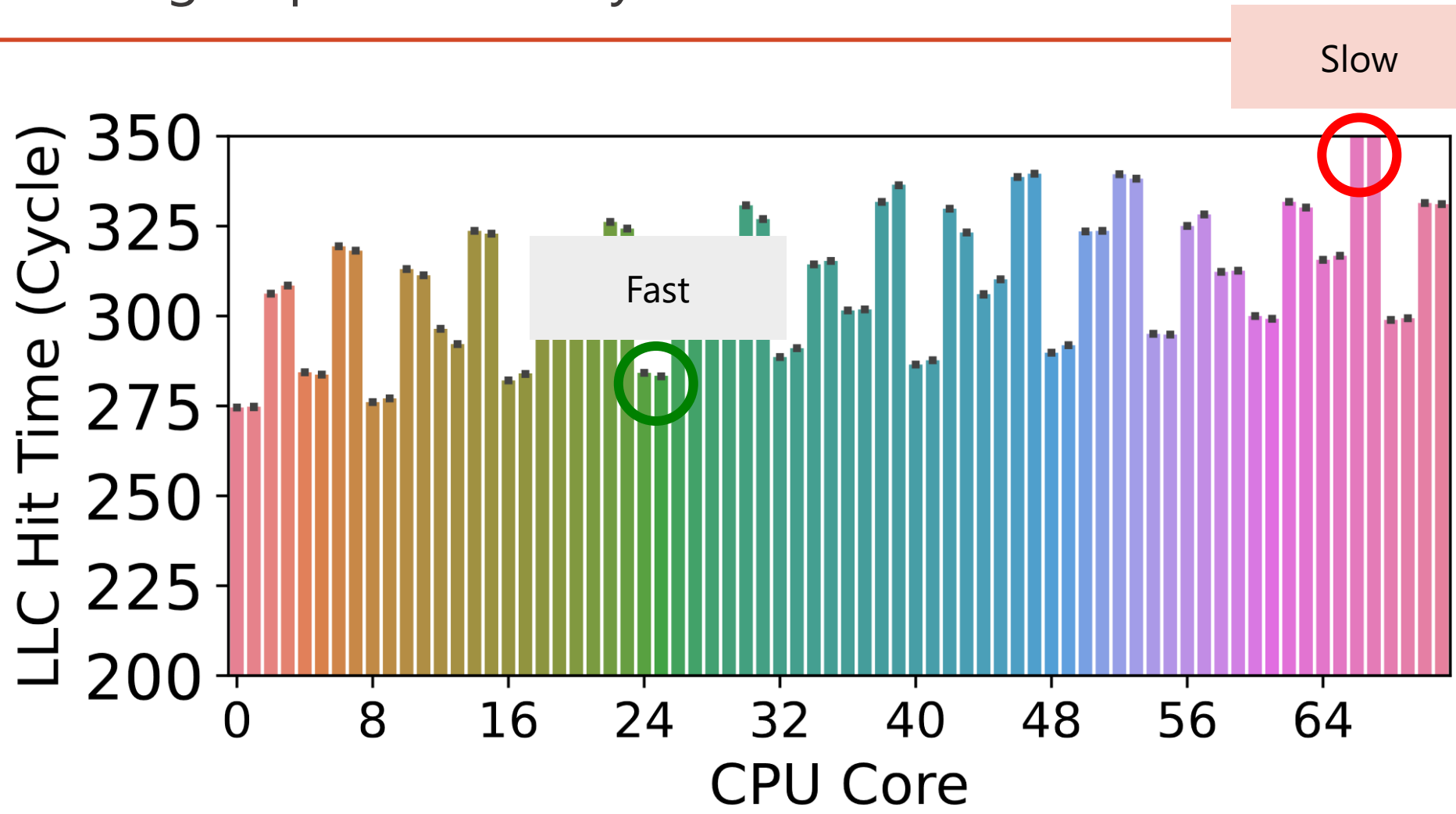


CHA = Caching and Homing Agent
- Part of the Tag Directory
- Tracks Locations

Core 4 requests Line A
- Request Goes to CHA A
- CHA Forwards Request to LLC hosting Line A
- LLC hosting Line A sends Response to Core 4

Core 0 requests Line A
- Request Goes to CHA A
- CHA Forwards Request to LLC hosting Line A
- LLC Hosting Line A sends Response to Core 0

LLC Hit Timing Depends on Physical Location



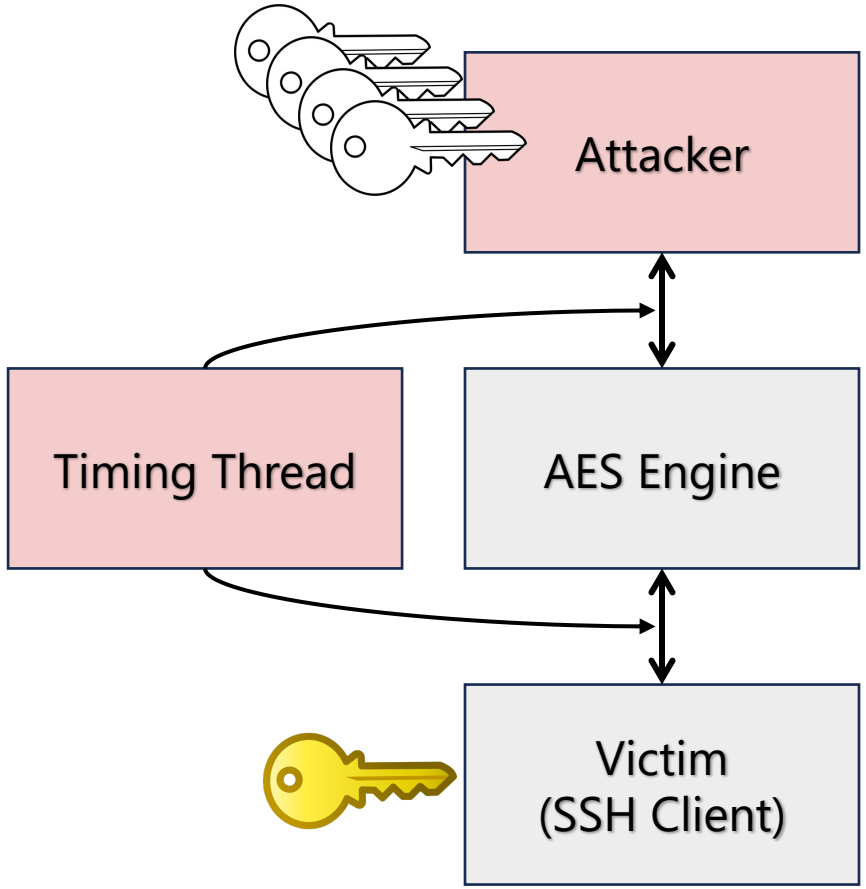
Attack on Intel Xeon Phi : Setup

Victim

- AES Decrypt
- Contains Secret Key
- Accesses Near/Far Tile based on Secret

Attacker

- Observe timing of Victim
- Does not have access to Secret
- Can run multiple iterations



Vulnerable Access Patterns in AES

AES has many decryption tables (Td tables) for improving performance

By default, no-asm AES use these tables now

Commit

aes: make the no-asm constant time code path not the default

```
After OMC and OTC discussions, the 95% performance loss resulting from
the constant time code was deemed excessive for something outside of
our security policy.

The option to use the constant time code exists as it was in OpenSSL 1.1.1.

Reviewed-by:
(Merged from #17600)
```

openssl-3.0 (#16786) + openssl-3.1

Vulnerable Access Patterns in AES

AES has many decryption tables (Td tables) for improving performance

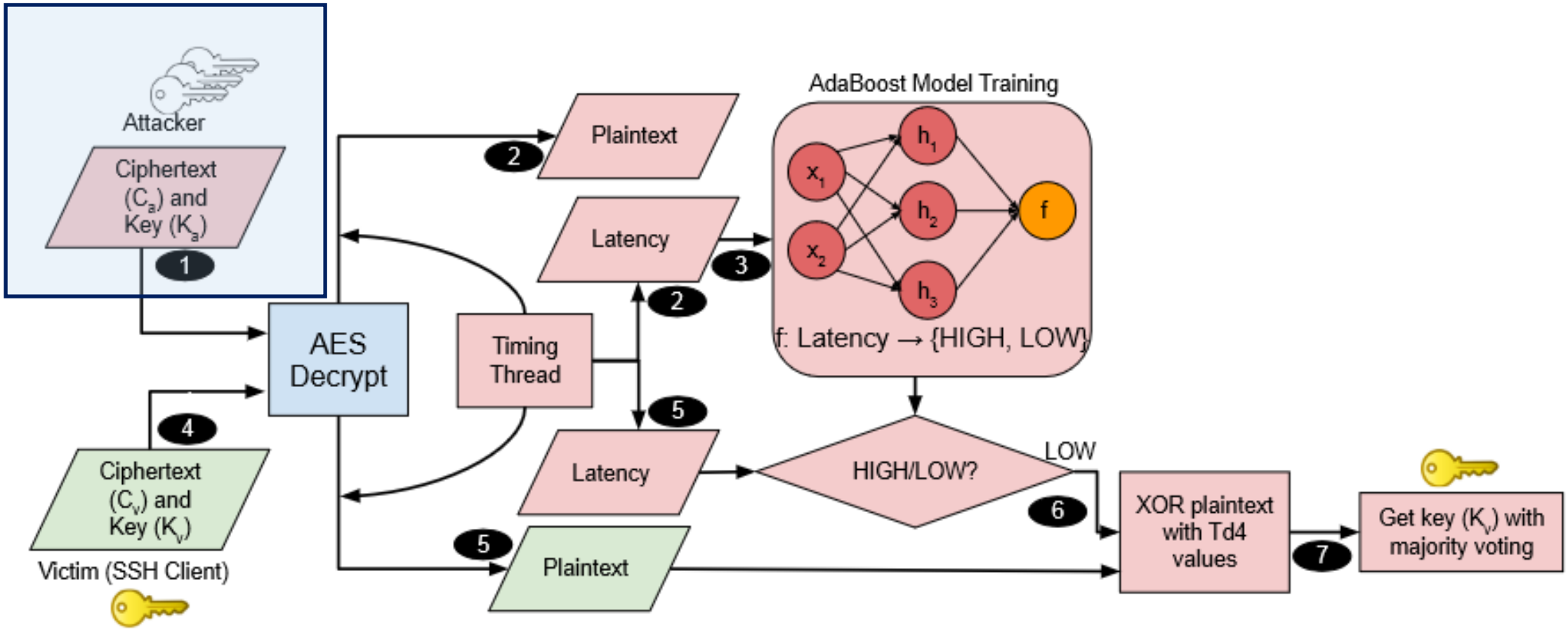
By default, no-asm AES use these tables now

Last round of decryption use part of secret key `rk[0]` and has secret dependent memory accesses

Output of last round AES decrypt is plaintext

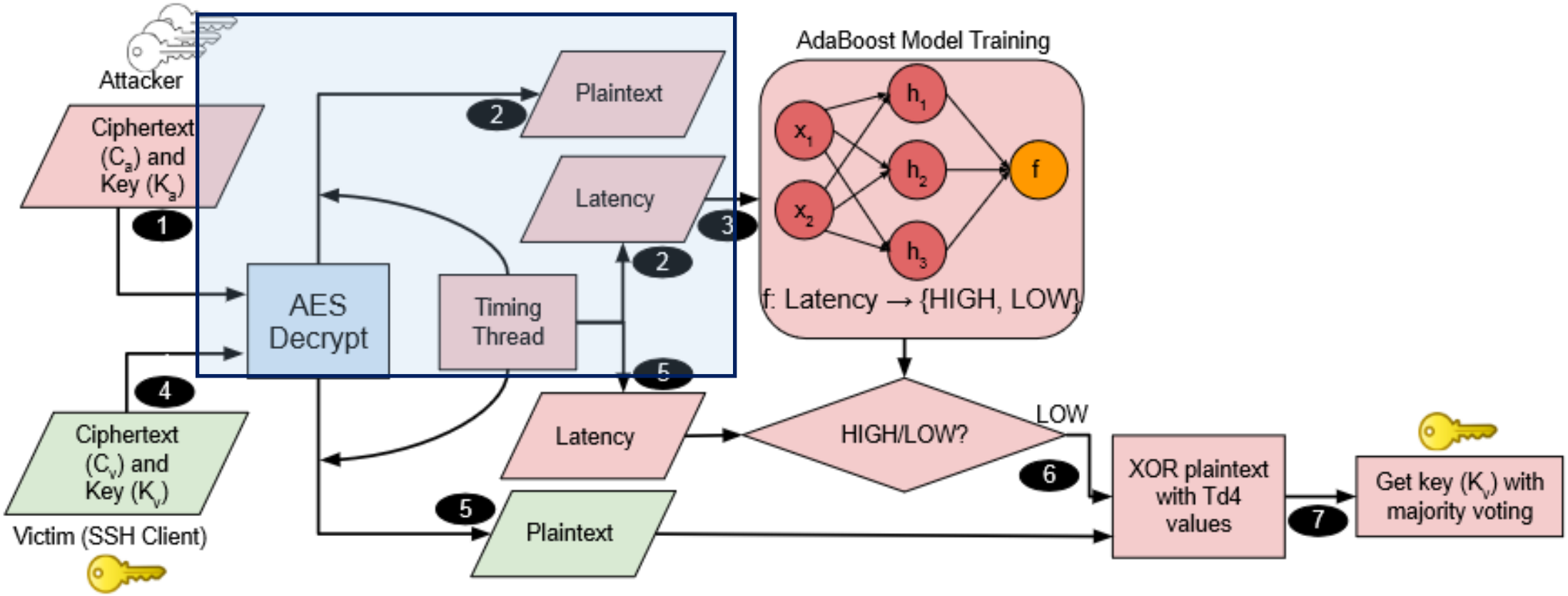
```
/* The last round */
s0 =
    ((u32)Td4[(t0 >> 24)       ] << 24) ^
    ((u32)Td4[(t3 >> 16) & 0xff] << 16) ^
    ((u32)Td4[(t2 >>  8) & 0xff] <<  8) ^
    ((u32)Td4[(t1      ) & 0xff])       ^
    rk[0];
PUTU32(out, s0);
s1 =
    ((u32)Td4[(t1 >> 24)       ] << 24) ^
    ((u32)Td4[(t0 >> 16) & 0xff] << 16) ^
    ((u32)Td4[(t3 >>  8) & 0xff] <<  8) ^
    ((u32)Td4[(t2      ) & 0xff])       ^
    rk[1];
PUTU32(out +  4, s1);
```

Attack Steps: Generate Keys



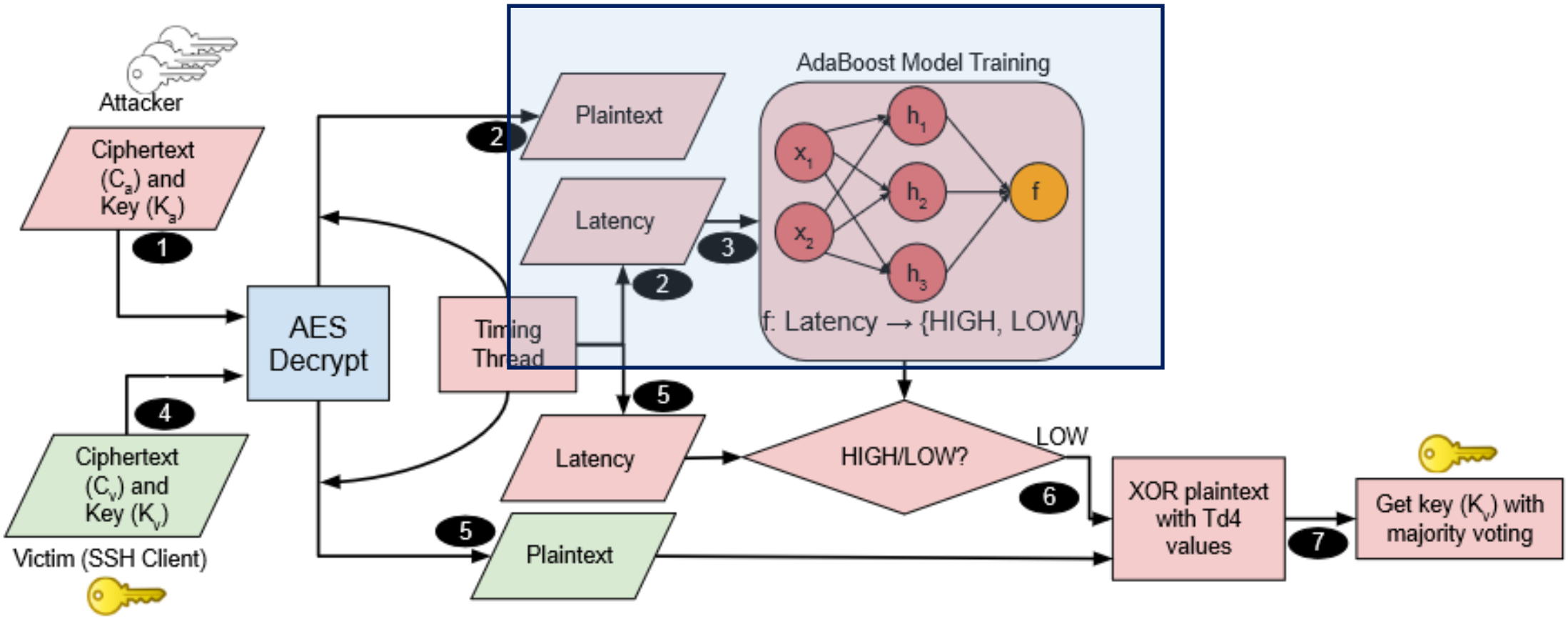
Attack Steps

Attack Steps: Measure Known Data



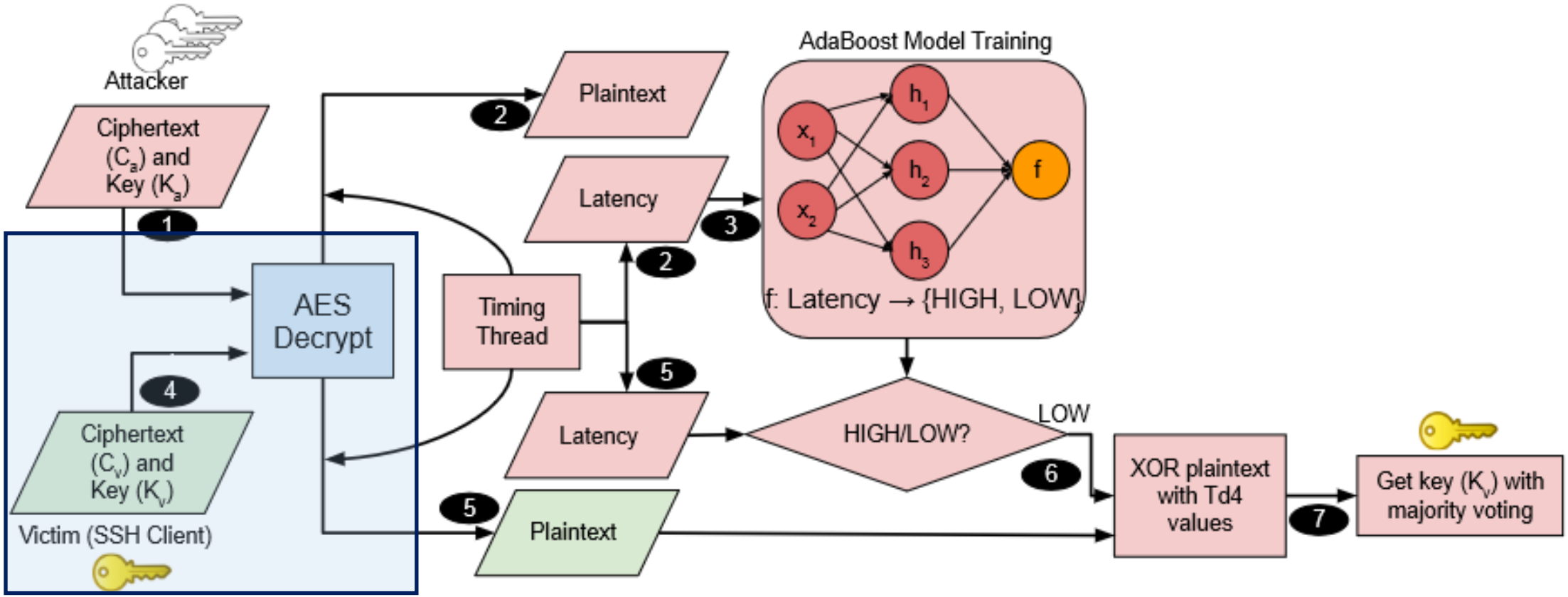
Attack Steps

Attack Steps: Train AdaBoost



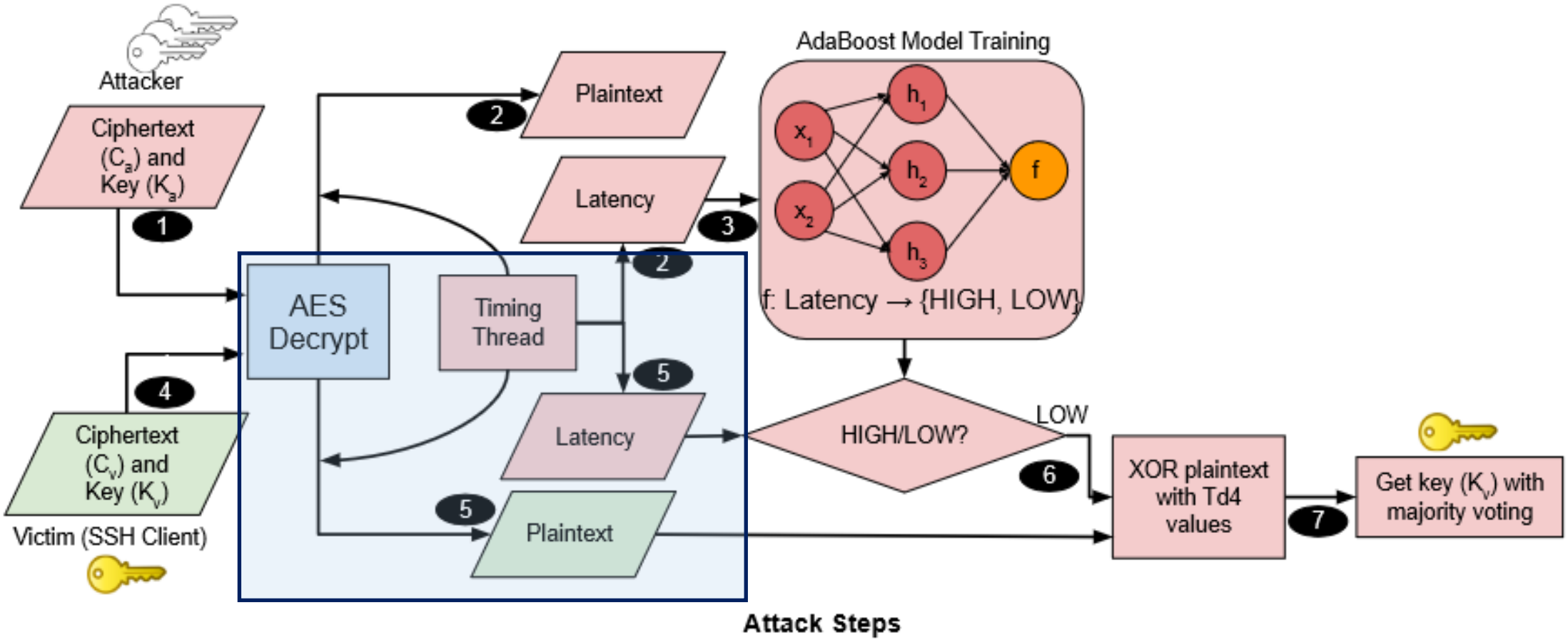
Attack Steps

Attack Steps: Allow Victim Access

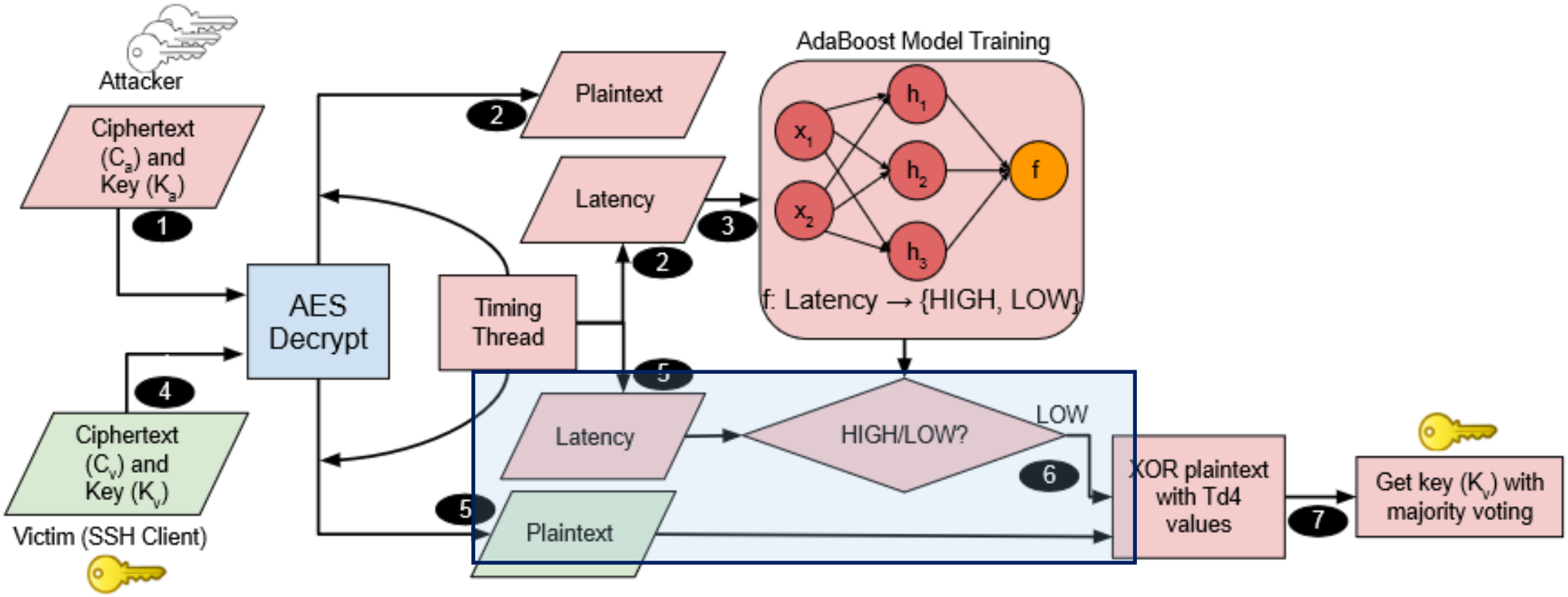


Attack Steps

Attack Steps: Measure Victim Access Time

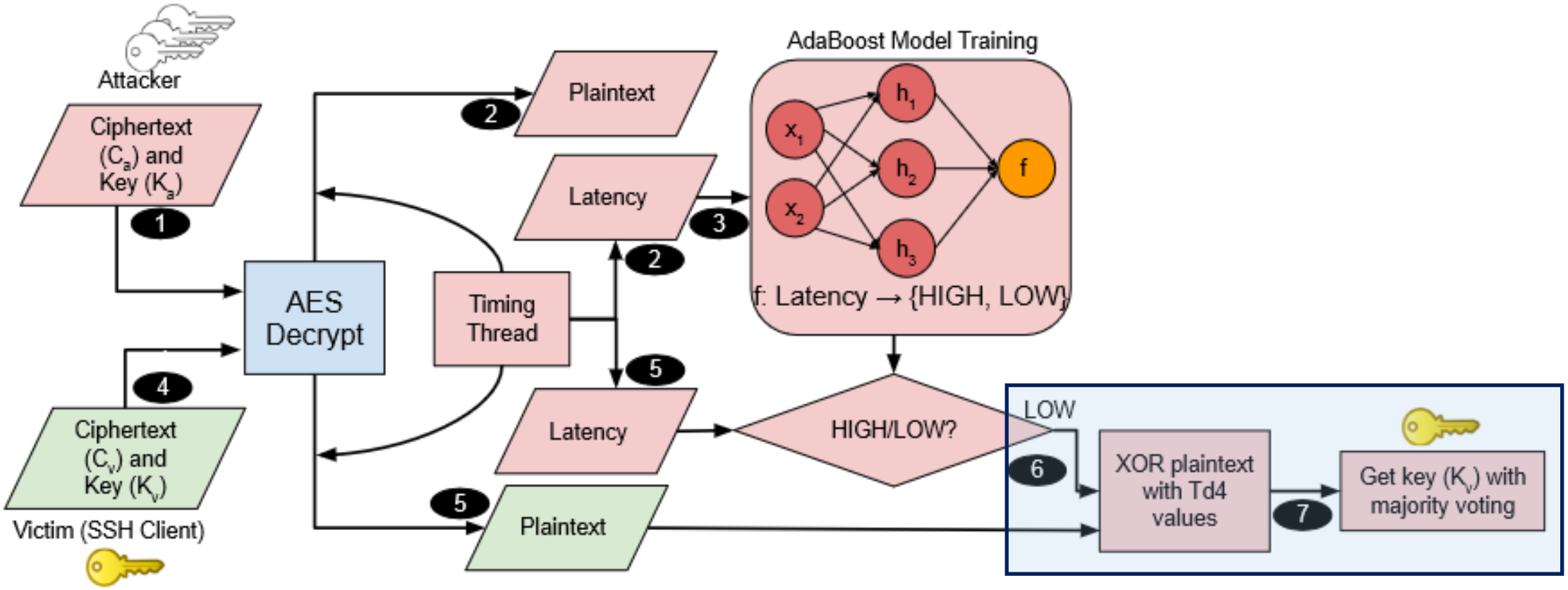


Attack Steps: Classify Victim Access Time



Attack Steps

Attack Steps: Recover Key



Attack Steps

Attack Challenges: Fine Grained Timing

Decrypt function contains many memory accesses

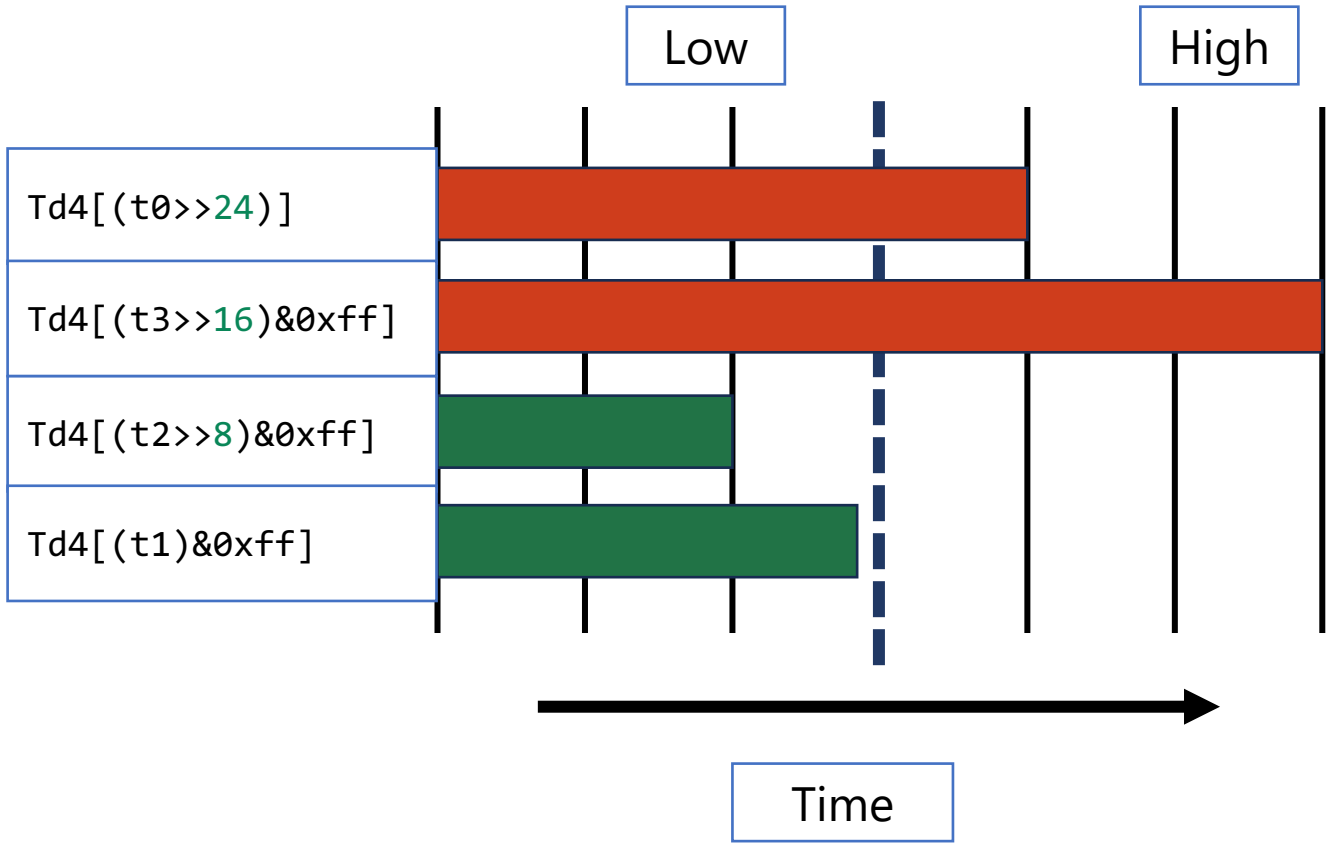
- Many `Td` table accesses are made
- End-to-end timing of Decrypt contains a lot of noise

Fine Grained Timing utilizes access to shared buffer

- Allows more precise measurement of LLC Hit Latency
- Only monitor accesses to `Td4` table
- Access the unprotected buffer (`out` buffer)

```
/* The last round */
s0 =
    ((u32)Td4[(t0 >> 24)       ] << 24) ^
    ((u32)Td4[(t3 >> 16) & 0xff] << 16) ^
    ((u32)Td4[(t2 >>  8) & 0xff] <<  8) ^
    ((u32)Td4[(t1      ) & 0xff])       ^
    rk[0];
PUTU32(out, s0);
s1 =
    ((u32)Td4[(t1 >> 24)       ] << 24) ^
    ((u32)Td4[(t0 >> 16) & 0xff] << 16) ^
    ((u32)Td4[(t3 >>  8) & 0xff] <<  8) ^
    ((u32)Td4[(t2      ) & 0xff])       ^
    rk[1];
PUTU32(out +  4, s1);
```

Attack Challenges : Longest Memory Accesses Hide Faster Accesses




```
/* The last round */
s0 =
    ((u32)Td4[(t0 >> 24)       ] << 24) ^
    ((u32)Td4[(t3 >> 16) & 0xff] << 16) ^
    ((u32)Td4[(t2 >>  8) & 0xff] <<  8) ^
    ((u32)Td4[(t1      ) & 0xff])       ^
    rk[0];
PUTU32(out, s0);
s1 =
    ((u32)Td4[(t1 >> 24)       ] << 24) ^
    ((u32)Td4[(t0 >> 16) & 0xff] << 16) ^
    ((u32)Td4[(t3 >>  8) & 0xff] <<  8) ^
    ((u32)Td4[(t2      ) & 0xff])       ^
    rk[1];
PUTU32(out +  4, s1);
```

Experiment Setup

Configuration Parameters

- Intel Xeon Phi 7290 CPU
- Cluster set to All-to-all configuration
- MCDRAM set as part of the memory

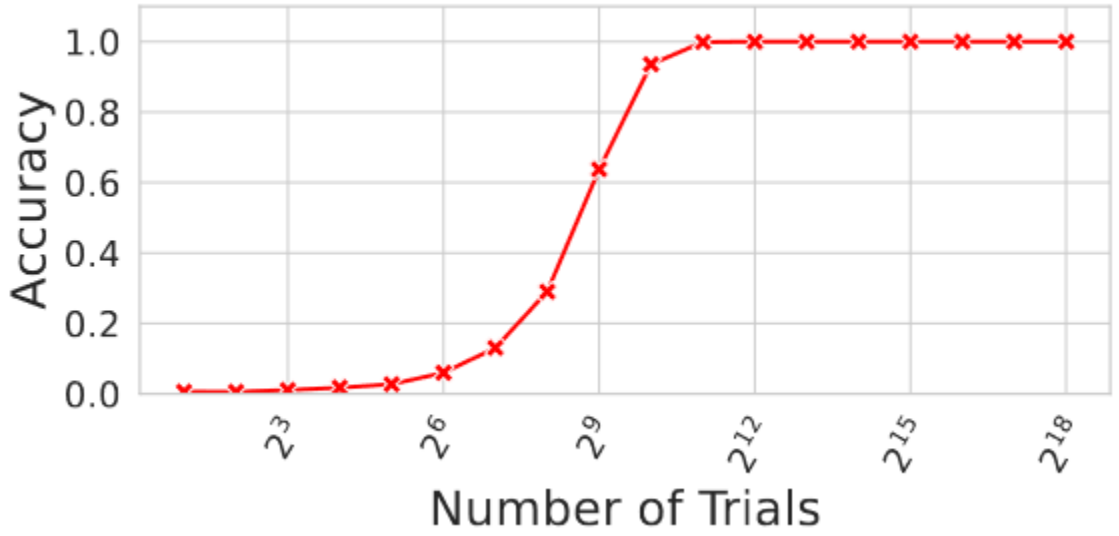
Side Channel Attack

- Number of different plaintexts [2, 2^{20}]
- Number of trials for same plaintext [1, 100]

Covert Channel Attack

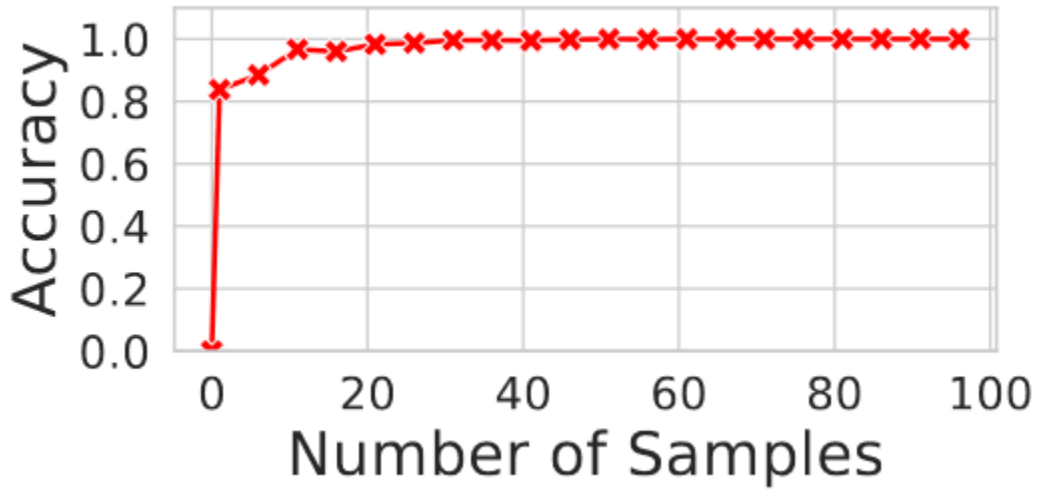
- Payload Size [2^{0}, 2^{17}]

Side-Channel Results: Key Extraction Accuracy



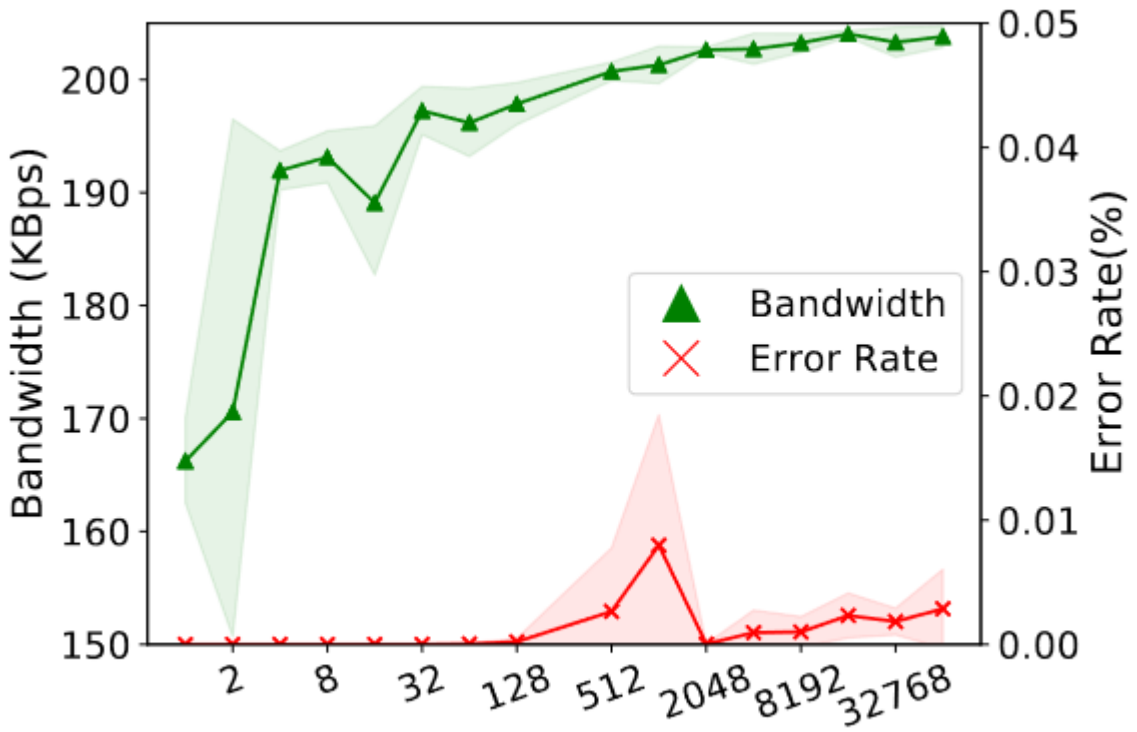
We can extract 4 bytes of any random key with 100% accuracy by using only ≃ 4000 trials.

Side-Channel Attack Result: ML Model Accuracy



100% accuracy for >40 samples for each plaintext using AdaBoost

Covert Channel Bandwidth & Error Rates



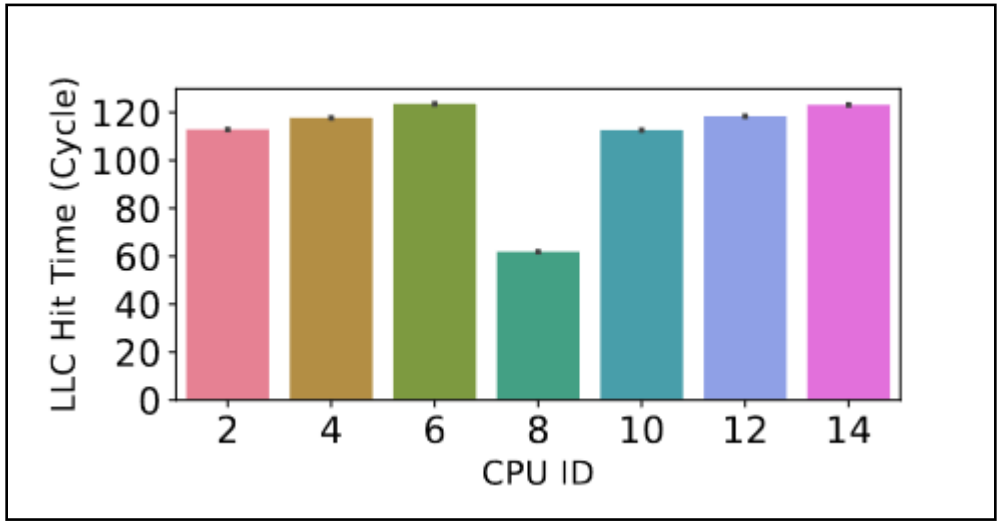
Max 0.02% error rate with 205 KBPS bandwidth

Generalizability: Beyond Xeon Phi

Intel Xeon SP Scalable and Intel Core processors have mesh network

Similar latency distribution found in Intel 10700k with 16 cores from Comet Lake processor family

Similar vulnerabilities may exist in other mesh network processors



Conclusions

- Implemented Covert Channel Intel Xeon Phi 7290
  - 205KBPS data bandwidth & 0.2% Error Rate
- Implemented Side-channel in Intel Xeon Phi 7290
  - 4000 trials to get 4 bytes of AES key with 100% accuracy
- Other processors with mesh network might be vulnerable
- Other cryptographic algorithms with similar T-table might be vulnerable

Artifacts Available: https://github.com/farabimahmud/aok_ae

Farabi Mahmud
Texas A&M University
farabi@tamu.edu
www.farabimahmud.com

Extra Slides


Allocate Large array to LLC
Let Victim Make LLC Hit
Measure Latency of LLC Hit
Classify LLC Hit to HIGH vs LOW

Using Cache replacement	Attacker run on a separate core
	Bring data to attacker's core before the Victim is allowed to execute
Using PREFETCHW Instruction	Attacker run on a separate core
	Invalidate L1D cache of the victim core with PREFETCHW from remote core

Two Key Requirements

Vulnerable T-table implementation that access different memory location based on secret key

Fine-grained timing channel that allows running timer thread based on a shared variable with AES engine

Out of Order CPUs can issue multiple load into pipeline

Latency of multiple loads can overlap

Highest latency load will dominate overall latencies

```
/* The last round */
s0 =
     ((u32)Td4[(t0 >> 24)       ] << 24) ^
     ((u32)Td4[(t3 >> 16) & 0xff] << 16) ^
     ((u32)Td4[(t2 >>  8) & 0xff] <<  8) ^
     ((u32)Td4[(t1      ) & 0xff])       ^
      rk[0];
PUTU32(out, s0);
s1 =
     ((u32)Td4[(t1 >> 24)       ] << 24) ^
     ((u32)Td4[(t0 >> 16) & 0xff] << 16) ^
     ((u32)Td4[(t3 >>  8) & 0xff] <<  8) ^
     ((u32)Td4[(t2      ) & 0xff])       ^
     rk[1];
PUTU32(out +  4, s1);
```

Problem

- Attacker can only monitor the timing of Decrypt function
- Decrypt contains multiple rounds of Td4 table usages

Solution:

- Use multiple trials and AdaBoost algorithm to decide

1. Attacker Generate N ciphertext and key pairs (Ca and Ka)
2. Attacker thread use AES Decryption to get plaintext
3. Timer thread measures latency during each decryption
4. Attacker classify labels with LOW and HIGH based on latency
5. Attacker Train AdaBoost Model with these labels

Classifying Victim's Accesses

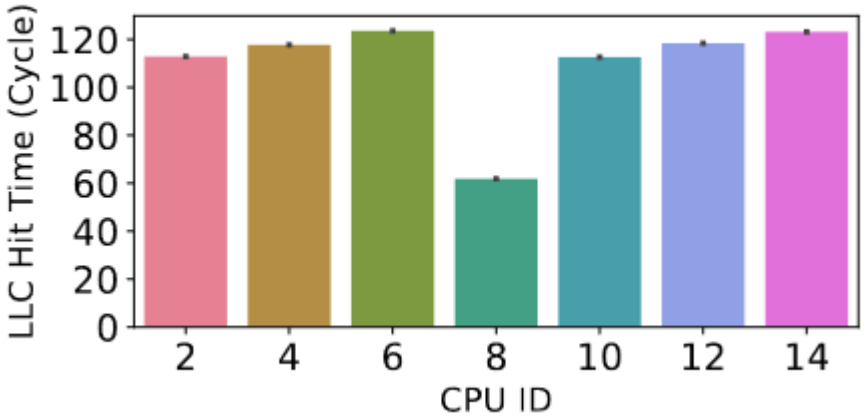
4. Victim use the AES engine to decrypt ciphertext Cv with its own secret key Kv
5. Timer monitors the victim accesses and measure latency
6. Latency is predicted to be HIGH/LOW using AdaBoost model
7. If the latency is classified as LOW, plaintext can be XORed with Td4 values associated with LOW label

Repeat Step 4-7 multiple times and take majority voting

Intel Xeon SP Scalable and Intel Core processors have mesh network

Similar latency distribution found in Intel 10700k with 16 cores from Comet Lake processor family

Similar vulnerabilities may exist in other mesh network processors



Similar T-table implementation in many cryptographic software

Recent AES version has disabled patch which would prevent this attack

Camellia & ARIA also have similar structure

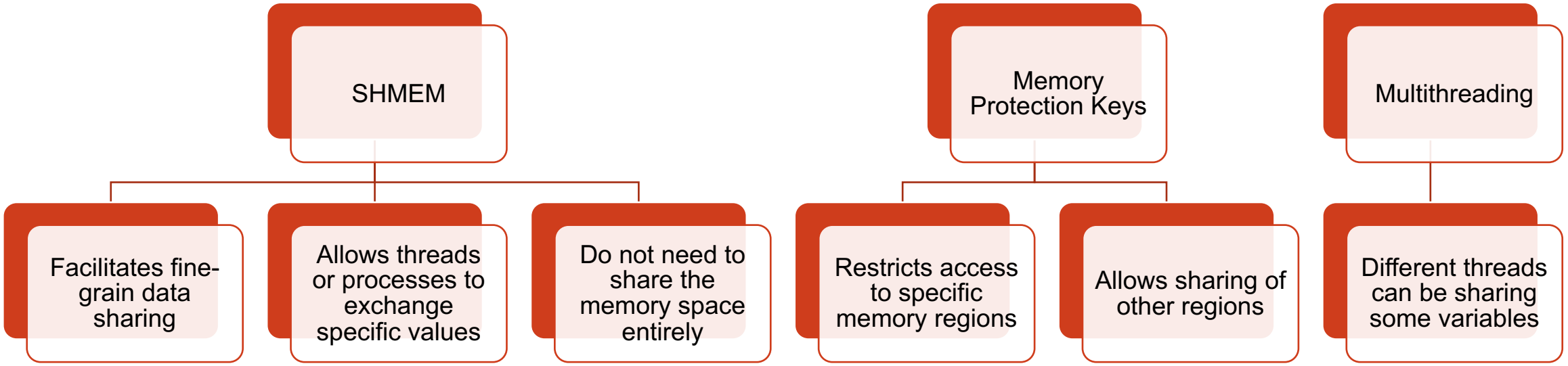
```
static void sl1(ARIA_u128 *o, const ARIA_u128 *x, const ARIA_u128 *y)
{
    unsigned int i;
    for (i = 0; i < ARIA_BLOCK_SIZE; i += 4) {
        o->c[i    ] = sb1[x->c[i    ] ^ y->c[i    ]];
        o->c[i + 1] = sb2[x->c[i + 1] ^ y->c[i + 1]];
        o->c[i + 2] = sb3[x->c[i + 2] ^ y->c[i + 2]];
        o->c[i + 3] = sb4[x->c[i + 3] ^ y->c[i + 3]];
    }
}
```

Decrypt function Contains many memory accesses

- Many of these accesses are made to Td table
- End-to-end timing of Decrypt contains noise

Fine-grained Timing utilizes access to shared buffer

- Allows more precise measurement of LLC Hit Latency
- Only monitor accesses to Td4 table


SHMEM
Facilitates fine-grain data sharing
Allows threads or processes to exchange specific values
Do not need to share the memory space entirely
Memory Protection Keys
Restricts access to specific memory regions
Allows sharing of other regions
Multithreading
Different threads can be sharing some variables

Software Targets

Hardware Platforms

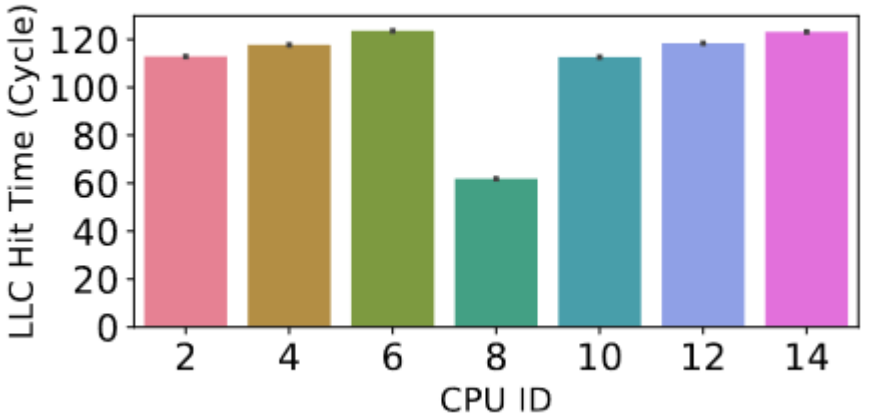
Configurations

- MCDRAM
- Cluster

Intel Xeon SP Scalable and Intel Core processors have mesh network

Similar latency distribution found in Intel 10700k with 16 cores from Comet Lake processor family

Similar vulnerabilities may exist in other mesh network processors



MCDRAM configuration will impact LLC Hit Latency

- Cache Mode
- Flat Mode
- Hybrid Mode

We have used Flat Mode

Three available modes

- All-to-all
- Quadrant/Hemisphere Mode
- Sub NUMA Cluster (SNC-2/SNC-4)

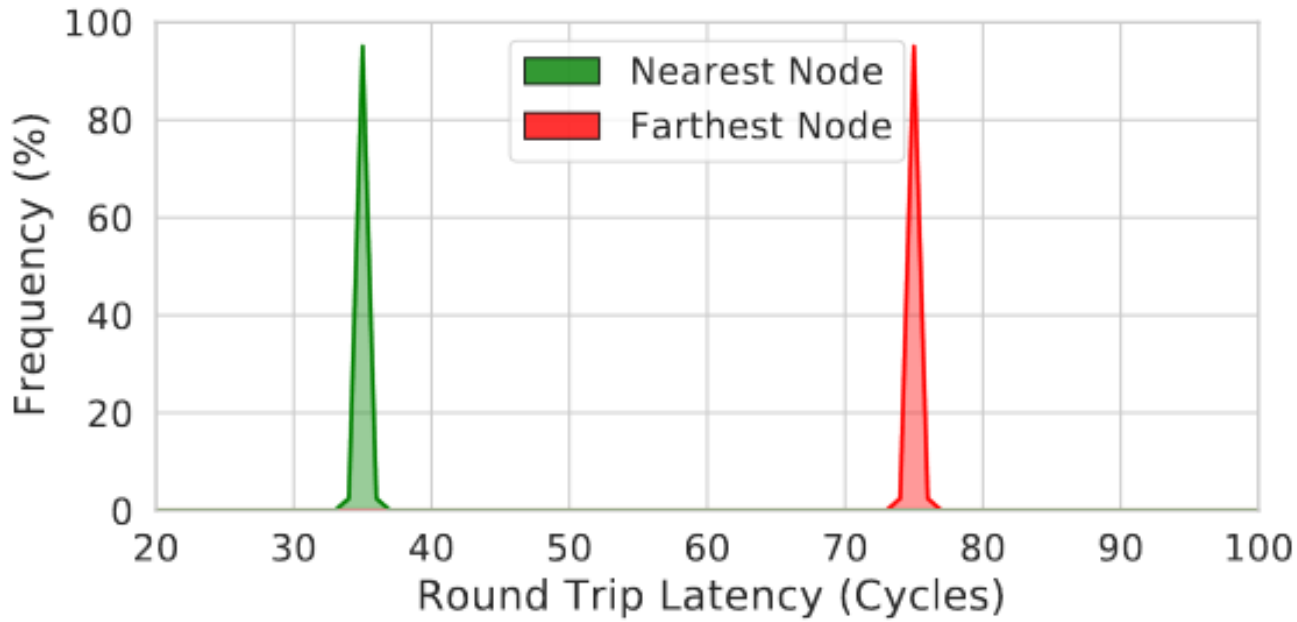
We have used All-to-all cluster mode

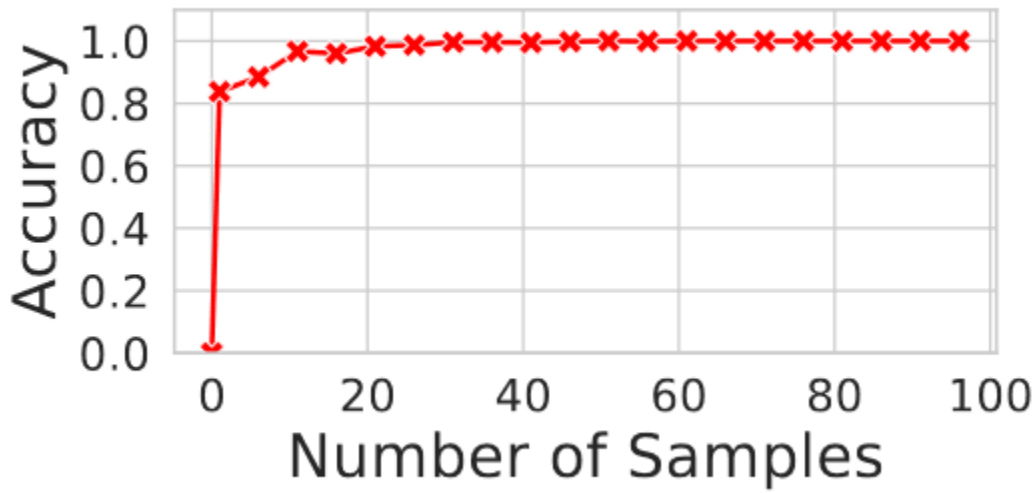
Problem

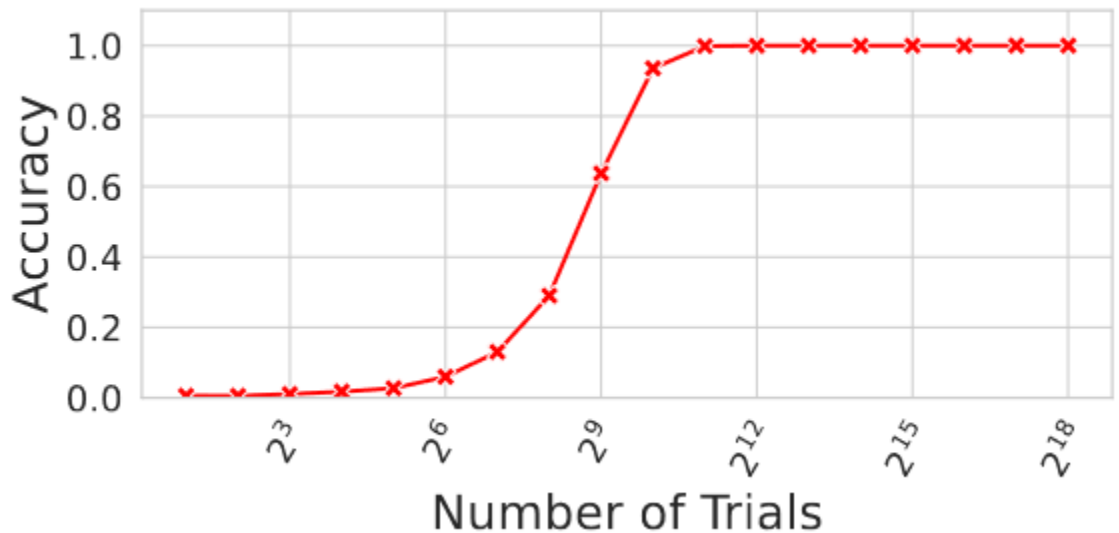
- Multiple loads overlap within the region of interest
- Measured latency is affected by overlapped loads

Solution

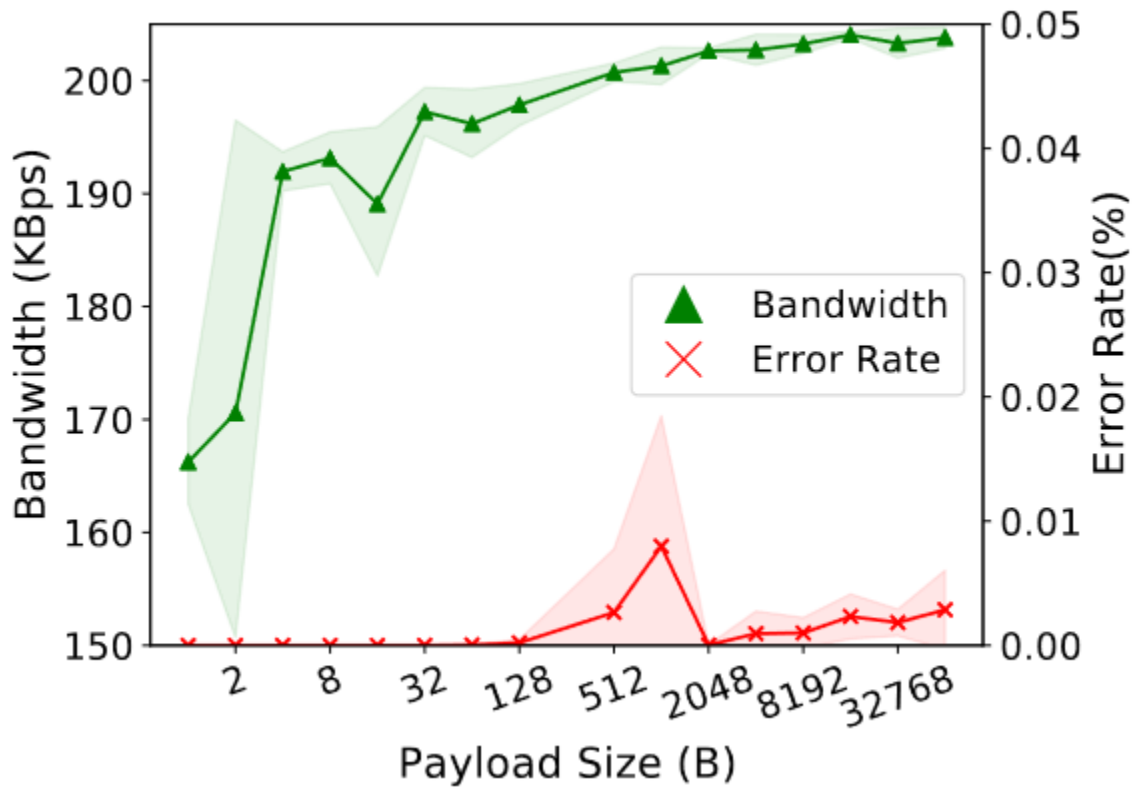
- Take multiple samples
- Use AdaBoost algorithm to classify samples


Nearest Node
Farthest Node
Frequency (%)
100
80
60
40
20
0
20
30
40
50
60
70
80
90
100
Round Trip Latency (Cycles)


Accuracy
1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100
Number of Samples



We can extract 4 bytes of any random key with 100% accuracy by using only ≃ 4000 trials.


Bandwidth (KBps)
Error Rate(%)
Payload Size (B)
Bandwidth
Error Rate
200
190
180
170
160
150
0.05
0.04
0.03
0.02
0.01
0.00
2
8
32
128
512
2048
8192
32768

Distance-based NUCA Cache Side-Channel Attack

Implemented in Gem5 Simulator

- 95% Accuracy even with Rodinia background application

Implemented Covert Channel Intel Xeon Phi 7290

- 205KBPS data bandwidth
- 0.2% Error Rate

Implemented Side-channel in Intel Xeon Phi 7290

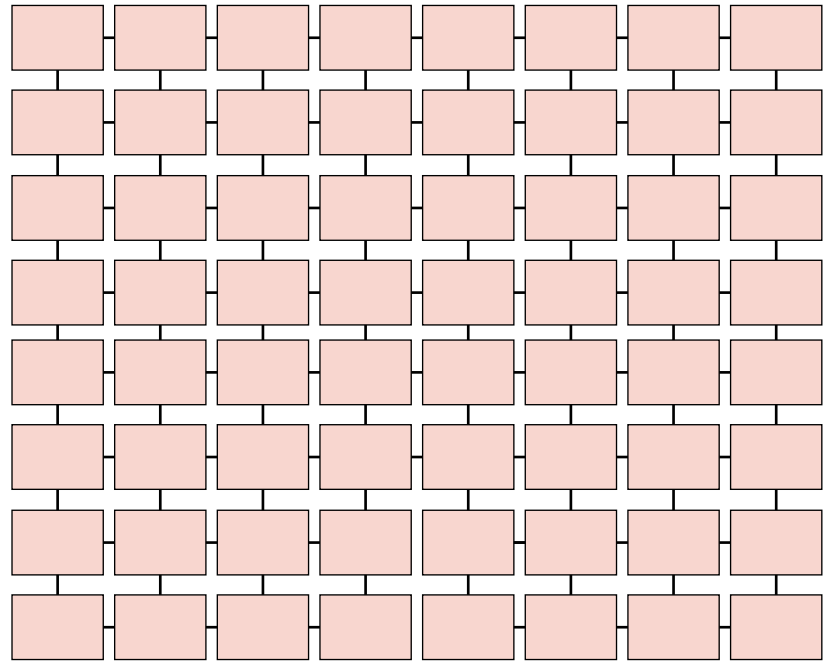
- 4000 trials to get 4 bytes of AES key with 100% accuracy

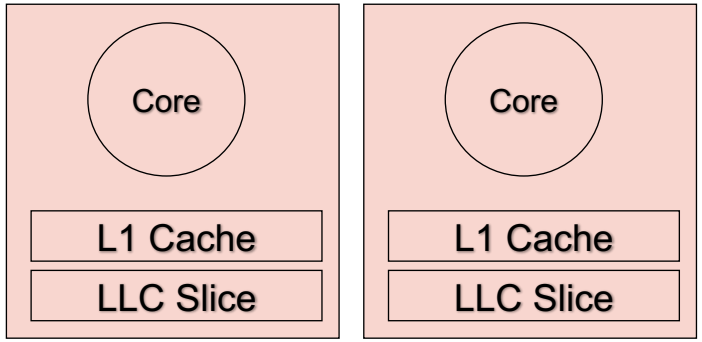
Other processors with mesh network might be vulnerable

Other cryptographic algorithms with similar T-table might be vulnerable

ATTACK EXAMPLE ON GEM5 SIMULATOR

Architecture	8x8 Cores
	Distributed Directory
	2D Mesh Network
Each tile contains	A Core
	Private L1I Cache
	Private L1D Cache
	Shared LLC (L2) Bank
L1D Cache	2-way associative
	4kB
	LRU Replacement
LLC	8-way associative
	2MB
	Distributed across 64 Tiles





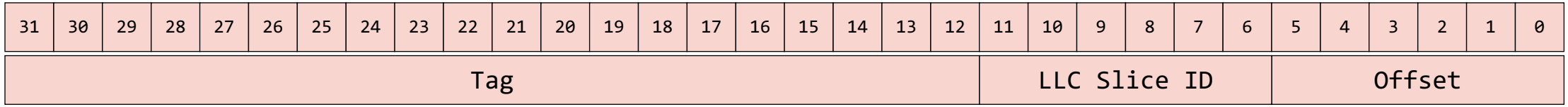
Step 1.
Identify vulnerable access pattern in Victim function

Step 2.
Prepare for L1 Miss but LLC Hit

Step 3.
Allow Victim to access entries that would be LLC hit

Step 4.
Measure the latency and classify accordingly

Step 1a. Reverse Engineer LLC Slice Selection Function



Step 1b. Determine Addresses belonging to Different LLC Slice

Array Index	Virtual Address	Physical Address	LLC Slice
117 * 64	0x4C7FC0	0XC6FC0	63
118 * 64	0x4C8000	0xC7000	0

ADDR	Slice-L2-L1
	0-0-0

LLC Miss

L1$	Set	Way 0	Way 1
	0	0-0-0	
	1		

LLC	Set	Way 0	Way 1	Way 2	Way 3	Way 4	Way 5	Way 6	Way 7
	0	0-0-0							
	1								

Slice-L2-L1
1-0-0

ADDR

LLC Miss

L1$	Set	Way 0	Way 1
	0	0-0-0	1-0-0
	1		

LLC	Set	Way 0	Way 1	Way 2	Way 3	Way 4	Way 5	Way 6	Way 7
	0	0-0-0	1-0-0						
	1								

ADDR	Slice-L2-L1
	2-0-0

LLC Miss

L1$	Set	Way 0	Way 1
	0	0-0-0	1-0-0
	1		

LLC	Set	Way 0	Way 1	Way 2	Way 3	Way 4	Way 5	Way 6	Way 7
	0	0-0-0	1-0-0	2-0-0					
	1								

Slice-L2-L1
1-0-0

ADDR

LLC Hit

L1$	Set	Way 0	Way 1
	0	2-0-0	3-0-0
	1		

LLC	Set	Way 0	Way 1	Way 2	Way 3	Way 4	Way 5	Way 6	Way 7
	0	0-0-0	1-0-0	2-0-0	3-0-0				
	1								

- Only allow Victim to have one memory access
- Memory Location dependent on Secret Bit

```
void victim(unsigned int mask) {
    uint8_t s = secret & mask;
    if (s == 0) s ^= arr[117 * 64]; // LLC bank 63
    else        s ^= arr[118 * 64]; // LLC bank 0
}
```

- Use RDTSCP to measure latency
- Based on the threshold, we can classify whether its bit 0 or 1

```
// Time the victim function
t1 = __rdtscp(&junk);
victim(mask);
t2 = __rdtscp(&junk) - t1;
// If the bit is 0, the latency > 100
printf("BIT[%d]: %d\n", i, t2 > THRESHOLD? 0:1);
```


Green Region Secret Bit 0
White Region Secret Bit 1
Trip Latency
(Cycles)
100
80
60
40
20
25
30
35
40
45
50
55
60
65
70
75
80
85
90
95
Timeline
> 95 % accuracy
Red Cross Bit Predicted 1
Blue Cross Bit Predicted 0