DATE 22

DESIGN, AUTOMATION & TEST IN EUROPE

14 – 15 March 2022 · on-site event
16 – 23 March 2022 · online event

The European Event for Electronic
System Design & Test

PREFENDER: A Prefetching Defender against Cache Side Channel Attacks as A Pretender

Luyi Li [†] , Jiayi Huang [‡], **Lang Feng** [†] , Zhongfeng Wang [†]

[†] Nanjing University

[‡] University of California, Santa Barbara

Outline

I. Motivation

II. Background

III. PREFENDER Design

IV. Experimental Evaluation

Motivation

- **More and more complex devices expose them to larger attack surfaces**
  - Cloud computing, IoT, etc.

- **Increasing threat of cache side channel attacks**
  - Vulnerabilities in hardware design: Spectre, Meltdown, Foreshadow, etc.

- **Urgent to defeat them effectively and efficiently**

Motivation

- **Related work**
  - **Cache isolation:** DAWG from MICRO2018
  - **Limit speculation:** Conditional Speculation from HPCA19
  - **Stateless speculative buffer:** InvisiSpec from MICRO2018
  - **Noise injection:** Reuse-trap from DAC2020
  - ...
- **Trade-off between security and performance in existing methods**

vs.

+?

- **Can we both enforce security and increase performance?**
  - **Insight:** Effective prefetching can both defend against attacks and save execution time.

Outline

I. Motivation

II. Background

- Cache Side Channel Attack
- Threat Model
- Prefetching

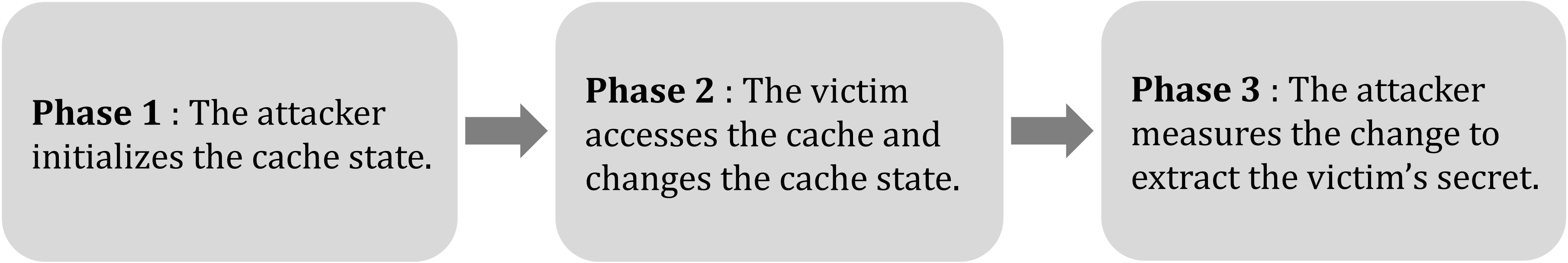
III. PREFENDER Design

IV. Experimental Evaluation

Cache Side Channel Attack

- **Side channel attack is to extract secrets from information inadvertently leaked by a system**
  - Time, Cache, Power, Electromagnetic, etc.

- **Cache side channel attack**
  - Victim leaves side channels in the cache while running.
  - Attacker exploits the cache side channels to extract secrets.



Phase 1 : The attacker initializes the cache state.

Phase 2 : The victim accesses the cache and changes the cache state.

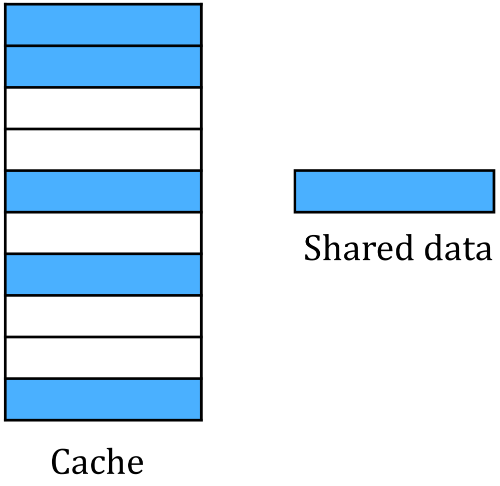
Phase 3 : The attacker measures the change to extract the victim's secret.

e.g. A typical attack flow

Threat Model

- **Cache timing side channel attack**
  - Attacker measures the cacheline access latency in phase 3
  - Flush+Reload, Evict+Reload, Prime+Probe, etc.

- **Example: Flush + Reload**
  - Shared memory between attacker and victim
  - Instruction support for cache flush

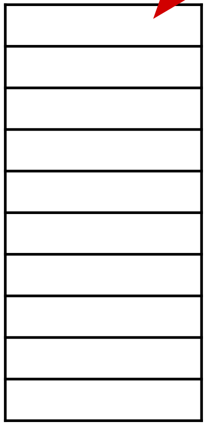


Threat Model

- **Cache timing side channel attack**
  - Attacker measures the cacheline access latency in phase 3
  - Flush+Reload, Evict+Reload, Prime+Probe, etc.

- **Example: Flush + Reload**
  - **Step1:** Attacker flushes the shared memory from cache

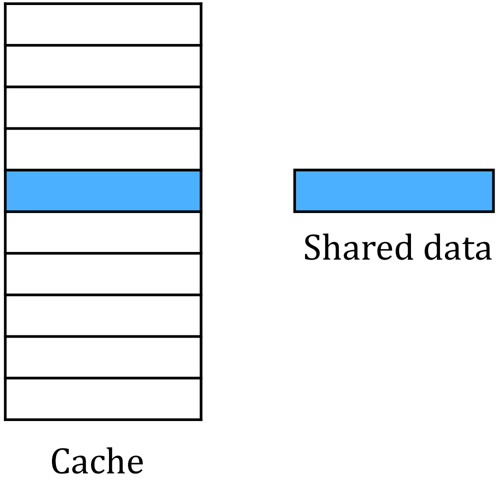
Eviction cacheline: cachelines that may be accessed by the victim



Threat Model

- **Cache timing side channel attack**
  - Attacker measures the cacheline access latency in phase 3
  - Flush+Reload, Evict+Reload, Prime+Probe, etc.

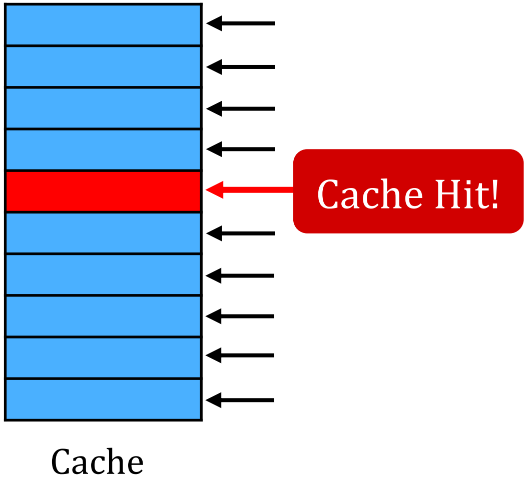
- **Example: Flush + Reload**
  - **Step1:** Attacker flushes the shared memory from cache
  - **Step2:** Victim accesses/does not access the shared memory



Threat Model

- **Cache timing side channel attack**
  - Attacker measures the cacheline access latency in phase 3
  - Flush+Reload, Evict+Reload, Prime+Probe, etc.

- **Example: Flush + Reload**
  - **Step1:** Attacker flushes the shared memory from cache
  - **Step2:** Victim accesses/does not access the shared memory
  - **Step3:** Attacker re-accesses the shared memory
    - Cache hit -> victim accessed
    - Cache miss -> victim did not access



Example: Spectre v1

```
if (x < array1_size) {
    y = array2[array1[x] * 512];
}
```

e.g. A victim code snippet

Secret is brought to cache before the bound check finishes and leaves a cache side channel !

- Attacker maliciously trains branch predictor to assume ‘ `if` ’ is likely true.
- **Speculative execution** allows instructions to be speculatively executed before the branch target is determined.
- Attacker invokes code with an **out-of-bounds** x.
  - **x = (address of a secret byte to read) – (base address of array1).**

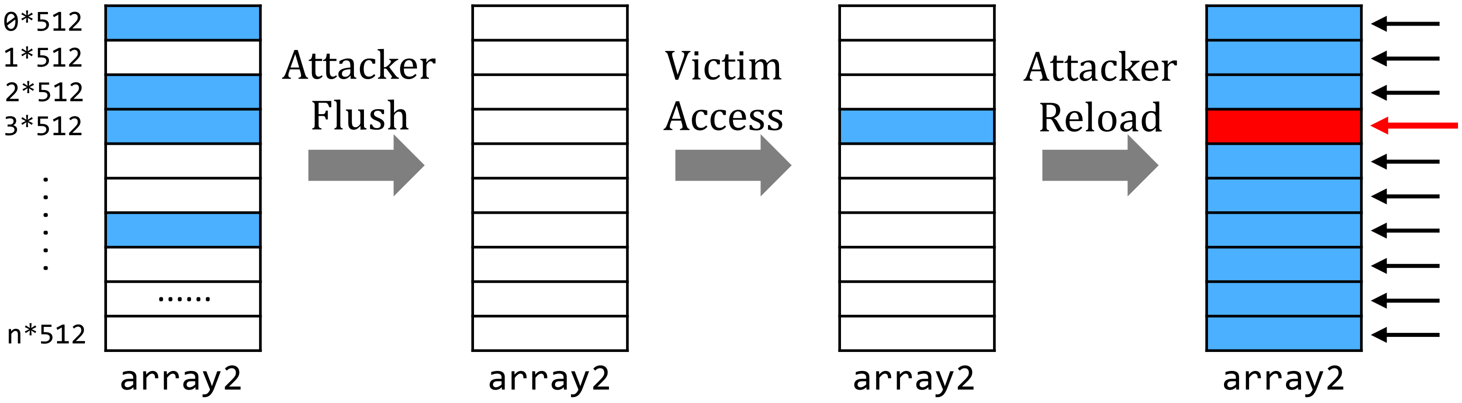
Example: Spectre v1

```
if (x < array1_size) {
        y = array2[array1[x] * 512];
}
```

e.g. A victim code snippet

Secret is brought to cache before the bound check finishes and leaves a cache side channel !

- Use cache side channel attack to extract the secret



Cache Hit! Secret is 3!

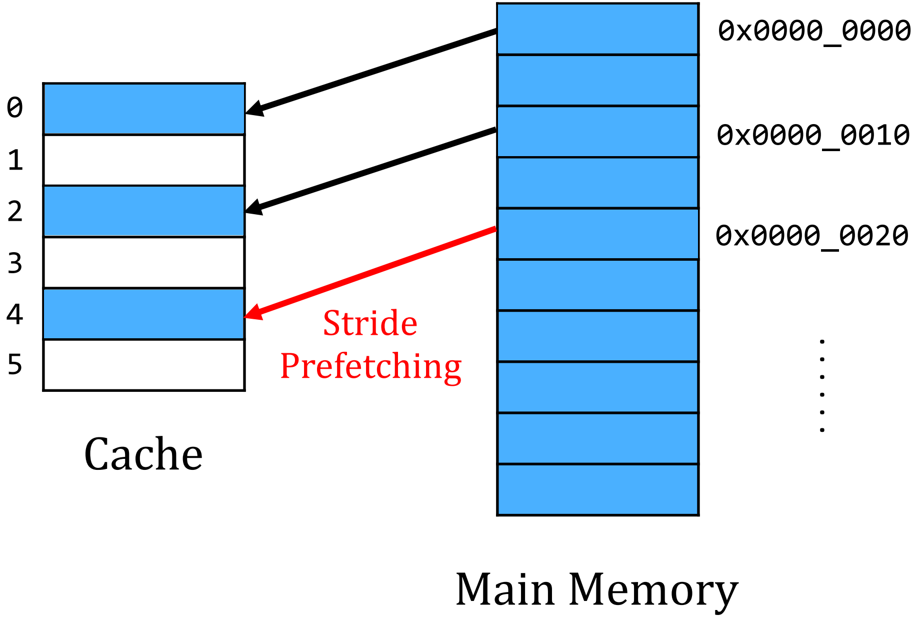
e.g. A Flush+Reload example

Prefetching

- **To reduce cache miss rate and improve performance**
  - Prefetch data into cache before the processor requests it.
  - Hardware prefetchers: Next-line Prefetcher, Stride Prefetcher, etc.

```
load r1, 0x0000_0000
load r2, 0x0000_0010
      ......
load r3, 0x0000_0020
```

Cache Hit!



Outline

I. Motivation

II. Background

III. PREFENDER Design

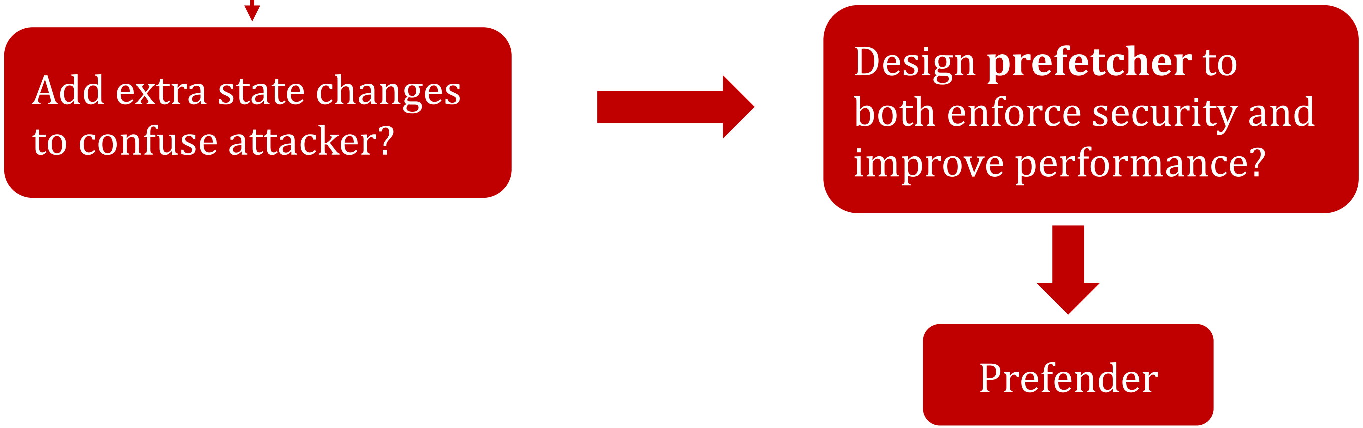
- Design Insight
- Architecture Overview
- Data Scale Tracker
- Access Pattern Tracker

IV. Experimental Evaluation

Design Insight

- **Observation 1**
  - Victim causes only one cache state change in phase 2.
  - Attacker utilizes the only one change to extract secrets in phase 3.

- **Observation 2**
  - Prefetching can cause extra cache state changes.
  - Prefetching can help enhance performance based on accurate prediction.



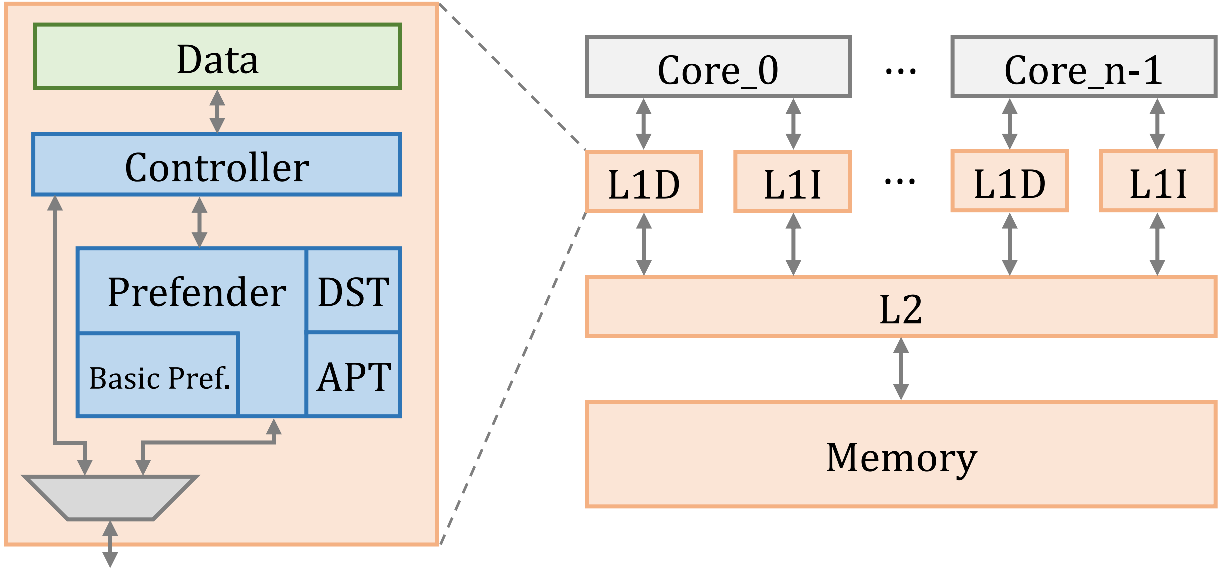
Add extra state changes to confuse attacker?

Design **prefetcher** to both enforce security and improve performance?

Prefender

Architecture Overview

- **Prefender: L1 Data Prefetcher**
  - Data scale tracker (DST) to interfere with phase 2
  - Access pattern tracker (APT) to interfere with phase 3
  - Support for basic hardware prefercher: next-line prefetcher, stride prefetcher

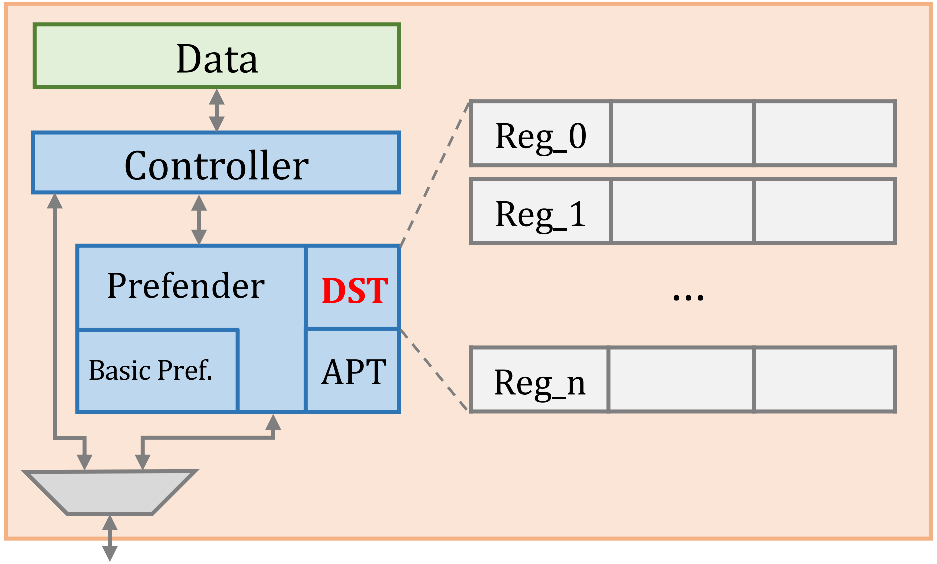


Data Scale Tracker

- **Goal**
  - To predict the eviction cachelines during victim's execution
- **Challenge**
  - In phase 2, victim may only access one secret-dependent eviction cacheline.
- **Observation**
  - Victim uses indirect memory access to load eviction cacheline.
  - `e.g. eviction[s * 128];`

➢ **Calculation History Buffer**

- To track how the load's target address is calculated.



DST - Calculation History Buffer

- **To track how the load's target address is calculated**
  - Record calculations related to each register
  - Addition (and subtraction) and Multiplication (and shifting)

- **Fixed Value (fva_r)**
  - Record immediate value used in the calculation history of register r

- **Scale (sc_r)**
  - Record scale of target addr in register r
    e.g. 128 * i -> scale is 128
  - To predict the cache access pattern

	fva_r	sc_r
Reg_0		
Reg_1		
…		
Reg_n		

Track sc_r with the help of fva_r by propagating them from source registers to destination registers.

Data Scale Tracker

- **Example**
  - Victim accesses `array[secret * 0x200]`
  - Finally, `r0 = secret_addr, r5 = array + secret * 0x200`

		fva_r	sc_r
r0			
r1			
r2			
r3			
r4			
r5			

```
load r0, 4(sp)
load r1, 0(r0)
load r2, array
load r3, 0x200
mul r4, r1, r3
add r5, r4, r2
load r6, 0(r5)
...
```

e.g. A victim assembly code snippet

Data Scale Tracker

- **1: `Load secret_addr to r0`**
- **2: `Load secret to r1`**
    - In data movement instructions, scale is initialized to 1.

		fva_r	sc_r
r0	secret_addr	NA	1
r1	secret	NA	1
r2			
r3			
r4			
r5			

```
1: load r0, 4(sp)
2: load r1, 0(r0)
3: load r2, array
4: load r3, 0x200
5: mul r4, r1, r3
6: add r5, r4, r2
7: load r6, 0(r5)
...
```

e.g. A victim assembly code snippet

Data Scale Tracker

- **3: `Load array to r2`**
- **4: `Load 0x200 to r3`**
  - If load an immediate number, set the fva_r .

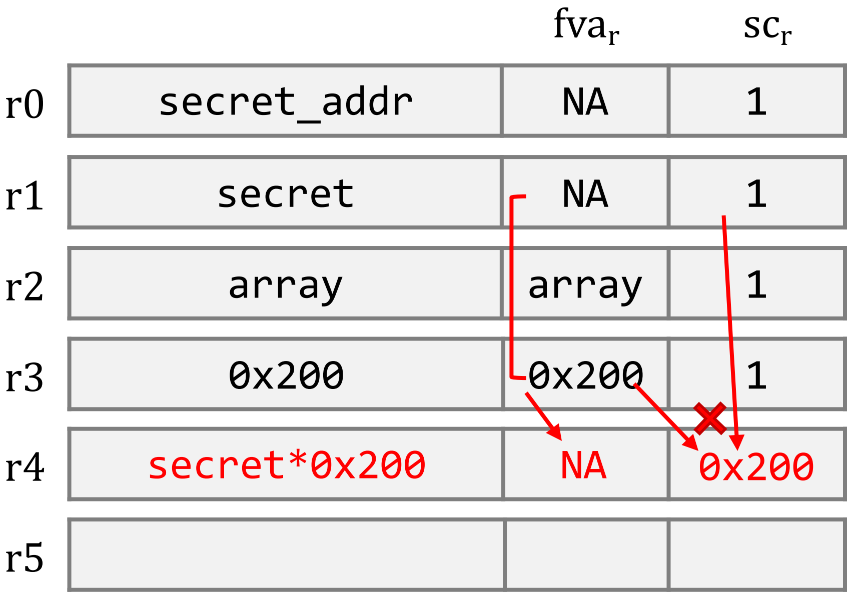
		fva_r	sc_r
r0	secret_addr	NA	1
r1	secret	NA	1
r2	array	array	1
r3	0x200	0x200	1
r4			
r5			

```
1: load r0, 4(sp)
2: load r1, 0(r0)
3: load r2, array
4: load r3, 0x200
5: mul r4, r1, r3
6: add r5, r4, r2
7: load r6, 0(r5)
...
```

e.g. A victim assembly code snippet

Data Scale Tracker

- **5: Calculate index `r4 = r1 * r3 (secret * 0x200)`**
  - $sc_{r4} = sc_{r1} * fva_{r3}$



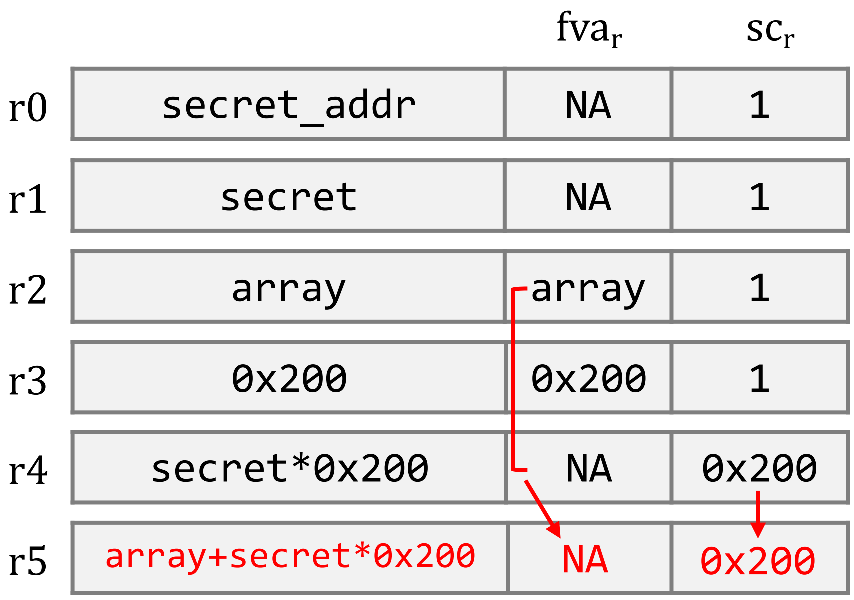
		fva_r	sc_r
r0	secret_addr	NA	1
r1	secret	NA	1
r2	array	array	1
r3	0x200	0x200	1
r4	secret*0x200	NA	0x200
r5			

```
1: load r0, 4(sp)
2: load r1, 0(r0)
3: load r2, array
4: load r3, 0x200
5: mul r4, r1, r3
6: add r5, r4, r2
7: load r6, 0(r5)
...
```

e.g. A victim assembly code snippet

Data Scale Tracker

- **6: Calculate** **target address**: **`r5 = r2 + r4 (array + secret*0x200)`**
  - fva_{r2} is valid, $sc_{r5} = sc_{r4}$



		fva_r	sc_r
r0	secret_addr	NA	1
r1	secret	NA	1
r2	array	array	1
r3	0x200	0x200	1
r4	secret*0x200	NA	0x200
r5	array+secret*0x200	NA	0x200

```
1: load r0, 4(sp)
2: load r1, 0(r0)
3: load r2, array
4: load r3, 0x200
5: mul r4, r1, r3
6: add r5, r2, r4
7: load r6, 0(r5)
...
```

e.g. A victim assembly code snippet

Data Scale Tracker

- **7: Load `array[secret * 0x200]` to r6**
  - sc_{r5} (0x200) > cacheline size (0x40 in the example) ! Do data prefetching!
  - Candidate address: `r5 + 0x200, r5 - 0x200` (prefetch data not in the cache).

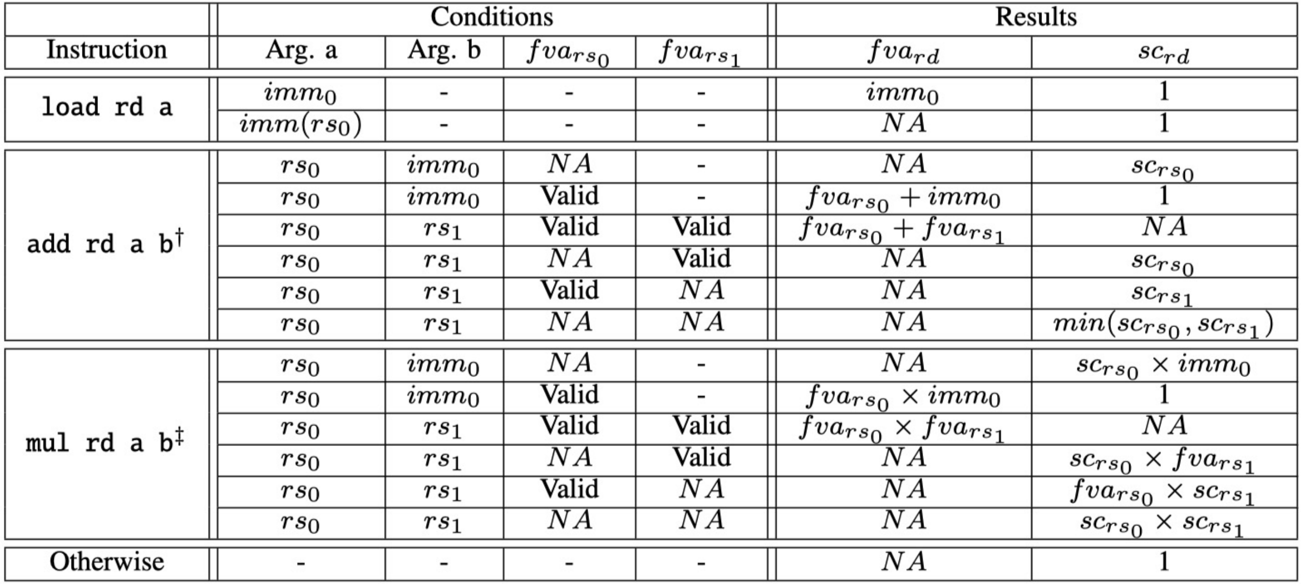
		fva_r	sc_r
r0	secret_addr	NA	1
r1	secret	NA	1
r2	array	array	1
r3	0x200	0x200	1
r4	secret*0x200	NA	0x200
r5	array+secret*0x200	NA	0x200

```
1: load r0, 4(sp)
2: load r1, 0(r0)
3: load r2, array
4: load r3, 0x200
5: mul r4, r1, r3
6: add r5, r2, r4
7: load r6, 0(r5)
...
```

e.g. A victim assembly code snippet

Data Scale Tracker

- **More complicated access pattern can also be handled**
  - 128 * i + 32 * j + imm, $(128i_0i_1i_2 + 32j_0 * 16j_1) * (48k_0 + imm)$, etc.
  - More analyses in the paper



	Conditions				Results	
Instruction	Arg. a	Arg. b	fva_{rs_0}	fva_{rs_1}	fva_{rd}	sc_{rd}
`load rd a`	imm_0	-	-	-	imm_0	1
	$imm(rs_0)$	-	-	-	NA	1
`add rd a b`†	rs_0	imm_0	NA	-	NA	sc_{rs_0}
	rs_0	imm_0	Valid	-	$fva_{rs_0} + imm_0$	1
	rs_0	rs_1	Valid	Valid	$fva_{rs_0} + fva_{rs_1}$	NA
	rs_0	rs_1	NA	Valid	NA	sc_{rs_0}
	rs_0	rs_1	Valid	NA	NA	sc_{rs_1}
	rs_0	rs_1	NA	NA	NA	$min(sc_{rs_0}, sc_{rs_1})$
`mul rd a b`‡	rs_0	imm_0	NA	-	NA	$sc_{rs_0} \times imm_0$
	rs_0	imm_0	Valid	-	$fva_{rs_0} \times imm_0$	1
	rs_0	rs_1	Valid	Valid	$fva_{rs_0} \times fva_{rs_1}$	NA
	rs_0	rs_1	NA	Valid	NA	$sc_{rs_0} \times fva_{rs_1}$
	rs_0	rs_1	Valid	NA	NA	$fva_{rs_0} \times sc_{rs_1}$
	rs_0	rs_1	NA	NA	NA	$sc_{rs_0} \times sc_{rs_1}$
Otherwise	-	-	-	-	NA	1

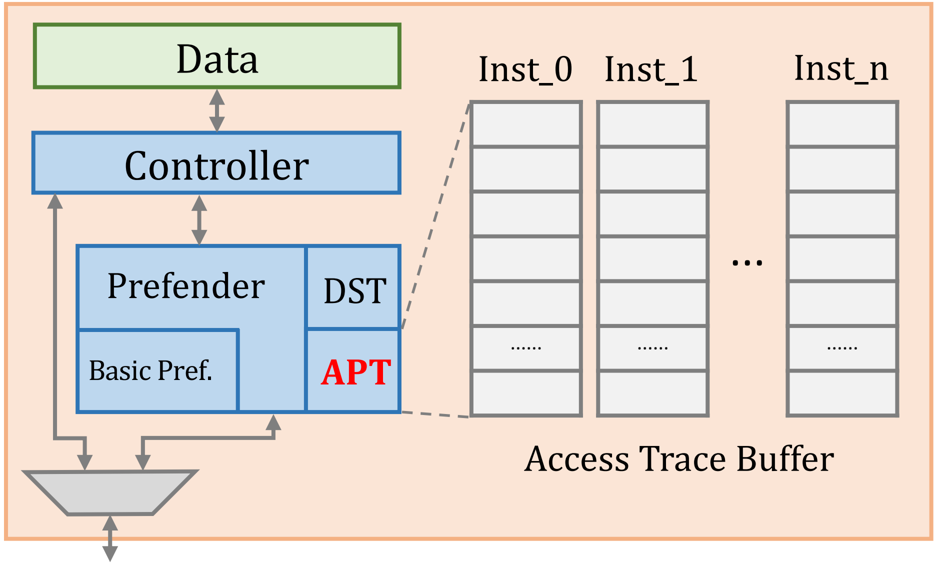
Table1: Rules to calculate fva_{rd} and sc_{rd} .

Access Pattern Tracker

- **Goal**
  - To predict the access patterns of attacker during its measurement
- **Challenge**
  - In phase 3, attacker randomly measures the latency to bypass prefetcher.
- **Observation**
  - Random accesses are associated with only a few load instructions.

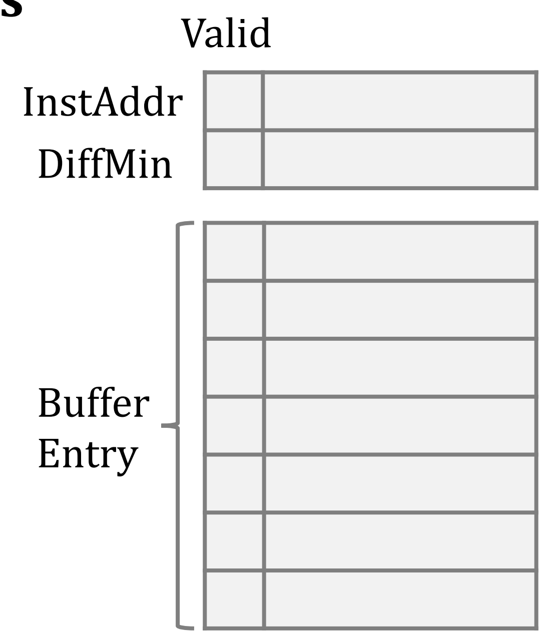
➢**Access trace buffer**

- Instruction-level granularity to detect attacks



APT - Access Trace Buffer

- **Instruction-level granularity to detect attacks**
  - Each buffer is associated with one load instruction
- **InstAddr register**
  - Record instruction address of its associated load
- **Buffer entry**
  - Record block address accessed by the load
- **DiffMin register**
  - Record minimum difference between two block addresses among all the entries



Access Trace Buffer

Access Pattern Tracker

- **Example**
  - **0x8008 load:** Randomly load array2[array1[x] * 0x200]
  - **0x8018 load:** Sequentially load safe_array[i]

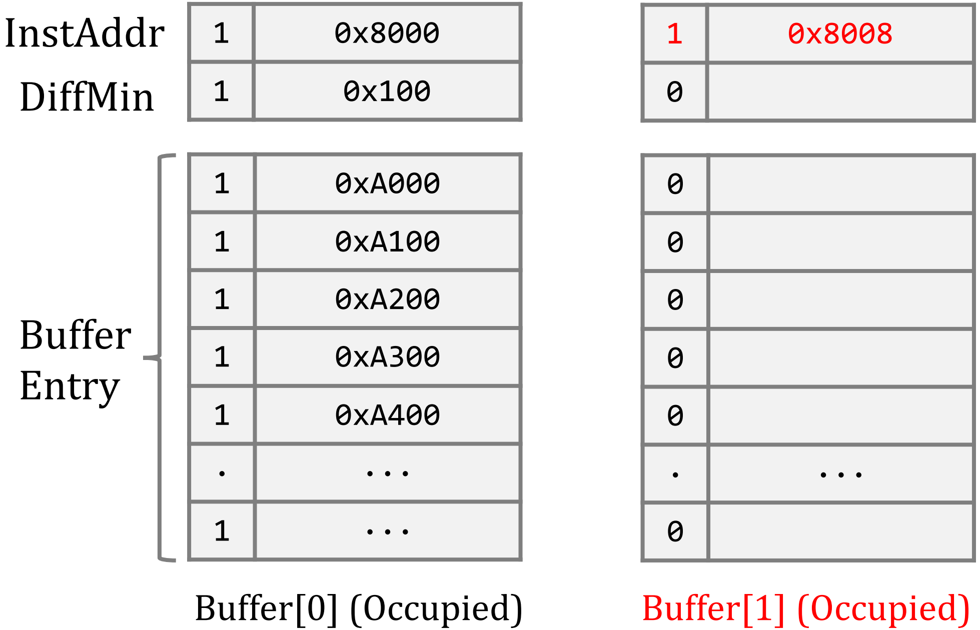
InstAddr	1	0x8000	0	
DiffMin	1	0x100	0	
Buffer Entry	1	0xA000	0	
	1	0xA100	0	
	1	0xA200	0	
	1	0xA300	0	
	1	0xA400	0	
	.	...	.	...
	1	...	0	
	Buffer[0] (Occupied)		Buffer[1] (Empty)	

```
0x8008: load r1, 0(r10)
        ......
0x8018: load r3, 0(r11)
```

e.g. An attacker assembly code snippet

Access Pattern Tracker

- ① **Buffer allocation**
  - Allocate an empty buffer.



InstAddr	1	0x8000
DiffMin	1	0x100
Buffer Entry	1	0xA000
	1	0xA100
	1	0xA200
	1	0xA300
	1	0xA400
	.	...
	1	...

Buffer[0] (Occupied)

1	0x8008
0	
0	
0	
0	
0	
0	
.	...
0	

Buffer[1] (Occupied)

```
0x8008: load r1, 0(r10)
        .......
0x8018: load r3, 0(r11)
```

Access Pattern Tracker

- ② **Entry updating**
  - If not recorded, store the block address (BlkAddr) in a new entry.

InstAddr	1	0x8000
DiffMin	1	0x100
Buffer Entry	1	0xA000
	1	0xA100
	1	0xA200
	1	0xA300
	1	0xA400
	.	...
	1	...

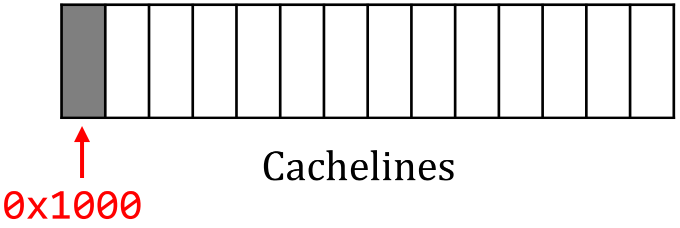
Buffer[0] (Occupied)

InstAddr	1	0x8008
DiffMin	0	
Buffer Entry	1	0x1000
	0	
	0	
	0	
	0	
	.	...
	0	

Buffer[1] (Occupied)

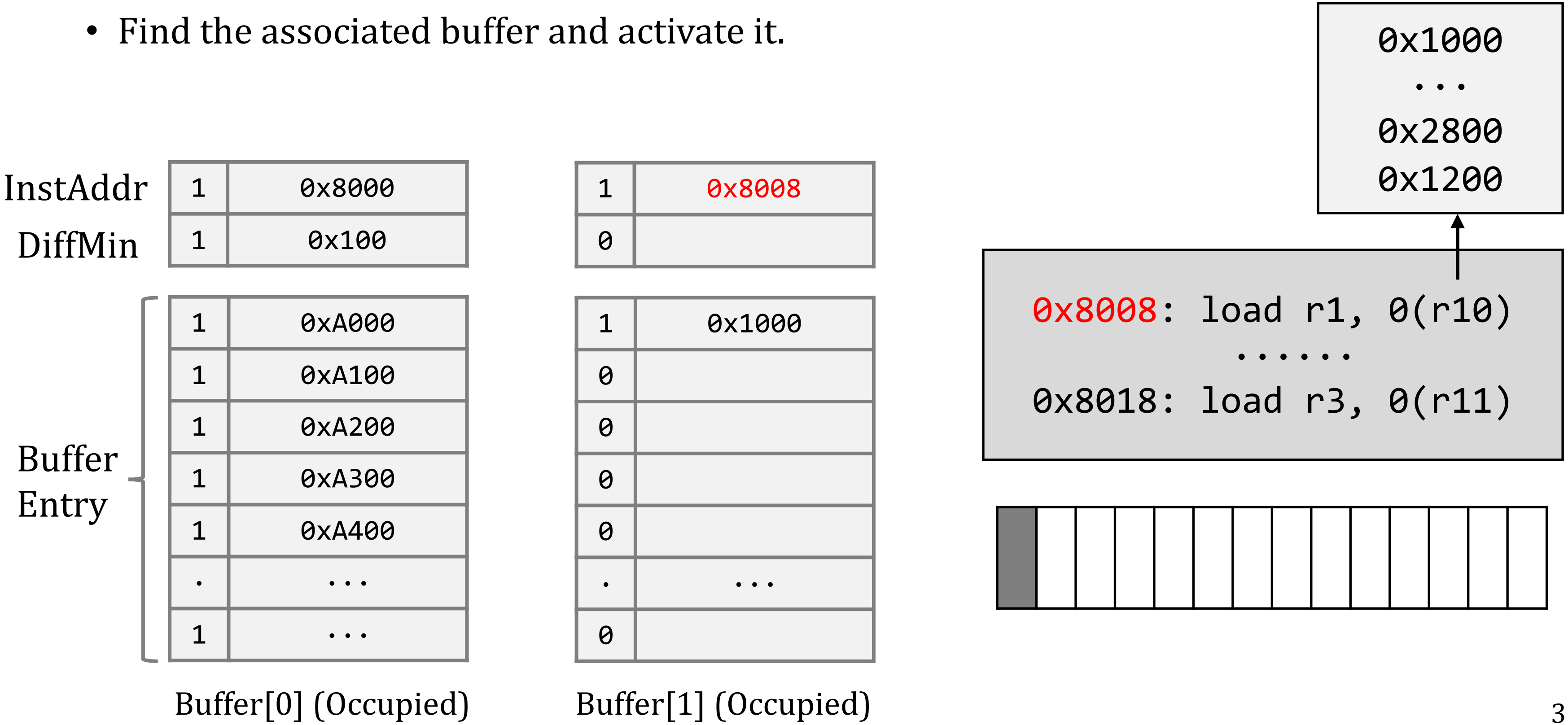
0x1000
...
0x2800
0x1200

```
0x8008: load r1, 0(r10)
       ......
0x8018: load r3, 0(r11)
```



Access Pattern Tracker

- ① **Buffer allocation**
  - Find the associated buffer and activate it.



InstAddr	1	0x8000	1	0x8008
DiffMin	1	0x100	0	
Buffer Entry	1	0xA000	1	0x1000
	1	0xA100	0	
	1	0xA200	0	
	1	0xA300	0	
	1	0xA400	0	
	.	...	.	...
	1	...	0	

Buffer[0] (Occupied) Buffer[1] (Occupied)

```
0x1000
...
0x2800
0x1200
```

```
0x8008: load r1, 0(r10)
......
0x8018: load r3, 0(r11)
```

Access Pattern Tracker

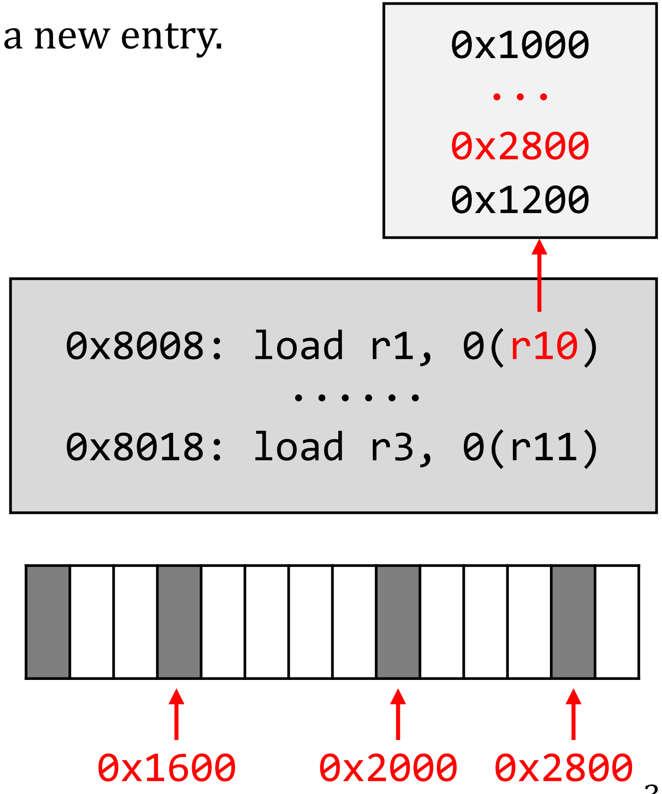
- ② **Entry updating**
  - If not recorded, store the block address (BlkAddr) in a new entry.
  - If all entries are valid, use LRU to replace.

InstAddr	1	0x8000
DiffMin	1	0x100
Buffer Entry	1	0xA000
	1	0xA100
	1	0xA200
	1	0xA300
	1	0xA400
	.	...
	1	...

Buffer[0] (Occupied)

1	0x8008
0	
1	0x1000
1	0x2000
1	0x1600
1	0x2800
0	
.	...
0	

Buffer[1] (Occupied)



```
0x8008: load r1, 0(r10)
              ......
0x8018: load r3, 0(r11)
```

Access Pattern Tracker

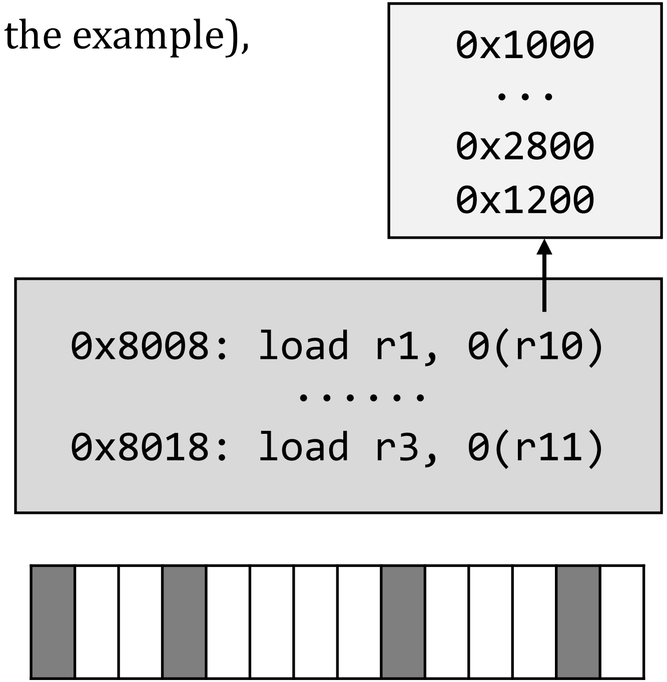
- ③ **DiffMin updating**
  - If the number valid entries reaches a threshold (4 in the example), calculate DiffMin.

InstAddr	1	0x8000
DiffMin	1	0x100
Buffer Entry	1	0xA000
	1	0xA100
	1	0xA200
	1	0xA300
	1	0xA400
	.	...
	1	...

Buffer[0] (Occupied)

InstAddr	1	0x8008
DiffMin	1	0x600
Buffer Entry	1	0x1000
	1	0x2000
	1	0x1600
	1	0x2800
	0	
	.	...
	0	

Buffer[1] (Occupied)



0x1000
...
0x2800
0x1200

```
0x8008: load r1, 0(r10)
......
0x8018: load r3, 0(r11)
```

Access Pattern Tracker

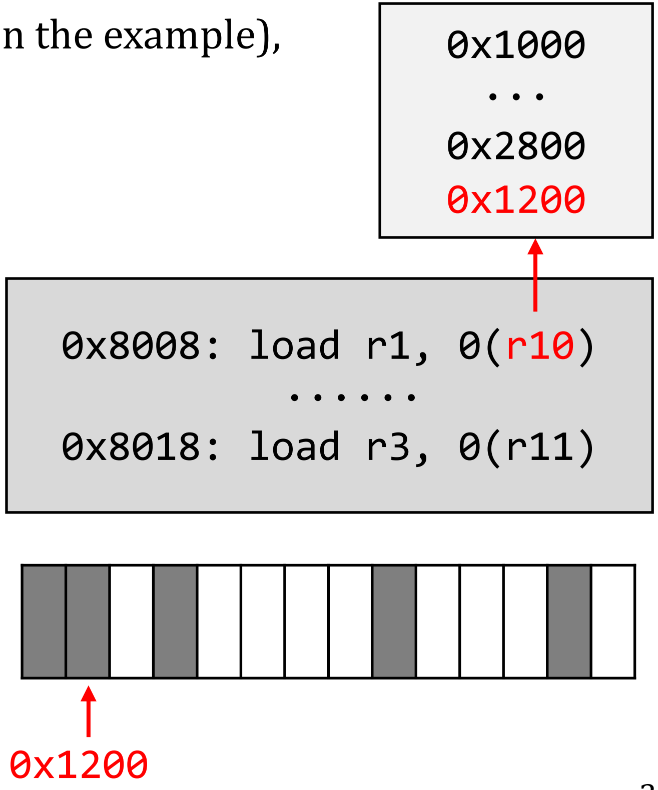
- ③ **DiffMin updating**
  - If the number valid entries surpasses a threshold (4 in the example), update DiffMin each time the buffer is activated.

InstAddr	1	0x8000
DiffMin	1	0x100
Buffer Entry	1	0xA000
	1	0xA100
	1	0xA200
	1	0xA300
	1	0xA400
	.	...
	1	...

Buffer[0] (Occupied)

1	0x8008
1	~~0x600~~ 0x200
1	0x1000
1	0x2000
1	0x1600
1	0x2800
1	0x1200
.	...
0	

Buffer[1] (Occupied)



0x1000
...
0x2800
0x1200

```
0x8008: load r1, 0(r10)
         ......
0x8018: load r3, 0(r11)
```

Access Pattern Tracker

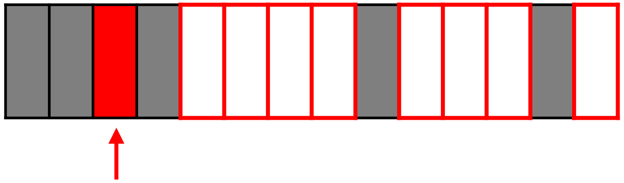
- ④ **Data prefetching**
  - If the number valid entries surpasses a threshold, do prefetching!
  - Candidate address: BlkAddr + DiffMin, BlkAddr - Diffmin (prefetch data not in the cache).

InstAddr	1	0x8000	1	0x8008
DiffMin	1	0x100	1	0x200
Buffer Entry	1	0xA000	1	0x1000
	1	0xA100	1	0x2000
	1	0xA200	1	0x1600
	1	0xA300	1	0x2800
	1	0xA400	1	0x1200
	.	...	.	...
	1	...	0	

Buffer[0] (Occupied) Buffer[1] (Occupied)

```
0x1000
...
0x2800
0x1200
```

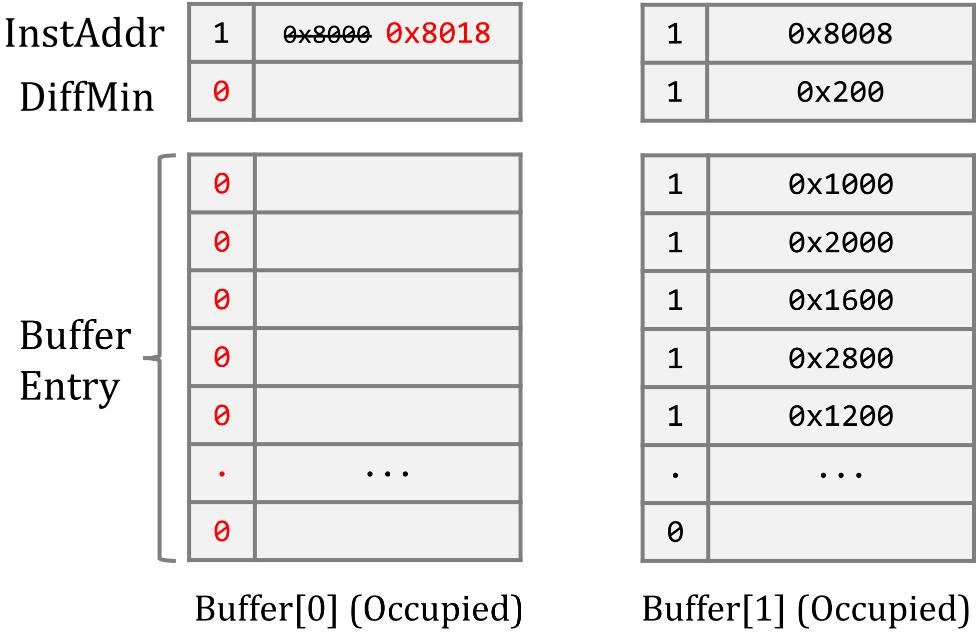
```
0x8008: load r1, 0(r10)
......
0x8018: load r3, 0(r11)
```



0x1000 in cache! Prefetch 0x1400

Access Pattern Tracker

- ① **Buffer allocation**
  - If all buffers are occupied, use LRU to select a buffer.



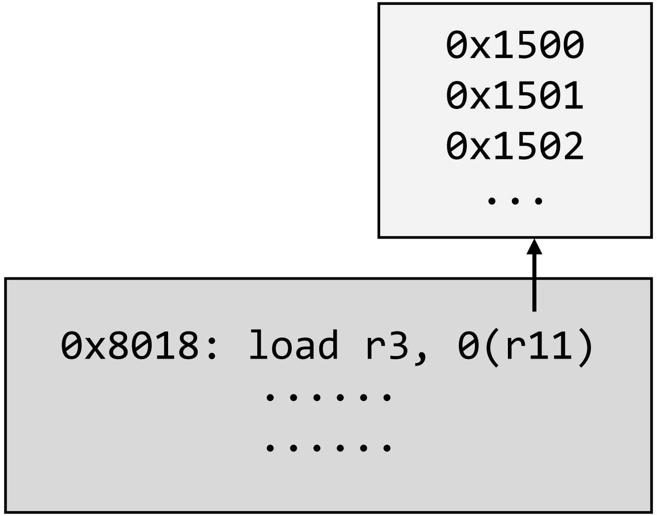
InstAddr	1	~~0x8000~~ 0x8018	1	0x8008
DiffMin	0		1	0x200
Buffer Entry	0		1	0x1000
	0		1	0x2000
	0		1	0x1600
	0		1	0x2800
	0		1	0x1200
	.	...	.	...
	0		0	
	Buffer[0] (Occupied)		Buffer[1] (Occupied)	

```
0x8018: load r3, 0(r11)
      ......
      ......
```

Access Pattern Tracker

- ② **Entry updating**
- ③ **DiffMin updating**
- ④ **Data prefetching**
- ①②③④, ①②③④, ①②③④ ……

InstAddr	1	0x8018	1	0x8008
DiffMin	1	0x1	1	0x200
Buffer Entry	1	0x1500	1	0x1000
	1	0x1501	1	0x2000
	1	0x1502	1	0x1600
	1	0x1503	1	0x2800
	1	0x1504	1	0x1200
	.	...	.	...
	1	...	0	
	Buffer[0] (Occupied)		Buffer[1] (Occupied)	



Outline

I. Motivation

II. Background

III. PREFENDER Design

IV. Experimental Evaluation

- Security Evaluation
- Performance Evaluation

Methodology

- **Tools**
  - Gem5 simulator

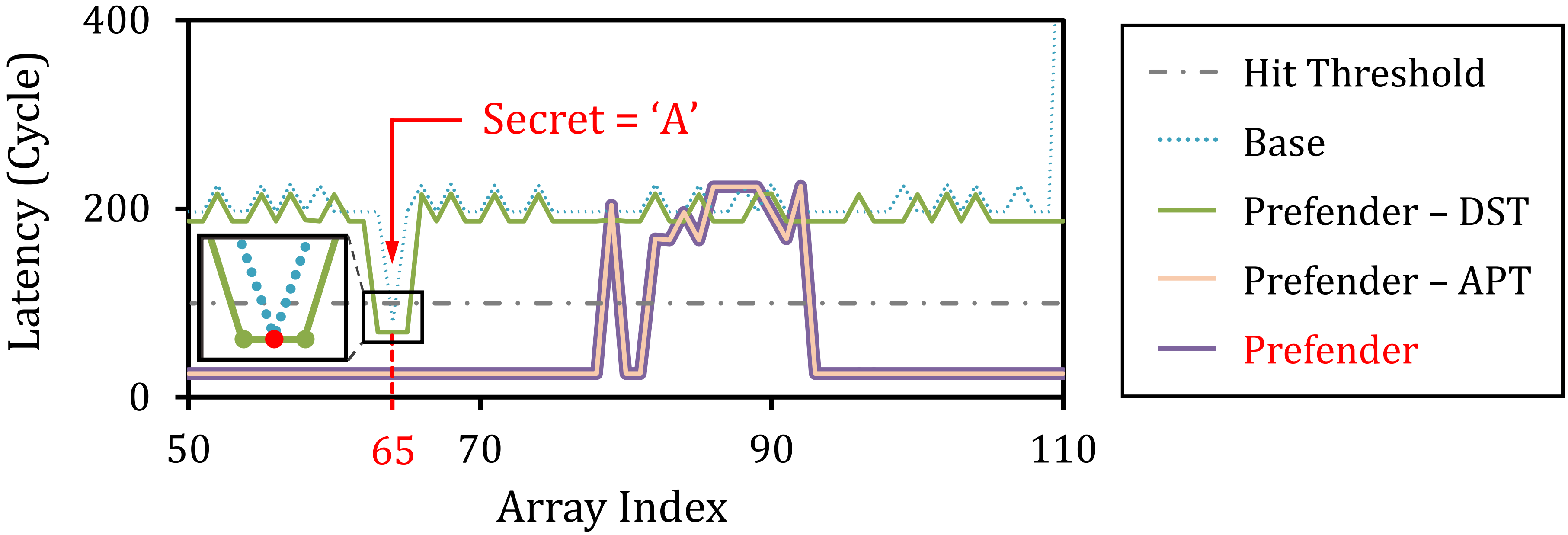
- **Configuration**
  - System call emulation (SE) mode
  - x86 O3 core at 2GHz
  - 32KB 2-way L1ICache, 64KB 2-way L1DCache, 2MB 8-way L2Cache

- **Testbench**
  - **Security:** Spectre v1 (Flush+Reload, Evict+Reload, Prime+Probe)
  - **Performance:** SPEC CPU 2006 benchmark

Security Evaluation

- **Spectre v1 (Flush + Reload)**

So many cache hits …
Which is the secret ???

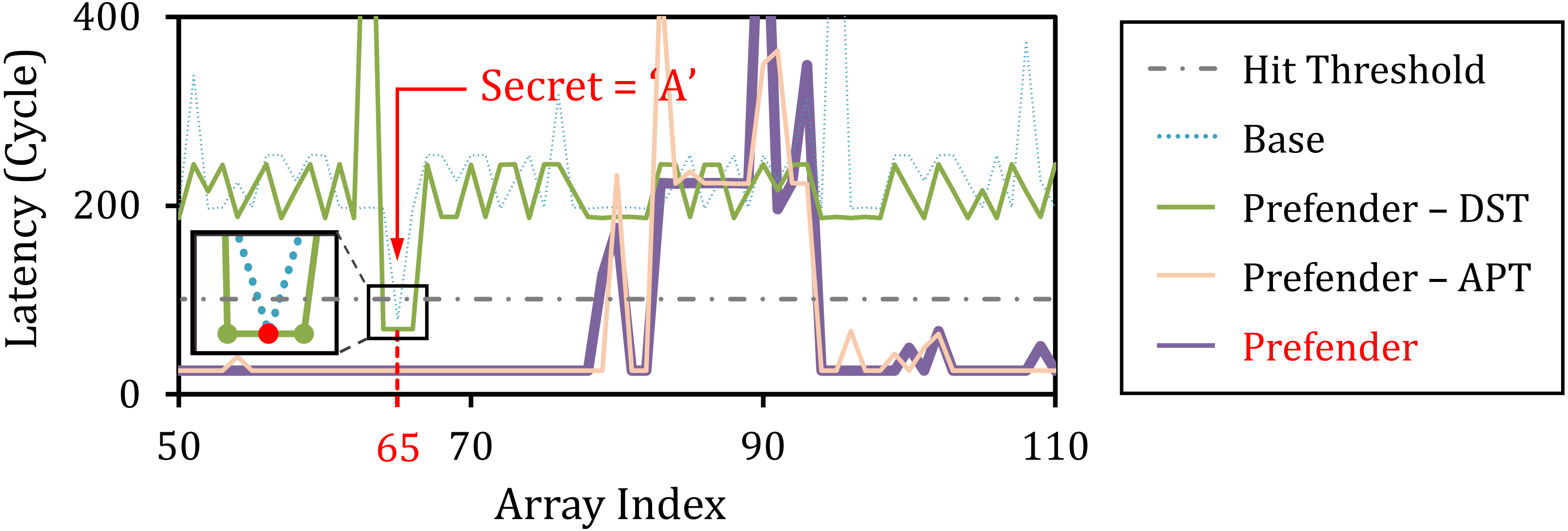


Security Evaluation

- **Spectre v1 (Evict + Reload)**

SPECTRE

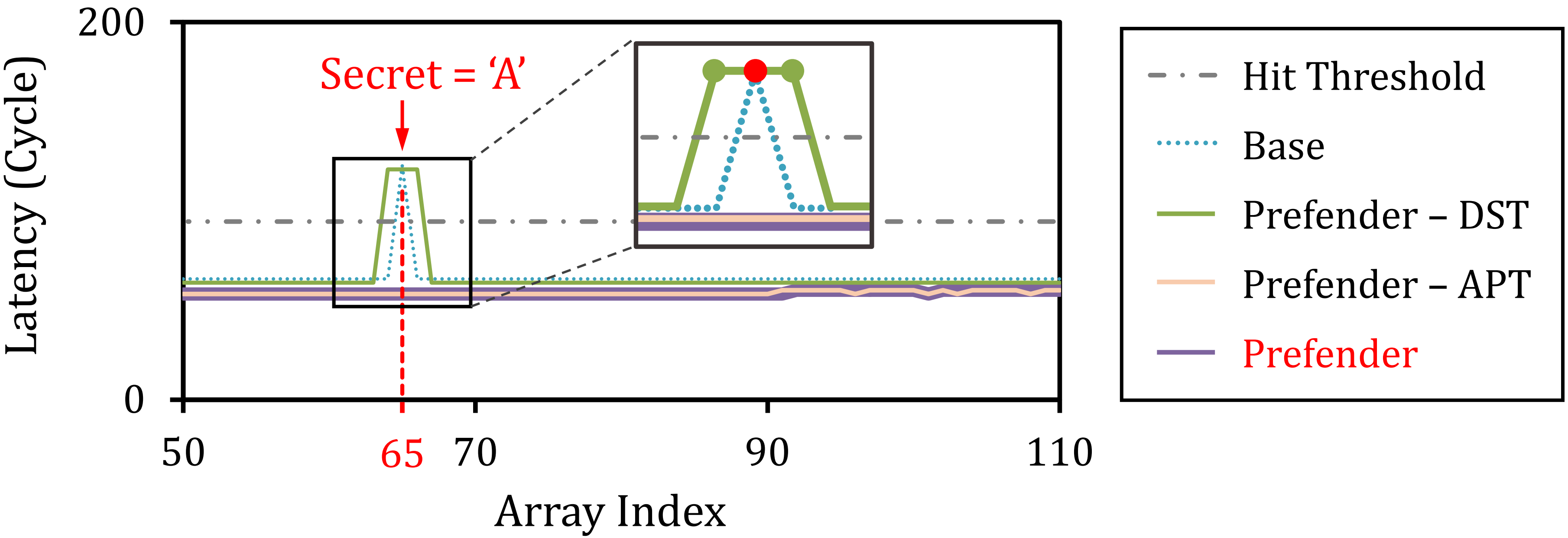
So many cache hits …
Fail again !!!



Security Evaluation

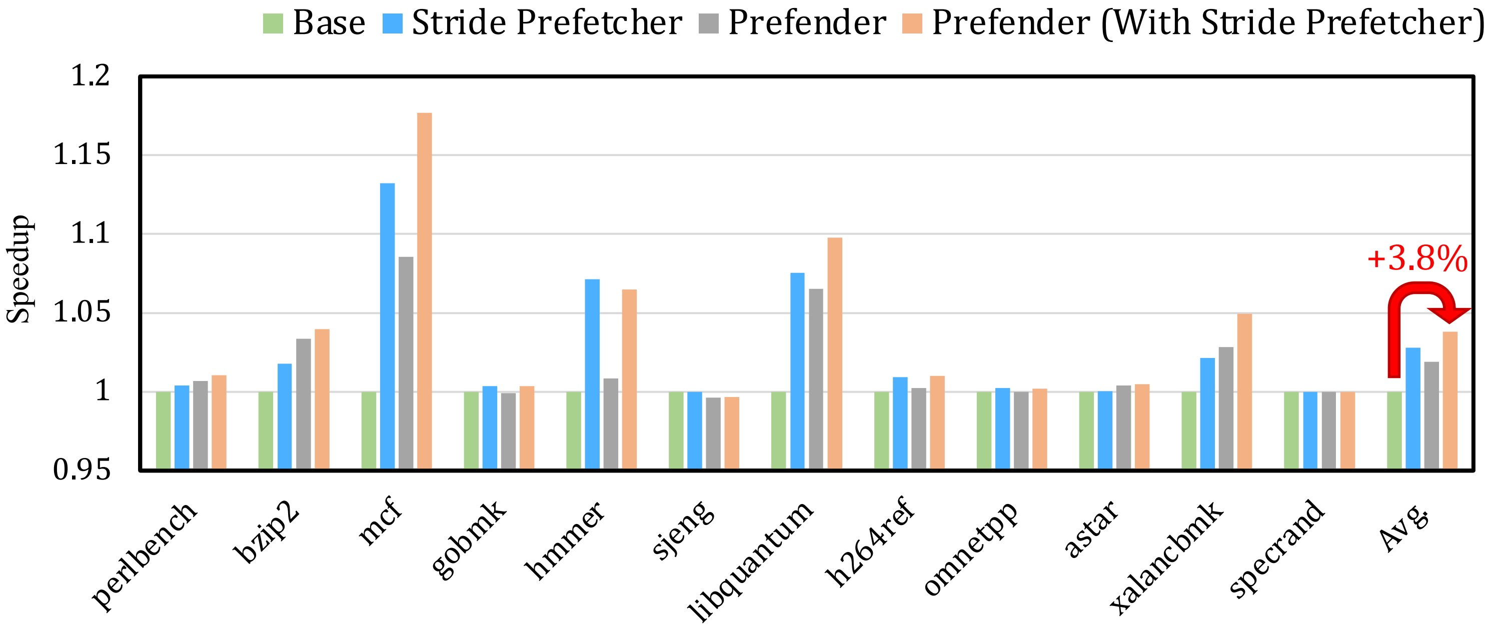
- **Spectre v1 (Prime + Probe)**

No cache miss ???
Prefender beats me !!!



Performance Evaluation

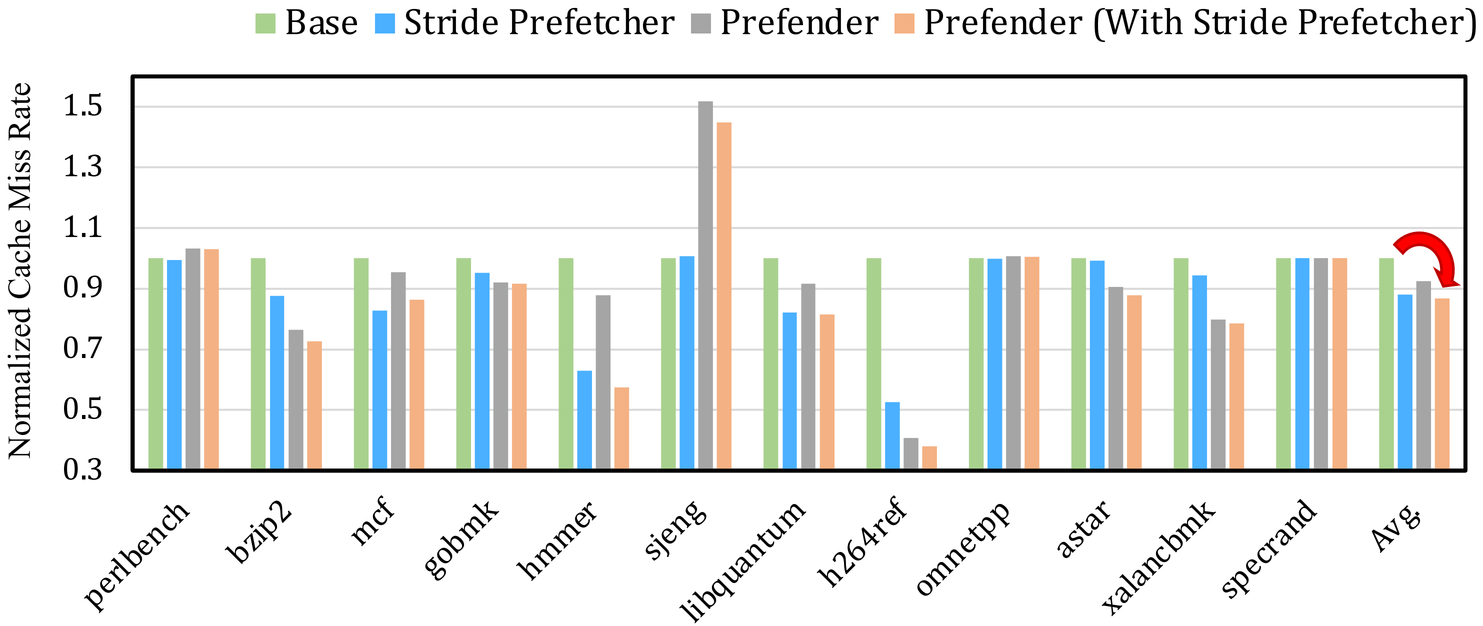
- **SPEC CPU 2006**
  - **APT:** 32 buffers, 8 entries



- More cases in the paper

Cache Miss Rate Evaluation

- **SPEC CPU 2006**
  - **APT:** 32 buffers, 8 entries



- More cases in the paper

Hardware Resource Consumption Analysis

- **Data Scale Tracker**
  - **Assumption:**
    - The prefetching is performed within one page
    - Page size is < 64KB, and each core has < 100 registers
    - Therefore, 16 bits are enough for each fixed value (fva) and each scale (sc)
  - **Memory:** < 16*2*100/8 Bytes, which is < 400 Bytes
  - **Datapath:** A 16-bit adder, a 16-bit multiplier, and a 16-bit comparator
- **Access Pattern Tracker**
  - **Assumption:**
    - In Access Trace Buffer, each entry, InstAddr, DiffMin, and the time for LRU are 64-bit
    - The target is to prefetch eviction cachelines, and the size of L1Dcache < 1MB
    - Therefore, 20 bits are enough for the calculation
    - There are 32 Access Trace Buffers, each of which has 8 entries
  - **Memory:** < 64*(8+3)*32/8 Bytes, which is < 2816 Bytes
  - **Datapath:** Several 20-bit comparators and 20-bit subtractors for each buffer

Conclusion

- Propose a secure prefetcher, which is able to **defeat cache side channel attacks** while **maintaining or even improving performance**.
- Design **Data Scale Tracker (DST)** to predict the eviction cachelines during the victim's execution.
- Design **Access Pattern Tracker (APT)** to predict the access patterns during the attacker's measurement.
- Prove **the defense effectiveness for Spectre** and achieve **a speedup for SPEC CPU 2006 benchmark**.

Thanks for Listening!

If You Have Any Question, Please Contact Us at
luyli@smail.nju.edu.com
flang@nju.edu.cn