DESIGN AUTOMATION CONFERENCE

58

FROM CHIPS TO SYSTEMS — LEARN TODAY, CREATE TOMORROW

DEC 5 - 9, 2021 ◆ San Francisco, California

Load-Step: A Precise TrustZone Execution Control Framework for Exploring New Side-channel Attacks Like Flush+Evict

Zili KOU [1], Wenjian HE [1], Wei ZHANG [1], and Sharad Sinha [2]

[1] Hong Kong University of Science and Technology

[2] Indian Institute of Technology Goa

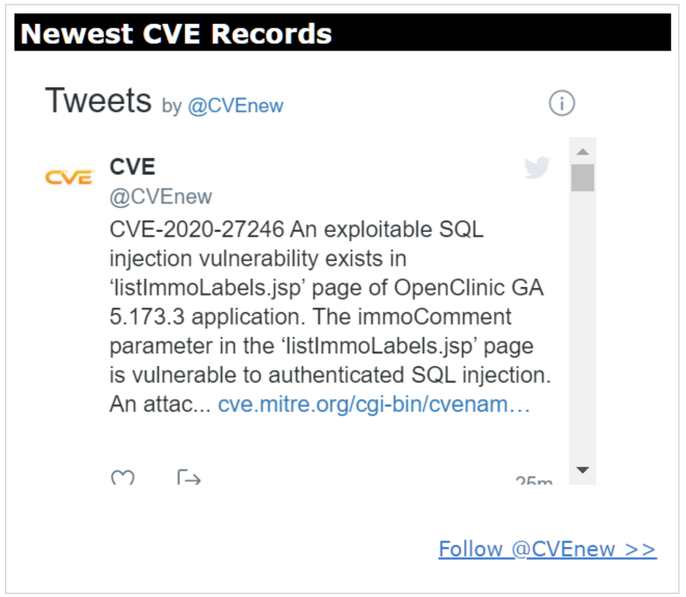
May 15, 2021

Security in Computing Systems

- Essential

- Suffers from software vulnerabilities…
  - Insecure codes can be exploit, e.g., stack overflow
  - Operating system(OS) itself has flaws

- Forever on the way to fix up!



Newest CVE Records

Tweets by @CVEnew

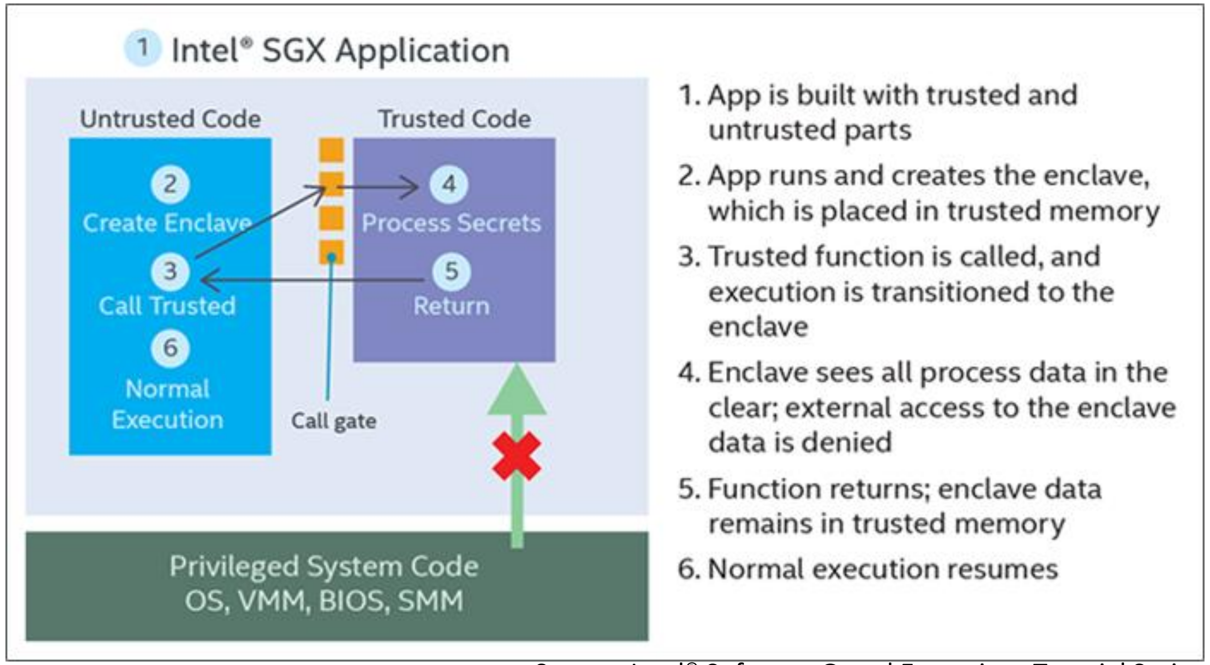
CVE
@CVEnew
CVE-2020-27246 An exploitable SQL injection vulnerability exists in 'listImmoLabels.jsp' page of OpenClinic GA 5.173.3 application. The immoComment parameter in the 'listImmoLabels.jsp' page is vulnerable to authenticated SQL injection. An attac... cve.mitre.org/cgi-bin/cvenam…

Follow @CVEnew >>

source: cve.mitre.org

Trusted Execution Environment (TEE)

- Hardware level isolate
  - Enclave or outside
- Commercial products
  - Intel SGX
  - Arm TrustZone



1. App is built with trusted and untrusted parts
2. App runs and creates the enclave, which is placed in trusted memory
3. Trusted function is called, and execution is transitioned to the enclave
4. Enclave sees all process data in the clear; external access to the enclave data is denied
5. Function returns; enclave data remains in trusted memory
6. Normal execution resumes

Source: Intel® Software Guard Extensions Tutorial Series

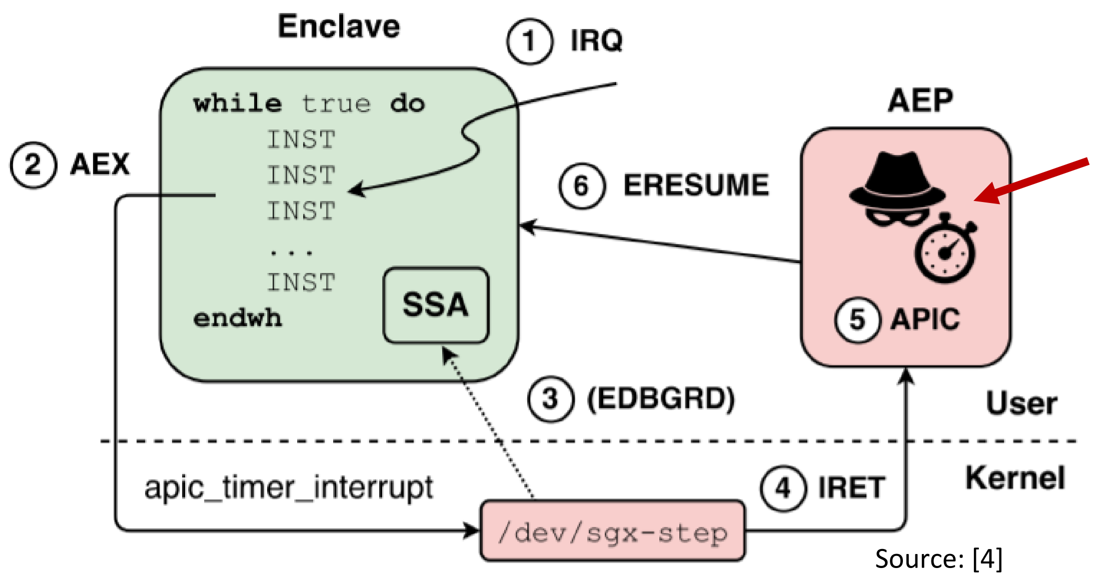
Upgraded protection from TEE

- Strong isolation
  - Accessing to secure memory is encrypted, authorized, and attested
- Fail to defend µarch side-channel attackers
  - Secrets can still be leaked by (cache) side-channels
    - Cache Attacks on Intel SGX [1], Armageddon[2]
  - Speculative execution vulnerabilities still exits
    - Foreshadow [3]…
- TEE is more privileged than OS
  - Protects enclaves against even “**malicious OS**”
  - Kernel (privileged) attacker should be considered!

Kernel Attacker on Intel SGX

SGX-Step [4] Interrupt the enclaves **per instruction**,

and then detects the µarch side-channel

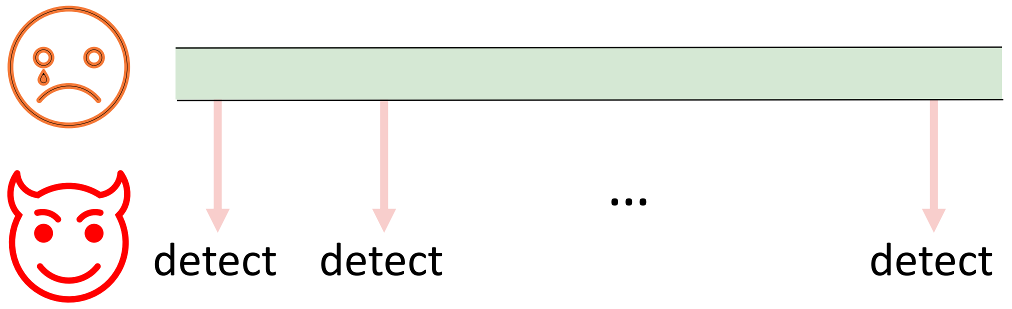


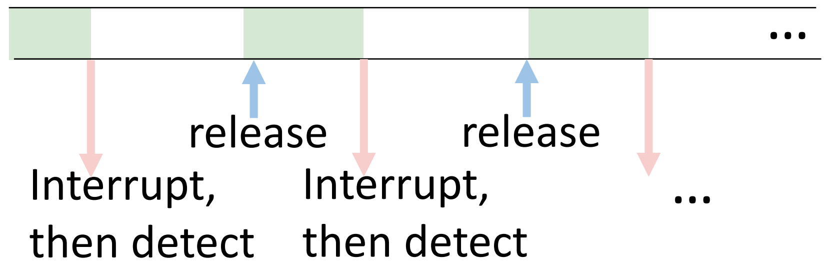
High precision hardware timer on intel CPU

Source: [4]

Kernel Attacker on Intel SGX

	User-space attack	Kernel-privileged attack
Victim	User application	Trusted application in enclave
Attacker	User application	Malicious OS
Noise	high	low
Temporal resolution	Low, as victim and attacker run **simultaneously**	High, depends on the **interrupt frequency**

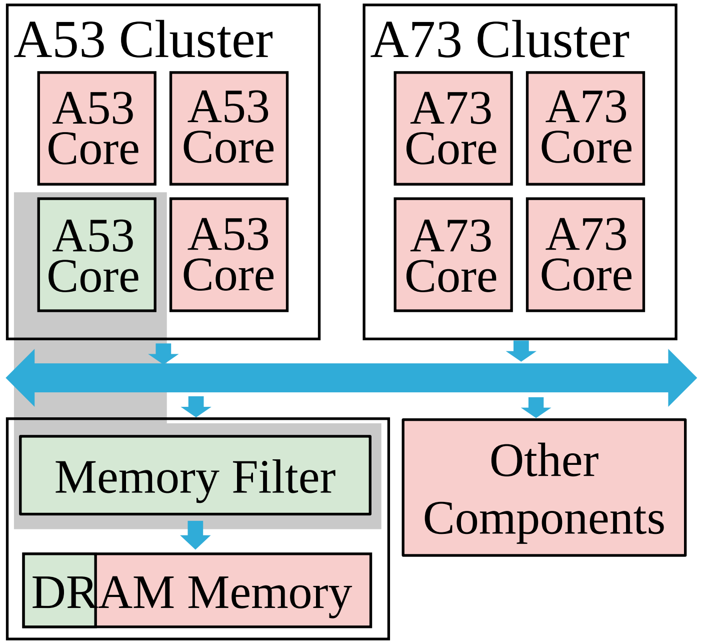




Research Gap

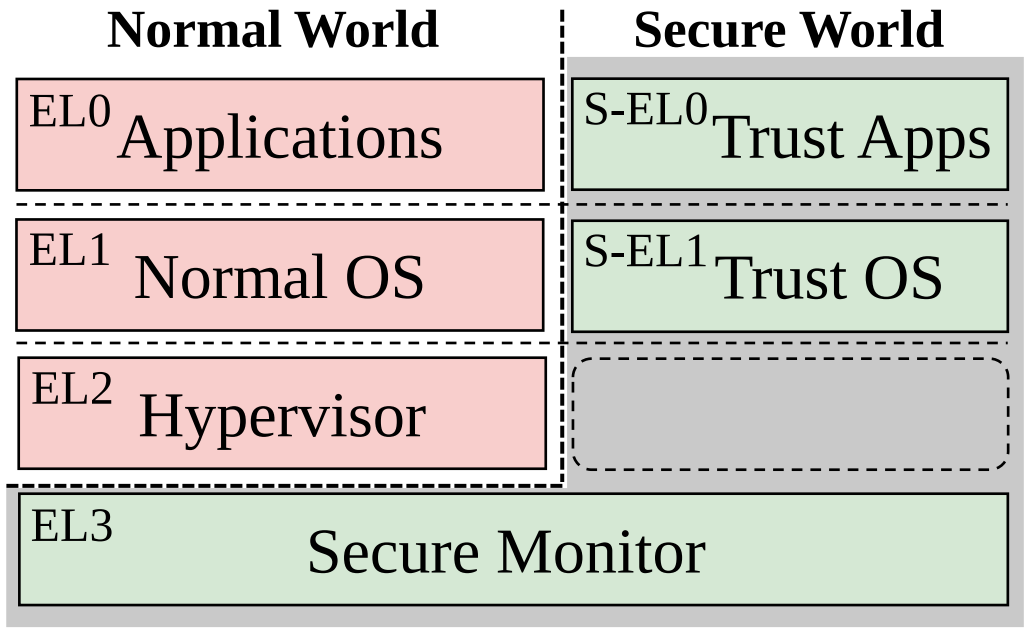
- User-space attacker on Arm TrustZone
  - Cache side-channel attacks still work on TrustZone [2][7]
- **Kernel attacker on Arm TrustZone?**
  - One nonsystematic work exits [8]
  - **Newly Arm-specific μarch side-channels?**
  - **Maximum temporal resolution?**

Arm TrustZone, in hardware



- Non-secure bit (NS-bit)
  - In each core's register
  - In tags of cache lines
- Memory filter
  - Isolate the secure memory

Arm TrustZone, in software



- Exception Level (EL)
  - Distinguish privileges
- World switch
  - Ensured by concrete protocol
- “smc” instruction
  - Communicate between secure world and normal world.

Threat Model

- Some cryptography program is implemented at S-EL0
- Attacker has full privilege of EL1 (Linux kernel)
  - Can install an external kernel module
  - Assign any core to run a Trusted Application
- Obey the threat model of TrustZone
  - No exploit software vulnerabilities
  - No privilege at EL3, S-ELs

Experiment Platform

In software,

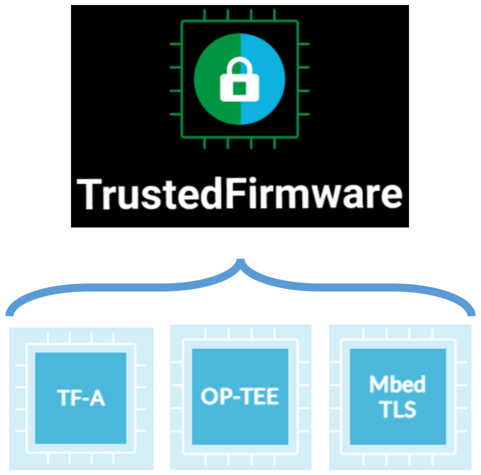
"TrustedFirmware.org": the reference implementation of TrustZone

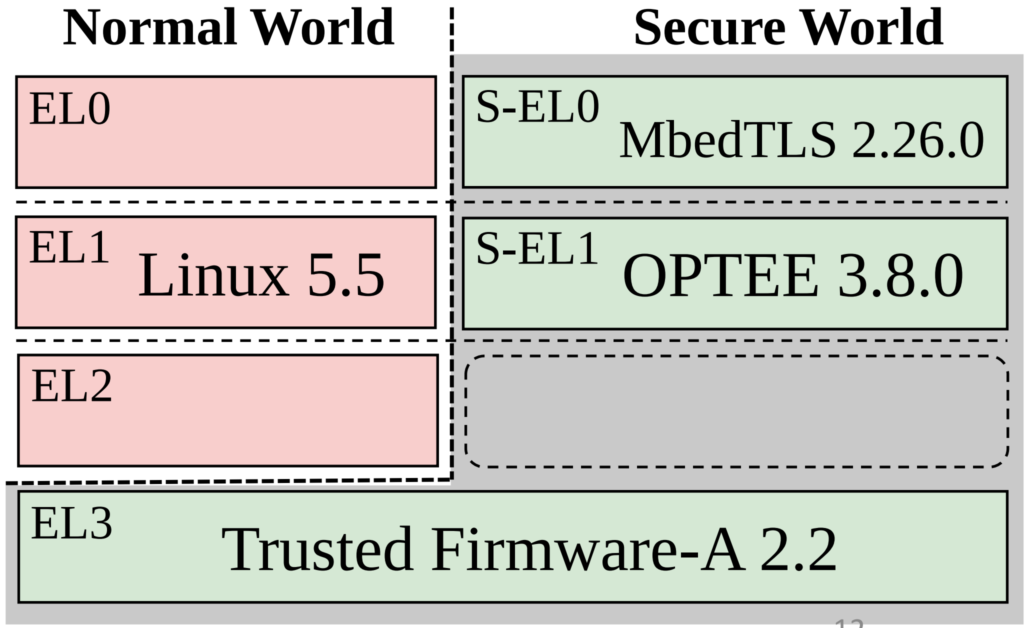
In Hardware,

Hikey960 board with Kirin 960 SoC

Arm big.LITTLE architecture,

4 Cortex A53s and 4 Cortex A73s
(533 - 1844 MHz) (903 - 2362 MHz)





Load-Step

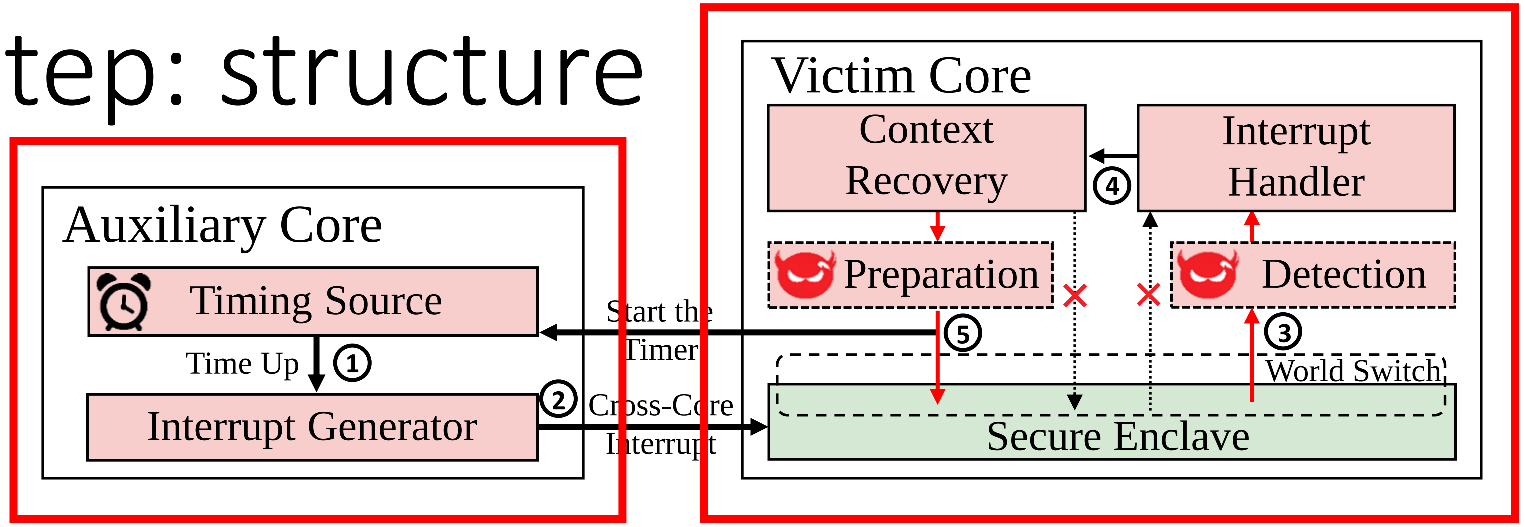
- A high precision framework to control the execution of TrustZone
    - Periodically generate interrupt forward to secure world
    - Detect μarch side-channels for every interrupt epochs

- Two challenges
    - Framework implementation
        - Stable with Low-noise
    - Timing source
        - High interrupt frequency

Load-Step: design

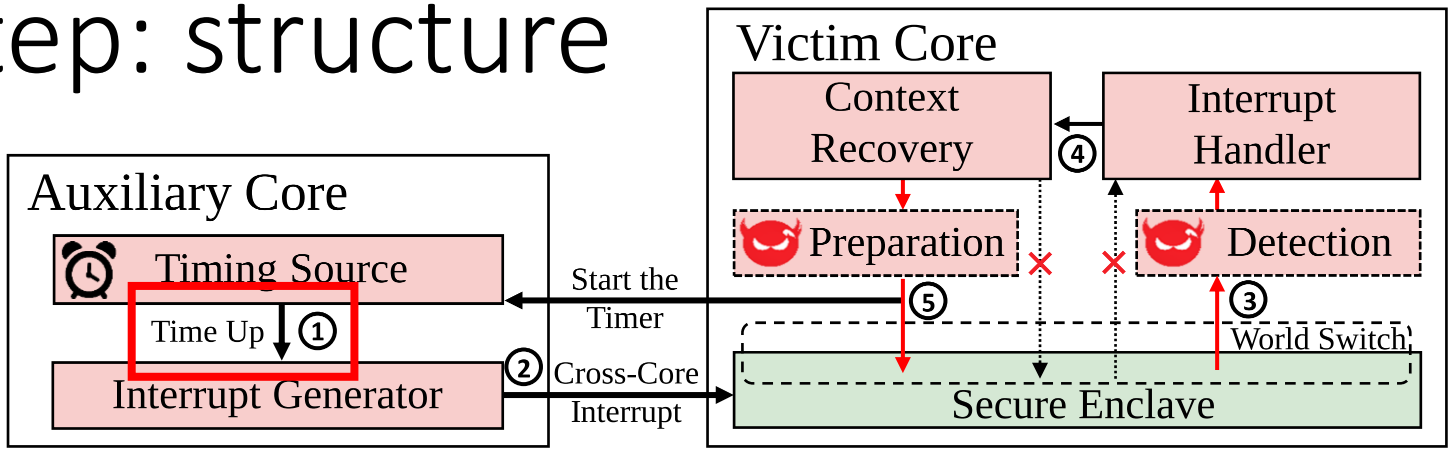
- Designed as an external kernel module
    - Exchange some kernel function, e.g., *irq_handler()*
- Cross-core interrupt instead of self-core interrupt
    - TrustZone “occupy” the whole physical core

Load-Step: structure



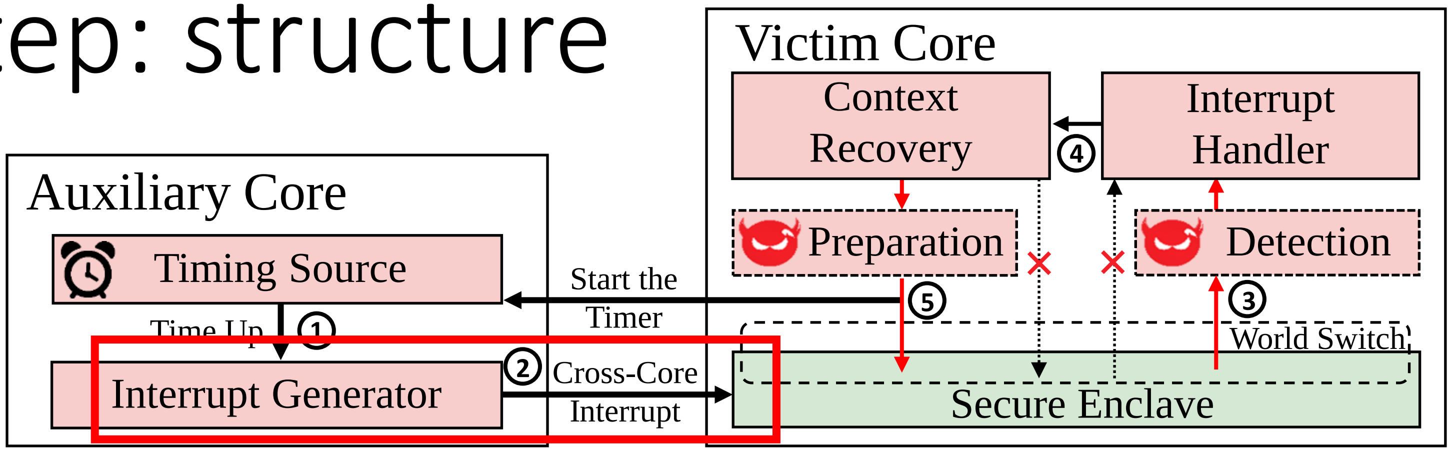
- Two-core framework
  - Auxiliary core
  - Victim core

Load-Step: structure



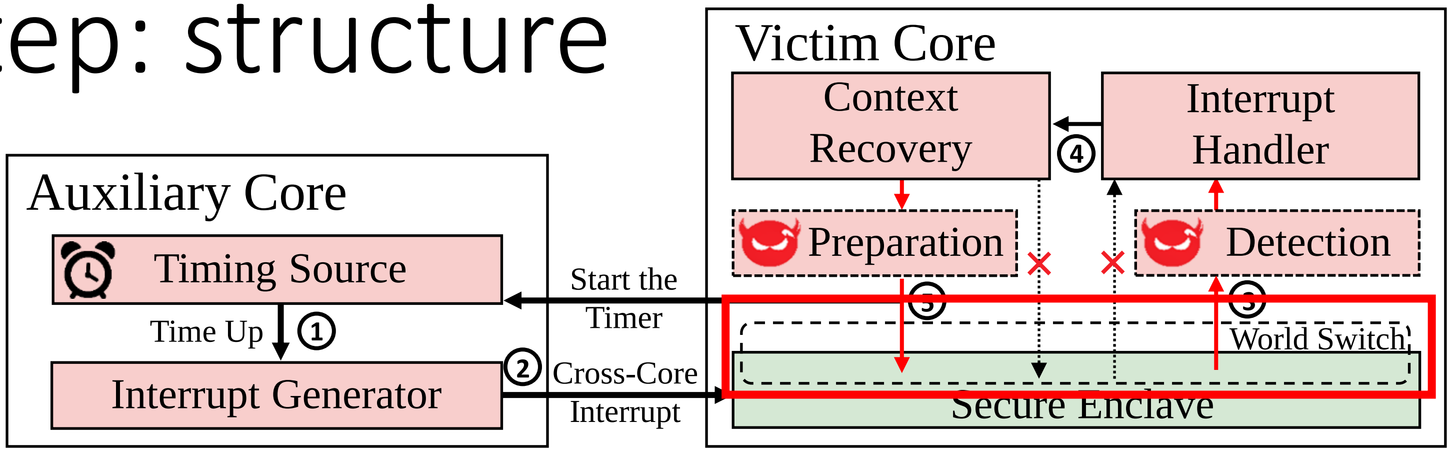
- Auxiliary core receives a time-up even from “timing source”
    - periodically

Load-Step: structure



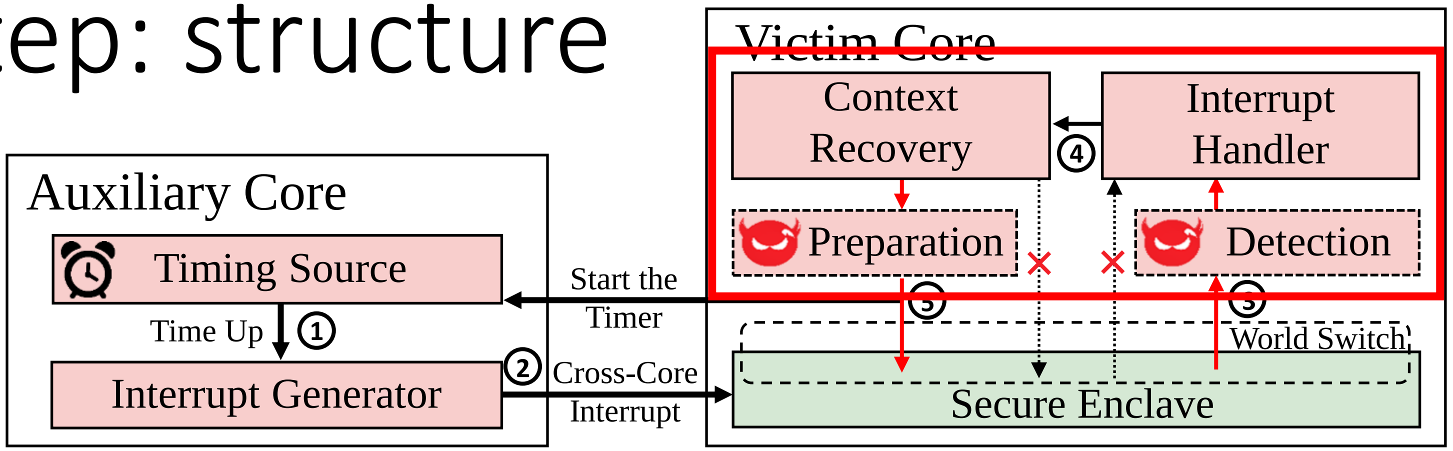
- Generate a cross-core interrupt (IRQ) forwarding to victim core
  - Achieved by Arm Generic Interrupt Controller (Arm-GIC)
  - This IRQ is an “insecure IRQ” (IRQ from normal world)

Load-Step: structure



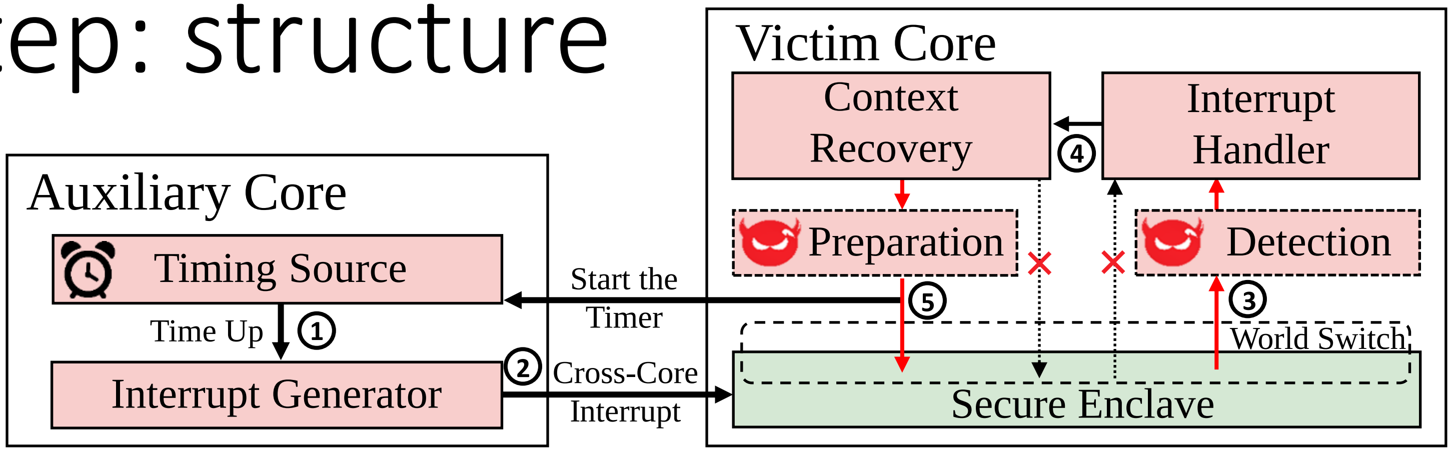
- “Insecure IRQ” must be handled by normal world!
  - World Switch happens

Load-Step: structure



- Preparation block
  - Prepare, pre-train µarch components, if needed
- Detection block
  - Collect data in µarch side-channel

Load-Step: structure



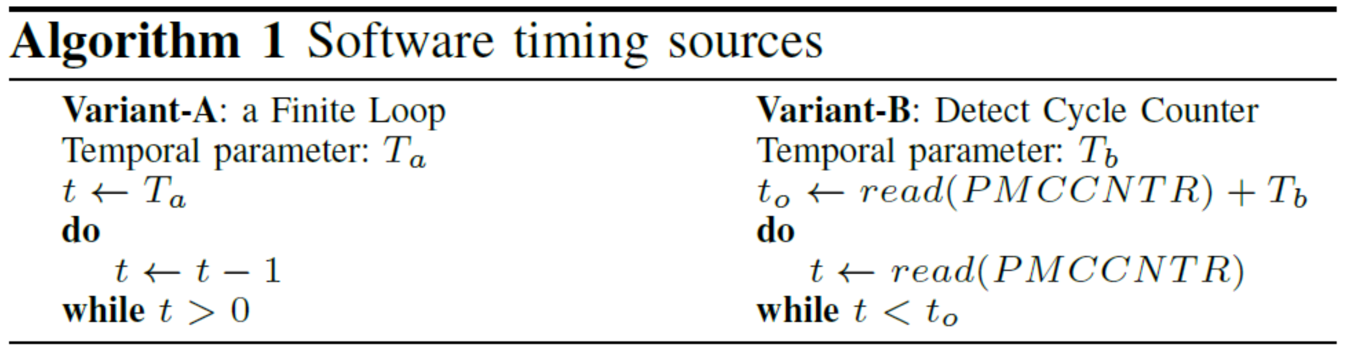
- Overview
  - Timing source is essential
    - Frequency determines “temporal resolution”
    - Stability determines “precision”

Timing source

- Hardware Timers
  - Exist in each core
  - Generate Hardware IRQ once time up
  - Frequency is usually fixed, typically ranges from 1MHz-50MHz

Timing source

- Software Timers
  - Finite count down loop
  - Check “cycle counter” of core



Algorithm 1 Software timing sources

```
**Variant-A**: a Finite Loop
Temporal parameter: $T_a$
$t \leftarrow T_a$
**do**
    $t \leftarrow t - 1$
**while** $t > 0$
```

```
**Variant-B**: Detect Cycle Counter
Temporal parameter: $T_b$
$t_o \leftarrow read(PMCCNTR) + T_b$
**do**
    $t \leftarrow read(PMCCNTR)$
**while** $t < t_o$
```

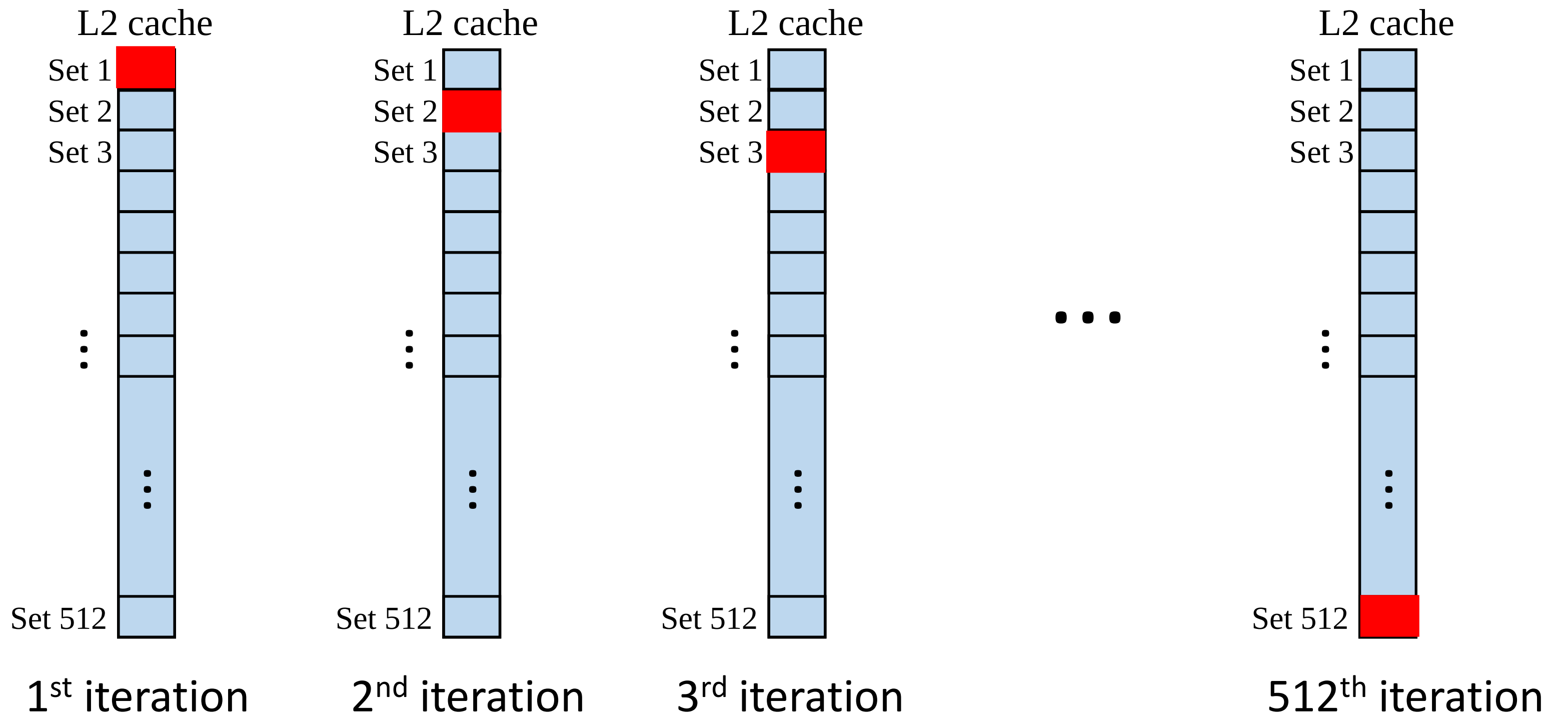
Timing source

- Hardware or Software Timers?

	Reliability	**Temporal Resolution**
Hardware Timer	Few jitters	200 ns to 1000 ns (Ours: 500 ns)
Software Timer	More software jitters	1 ns in a 1 GHz core

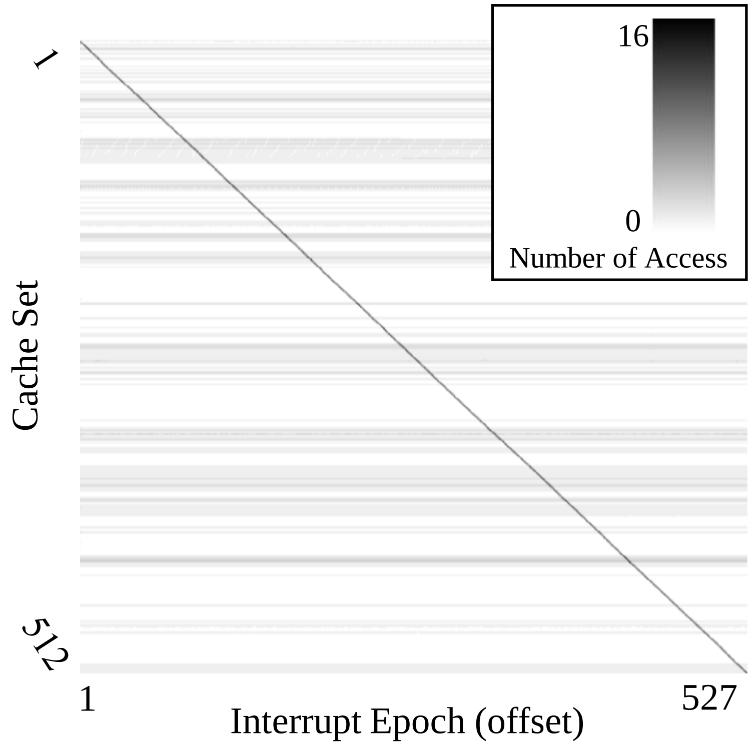
Temporal Resolution

- Benchmark trusted application
  - Load data that maps to every cache set, in order



Temporal Resolution

- Prime+Probe reinforced by Load-Step
  - Detection block: “Probe” every L2 cache sets
  - Preparation block: “Prime” every L2 cache sets
- CPU: 512 sets with 16 way set associative
  - Load 16 data per iteration, totally 512 iterations

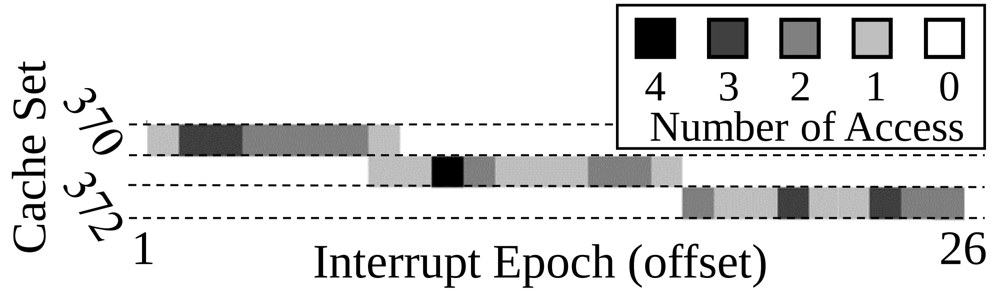


IRQ interval: $T_H = \mathbf{51}(2MHz)^{-1}$

Temporal Resolution: ≈ 16 loads

Temporal Resolution

- Hardware Timer



IRQ interval: $T_H = \mathbf{41}(2MHz)^{-1}$

Temporal Resolution: ≈ **2 loads**

- Decrease time parameter? 40? 39?
  - Endless loop
    - interrupt interval is even less than the time of world switch!

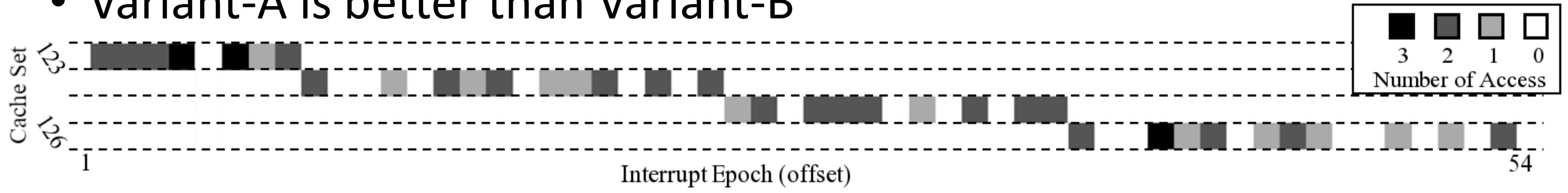
Temporal Resolution

Algorithm 1 Software timing sources

```
**Variant-A**: a Finite Loop
Temporal parameter: $T_a$
$t \leftarrow T_a$
do
    $t \leftarrow t - 1$
while $t > 0$
```

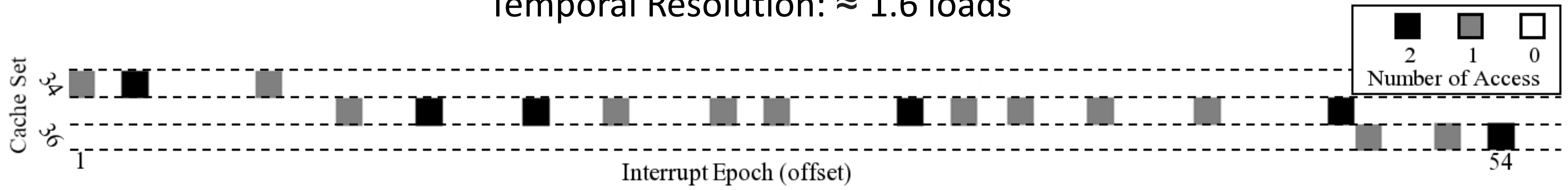
```
**Variant-B**: Detect Cycle Counter
Temporal parameter: $T_b$
$t_o \leftarrow read(PMCCNTR) + T_b$
do
    $t \leftarrow read(PMCCNTR)$
while $t < t_o$
```

- Software Timer
  - Variant-A is better than Variant-B



Variant-B with IRQ interval: $T_b = \mathbf{680}$

Temporal Resolution: ≈ 1.6 loads

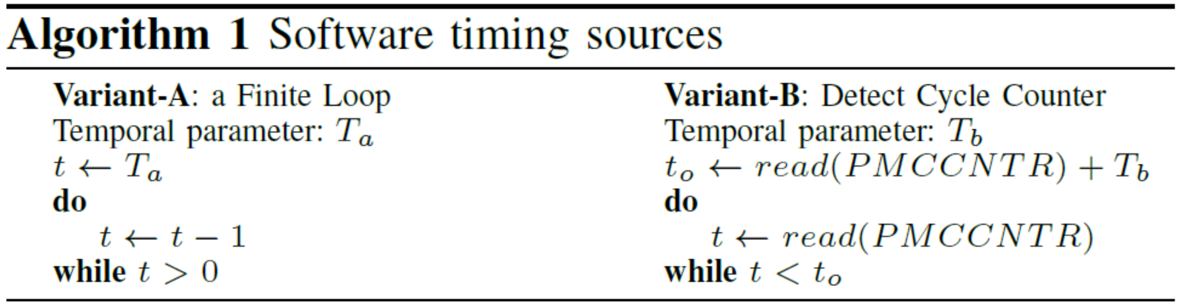


Variant-A with IRQ interval: $T_b = \mathbf{515}$

Temporal Resolution: ≈ 1.3 loads

Temporal Resolution

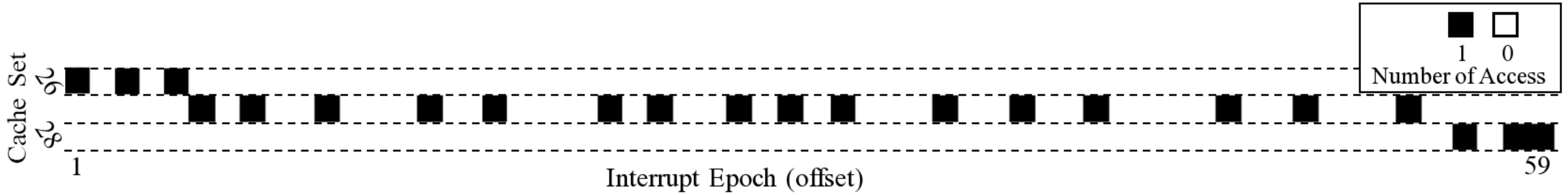
- Software Timer



Algorithm 1 Software timing sources

```
Variant-A: a Finite Loop
Temporal parameter: $T_a$
$t \leftarrow T_a$
do
    $t \leftarrow t - 1$
while $t > 0$
```

```
Variant-B: Detect Cycle Counter
Temporal parameter: $T_b$
$t_o \leftarrow read(PMCCNTR) + T_b$
do
    $t \leftarrow read(PMCCNTR)$
while $t < t_o$
```



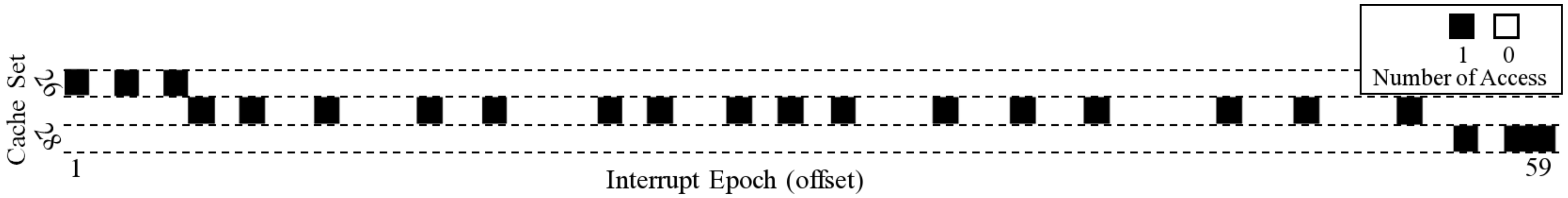
Variant-A with IRQ interval: $T_b = \mathbf{505}$

Temporal Resolution: **1 loads per interrupt**

Temporal Resolution

- Achieve load-instruction precision

- Higher precision?
  - Hardware timer: not enough temporal resolution
  - Software timer: affect by software jitters



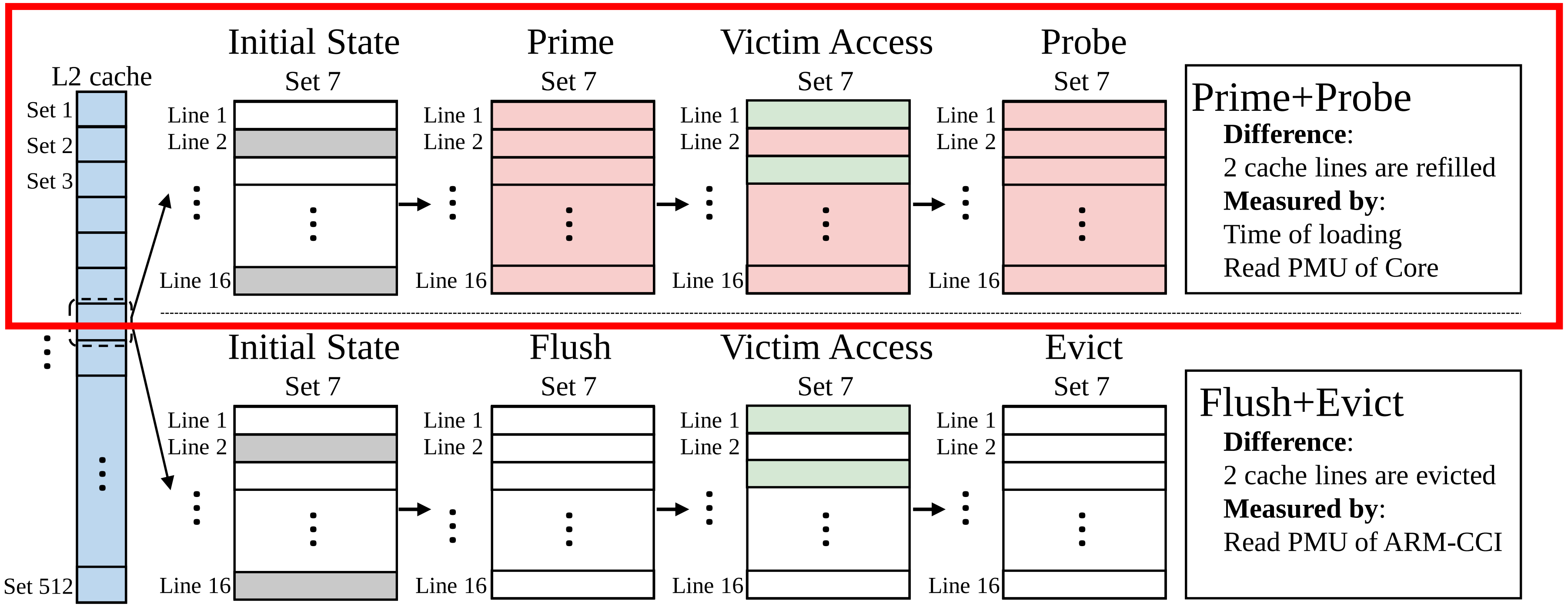
Temporal Resolution: **1 loads per interrupt**

Flush+Evict

- Arm-specific instruction: “DC CISW”
  - Clear and invalidate cache line by set/way
  - Flush cache lines without sharing the victim’s memory space

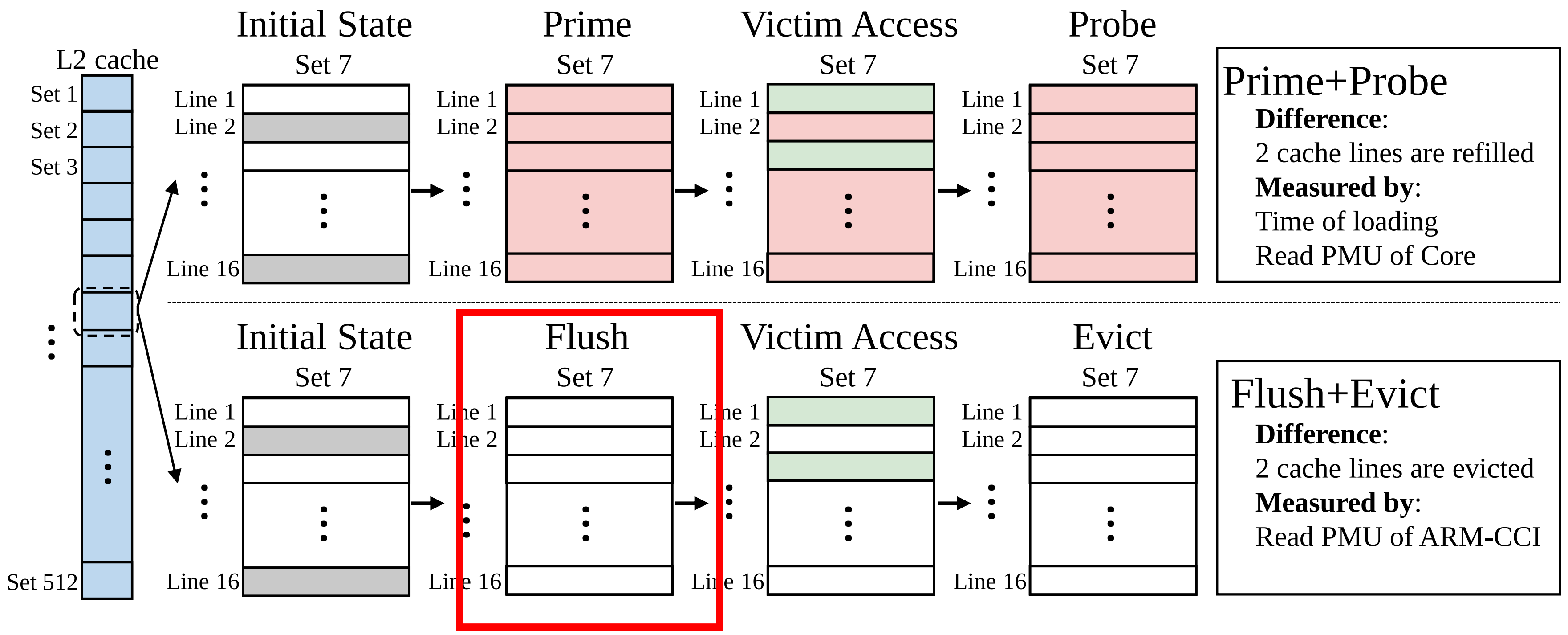
- Flushed cache lines emit “evict transaction”
  - Counted by Arm Cache Coherent Interconnect (Arm-CCI)

Flush+Evict



Prime+Probe attack

Flush+Evict

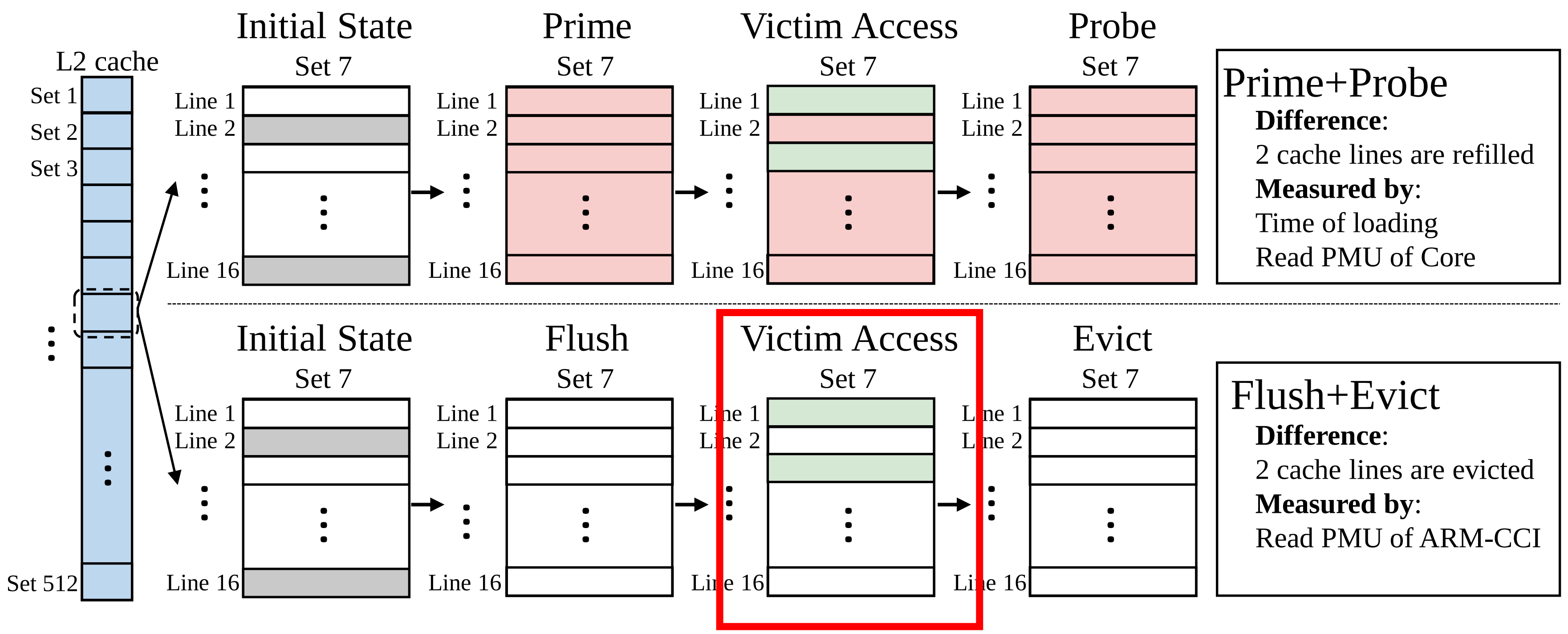


Prime+Probe
Difference:
2 cache lines are refilled
Measured by:
Time of loading
Read PMU of Core

Flush+Evict
Difference:
2 cache lines are evicted
Measured by:
Read PMU of ARM-CCI

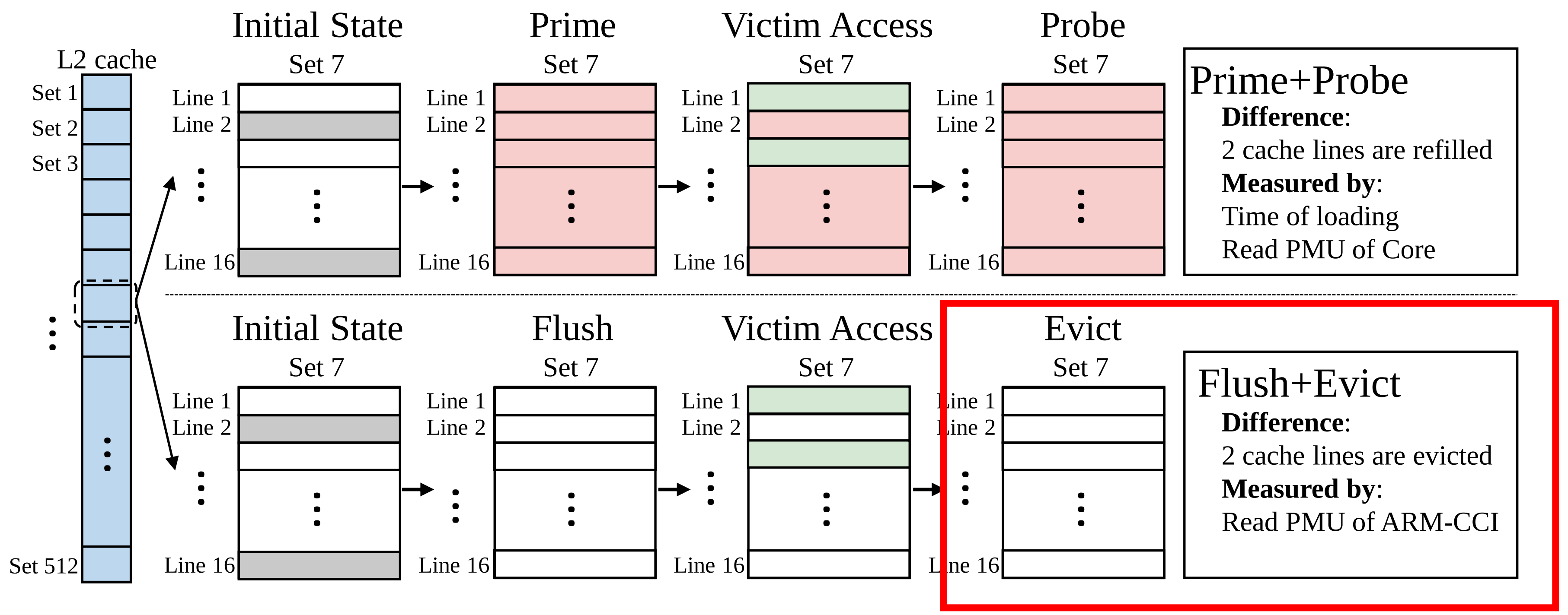
Flush a cache set by "DC CISW"

Flush+Evict



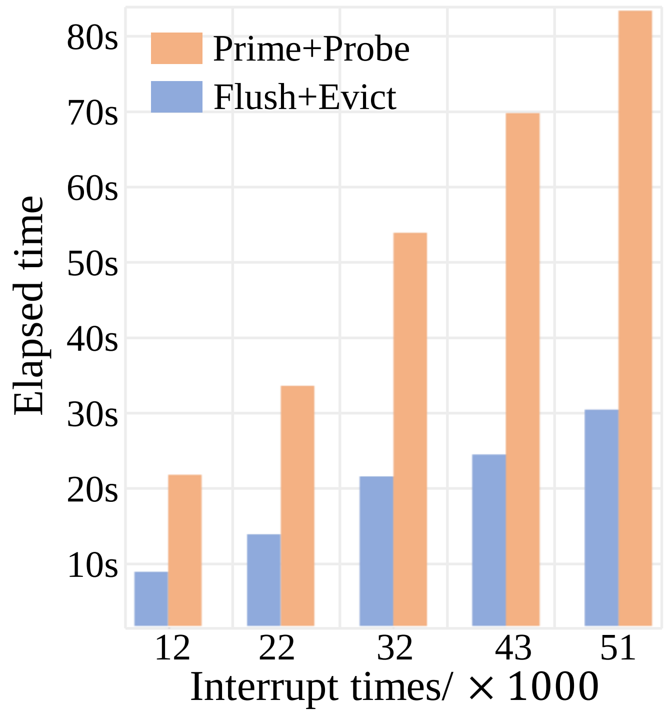
Victim's activity leaves some cache lines here

Flush+Evict

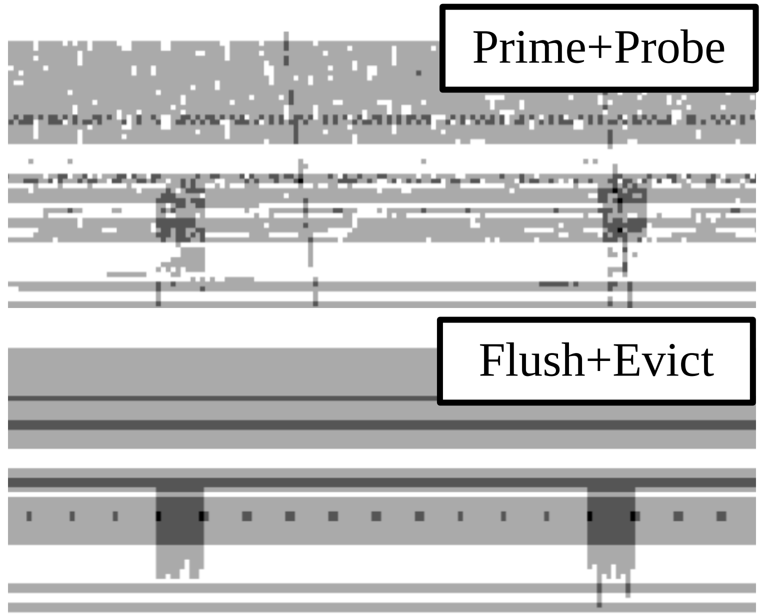


Flush a cache set by "DC CISW", again,
And count the "evict transaction" events

Flush+Evict



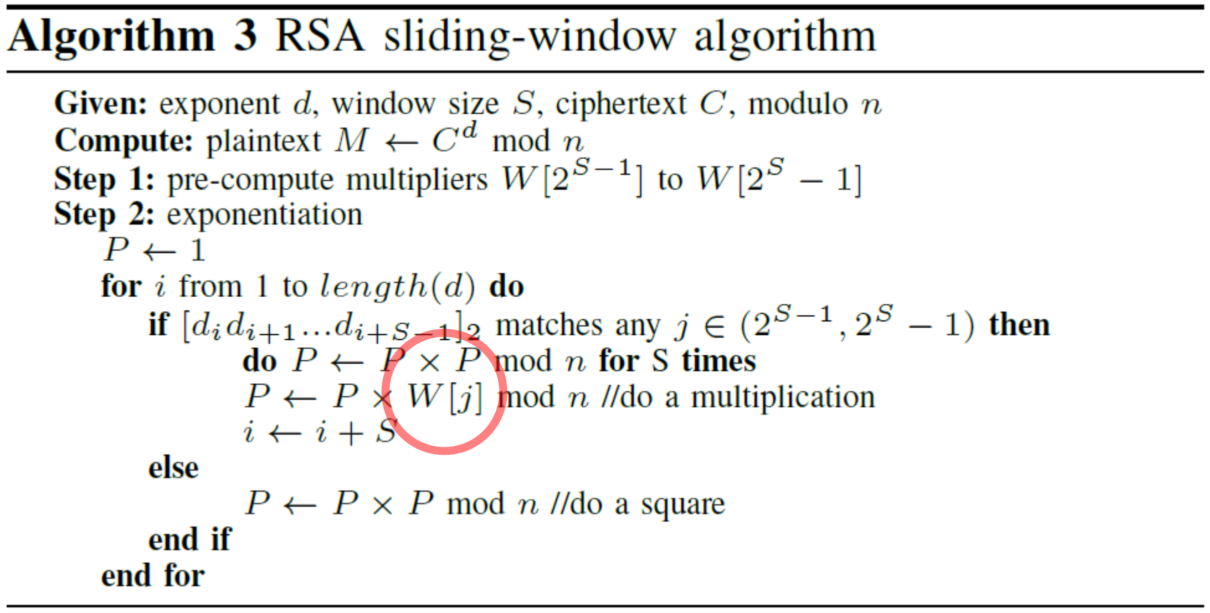
Faster profiling speed



Lower profiling noise

Attack RSA in MbedTLS

- RSA sliding window algorithm suffers from cache side-channel attacks [8][9]



Algorithm 3 RSA sliding-window algorithm

```
Given: exponent $d$, window size $S$, ciphertext $C$, modulo $n$
Compute: plaintext $M \leftarrow C^d \bmod n$
Step 1: pre-compute multipliers $W[2^{S-1}]$ to $W[2^S - 1]$
Step 2: exponentiation
    $P \leftarrow 1$
    for $i$ from 1 to $length(d)$ do
        if $[d_i d_{i+1} \dots d_{i+S-1}]_2$ matches any $j \in (2^{S-1}, 2^S - 1)$ then
                do $P \leftarrow P \times P \bmod n$ for S times
                $P \leftarrow P \times W[j] \bmod n$ //do a multiplication
                $i \leftarrow i + S$
        else
                $P \leftarrow P \times P \bmod n$ //do a square
        end if
    end for
```

Attack RSA in MbedTLS

- “Exponent Blinding” (k=64bit) is imported into MbedTLS
  - K bits of exponent is randomized for every decryption
  - [9] profiles 11 traces to recover the full key, **now need $\mathbf{3.27 \times 10^{17}}$ traces**

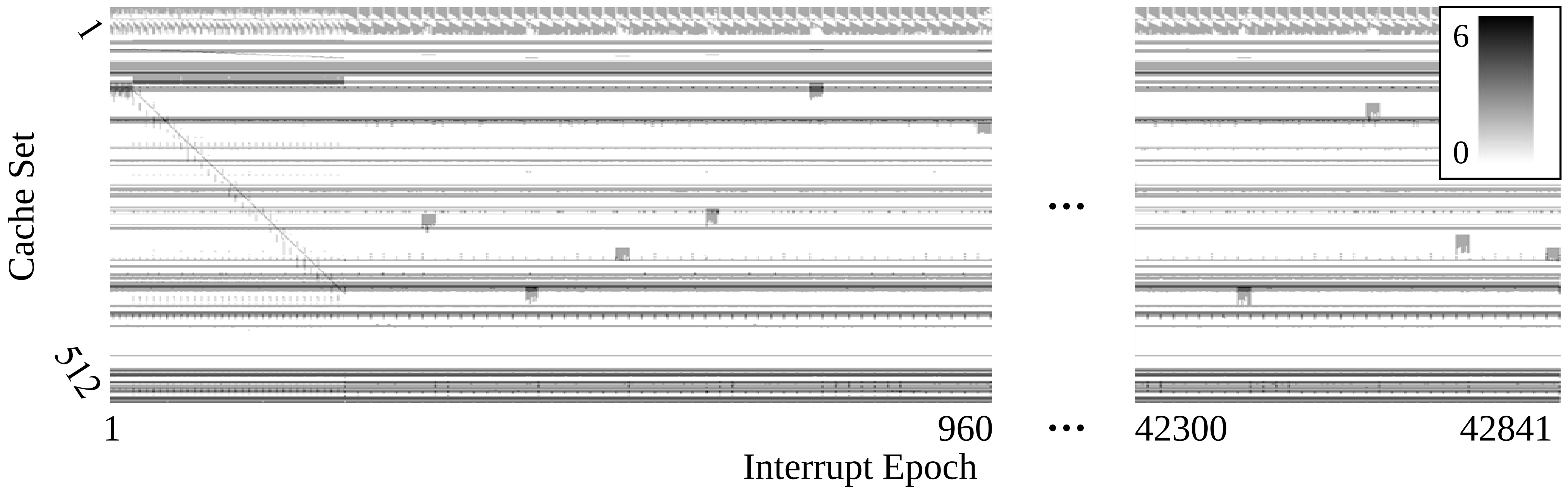
- “Exponent Blinding” completely fails if:
  - Side-channel attacker can figure out the full by single trace

Attack RSA in MbedTLS

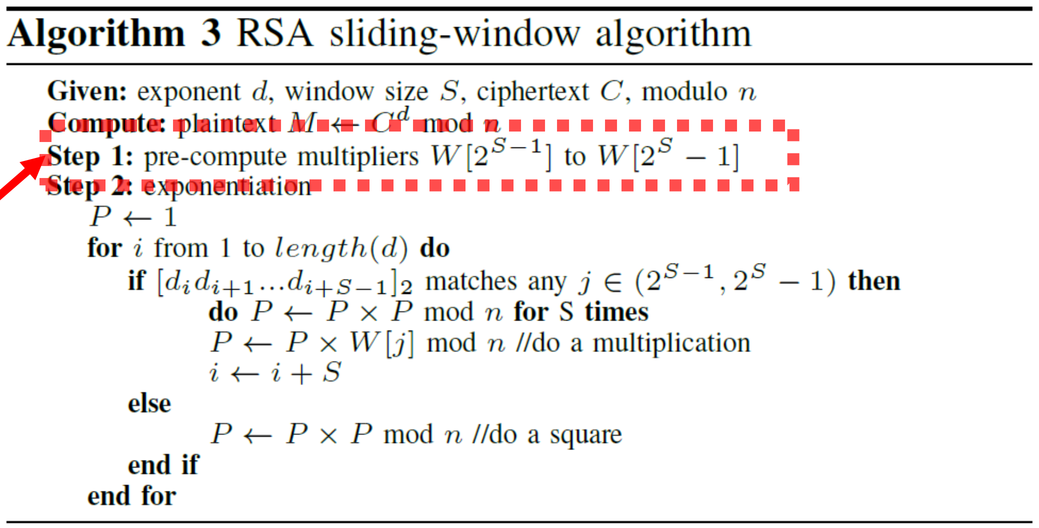
- Software configurations
  - RSA decryption in MbedTLS 2.22.0
  - 4096-bit key size
  - ***window_size = 6***
  - “exponent blinding” is enabled

Attack RSA in MbedTLS

- Load-Step & Flush+Evict



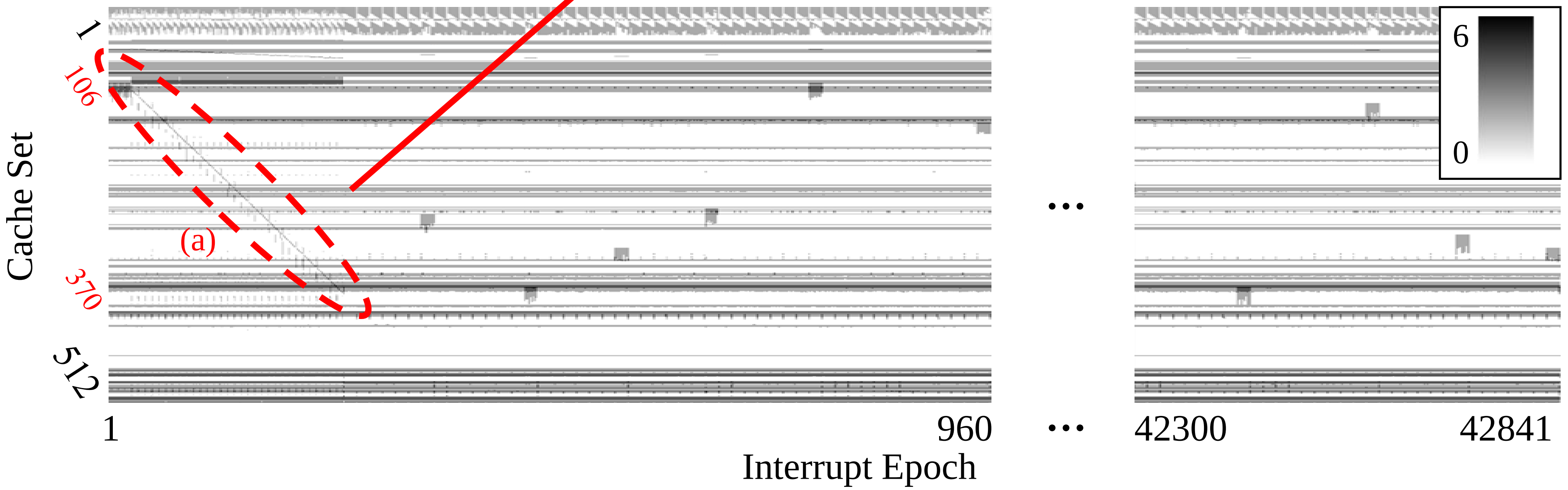
Attack RSA in MbedTLS



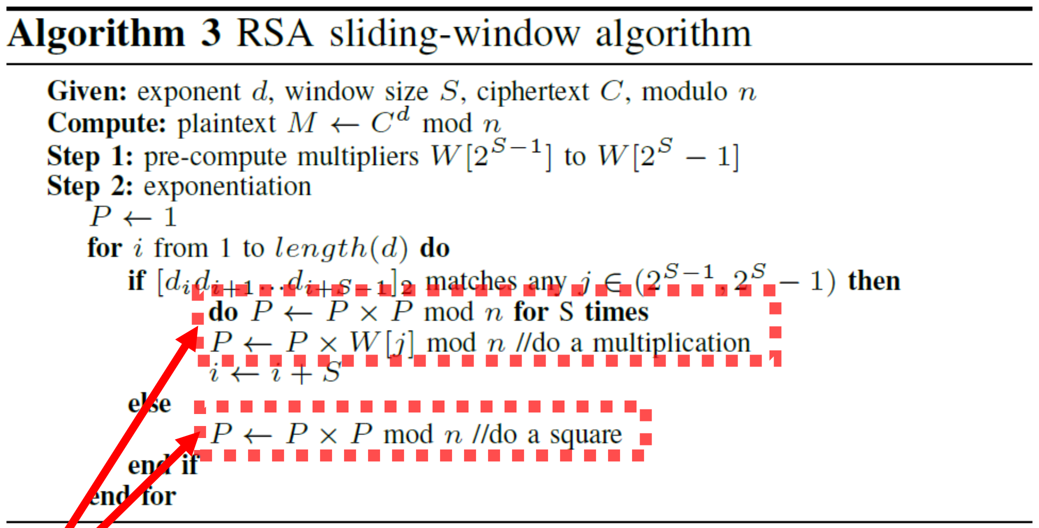
Algorithm 3 RSA sliding-window algorithm

```
Given: exponent $d$, window size $S$, ciphertext $C$, modulo $n$
Compute: plaintext $M \leftarrow C^d \bmod n$
Step 1: pre-compute multipliers $W[2^{S-1}]$ to $W[2^S - 1]$
Step 2: exponentiation
    $P \leftarrow 1$
    for $i$ from 1 to $length(d)$ do
        if $[d_i d_{i+1} \ldots d_{i+S-1}]_2$ matches any $j \in (2^{S-1}, 2^S - 1)$ then
            do $P \leftarrow P \times P \bmod n$ for S times
            $P \leftarrow P \times W[j] \bmod n$ //do a multiplication
            $i \leftarrow i + S$
        else
            $P \leftarrow P \times P \bmod n$ //do a square
        end if
    end for
```

- Pattern of pre-compute



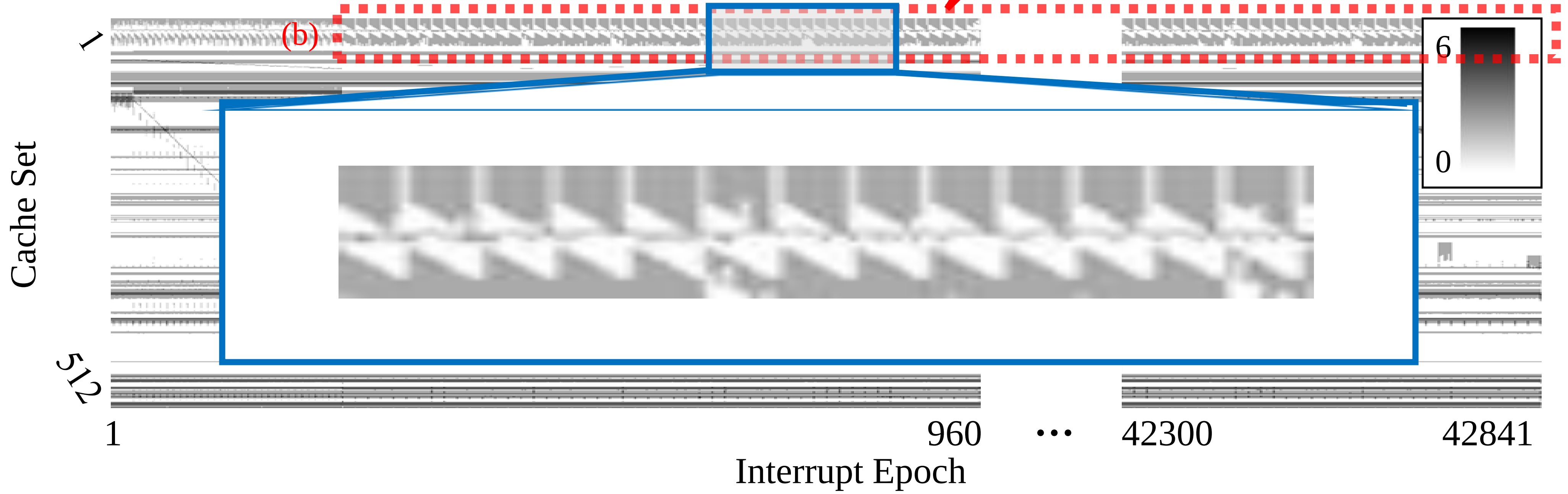
Attack RSA in MbedTLS



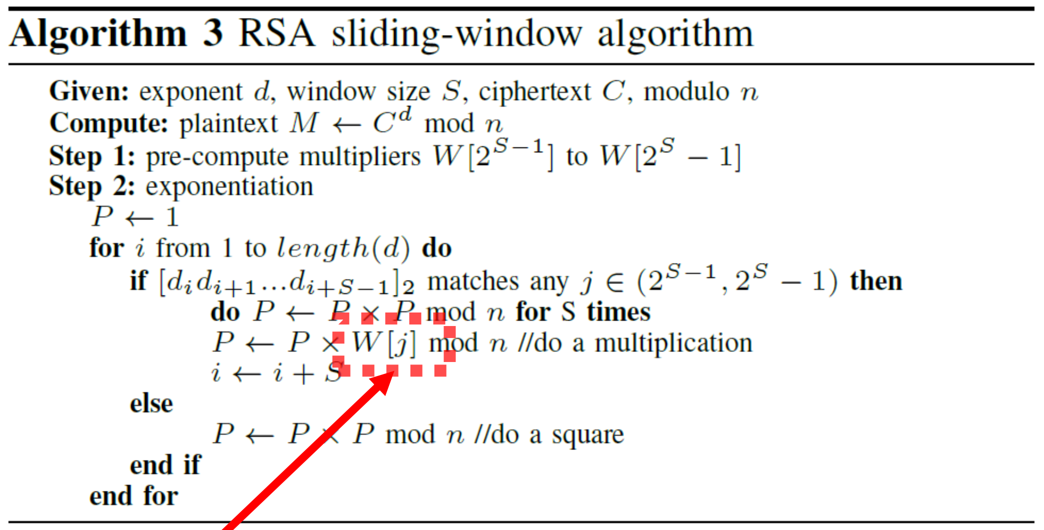
Algorithm 3 RSA sliding-window algorithm

```
Given: exponent $d$, window size $S$, ciphertext $C$, modulo $n$
Compute: plaintext $M \leftarrow C^d$ mod $n$
Step 1: pre-compute multipliers $W[2^{S-1}]$ to $W[2^S - 1]$
Step 2: exponentiation
    $P \leftarrow 1$
    for $i$ from 1 to $length(d)$ do
        if $[d_i d_{i+1} \dots d_{i+S-1}]_2$ matches any $j \in (2^{S-1}, 2^S - 1)$ then
            do $P \leftarrow P \times P$ mod $n$ for S times
            $P \leftarrow P \times W[j]$ mod $n$ //do a multiplication
            $i \leftarrow i + S$
        else
            $P \leftarrow P \times P$ mod $n$ //do a square
        end if
    end for
```

- Pattern of *multiply()*



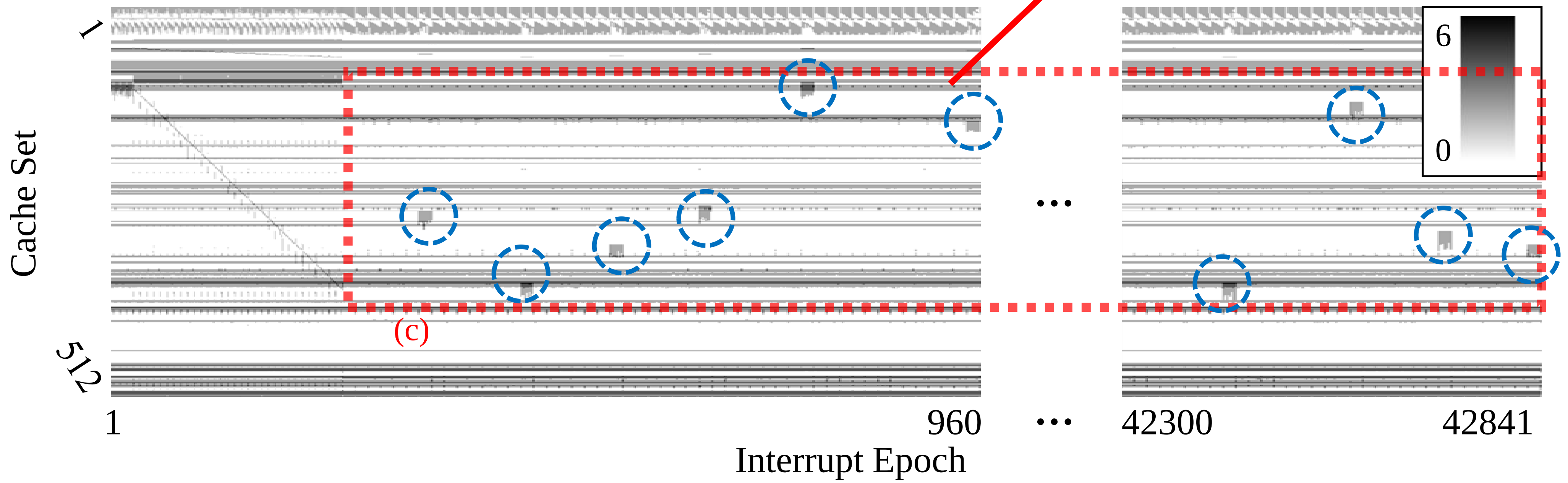
Attack RSA in MbedTLS



Algorithm 3 RSA sliding-window algorithm

```
Given: exponent $d$, window size $S$, ciphertext $C$, modulo $n$
Compute: plaintext $M \leftarrow C^d \bmod n$
Step 1: pre-compute multipliers $W[2^{S-1}]$ to $W[2^S - 1]$
Step 2: exponentiation
    $P \leftarrow 1$
    for $i$ from 1 to $length(d)$ do
        if $[d_i d_{i+1} \ldots d_{i+S-1}]_2$ matches any $j \in (2^{S-1}, 2^S - 1)$ then
                do $P \leftarrow P \times P \bmod n$ for S times
                $P \leftarrow P \times W[j] \bmod n$ //do a multiplication
                $i \leftarrow i + S$
        else
                $P \leftarrow P \times P \bmod n$ //do a square
        end if
    end for
```

- Pattern of multipliers

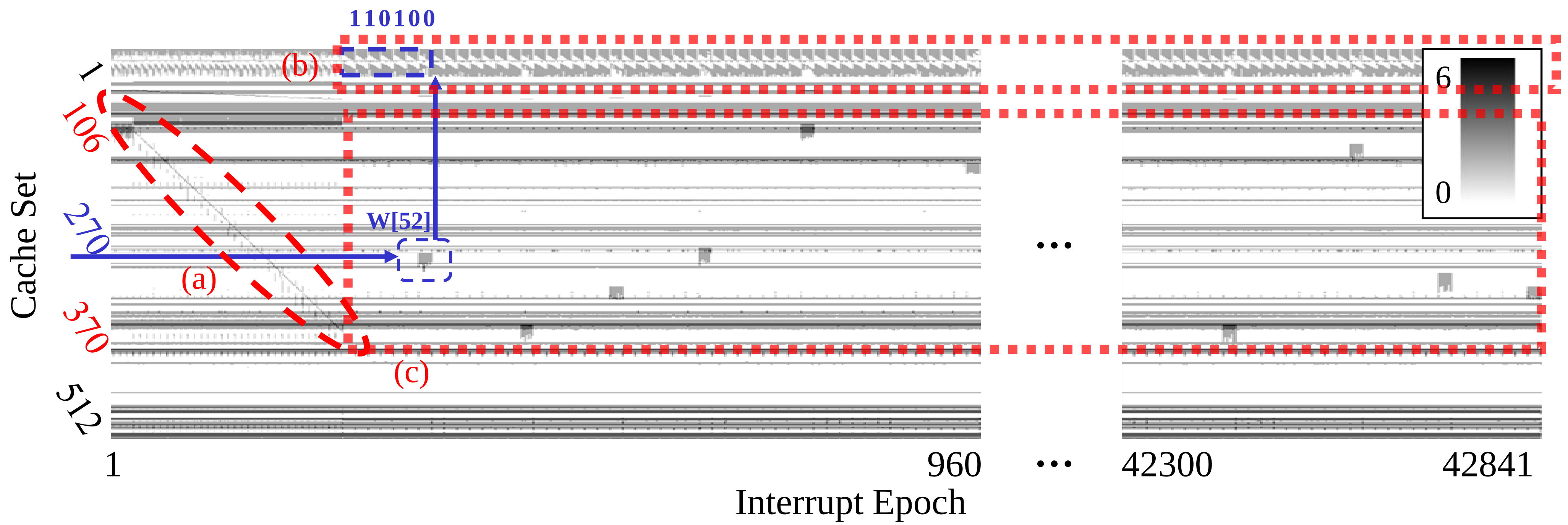


Attack RSA in MbedTLS

- "decode" one multiplier

Algorithm 3 RSA sliding-window algorithm

```
Given: exponent $d$, window size $S$, ciphertext $C$, modulo $n$
Compute: plaintext $M \leftarrow C^d \bmod n$
Step 1: pre-compute multipliers $W[2^{S-1}]$ to $W[2^S - 1]$
Step 2: exponentiation
    $P \leftarrow 1$
    for $i$ from 1 to $length(d)$ do
        if $[d_i d_{i+1} \ldots d_{i+S-1}]_2$ matches any $j \in (2^{S-1}, 2^S - 1)$ then
                do $P \leftarrow P \times P \bmod n$ for S times
                $P \leftarrow P \times W[j] \bmod n$ //do a multiplication
                $i \leftarrow i + S$
        else
                $P \leftarrow P \times P \bmod n$ //do a square
        end if
    end for
```

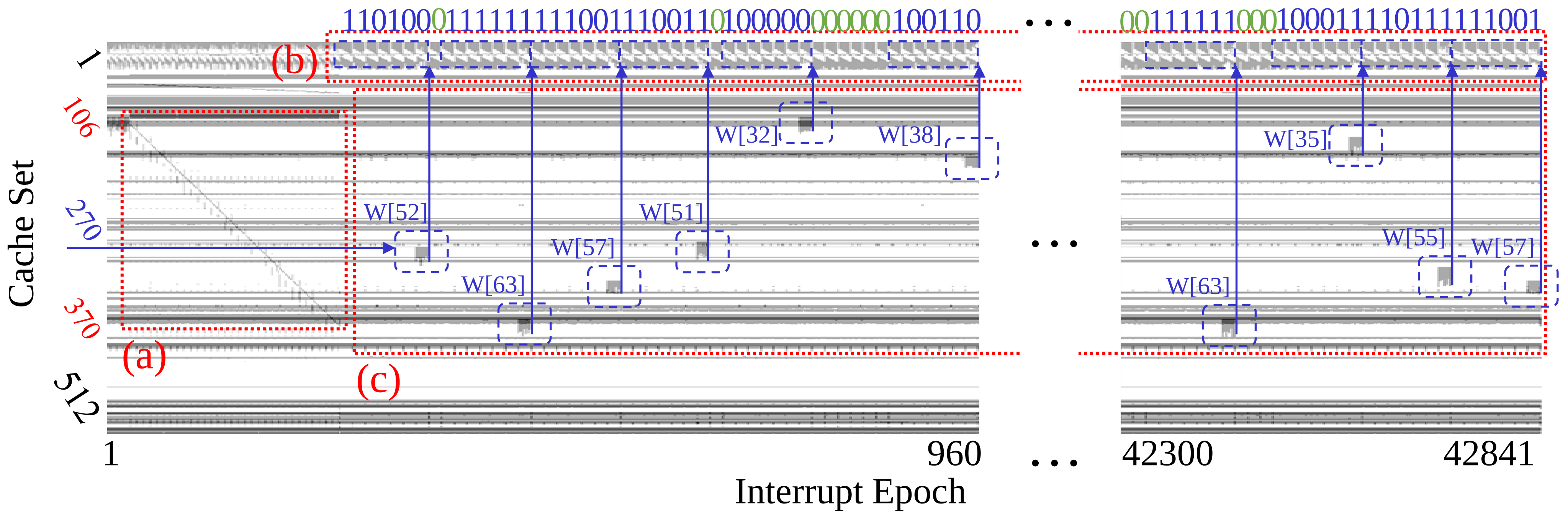


Attack RSA in MbedTLS

Algorithm 3 RSA sliding-window algorithm

```
Given: exponent $d$, window size $S$, ciphertext $C$, modulo $n$
Compute: plaintext $M \leftarrow C^d \bmod n$
Step 1: pre-compute multipliers $W[2^{S-1}]$ to $W[2^S - 1]$
Step 2: exponentiation
    $P \leftarrow 1$
    for $i$ from 1 to $length(d)$ do
        if $[d_i d_{i+1} \ldots d_{i+S-1}]_2$ matches any $j \in (2^{S-1}, 2^S - 1)$ then
            do $P \leftarrow P \times P \bmod n$ for S times
            $P \leftarrow P \times W[j] \bmod n$ //do a multiplication
            $i \leftarrow i + S$
        else
            $P \leftarrow P \times P \bmod n$ //do a square
        end if
    end for
```

- Decode all multipliers and 0's window



Attack RSA in MbedTLS

- performance

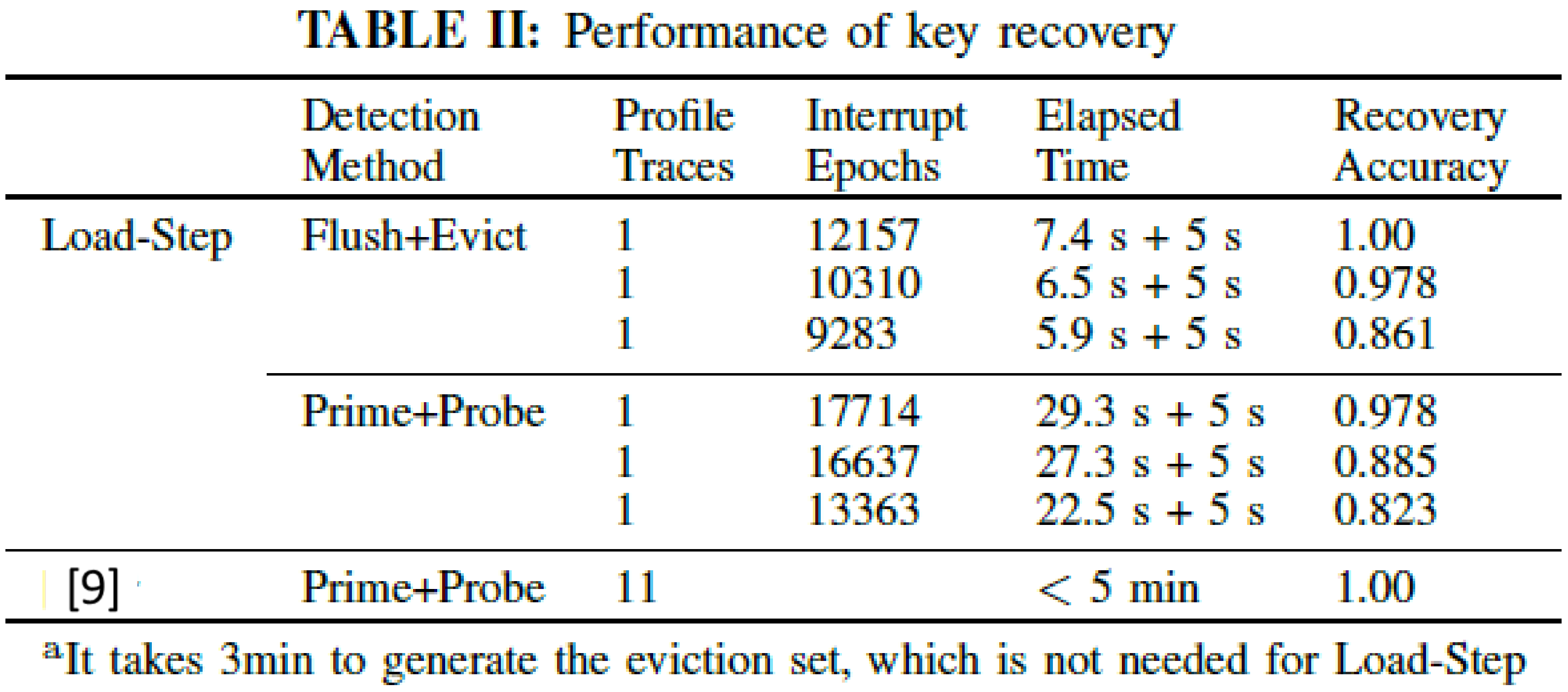


TABLE II: Performance of key recovery

	Detection Method	Profile Traces	Interrupt Epochs	Elapsed Time	Recovery Accuracy
Load-Step	Flush+Evict	1	12157	7.4 s + 5 s	1.00
		1	10310	6.5 s + 5 s	0.978
		1	9283	5.9 s + 5 s	0.861
	Prime+Probe	1	17714	29.3 s + 5 s	0.978
		1	16637	27.3 s + 5 s	0.885
		1	13363	22.5 s + 5 s	0.823
[9]	Prime+Probe	11		< 5 min	1.00

[a]It takes 3min to generate the eviction set, which is not needed for Load-Step

Conclusion

- Load-Step: a precise Arm TrustZone execution control framework
- Flush+Evict: a new Arm-specific cache side-channel attack
- Insights into μarch attacks on Arm TrustZone

Thank you~

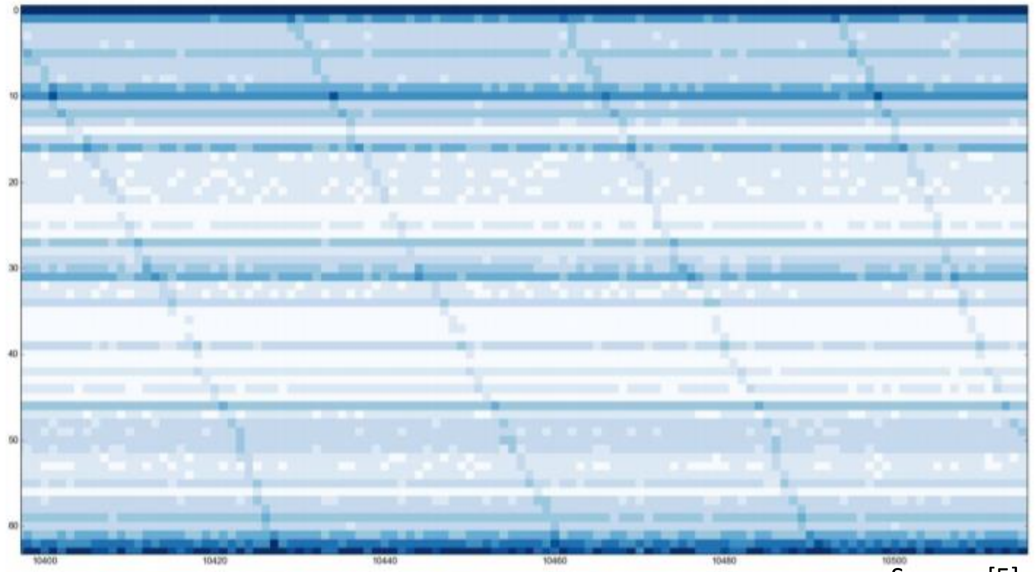
[1]J. Götzfried et all., Cache Attacks on Intel SGX. In Proceedings of the EuroSec'17. Association for Computing Machinery, New York, NY, USA, Article 2, 1–6.
[2] M. Lipp et al., Armageddon: Cache Attacks on Mobile Devices, in Proc. of USENIX Security, 2016.
[3] J. Bulck, et al., Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution, in USENIX Security 18, 2018, pp. 991–1008.
[4] J. Bulck, er al., SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In Proceedings of SysTEX'17, Article 4, 1–6.
[5] A. Moghimi et al., "Cachezoom: How SGX Amplifies the Power of Cache Attacks," in Proc. of CHES, 2017.
[6] D. Moghimi, et al., CopyCat: Controlled Instruction-Level Attacks on Enclaves. USENIX Security 20.
[7] N. Zhang et al., "TruSpy: Cache Side-channel Information Leakage from the Secure World on Arm Devices." Trans. on IACR Cryptol, 2016.
[8] F. Liu, et al., "Last-Level Cache Side-Channel Attacks are Practical," 2015 IEEE Symposium on Security and Privacy, 2015, pp. 605-622
[9] M. Schwarz et al., "Malware Guard Extension: Using SGX to Conceal Cache Attacks," in Proc. of DIMVA, 2017.

Reported by Zili KOU (zkou@connect.ust.hk)

Kernel Attacker on Intel SGX

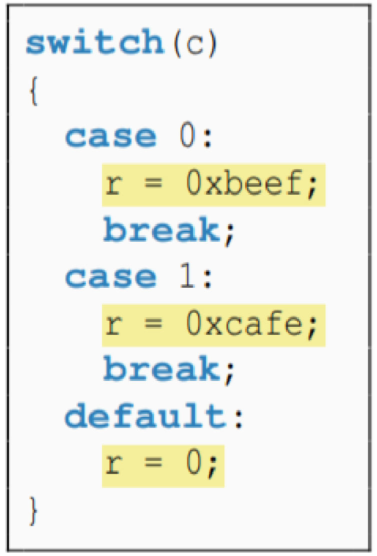
For every interrupt epoch, detect the µarch side-channels…

Cache (by CacheZoom)

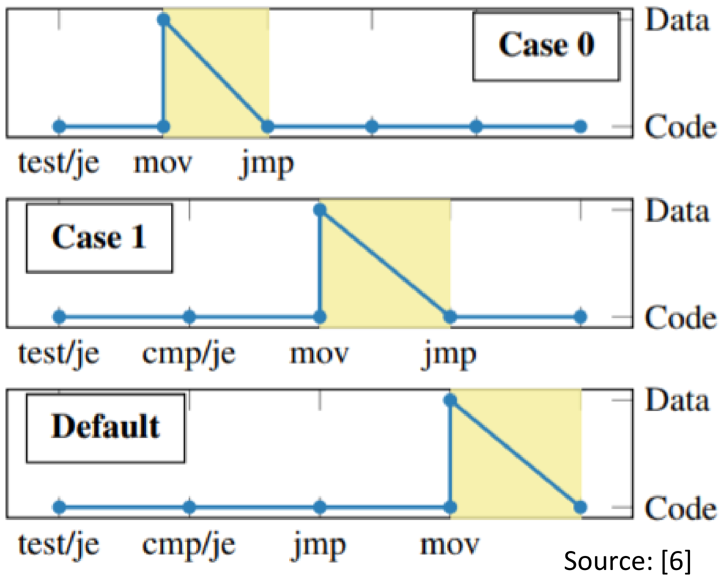


Source: [5]

Page tables (by COPYCAT)



```
switch(c)
{
  case 0:
    r = 0xbeef;
    break;
  case 1:
    r = 0xcafe;
    break;
  default:
    r = 0;
}
```



Source: [6]

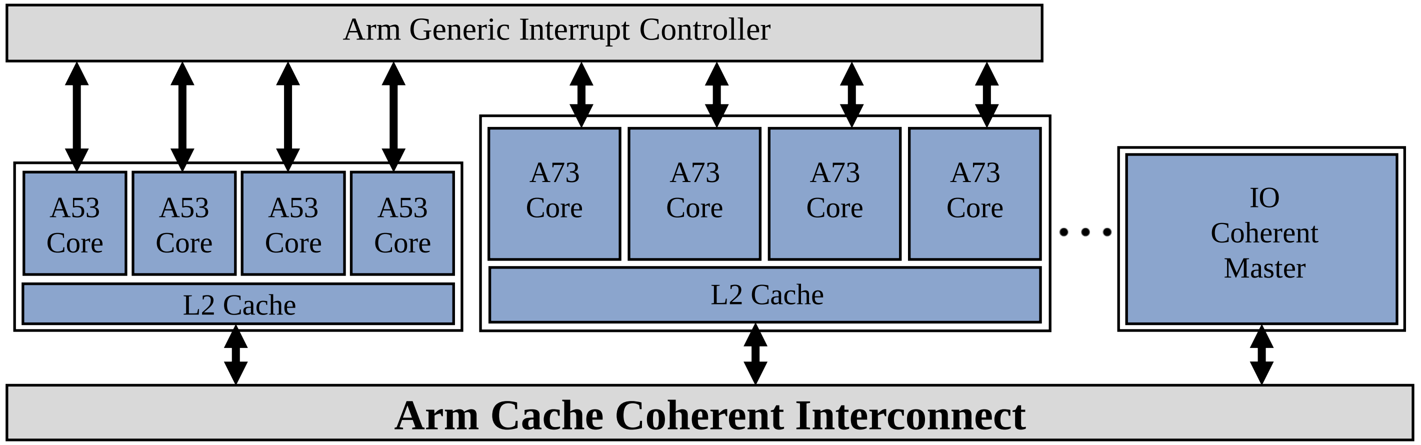
Flush-based cache side-channel attack

- Flush+Reload and Flush+Flush
  - No need to “fill” a cache set: faster and more precise than Prime+Probe
  - **Require shared memory space with victim (need to flush by address)**

Impossible! TEE isolates memory between attacker and victim!

Flush+Evict

- No timing difference, but emit a “evict transaction”
- Count the event of “evict transaction”
  - No such event in core’s performance counter
  - Counted by Arm Cache Coherent Interconnect (Arm-CCI)



Theories in Exponent Blinding:

Add a random number r to d for every decryption

C: ciphertext; **P**: plaintext; **d**: private key; **N**: modulus; **r**: random number (changes for every decryption)

$$P = C^{d} \, mod \, N$$

without blinding

with blinding

$$P' = C^{d+r} \, mod \, N$$

followed with some un-blinding steps…

If the r is 8 bytes length:

[9] can only leak the 96% value of the blinded key $d' = d + r$, which is randomized and different for every trace. Thus, it needs try $\mathbf{3.27 \times 10^{17}}$ times to get the desired 11 traces(in probability)

(Desired 11 traces means the traces generated by the same random number r)

Defense the cache side-channel attack well, as **most attacks cannot recover the key from a single trace**

Exponent Blinding failed when attacker can recover the key from a single trace:

If Attacker knows the values of **P**, **N**, **C**, **(d + r)**,
value of **d** can be calculated out by simply doing some factoring:

$$\begin{aligned} P &= C^{d} \, mod \, N \\ P' &= C^{(d+r)} \, mod \, N \\ P &= P' I mod \, N \\ I &= (C^{r})^{-1} \, mod \, N \end{aligned} \quad \Longrightarrow \quad d$$