

Micro-architectural Attacks 1

Milan Patnaik

Indian Institute of Technology Madras

Credits: Prof Chester Rebeiro and my colleagues of RISE Lab, IIT Madras

Agenda : Class

- **Cache Timing Attacks**
 - Cache Covert Channels.
 - Flush+Reload Attacks.
- **Cache Collision Attacks**
 - Prime+Probe Attacks.
 - Time Driven Attacks.
- **Case Studies**
 - Meltdown
 - Spectre
 - Rowhammer

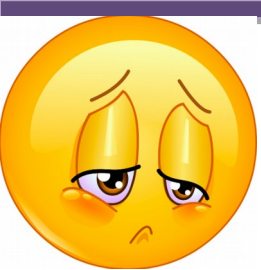
Agenda : Labs

- **Lab1.**
 - Cache Covert Channel.
- **Lab2.**
 - Cache Timing Attack.



Things we thought gave us security!

- Cryptography
- Passwords
- Information Flow Policies
- Privileged Rings
- ASLR
- Virtual Machines and confinement
- Javascript and HTML5
(due to restricted access to system
resources)
- Enclaves (SGX and Trustzone)



Micro-Architectural Attacks (can break all of this)

- Cryptography
- Passwords
- Information Flow Policies
- Privileged Rings
- ASLR
- Virtual Machines and confinement
- Javascript and HTML5
(due to restricted access to system resources)
- Enclaves (SGX and Trustzone)

Cache timing attack

Branch prediction attack

Speculation Attacks

Row hammer

Fault Injection Attacks

cold boot attacks

DRAM Row buffer (DRAMA)

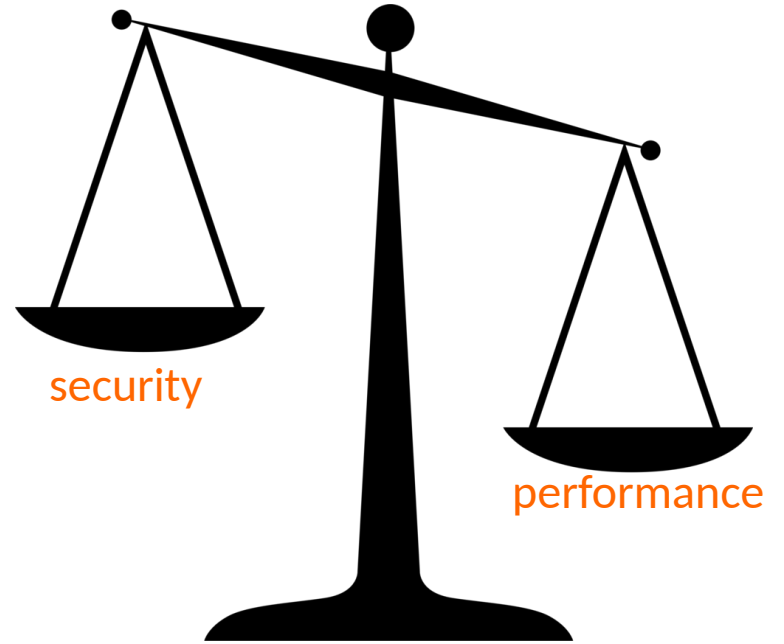
..... and many more

Causes

Most micro-architectural attacks caused by performance optimizations

Others due to inherent device properties

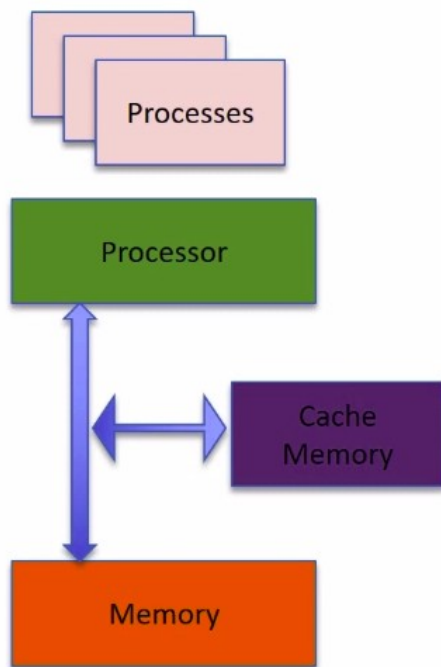
Third, due to stronger attackers



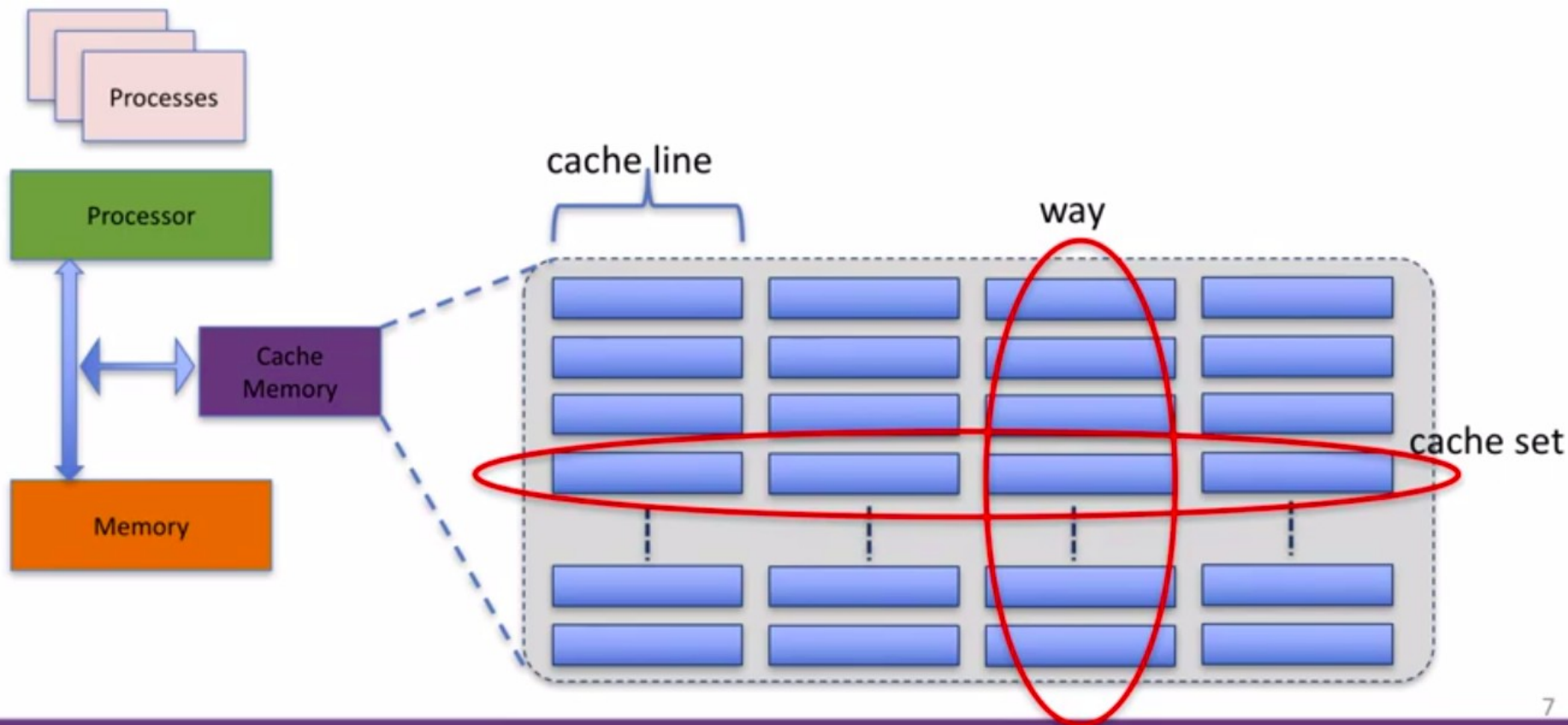
Cache Timing Attacks

Cache Covert Channels

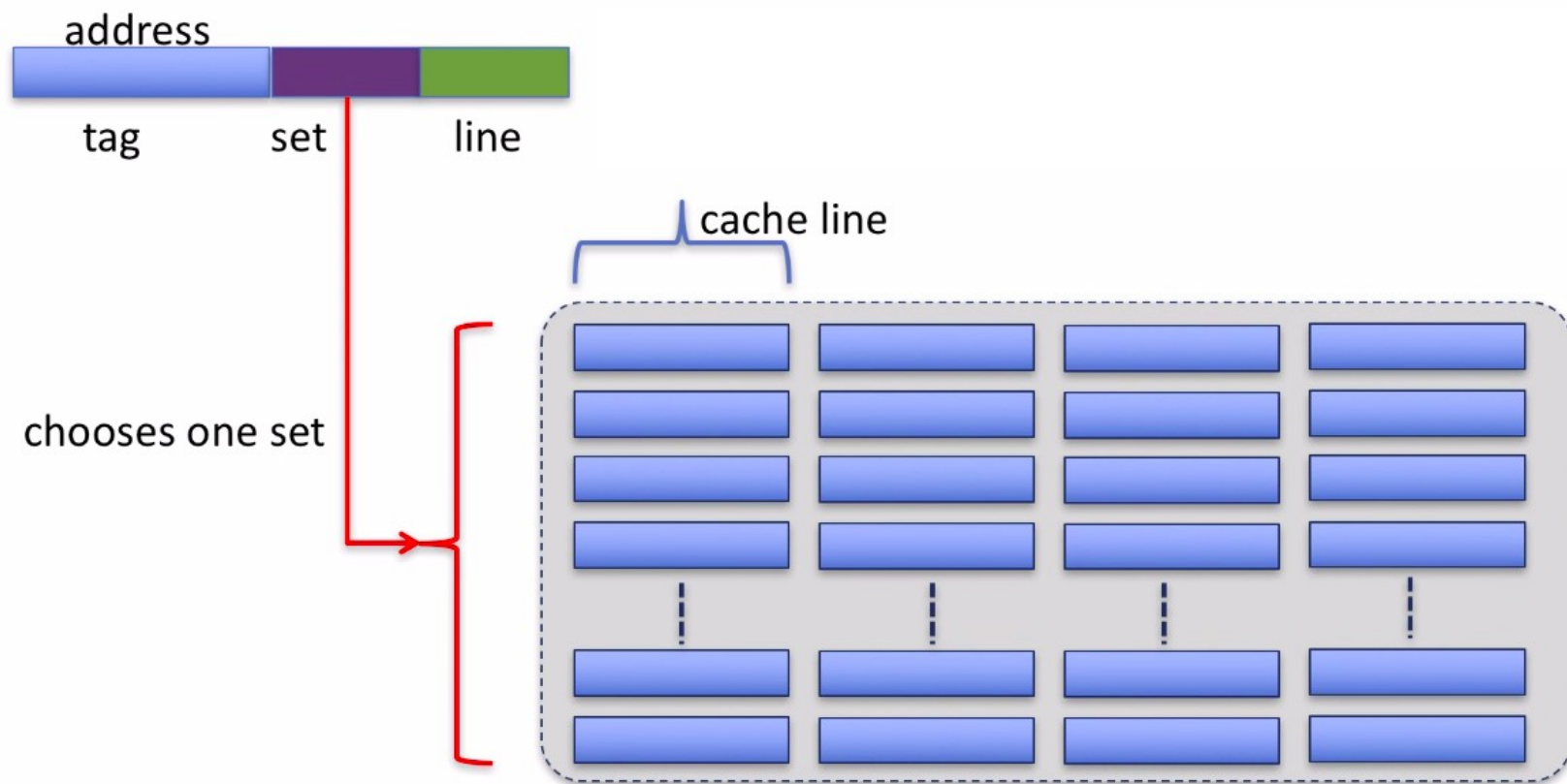
Cache Covert Channel



Cache Covert Channel



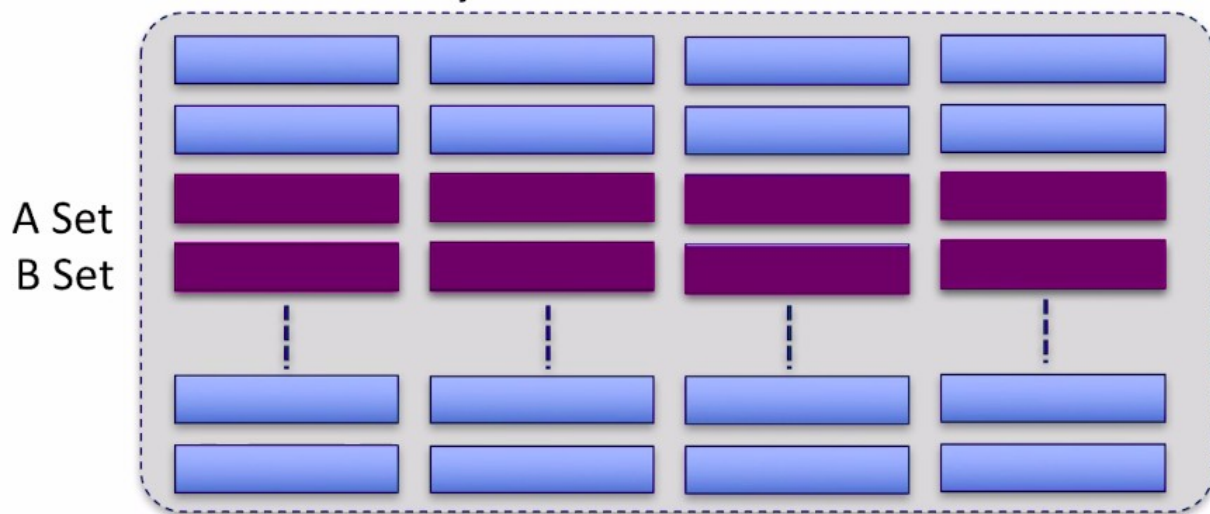
Cache Covert Channel



Cache Covert Channel

```
while(1){  
  load A1p2; load A2p2  
  load A3p2; load A4p2  
  load B1p2; load B2p2  
  load B3p2; load B4p2  
}
```

Process P2



Cache Covert Channel



statistically
time A ~ time B

```
while(1){
```

```
load A1p2; load A2p2
```

```
load A3p2; load A4p2
```

```
load B1p2; load B2p2
```

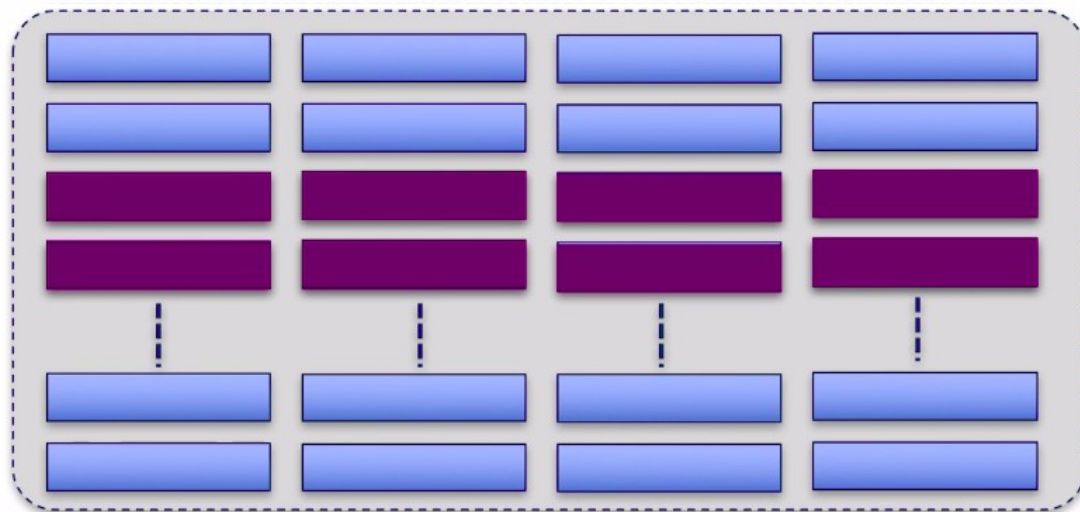
```
load B3p2; load B4p2
```

```
}
```

Process P2

A Set

B Set



Cache Covert Channel

Process P1

```
If (bit == 1)
  load Ap1
Else
  load Bp1
```

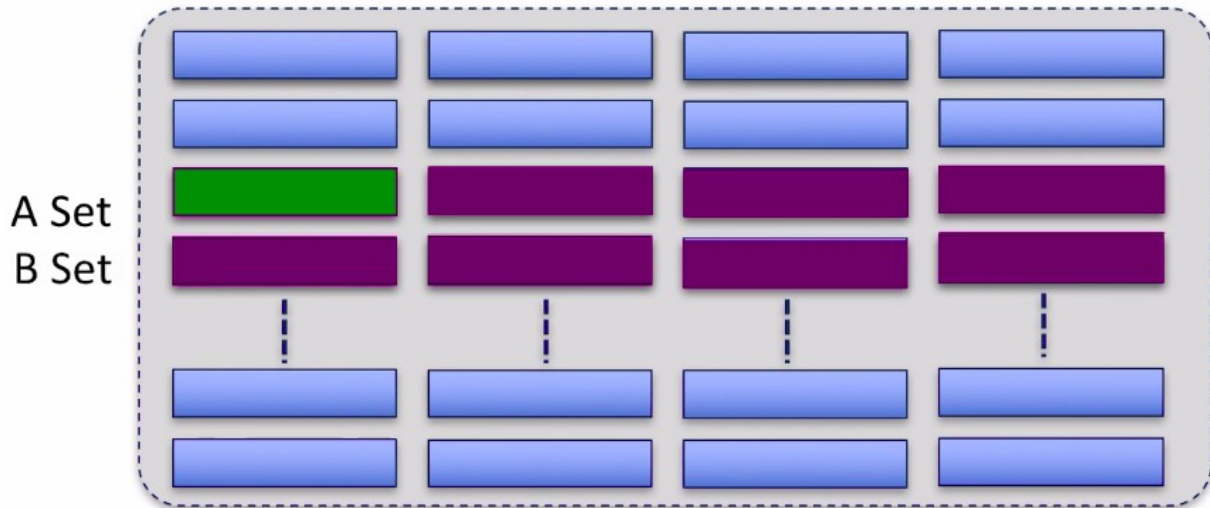


statistically
time A > time B

```
while(1){
```

```
  load A1p2; load A2p2
  load A3p2; load A4p2
  load B1p2; load B2p2
  load B3p2; load B4p2
}
```

Process P2



Cache Covert Channel

Process P1

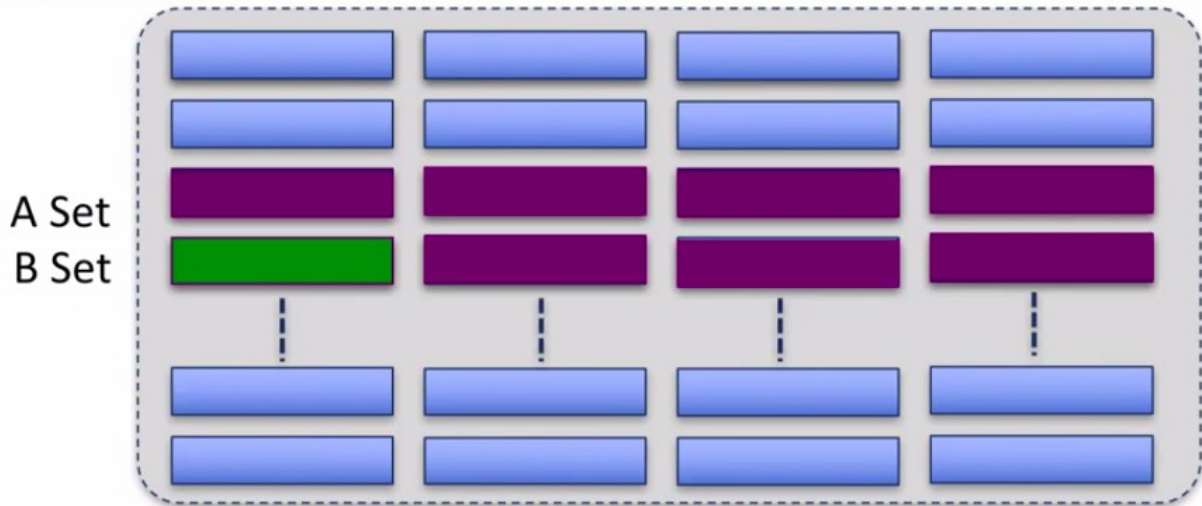
```
If (bit == 1)  
  load Ap1  
Else  
  load Bp1
```



statistically
time A < time B

```
while(1){  
  load A1p2; load A2p2  
  load A3p2; load A4p2  
  load B1p2; load B2p2  
  load B3p2; load B4p2  
}
```

Process P2



Cache Covert Channel



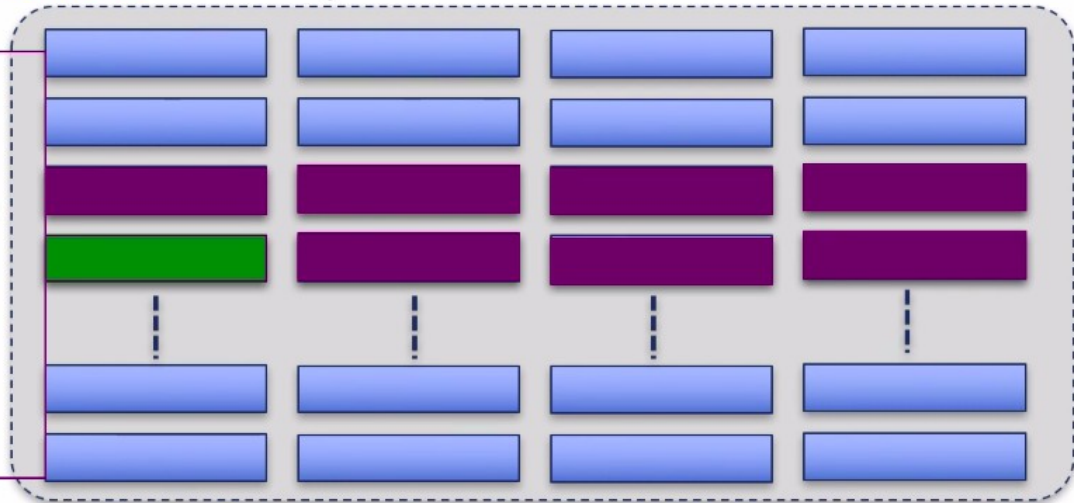
statistically
time A > time B

```
while(1){
```

```
  load A1p2; load A2p2  
  load A3p2; load A4p2  
  load B1p2; load B2p2  
  load B3p2; load B4p2
```

```
}
```

Process P2



```
bit = message
```

```
while(bit[i] != '\0')
```

```
  for(some number of iterations)
```

```
    If (bit[i] == 1)
```

```
      load Ap1
```

```
    else
```

```
      load Bp1
```

Covert Channels

- Identifying: Not easy because simple things like the existence of a file, time, etc. could be a source for a covert channel.
- Quantification: communication rate (bps)
- Elimination: Careful design, separation, characteristics of operation (eg. rate of opening / closing a file)

Cache Timing Attacks

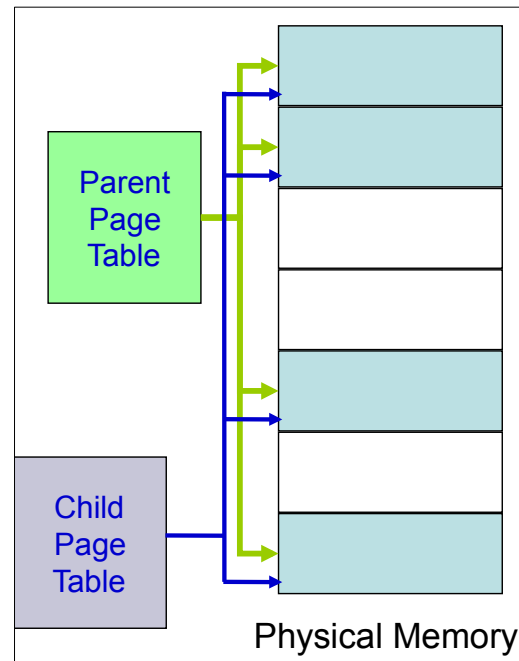
Flush + Reload Attack

Copy on Write

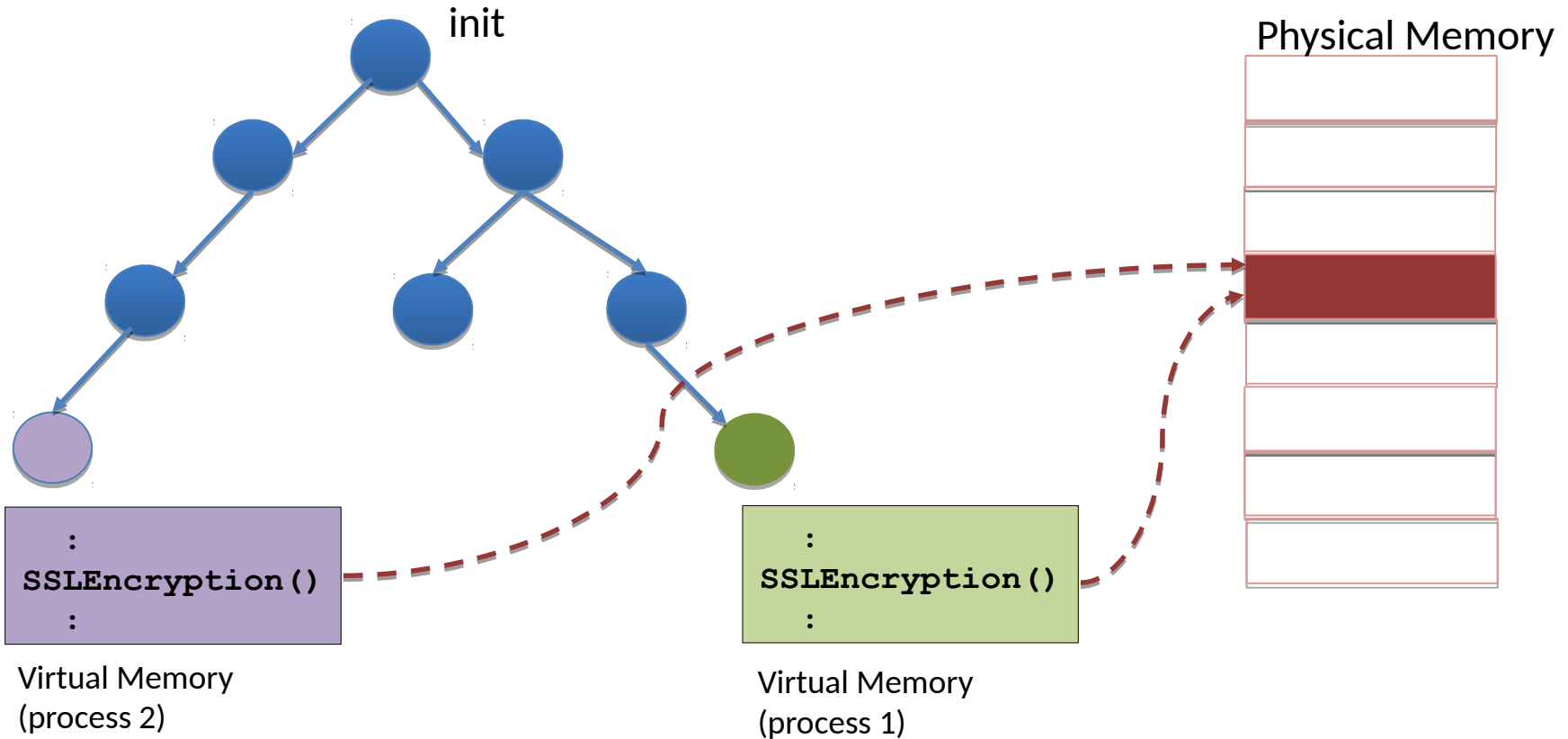
```
if (fork() > 0){  
    // in parent process  
} else{  
    // in child process  
}
```

Child created is an exact replica of the parent process.

- Page tables of the parent duplicated in the child
- New pages created only when parent (or child) modifies data
 - Postpone copying of pages as much as possible, thus optimizing performance
 - Thus, common code sections (like libraries) would be shared across processes.



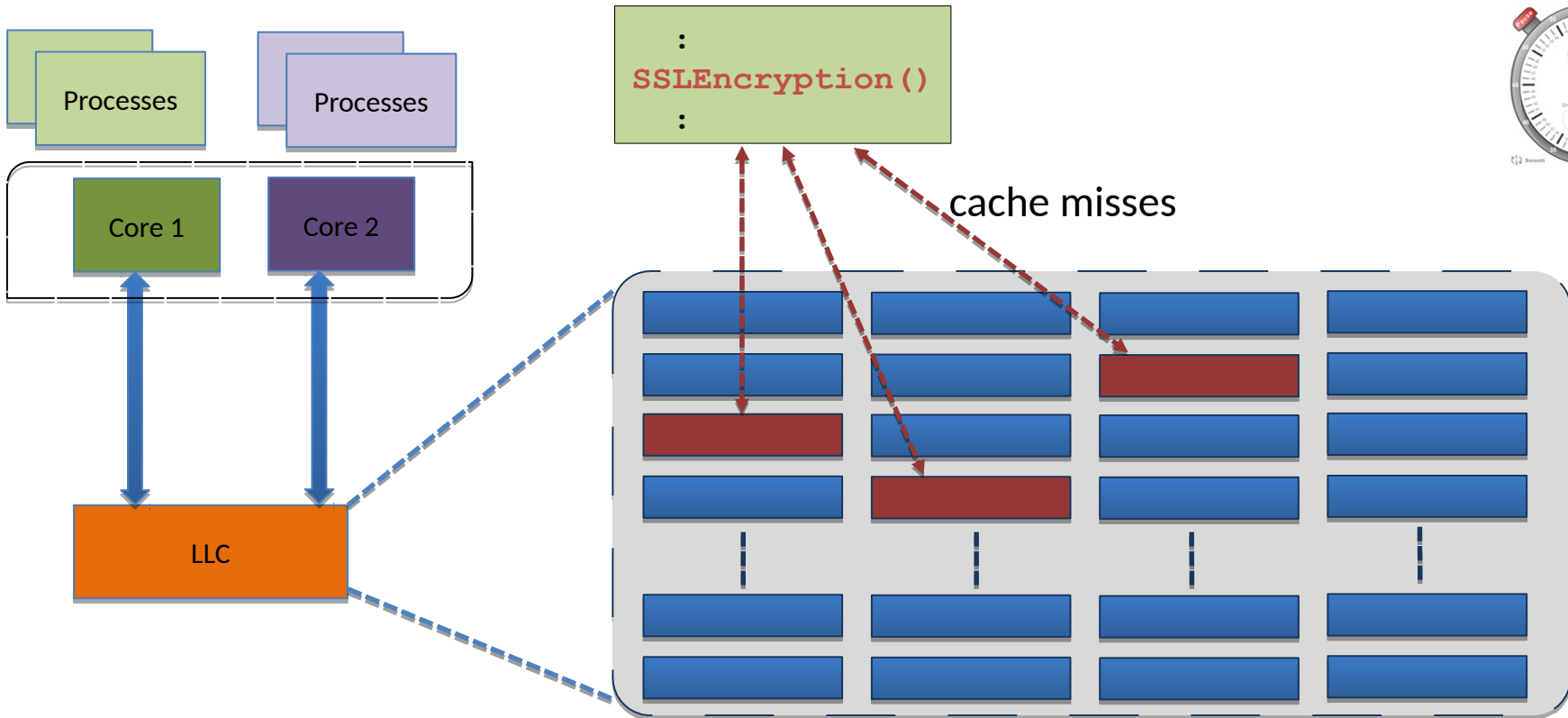
Process Tree



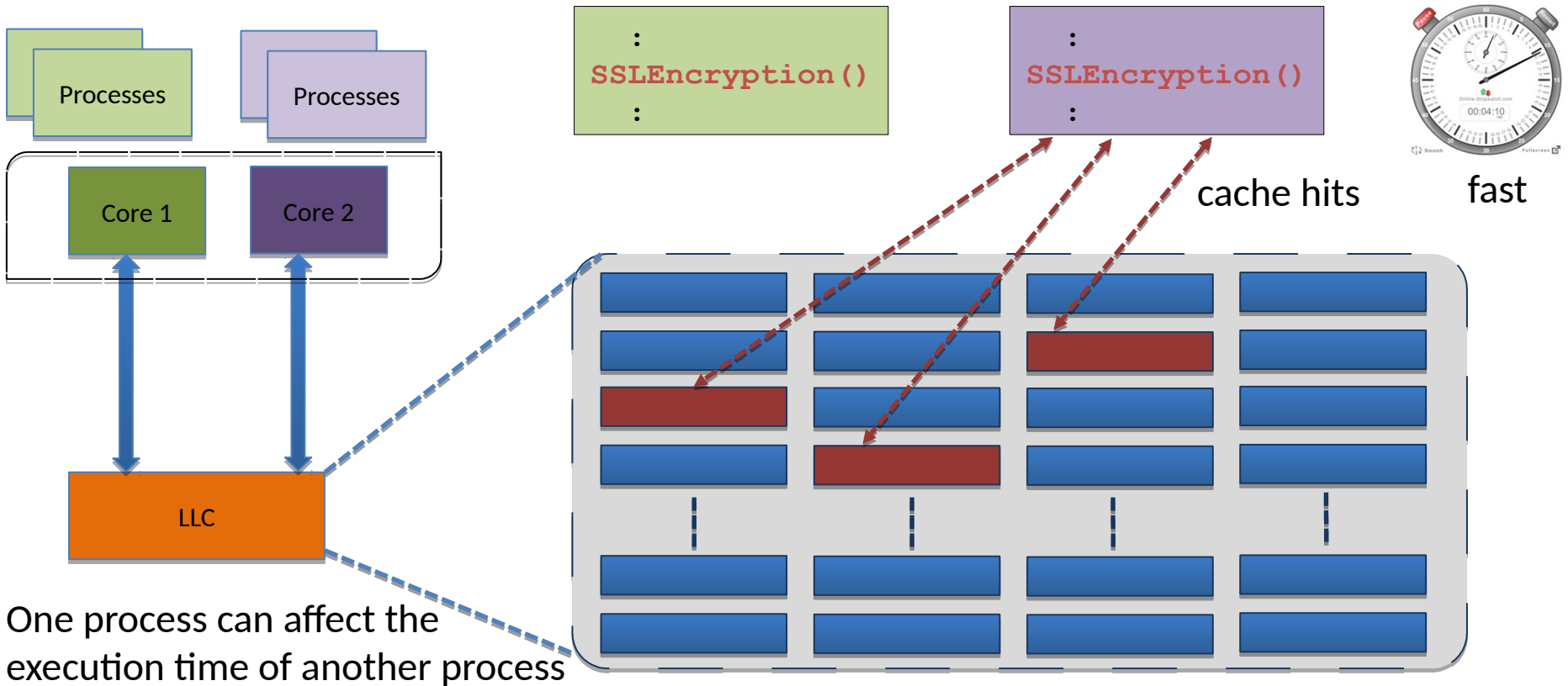
Interaction with the LLC



slow



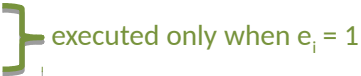
Interaction with the LLC



Flush + Reload Attack on LLC

Part of an encryption algorithm

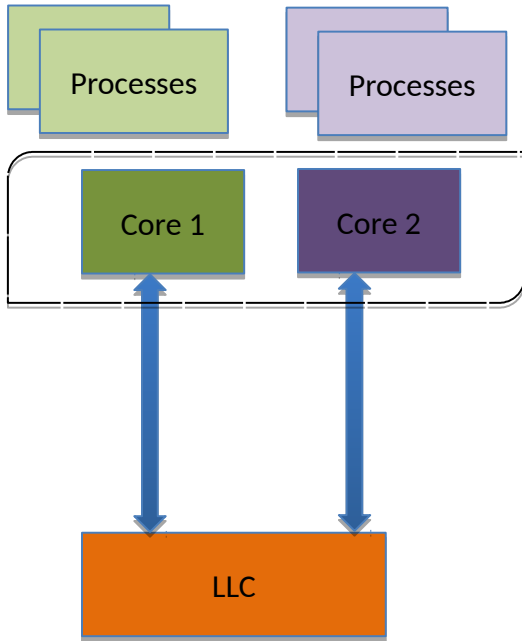
```
1 function exponent( $b, e, m$ )
2 begin
3    $x \leftarrow 1$ 
4   for  $i \leftarrow |e| - 1$  downto 0 do
5      $x \leftarrow x^2$ 
6      $x \leftarrow x \bmod m$ 
7     if ( $e_i = 1$ ) then
8        $x \leftarrow xb$ 
9        $x \leftarrow x \bmod m$ 
10    endif
11  done
12  return  $x$ 
13 end
```

 executed only when $e_i = 1$

cflush Instruction

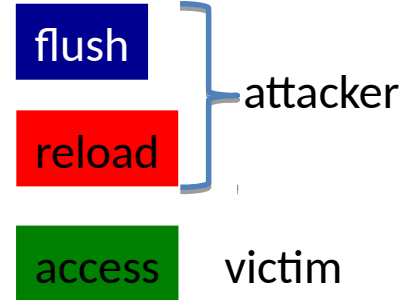
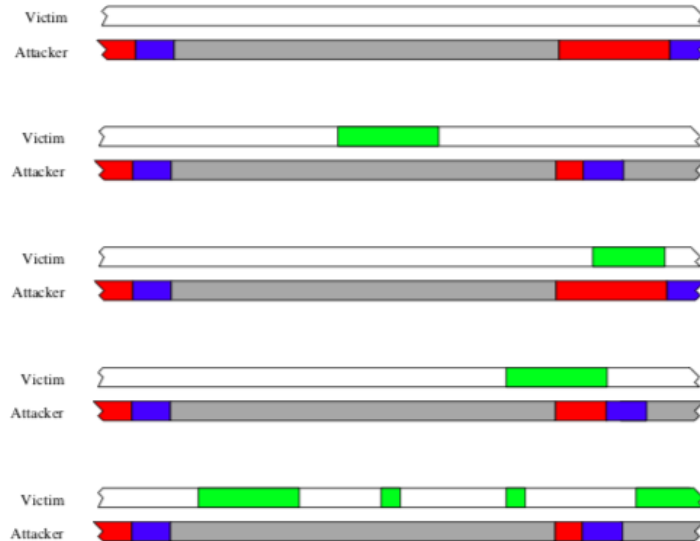
Takes an address as input.
Flushes that address from all caches
cflush (line 8)

Flush + Reload Attack



```
:  
SSLEncryption()  
:
```

```
:  
Clflush(line 8)  
:
```



Flush+Reload Attack

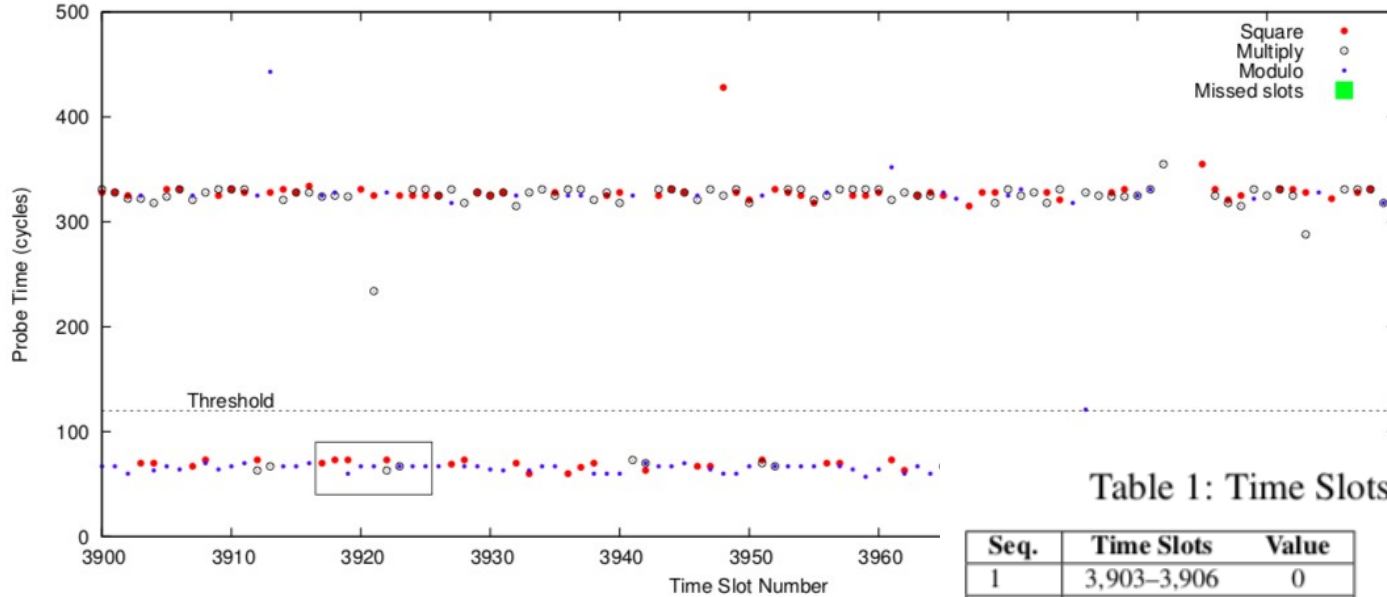


Table 1: Time Slots for Bit Sequence

Seq.	Time Slots	Value
1	3,903–3,906	0
2	3,907–3,916	1
3	3,917–3,926	1
4	3,927–3,931	0
5	3,932–3,935	0
6	3,936–3,945	1
7	3,946–3,955	1

Seq.	Time Slots	Value
8	3,956–3,960	0
9	3,961–3,969	1
10	3,970–3,974	0
11	3,975–3,979	0
12	3,980–3,988	1
13	3,989–3,998	1

Countermeasures

- Do not use copy-on-write
 - Implemented by cloud providers
- Permission checks for cflush
 - Do we need cflush?
- Non-inclusive cache memories
 - AMD
 - Intel i9 versions
- Fuzzing Clocks
- Software Diversification
 - Permute location of objects in memory (statically and dynamically)

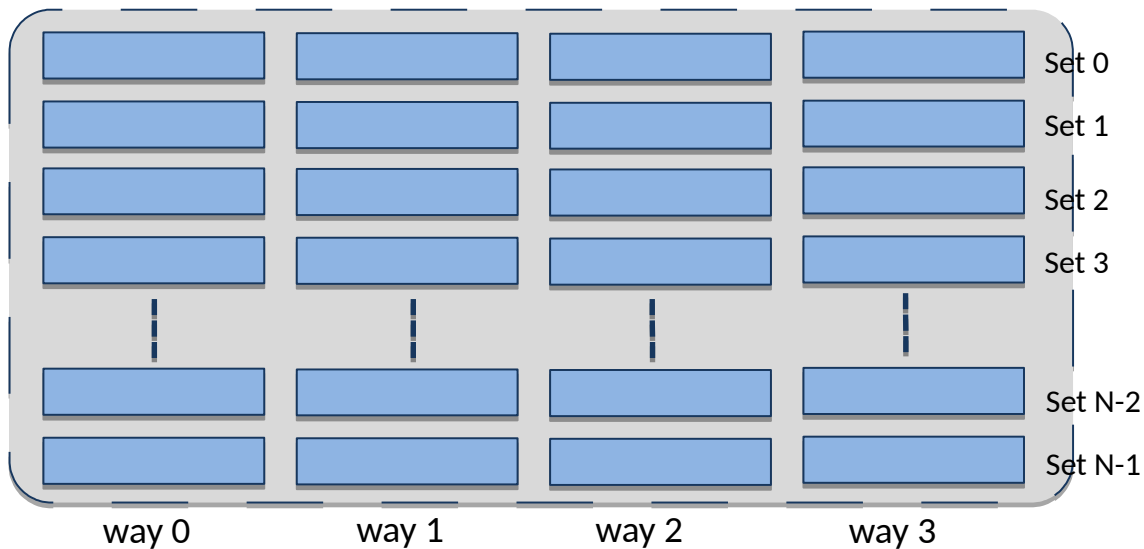
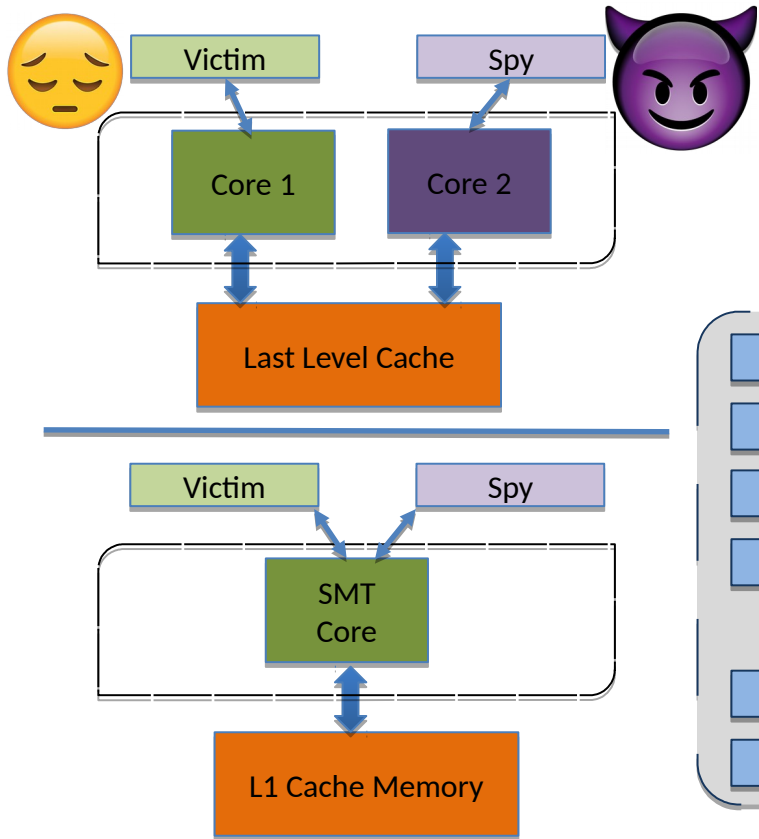
Cache Collision Attacks

Prime + Probe Attack

Cache Collision Attacks

- External Collision Attacks
 - Prime + Probe
- Internal Collision Attacks
 - Time-driven attacks

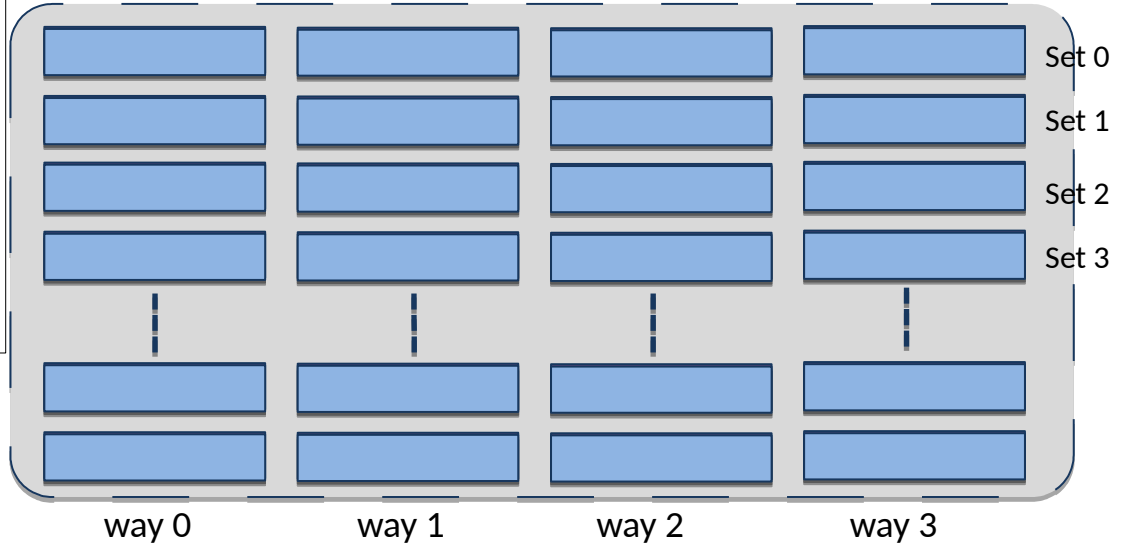
Prime + Probe Attack



Prime Phase



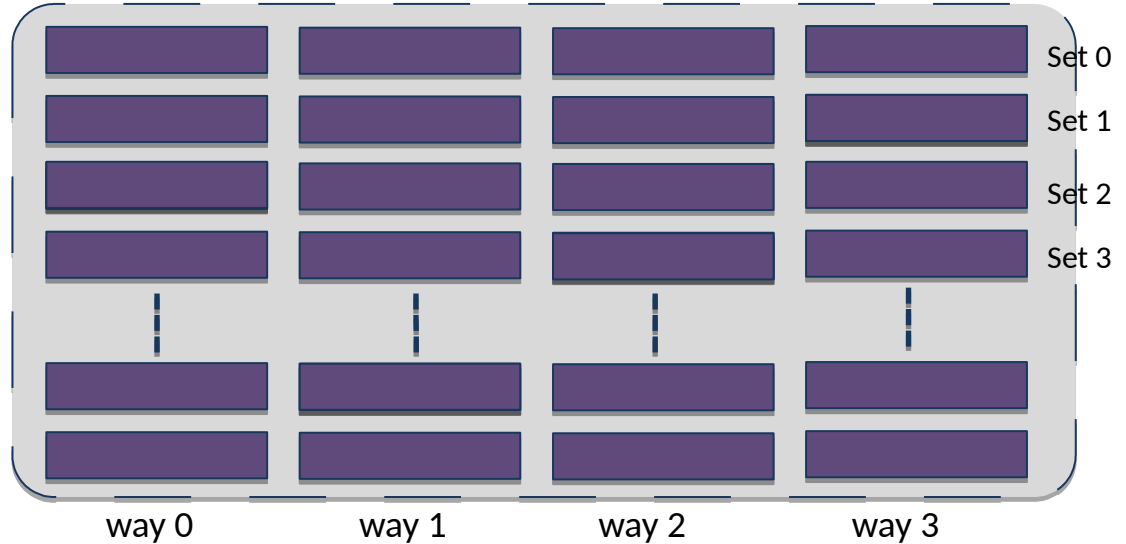
```
While(1){  
  for(each cache set){  
    start = time();  
    access all cache ways  
    end = time();  
    access_time = end - start  
  }  
  wait for some time  
}
```



Victim Execution



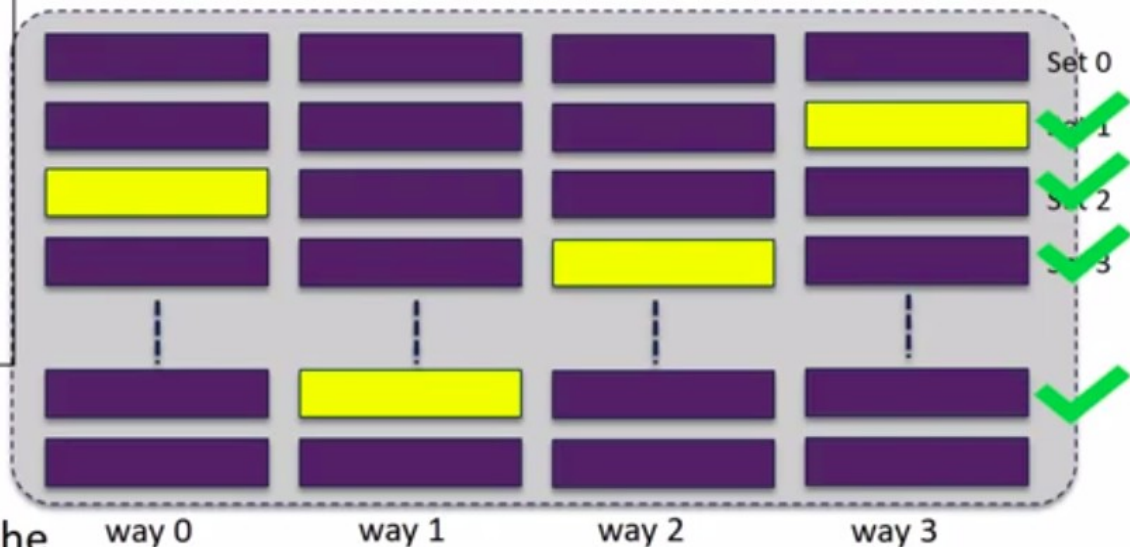
The execution causes some of the spy data to get evicted



Probe Phase



```
While(1){  
  for(each cache set){  
    start = time();  
    access all cache ways  
    end = time();  
    access_time = end - start  
  }  
  wait for some time  
}
```

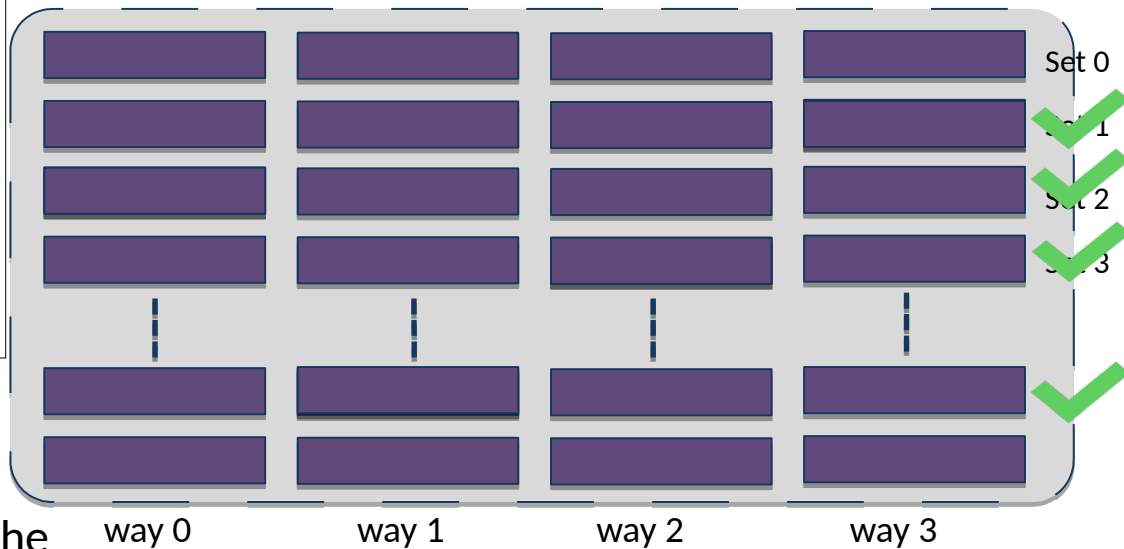


Time taken by sets that have victim data is more due to the cache misses

Probe Phase

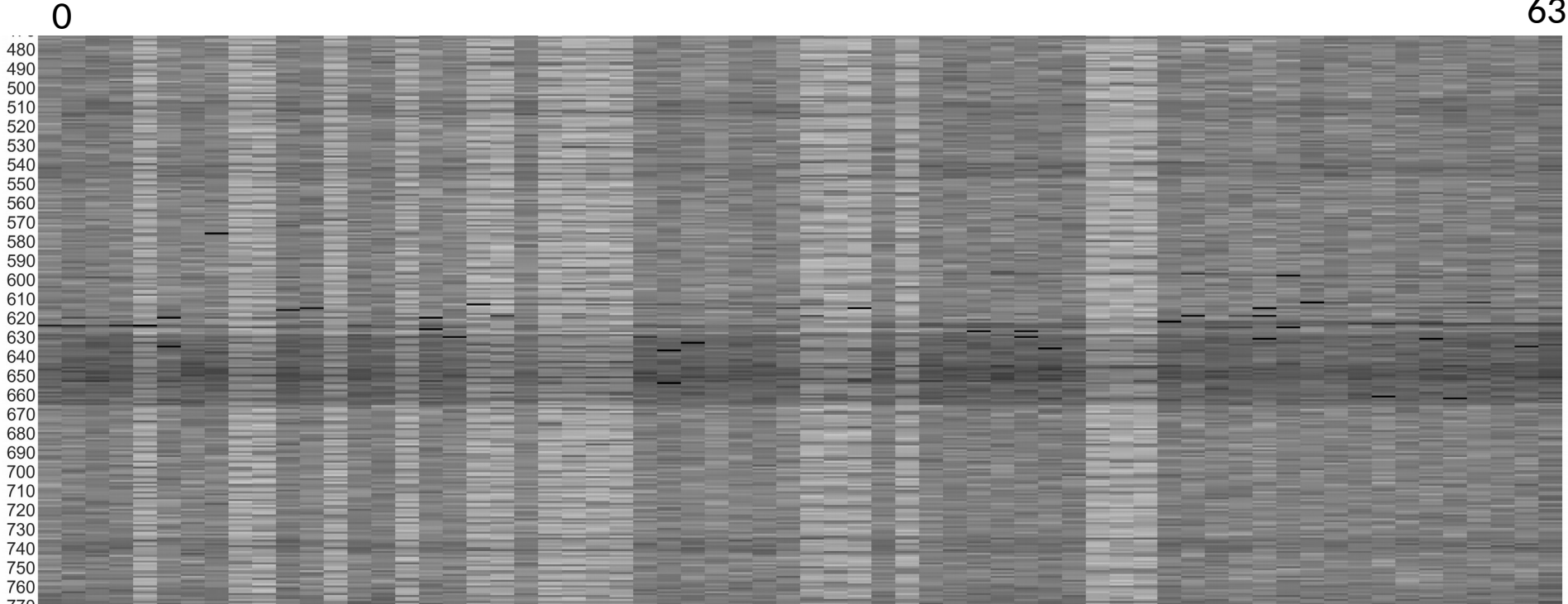


```
While(1){  
  for(each cache set){  
    start = time();  
    access all cache ways  
    end = time();  
    access_time = end - start  
  }  
  wait for some time  
}
```



Time taken by sets that have victim data is more due to the cache misses

Probe Time Plot



Each row is an iteration of the while loop; darker shades imply higher memory access time



Prime + Probe in Cryptography

```
char Lookup[] = {x, x, x, . . . x};

char RecvDecrypt(socket){
    char key = 0x12;
    char pt, ct;

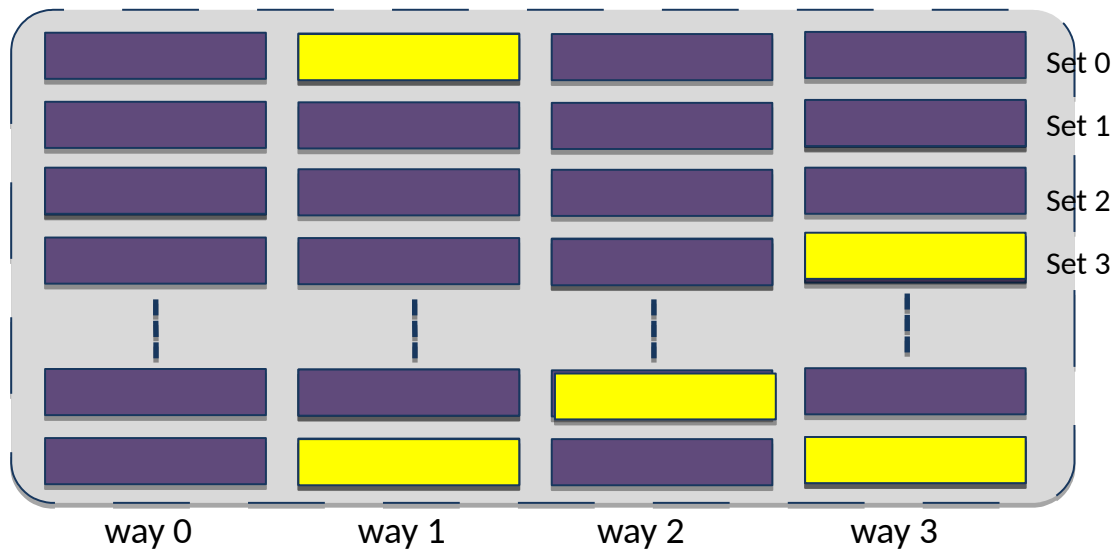
    read(socket, &ct, 1);
    pt = Lookup[key ^ ct];
    return pt;
}
```

Key dependent memory accesses

The attacker know the address of Lookup and the ciphertext (ct)
The memory accessed in Lookup depends on the value of key
Given the set number, one can identify bits of $\text{key} \wedge \text{ct}$.

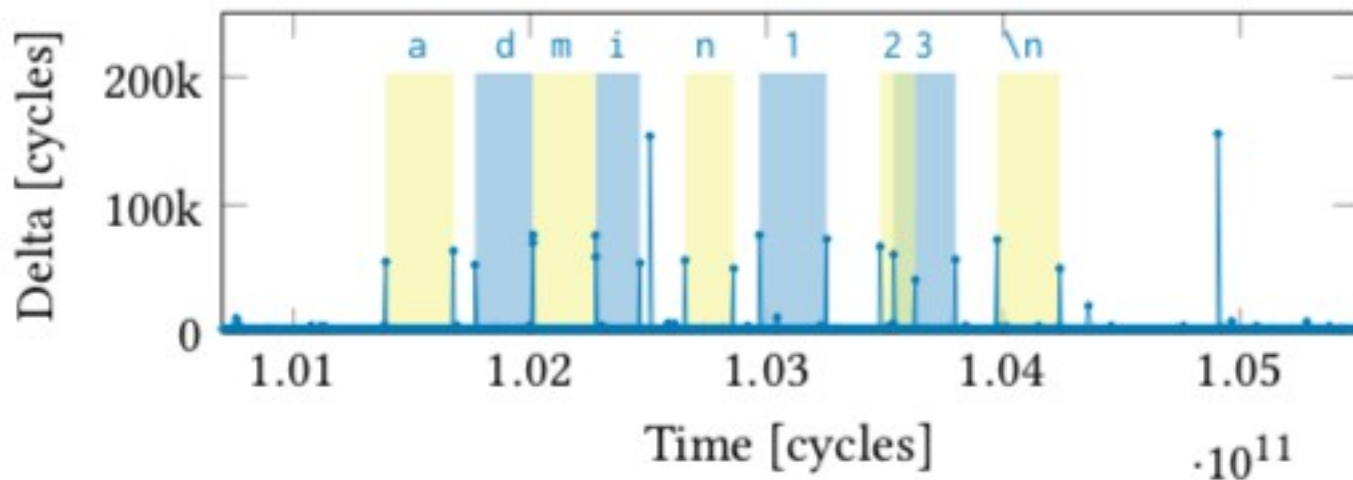
Keystroke Sniffing

- Keystroke → interrupt → kernel mode switch → ISR execution → add to keyboard buffer → ... → return from interrupt



Keystroke Sniffing

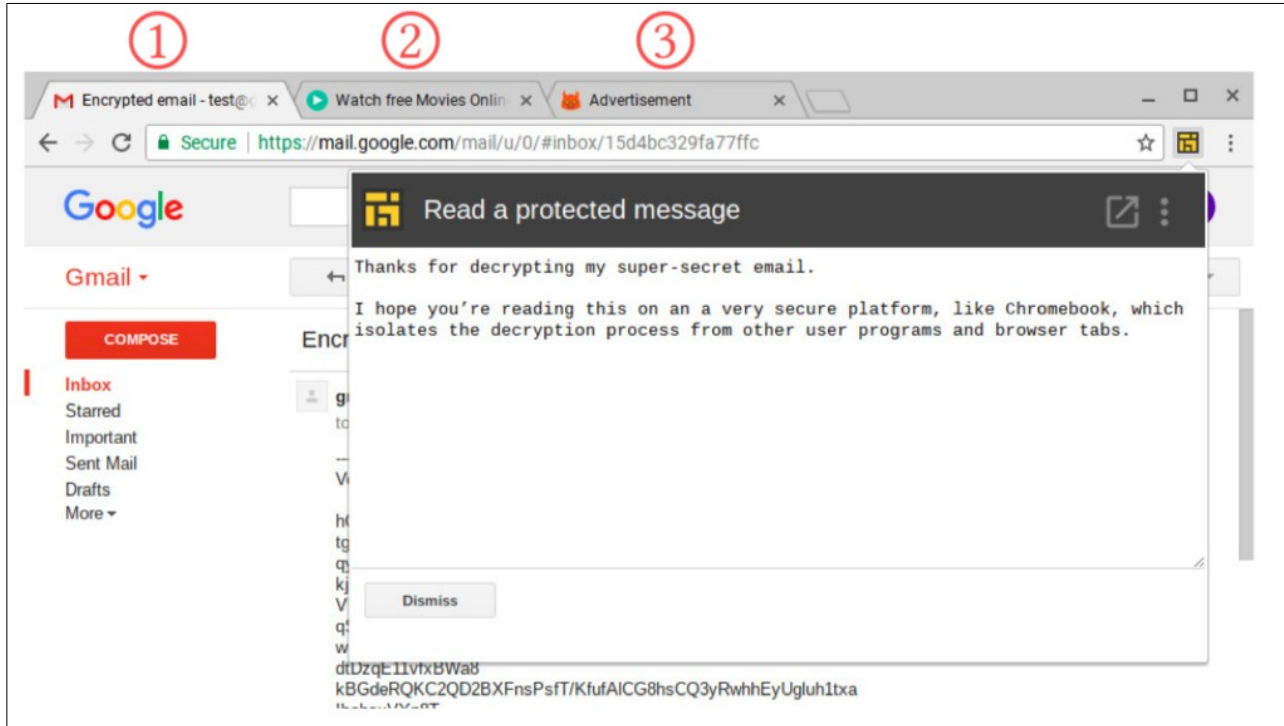
- Regular disturbance seen in Probe Time Plot
- Period between disturbance used to predict passwords



Web Browser Attacks

- Prime+Probe in
 - Javascript
 - pNACL
 - Web assembly

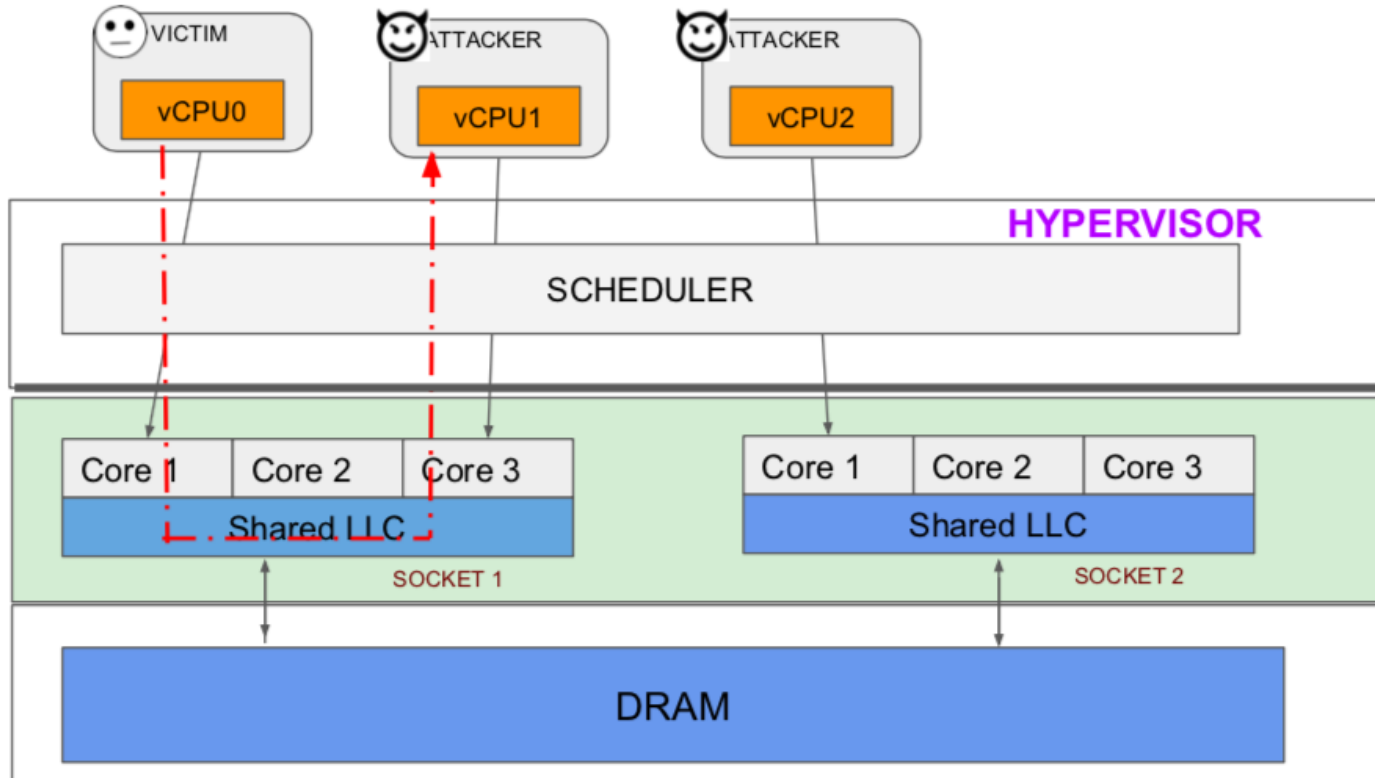
Extract Gmail secret key



Website Fingerprinting

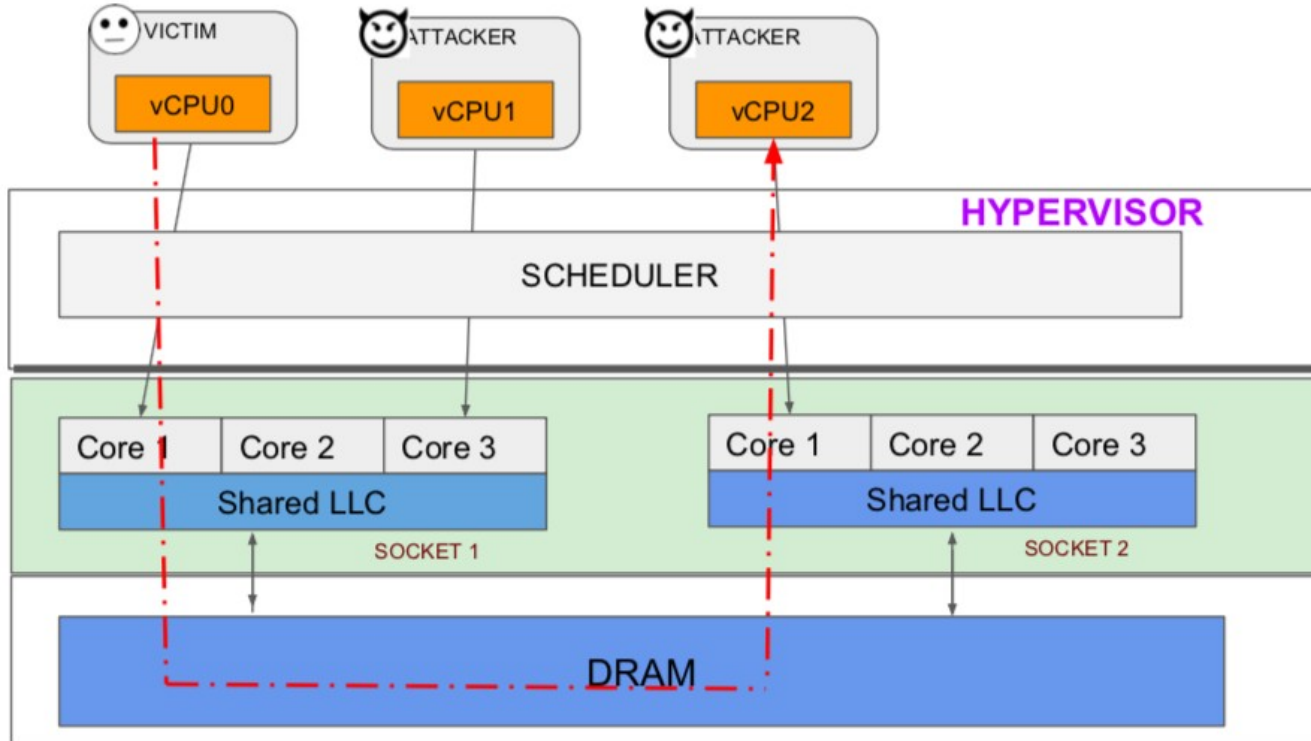
- Privacy: Find out what websites are being browsed.

Cross VM Attacks (Cache)



*Ristenpart et al., *Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds*, CCS- 2009

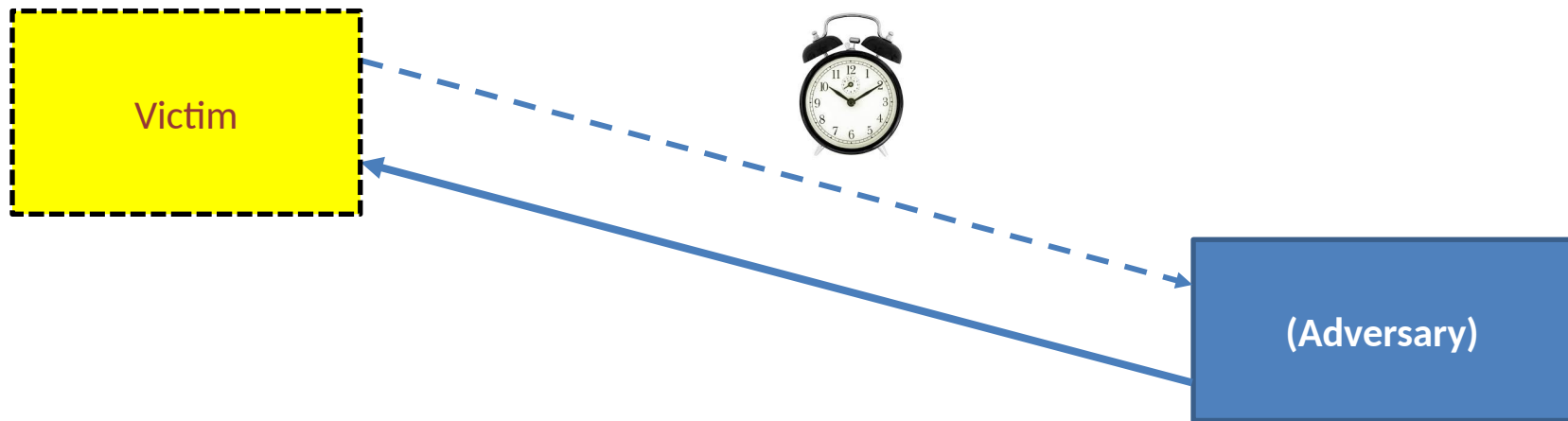
Cross VM Attacks (DRAM)



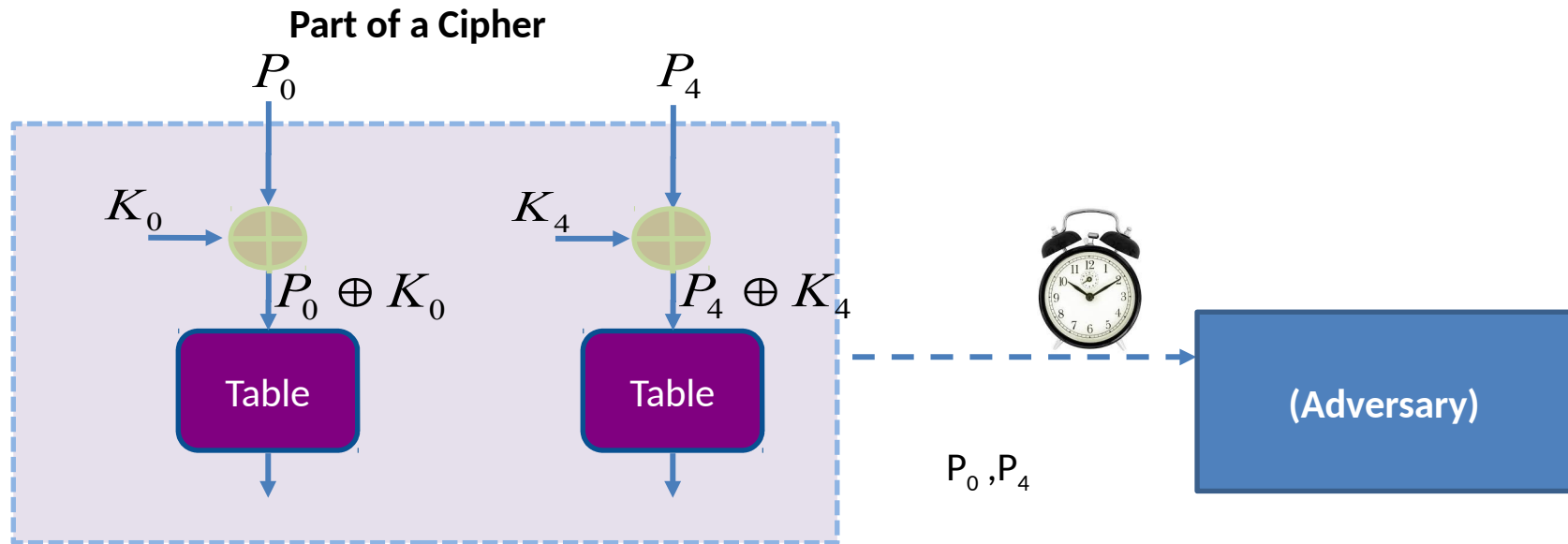
Cache Collision Attacks

Time Driven Attacks

Internal Collision Attacks



Internal Collisions on a Cipher



If cache hit (less time) :

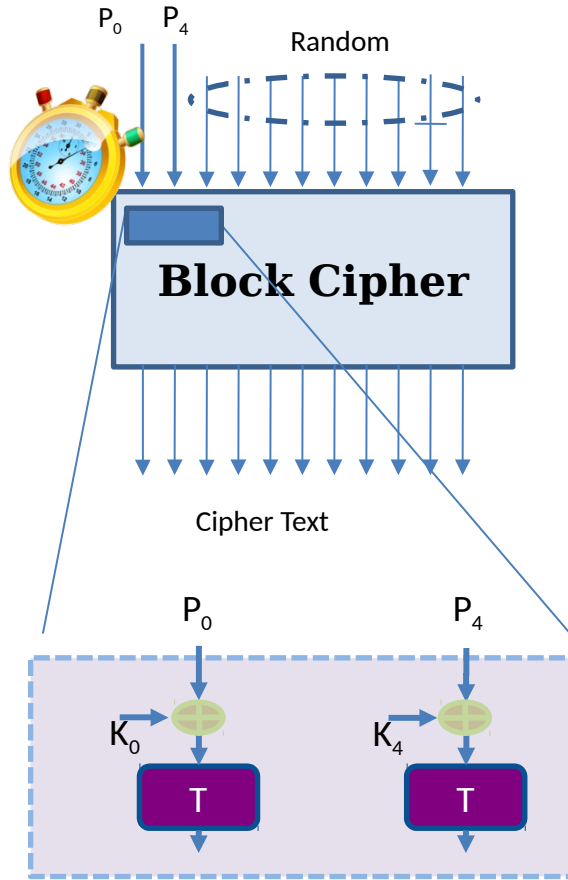
$$\langle P_0 \oplus K_0 \rangle = \langle P_4 \oplus K_4 \rangle$$
$$\Rightarrow \langle K_0 \oplus K_4 \rangle = \langle P_0 \oplus P_4 \rangle$$

If cache miss (more time):

$$\langle P_0 \oplus K_0 \rangle \neq \langle P_4 \oplus K_4 \rangle$$
$$\Rightarrow \langle K_0 \oplus K_4 \rangle \neq \langle P_0 \oplus P_4 \rangle$$

Suppose
 ($K_0 = 00$ and $k_4 = 50$)

- $P_0 = 0$, all other inputs are random
- Make N time measurements
- Segregate into Y buckets based on value of P_4
- Find average time of each bucket
- Find deviation of each average from overall average (DOM)



$$\langle K_0 \oplus K_4 \rangle = \langle P_0 \oplus P_4 \rangle$$

P4	Average Time	DOM
00	2945.3	1.8
10	2944.4	0.9
20	2943.7	0.2
30	2943.7	0.2
40	2944.8	1.3
50	2937.4	6.3
60	2943.3	-0.2
70	2945.8	2.3
:	:	:
F0	2941.8	-1.7

Average : 2943.57
 Maximum : -6.3

That's for the Day !!