

Micro-architectural Attacks

Chester Rebeiro
IIT Madras



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)

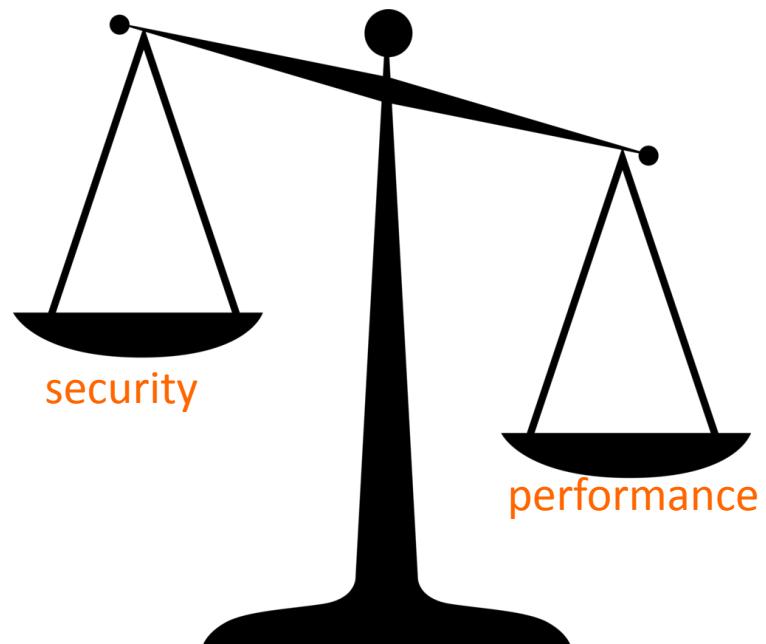
- Cache timing attack
- Branch prediction attack
- Speculation Attacks
- Row hammer
- Fault Injection Attacks
- cold boot attacks
- DRAM Row buffer (DRAM)
..... and many more

Causes

Most micro-architectural attacks caused by performance optimizations

Others due to inherent device properties

Third, due to stronger attackers

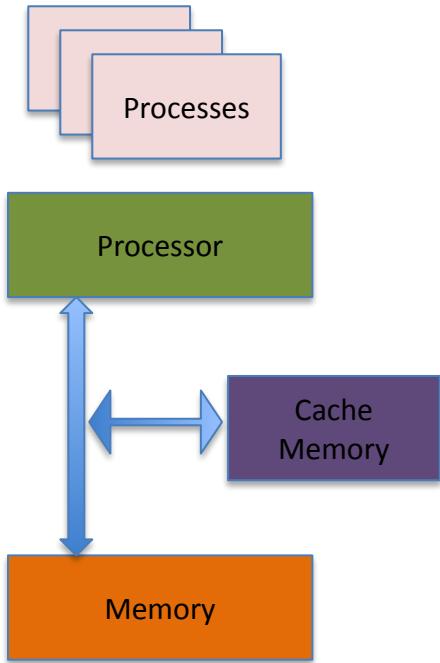


Cache Covert Channel

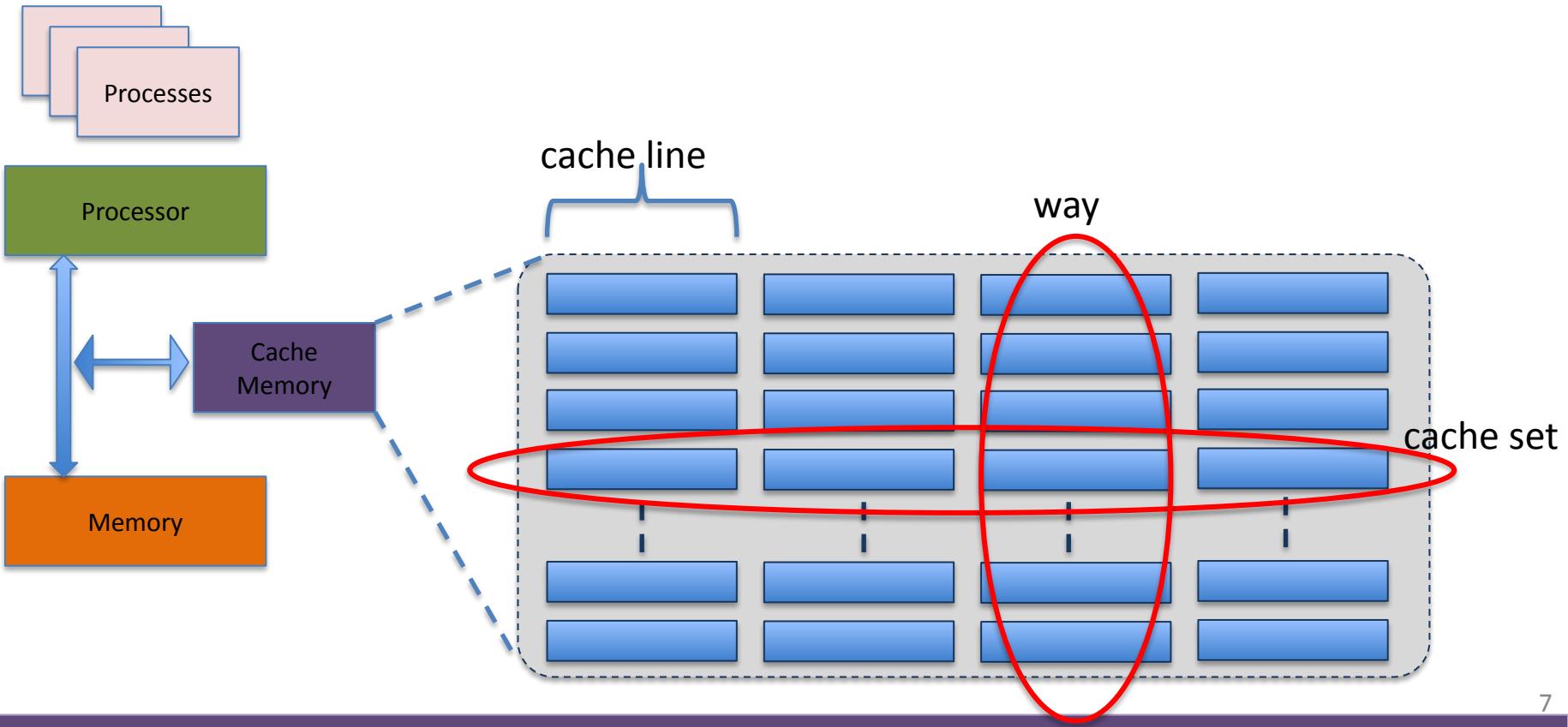
Chester Rebeiro

Indian Institute of Technology Madras

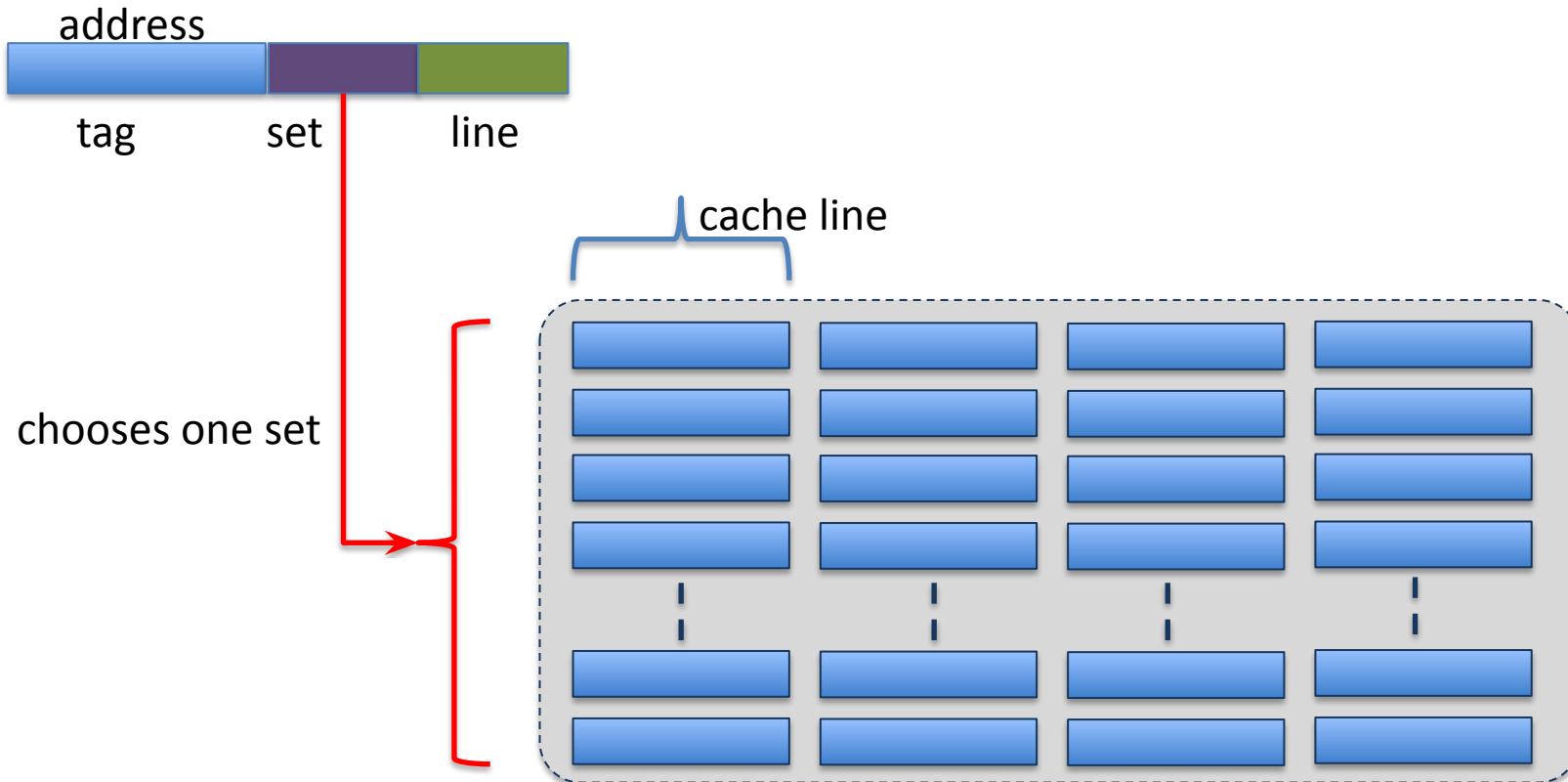
Cache Covert Channel



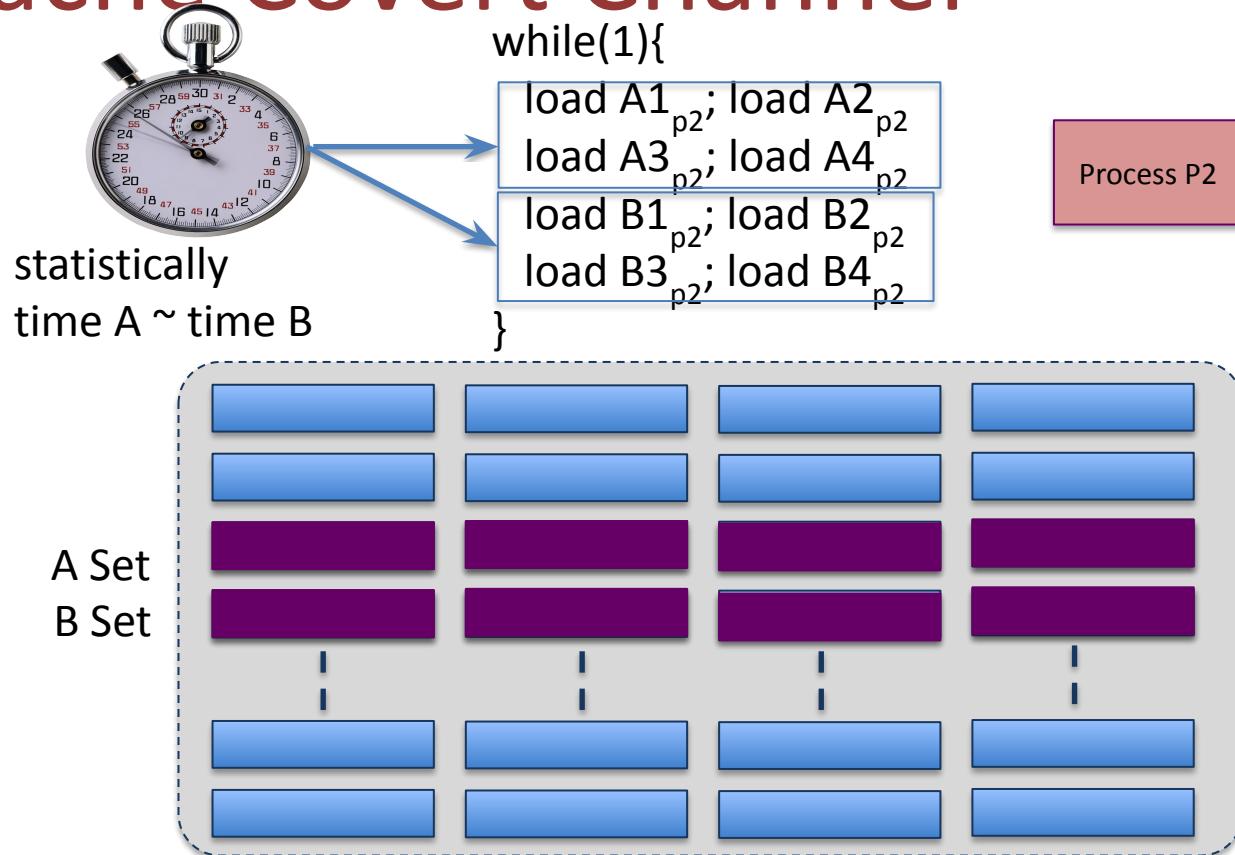
Cache Covert Channel



Cache Covert Channel



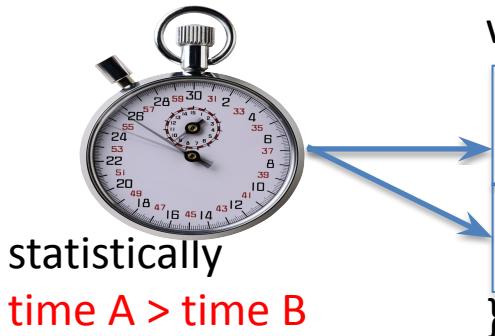
Cache Covert Channel



Cache Covert Channel

Process P1

```
If (bit == 1)  
    load AP1  
Else  
    load BP1
```



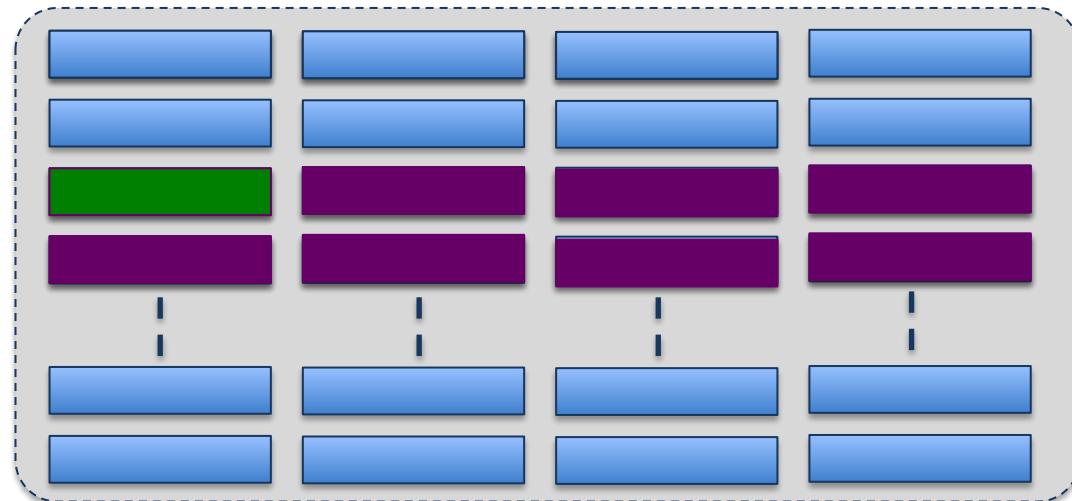
while(1){

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

}

Process P2

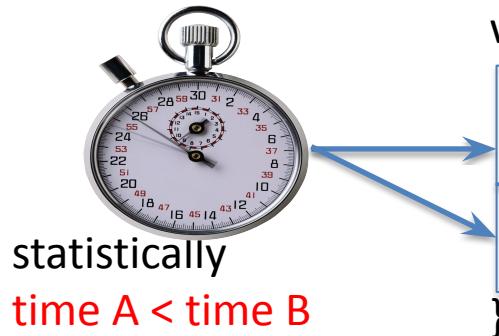
A Set
B Set



Cache Covert Channel

Process P1

```
If (bit == 1)  
    load AP1  
Else  
    load BP1
```

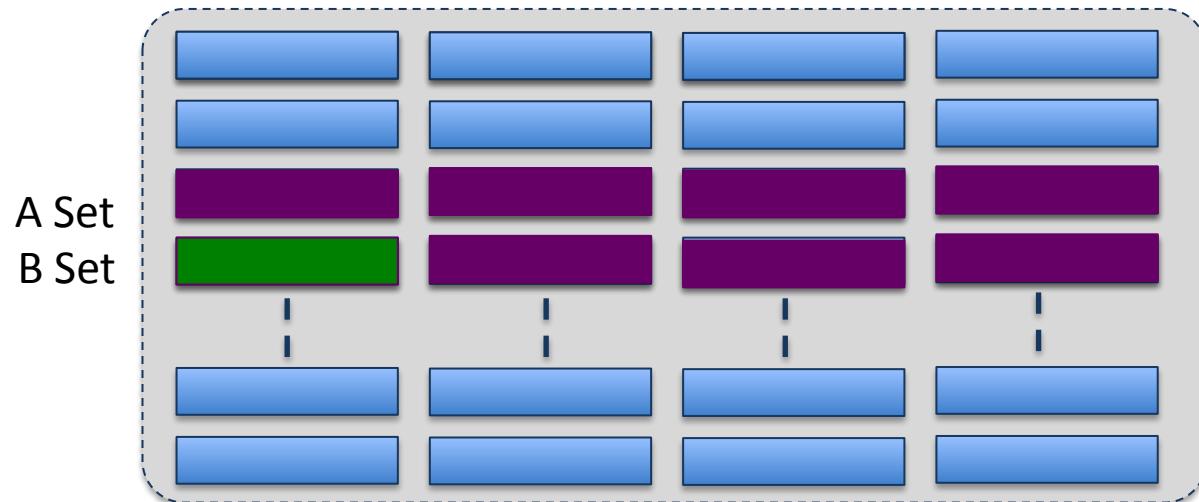


while(1){

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

}

Process P2



Cache Covert Channel

Process P1

Process P2



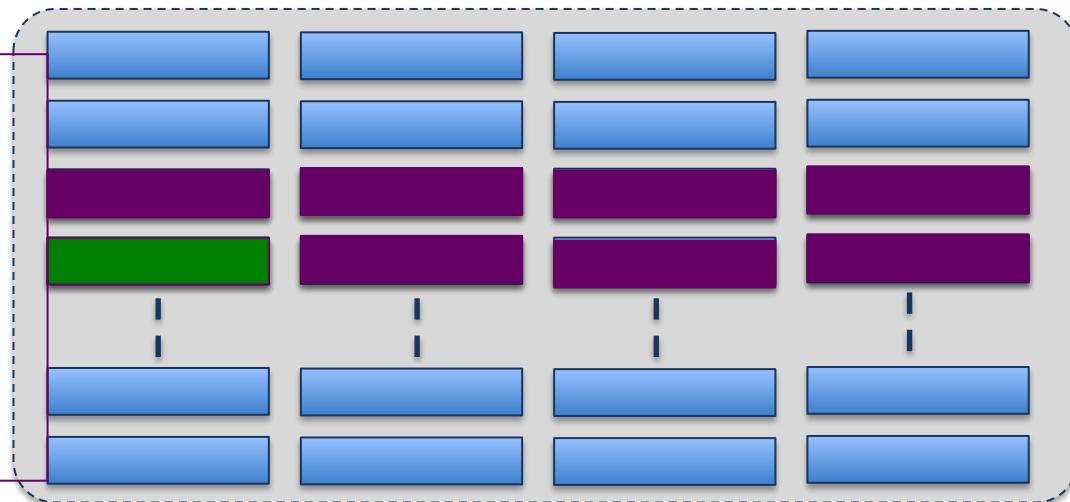
statistically
time A < time B

while(1){

load A_{1 p2}; load A_{2 p2}
load A_{3 p2}; load A_{4 p2}
load B_{1 p2}; load B_{2 p2}
load B_{3 p2}; load B_{4 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)

Flush+Reload Attacks

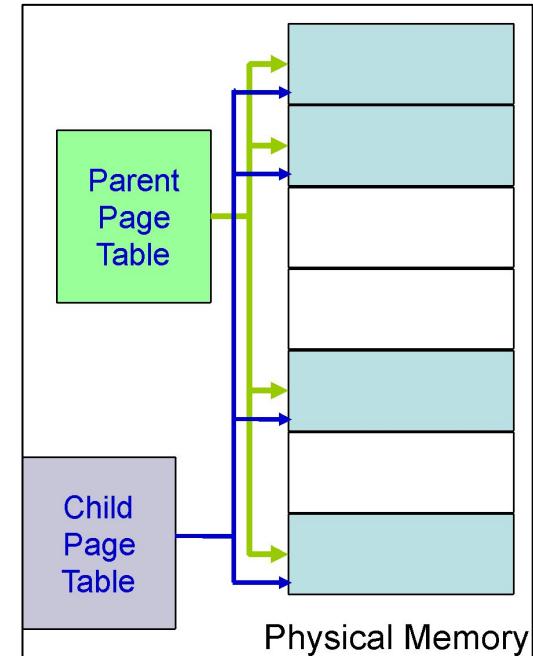
Chester Rebeiro
IIT Madras

Copy on Write

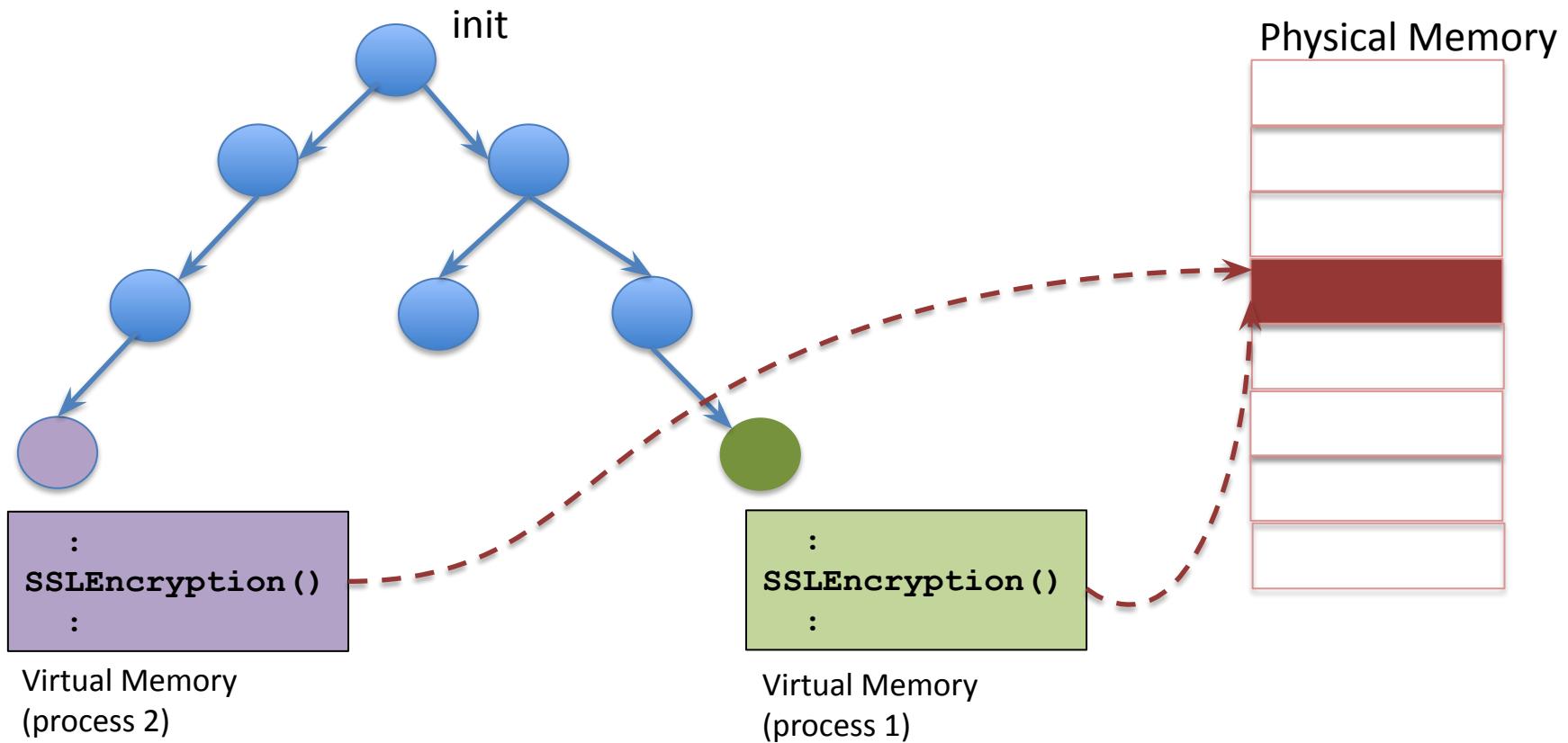
```
if (fork() > 0){  
    // in parent process  
} else{  
    // in child process  
}
```

Child created is an exact replica of the parent process.

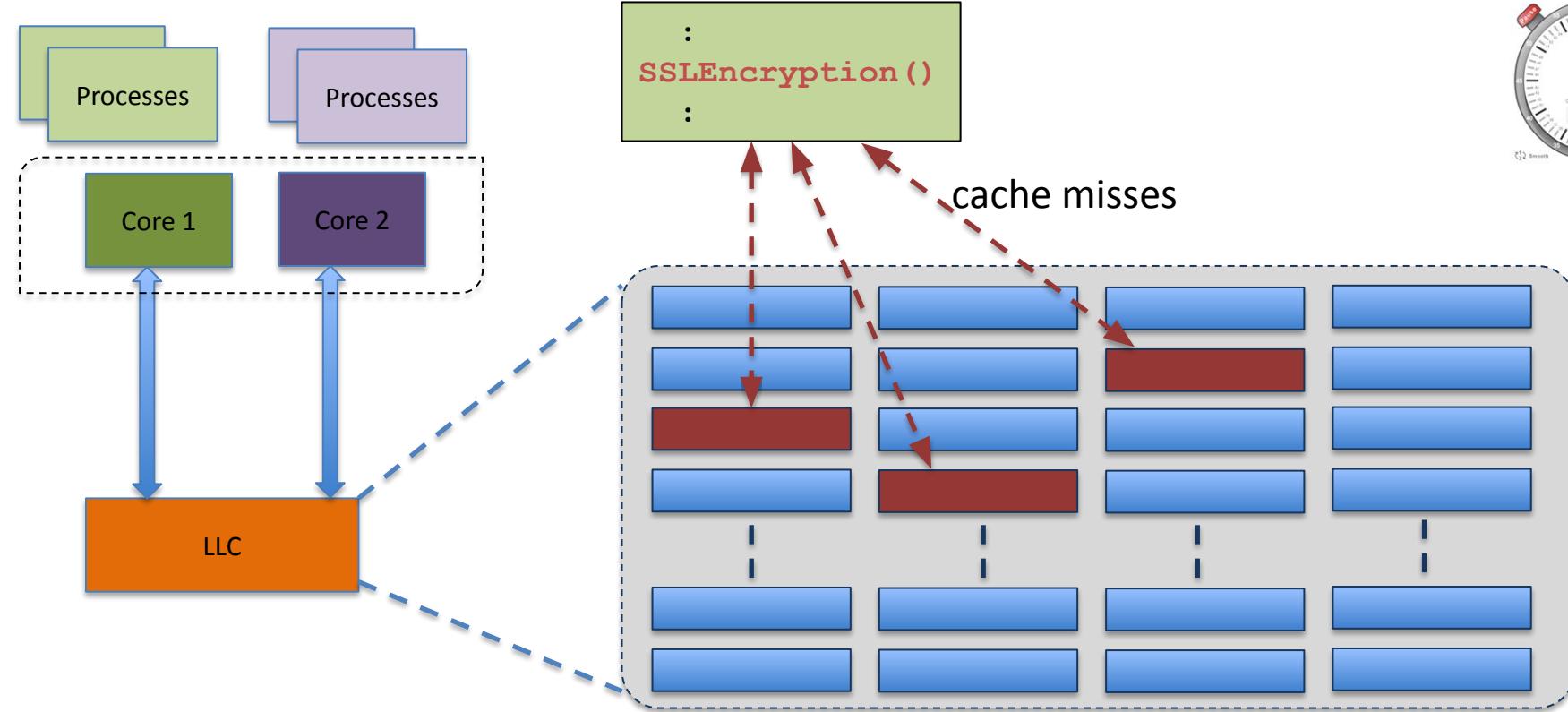
- Page tables of the parent duplicated in the child
- New pages created only when parent (or child) modifies data
 - Postpone copying of pages as much as possible, thus optimizing performance
 - Thus, common code sections (like libraries) would be shared across processes.



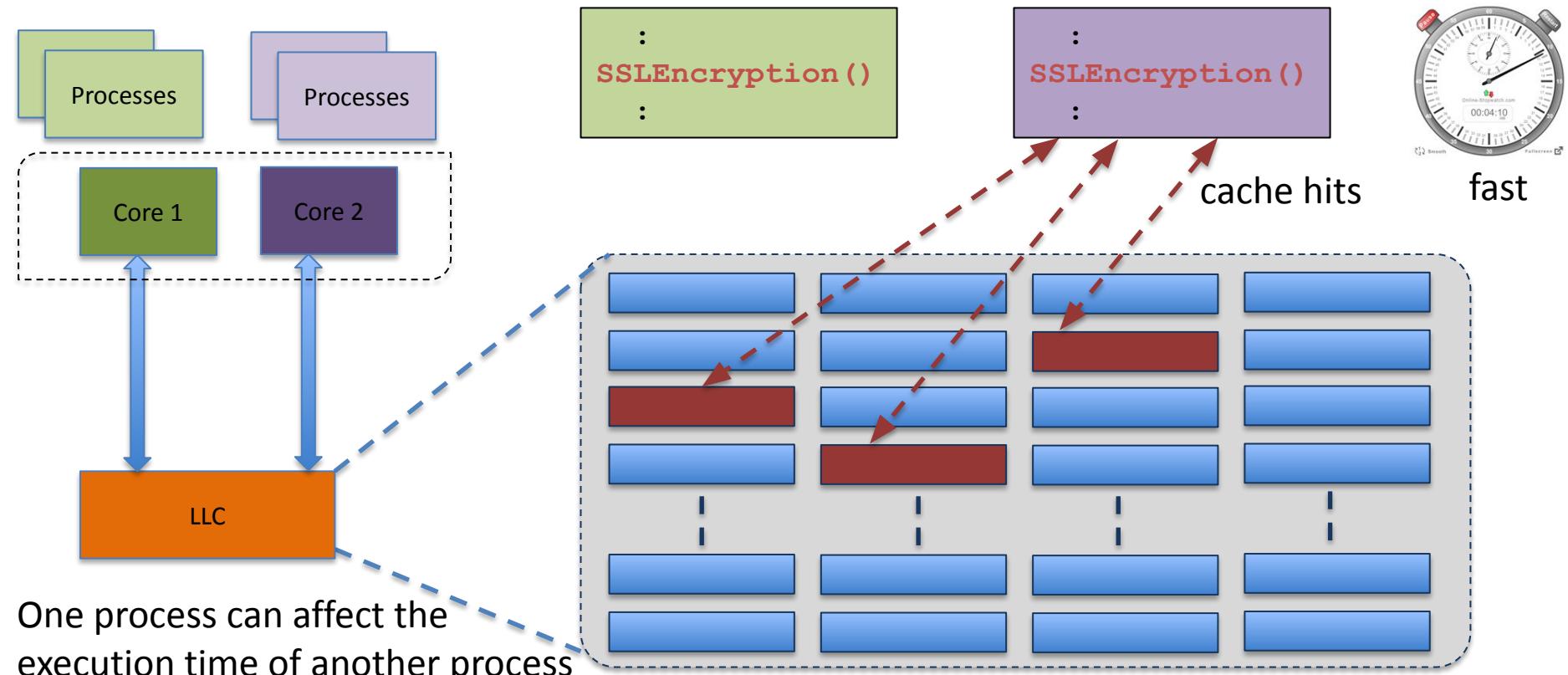
Process Tree



Interaction with the LLC



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
```

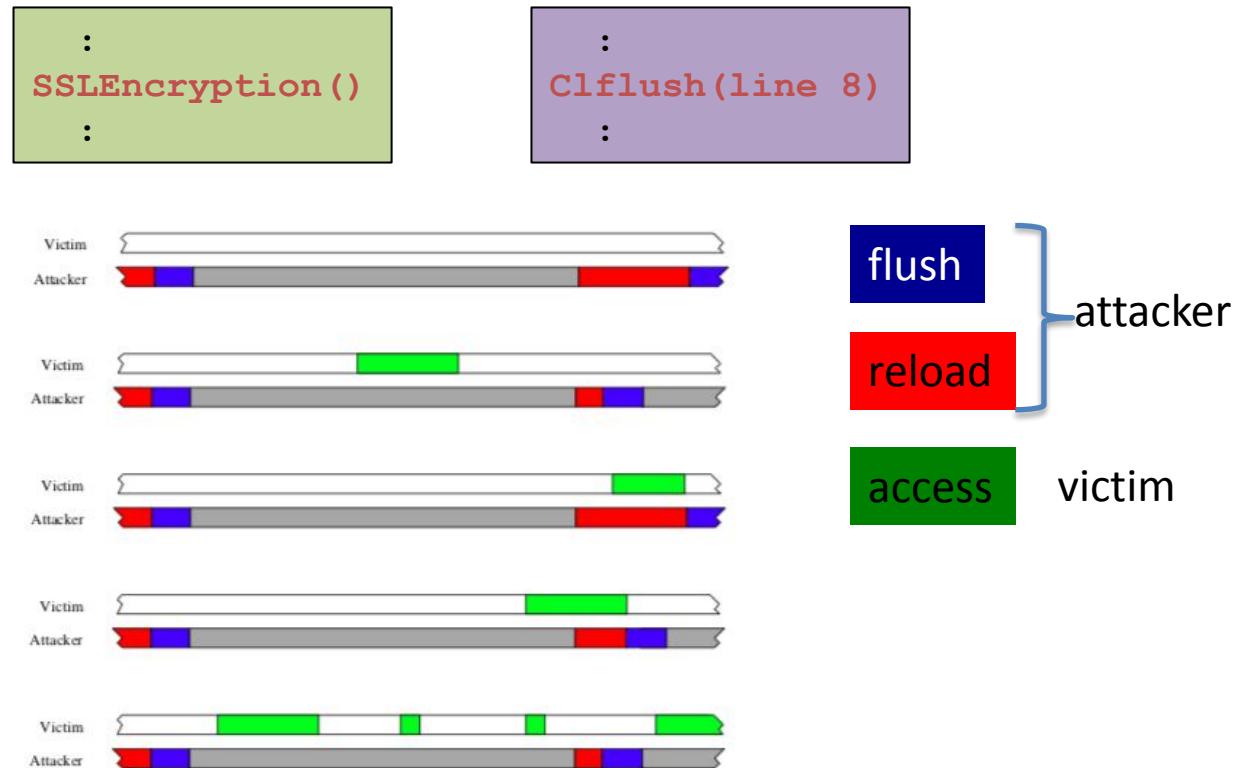
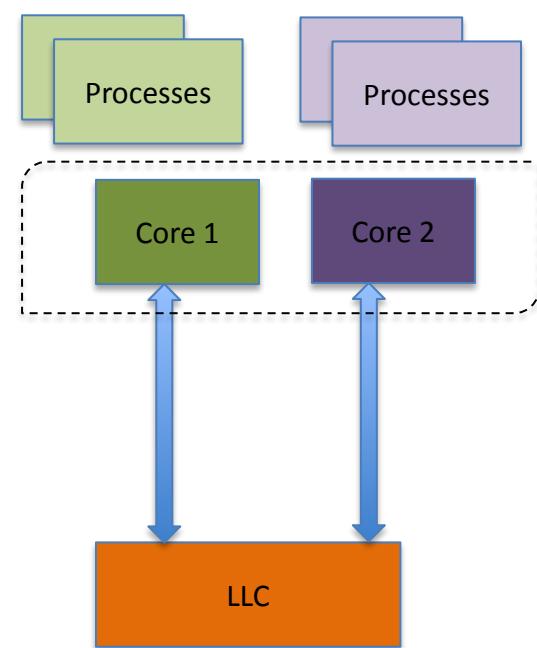


} executed only when $e_i = 1$

clflush Instruction

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

Flush + Reload Attack



Flush+Reload Attack

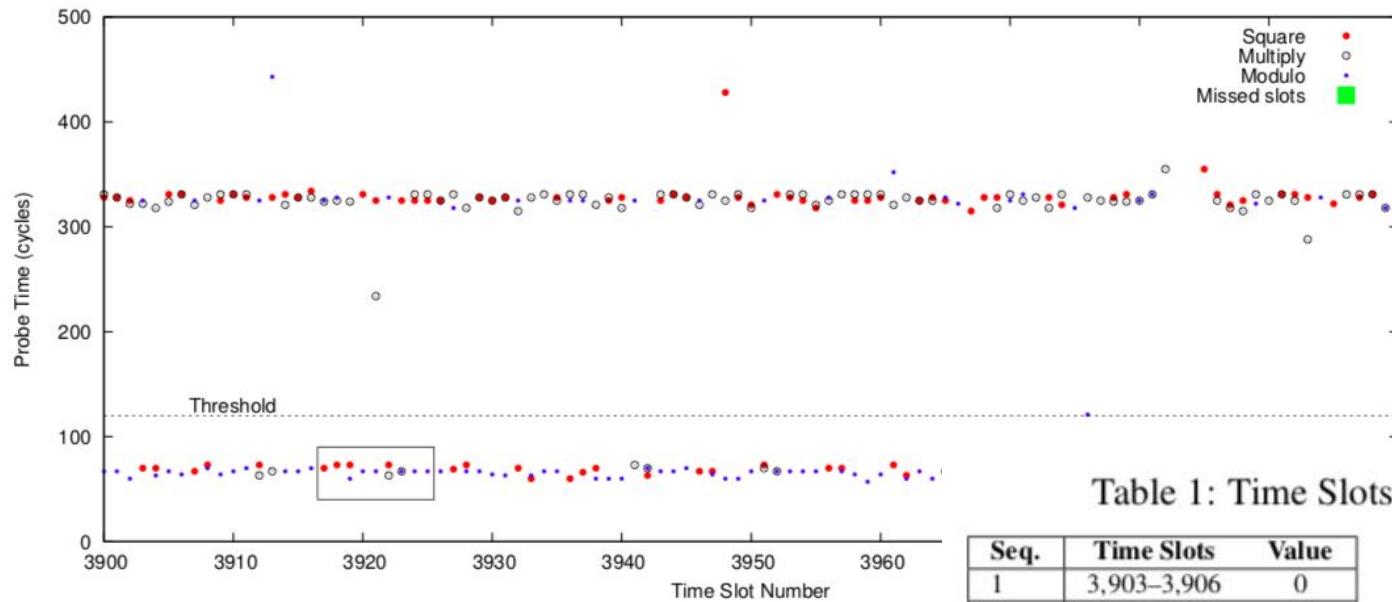


Table 1: Time Slots for Bit Sequence

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 clflush
 - Do we need clflush?
- Non-inclusive cache memories
 - AMD
 - Intel i9 versions
- Fuzzing Clocks
- Software Diversification
 - Permute location of objects in memory (statically and dynamically)

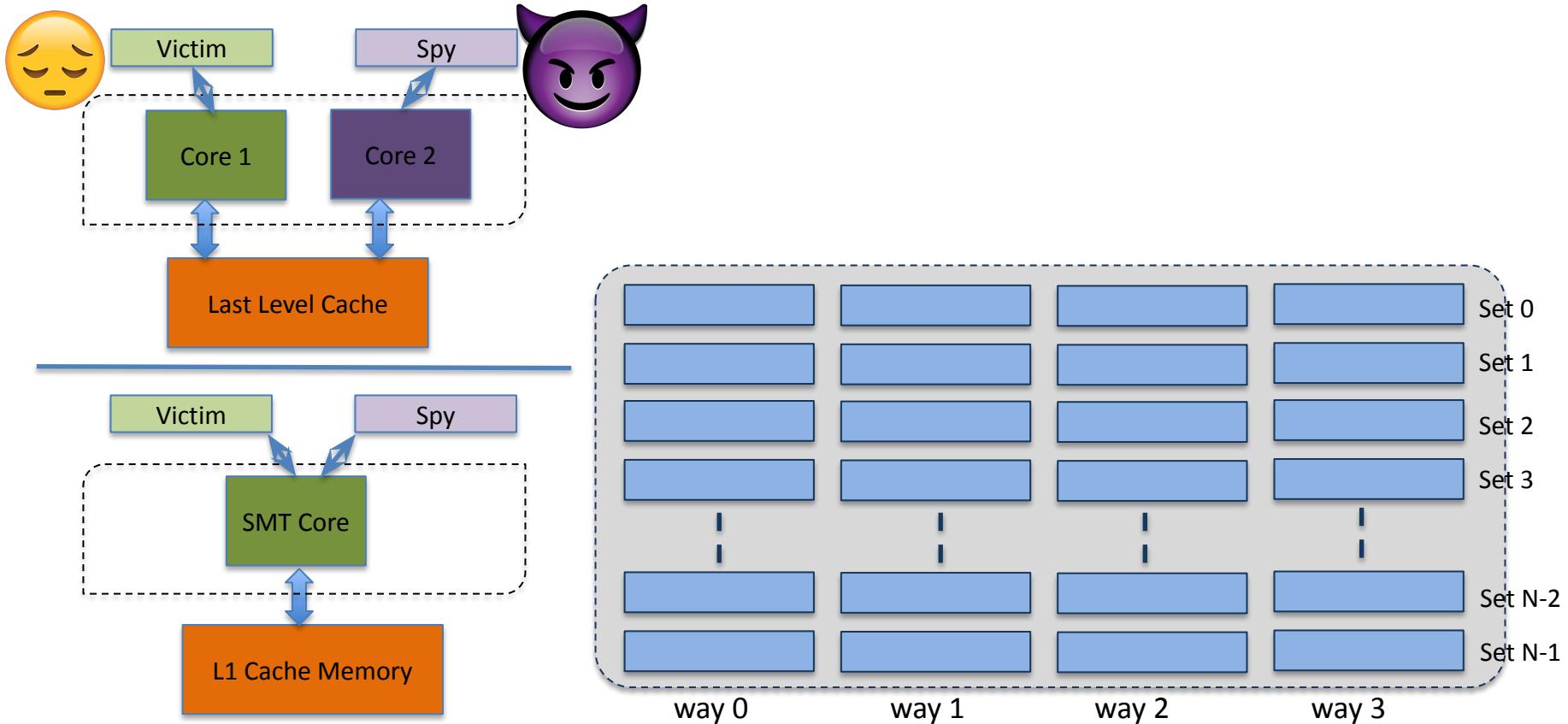
Cache Collision Attacks

Chester Rebeiro
IIT Madras

Cache Collision Attacks

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

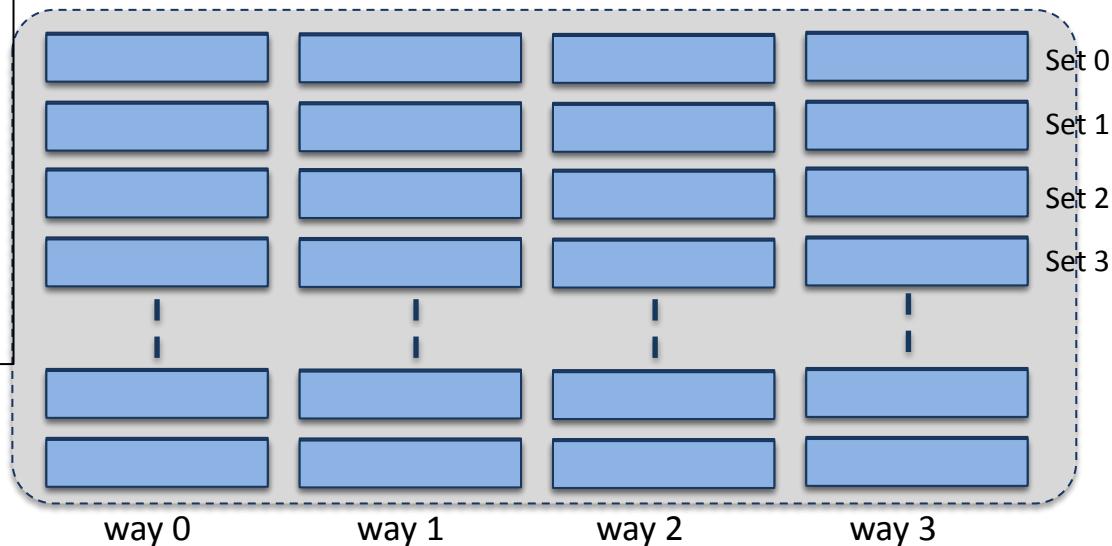
Prime + Probe Attack



Prime Phase



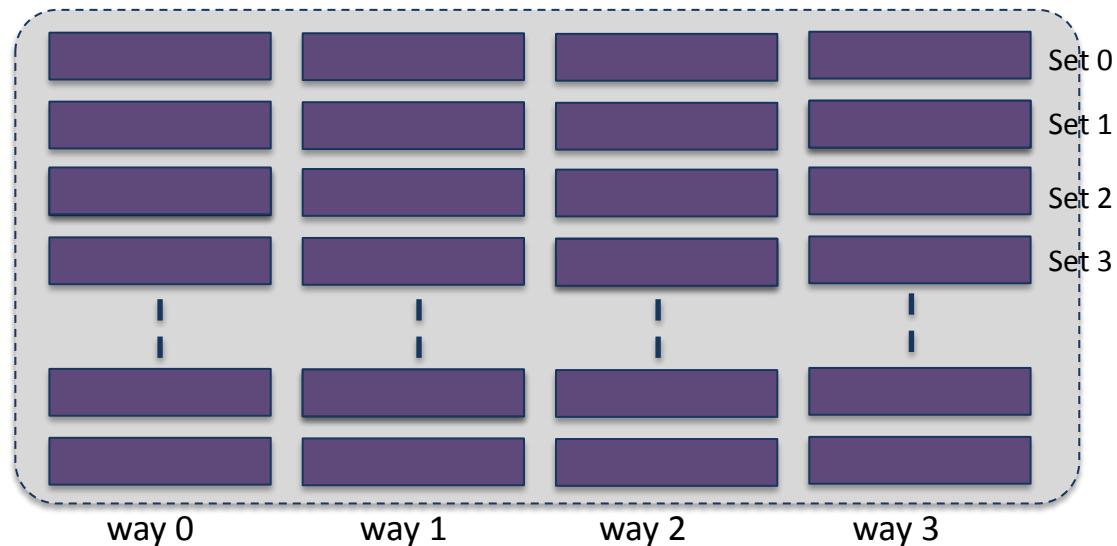
```
While(1) {
    for(each cache set) {
        start = time();
        access all cache ways
        end = time();
        access_time = end - start
    }
    wait for some time
}
```



Victim Execution



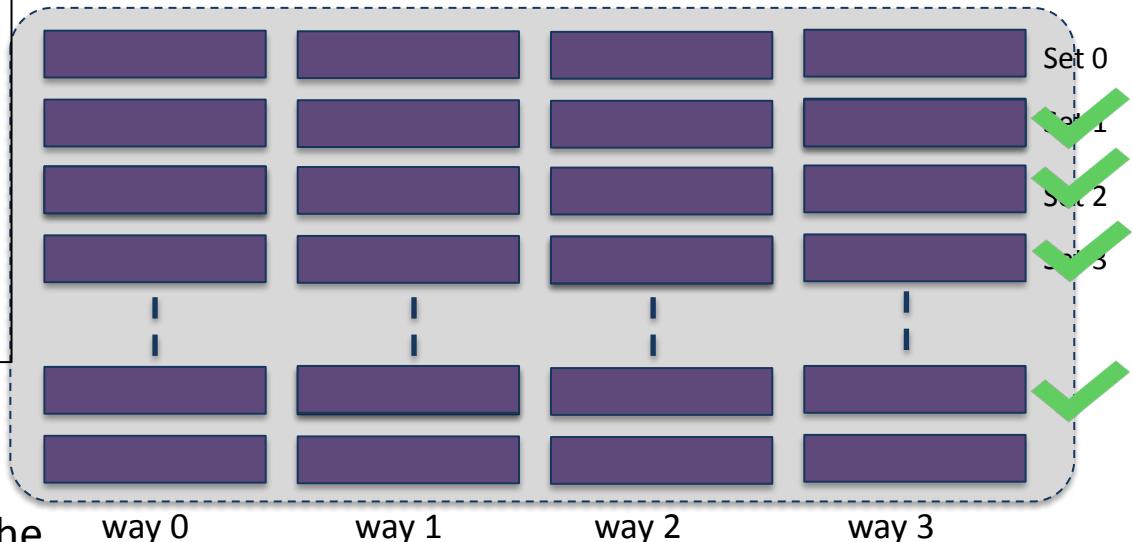
The execution causes some of the spy data to get evicted



Probe Phase

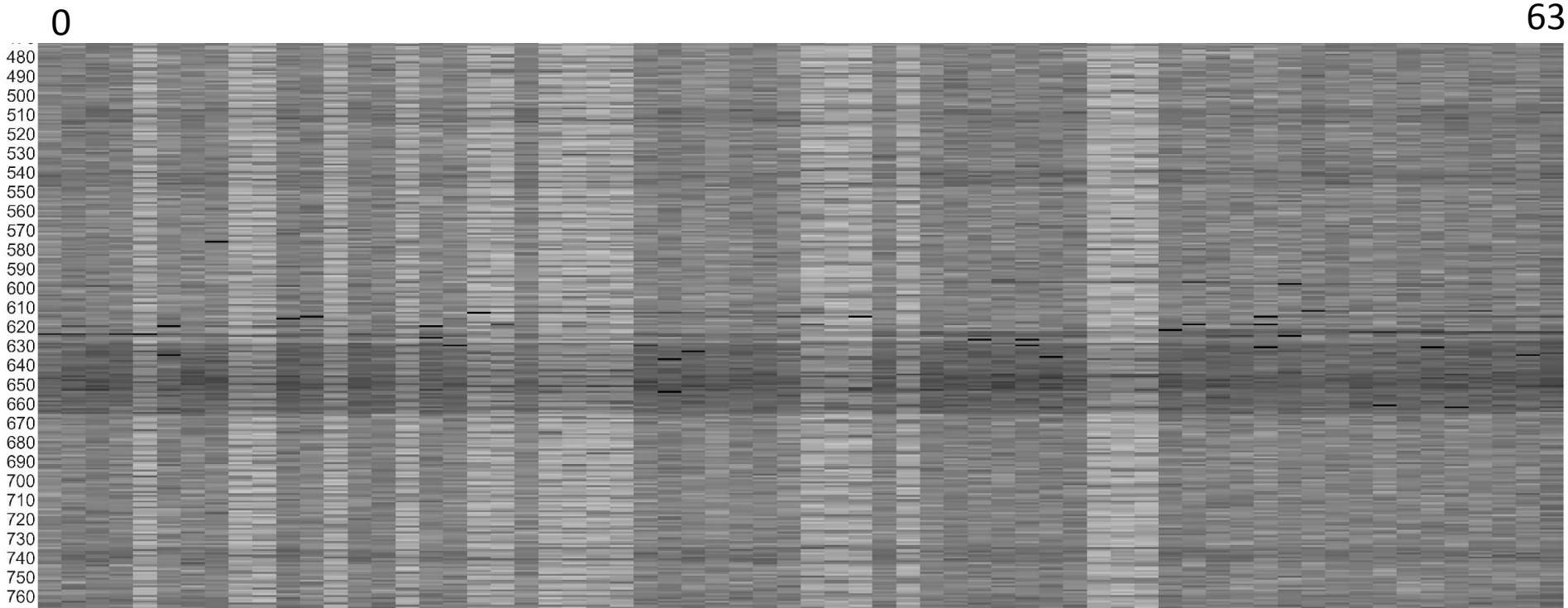


```
While(1) {
    for(each cache set) {
        start = time();
        access all cache ways
        end = time();
        access_time = end - start
    }
    wait for some time
}
```



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

Probe 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;

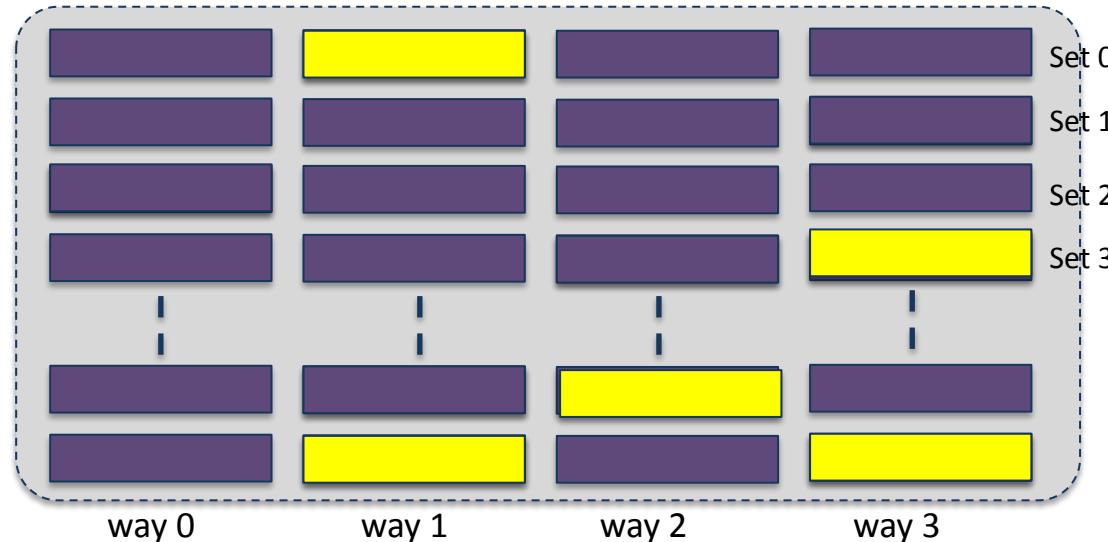
    read(socket, &ct, 1);
    pt = Lookup[key ^ ct];
    return pt;
}
```

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 key ^ ct.

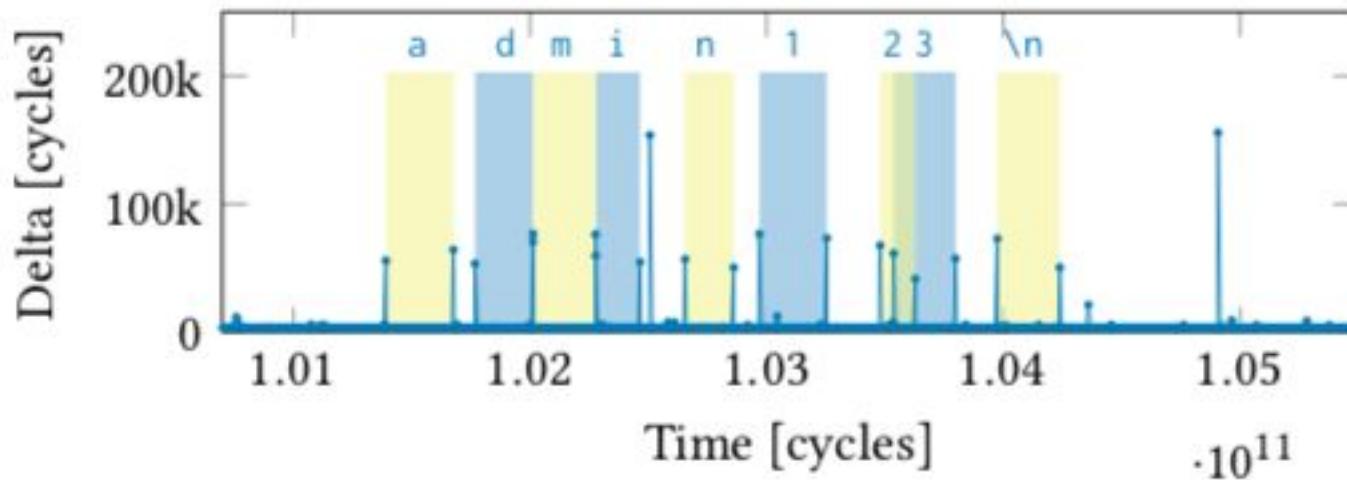
Keystroke Sniffing

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



Keystroke Sniffing

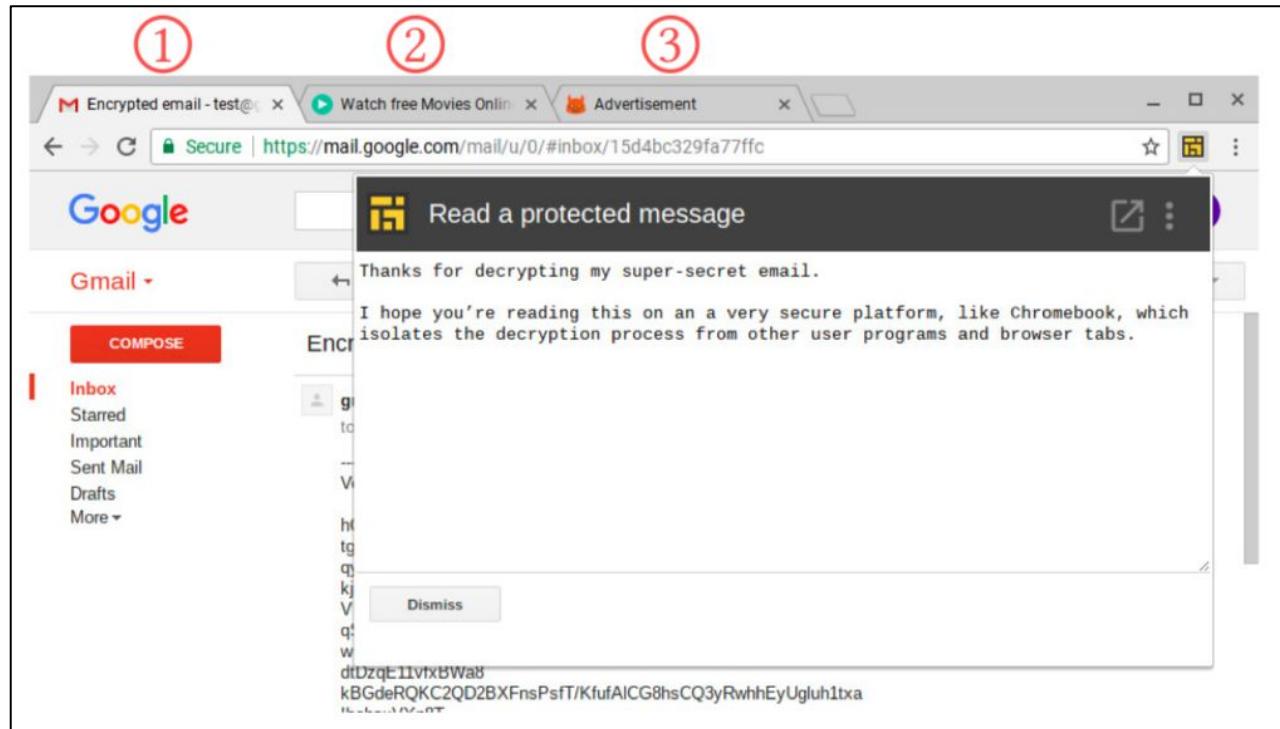
- Regular disturbance seen in Probe Time Plot
- Period between disturbance used to predict passwords



Web Browser Attacks

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

Extract Gmail secret key

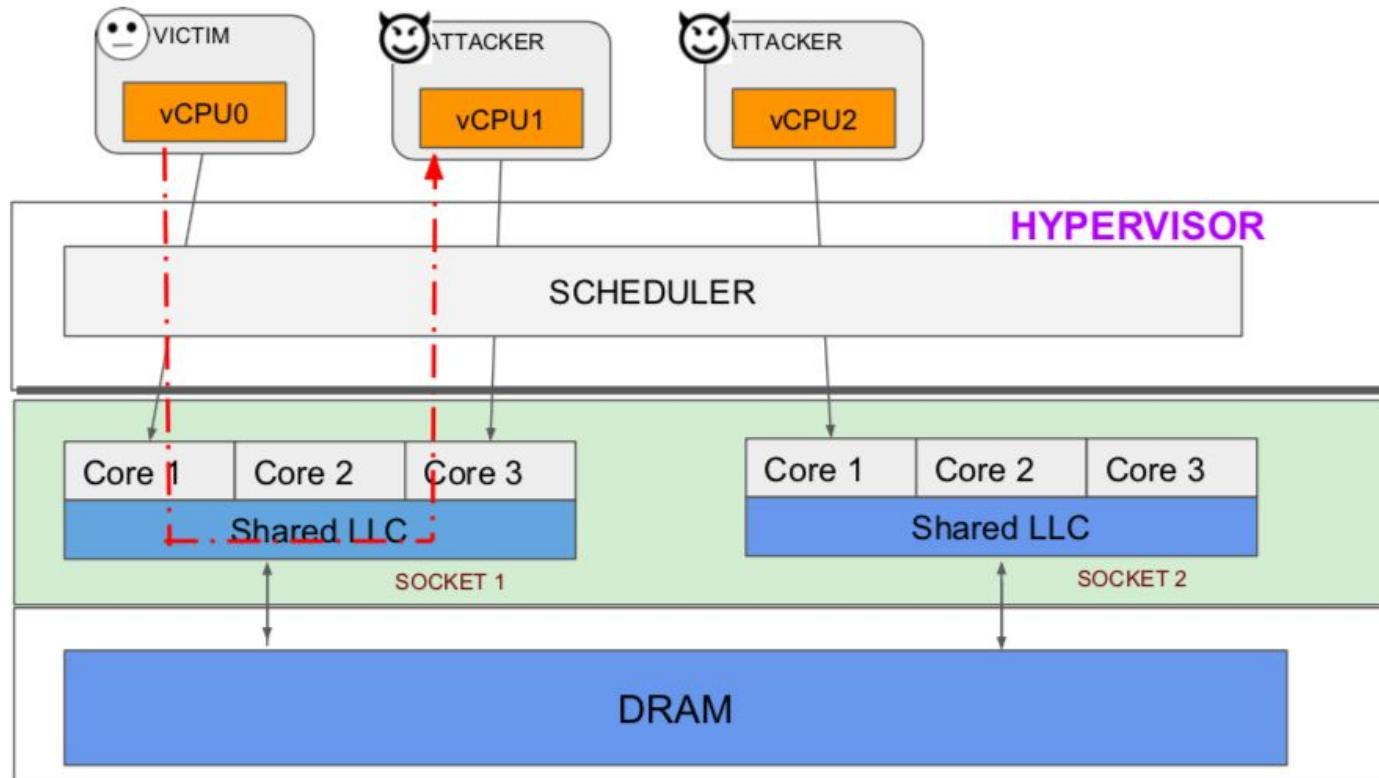


<https://www.cs.tau.ac.il/~tromer/drivebycache/drivebycache.pdf>

Website Fingerprinting

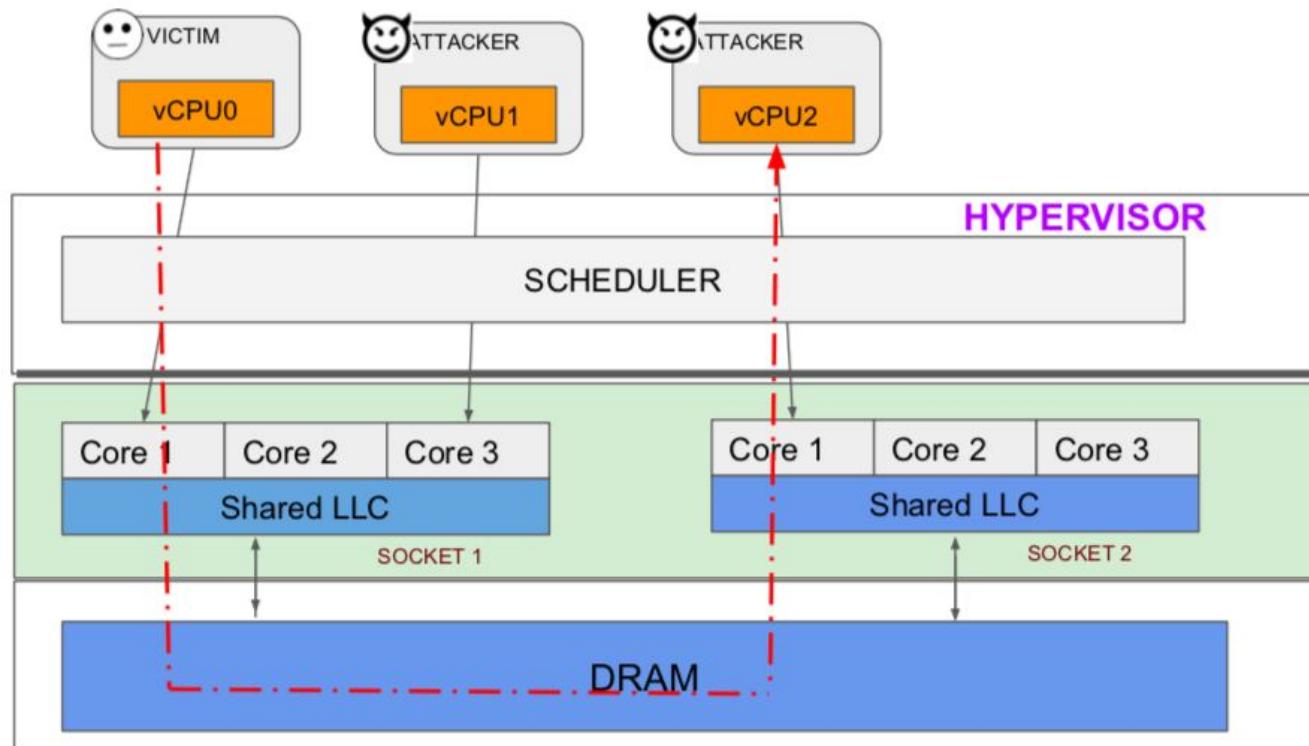
- Privacy: Find out what websites are being browsed.

Cross VM Attacks (Cache)

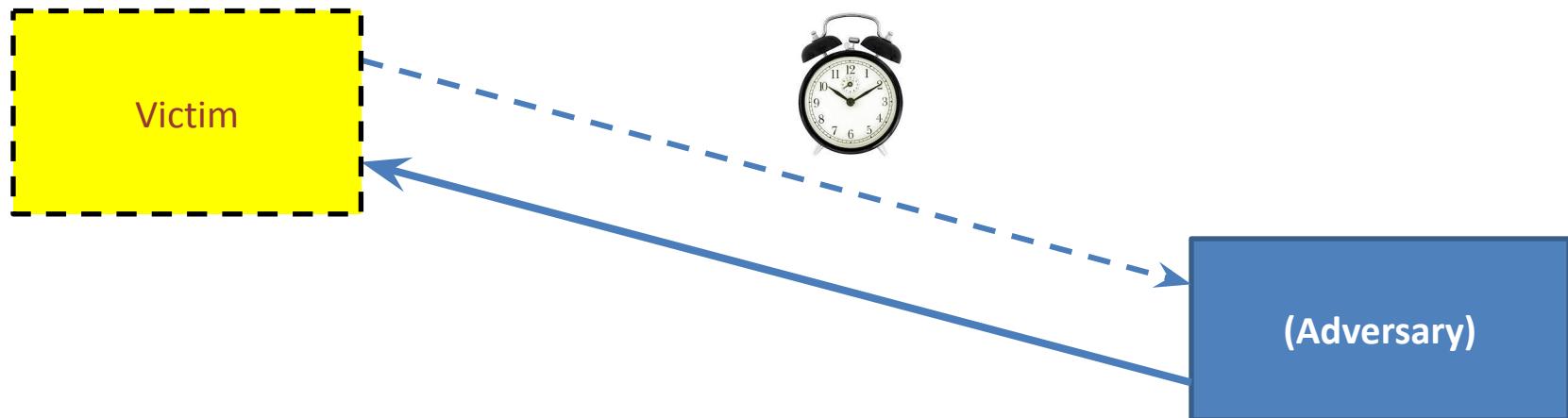


*Ristenpart et.al., Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds, CCS- 2009

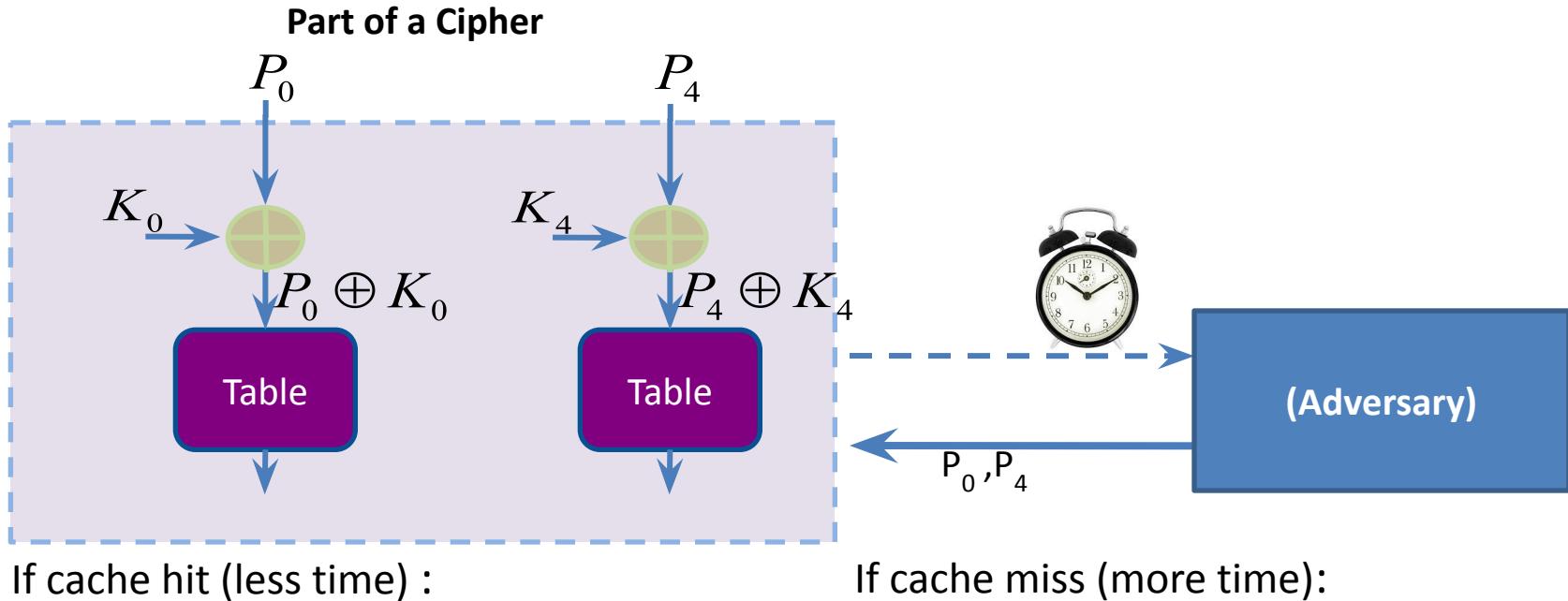
Cross VM Attacks (DRAM)



Internal Collision Attacks



Internal Collisions on a Cipher

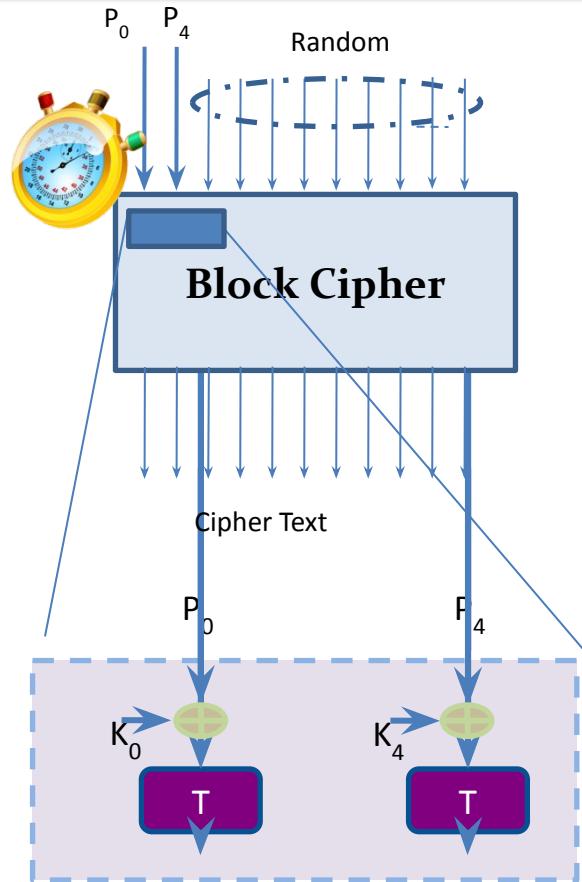


$$\begin{aligned}\langle P_0 \oplus K_0 \rangle &= \langle P_4 \oplus K_4 \rangle \\ \Rightarrow \langle K_0 \oplus K_4 \rangle &= \langle P_0 \oplus P_4 \rangle\end{aligned}$$

$$\begin{aligned}\langle P_0 \oplus K_0 \rangle &\neq \langle P_4 \oplus K_4 \rangle \\ \Rightarrow \langle K_0 \oplus K_4 \rangle &\neq \langle P_0 \oplus P_4 \rangle\end{aligned}$$

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)

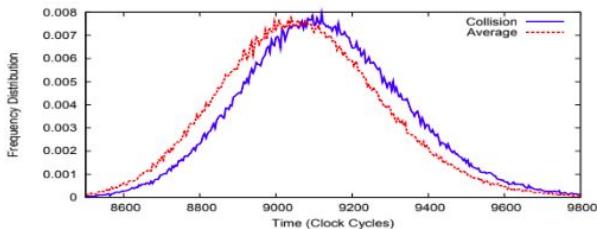


$$\langle K_0 \oplus K_4 \rangle = \langle P_0 \oplus P_4 \rangle$$

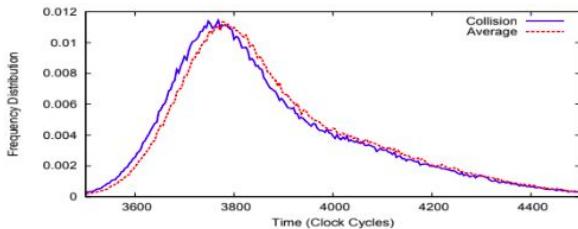
P4	Average Time	DOM
00	2945.3	1.8
10	2944.4	0.9
20	2943.7	0.2
30	2943.7	0.2
40	2944.8	1.3
50	2937.4	-6.3
60	2943.3	-0.2
70	2945.8	2.3
:	:	:
Average : 2943.57		-1.7
Maximum : -6.3		

Easiness to attack ↓

Implementation	Difference of Means
AES (OpenSSL 0.9.8a)	-6.5
DES (PolarSSL 1.1.1)	+11
CAMELLIA (PolarSSL 1.1.1)	19.2
CLEFIA (Ref. Implementation 1.0)	23.4



(a) CLEFIA



(b) AES

Speculation Attacks

Some of the slides motivated from Yuval Yarom's talk on Meltdown and Spectre at the Cyber security research bootcamp 2018

Out-of-order execution

How instructions are fetched

```
load r0, addr1  
mov r2, r1  
add r2, r2, r3  
store r1, add2  
sub r4, r5, r6
```

inorder

How they may be executed

```
sub r4, r5, r6  
store r1, add2  
mov r2, r1  
add r2, r2, r3  
load r0, addr1
```

out-of-order

How the results are committed

```
r0  
r2  
r2  
addr2  
r4
```

order restored

*Out the processor core, execution looks in-order
Insider the processor core, execution is done out-of-order*

Speculative Execution

```
    cmp r0, r1
    jnz label
    load r0, addr1
    mov r2, r1
    add r2, r2, r3
    store r1, add2
    sub r4, r5, r6
```

label:
more instructions

Speculative execution (transient instructions)

```
    cmp r0, r1
    jnz label
    load r0, addr1
    mov r2, r1
    add r2, r2, r3
    store r1, add2
    sub r4, r5, r6
    :
    :
    :
label:
more instructions
```

How instructions are executed

r0
r2
r2
add2
r4

•

How results are committed when speculation is **correct**

Speculative Execution

```
cmp r0, r1  
jnz label  
load r0, addr1  
mov r2, r1  
add r2, r2, r3  
store r1, add2  
sub r4, r5, r6  
:  
:  
:  
label:  
more instructions
```

How instructions are fetched

Speculative execution
(transient instructions)

```
cmp r0, r1  
jnz label  
load r0, addr1  
mov r2, r1  
add r2, r2, r3  
store r1, add2  
sub r4, r5, r6  
:  
:  
:  
label:  
more instructions
```

How instructions are executed

Speculated results discarded

How results are committed when speculation is **incorrect**

Speculative Execution

```
cmp r0, r1  
div r0, r1  
load r0, addr1  
mov r2, r1  
add r2, r2, r3  
store r1, add2  
sub r4, r5, r6  
:  
:  
:  
label:  
more instructions
```

How instructions are fetched

Speculative execution

```
cmp r0, r1  
div r0, r1  
load r0, addr1  
mov r2, r1  
add r2, r2, r3  
store r1, add2  
sub r4, r5, r6  
:  
:  
:  
label:  
more instructions
```

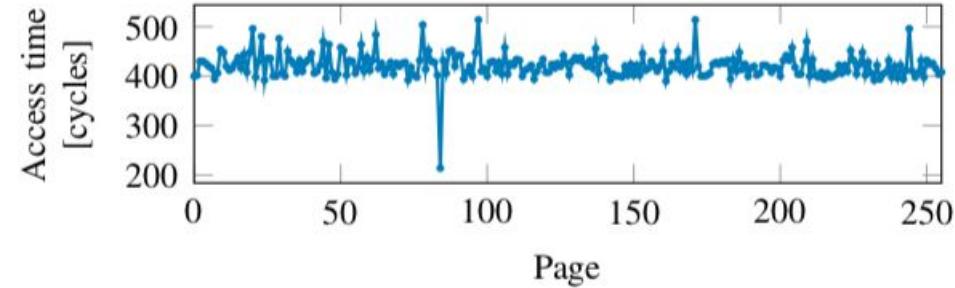
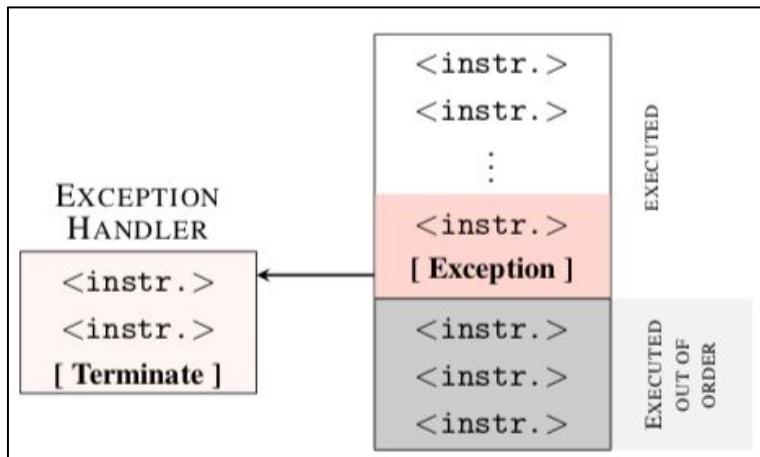
How instructions are executed

Speculated results discarded

How results are committed when speculation is **incorrect**
(eg. If $r1 = 0$)

Speculative Execution and Micro-architectural State

```
1 raise_exception();  
2 // the line below is never reached  
3 access(probe_array[data * 4096]);
```



Even though line 3 is not reached, the micro-architectural state is modified due to Line 3.

Meltdown

Virtual address space of process

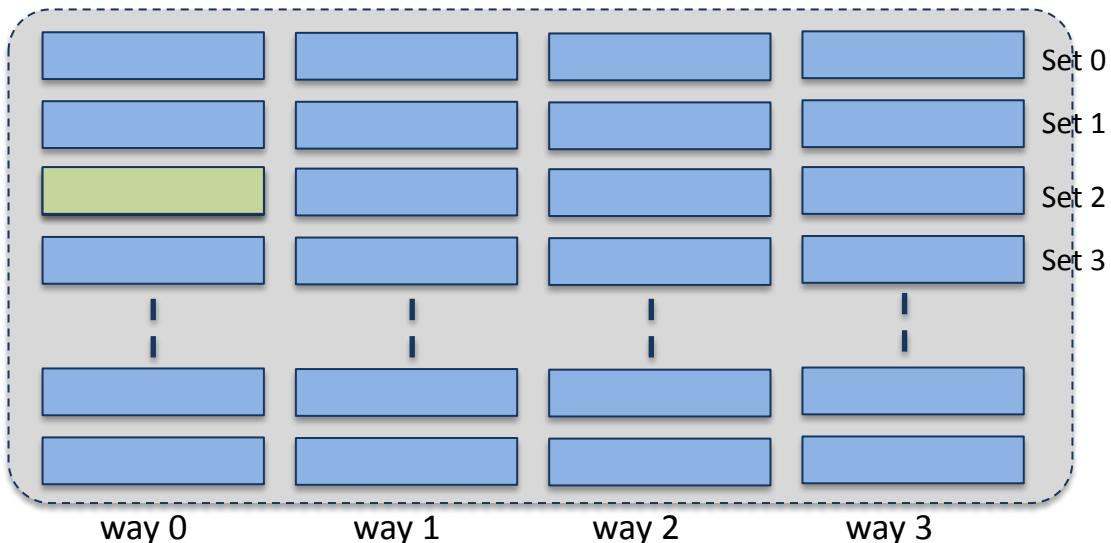
Kernel space

*pointer
array

```
i = *pointer  
y = array[i * 256]
```

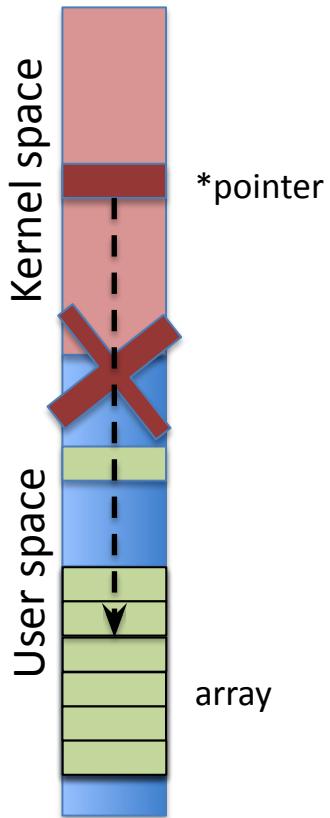
Normal Circumstances

Cache Memory



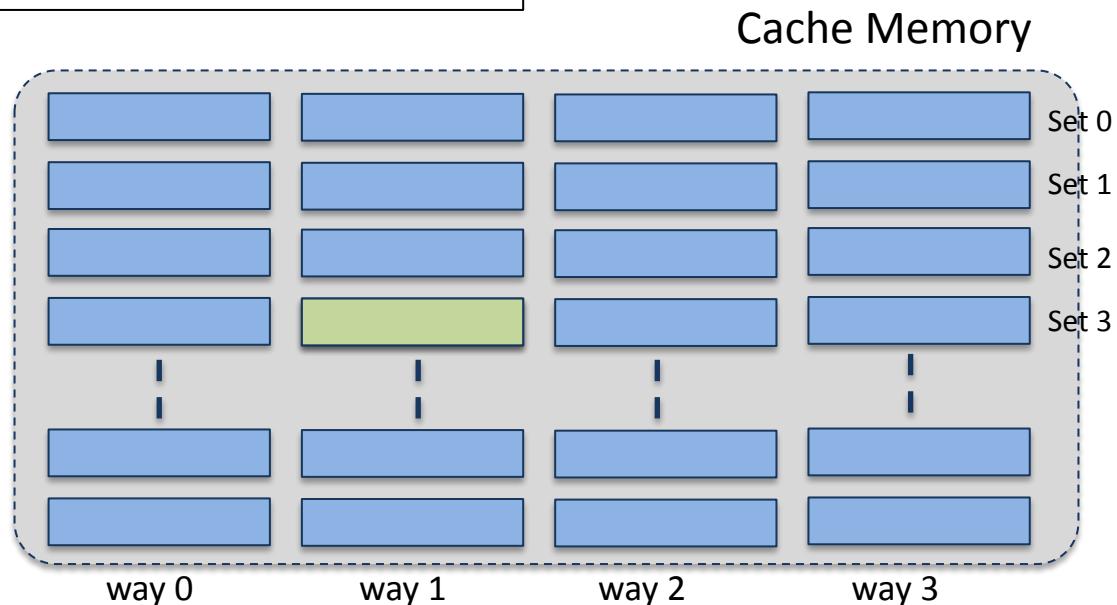
Meltdown

Virtual address space of process



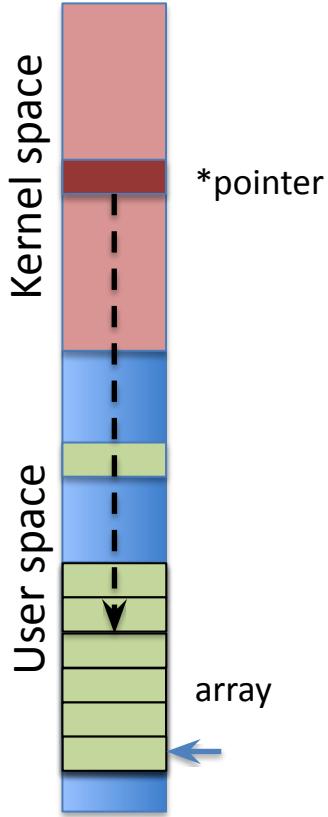
```
i = *pointer  
y = array[i * 256]
```

Not normal Circumstances



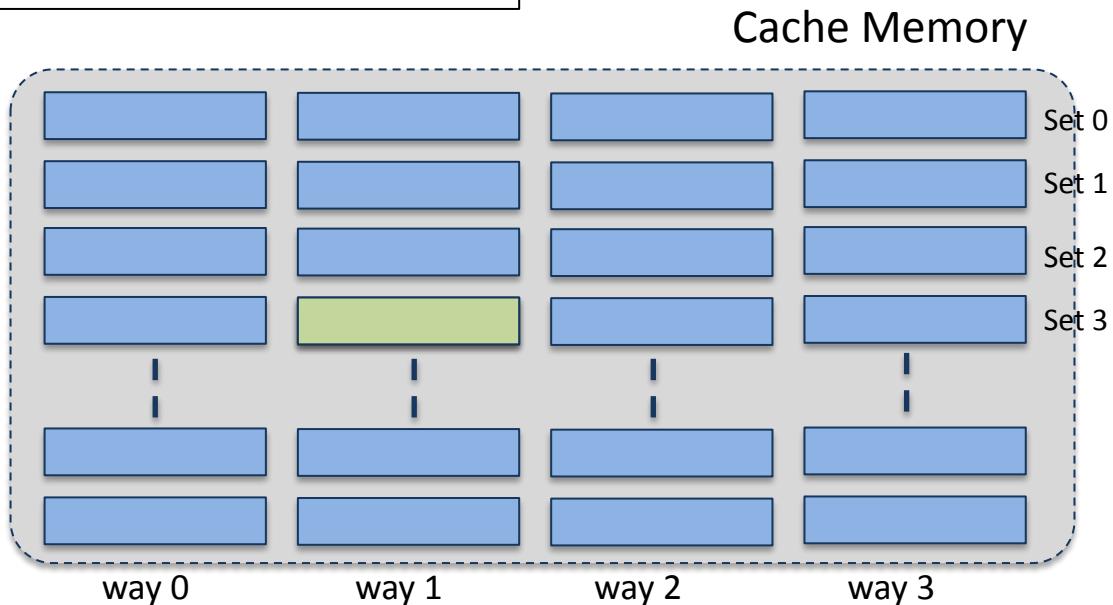
Meltdown

Virtual address space of process



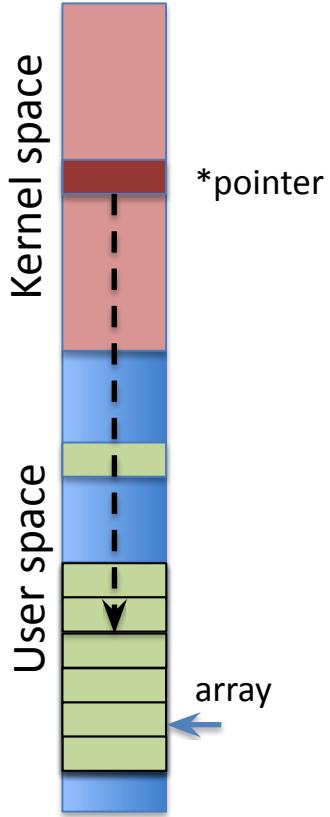
```
i = *pointer  
y = array[i * 256]
```

Not normal Circumstances



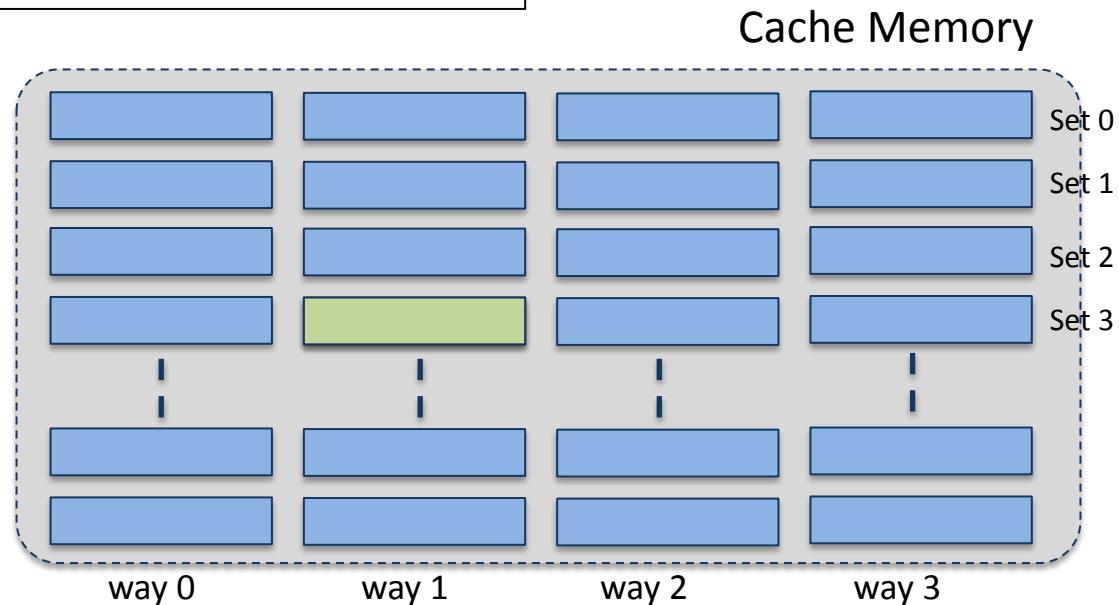
Meltdown

Virtual address space of process



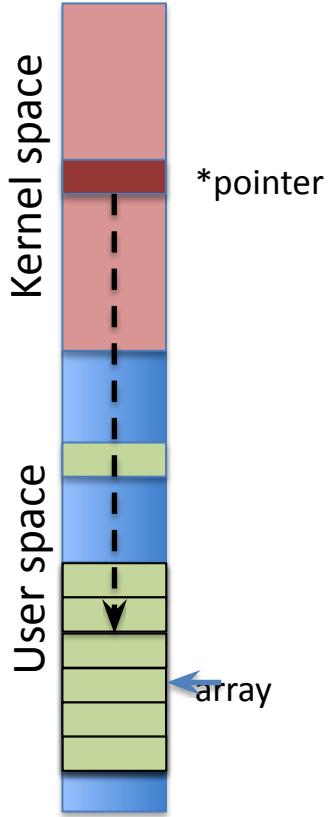
```
i = *pointer  
y = array[i * 256]
```

Not normal Circumstances



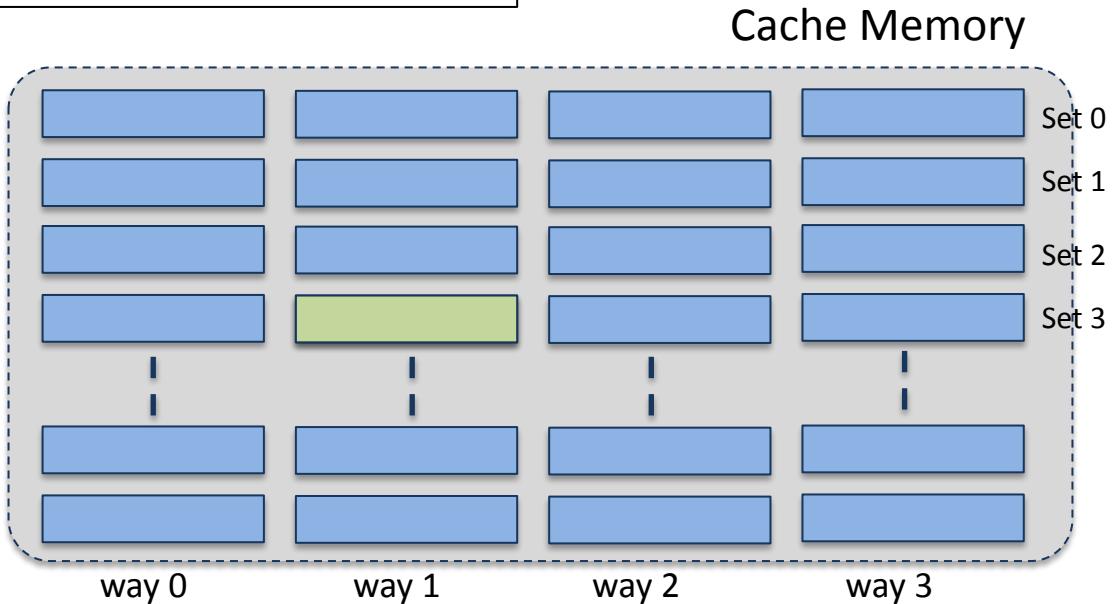
Meltdown

Virtual address space of process



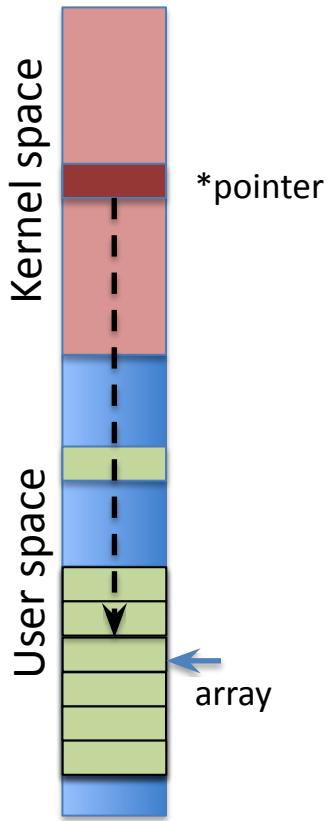
```
i = *pointer  
y = array[i * 256]
```

Not normal Circumstances



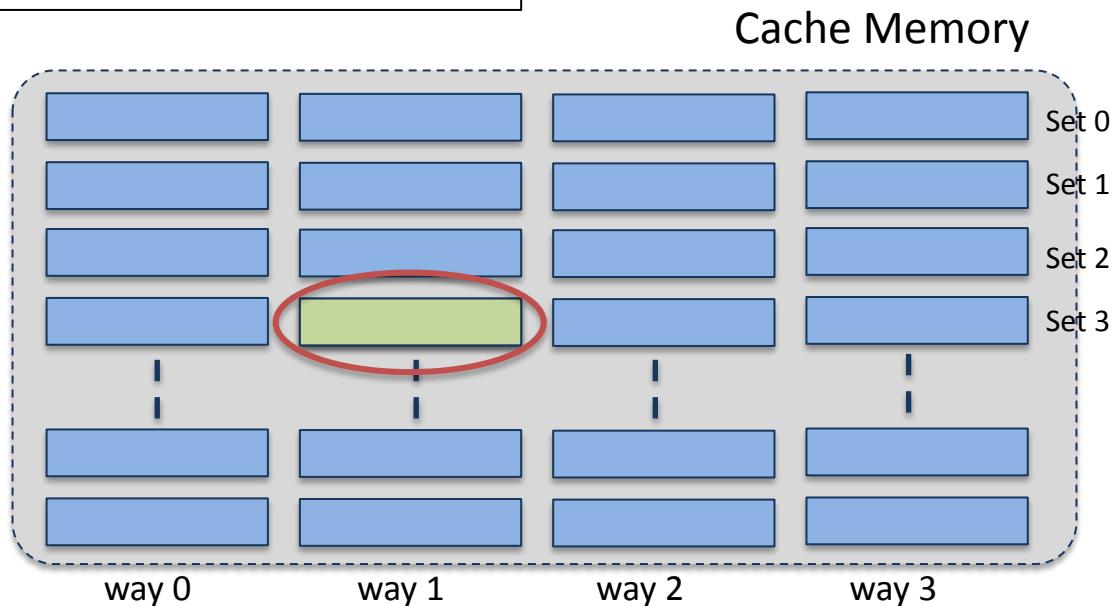
Meltdown

Virtual address space of process



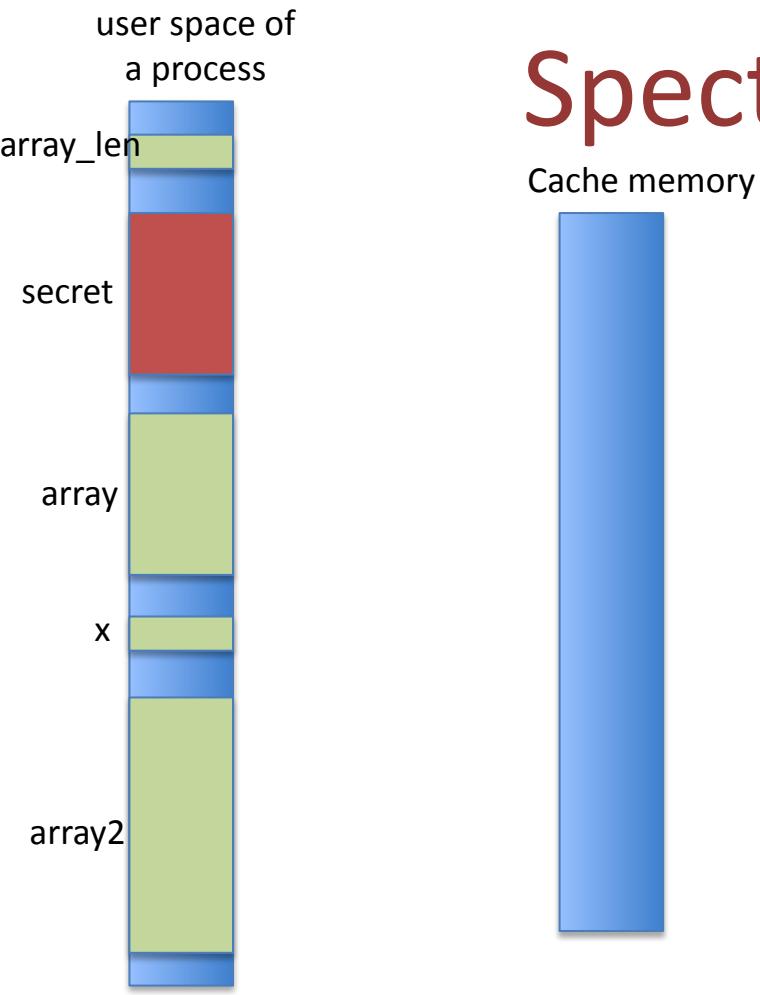
```
i = *pointer  
y = array[i * 256]
```

Not normal Circumstances



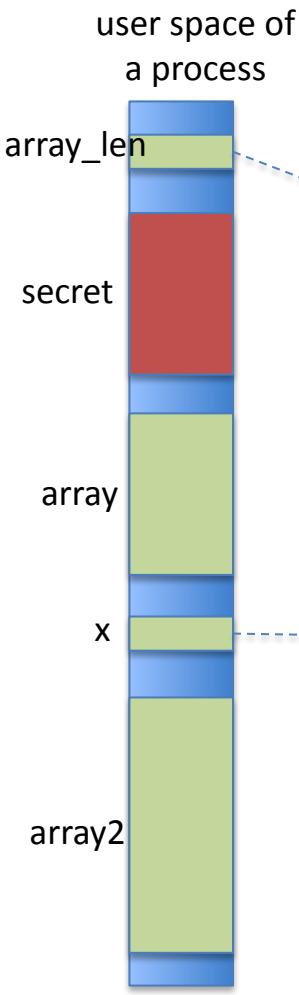
Spectre

Slides motivated from Yuval Yarom's talk on Meltdown and Spectre at the
Cyber security research bootcamp 2018



Spectre (variant 1)

```
if (x < array_len) {  
    i = array[x];  
    y = array2[i * 256];  
}
```

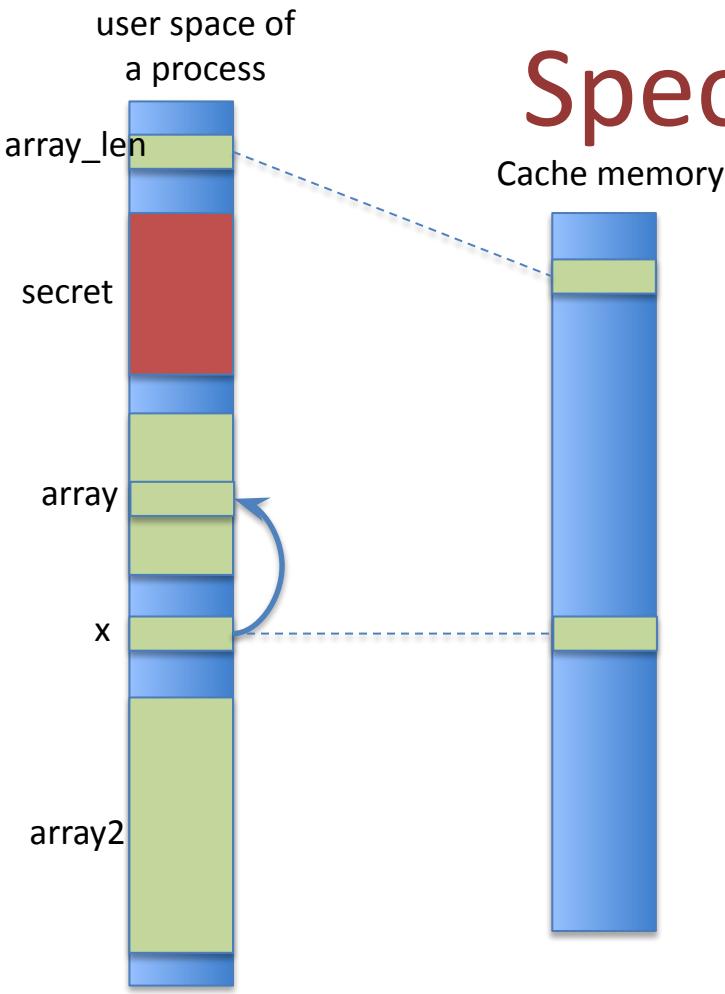


Spectre (variant 1)

Cache memory

```
if (x < array_len) {  
    i = array[x];  
    y = array2[i * 256];  
}
```

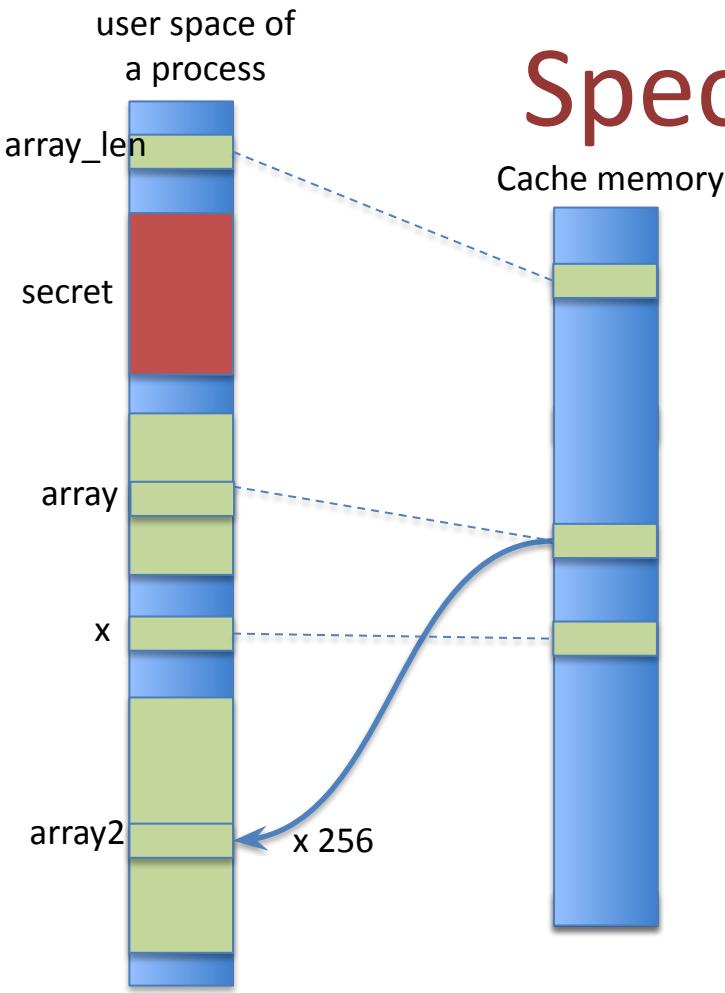
Spectre (variant 1)



Normal Behavior

```
if (x < array_len) {  
    i = array[x];  
    y = array2[i * 256];  
}
```

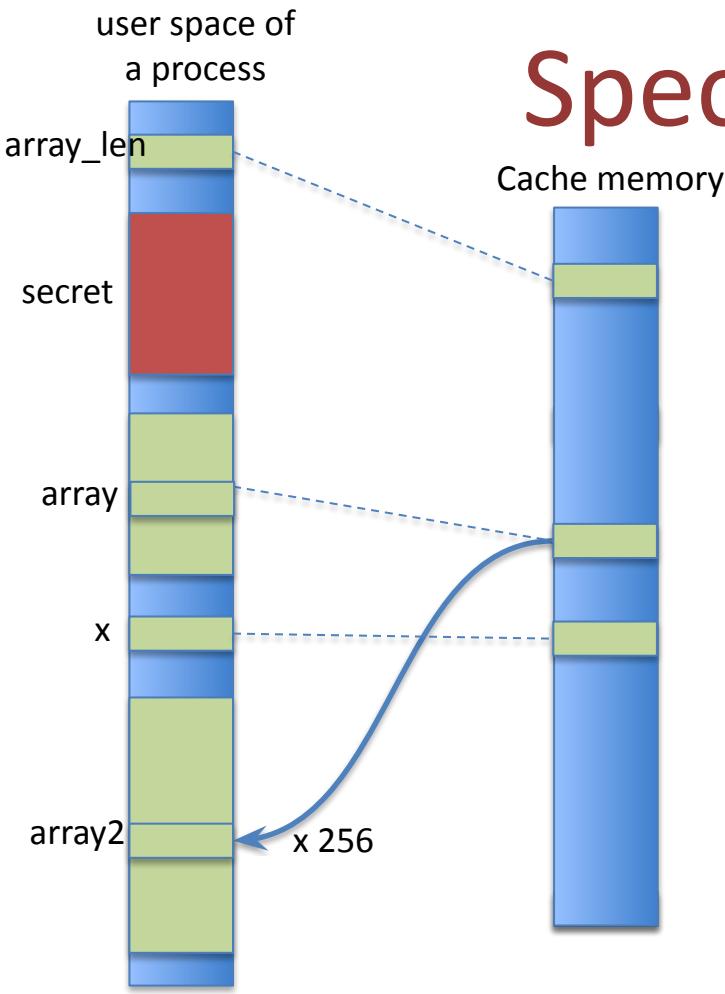
Spectre (variant 1)



Normal Behavior

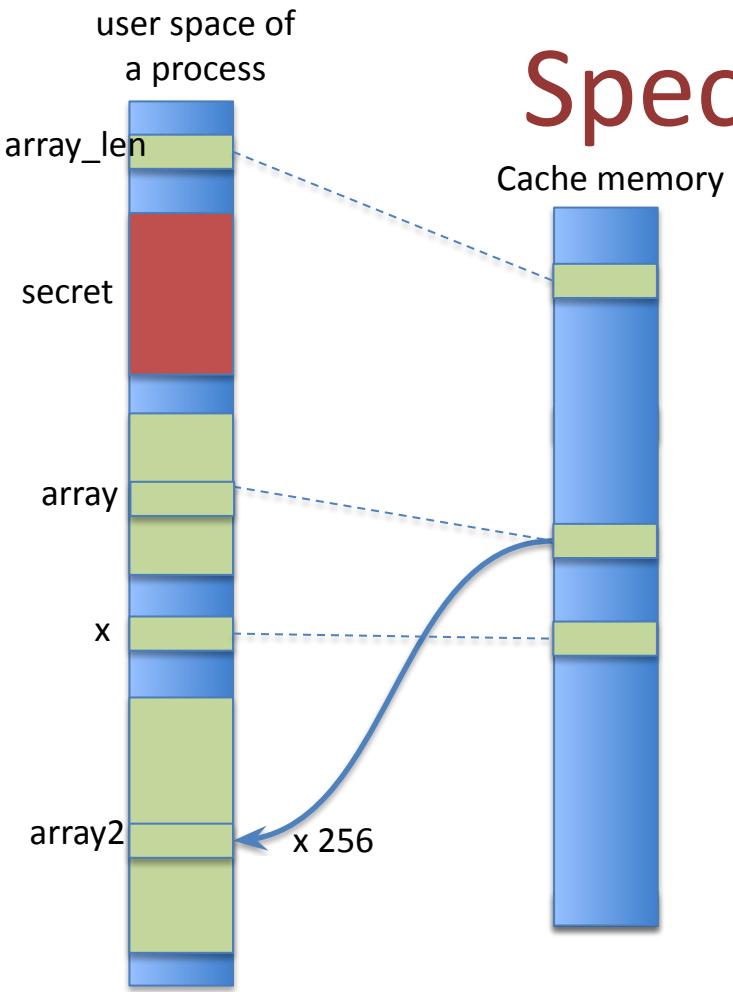
```
if (x < array_len) {  
    i = array[x];  
    y = array2[i * 256];  
}
```

Spectre (variant 1)



Normal Behavior

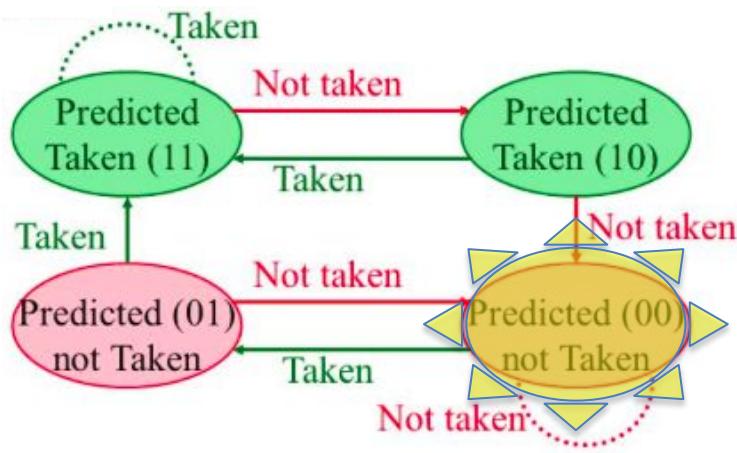
```
if (x < array_len) {  
    i = array[x];  
    y = array2[i * 256];  
}
```

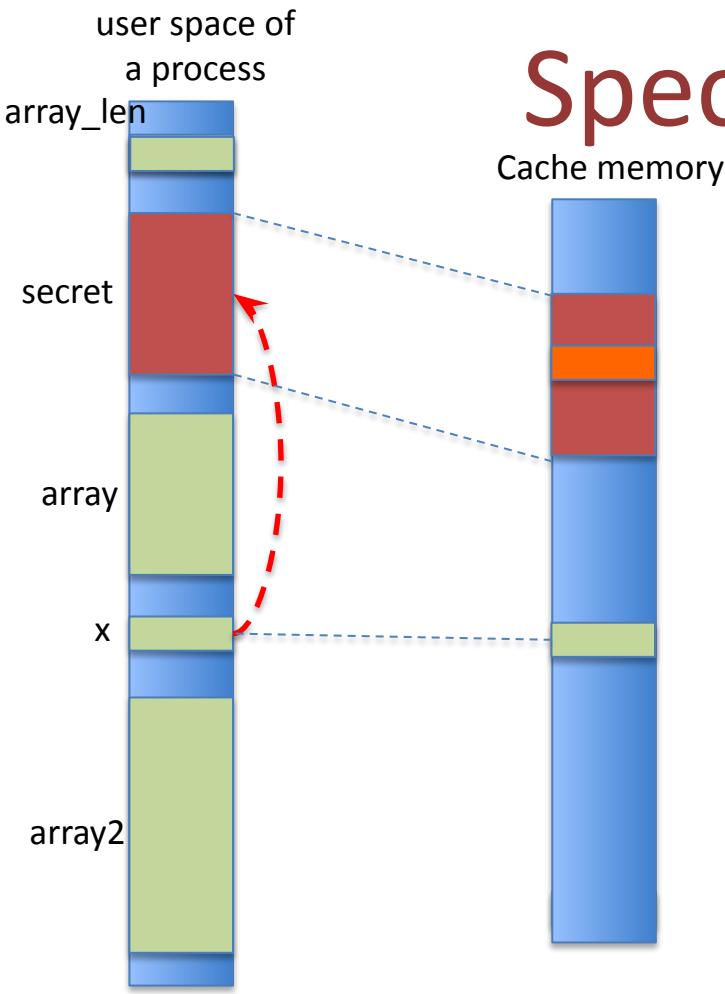


Spectre (variant 1)

Normal Behavior

```
if (x < array_len){  
    i = array[x];  
    y = array2[i * 256];
```





Spectre (variant 1)

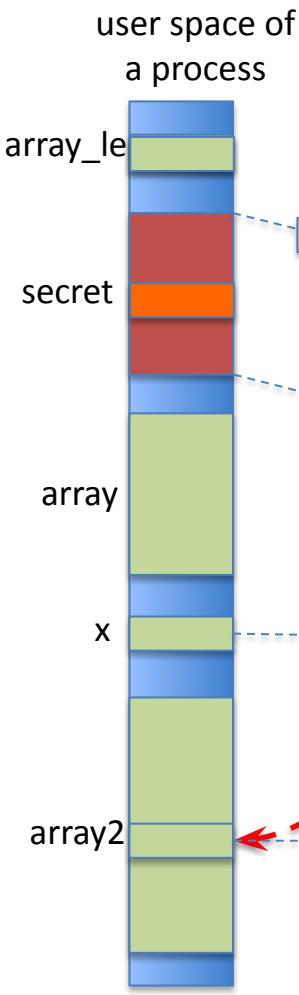
Under Attack

```

if (x < array_len) {
    i = array[x];
    y = array2[i * 256];
}

```

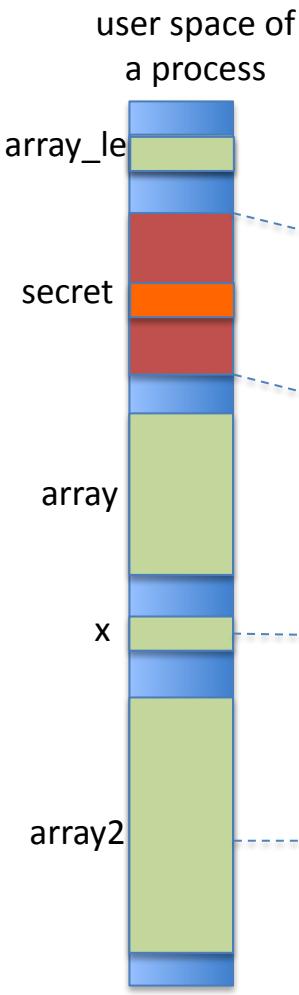
- $x > \text{array_len}$
- array_len not in cache
- secret in cache memory



Spectre (variant 1)

```
if (x < array_len) {  
    i = array[x];  
    y = array2[i * 256];  
}
```

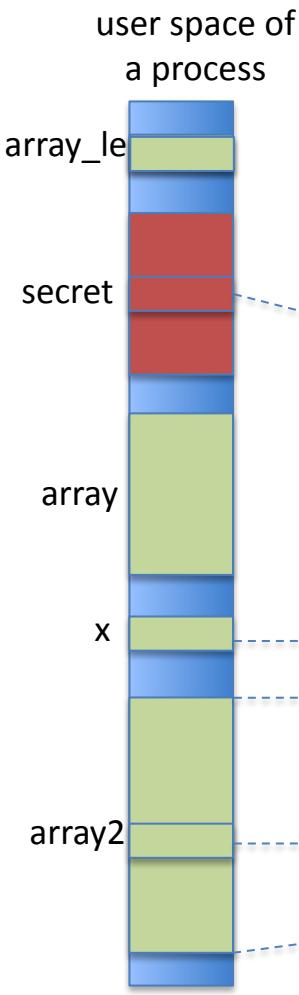




Spectre (variant 1)

```
if (x < array_len) {  
    i = array[x];  
    y = array2[i * 256];  
}
```





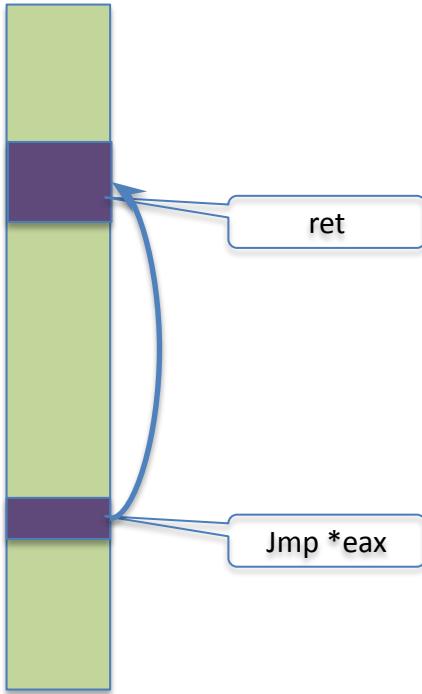
Spectre (variant 1)

```
if (x < array_len) {  
    i = array[x];  
    y = array2[i * 256];  
}
```

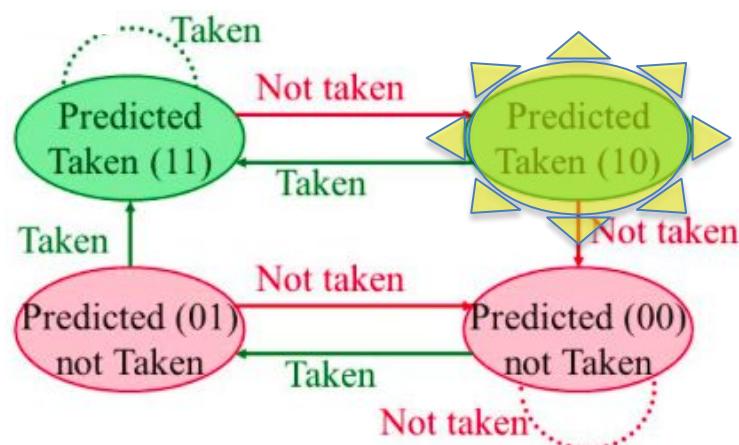
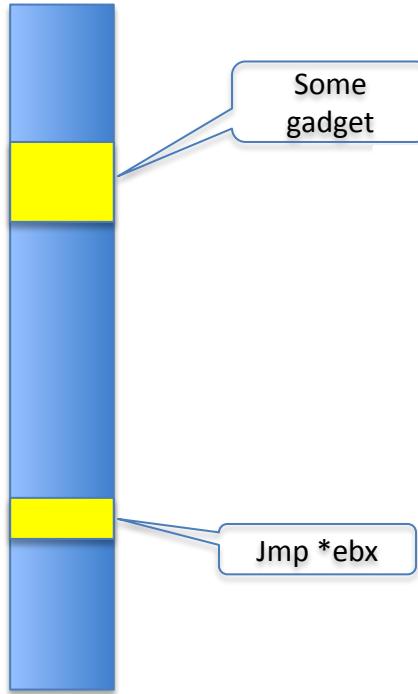
Cache hit found
only here

Spectre (variant 2)

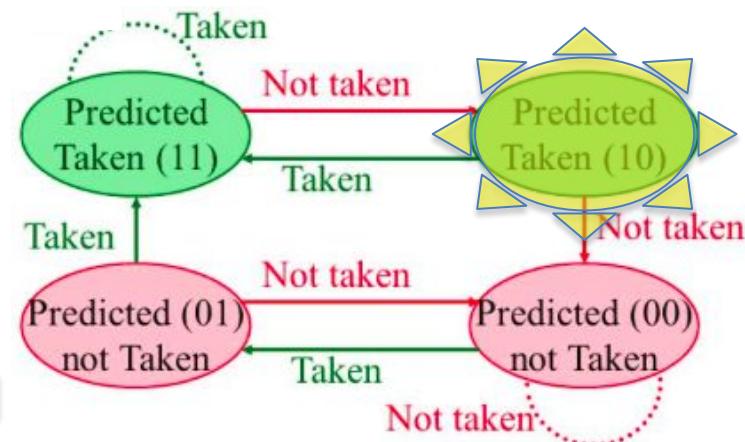
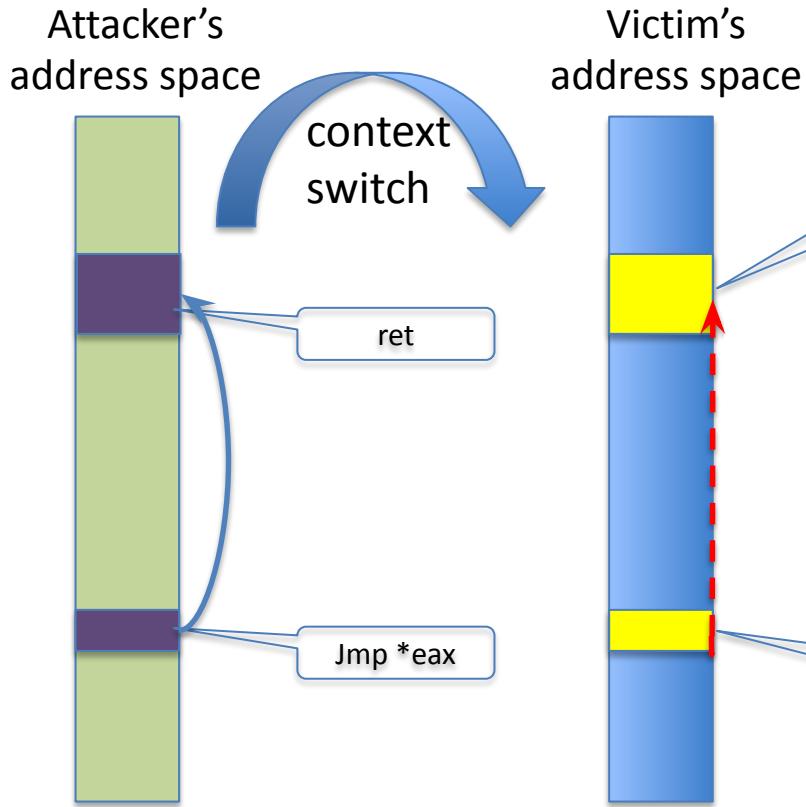
Attacker's address space



Victim's address space

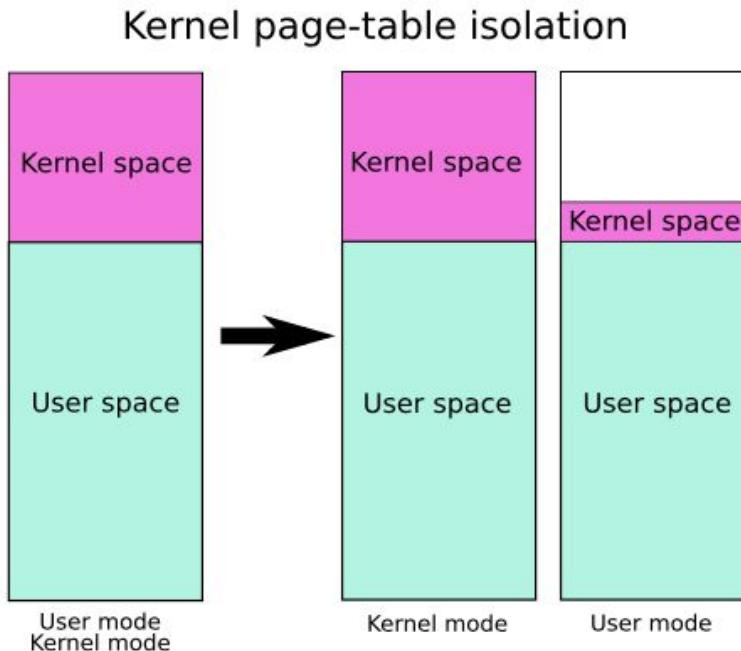


Spectre (variant 2)



Countermeasures

For meltdown: kpti (kernel page table isolation)



Countermeasures

For Spectre (variant 1): compiler patches
use barriers (LFENCE instruction) to prevent speculation
static analysis to identify locations where attackers can control
speculation

Countermeasures

- For Spectre (Variant 2): Separate BTBs for each process
 - Prevent BTBs across SMT threads
 - Prevent user code does not learn from lower security execution

Countermeasures

- For all: at hardware
 - Every speculative load and store should bypass cache and stored in a special buffer known as speculative buffer