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 resouces)
- 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 resouces)
- Enclaves (SGX and Trustzone)

Cache timing attack

Branch prediction attack

Speculation Attacks

Row hammer

Fault Injection Attacks

cold boot attacks

DRAM Row buffer (DRAMA)

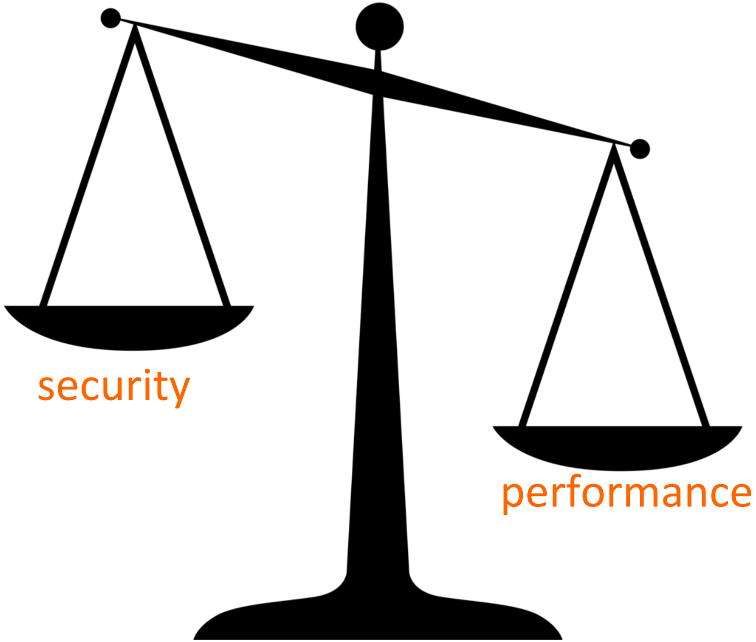
….. and many more

Causes

Most micro-architectural attacks caused by performance optimizations

Others due to inherent device properties

Third, due to stronger attackers

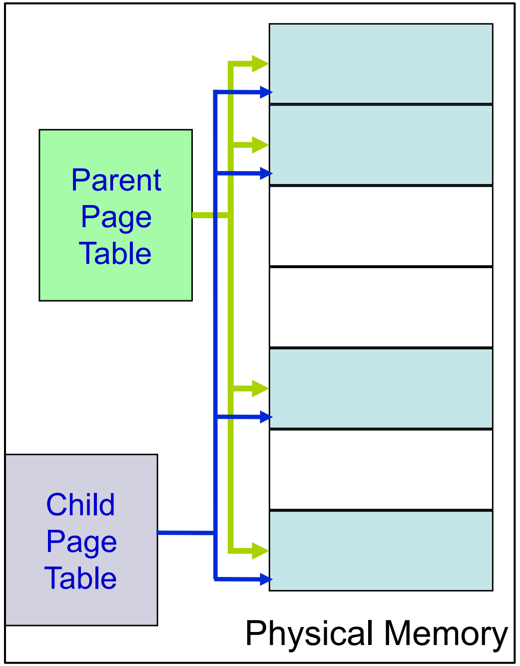


Copy on Write

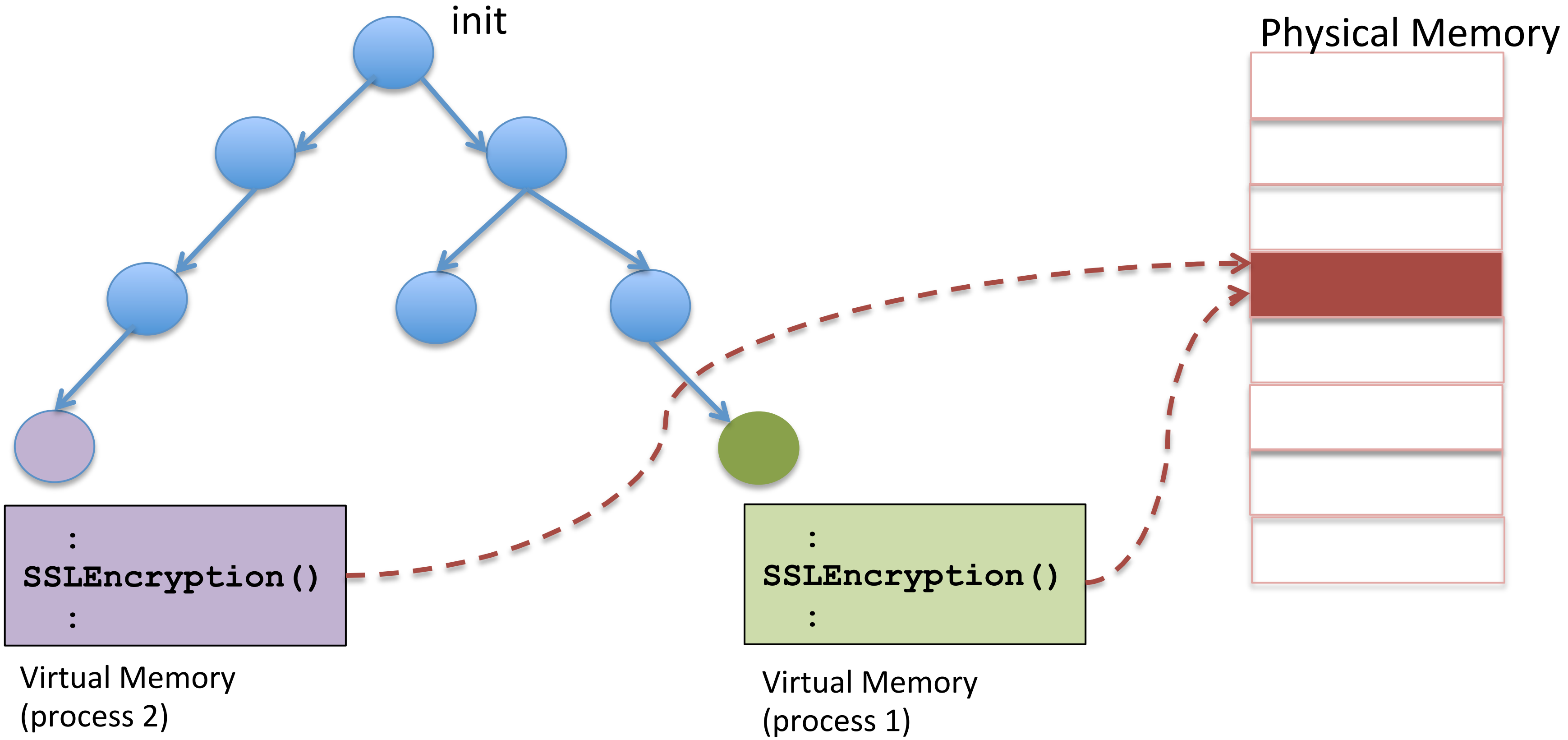
```
if (fork() > 0){
   // in parent process
} else{
   // in child process
}
```

Child created is an exact replica of the parent process.

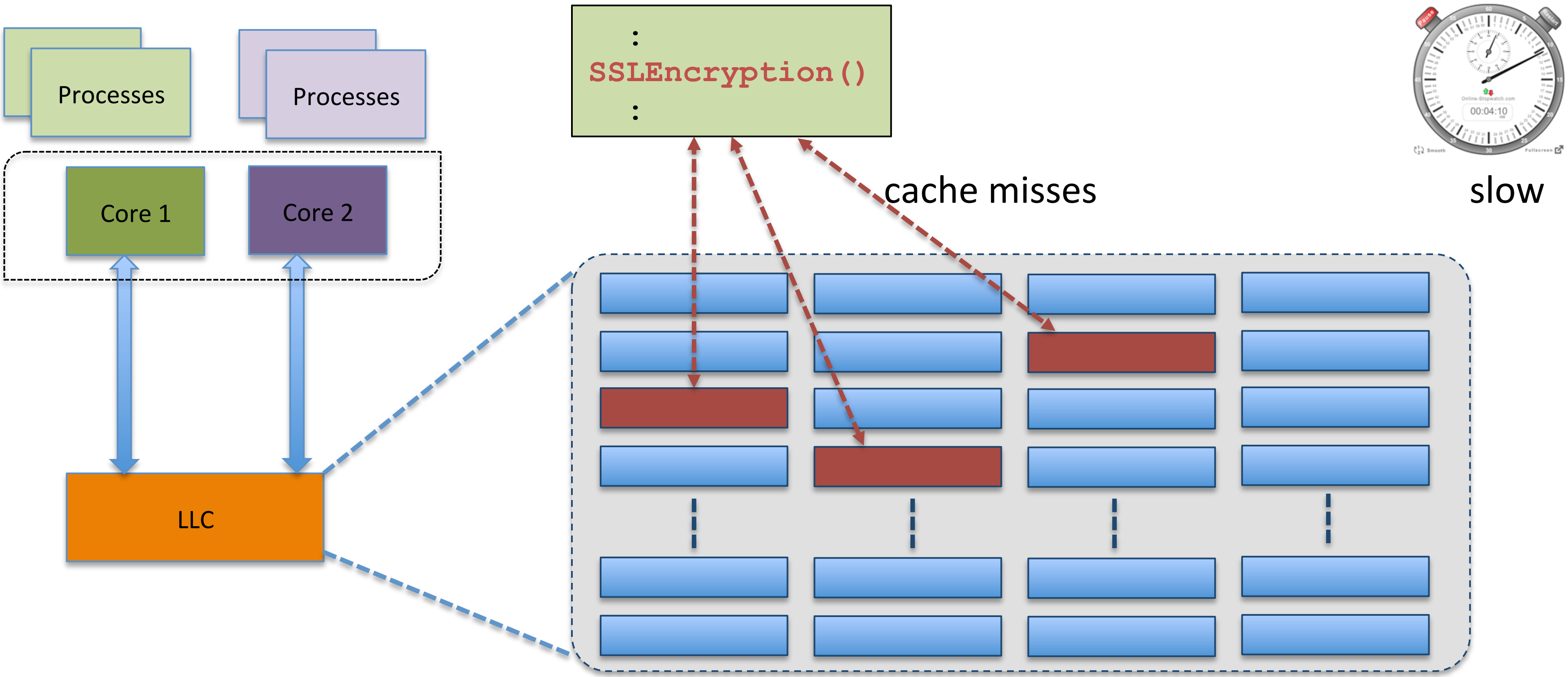
- Page tables of the parent duplicated in the child
- New pages created only when parent (or child) modifies data
    - Postpone copying of pages as much as possible, thus optimizing performance
    - Thus, common code sections (like libraries) would be shared across processes.



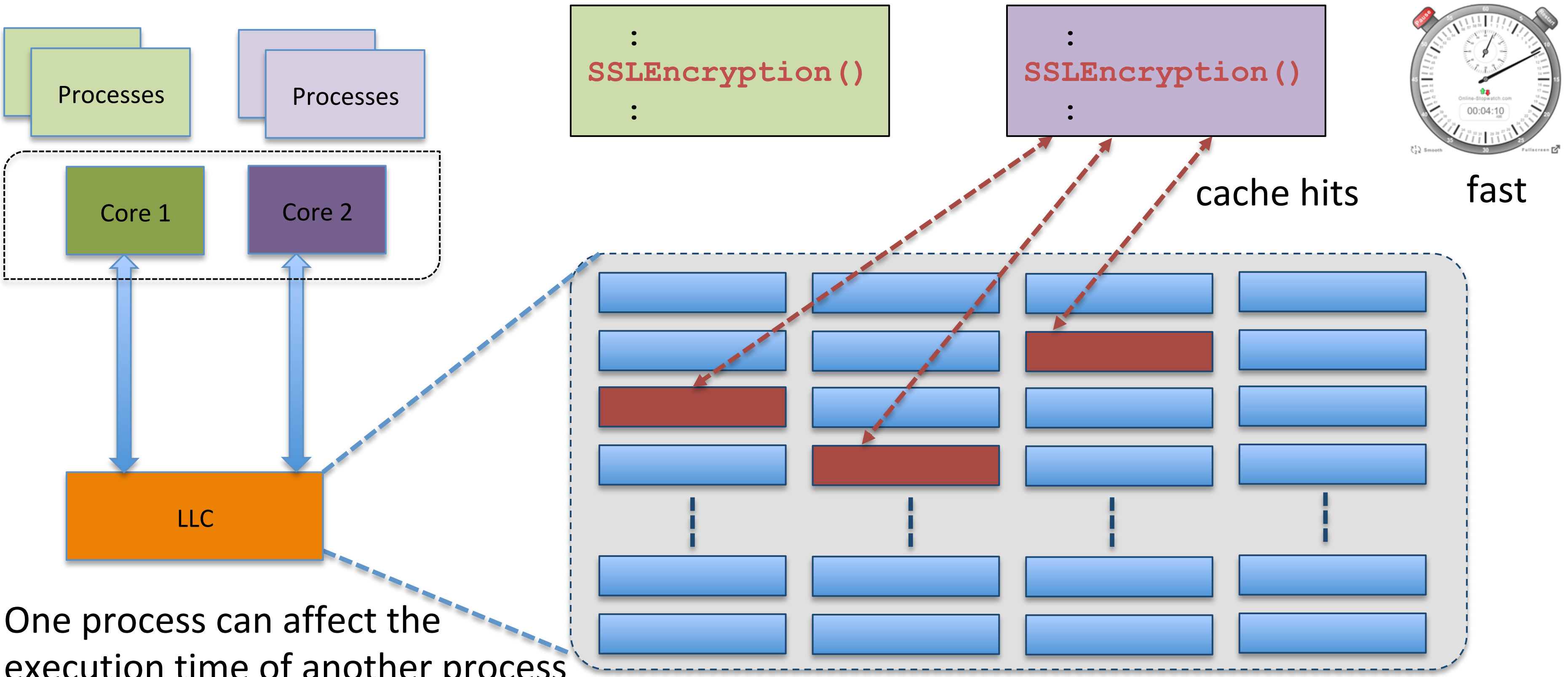
Process Tree



Interaction with the LLC



Interaction with the LLC



One process can affect the execution time of another process

Flush + Reload Attack on LLC

Part of an encryption algorithm

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

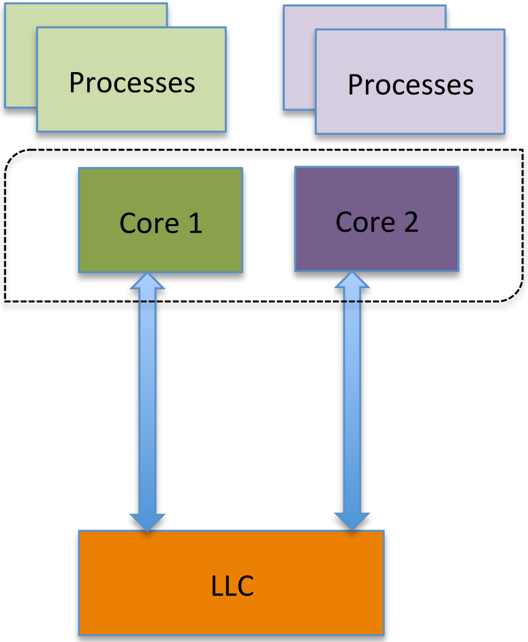
executed only when $e_i = 1$ (lines 8–9)

clflush Instruction

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

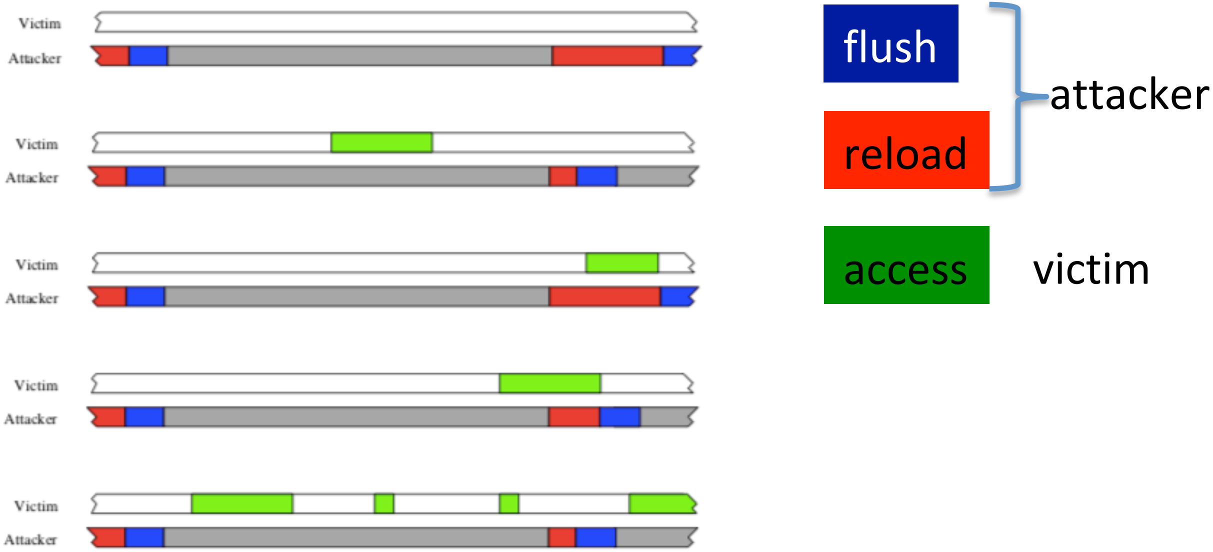
Flush+Reload Attack, Yuval Yarom and Katrina Falkner (https://eprint.iacr.org/2013/448.pdf)

Flush + Reload Attack




```
:
SSLEncryption()
:
```

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



Flush+Reload Attack

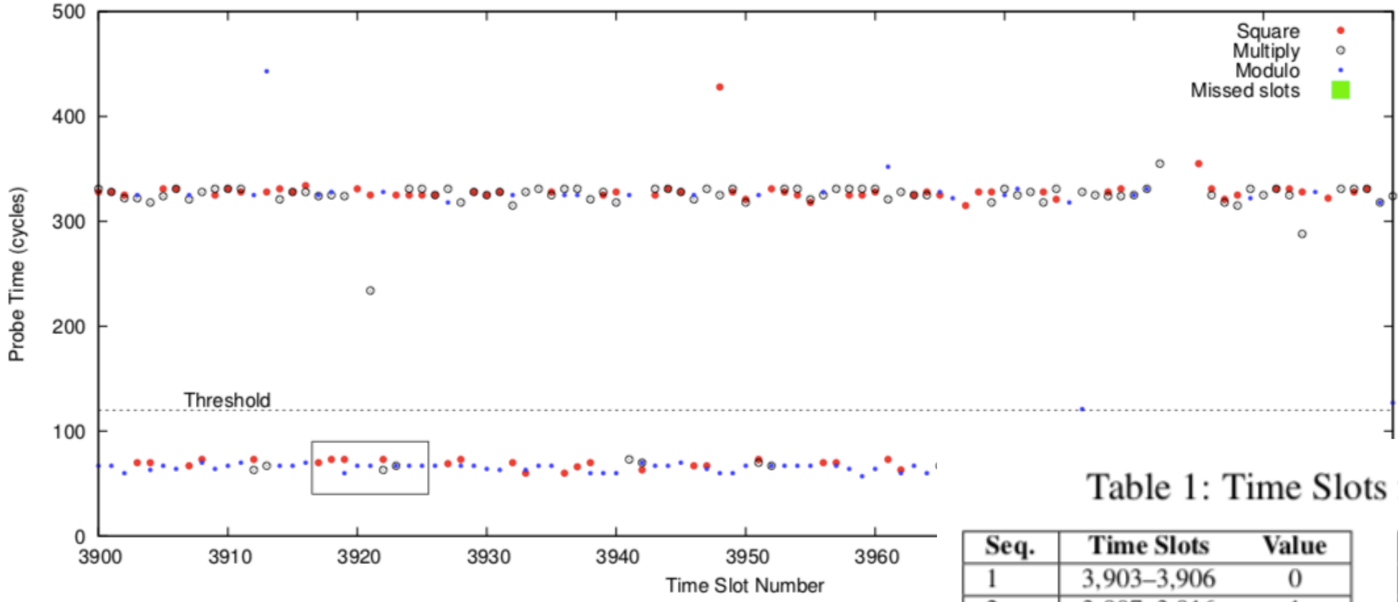


Table 1: Time Slots for Bit Sequence

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

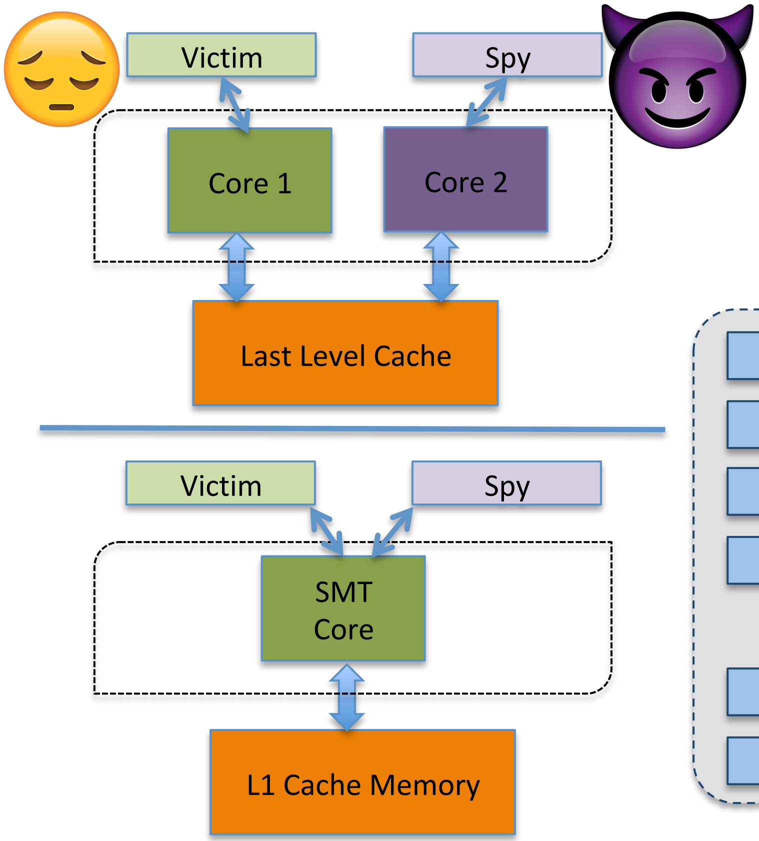
Countermeasures

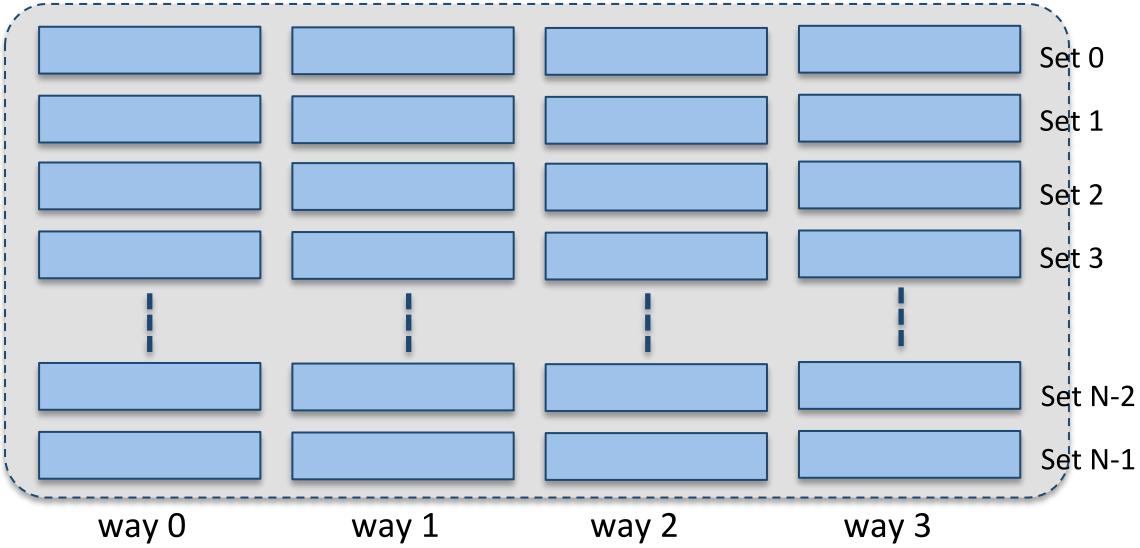
- Do not use copy-on-write
  - Implemented by cloud providers
- Permission checks for clflush
  - Do we need clflush?
- Non-inclusive cache memories
  - AMD
  - Intel i9 versions
- Fuzzing Clocks
- Software Diversification
  - Permute location of objects in memory (statically and dynamically)

Cache Collision Attacks

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

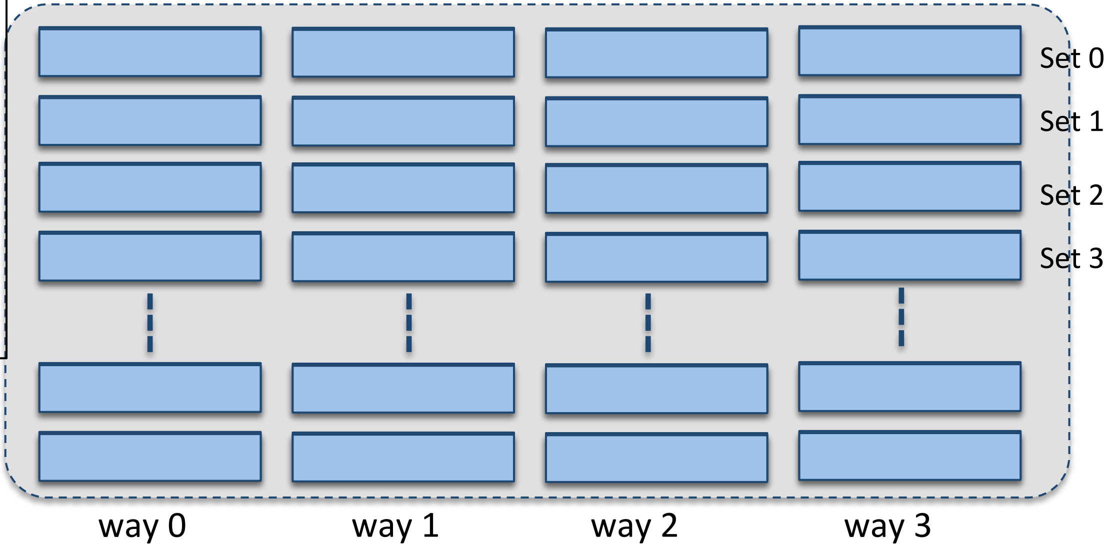
Prime + Probe Attack





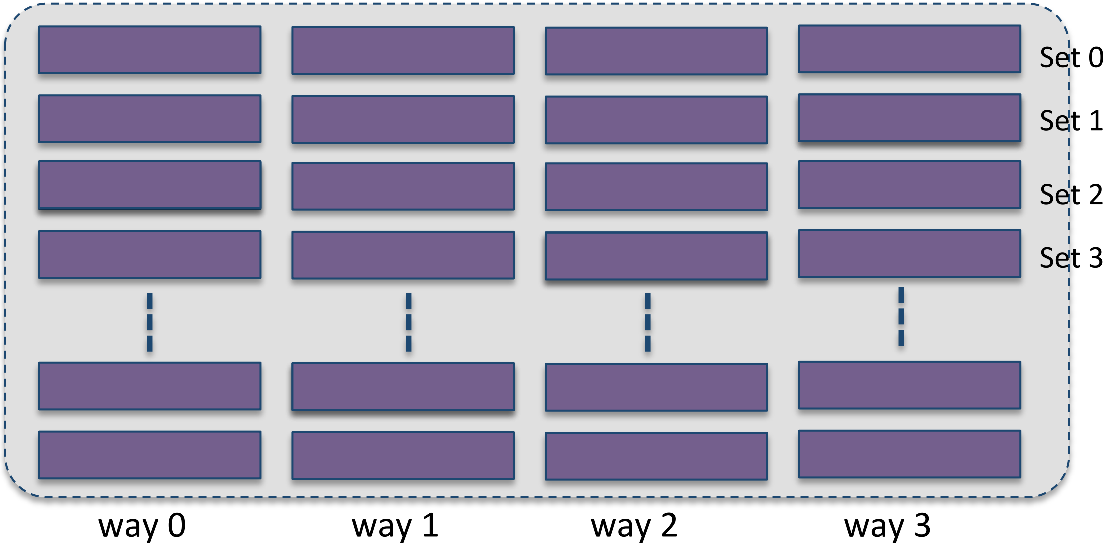
Prime Phase

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



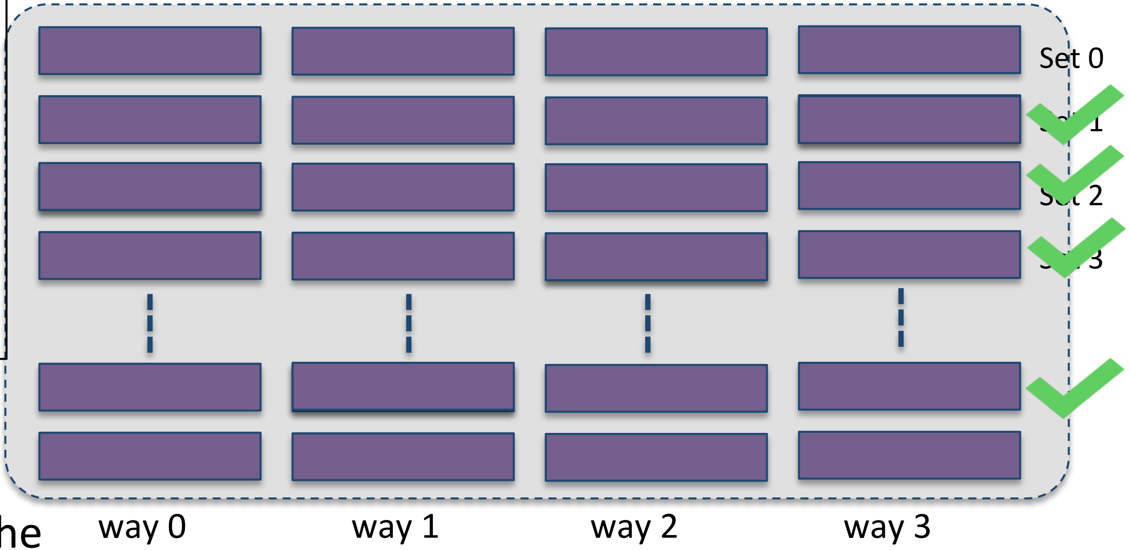
Victim Execution

The execution causes some of the spy data to get evicted



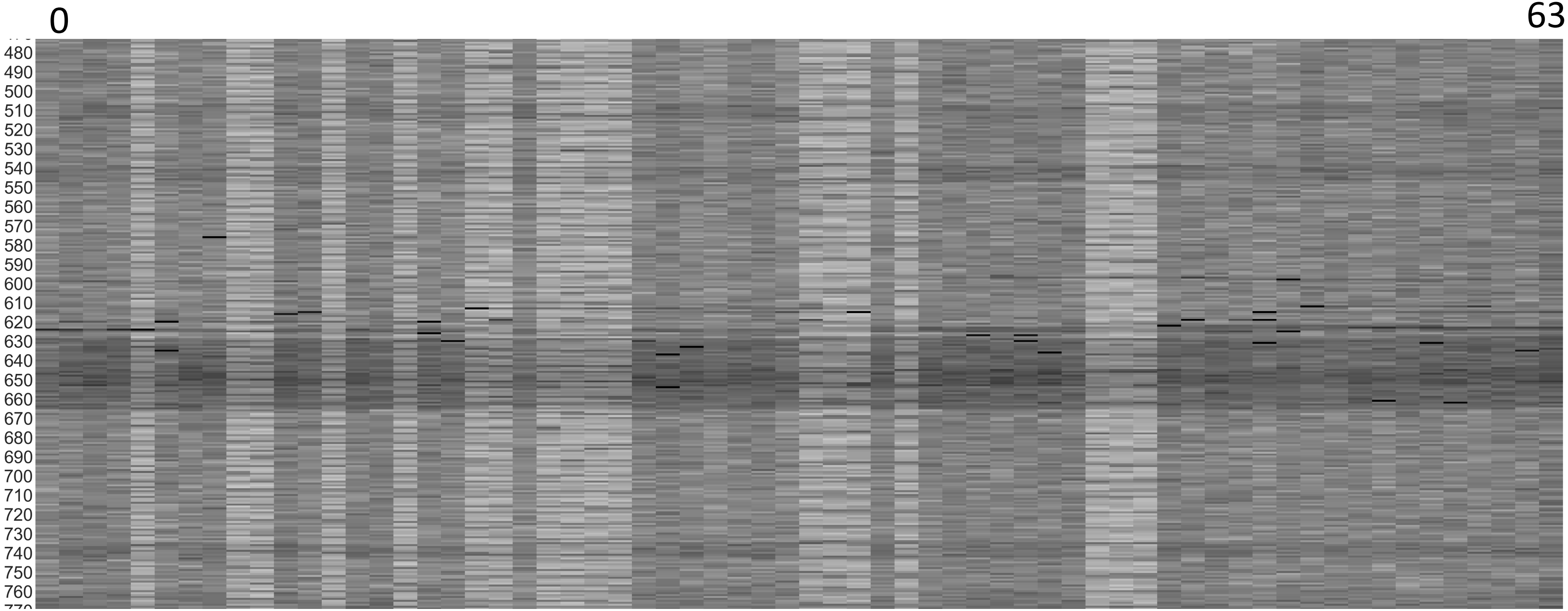
Probe Phase

```
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

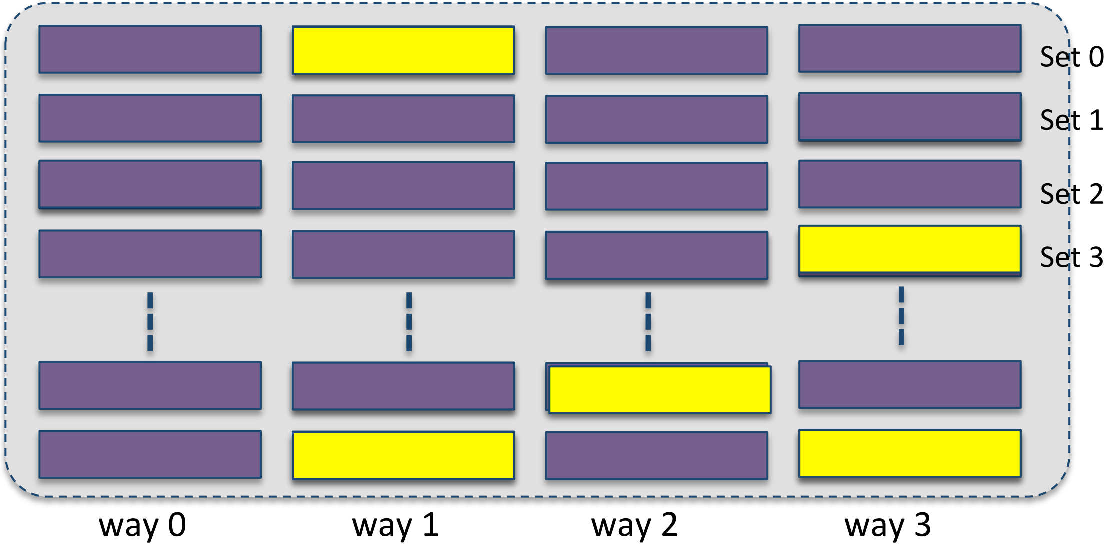
```
char Lookup[] = {x, x, x, . . . x};

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.

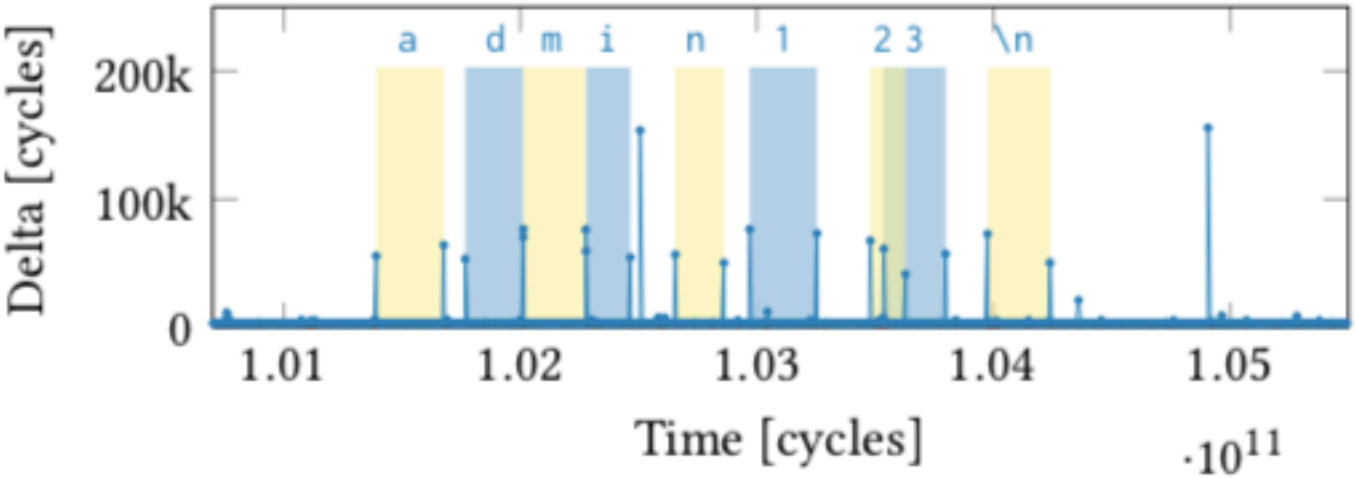
Keystroke Sniffing

- Keystroke → 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

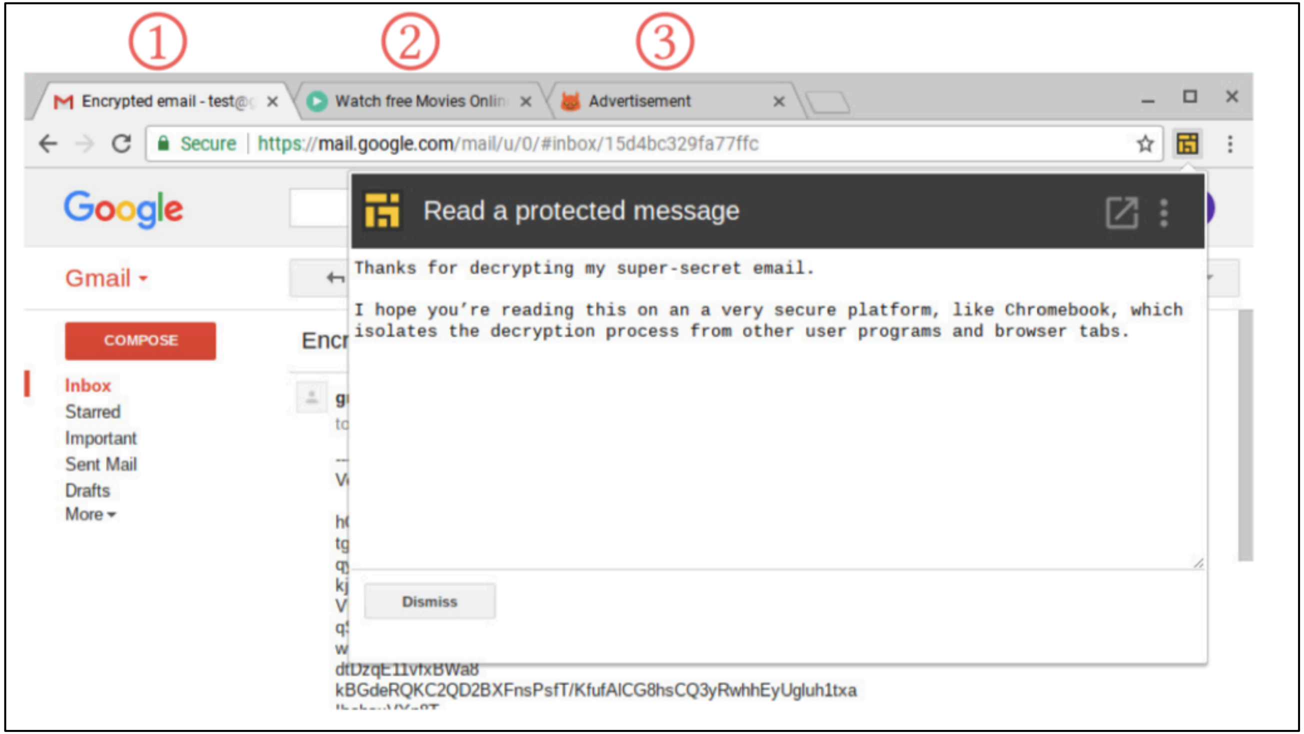


Svetlana Pinet, Johannes C. Ziegler, and F.-Xavier Alario. 2016. Typing Is Writing: Linguistic Properties Modulate Typing Execution. Psychon Bull Rev 23, 6

Web Browser Attacks

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

Extract Gmail secret key

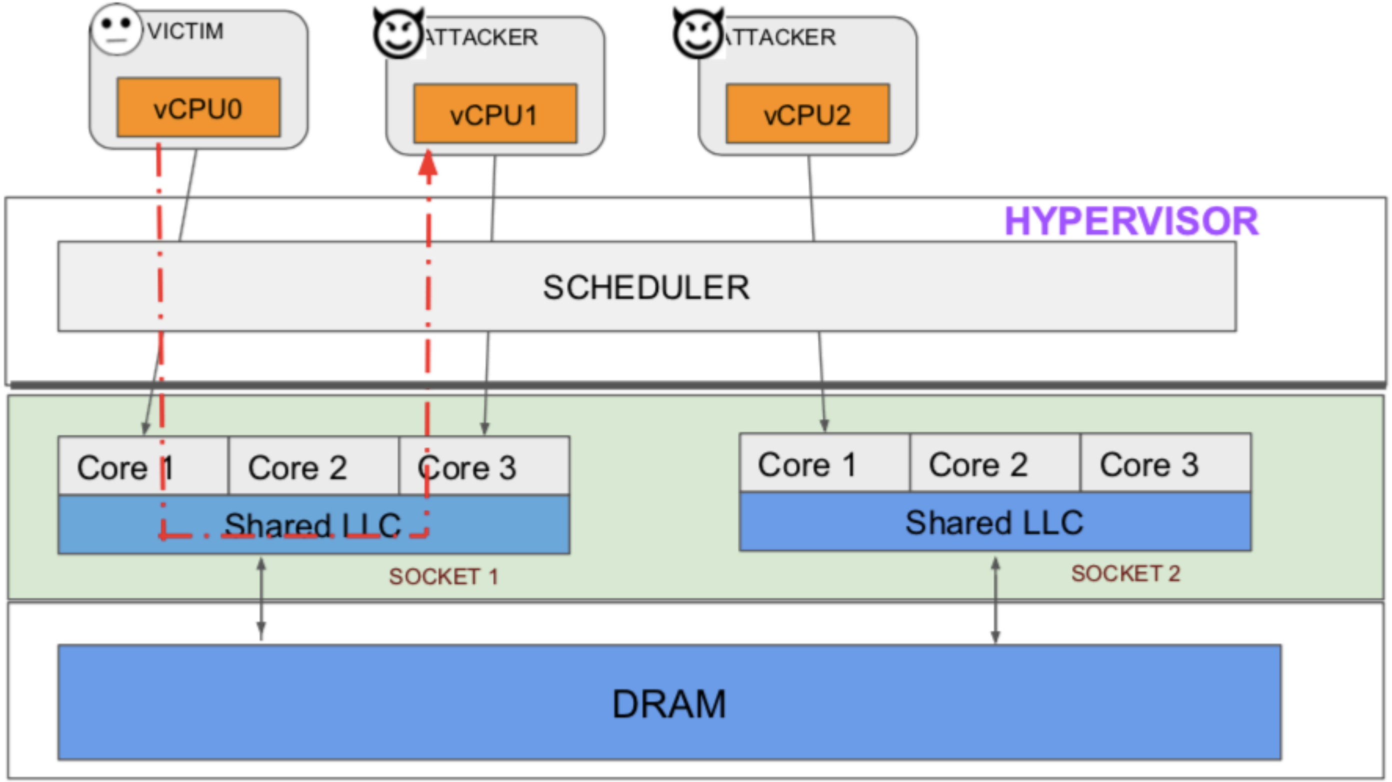


https://www.cs.tau.ac.il/~tromer/drivebycache/drivebycache.pdf

Website Fingerprinting

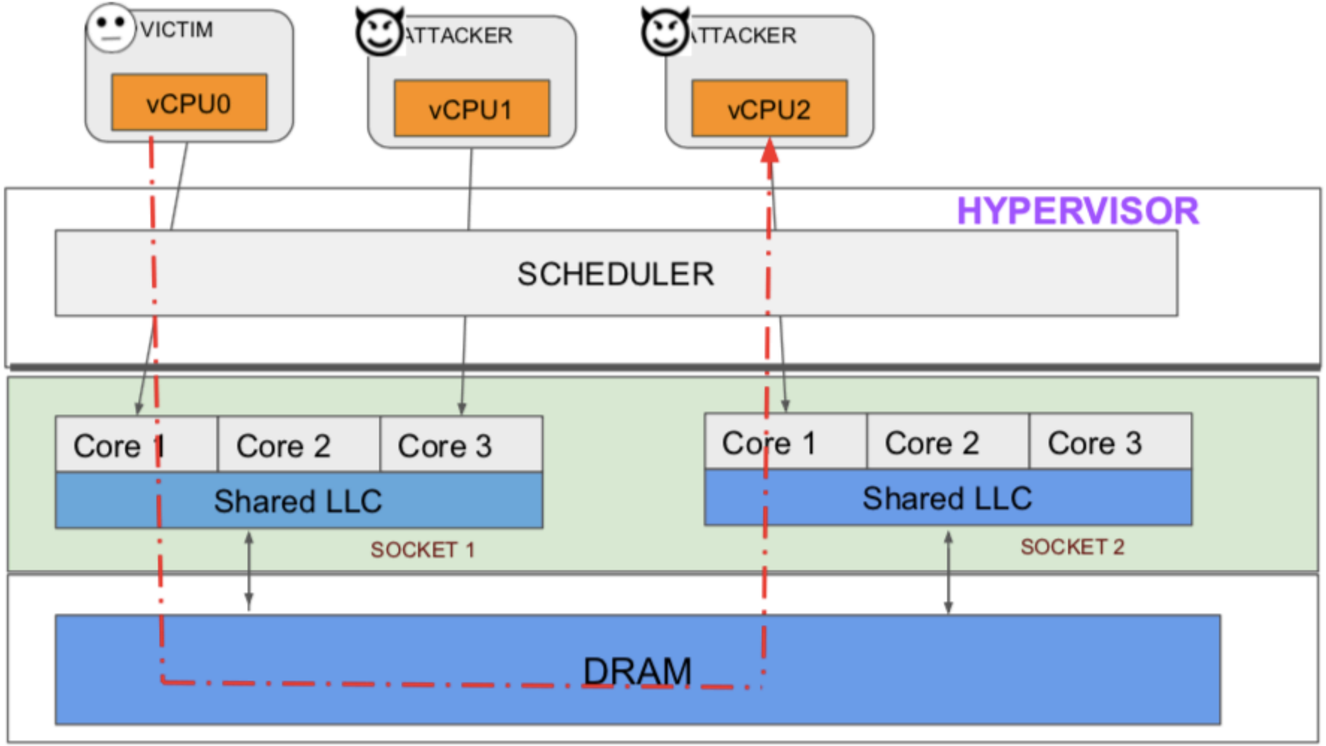
- Privacy: Find out what websites are being browsed.

Cross VM Attacks (Cache)

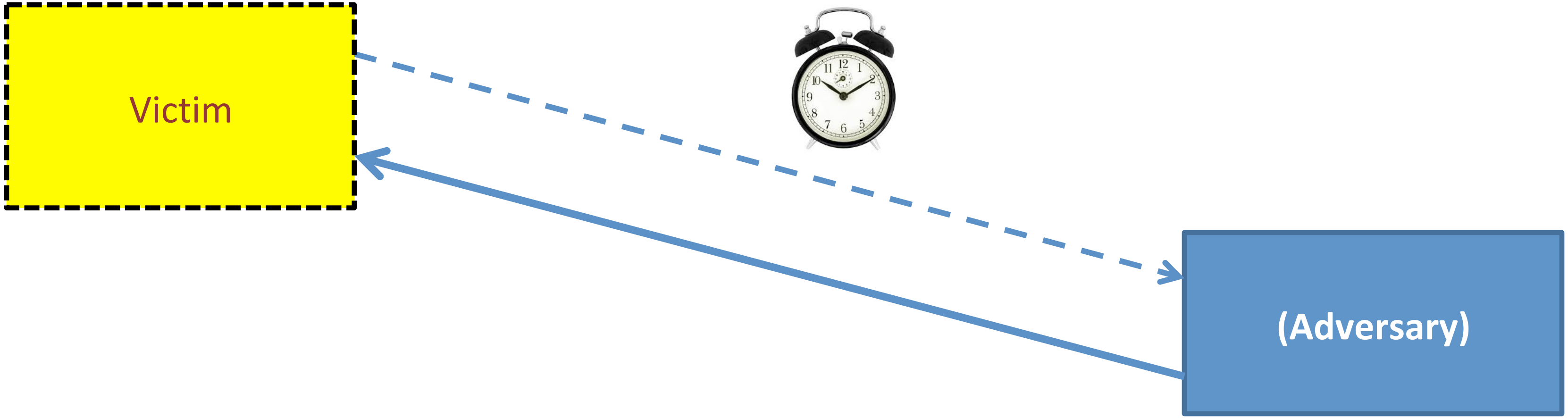


*Ristenpart et.al., *Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds*, CCS- 2009

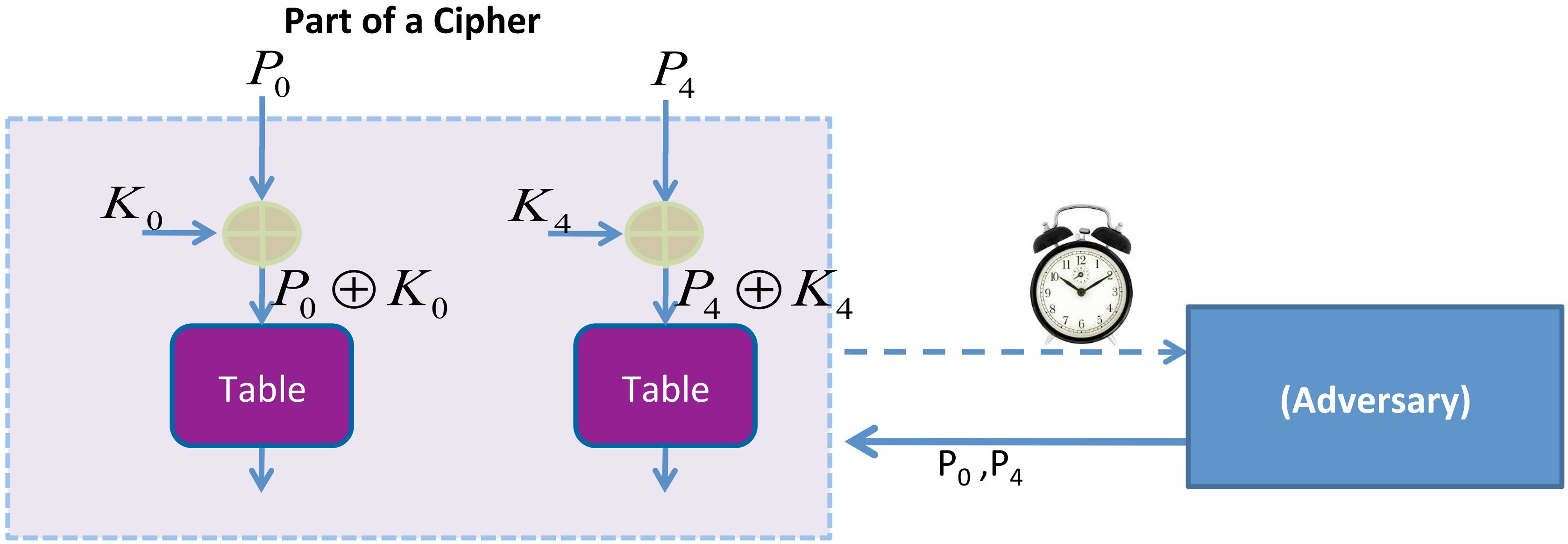
Cross VM Attacks (DRAM)



Internal Collision Attacks



Internal Collisions on a Cipher



If cache hit (less time) :

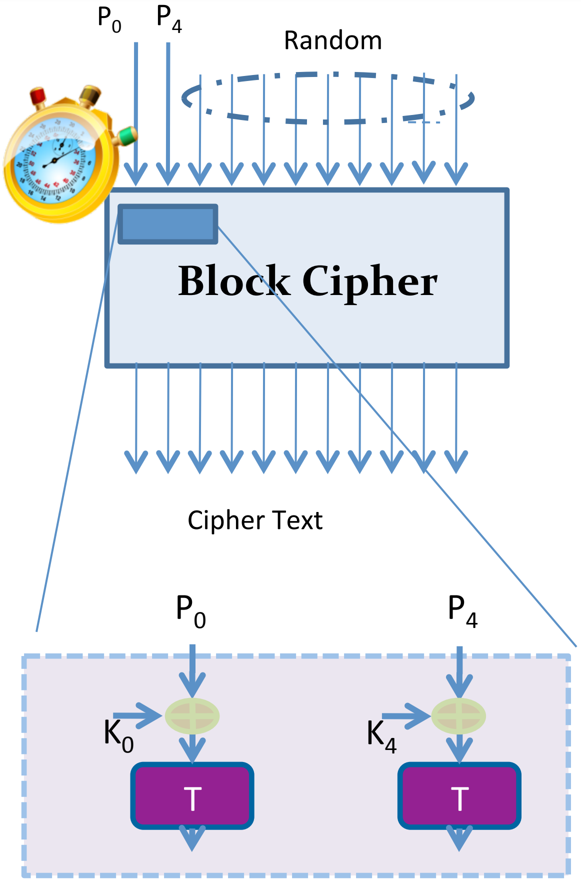
$$\langle P_0 \oplus K_0 \rangle = \langle P_4 \oplus K_4 \rangle$$
$$\Rightarrow \langle K_0 \oplus K_4 \rangle = \langle P_0 \oplus P_4 \rangle$$

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)

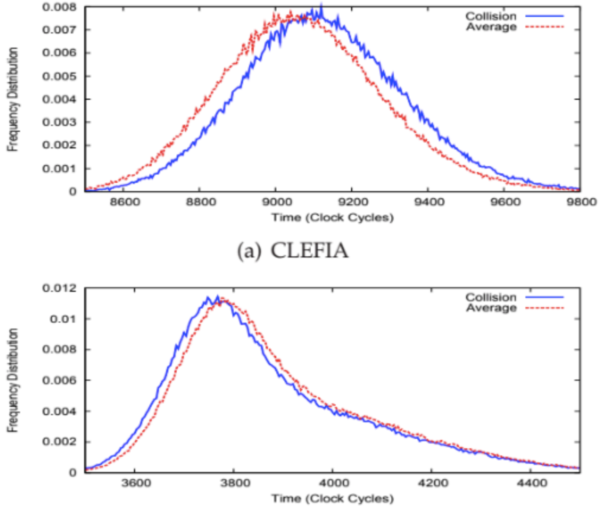


$$\langle K_0 \oplus K_4 \rangle = \langle P_0 \oplus P_4 \rangle$$

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
:	:	:
Average : 2943.57 Maximum : -6.3		-1.7

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



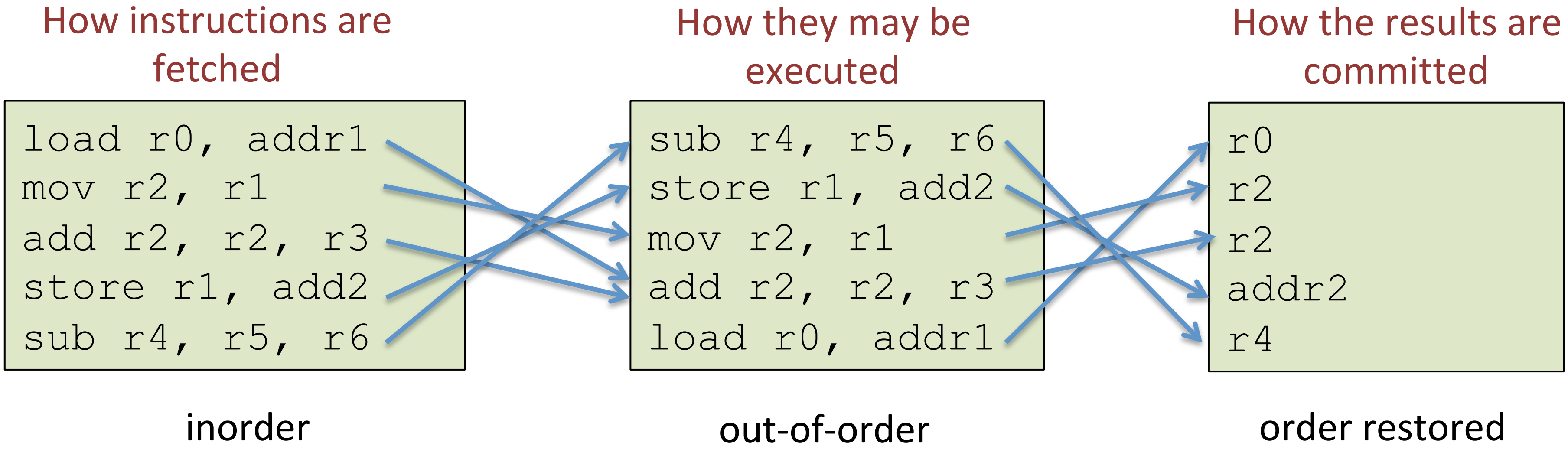
(a) CLEFIA

(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



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
```

How instructions are fetched

```
  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 (transient instructions)

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

```
  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

Speculative execution (transient instructions)

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

```
  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

Speculative execution

Speculated results discarded

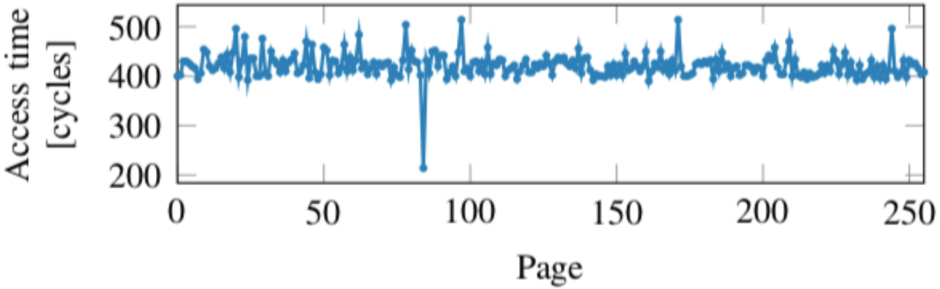
```
   :
   :
   :
```

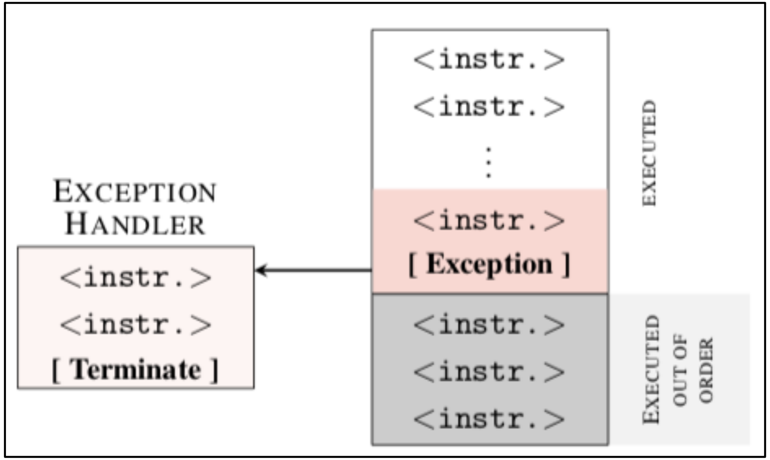
How results are committed when speculation is **incorrect** (eg. If r1 = 0)

Speculative Execution and Micro-architectural State

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

data=84



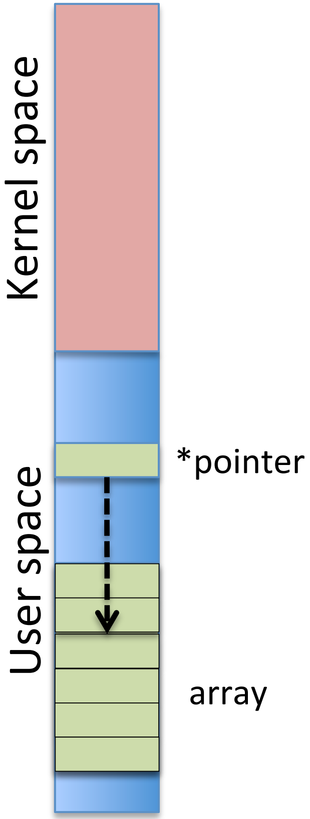


Even though line 3 is not reached, the micro-architectural state is modified due to Line 3.

Meltdown

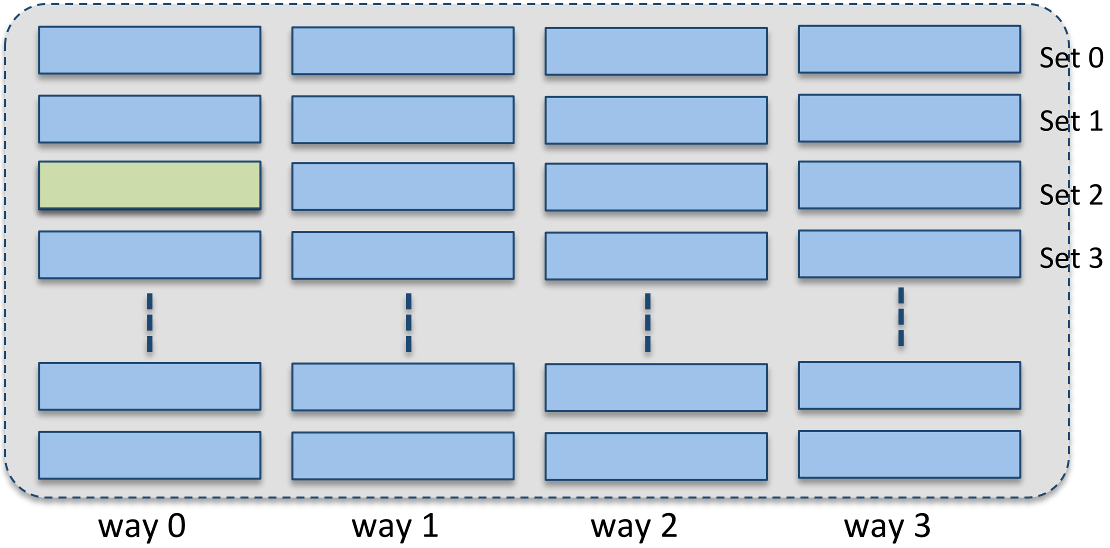
Normal Circumstances

Virtual address space of process




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

Cache Memory

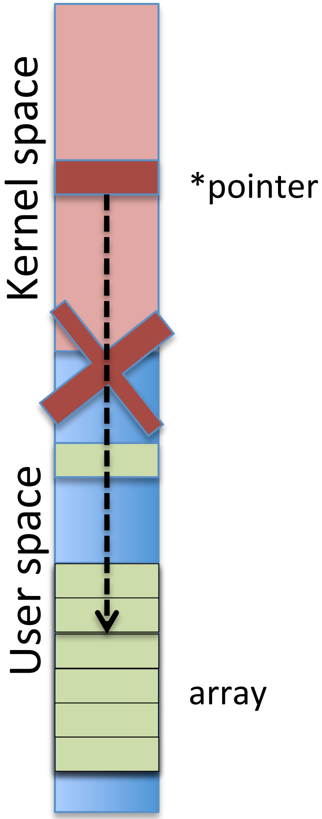


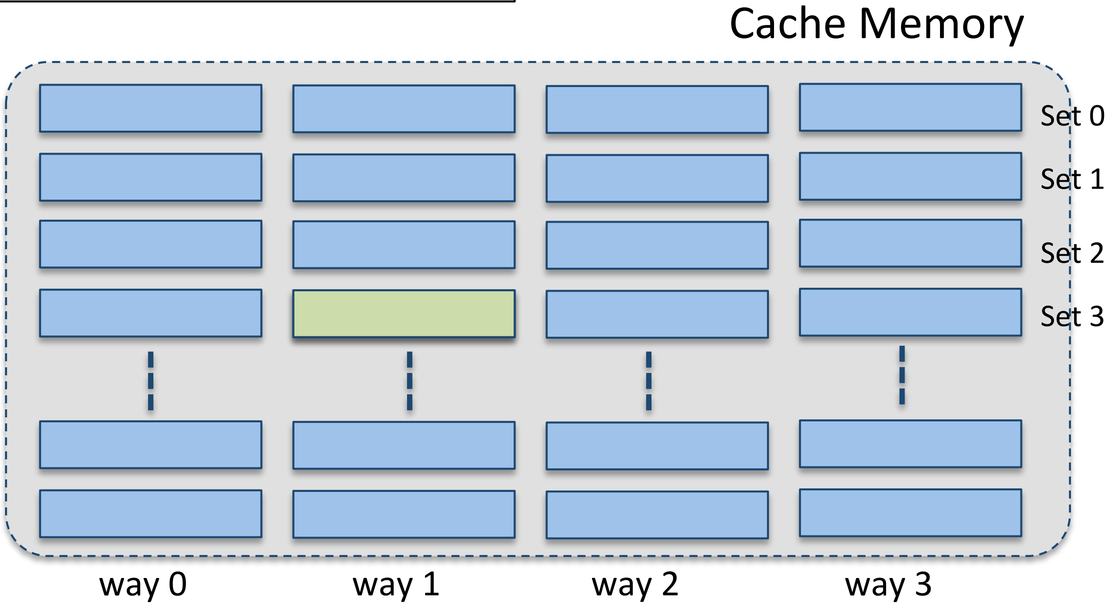
Meltdown

Not normal Circumstances

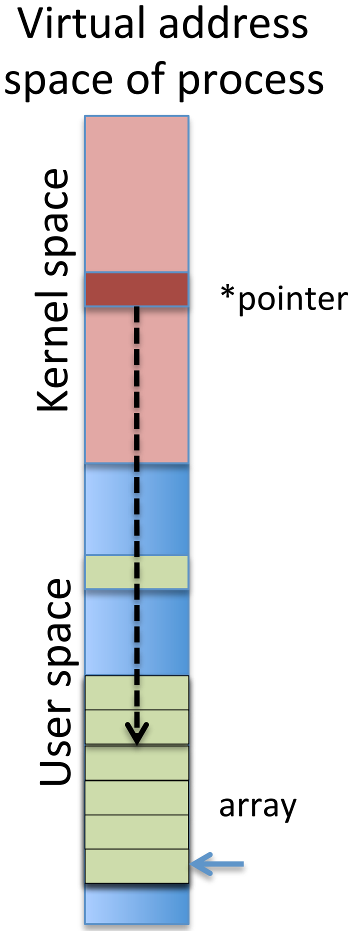
```
i = *pointer
y = array[i * 256]
```

Virtual address space of process



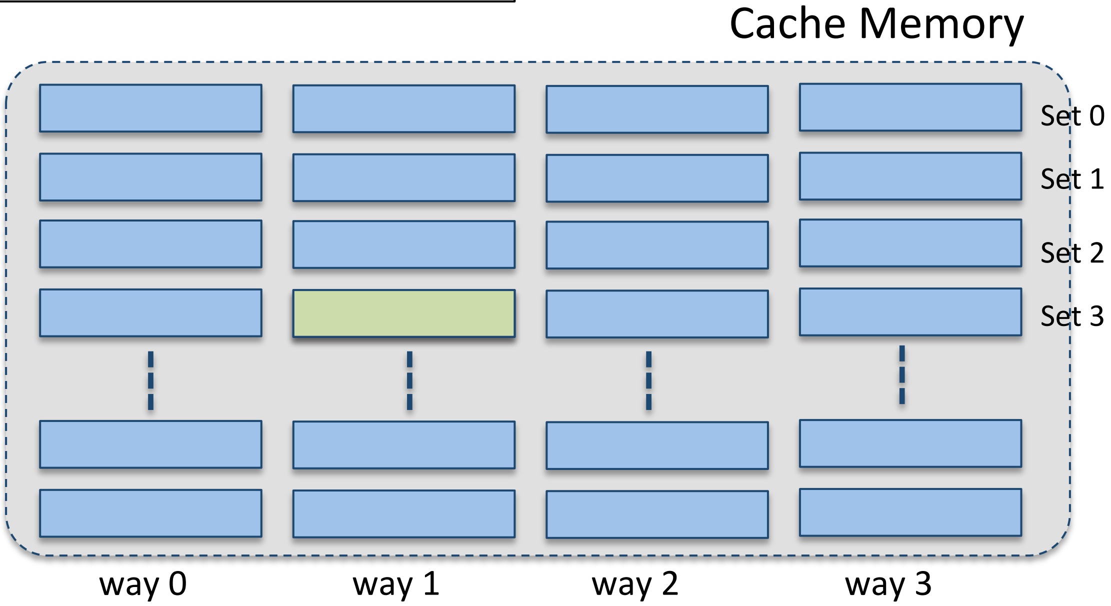


Meltdown



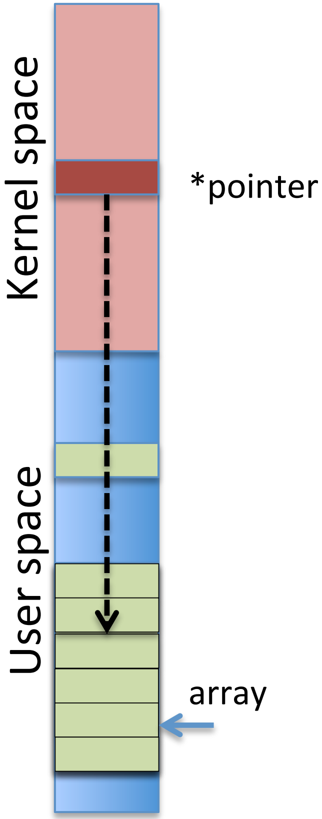
Not normal Circumstances

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



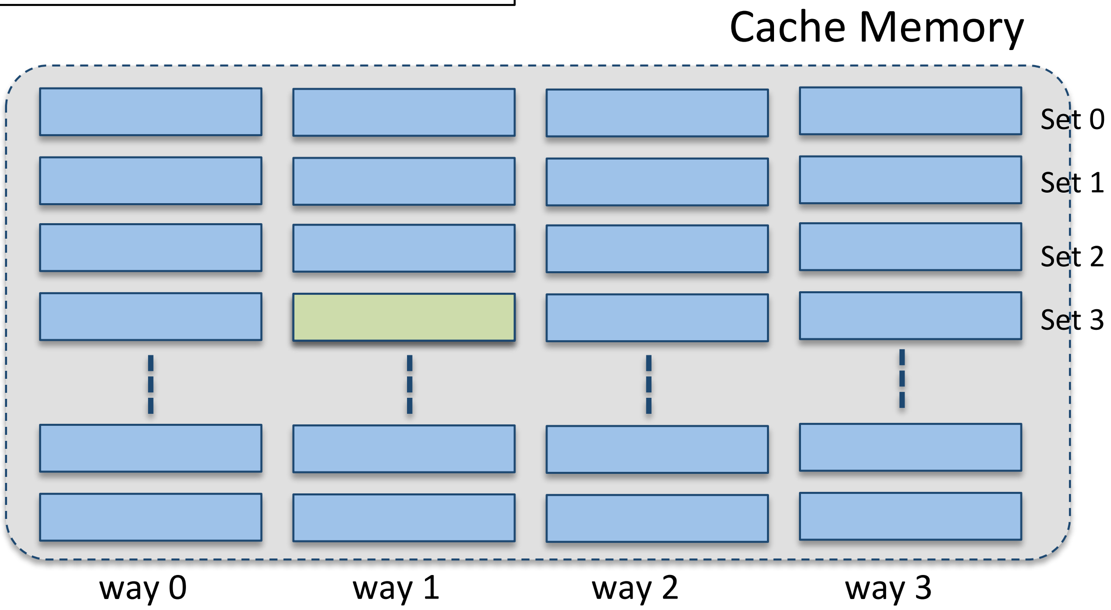
cache miss

Meltdown



Not normal Circumstances

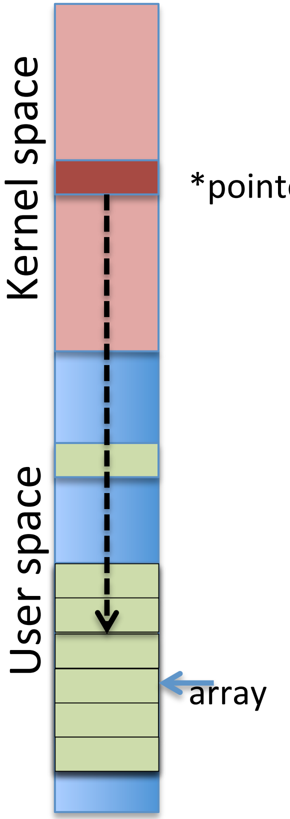
```
i = *pointer
y = array[i * 256]
```



cache miss

Meltdown

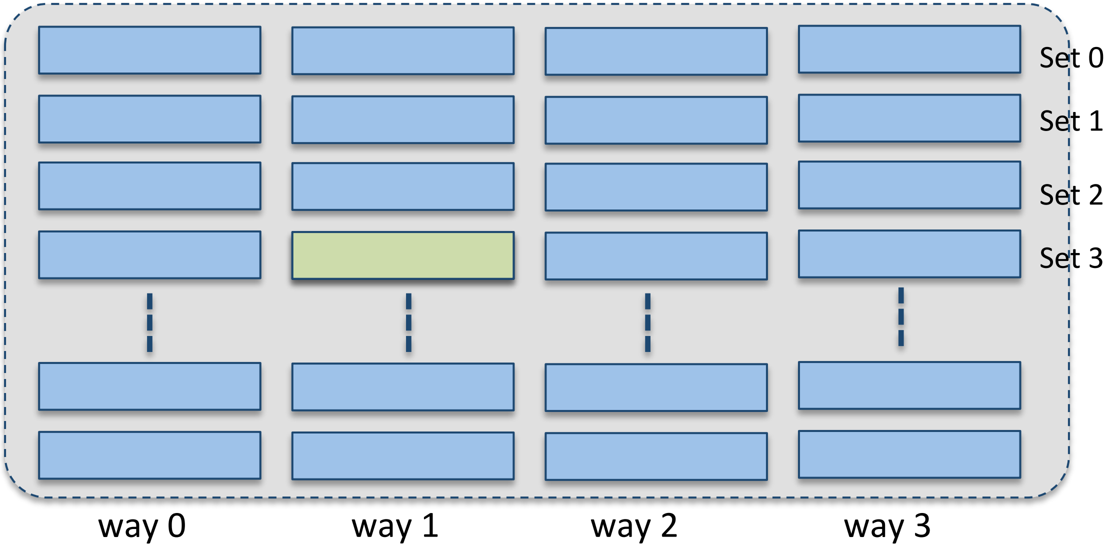
Virtual address space of process



Not normal Circumstances

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

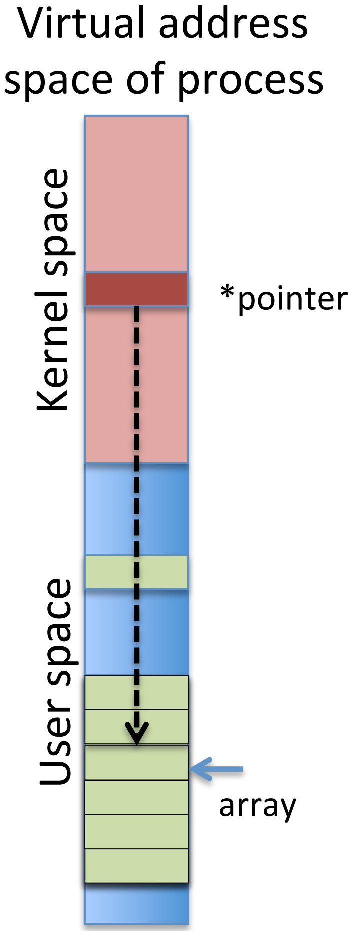
Cache Memory



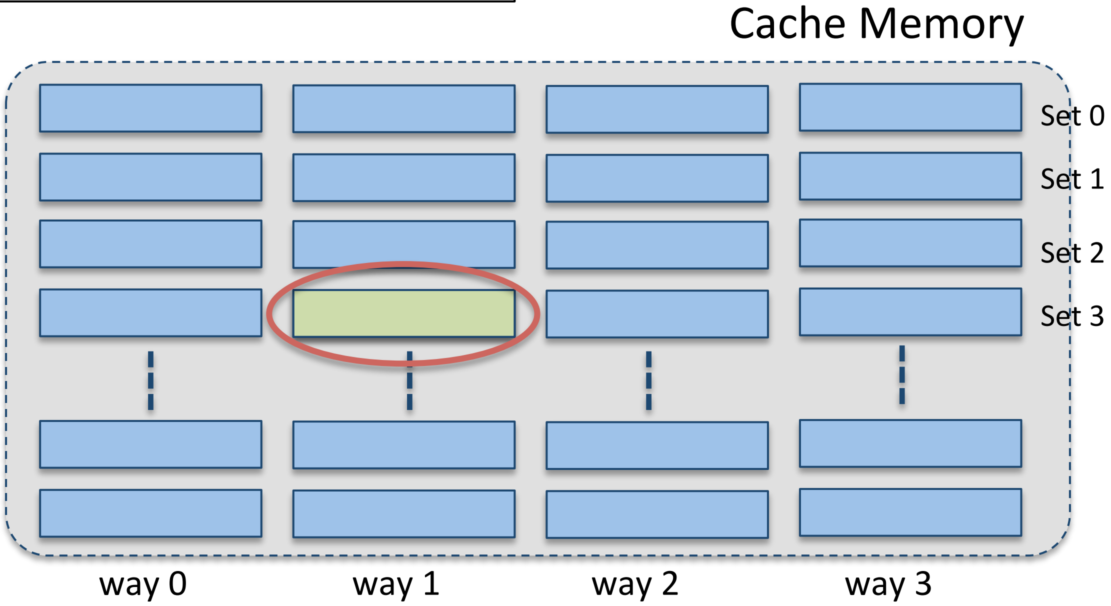
cache miss

Meltdown

Not normal Circumstances




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

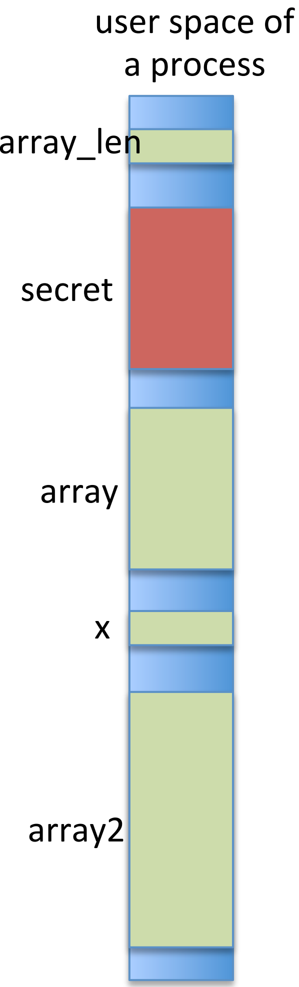


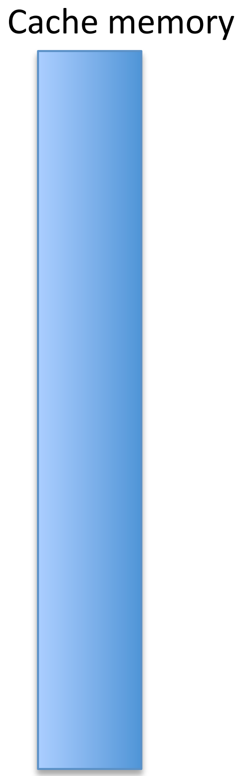
cache hit

Spectre

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

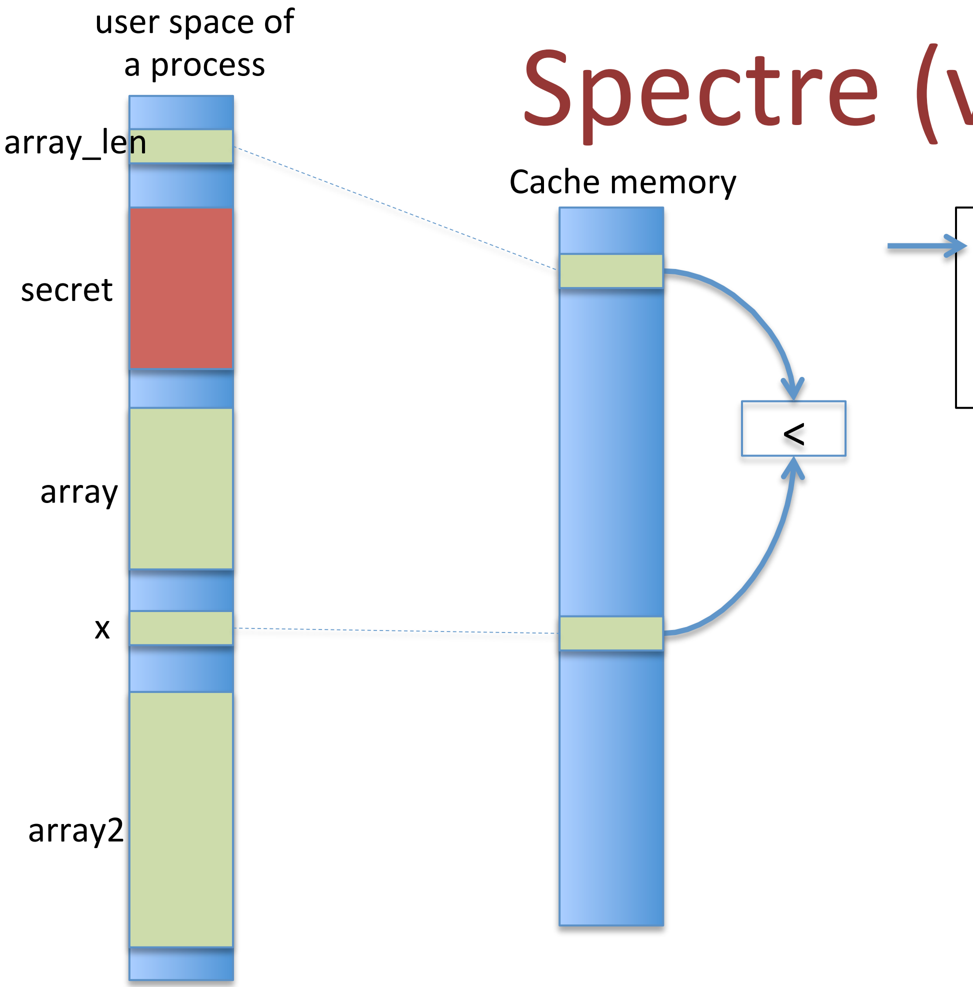
Spectre (variant 1)






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

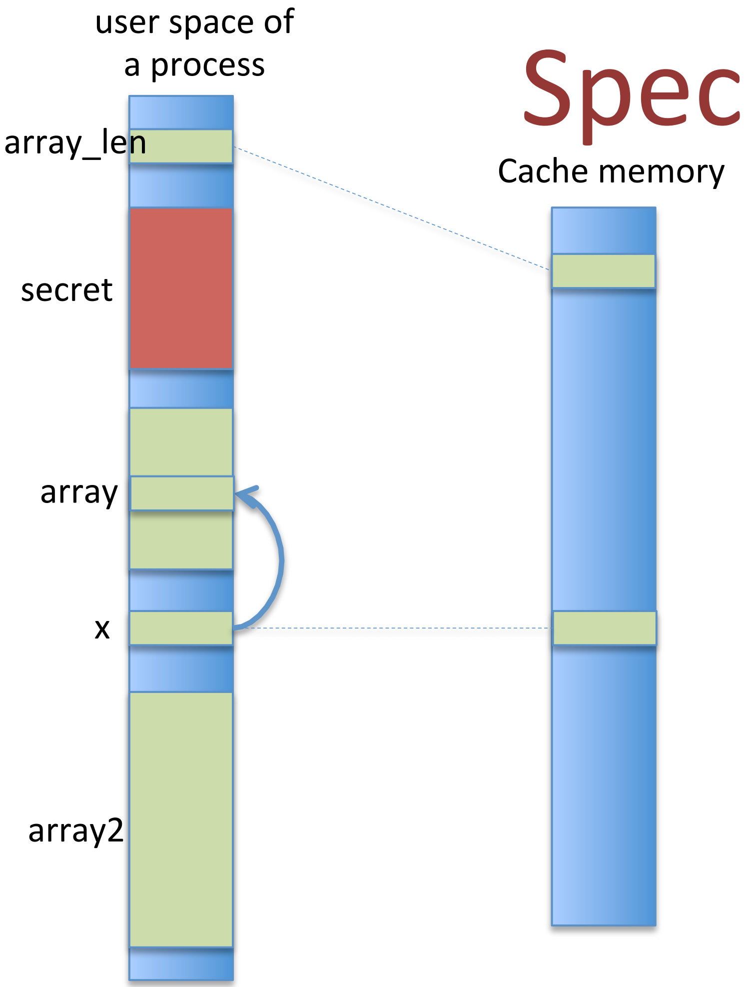
Spectre (variant 1)




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

Spectre (variant 1)

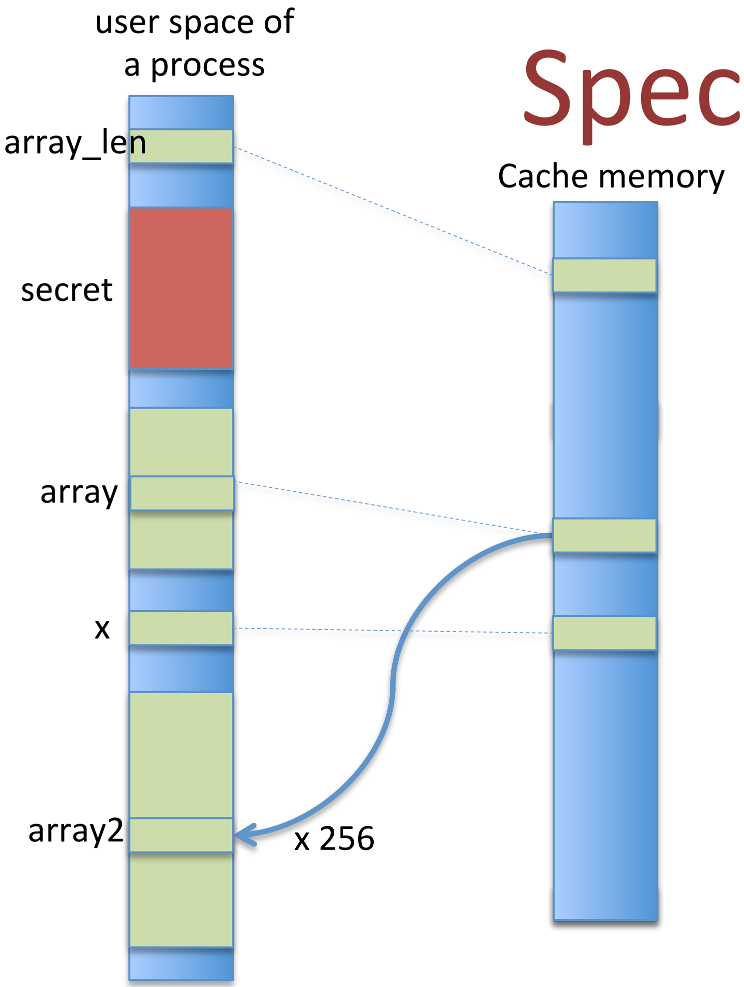
Normal Behavior




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

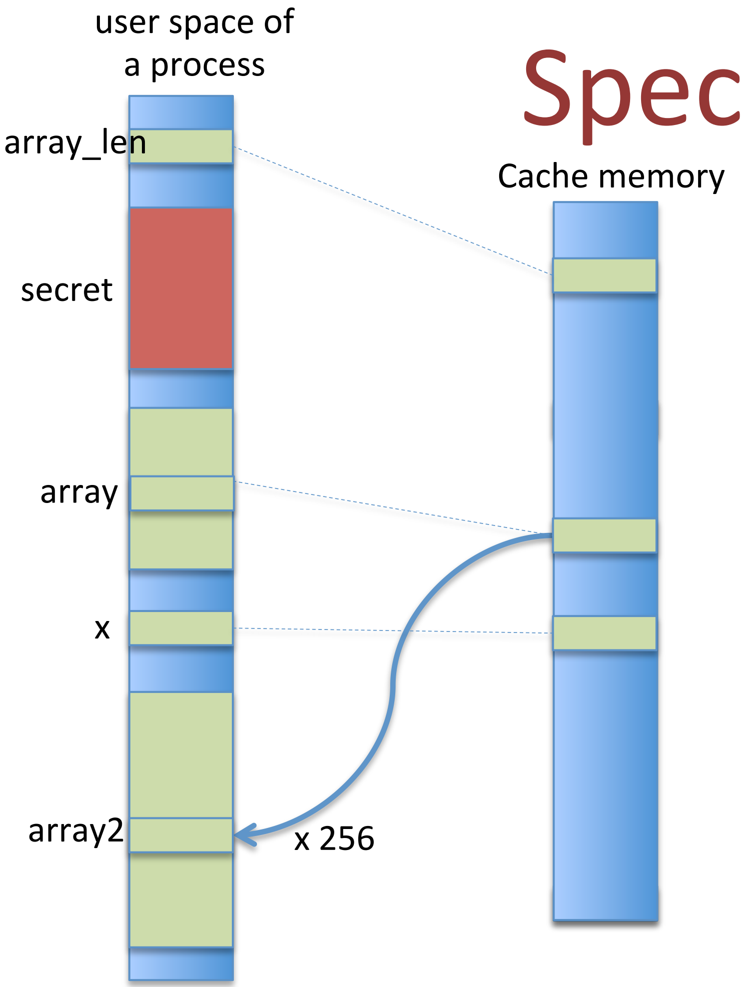
Spectre (variant 1)

Normal Behavior




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

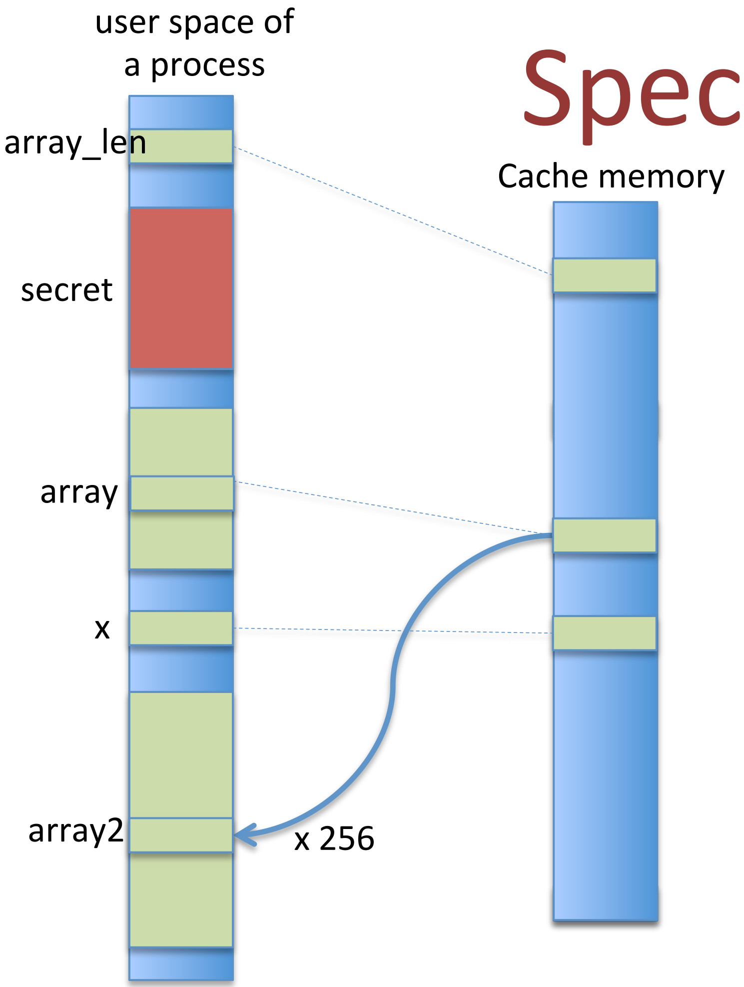
Spectre (variant 1)



Normal Behavior

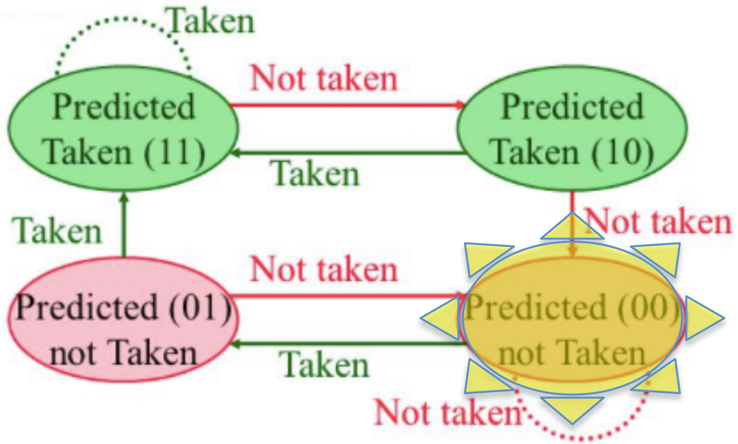
```
if (x < array_len){
    i = array[x];
    y = array2[i * 256];
}
```

Spectre (variant 1)

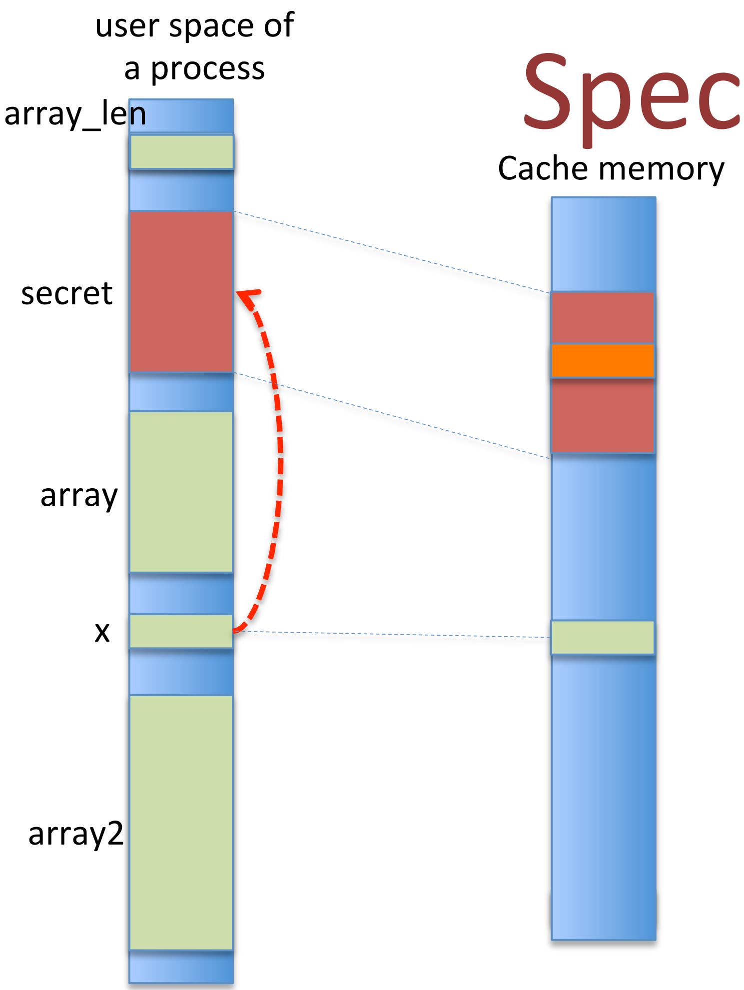


Normal Behavior

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



Spectre (variant 1)



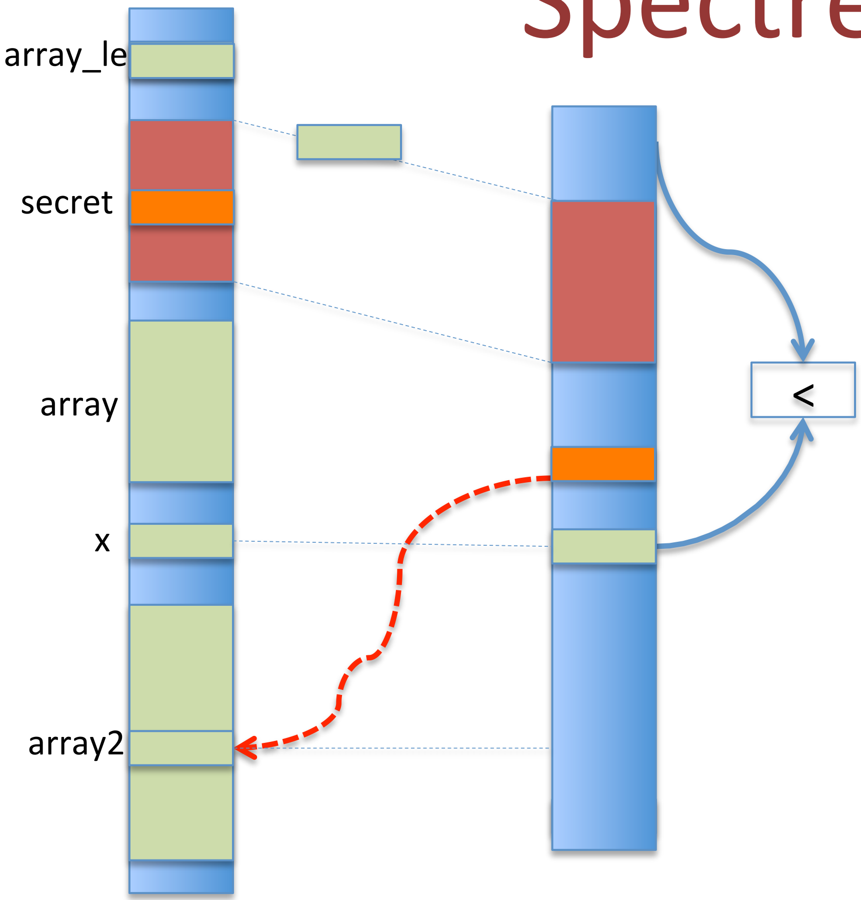
Under Attack

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

- x > array_len
- array_len not in cache
- secret in cache memory

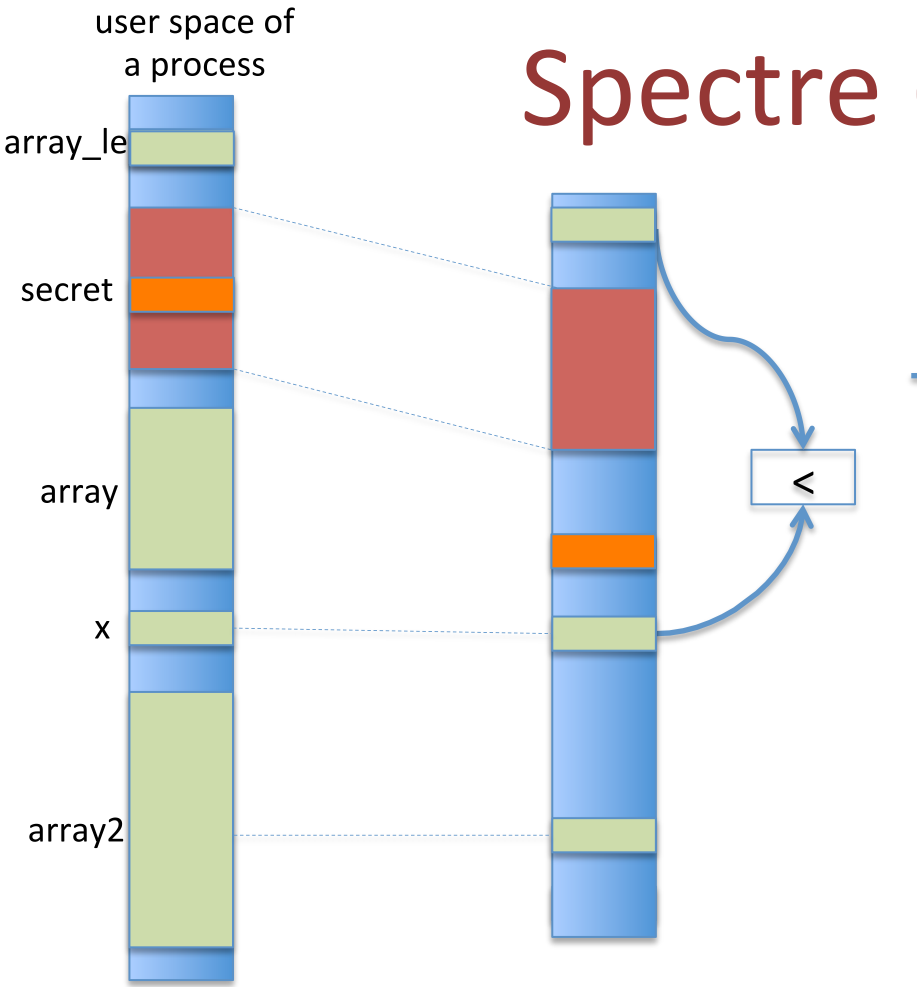
Spectre (variant 1)

user space of
a process




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

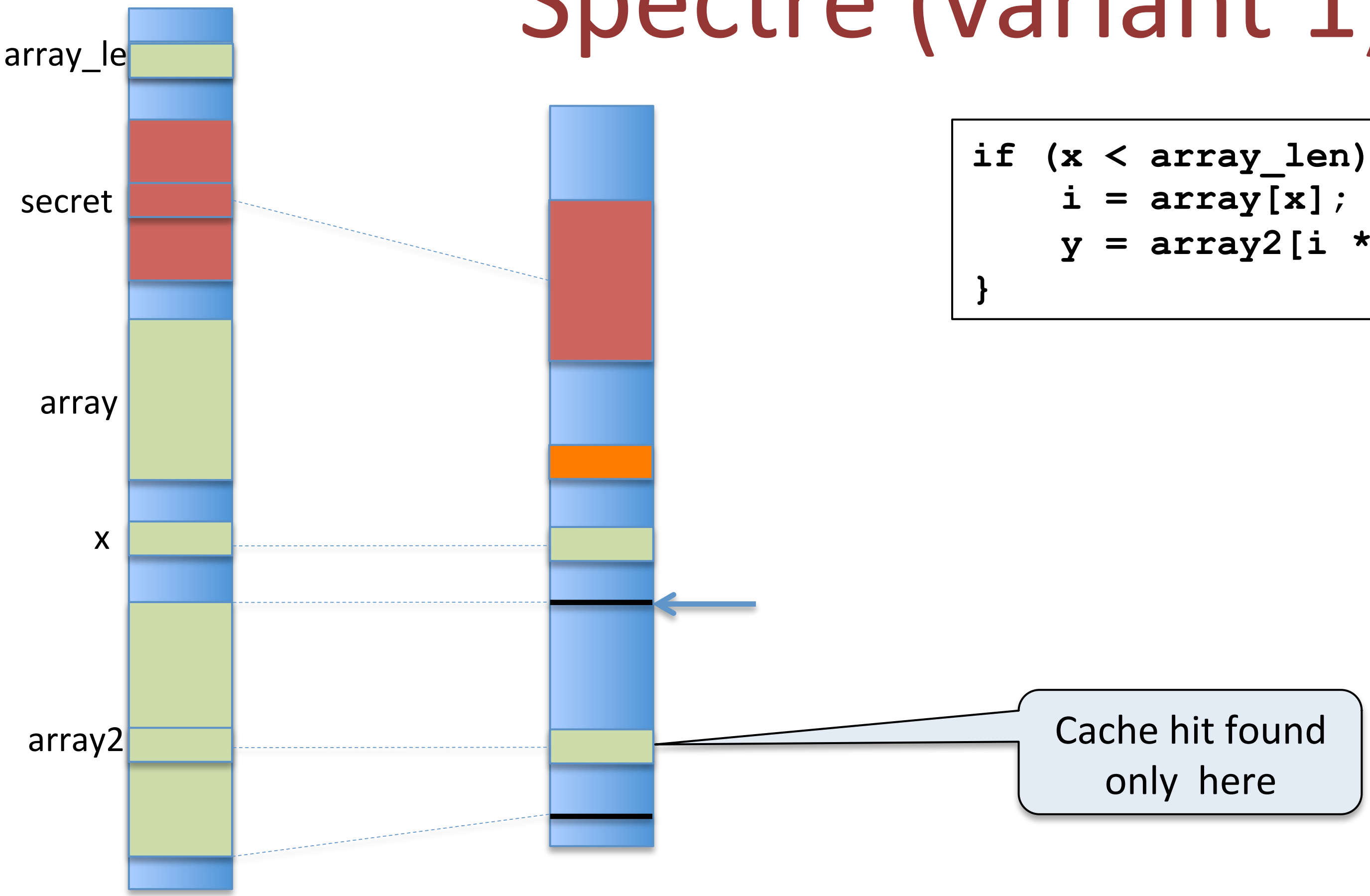
Spectre (variant 1)



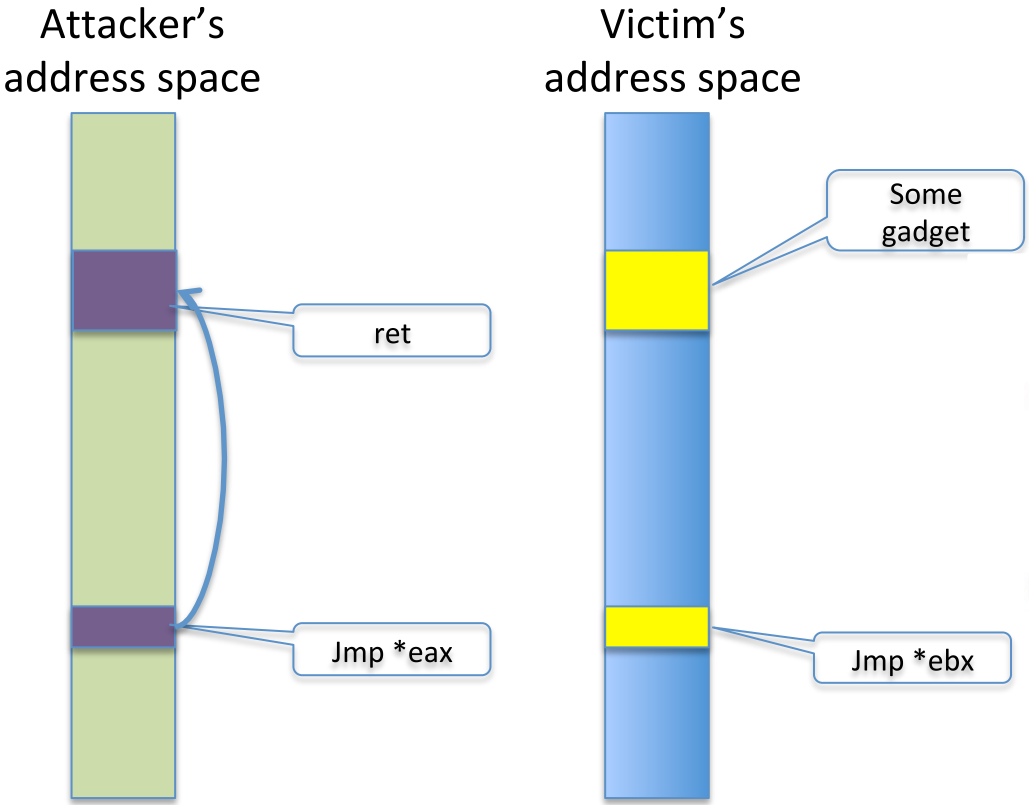

```
if (x < array_len){
    i = array[x];
    y = array2[i * 256];
}
```

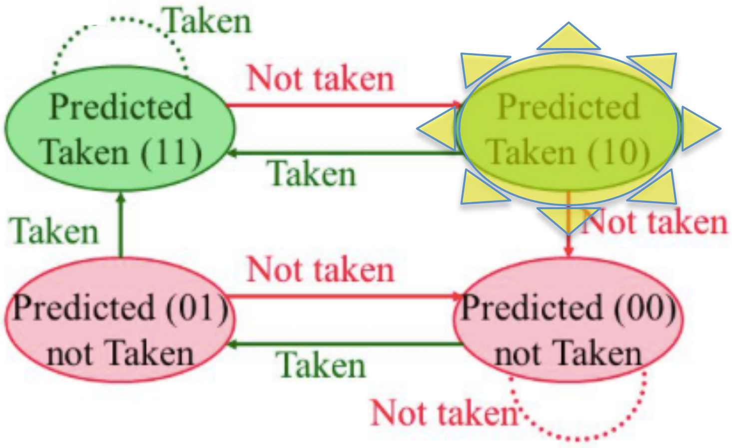
Spectre (variant 1)

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

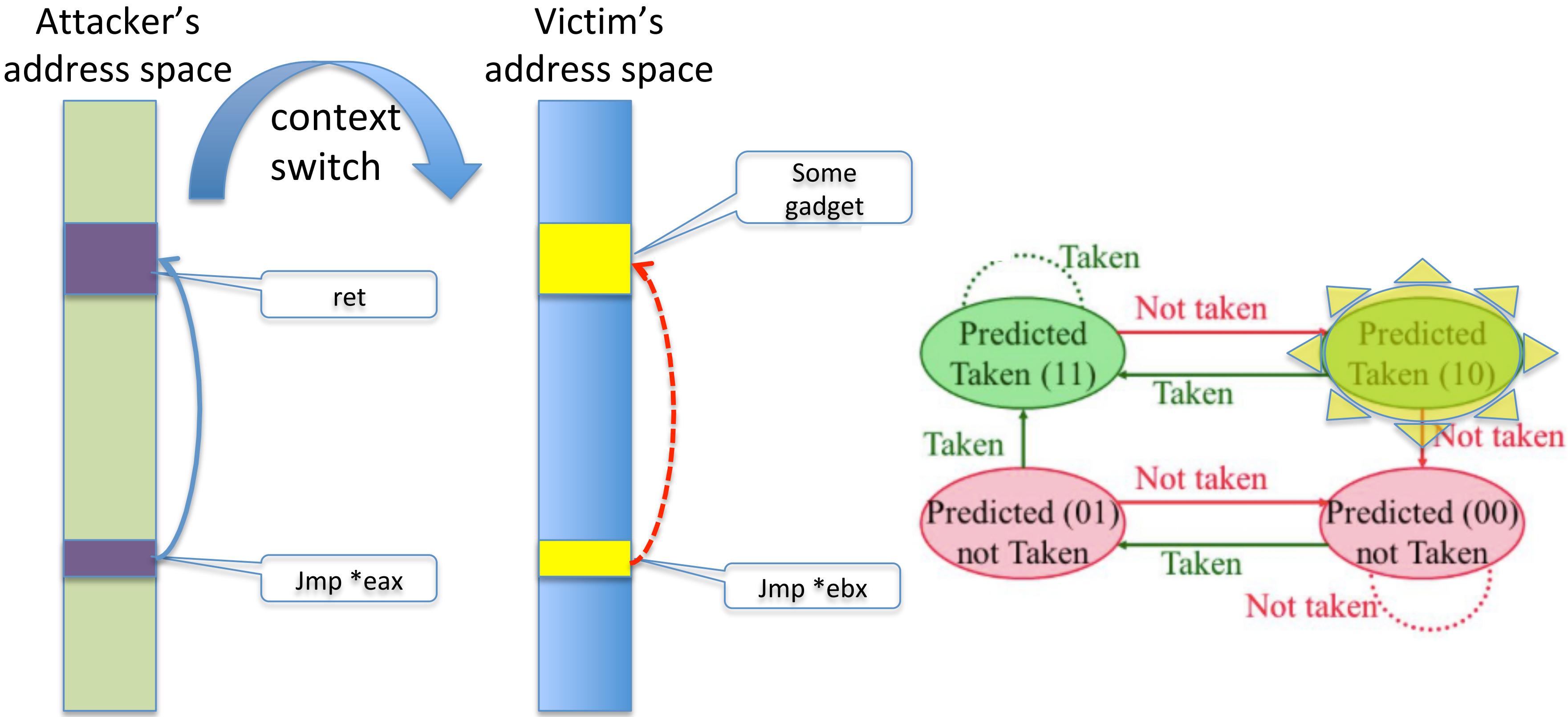


Spectre (variant 2)



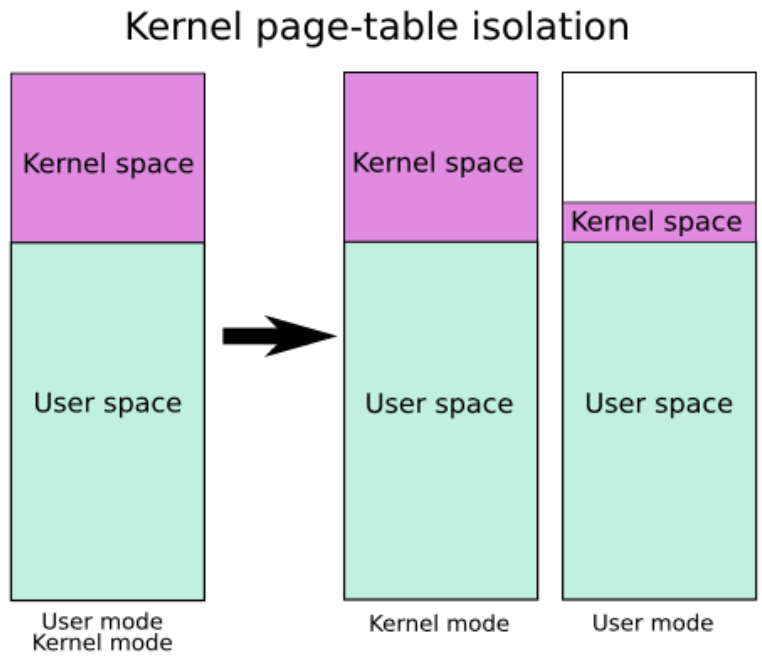


Spectre (variant 2)



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