Micro-architectural Attacks 2

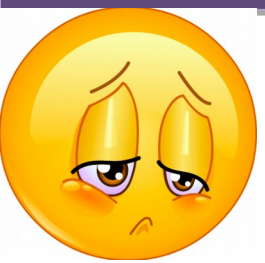
Milan Patnaik

Indian Institute of Technology Madras

Credits: Prof Chester Rebeiro and my colleages of RISE Lab, 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

Speculation Attacks

Branch prediction attack

Row hammer

Fault Injection Attacks

cold boot attacks

DRAM Row buffer (DRAMA)

..... and many more

Meltdown

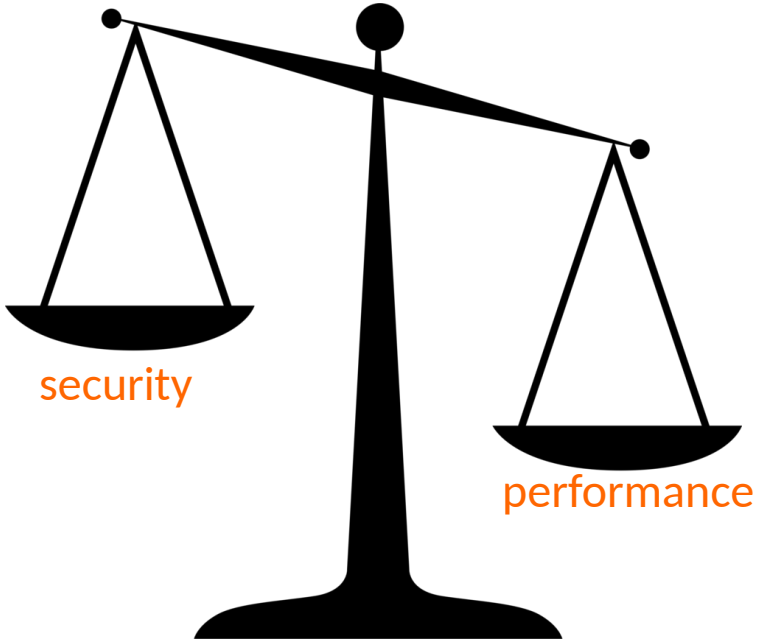
Spectre

Causes

Most micro-architectural attacks caused by performance optimizations

Others due to inherent device properties

Third, due to stronger attackers



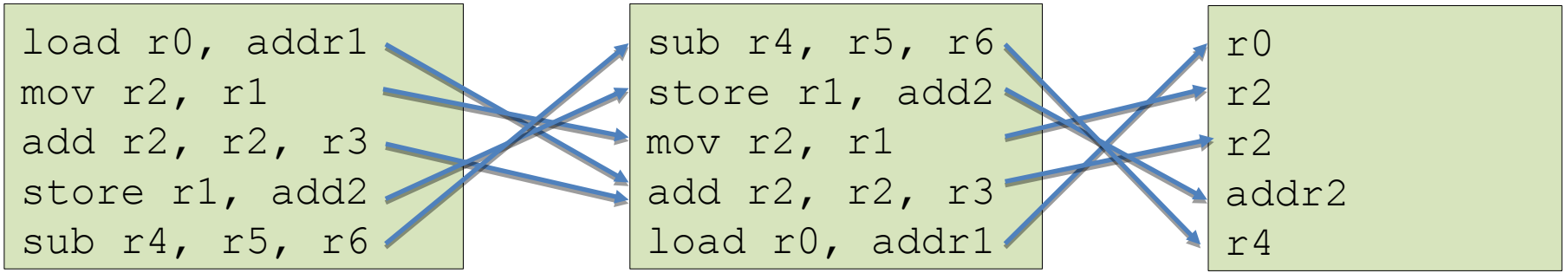
Instruction Level Parallelism

Out-of-order execution

How instructions are fetched

How they may be executed

How the results are committed



inorder

out-of-order

order restored

Out of the processor core, execution looks in-order

Inside the processor core, execution is done out-of-order

Speculative Execution: Case 1

```
   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 : Case 1

```
   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

Speculated results discarded

:
:
:

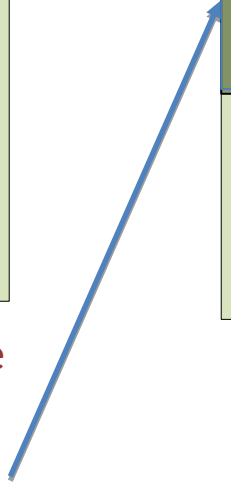
How results are committed when speculation is **incorrect**

Speculative execution
(transient instructions)

Speculative Execution : Case 2

```
   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)

ILP Paradigms in Modern Processors

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the ARM Cortex-A8
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

Speculation Attacks

Meltdown and Spectre

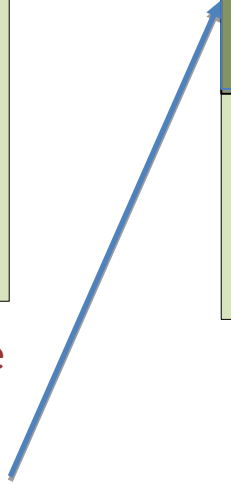
Meltdown

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

Speculative Execution : Case 2

```
   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
```

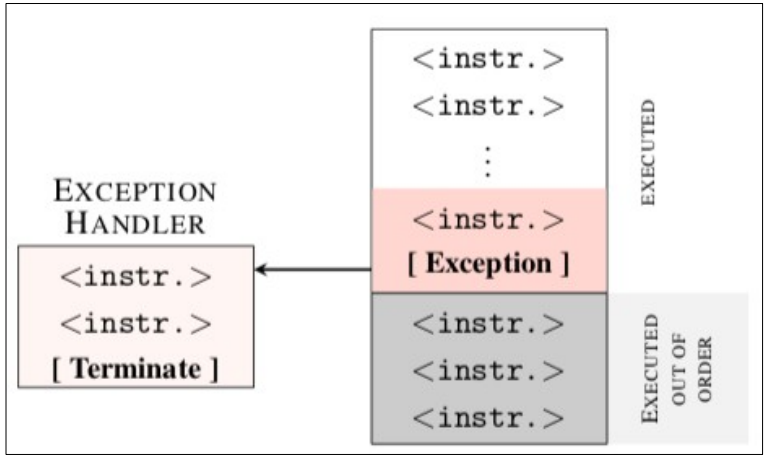
How instructions are executed

Speculated results discarded

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

Speculative Execution and Micro-architectural State

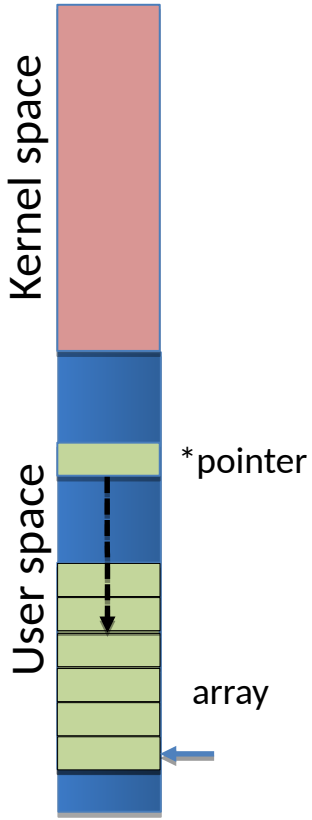
```
1 raise_exception();
2 // the line below is never reached
3 access(probe_array[data * 4096]);
```



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

Meltdown Concept

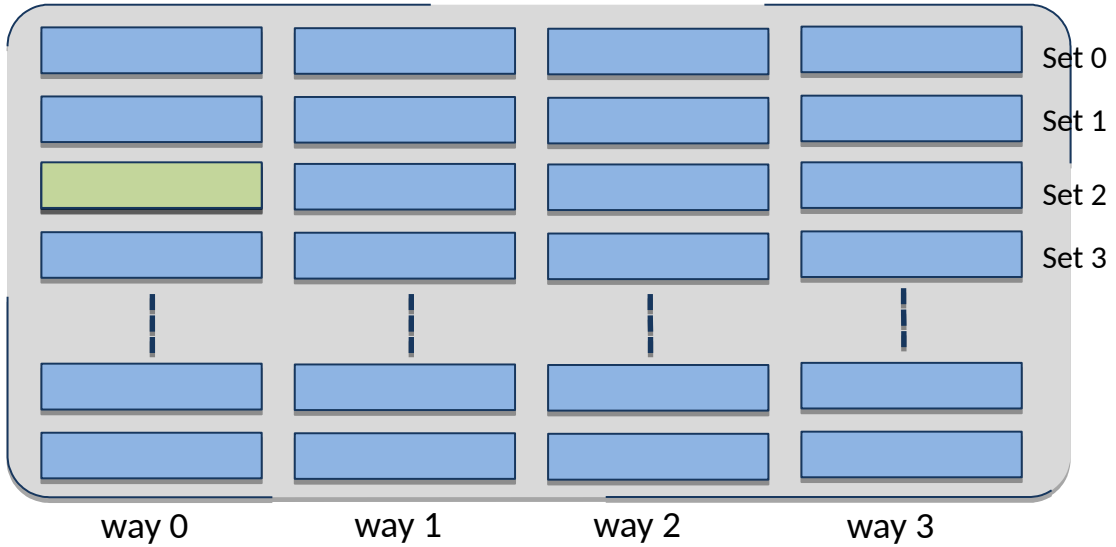
Virtual address space of process



Normal Circumstances

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

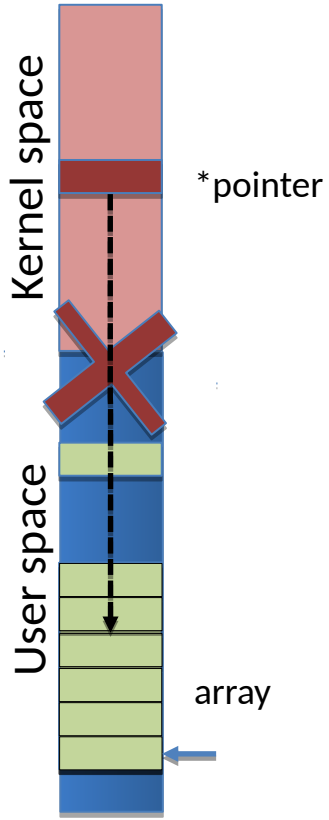
Cache Memory



Meltdown Concept

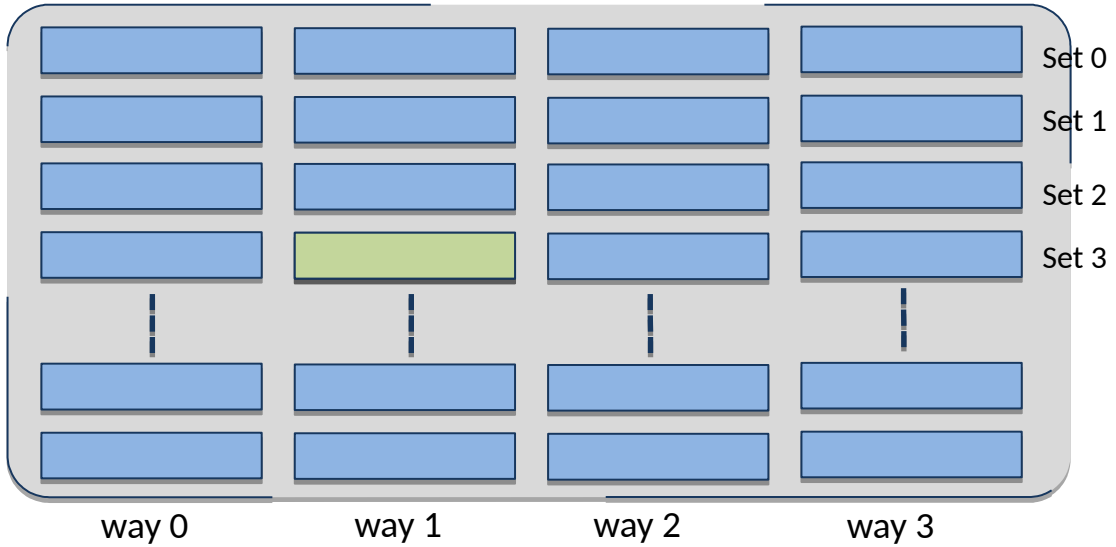
Not normal Circumstances

Virtual address space of process




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

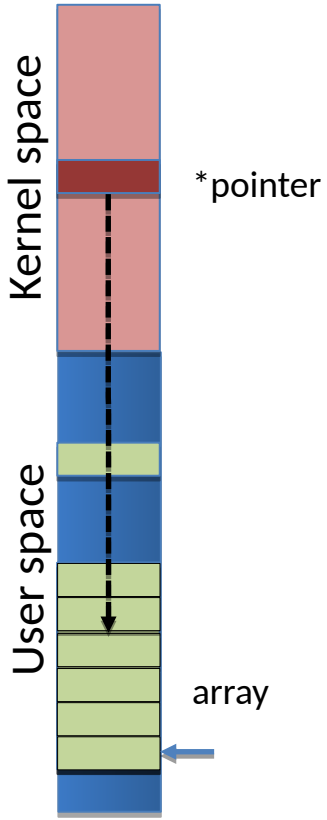
Cache Memory



Meltdown Concept

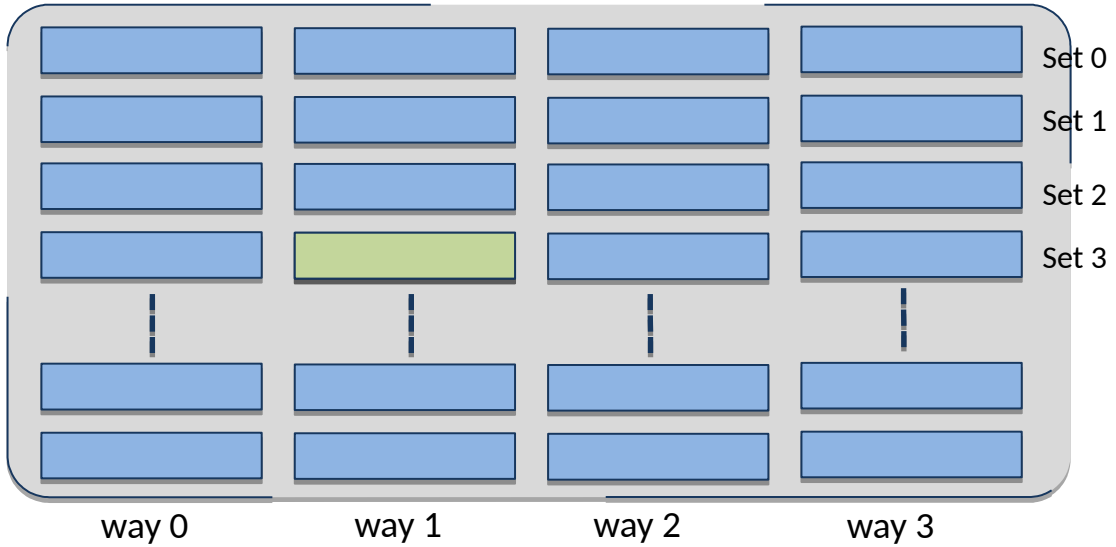
Not normal Circumstances

Virtual address space of process




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

Cache Memory

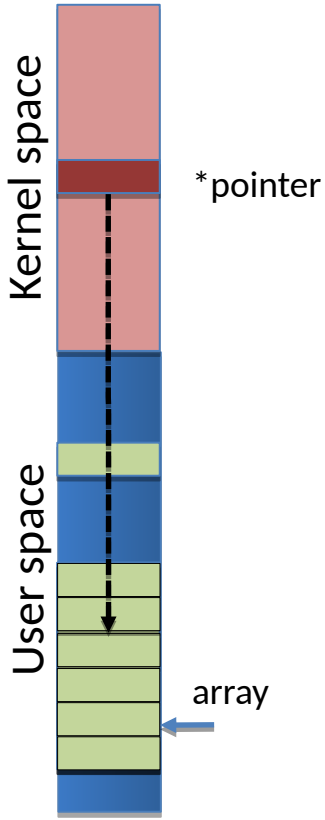


cache miss

Meltdown Concept

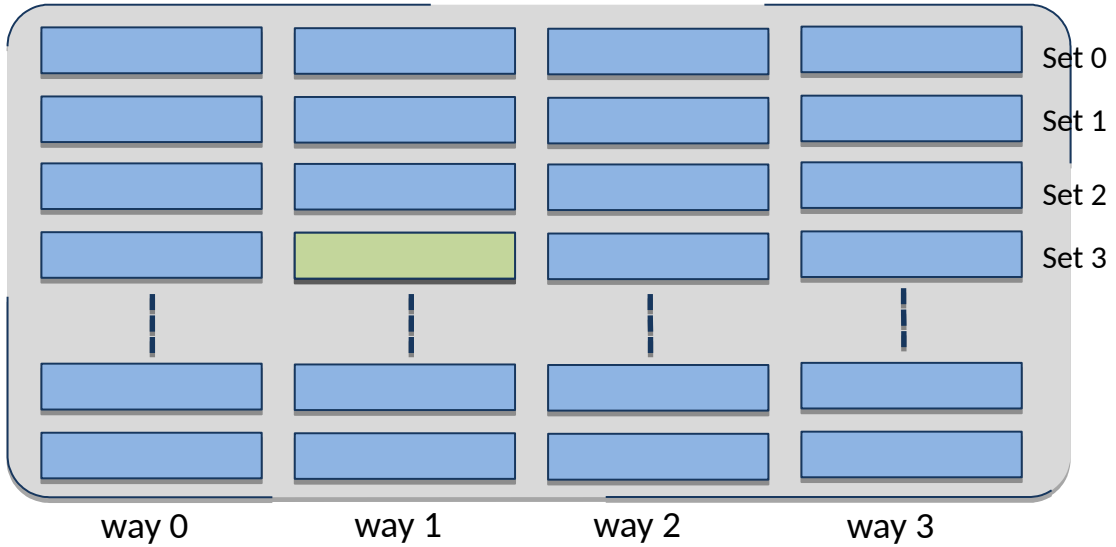
Not normal Circumstances

Virtual address space of process




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

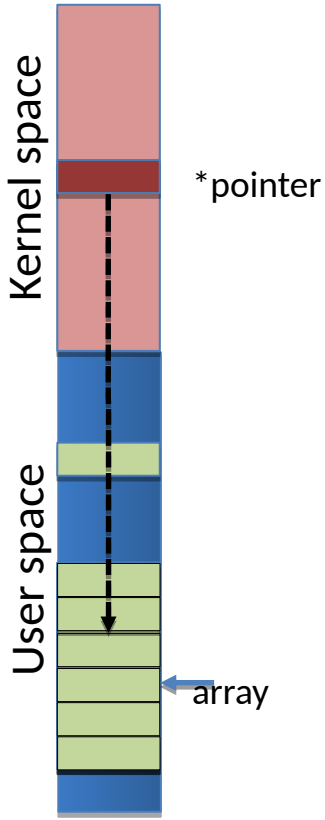
Cache Memory



cache miss

Meltdown Concept

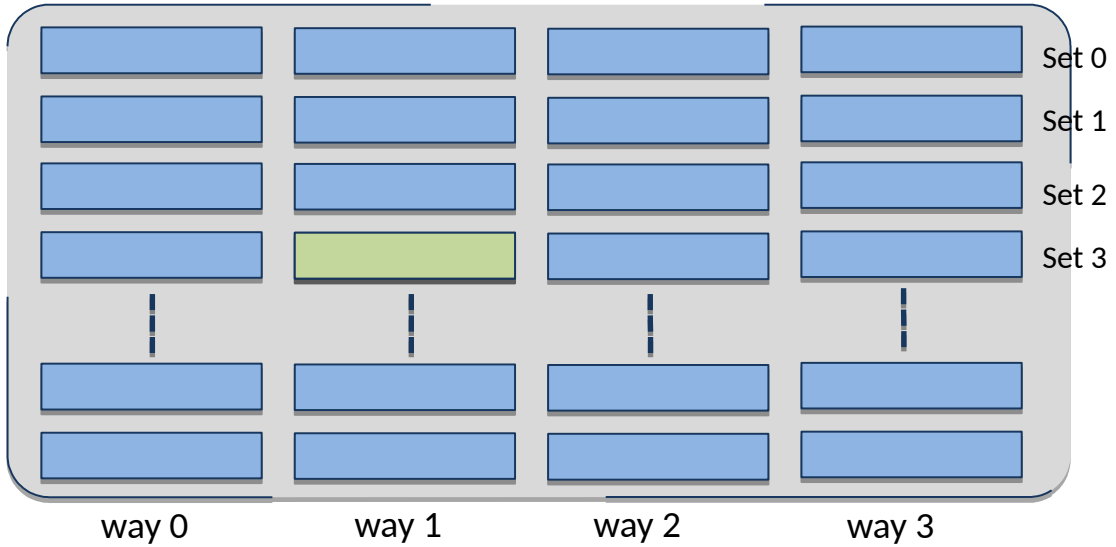
Virtual address space of process



Not normal Circumstances

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

Cache Memory

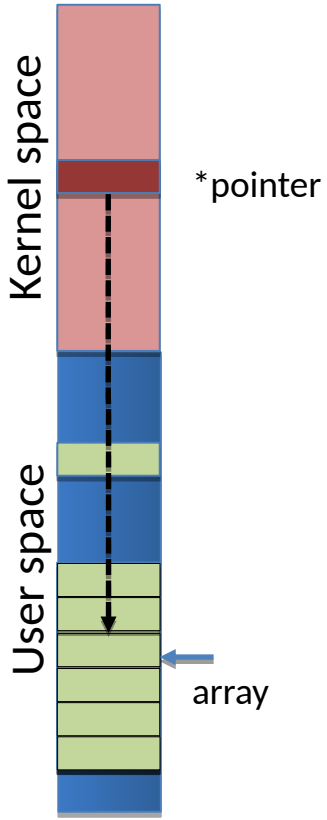


cache miss

Meltdown Concept

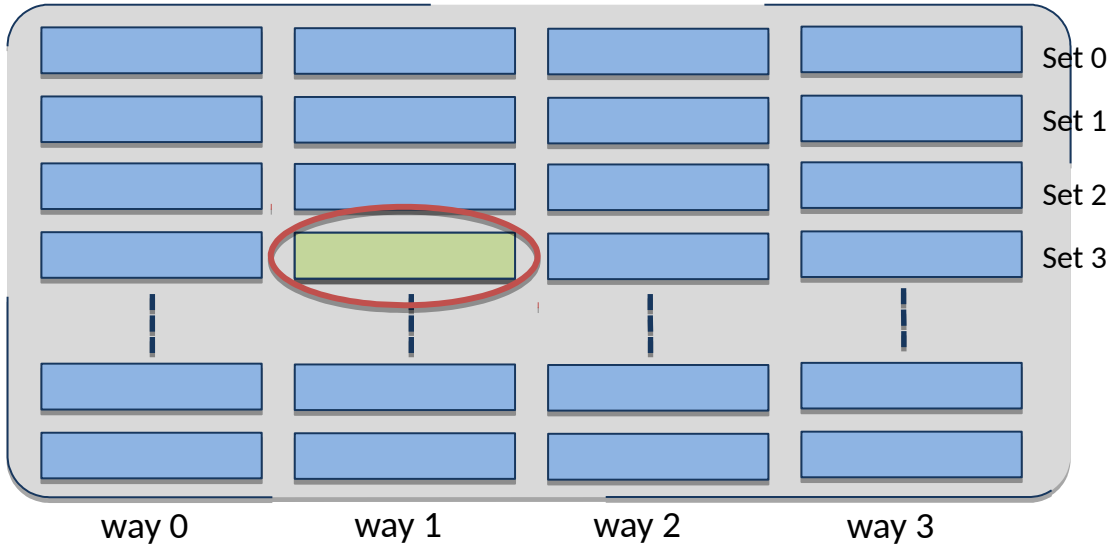
Not normal Circumstances

Virtual address space of process



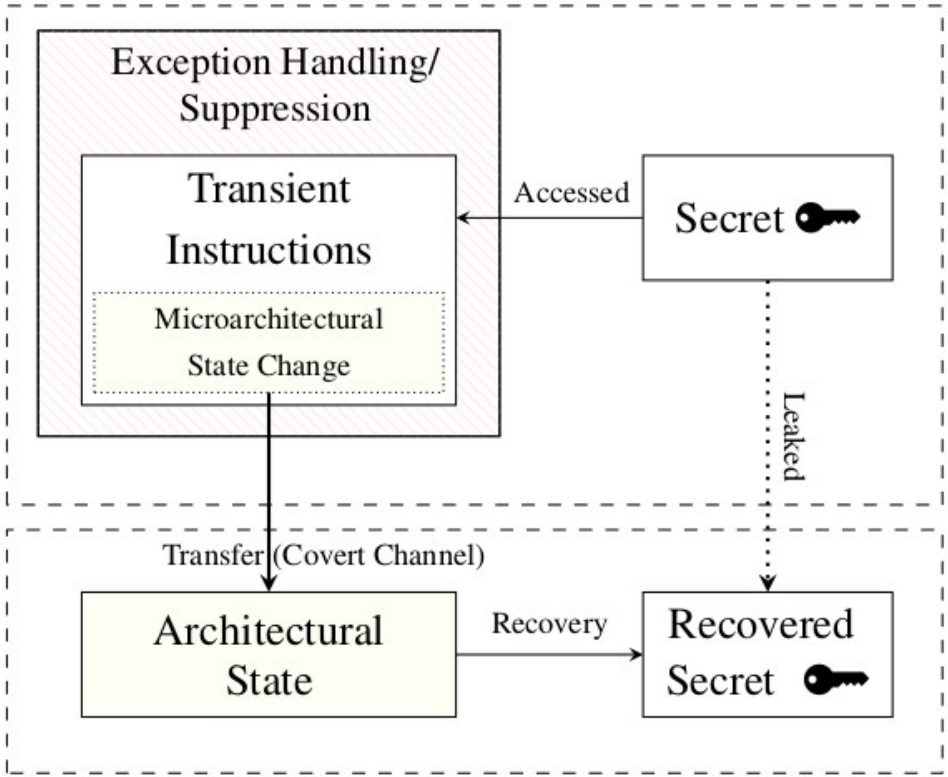

```
i = *pointer
y = array[i * 256]
```

Cache Memory



cache hit

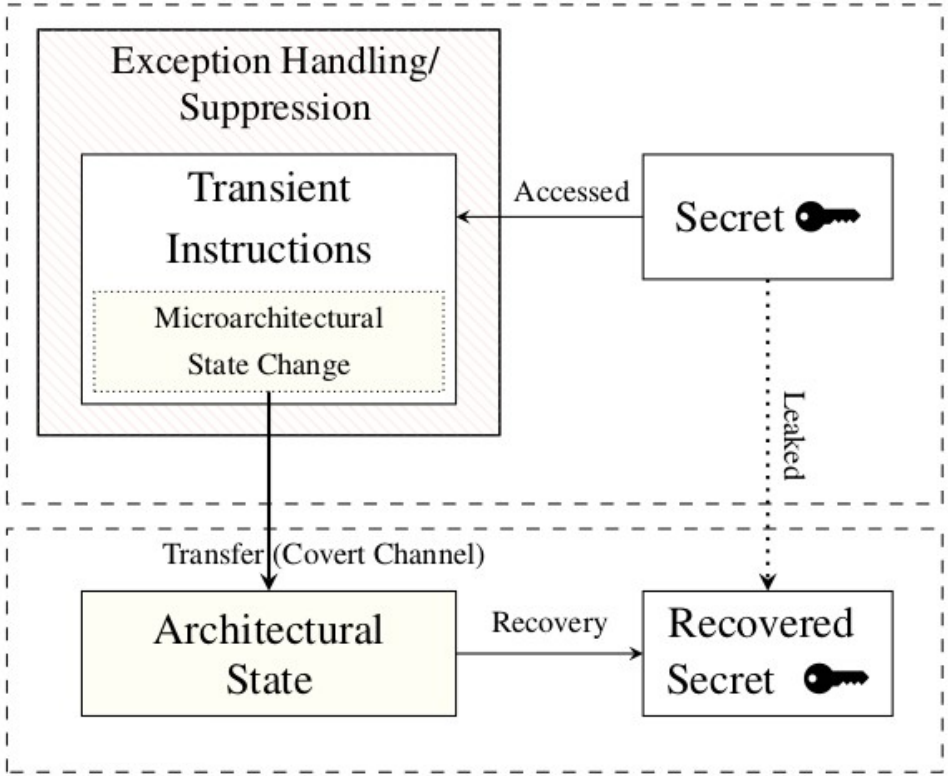
Meltdown : The Attack



* **<u>Executing Transient Instructions</u>**
  - Exception Handling
  - Exception Supression
* **<u>Building a Covert Channel</u>**

Credits : Moritz Lipp et al

Meltdown : The Attack



Step 1
Content of attacker-chosen memory location loaded into register.

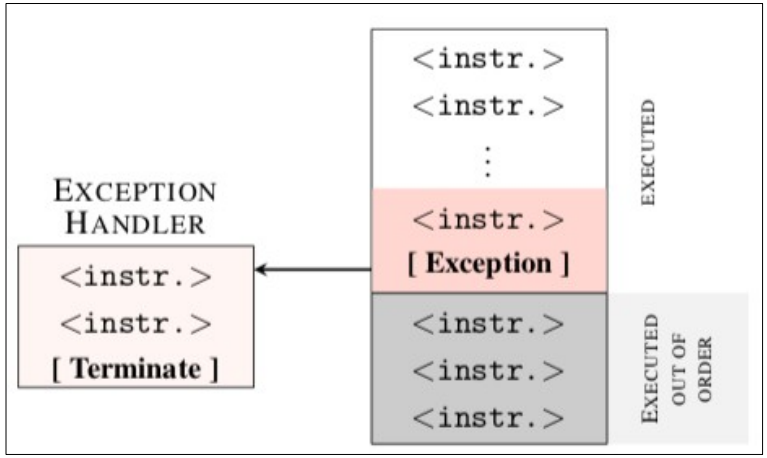
Step 2
Transient instruction accesses cache line based on secret content of register.

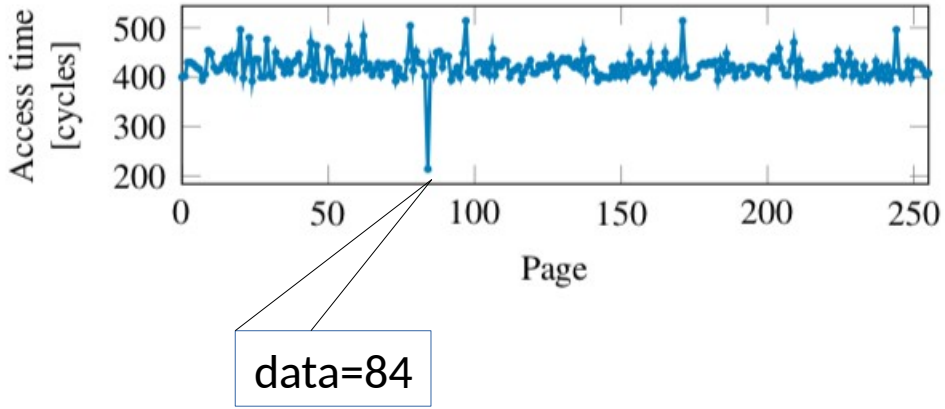
Step 3
Attacker uses Flush+Reload to determine accessed cache line and secret stored at chosen memory location.

Credits : Moritz Lipp et al

Speculative Execution and Micro-architectural State

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





Credits : Moritz Lipp et al

Spectre

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

Speculative Execution : Case 1

```
   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

Speculated results discarded

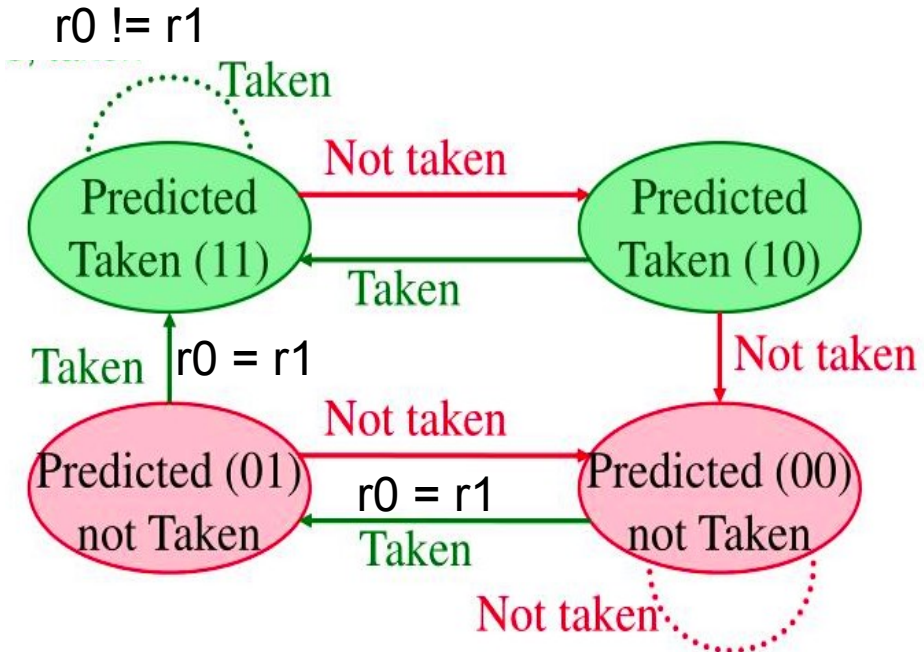
:
:
:

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

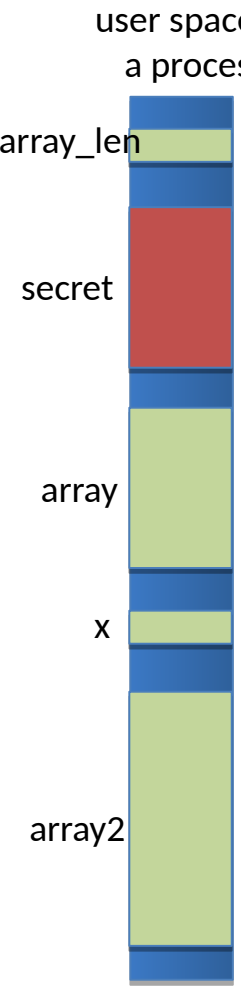
Speculative execution
(transient instructions)

Branch Prediction

```
   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
```



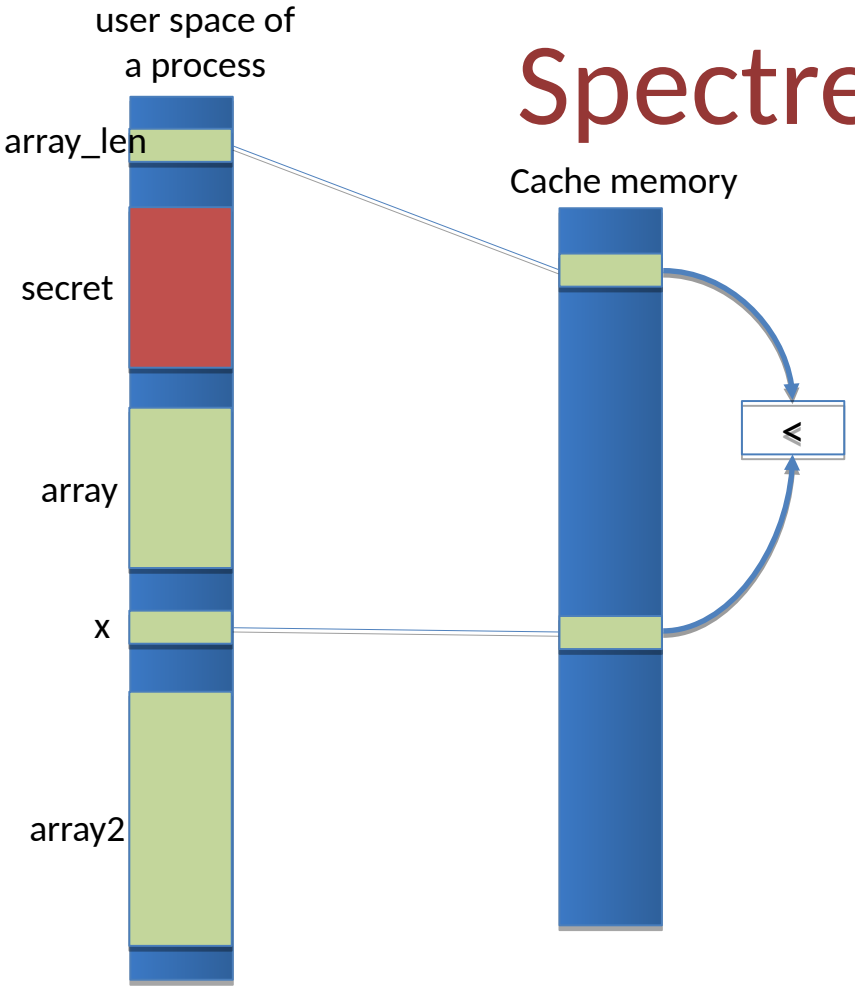
Spectre (Variant 1)



Cache memory

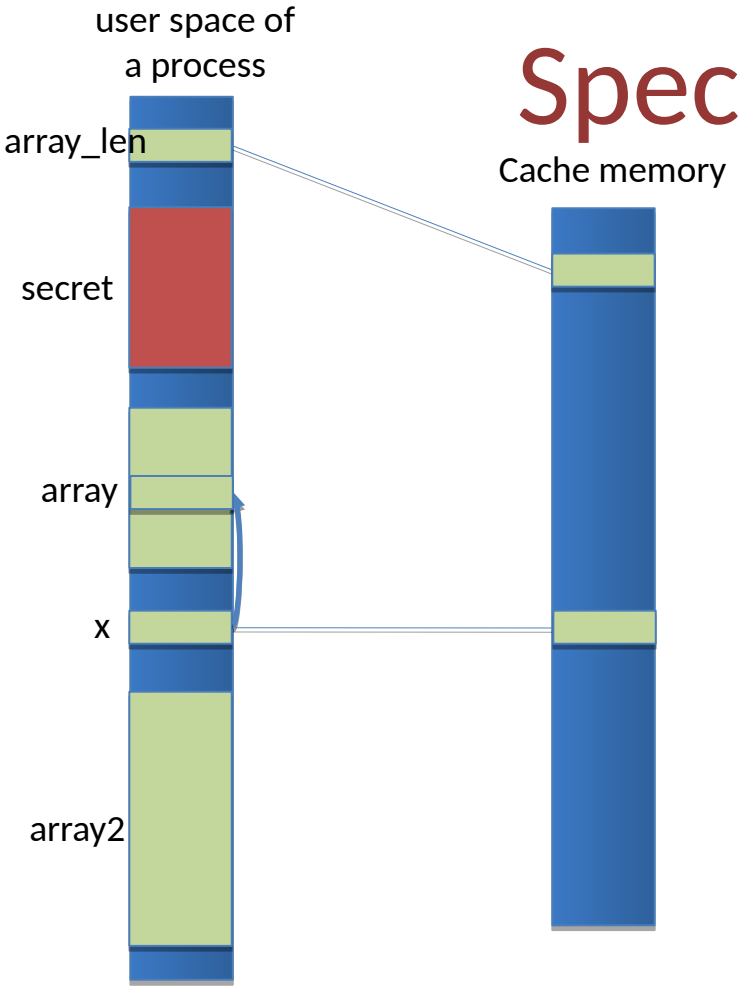
```
if (x < array_len){
    i = array[x];
    y = array2[i * 256];
}
```

Spectre (Variant 1)




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

Spectre (Variant 1)

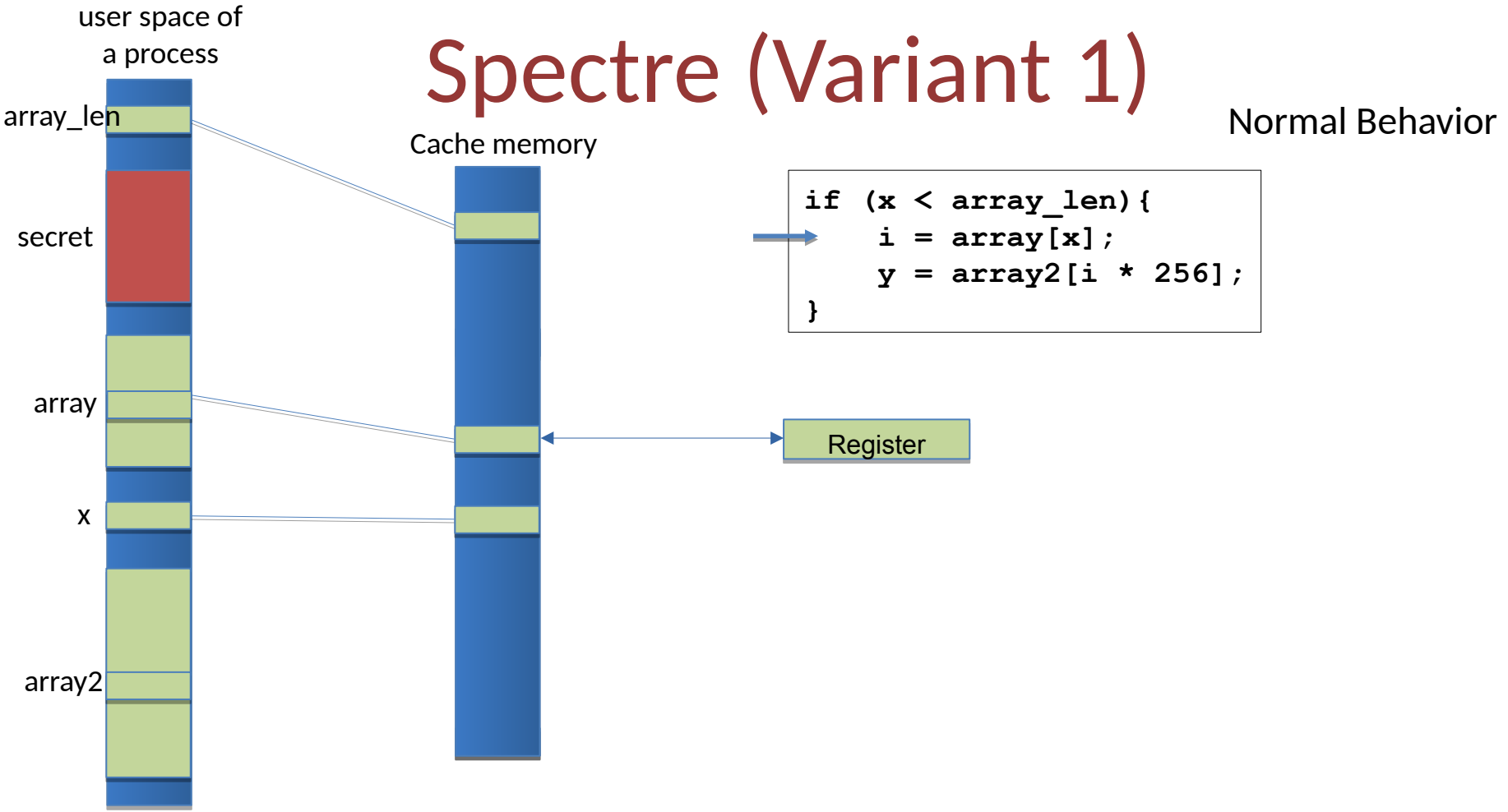


Normal Behavior

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

Spectre (Variant 1)

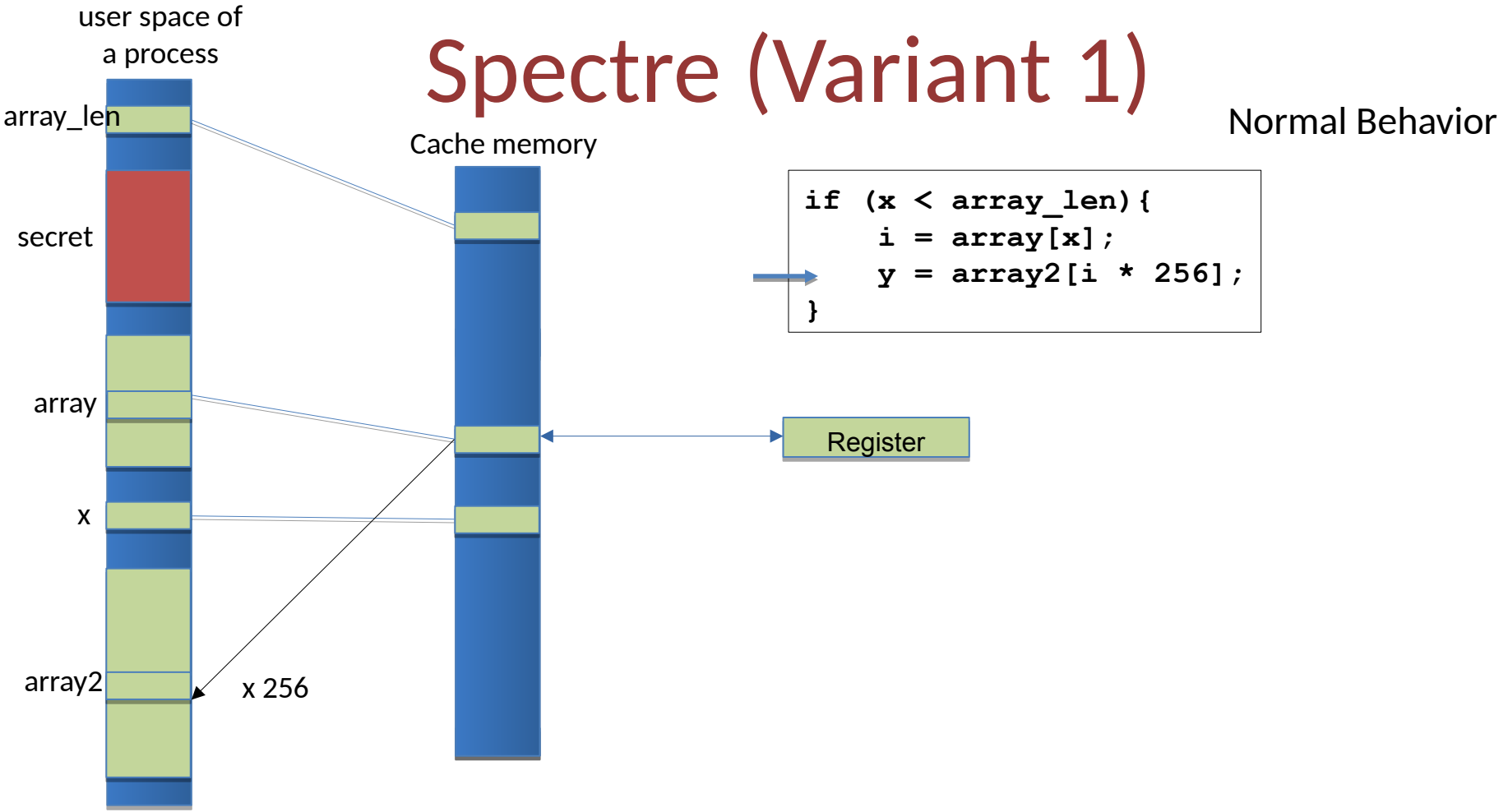
Normal Behavior




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

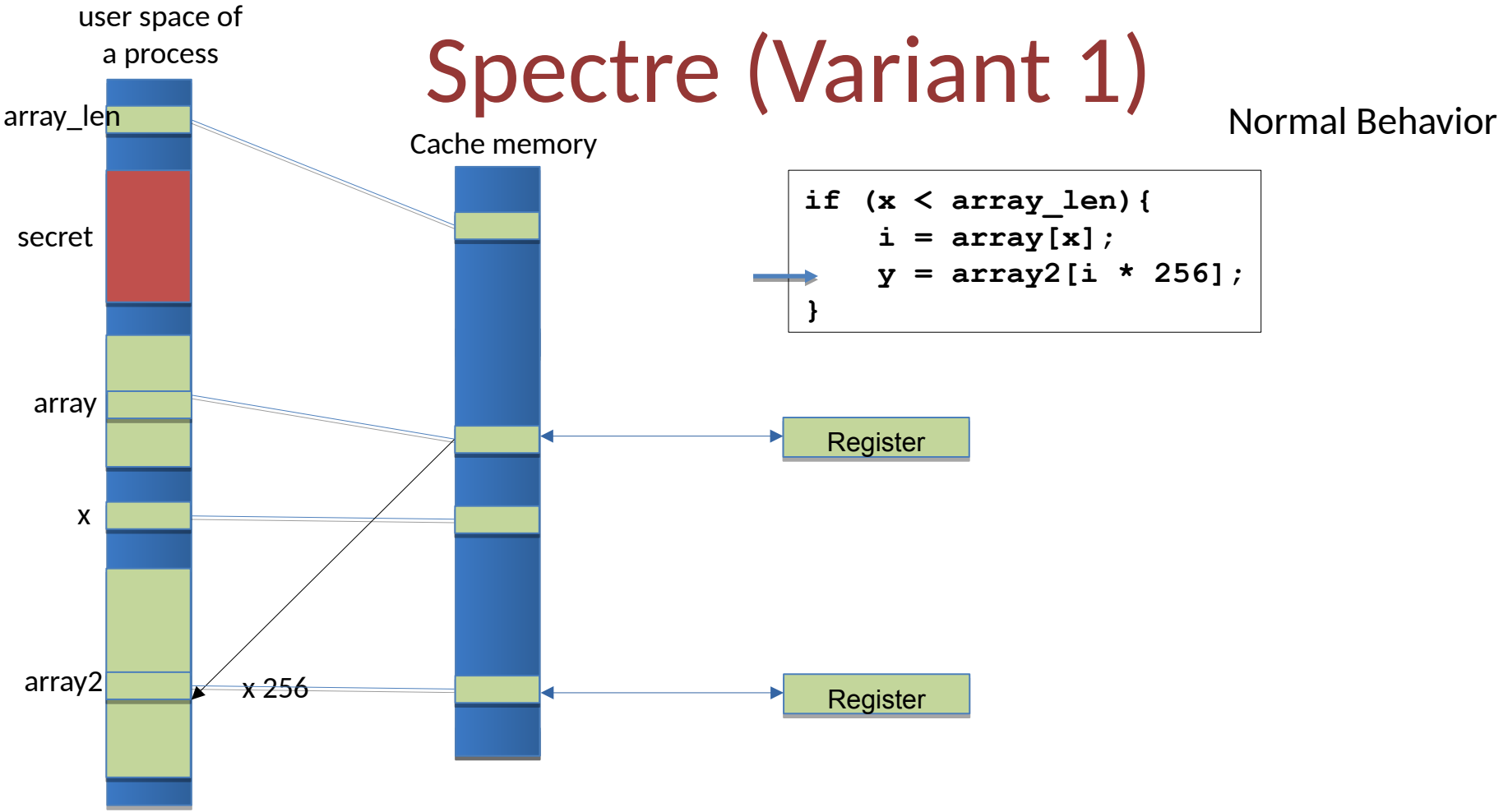
Spectre (Variant 1)

Normal Behavior



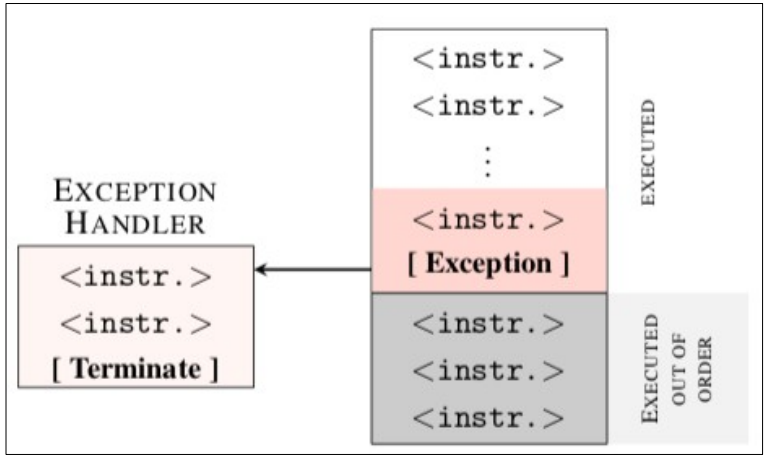
Spectre (Variant 1)

Normal Behavior



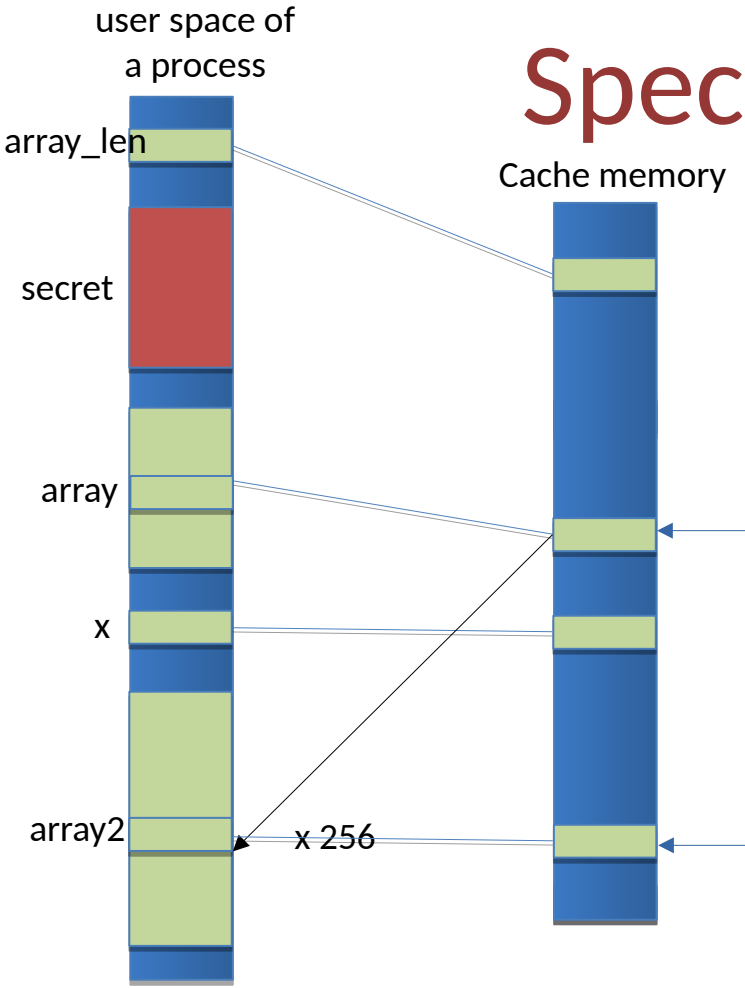
Speculative Execution and Micro-architectural State

```
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.

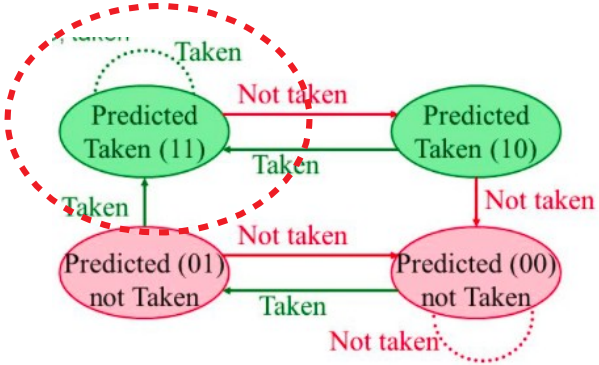
Spectre (variant 1)



Normal Behavior

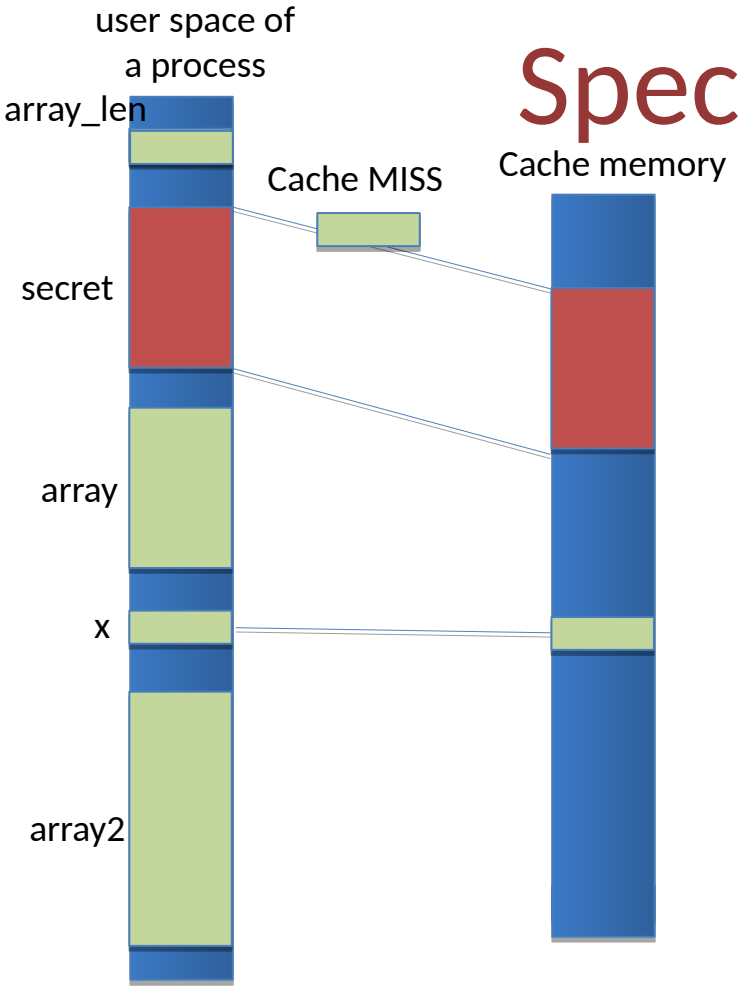
```
if (x < array_len){
    i = array[x];
    y = array2[i * 256];
}
```

Multiple TAKEN Loops



Branch TAKEN = TRUE if Condition

Spectre (Variant 1)

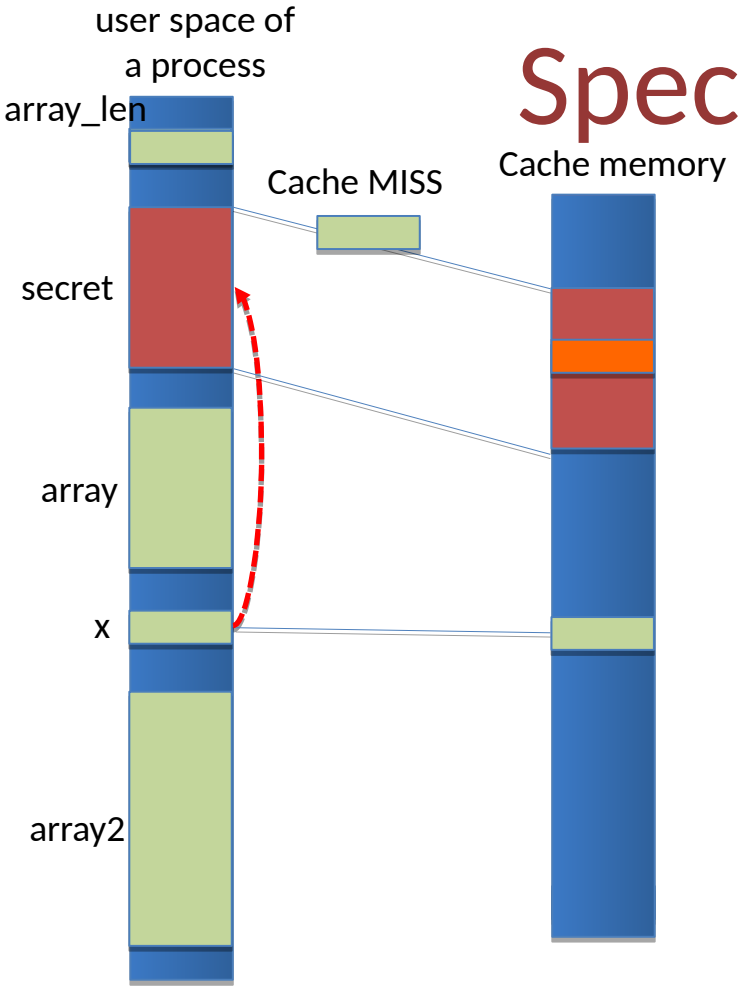


Under Attack

```
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)



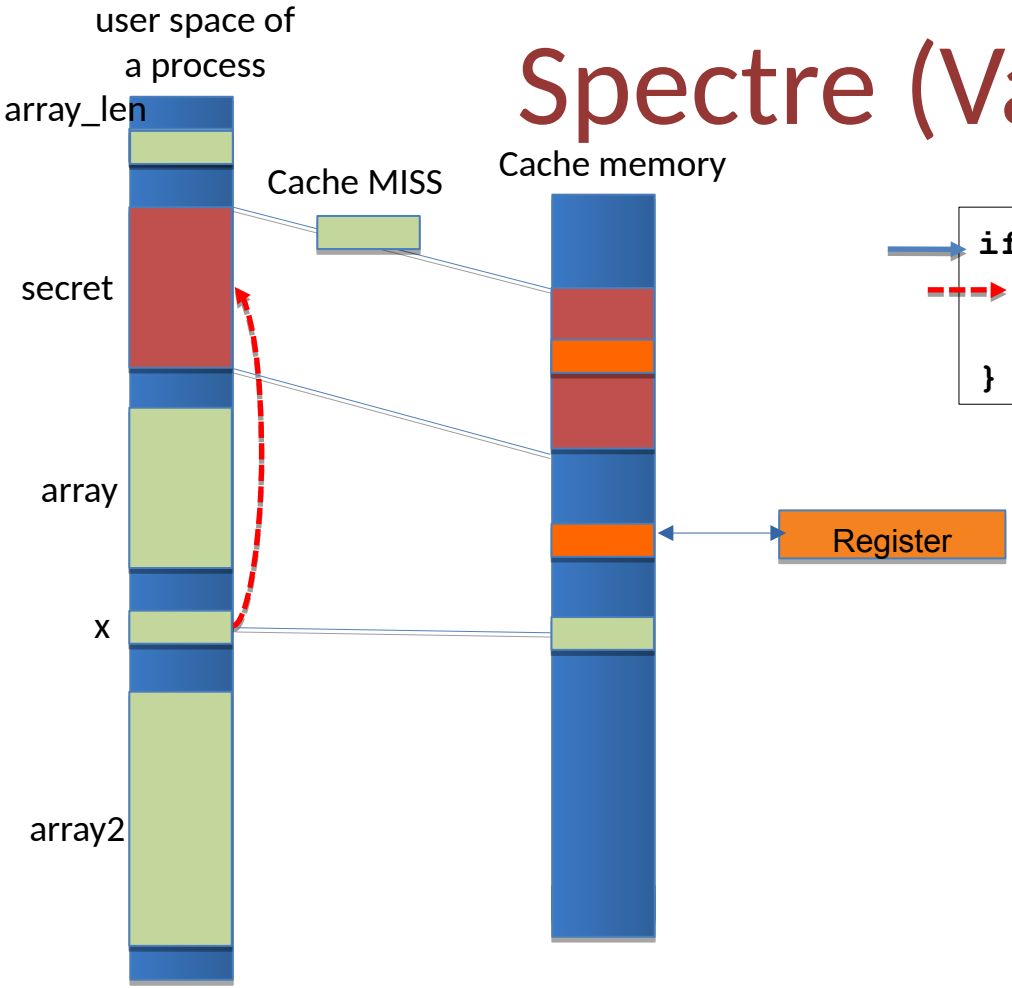
Under Attack

```
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)

Under Attack

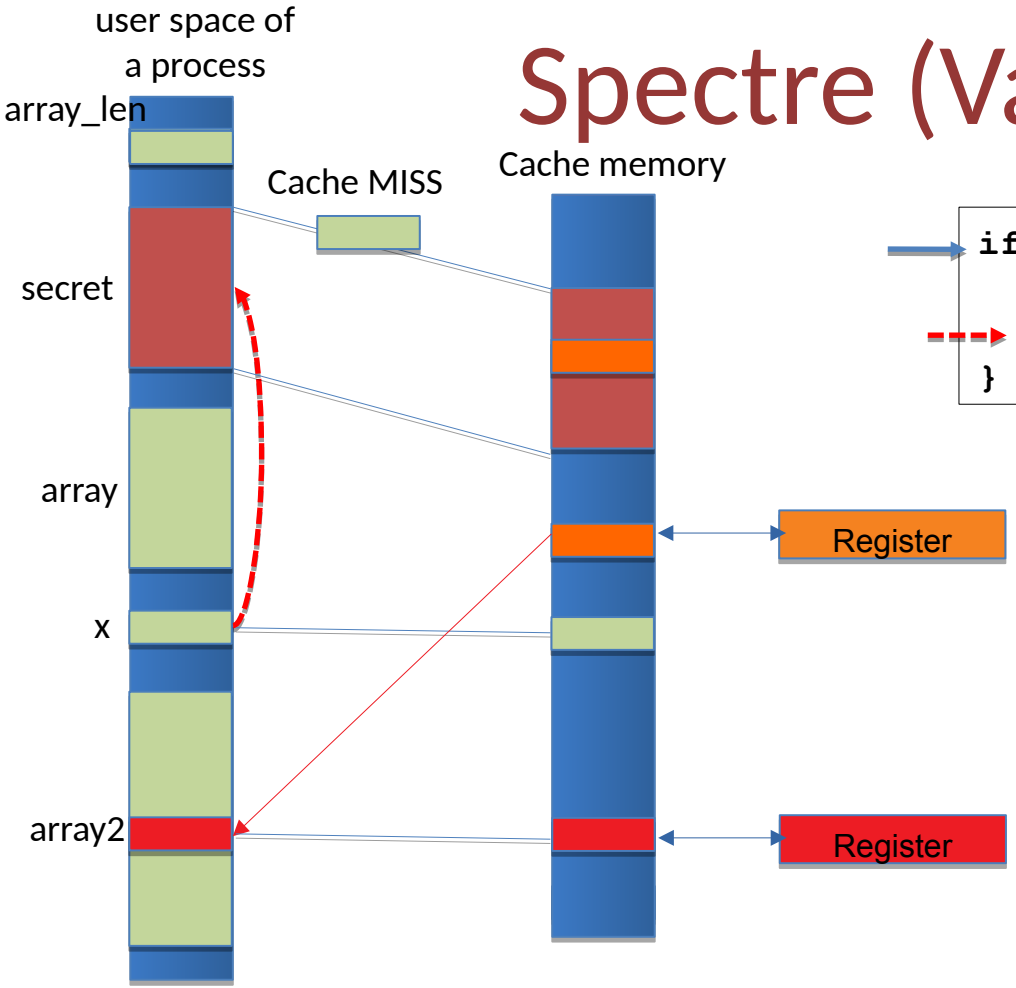



```
if (x < array_len){
    i = array[x];
    y = array2[i * 256];
}
```

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

Spectre (Variant 1)

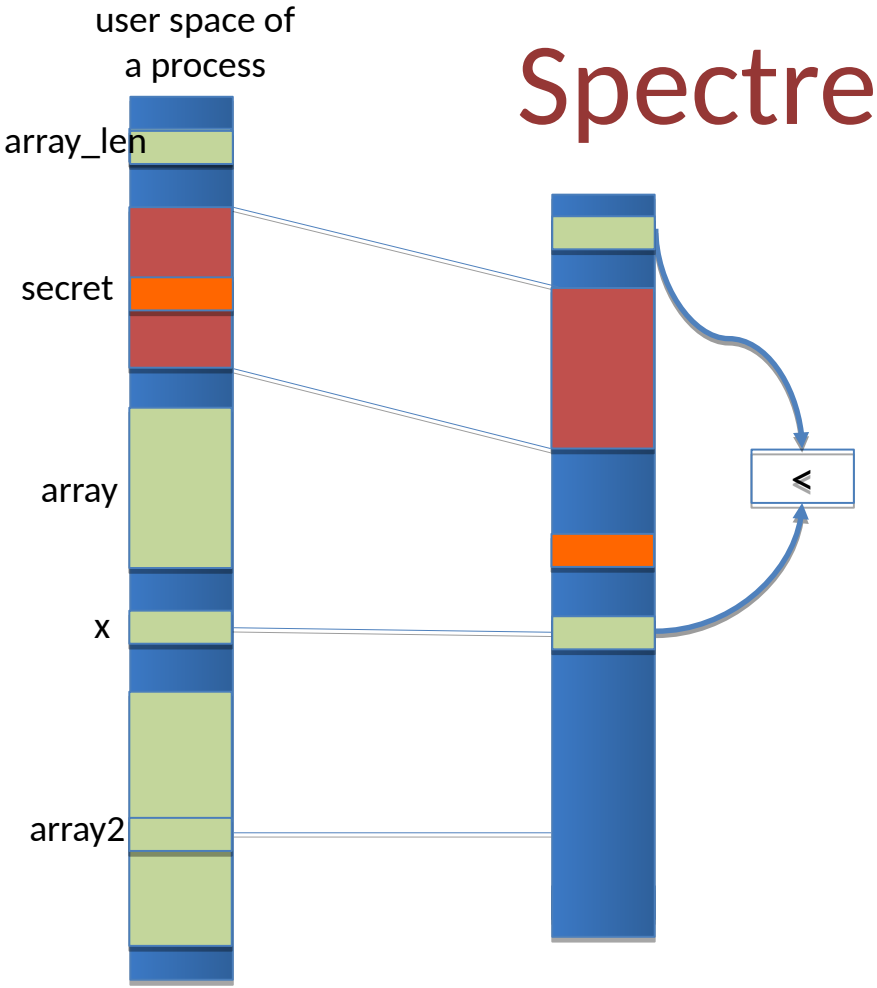
Under Attack




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

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

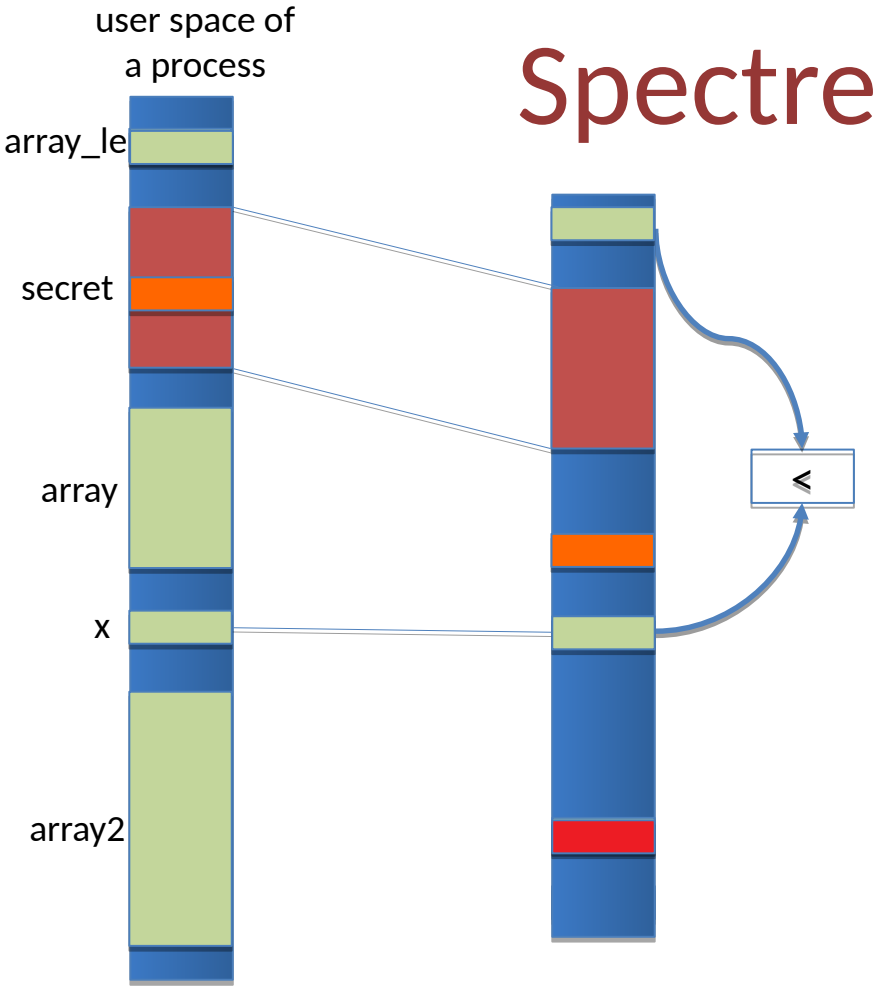
Spectre (Variant 1)




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

Misprediction!

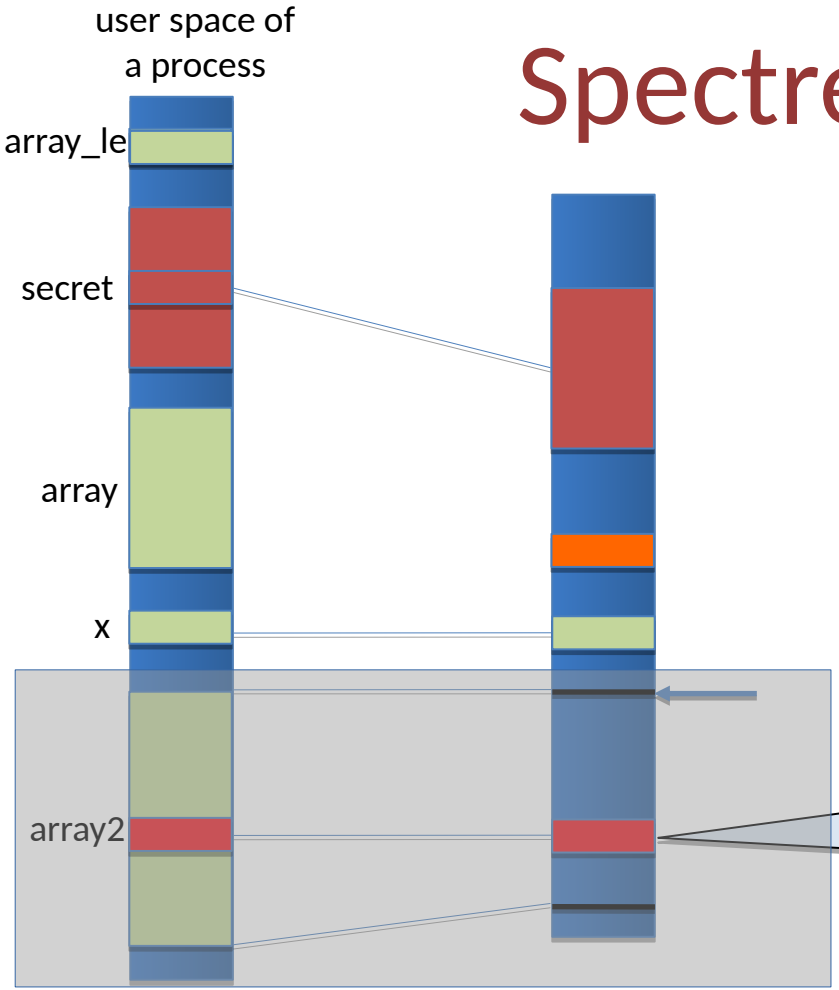
Spectre (Variant 1)




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

Misprediction!

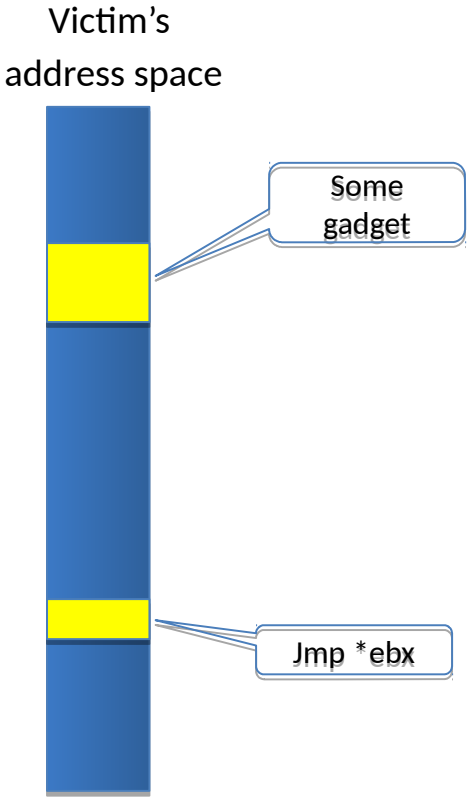
Spectre (Variant 1)



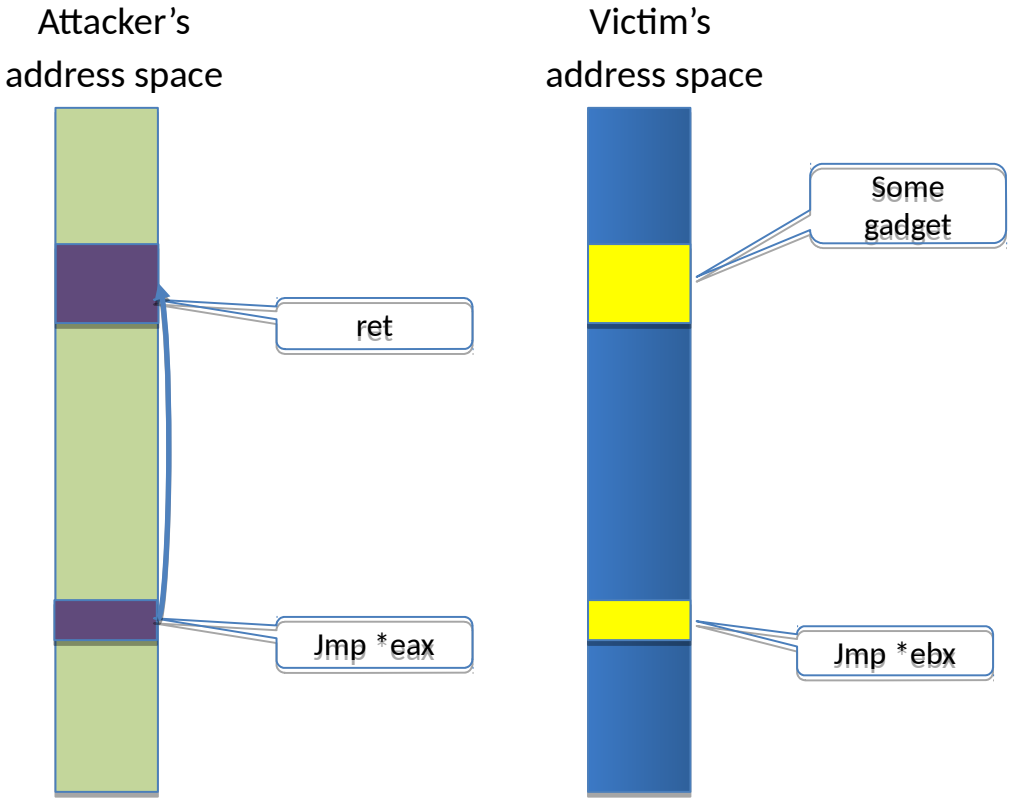

```
if (x < array_len){
    i = array[x];
    y = array2[i * 256];
}
```

Cache hit found here by FLUSH_RELOAD attack

Spectre (Variant 2)



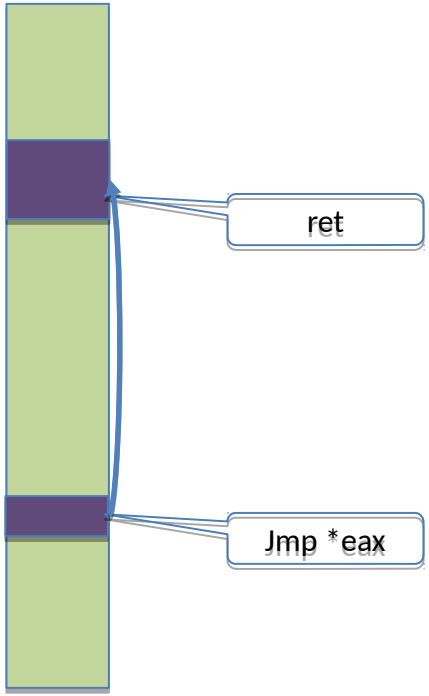
Spectre (Variant 2)

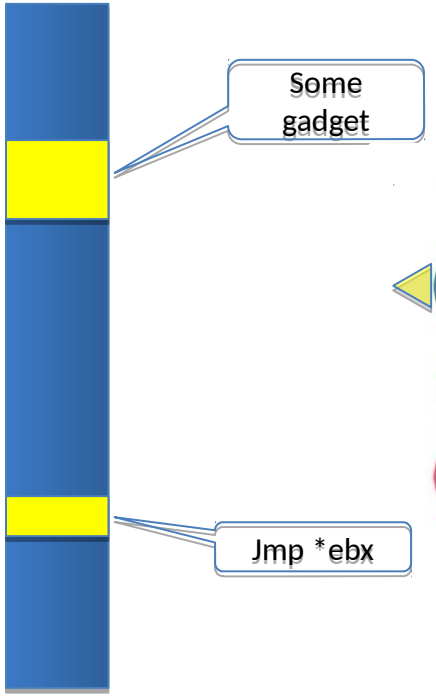


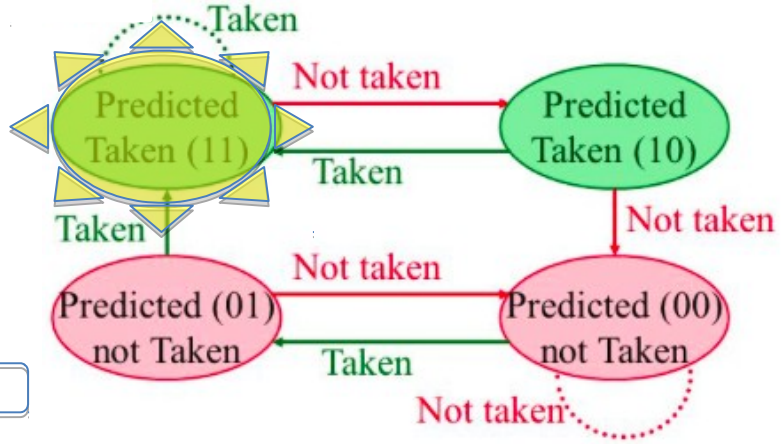
Spectre (Variant 2)

Attacker's address space

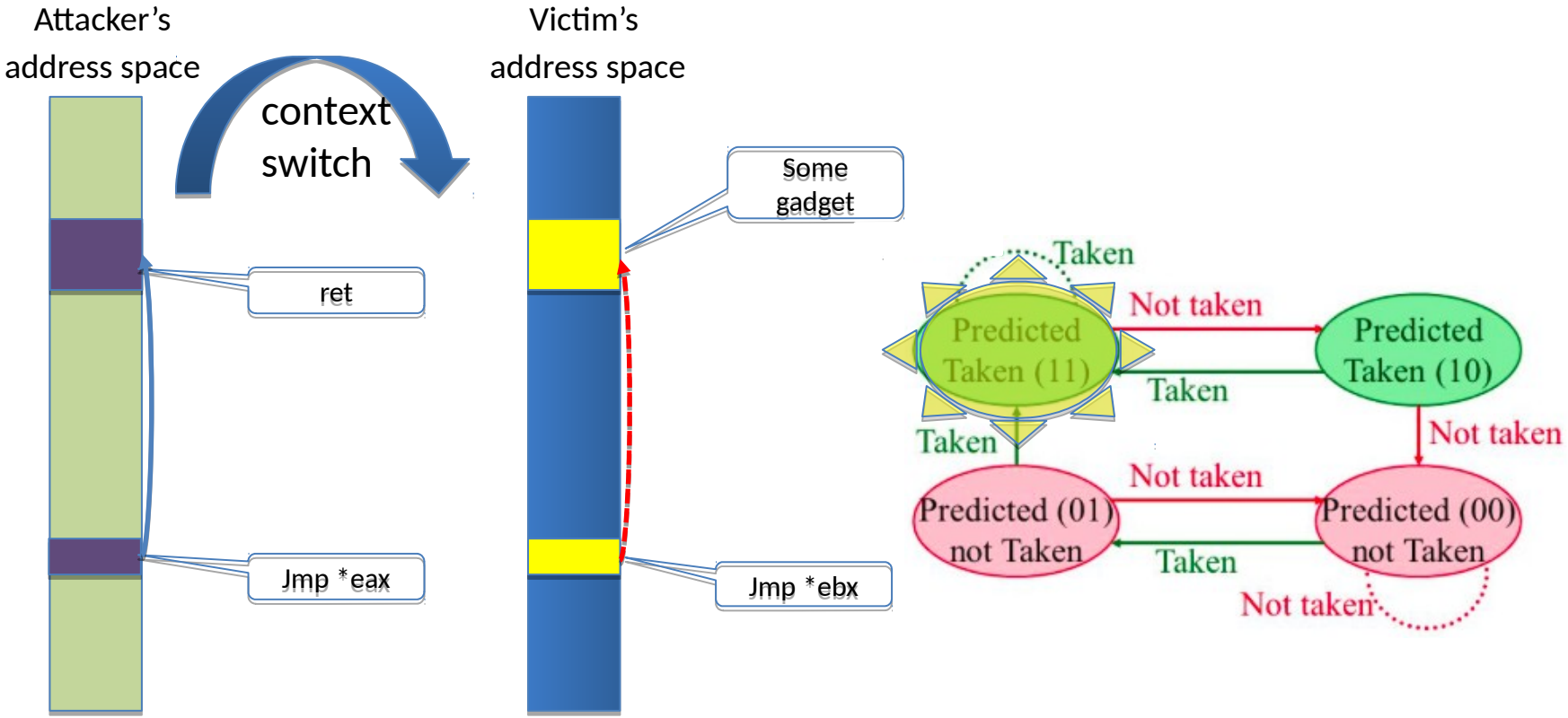
Victim's address space





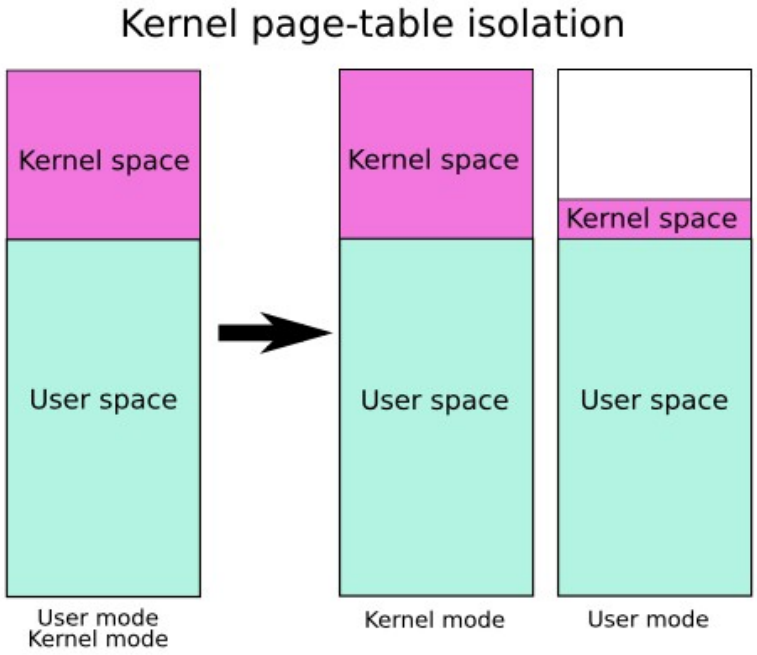


Spectre (Variant 2)



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

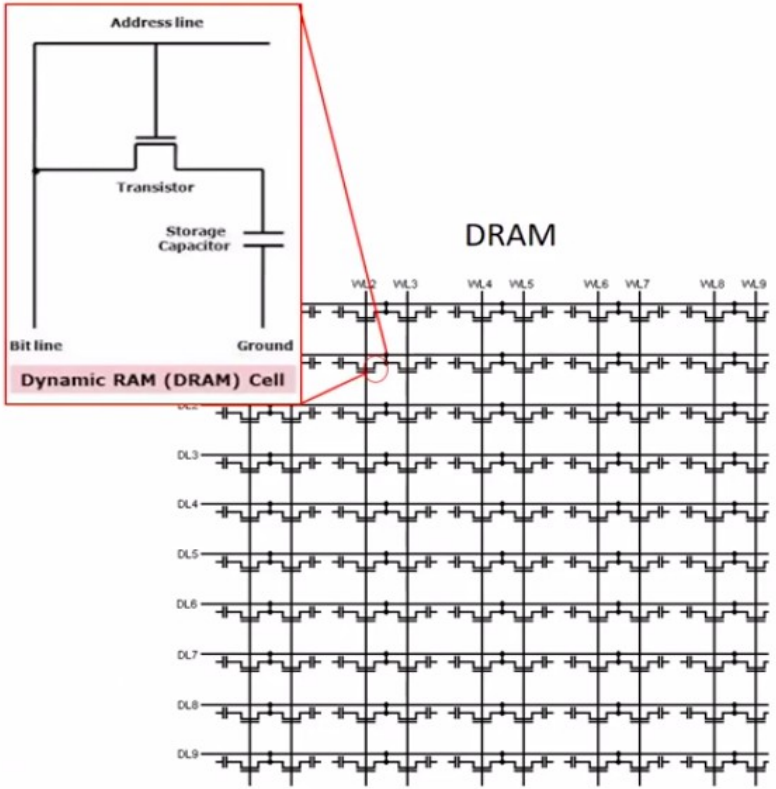
Countermeasures

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

The Rowhammer Attack

Some slides borrowed from Prof. Onur Mutlu's talk at DATE
Invited talk | March 30, 2017

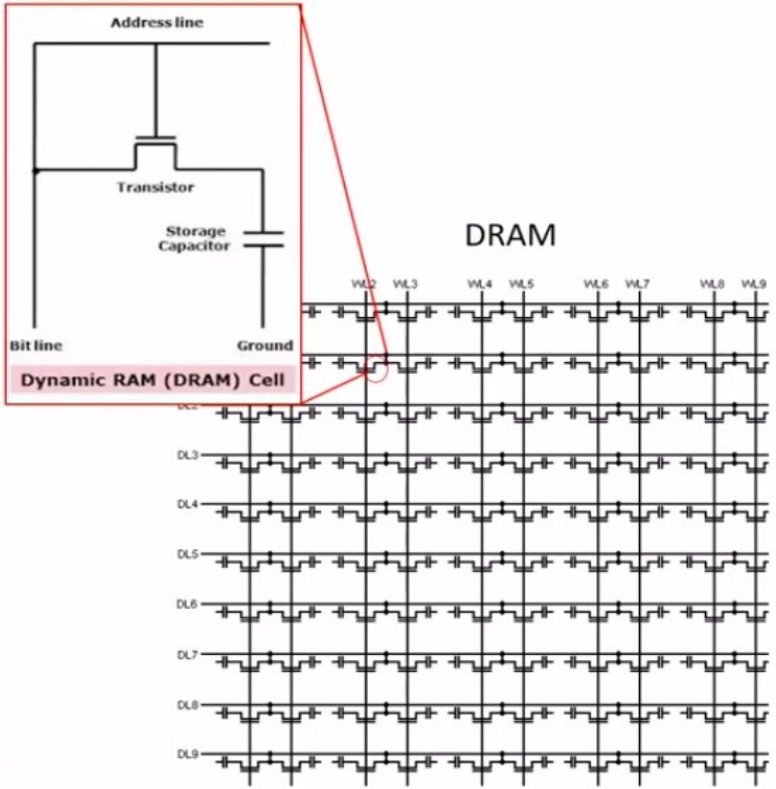
DRAM


Address line
Transistor
Storage Capacitor
Bit line
Ground
Dynamic RAM (DRAM) Cell
DRAM
WL2
WL3
WL4
WL5
WL6
WL7
WL8
WL9
DL3
DL4
DL5
DL6
DL7
DL8
DL9

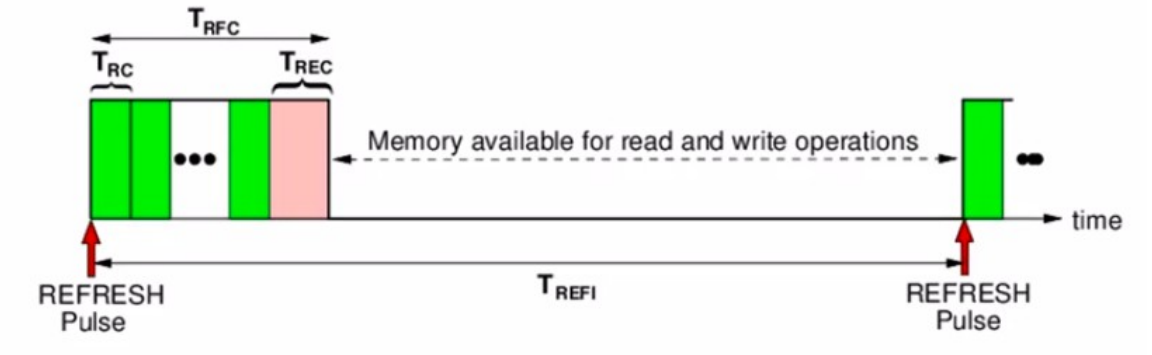
DRAM

- DRAM stores charge in a capacitor

- Capacitor must be large for reliable sensing



DRAM Refresh Cycles

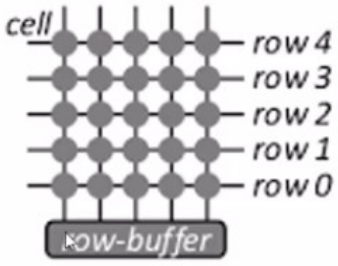


- Stored data gets lost due to leakage

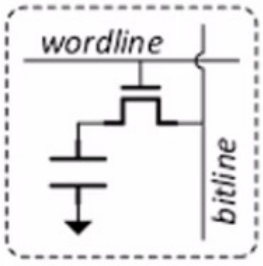
- Need for capacitor refresh

- Refresh cycles are in order of miliseconds still consuming some memory bandwidth

DRAM Cells



a. Rows of cells



b. A single cell

- Scaling beyond 35-40 nm is challenging

- DRAM cells become smaller with reduction in transistor size

- Space between cells also reduces

- Closer the two charged bodies, higher is the electro magnetic interference

Rowhammer

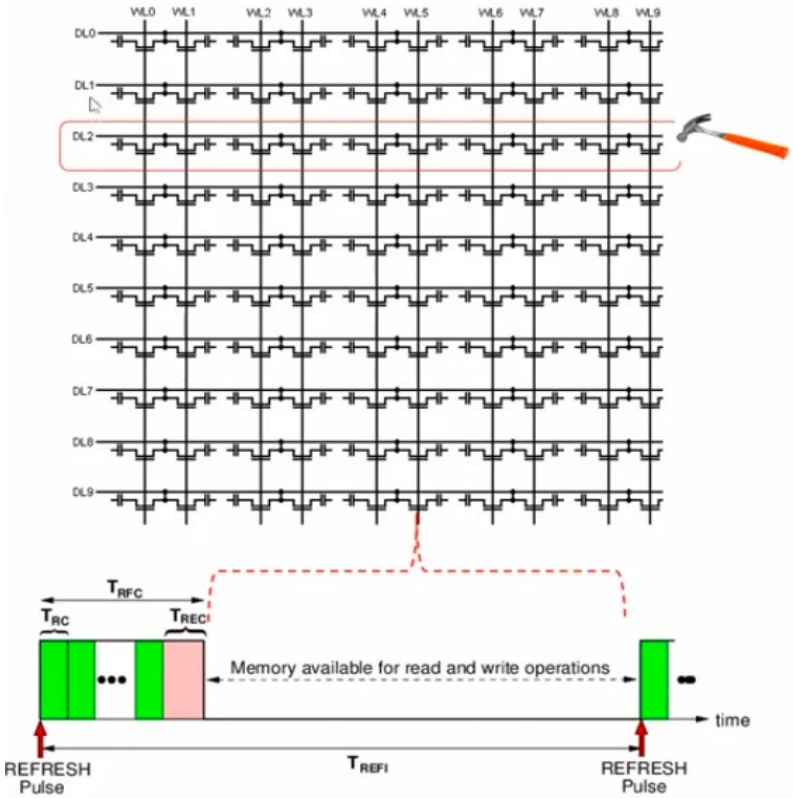
- Access during refresh cycle causes neighbouring cells to loose charge faster

  - Electromagnetoc decoupling

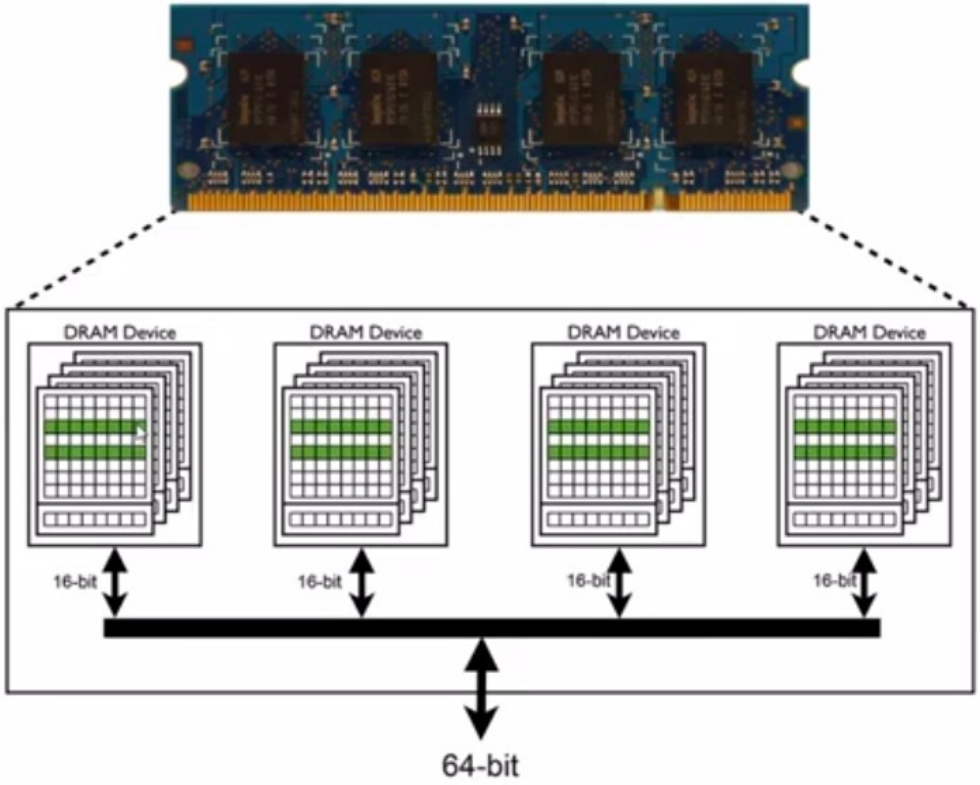
  - Toggling a row increases the adjacent row voltage

  - Opens adjacent row : Charge leakage

- Data corrupted, read and written back

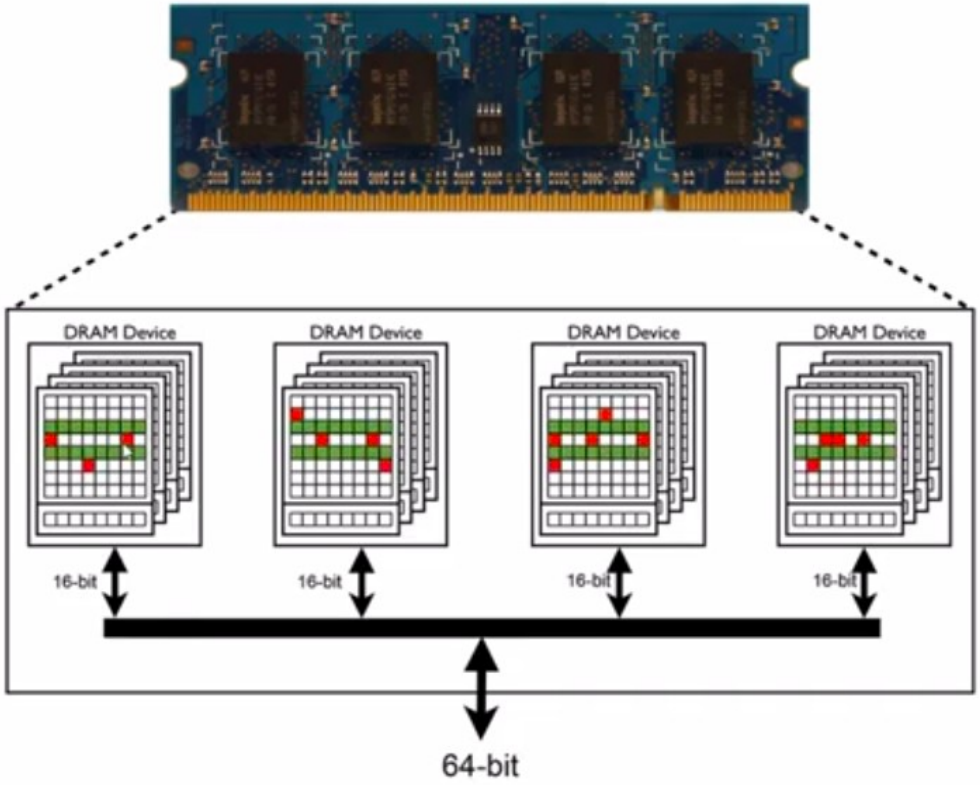


Rowhammer



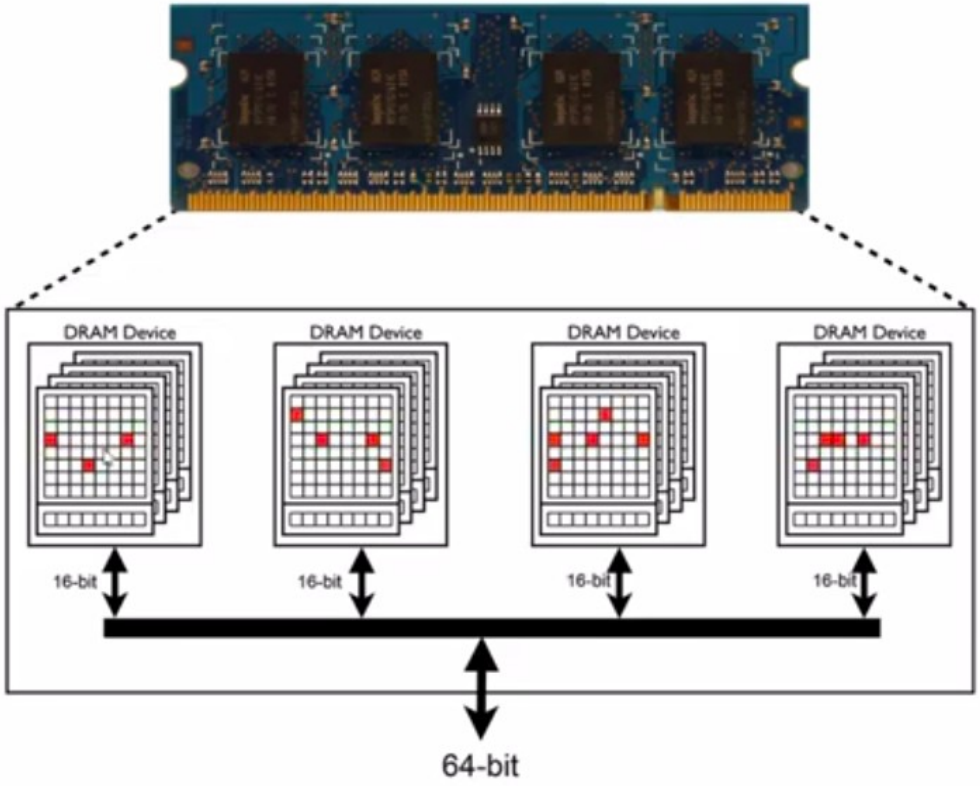
http://apt.cs.manchester.ac.uk/projects/ARMOR/RowHammer/

Rowhammer



http://apt.cs.manchester.ac.uk/projects/ARMOR/RowHammer/

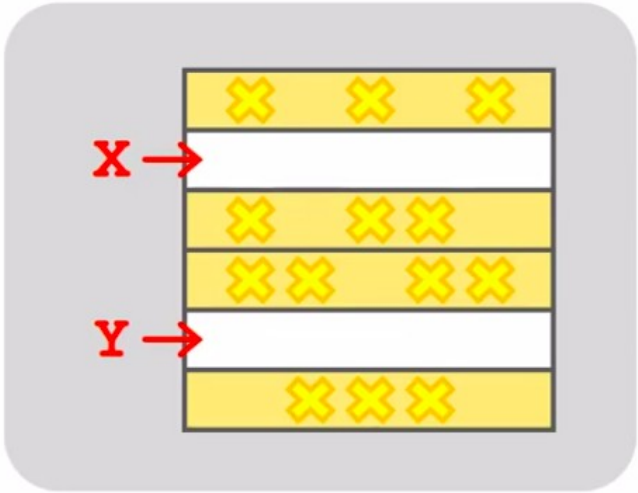
Rowhammer



http://apt.cs.manchester.ac.uk/projects/ARMOR/RowHammer/

Rowhammer Example

```
loop:
  mov (X), %eax
  mov (Y), %ebx
  clflush (X)
  clflush (Y)
  mfence
  jmp loop
```



To Avoid cache hits => Flush x from cache
To void row hits to x in the row buffer => Read y in another row

Download from: https://github.com/CMU-SAFARI/rowhammer

Countermeasures

- ❑ Increase the access interval of the aggressor (attacking row). Less frequent accesses => fewer errors
- ❑ Decrease the refresh cycles. More frequent refresh => fewer errors
- ❑ Pattern of storing data in DRAMs.
- ❑ Sophisticated Error Corrections. (As many as 4 errors were found per cache line)

That’s for the Lectures !!