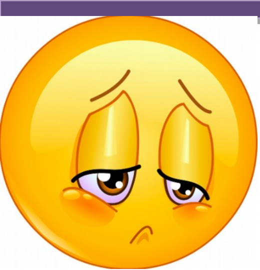
Micro-architectural Attacks 2

Milan Patnaik
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)

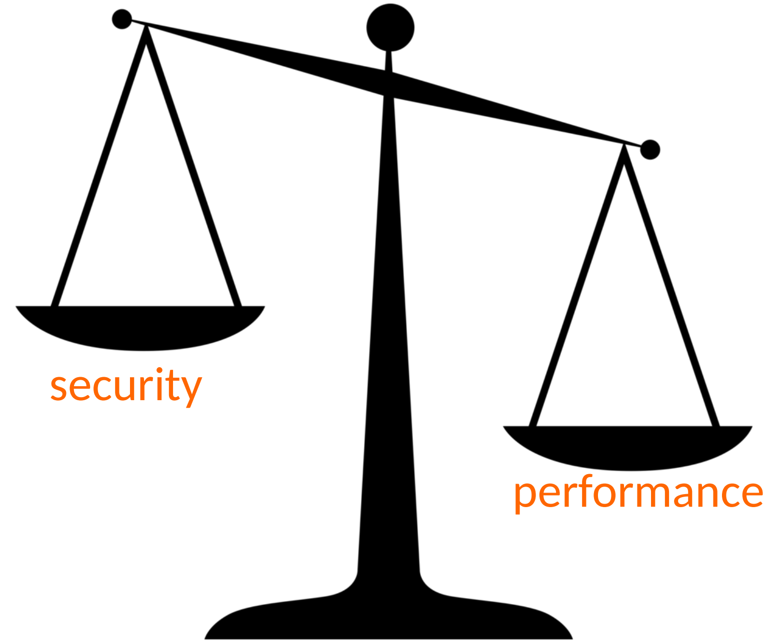
….. and many more

Causes

Most micro-architectural attacks caused by performance optimizations

Others due to inherent device properties

Third, due to stronger attackers



Instruction Level Parallelism

Out-of-order execution

How instructions are fetched	How they may be executed	How the results are committed
`load r0, addr1`	`sub r4, r5, r6`	`r0`
`mov r2, r1`	`store r1, add2`	`r2`
`add r2, r2, r3`	`mov r2, r1`	`r2`
`store r1, add2`	`add r2, r2, r3`	`addr2`
`sub r4, r5, r6`	`load r0, addr1`	`r4`
inorder	out-of-order	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
```

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

Speculated results discarded

:
:
:

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

Speculative execution (transient instructions)

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

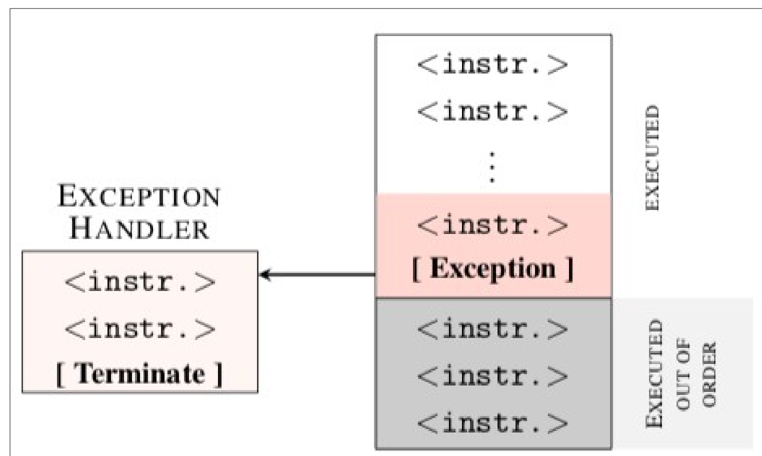
Speculated results discarded

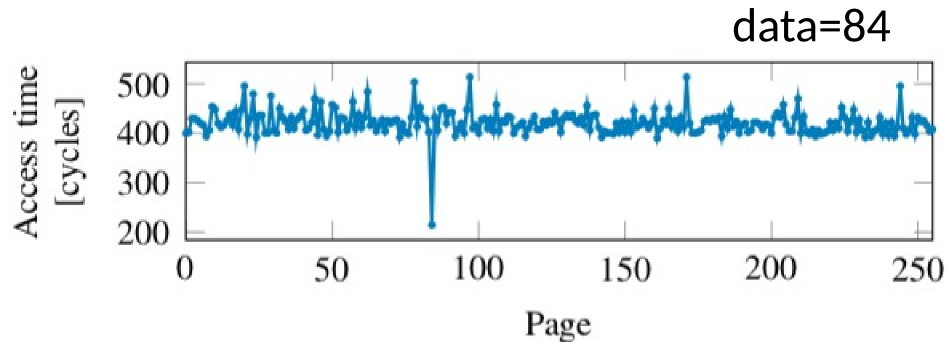
:
:
:

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]);
```





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

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

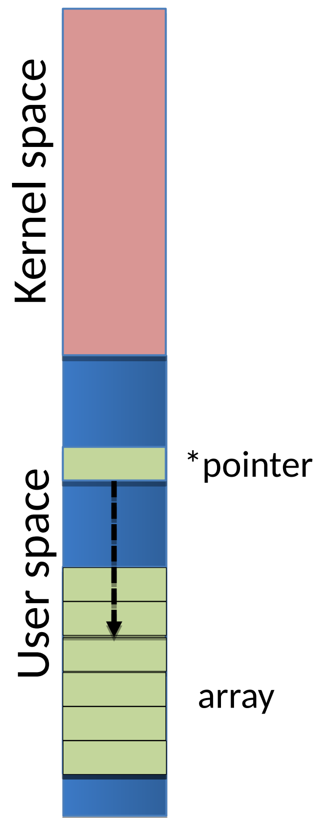
Meltdown

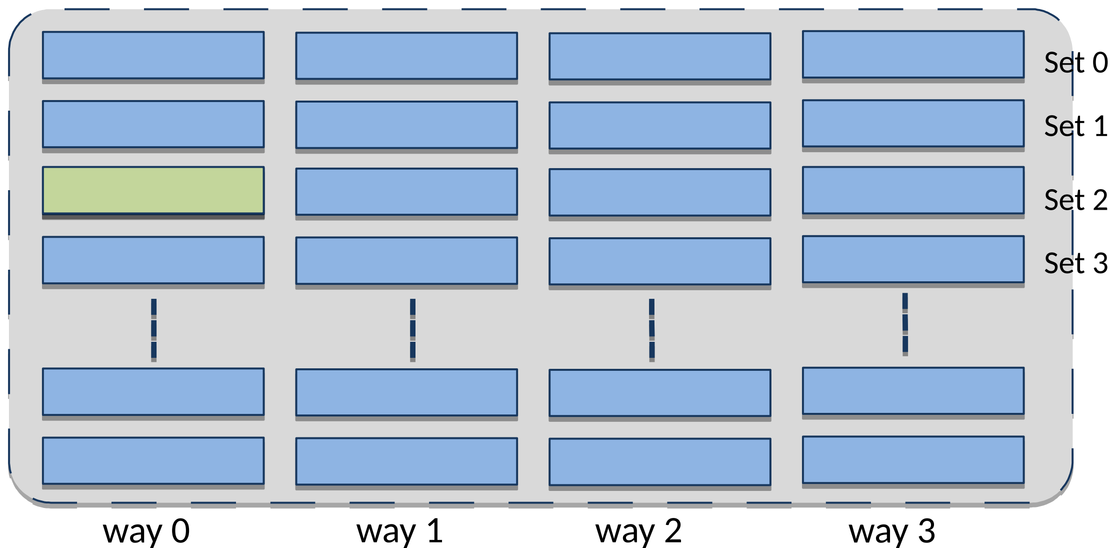
Slides motivated from Yuval Yarom’s talk on Meltdown and Spectre at the Cyber security research bootcamp 2018

Meltdown

Normal Circumstances

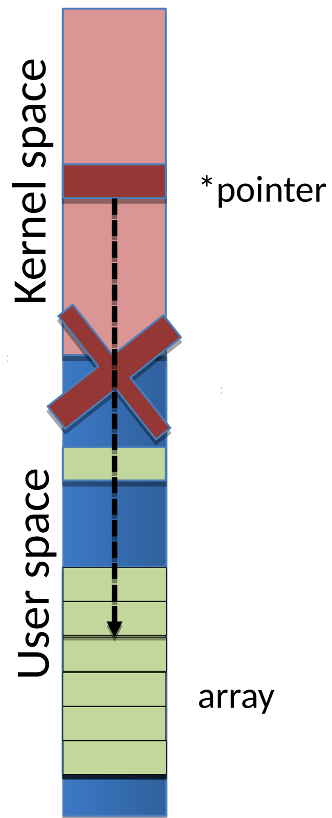
```
i = *pointer
y = array[i * 256]
```





Meltdown

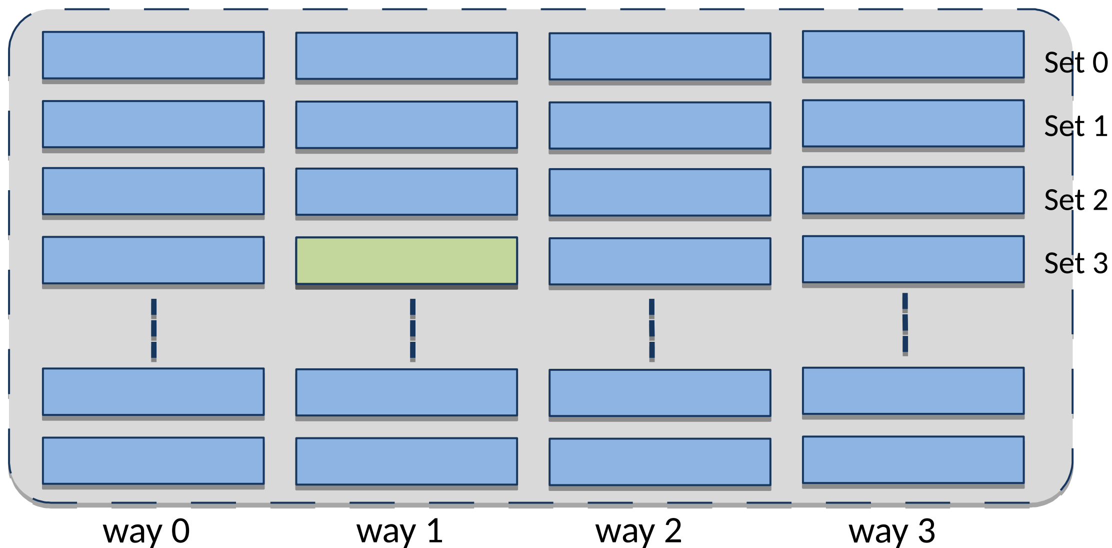
Virtual address space of process



Not normal Circumstances

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

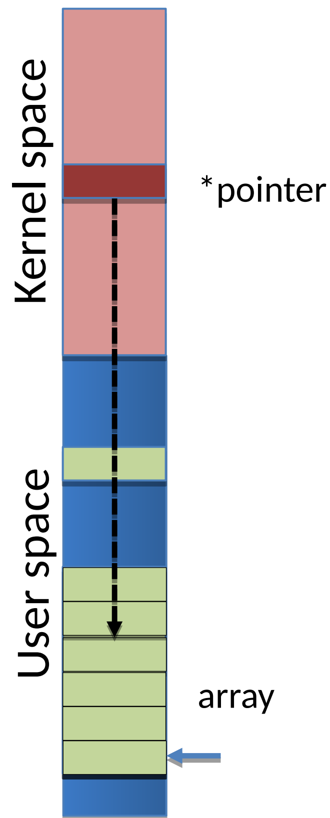
Cache Memory



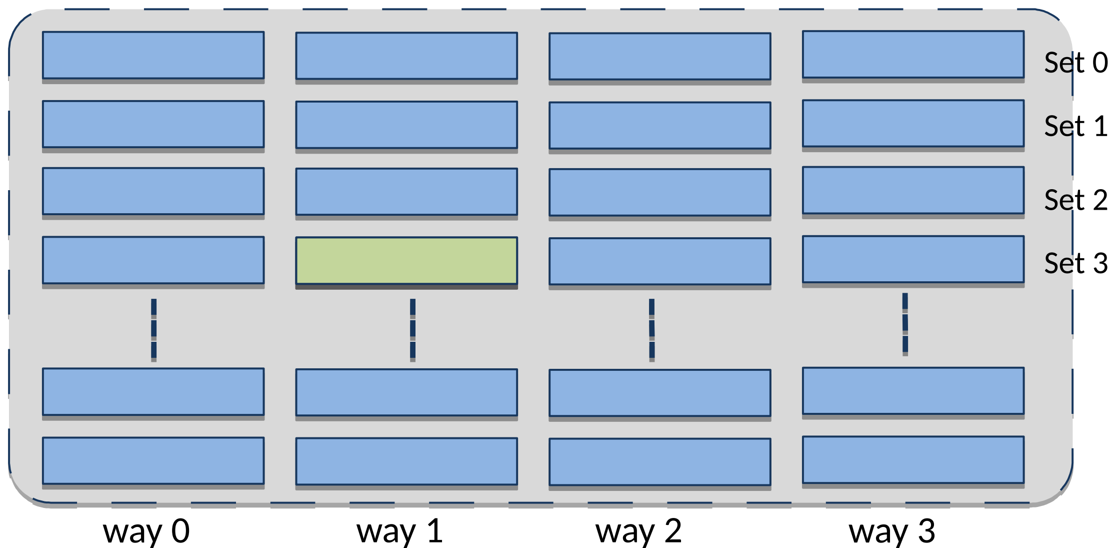
Meltdown

Not normal Circumstances

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

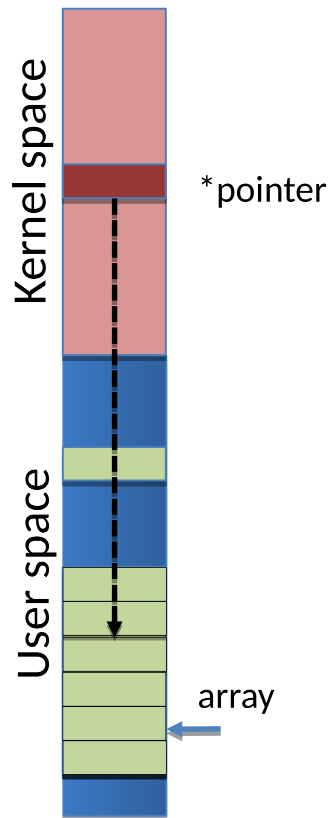


cache miss



Meltdown

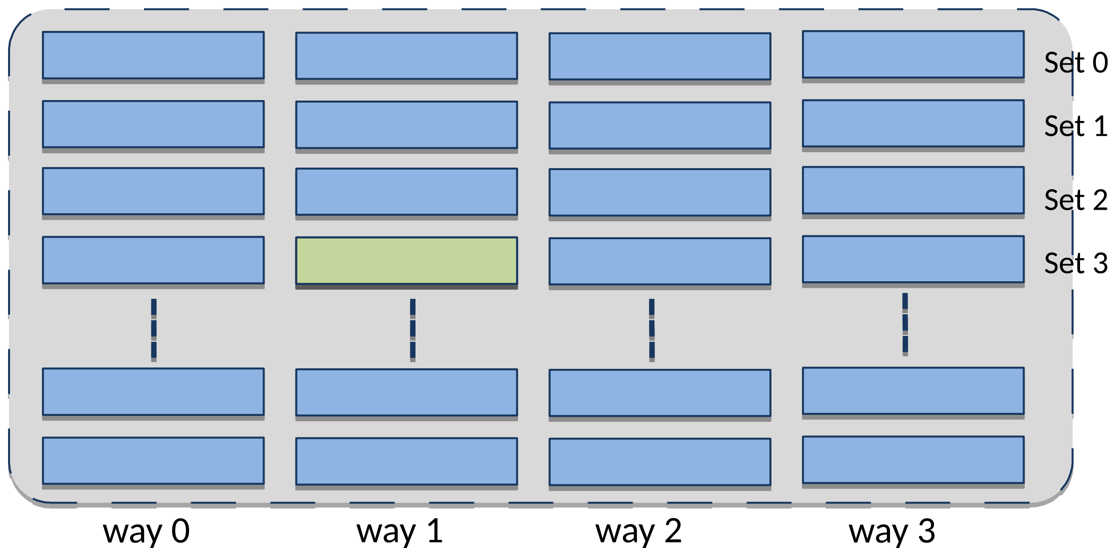
Virtual address space of process



Not normal Circumstances

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

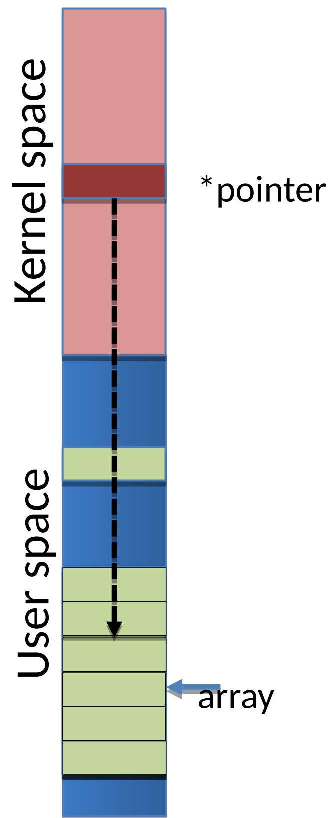
Cache Memory



cache miss

Meltdown

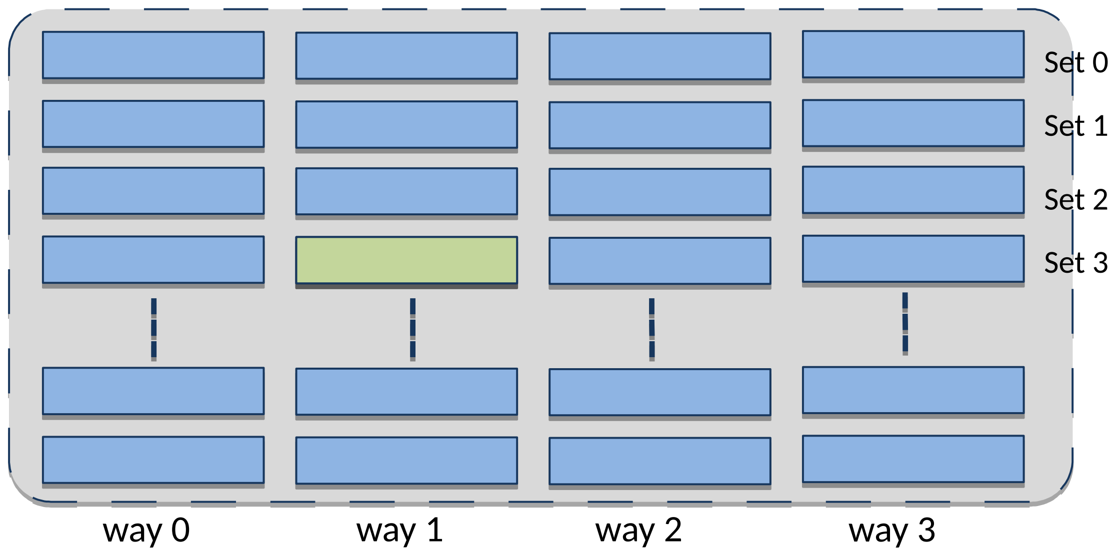
Virtual address space of process



Not normal Circumstances

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

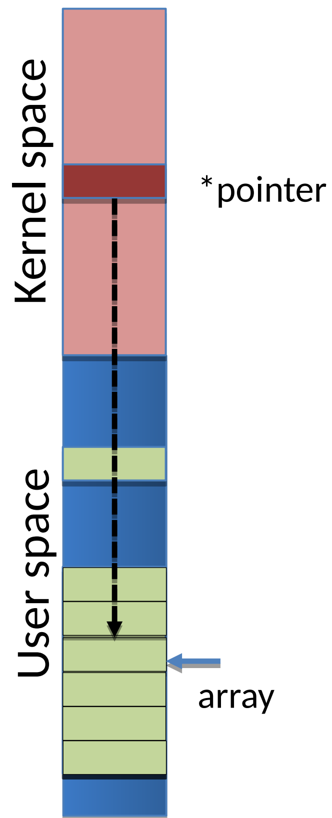
Cache Memory



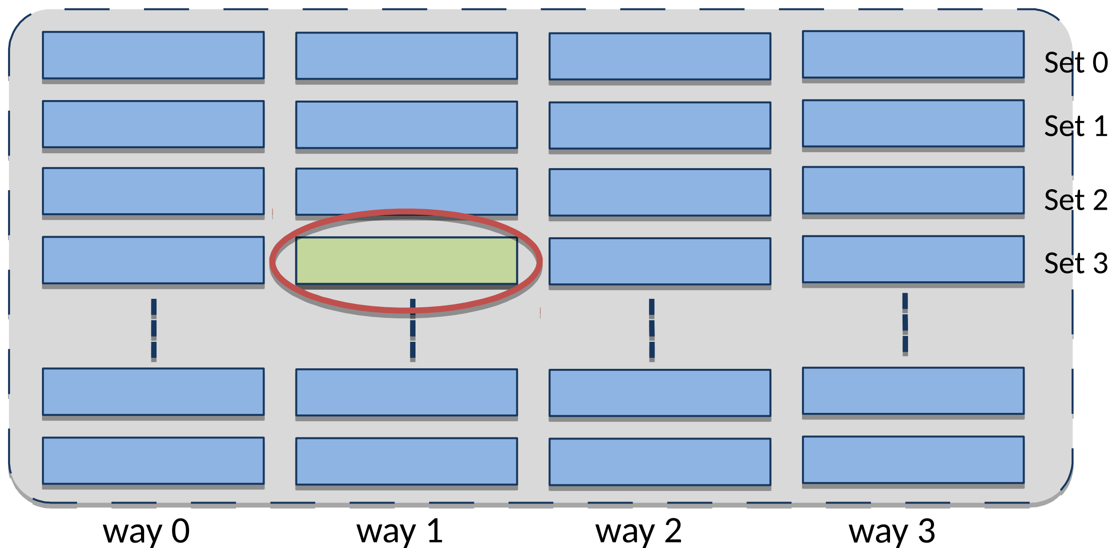
cache miss

Meltdown

Not normal Circumstances



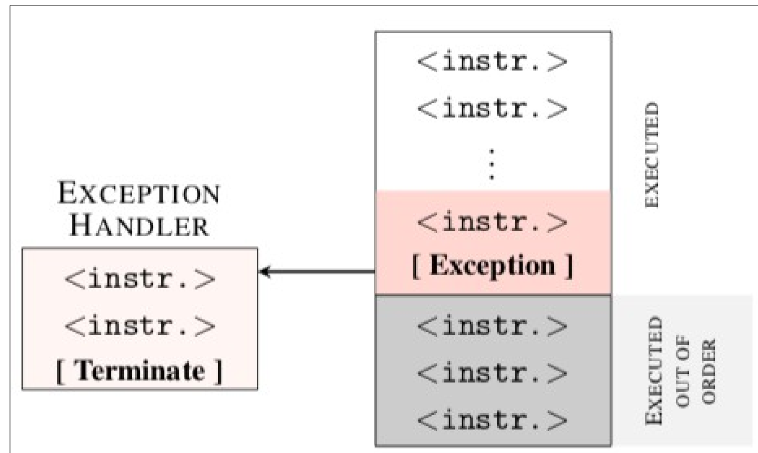

```
i = *pointer
y = array[i * 256]
```

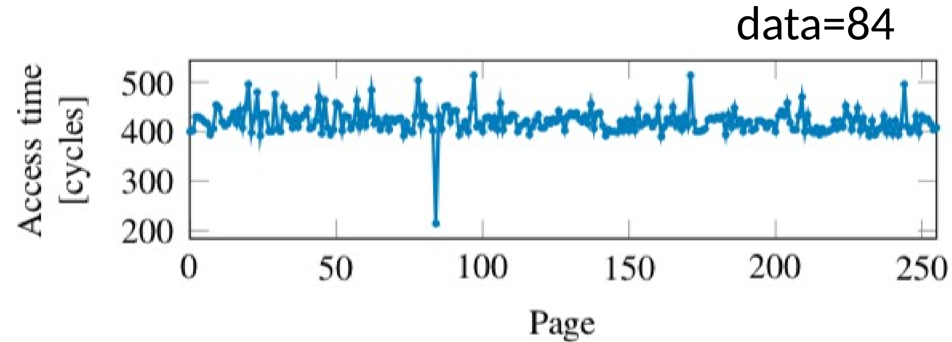


cache hit

Speculative Execution
and Micro-architectural State

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

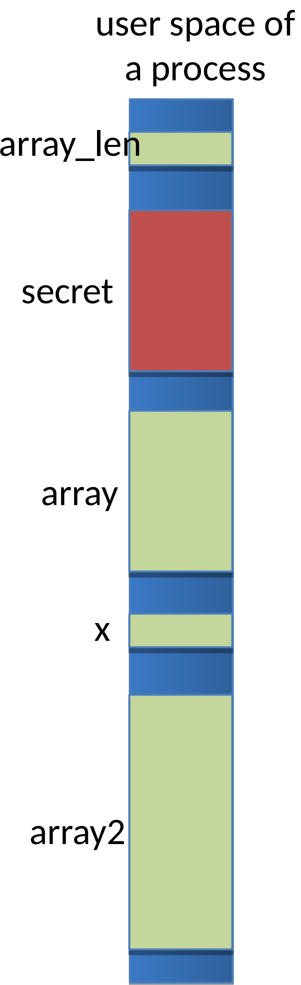




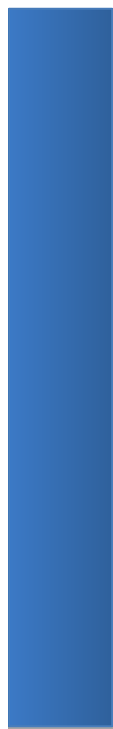
Spectre

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

Spectre (variant 1)

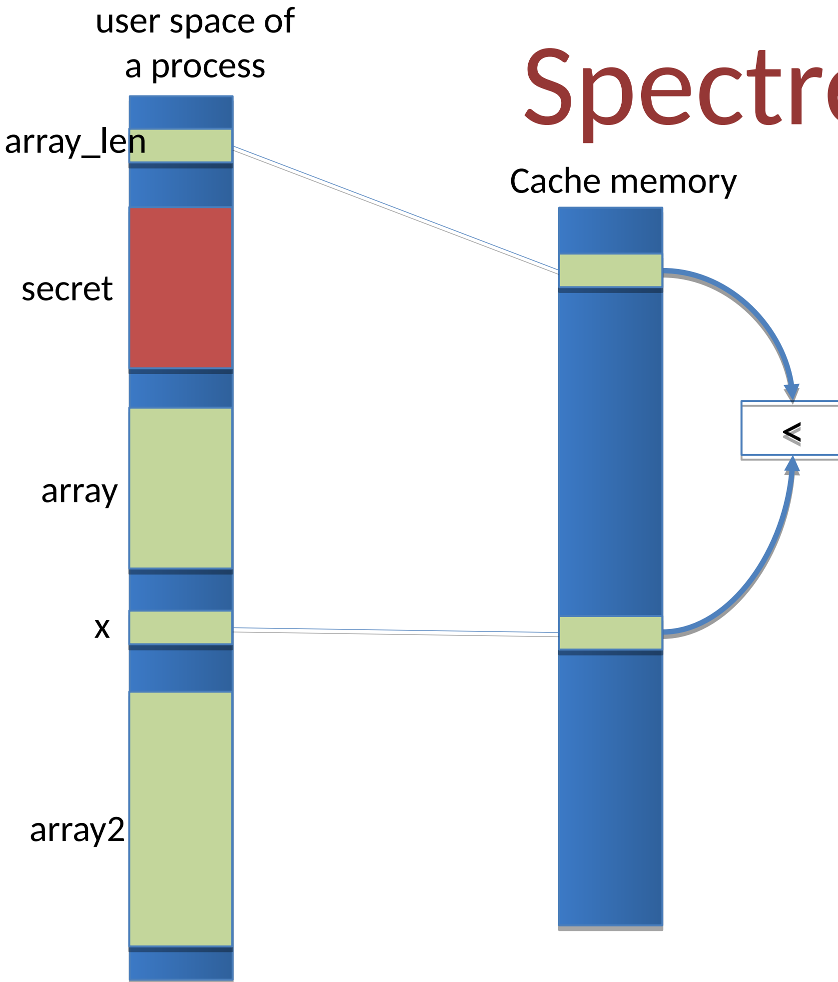


Cache memory



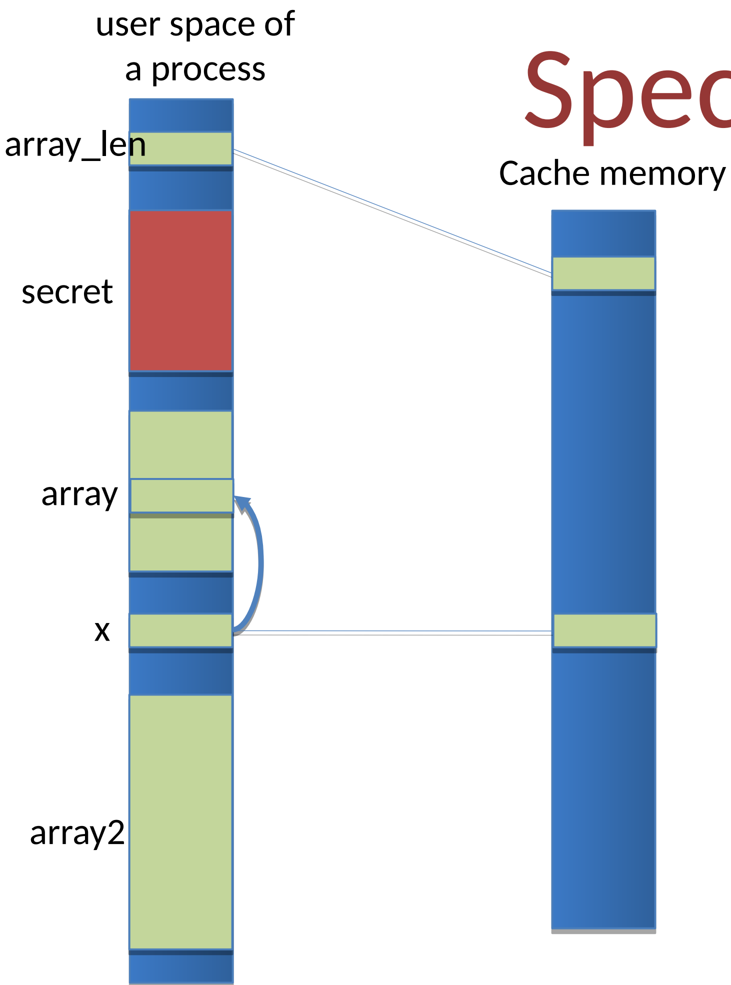
```
if (x < array_len){
    i = array[x];
    y = array2[i * 256];
}
```

Spectre (variant 1)




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

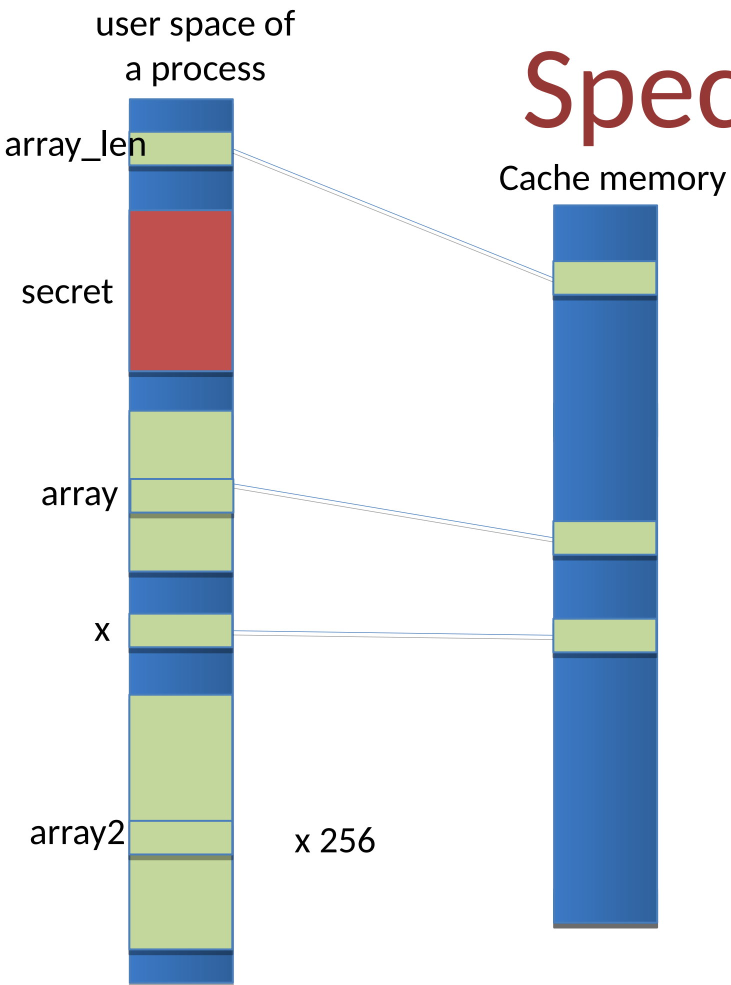
Spectre (variant 1)



Normal Behavior

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

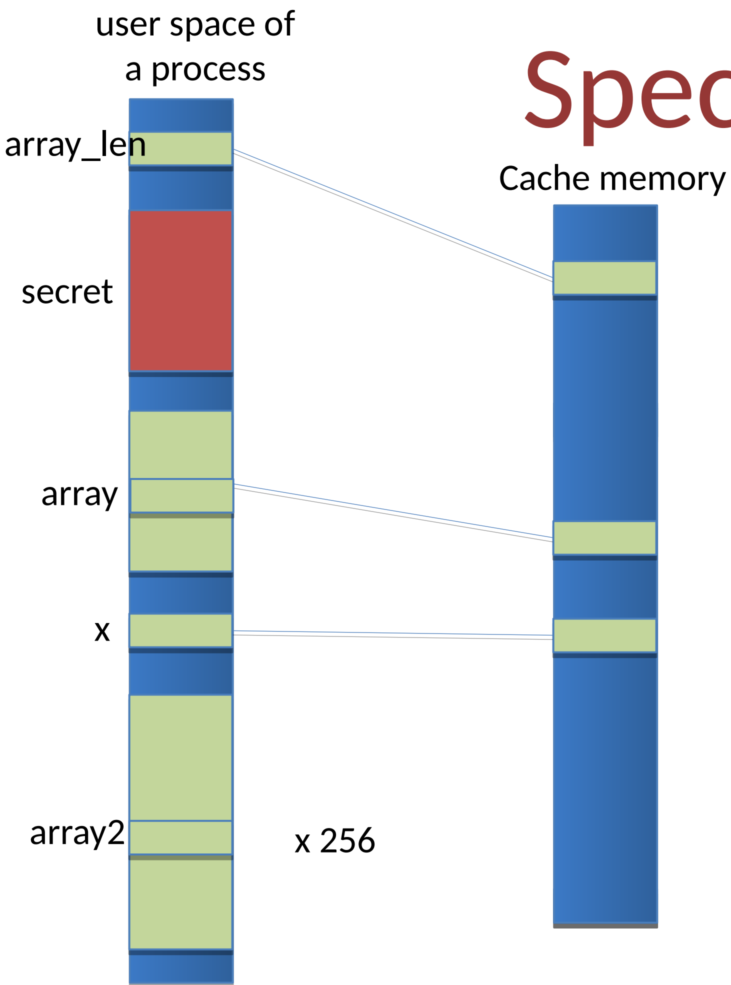
Spectre (variant 1)



Normal Behavior

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

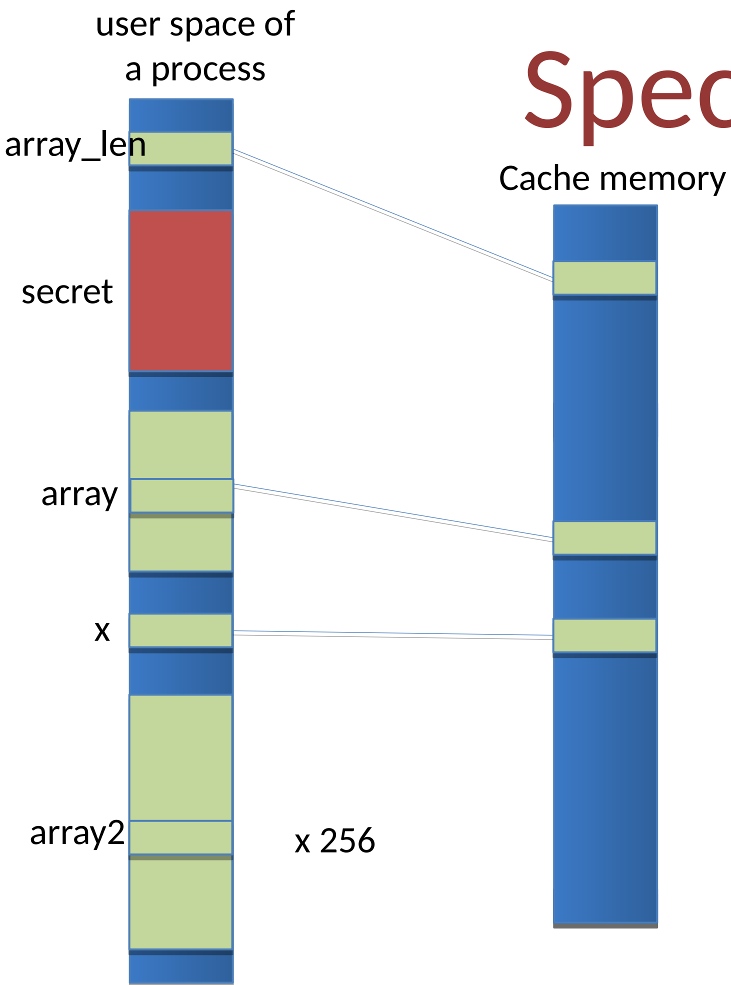
Spectre (variant 1)



Normal Behavior

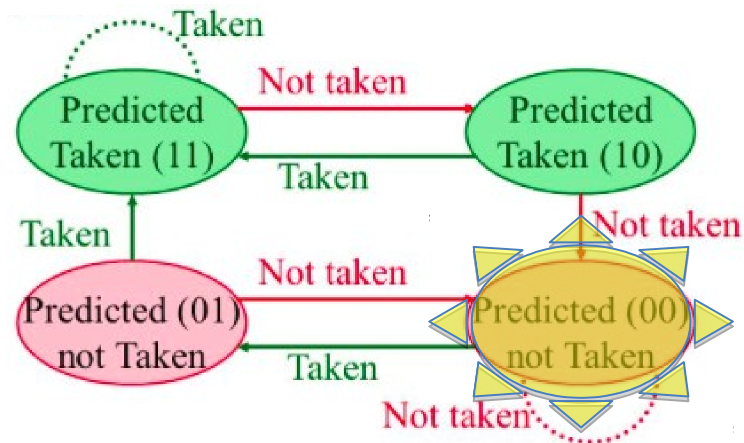
```
if (x < array_len){
    i = array[x];
    y = array2[i * 256];
}
```

Spectre (variant 1)

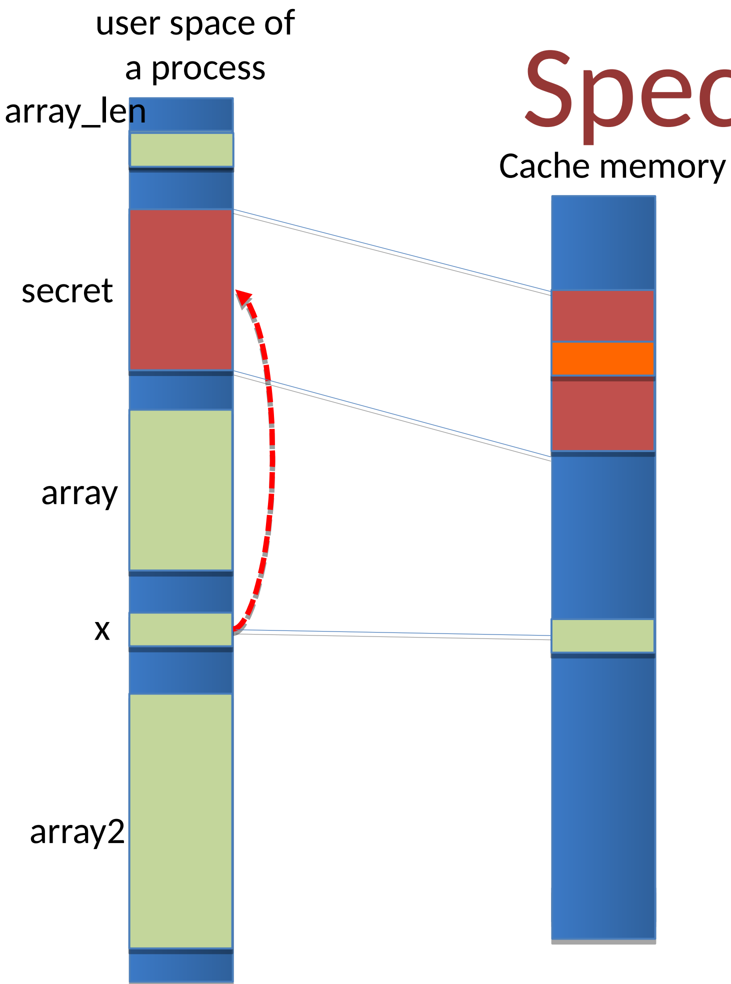


Normal Behavior

```
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

array_le

secret

array

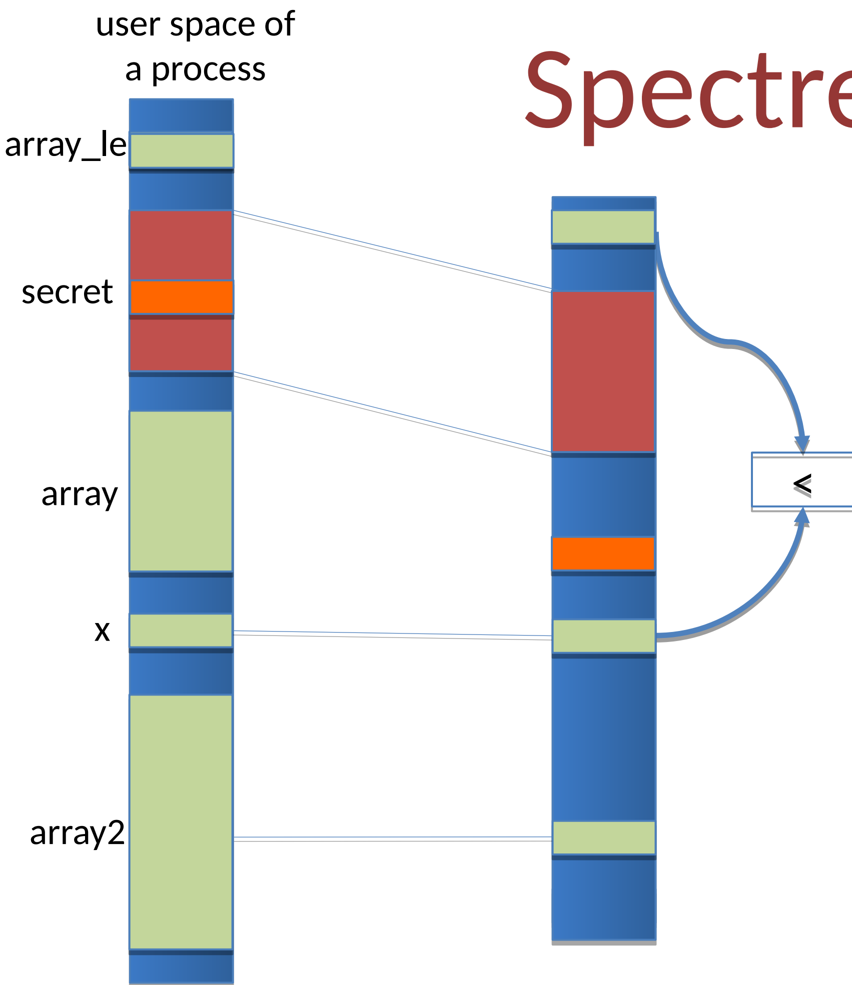
x

array2

<

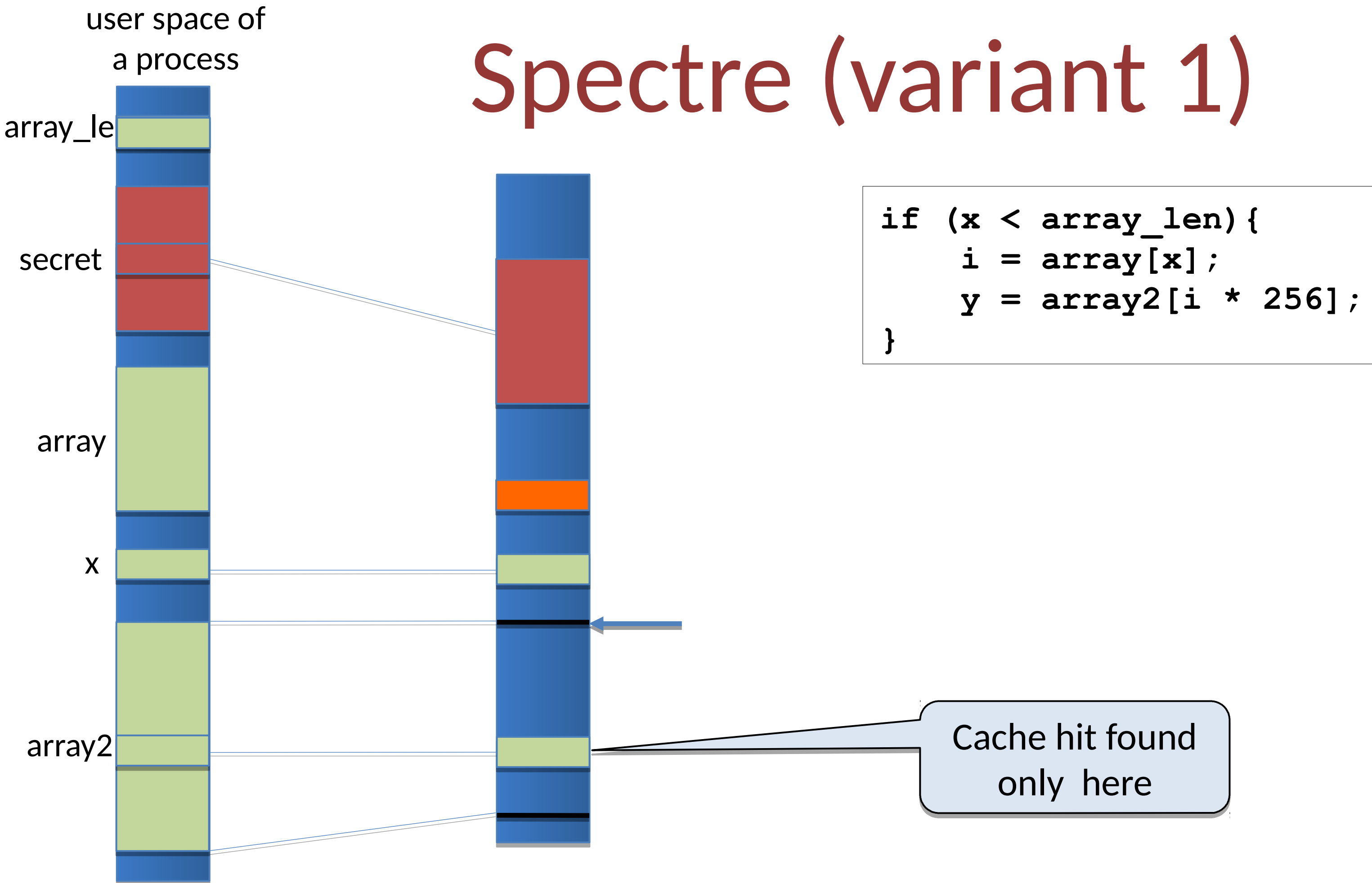
```
if (x < array_len){
    i = array[x];
    y = array2[i * 256];
}
```

Spectre (variant 1)

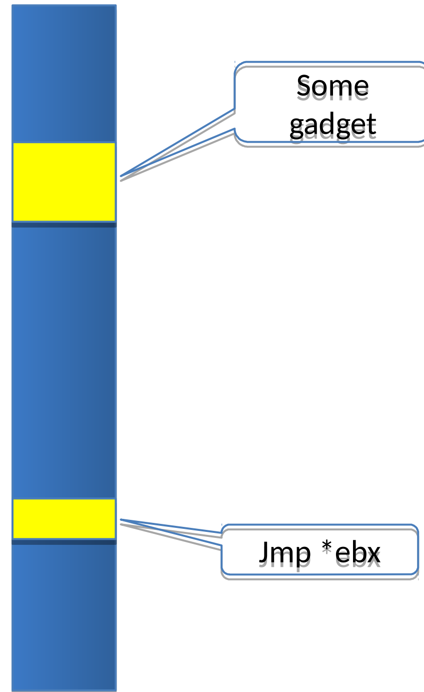



```
if (x < array_len){
    i = array[x];
    y = array2[i * 256];
}
```

Misprediction!

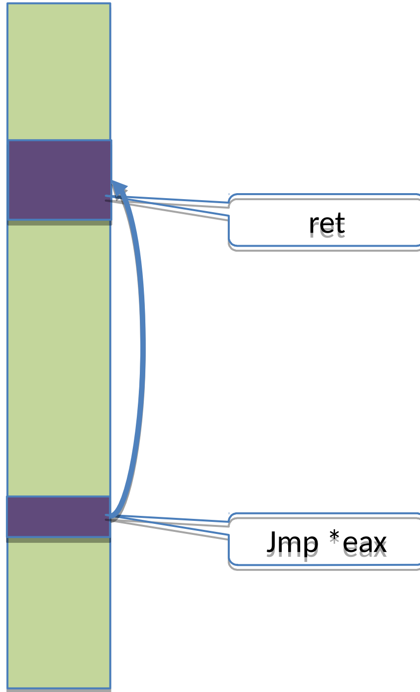

user space of
a process
Spectre (variant 1)
array_le
secret
array
x
array2
if (x < array_len){
i = array[x];
y = array2[i * 256];
}
Cache hit found
only here

Spectre (variant 2)

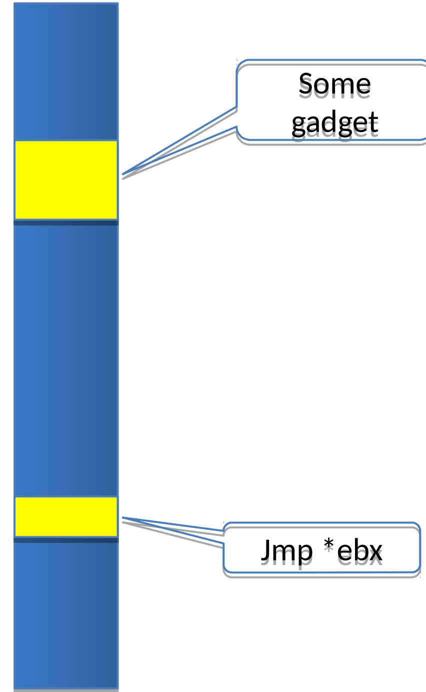


Spectre (variant 2)

Attacker's address space

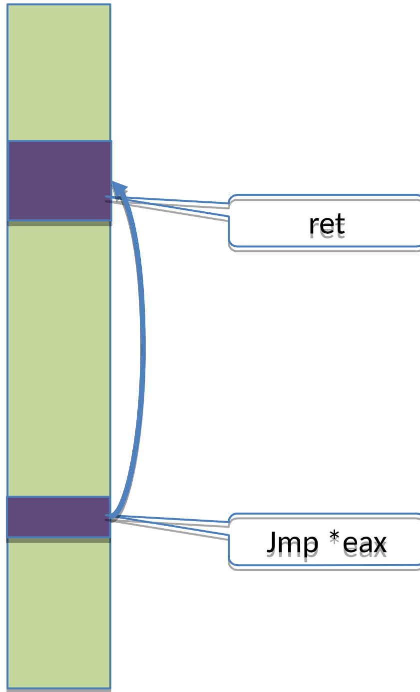


Victim's address space

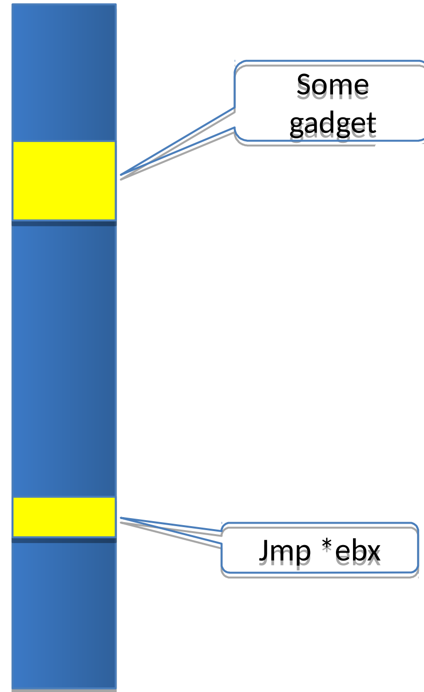


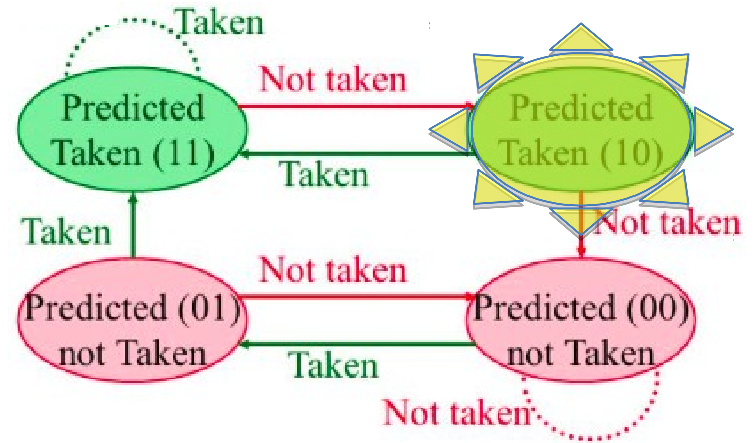
Spectre (variant 2)

Attacker's address space

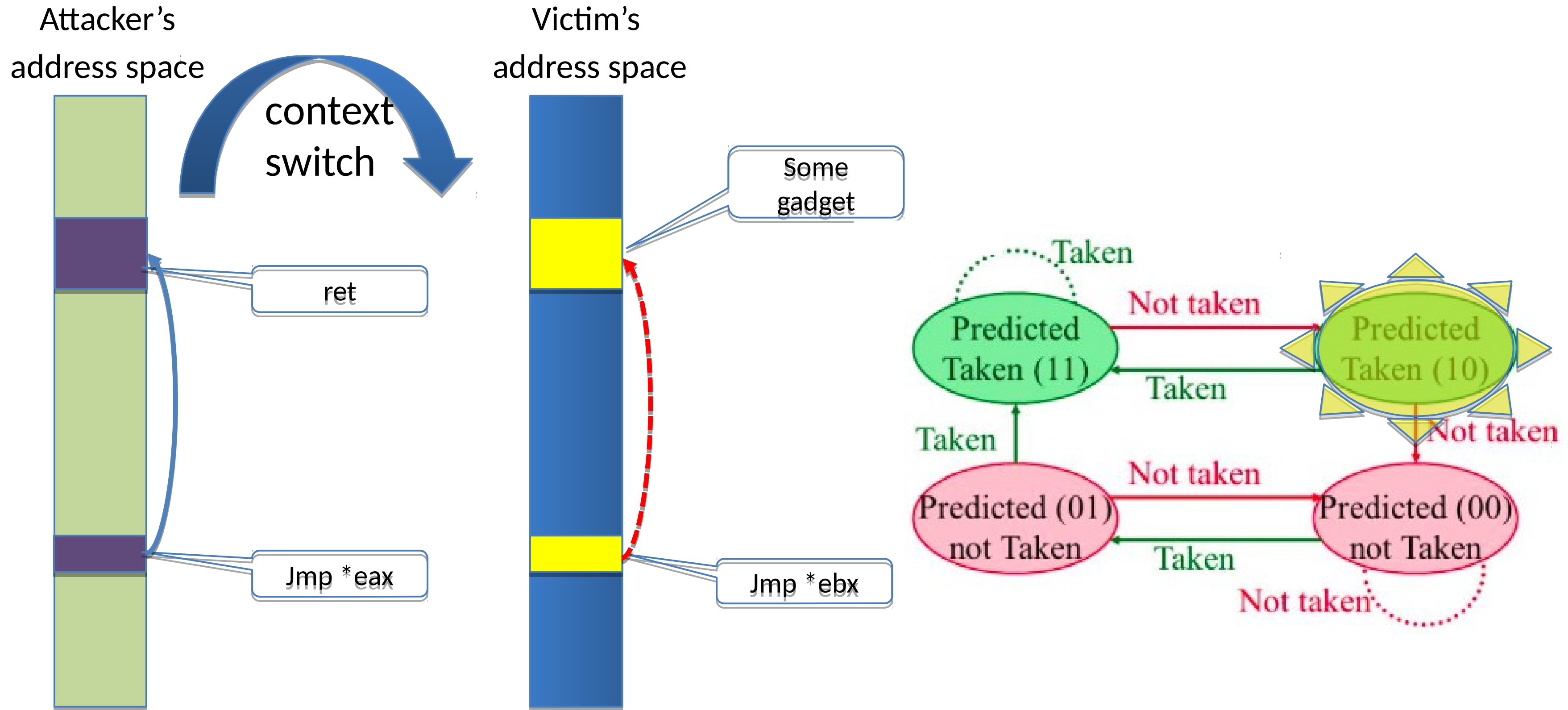


Victim's address space



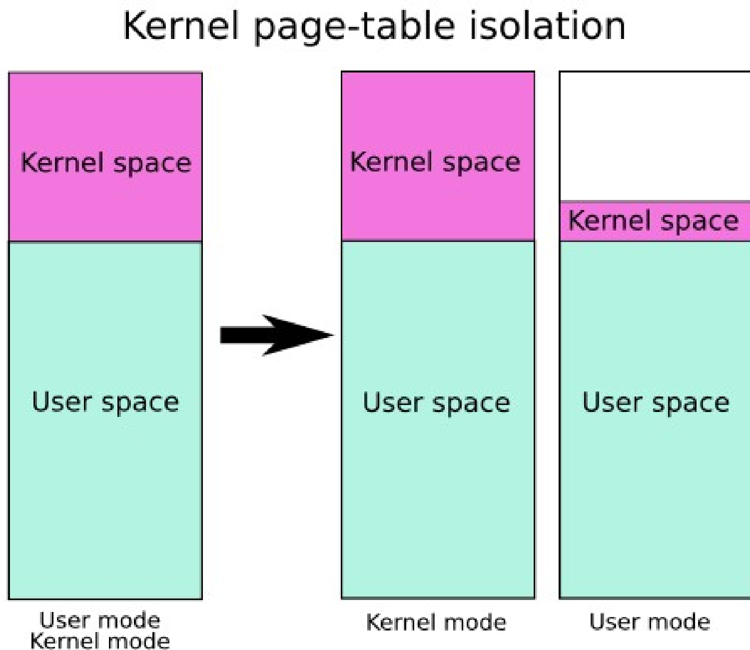


Spectre (variant 2)



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

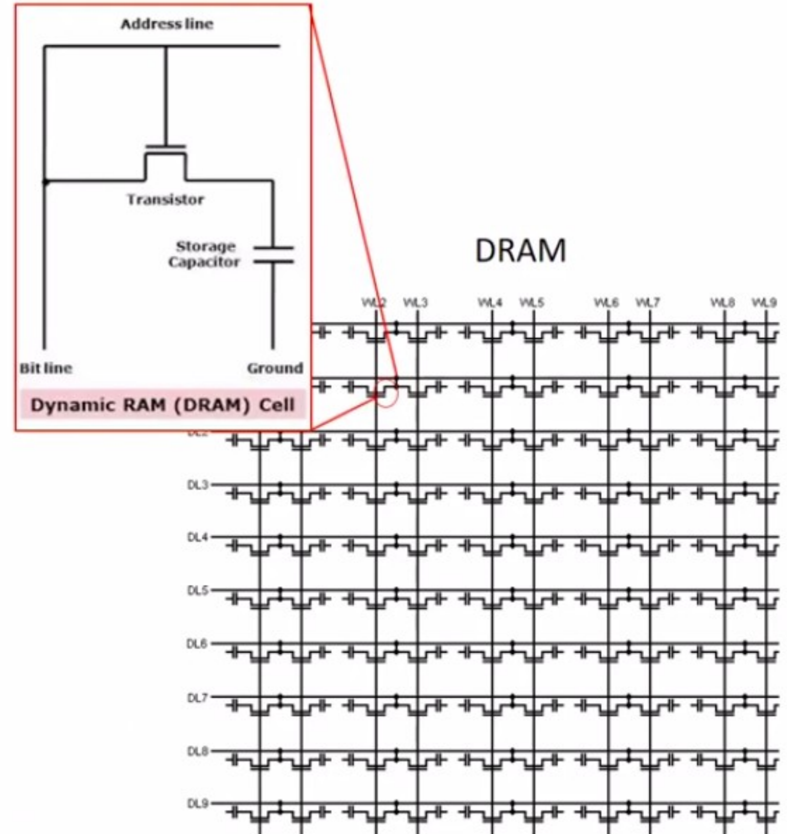
Countermeasures

- 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

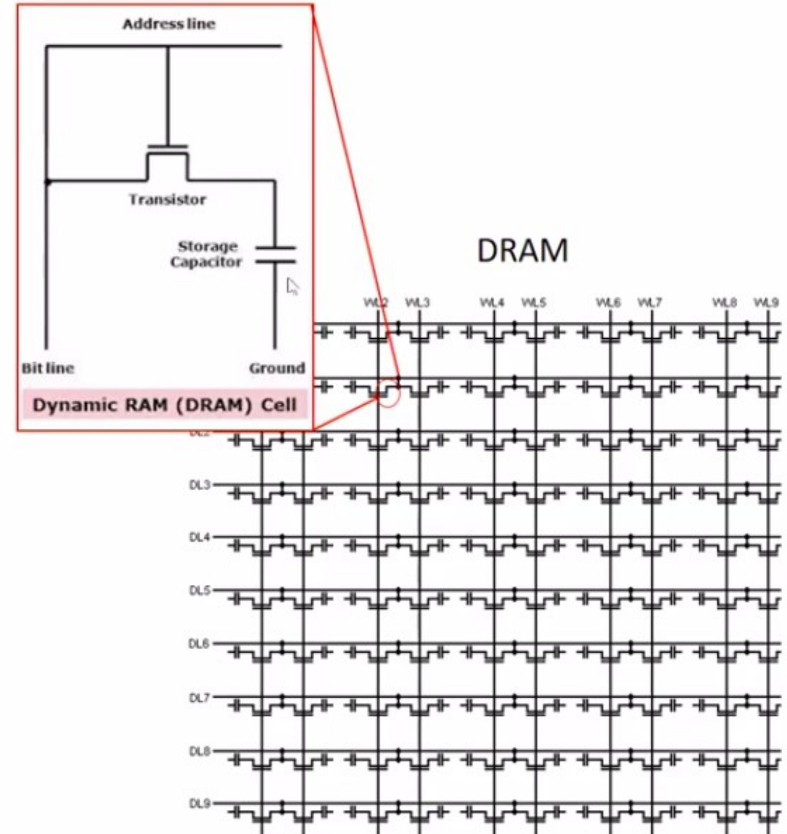
Some slides borrowed from Prof. Onur Mutlu's talk at DATE
Invited talk | March 30, 2017

DRAM

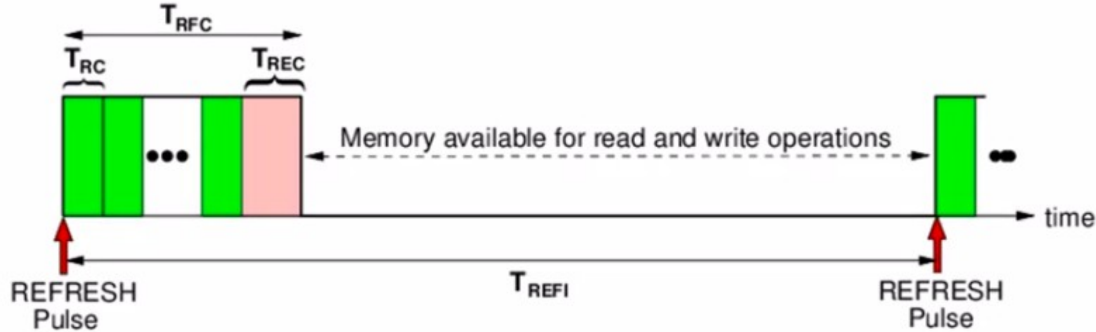


DRAM

- ❑ DRAM stores charge in a **capacitor** (charge-based memory)
- ❑ Capacitor must be large enough for reliable sensing



DRAM Refresh Cycles

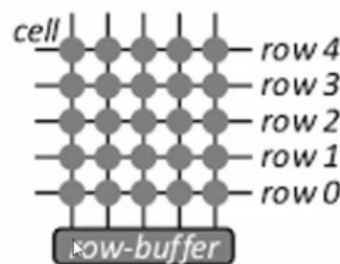


- ❑ As time passes, the charges in the memory cells (capacitors) leak away, so without being refreshed the stored data would eventually be lost.

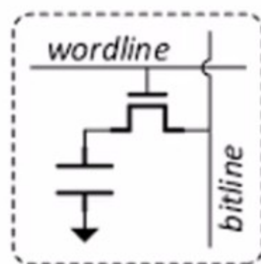
To prevent this, external circuitry periodically reads each cell and rewrites it, restoring the charge on the capacitor to its original level.

- ❑ Refresh cycles are in orders of milliseconds though, consume some memory bandwidth

DRAM Cells arrangement



a. Rows of cells

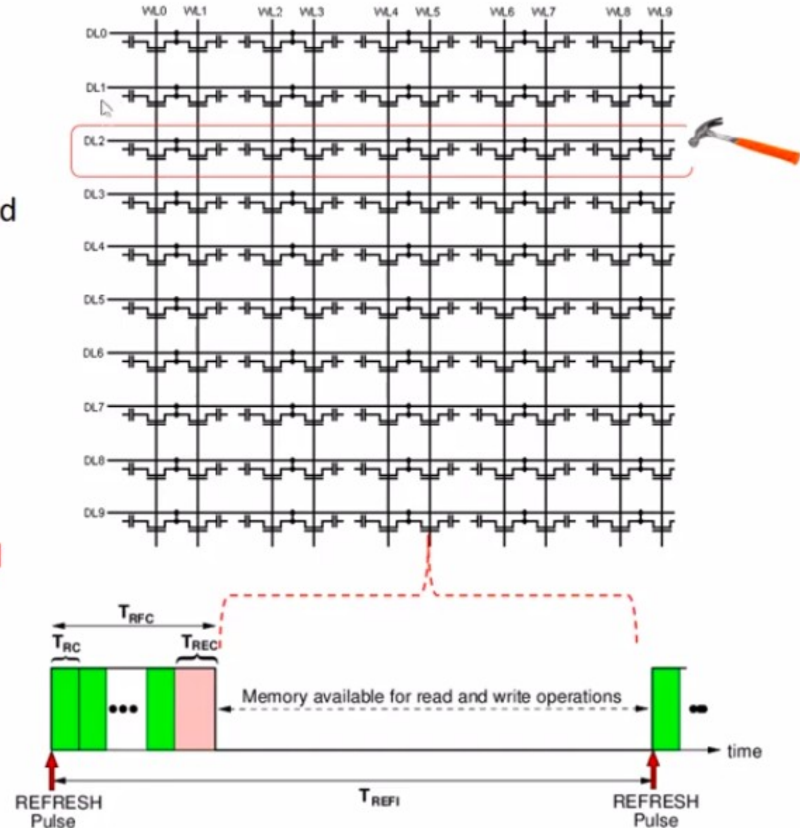


b. A single cell

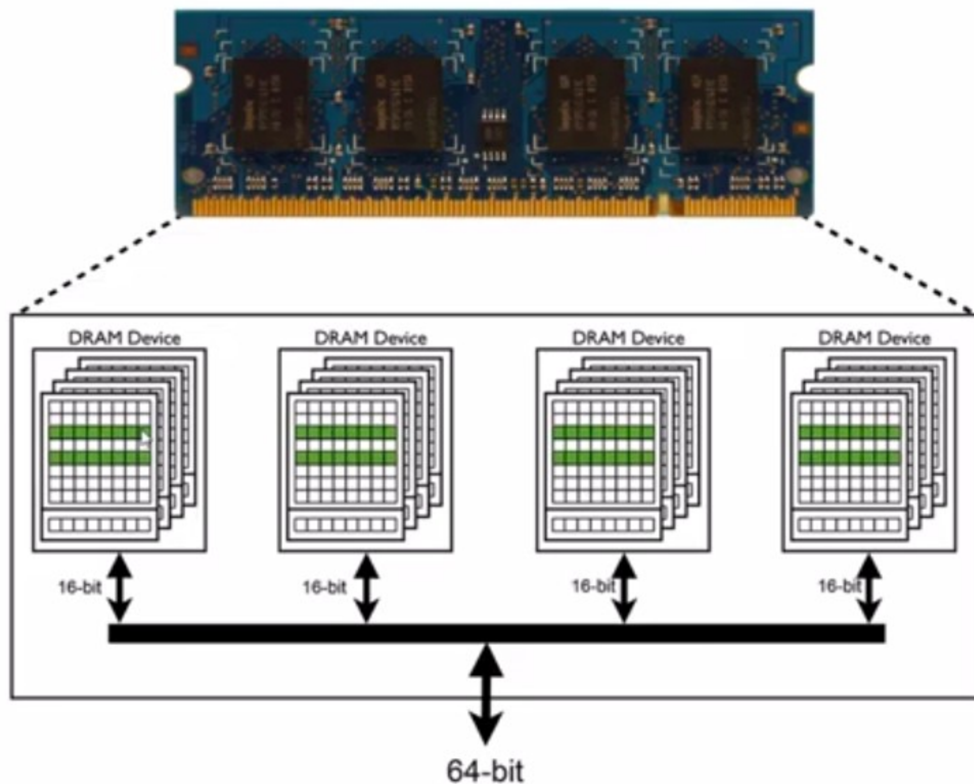
- Scaling beyond 40-35nm (2013) is challenging [ITRS, 2009]
- With reduction in transistor sizes, DRAM cells became smaller
- As DRAM cells became smaller, the space between two such cells reduces
- Closer the two charged bodies, higher the electromagnetic interference

Rowhammer

- ❑ As we keep accessing a particular row repeatedly during a refresh interval, neighboring cells have been found to leak charge at a faster rate due to electromagnetic coupling.
  - Toggling a row voltage briefly increases the voltage of adjacent rows
  - This slightly opens adjacent rows => Charge leakage
  - This leads to a decrease in their retention time.
- ❑ Thus during refresh, the corrupted data will be read and written back again to the DRAM cell.

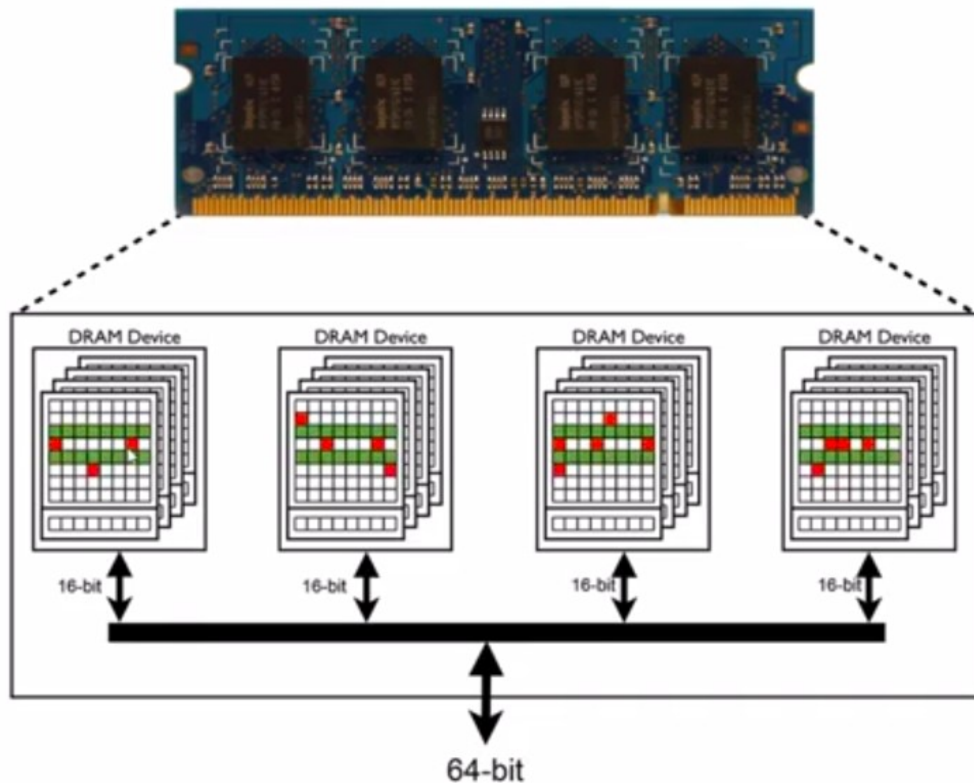


Rowhammer



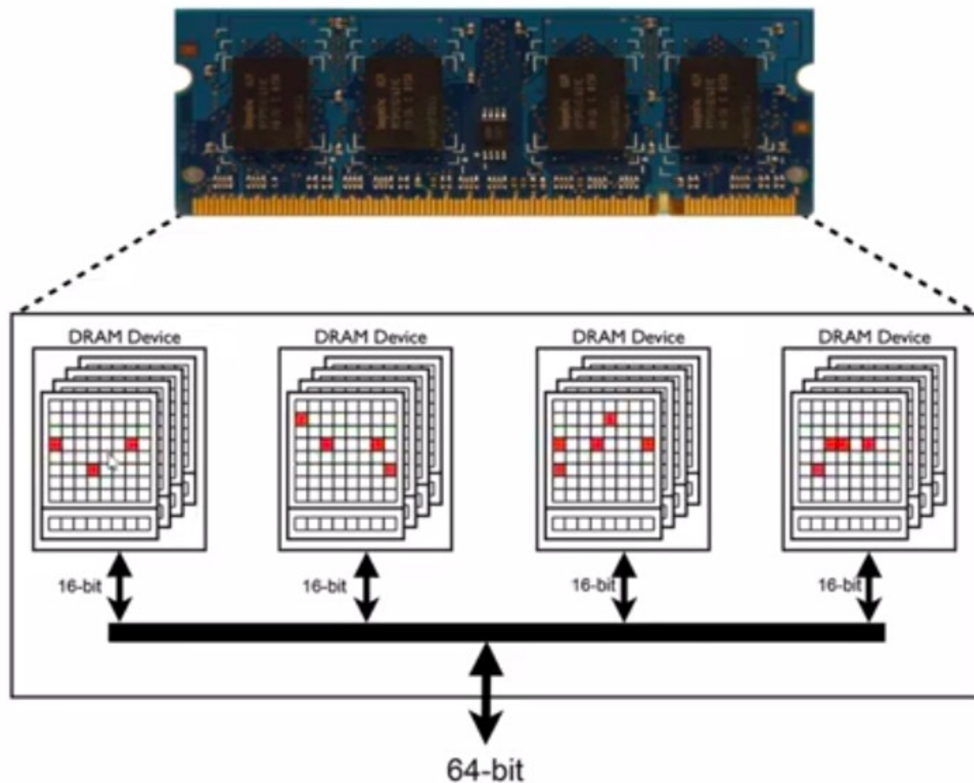
http://apt.cs.manchester.ac.uk/projects/ARMOR/RowHammer/

Rowhammer



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

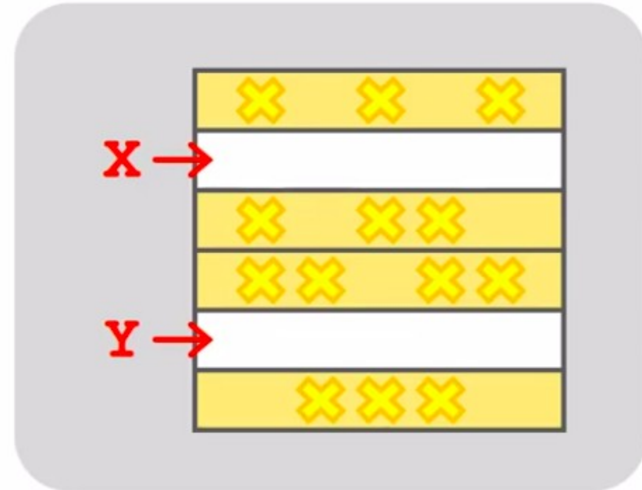
Rowhammer



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

A Simple program

```
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

Security implications

One bit can make a lot of difference

Solutions

- 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 Day !!