

Microarchitectural Attacks:

From the Basics to Arbitrary Read and Write Primitives without any Software Bugs

Daniel Gruss

June 19, 2018

Graz University of Technology











1337 4242

FOOD CACHE

Revolutionary concept!

Store your food at home,
never go to the grocery store
during cooking.

Can store **ALL** kinds of food.

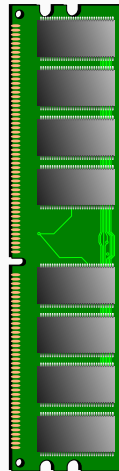
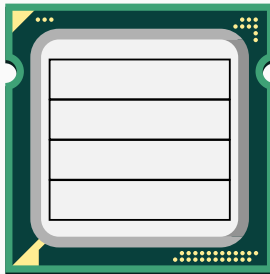
ONLY TODAY INSTEAD OF ~~\$1,300~~

\$1,299

ORDER VIA PHONE: +555 12345

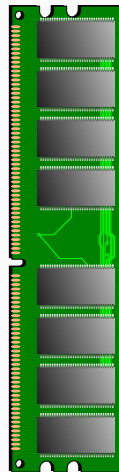
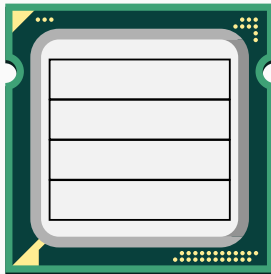


```
printf("%d", i);  
printf("%d", i);
```



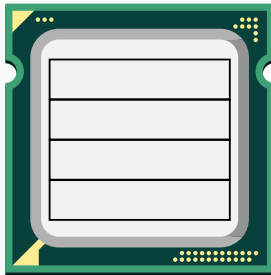

```
printf("%d", i);  
printf("%d", i);
```

Cache miss

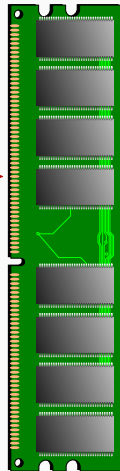


```
printf("%d", i);  
printf("%d", i);
```

Cache miss

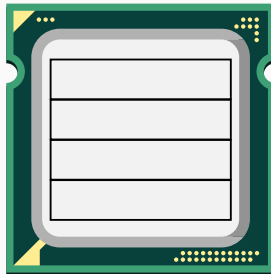


Request



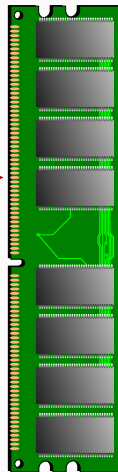
```
printf("%d", i);  
printf("%d", i);
```

Cache miss



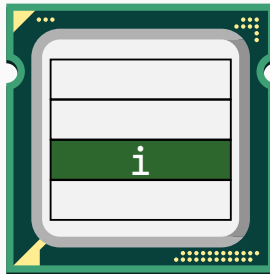
Request

Response



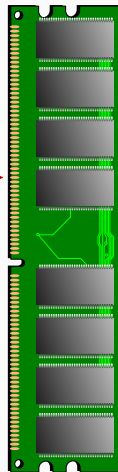
```
printf("%d", i);  
printf("%d", i);
```

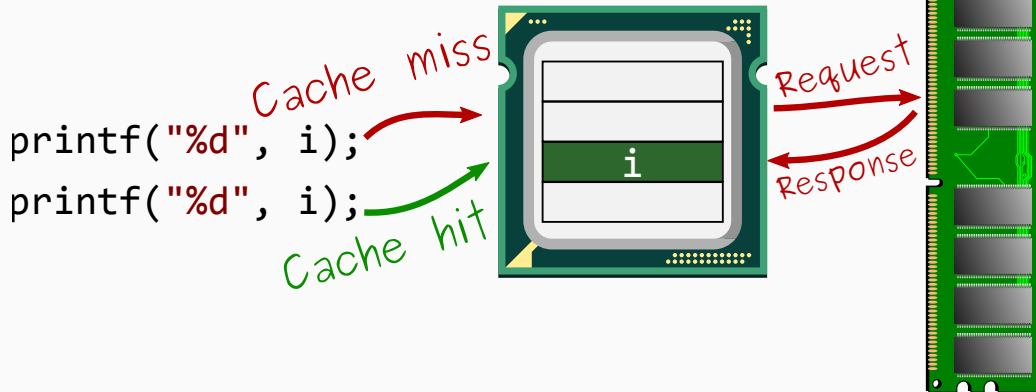
Cache miss

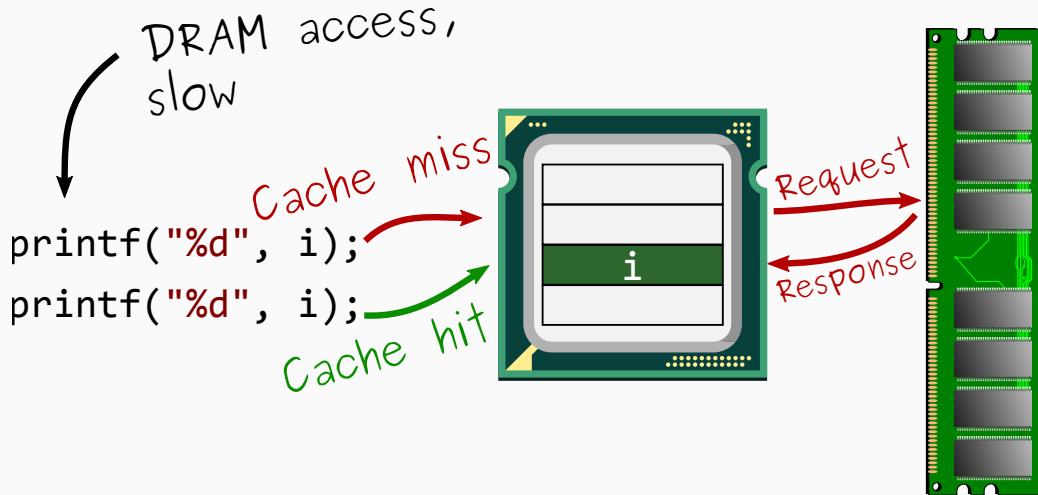


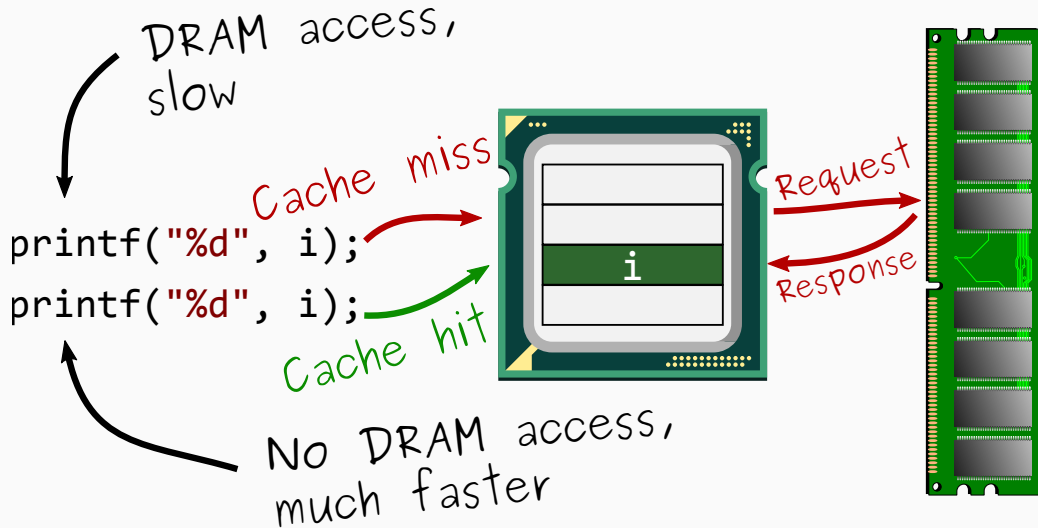
Request

Response





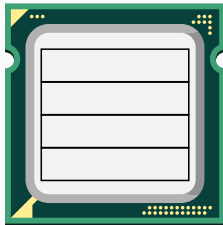




Shared Memory

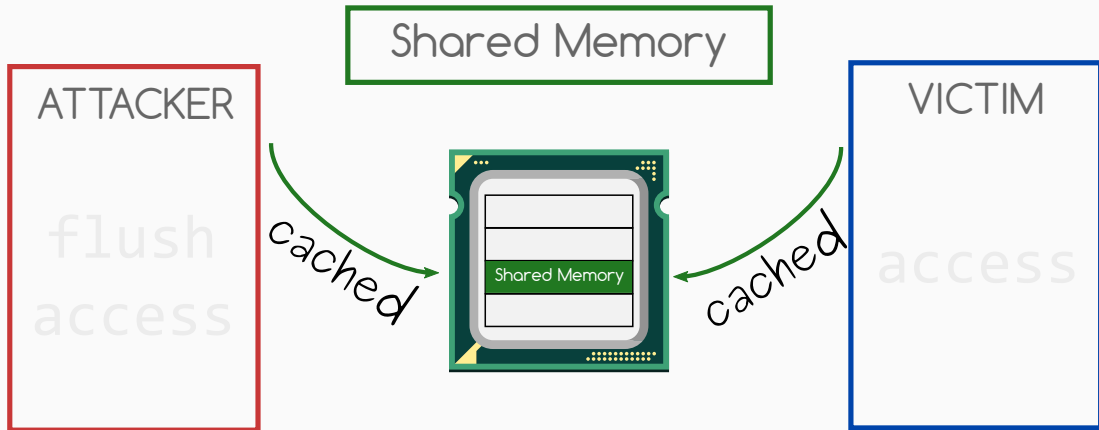
ATTACKER

flush
access



VICTIM

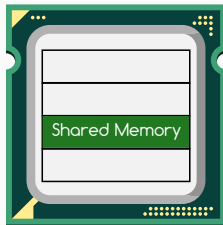
access



Shared Memory

ATTACKER

flush
access



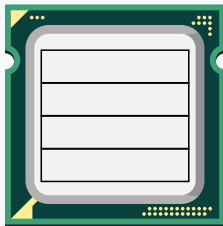
VICTIM

access

Shared Memory

ATTACKER

flush
access



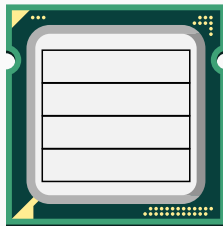
VICTIM

access

Shared Memory

ATTACKER

flush
access



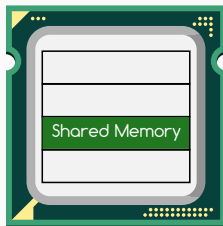
VICTIM

access

Shared Memory

ATTACKER

flush
access



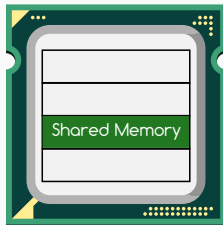
VICTIM

access

Shared Memory

ATTACKER

flush
access



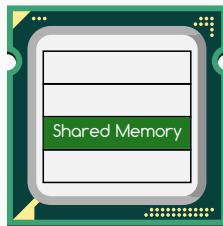
VICTIM

access

Shared Memory

ATTACKER

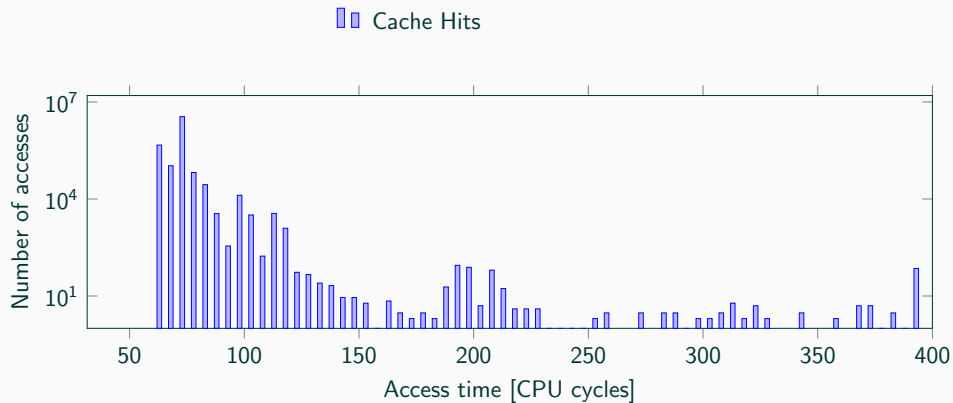
flush
access

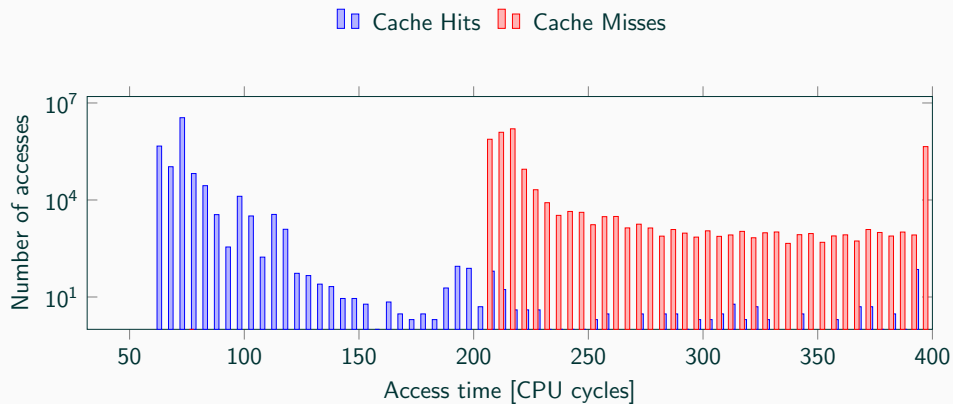


VICTIM

access

fast if victim accessed data,
slow otherwise





Terminal

File Edit View Search Terminal Help

```
% sleep 2; ./spy 300 7f05140a4000-7f051417b000 r-xp 0x20000 08:02 26
8050 /usr/lib/x86_64-linux-gnu/gedit/libgedit.so
```

gnome-terminal

gnome-terminal

Terminal

File Edit View Search Terminal Help

```
shark% ./spy
```

gnome-terminal

Open +

Untitled Document 1

Save

1

I

Plain Text Tab Width: 2 Ln 1, Col 1 INS





Back to Work

*7. Serve with cooked
and peeled potatoes*





Wait for an hour





Wait for an hour

LATENCY

1. *Wash and cut
vegetables*

2. *Pick the basil leaves
and set aside*

3. *Heat 2 tablespoons of
oil in a pan*

4. *Fry vegetables until
golden and softened*



Dependency

1. Wash and cut vegetables

2. Pick the basil leaves and set aside

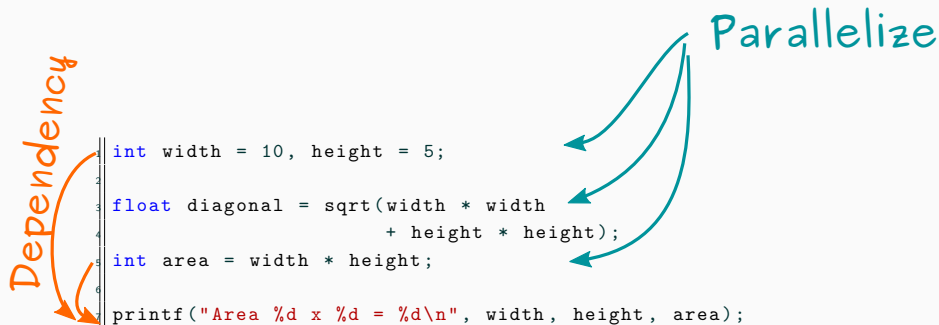
3. Heat 2 tablespoons of oil in a pan

4. Fry vegetables until golden and softened

Parallelize



```
1 int width = 10, height = 5;
2
3 float diagonal = sqrt(width * width
4                       + height * height);
5 int area = width * height;
6
7 printf("Area %d x %d = %d\n", width, height, area);
```



```
1 | char data = *(char*)0xffffffff81a000e0;  
2 | printf("%c\n", data);
```



```
1 | char data = *(char*)0xffffffff81a000e0;  
2 | printf("%c\n", data);
```



```
1 segfault at ffffffff81a000e0 ip 0000000000400535  
2          sp 00007ffce4a80610 error 5 in reader
```

```
1 char data = *(char*)0xffffffff81a000e0;  
2 printf("%c\n", data);
```



```
1 segfault at ffffffff81a000e0 ip 0000000000400535  
2          sp 00007ffce4a80610 error 5 in reader
```

- Kernel addresses are not accessible

```
1 char data = *(char*)0xffffffff81a000e0;  
2 printf("%c\n", data);
```



```
1 segfault at ffffffff81a000e0 ip 0000000000400535  
2          sp 00007ffce4a80610 error 5 in reader
```

- Kernel addresses are not accessible
- Are privilege checks also done when executing instructions out of order?

- Adapted code



```
1 | *(volatile char*)0;  
2 | array[84 * 4096] = 0; // unreachable
```



- Adapted code

```
1 | *(volatile char*)0;  
2 | array[84 * 4096] = 0; // unreachable
```

- Static code analyzer is not happy

```
1 | warning: Dereference of null pointer  
2 |     *(volatile char*)0;
```



- Flush+Reload over all pages of the array



- “Unreachable” code line was actually executed



- Flush+Reload over all pages of the array



- “Unreachable” code line was actually executed
- Exception was only thrown afterwards



- Combine the two things

```
1 char data = *(char*)0xffffffff81a000e0;  
2 array[data * 4096] = 0;
```



- Combine the two things

```
1 char data = *(char*)0xffffffff81a000e0;  
2 array[data * 4096] = 0;
```

- Then check whether any part of array is cached



- Flush+Reload over all pages of the array



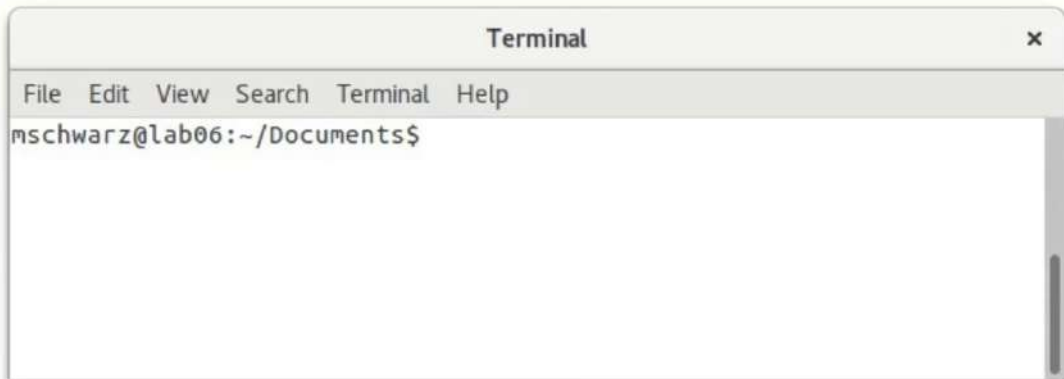
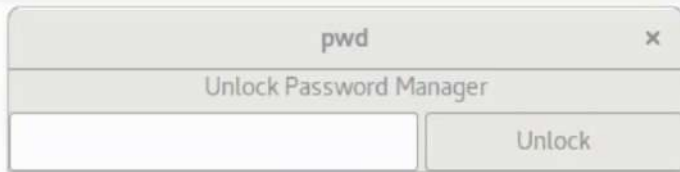
- Index of cache hit reveals data

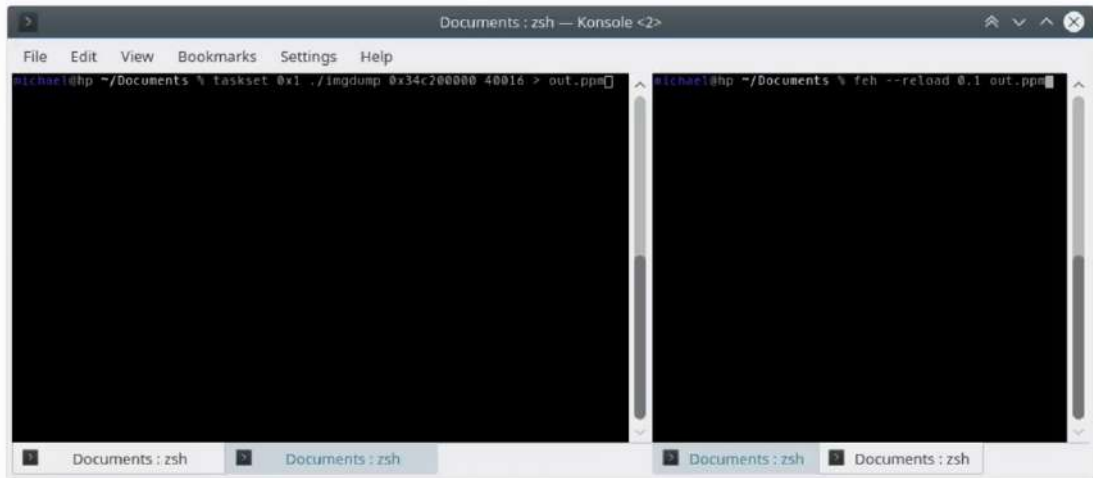


- Flush+Reload over all pages of the array



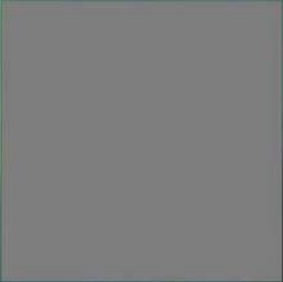
- Index of cache hit reveals data
- Permission check is in some cases not fast enough





A man and a woman are shown in a close-up, looking off-camera with serious expressions. The woman is on the left, and the man is on the right. The lighting is dim and blue-toned, suggesting a nighttime or indoor setting with artificial light. The man's ear is highlighted with a red glow.

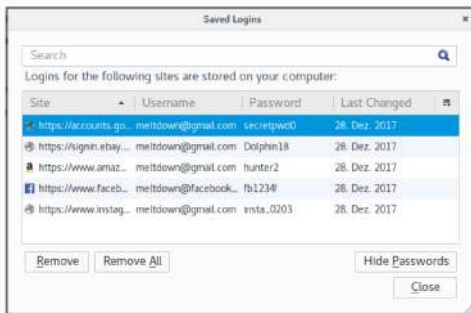
CAN YOU
ENHANCE THAT



```
meltdown@meltdown ~/ppm2 % taskset 1 ./imgdump 0x375a00000 14919 > output.flif  
Reading from 0xffff880375a00000
```



I

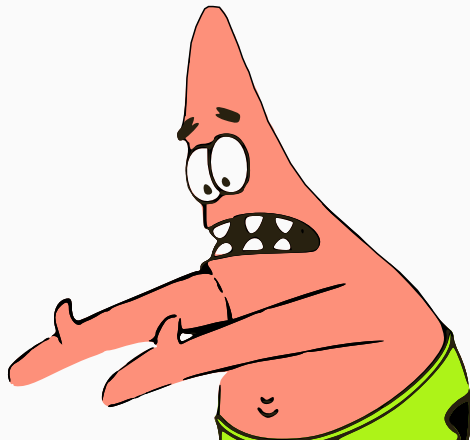


```
f94b7690: e5 e5 e5 e5 e5 e5 e5 e5 e5 e5 e5 e5 | .....|
f94b76a0: e5 e5 e5 e5 e5 e5 e5 e5 e5 e5 e5 e5 | .....|
f94b76b0: 70 52 b8 6b 96 7f XX XX XX XX XX XX XX XX |pR.k.....|
f94b76c0: 09 XX XX XX XX XX XX XX XX XX XX XX XX XX | .....|
f94b76d0: XX XX XX XX XX XX XX XX XX XX XX XX XX XX | .....|
f94b76e0: XX XX XX XX XX XX XX XX XX XX XX XX XX 81 | .....|
f94b76f0: 12 XX e0 81 19 XX e0 81 44 6f 6c 70 68 69 6e 31 |.....Dolphin1|
f94b7700: 38 e5 e5 e5 e5 e5 e5 e5 e5 e5 e5 e5 |8.....|
f94b7710: 70 52 b8 6b 96 7f XX XX XX XX XX XX XX XX XX |pR.k.....|
f94b7720: XX XX XX XX XX XX XX XX XX XX XX XX XX XX | .....|
f94b7730: XX XX XX XX 4a XX XX XX XX XX XX XX XX XX |.....J.....|
f94b7740: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX | .....|
f94b7750: XX XX XX XX XX XX XX XX XX XX e0 81 69 6e 73 74 | .....inst|
f94b7760: 61 5f 30 32 30 33 e5 e5 e5 e5 e5 e5 e5 |a_0203.....|
f94b7770: 70 52 18 7d 28 7f XX XX XX XX XX XX XX XX XX |pR.}(.|
f94b7780: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX | .....|
f94b7790: XX XX XX XX 54 XX XX XX XX XX XX XX XX XX XX |.....T.....|
f94b77a0: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX | .....|
f94b77b0: XX XX XX XX XX XX XX XX XX XX XX XX XX 73 65 63 72 | .....secl|
f94b77c0: 65 74 70 77 64 30 e5 e5 e5 e5 e5 e5 e5 |etpwd0.....|
f94b77d0: 30 b4 18 7d 28 7f XX XX XX XX XX XX XX XX XX |0..}(.|
f94b77e0: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX | .....|
f94b77f0: XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX | .....|
f94b7800: e5 e5 e5 e5 e5 e5 e5 e5 e5 e5 e5 e5 | .....|
f94b7810: 68 74 74 70 73 3a 2f 2f 61 64 64 6f 6e 73 2e 63 |https://addons.c|
f94b7820: 64 6e 2e 6d 6f 7a 69 6c 6c 61 2e 6e 65 74 2f 75 |dn.mozilla.net/u|
f94b7830: 73 65 72 2d 6d 65 64 69 61 2f 61 64 64 6f 6e 5f |ser-media/addon_|
f94b7840: 69 63 6f 6e 73 2f 33 35 34 2f 33 35 34 33 39 39 |icons/354/354399|
f94b7850: 2d 36 34 2e 70 6e 67 3f 6d 6f 64 69 66 69 65 64 |-64.png?modified|
f94b7860: 3d 31 34 35 32 32 34 34 38 31 35 XX XX XX XX XX |1452244815.....|
```

How to mitigate Meltdown?

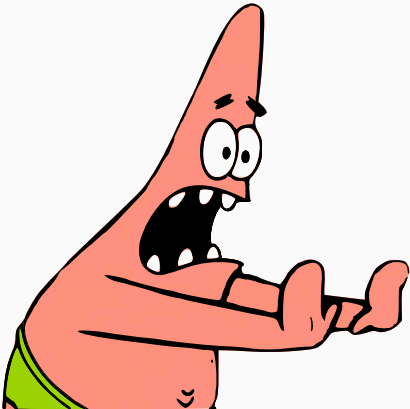
- Kernel addresses in user space are a problem

- Kernel addresses in user space are a problem
- Why don't we take the kernel addresses...



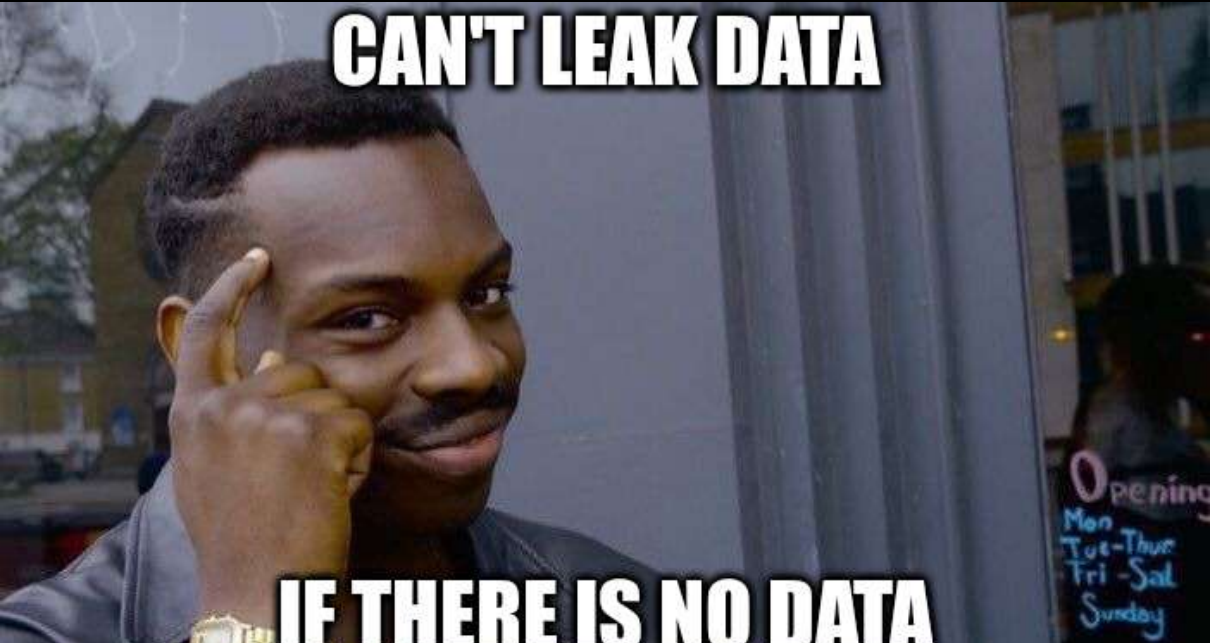


- ...and remove them if not needed?



- ...and remove them if not needed?
- User accessible check in hardware is not reliable

CAN'T LEAK DATA



IF THERE IS NO DATA





Kernel **A**ddress **I**solation to have **S**ide channels **E**fficiently **R**emoved

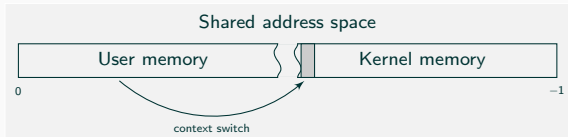
KAISER /ˈkAɪzə/

1. [german] Emperor, ruler of an empire
2. largest penguin, emperor penguin

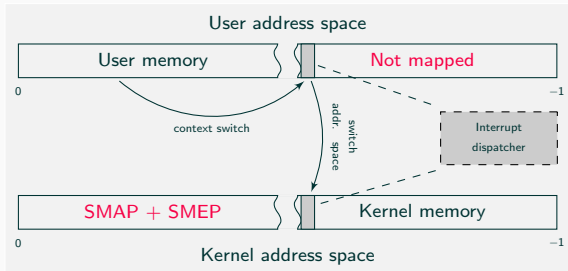


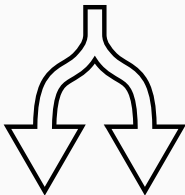
Kernel **A**ddress **I**solation to have **S**ide channels **E**fficiently **R**emoved

Without KAISER:

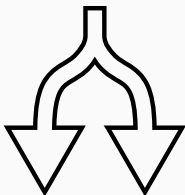


With KAISER:

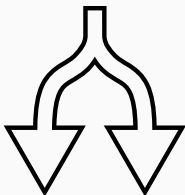




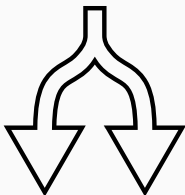
- We published **KAISER** in July 2017



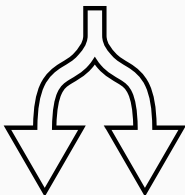
- We published **KAISER** in July 2017
- Intel and others improved and merged it into Linux as **KPTI** (Kernel Page Table Isolation)



- We published **KAISER** in July 2017
- Intel and others improved and merged it into Linux as **KPTI** (Kernel Page Table Isolation)
- Microsoft implemented similar concept in Windows 10



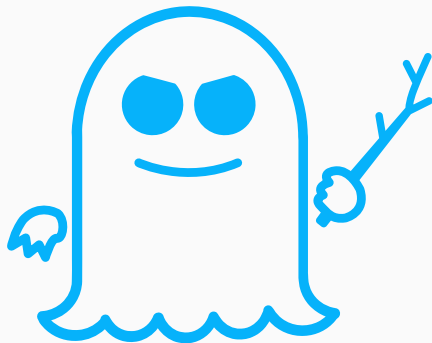
- We published **KAISER** in July 2017
- Intel and others improved and merged it into Linux as **KPTI** (Kernel Page Table Isolation)
- Microsoft implemented similar concept in Windows 10
- Apple implemented it in macOS 10.13.2 and called it “**Double Map**”



- We published **KAISER** in July 2017
- Intel and others improved and merged it into Linux as **KPTI** (Kernel Page Table Isolation)
- Microsoft implemented similar concept in Windows 10
- Apple implemented it in macOS 10.13.2 and called it “**Double Map**”
- All share the same idea: switching address spaces on context switch



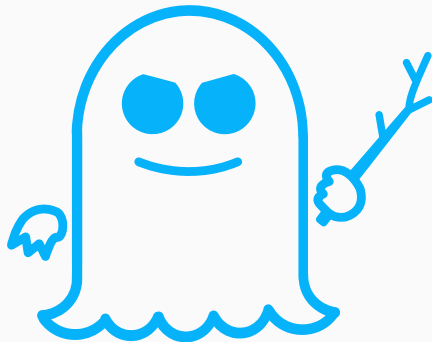
MELTDOWN



SPECTRE



MELTDOWN



SPECTRE

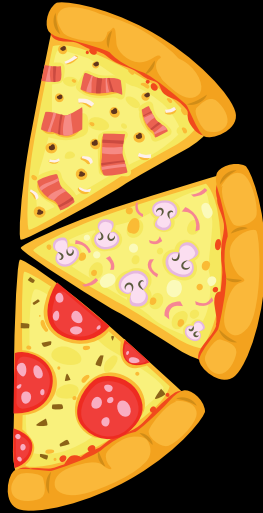




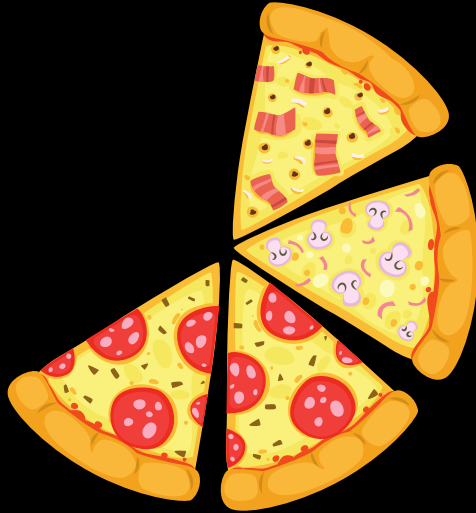
Prosciutto



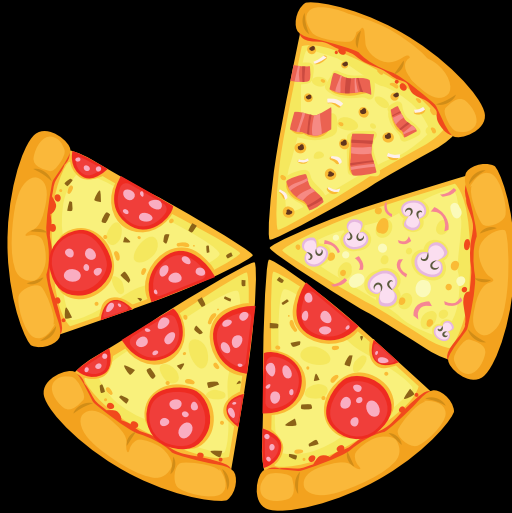
Funghi



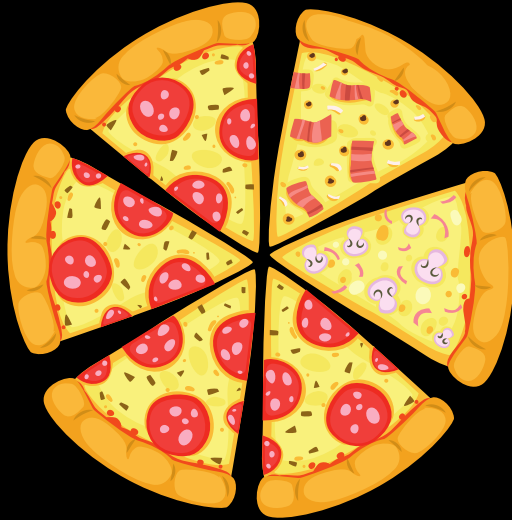
Diavolo



Diavolo



Diavolo



Diavolo

»A table for 6 please«





Speculative Cooking



»A table for 6 please«











- Mistrains branch prediction



- Mistrains branch prediction
- CPU speculatively executes code which should not be executed



- Mistrains branch prediction
- CPU speculatively executes code which should not be executed
- Can also mistrain indirect calls



- Mistrains branch prediction
 - CPU speculatively executes code which should not be executed
 - Can also mistrain indirect calls
- Spectre “convinces” program to execute code



```
index = 0;
```

```
char* data = "textKEY";
```

```
if (index < 4)
```

then



Prediction

else

```
LUT[data[index] * 4096]
```

```
0
```

```
index = 0;
```

```
char* data = "textKEY";
```

```
if (index < 4)
```

then

```
LUT[data[index] * 4096]
```



Prediction

else

```
0
```

```
index = 0;
```

```
char* data = "textKEY";
```

```
if (index < 4)
```

then

```
LUT[data[index] * 4096]
```



else

Speculate

0

```
index = 0;
```

```
char* data = "textKEY";
```

```
if (index < 4)
```

Execute

then

```
LUT[data[index] * 4096]
```



else

0

```
index = 1;
```

```
char* data = "textKEY";
```

```
if (index < 4)
```

then

else



Prediction

```
LUT[data[index] * 4096]
```

```
0
```

```
index = 1;
```



```
char* data = "textKEY";
```

```
if (index < 4)
```

then

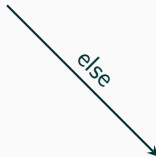


```
LUT[data[index] * 4096]
```



Prediction

else



```
0
```

```
index = 1;
```

```
char* data = "textKEY";
```

```
if (index < 4)
```

Speculate

then

```
LUT[data[index] * 4096]
```



Prediction

else

0


```
index = 1;
```

```
char* data = "textKEY";
```

```
if (index < 4)
```

then

```
LUT[data[index] * 4096]
```



Prediction

else

```
0
```

```
index = 2;
```

```
char* data = "textKEY";
```

```
if (index < 4)
```

then

else



Prediction

```
LUT[data[index] * 4096]
```

```
0
```

```
index = 2;
```

```
char* data = "textKEY";
```

```
if (index < 4)
```

then

```
LUT[data[index] * 4096]
```



Prediction

else

```
0
```

```
index = 2;
```

```
char* data = "textKEY";
```

```
if (index < 4)
```

Speculate

then

```
LUT[data[index] * 4096]
```



Prediction

else

0

```
index = 2;
```

```
char* data = "textKEY";
```

```
if (index < 4)
```

then

```
LUT[data[index] * 4096]
```



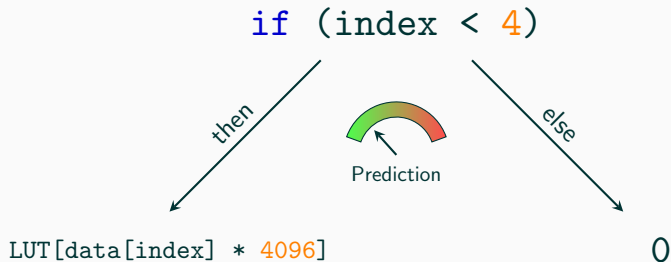
Prediction

else

```
0
```

```
index = 3;
```

```
char* data = "textKEY";
```



```
index = 3;
```

```
char* data = "textKEY";
```

```
if (index < 4)
```

then

```
LUT[data[index] * 4096]
```



Prediction

else

```
0
```

```
index = 3;
```

```
char* data = "textKEY";
```

```
if (index < 4)
```

Speculate

then

```
LUT[data[index] * 4096]
```



else

0


```
index = 3;
```

```
char* data = "textKEY";
```

```
if (index < 4)
```

then

```
LUT[data[index] * 4096]
```



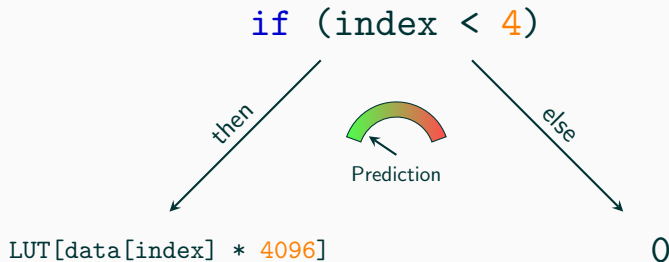
Prediction

else

```
0
```

```
index = 4;
```

```
char* data = "textKEY";
```



```
index = 4;
```

```
char* data = "textKEY";
```

```
if (index < 4)
```

then

```
LUT[data[index] * 4096]
```



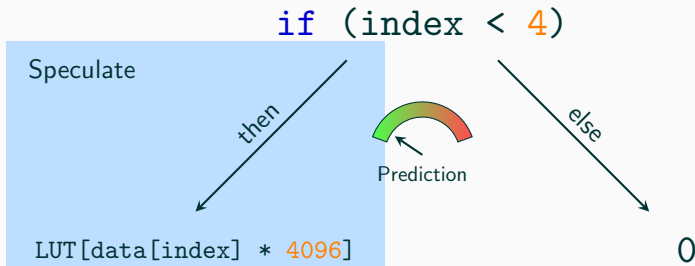
Prediction

else

```
0
```

```
index = 4;
```

```
char* data = "textKEY";
```



```
index = 4;
```

```
char* data = "textKEY";
```

```
if (index < 4)
```

then

```
LUT[data[index] * 4096]
```



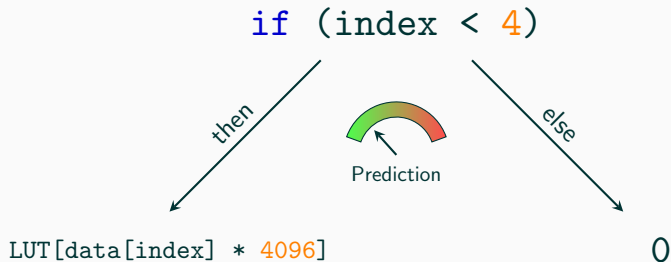
else

Execute

0

```
index = 5;
```

```
char* data = "textKEY";
```



```
index = 5;
```

```
char* data = "textKEY";
```

```
if (index < 4)
```

then

```
LUT[data[index] * 4096]
```



Prediction

else

```
0
```

```
index = 5;
```

```
char* data = "textKEY";
```

```
if (index < 4)
```

Speculate

then

```
LUT[data[index] * 4096]
```



else

0


```
index = 5;
```

```
char* data = "textKEY";
```

```
if (index < 4)
```

then

```
LUT[data[index] * 4096]
```



else

Execute

0

```
index = 6;
```

```
char* data = "textKEY";
```

```
if (index < 4)
```

then

else



Prediction

```
LUT[data[index] * 4096]
```

```
0
```

```
index = 6;
```

```
char* data = "textKEY";
```

```
if (index < 4)
```

then

```
LUT[data[index] * 4096]
```



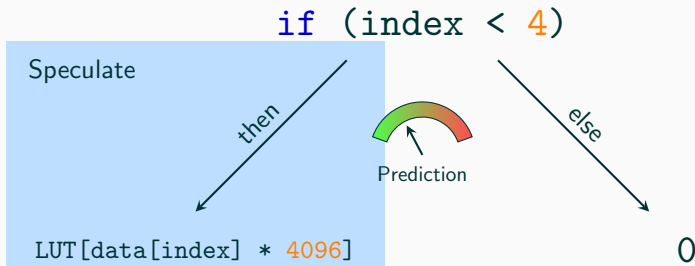
Prediction

else

```
0
```

```
index = 6;
```

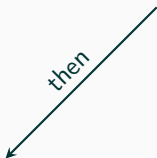
```
char* data = "textKEY";
```



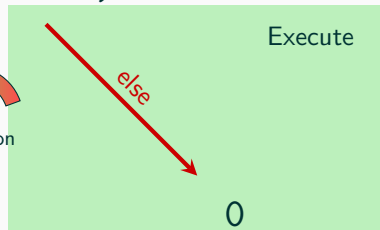
```
index = 6;
```

```
char* data = "textKEY";
```

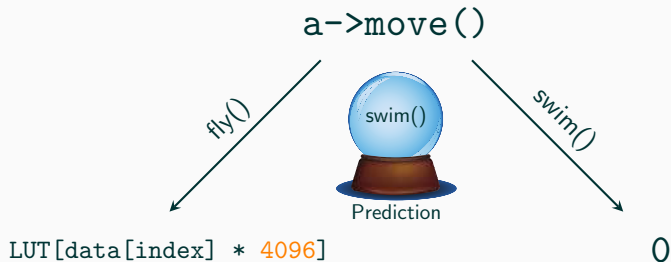
```
if (index < 4)
```



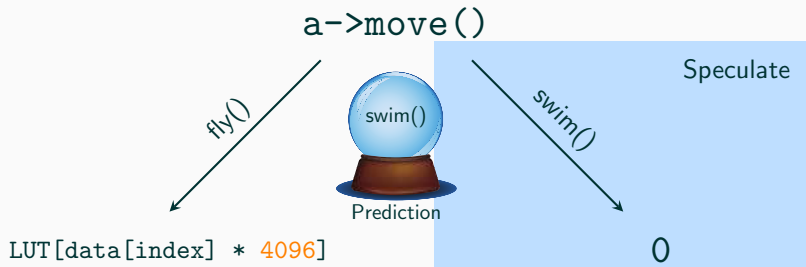
```
LUT[data[index] * 4096]
```



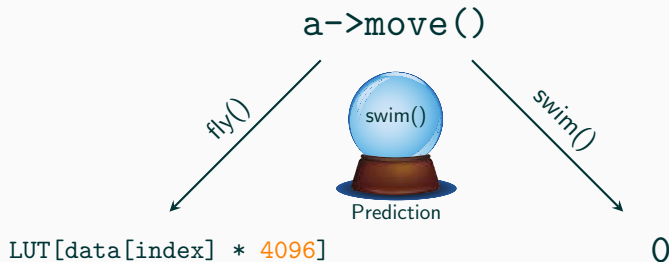
```
Animal* a = bird;
```



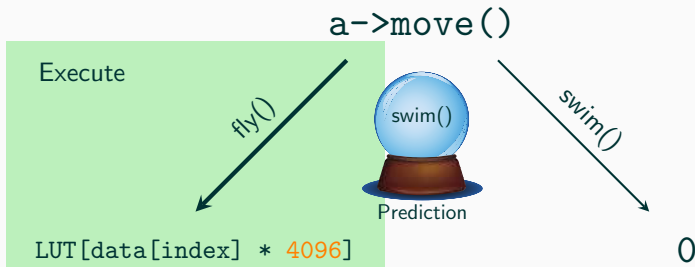
```
Animal* a = bird;
```



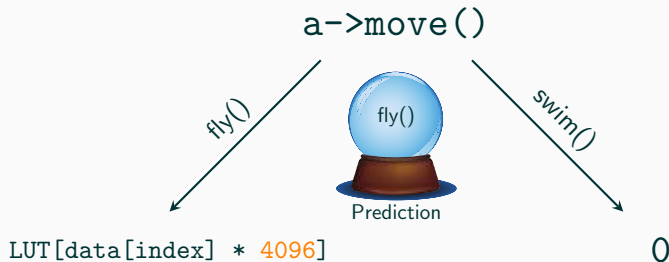
```
Animal* a = bird;
```



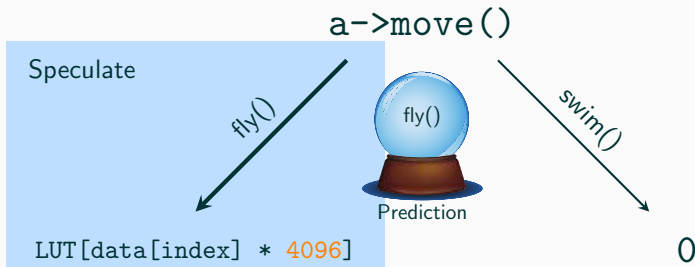

```
Animal* a = bird;
```



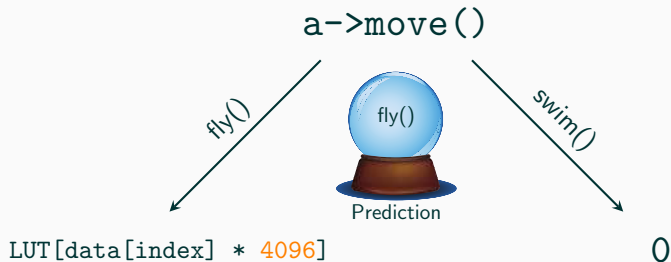
```
Animal* a = bird;
```



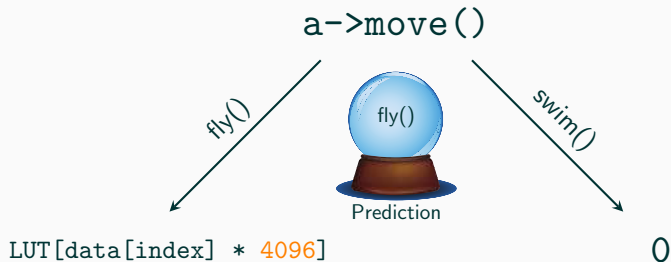
```
Animal* a = bird;
```



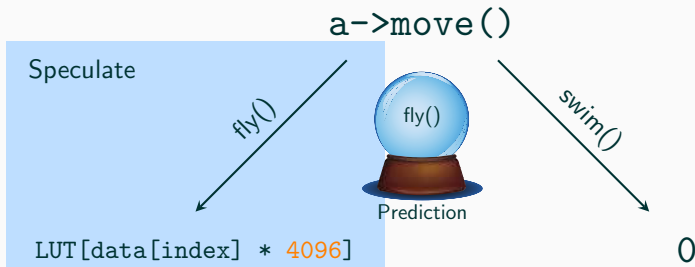
```
Animal* a = bird;
```



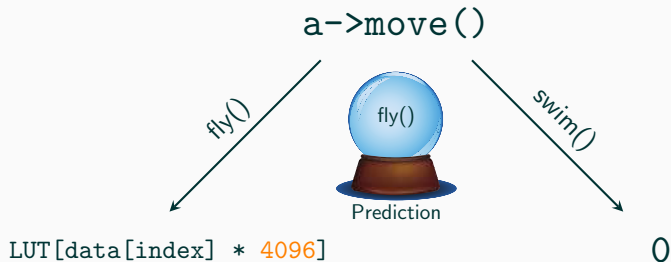
```
Animal* a = fish;
```



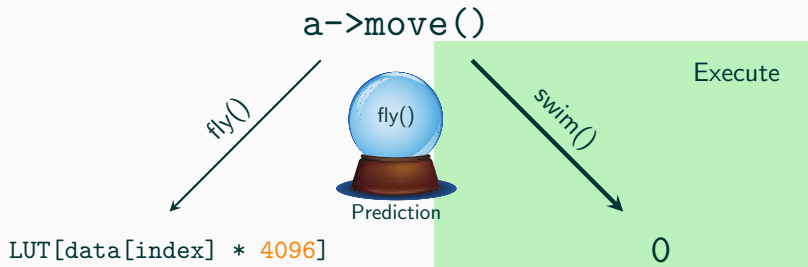
```
Animal* a = fish;
```



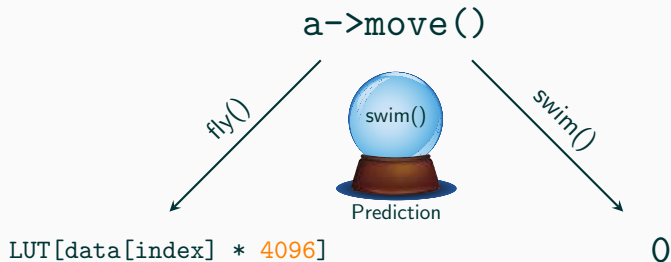
```
Animal* a = fish;
```



```
Animal* a = fish;
```



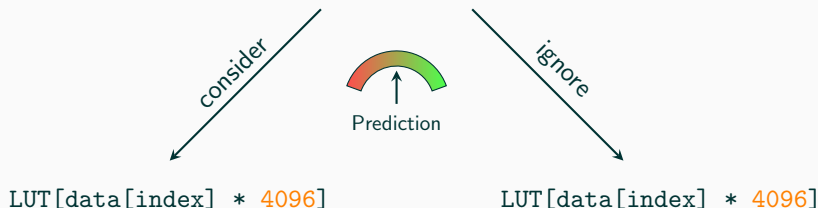

```
Animal* a = fish;
```

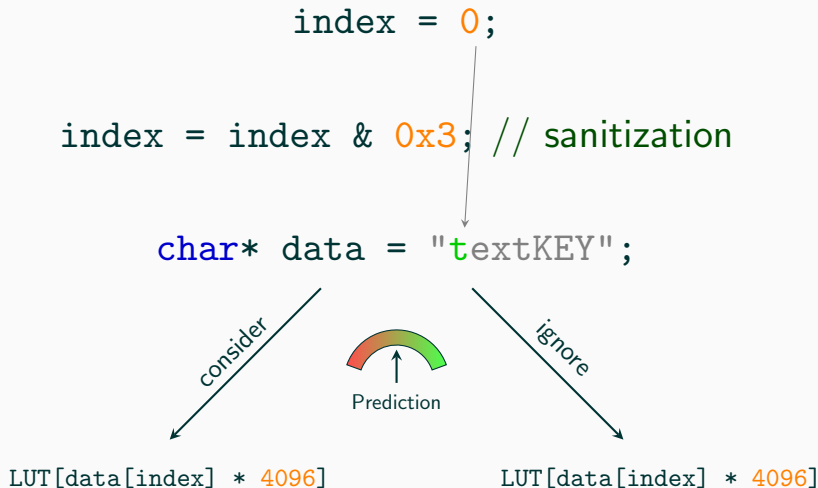


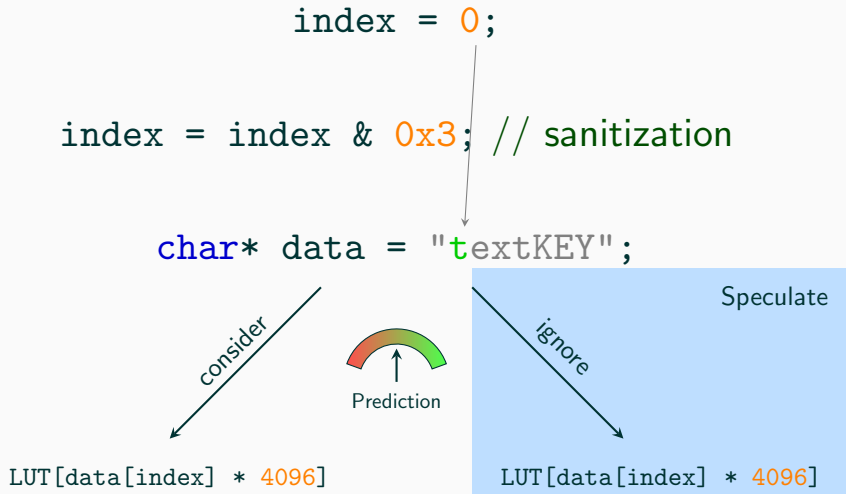
```
index = 0;
```

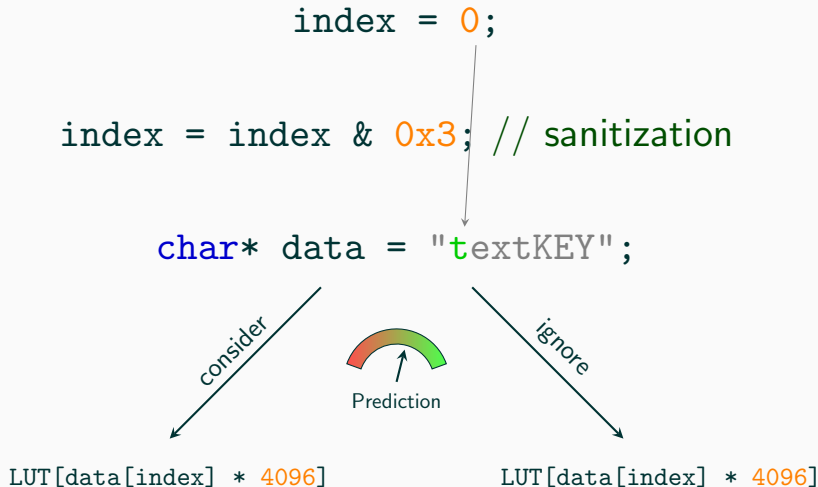
```
index = index & 0x3; // sanitization
```

```
char* data = "textKEY";
```





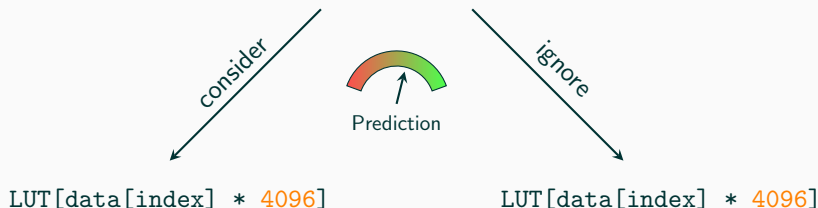


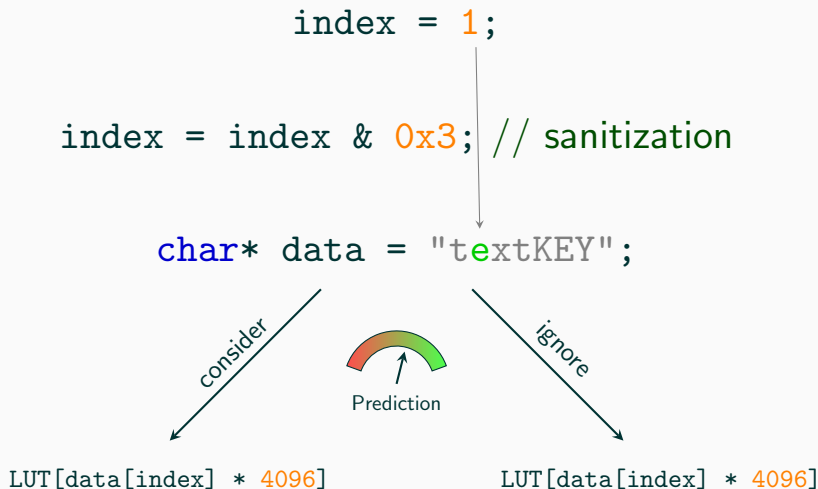


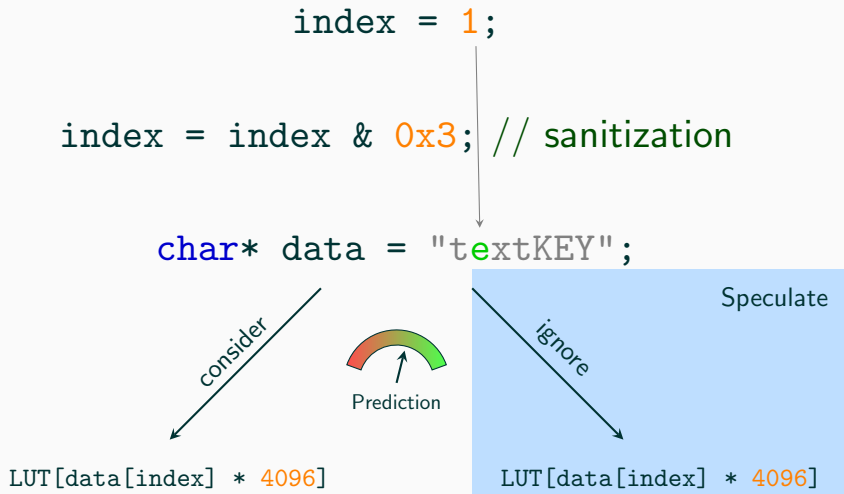
```
index = 1;
```

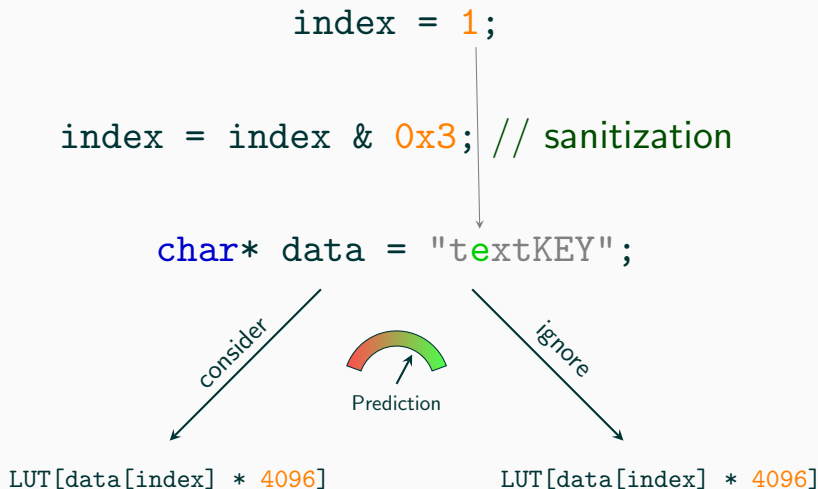
```
index = index & 0x3; // sanitization
```

```
char* data = "textKEY";
```





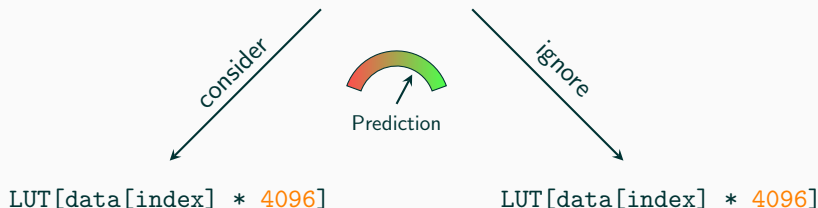


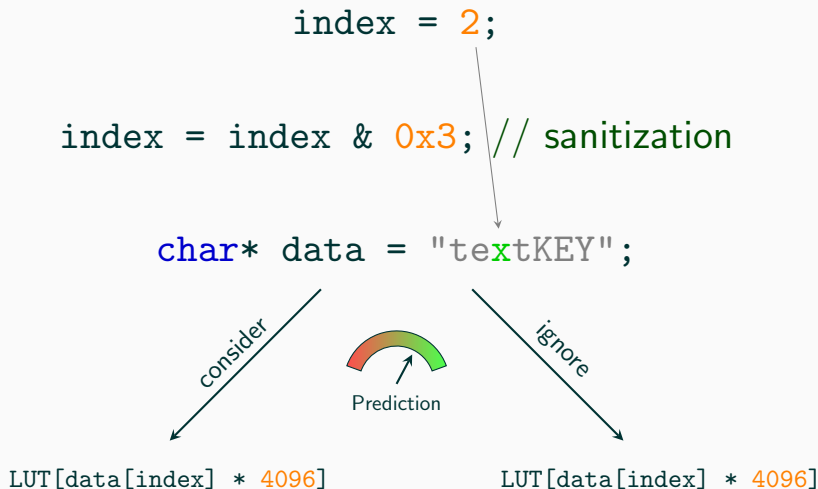


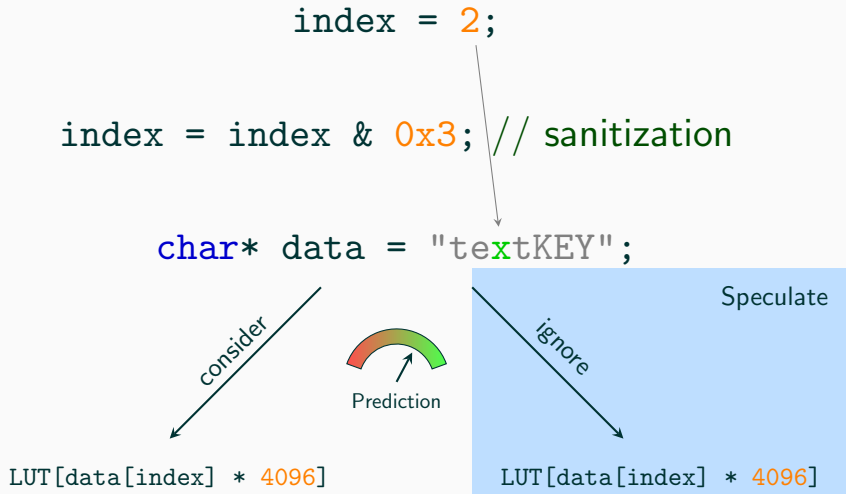
```
index = 2;
```

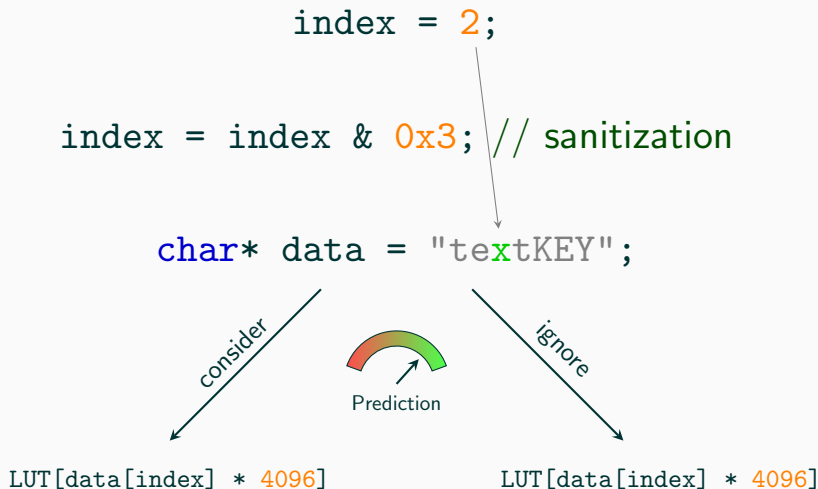
```
index = index & 0x3; // sanitization
```

```
char* data = "textKEY";
```





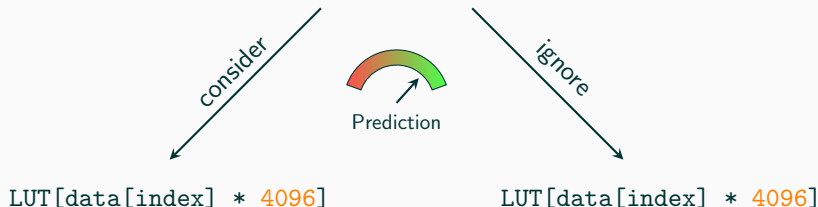


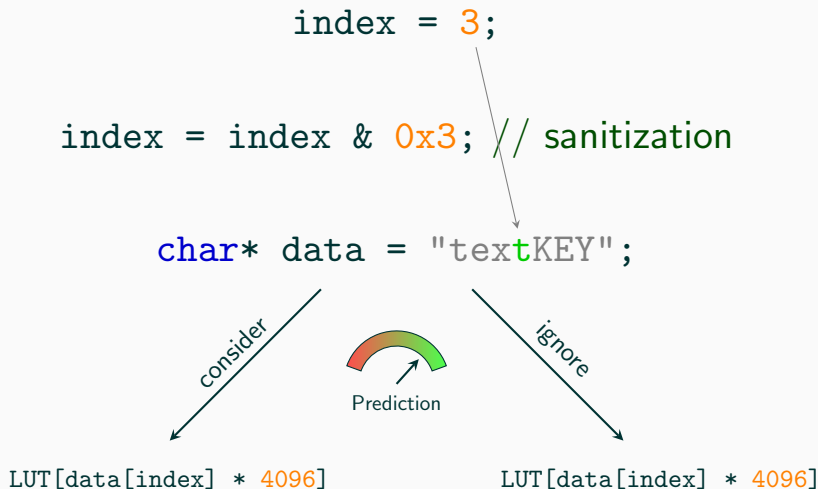


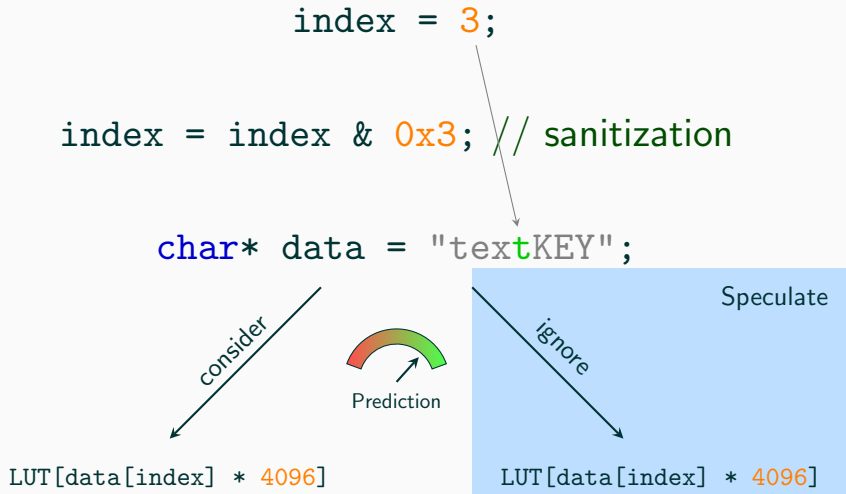
```
index = 3;
```

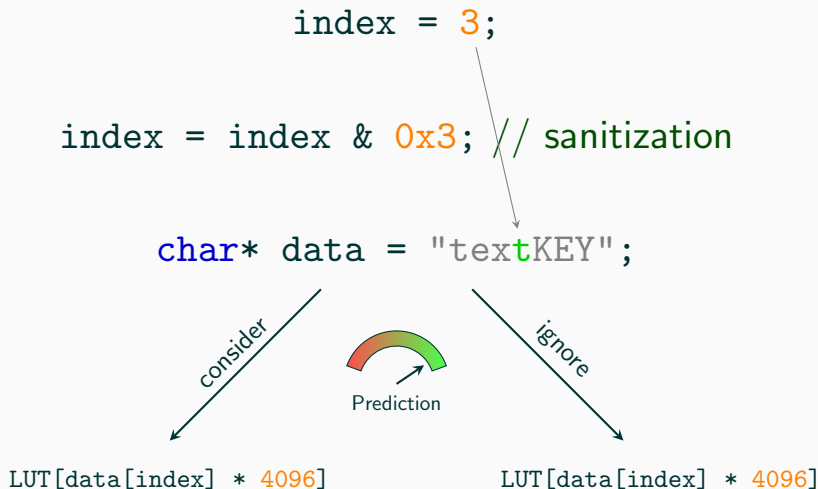
```
index = index & 0x3; // sanitization
```

```
char* data = "textKEY";
```





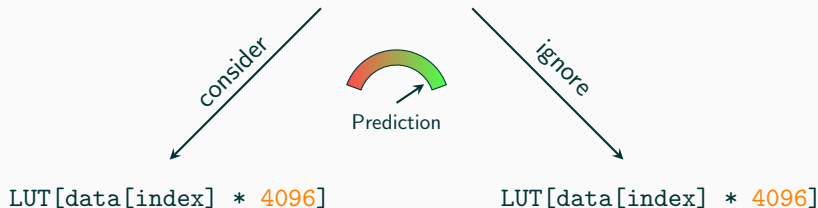


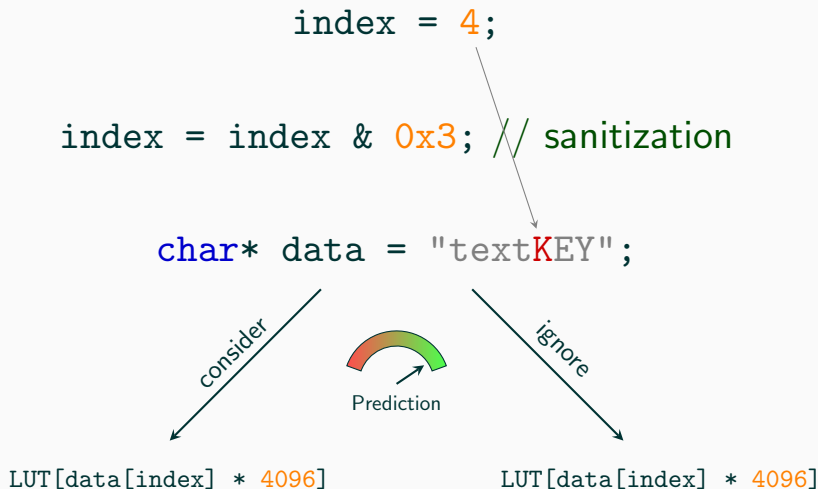


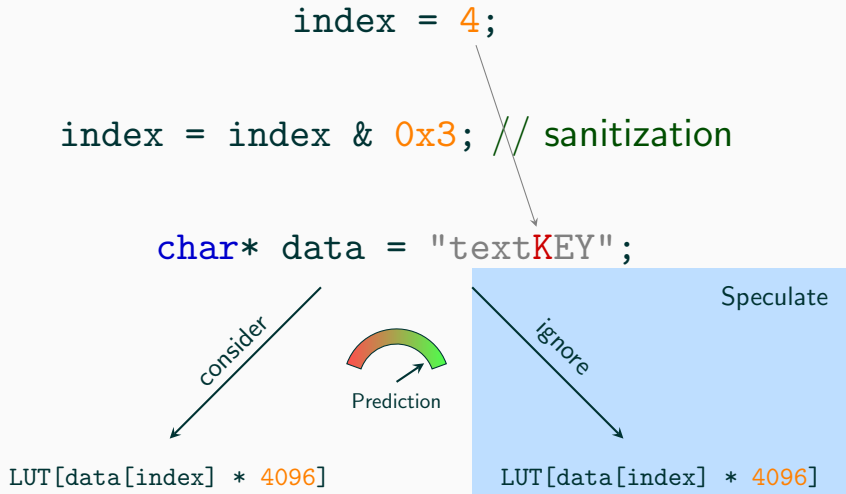
```
index = 4;
```

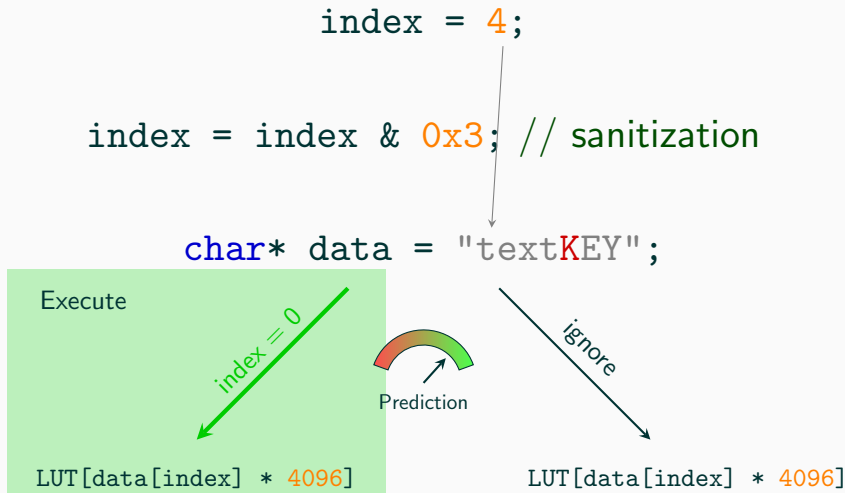
```
index = index & 0x3; // sanitization
```

```
char* data = "textKEY";
```





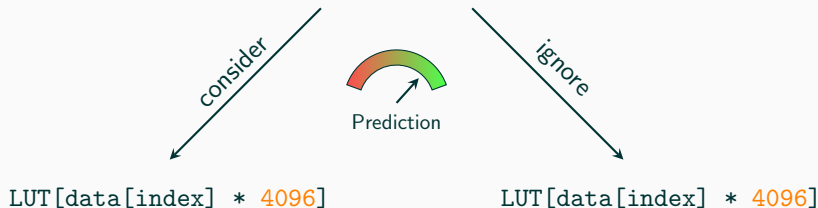


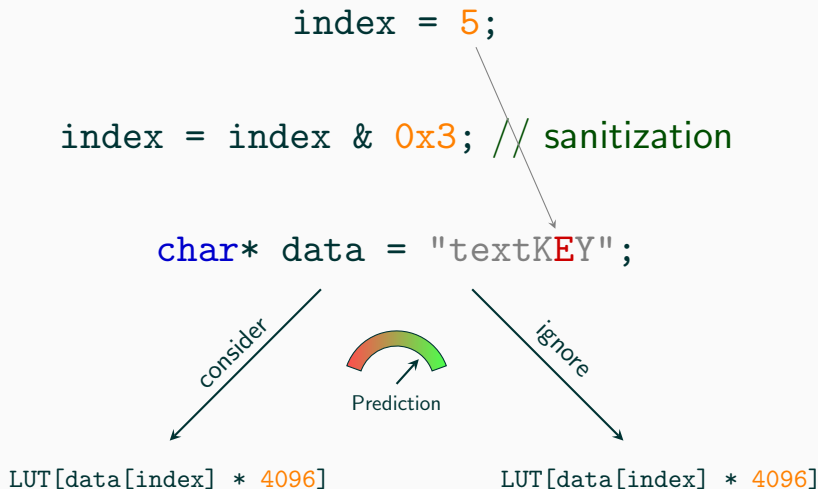


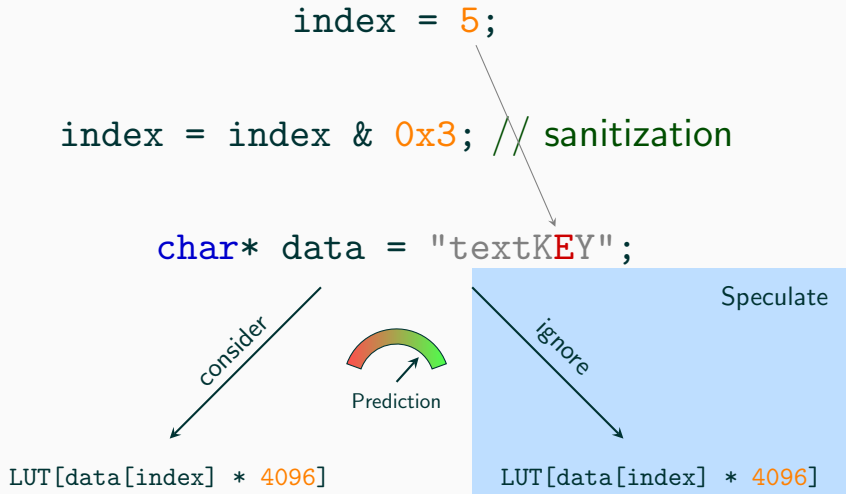
```
index = 5;
```

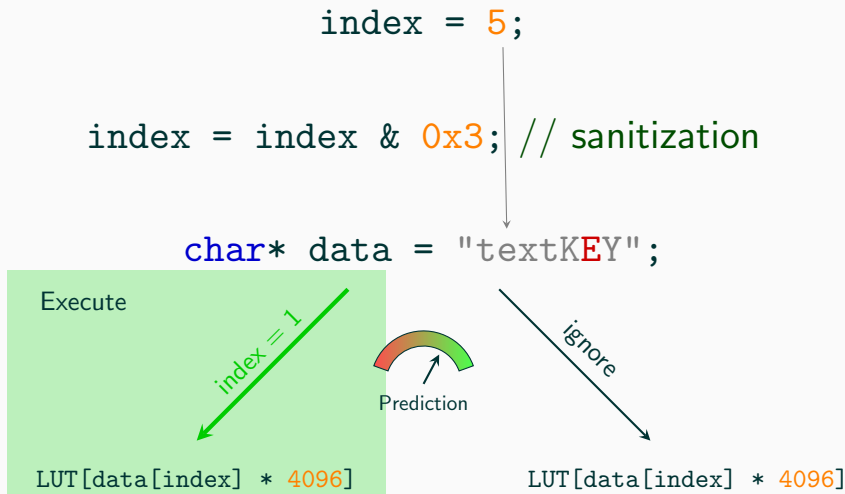
```
index = index & 0x3; // sanitization
```

```
char* data = "textKEY";
```





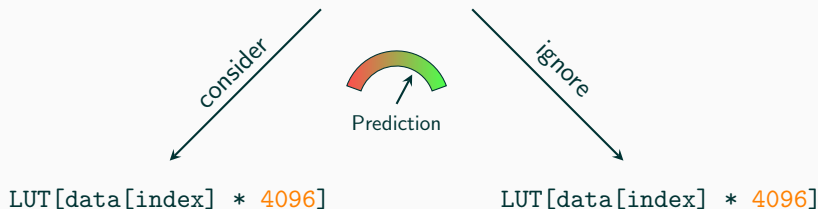


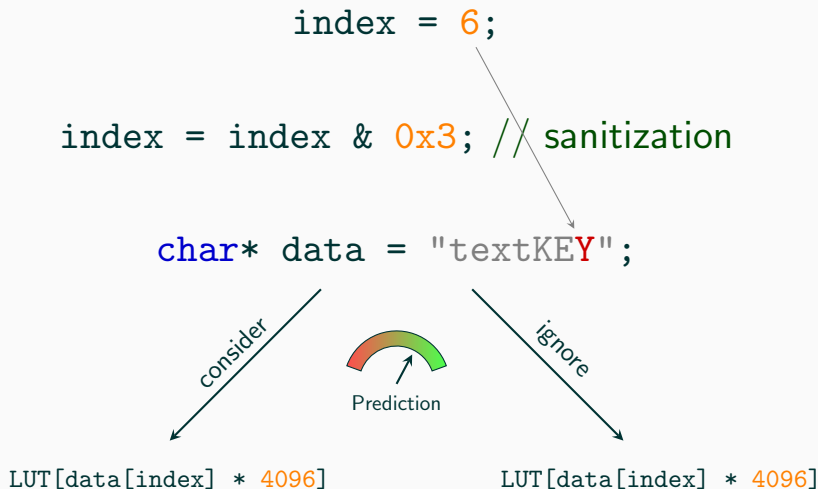


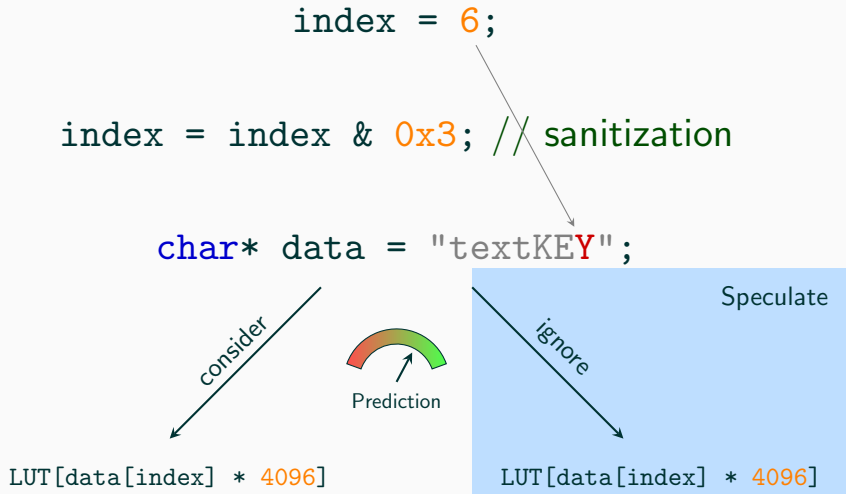
```
index = 6;
```

```
index = index & 0x3; // sanitization
```

```
char* data = "textKEY";
```







```
index = 6;
```

```
index = index & 0x3; // sanitization
```

```
char* data = "textKEY";
```

Execute

$\text{index} = 2$

$\text{LUT}[\text{data}[\text{index}]] * 4096$

Prediction

ignore

$\text{LUT}[\text{data}[\text{index}]] * 4096$



- Trivial approach: disable speculative execution



- Trivial approach: disable speculative execution
- No wrong speculation if there is no speculation



- Trivial approach: disable speculative execution
- No wrong speculation if there is no speculation
- Problem: massive performance hit!



- Trivial approach: disable speculative execution
- No wrong speculation if there is no speculation
- Problem: massive performance hit!
- Also: How to disable it?



- Trivial approach: disable speculative execution
- No wrong speculation if there is no speculation
- Problem: massive performance hit!
- Also: How to disable it?
- Speculative execution is deeply integrated into CPU





- Workaround: insert instructions stopping speculation



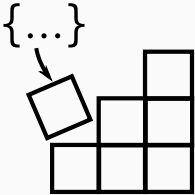
- Workaround: insert instructions stopping speculation
→ insert after every bounds check

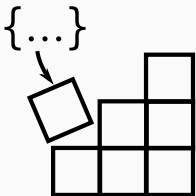


- Workaround: insert instructions stopping speculation
- insert after every bounds check
- x86: LFENCE, ARM: CSDB

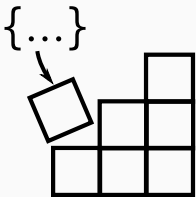


- Workaround: insert instructions stopping speculation
- insert after every bounds check
- x86: LFENCE, ARM: CSDB
 - Available on all Intel CPUs, retrofitted to existing ARMv7 and ARMv8

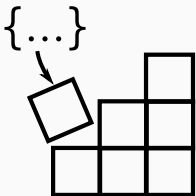




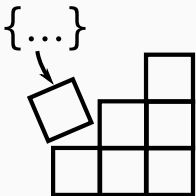
- Speculation barrier requires compiler supported



- Speculation barrier requires compiler supported
- Already implemented in GCC, LLVM, and MSVC



- Speculation barrier requires compiler supported
- Already implemented in GCC, LLVM, and MSVC
- Can be automated (MSVC) → not really reliable



- Speculation barrier requires compiler supported
- Already implemented in GCC, LLVM, and MSVC
- Can be automated (MSVC) → not really reliable
- Explicit use by programmer: `__builtin_load_no_speculate`

```
// Unprotected

int array[N];

int get_value(unsigned int n) {
    int tmp;

    if (n < N) {
        tmp = array[n]
    } else {
        tmp = FAIL;
    }

    return tmp;
}
```

```
// Unprotected

int array[N];

int get_value(unsigned int n) {
    int tmp;

    if (n < N) {
        tmp = array[n]
    } else {
        tmp = FAIL;
    }

    return tmp;
}
```

```
// Protected

int array[N];

int get_value(unsigned int n) {

    int *lower = array;
    int *ptr = array + n;
    int *upper = array + N;

    return
        __builtin_load_no_speculate
            (ptr, lower, upper, FAIL);
}
```





- Speculation barrier works if affected code constructs are known



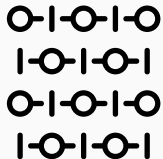
- Speculation barrier works if affected code constructs are known
- Programmer has to fully understand vulnerability



- Speculation barrier works if affected code constructs are known
- Programmer has to fully understand vulnerability
- Automatic detection is not reliable

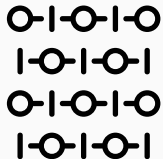


- Speculation barrier works if affected code constructs are known
- Programmer has to fully understand vulnerability
- Automatic detection is not reliable
- Non-negligible performance overhead of barriers



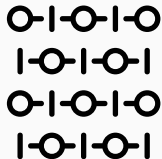
Intel released microcode updates

- Indirect Branch Restricted Speculation (IBRS):



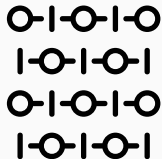
Intel released microcode updates

- Indirect Branch Restricted Speculation (IBRS):
 - Do not speculate based on anything before entering IBRS mode



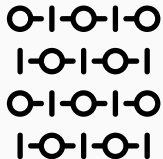
Intel released microcode updates

- Indirect Branch Restricted Speculation (IBRS):
 - Do not speculate based on anything before entering IBRS mode
 - lesser privileged code cannot influence predictions



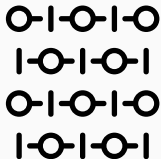
Intel released microcode updates

- Indirect Branch Restricted Speculation (IBRS):
 - Do not speculate based on anything before entering IBRS mode
→ lesser privileged code cannot influence predictions
- Indirect Branch Predictor Barrier (IBPB):



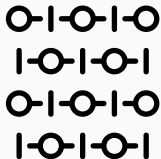
Intel released microcode updates

- Indirect Branch Restricted Speculation (IBRS):
 - Do not speculate based on anything before entering IBRS mode
→ lesser privileged code cannot influence predictions
- Indirect Branch Predictor Barrier (IBPB):
 - Flush branch-target buffer



Intel released microcode updates

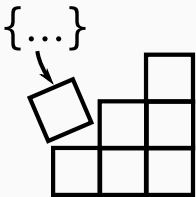
- Indirect Branch Restricted Speculation (IBRS):
 - Do not speculate based on anything before entering IBRS mode
→ lesser privileged code cannot influence predictions
- Indirect Branch Predictor Barrier (IBPB):
 - Flush branch-target buffer
- Single Thread Indirect Branch Predictors (STIBP):



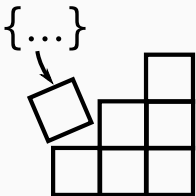
Intel released microcode updates

- Indirect Branch Restricted Speculation (IBRS):
 - Do not speculate based on anything before entering IBRS mode
→ lesser privileged code cannot influence predictions
- Indirect Branch Predictor Barrier (IBPB):
 - Flush branch-target buffer
- Single Thread Indirect Branch Predictors (STIBP):
 - Isolates branch prediction state between two hyperthreads

Retpoline (compiler extension)



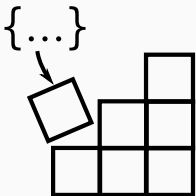
Retpoline (compiler extension)



```
1      push <call_target>
2      call 1f
3
4      2:                                ; speculation will continue here
5      lfence                            ; speculation barrier
6      jmp 2b                             ; endless loop
7
8      1:                                ; the actual call to <call_target>
9      lea 8(%rsp), %rsp ; restore stack pointer
10     ret
```

→ always predict to enter an endless loop

Retpoline (compiler extension)

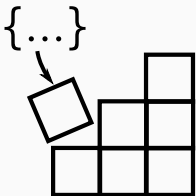


```
1      push <call_target>
2      call 1f
3
4      2:                                ; speculation will continue here
5      lfence                            ; speculation barrier
6      jmp 2b                            ; endless loop
7
8      1:                                ; the actual call to <call_target>
9      lea 8(%rsp), %rsp ; restore stack pointer
10     ret
```

→ always predict to enter an endless loop

- instead of the correct (or wrong) target function

Retpoline (compiler extension)

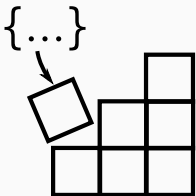


```
1      push <call_target>
2      call 1f
3
4      2:                                ; speculation will continue here
5      lfence                            ; speculation barrier
6      jmp 2b                             ; endless loop
7
8      1:                                ; the actual call to <call_target>
9      lea 8(%rsp), %rsp ; restore stack pointer
10     ret
```

→ always predict to enter an endless loop

- instead of the correct (or wrong) target function → performance?

Retpoline (compiler extension)

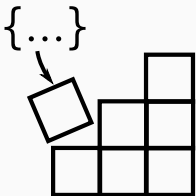


```
1      push <call_target>
2      call 1f
3
4      2:                                ; speculation will continue here
5      lfence                            ; speculation barrier
6      jmp 2b                             ; endless loop
7
8      1:                                ; the actual call to <call_target>
9      lea 8(%rsp), %rsp ; restore stack pointer
10     ret
```

→ always predict to enter an endless loop

- instead of the correct (or wrong) target function → performance?
- On Broadwell or newer:

Retpoline (compiler extension)

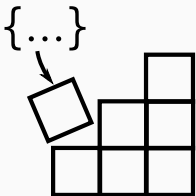


```
1      push <call_target>
2      call 1f
3
4      2:                                ; speculation will continue here
5      lfence                            ; speculation barrier
6      jmp 2b                            ; endless loop
7
8      1:                                ; the actual call to <call_target>
9      lea 8(%rsp), %rsp ; restore stack pointer
10     ret
```

→ always predict to enter an endless loop

- instead of the correct (or wrong) target function → performance?
- On Broadwell or newer:
 - **ret** may fall-back to the BTB for prediction

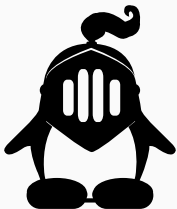
Retpoline (compiler extension)



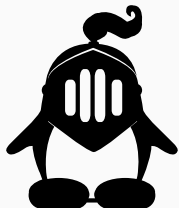
```
1      push <call_target>
2      call 1f
3
4      2:                                ; speculation will continue here
5      lfence                            ; speculation barrier
6      jmp 2b                            ; endless loop
7
8      1:                                ; restore stack pointer
9      lea 8(%rsp), %rsp                ; the actual call to <call_target>
10     ret
```

→ always predict to enter an endless loop

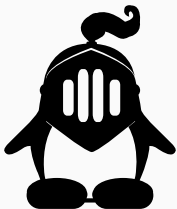
- instead of the correct (or wrong) target function → performance?
 - On Broadwell or newer:
 - **ret** may fall-back to the BTB for prediction
- microcode patches to prevent that



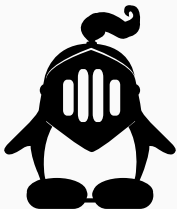
- ARM provides hardened Linux kernel



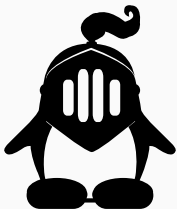
- ARM provides hardened Linux kernel
- Clears branch-predictor state on context switch



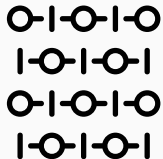
- ARM provides hardened Linux kernel
- Clears branch-predictor state on context switch
- Either via instruction (BPIALL)...



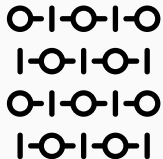
- ARM provides hardened Linux kernel
- Clears branch-predictor state on context switch
- Either via instruction (BPIALL)...
- ...or workaround (disable/enable MMU)



- ARM provides hardened Linux kernel
- Clears branch-predictor state on context switch
- Either via instruction (BPIALL)...
- ...or workaround (disable/enable MMU)
- Non-negligible performance overhead ($\approx 200\text{-}300\text{ ns}$)



Intel released microcode updates



Intel released microcode updates

- Disable store-to-load-forward speculation
- Performance impact of 2–8%



- Prevent access to high-resolution timer



- Prevent access to high-resolution timer
- Own timer using timing thread



- Prevent access to high-resolution timer
- Own timer using timing thread
- Flush instruction only privileged



- Prevent access to high-resolution timer
- Own timer using timing thread
- Flush instruction only privileged
- Cache eviction through memory accesses



- Prevent access to high-resolution timer
- Own timer using timing thread
- Flush instruction only privileged
- Cache eviction through memory accesses
- Just move secrets into secure world



- Prevent access to high-resolution timer
- Own timer using timing thread
- Flush instruction only privileged
- Cache eviction through memory accesses
- Just move secrets into secure world
- Spectre works on secure enclaves

Meltdown

Spectre

Meltdown

- Out-of-Order Execution

Spectre

- Speculative Execution (subset of Out-of-Order Execution)

Meltdown

- Out-of-Order Execution
- has nothing to do with branch prediction

Spectre

- Speculative Execution (subset of Out-of-Order Execution)
- fundamentally builds on branch (mis)prediction

Meltdown

- Out-of-Order Execution
- has **nothing to do with branch prediction**
- turning off speculative execution **entirely** has no effect on Meltdown

Spectre

- **Speculative Execution** (subset of Out-of-Order Execution)
- fundamentally builds on branch (mis)prediction
- turning off speculative execution **entirely** would work

Meltdown

- Out-of-Order Execution
 - has **nothing to do with branch prediction**
 - turning off speculative execution **entirely** has no effect on Meltdown
- **melts down** the isolation provided by the `user_accessible-bit`

Spectre

- **Speculative Execution** (subset of Out-of-Order Execution)
- fundamentally builds on branch (mis)prediction
- turning off speculative execution **entirely** would work
- has nothing to do with the `user_accessible-bit`

Meltdown

- Out-of-Order Execution
 - has **nothing to do with branch prediction**
 - turning off speculative execution **entirely** has no effect on Meltdown
- **melts down** the isolation provided by the `user_accessible-bit`
- in theory: OoO not required, pipelining can be sufficient

Spectre

- **Speculative Execution** (subset of Out-of-Order Execution)
- fundamentally builds on branch (mis)prediction
- turning off speculative execution **entirely** would work
- has nothing to do with the `user_accessible-bit`
- KAISER has no effect on Spectre at all

Meltdown

- Out-of-Order Execution
 - has **nothing to do with branch prediction**
 - turning off speculative execution **entirely** has no effect on Meltdown
- **melts down** the isolation provided by the `user_accessible-bit`
- in theory: OoO not required, pipelining can be sufficient
 - mitigated by KAISER

Spectre

- **Speculative Execution** (subset of Out-of-Order Execution)
- fundamentally builds on branch (mis)prediction
- turning off speculative execution **entirely** would work
- has nothing to do with the `user_accessible-bit`
- KAISER has no effect on Spectre at all

Meltdown

Spectre

Meltdown

- performs illegal memory accesses → we need to take care of processor exceptions

Spectre

- performs only legal memory accesses

Meltdown

- performs illegal memory accesses → we need to take care of processor exceptions
 - exception handling

Spectre

- performs only legal memory accesses
 - has nothing to do with exception handling

Meltdown

- performs illegal memory accesses → we need to take care of processor exceptions
 - exception handling
 - exception suppression with TSX

Spectre

- performs only legal memory accesses
 - has nothing to do with exception handling or suppression

Meltdown

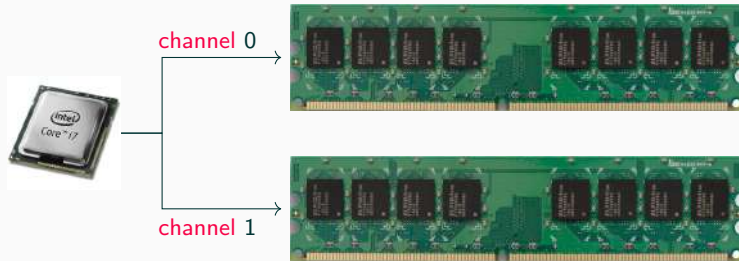
- performs illegal memory accesses → we need to take care of processor exceptions
 - exception handling
 - exception suppression with TSX
 - exception suppression with branch misprediction

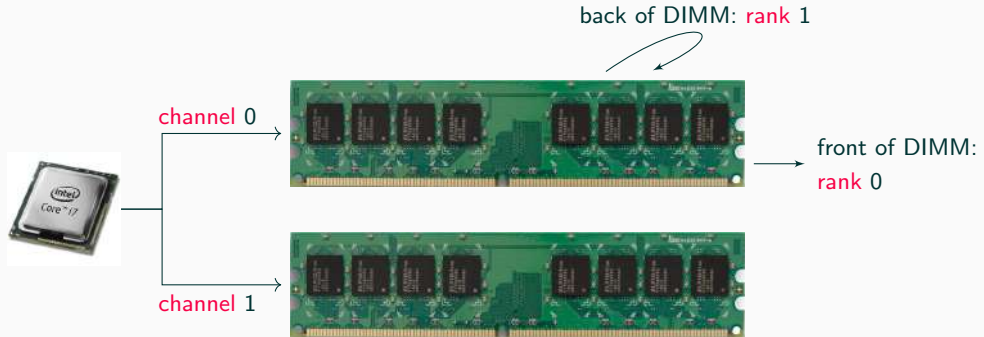
Spectre

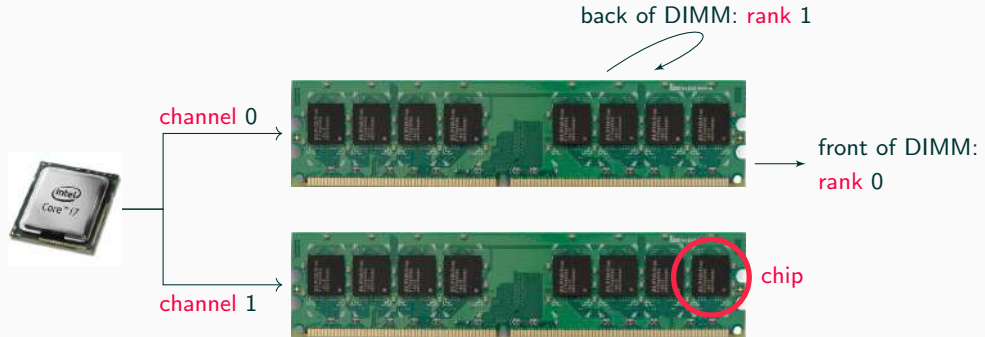
- performs only legal memory accesses
 - has nothing to do with exception handling or suppression

What if we want to modify data?

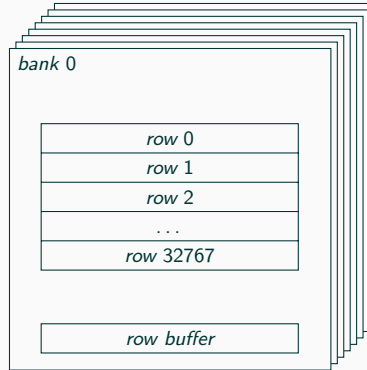


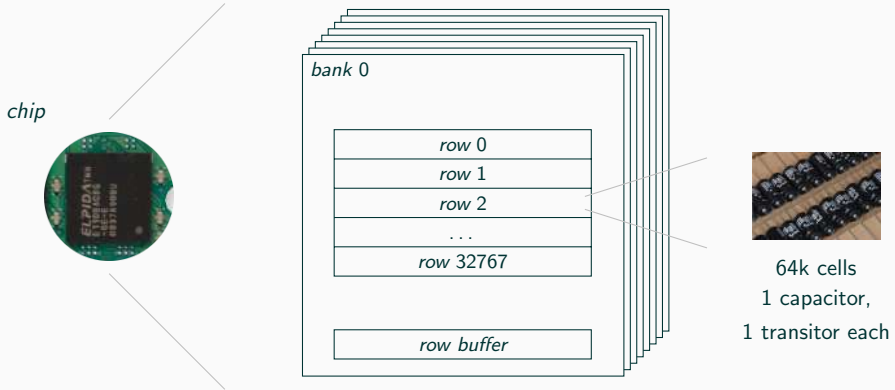


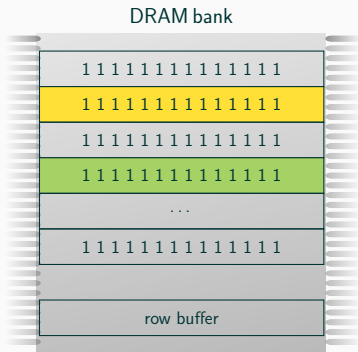




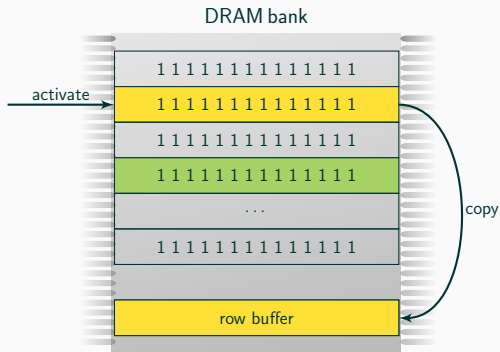
chip



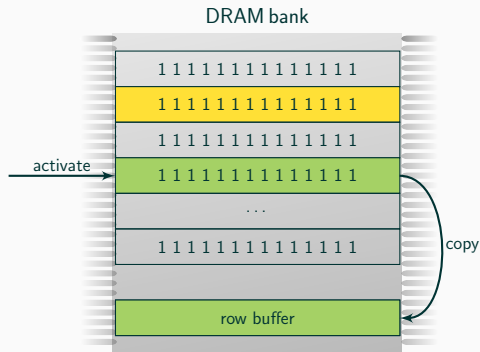




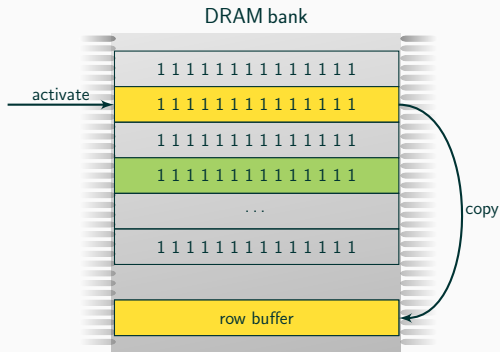
- Cells leak → repetitive **refresh** necessary
- Maximum interval between refreshes to guarantee **data integrity**
- Cells leak faster upon proximate accesses → Rowhammer



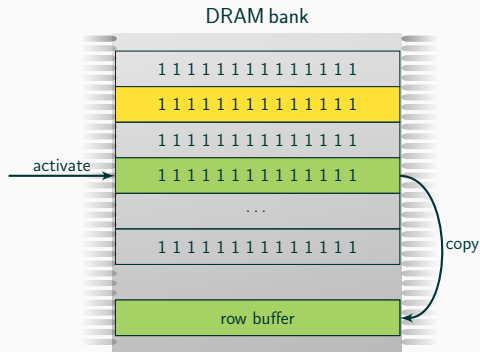
- Cells leak → repetitive **refresh** necessary
- Maximum interval between refreshes to guarantee **data integrity**
- Cells leak faster upon proximate accesses → Rowhammer



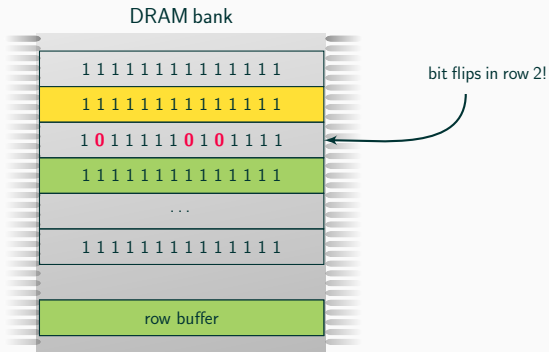
- Cells leak → repetitive **refresh** necessary
- Maximum interval between refreshes to guarantee **data integrity**
- Cells leak faster upon proximate accesses → Rowhammer



- Cells leak → repetitive **refresh** necessary
- Maximum interval between refreshes to guarantee **data integrity**
- Cells leak faster upon proximate accesses → Rowhammer



- Cells leak → repetitive **refresh** necessary
- Maximum interval between refreshes to guarantee **data integrity**
- Cells leak faster upon proximate accesses → Rowhammer



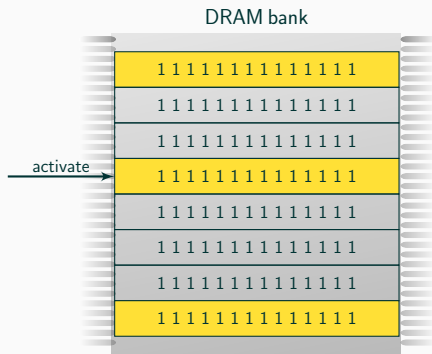
- Cells leak → repetitive **refresh** necessary
- Maximum interval between refreshes to guarantee **data integrity**
- Cells leak faster upon proximate accesses → Rowhammer

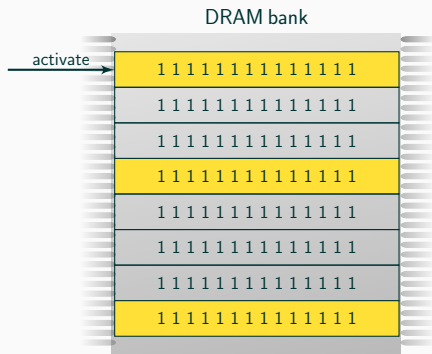
- There are two different hammering techniques

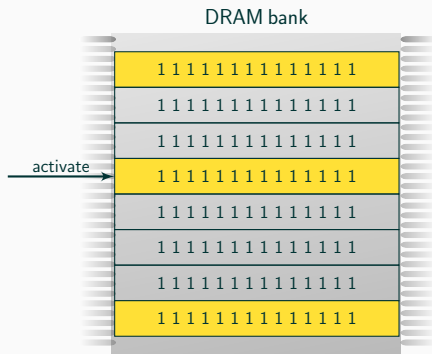
- There are two different hammering techniques
- #1: Hammer one row next to victim row and other random rows

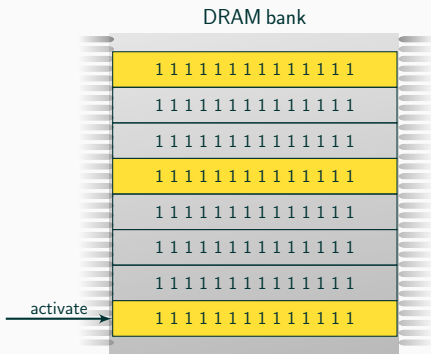
- There are two different hammering techniques
- #1: Hammer one row next to victim row and other random rows
- #2: Hammer two rows neighboring victim row

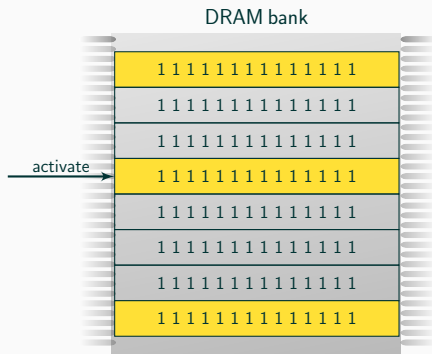
- There are **three** different hammering techniques
- #1: Hammer one row next to victim row and other random rows
- #2: Hammer two rows neighboring victim row
- #3: Hammer only one row next to victim row

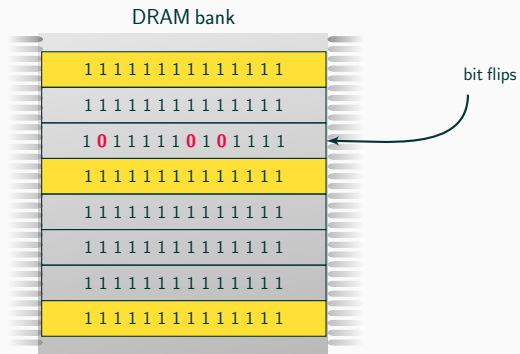


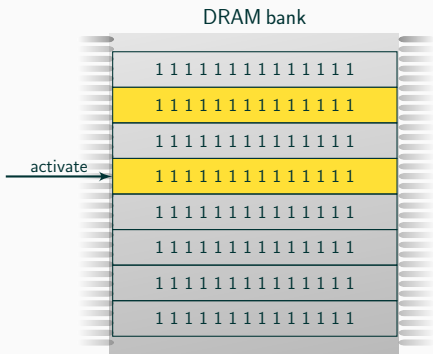


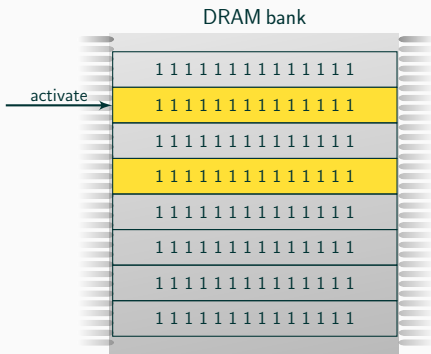


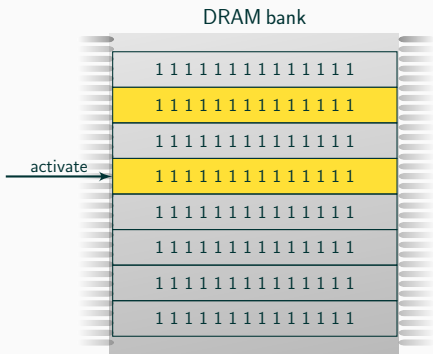


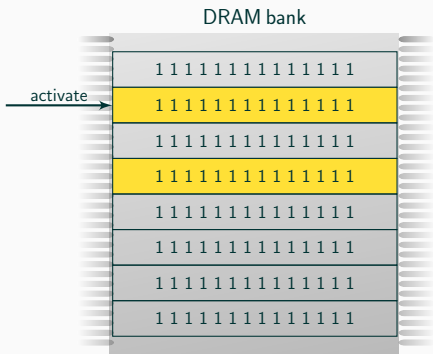


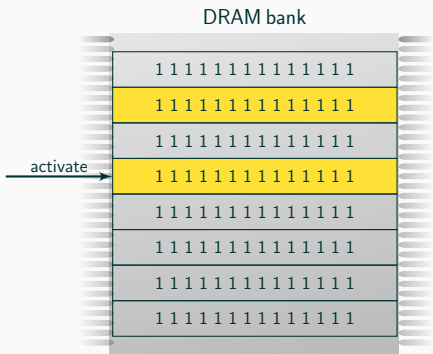


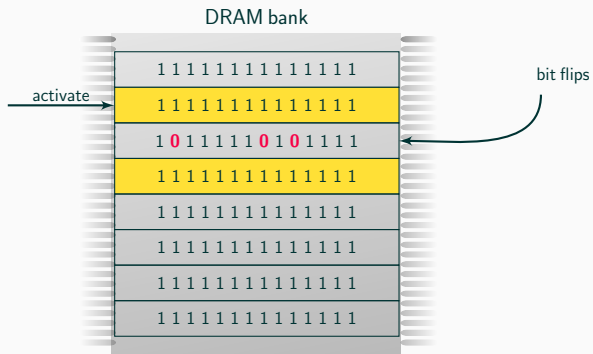


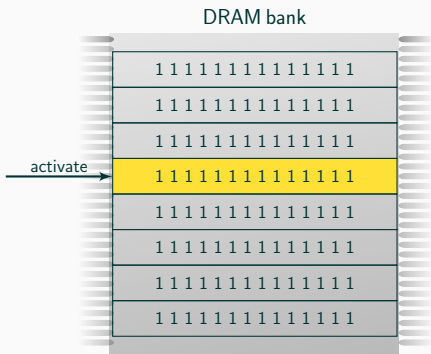


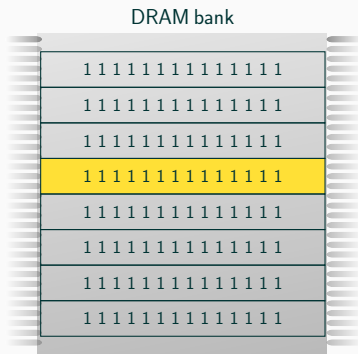


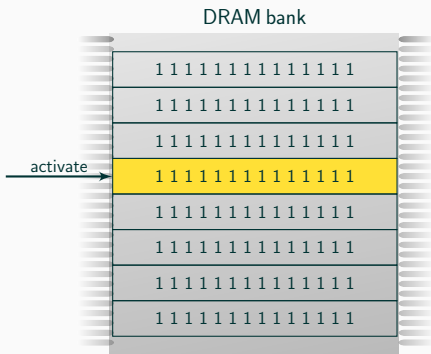


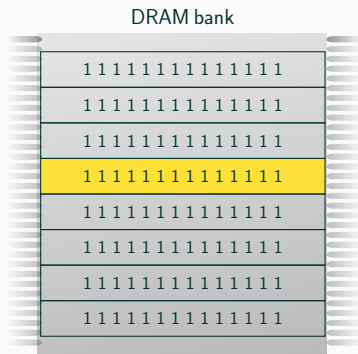


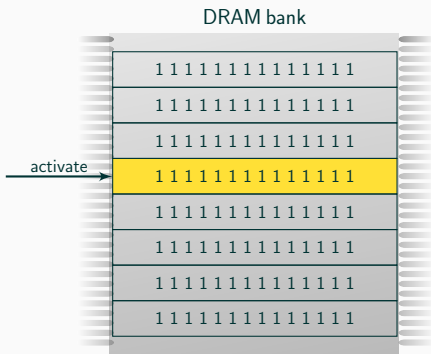


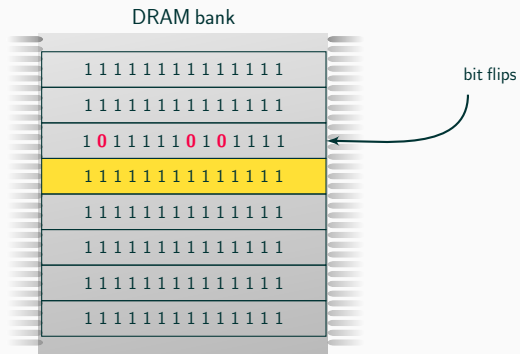


















- They are not random → highly reproducible flip pattern!



- They are not random → highly reproducible flip pattern!
 1. Choose a data structure that you can place at arbitrary memory locations



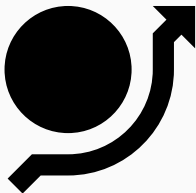
- They are not random → highly reproducible flip pattern!
 1. Choose a data structure that you can place at arbitrary memory locations
 2. Scan for “good” flips

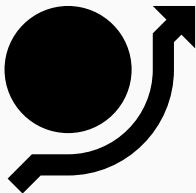


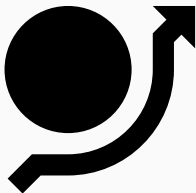
- They are not random → highly reproducible flip pattern!
 1. Choose a data structure that you can place at arbitrary memory locations
 2. Scan for “good” flips
 3. Place data structure there



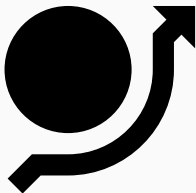
- They are not random → highly reproducible flip pattern!
 1. Choose a data structure that you can place at arbitrary memory locations
 2. Scan for “good” flips
 3. Place data structure there
 4. Trigger bit flip again



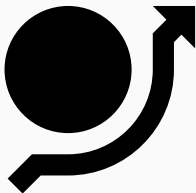




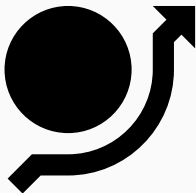
- Many applications perform actions as root



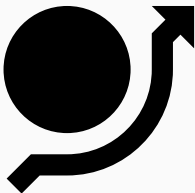
- Many applications perform actions as root



- Many applications perform actions as root
- They can be used by unprivileged users as well



- Many applications perform actions as root
- They can be used by unprivileged users as well



- Many applications perform actions as root
- They can be used by unprivileged users as well
- `sudo`

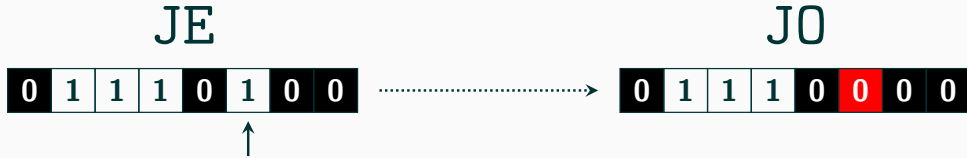


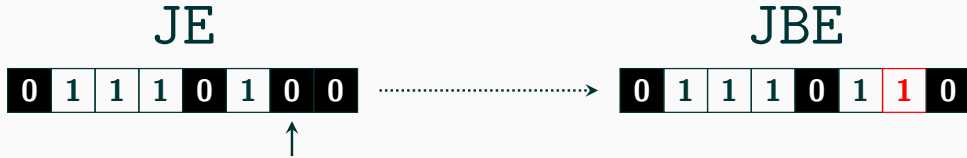
















Apple had a great idea:

- lowering the refresh rate saves energy but produces more bit flips



Apple had a great idea:

- lowering the refresh rate saves energy but produces more bit flips
- use ECC memory to mitigate bit flips



Apple had a great idea:

- lowering the refresh rate saves energy but produces more bit flips
- use ECC memory to mitigate bit flips
- in the end: it's an optimization problem.



Apple had a great idea:

- lowering the refresh rate saves energy but produces more bit flips
- use ECC memory to mitigate bit flips
- in the end: it's an optimization problem.
 - too aggressive? bit flips will be possible



Apple had a great idea:

- lowering the refresh rate saves energy but produces more bit flips
- use ECC memory to mitigate bit flips
- in the end: it's an optimization problem.
 - too aggressive? bit flips will be possible
 - too cautious? waste of energy



Apple had a great idea:

- lowering the refresh rate saves energy but produces more bit flips
- use ECC memory to mitigate bit flips
- in the end: it's an optimization problem.
 - too aggressive? bit flips will be possible
 - too cautious? waste of energy
 - what if the “too aggressive” changes over time?



Apple had a great idea:

- lowering the refresh rate saves energy but produces more bit flips
- use ECC memory to mitigate bit flips
- in the end: it's an optimization problem.
 - too aggressive? bit flips will be possible
 - too cautious? waste of energy
 - what if the “too aggressive” changes over time?
 - what if attackers come up with slightly better attacks?



Apple had a great idea:

- lowering the refresh rate saves energy but produces more bit flips
- use ECC memory to mitigate bit flips
- in the end: it's an optimization problem.
 - too aggressive? bit flips will be possible
 - too cautious? waste of energy
 - what if the “too aggressive” changes over time?
 - what if attackers come up with slightly better attacks?
- difficult to optimize with an intelligent adversary



We have ignored microarchitectural attacks for many many years:



We have ignored microarchitectural attacks for many many years:

- attacks on crypto



We have ignored microarchitectural attacks for many many years:

- attacks on crypto → “software should be fixed”



We have ignored microarchitectural attacks for many many years:

- attacks on crypto → “software should be fixed”
- attacks on ASLR



We have ignored microarchitectural attacks for many many years:

- attacks on crypto → “software should be fixed”
- attacks on ASLR → “ASLR is broken anyway”



We have ignored microarchitectural attacks for many many years:

- attacks on crypto → “software should be fixed”
- attacks on ASLR → “ASLR is broken anyway”
- attacks on SGX and TrustZone



We have ignored microarchitectural attacks for many many years:

- attacks on crypto → “software should be fixed”
- attacks on ASLR → “ASLR is broken anyway”
- attacks on SGX and TrustZone → “not part of the threat model”



We have ignored microarchitectural attacks for many many years:

- attacks on crypto → “software should be fixed”
- attacks on ASLR → “ASLR is broken anyway”
- attacks on SGX and TrustZone → “not part of the threat model”
- Rowhammer attacks



We have ignored microarchitectural attacks for many many years:

- attacks on crypto → “software should be fixed”
- attacks on ASLR → “ASLR is broken anyway”
- attacks on SGX and TrustZone → “not part of the threat model”
- Rowhammer attacks → “only affects cheap sub-standard modules”



We have ignored microarchitectural attacks for many many years:

- attacks on crypto → “software should be fixed”
 - attacks on ASLR → “ASLR is broken anyway”
 - attacks on SGX and TrustZone → “not part of the threat model”
 - Rowhammer attacks → “only affects cheap sub-standard modules”
- for years we solely optimized for performance



After learning about a side channel you realize:



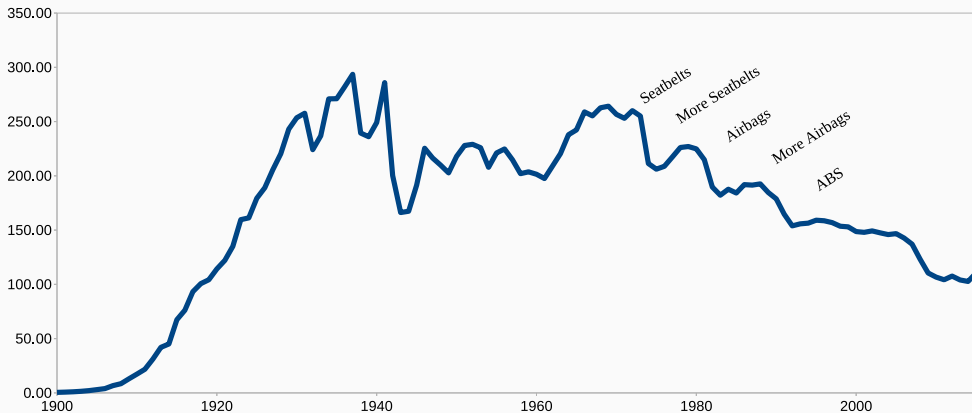
After learning about a side channel you realize:

- the side channels were documented in the Intel manual



After learning about a side channel you realize:

- the side channels were documented in the Intel manual
- only now we understand the implications



Motor Vehicle Deaths in U.S. by Year





- moral obligation to invest more time on defenses than on attacks



- moral obligation to invest more time on defenses than on attacks
- **dangerous**: we overlooked Meltdown and Spectre for decades



- moral obligation to invest more time on defenses than on attacks
- **dangerous**: we overlooked Meltdown and Spectre for decades
- we don't know all problems. do we know at least the most important subset?



- moral obligation to invest more time on defenses than on attacks
- **dangerous**: we overlooked Meltdown and Spectre for decades
- we don't know all problems. do we know at least the most important subset?
- are we hammering on a small subset of problems and forgot about the bigger picture?





A unique chance to

- rethink processor design



A unique chance to

- rethink processor design
- grow up, like other fields (car industry, construction industry)



A unique chance to

- rethink processor design
- grow up, like other fields (car industry, construction industry)
- dedicate more time into identifying problems and not solely in mitigating known problems

Microarchitectural Attacks:

From the Basics to Arbitrary Read and Write Primitives without any Software Bugs

Daniel Gruss

June 19, 2018

Graz University of Technology