Microarchitectural Side-Channel Attacks

From the Basics to Transient Execution Attacks

Claudio Canella

June 17, 2018

IAIK – Graz University of Technology

Who am I?

Claudio Canella

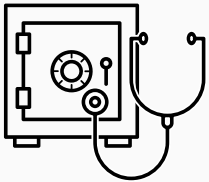
PhD student @ Graz University of Technology

@cc0x1f

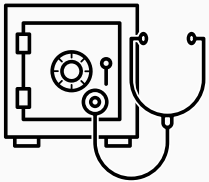
claudio.canella@iaik.tugraz.at

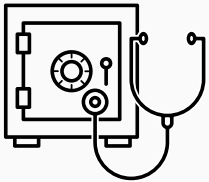
Side-Channel Attacks

- Safe software infrastructure does not mean safe execution

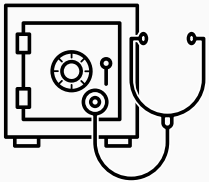


- Safe software infrastructure does not mean safe execution
- Information leaks because of the underlying hardware





- Safe software infrastructure does not mean safe execution
- Information leaks because of the underlying hardware
- Exploit unintentional information leakage by side-effects



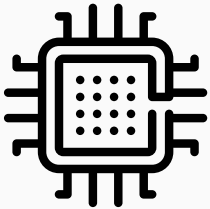
- Safe software infrastructure does not mean safe execution
- Information leaks because of the underlying hardware
- Exploit unintentional information leakage by side-effects

Power consumption

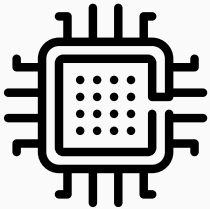
Execution time

CPU caches

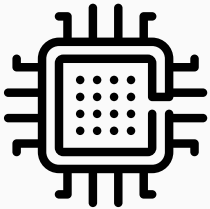
Caches and Cache Attacks



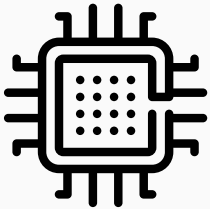
- Cache-based keylogging



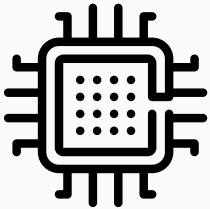
- Cache-based keylogging
- Crypto key recovery



- Cache-based keylogging
- Crypto key recovery
  - various implementations (AES, RSA, ECC, ...)
  - up to 97% key bits recovered after 1 encryption

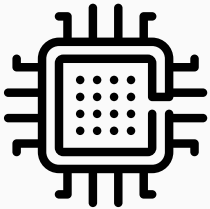


- Cache-based keylogging
- Crypto key recovery
  - various implementations (AES, RSA, ECC, ...)
  - up to 97% key bits recovered after 1 encryption
- Cross-VM, cross-core, even cross-CPU



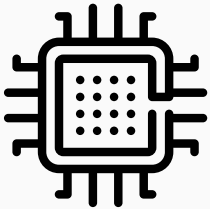
- Cache-based keylogging
- Crypto key recovery
  - various implementations (AES, RSA, ECC, ...)
  - up to 97% key bits recovered after 1 encryption
- Cross-VM, cross-core, even cross-CPU
- Any CPU vendor

Cross-Core Attacks?



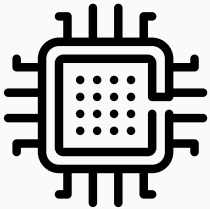
- using the *inclusive* property

Cross-Core Attacks?



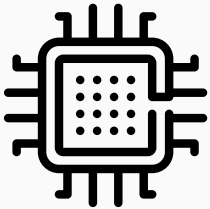
- using the *inclusive* property
- last-level cache is a superset of L1 and L2

Cross-Core Attacks?



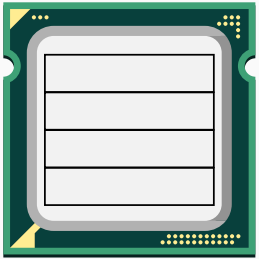
- using the *inclusive* property
- last-level cache is a superset of L1 and L2
- data evicted from last-level cache $\rightarrow$ evicted from L1 and L2

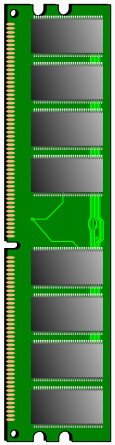
Cross-Core Attacks?



- using the *inclusive* property
- last-level cache is a superset of L1 and L2
- data evicted from last-level cache $\rightarrow$ evicted from L1 and L2
- a core can evict lines in the private L1 of another core

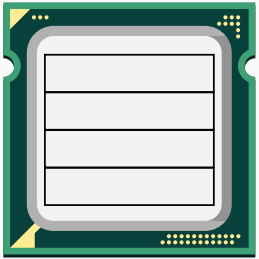
```
printf("%d", i);
printf("%d", i);
```

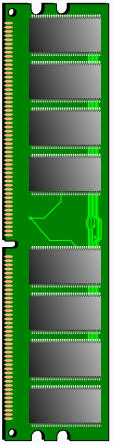




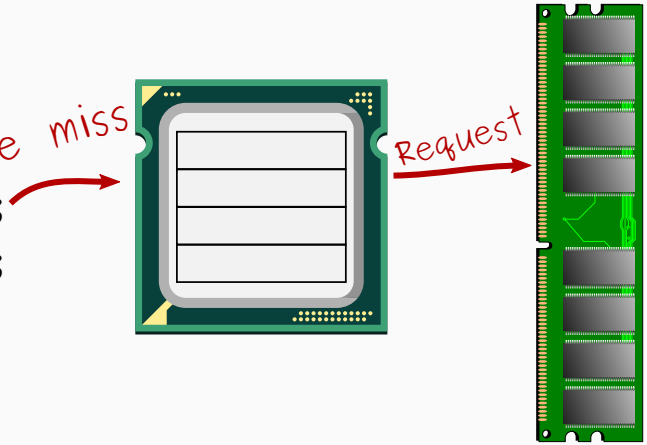
```
printf("%d", i);
printf("%d", i);
```

Cache miss

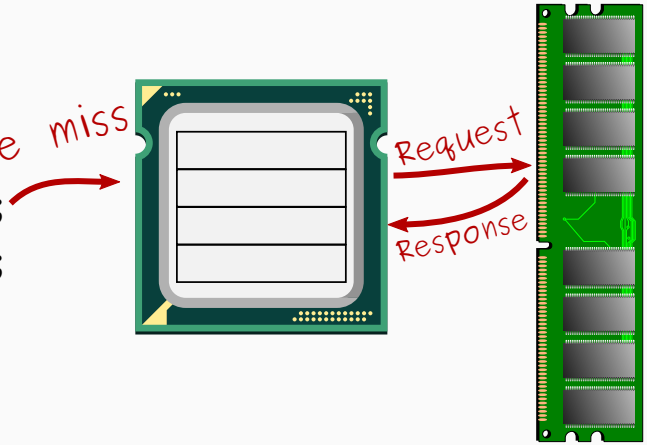




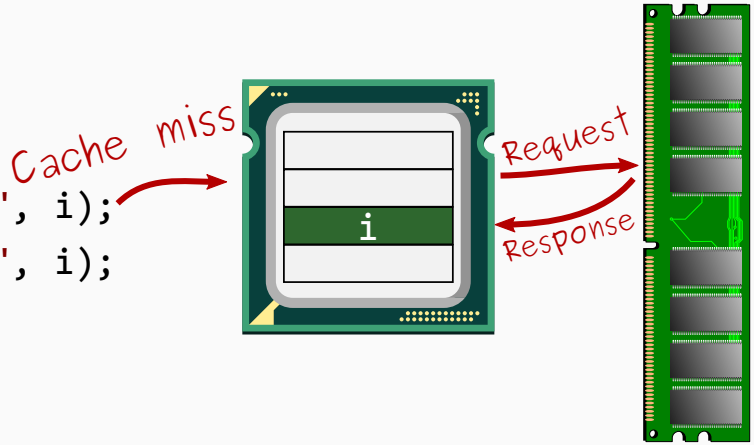
```
printf("%d", i);
printf("%d", i);
```

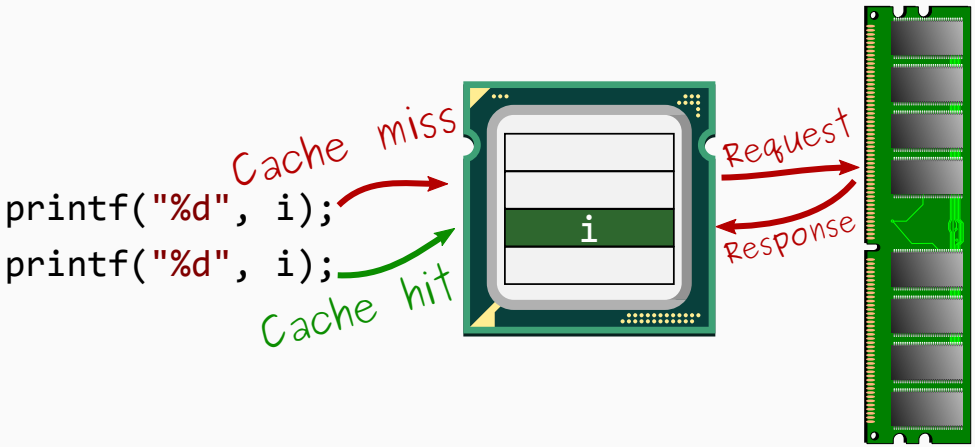


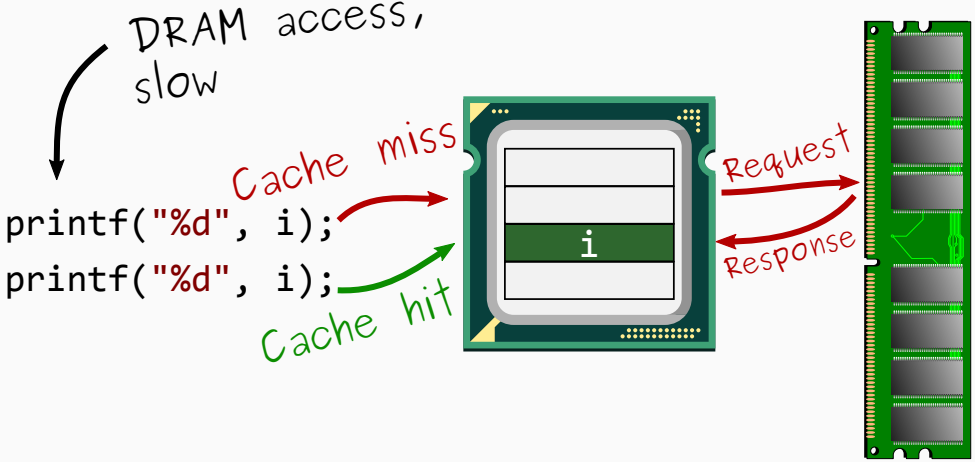
```
printf("%d", i);
printf("%d", i);
```

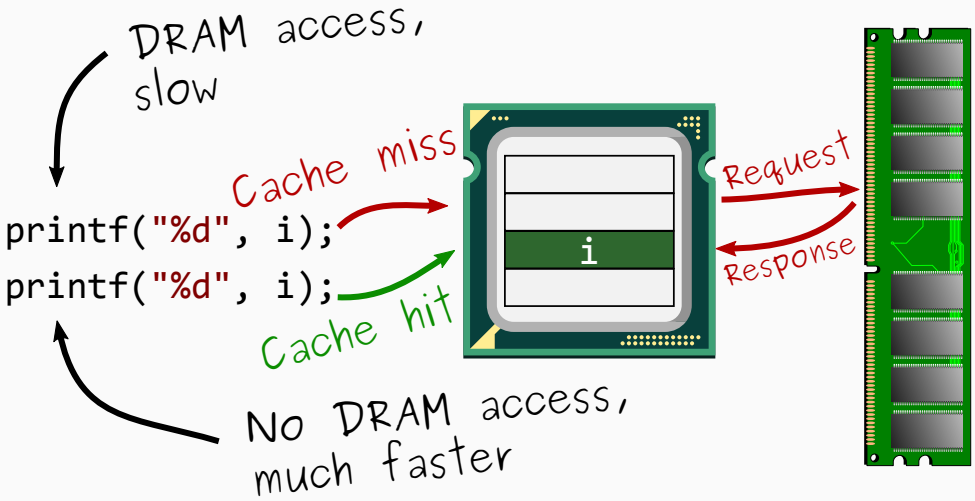


```
printf("%d", i);
printf("%d", i);
```




printf("%d", i);
printf("%d", i);
Cache miss
Cache hit
i
Request
Response


DRAM access, slow
printf("%d", i);
printf("%d", i);
Cache miss
Cache hit
i
Request
Response

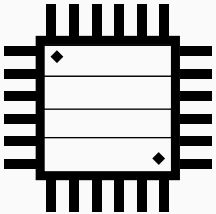

DRAM access,
slow
printf("%d", i);
printf("%d", i);
Cache miss
Cache hit
i
Request
Response
No DRAM access,
much faster

Shared Memory

ATTACKER

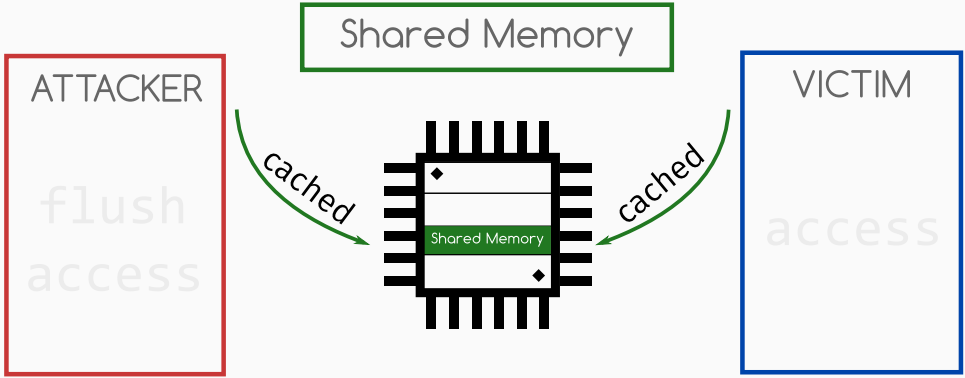
flush

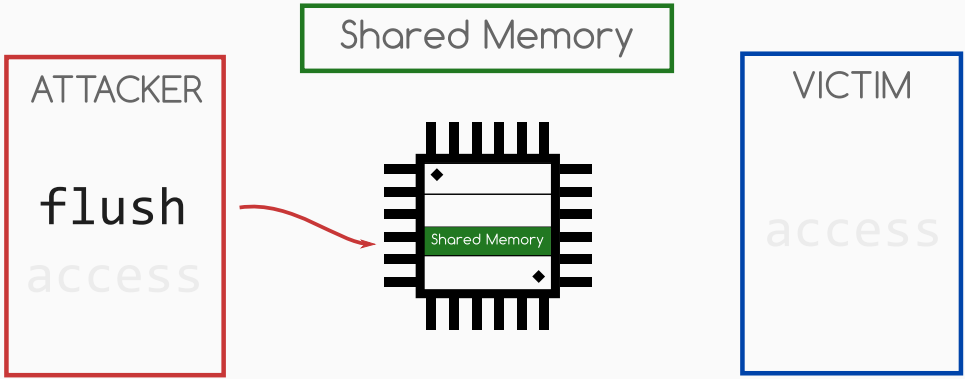
access

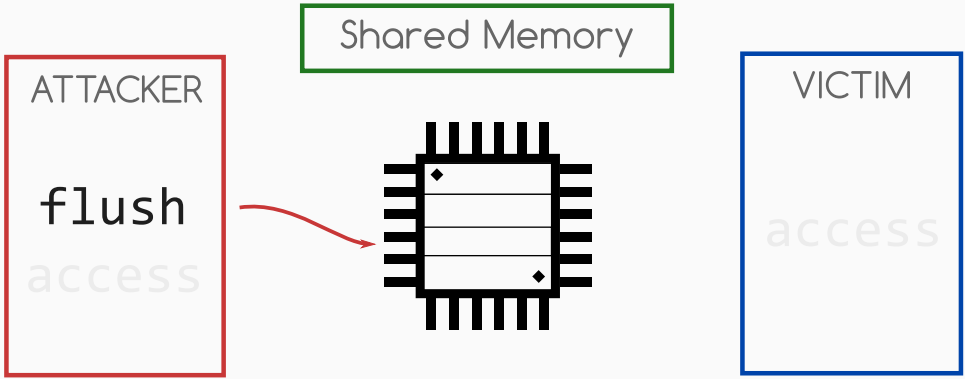


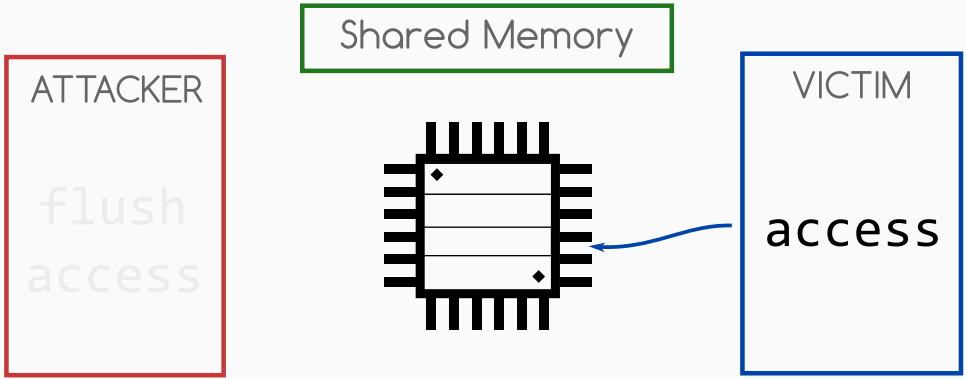
VICTIM

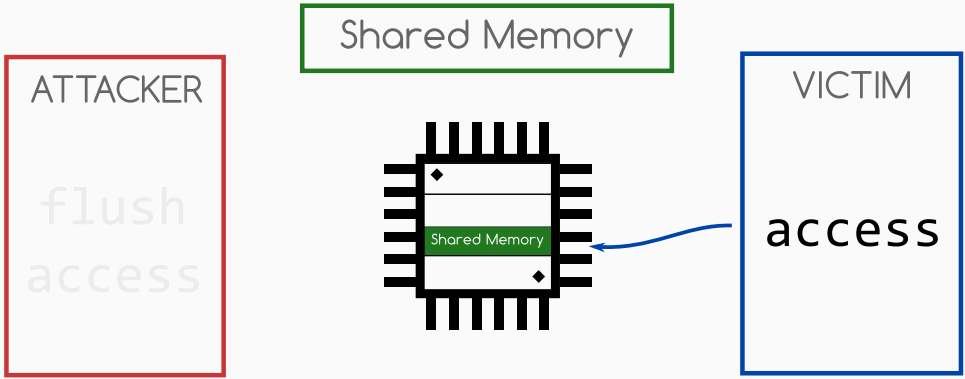
access

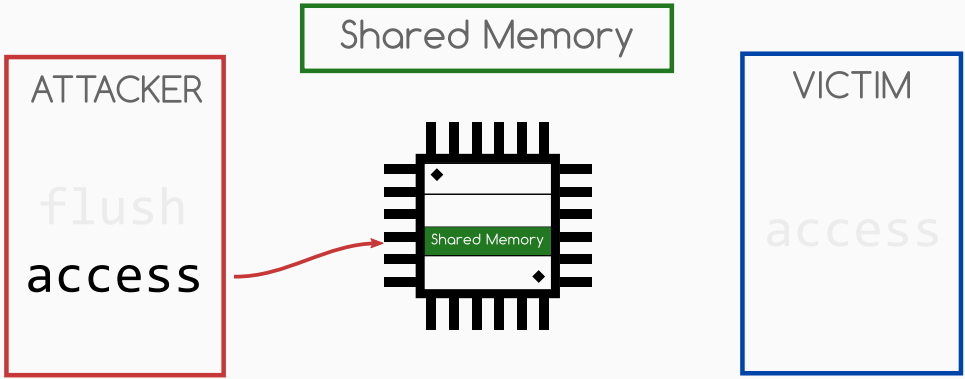

Shared Memory
ATTACKER
flush
access
cached
Shared Memory
cached
VICTIM
access

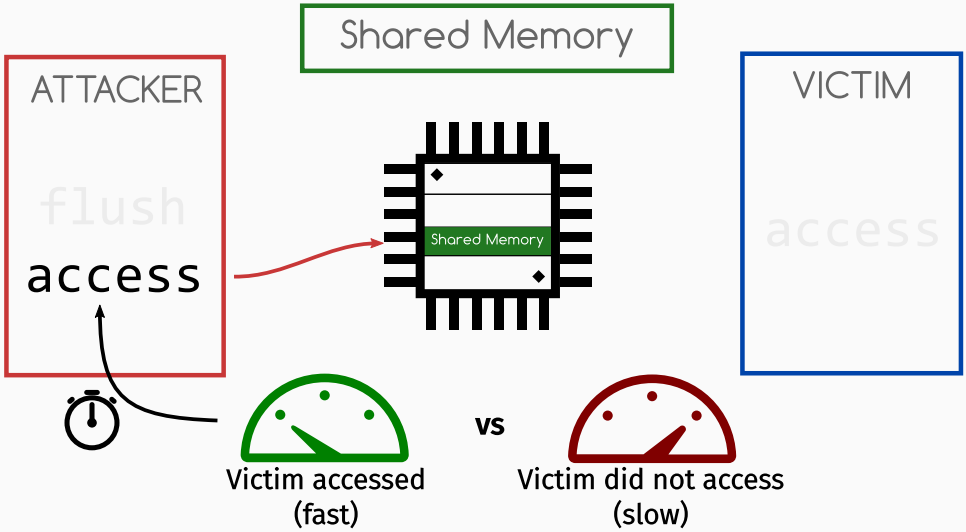

Shared Memory
ATTACKER
flush
access
Shared Memory
VICTIM
access


Shared Memory
ATTACKER
flush
access
VICTIM
access

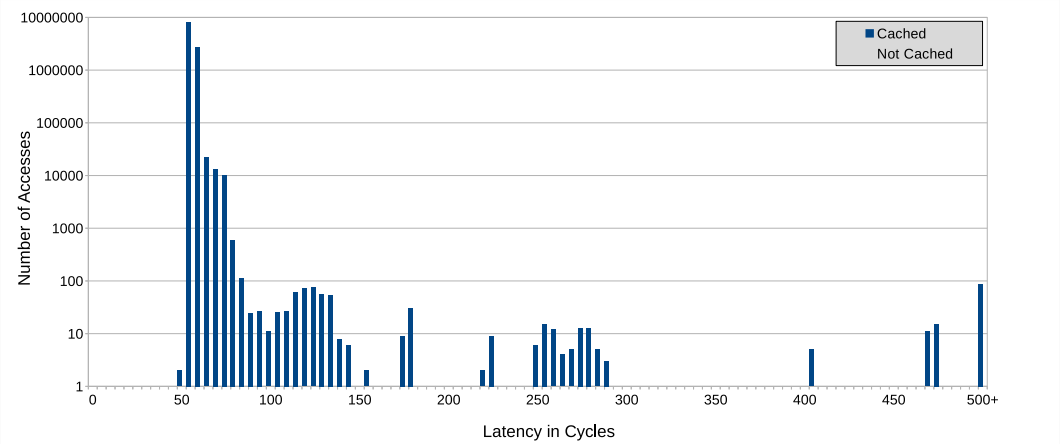

Shared Memory
ATTACKER
flush
access
VICTIM
access


Shared Memory
ATTACKER
flush
access
Shared Memory
VICTIM
access

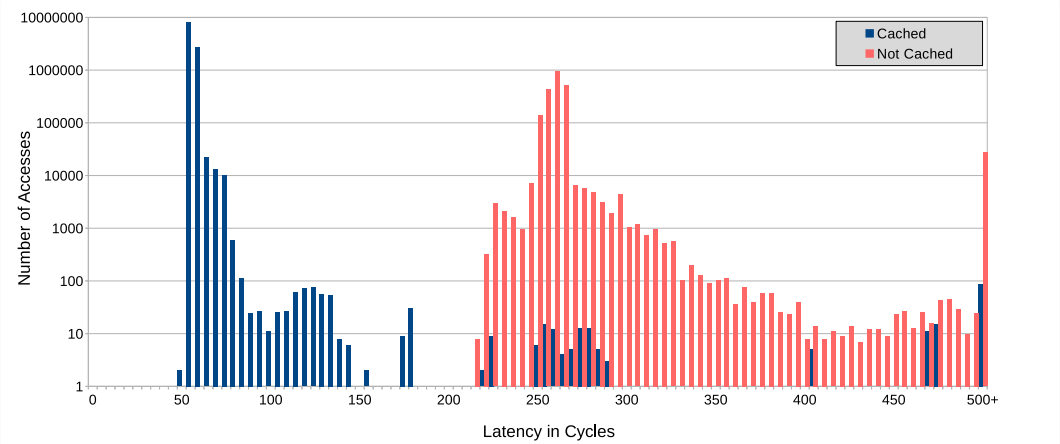

Shared Memory
ATTACKER
flush
access
Shared Memory
VICTIM
access


Shared Memory
ATTACKER
flush
access
Shared Memory
VICTIM
access
vs
Victim accessed
(fast)
Victim did not access
(slow)

Memory Access Latency



Memory Access Latency



Cache Template Attacks

Profiling Phase

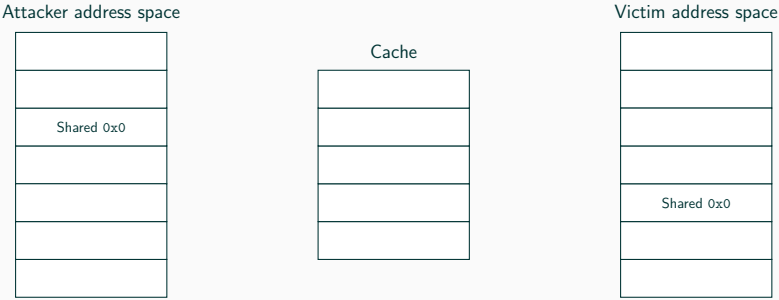
- Preprocessing step to find exploitable addresses automatically
  - w.r.t. “events” (keystrokes, encryptions, ...)
  - Called “Cache Template”

Profiling Phase

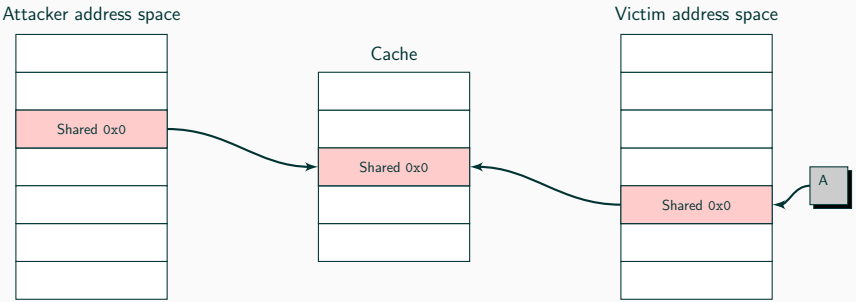
- Preprocessing step to find exploitable addresses automatically
  - w.r.t. “events” (keystrokes, encryptions, ...)
  - Called “Cache Template”

Exploitation Phase

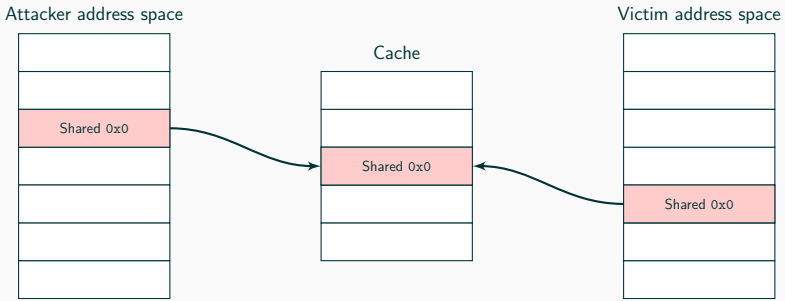
- Monitor exploitable addresses



Cache is empty

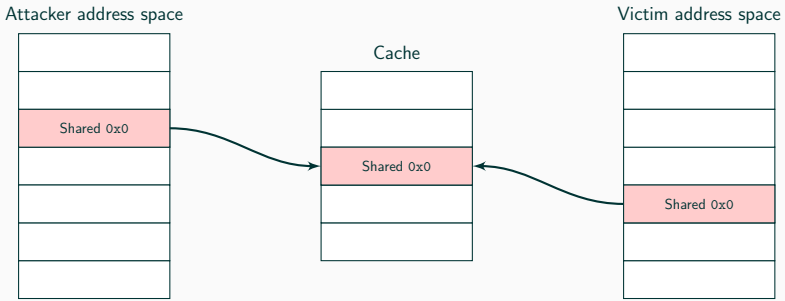


Attacker triggers an event

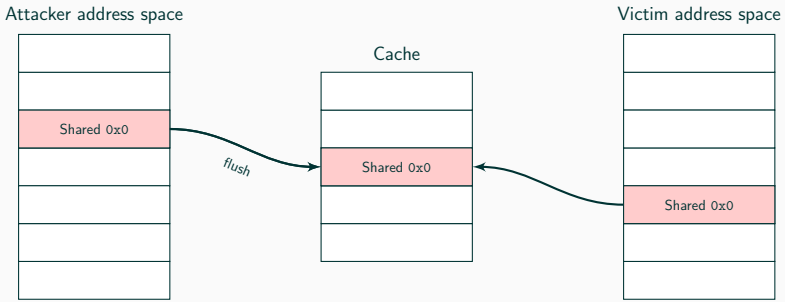


Attacker checks one address for cache hits ("Reload")

Profiling Phase

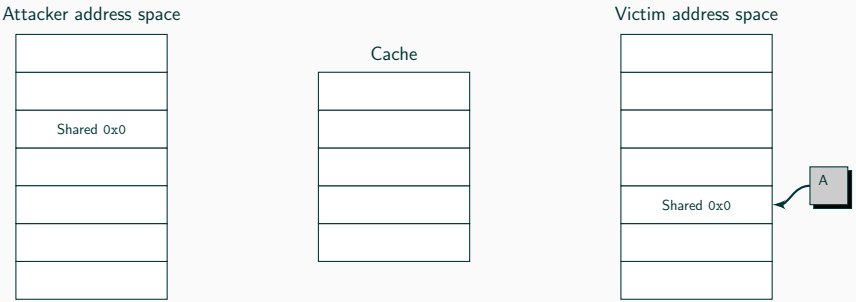


Update cache hit ratio (per event and address)

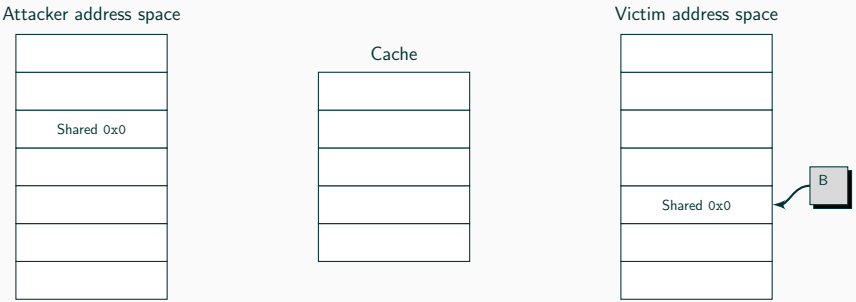


Attacker flushes shared memory

Profiling Phase

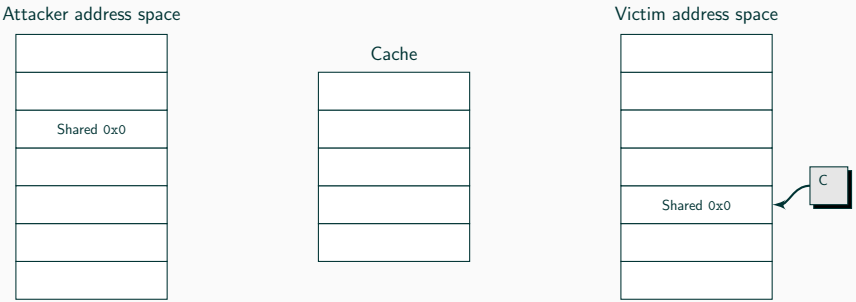


Repeat for higher accuracy

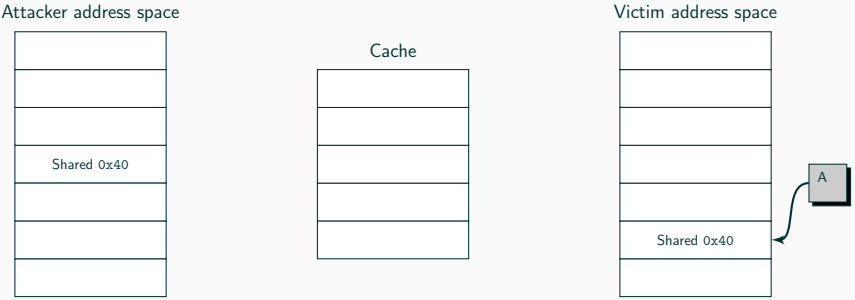


Repeat for all events

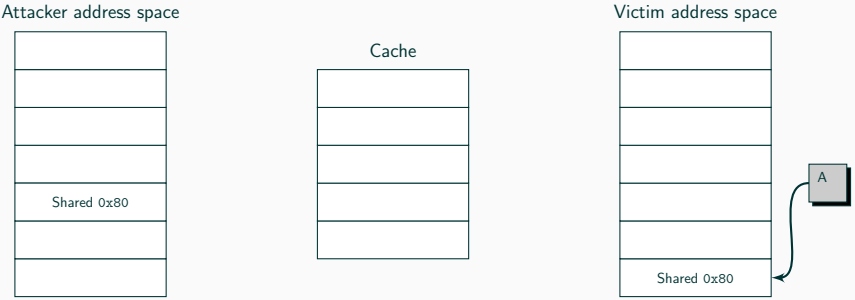
Profiling Phase



Repeat for all events

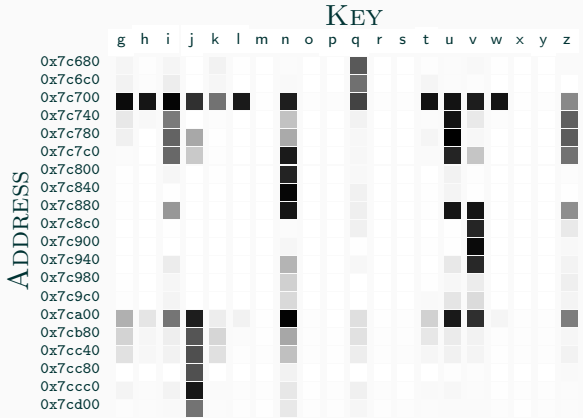


Continue with next address



Continue with next address

Profiling Phase: All Events, All Addresses



- Monitor addresses from Cache Template

- Monitor addresses from Cache Template
- Report to log file / attacker

- Monitor addresses from Cache Template
- Report to log file / attacker
- Manual analysis of log file
  - Find password in keypress log, etc.

AES uses T-Tables (precomputed from S-Boxes)

AES uses T-Tables (precomputed from S-Boxes)

- 4 T-Tables

AES uses T-Tables (precomputed from S-Boxes)

- 4 T-Tables
-

$$T_0\left[k_{\{0,4,8,12\}} \oplus p_{\{0,4,8,12\}}\right]$$

$$T_1\left[k_{\{1,5,9,13\}} \oplus p_{\{1,5,9,13\}}\right]$$

...

AES uses T-Tables (precomputed from S-Boxes)

- 4 T-Tables
- $$T_0\left[k_{\{0,4,8,12\}} \oplus p_{\{0,4,8,12\}}\right]$$
  $$T_1\left[k_{\{1,5,9,13\}} \oplus p_{\{1,5,9,13\}}\right]$$
  ...
- If we know which entry of T is accessed, we know the result of $k_i \oplus p_i$.

AES uses T-Tables (precomputed from S-Boxes)

- 4 T-Tables
- $$T_0\left[k_{\{0,4,8,12\}} \oplus p_{\{0,4,8,12\}}\right]$$
  $$T_1\left[k_{\{1,5,9,13\}} \oplus p_{\{1,5,9,13\}}\right]$$
  ...
- If we know which entry of T is accessed, we know the result of $k_i \oplus p_i$.
- Known-plaintext attack (p_i is known) $\rightarrow$ k_i can be determined

AES T-Table implementation from OpenSSL 1.0.2

AES T-Table implementation from OpenSSL 1.0.2

- Most addresses in two groups:
  - Cache hit ratio 100% (always cache hits)
  - Cache hit ratio 0% (no cache hits)

AES T-Table implementation from OpenSSL 1.0.2

- Most addresses in two groups:
  - Cache hit ratio 100% (always cache hits)
  - Cache hit ratio 0% (no cache hits)
- One 4096 byte memory block:
  - Cache hit ratio of 92%
  - Cache hits depend on key value and plaintext value
  - The T-Tables

AES T-Table implementation from OpenSSL 1.0.2

- Known-plaintext attack

AES T-Table implementation from OpenSSL 1.0.2

- Known-plaintext attack
- Events: encryption with only one fixed key byte

AES T-Table implementation from OpenSSL 1.0.2

- Known-plaintext attack
- Events: encryption with only one fixed key byte
- Profile each event

AES T-Table implementation from OpenSSL 1.0.2

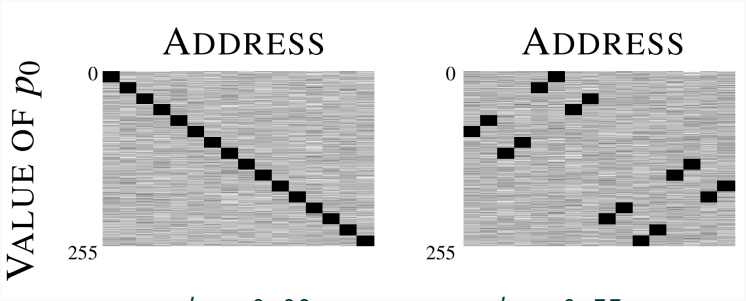
- Known-plaintext attack
- Events: encryption with only one fixed key byte
- Profile each event
- Exploitation phase:
  - Eliminate key candidates

AES T-Table implementation from OpenSSL 1.0.2

- Known-plaintext attack
- Events: encryption with only one fixed key byte
- Profile each event
- Exploitation phase:
  - Eliminate key candidates
  - Reduction of key space in first-round attack:
    - 64 bits after 16–160 encryptions

AES T-Table implementation from OpenSSL 1.0.2

- Known-plaintext attack
- Events: encryption with only one fixed key byte
- Profile each event
- Exploitation phase:
  - Eliminate key candidates
  - Reduction of key space in first-round attack:
    - 64 bits after 16–160 encryptions
  - State of the art: full key recovery after 30000 encryptions

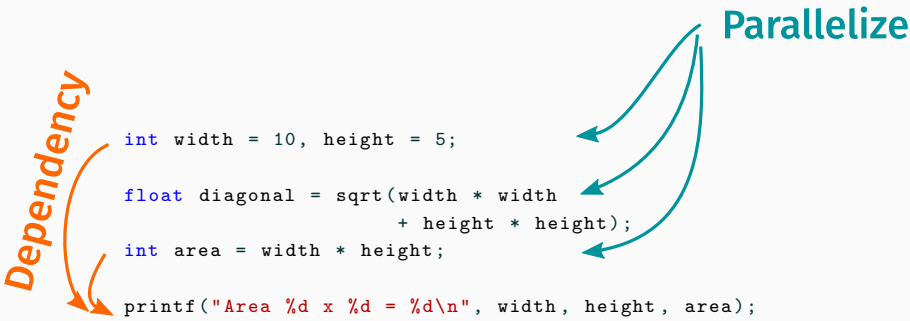


(transposed)

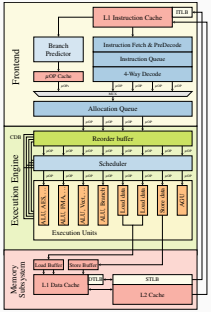
Transient Execution Attacks

Out-of-order Execution

```
int width = 10, height = 5;

float diagonal = sqrt(width * width
                      + height * height);
int area = width * height;

printf("Area %d x %d = %d\n", width, height, area);
```


Parallelize
Dependency
int width = 10, height = 5;
float diagonal = sqrt(width * width
+ height * height);
int area = width * height;
printf("Area %d x %d = %d\n", width, height, area);

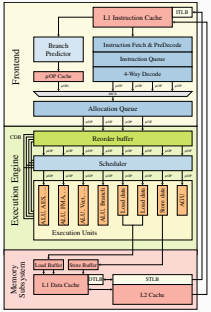
Out-of-Order Execution



Instructions

- are executed out-of-order

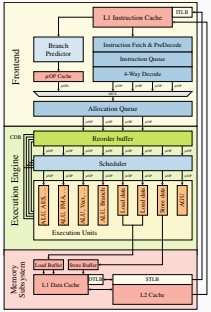
Out-of-Order Execution



Instructions

- are executed out-of-order
- wait until their dependencies are ready

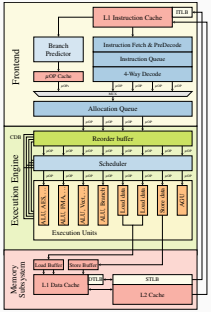
Out-of-Order Execution



Instructions

- are executed out-of-order
- wait until their dependencies are ready
  - Later instructions might execute prior earlier instructions

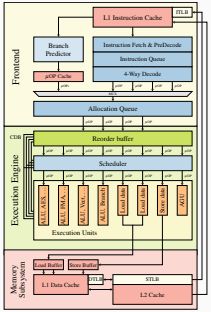
Out-of-Order Execution



Instructions

- are executed out-of-order
- wait until their dependencies are ready
  - Later instructions might execute prior earlier instructions
- retire in-order

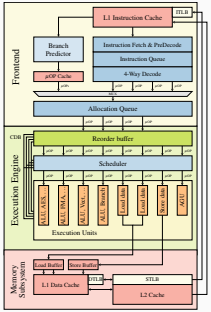
Out-of-Order Execution



Instructions

- are executed out-of-order
- wait until their dependencies are ready
  - Later instructions might execute prior earlier instructions
- retire in-order
  - State becomes architecturally visible

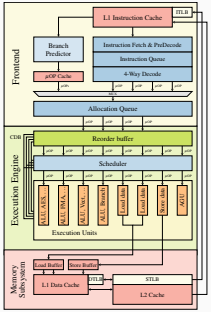
Out-of-Order Execution



Instructions

- are executed out-of-order
- wait until their dependencies are ready
  - Later instructions might execute prior earlier instructions
- retire in-order
  - State becomes architecturally visible
- Exceptions are checked during retirement

Out-of-Order Execution



Instructions

- are executed out-of-order
- wait until their dependencies are ready
  - Later instructions might execute prior earlier instructions
- retire in-order
  - State becomes architecturally visible
- Exceptions are checked during retirement
  - Flush pipeline and recover state

- CPU tries to predict the future (branch predictor), ...

- CPU tries to predict the future (branch predictor), ...
  - ... based on events learned in the past

Speculative Execution

- CPU tries to predict the future (branch predictor), ...
  - ... based on events learned in the past
- Speculative execution of instructions

Speculative Execution

- CPU tries to predict the future (branch predictor), ...
  - ... based on events learned in the past
- Speculative execution of instructions
- If the prediction was correct, ...

- CPU tries to predict the future (branch predictor), ...
  - ... based on events learned in the past
- Speculative execution of instructions
- If the prediction was correct, ...
  - ... very fast

- CPU tries to predict the future (branch predictor), ...
  - ... based on events learned in the past
- Speculative execution of instructions
- If the prediction was correct, ...
  - ... very fast
  - otherwise: Discard results

- CPU tries to predict the future (branch predictor), ...
  - ... based on events learned in the past
- Speculative execution of instructions
- If the prediction was correct, ...
  - ... very fast
  - otherwise: Discard results
- Measurable side-effects?

Transient: What's that?

- Out-of-order/speculatively executed instructions leave microarchitectural traces

Transient: What's that?

- Out-of-order/speculatively executed instructions leave microarchitectural traces
  - We can see them for example in the cache

Transient: What's that?

- Out-of-order/speculatively executed instructions leave microarchitectural traces
  - We can see them for example in the cache
- Give such instructions a name: transient instructions

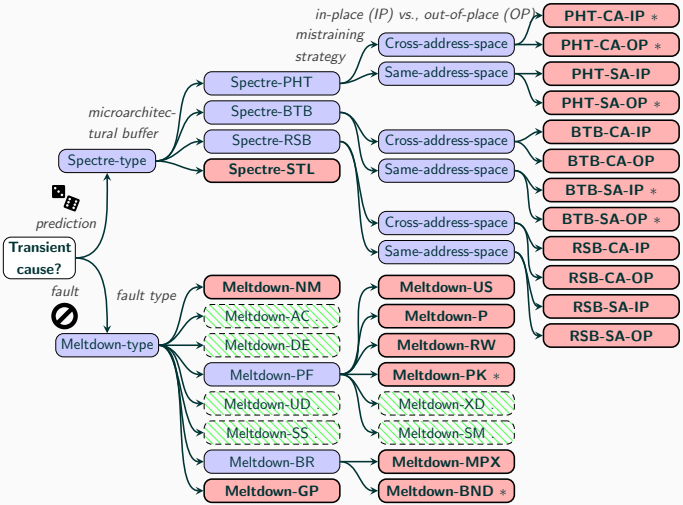
Transient: What's that?

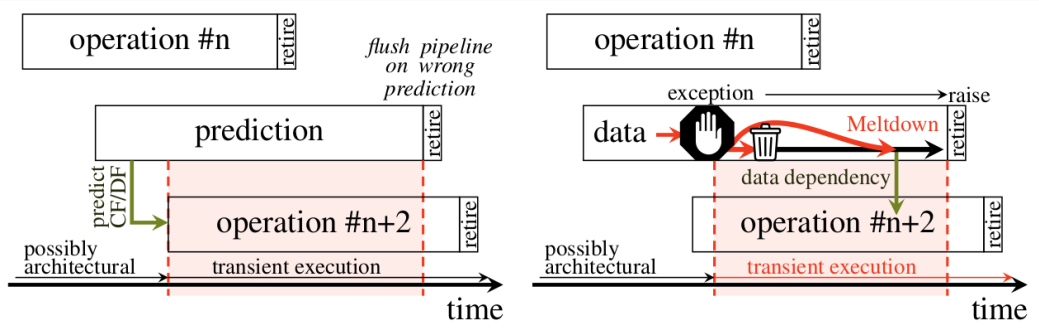
- Out-of-order/speculatively executed instructions leave microarchitectural traces
  - We can see them for example in the cache
- Give such instructions a name: transient instructions
- We can indirectly observe the execution of transient instructions

MELTDOWN

SPECTRE

Transient Execution Attacks: Classification




operation #n
retire
flush pipeline on wrong prediction
prediction
retire
predict CF/DF
operation #n+2
retire
possibly architectural
transient execution
time
operation #n
retire
exception
raise
data
Meltdown
retire
data dependency
operation #n+2
retire
possibly architectural
transient execution
time

- Attacker loads inaccesibble data into register

- Attacker loads inaccesibble data into register $\rightarrow$ page-fault

- Attacker loads inaccesibble data into register $\rightarrow$ page-fault
- Transient instruction stream accesses cache line based on loaded data

- Attacker loads inaccesibble data into register $\rightarrow$ page-fault
- Transient instruction stream accesses cache line based on loaded data
- Determine accessed cache line using Flush+Reload

- Attacker loads inaccesibble data into register → page-fault
- Transient instruction stream accesses cache line based on loaded data
- Determine accessed cache line using Flush+Reload
- Exploit lazy-enforcement of bits in the page-table entry

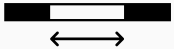
- OS must save registers on context switch

- OS must save registers on context switch → lazy-switching

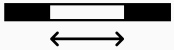
- OS must save registers on context switch $\rightarrow$ lazy-switching
- Next FPU instruction causes #NM exception

- OS must save registers on context switch $\rightarrow$ lazy-switching
- Next FPU instruction causes #NM exception
- Execution continues with previous data in register

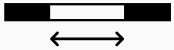
- OS must save registers on context switch $\rightarrow$ lazy-switching
- Next FPU instruction causes #NM exception
- Execution continues with previous data in register
- Extract data using Flush+Reload



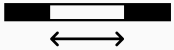
- x86 provides dedicated instruction raising #BR exception if bound-range is exceeded



- x86 provides dedicated instruction raising #BR exception if bound-range is exceeded
- Subsequent data is again used in transient execution



- x86 provides dedicated instruction raising #BR exception if bound-range is exceeded
- Subsequent data is again used in transient execution
- Attacker determines accessed cache line using Flush+Reload



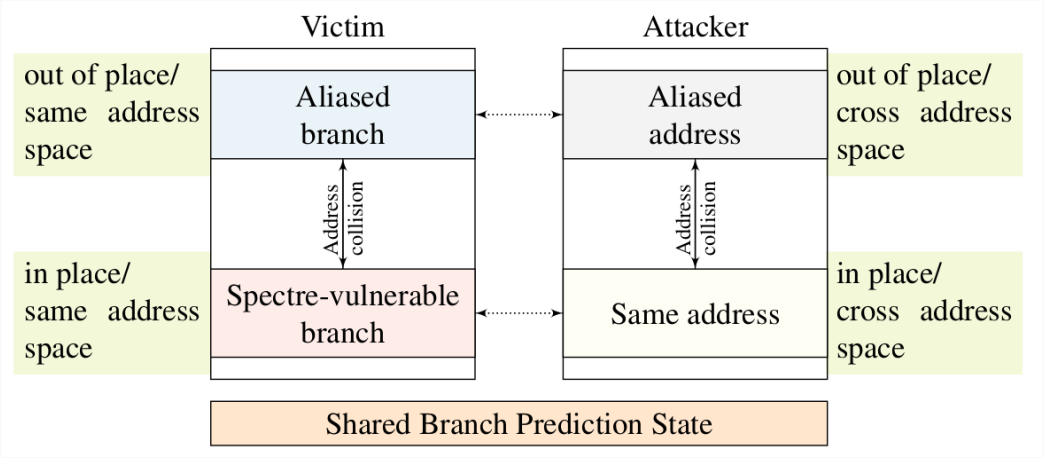
- x86 provides dedicated instruction raising #BR exception if bound-range is exceeded
- Subsequent data is again used in transient execution
- Attacker determines accessed cache line using Flush+Reload
- First Meltdown-type attack on AMD

MELTDOWN

SPECTRE

MELTDOWN

SPECTRE


Victim
Attacker
out of place/ same address space
Aliased branch
Aliased address
out of place/ cross address space
Address collision
Address collision
in place/ same address space
Spectre-vulnerable branch
Same address
in place/ cross address space
Shared Branch Prediction State

```
index = 0;

char* data = "textKEY";

if (index < 4)
```

`LUT[data[index] * 4096]` 0

```
index = 0;
```

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

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

then

else

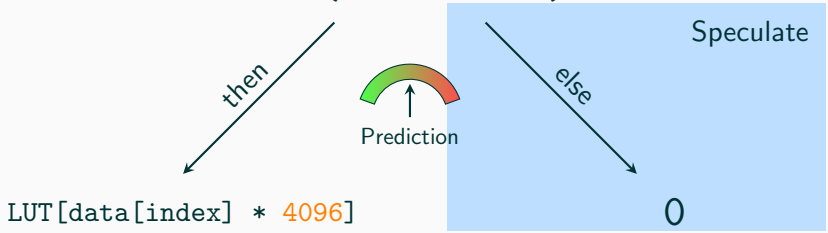
```
LUT[data[index] * 4096]
```

0

```
index = 0;
```

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

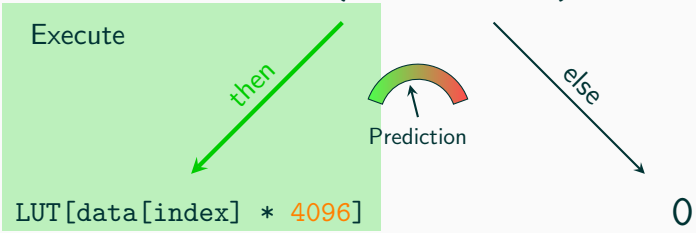
```
if (index < 4)
```



```
index = 0;
```

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

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



```
index = 1;
```

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

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


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

0

```
index = 1;
```

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

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

then

else

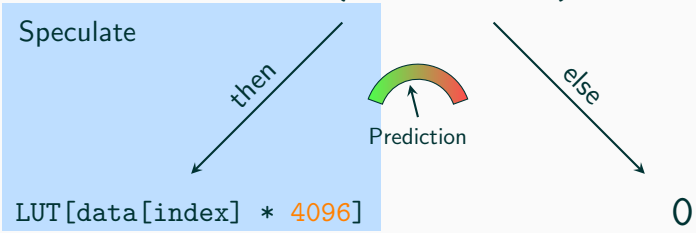
```
LUT[data[index] * 4096]
```

0

```
index = 1;
```

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

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



```
index = 1;
```

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

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

```
index = 2;
```

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

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

```
index = 2;
```

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

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

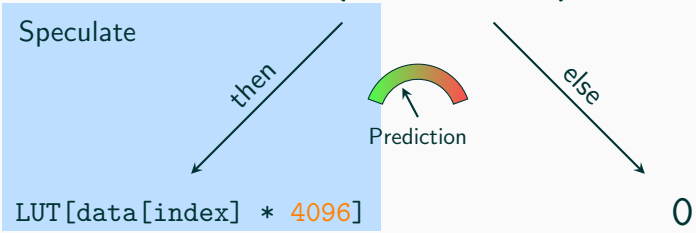

```
LUT[data[index] * 4096]
```

0

```
index = 2;
```

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

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



```
index = 2;
```

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

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

```
index = 3;
```

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

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


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

0

```
index = 3;
```

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

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

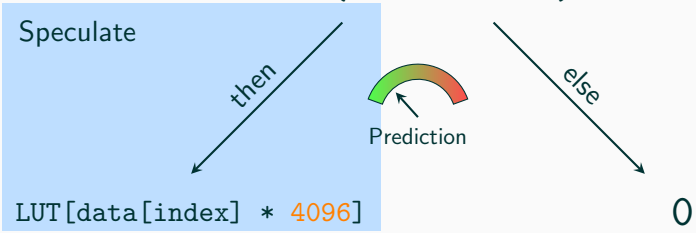

```
LUT[data[index] * 4096]
```

0

```
index = 3;
```

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

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



```
index = 3;
```

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

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

then

Prediction

else

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

0

```
index = 4;
```

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

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


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

0

```
index = 4;
```

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

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

then

else

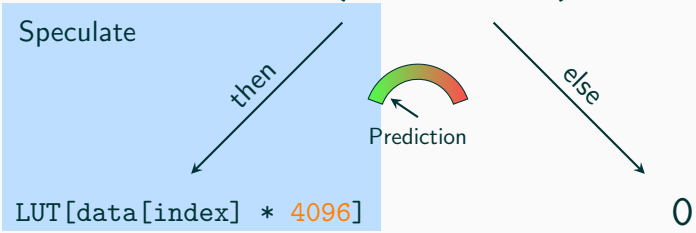
```
LUT[data[index] * 4096]
```

0

```
index = 4;
```

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

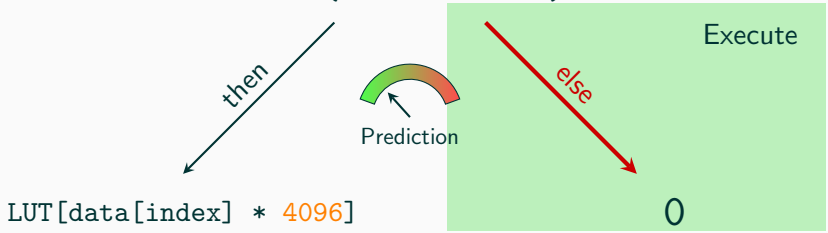
```
if (index < 4)
```



```
index = 4;
```

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

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



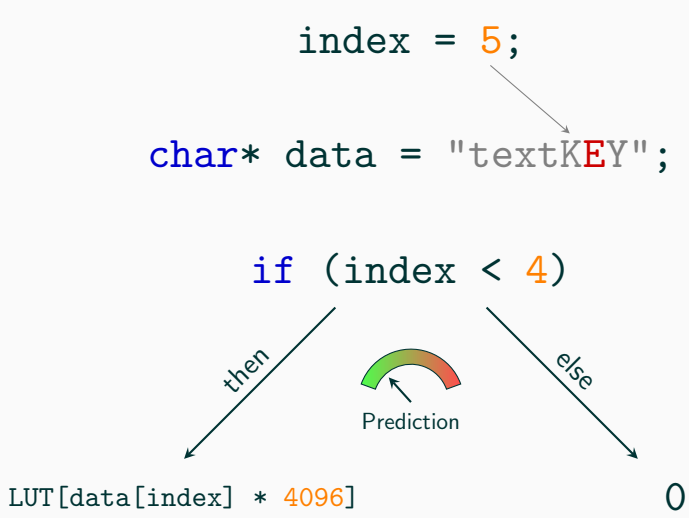
```
index = 5;
```

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

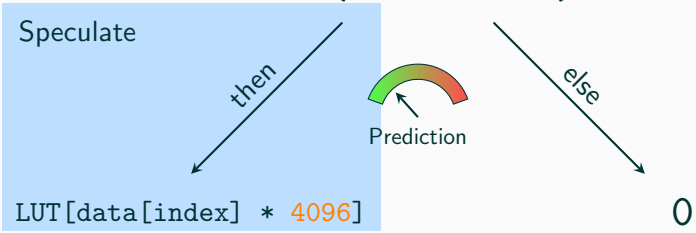
```
if (index < 4)
```


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

0


index = 5;
char* data = "textKEY";
if (index < 4)
then
Prediction
else
LUT[data[index] * 4096]
0

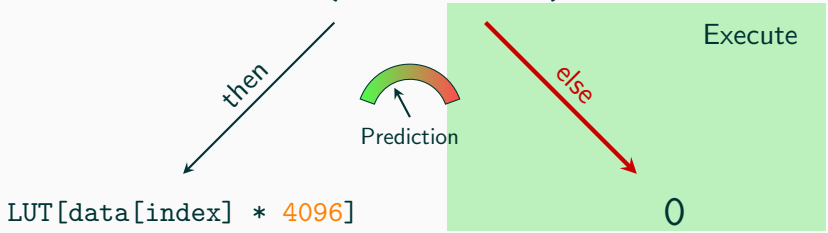
```
index = 5;

char* data = "textKEY";

if (index < 4)
```



```
index = 5;
```

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

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



```
index = 6;

char* data = "textKEY";

if (index < 4)
```

`LUT[data[index] * 4096]` 0

```
index = 6;
```

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

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

then

else

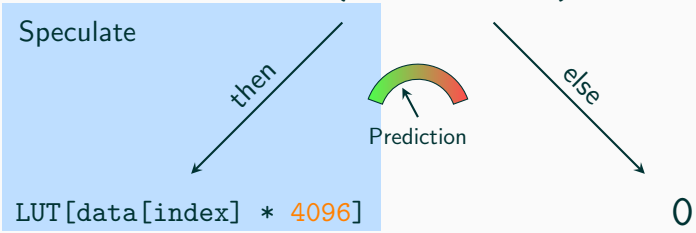
```
LUT[data[index] * 4096]
```

0

```
index = 6;
```

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

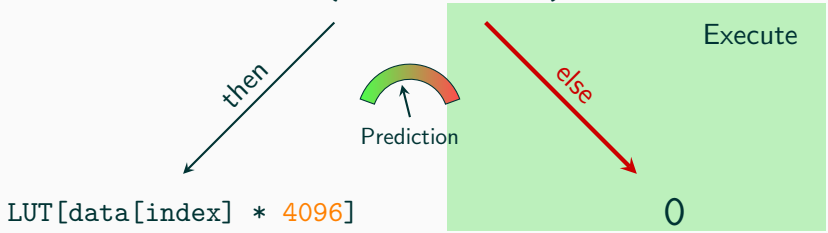
```
if (index < 4)
```



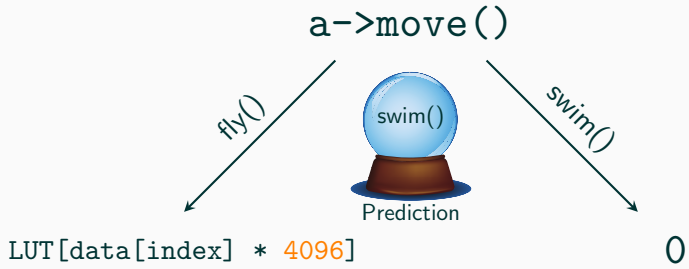
```
index = 6;
```

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

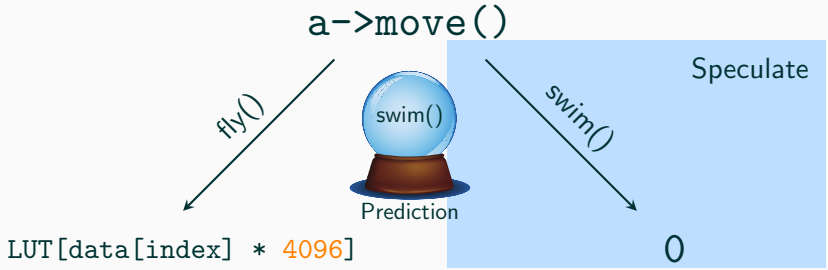
```
if (index < 4)
```



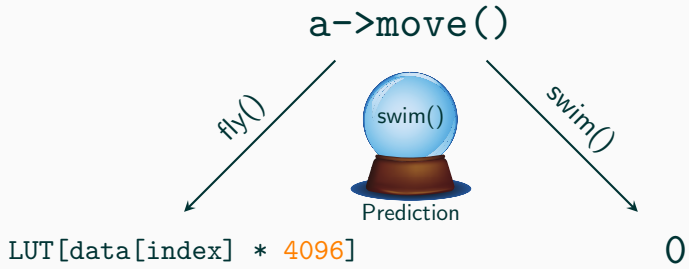
```
Animal* a = bird;
```



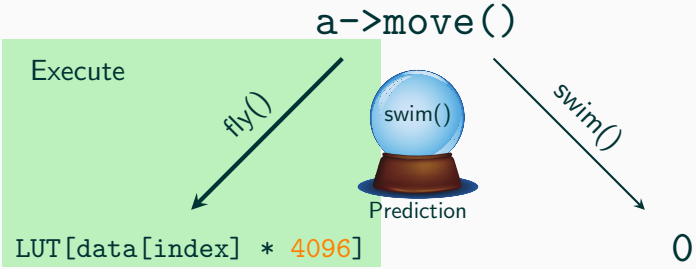
```
Animal* a = bird;
```



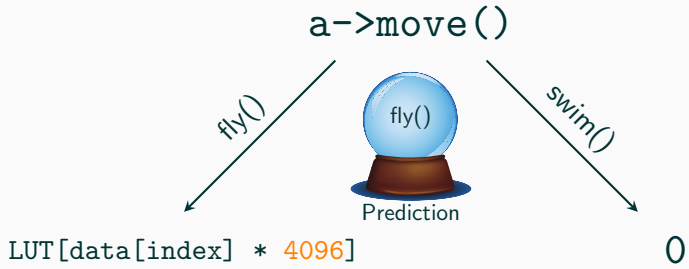
```
Animal* a = bird;
```



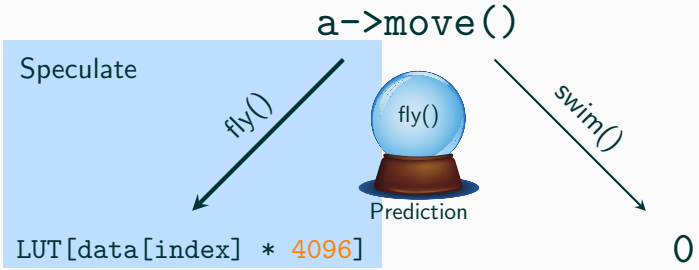
```
Animal* a = bird;
```



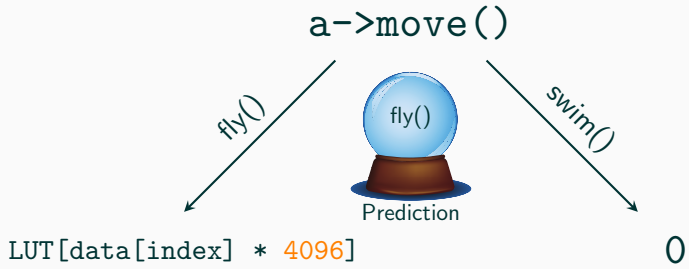
```
Animal* a = bird;
```



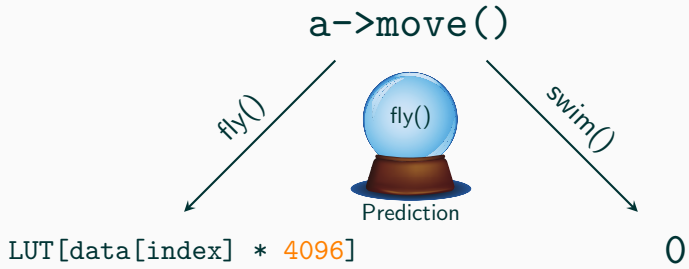
```
Animal* a = bird;
```



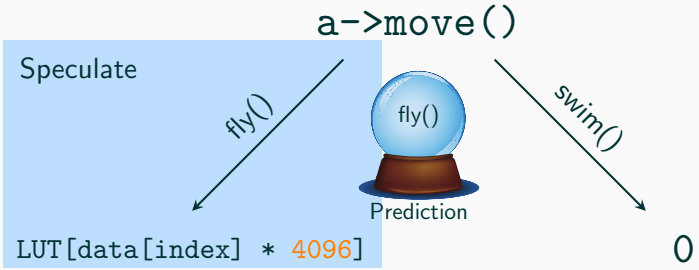
```
Animal* a = bird;
```



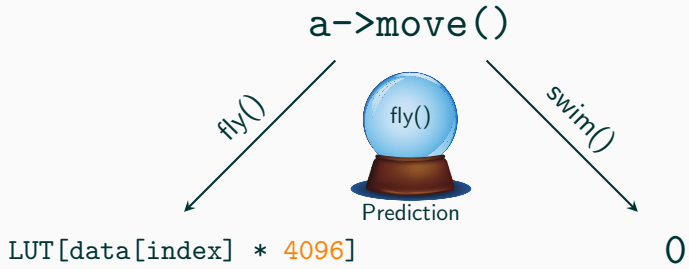
```
Animal* a = fish;
```



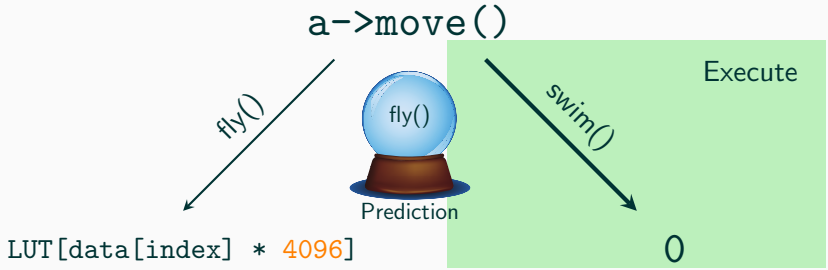
```
Animal* a = fish;
```



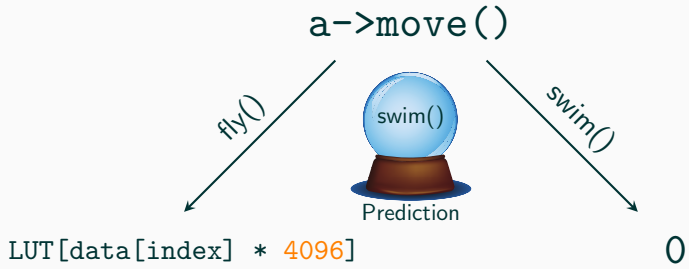
```
Animal* a = fish;
```



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



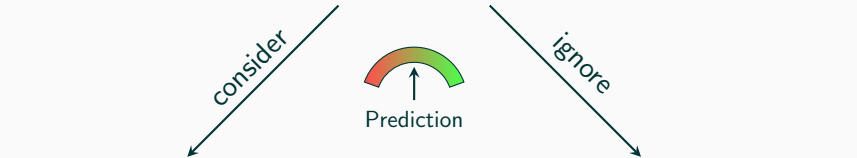
```
Animal* a = fish;
```

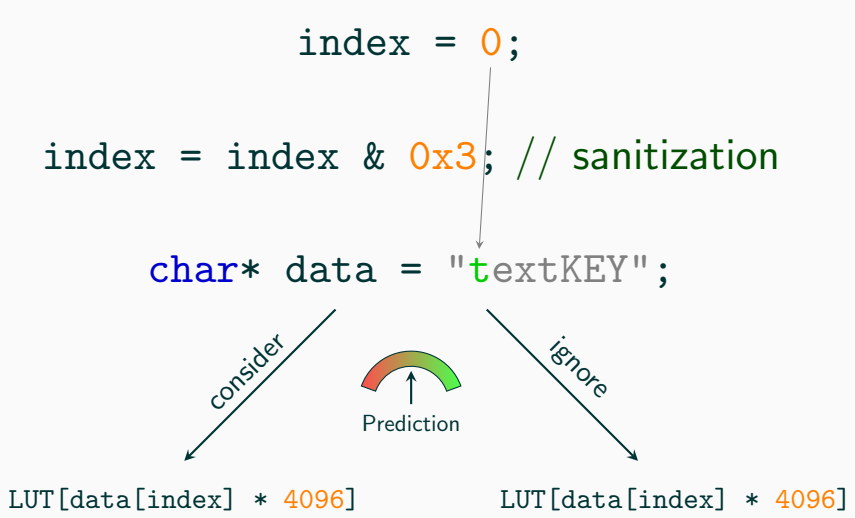


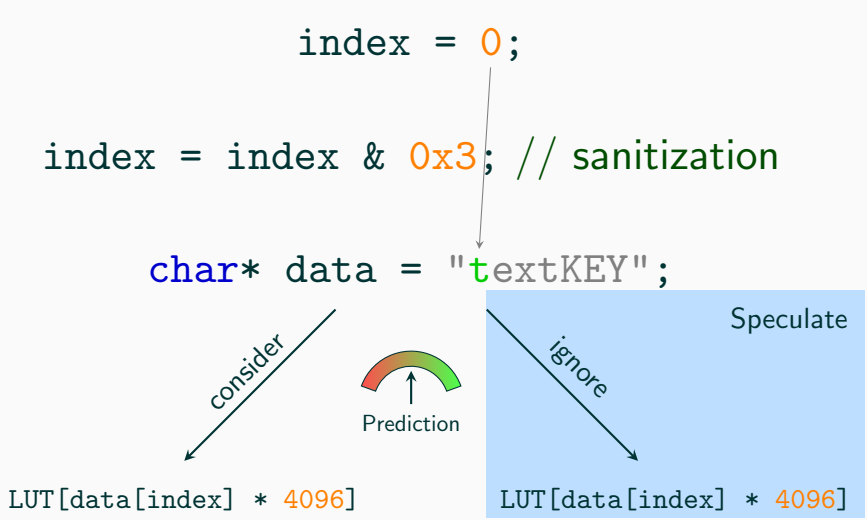
```
index = 0;
```

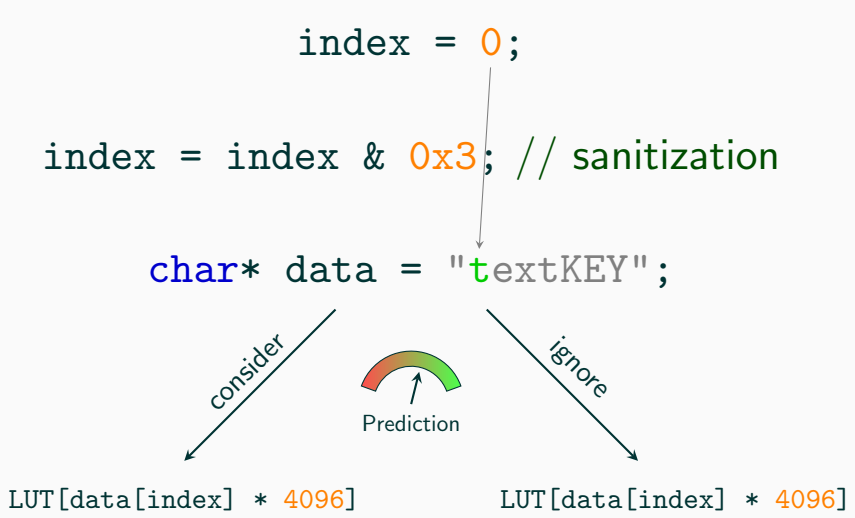
```
index = index & 0x3; // sanitization
```

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




index = 0;
index = index & 0x3; // sanitization
char* data = "textKEY";
consider
Prediction
ignore
LUT[data[index] * 4096]
LUT[data[index] * 4096]

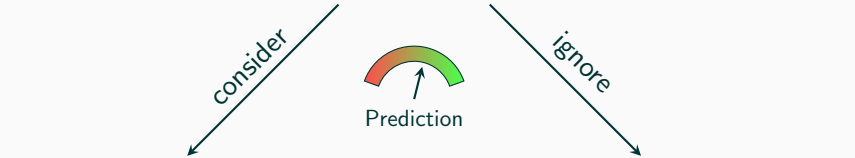

index = 0;
index = index & 0x3; // sanitization
char* data = "textKEY";
consider
Prediction
Speculate
ignore
LUT[data[index] * 4096]
LUT[data[index] * 4096]


index = 0;
index = index & 0x3; // sanitization
char* data = "textKEY";
consider
Prediction
ignore
LUT[data[index] * 4096]
LUT[data[index] * 4096]

```
index = 1;
```

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

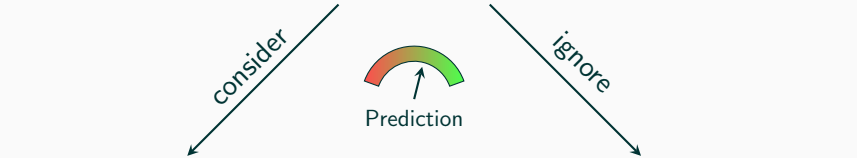
```
char* data = "textKEY";
```



```
index = 1;
```

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

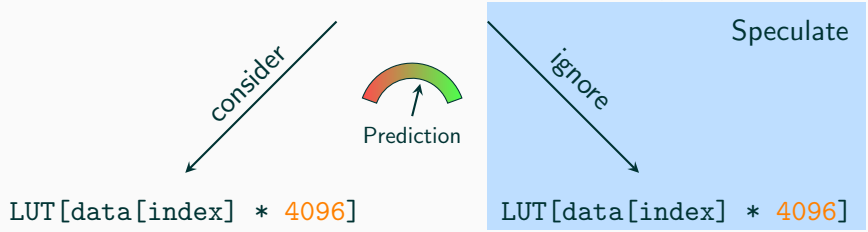
```
char* data = "textKEY";
```



```
index = 1;
```

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

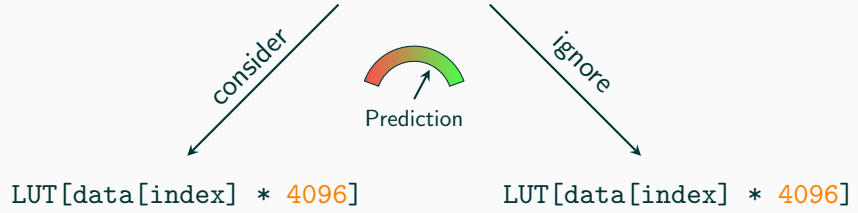
```
char* data = "textKEY";
```



```
index = 1;
```

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

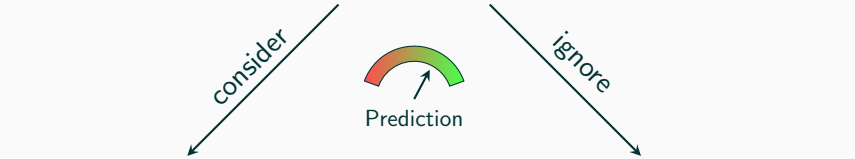
```
char* data = "textKEY";
```



```
index = 2;
```

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

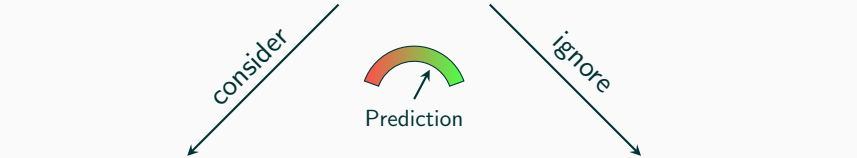
```
char* data = "textKEY";
```

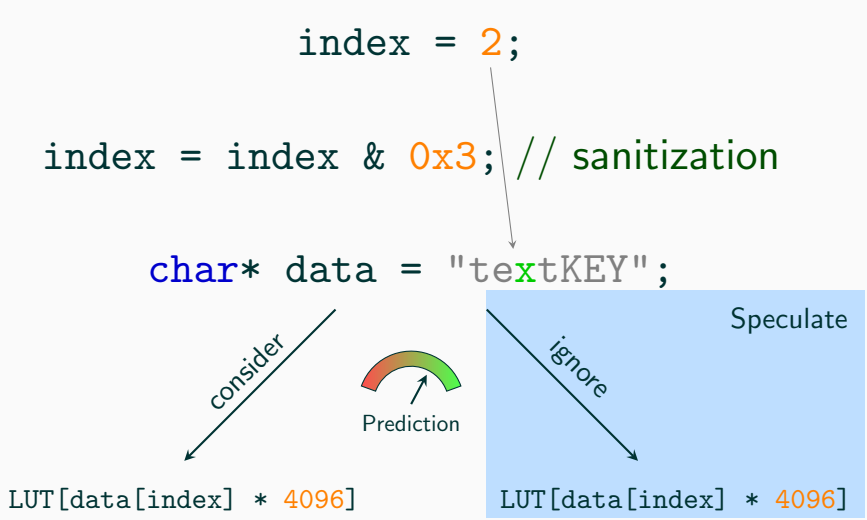


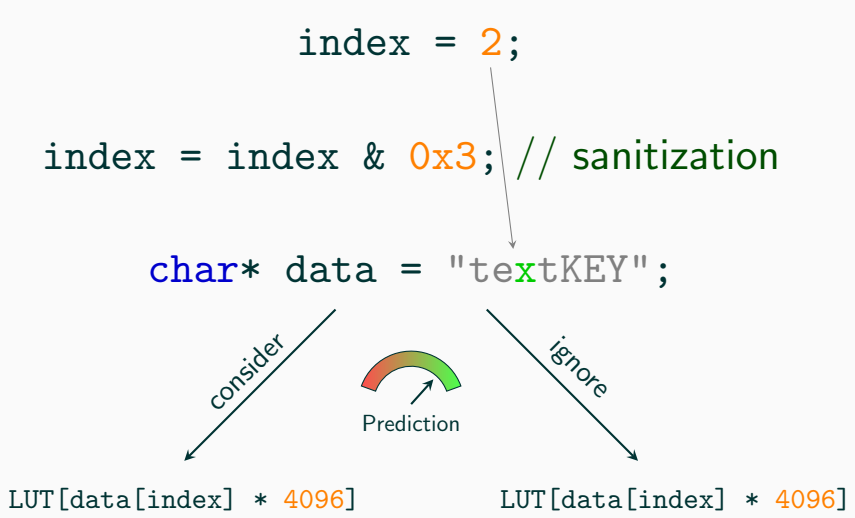
```
index = 2;
```

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

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



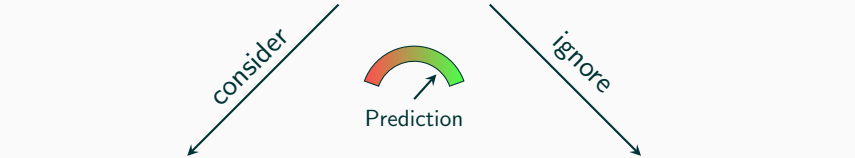

index = 2;
index = index & 0x3; // sanitization
char* data = "textKEY";
consider
Prediction
Speculate
ignore
LUT[data[index] * 4096]
LUT[data[index] * 4096]


index = 2;
index = index & 0x3; // sanitization
char* data = "textKEY";
consider
Prediction
ignore
LUT[data[index] * 4096]
LUT[data[index] * 4096]

```
index = 3;
```

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

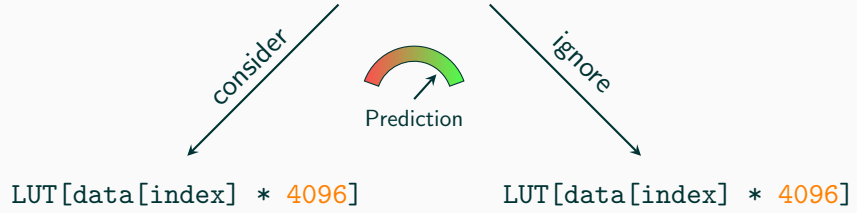
```
char* data = "textKEY";
```

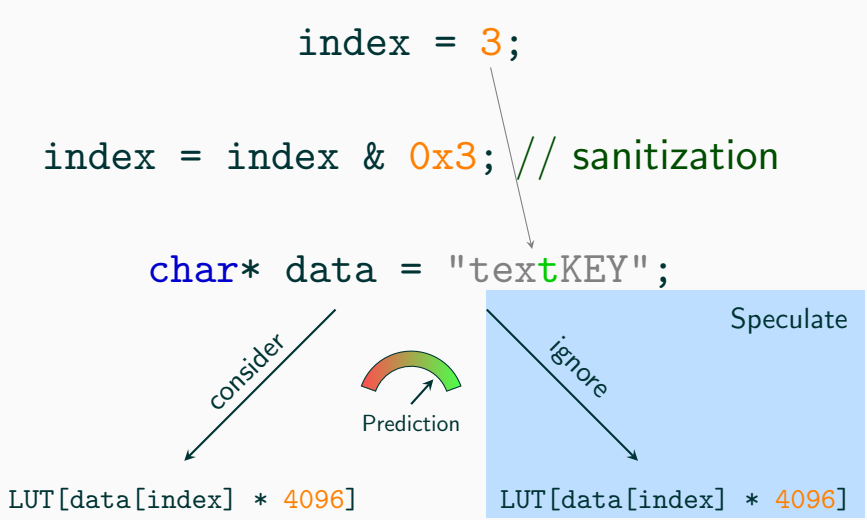


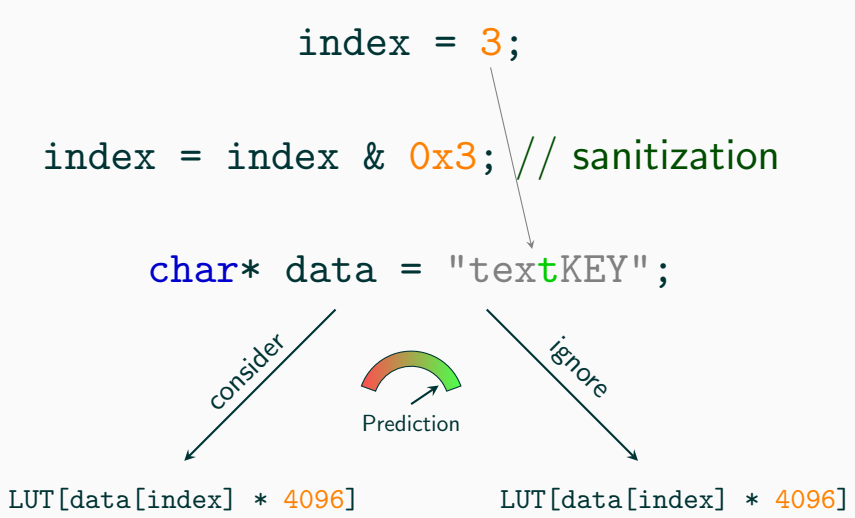
```
index = 3;
```

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

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



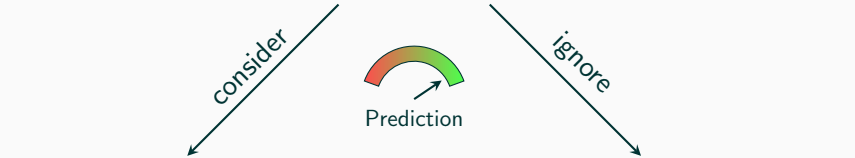

index = 3;
index = index & 0x3; // sanitization
char* data = "textKEY";
consider
Prediction
Speculate
ignore
LUT[data[index] * 4096]
LUT[data[index] * 4096]

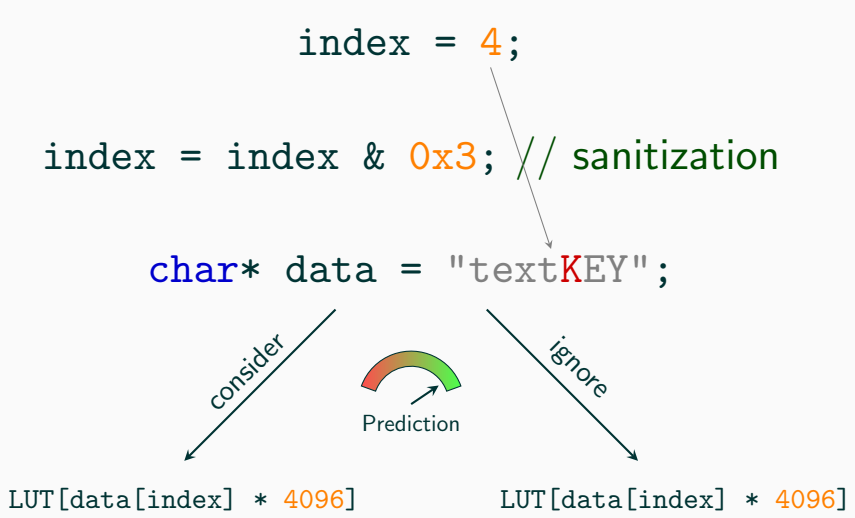

index = 3;
index = index & 0x3; // sanitization
char* data = "textKEY";
consider
Prediction
ignore
LUT[data[index] * 4096]
LUT[data[index] * 4096]

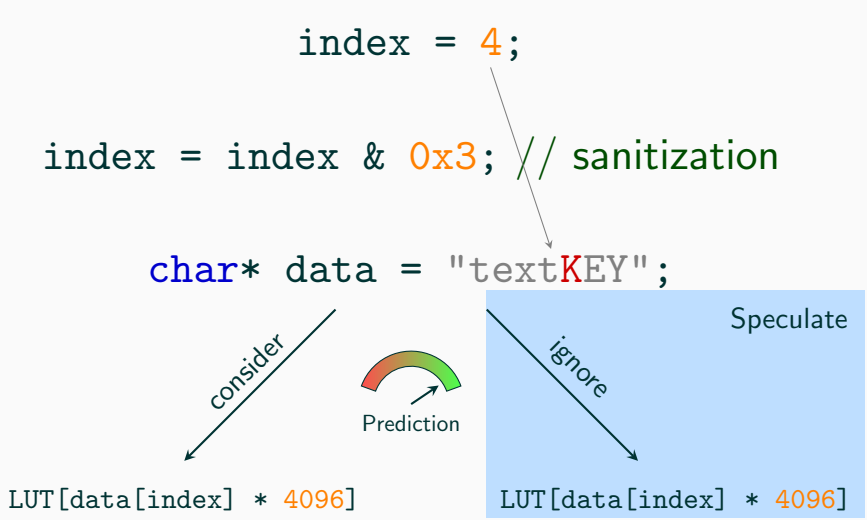
```
index = 4;
```

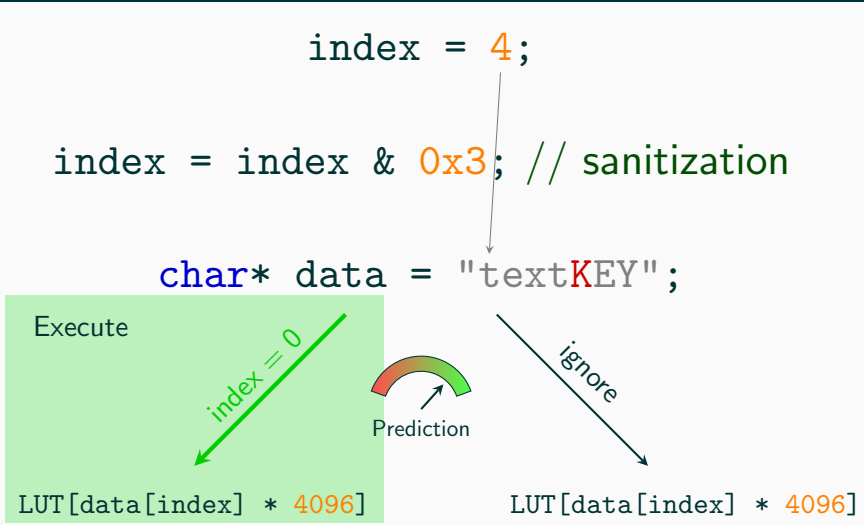
```
index = index & 0x3; // sanitization
```

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




index = 4;
index = index & 0x3; // sanitization
char* data = "textKEY";
consider
Prediction
ignore
LUT[data[index] * 4096]
LUT[data[index] * 4096]

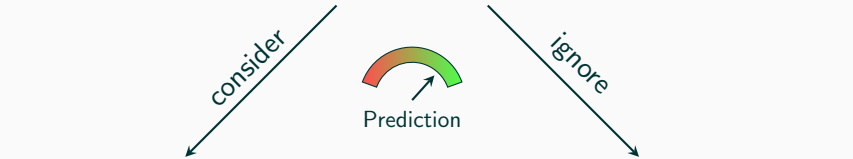

index = 4;
index = index & 0x3; // sanitization
char* data = "textKEY";
consider
Prediction
Speculate
ignore
LUT[data[index] * 4096]
LUT[data[index] * 4096]

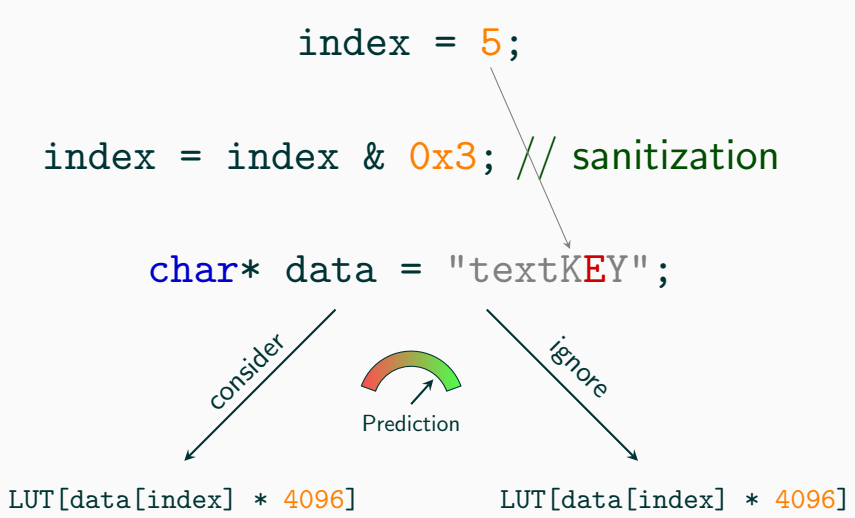

index = 4;
index = index & 0x3; // sanitization
char* data = "textKEY";
Execute
index = 0
Prediction
ignore
LUT[data[index] * 4096]
LUT[data[index] * 4096]

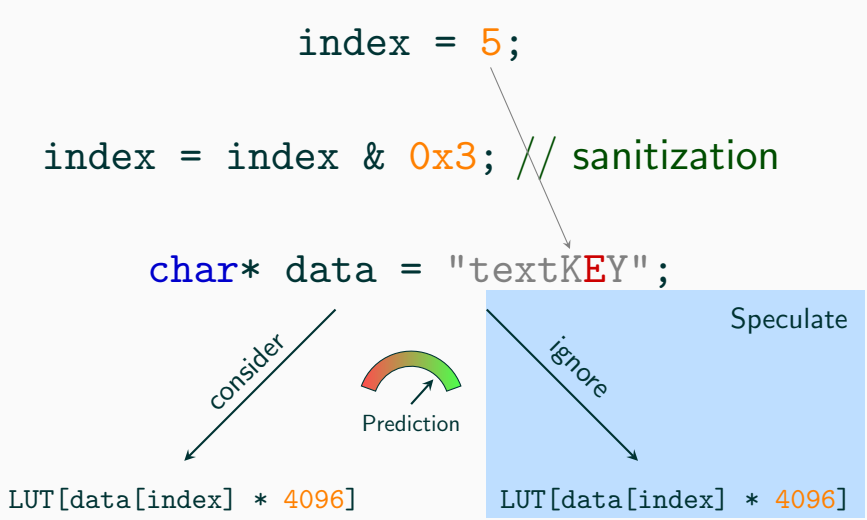
```
index = 5;
```

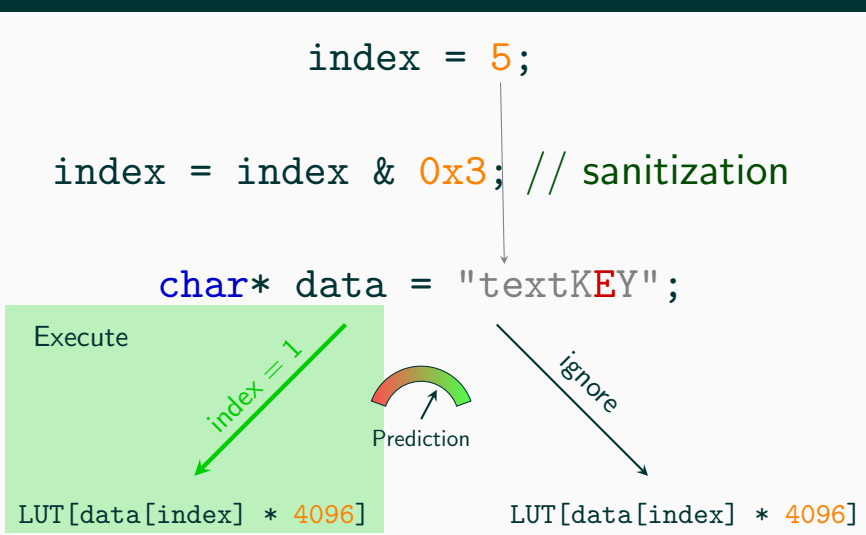
```
index = index & 0x3; // sanitization
```

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




index = 5;
index = index & 0x3; // sanitization
char* data = "textKEY";
consider
Prediction
ignore
LUT[data[index] * 4096]
LUT[data[index] * 4096]

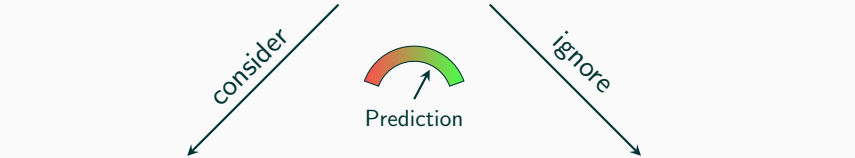

index = 5;
index = index & 0x3; // sanitization
char* data = "textKEY";
consider
Prediction
Speculate
ignore
LUT[data[index] * 4096]
LUT[data[index] * 4096]

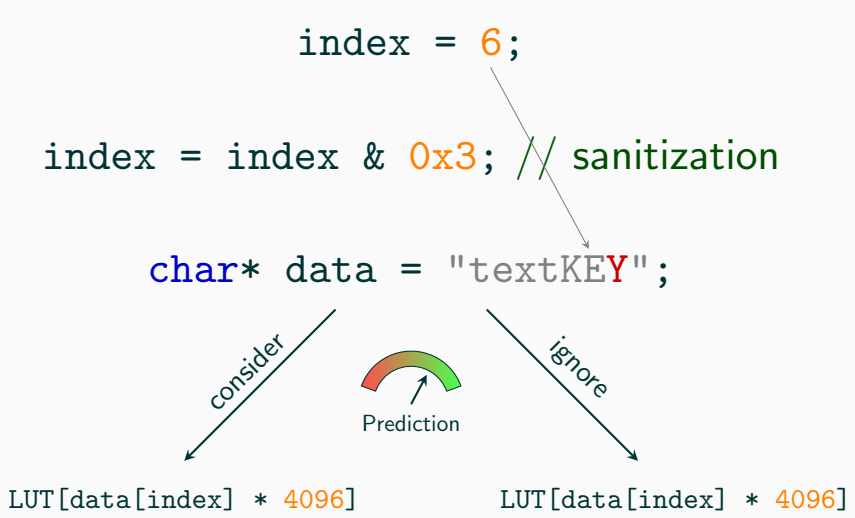

index = 5;
index = index & 0x3; // sanitization
char* data = "textKEY";
Execute
index = 1
Prediction
ignore
LUT[data[index] * 4096]
LUT[data[index] * 4096]

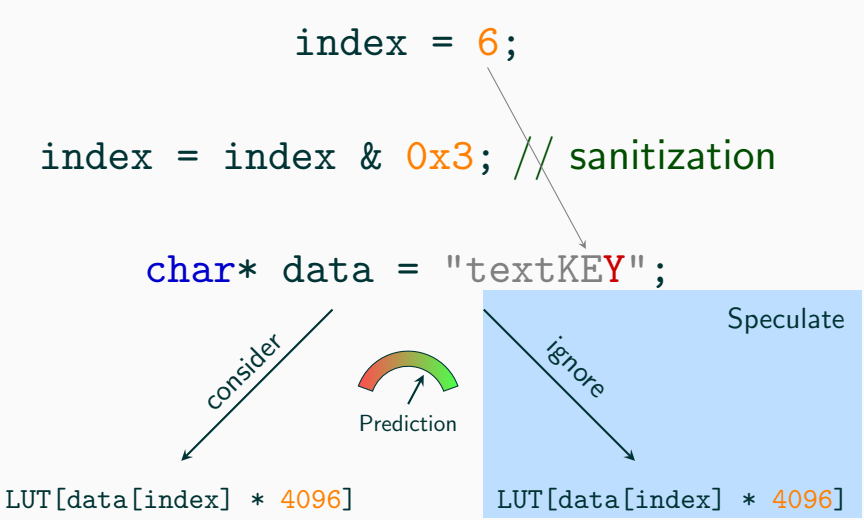
```
index = 6;
```

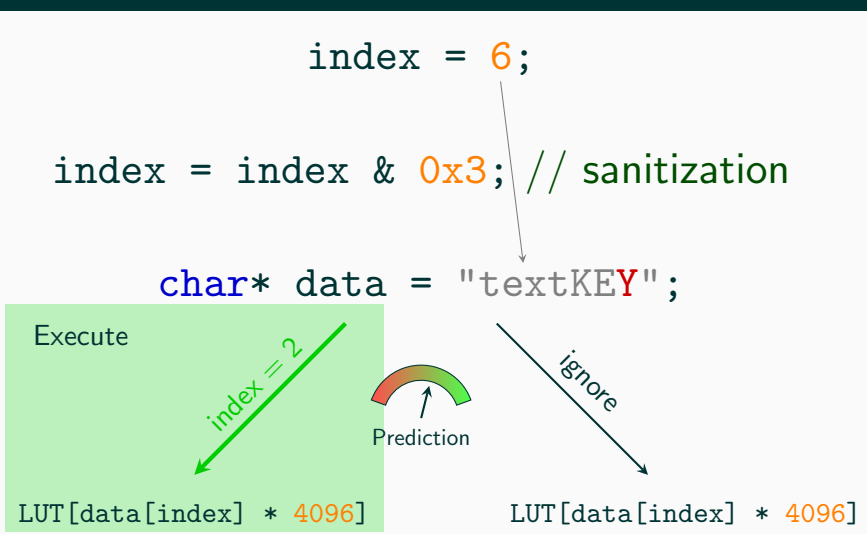
```
index = index & 0x3; // sanitization
```

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




index = 6;
index = index & 0x3; // sanitization
char* data = "textKEY";
consider
Prediction
ignore
LUT[data[index] * 4096]
LUT[data[index] * 4096]

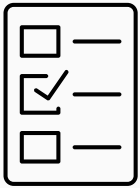

index = 6;
index = index & 0x3; // sanitization
char* data = "textKEY";
consider
Prediction
Speculate
ignore
LUT[data[index] * 4096]
LUT[data[index] * 4096]


index = 6;
index = index & 0x3; // sanitization
char* data = "textKEY";
Execute
index = 2
Prediction
ignore
LUT[data[index] * 4096]
LUT[data[index] * 4096]

- Small hardware stack storing return addresses of recent calls

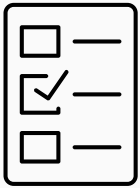
- Small hardware stack storing return addresses of recent calls
- Redirect an indirect branch (a return in this case)

- Small hardware stack storing return addresses of recent calls
- Redirect an indirect branch (a return in this case)
- Fill buffer with “wrong” values



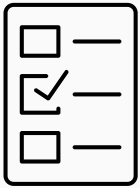
- Transient execution attacks present a new class of attacks

Conclusion

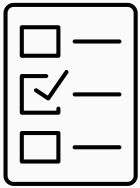


- Transient execution attacks present a new class of attacks
- Optimizations often have security implications

Conclusion



- Transient execution attacks present a new class of attacks
- Optimizations often have security implications
- Many problems to solve around transient execution attacks



- Transient execution attacks present a new class of attacks
- Optimizations often have security implications
- Many problems to solve around transient execution attacks
  $\rightarrow$ still no satisfying solution

Thank you!

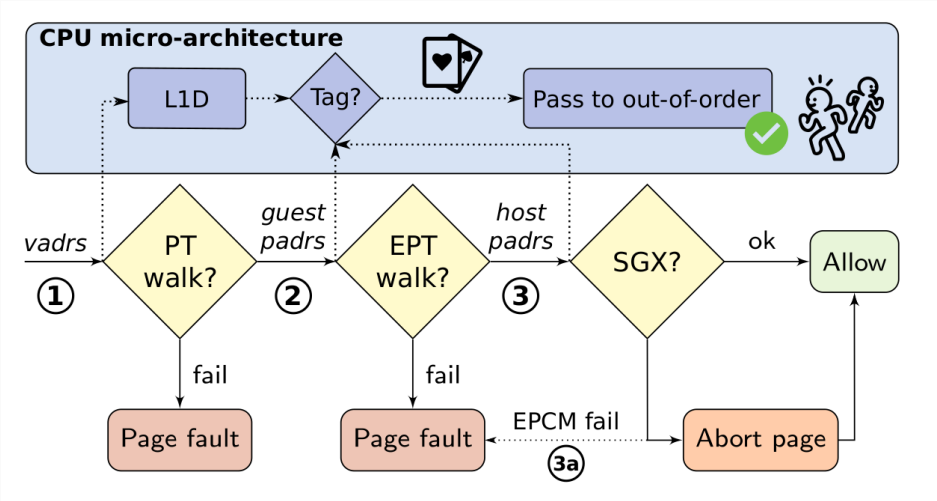
Microarchitectural Side-Channel Attacks

From the Basics to Transient Execution Attacks

Claudio Canella

June 17, 2018

IAIK – Graz University of Technology


CPU micro-architecture
L1D
Tag?
Pass to out-of-order
vadrs
PT walk?
guest padrs
EPT walk?
host padrs
SGX?
ok
Allow
1
2
3
fail
fail
Page fault
Page fault
EPCM fail
3a
Abort page