

Microarchitectural Side-Channel Attacks

From the Basics to Transient Execution Attacks

Claudio Canella

June 17, 2018

IAIK – Graz University of Technology



Claudio Canella

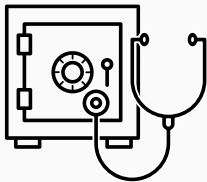
PhD student @ Graz University of Technology

🐦 @cc0x1f

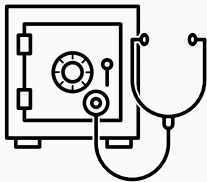
✉ claudio.canella@iaik.tugraz.at

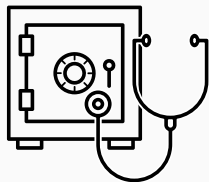
Side-Channel Attacks

- Safe software infrastructure does not mean safe execution



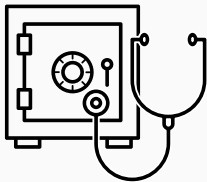
- Safe software infrastructure does not mean safe execution
- Information leaks because of the **underlying hardware**





- 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



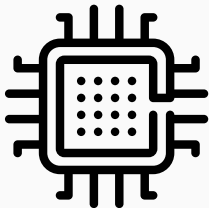
Execution
time



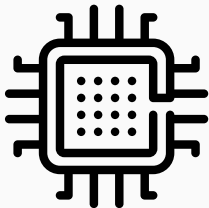
CPU caches



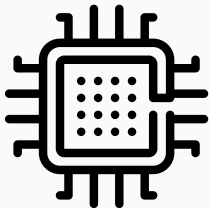
Caches and Cache Attacks



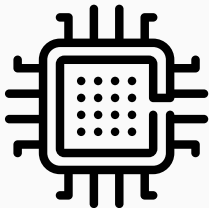
- Cache-based keylogging



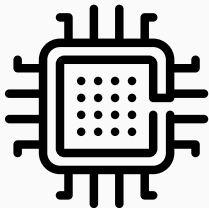
- Cache-based keylogging
- Crypto key recovery



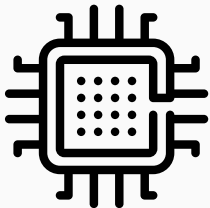
- Cache-based keylogging
- Crypto key recovery
 - various implementations (AES, RSA, ECC, ...)
 - up to 97% key bits recovered after 1 encryption



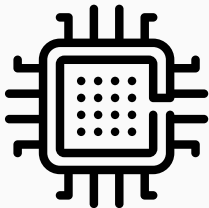
- Cache-based keylogging
- Crypto key recovery
 - various implementations (AES, RSA, ECC, ...)
 - up to 97% key bits recovered after 1 encryption
- Cross-VM, cross-core, even cross-CPU



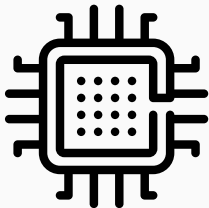
- Cache-based keylogging
- Crypto key recovery
 - various implementations (AES, RSA, ECC, ...)
 - up to 97% key bits recovered after 1 encryption
- Cross-VM, cross-core, even cross-CPU
- Any CPU vendor



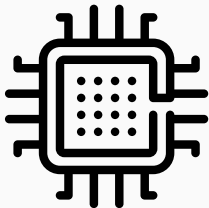
- using the *inclusive* property



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

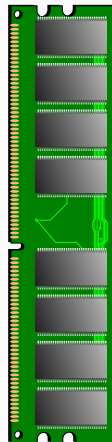
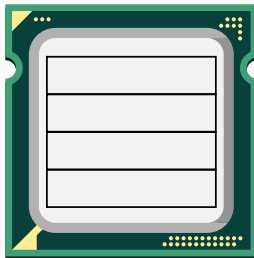


- using the *inclusive* property
- last-level cache is a superset of L1 and L2
- data evicted from last-level cache → evicted from L1 and L2



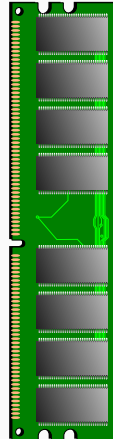
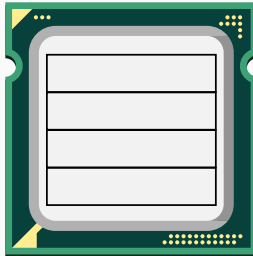
- using the *inclusive* property
- last-level cache is a superset of L1 and L2
- data evicted from last-level cache → evicted from L1 and L2
- a core can evict lines in the private L1 of another core

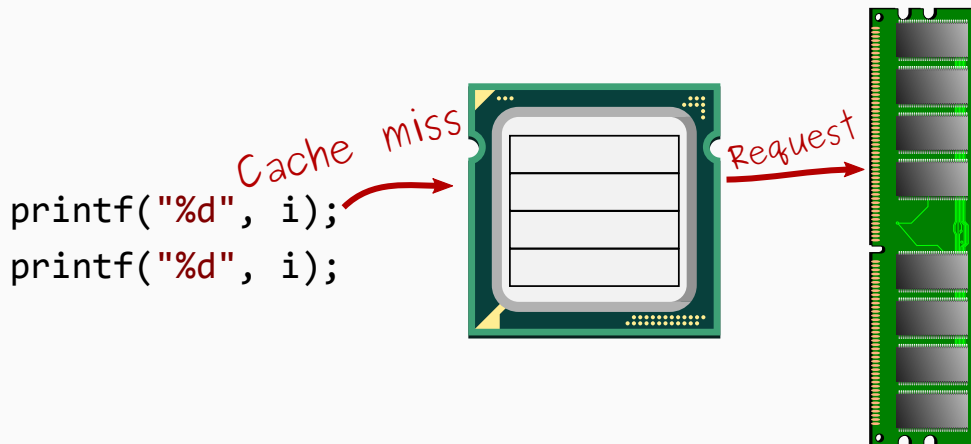
```
printf("%d", i);  
printf("%d", i);
```

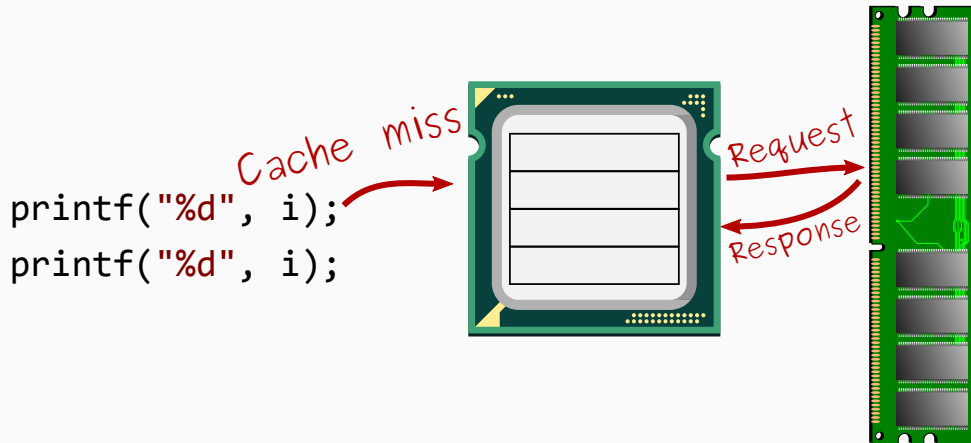


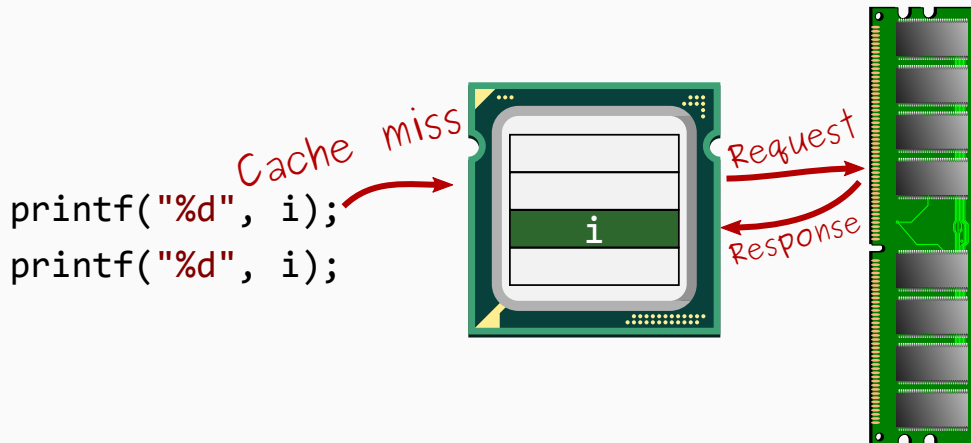
```
printf("%d", i);  
printf("%d", i);
```

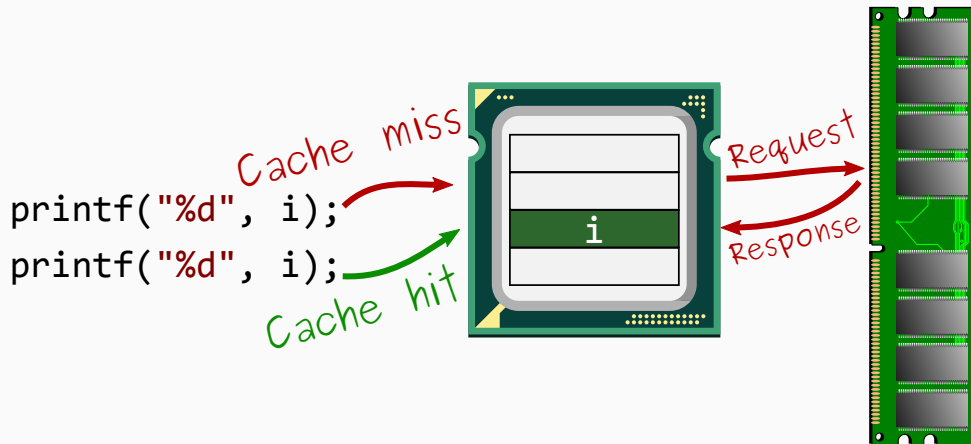
Cache miss

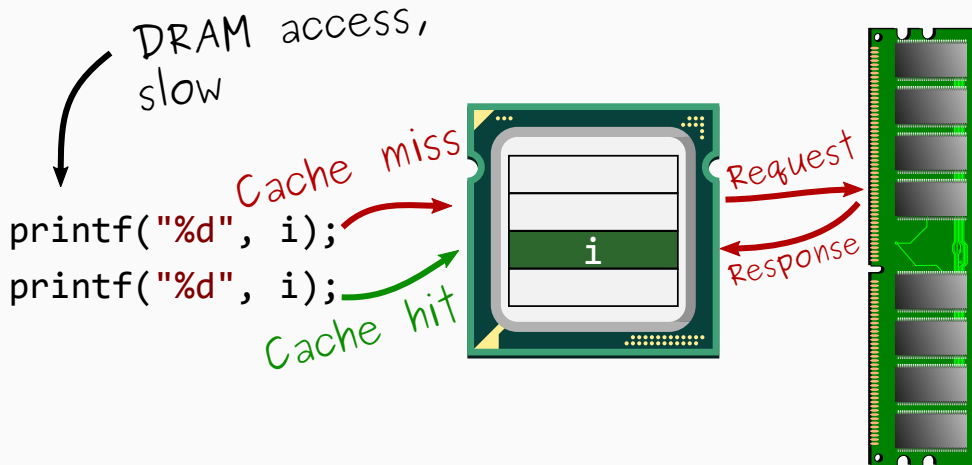


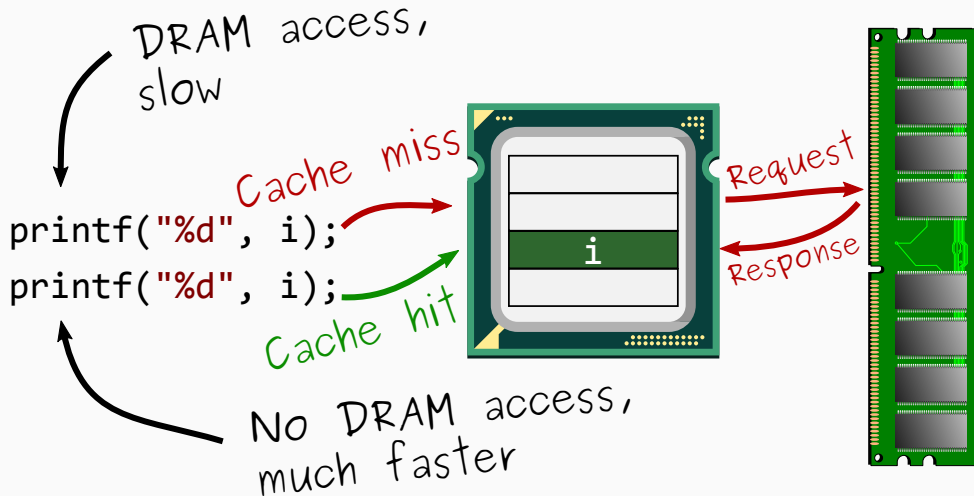


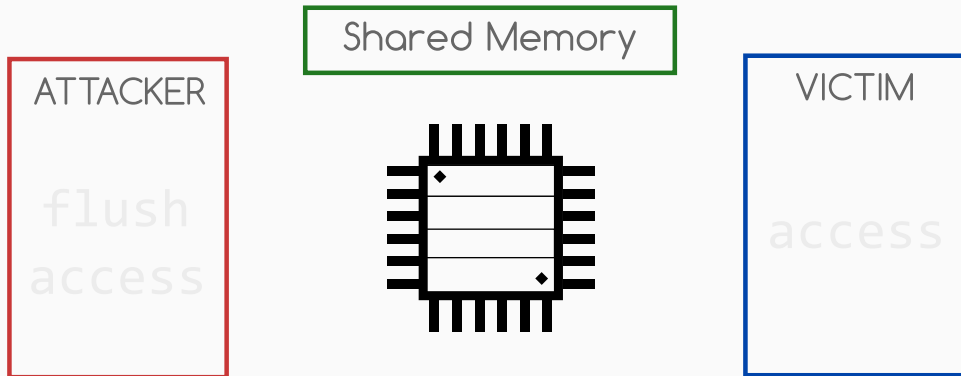


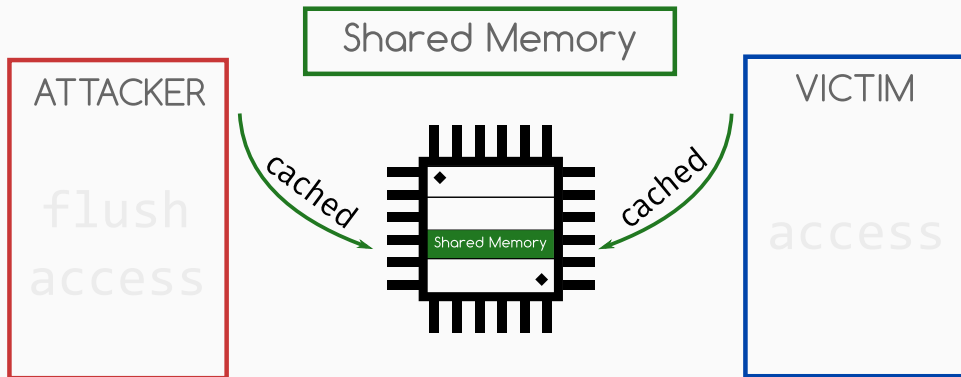


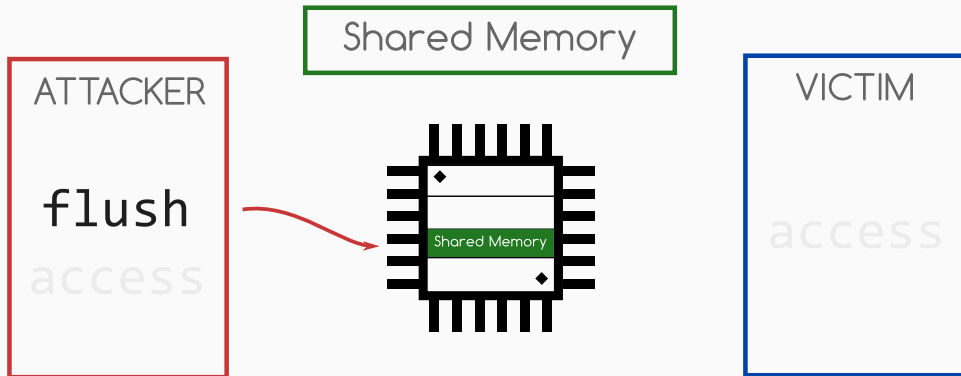


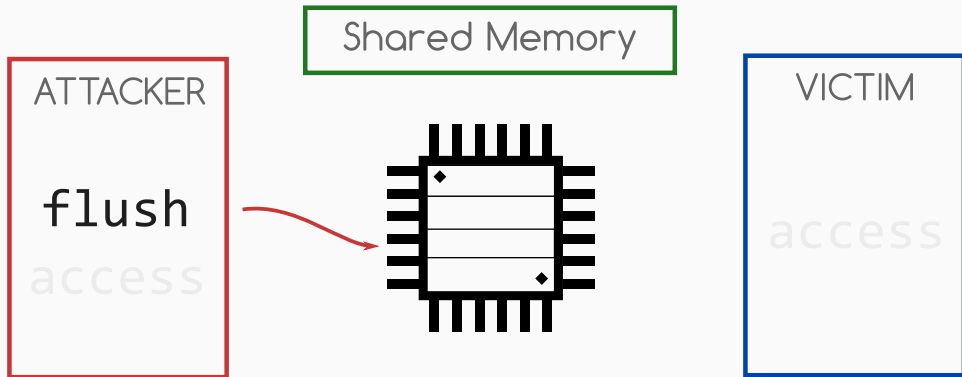


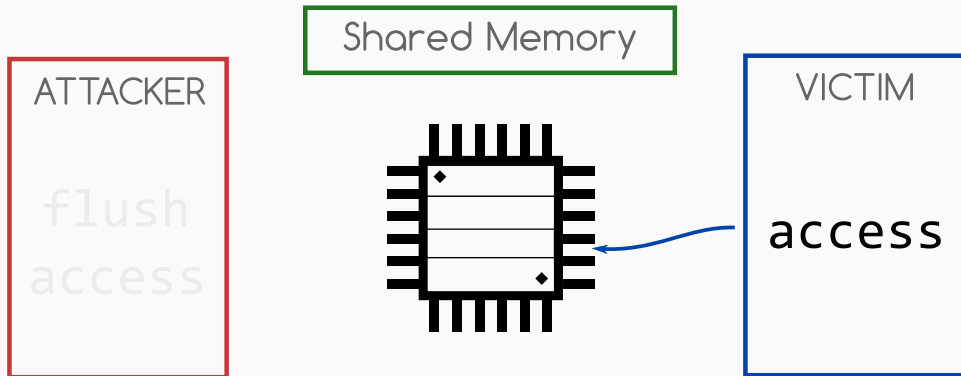


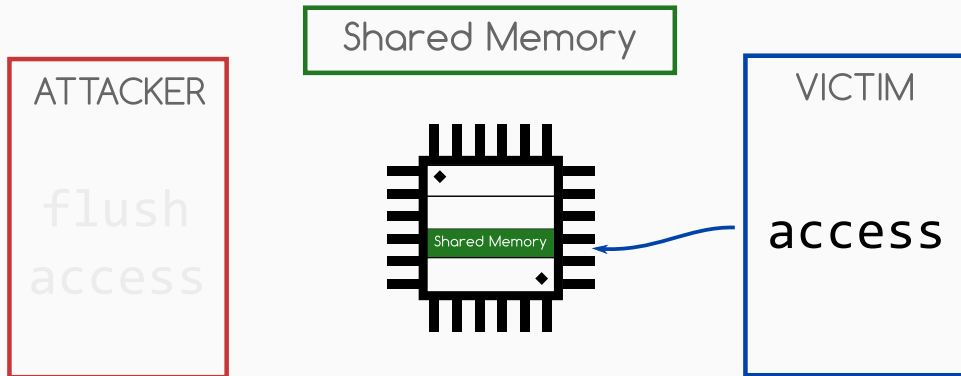


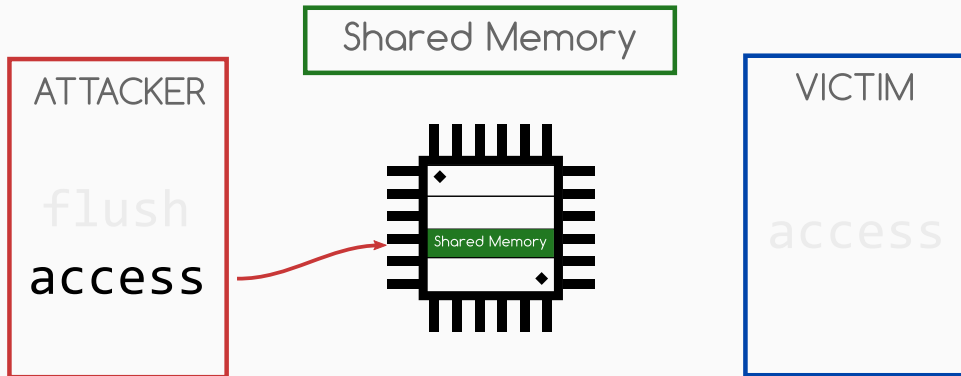


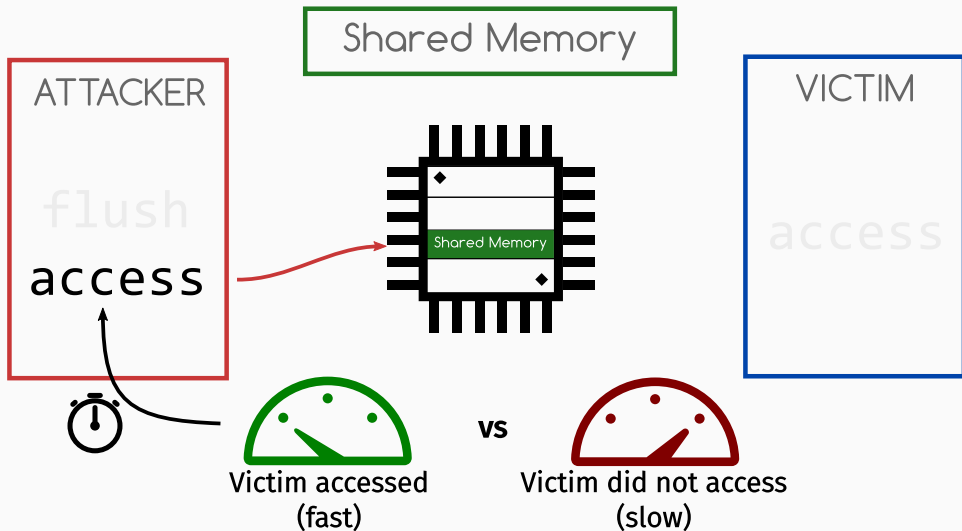


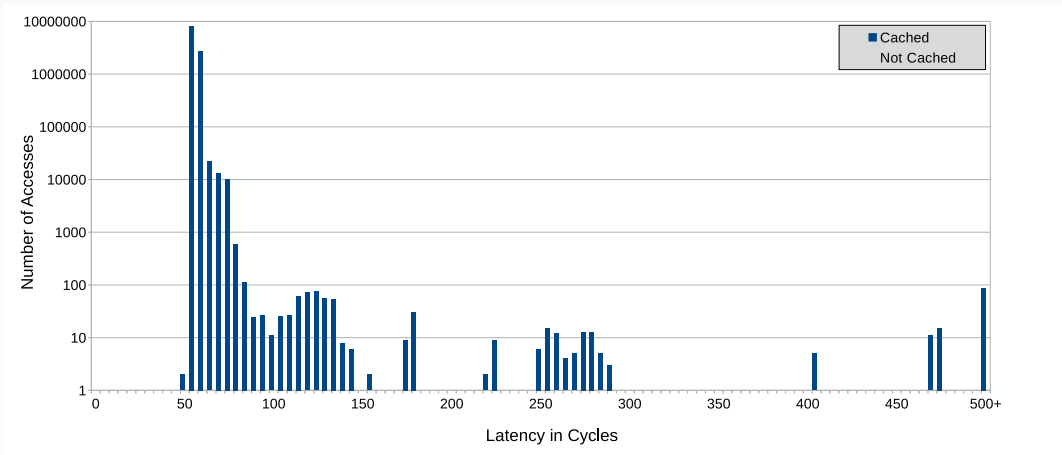


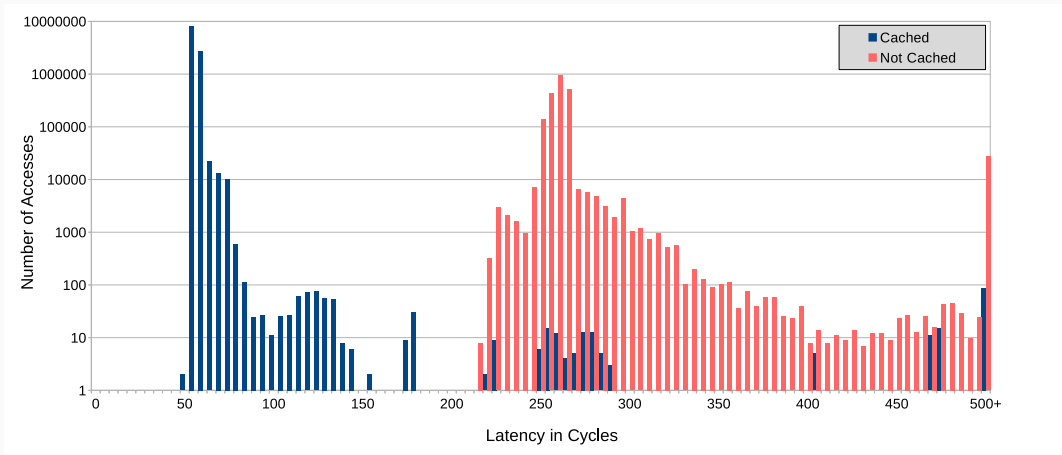












Cache Template Attacks

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

Attacker address space



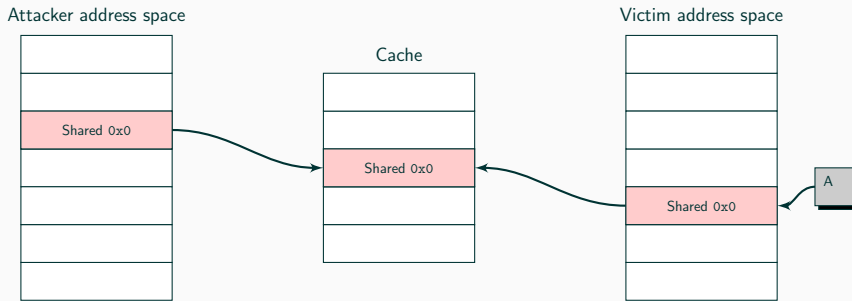
Cache



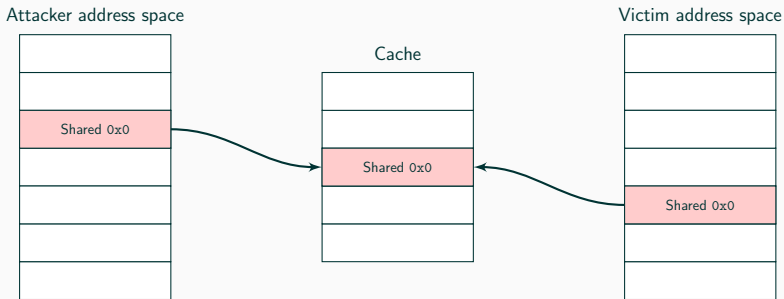
Victim address space



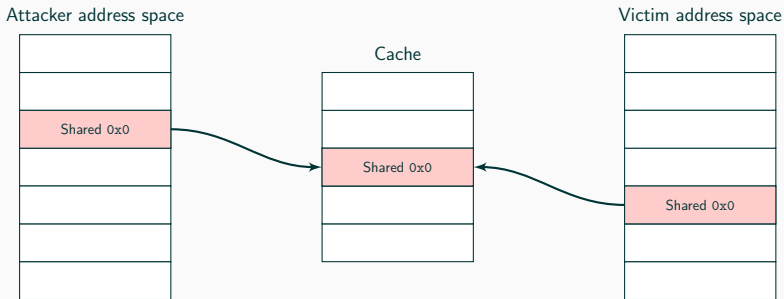
Cache is empty



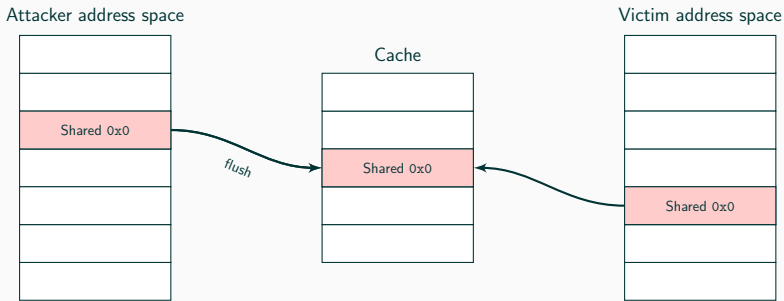
Attacker triggers an event



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



Update cache hit ratio (per event and address)

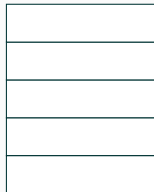


Attacker flushes shared memory

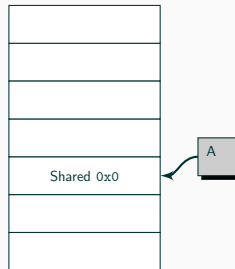
Attacker address space



Cache



Victim address space

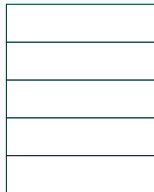


Repeat for higher accuracy

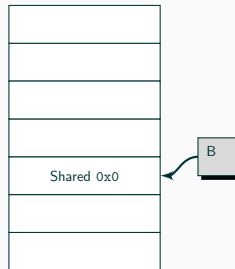
Attacker address space



Cache



Victim address space

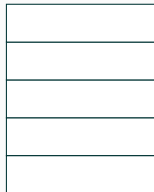


Repeat for all events

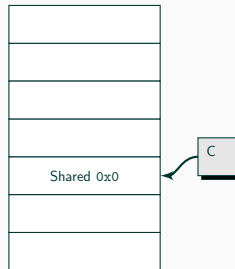
Attacker address space



Cache



Victim address space

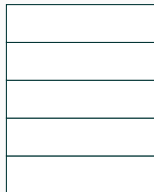


Repeat for all events

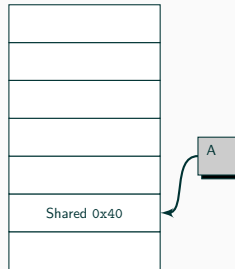
Attacker address space



Cache



Victim address space

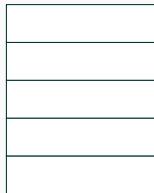


Continue with next address

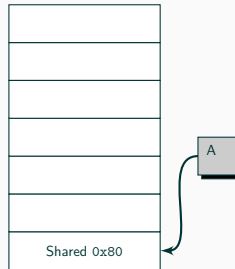
Attacker address space



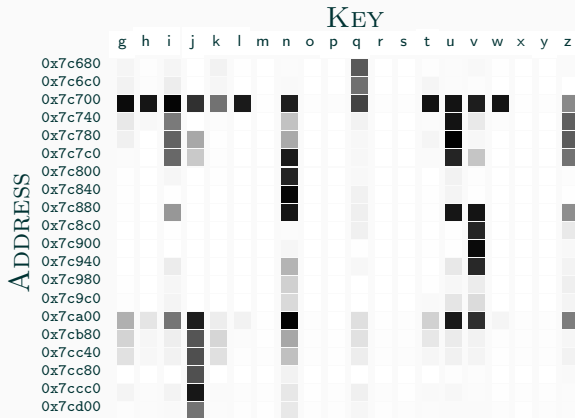
Cache



Victim address space

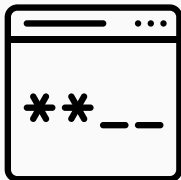


Continue with next address





- 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

-

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

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

...





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

- 4 T-Tables

-

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

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

...

- 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

-

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

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

...

- 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



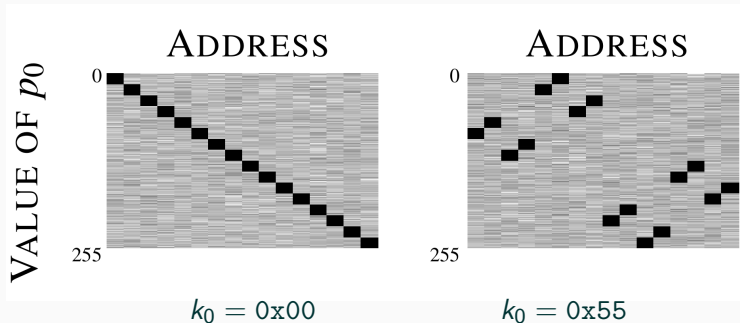
AES T-Table implementation from OpenSSL 1.0.2

- 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

```
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

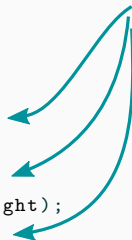
Dependency

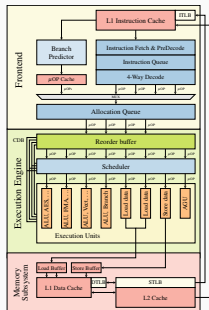
```
int width = 10, height = 5;

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

int area = width * height;

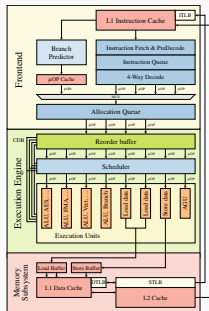
printf("Area %d x %d = %d\n", width, height, area);
```





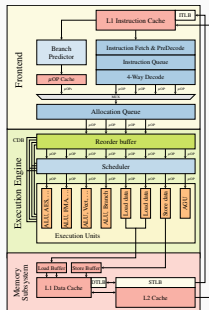
Instructions

- are executed **out-of-order**



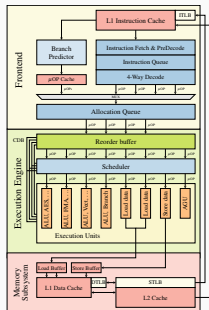
Instructions

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



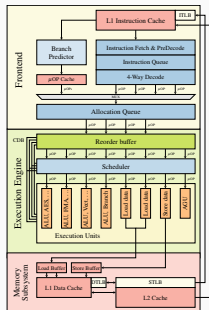
Instructions

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



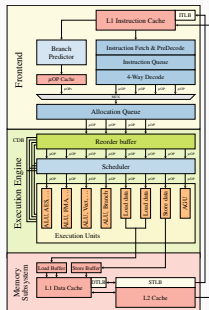
Instructions

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



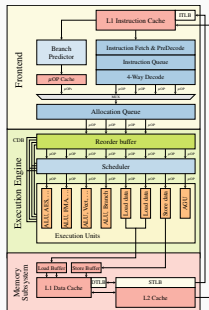
Instructions

- 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



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



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



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



- 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?



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



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



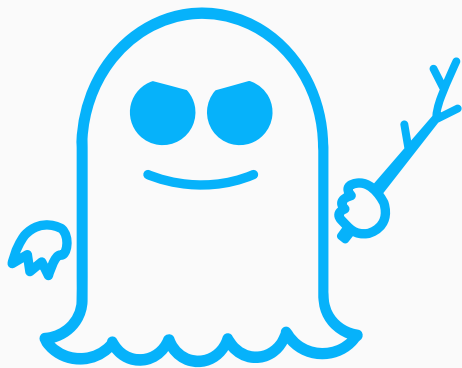
- Out-of-order/speculatively executed instructions **leave microarchitectural traces**
 - We can see them for example in the cache
- Give such instructions a name: **transient instructions**



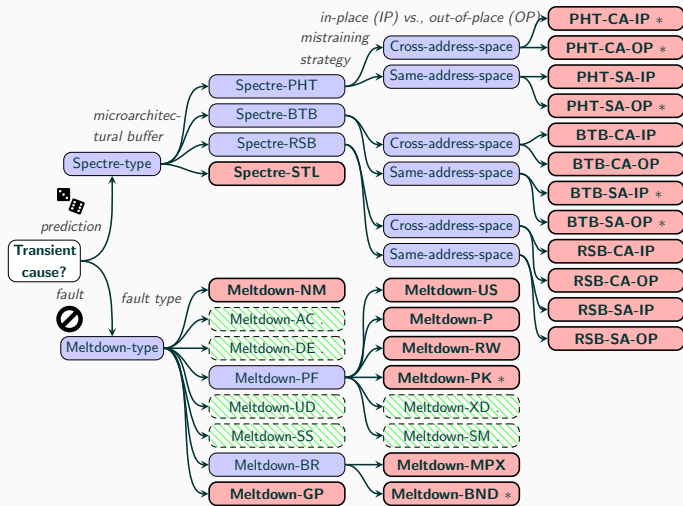
- Out-of-order/speculatively executed instructions **leave microarchitectural traces**
 - We can see them for example in the cache
- Give such instructions a name: **transient instructions**
- We can indirectly observe the **execution of transient instructions**

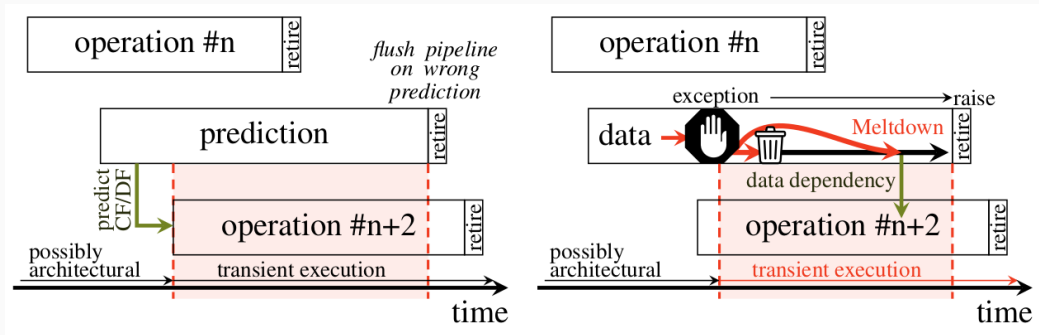


MELTDOWN



SPECTRE







- Attacker loads inaccessible data into register



- Attacker loads inaccessible data into register → page-fault



- Attacker loads inaccessible data into register → page-fault
- Transient instruction stream accesses cache line based on loaded data



- Attacker loads inaccessible data into register → page-fault
- Transient instruction stream accesses cache line based on loaded data
- Determine accessed cache line using Flush+Reload



- Attacker loads inaccessible 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



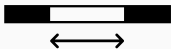
- OS must save registers on context switch → lazy-switching
- Next FPU instruction causes $\#NM$ exception



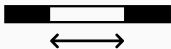
- OS must save registers on context switch → lazy-switching
- Next FPU instruction causes $\#NM$ exception
- Execution continues with previous data in register



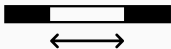
- OS must save registers on context switch → lazy-switching
- Next FPU instruction causes $\#NM$ exception
- Execution continues with previous data in register
- Extract data using Flush+Reload



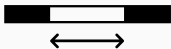
- x86 provides dedicated instruction raising #BR exception if bound-range is exceeded



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



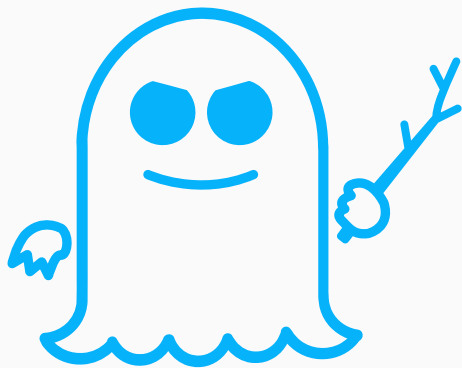
- x86 provides dedicated instruction raising `#BR` exception if bound-range is exceeded
- Subsequent data is again used in transient execution
- Attacker determines accessed cache line using Flush+Reload



- 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



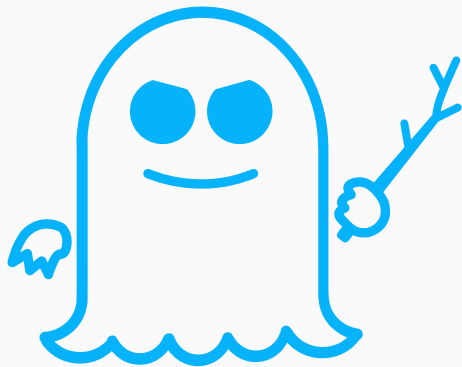
MELTDOWN



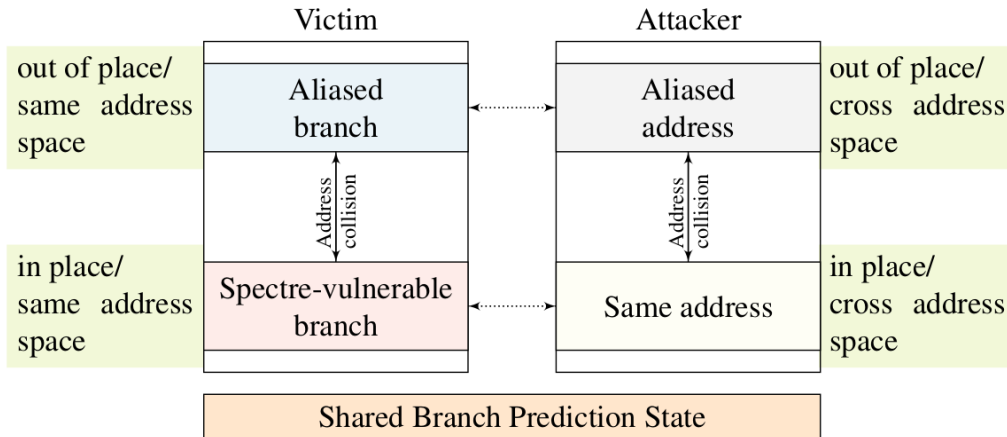
SPECTRE



MELTDOWN



SPECTRE



```
index = 0;
```

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

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

then



else

```
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)
```

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



Prediction

else

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

```
0
```

```
index = 1;
```

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

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

then



Prediction

else

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

```
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



Prediction

else

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

```
0
```

```
index = 2;
```

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

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

then



Prediction

else

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

```
0
```

```
index = 2;
```

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

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

then



Prediction

else

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

```
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



else

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

```
0
```

```
index = 3;
```

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

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

then



Prediction

else

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

```
0
```

```
index = 3;
```

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

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

then



Prediction

else

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

```
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



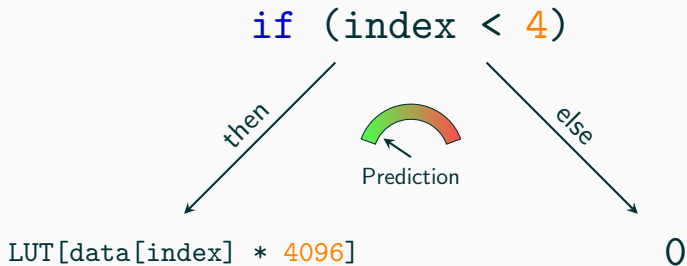
else

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

```
0
```

```
index = 4;
```

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



```
index = 4;
```

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

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

then



Prediction

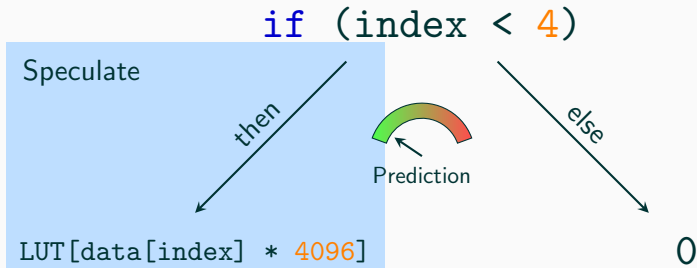
else

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

```
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";
```

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

then



Prediction

else

```
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)
```

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



Prediction

else

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

```
0
```

```
index = 6;
```

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

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

then



Prediction

else

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

```
0
```

```
index = 6;
```

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

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

Speculate

then

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



Prediction

else

0

```
index = 6;
```

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

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

then

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

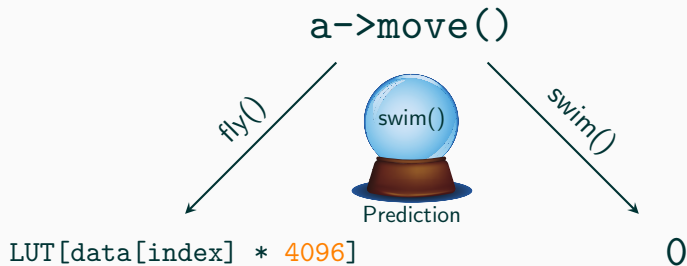


else

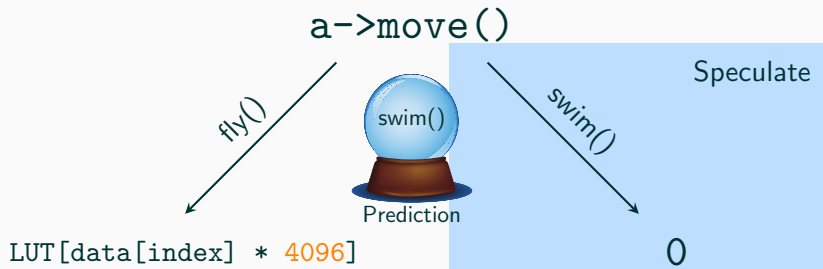
Execute

0

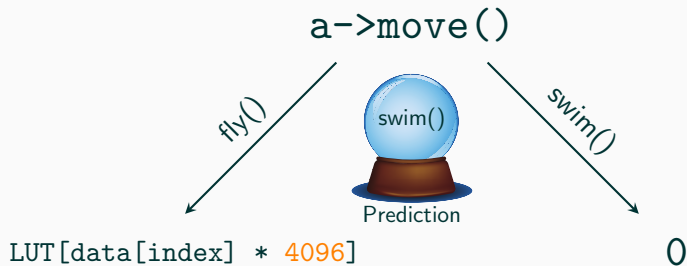
```
Animal* a = bird;
```



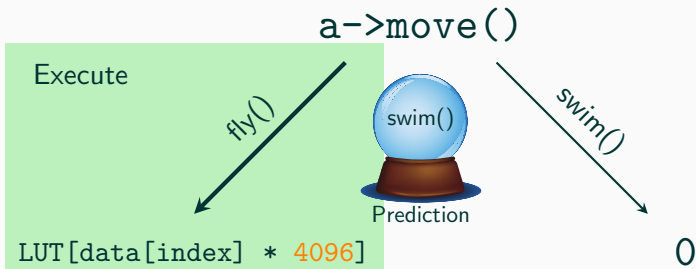
```
Animal* a = bird;
```



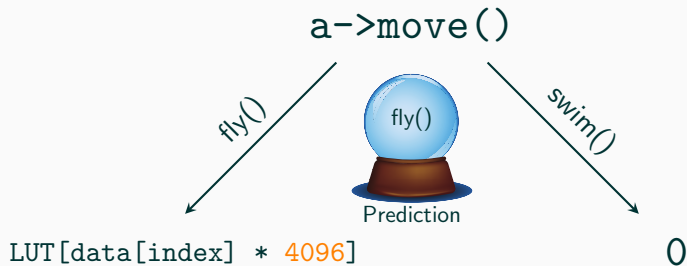
```
Animal* a = bird;
```



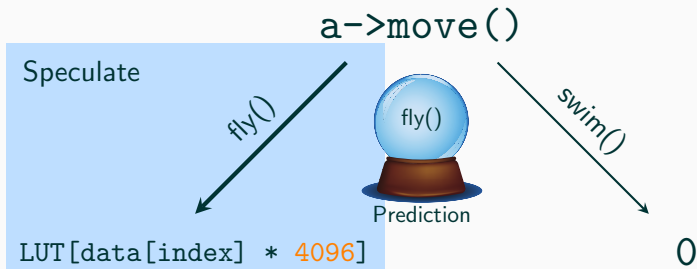
```
Animal* a = bird;
```



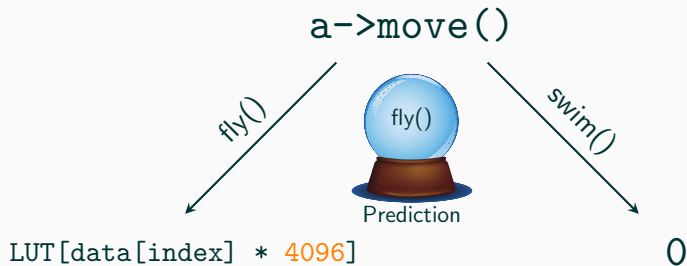
```
Animal* a = bird;
```



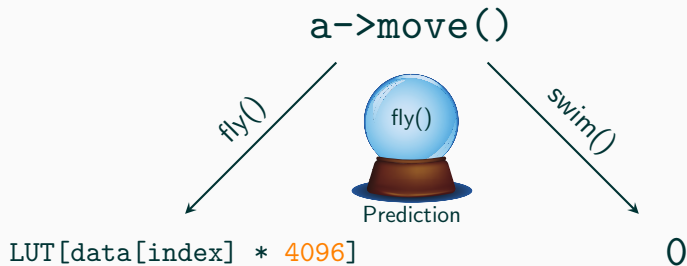
```
Animal* a = bird;
```



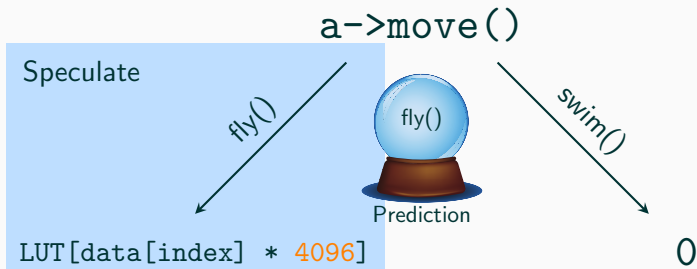
```
Animal* a = bird;
```



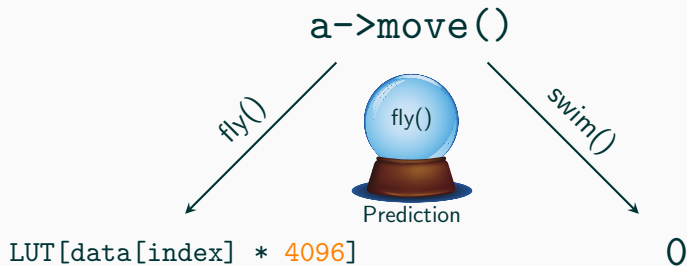
```
Animal* a = fish;
```



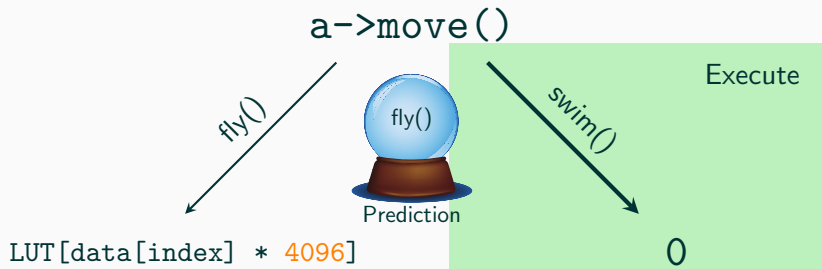
```
Animal* a = fish;
```



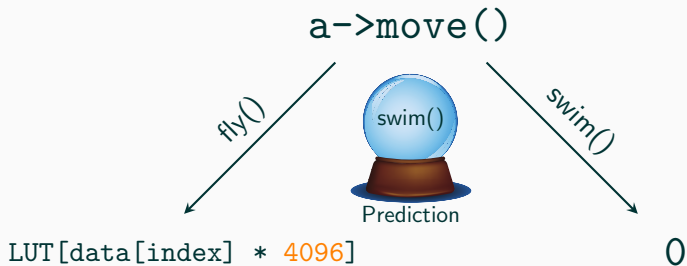
```
Animal* a = fish;
```



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



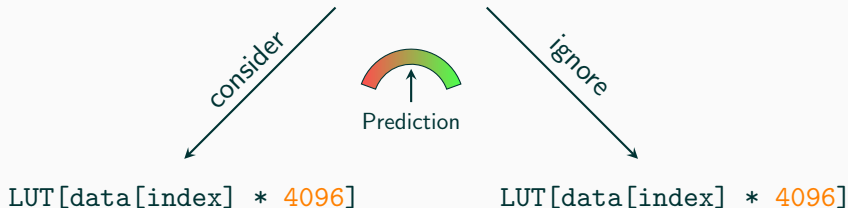
```
Animal* a = fish;
```

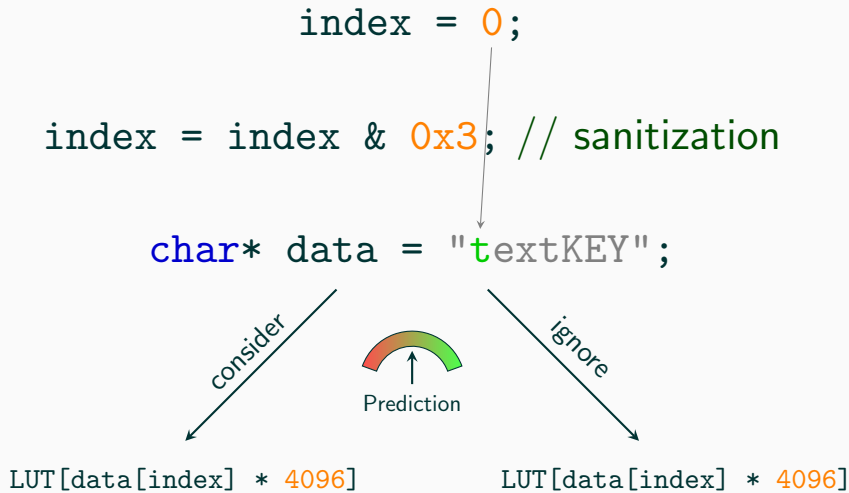


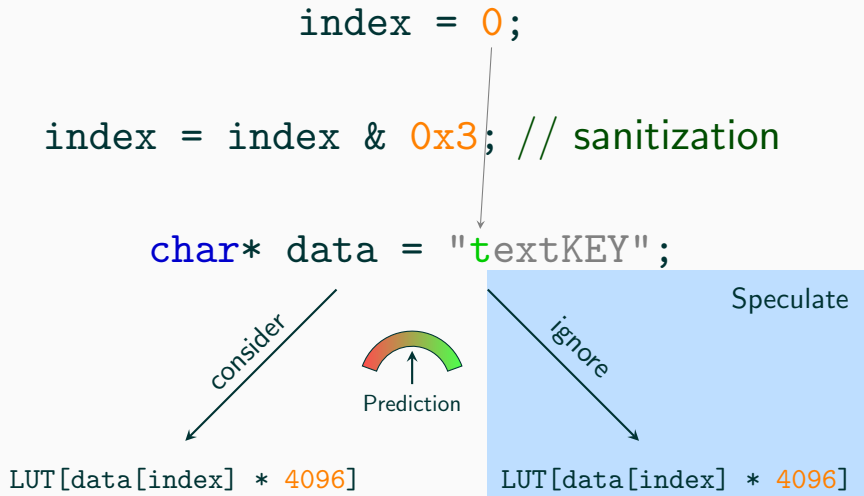
```
index = 0;
```

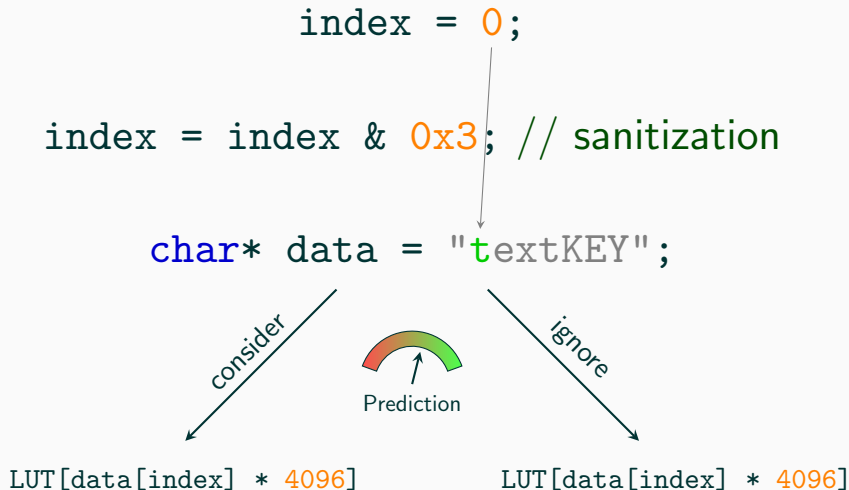
```
index = index & 0x3; // sanitization
```

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





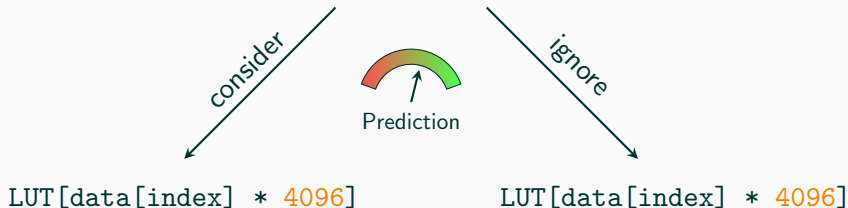


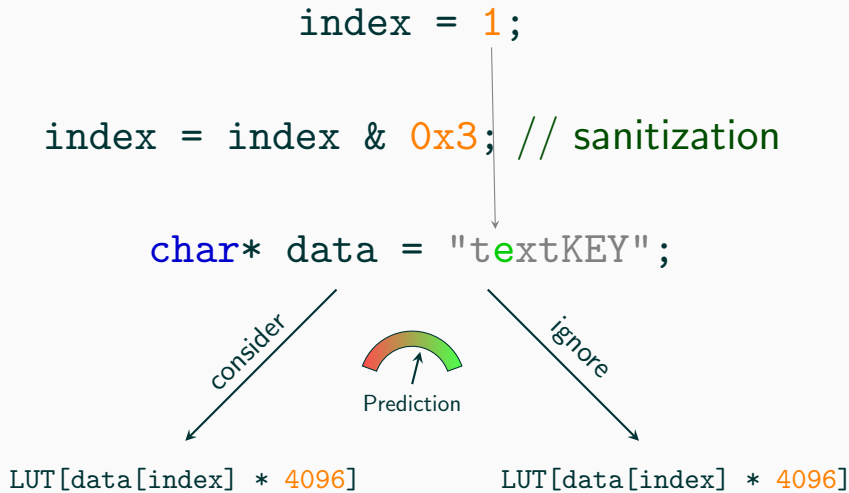


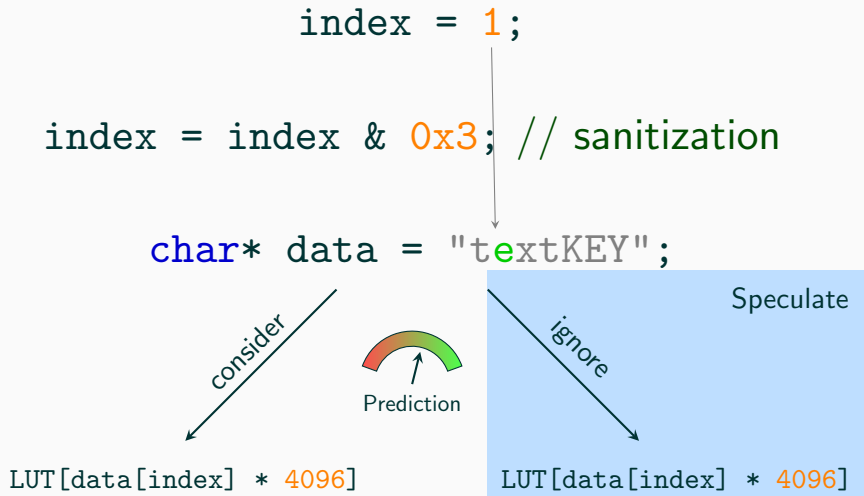
```
index = 1;
```

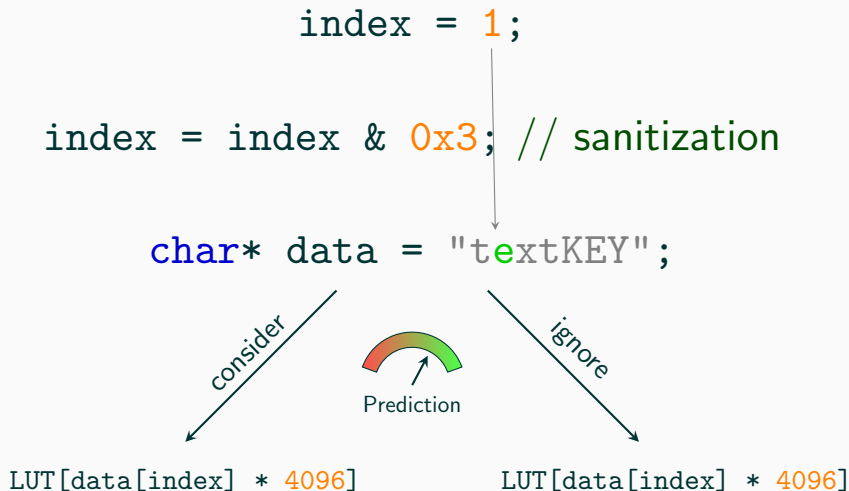
```
index = index & 0x3; // sanitization
```

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





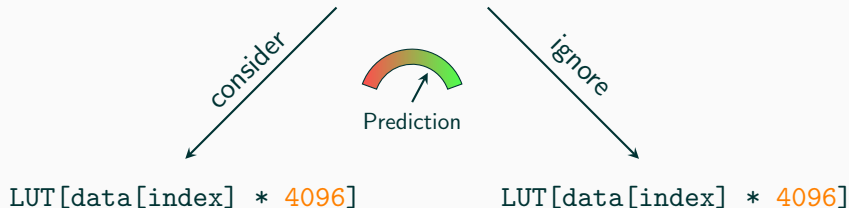


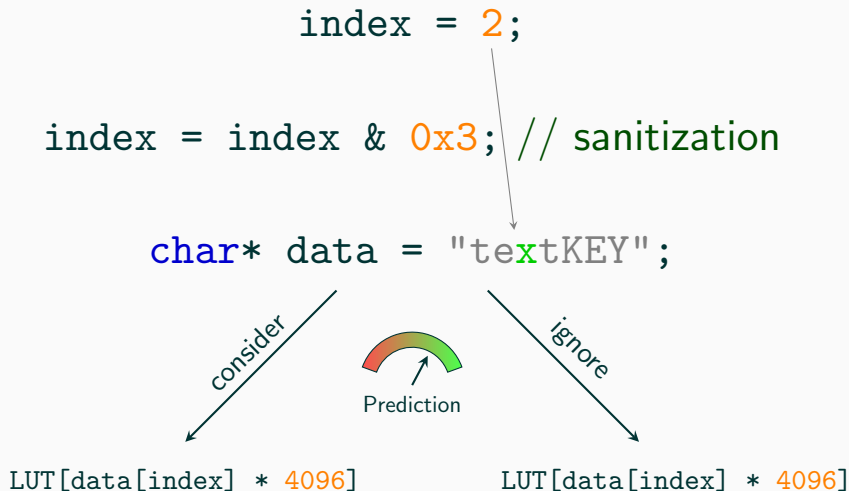


```
index = 2;
```

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

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

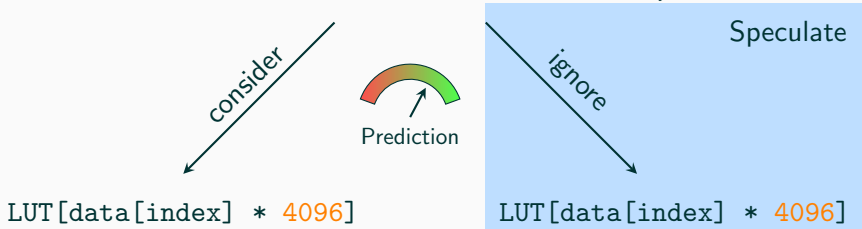


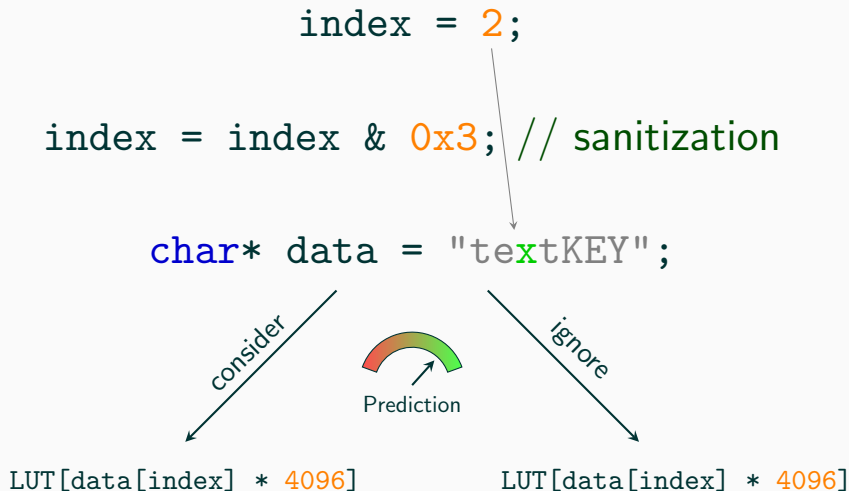


```
index = 2;
```

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

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

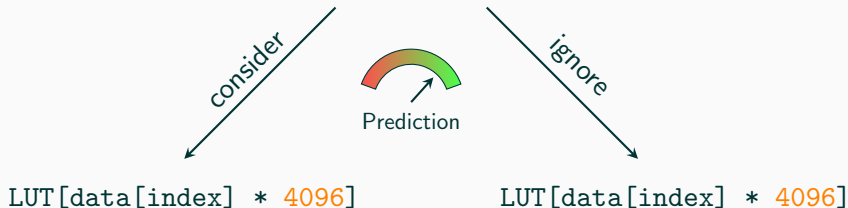




```
index = 3;
```

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

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



```
index = 3;
```

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

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

consider



Prediction

ignore

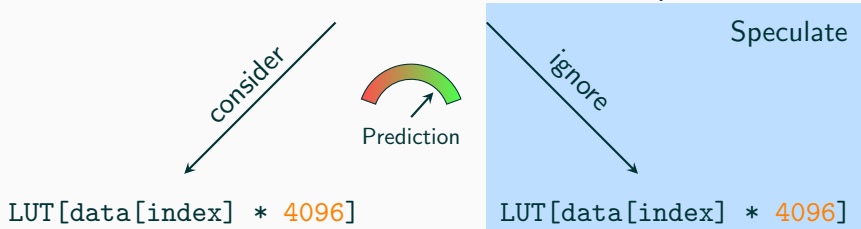
```
LUT[data[index] * 4096]
```

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

```
index = 3;
```

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

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



```
index = 3;
```

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

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

consider



Prediction

ignore

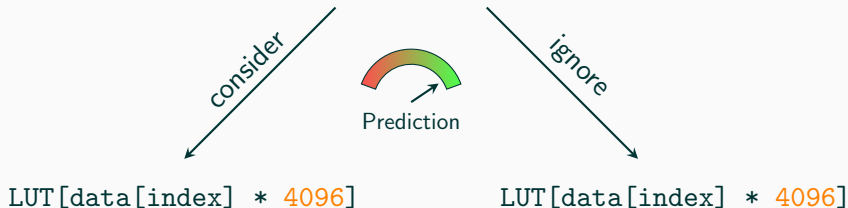
```
LUT[data[index] * 4096]
```

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

```
index = 4;
```

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

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



```
index = 4;
```

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

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

consider

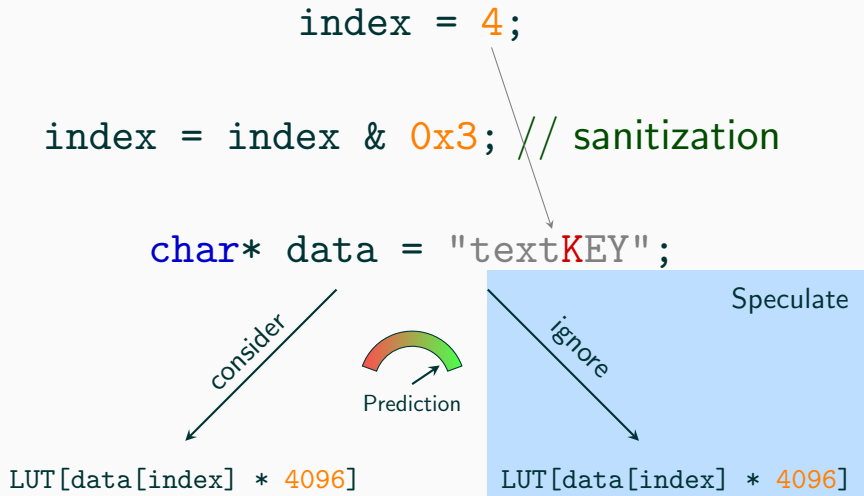


Prediction

ignore

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

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



```
index = 4;
```

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

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

Execute

$\text{index} = 0$

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



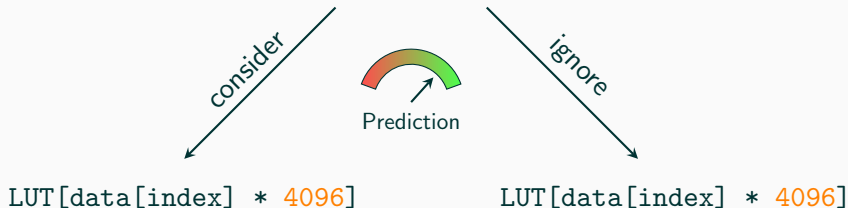
ignore

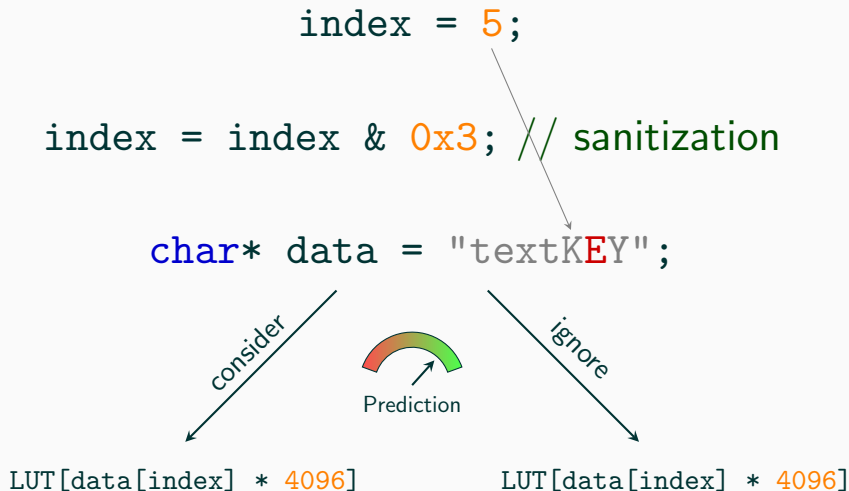
$\text{LUT}[\text{data}[\text{index}] * 4096]$

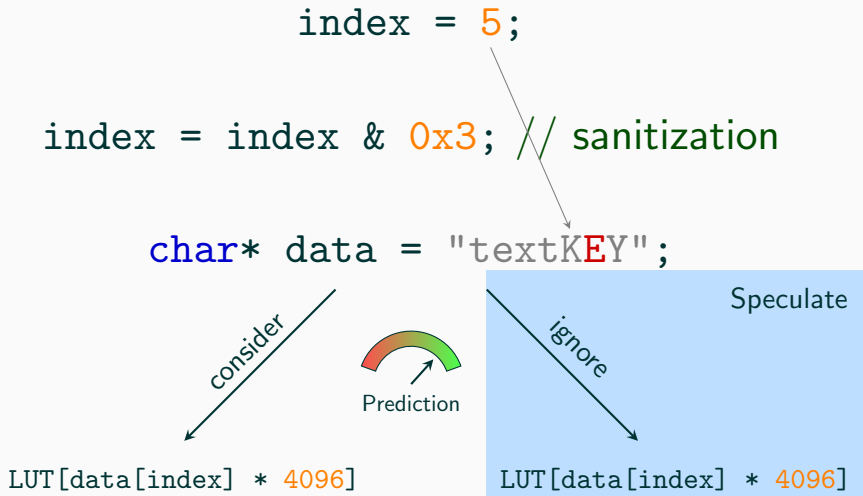
```
index = 5;
```

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

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







```
index = 5;
```

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

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

Execute

index = 1



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

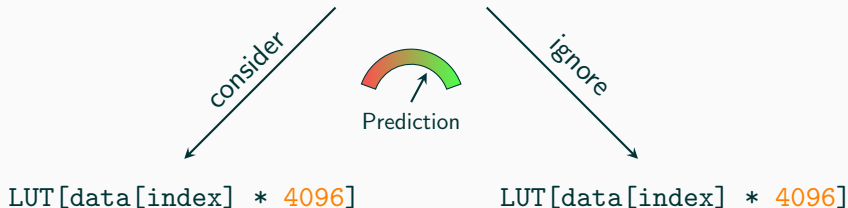
ignore

```
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

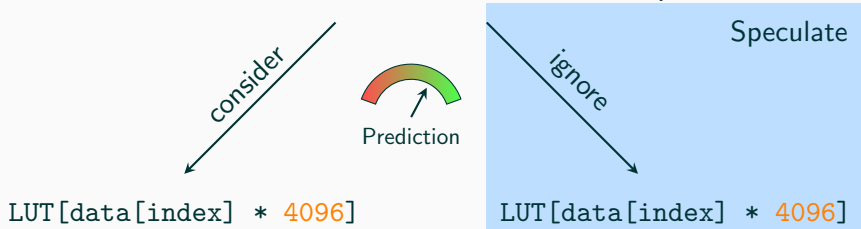
```
LUT[data[index] * 4096]
```

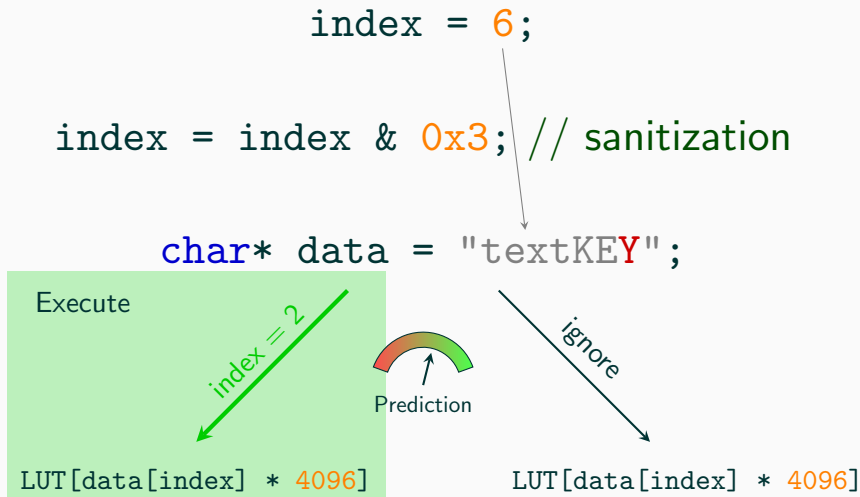
```
LUT[data[index] * 4096]
```

```
index = 6;
```

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

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







- 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



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



- 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
→ 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

