# Hardware-Software Co-Design against Microarchitectural Attacks

**Daniel Gruss**

September 4, 2019

Graz University of Technology


National Geographic


side channel
= obtaining meta-data and
deriving secrets from it
CHANGE MY MIND

# Side Channel or not?

# Side Channel or not?

- Profiling cache utilization with performance counters?

# Side Channel or not?

- Profiling cache utilization with performance counters? $\rightarrow$ No

- Profiling cache utilization with performance counters? $\rightarrow$ No
- Observing cache utilization with performance counters and using it to infer a crypto key?

# Side Channel or not?

- Profiling cache utilization with performance counters? $\rightarrow$ No
- Observing cache utilization with performance counters and using it to infer a crypto key? $\rightarrow$ Yes

# Side Channel or not?

- Profiling cache utilization with performance counters? $\rightarrow$ No
- Observing cache utilization with performance counters and using it to infer a crypto key? $\rightarrow$ Yes
- Measuring memory access latency with Flush+Reload?

# Side Channel or not?

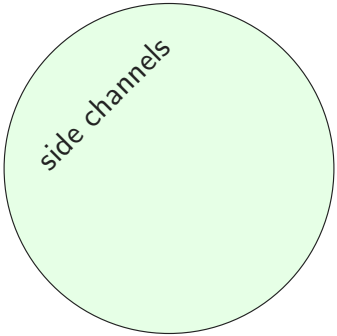
- Profiling cache utilization with performance counters? → No
- Observing cache utilization with performance counters and using it to infer a crypto key? → Yes
- Measuring memory access latency with Flush+Reload? → No

- Profiling cache utilization with performance counters? $\rightarrow$ No
- Observing cache utilization with performance counters and using it to infer a crypto key? $\rightarrow$ Yes
- Measuring memory access latency with Flush+Reload? $\rightarrow$ No
- Measuring memory access latency with Flush+Reload and using it to infer keystroke timings?

- Profiling cache utilization with performance counters? $\rightarrow$ No
- Observing cache utilization with performance counters and using it to infer a crypto key? $\rightarrow$ Yes
- Measuring memory access latency with Flush+Reload? $\rightarrow$ No
- Measuring memory access latency with Flush+Reload and using it to infer keystroke timings? $\rightarrow$ Yes

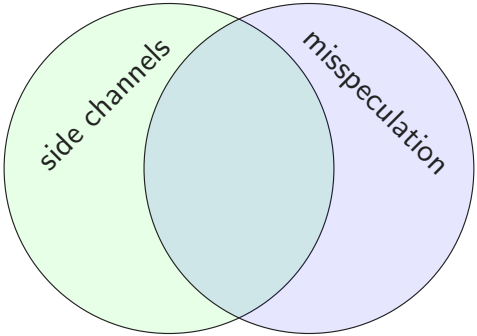
# Speculative Side-Channel Attacks?

# Speculative Side-Channel Attacks?



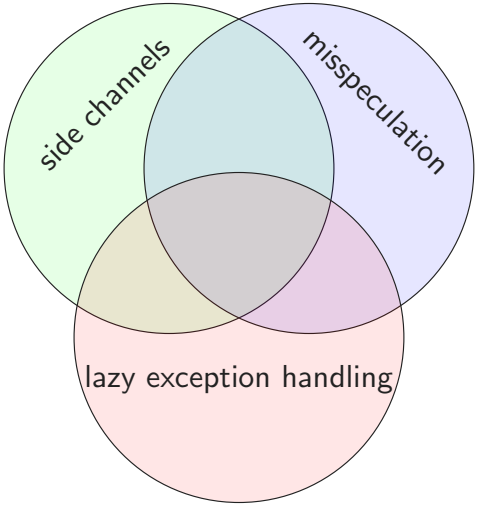

- traditional cache attacks (crypto, keys, etc)

# Speculative Side-Channel Attacks?



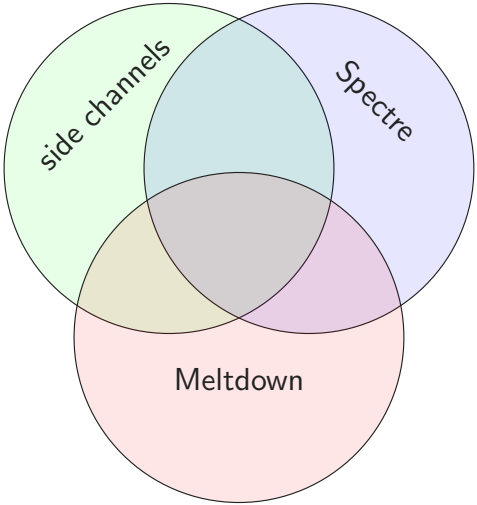

- traditional cache attacks (crypto, keys, etc)
- actual misspeculation (e.g., branch misprediction)

# Speculative Side-Channel Attacks?



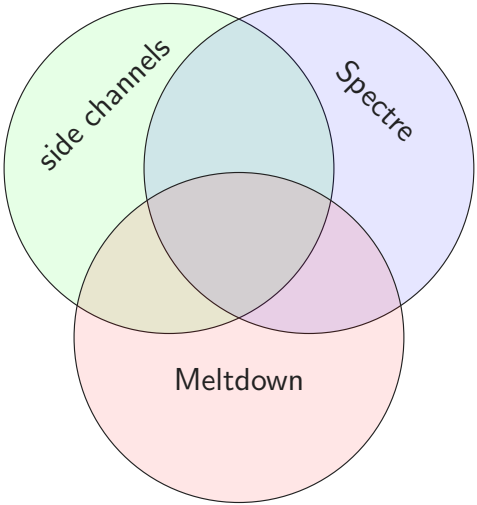

- traditional cache attacks (crypto, keys, etc)
- actual misspeculation (e.g., branch misprediction)
- Meltdown, Foreshadow, ZombieLoad, etc

# Speculative Side-Channel Attacks?



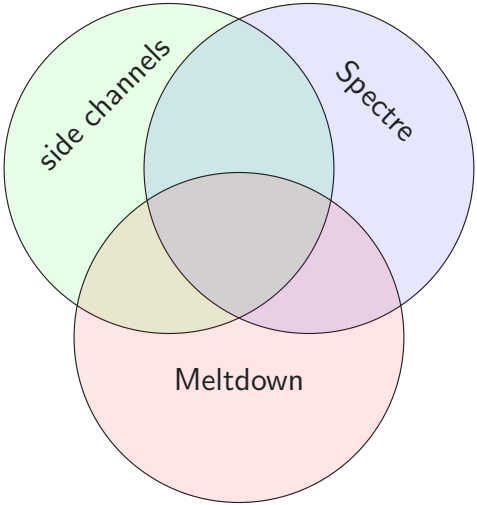

- traditional cache attacks (crypto, keys, etc)
- actual misspeculation (e.g., branch misprediction)
- Meltdown, Foreshadow, ZombieLoad, etc

# Speculative Side-Channel Attacks?



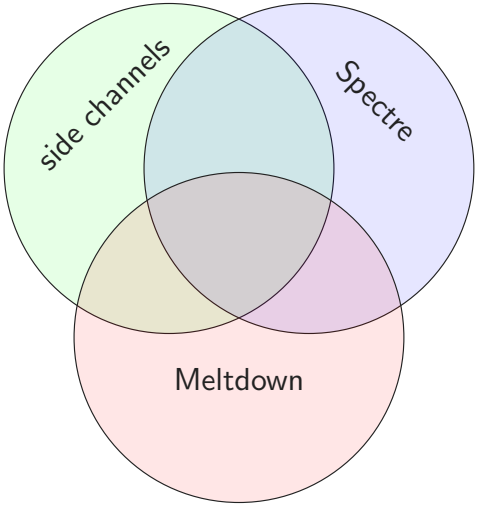

- traditional cache attacks (crypto, keys, etc)
- actual misspeculation (e.g., branch misprediction)
- Meltdown, Foreshadow, ZombieLoad, etc
- **Let's avoid the term Speculative Side-Channel Attacks**

# Speculative Side-Channel Attacks?



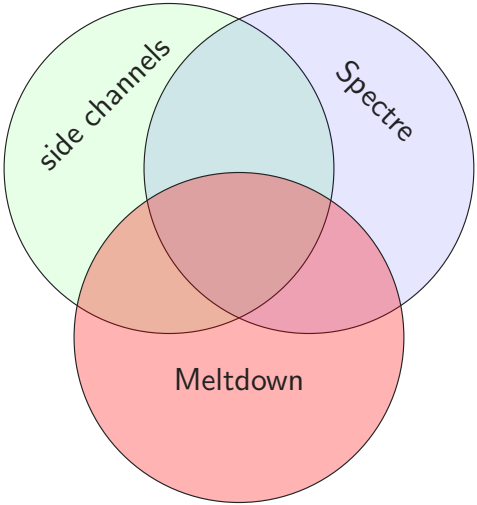

- traditional cache attacks (crypto, keys, etc)
- actual misspeculation (e.g., branch misprediction)
- Meltdown, Foreshadow, ZombieLoad, etc
- **Let's avoid the term Speculative Side-Channel Attacks**
- Let's be more precise

# Speculative Side-Channel Attacks?

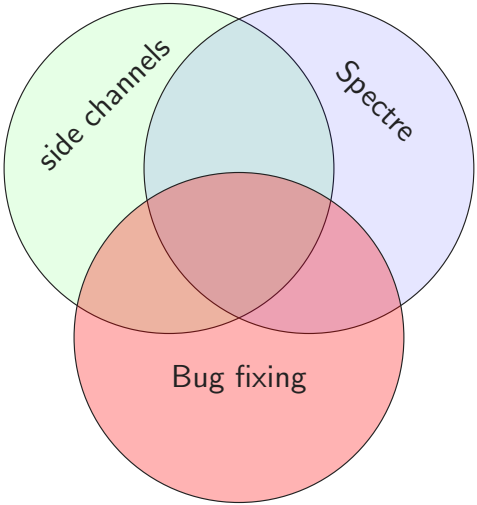



- traditional cache attacks (crypto, keys, etc)
- actual misspeculation (e.g., branch misprediction)
- Meltdown, Foreshadow, ZombieLoad, etc
- **Let's avoid the term Speculative Side-Channel Attacks**
- Let's be more precise
- $\rightarrow$ then we can think about actual mitigations

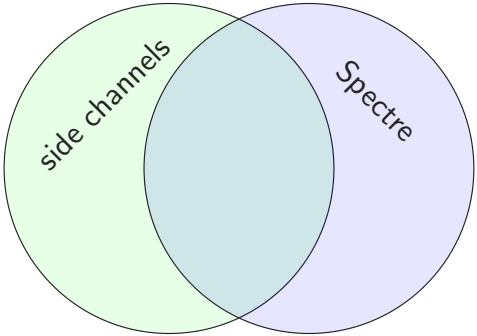
# Speculative Side-Channel Attacks?



# Speculative Side-Channel Attacks?



# Speculative Side-Channel Attacks?




But what about these two?

1337 4242

# FOOD CACHE

**Revolutionary** concept!

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

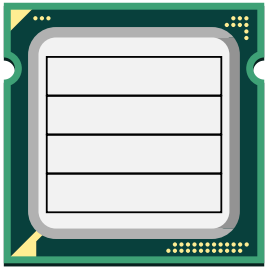
Can store **ALL** kinds of food.

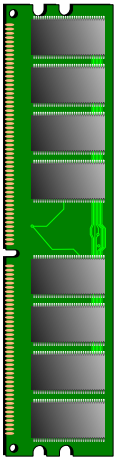
ONLY TODAY INSTEAD OF ~~$1,300~~

**$1,299**

ORDER VIA PHONE: +555 12345

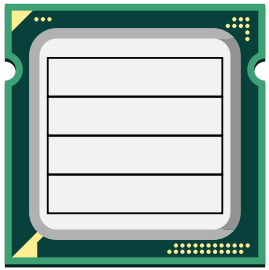
```
printf("%d", i);
printf("%d", i);
```

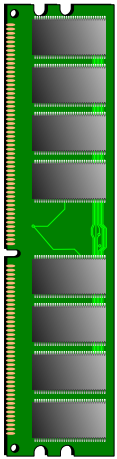




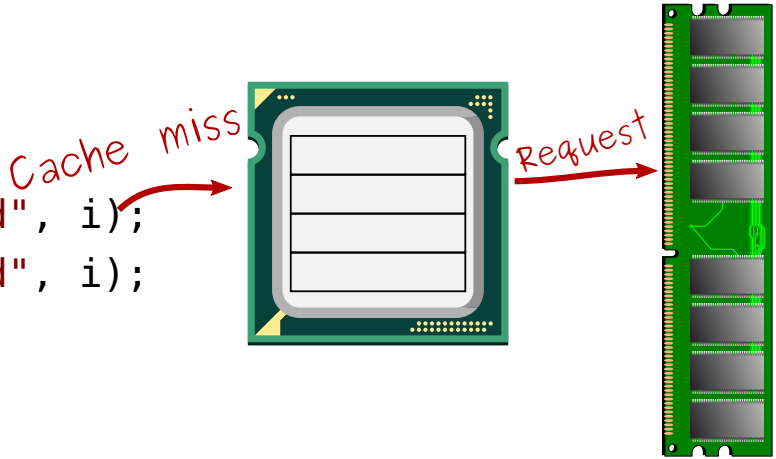
```
printf("%d", i);
printf("%d", i);
```

Cache miss

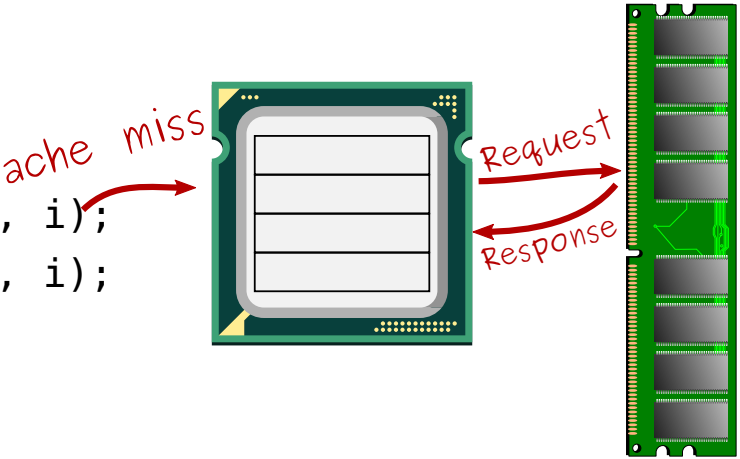




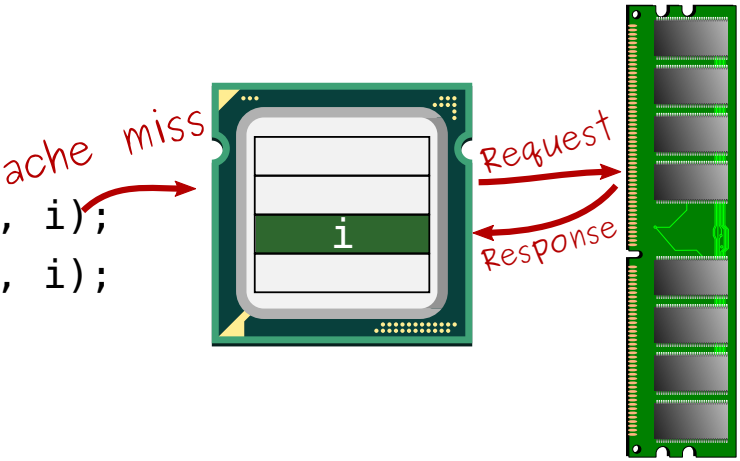
```
printf("%d", i);
printf("%d", i);
```

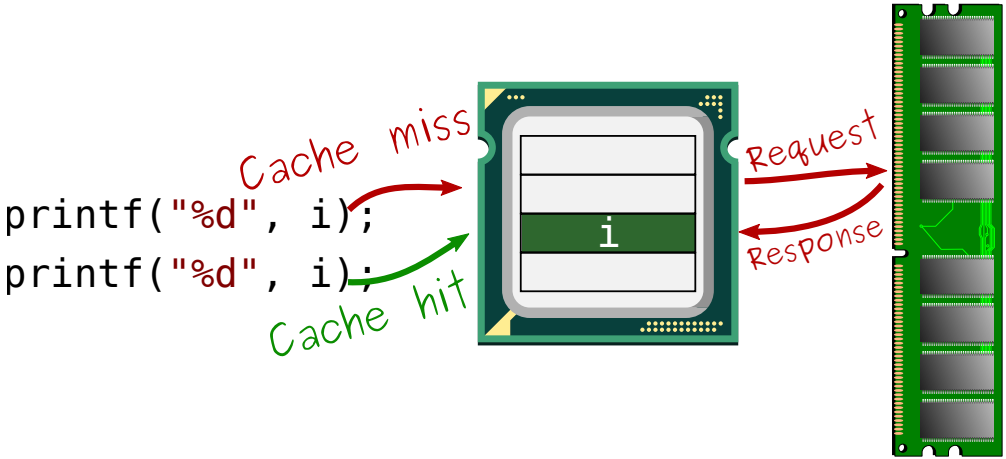


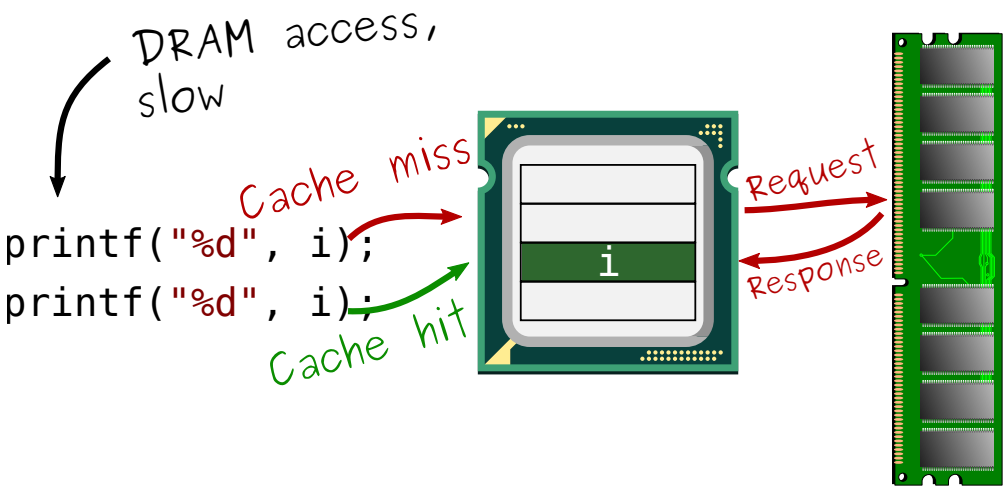
```
printf("%d", i);
printf("%d", i);
```

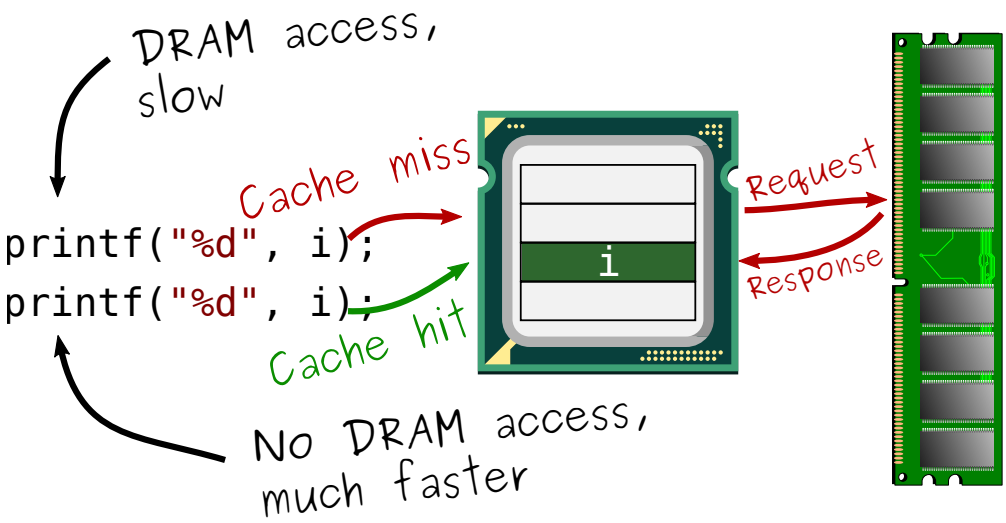


```
printf("%d", i);
printf("%d", i);
```




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


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

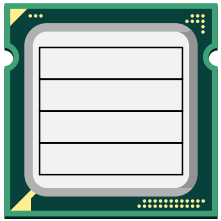

DRAM access, slow
printf("%d", i);
printf("%d", i);
Cache miss
Cache hit
i
Request
Response
No DRAM access, much faster

Shared Memory

ATTACKER

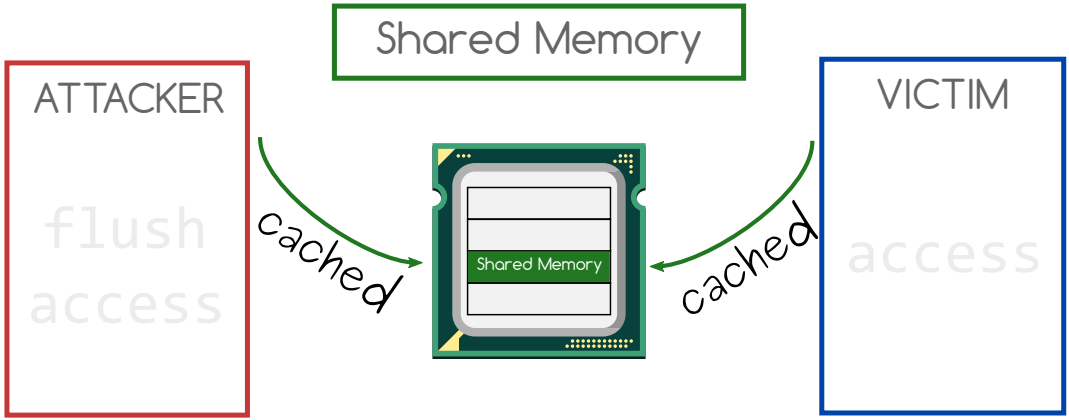
flush

access



VICTIM

access

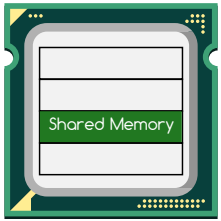

Shared Memory
ATTACKER
flush
access
cached
Shared Memory
cached
VICTIM
access

Shared Memory

ATTACKER

flush

access




VICTIM

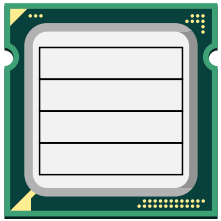
access

Shared Memory

ATTACKER

flush

access



VICTIM

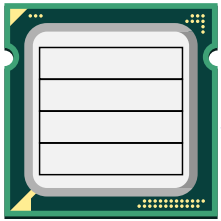
access

Shared Memory

ATTACKER

flush

access



VICTIM

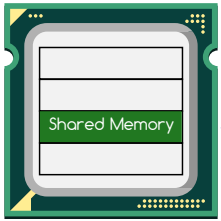
access

Shared Memory

ATTACKER

flush

access




VICTIM

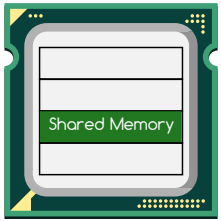
access

Shared Memory

ATTACKER

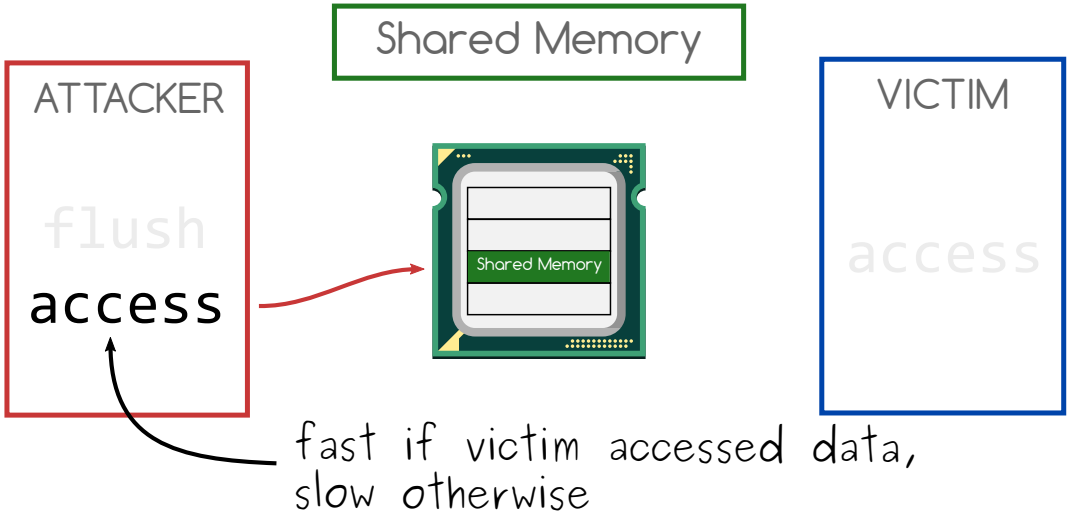
flush

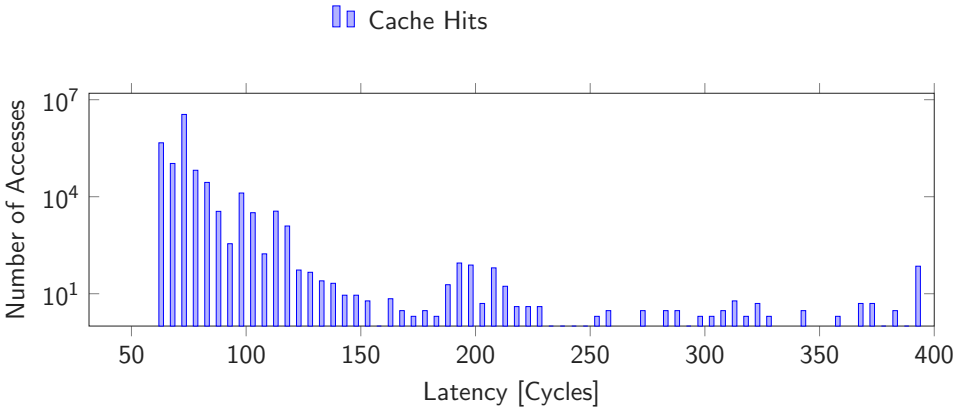
access

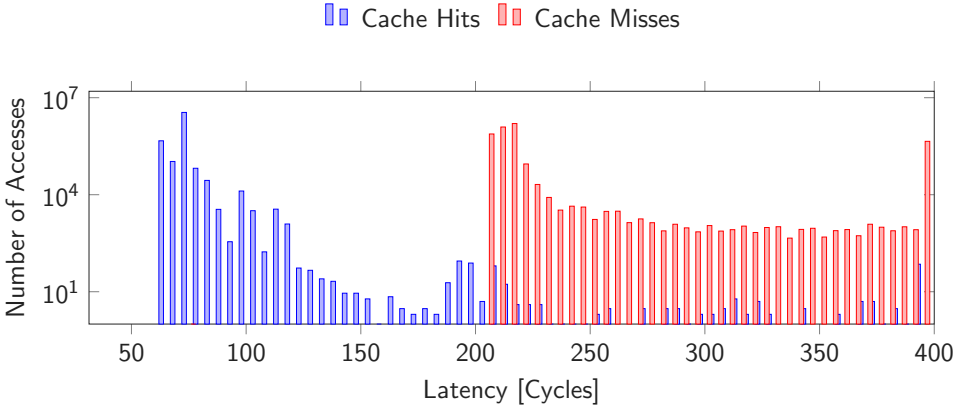


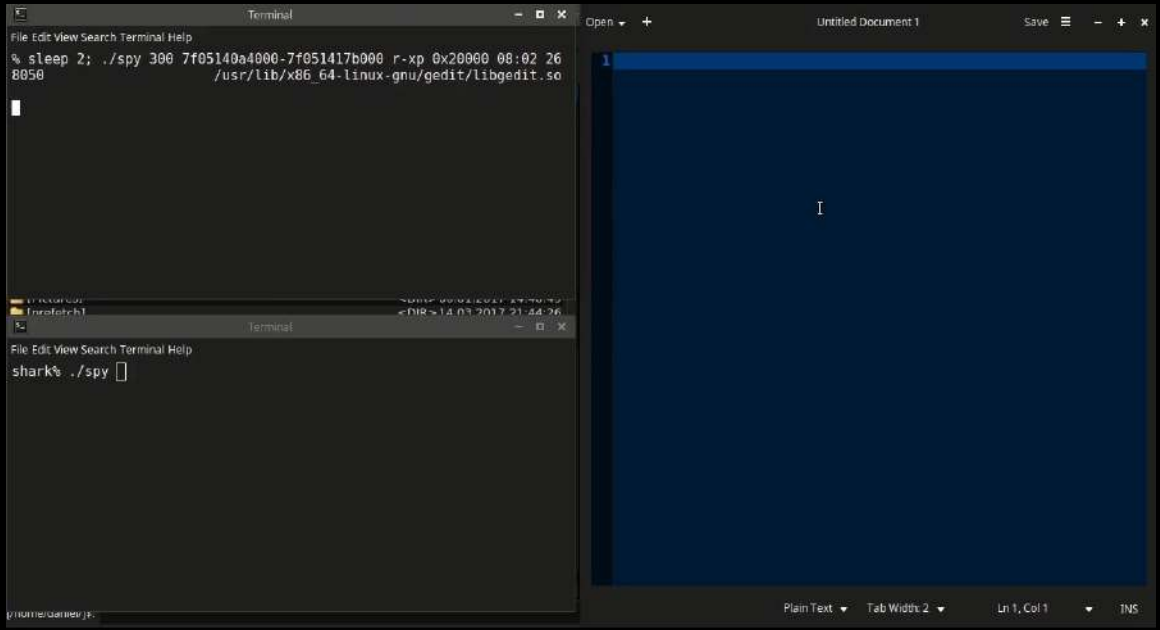

VICTIM

access


Shared Memory
ATTACKER
flush
access
Shared Memory
VICTIM
access
fast if victim accessed data,
slow otherwise

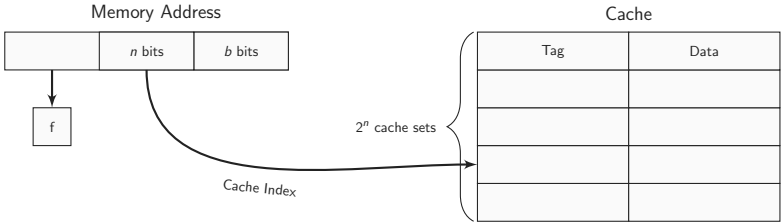

Cache Hits
Number of Accesses
$10^7$
$10^4$
$10^1$
50
100
150
200
250
300
350
400
Latency [Cycles]


Cache Hits
Cache Misses
Number of Accesses
$10^7$
$10^4$
$10^1$
50
100
150
200
250
300
350
400
Latency [Cycles]

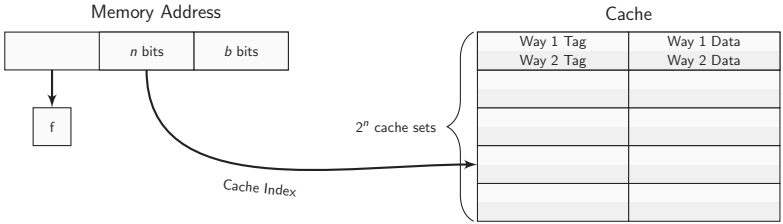



Cache Template Attack Demo

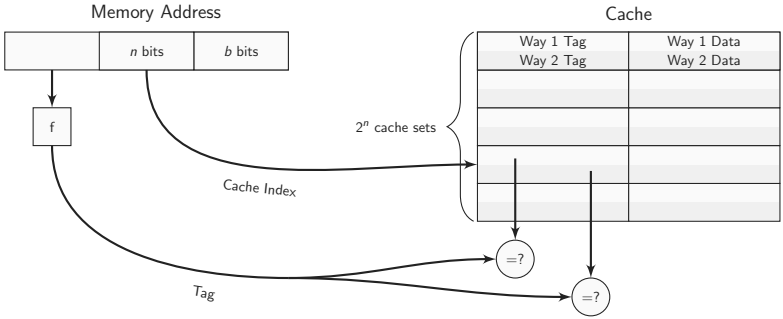
# 2-way set associativity



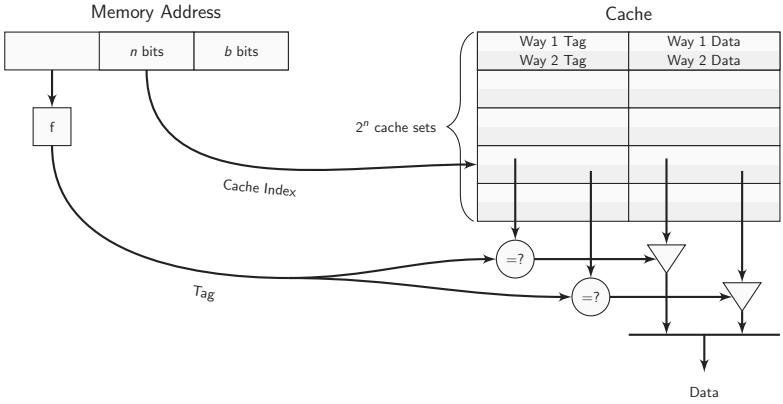
# 2-way set associativity



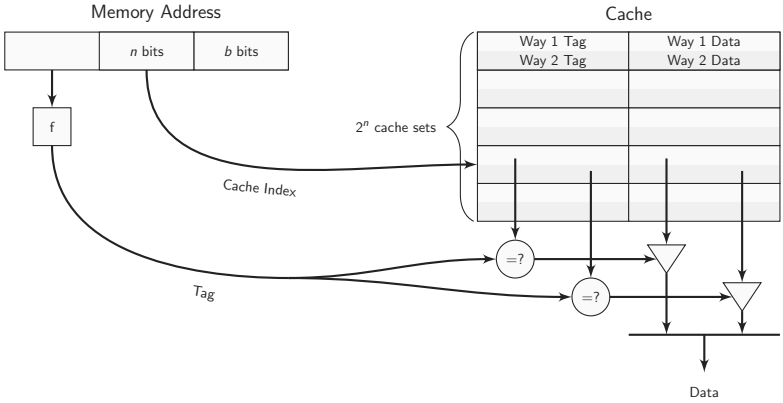
# 2-way set associativity



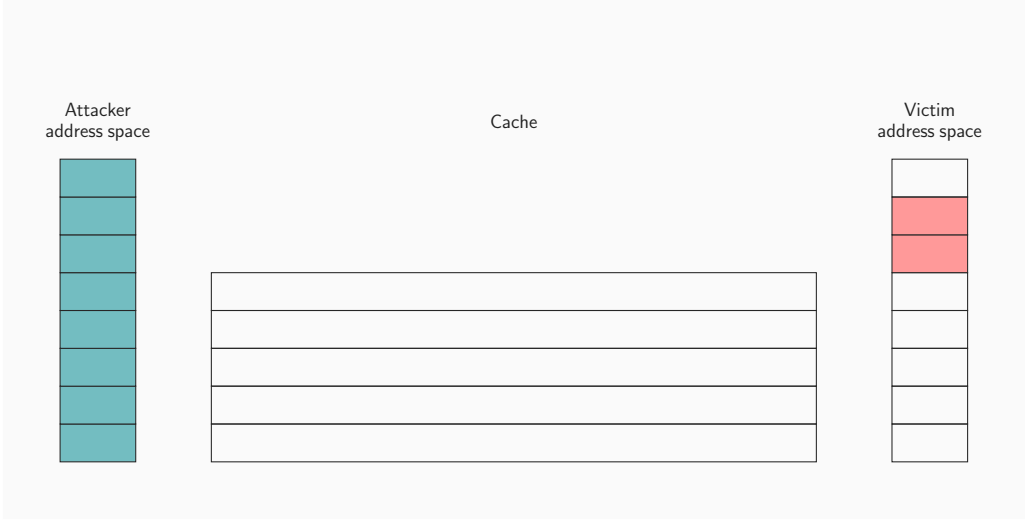
# 2-way set associativity

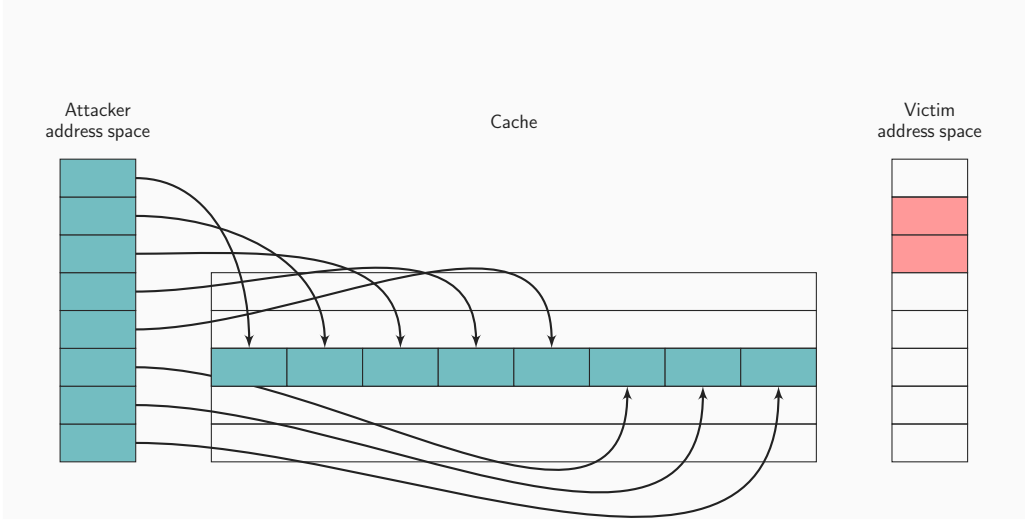


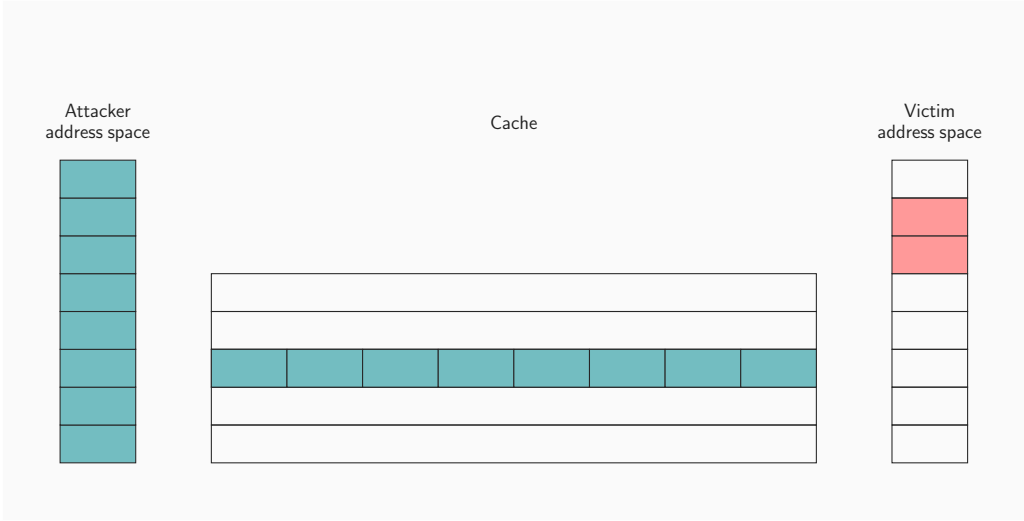
# 2-way set associativity

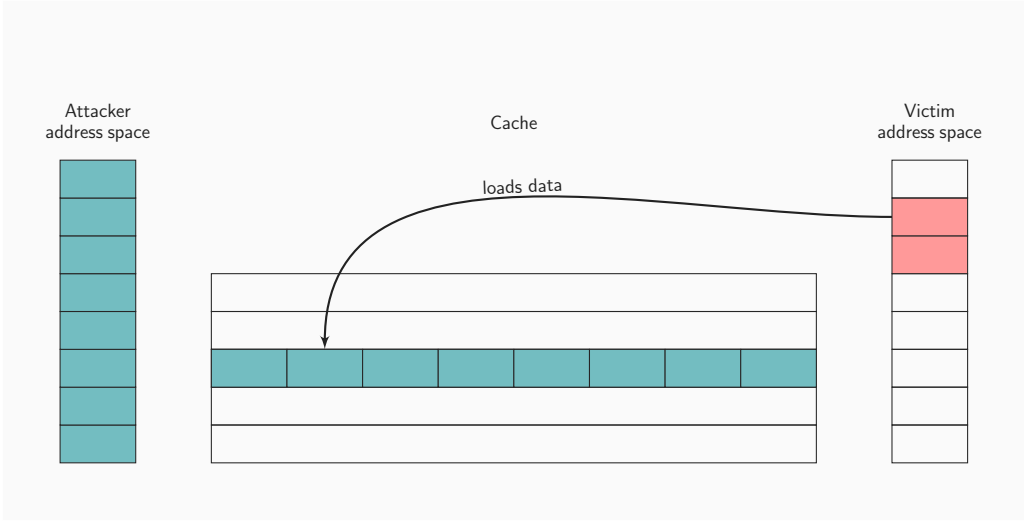


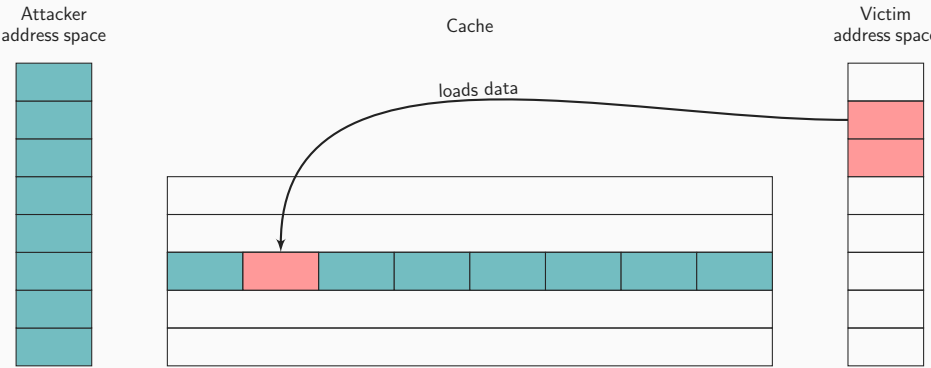

→ replacement policy

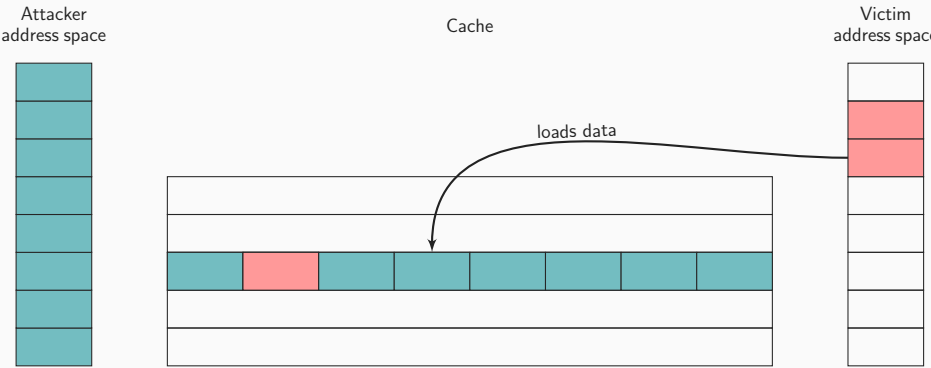

Attacker
address space
Cache
Victim
address space

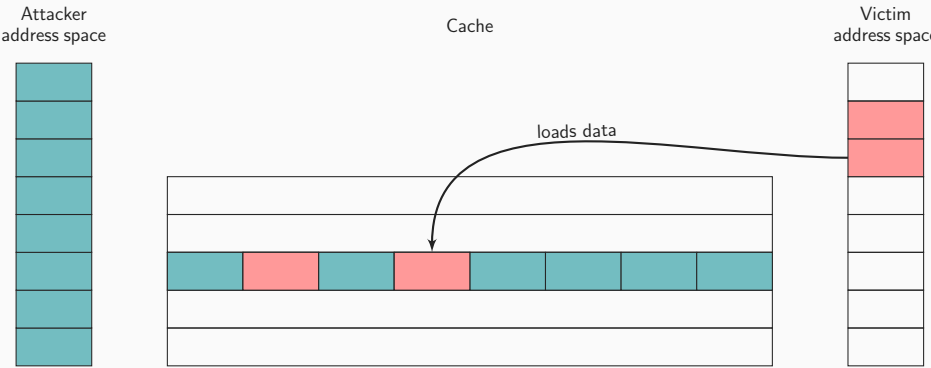

Attacker
address space
Cache
Victim
address space

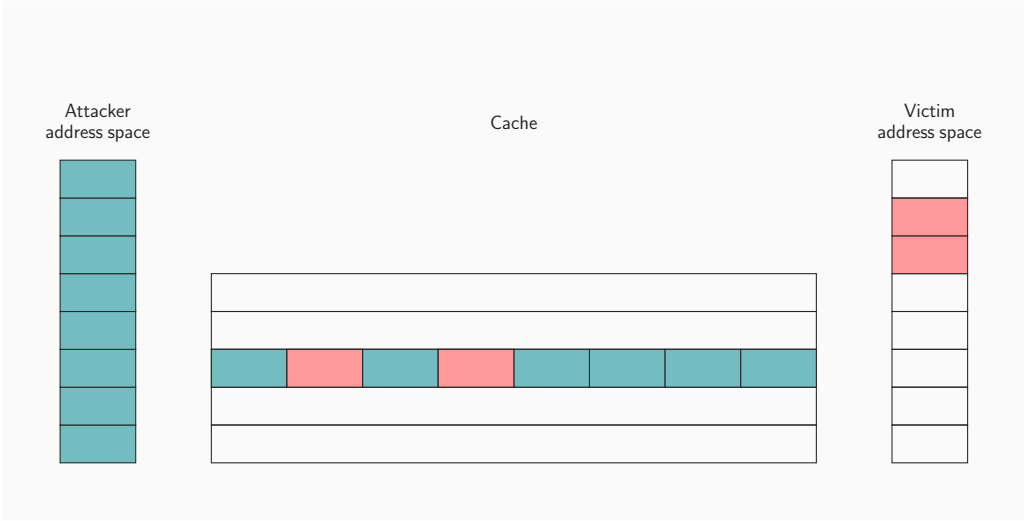

Attacker
address space
Cache
Victim
address space

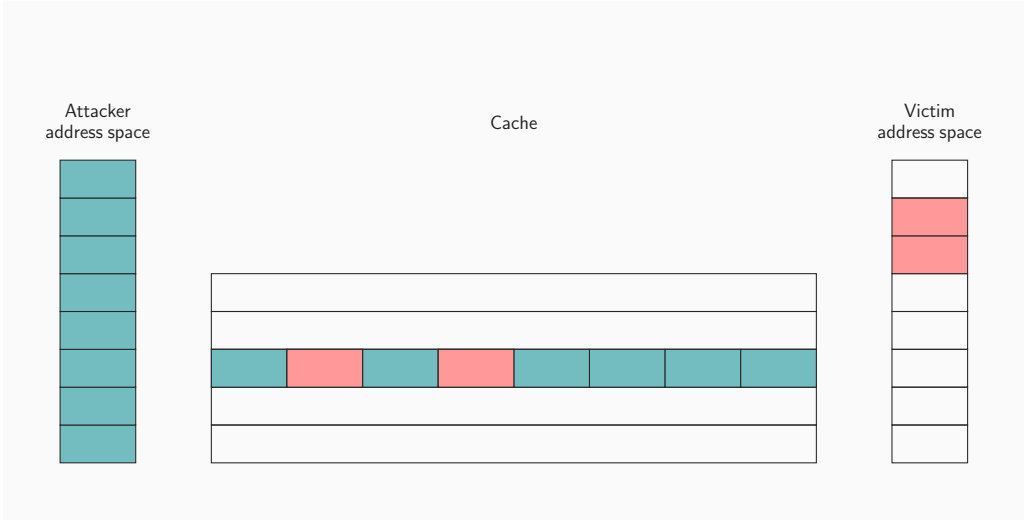

Attacker
address space
Cache
Victim
address space
loads data

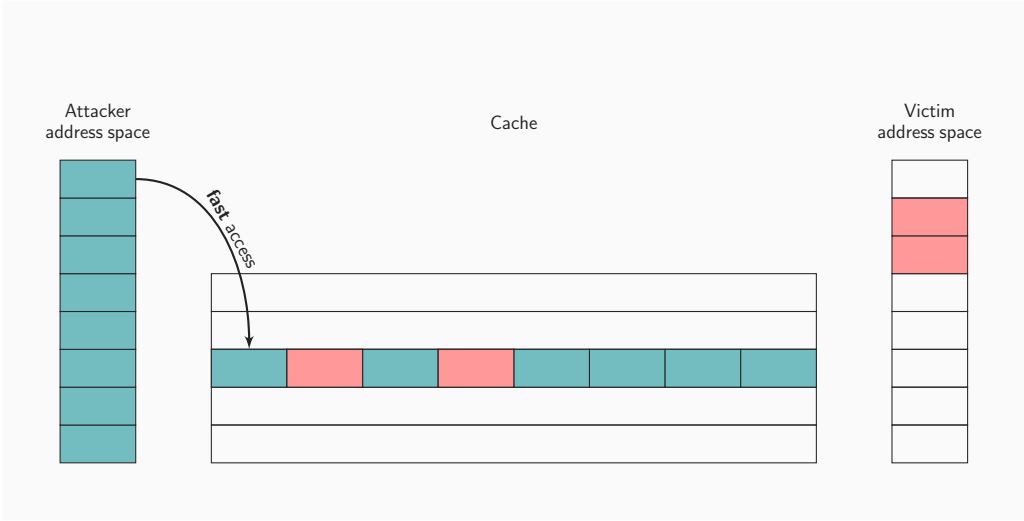

Attacker
address space
Cache
Victim
address space
loads data

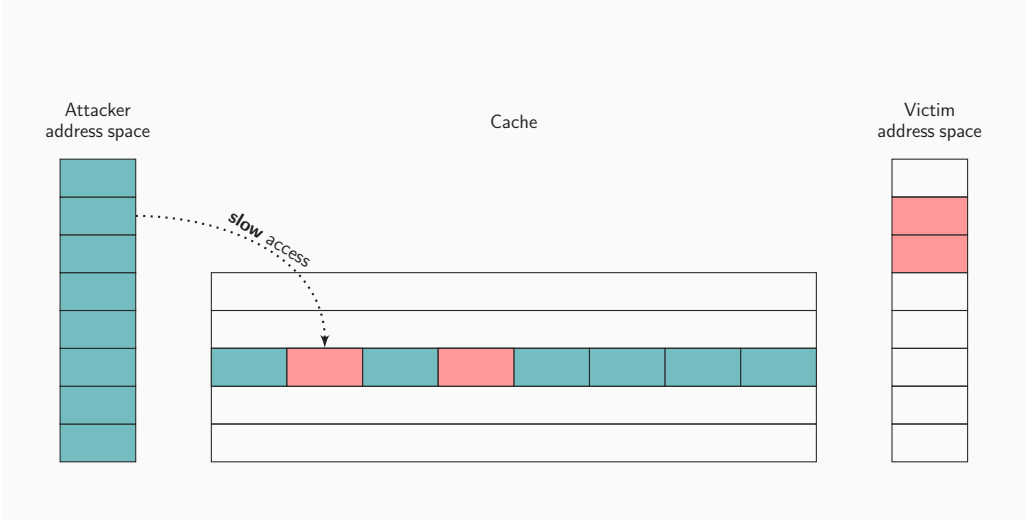

Attacker
address space
Cache
Victim
address space
loads data


Attacker
address space
Cache
Victim
address space
loads data


Attacker
address space
Cache
Victim
address space


Attacker
address space
Cache
Victim
address space


Attacker
address space
Cache
Victim
address space
fast access


Attacker
address space
Cache
Victim
address space
slow access

Pros: less restrictive

1. no need for `clflush` instruction (not available e.g., in JS)

Pros: less restrictive

1. no need for `clflush` instruction (not available e.g., in JS)
2. no need for shared memory ($\rightarrow$ cross-VM)

Pros: less restrictive

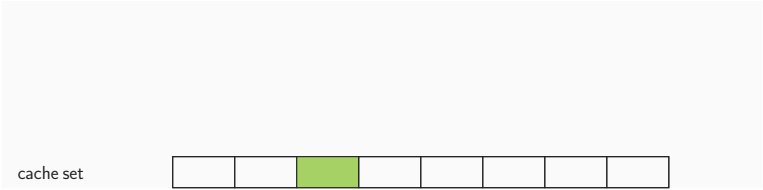
1. no need for `clflush` instruction (not available e.g., in JS)
2. no need for shared memory ($\rightarrow$ cross-VM)

Pros: less restrictive

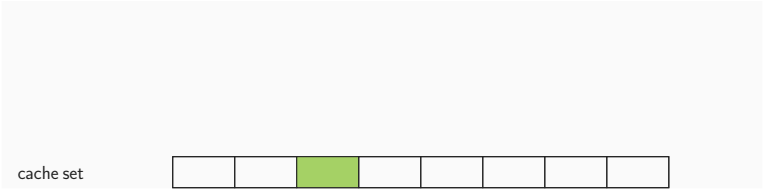
1. no need for `clflush` instruction (not available e.g., in JS)
2. no need for shared memory ($\rightarrow$ cross-VM)

Cons: coarser granularity (1 set)

"LRU eviction" memory accesses

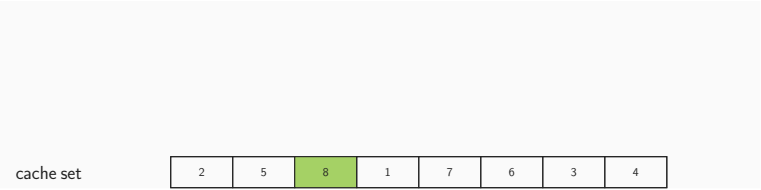


"LRU eviction" memory accesses



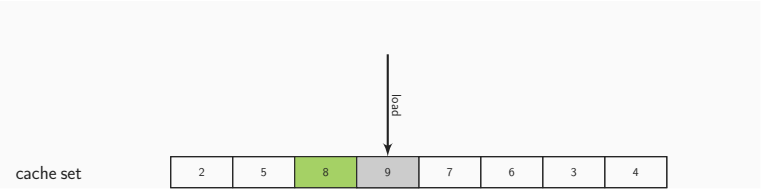

- LRU replacement policy: oldest entry first

"LRU eviction" memory accesses



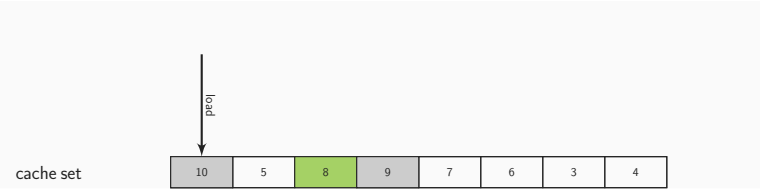

- LRU replacement policy: oldest entry first
- timestamps for every cache line

"LRU eviction" memory accesses



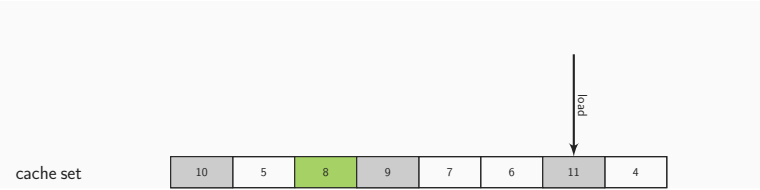

- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp

"LRU eviction" memory accesses



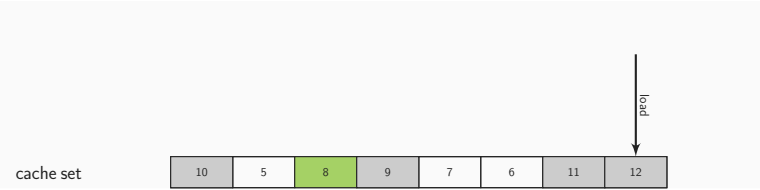

- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp

"LRU eviction" memory accesses



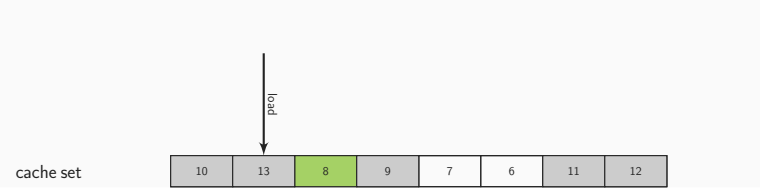

- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp

"LRU eviction" memory accesses




- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp

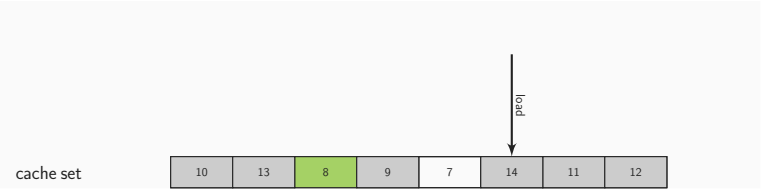
"LRU eviction" memory accesses




- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp

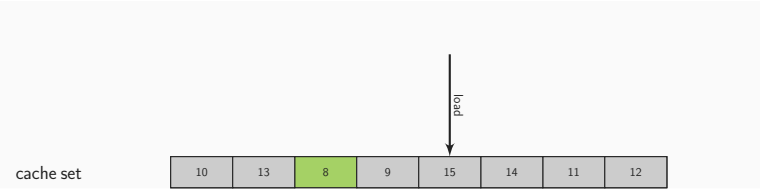
# Replacement policy on older CPUs

"LRU eviction" memory accesses



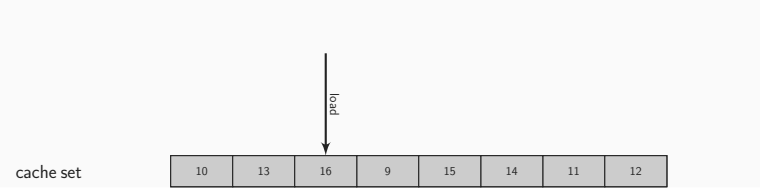

- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp

"LRU eviction" memory accesses

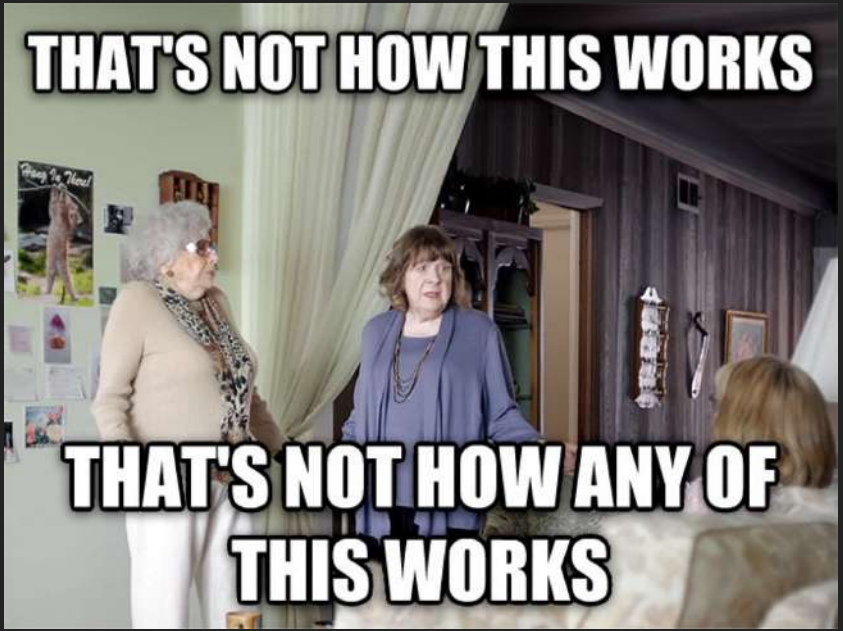



- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp

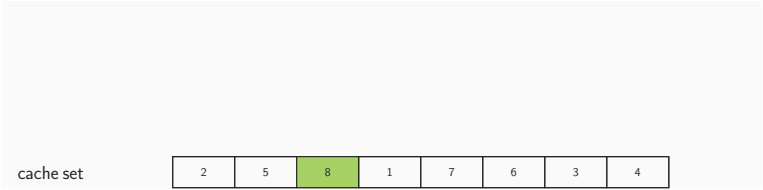
"LRU eviction" memory accesses




- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp

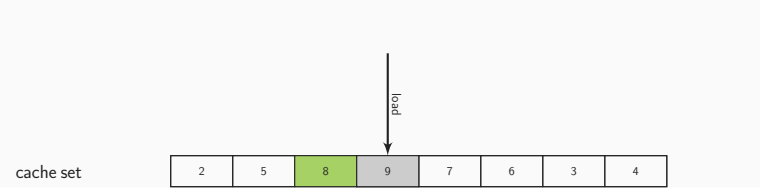

THAT'S NOT HOW THIS WORKS
THAT'S NOT HOW ANY OF THIS WORKS

"LRU eviction" memory accesses



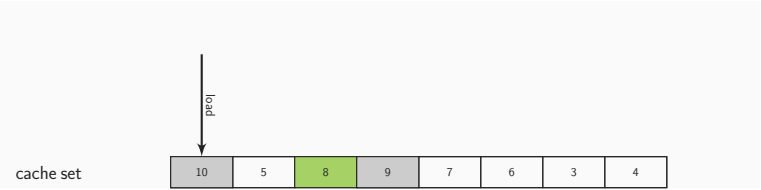

- no LRU replacement

"LRU eviction" memory accesses



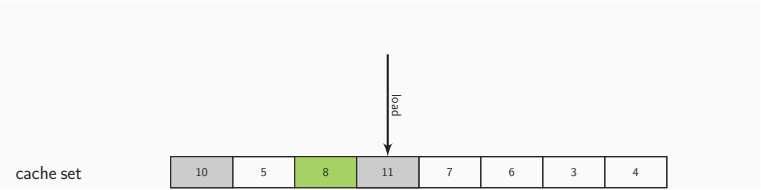

- no LRU replacement

"LRU eviction" memory accesses



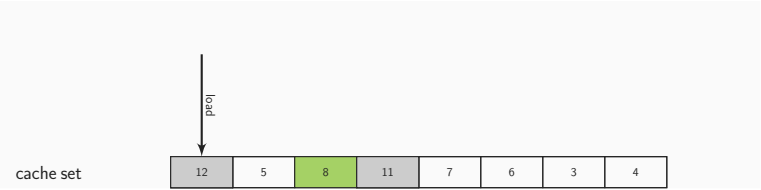

- no LRU replacement

"LRU eviction" memory accesses



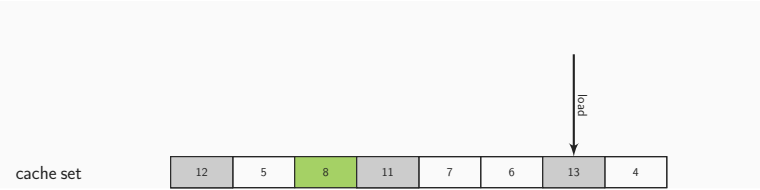

- no LRU replacement

"LRU eviction" memory accesses



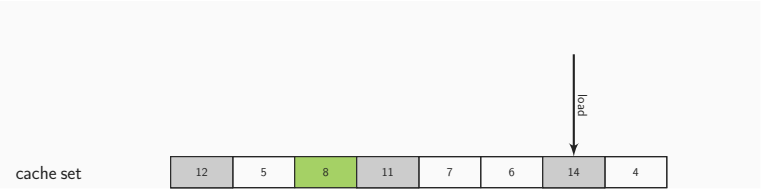

- no LRU replacement

"LRU eviction" memory accesses



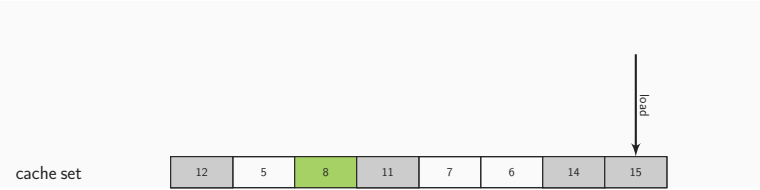

- no LRU replacement

"LRU eviction" memory accesses



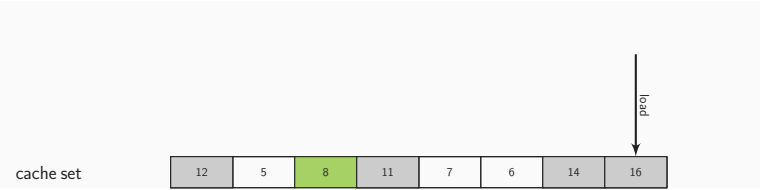

- no LRU replacement

"LRU eviction" memory accesses




- no LRU replacement

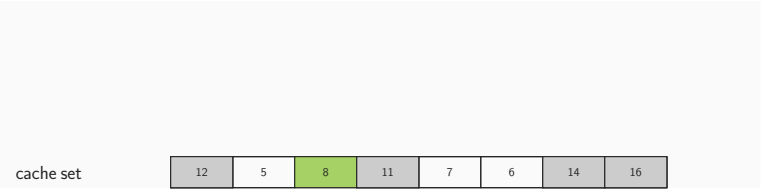
"LRU eviction" memory accesses




- no LRU replacement

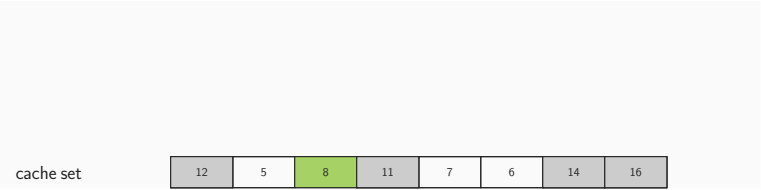
# Replacement policy on recent CPUs

"LRU eviction" memory accesses




- no LRU replacement
- only 75% success rate on Haswell

"LRU eviction" memory accesses



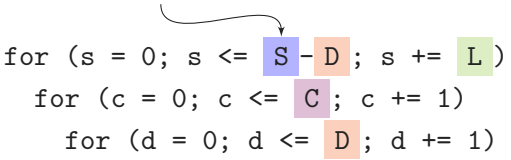

- no LRU replacement
- only 75% success rate on Haswell
- more accesses → higher success rate, but **too slow**

Write eviction strategies as: *P-C-D-L-S*

```
for (s = 0; s <= S-D; s += L)
  for (c = 0; c <= C; c += 1)
    for (d = 0; d <= D; d += 1)
      *a[s+d];
```

Write eviction strategies as: *P-C-D-L-S*

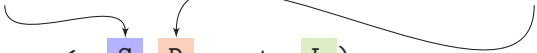
*S*: total number of different addresses
(= set size)



Write eviction strategies as: *P-C-D-L-S*

*S*: total number of different addresses (= set size)

*D*: different addresses per inner access loop

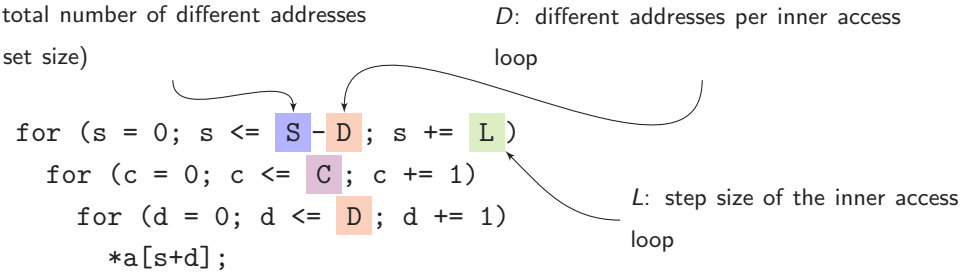


```
for (s = 0; s <= S - D; s += L)
  for (c = 0; c <= C; c += 1)
    for (d = 0; d <= D; d += 1)
      *a[s+d];
```

Write eviction strategies as: *P-C-D-L-S*

*S*: total number of different addresses (= set size)

*D*: different addresses per inner access loop

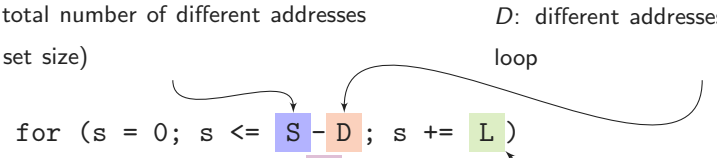



*L*: step size of the inner access loop

Write eviction strategies as: *P-C-D-L-S*

*S*: total number of different addresses (= set size)

*D*: different addresses per inner access loop




*L*: step size of the inner access loop

*C*: number of repetitions of the inner access loop

We evaluated more than 10000 strategies...[1]

| strategy | # accesses | eviction rate | loop time |
|---|---|---|---|
| *P*-1-1-1-17 | 17 | | |
| *P*-1-1-1-20 | 20 | | |

[1] Executed in a loop, on a Haswell with a 16-way last-level cache

We evaluated more than 10000 strategies...[1]

| strategy | # accesses | eviction rate | loop time |
|---|---|---|---|
| *P*-1-1-1-17 | 17 | 74.46% ✗ | |
| *P*-1-1-1-20 | 20 | 99.82% ✓ | |

[1] Executed in a loop, on a Haswell with a 16-way last-level cache

We evaluated more than 10000 strategies...[1]

| strategy | # accesses | eviction rate | loop time |
|---|---|---|---|
| *P*-1-1-1-17 | 17 | 74.46% ✗ | 307 ns ✓ |
| *P*-1-1-1-20 | 20 | 99.82% ✓ | 934 ns ✗ |

[1] Executed in a loop, on a Haswell with a 16-way last-level cache

We evaluated more than 10000 strategies...[1]

| strategy | # accesses | eviction rate | loop time |
|---|---|---|---|
| $P$-1-1-1-17 | 17 | 74.46% ✗ | 307 ns ✓ |
| $P$-1-1-1-20 | 20 | 99.82% ✓ | 934 ns ✗ |
| $P$-2-1-1-17 | 34 | | |

[1] Executed in a loop, on a Haswell with a 16-way last-level cache

We evaluated more than 10000 strategies...[1]

| strategy | # accesses | eviction rate | loop time |
|---|---|---|---|
| *P*-1-1-1-17 | 17 | 74.46% ✗ | 307 ns ✓ |
| *P*-1-1-1-20 | 20 | 99.82% ✓ | 934 ns ✗ |
| *P*-2-1-1-17 | 34 | 99.86% ✓ | |

[1] Executed in a loop, on a Haswell with a 16-way last-level cache

We evaluated more than 10000 strategies...[1]

| strategy | # accesses | eviction rate | loop time |
|---|---|---|---|
| *P*-1-1-1-17 | 17 | 74.46% ✗ | 307 ns ✓ |
| *P*-1-1-1-20 | 20 | 99.82% ✓ | 934 ns ✗ |
| *P*-2-1-1-17 | 34 | 99.86% ✓ | 191 ns ✓ |

[1] Executed in a loop, on a Haswell with a 16-way last-level cache

We evaluated more than 10000 strategies...[1]

| strategy | # accesses | eviction rate | loop time |
|---|---|---|---|
| *P*-1-1-1-17 | 17 | 74.46% ✗ | 307 ns ✓ |
| *P*-1-1-1-20 | 20 | 99.82% ✓ | 934 ns ✗ |
| *P*-2-1-1-17 | 34 | 99.86% ✓ | 191 ns ✓ |
| *P*-2-2-1-17 | 64 | | |

[1] Executed in a loop, on a Haswell with a 16-way last-level cache

We evaluated more than 10000 strategies...[1]

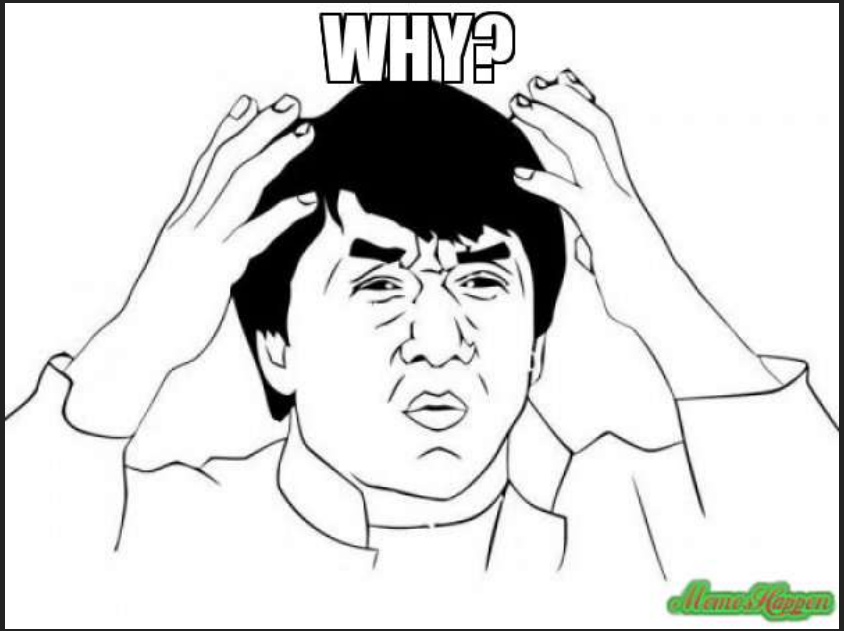
| strategy | # accesses | eviction rate | loop time |
|---|---|---|---|
| *P*-1-1-1-17 | 17 | 74.46% ✗ | 307 ns ✓ |
| *P*-1-1-1-20 | 20 | 99.82% ✓ | 934 ns ✗ |
| *P*-2-1-1-17 | 34 | 99.86% ✓ | 191 ns ✓ |
| *P*-2-2-1-17 | 64 | 99.98% ✓ | |

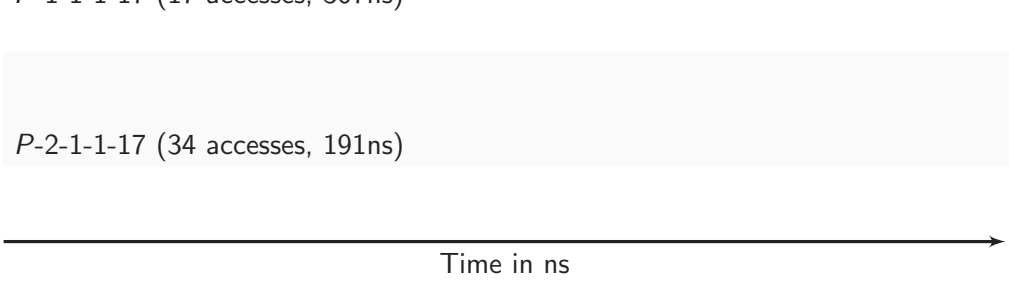
[1] Executed in a loop, on a Haswell with a 16-way last-level cache

We evaluated more than 10000 strategies...[1]

| strategy | # accesses | eviction rate | loop time |
|---|---|---|---|
| *P*-1-1-1-17 | 17 | 74.46% ✗ | 307 ns ✓ |
| *P*-1-1-1-20 | 20 | 99.82% ✓ | 934 ns ✗ |
| *P*-2-1-1-17 | 34 | 99.86% ✓ | 191 ns ✓ |
| *P*-2-2-1-17 | 64 | 99.98% ✓ | 180 ns ✓ |

[1] Executed in a loop, on a Haswell with a 16-way last-level cache


WHY?
MemeHappen


P-1-1-1-17 (17 accesses, 307ns)
P-2-1-1-17 (34 accesses, 191ns)
Time in ns

# Cache eviction strategies: Illustration

# Cache eviction strategies: Illustration

# Cache eviction strategies: Illustration

*P*-1-1-1-17 (17 accesses, 307ns)


*P*-2-1-1-17 (34 accesses, 191ns)


Time in ns

*P*-1-1-1-17 (17 accesses, 307ns)


*P*-2-1-1-17 (34 accesses, 191ns)


Time in ns

*P*-1-1-1-17 (17 accesses, 307ns)


*P*-2-1-1-17 (34 accesses, 191ns)


Time in ns

*P*-1-1-1-17 (17 accesses, 307ns)


*P*-2-1-1-17 (34 accesses, 191ns)


Time in ns

# Cache eviction strategies: Illustration

*P*-1-1-1-17 (17 accesses, 307ns)


*P*-2-1-1-17 (34 accesses, 191ns)


Time in ns

*P*-1-1-1-17 (17 accesses, 307ns)


*P*-2-1-1-17 (34 accesses, 191ns)


Time in ns

*P*-1-1-1-17 (17 accesses, 307ns)


*P*-2-1-1-17 (34 accesses, 191ns)

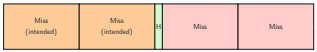

Time in ns

*P*-1-1-1-17 (17 accesses, 307ns)

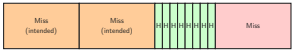

*P*-2-1-1-17 (34 accesses, 191ns)


Time in ns

*P*-1-1-1-17 (17 accesses, 307ns)

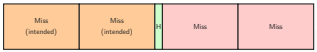



*P*-2-1-1-17 (34 accesses, 191ns)




Time in ns

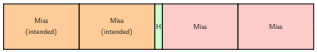
*P*-1-1-1-17 (17 accesses, 307ns)



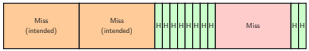

*P*-2-1-1-17 (34 accesses, 191ns)


Time in ns

*P*-1-1-1-17 (17 accesses, 307ns)



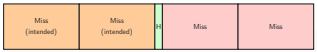

*P*-2-1-1-17 (34 accesses, 191ns)




Time in ns

# Cache eviction strategies: Illustration

*P*-1-1-1-17 (17 accesses, 307ns)



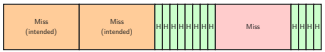

*P*-2-1-1-17 (34 accesses, 191ns)


Time in ns

# Cache eviction strategies: Illustration

*P*-1-1-1-17 (17 accesses, 307ns)


*P*-2-1-1-17 (34 accesses, 191ns)




Time in ns

# Cache eviction strategies: Illustration

*P*-1-1-1-17 (17 accesses, 307ns)

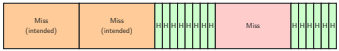

*P*-2-1-1-17 (34 accesses, 191ns)


Time in ns

# Cache eviction strategies: Illustration

*P*-1-1-1-17 (17 accesses, 307ns)


*P*-2-1-1-17 (34 accesses, 191ns)

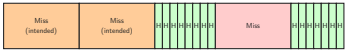



Time in ns

# Cache eviction strategies: Illustration

*P*-1-1-1-17 (17 accesses, 307ns)


*P*-2-1-1-17 (34 accesses, 191ns)

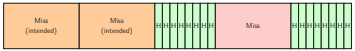



Time in ns

# Cache eviction strategies: Illustration

*P*-1-1-1-17 (17 accesses, 307ns)


*P*-2-1-1-17 (34 accesses, 191ns)




Time in ns

# Cache eviction strategies: Illustration

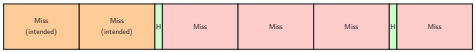
*P*-1-1-1-17 (17 accesses, 307ns)


*P*-2-1-1-17 (34 accesses, 191ns)


Time in ns

# Cache eviction strategies: Illustration

*P*-1-1-1-17 (17 accesses, 307ns)

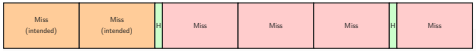



*P*-2-1-1-17 (34 accesses, 191ns)


Time in ns

# Cache eviction strategies: Illustration

*P*-1-1-1-17 (17 accesses, 307ns)

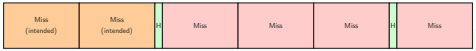



*P*-2-1-1-17 (34 accesses, 191ns)


Time in ns

# Cache eviction strategies: Illustration

*P*-1-1-1-17 (17 accesses, 307ns)

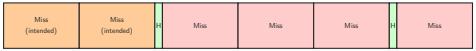



*P*-2-1-1-17 (34 accesses, 191ns)


Time in ns

# Cache eviction strategies: Illustration

*P*-1-1-1-17 (17 accesses, 307ns)

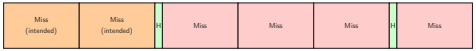



*P*-2-1-1-17 (34 accesses, 191ns)


Time in ns

# Cache eviction strategies: Illustration

*P*-1-1-1-17 (17 accesses, 307ns)

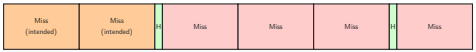



*P*-2-1-1-17 (34 accesses, 191ns)


Time in ns

# Cache eviction strategies: Illustration

*P*-1-1-1-17 (17 accesses, 307ns)

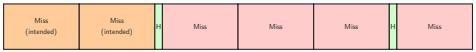



*P*-2-1-1-17 (34 accesses, 191ns)


Time in ns

# Cache eviction strategies: Illustration

*P*-1-1-1-17 (17 accesses, 307ns)




*P*-2-1-1-17 (34 accesses, 191ns)


Time in ns

# Cache eviction strategies: Illustration

*P*-1-1-1-17 (17 accesses, 307ns)


*P*-2-1-1-17 (34 accesses, 191ns)


Time in ns

# Cache eviction strategies: Illustration

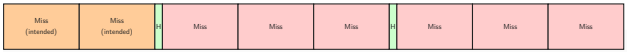
*P*-1-1-1-17 (17 accesses, 307ns)


*P*-2-1-1-17 (34 accesses, 191ns)

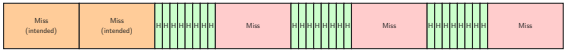

Time in ns

# Cache eviction strategies: Illustration

*P*-1-1-1-17 (17 accesses, 307ns)



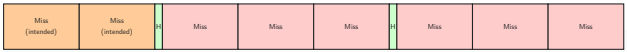

*P*-2-1-1-17 (34 accesses, 191ns)




Time in ns

# Cache eviction strategies: Illustration

*P*-1-1-1-17 (17 accesses, 307ns)

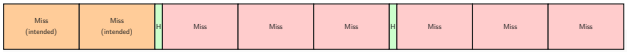



*P*-2-1-1-17 (34 accesses, 191ns)


Time in ns

# Cache eviction strategies: Illustration

*P*-1-1-1-17 (17 accesses, 307ns)

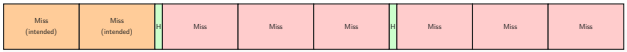



*P*-2-1-1-17 (34 accesses, 191ns)

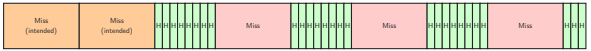

Time in ns

# Cache eviction strategies: Illustration

*P*-1-1-1-17 (17 accesses, 307ns)



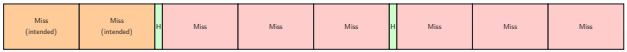

*P*-2-1-1-17 (34 accesses, 191ns)



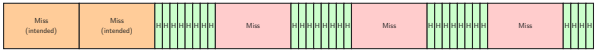

Time in ns

# Cache eviction strategies: Illustration

*P*-1-1-1-17 (17 accesses, 307ns)



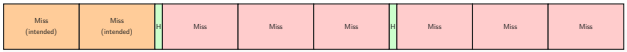

*P*-2-1-1-17 (34 accesses, 191ns)



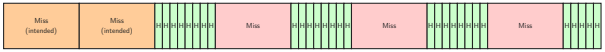

Time in ns

# Cache eviction strategies: Illustration

*P*-1-1-1-17 (17 accesses, 307ns)



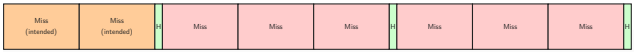

*P*-2-1-1-17 (34 accesses, 191ns)



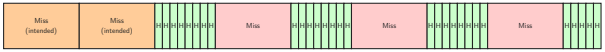

Time in ns

# Cache eviction strategies: Illustration

*P*-1-1-1-17 (17 accesses, 307ns)



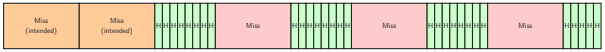

*P*-2-1-1-17 (34 accesses, 191ns)




Time in ns

*P*-1-1-1-17 (17 accesses, 307ns)

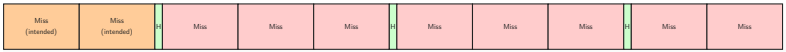

*P*-2-1-1-17 (34 accesses, 191ns)




Time in ns

# Cache eviction strategies: Illustration

*P*-1-1-1-17 (17 accesses, 307ns)



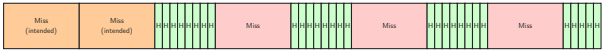

*P*-2-1-1-17 (34 accesses, 191ns)


Time in ns

# Cache eviction strategies: Illustration

*P*-1-1-1-17 (17 accesses, 307ns)


*P*-2-1-1-17 (34 accesses, 191ns)



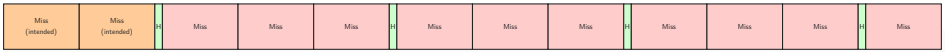

Time in ns

*P*-1-1-1-17 (17 accesses, 307ns)


*P*-2-1-1-17 (34 accesses, 191ns)


Time in ns

*P*-1-1-1-17 (17 accesses, 307ns)

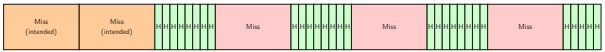



*P*-2-1-1-17 (34 accesses, 191ns)


Time in ns

*P*-1-1-1-17 (17 accesses, 307ns)


*P*-2-1-1-17 (34 accesses, 191ns)




Time in ns

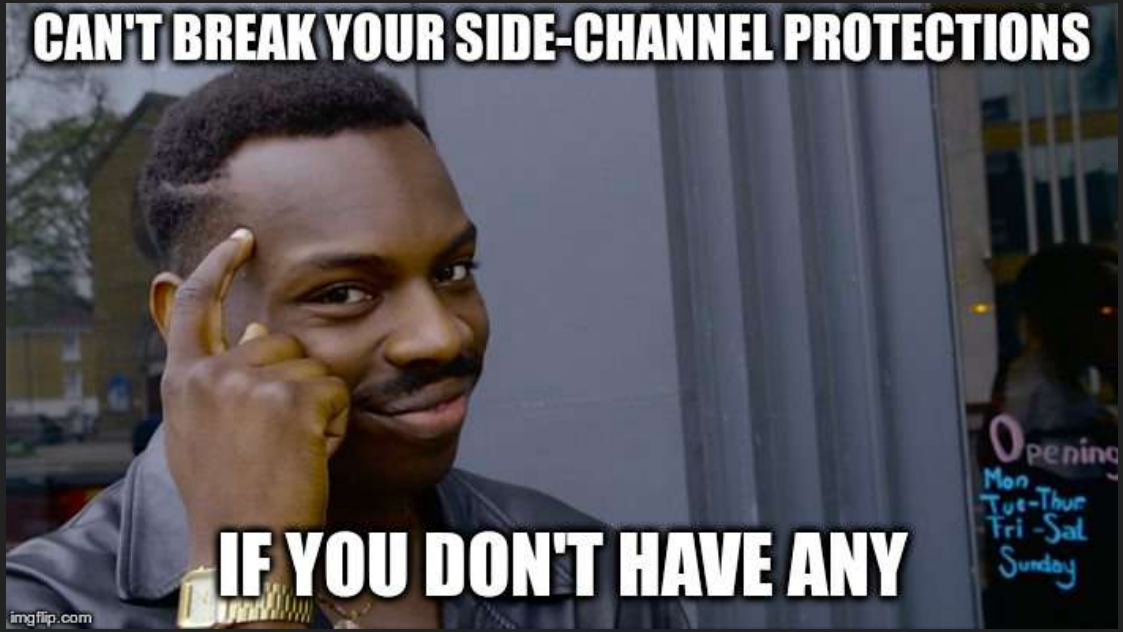
## Protection from Side-Channel Attacks

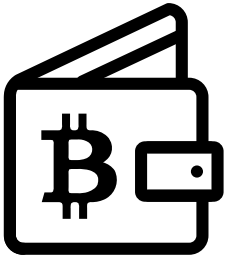
## Protection from Side-Channel Attacks

Intel SGX does not provide explicit protection from side-channel attacks.

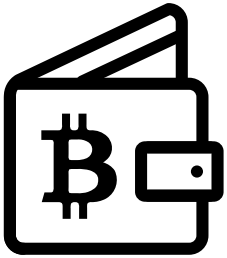
## Protection from Side-Channel Attacks

Intel SGX does not provide explicit protection from side-channel attacks. It is the enclave developer's responsibility to address side-channel attack concerns.


CAN'T BREAK YOUR SIDE-CHANNEL PROTECTIONS
IF YOU DON'T HAVE ANY
Opening
Mon
Tue-Thur
Fri-Sat
Sunday
imgflip.com



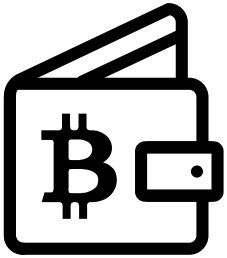
- Ledger SGX Enclave for blockchain applications
- BitPay Copay Bitcoin wallet
- Teechain payment channel using SGX



- Ledger SGX Enclave for blockchain applications
- BitPay Copay Bitcoin wallet
- Teechain payment channel using SGX

**Teechain**

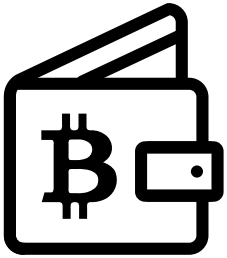
[...] We assume the TEE guarantees to hold



- Ledger SGX Enclave for blockchain applications
- BitPay Copay Bitcoin wallet
- Teechain payment channel using SGX

**Teechain**

[...] We assume the TEE guarantees to hold and do not consider side-channel attacks [5, 35, 46] on the TEE.

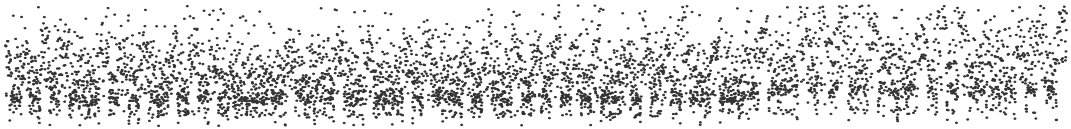


- Ledger SGX Enclave for blockchain applications
- BitPay Copay Bitcoin wallet
- Teechain payment channel using SGX

**Teechain**

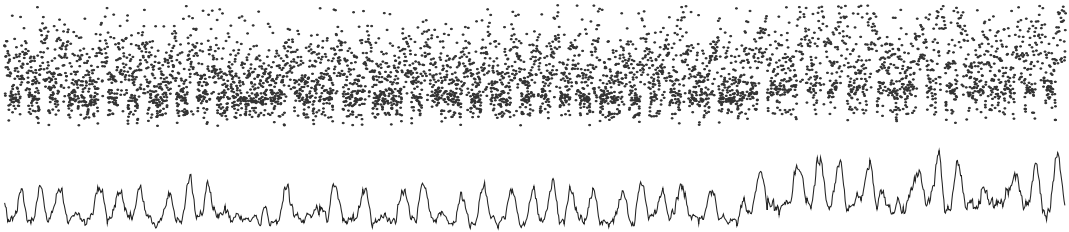
[...] We assume the TEE guarantees to hold and do not consider side-channel attacks [5, 35, 46] on the TEE. Such attacks and their mitigations [36, 43] are outside the scope of this work. [...]

Raw Prime+Probe trace...[2]



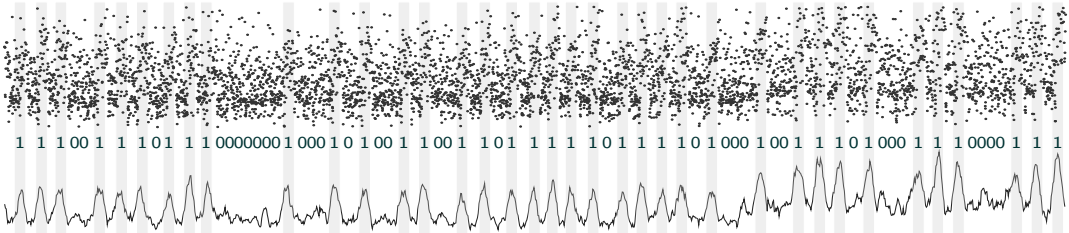
[2] M. Schwarz, D. Gruss, S. Weiser, C. Maurice, and S. Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA. 2017.

...processed with a simple moving average...[3]

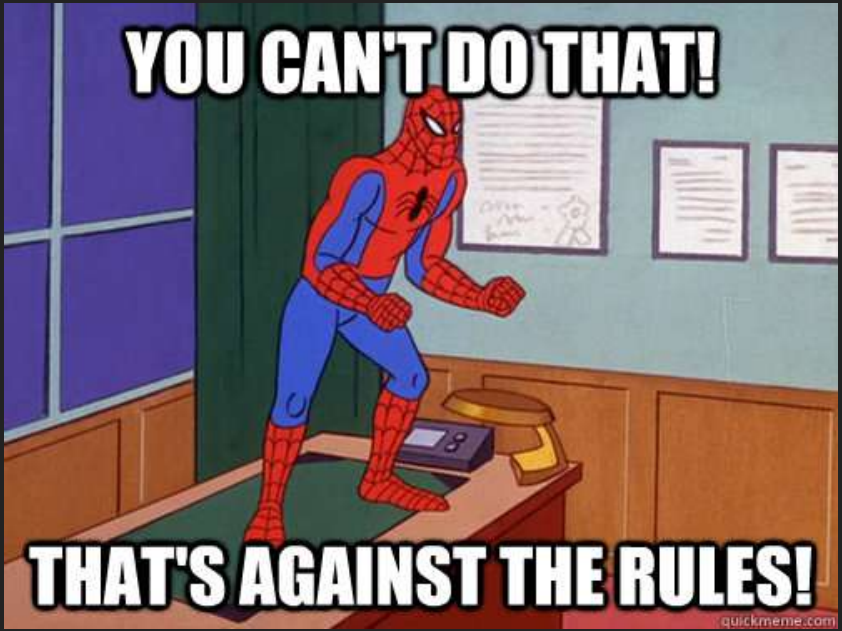


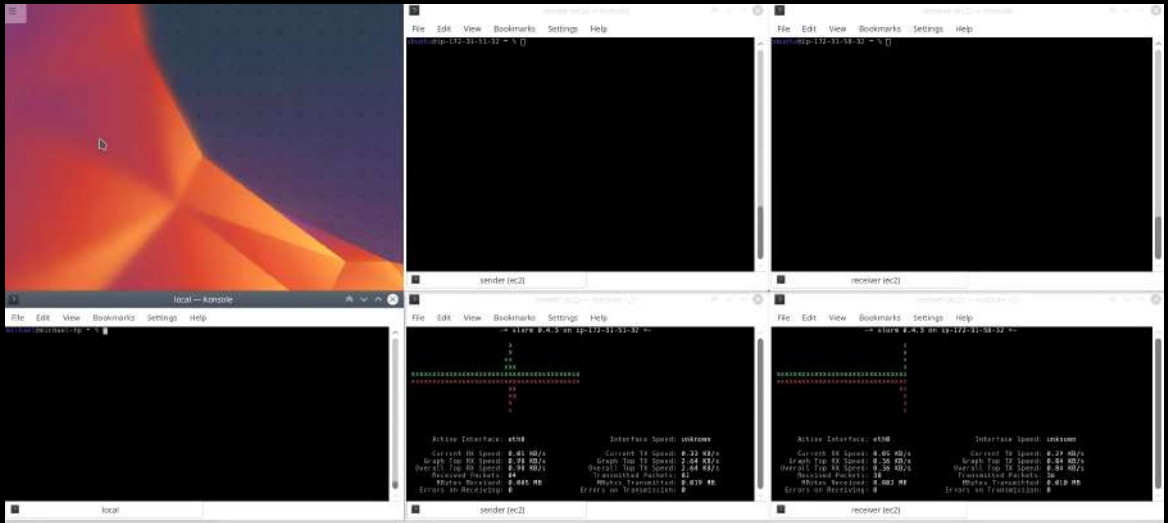
[3] M. Schwarz, D. Gruss, S. Weiser, C. Maurice, and S. Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA. 2017.

...allows to clearly see the bits of the exponent[4]



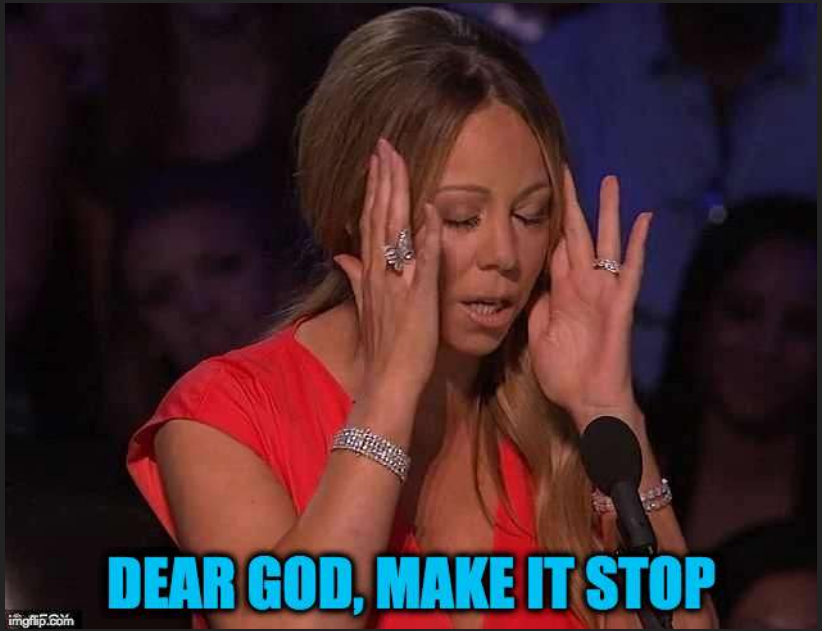

[4]M. Schwarz, D. Gruss, S. Weiser, C. Maurice, and S. Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA. 2017.


YOU CAN'T DO THAT!
THAT'S AGAINST THE RULES!
quickmeme.com




# Hello from the other side (DEMO): Video streaming over cache covert channel

Line 17, Column 1 INSERT Soft Tabs: 4(8) UTF-8 Normal


DEAR GOD, MAKE IT STOP
imgflip.com

# ScatterCache

# What is ScatterCache?

- Secure design for n-way set associative caches

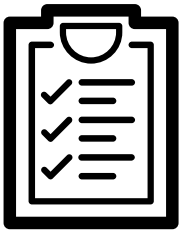
# What is ScatterCache?

- Secure design for n-way set associative caches
  - Addresses $\rightarrow$ cache sets
  - Increases number of possible cache sets
  - IDs for security domains

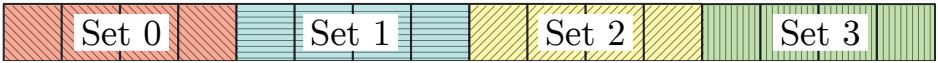
# What is ScatterCache?

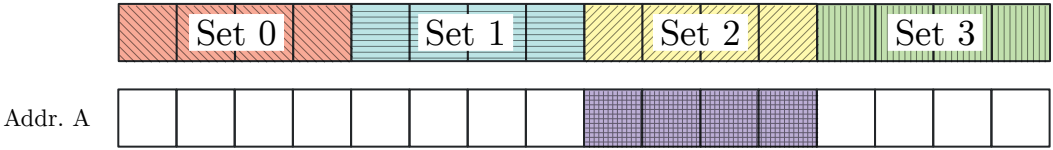
- Secure design for n-way set associative caches
  - Addresses $\rightarrow$ ??? $\rightarrow$ cache sets
  - Increases number of possible cache sets
  - IDs for security domains

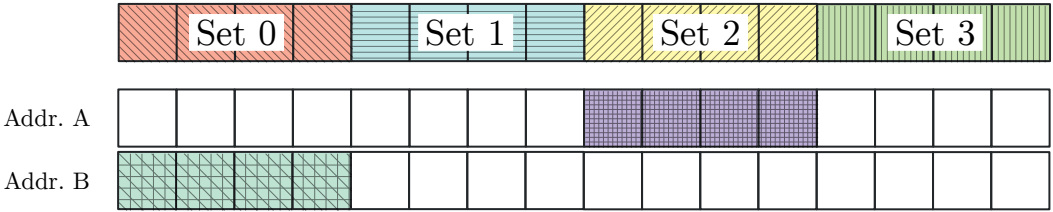
# What is ScatterCache?

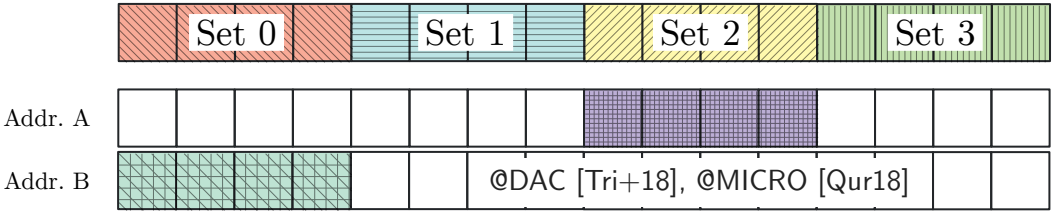


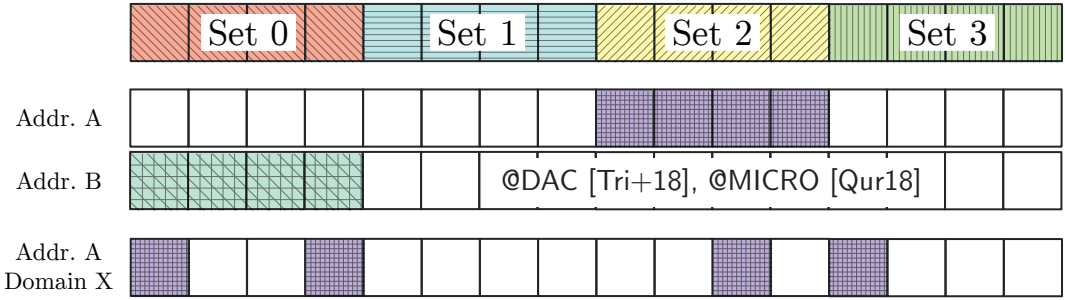
- Secure design for n-way set associative caches
  - Addresses $\rightarrow$ ??? $\rightarrow$ cache sets
  - Increases number of possible cache sets
  - IDs for security domains
- Established/existing concepts
  - Skewed caches / slices
  - Low latency cryptography (e.g., QARMA-64 [Ava17])

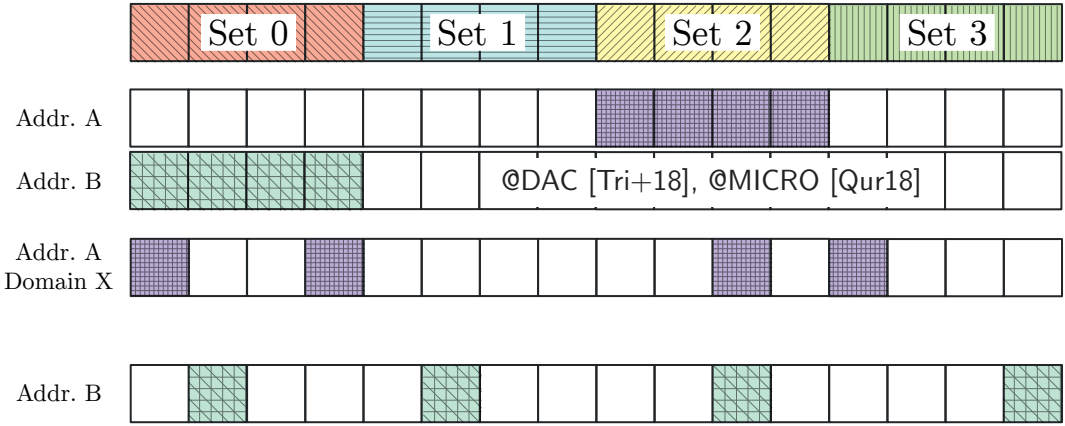

Set 0
Set 1
Set 2
Set 3

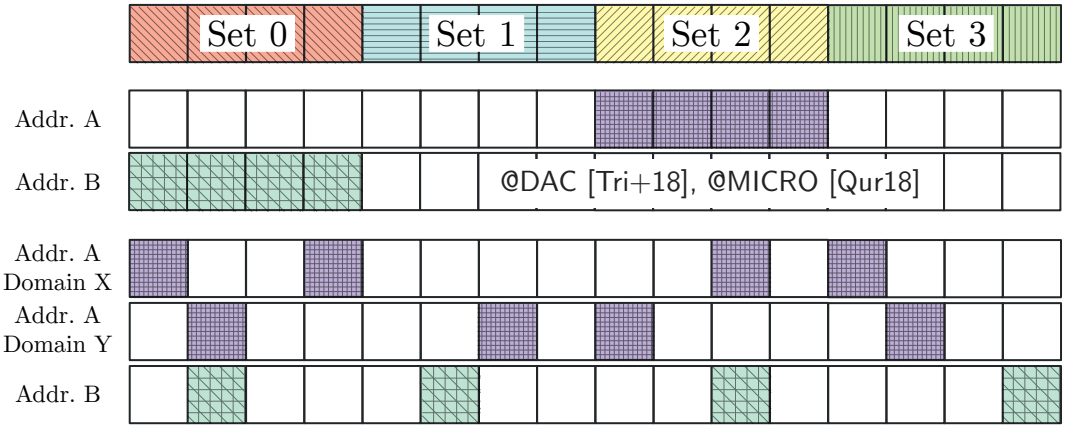

Set 0
Set 1
Set 2
Set 3
Addr. A

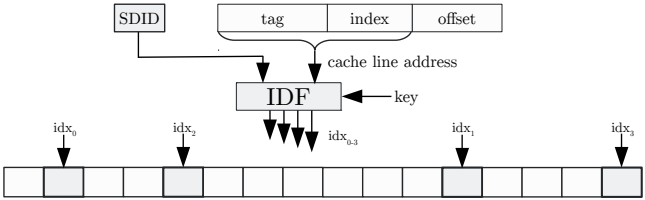

Set 0
Set 1
Set 2
Set 3
Addr. A
Addr. B


Set 0
Set 1
Set 2
Set 3
Addr. A
Addr. B
@DAC [Tri+18], @MICRO [Qur18]


Set 0
Set 1
Set 2
Set 3
Addr. A
Addr. B
@DAC [Tri+18], @MICRO [Qur18]
Addr. A
Domain X

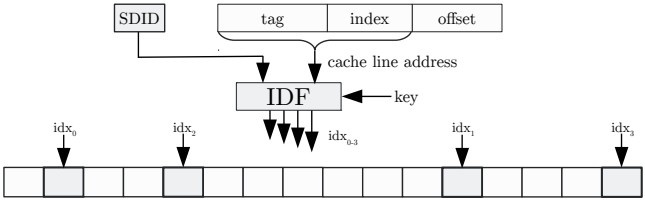

Set 0
Set 1
Set 2
Set 3
Addr. A
Addr. B
@DAC [Tri+18], @MICRO [Qur18]
Addr. A
Domain X
Addr. B


Set 0
Set 1
Set 2
Set 3
Addr. A
Addr. B
@DAC [Tri+18], @MICRO [Qur18]
Addr. A Domain X
Addr. A Domain Y
Addr. B




- Index Derivation Function (IDF) takes an address and returns a cache set
- Depends on hardware key and optional Security Domain ID (SDID)
- $\rightarrow$ Unique combination of cache lines for each address

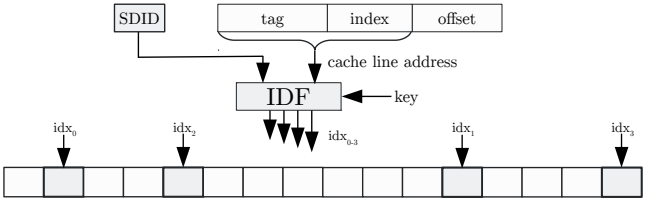
# ScatterCache - Naive Concept




$\binom{n_{ways} \cdot 2^{b_{indices}} + n_{ways} - 1}{n_{ways}}$ possible cache sets

- Index Derivation Function (IDF) takes an address and returns a cache set
- Depends on hardware key and optional Security Domain ID (SDID)
- $\rightarrow$ Unique combination of cache lines for each address

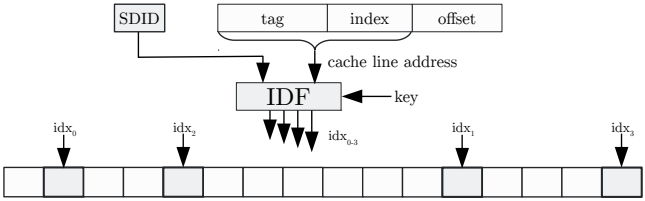
# ScatterCache - Naive Concept




$\binom{n_{ways} \cdot 2^{b_{indices}} + n_{ways} - 1}{n_{ways}}$ possible cache sets

512 KiB (32 B lines), $n_{ways} = 8$, $b_{indices} = 11$
$\rightarrow 2^{96.7}$ sets

- Index Derivation Function (IDF) takes an address and returns a cache set
- Depends on hardware key and optional Security Domain ID (SDID)
- $\rightarrow$ Unique combination of cache lines for each address




$\binom{n_{ways} \cdot 2^{b_{indices}} + n_{ways} - 1}{n_{ways}}$ possible cache sets

512 KiB (32 B lines), $n_{ways} = 8$, $b_{indices} = 11$
$\rightarrow 2^{96.7}$ sets

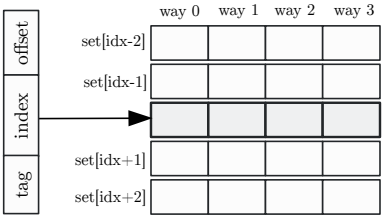
- Index Derivation Function (IDF) takes an address and returns a cache set
- Depends on hardware key and optional Security Domain ID (SDID)
- $\rightarrow$ Unique combination of cache lines for each address

– Potential index collisions
– One $n_{ways}$ multi-port memory

We want something that is closer to a traditional cache!

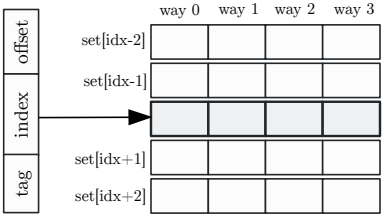
We want something that is closer to a traditional cache!

instead of this:

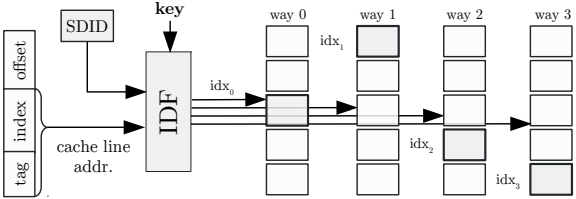


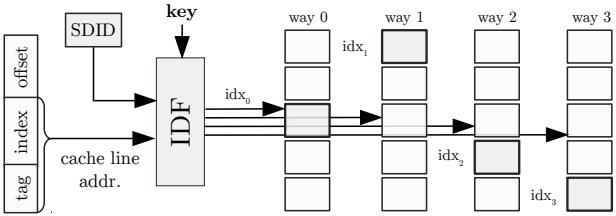
We want something that is closer to a traditional cache!

instead of this:

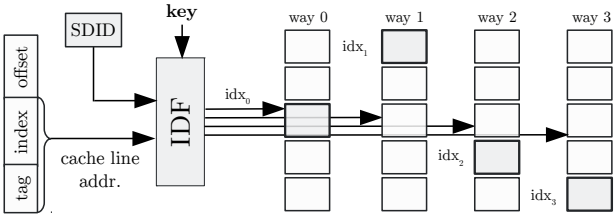



let's do this:





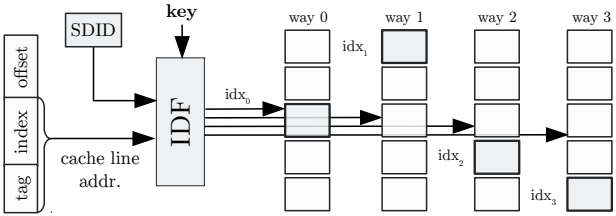

- Skewed cache [Sez93] (*i.e.*, traditional cache + more addressing logic) and an IDF
- Like cache slices




$2^{b_{indices} \cdot n_{ways}}$ possible cache sets

- Skewed cache [Sez93] (*i.e.*, traditional cache + more addressing logic) and an IDF
- Like cache slices

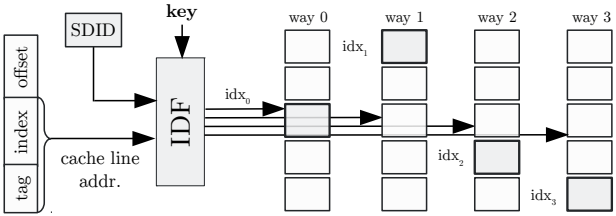
# ScatterCache - Concept




$2^{b_{indices} \cdot n_{ways}}$ possible cache sets

512 KiB (32 B lines), $n_{ways} = 8$, $b_{indices} = 11$
$\rightarrow 2^{88}$ sets

- Skewed cache [Sez93] (*i.e.*, traditional cache + more addressing logic) and an IDF
- Like cache slices

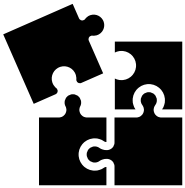


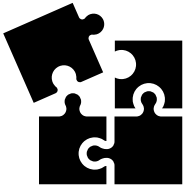

$2^{b_{indices} \cdot n_{ways}}$ possible cache sets

512 KiB (32 B lines), $n_{ways} = 8$, $b_{indices} = 11$
$\rightarrow 2^{88}$ sets

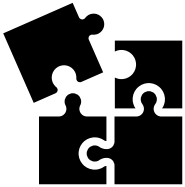
- Skewed cache [Sez93] (*i.e.*, traditional cache + more addressing logic) and an IDF
- Like cache slices
- Random replacement policy (for now)

- Inputs: cache line address, SDID, key
- Outputs: $n_{ways}$ indices ($b_{indices}$ bits)

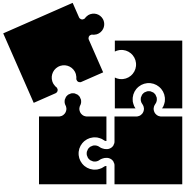




- Inputs: cache line address, SDID, key
- Outputs: $n_{ways}$ indices ($b_{indices}$ bits)
- Existing concepts and crypto primitives

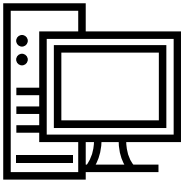


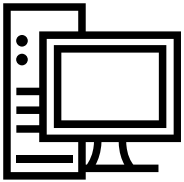
- Inputs: cache line address, SDID, key
- Outputs: $n_{ways}$ indices ($b_{indices}$ bits)
- Existing concepts and crypto primitives
- SCv1: hashing
  - Block ciphers (e.g., PRINCE [Bor+12])
  - Tweakable block ciphers (e.g., QARMA [Ava17])
  - Permutation-based primitives (e.g., Keccak-$p$ [Ber+11])



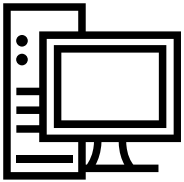
- Inputs: cache line address, SDID, key
- Outputs: $n_{ways}$ indices ($b_{indices}$ bits)
- Existing concepts and crypto primitives
- SCv1: hashing
  - Block ciphers (e.g., PRINCE [Bor+12])
  - Tweakable block ciphers (e.g., QARMA [Ava17])
  - Permutation-based primitives (e.g., Keccak-$p$ [Ber+11])
- SCv2: permutation
  - Prevents birthday-bound index collisions
  - No off-the-shelf primitives

- ScatterCache → last-level cache

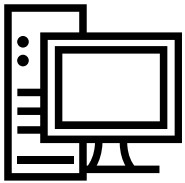




- ScatterCache → last-level cache
- Hardware managed key
  - Randomly generated at boot time
  - Rekeying with full cache flush



- ScatterCache → last-level cache
- Hardware managed key
  - Randomly generated at boot time
  - Rekeying with full cache flush
  - Potential for iterative rekeying
    → concurrent: CEASER-S @ISCA [Qur19]



- ScatterCache → last-level cache
- Hardware managed key
  - Randomly generated at boot time
  - Rekeying with full cache flush
  - Potential for iterative rekeying
    → concurrent: CEASER-S @ISCA [Qur19]
- SDID management via page table (indirection)
  - x86: Page Attribute Tables (PATs)
  - ARM: Memory Attribute Indirection Register (MAIRs)

- no OS support? $\rightarrow$ default SDID $= 0$
- OS support? $\rightarrow$ page-wise security domains

- no OS support? → default SDID = 0
- OS support? → page-wise security domains
  → shared read-only pages can be private in the cache!

- no OS support? → default SDID = 0
- OS support? → page-wise security domains
  → shared read-only pages can be private in the cache!
- OS defines domains
  (pages, processes, containers, VMs, . . . )

- no OS support? → default SDID = 0
- OS support? → page-wise security domains
  → shared read-only pages can be private in the cache!
- OS defines domains
  (pages, processes, containers, VMs, . . . )
- Software-based page *rekeying* by changing the SDID

- **Unshared memory** has no shared (physical) addresses
  - → No Flush+Reload, Evict+Reload, Flush+Flush
  - → Specialized Prime+Probe is possible

- **Unshared memory** has no shared (physical) addresses
  - → No Flush+Reload, Evict+Reload, Flush+Flush
  - → Specialized Prime+Probe is possible
- **Shared, read-only memory**
  - → Like unshared memory given OS support
  - → Otherwise, eviction-based attacks are hindered

# Applicable Cache Attacks

- **Unshared memory** has no shared (physical) addresses
  - → No Flush+Reload, Evict+Reload, Flush+Flush
  - → Specialized Prime+Probe is possible
- **Shared, read-only memory**
  - → Like unshared memory given OS support
  - → Otherwise, eviction-based attacks are hindered
- **Shared, writable memory** can't be separated
  - → Eviction-based attacks are hindered

- No end-to-end attack yet
  - → Simplified setting: perfect control, single access, no noise
  - → Investigate the building blocks in simulation and analytically

- No end-to-end attack yet
  - → Simplified setting: perfect control, single access, no noise
  - → Investigate the building blocks in simulation and analytically
- Finding congruent addresses ($n_{ways} = 8, b_{indices} = 11$)
  - Full collisions are unlikely → use partial collisions
  - Approach in the paper: $\approx 2^{25}$ profiled victim accesses

- No end-to-end attack yet
  - → Simplified setting: perfect control, single access, no noise
  - → Investigate the building blocks in simulation and analytically
- Finding congruent addresses ($n_{ways} = 8, b_{indices} = 11$)
  - Full collisions are unlikely → use partial collisions
  - Approach in the paper: $\approx 2^{25}$ profiled victim accesses
- Evicting one set with 99 % needs 275 addresses

- No end-to-end attack yet
  - → Simplified setting: perfect control, single access, no noise
  - → Investigate the building blocks in simulation and analytically
- Finding congruent addresses ($n_{ways} = 8$, $b_{indices} = 11$)
  - Full collisions are unlikely → use partial collisions
  - Approach in the paper: $\approx 2^{25}$ profiled victim accesses
- Evicting one set with 99 % needs 275 addresses
- Two Prime+Probe variants ($n_{ways} = 8$, $b_{indices} = 12$)
  - 99 % confidence: 35 to 152 victim accesses (repetitions)
  - Between 9870 and 1216 congruent addresses

- No end-to-end attack yet
  - → Simplified setting: perfect control, single access, no noise
  - → Investigate the building blocks in simulation and analytically
- Finding congruent addresses ($n_{ways} = 8$, $b_{indices} = 11$)
  - Full collisions are unlikely → use partial collisions
  - Approach in the paper: $\approx 2^{25}$ profiled victim accesses
- Evicting one set with 99 % needs 275 addresses
- Two Prime+Probe variants ($n_{ways} = 8$, $b_{indices} = 12$)
  - 99 % confidence: 35 to 152 victim accesses (repetitions)
  - Between 9870 and 1216 congruent addresses
- Noise? Leakage analysis?

- No end-to-end attack yet
  - → Simplified setting: perfect control, single access, no noise
  - → Investigate the building blocks in simulation and analytically
- Finding congruent addresses ($n_{ways} = 8$, $b_{indices} = 11$)
  - Full collisions are unlikely → use partial collisions
  - Approach in the paper: $\approx 2^{25}$ profiled victim accesses
  - Generalized by Purnal and Verbauwhede [PV19]: $\approx 2^{10}$
- Evicting one set with 99 % needs 275 addresses
- Two Prime+Probe variants ($n_{ways} = 8$, $b_{indices} = 12$)
  - 99 % confidence: 35 to 152 victim accesses (repetitions)
  - Between 9870 and 1216 congruent addresses
- Noise? Leakage analysis?

- Micro benchmarks using the gem5 full system simulator (ARM)
  - Poky Linux from Yocto 2.5 (kernel version 4.14.67)
  - GAP, MiBench, lmbench, scimark2
- SPEC CPU 2017 on custom cache simulator

- Micro benchmarks using the gem5 full system simulator (ARM)
  - Poky Linux from Yocto 2.5 (kernel version 4.14.67)
  - GAP, MiBench, lmbench, scimark2
- SPEC CPU 2017 on custom cache simulator
- Cache hit rate $\geq$ set-associative caches (random replacement)
- $2\,\% - 4\,\%$ below LRU on micro benchmarks, $0\,\% - 2\,\%$ for SPEC

all these cache attacks over the past decades...


IT'S OVER
IT'S FINALLY OVER
quickmeme.com

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

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

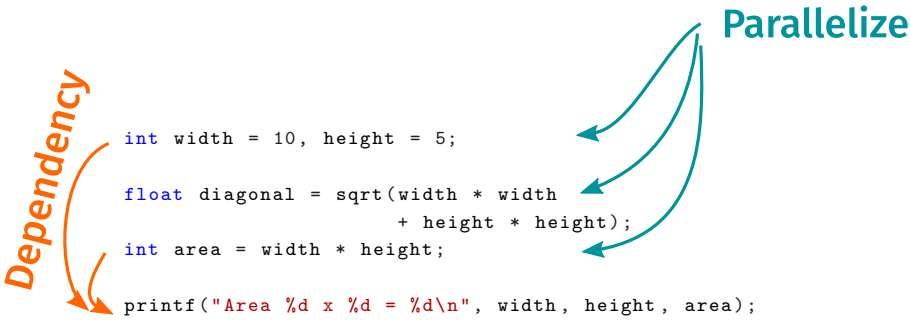
Dependency

Parallelize

```
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);

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

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

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

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

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

- Kernel addresses are not accessible

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

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

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

- Adapted code

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

# Let's see whether this works...

- Adapted code

```
*(volatile char*)0;
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

# Let's hope this does not work...

- Maybe there is no permission check in transient instructions...

# Let's hope this does not work...

- Maybe there is no permission check in transient instructions...
- ...or it is only done when commiting them

# Let's hope this does not work...

- Maybe there is no permission check in transient instructions...
- ...or it is only done when commiting them
- Indirection through microarchitectural traces:

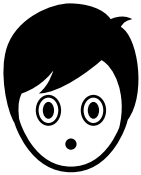
```
char data = *(char*)0xffffffff81a000e0;
array[data * 4096] = 0;
```

- Maybe there is no permission check in transient instructions...
- ...or it is only done when commiting them
- Indirection through microarchitectural traces:

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

- Check whether any part of `array` is cached

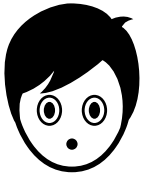
# It does work...



- Flush+Reload over all pages of the array


- Index of cache hit reveals data

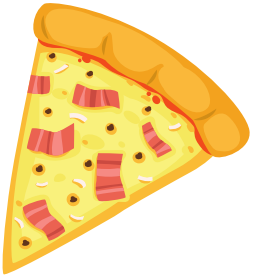
# It does work...

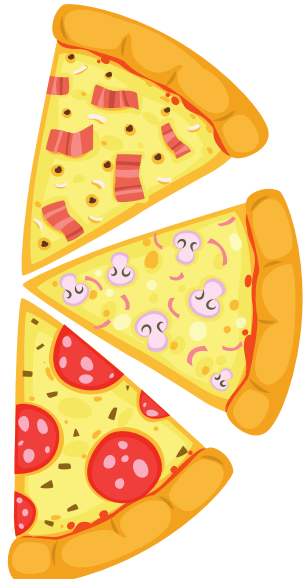


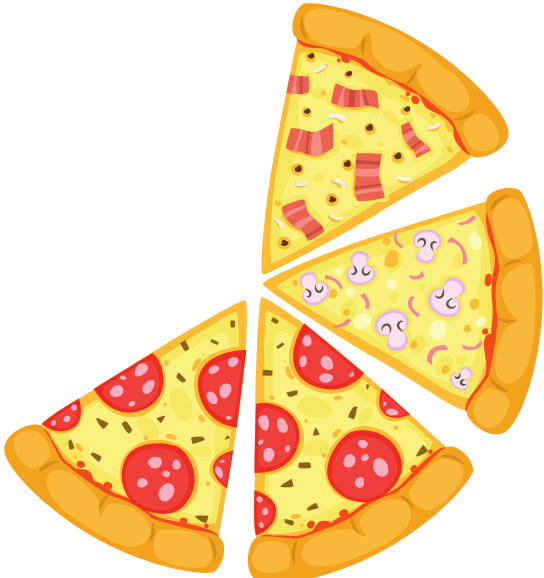
- Flush+Reload over all pages of the array

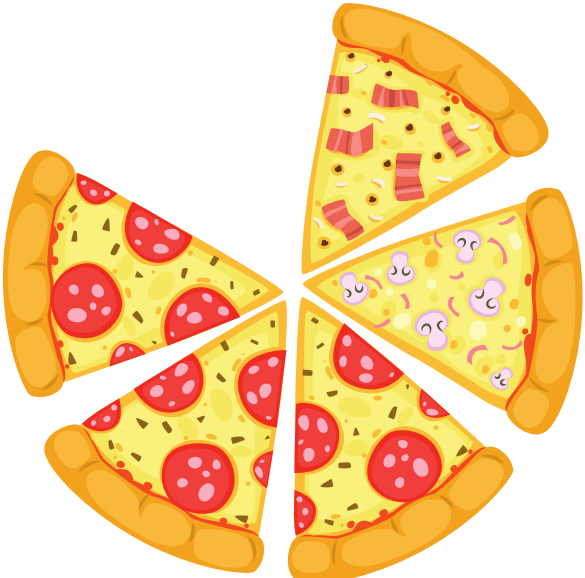

- Index of cache hit reveals data
- Permission check is in some cases not fast enough

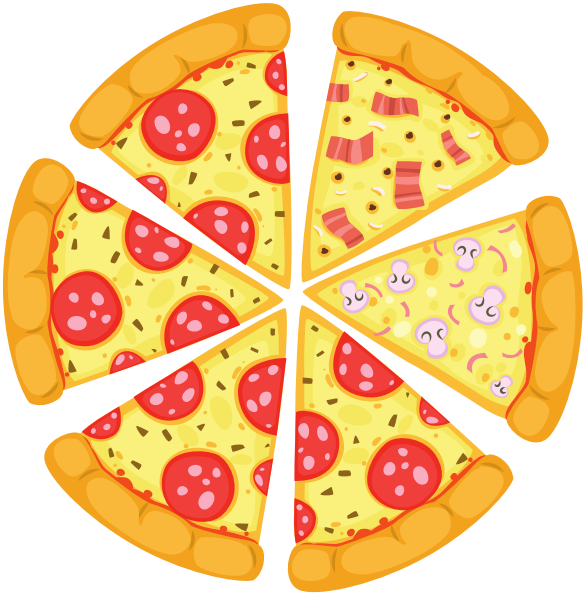

PIZZA
SPECIAL RECIPES












»A table for 6 please«

# Speculative Cooking


»A table for 6 please«


PIZZA
SPECIAL RECIPES


PIZZA
SPECIAL RECIPES

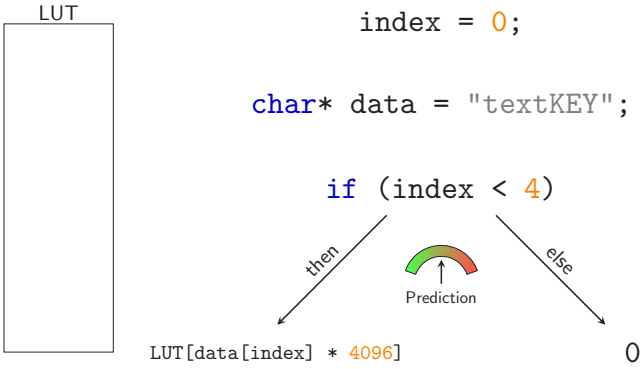

Pizza

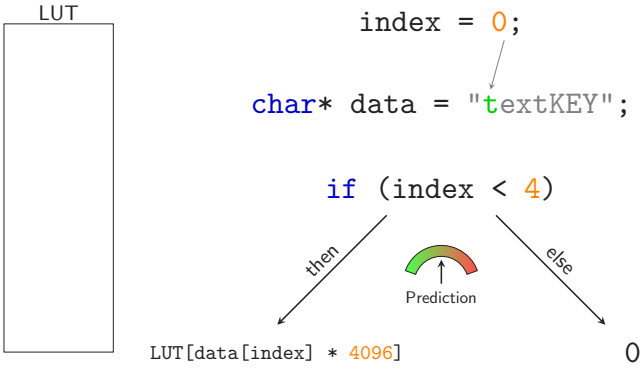

PIZZA
SPECIAL RECIPES

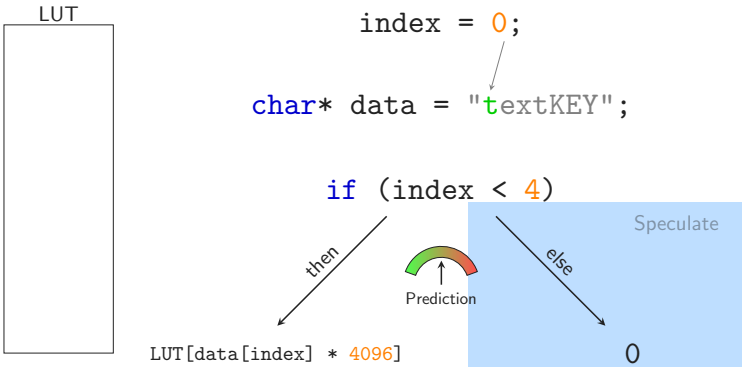

Pizza

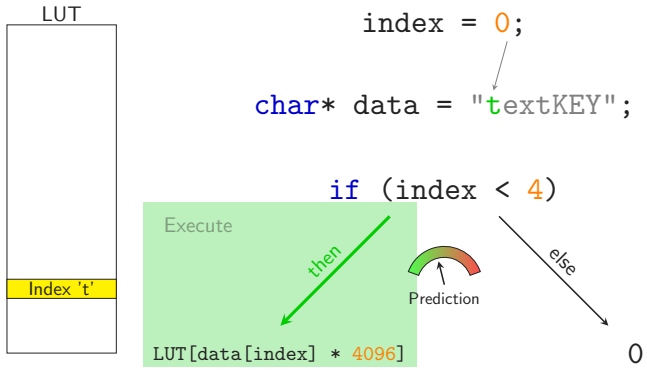

PIZZA
SPECIAL RECIPES

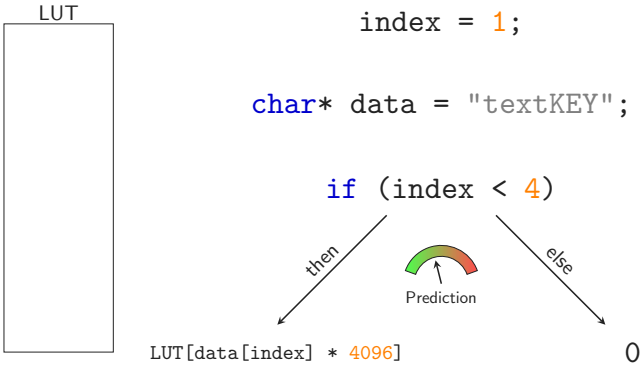

Pizza

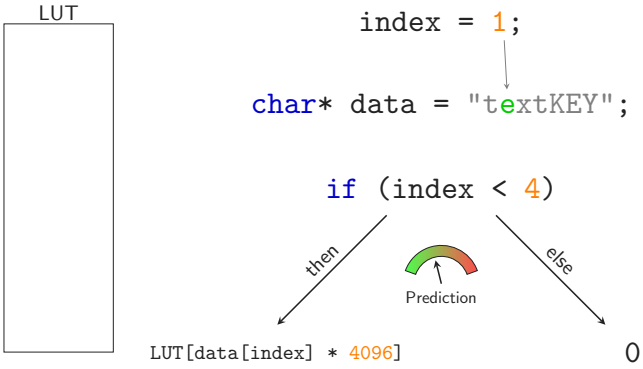

LUT
index = 0;
char* data = "textKEY";
if (index < 4)
then
else
Prediction
LUT[data[index] * 4096]
0

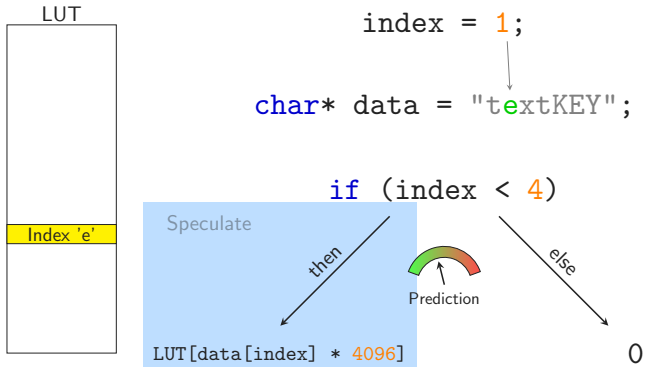

LUT
index = 0;
char* data = "textKEY";
if (index < 4)
then
else
Prediction
LUT[data[index] * 4096]
0

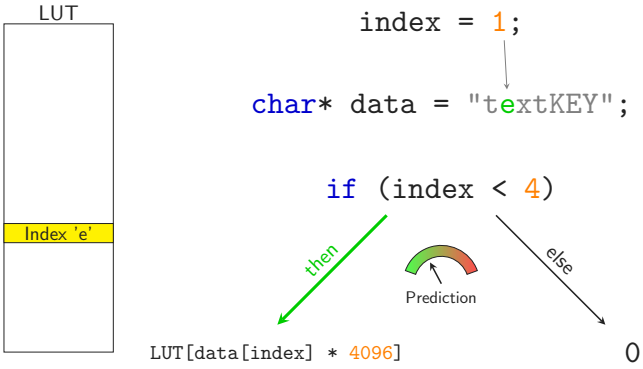

LUT
index = 0;
char* data = "textKEY";
if (index < 4)
Speculate
then
else
Prediction
LUT[data[index] * 4096]
0

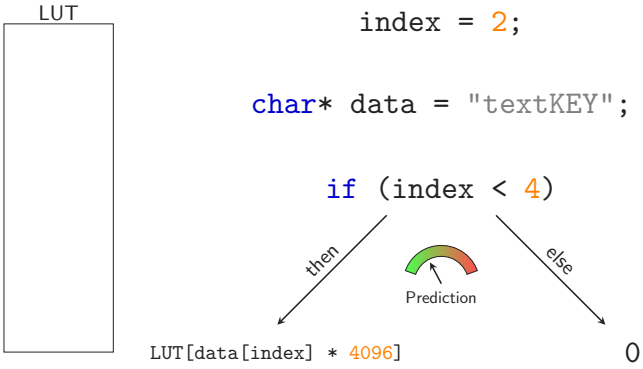

LUT
Index 't'
index = 0;
char* data = "textKEY";
if (index < 4)
Execute
then
else
Prediction
LUT[data[index] * 4096]
0

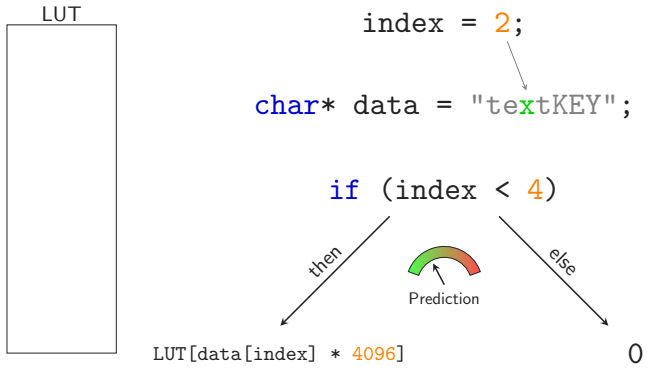

LUT
index = 1;
char* data = "textKEY";
if (index < 4)
then
Prediction
else
LUT[data[index] * 4096]
0

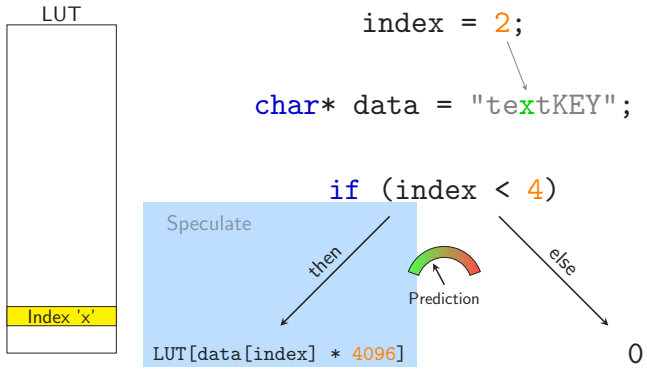

LUT
index = 1;
char* data = "textKEY";
if (index < 4)
then
Prediction
else
LUT[data[index] * 4096]
0

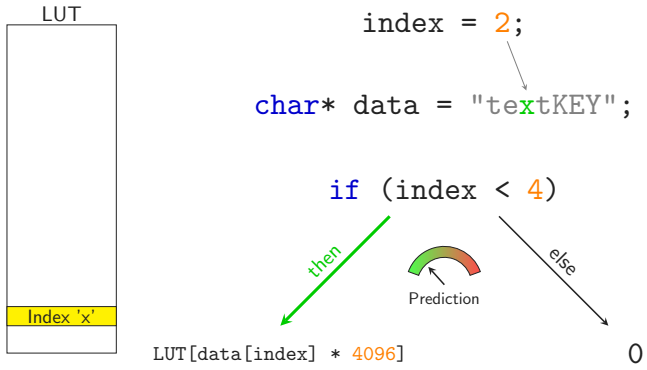

LUT
Index 'e'
index = 1;
char* data = "textKEY";
if (index < 4)
Speculate
then
Prediction
else
LUT[data[index] * 4096]
0

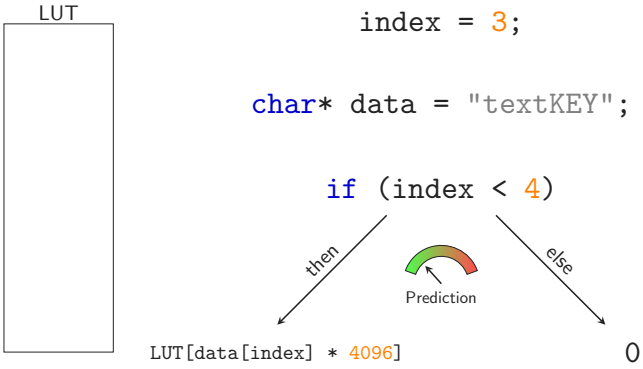

LUT
Index 'e'
index = 1;
char* data = "textKEY";
if (index < 4)
then
Prediction
else
LUT[data[index] * 4096]
0

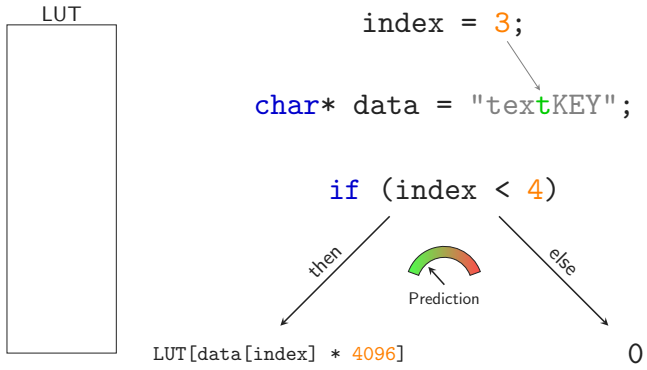

LUT
index = 2;
char* data = "textKEY";
if (index < 4)
then
else
Prediction
LUT[data[index] * 4096]
0

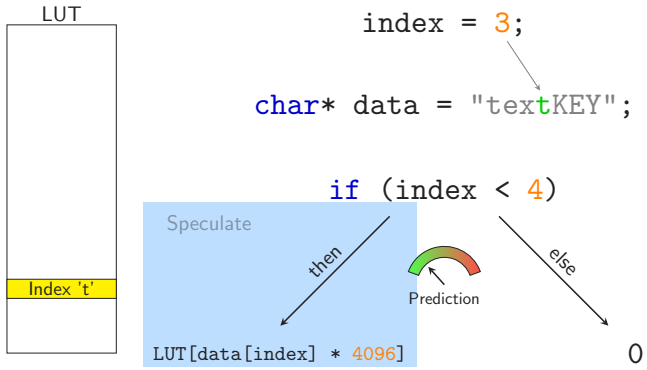

LUT
index = 2;
char* data = "textKEY";
if (index < 4)
then
Prediction
else
LUT[data[index] * 4096]
0

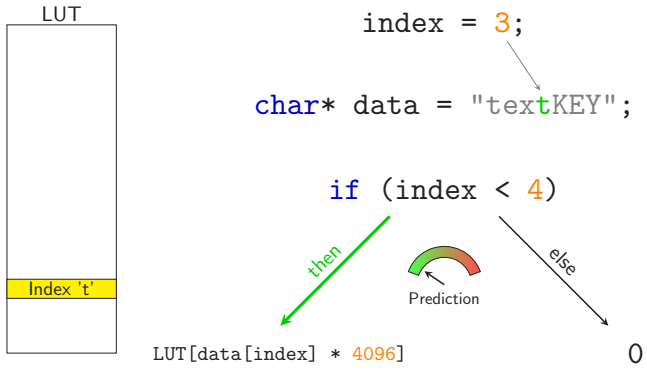

LUT
Index 'x'
index = 2;
char* data = "textKEY";
if (index < 4)
Speculate
then
else
Prediction
LUT[data[index] * 4096]
0

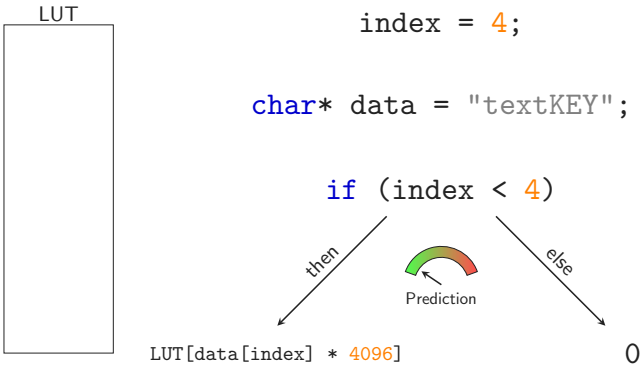

LUT
Index 'x'
index = 2;
char* data = "textKEY";
if (index < 4)
then
Prediction
else
LUT[data[index] * 4096]
0

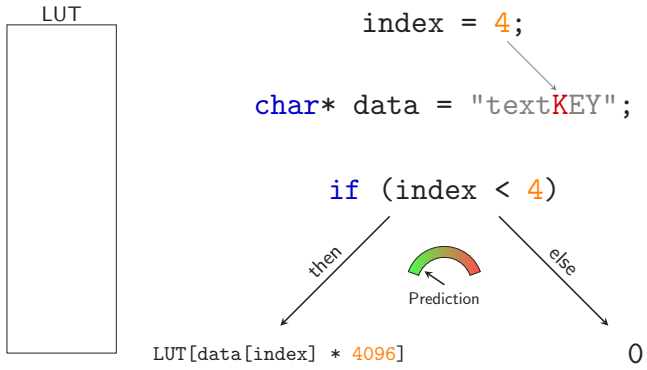

LUT
index = 3;
char* data = "textKEY";
if (index < 4)
then
else
Prediction
LUT[data[index] * 4096]
0

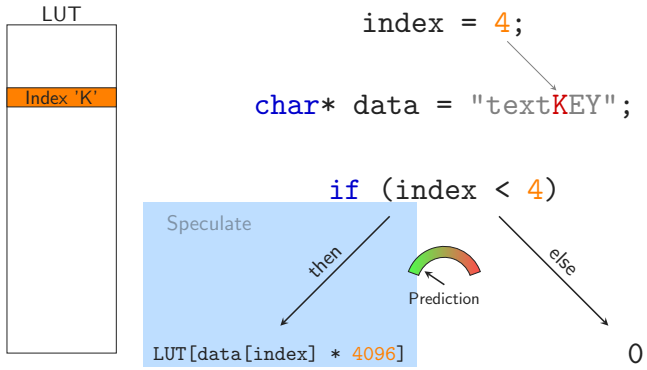

LUT
index = 3;
char* data = "textKEY";
if (index < 4)
then
Prediction
else
LUT[data[index] * 4096]
0

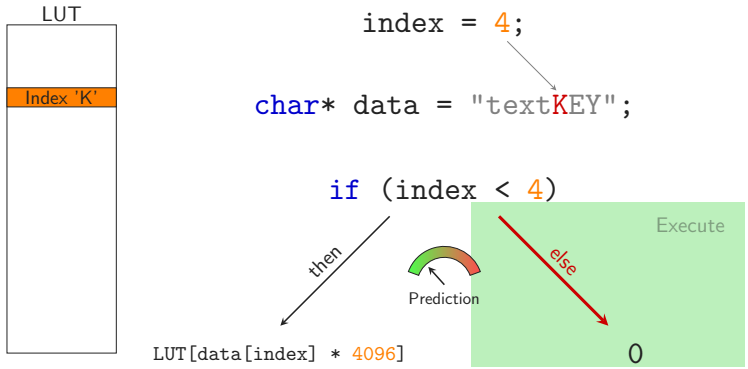

LUT
Index 't'
index = 3;
char* data = "textKEY";
if (index < 4)
Speculate
then
Prediction
else
LUT[data[index] * 4096]
0

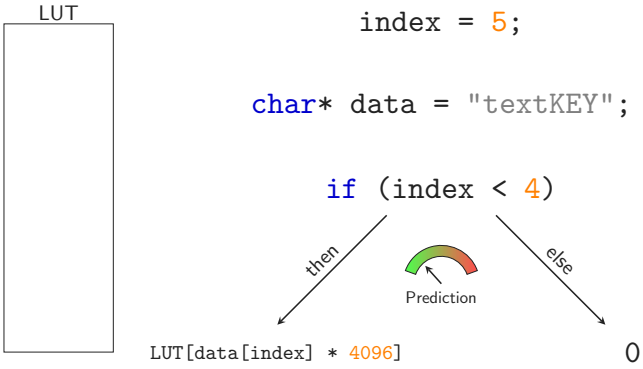

LUT
Index 't'
index = 3;
char* data = "textKEY";
if (index < 4)
then
Prediction
else
LUT[data[index] * 4096]
0

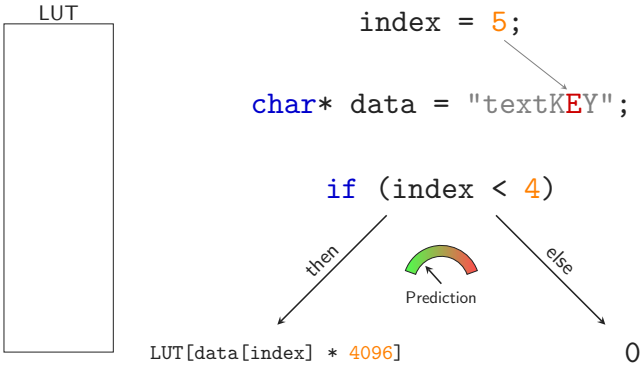

LUT
index = 4;
char* data = "textKEY";
if (index < 4)
then
Prediction
else
LUT[data[index] * 4096]
0

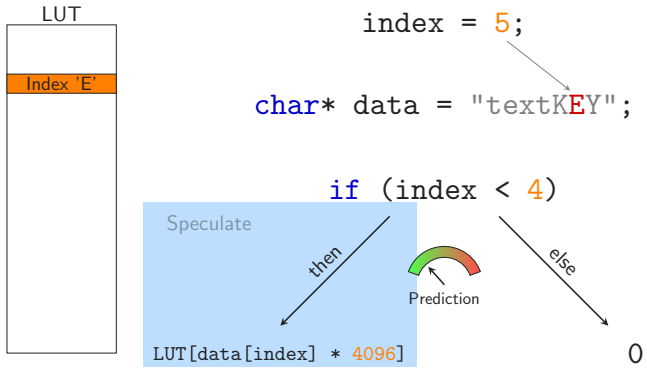

LUT
index = 4;
char* data = "textKEY";
if (index < 4)
then
else
Prediction
LUT[data[index] * 4096]
0

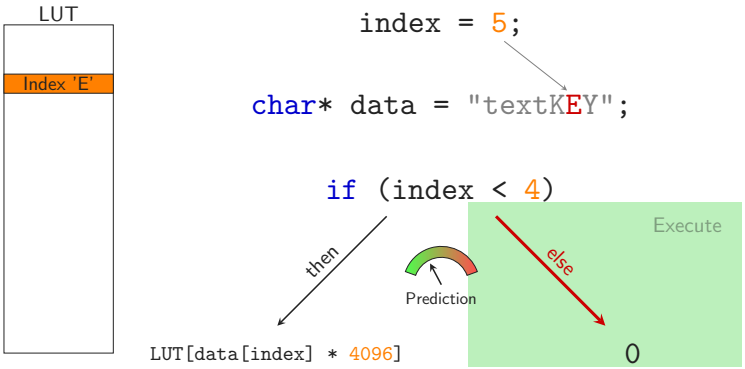

LUT
Index 'K'
index = 4;
char* data = "textKEY";
if (index < 4)
Speculate
then
else
Prediction
LUT[data[index] * 4096]
0

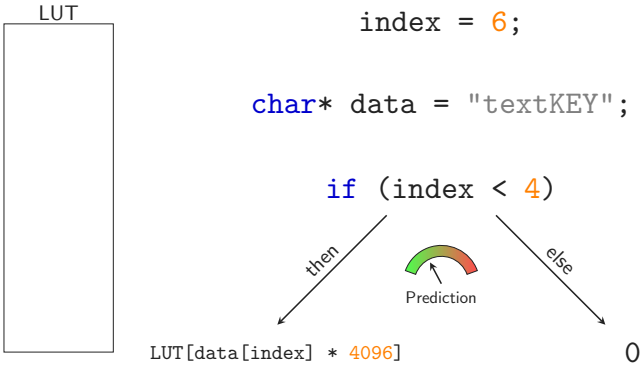

LUT
Index 'K'
index = 4;
char* data = "textKEY";
if (index < 4)
then
Prediction
else
Execute
LUT[data[index] * 4096]
0

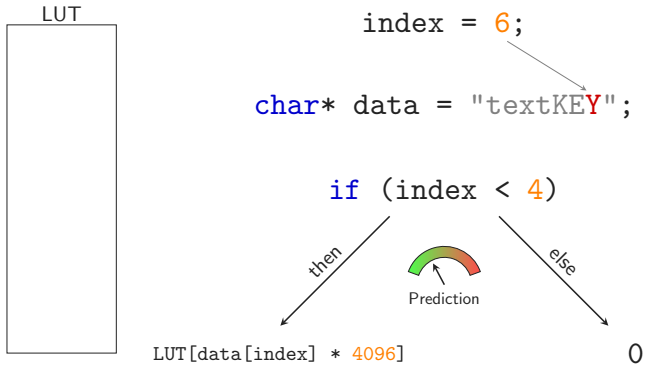

LUT
index = 5;
char* data = "textKEY";
if (index < 4)
then
Prediction
else
LUT[data[index] * 4096]
0

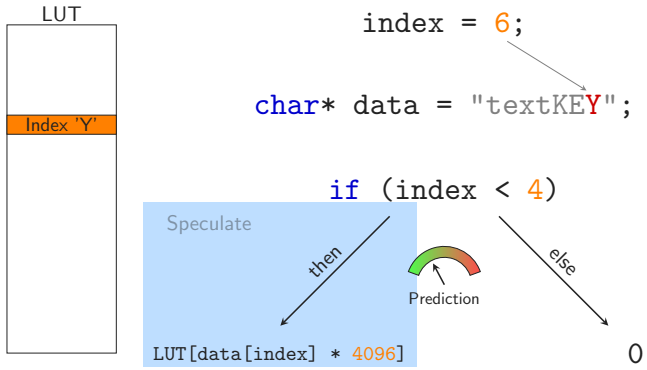

LUT
index = 5;
char* data = "textKEY";
if (index < 4)
then
else
Prediction
LUT[data[index] * 4096]
0

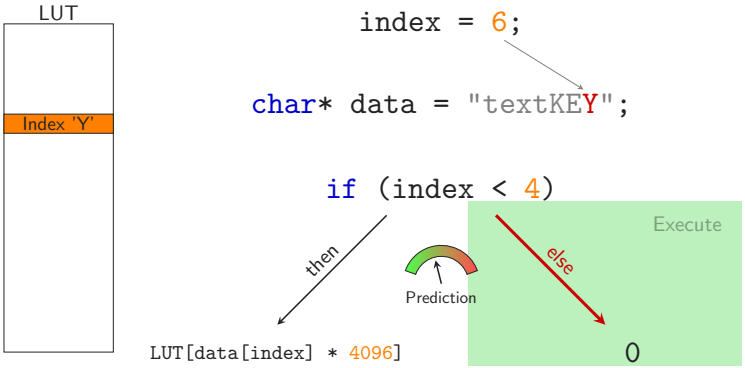

LUT
Index 'E'
index = 5;
char* data = "textKEY";
if (index < 4)
Speculate
then
Prediction
else
LUT[data[index] * 4096]
0

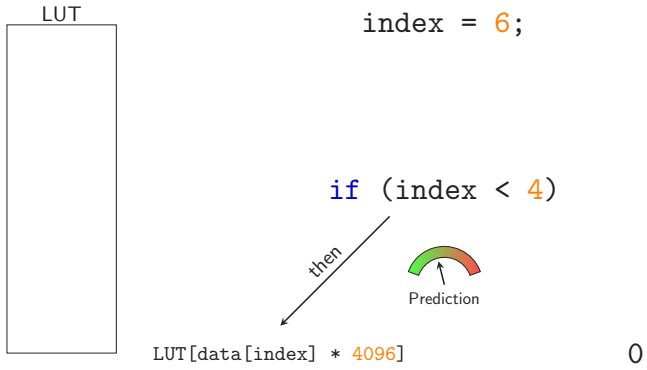

LUT
Index 'E'
index = 5;
char* data = "textKEY";
if (index < 4)
then
Prediction
else
Execute
LUT[data[index] * 4096]
0


LUT
index = 6;
char* data = "textKEY";
if (index < 4)
then
Prediction
else
LUT[data[index] * 4096]
0


LUT
index = 6;
char* data = "textKEY";
if (index < 4)
then
else
Prediction
LUT[data[index] * 4096]
0

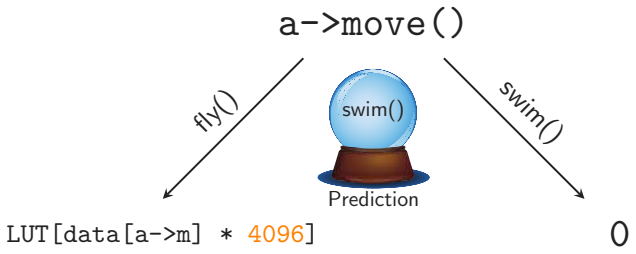

LUT
Index 'Y'
index = 6;
char* data = "textKEY";
if (index < 4)
Speculate
then
else
Prediction
LUT[data[index] * 4096]
0


LUT
Index 'Y'
index = 6;
char* data = "textKEY";
if (index < 4)
then
Prediction
else
Execute
LUT[data[index] * 4096]
0

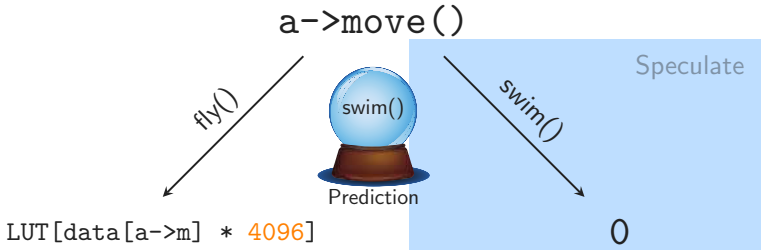



Spectre-STL (v4): Ignore sanitizing write access and use unsanitized old value instead

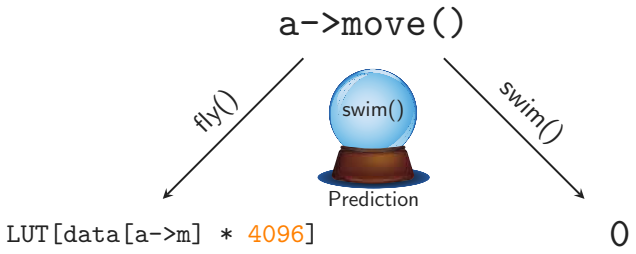
```
Animal* a = bird;
```



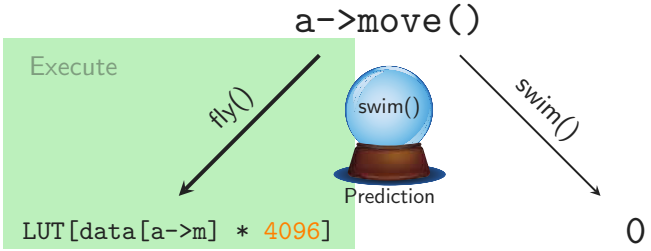
```
Animal* a = bird;
```



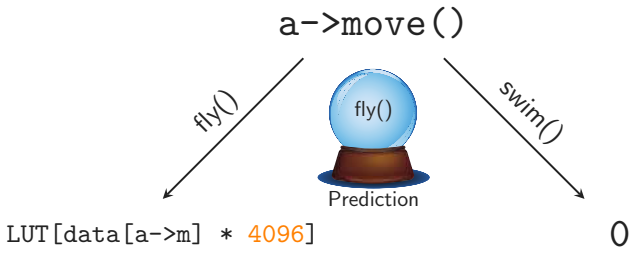
```
Animal* a = bird;
```



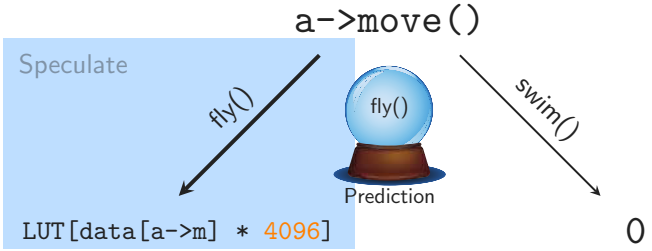
```
Animal* a = bird;
```



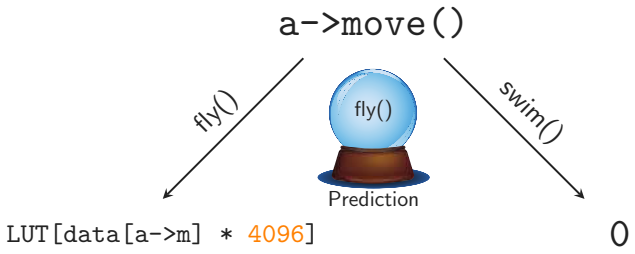
```
Animal* a = bird;
```



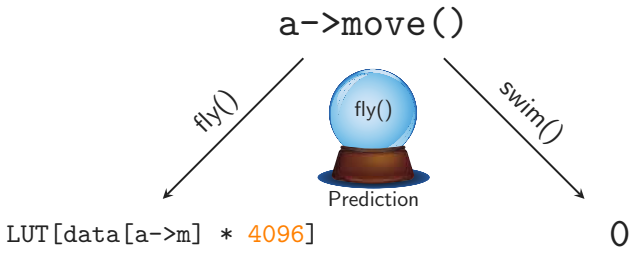
```
Animal* a = bird;
```



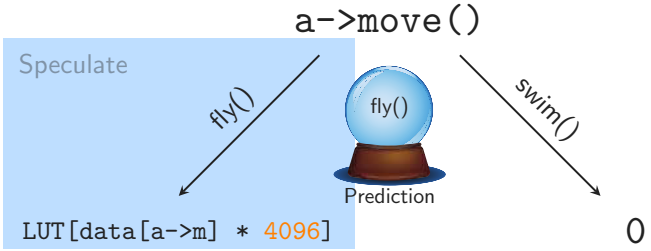
```
Animal* a = bird;
```



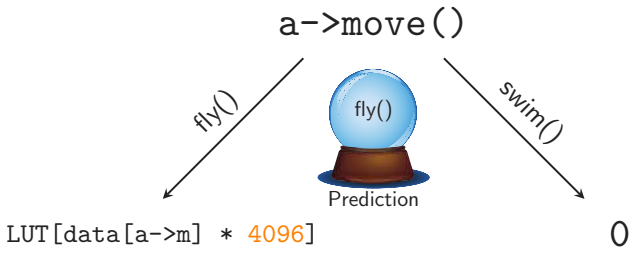
```
Animal* a = fish;
```



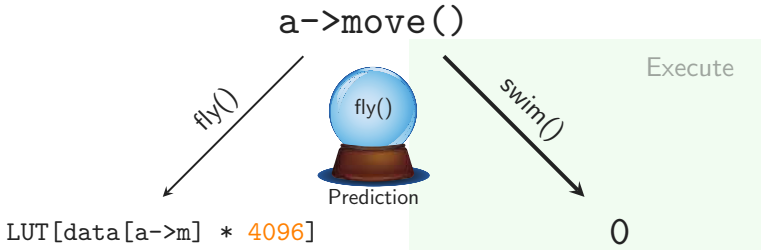
```
Animal* a = fish;
```



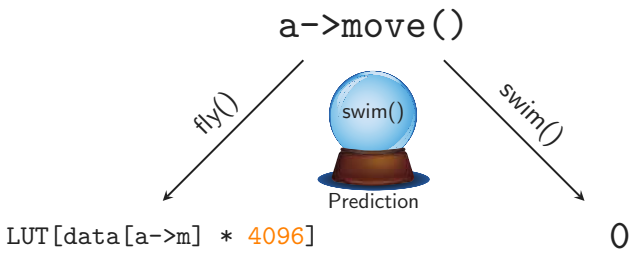
```
Animal* a = fish;
```



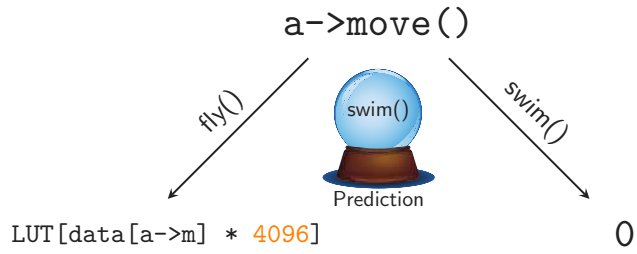
```
Animal* a = fish;
```



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

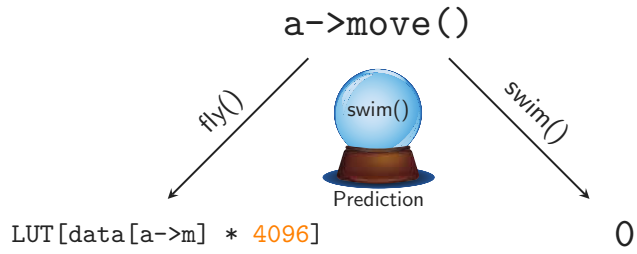


```
Animal* a = fish;
```




Spectre-BTB (v2): mistrain BTB → mispredict indirect jump/call

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




Spectre-BTB (v2): mistrain BTB → mispredict indirect jump/call

Spectre-RSB (v5): mistrain RSB → mispredict return

- v1.1: Speculatively write to memory locations

[5] V. Kiriansky and C. Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. In: arXiv:1807.03757 (2018).

- v1.1: Speculatively write to memory locations

→ Many more gadgets than previously anticipated n

[5] V. Kiriansky and C. Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. In: arXiv:1807.03757 (2018).

# “Speculative Buffer Overflows”[5]

- v1.1: Speculatively write to memory locations

→ Many more gadgets than previously anticipated n

- v1.2: Ignore writable bit

[5] V. Kiriansky and C. Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. In: arXiv:1807.03757 (2018).

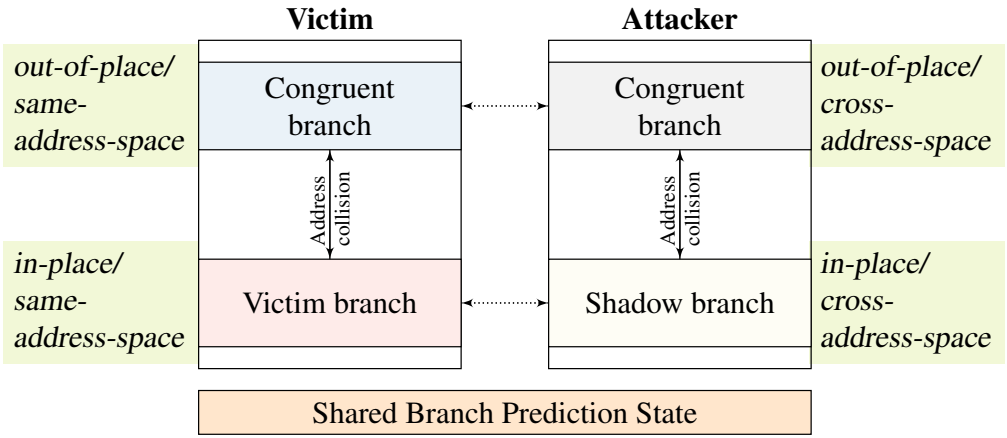
- v1.1: Speculatively write to memory locations

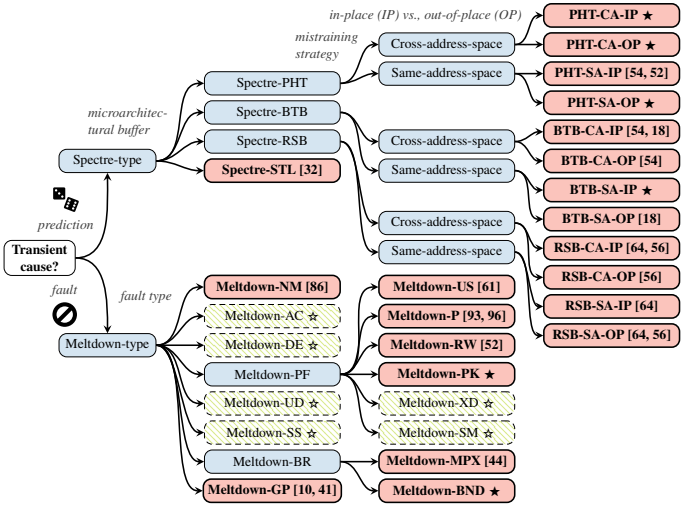
→ Many more gadgets than previously anticipated n

- v1.2: Ignore writable bit

→ = Meltdown-RW

[5] V. Kiriansky and C. Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. In: arXiv:1807.03757 (2018).


Victim
Attacker
out-of-place/ same-address-space
Congruent branch
Congruent branch
out-of-place/ cross-address-space
Address collision
Address collision
in-place/ same-address-space
Victim branch
Shadow branch
in-place/ cross-address-space
Shared Branch Prediction State


Transient cause?
prediction
fault
Spectre-type
microarchitec-tural buffer
Spectre-PHT
Spectre-BTB
Spectre-RSB
Spectre-STL [32]
mistraining strategy
in-place (IP) vs., out-of-place (OP)
Cross-address-space
Same-address-space
PHT-CA-IP ★
PHT-CA-OP ★
PHT-SA-IP [54, 52]
PHT-SA-OP ★
BTB-CA-IP [54, 18]
BTB-CA-OP [54]
BTB-SA-IP ★
BTB-SA-OP [18]
RSB-CA-IP [64, 56]
RSB-CA-OP [56]
RSB-SA-IP [64]
RSB-SA-OP [64, 56]
fault type
Meltdown-type
Meltdown-NM [86]
Meltdown-AC ☆
Meltdown-DE ☆
Meltdown-PF
Meltdown-UD ☆
Meltdown-SS ☆
Meltdown-BR
Meltdown-GP [10, 41]
Meltdown-US [61]
Meltdown-P [93, 96]
Meltdown-RW [52]
Meltdown-PK ★
Meltdown-XD ☆
Meltdown-SM ☆
Meltdown-MPX [44]
Meltdown-BND ★

# Mitigations?


BLOCKCHAIN
HD
HISTORY.COM

# Computer Architecture Today

Informing the broad computing community about current activities, advances and future directions in computer architecture.

## Let's Keep it to Ourselves: Don't Disclose Vulnerabilities

by Gus Uht on Jan 31, 2019 | Tags: Opinion, Security

| Attack | | InvisiSpec | SafeSpec | DAWG | RSB Stuffing | Retpoline | Poison Value | Index Masking | Site Isolation | SLH | YSNB | IBRS | STIPB | IBPB | Serialization | Taint Tracking | Timer Reduction | Sloth | SSBD / SSBB |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Intel | Spectre-PHT | □ | □ | □ | ◇ | ◇ | ● | ◐ | ◐ | ● | ○ | ◇ | ◇ | ◇ | ◐ | ■ | ◐ | ◧ | ◇ |
| | Spectre-BTB | □ | □ | □ | ◇ | ● | ◇ | ◇ | ◐ | ◇ | ◇ | ● | ◐ | ◐ | ◇ | ■ | ◐ | ◇ | ◇ |
| | Spectre-RSB | □ | □ | □ | ◐ | ◇ | ◇ | ◇ | ◐ | ◇ | ◇ | ◇ | ◇ | ◇ | ◇ | ■ | ◐ | ◇ | ◇ |
| | Spectre-STL | □ | □ | □ | ◇ | ◇ | ◇ | ◇ | ◐ | ◇ | ◇ | ◇ | ◇ | ◇ | ◇ | ■ | ◐ | ■ | ● |
| ARM | Spectre-PHT | □ | □ | □ | ◇ | ◇ | ● | ◐ | ◐ | ● | ○ | ◇ | ◇ | ◇ | ◐ | ■ | ◐ | ◧ | ◇ |
| | Spectre-BTB | □ | □ | □ | ◇ | ● | ◇ | ◇ | ◐ | ◇ | ◇ | ◇ | ◇ | ◇ | ◇ | ■ | ◐ | ◇ | ◇ |
| | Spectre-RSB | □ | □ | □ | ◐ | ◇ | ◇ | ◇ | ◐ | ◇ | ◇ | ◇ | ◇ | ◇ | ◇ | ■ | ◐ | ◇ | ◇ |
| | Spectre-STL | □ | □ | □ | ◇ | ◇ | ◇ | ◇ | ◐ | ◇ | ◇ | ◇ | ◇ | ◇ | ◇ | ■ | ◐ | ■ | ● |
| AMD | Spectre-PHT | □ | □ | □ | ◇ | ◇ | ● | ◐ | ◐ | ● | ○ | ◇ | ◇ | ◇ | ◐ | ■ | ◐ | ◧ | ◇ |
| | Spectre-BTB | □ | □ | □ | ◇ | ● | ◇ | ◇ | ◐ | ◇ | ◇ | ■ | ◧ | ◧ | ◇ | ■ | ◐ | ◇ | ◇ |
| | Spectre-RSB | □ | □ | □ | ◐ | ◇ | ◇ | ◇ | ◐ | ◇ | ◇ | ◇ | ◇ | ◧ | ◇ | ■ | ◐ | ◇ | ◇ |
| | Spectre-STL | □ | □ | □ | ◇ | ◇ | ◇ | ◇ | ◐ | ◇ | ◇ | ◇ | ◇ | ◇ | ◇ | ■ | ◐ | ■ | ● |

Symbols show if an attack is mitigated (●), partially mitigated (◐), not mitigated (○), theoretically mitigated (■), theoretically impeded (◧), not theoretically impeded (□), or out of scope (◇).

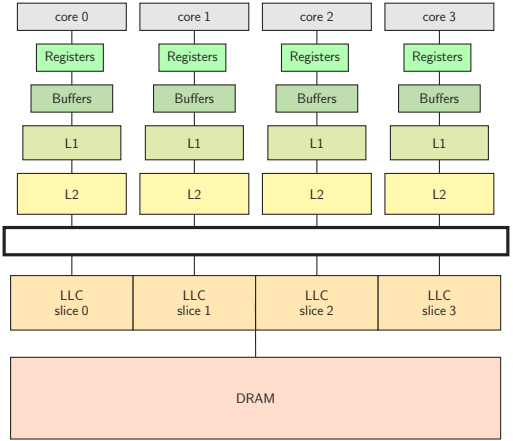
**Table 2:** Reported performance impacts of countermeasures

| Defense \ Impact | Performance Loss | Benchmark |
|---|---|---|
| InvisiSpec | 22% | SPEC |
| SafeSpec | 3% (improvement) | SPEC2017 on MARSSx86 |
| DAWG | 2–12%, 1–15% | PARSEC, GAPBS |
| RSB Stuffing | no reports | |
| Retpoline | 5–10% | real-world workload servers |
| Site Isolation | only memory overhead | |
| SLH | 36.4%, 29% | Google microbenchmark suite |
| YSNB | 60% | Phoenix |
| IBRS | 20–30% | two sysbench 1.0.11 benchmarks |
| STIPB | 30– 50% | Rodinia OpenMP, DaCapo |
| IBPB | no individual reports | |
| Serialization | 62%, 74.8% | Google microbenchmark suite |
| SSBD/SSBB | 2–8% | SYSmark®2014 SE & SPEC integer |
| KAISER/KPTI | 0–2.6% | system call rates |
| L1TF mitigations | -3–31% | various SPEC |

# ConTExT

# Where can data be?





# ConTExT Idea

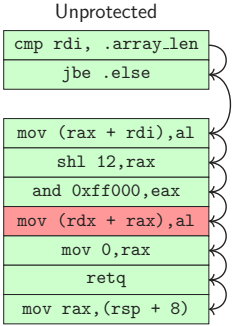
- Mark secrets in source code

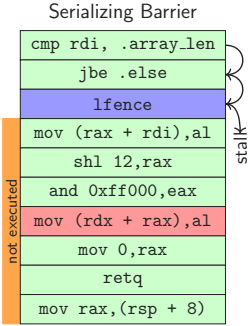
- Mark secrets in source code
- Propagate taint through memory hierarchy:

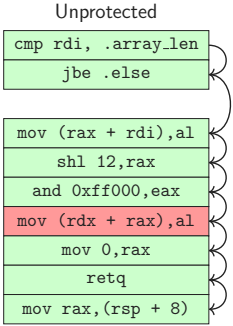
- Mark secrets in source code
- Propagate taint through memory hierarchy:
  - Pages

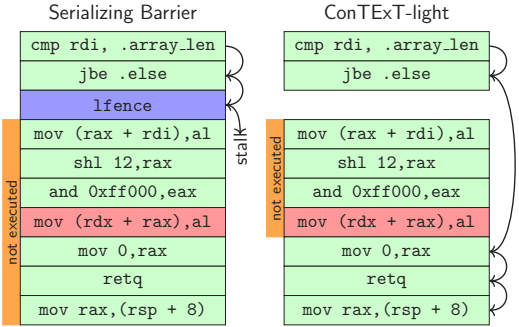
- Mark secrets in source code
- Propagate taint through memory hierarchy:
  - Pages
  - Cache Lines (in caches and buffers)

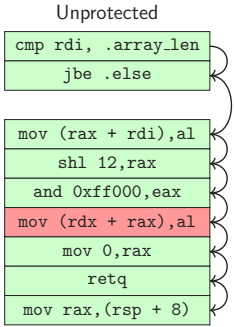
- Mark secrets in source code
- Propagate taint through memory hierarchy:
  - Pages
  - Cache Lines (in caches and buffers)
  - Registers


Unprotected
cmp rdi, .array_len
jbe .else
mov (rax + rdi),al
shl 12,rax
and 0xff000,eax
mov (rdx + rax),al
mov 0,rax
retq
mov rax,(rsp + 8)

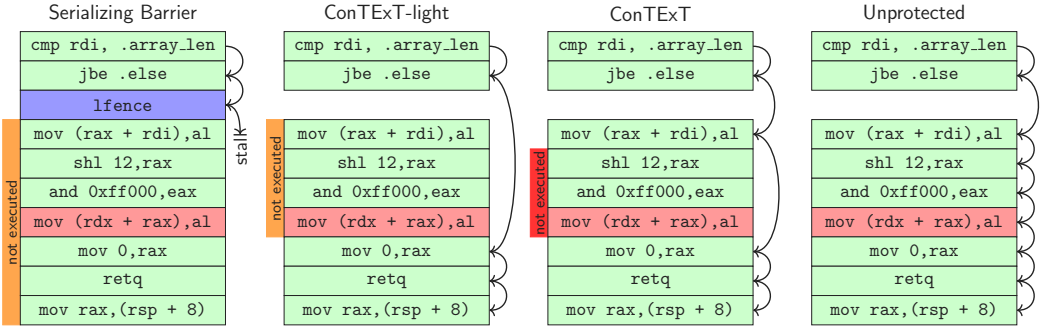

Serializing Barrier
cmp rdi, .array_len
jbe .else
lfence
mov (rax + rdi),al
shl 12,rax
and 0xff000,eax
mov (rdx + rax),al
mov 0,rax
retq
mov rax,(rsp + 8)
stalk
not executed


Unprotected
cmp rdi, .array_len
jbe .else
mov (rax + rdi),al
shl 12,rax
and 0xff000,eax
mov (rdx + rax),al
mov 0,rax
retq
mov rax,(rsp + 8)


Serializing Barrier
cmp rdi, .array_len
jbe .else
lfence
mov (rax + rdi),al
shl 12,rax
and 0xff000,eax
mov (rdx + rax),al
mov 0,rax
retq
mov rax,(rsp + 8)
not executed
stalk
ConTExT-light
cmp rdi, .array_len
jbe .else
mov (rax + rdi),al
shl 12,rax
and 0xff000,eax
mov (rdx + rax),al
mov 0,rax
retq
mov rax,(rsp + 8)
not executed

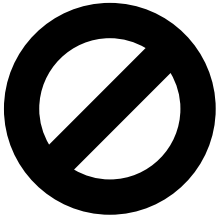

Unprotected
cmp rdi, .array_len
jbe .else
mov (rax + rdi),al
shl 12,rax
and 0xff000,eax
mov (rdx + rax),al
mov 0,rax
retq
mov rax,(rsp + 8)

# ConTExT Comparison

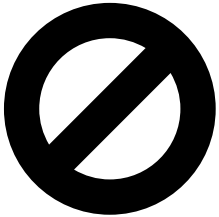


# No generic taint tracking

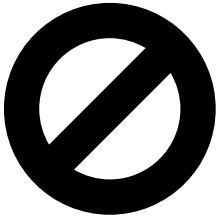
# No generic taint tracking



# No generic taint tracking

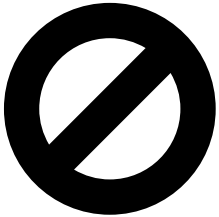


# No generic taint tracking



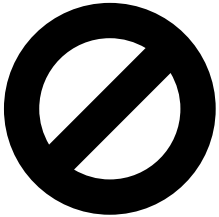
- Taint tracking is complicated and expensive

# No generic taint tracking



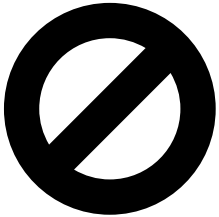
- Taint tracking is complicated and expensive

# No generic taint tracking



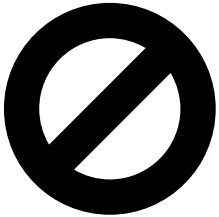
- Taint tracking is complicated and expensive → we don't do that
- Software unintentionally losing taint?

# No generic taint tracking



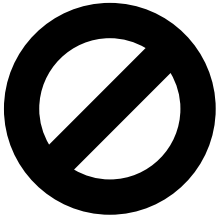
- Taint tracking is complicated and expensive → we don't do that
- Software unintentionally losing taint?

# No generic taint tracking



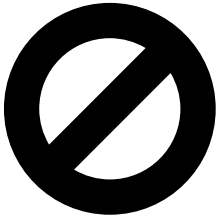
- Taint tracking is complicated and expensive → we don't do that
- Software unintentionally losing taint? → software problem
- Variable copies?

# No generic taint tracking



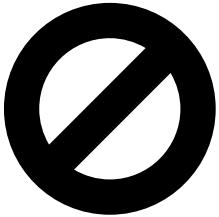
- Taint tracking is complicated and expensive → we don't do that
- Software unintentionally losing taint? → software problem
- Variable copies?

# No generic taint tracking



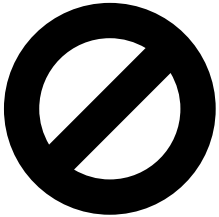
- Taint tracking is complicated and expensive → we don't do that
- Software unintentionally losing taint? → software problem
- Variable copies? → compiler should avoid / warn
- Otherwise: always load/store from the same location

# No generic taint tracking



- Taint tracking is complicated and expensive → we don't do that
- Software unintentionally losing taint? → software problem
- Variable copies? → compiler should avoid / warn
- Otherwise: always load/store from the same location

# No generic taint tracking



- Taint tracking is complicated and expensive → we don't do that
- Software unintentionally losing taint? → software problem
- Variable copies? → compiler should avoid / warn
- Otherwise: always load/store from the same location (compiler does that anyway)

# Avoiding Taint Explosion

# Avoiding Taint Explosion

- Fully overwriting a tainted register $\rightarrow$ untaint

- Fully overwriting a tainted register $\rightarrow$ untaint
- Writing to unprotected memory exposes value to attackers

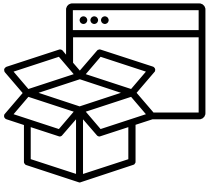
- Fully overwriting a tainted register → untaint
- Writing to unprotected memory exposes value to attackers
  → Untaint register

- Fully overwriting a tainted register → untaint
- Writing to unprotected memory exposes value to attackers
  → Untaint register
- Split stack into protected and unprotected half

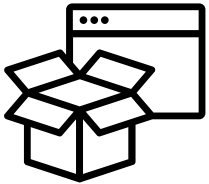
- Fully overwriting a tainted register → untaint
- Writing to unprotected memory exposes value to attackers
  → Untaint register
- Split stack into protected and unprotected half
- Stack spills of unprotected data → stay unprotected as long as they stay in the cache

# What about function arguments?

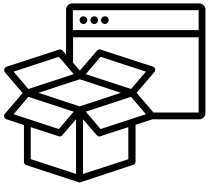
# What about function arguments?



# What about function arguments?

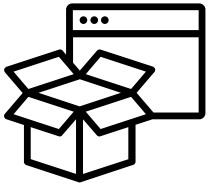


# What about function arguments?



- Parameter declared secret? → non-transient (over-tainting)

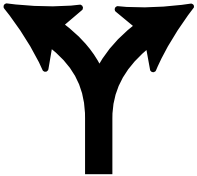
# What about function arguments?



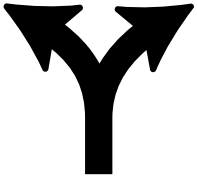
- Parameter declared secret? → non-transient (over-tainting)
- Copy to non-secret local variable if necessary (e.g., based on arguments)

# What about branching on secrets?

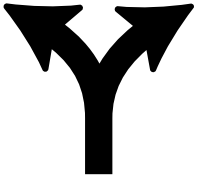
# What about branching on secrets?



# What about branching on secrets?

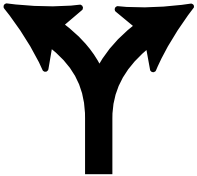


# What about branching on secrets?



- Then you already have a side channel

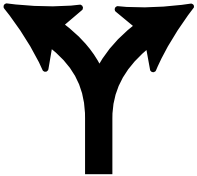
# What about branching on secrets?



- Then you already have a side channel

$\rightarrow$ No need for a more complicated transient-execution attack

# What about branching on secrets?



- Then you already have a side channel

→ No need for a more complicated transient-execution attack

→ Compiler issues a warning

- different classes of attacks need different classes of defenses

- different classes of attacks need different classes of defenses
- interesting opportunities for new defensive research

- different classes of attacks need different classes of defenses
- interesting opportunities for new defensive research
- we need more cat-and-mouse game in system security (like in crypto algorithm design)

# Hardware-Software Co-Design against Microarchitectural Attacks

**Daniel Gruss**

September 4, 2019

Graz University of Technology