# Side-Channel Security

Chapter 4: Transient-Execution Attacks - Meltdown and Spectre

**Lukas Giner**

March 21, 2024

Graz University of Technology

# Vulnerability Assessment

- Meltdown[4] and Spectre [2] are two CPU vulnerabilities

## Vulnerability Assessment

- Meltdown[4] and Spectre [2] are two CPU vulnerabilities
- Discovered in 2017 by 4 independent teams

## Vulnerability Assessment

- Meltdown[4] and Spectre [2] are two CPU vulnerabilities
- Discovered in 2017 by 4 independent teams
- Due to an embargo, released at the beginning of 2018

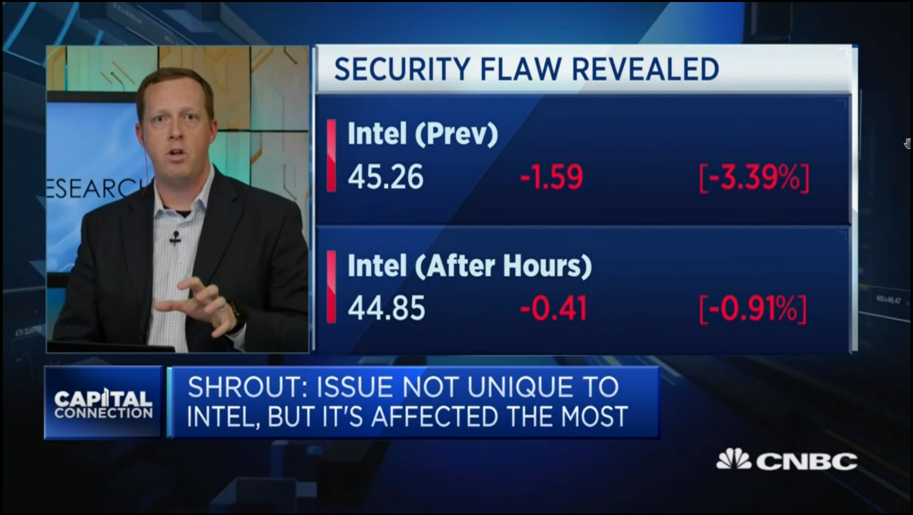
## Vulnerability Assessment

- Meltdown[4] and Spectre [2] are two CPU vulnerabilities
- Discovered in 2017 by 4 independent teams
- Due to an embargo, released at the beginning of 2018
- News coverage followed by a lot of panic

FOX BUSINESS WASHINGTON, D.C.
WASHINGTON, D.C.
WINTER STORM
NEWS ALERT
INTEL REVEALS DESIGN FLAW THAT COULD ALLOW HACKERS TO ACCESS DATA
FOX BUSINESS NETWORK
@FOXBUSINESS

FIREEYE CYBER THREAT MAP
DEVELOPING STORY
COMPUTER CHIP FLAWS IMPACT BILLIONS OF DEVICES
LIVE
CNN
DAX ▲164.69
NEWS STREAM

GLOBAL
COMPUTER CHIP SCARE
The bugs are known as 'Spectre' and 'Meltdown'
BBC WORLD NEWS
£:HKS 10.58
EURO:£ 0.891



SECURITY FLAW REVEALED

| | | |
|---|---|---|
| Intel (Prev) | | |
| 45.26 | -1.59 | [-3.39%] |
| Intel (After Hours) | | |
| 44.85 | -0.41 | [-0.91%] |

SHROUT: ISSUE NOT UNIQUE TO INTEL, BUT IT'S AFFECTED THE MOST

## Vulnerability Assesment

A lot of confusion fueled the panic

- Which CPUs/vendors are affected?

## Vulnerability Assesment

A lot of confusion fueled the panic

- Which CPUs/vendors are affected?
- Are smartphones/IoT devices affected?

## Vulnerability Assesment

A lot of confusion fueled the panic

- Which CPUs/vendors are affected?
- Are smartphones/IoT devices affected?
- Can the vulnerabilities be exploited remotely?

## Vulnerability Assesment

A lot of confusion fueled the panic

- Which CPUs/vendors are affected?
- Are smartphones/IoT devices affected?
- Can the vulnerabilities be exploited remotely?
- What data is at risk?

## Vulnerability Assesment

A lot of confusion fueled the panic

- Which CPUs/vendors are affected?
- Are smartphones/IoT devices affected?
- Can the vulnerabilities be exploited remotely?
- What data is at risk?
- How hard is it to exploit the vulnerabilities?

## Vulnerability Assesment

A lot of confusion fueled the panic

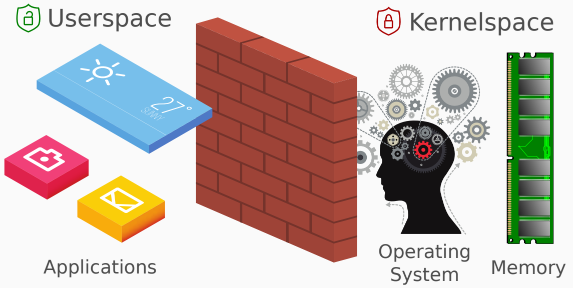
- Which CPUs/vendors are affected?
- Are smartphones/IoT devices affected?
- Can the vulnerabilities be exploited remotely?
- What data is at risk?
- How hard is it to exploit the vulnerabilities?
- Is it already exploited?

## Let’s try to clarify these questions

MELTDOWN

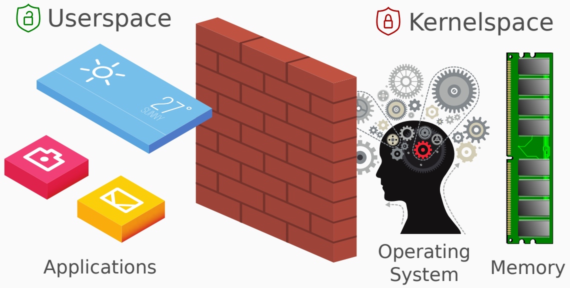
# Hardware Isolation

- Kernel is isolated from user space



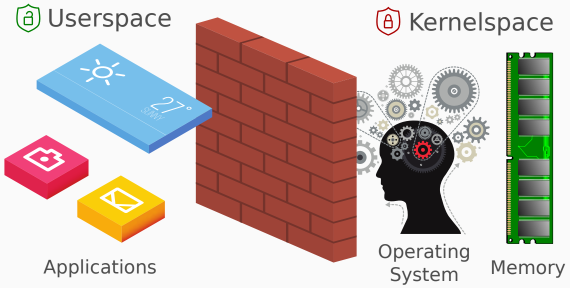
## Hardware Isolation

- Kernel is isolated from user space
- This isolation is a combination of hardware and software



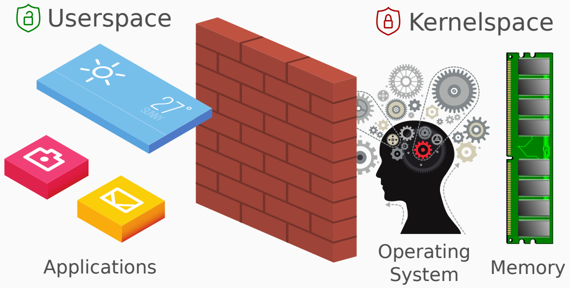
## Hardware Isolation

- Kernel is isolated from user space
- This isolation is a combination of hardware and software
- User applications cannot access anything from the kernel



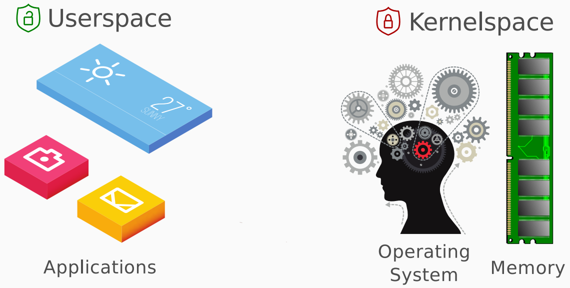
## Hardware Isolation

- Kernel is isolated from user space
- This isolation is a combination of hardware and software
- User applications cannot access anything from the kernel
- There is only a well-defined interface → syscalls



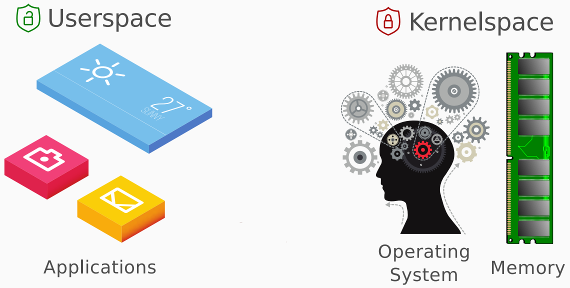
## Meltdown Briefing

- Breaks isolation between applications and kernel



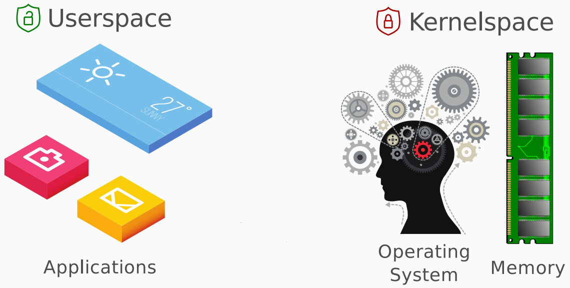
## Meltdown Briefing

- Breaks isolation between applications and kernel
- User applications can access kernel addresses



## Meltdown Briefing

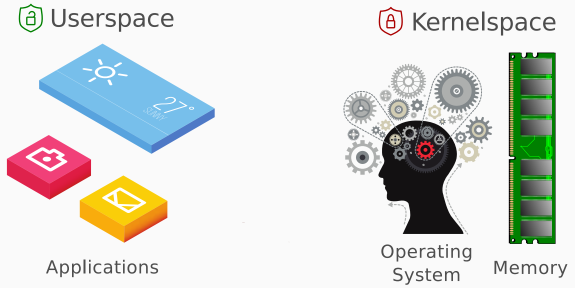
- Breaks isolation between applications and kernel
- User applications can access kernel addresses
- Entire physical memory is mapped in the kernel



## Meltdown Briefing

- Breaks isolation between applications and kernel
- User applications can access kernel addresses
- Entire physical memory is mapped in the kernel

→ Meltdown can read whole DRAM



# Meltdown Requirements

- Only on Intel CPUs and some ARMs (e.g. Cortex A15,A57,A72,A75)

## Meltdown Requirements

- Only on Intel CPUs and some ARMs (e.g. Cortex A15,A57,A72,A75)
- AMD and other ARMs seem to be unaffected

## Meltdown Requirements

- Only on Intel CPUs and some ARMs (e.g. Cortex A15,A57,A72,A75)
- AMD and other ARMs seem to be unaffected
- Common cause: permission check done in parallel to load instruction

## Meltdown Requirements

- Only on Intel CPUs and some ARMs (e.g. Cortex A15,A57,A72,A75)
- AMD and other ARMs seem to be unaffected
- Common cause: permission check done in parallel to load instruction
- Race condition between permission check and dependent operation(s)

## Meltdown Variant Requirements

- Meltdown variant: read privileged registers

## Meltdown Variant Requirements

- Meltdown variant: read privileged registers
- Limited to some registers, no memory content

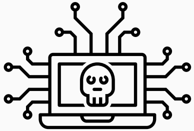
## Meltdown Variant Requirements

- Meltdown variant: read privileged registers
- Limited to some registers, no memory content
- Reported by ARM

## Meltdown Variant Requirements

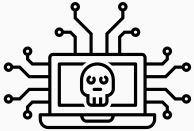
- Meltdown variant: read privileged registers
- Limited to some registers, no memory content
- Reported by ARM
- Affects some ARMs (Cortex A15, A57, and A72)

## Meltdown Exploitability



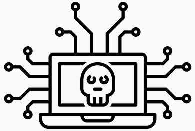
- Meltdown requires code execution on the device (e.g. Apps)

## Meltdown Exploitability



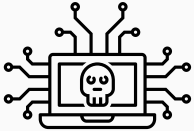
- Meltdown requires code execution on the device (e.g. Apps)
- Untrusted code can read entire memory of device

## Meltdown Exploitability



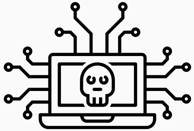
- Meltdown requires code execution on the device (e.g. Apps)
- Untrusted code can read entire memory of device
- Cannot be triggered remotely

## Meltdown Exploitability



- Meltdown requires code execution on the device (e.g. Apps)
- Untrusted code can read entire memory of device
- Cannot be triggered remotely
- Proof-of-concept code available online

## Meltdown Exploitability



- Meltdown requires code execution on the device (e.g. Apps)
- Untrusted code can read entire memory of device
- Cannot be triggered remotely
- Proof-of-concept code available online
- No info about environment required $\rightarrow$ easy to reproduce

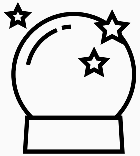
SPECTRE

## Spectre Briefing

- Mistrains branch prediction

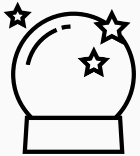
## Spectre Briefing

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



## Spectre Briefing

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



## Spectre Briefing

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

→ Spectre “convinces” program to execute code

## Spectre Requirements

- On Intel and AMD CPUs

## Spectre Requirements

- On Intel and AMD CPUs
- Some ARMs (Cortex R and Cortex A) are also affected

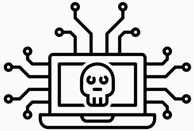
## Spectre Requirements

- On Intel and AMD CPUs
- Some ARMs (Cortex R and Cortex A) are also affected
- Common cause: speculative execution of branches

## Spectre Requirements

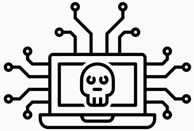
- On Intel and AMD CPUs
- Some ARMs (Cortex R and Cortex A) are also affected
- Common cause: speculative execution of branches
- Speculative execution leaves microarchitectural traces which leak secret

## Spectre Exploitability



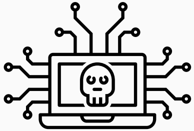
- Spectre (typically) requires code execution on the device (e.g. Apps)

## Spectre Exploitability



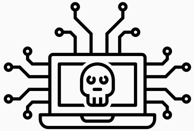
- Spectre (typically) requires code execution on the device (e.g. Apps)
- Untrusted code can convince trusted code to reveal secrets

## Spectre Exploitability



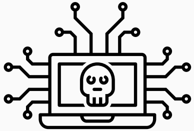
- Spectre (typically) requires code execution on the device (e.g. Apps)
- Untrusted code can convince trusted code to reveal secrets
- Can be triggered remotely (e.g. in the browser, NetSpectre)

## Spectre Exploitability



- Spectre (typically) requires code execution on the device (e.g. Apps)
- Untrusted code can convince trusted code to reveal secrets
- Can be triggered remotely (e.g. in the browser, NetSpectre)
- Proof-of-concept code available online

## Spectre Exploitability



- Spectre (typically) requires code execution on the device (e.g. Apps)
- Untrusted code can convince trusted code to reveal secrets
- Can be triggered remotely (e.g. in the browser, NetSpectre)
- Proof-of-concept code available online
- Info about environment required $\rightarrow$ hard to reproduce

## Background

## Out-of-order Execution

5. Cook everything until vegetables are soft

6. Add spices to taste and stir for 10 minutes

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*

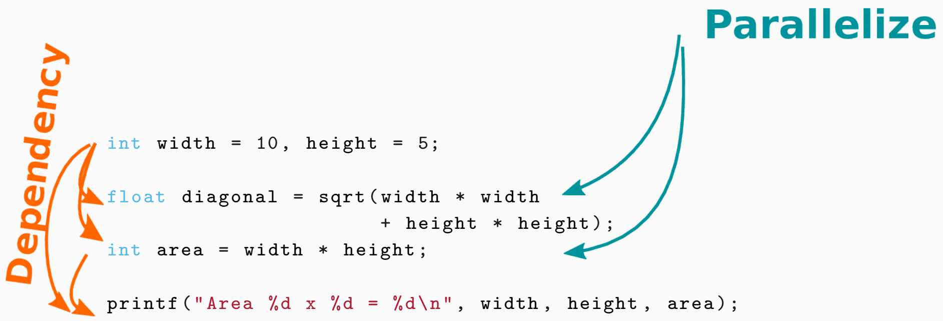
Parallelize

Dependency

## Out-of-order Execution

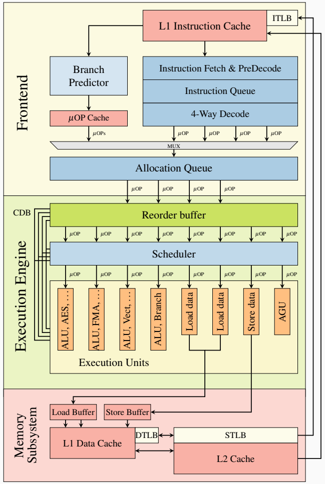
```c
int width = 10, height = 5;

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

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

# Out-of-order Execution



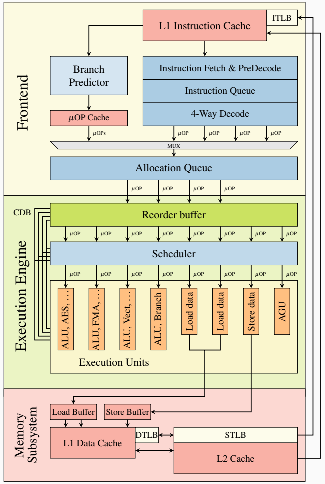
```c
int width = 10, height = 5;

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

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

# Out-of-order execution



- Instructions are fetched and decoded in the front-end
- Instructions are dispatched to the backend
- Instructions are processed by individual execution units

# Out-of-order execution



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

## We are ready for the gory details of Meltdown

## Building the Code

- Find something human readable, e.g., the Linux version

```
# sudo grep linux_banner /proc/kallsyms
ffffffff81a000e0 R linux_banner
```

## Building the Code

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

## Building the Code

- Compile and run

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

## Building the Code

- Compile and run

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

- Kernel addresses are of course not accessible

## Building the Code

- Compile and run

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

- Kernel addresses are of course not accessible
- Any invalid access throws an exception → segmentation fault

## Building the Code

- Just catch the segmentation fault!

## Building the Code

- Just catch the segmentation fault!
- We can simply install a signal handler

## Building the Code

- Just catch the segmentation fault!
- We can simply install a signal handler
- And if an exception occurs, just jump back and continue

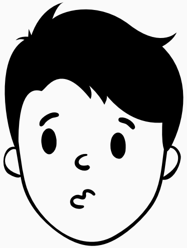
## Building the Code

- Just catch the segmentation fault!
- We can simply install a signal handler
- And if an exception occurs, just jump back and continue
- Then we can read the value

## Building the Code

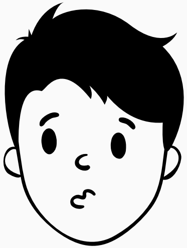
- Just catch the segmentation fault!
- We can simply install a signal handler
- And if an exception occurs, just jump back and continue
- Then we can read the value
- Sounds like a good idea

## Building the Code



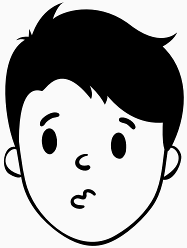
- Still no kernel memory

## Building the Code



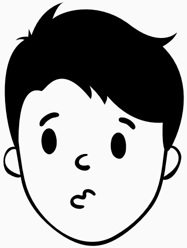
- Still no kernel memory
- Maybe it is not that straight forward

## Building the Code



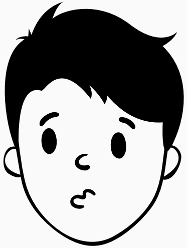
- Still no kernel memory
- Maybe it is not that straight forward
- Privilege checks seem to work

## Building the Code



- Still no kernel memory
- Maybe it is not that straight forward
- Privilege checks seem to work
- Are privilege checks also done when executing instructions out of order?

## Building the Code

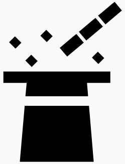


- Still no kernel memory
- Maybe it is not that straight forward
- Privilege checks seem to work
- Are privilege checks also done when executing instructions out of order?
- Problem: out-of-order instructions are not visible

## Building the Code

- Adapted code

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



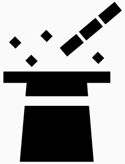
## Building the Code

- Adapted code

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

- volatile because compiler was not happy

```
warning: statement with no effect [-Wunused-value]
           *(char*) 0;
```



# Building the Code

- Adapted code

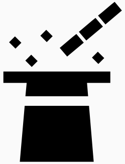
```
*(volatile char*) 0;
array[0] = 0;
```

- `volatile` because compiler was not happy

```
warning: statement with no effect [-Wunused-value]
          *(char*) 0;
```

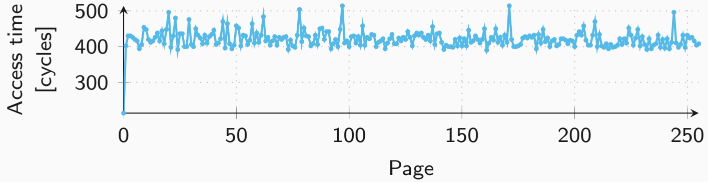
- Static code analyzer is still not happy

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



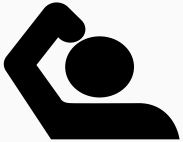
## Building Meltdown

- Flush+Reload over all pages of the array

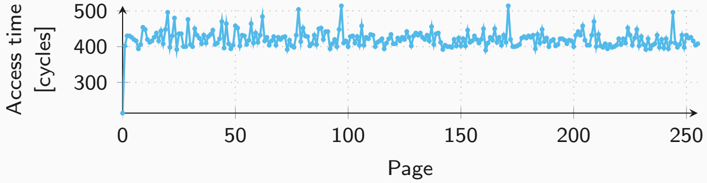


- "Unreachable" code line was actually executed

# Building Meltdown



- Flush+Reload over all pages of the array



- "Unreachable" code line was actually executed
- Exception was only thrown afterwards

## Building Meltdown

- Out-of-order instructions leave microarchitectural traces

## Building Meltdown

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

## Building Meltdown

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

## Building Meltdown

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

## Building Meltdown

- Combine the two things

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

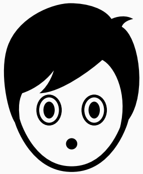
## Building Meltdown

- Combine the two things

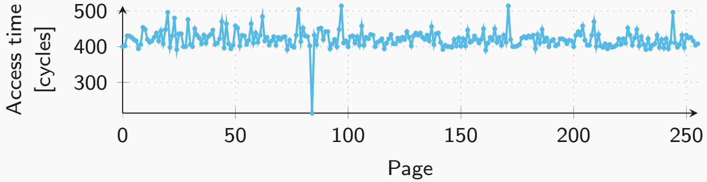
```
char data = *(char*)0xffffffff81a000e0;
array[data * 4096] = 0;
```

- Then check whether any part of array is cached

# Building Meltdown

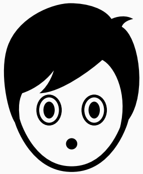


- Flush+Reload over all pages of the array

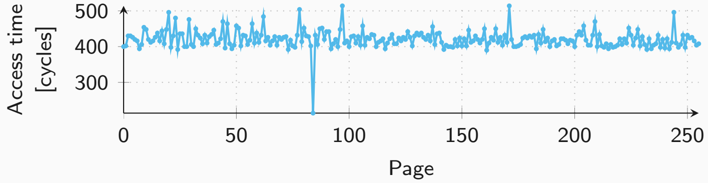


- Index of cache hit reveals data

## Building Meltdown



- Flush+Reload over all pages of the array



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

## Building Meltdown

- Using out-of-order execution, we can read data at any address

## Building Meltdown

MELTDOWN

- Using out-of-order execution, we can read data at any address
- Privilege checks are sometimes too slow

## Building Meltdown

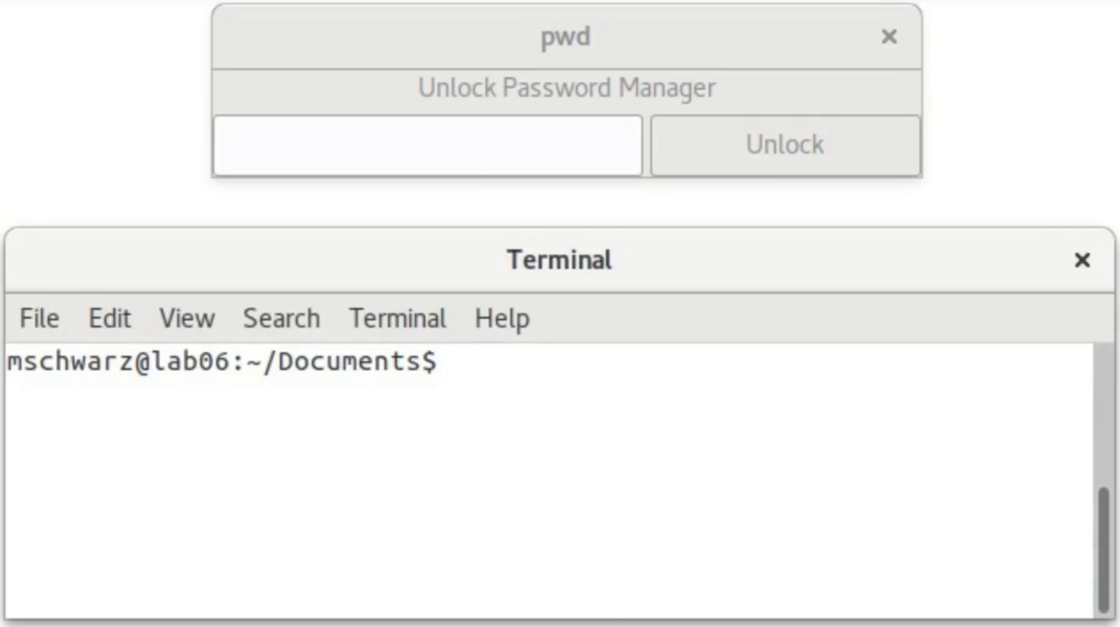
MELTDOWN

- Using out-of-order execution, we can read data at any address
- Privilege checks are sometimes too slow
- Allows to leak kernel memory

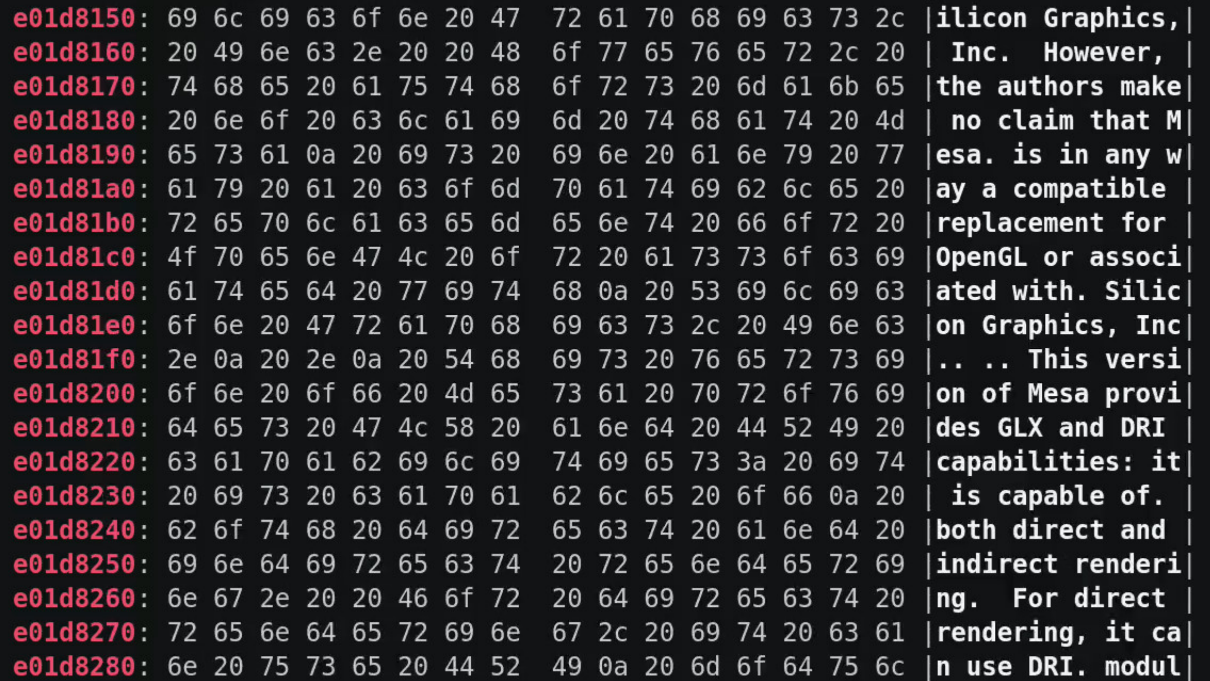
## Building Meltdown

- Using out-of-order execution, we can read data at any address
- Privilege checks are sometimes too slow
- Allows to leak kernel memory
- Entire physical memory is typically also accessible in kernel address space

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



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



```
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 0a 20 | 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|
```

## Details: Exception Handling

- Basic Meltdown code leads to a crash (segfault)

## Details: Exception Handling

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

# Details: Exception Handling

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

Fault Handling

Fault Suppression

Fault Prevention

## Meltdown with Fault Suppression

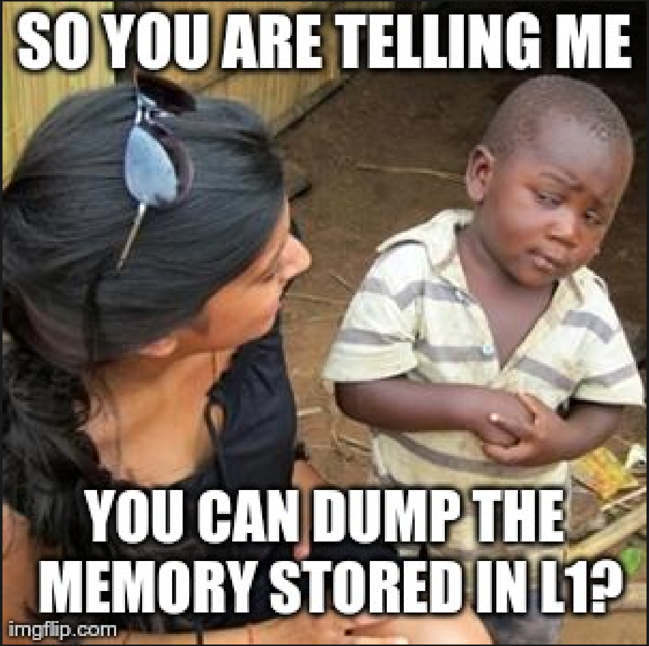
- Intel TSX to suppress exceptions instead of signal handler

```
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);
  }
}
```

## Meltdown with Fault Prevention

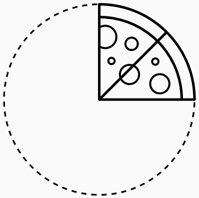
- Speculative execution to prevent exceptions

```
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);
  }
}
```


SO YOU ARE TELLING ME
YOU CAN DUMP THE MEMORY STORED IN L1?
imgflip.com

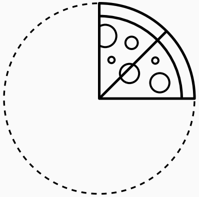
WHAT IF I TOLD YOU
YOU CAN LEAK THE ENTIRE MEMORY
imgflip.com

# Uncached memory



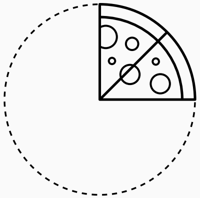
- Initial assumption: we can only read data stored in the L1 with Meltdown

## Uncached memory



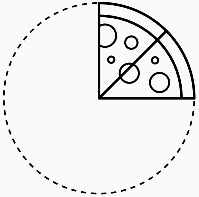
- Initial assumption: we can only read data stored in the L1 with Meltdown. And that's true, sort of:

## Uncached memory



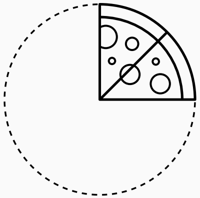
- Initial assumption: we can only read data stored in the L1 with Meltdown. And that's true, sort of:
- Experiment where a thread flushes the value constantly and a thread on a different core reloads the value

## Uncached memory



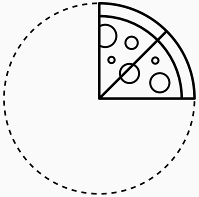
- Initial assumption: we can only read data stored in the L1 with Meltdown. And that's true, sort of:
- Experiment where a thread flushes the value constantly and a thread on a different core reloads the value
  - Target data is not in the L1 cache of the attacking core

## Uncached memory



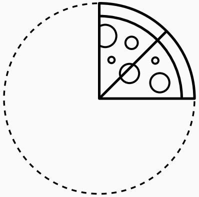
- Initial assumption: we can only read data stored in the L1 with Meltdown. And that's true, sort of:
- Experiment where a thread flushes the value constantly and a thread on a different core reloads the value
    - Target data is not in the L1 cache of the attacking core
- We can still leak the data at a lower reading rate, why?

## Uncached memory



- Initial assumption: we can only read data stored in the L1 with Meltdown. And that's true, sort of:
- Experiment where a thread flushes the value constantly and a thread on a different core reloads the value
  - Target data is not in the L1 cache of the attacking core
- We can still leak the data at a lower reading rate, why?
- $\rightarrow$ *Original* Meltdown only leaks from the L1

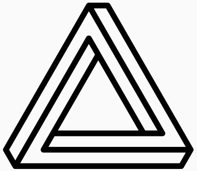
## Uncached memory



- Initial assumption: we can only read data stored in the L1 with Meltdown. And that's true, sort of:
- Experiment where a thread flushes the value constantly and a thread on a different core reloads the value
  - Target data is not in the L1 cache of the attacking core
- We can still leak the data at a lower reading rate, why?
- $\rightarrow$ *Original* Meltdown only leaks from the L1, but we can get data there with load gadgets [6]

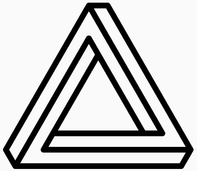
I'LL JUST QUICKLY DUMP THE ENTIRE MEMORY VIA MELTDOWN
imgflip.com

## Practical attacks



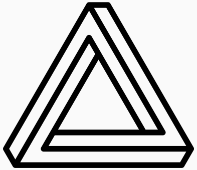
- Dumping the entire physical memory takes some time

## Practical attacks



- Dumping the entire physical memory takes some time
  - Not very practical in most scenarios

## Practical attacks



- Dumping the entire physical memory takes some time
  - Not very practical in most scenarios
- Can we mount more targeted attacks?

## VeraCrypt

- Open-source utility for disk encryption

## VeraCrypt

- Open-source utility for disk encryption
- Fork of TrueCrypt

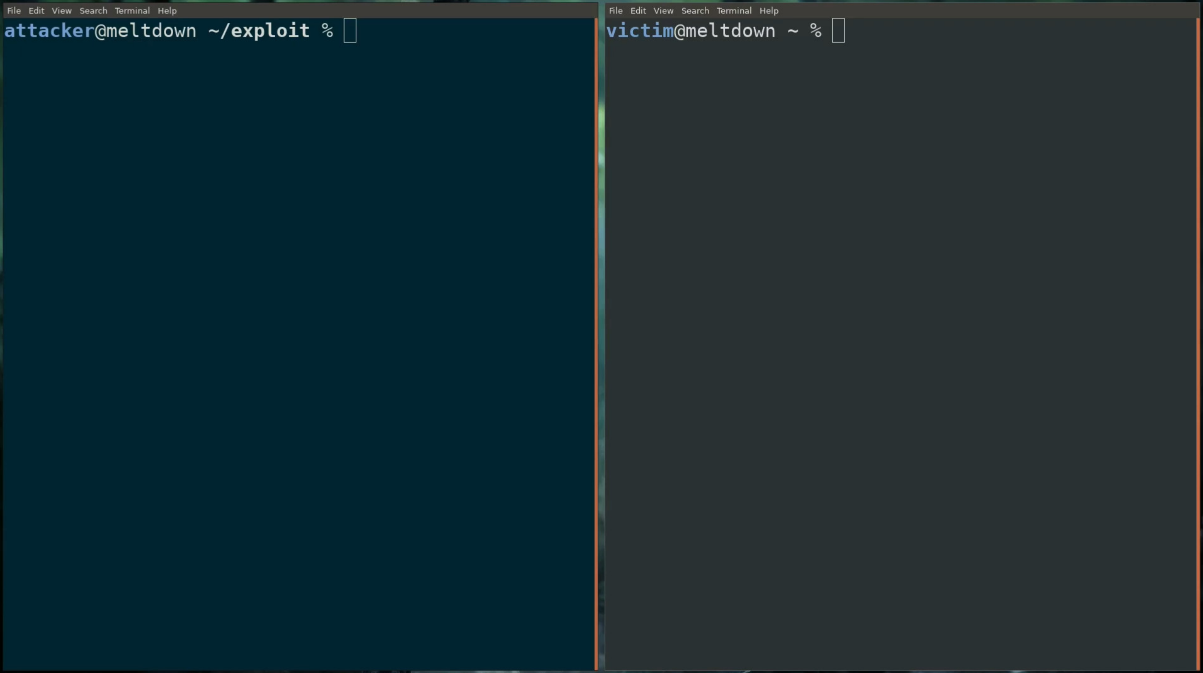
## VeraCrypt

- Open-source utility for disk encryption
- Fork of TrueCrypt
- Cryptographic keys are stored in RAM

## VeraCrypt

VeraCrypt

- Open-source utility for disk encryption
- Fork of TrueCrypt
- Cryptographic keys are stored in RAM
    - With Meltdown, we can extract the keys from DRAM

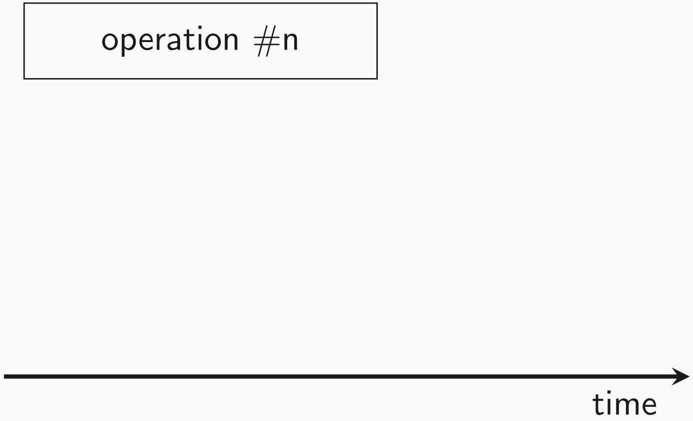

File Edit View Search Terminal Help

```
attacker@meltdown ~/exploit %
```

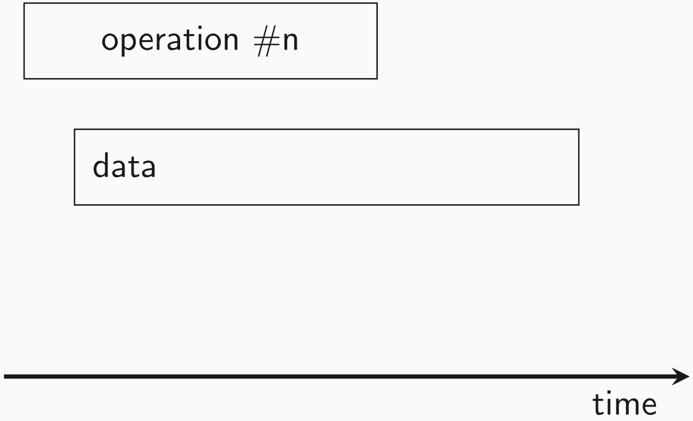
File Edit View Search Terminal Help

```
victim@meltdown ~ %
```

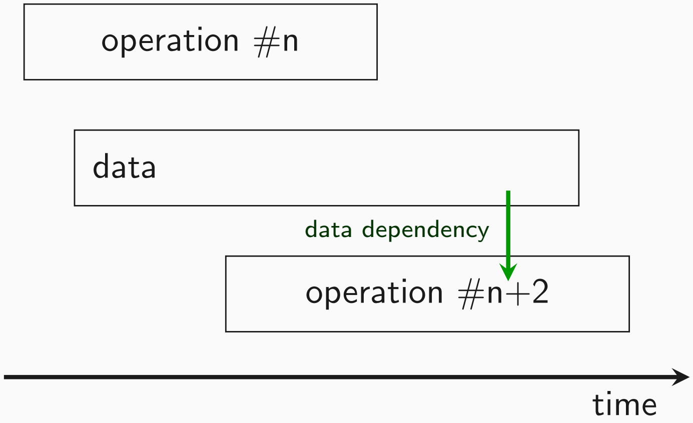
## Meltdown Root Cause



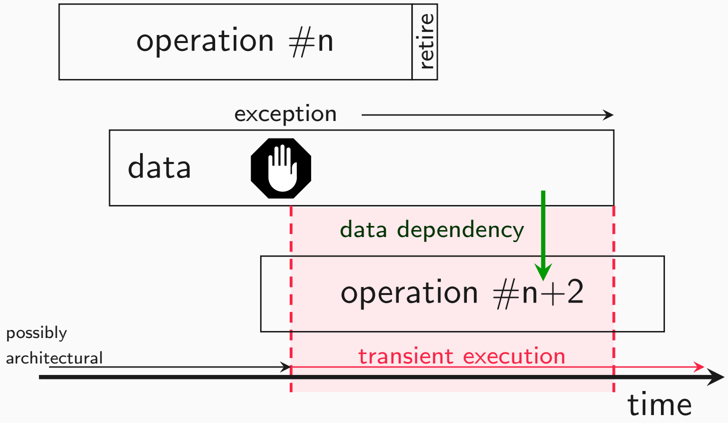
## Meltdown Root Cause



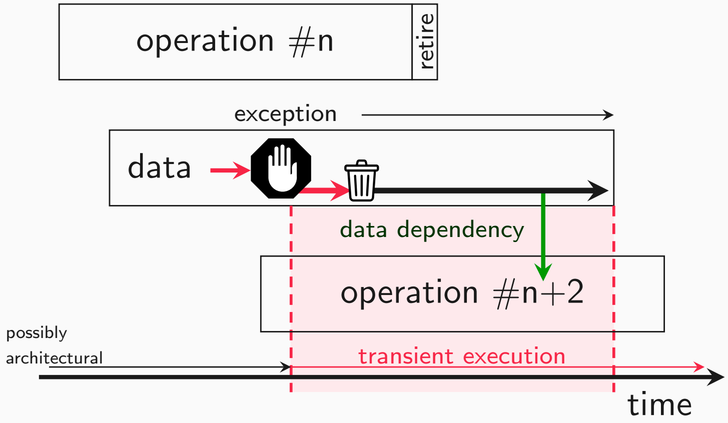
# Meltdown Root Cause



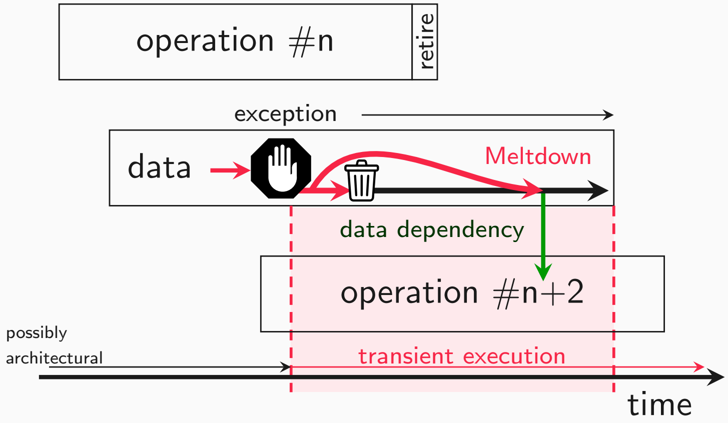
## Meltdown Root Cause



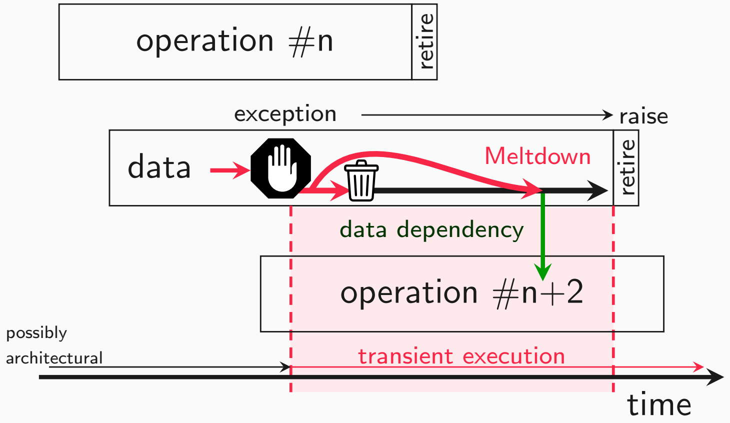
# Meltdown Root Cause



## Meltdown Root Cause



# Meltdown Root Cause



## Generalization

- Meltdown is a whole category of vulnerabilities

## Generalization

- Meltdown is a whole category of vulnerabilities
- Not only the user-accessible check

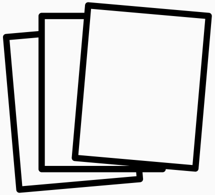
## Generalization

- Meltdown is a whole category of vulnerabilities
- Not only the user-accessible check
- Looking closer at the check...

## Paging

- CPU uses virtual address spaces to isolate processes

## Paging

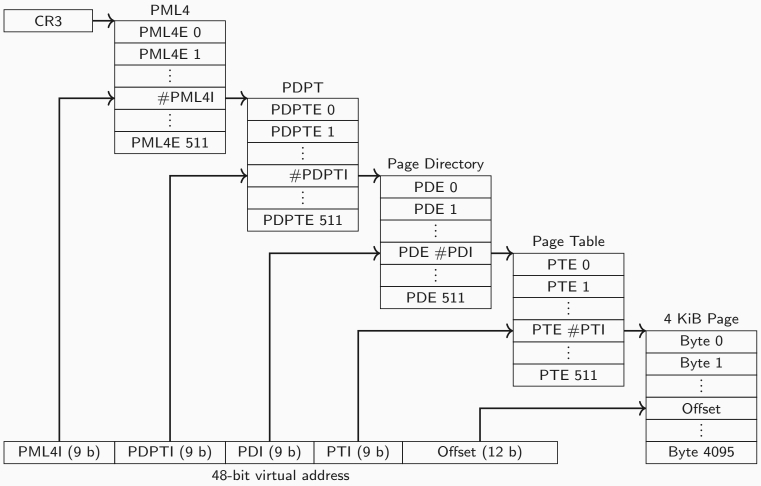


- CPU uses virtual address spaces to isolate processes
- Physical memory is organized in page frames

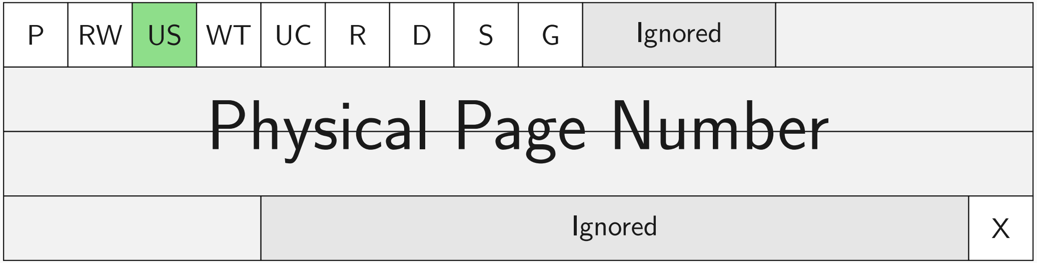
# Paging

- CPU uses virtual address spaces to isolate processes
- Physical memory is organized in page frames
- Virtual memory pages are mapped to page frames using page tables

# Address Translation on x86-64

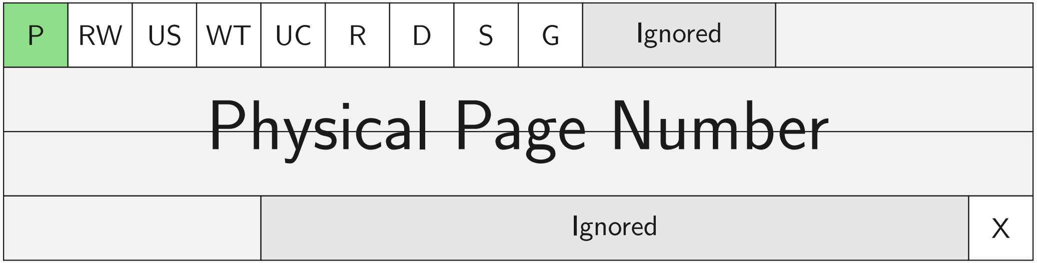


## Page Table Entry

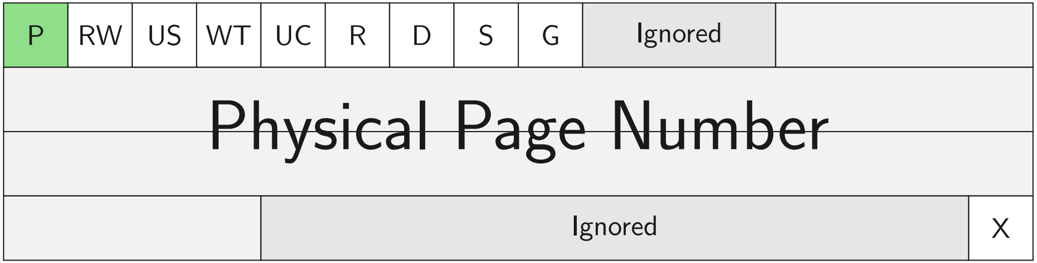


- User/Supervisor bit defines in which privilege level the page can be accessed

## Page Table Entry



## Page Table Entry



- Present bit is the next obvious bit

## Foreshadow-NG [7]

- An even worse bug → Foreshadow-NG/L1TF

## Foreshadow-NG [7]

- An even worse bug → Foreshadow-NG/L1TF
- Exploitable from VMs

## Foreshadow-NG [7]

- An even worse bug $\rightarrow$ Foreshadow-NG/L1TF
- Exploitable from VMs
- Allows leaking data from the L1 cache

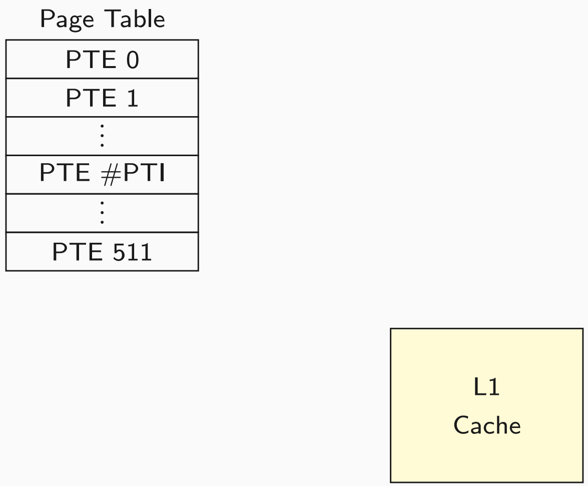
## Foreshadow-NG [7]

- An even worse bug $\rightarrow$ Foreshadow-NG/L1TF
- Exploitable from VMs
- Allows leaking data from the L1 cache
- Same mechanism as Meltdown

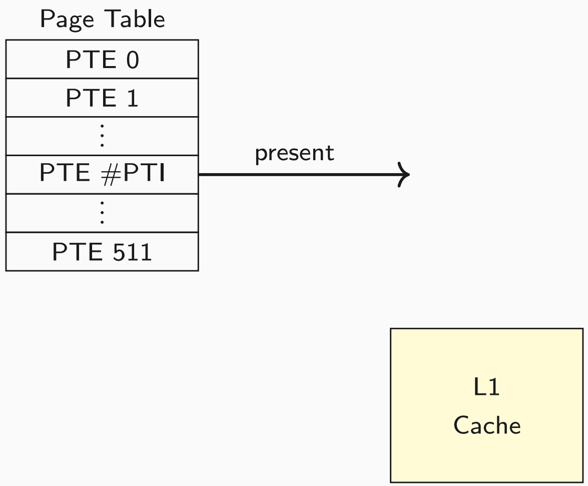
## Foreshadow-NG [7]

- An even worse bug → Foreshadow-NG/L1TF
- Exploitable from VMs
- Allows leaking data from the L1 cache
- Same mechanism as Meltdown
- Just a different bit in the PTE

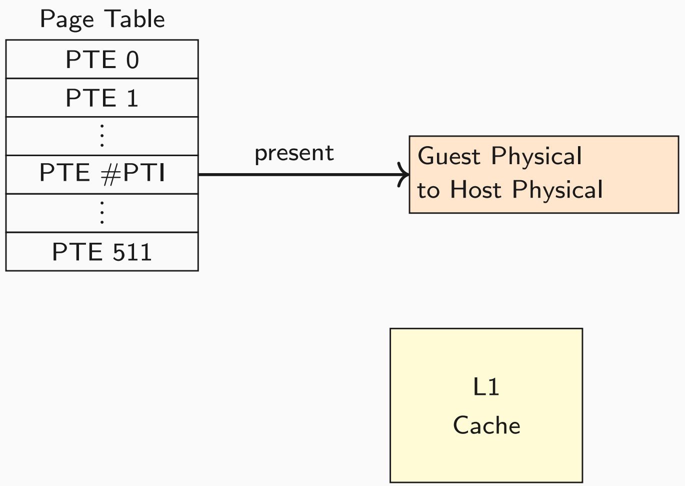
# Foreshadow-NG



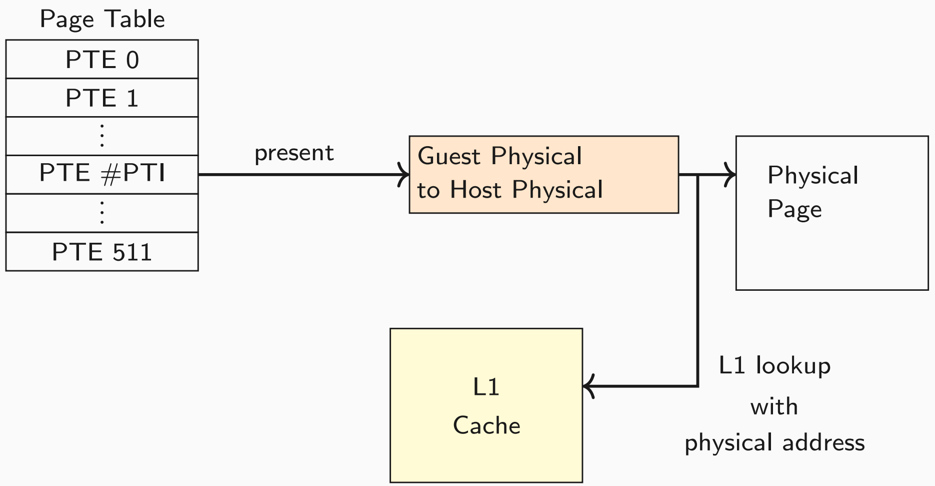
# Foreshadow-NG



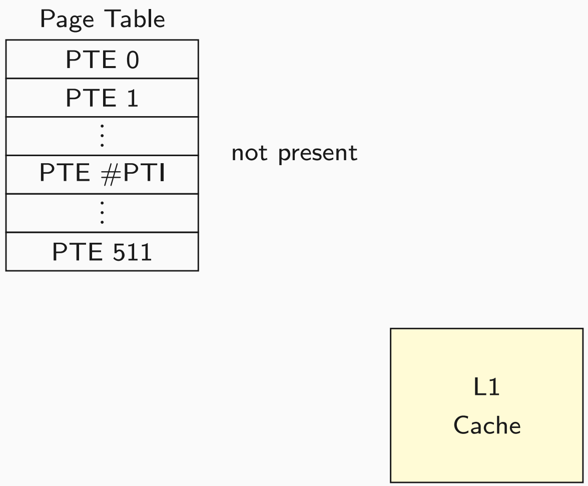
## Foreshadow-NG



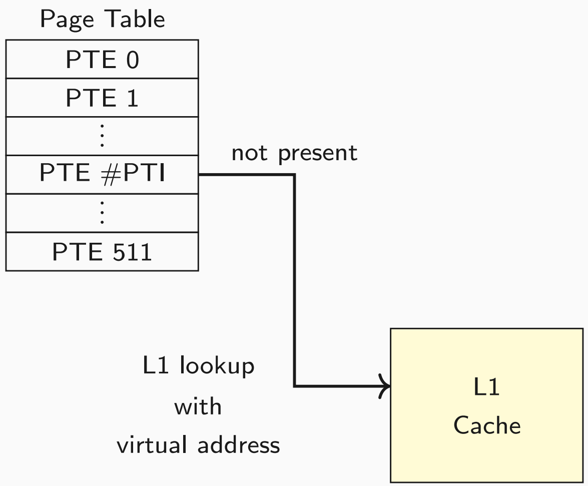
# Foreshadow-NG

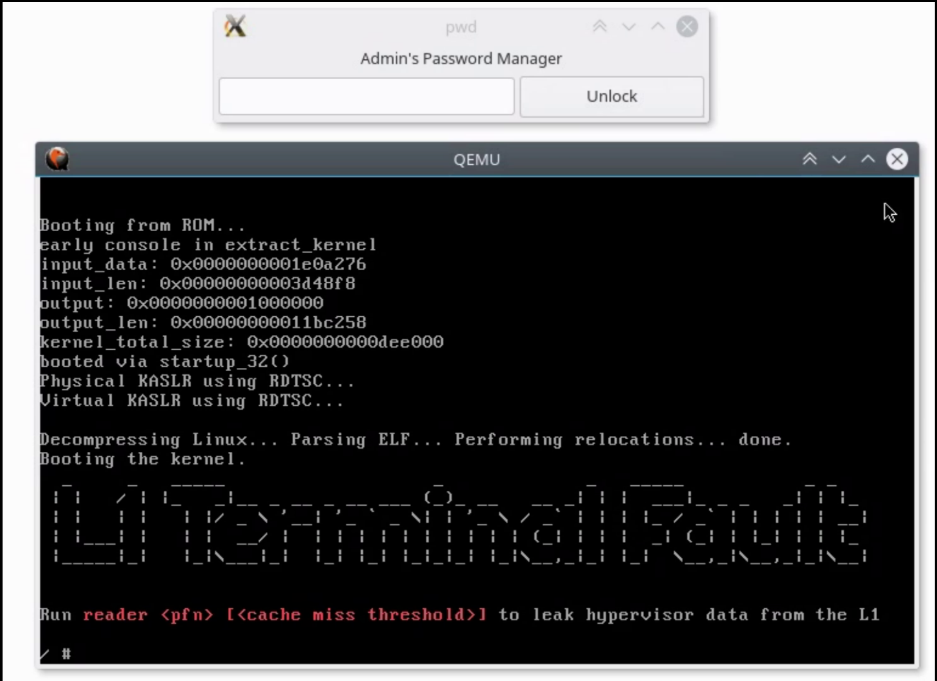


## Foreshadow-NG



## Foreshadow-NG





```
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

/ #
```

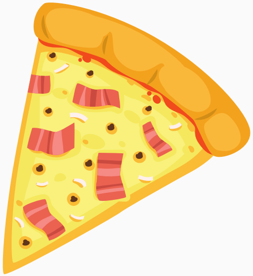
MELTDOWN

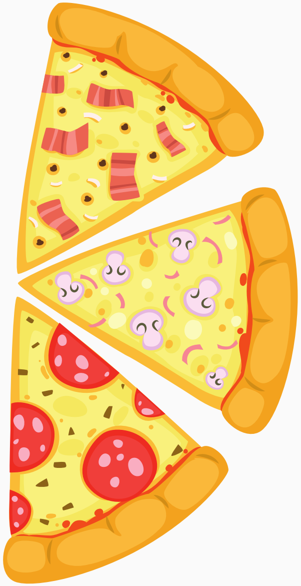
SPECTRE

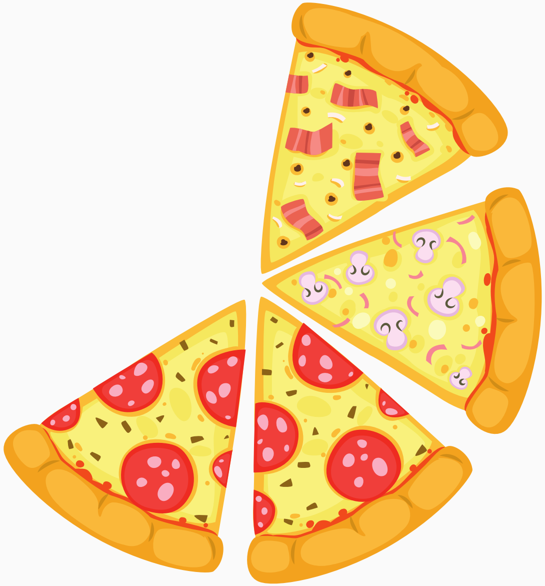
MELTDOWN

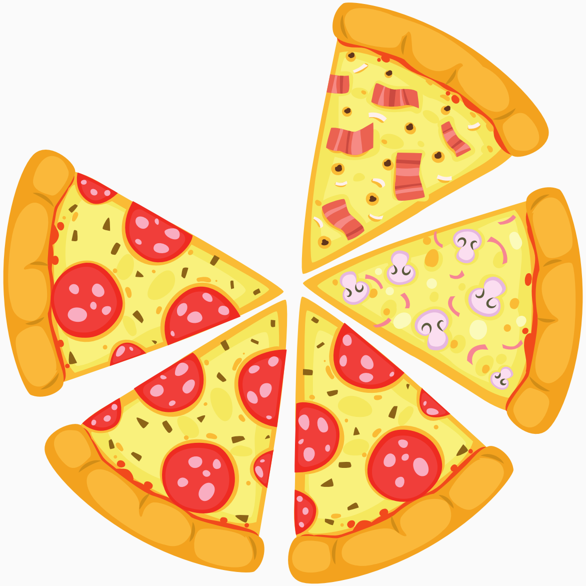
SPECTRE

PIZZA
SPECIAL RECIPES









»A table for 6 please«

## Speculative Cooking

»A table for 6 please«

PIZZA
SPECIAL RECIPES

PIZZA
SPECIAL RECIPES

Pizza

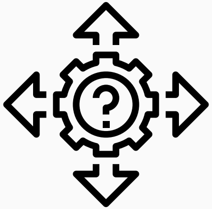
PIZZA
SPECIAL RECIPES

Pizza

PIZZA
SPECIAL RECIPES

Pizza

## Speculation Causes



- Many predictors in modern CPUs

## Speculation Causes

- Many predictors in modern CPUs
    - Branch taken/not taken (PHT)

## Speculation Causes

- Many predictors in modern CPUs
  - Branch taken/not taken (PHT)
  - Call/Jump destination (BTB)

## Speculation Causes

- Many predictors in modern CPUs
    - Branch taken/not taken (PHT)
    - Call/Jump destination (BTB)
    - Function return destination (RSB)

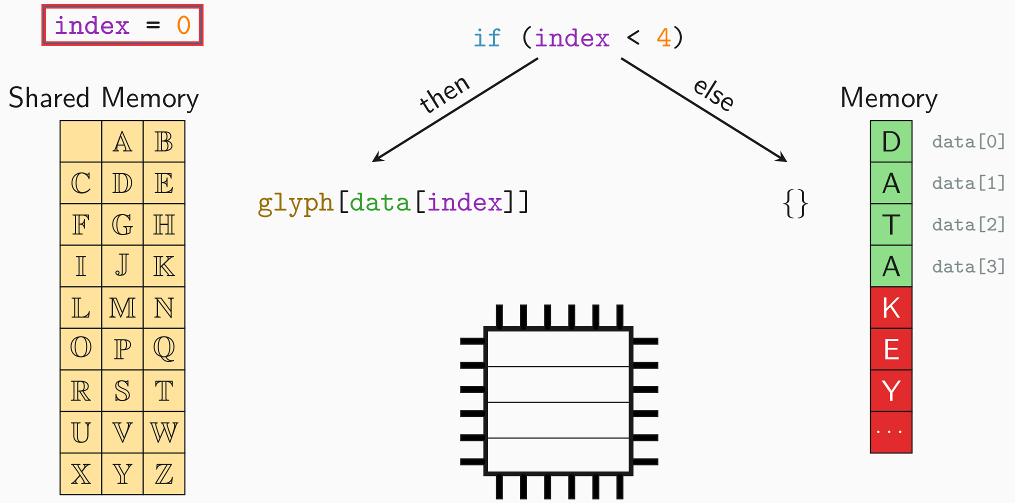
## Speculation Causes

- Many predictors in modern CPUs
  - Branch taken/not taken (PHT)
  - Call/Jump destination (BTB)
  - Function return destination (RSB)
  - Load matches previous store (STL)

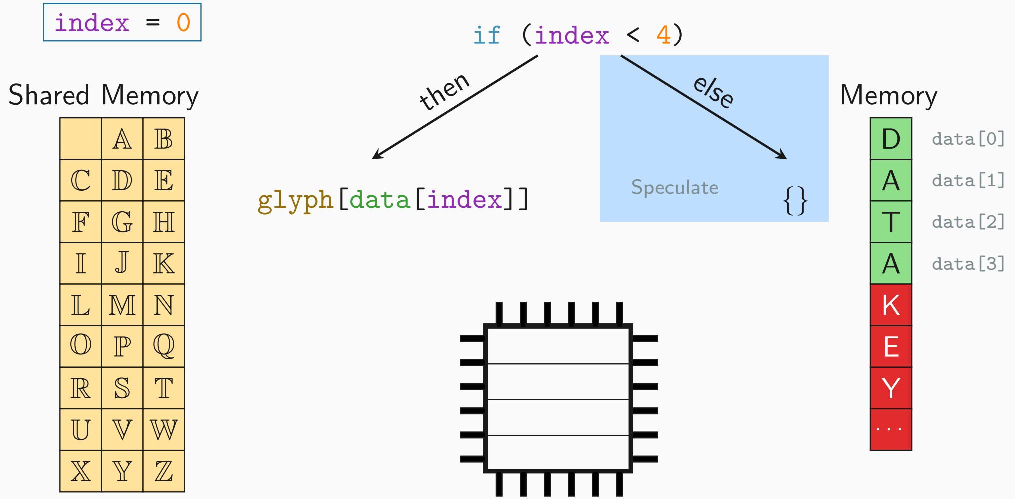
## Speculation Causes

- Many predictors in modern CPUs
  - Branch taken/not taken (PHT)
  - Call/Jump destination (BTB)
  - Function return destination (RSB)
  - Load matches previous store (STL)
- Most are even shared among processes

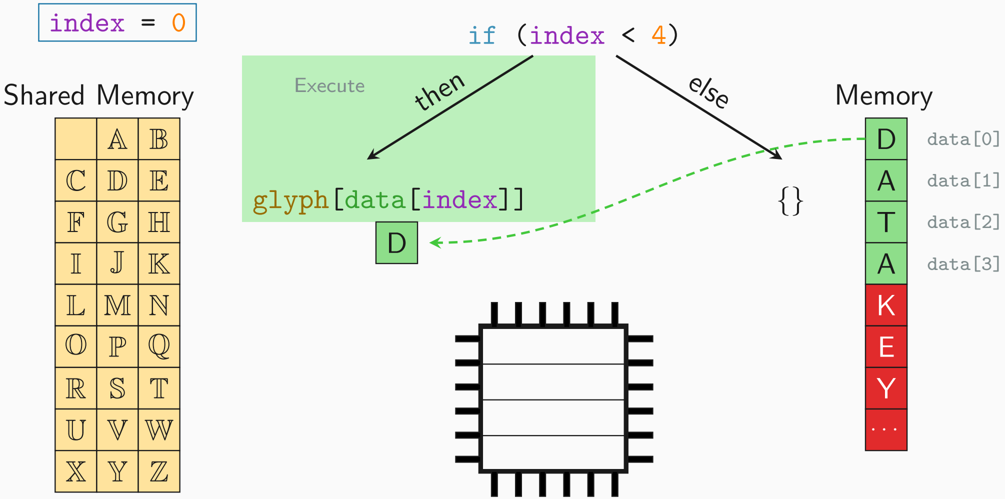
## Spectre-PHT (Variant 1)



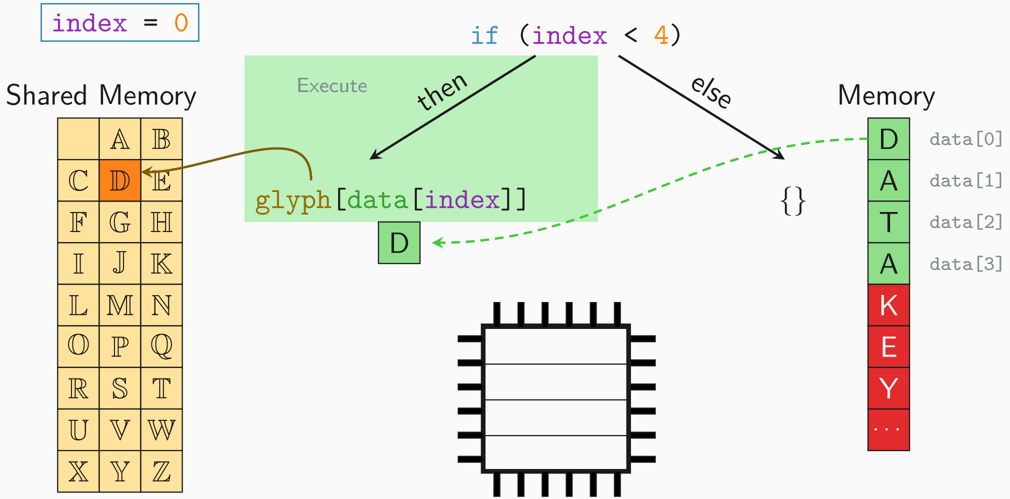
# Spectre-PHT (Variant 1)



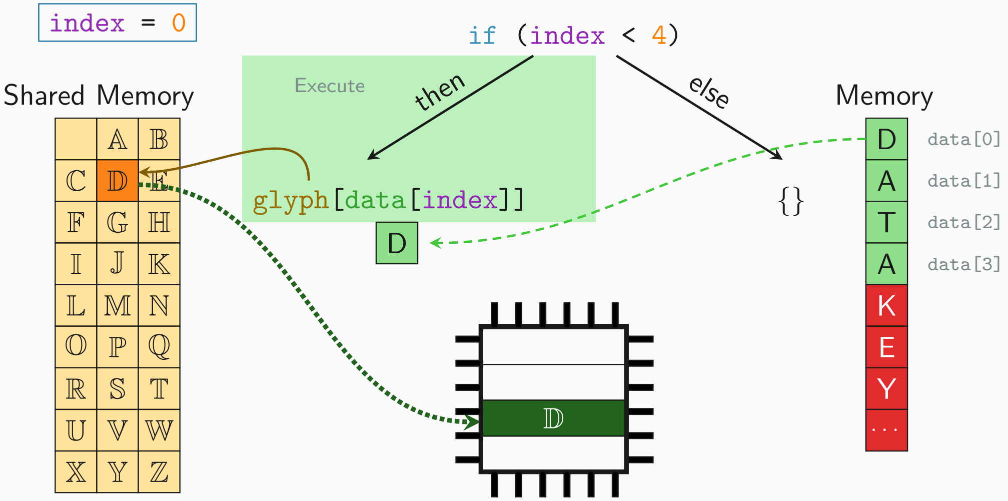
## Spectre-PHT (Variant 1)



## Spectre-PHT (Variant 1)



## Spectre-PHT (Variant 1)



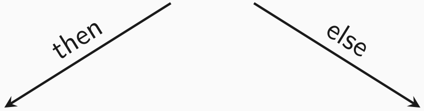
# Spectre-PHT (Variant 1)

```
index = 1
```

Shared Memory

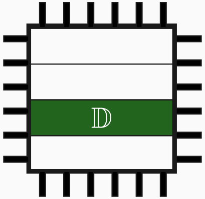
| | | |
|---|---|---|
| | A | B |
| C | D | E |
| F | G | H |
| I | J | K |
| L | M | N |
| O | P | Q |
| R | S | T |
| U | V | W |
| X | Y | Z |

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



```
glyph[data[index]]
```

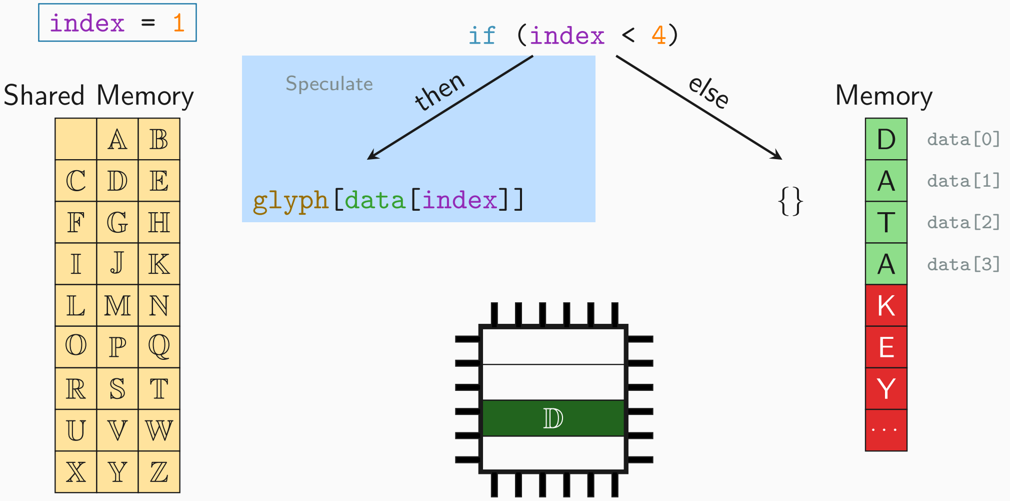
```
{}
```



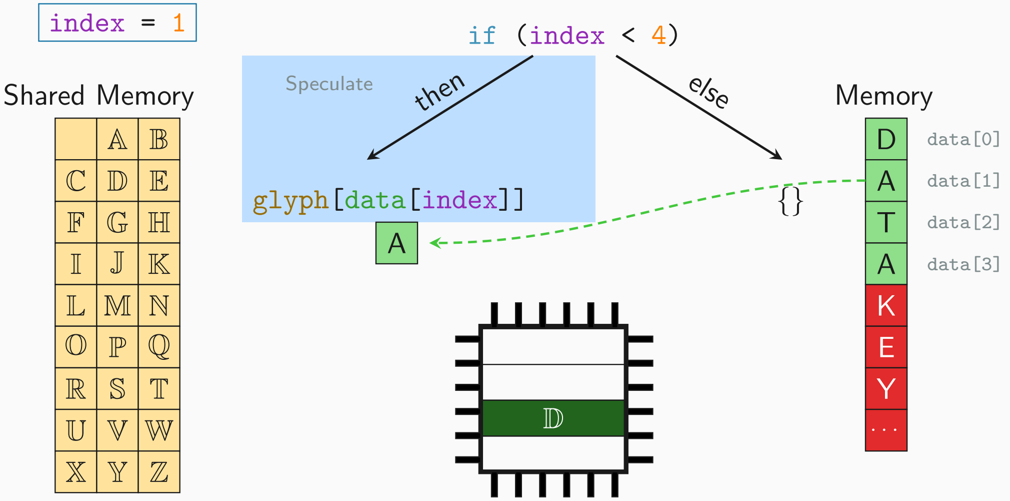
Memory

| | |
|---|---|
| D | data[0] |
| A | data[1] |
| T | data[2] |
| A | data[3] |
| K | |
| E | |
| Y | |
| ... | |

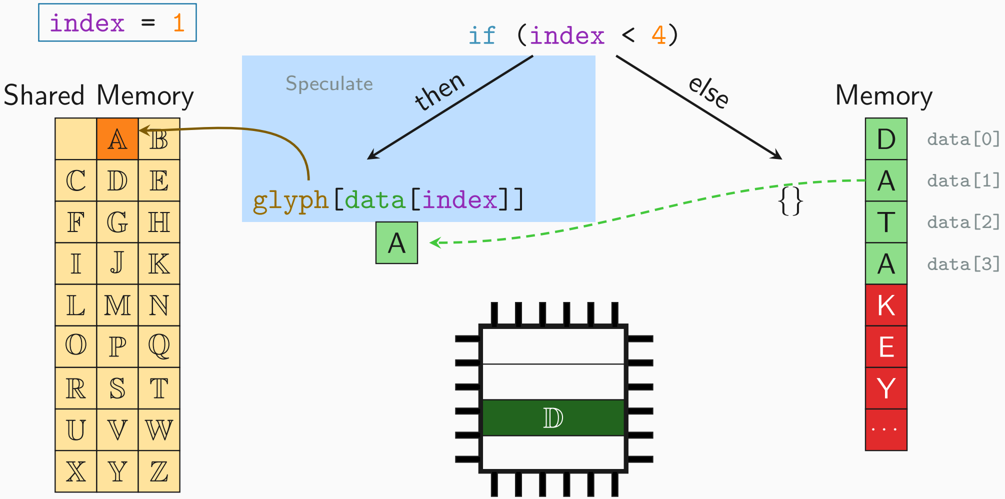
## Spectre-PHT (Variant 1)



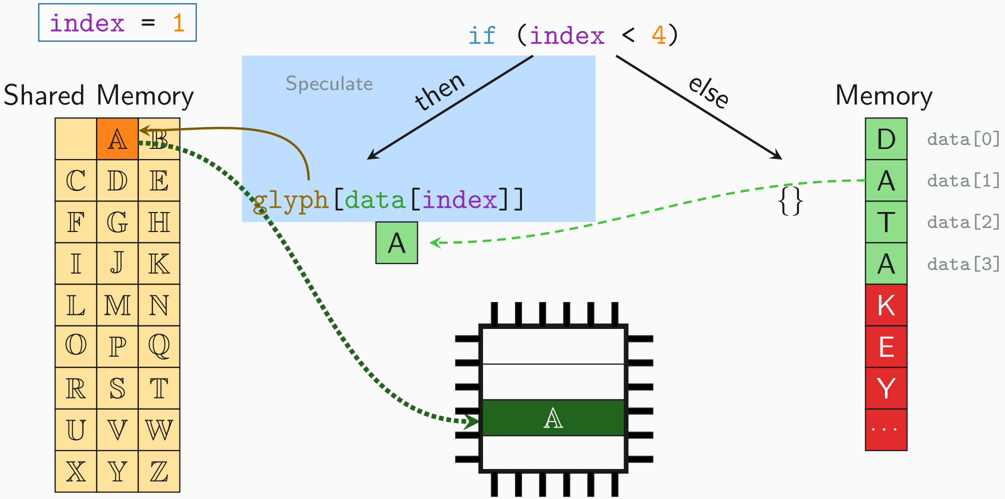
# Spectre-PHT (Variant 1)



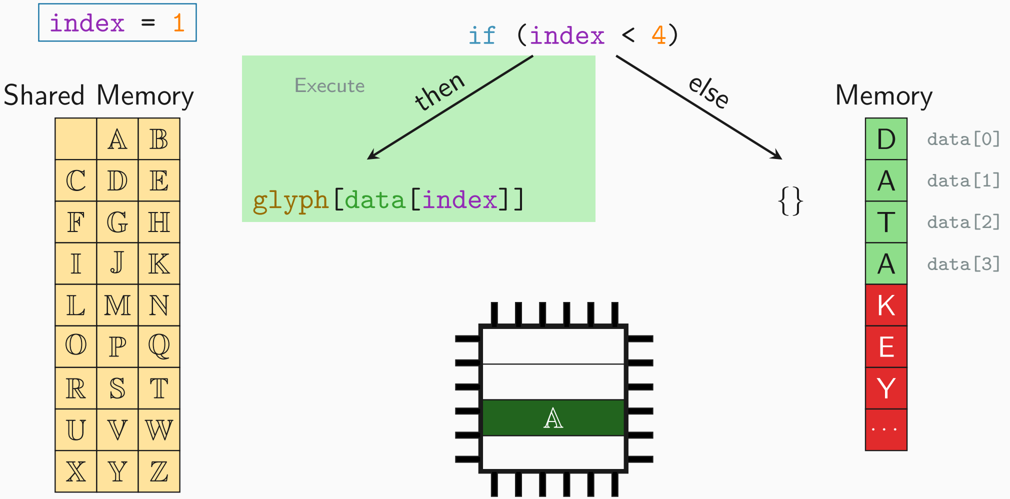
## Spectre-PHT (Variant 1)



# Spectre-PHT (Variant 1)



# Spectre-PHT (Variant 1)

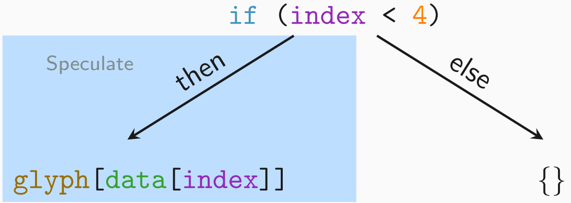


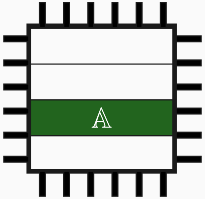
# Spectre-PHT (Variant 1)

`index = 2`

Shared Memory

| | | |
|---|---|---|
| | A | B |
| C | D | E |
| F | G | H |
| I | J | K |
| L | M | N |
| O | P | Q |
| R | S | T |
| U | V | W |
| X | Y | Z |

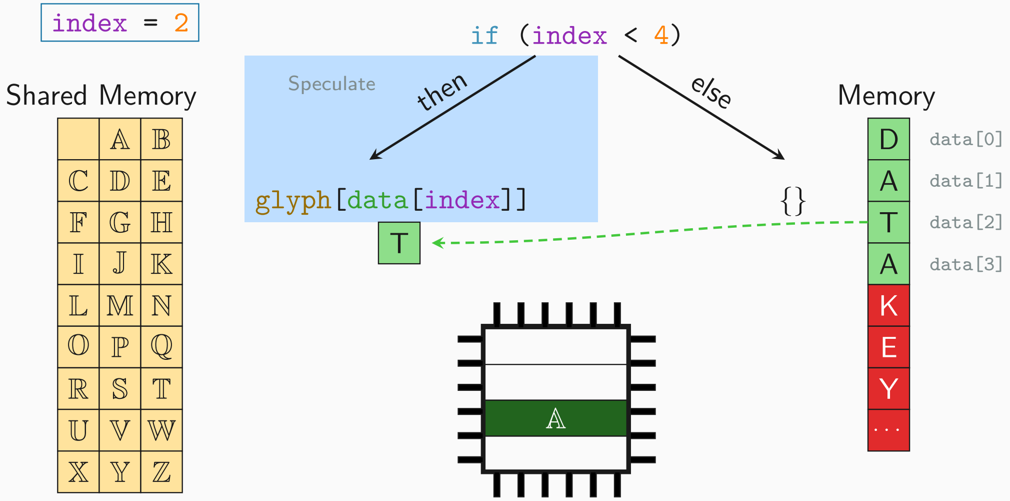




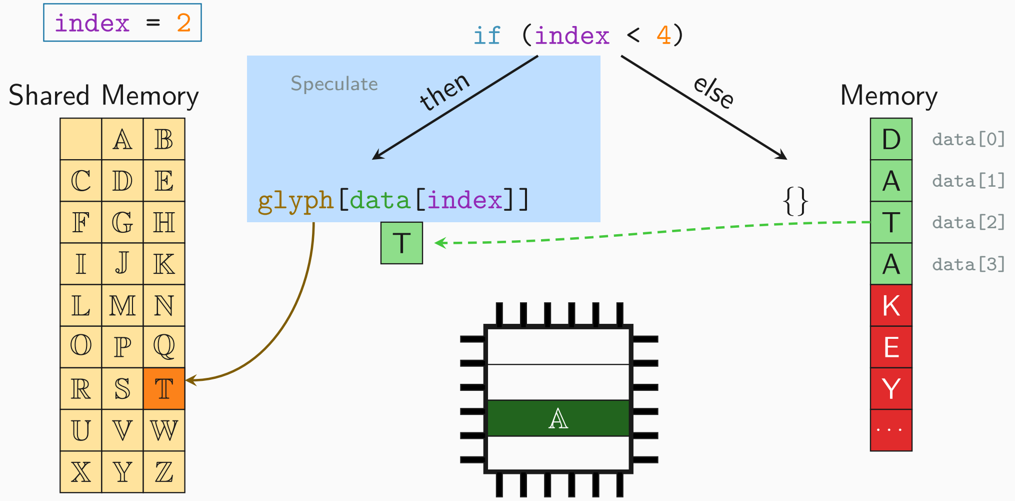
Memory

| | |
|---|---|
| D | data[0] |
| A | data[1] |
| T | data[2] |
| A | data[3] |
| K | |
| E | |
| Y | |
| ... | |

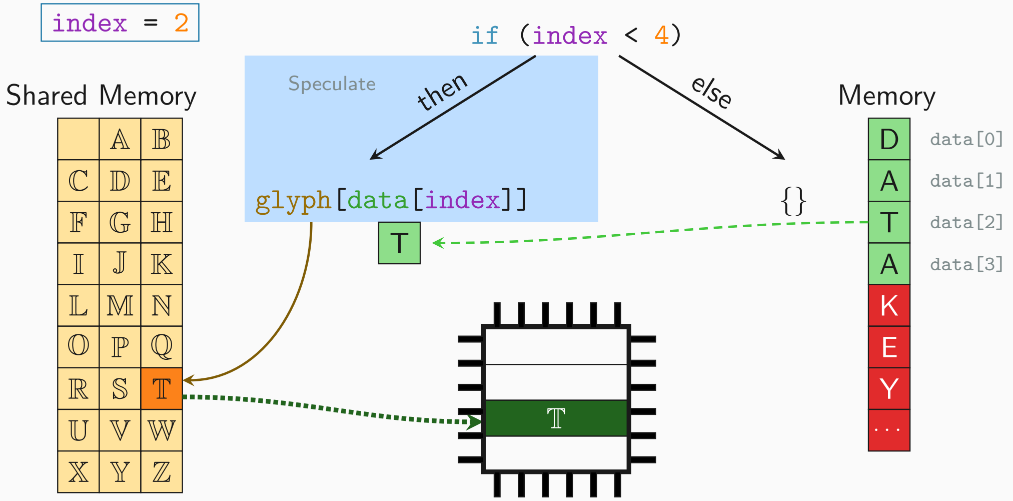
## Spectre-PHT (Variant 1)



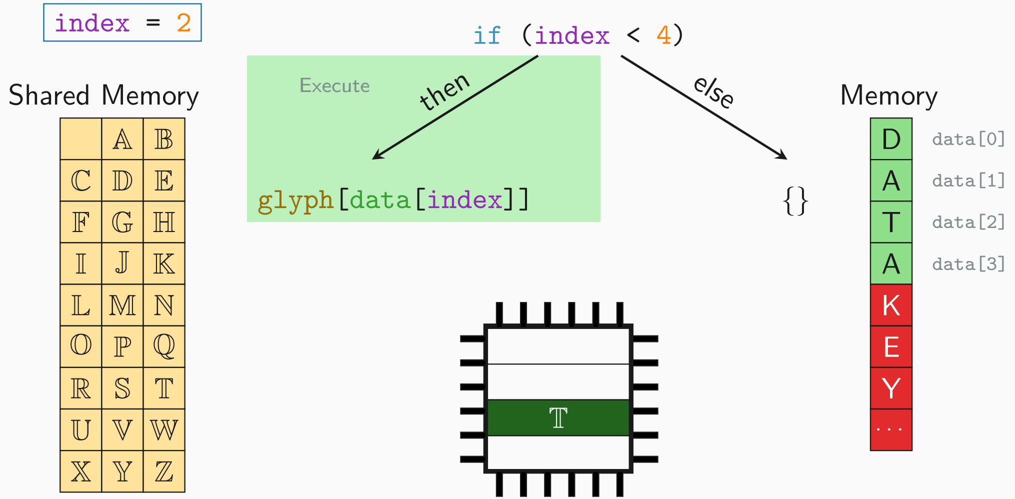
## Spectre-PHT (Variant 1)



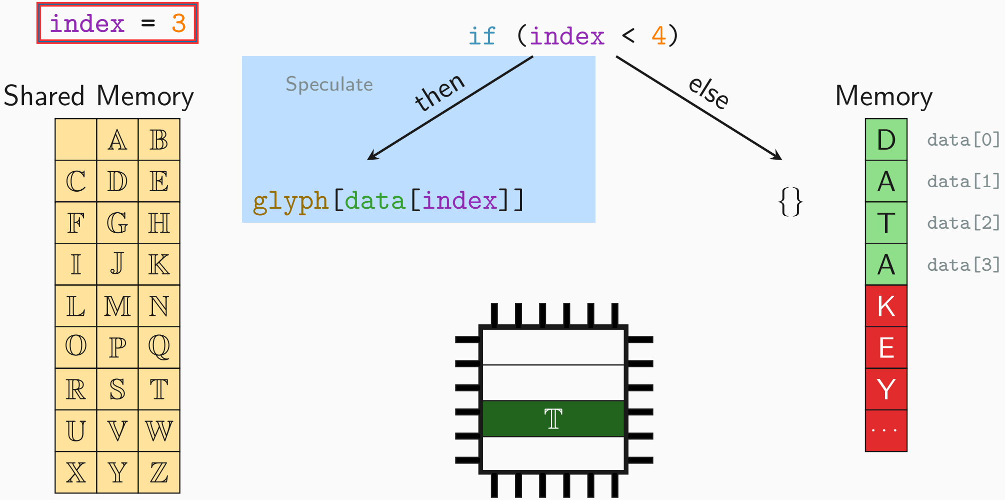
# Spectre-PHT (Variant 1)



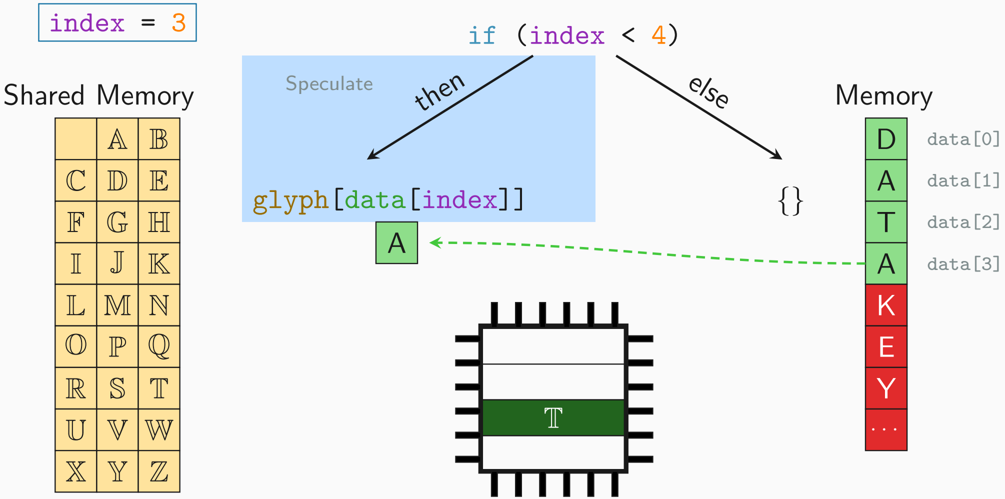
# Spectre-PHT (Variant 1)



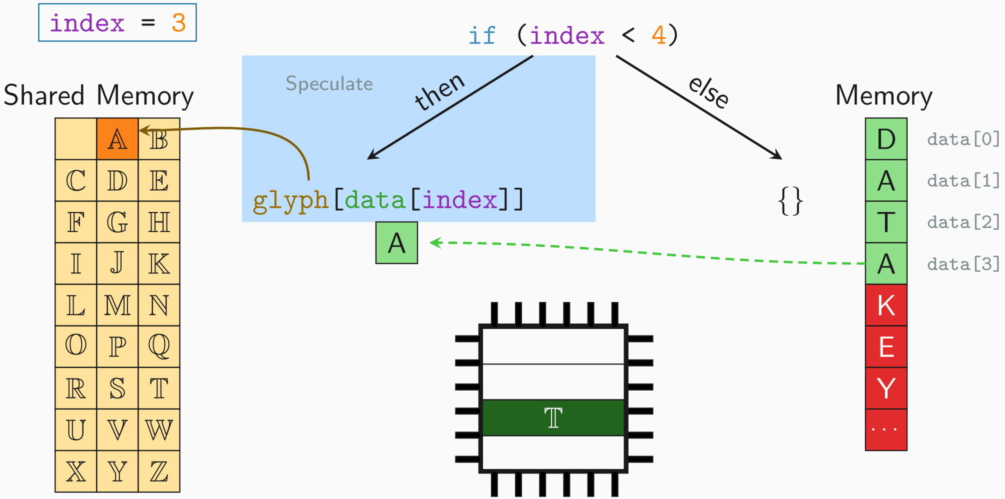
# Spectre-PHT (Variant 1)



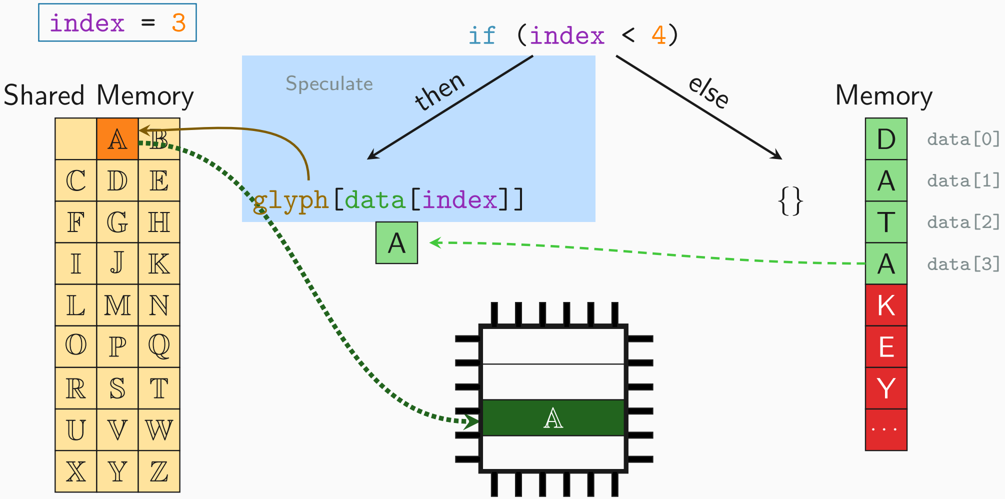
## Spectre-PHT (Variant 1)



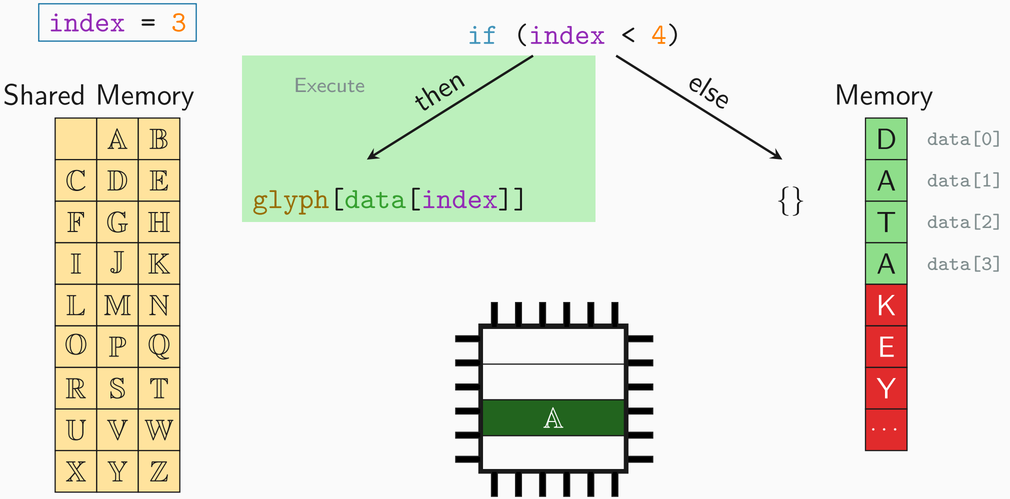
# Spectre-PHT (Variant 1)



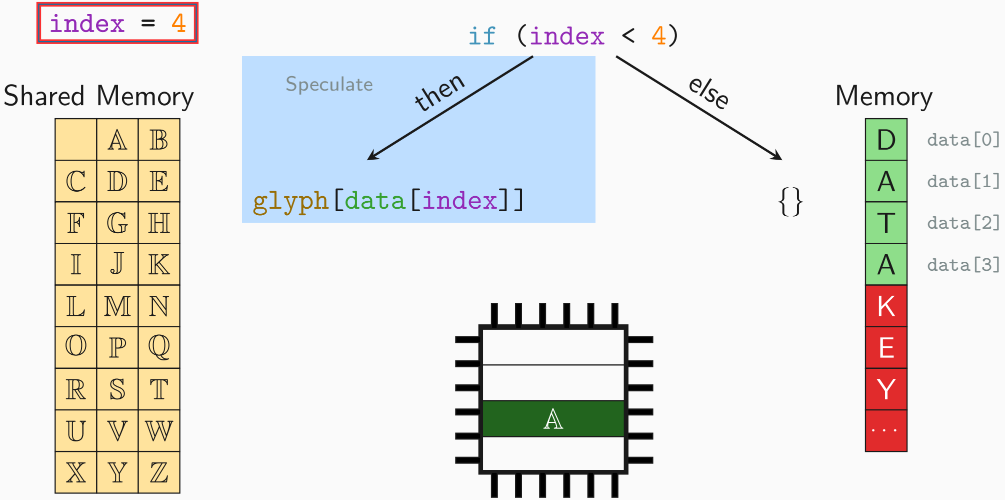
# Spectre-PHT (Variant 1)



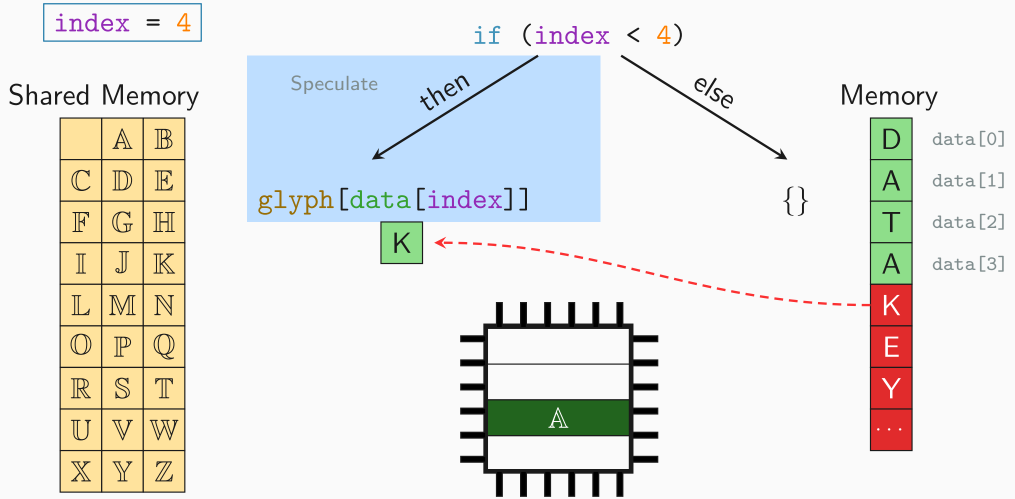
# Spectre-PHT (Variant 1)



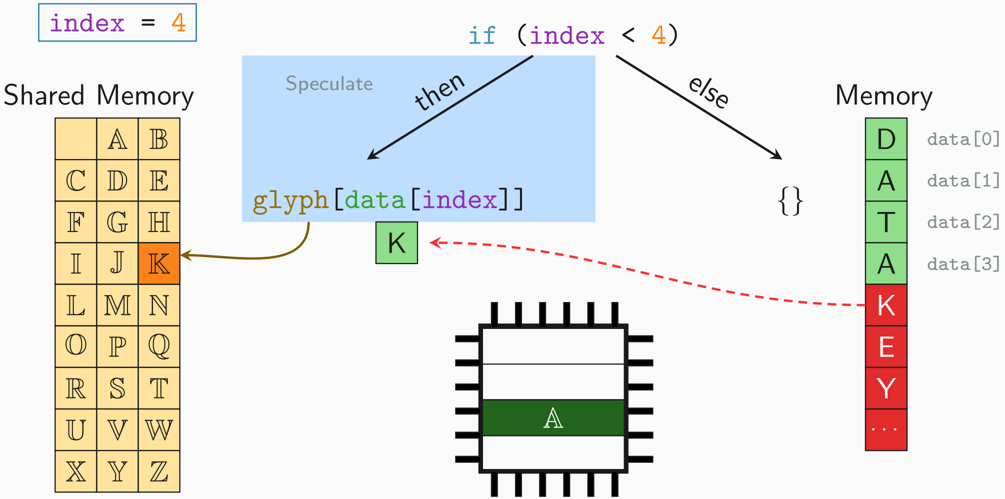
# Spectre-PHT (Variant 1)



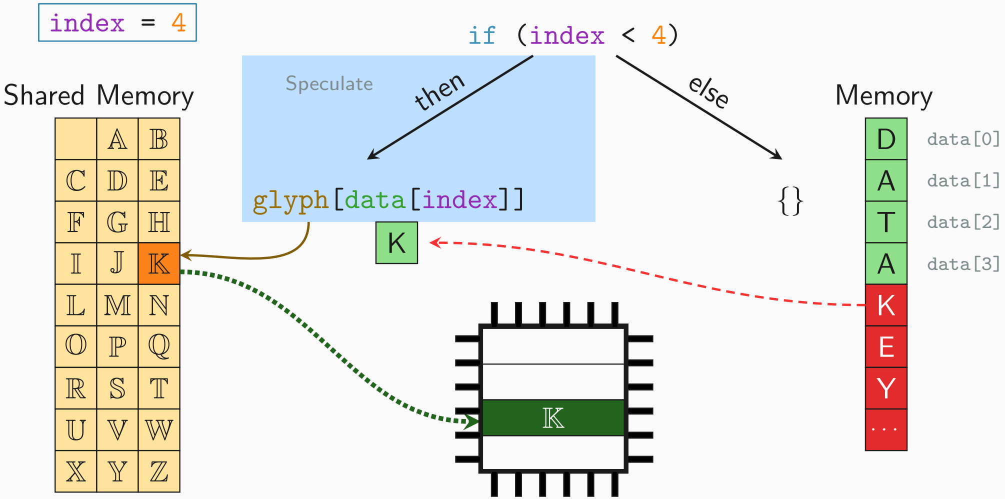
## Spectre-PHT (Variant 1)



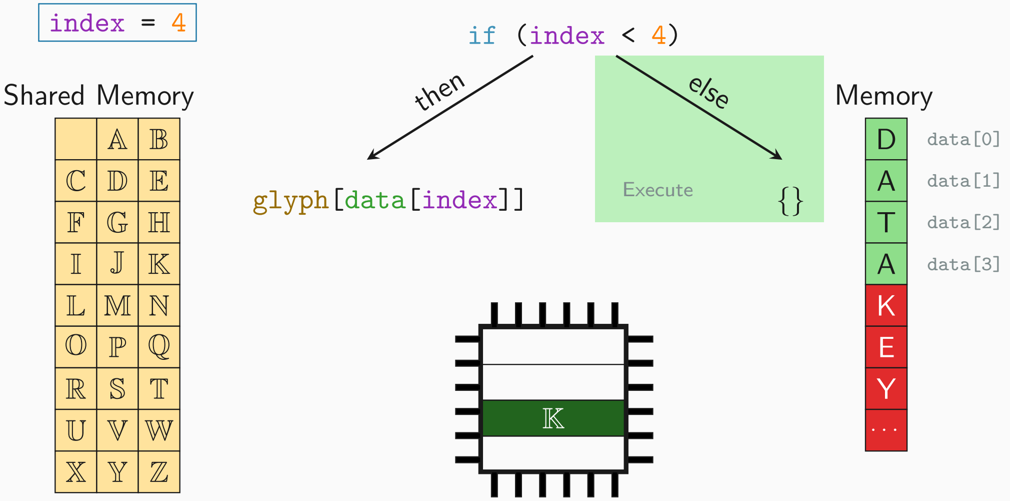
# Spectre-PHT (Variant 1)



## Spectre-PHT (Variant 1)

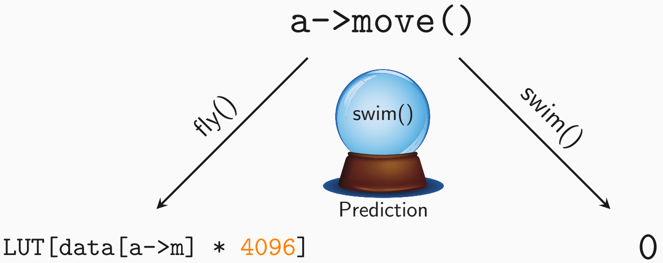


## Spectre-PHT (Variant 1)



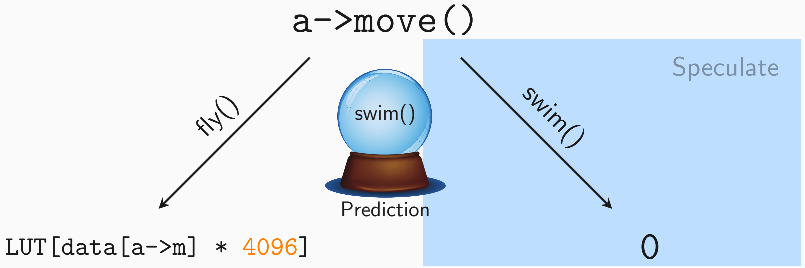
# Spectre-BTB (Variant 2)

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



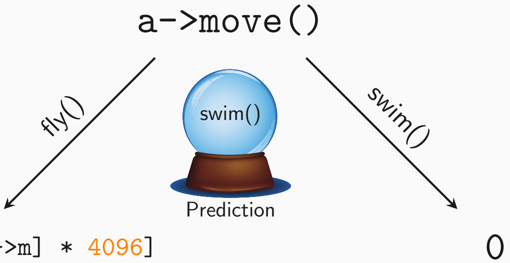
## Spectre-BTB (Variant 2)

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



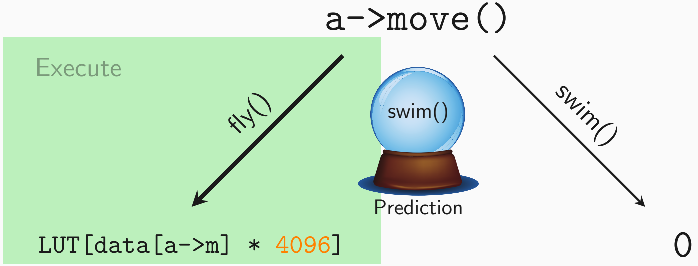
## Spectre-BTB (Variant 2)

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



## Spectre-BTB (Variant 2)

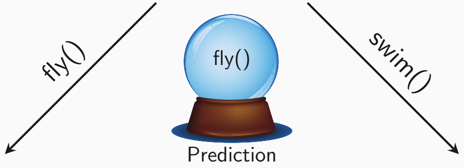
```
Animal* a = bird;
```



## Spectre-BTB (Variant 2)

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

```
a->move()
```

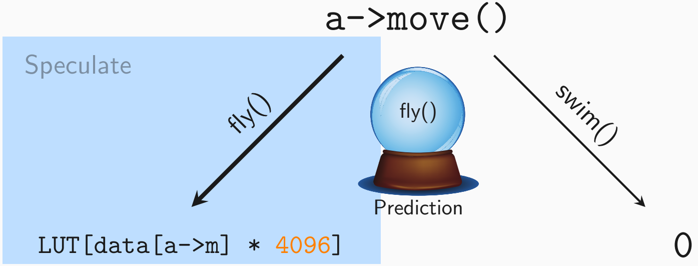


```
LUT[data[a->m] * 4096]
```

```
0
```

## Spectre-BTB (Variant 2)

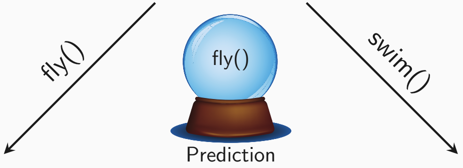
```
Animal* a = bird;
```



## Spectre-BTB (Variant 2)

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

```
a->move()
```

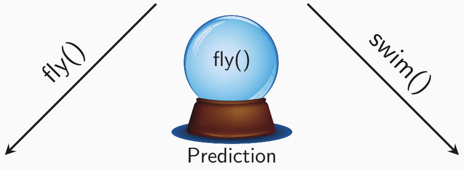


```
LUT[data[a->m] * 4096]          0
```

## Spectre-BTB (Variant 2)

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

```
a->move()
```



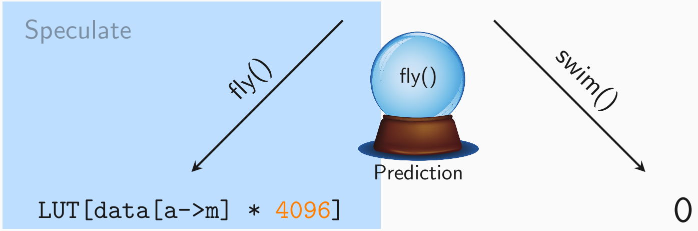
```
LUT[data[a->m] * 4096]
```

```
0
```

# Spectre-BTB (Variant 2)

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

```
a->move()
```

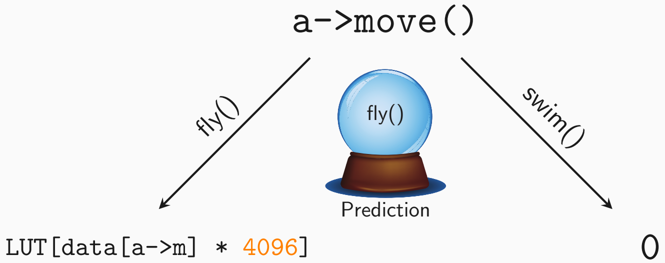


```
LUT[data[a->m] * 4096]
```

```
0
```

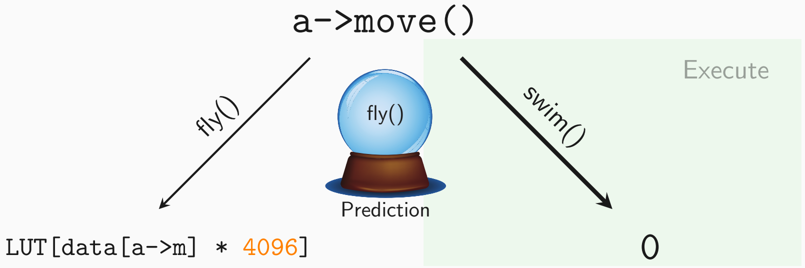
# Spectre-BTB (Variant 2)

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



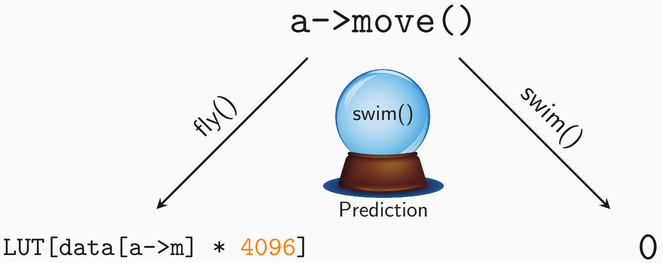
## Spectre-BTB (Variant 2)

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



## Spectre-BTB (Variant 2)

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



## Spectre-STL (Variant 4) [1]

- Loads can be executed out-of-order

## Spectre-STL (Variant 4) [1]

- Loads can be executed out-of-order $\rightarrow$ need to check for previous stores

## Spectre-STL (Variant 4) [1]

- Loads can be executed out-of-order $\rightarrow$ need to check for previous stores
- Check is time consuming

## Spectre-STL (Variant 4) [1]

- Loads can be executed out-of-order $\rightarrow$ need to check for previous stores
- Check is time consuming
- Optimization: Speculate whether a store happened or not

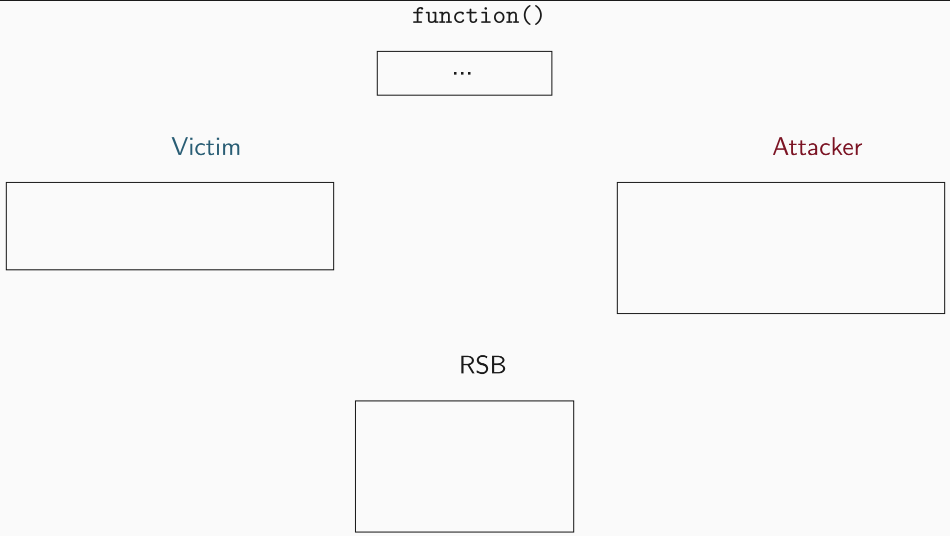
## Spectre-STL (Variant 4) [1]

- Loads can be executed out-of-order $\rightarrow$ need to check for previous stores
- Check is time consuming
- Optimization: Speculate whether a store happened or not
  - no store: bypass check

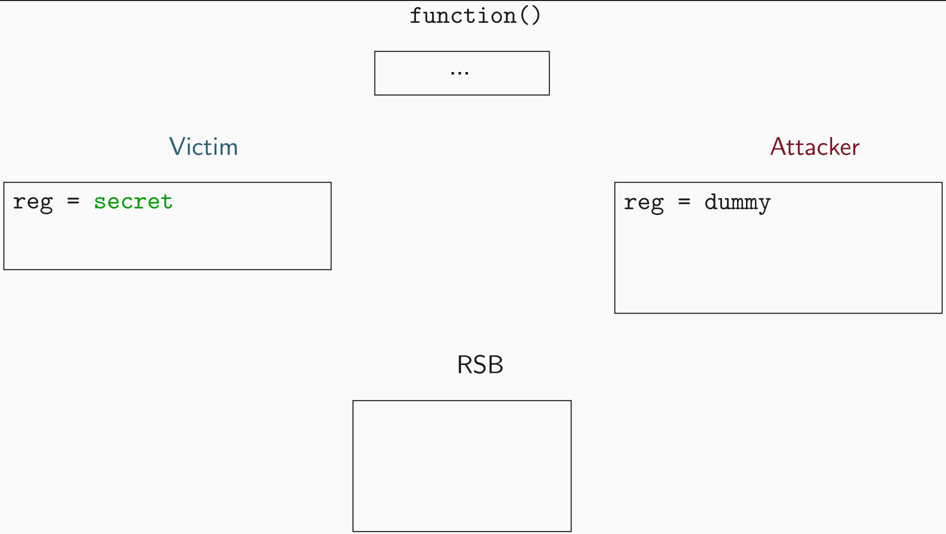
## Spectre-STL (Variant 4) [1]

- Loads can be executed out-of-order $\rightarrow$ need to check for previous stores
- Check is time consuming
- Optimization: Speculate whether a store happened or not
    - no store: bypass check
    - stall

## Spectre-RSB (Variant 5) [3, 5]



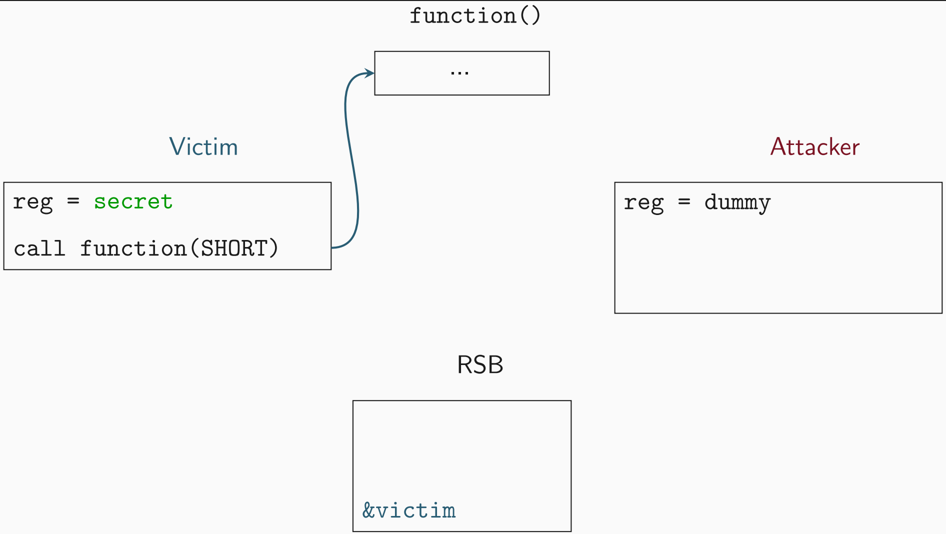
## Spectre-RSB (Variant 5) [3, 5]



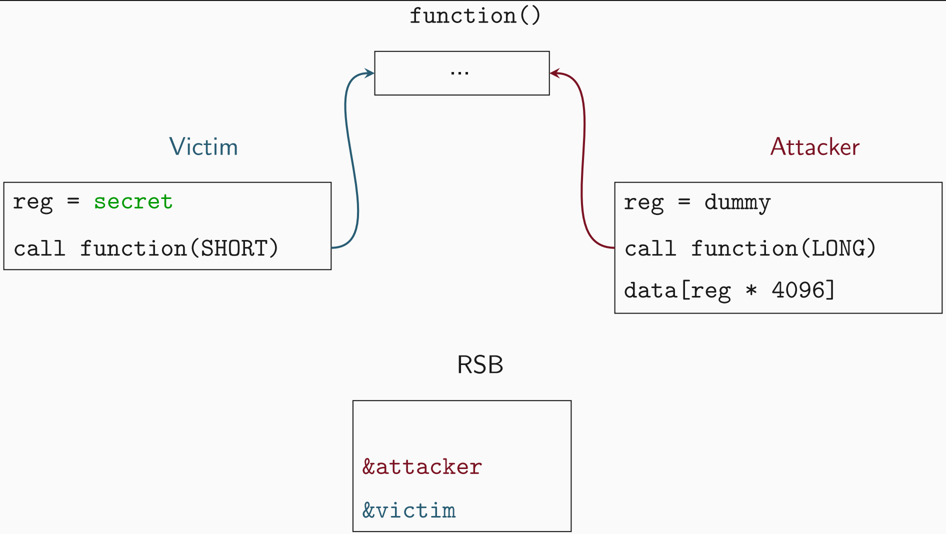
```
reg = secret
```

```
reg = dummy
```

## Spectre-RSB (Variant 5) [3, 5]



## Spectre-RSB (Variant 5) [3, 5]

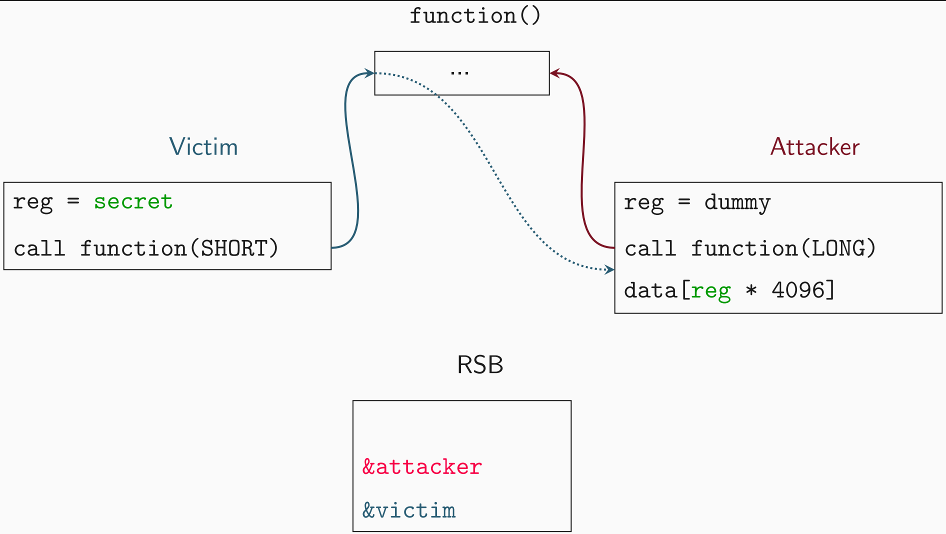


```
reg = secret
call function(SHORT)
```

```
reg = dummy
call function(LONG)
data[reg * 4096]
```

```
&attacker
&victim
```

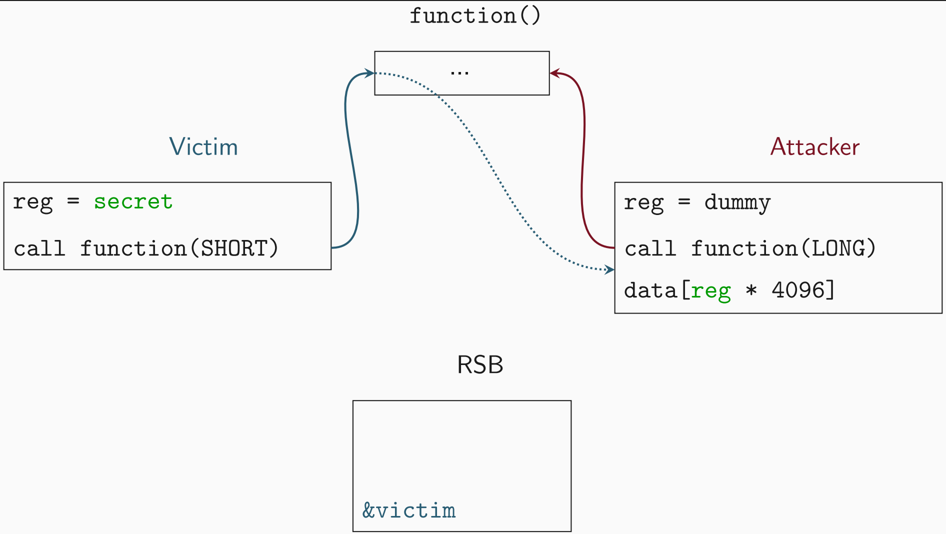
# Spectre-RSB (Variant 5) [3, 5]



```
reg = secret
call function(SHORT)
```

```
reg = dummy
call function(LONG)
data[reg * 4096]
```

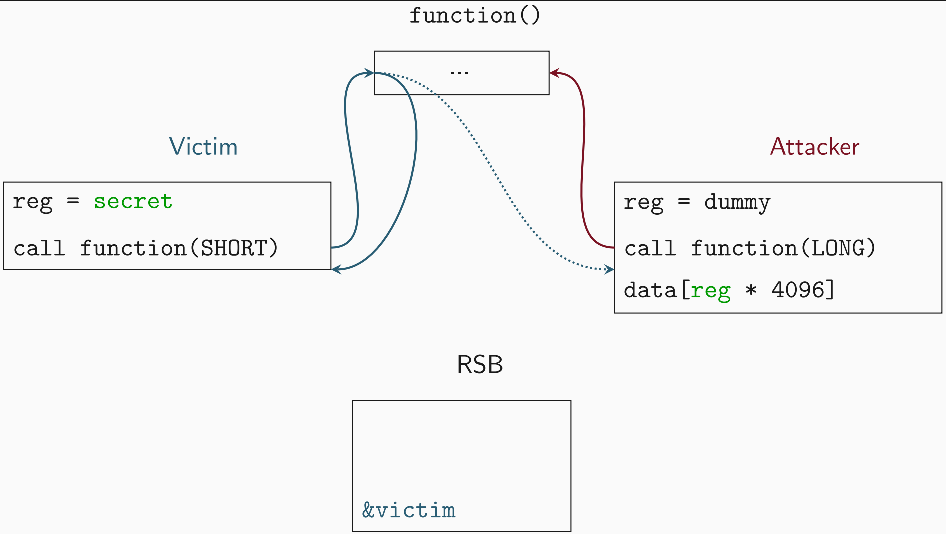
# Spectre-RSB (Variant 5) [3, 5]



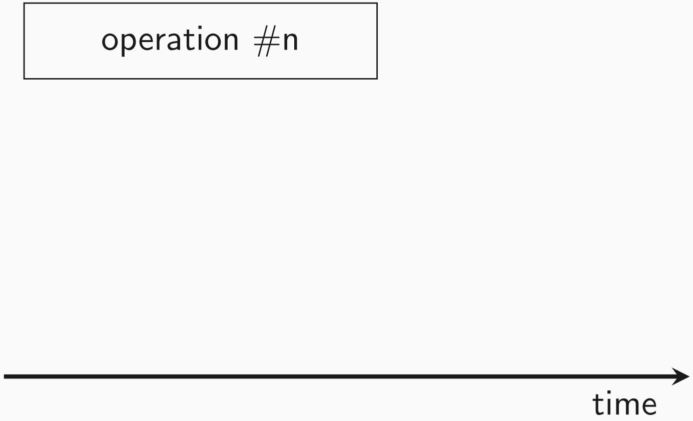
```
reg = secret
call function(SHORT)
```

```
reg = dummy
call function(LONG)
data[reg * 4096]
```

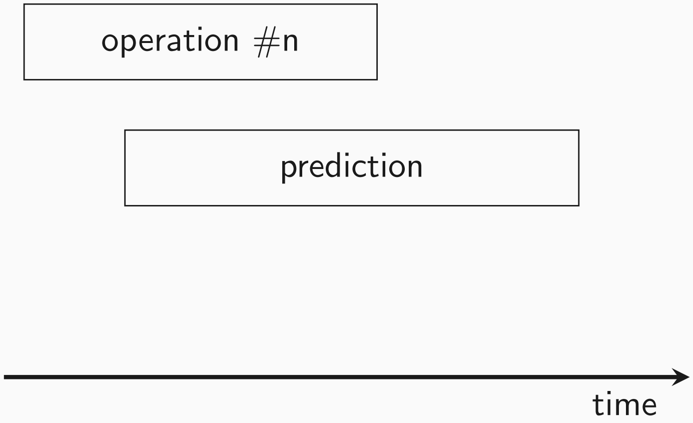
## Spectre-RSB (Variant 5) [3, 5]



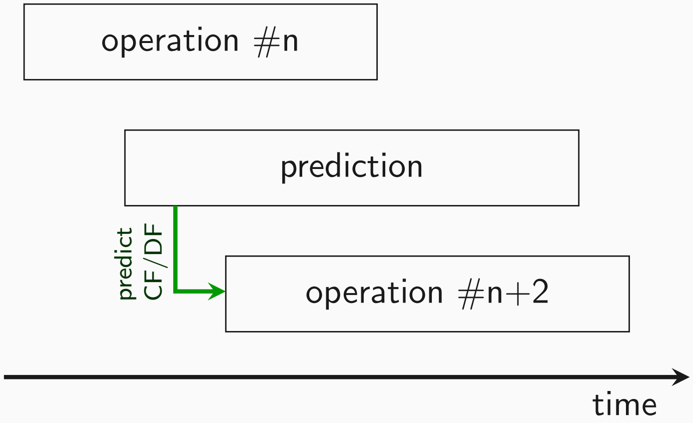
## Spectre Root Cause



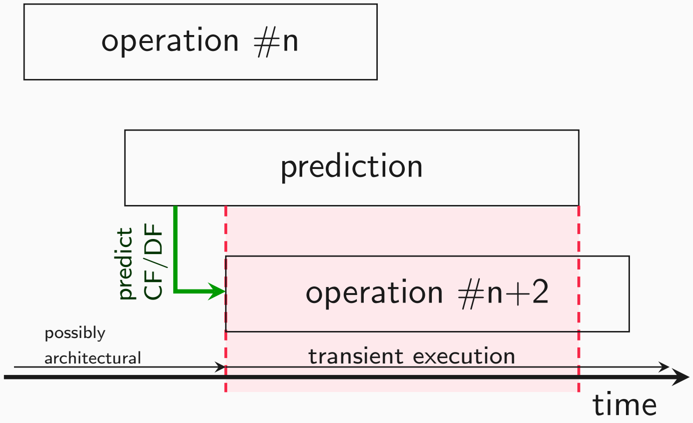
## Spectre Root Cause



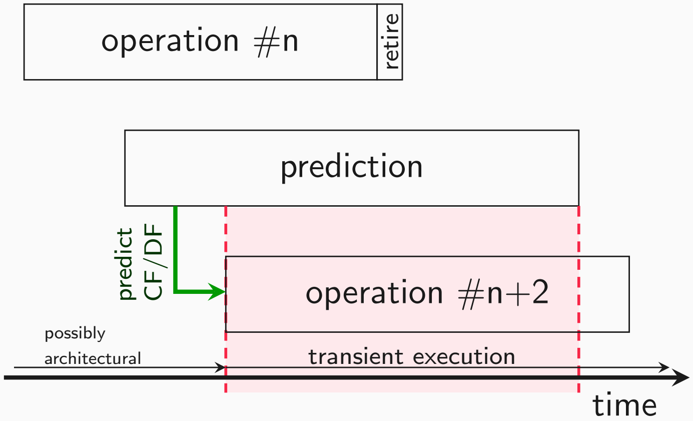
## Spectre Root Cause



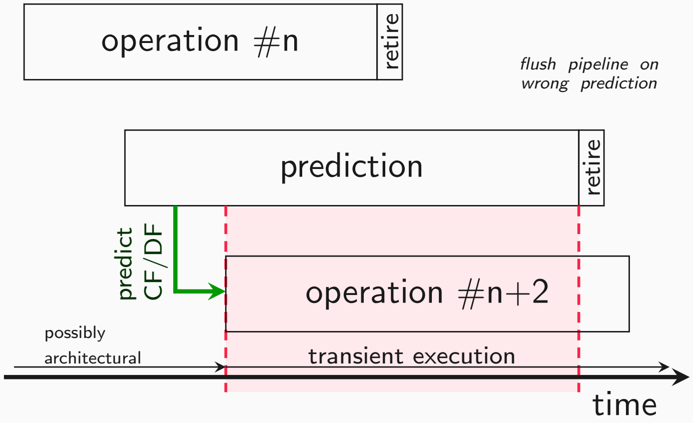
## Spectre Root Cause



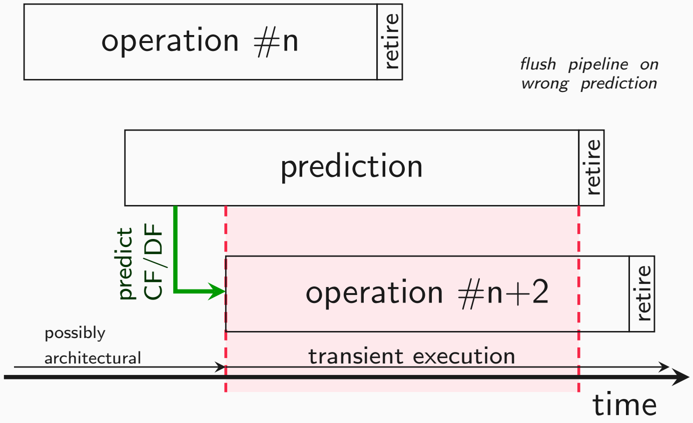
## Spectre Root Cause



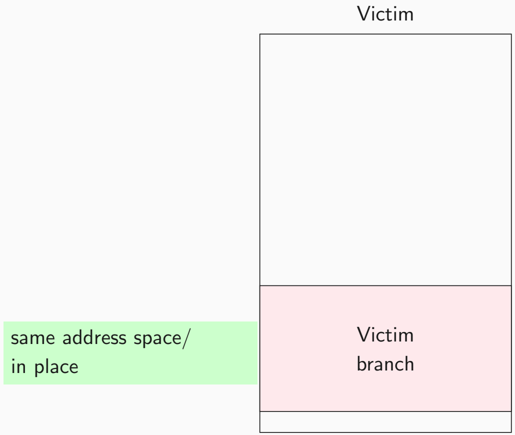
# Spectre Root Cause



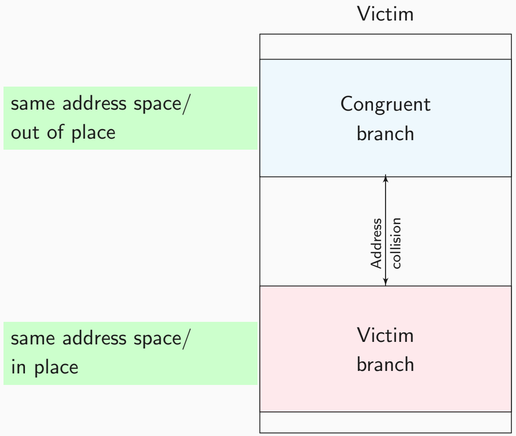
## Spectre Root Cause



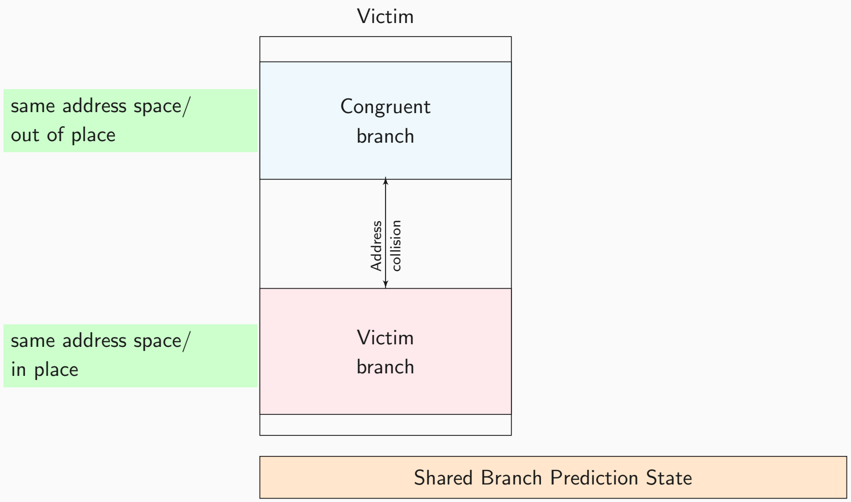
## Mistraining Location



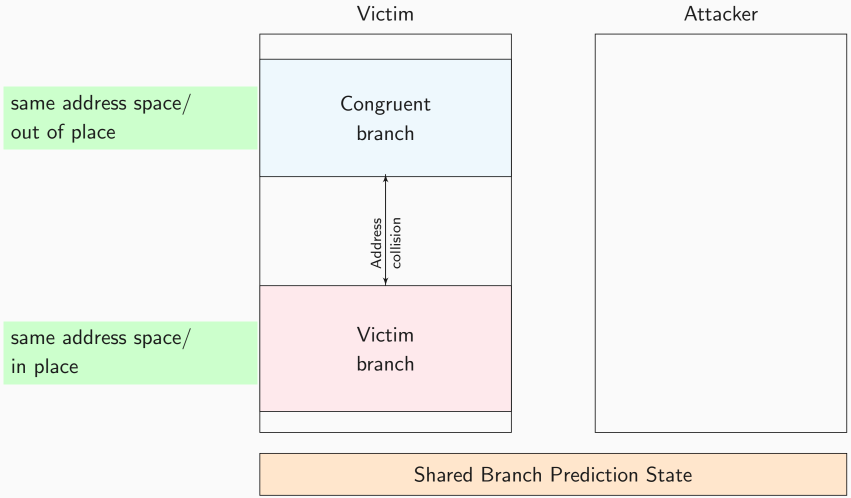
## Mistraining Location



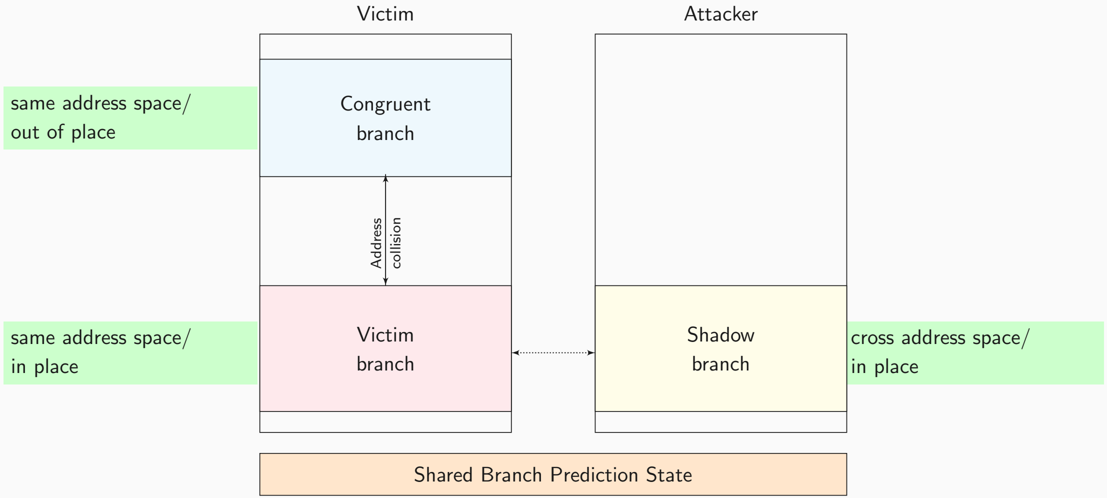
# Mistraining Location


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

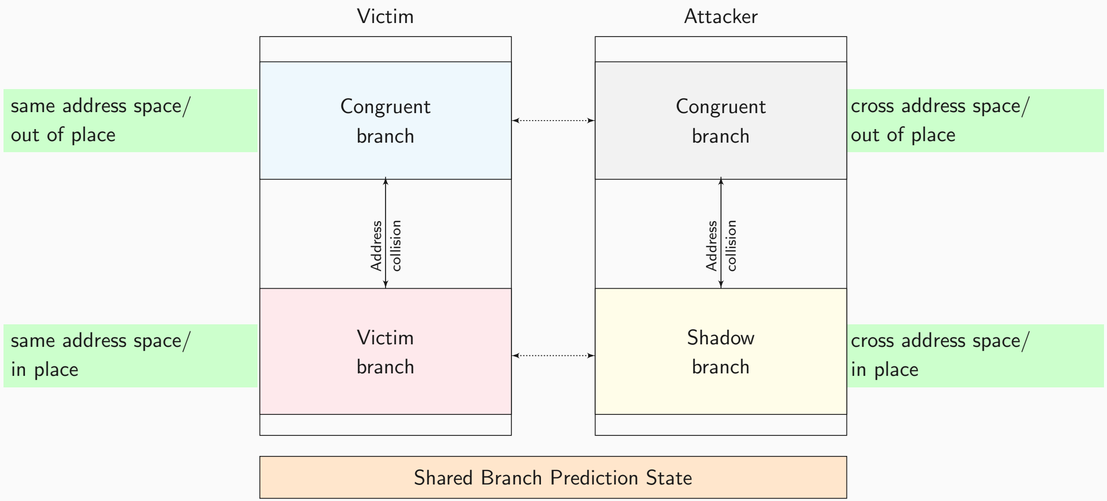
# Mistraining Location


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

# Mistraining Location

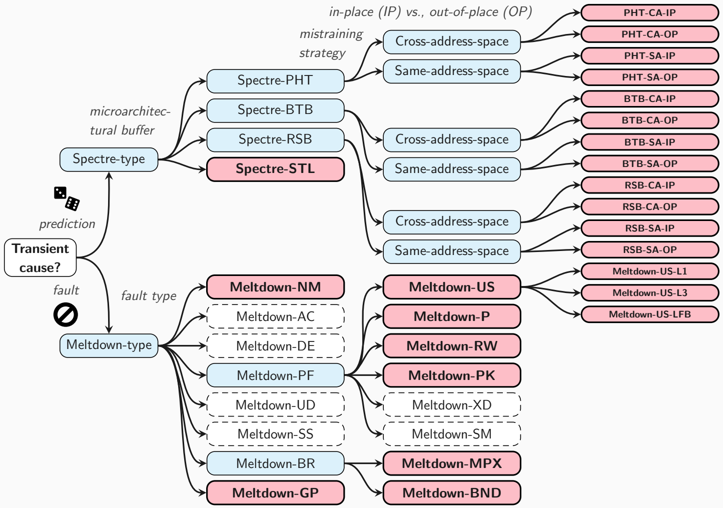

Victim
Attacker
same address space/ out of place
Congruent branch
Address collision
same address space/ in place
Victim branch
Shadow branch
cross address space/ in place
Shared Branch Prediction State

# Mistraining Location

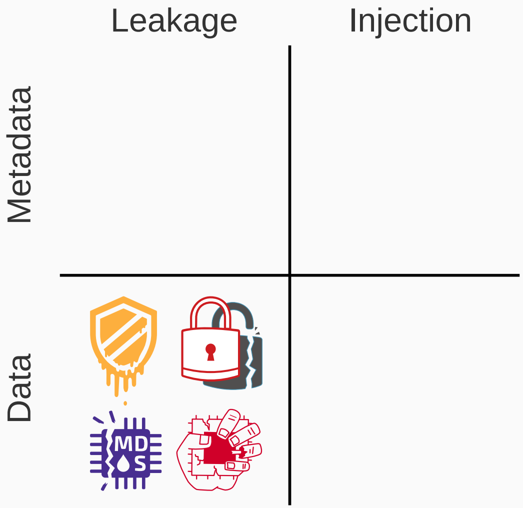


Victim
Attacker
same address space/ out of place
Congruent branch
Congruent branch
cross address space/ out of place
Address collision
Address collision
same address space/ in place
Victim branch
Shadow branch
cross address space/ in place
Shared Branch Prediction State

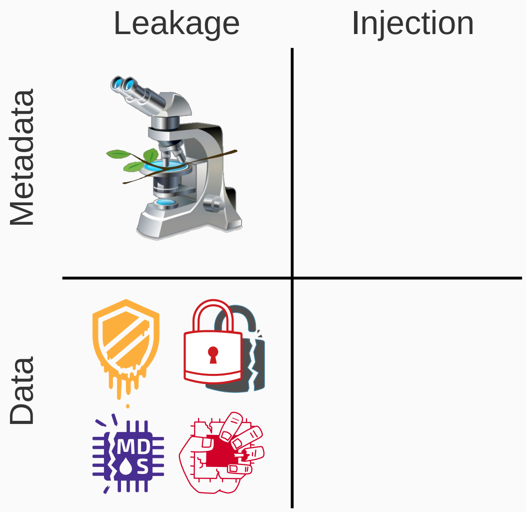
# Transient Execution Attacks



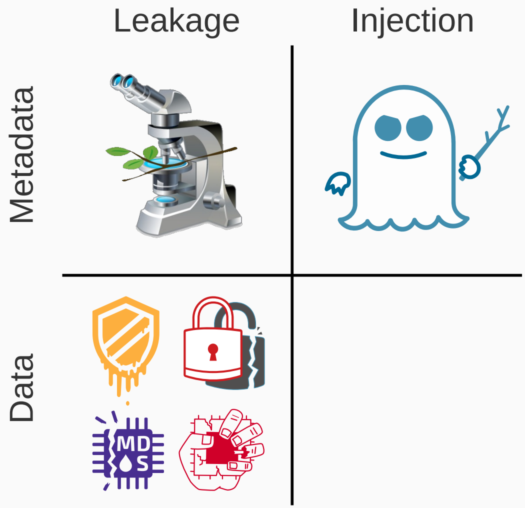
## Transient Execution Attacks



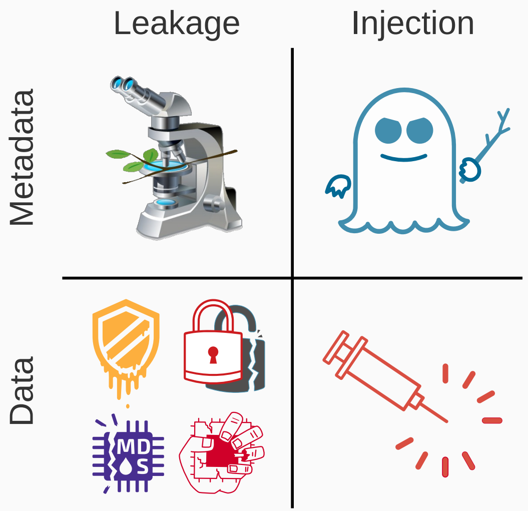
# Transient Execution Attacks



## Transient Execution Attacks



# Transient Execution Attacks



## Learn from it

We have ignored software side-channels for many many years:

## Learn from it

We have ignored software side-channels for many many years:

- attacks on crypto

## Learn from it

We have ignored software side-channels for many many years:

- attacks on crypto $\rightarrow$ “software should be fixed”

## Learn from it

We have ignored software side-channels for many many years:

- attacks on crypto → "software should be fixed"
- attacks on ASLR

## Learn from it

We have ignored software side-channels for many many years:

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

## Learn from it

We have ignored software side-channels for many many years:

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

## Learn from it

We have ignored software side-channels for many many years:

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

## Learn from it

We have ignored software side-channels for many many years:

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

→ for years we solely optimized for performance

## When you read the manuals...

After learning about a side channel you realize:

## When you read the manuals...

After learning about a side channel you realize:

- the side channels were documented in the Intel manual

## When you read the manuals...

After learning about a side channel you realize:

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

## Conclusion

- Underestimated microarchitectural attacks for a long time

## Conclusion

- Underestimated microarchitectural attacks for a long time
- Meltdown, Spectre and Foreshadow exploit performance optimizations
    - Allow to leak arbitrary memory

## Conclusion

- Underestimated microarchitectural attacks for a long time
- Meltdown, Spectre and Foreshadow exploit performance optimizations
  - Allow to leak arbitrary memory
- CPUs are deterministic - there is no noise

# Side-Channel Security

Chapter 4: Transient-Execution Attacks - Meltdown and Spectre

**Lukas Giner**

March 21, 2024

Graz University of Technology

## References i

[1] Jann Horn. *speculative execution, variant 4: speculative store bypass*. 2018.

[2] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. "Spectre Attacks: Exploiting Speculative Execution". In: *S&P*. 2019.

[3] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. "Spectre Returns! Speculation Attacks using the Return Stack Buffer". In: *WOOT*. 2018.

[4] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. "Meltdown: Reading Kernel Memory from User Space". In: *USENIX Security*. 2018.

# References ii

[5] G. Maisuradze and C. Rossow. “ret2spec: Speculative Execution Using Return Stack Buffers”. In: *CCS*. 2018.

[6] Martin Schwarzl, Thomas Schuster, Michael Schwarz, and Daniel Gruss. “Speculative Dereferencing of Registers: Reviving Foreshadow”. In: *arXiv:2008.02307* (2020).

[7] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. *Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution*. 2018. URL: `https://foreshadowattack.eu/foreshadow-NG.pdf`.