Transient Execution Attacks

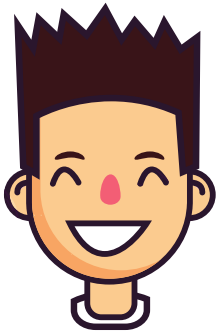
Exploiting the CPU's Microarchitecture

Michael Schwarz (@misc0110), Moritz Lipp (@mlqxyz)

October 7, 2019

Graz University of Technology

Who am I?



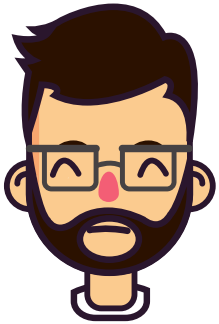
Michael Schwarz

PhD candidate @ Graz University of Technology

@misc0110

michael.schwarz@iaik.tugraz.at

Who am I?



Moritz Lipp

PhD candidate @ Graz University of Technology

@mlqxyz

mail@mlq.me

Let's Read Kernel Memory from User Space!

- 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);
```

- Compile and run

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

- Compile and run

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

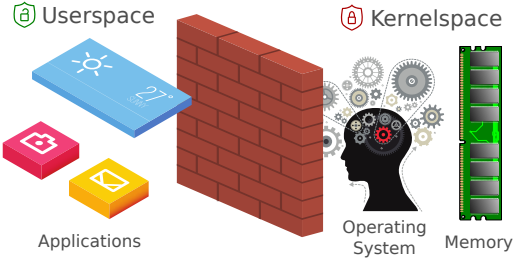
- Kernel addresses are of course not accessible

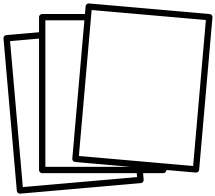
- Compile and run

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

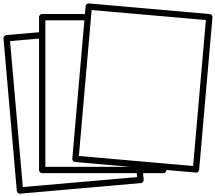
- Kernel addresses are of course not accessible
- Any invalid access throws an exception → segmentation fault

- Kernel is isolated from user space
- Isolation: combination of hardware and software
- Applications cannot access kernel
- Well-defined interface → system calls

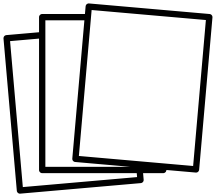




- Virtual address spaces isolate processes

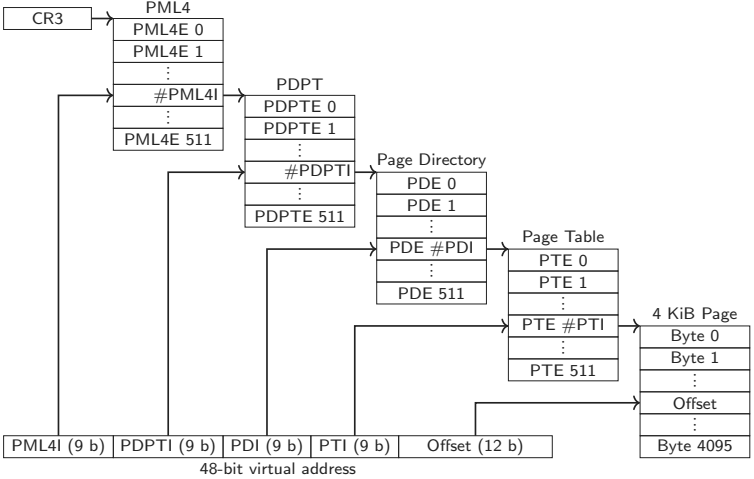


- Virtual address spaces isolate processes
- Physical memory organized in page frames



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

Address Translation on x86-64

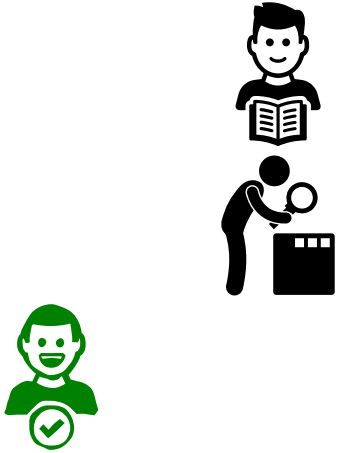


- User/Supervisor bit defines in which privilege level the page can be accessed

Loading an address

Loading an address

Loading an address



Loading an address

Loading an address

Loading an address

- Catch the segmentation fault!

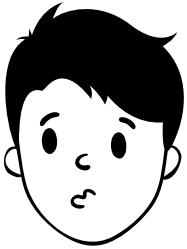
- Catch the segmentation fault!

→ Install a signal handler

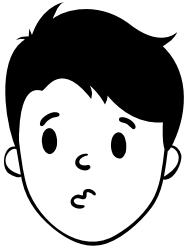
- Catch the segmentation fault!

→ Install a signal handler

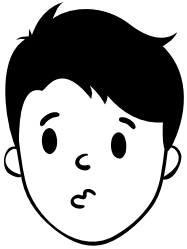
- On exception → jump back and continue



- Still no kernel memory



- Still no kernel memory
- Privilege checks seem to work



- Still no kernel memory
- Privilege checks seem to work
- Back to the drawing board

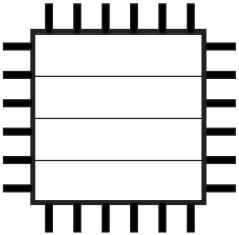
- We cannot see architectural changes

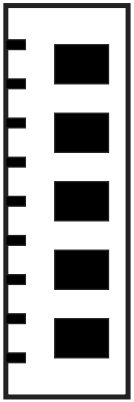
- We cannot see architectural changes
- What about the CPU internals?

- We cannot see architectural changes
- What about the CPU internals?

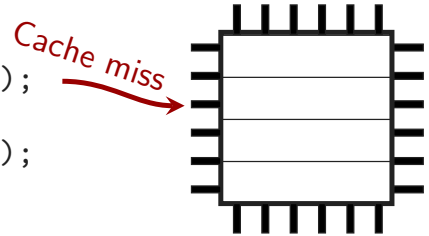
→ The microarchitectural state

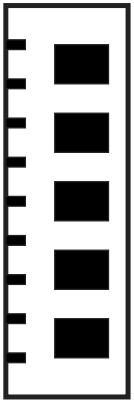
```
printf("%d", i);

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



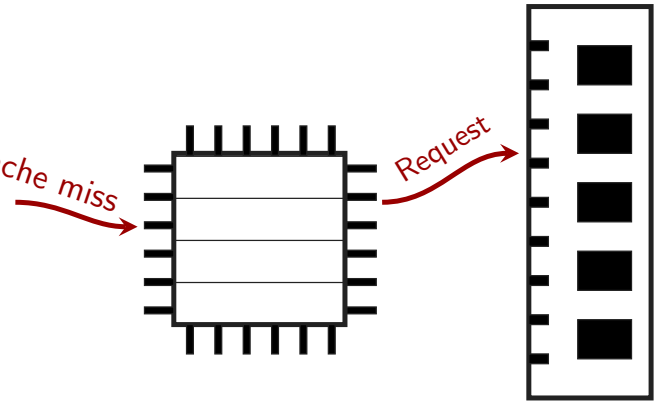


```
printf("%d", i);

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

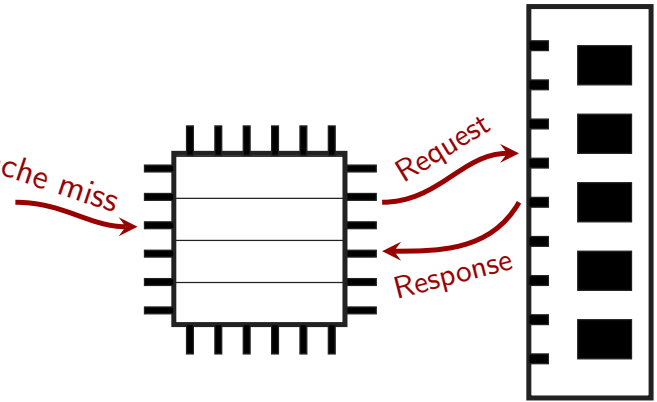




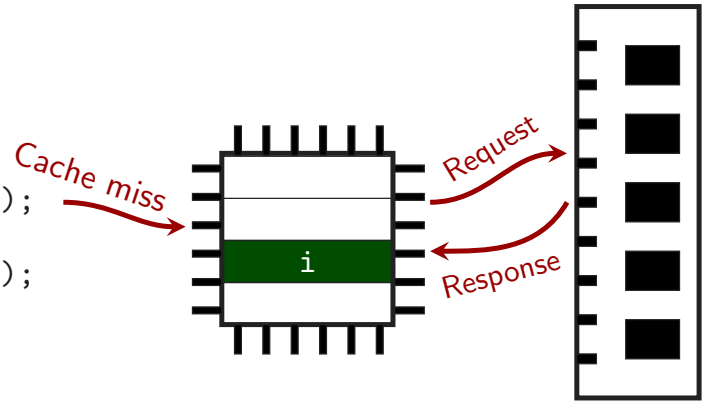
```
printf("%d", i);

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

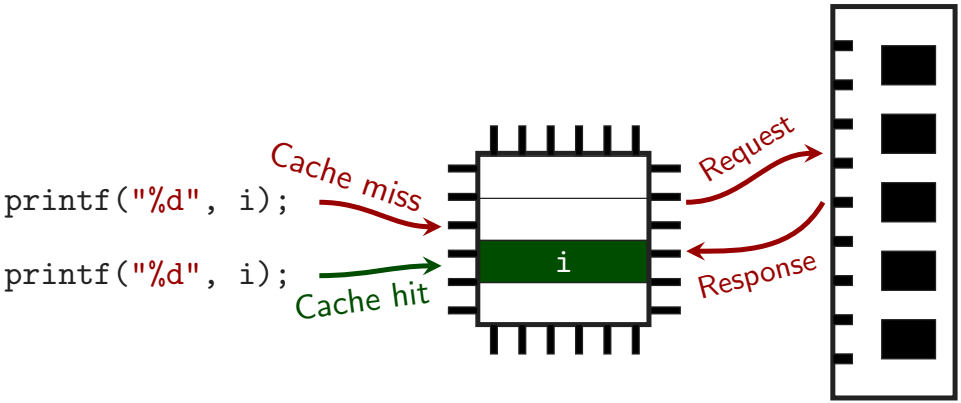


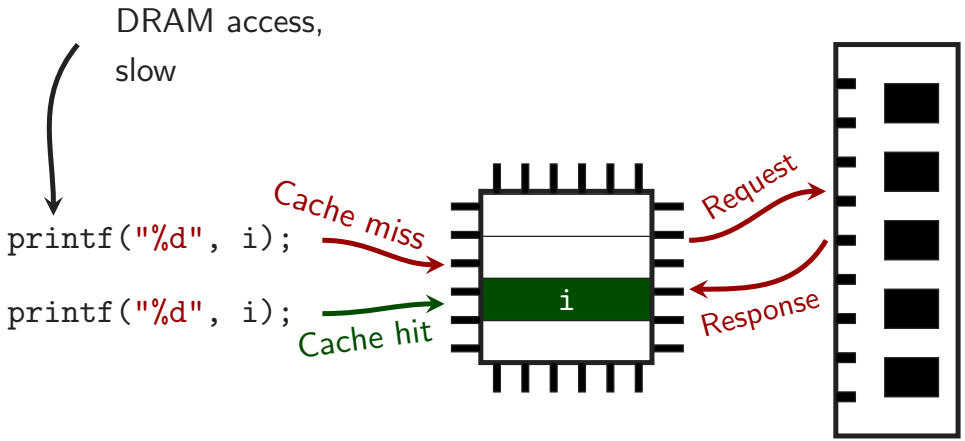
```
printf("%d", i);

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

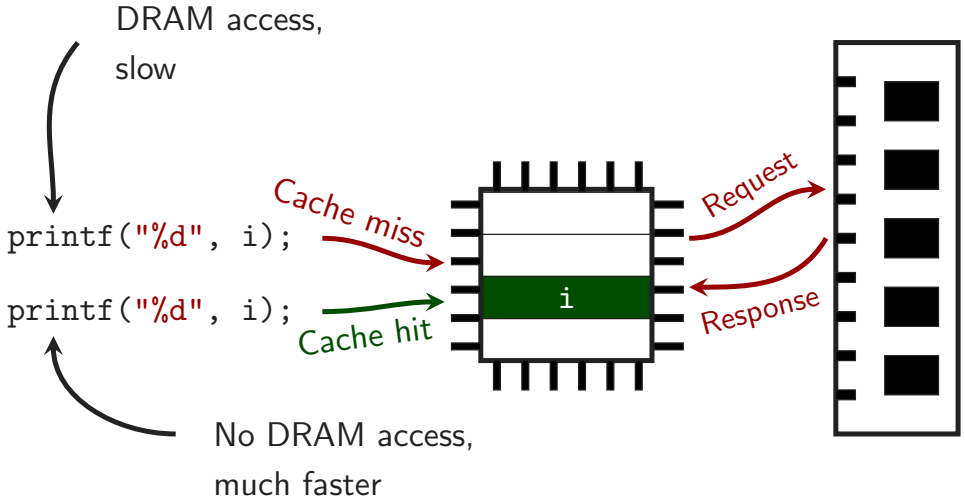


```
printf("%d", i);

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

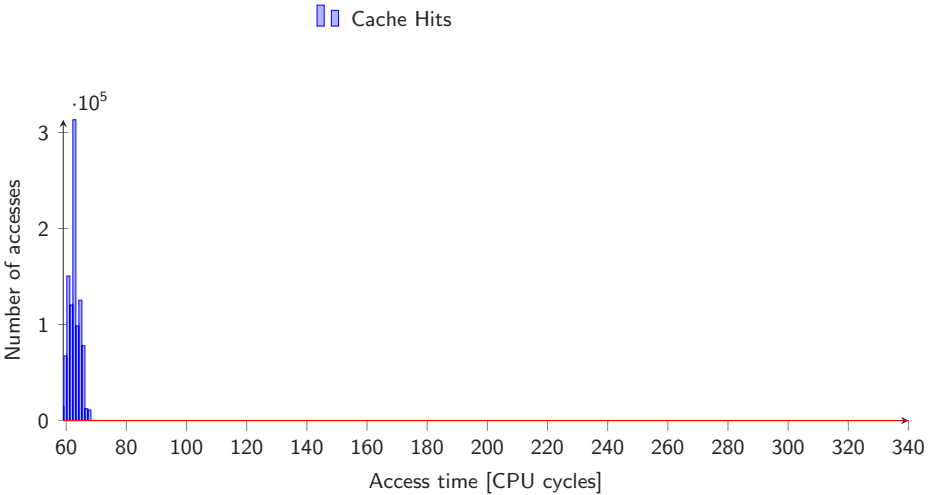



printf("%d", i);
printf("%d", i);
Cache miss
Cache hit
i
Request
Response

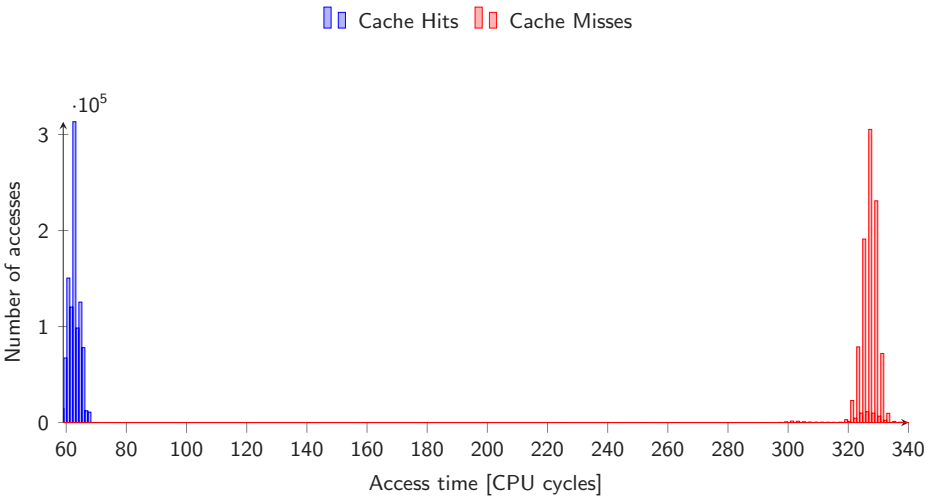

DRAM access,
slow
printf("%d", i);
printf("%d", i);
Cache miss
Cache hit
i
Request
Response

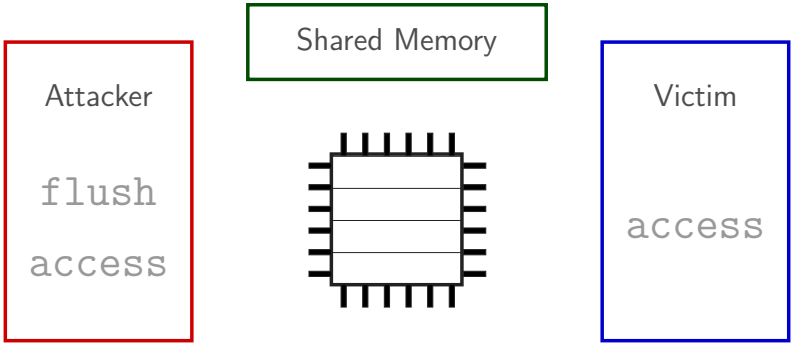

DRAM access,
slow
printf("%d", i);
printf("%d", i);
Cache miss
Cache hit
i
Request
Response
No DRAM access,
much faster

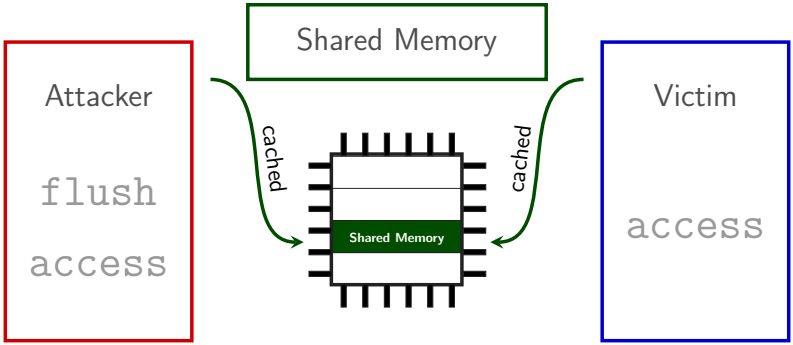
Caching speeds up Memory Accesses

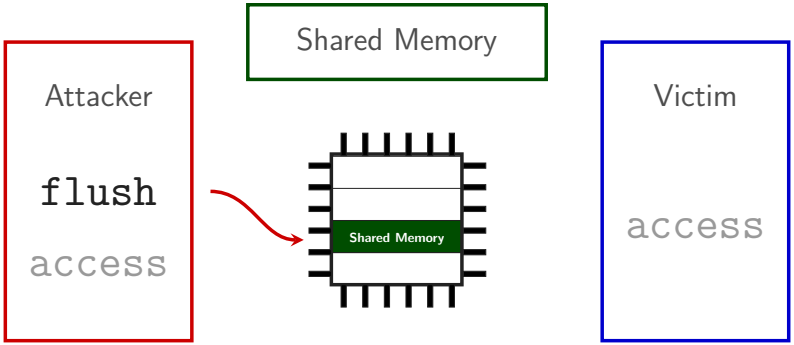


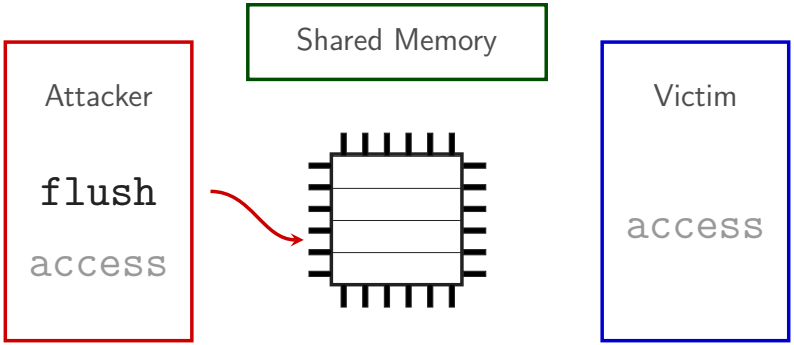
Caching speeds up Memory Accesses

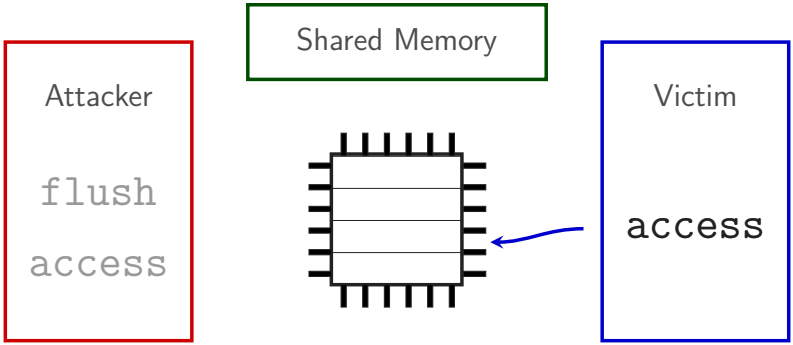


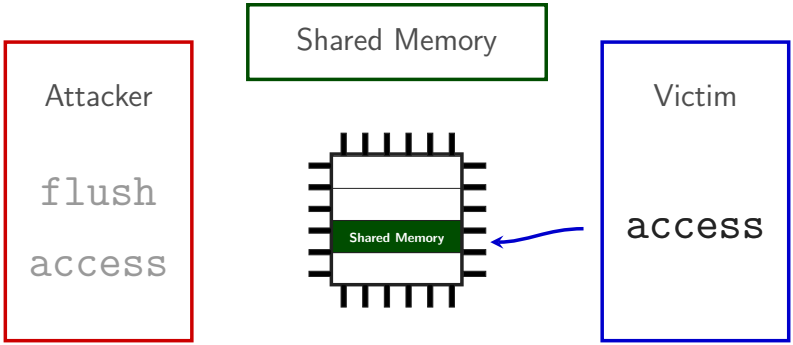

Shared Memory
Attacker
flush
access
Victim
access

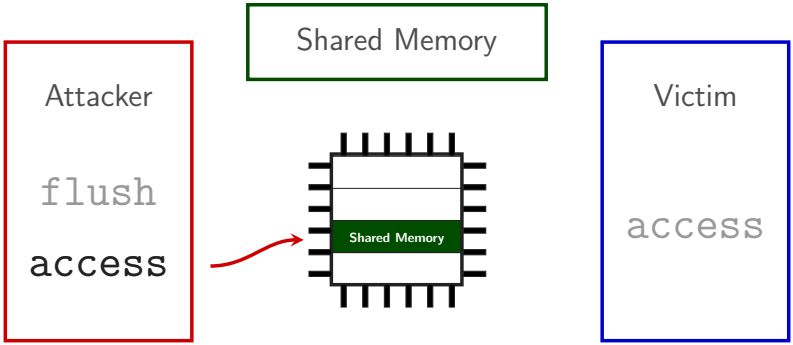

Shared Memory
Attacker
flush
access
cached
cached
Shared Memory
Victim
access

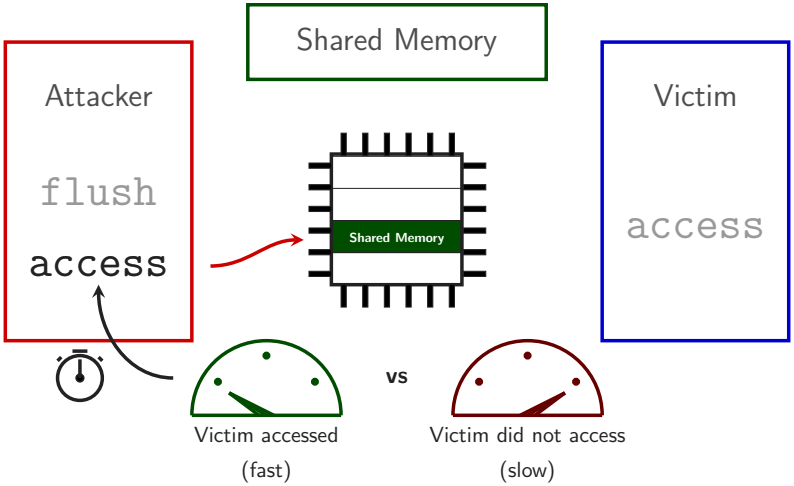

Shared Memory
Attacker
flush
access
Shared Memory
Victim
access


Shared Memory
Attacker
flush
access
Victim
access


Shared Memory
Attacker
flush
access
Victim
access


Shared Memory
Attacker
flush
access
Shared Memory
Victim
access


Shared Memory
Attacker
flush
access
Shared Memory
Victim
access


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

- Can we see illegal loads in the cache?

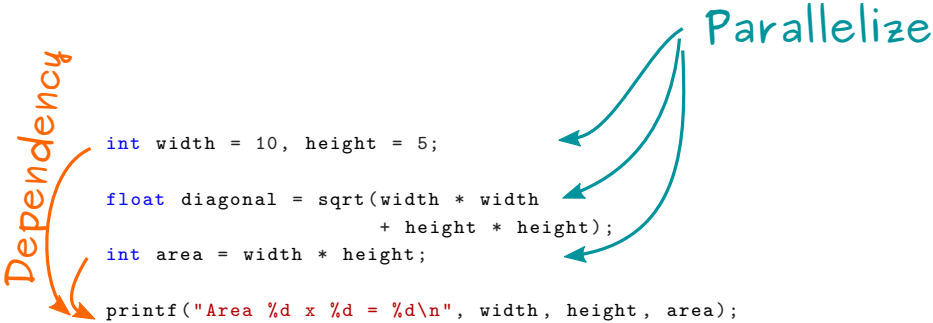
Illegal Loads and Caches

- Can we see illegal loads in the cache?
- What happens if the load cannot continue?

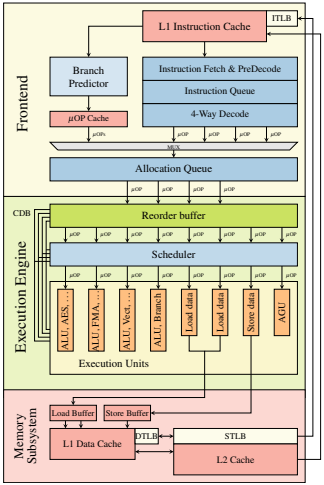
Out-of-order Execution

Out-of-order Execution

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

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

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

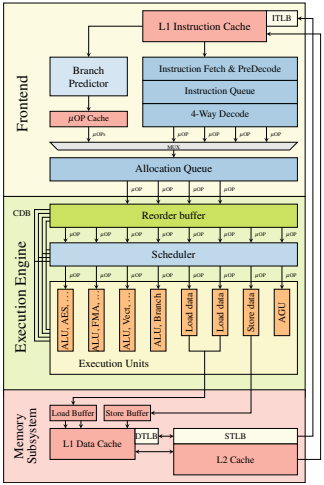

Parallelize
Dependency
int width = 10, height = 5;
float diagonal = sqrt(width * width
+ height * height);
int area = width * height;
printf("Area %d x %d = %d\n", width, height, area);

Out-of-order Execution



- Instructions are 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

- Adapted code

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

- Adapted code

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

- `volatile` because compiler was not happy

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

- Adapted code

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

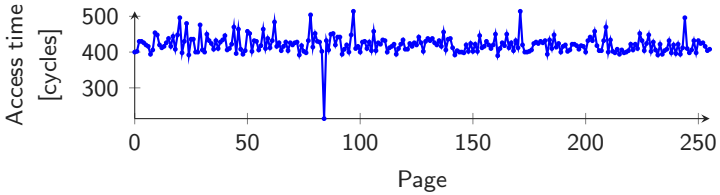
- `volatile` because compiler was not happy

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

- Static code analyzer is still not happy

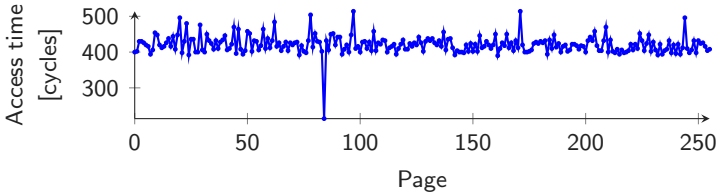
```
warning: Dereference of null pointer
         *(volatile char*)0;
```

- 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 in the cache

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

- 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

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

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

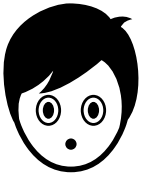
- Maybe there is no permission check in transient instructions...
- ...or it is only done when commiting them
- Add another layer of indirection to test

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

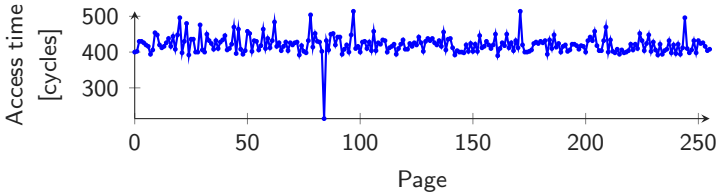
- Maybe there is no permission check in transient instructions...
- ...or it is only done when commiting them
- Add another layer of indirection to test

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

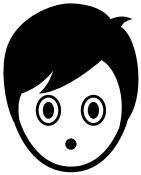
- Then check whether any part of `array` is cached



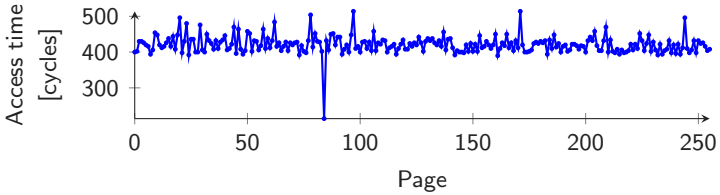
- Flush+Reload over all pages of the array



- 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

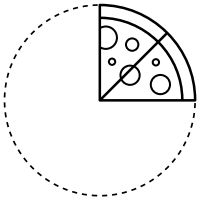
- Using out-of-order execution, we can read data at any address

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

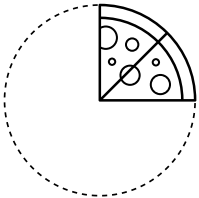
- Using out-of-order execution, we can read data at any address
- Privilege checks are sometimes too slow
- Allows to leak kernel memory

- 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

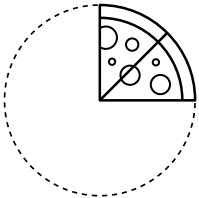
- Using out-of-order execution, we can read data at any address
- Privilege checks are sometimes too slow
- Allows to leak kernel memory
- Entire physical memory is typically also accessible in kernel address space
- Works on Intel CPUs and ARM Cortex-A75



- Assumed Meltdown can one only read data from the L1

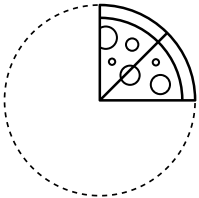


- Assumed Meltdown can one only read data from the L1
- Leakage from L3 or memory is possible, just slower

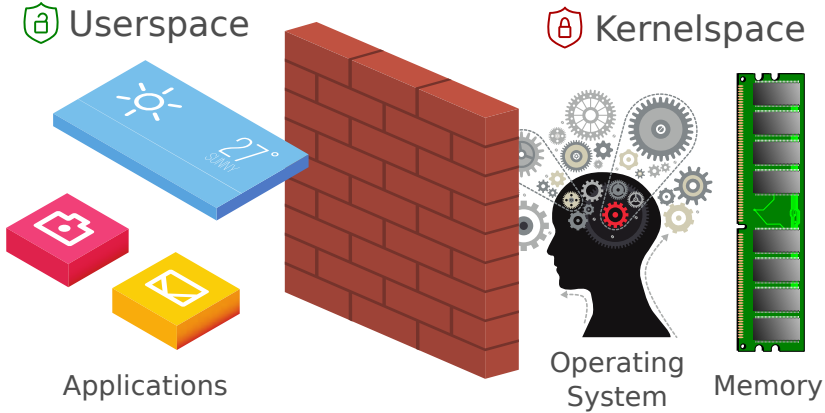


- Assumed Meltdown can one only read data from the L1
- Leakage from L3 or memory is possible, just slower
- Even leakage of UC (uncachable) memory regions...

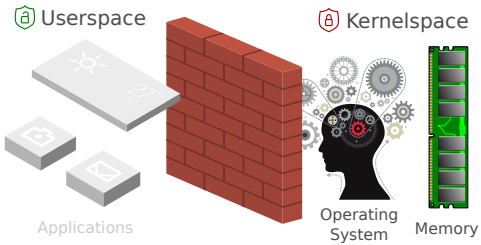
Uncached and uncachable memory



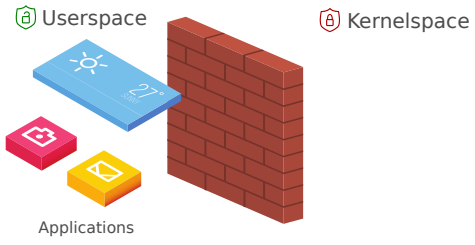
- Assumed Meltdown can one only read data from the L1
- Leakage from L3 or memory is possible, just slower
- Even leakage of UC (uncachable) memory regions...
  - ...if other hyperthread (legally) accesses the data


Userspace
Kernelspace
27°
Applications
Operating System
Memory

Kernel View



User View

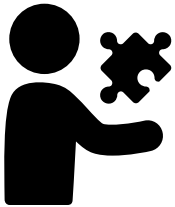


- **Linux**: Kernel Page-table Isolation (KPTI)

- **Linux**: Kernel Page-table Isolation (KPTI)
- **Apple**: Released updates

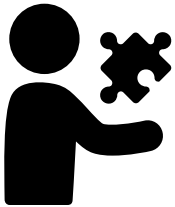
- **Linux**: Kernel Page-table Isolation (KPTI)
- **Apple**: Released updates
- **Windows**: Kernel Virtual Address (KVA) Shadow

Problem Solved?



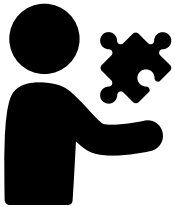
- Meltdown fully mitigated in software

Problem Solved?



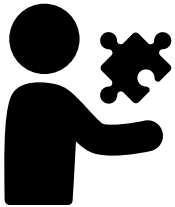
- Meltdown fully mitigated in software
- Problem seemed to be solved

Problem Solved?



- Meltdown fully mitigated in software
- Problem seemed to be solved
- No attack surface left

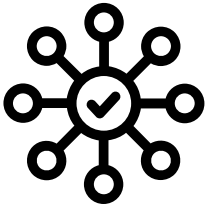
Problem Solved?



- Meltdown fully mitigated in software
- Problem seemed to be solved
- No attack surface left
- That is what everyone thought

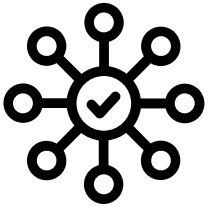

There are no bugs,
just happy little accidents

Generalization

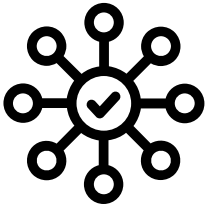


- Meltdown is a whole category of vulnerabilities

Generalization



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



- Meltdown is a whole category of vulnerabilities
- Not only the user-accessible check
- There are more bits...


P
RW
US
WT
UC
R
D
S
G
Ignored
Physical Page Number
Ignored
X

- Present bit is the next obvious bit

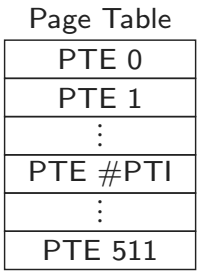
- An even worse bug → Foreshadow-NG/L1TF

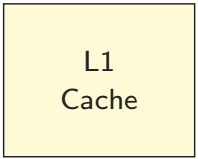
- An even worse bug → Foreshadow-NG/L1TF
- Exploitable from VMs

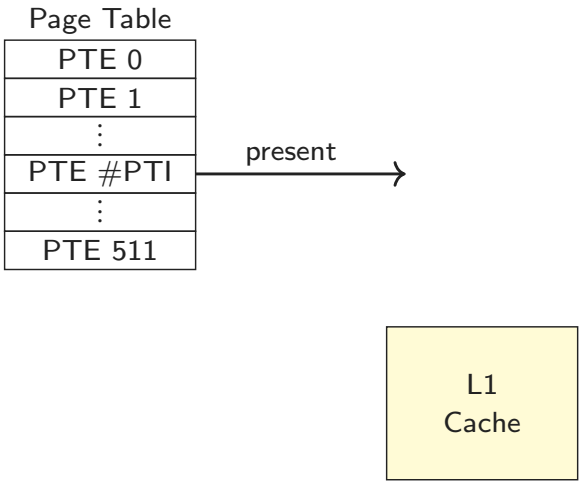
- An even worse bug → Foreshadow-NG/L1TF
- Exploitable from VMs
- Allows leaking data from the L1 cache

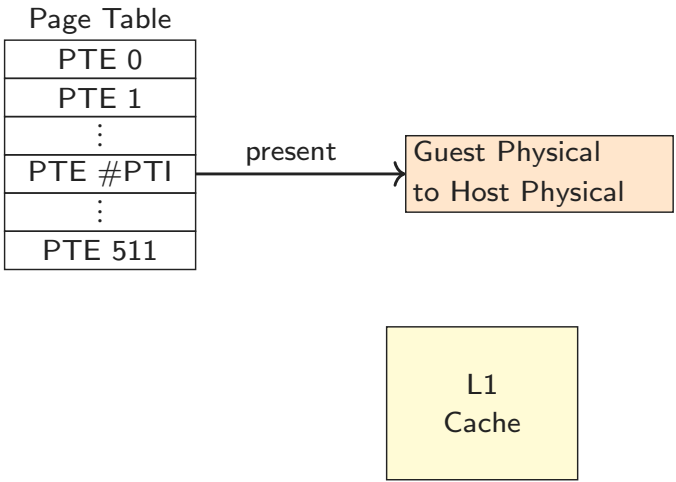
- An even worse bug → Foreshadow-NG/L1TF
- Exploitable from VMs
- Allows leaking data from the L1 cache
- Same mechanism as Meltdown

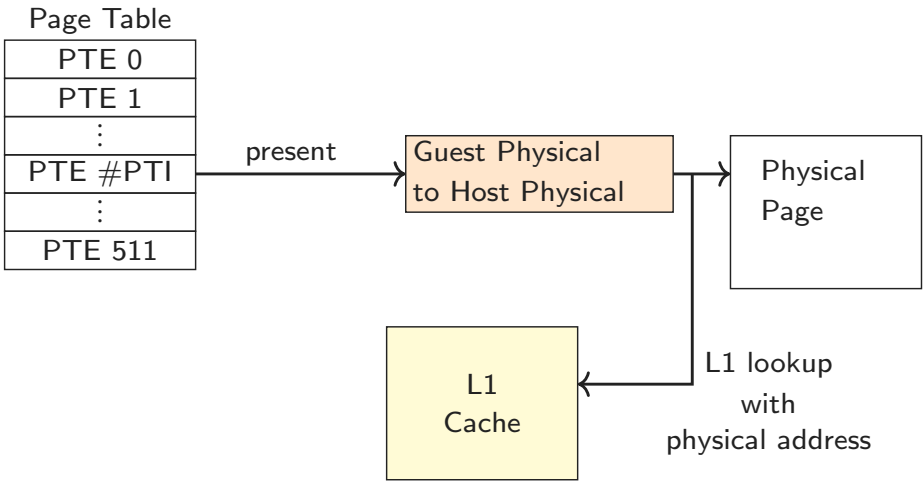
- An even worse bug → Foreshadow-NG/L1TF
- Exploitable from VMs
- Allows leaking data from the L1 cache
- Same mechanism as Meltdown
- Just a different bit in the PTE

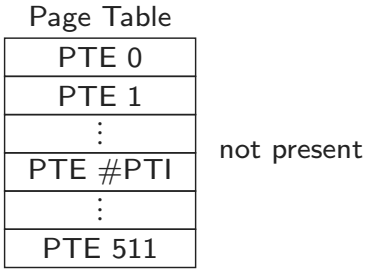

Page Table
PTE 0
PTE 1
PTE #PTI
PTE 511

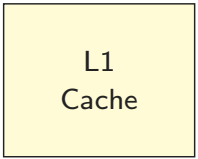

L1
Cache

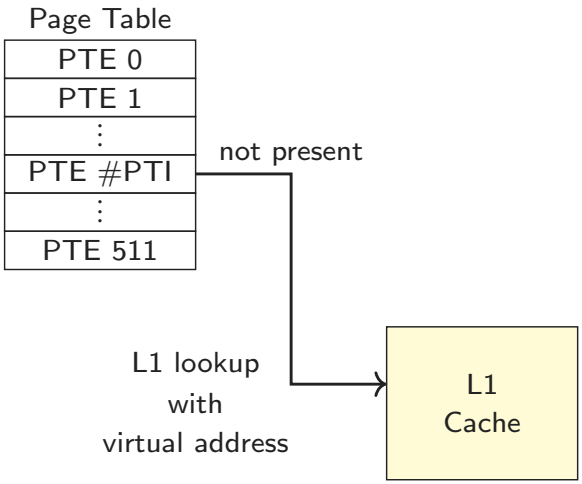

Page Table
PTE 0
PTE 1
PTE #PTI
PTE 511
present
L1
Cache


Page Table
PTE 0
PTE 1
PTE #PTI
PTE 511
present
Guest Physical
to Host Physical
L1
Cache

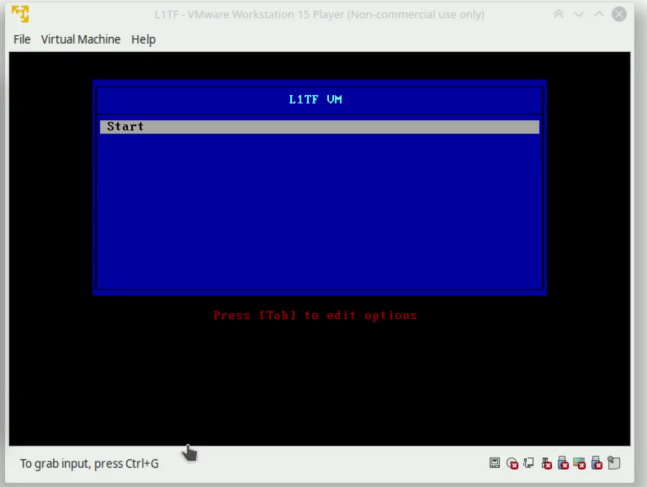

Page Table
PTE 0
PTE 1
PTE #PTI
PTE 511
present
Guest Physical
to Host Physical
Physical
Page
L1
Cache
L1 lookup
with
physical address


Page Table
PTE 0
PTE 1
PTE #PTI
PTE 511
not present


L1
Cache


Page Table
PTE 0
PTE 1
PTE #PTI
PTE 511
not present
L1 lookup
with
virtual address
L1
Cache


Research Manager
***
Unlock Research Manager
Unlock


L1TF - VMware Workstation 15 Player (Non-commercial use only)
File
Virtual Machine
Help
L1TF VM
Start
Press [Tab] to edit options
To grab input, press Ctrl+G

- KAISER/KPTI/KVA does not help

- KAISER/KPTI/KVA does not help
- Only software workarounds

- KAISER/KPTI/KVA does not help
- Only software workarounds
  → Flush L1 on VM entry

- KAISER/KPTI/KVA does not help
- Only software workarounds
  - → Flush L1 on VM entry
  - → Disable HyperThreading

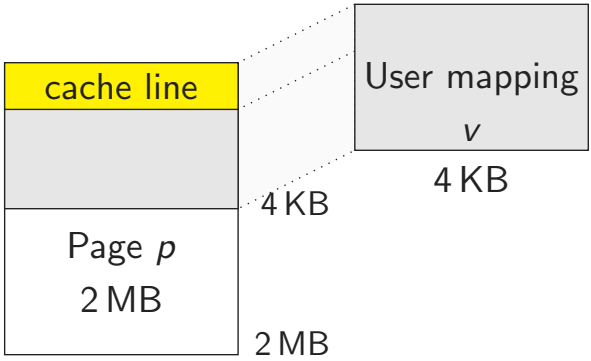
- KAISER/KPTI/KVA does not help
- Only software workarounds
  - → Flush L1 on VM entry
  - → Disable HyperThreading
- Workarounds might not be complete

Not cached?

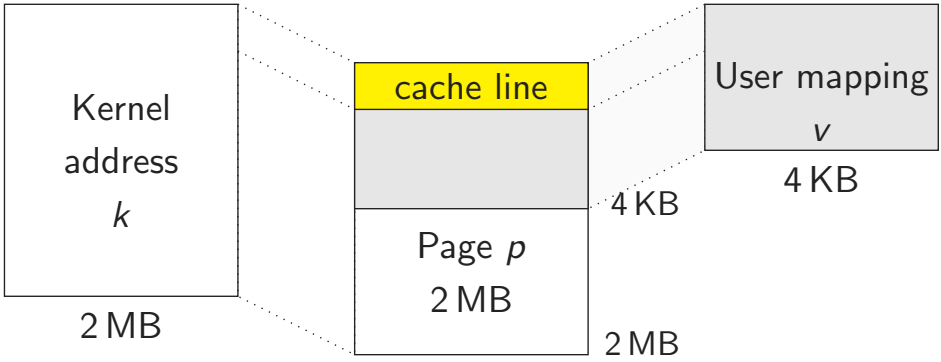
- What if the memory is not cached?

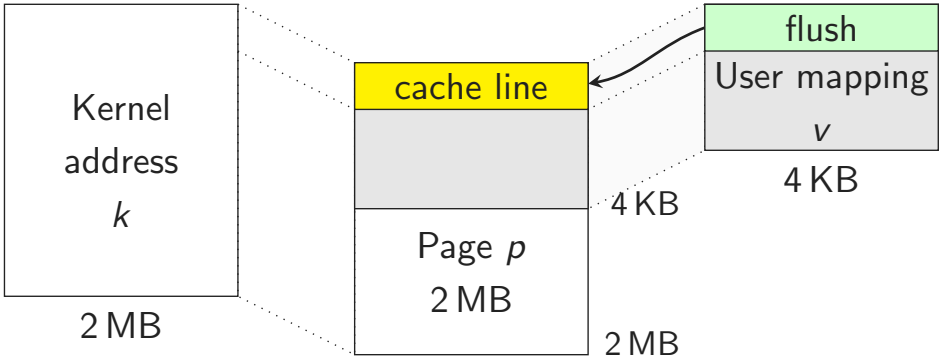
Not cached?

- What if the memory is not cached?
- No data $\rightarrow$ no leakage?

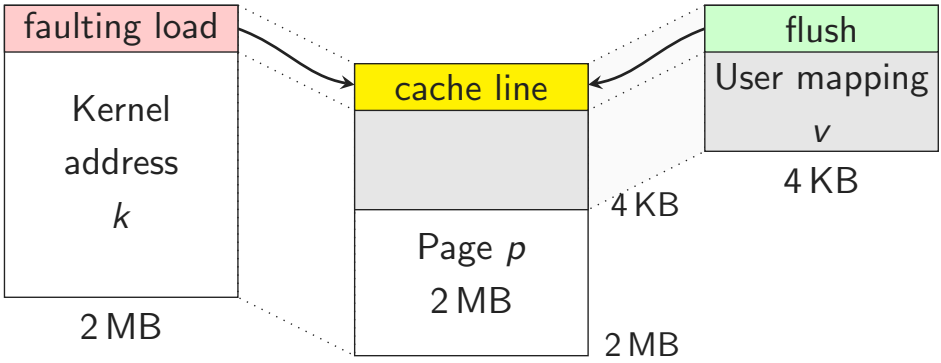

cache line
User mapping
v
4 KB
4 KB
Page p
2 MB
2 MB

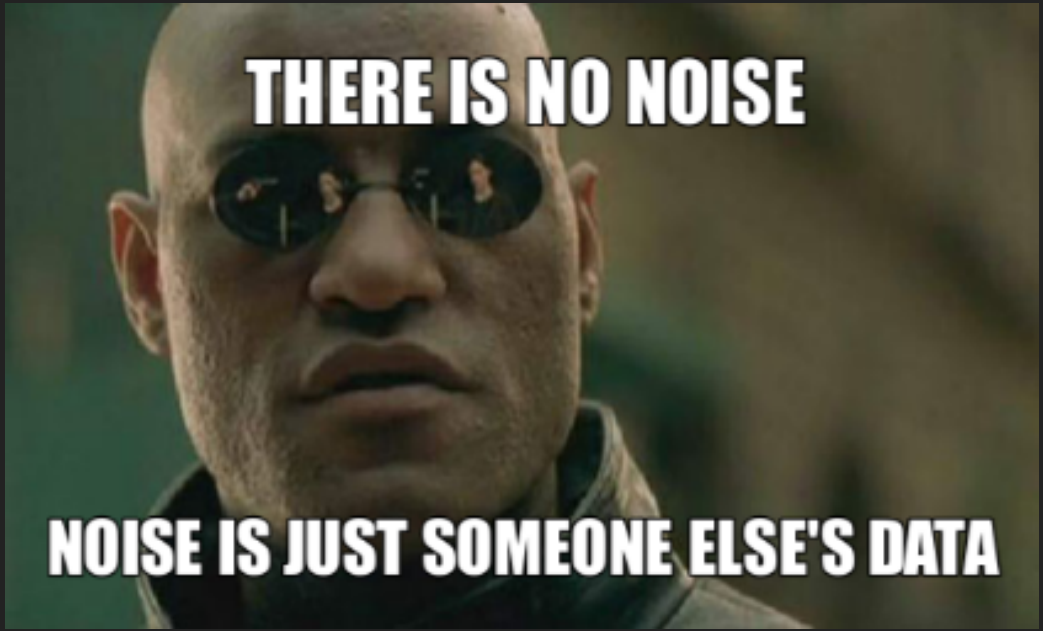
Flush the Target



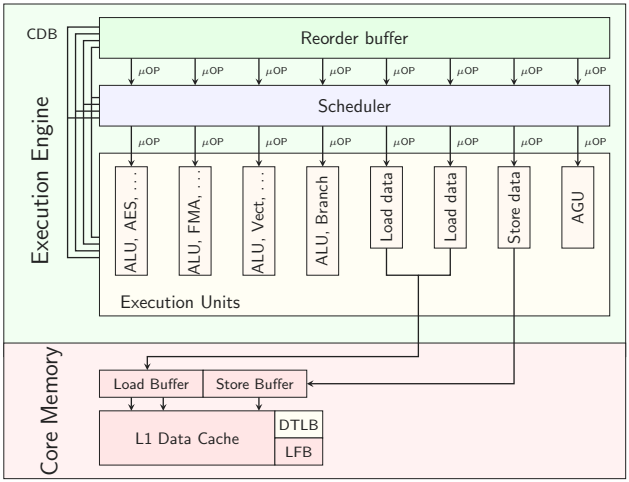

Kernel
address
k
2 MB
cache line
Page p
2 MB
4 KB
2 MB
flush
User mapping
v
4 KB

Flush the Target

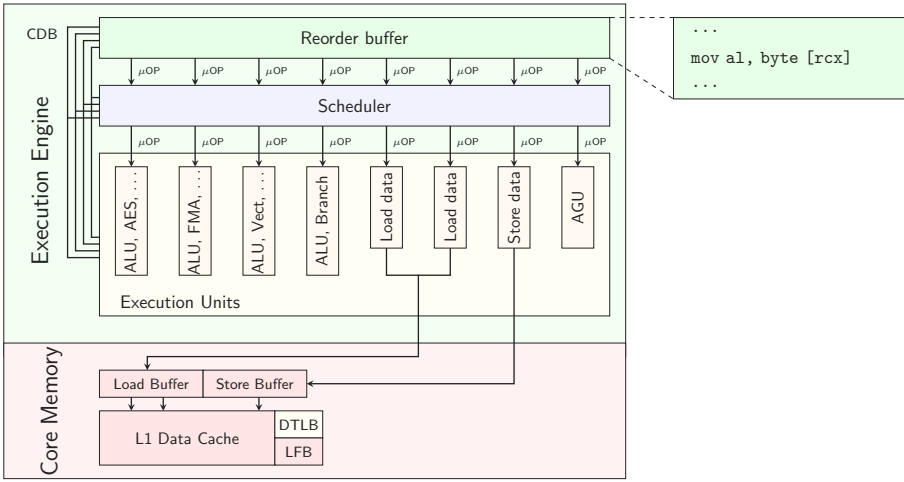


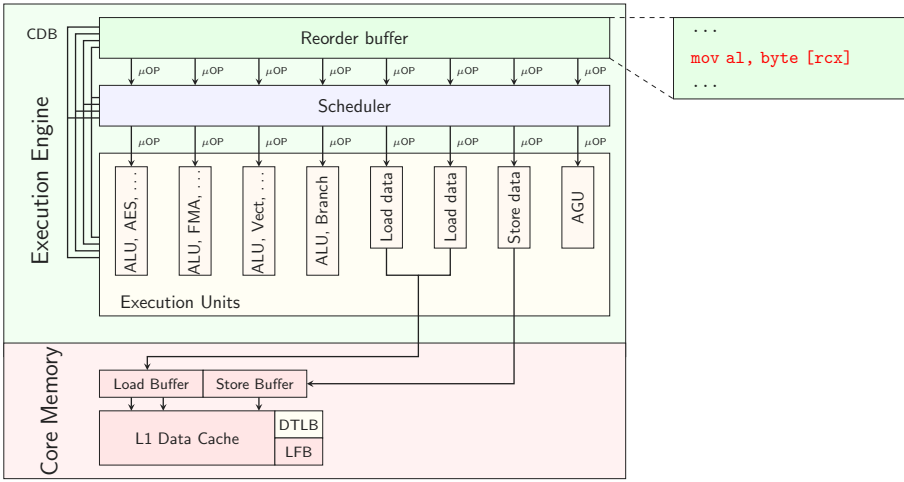

THERE IS NO NOISE
NOISE IS JUST SOMEONE ELSE'S DATA

Line-fill Buffers

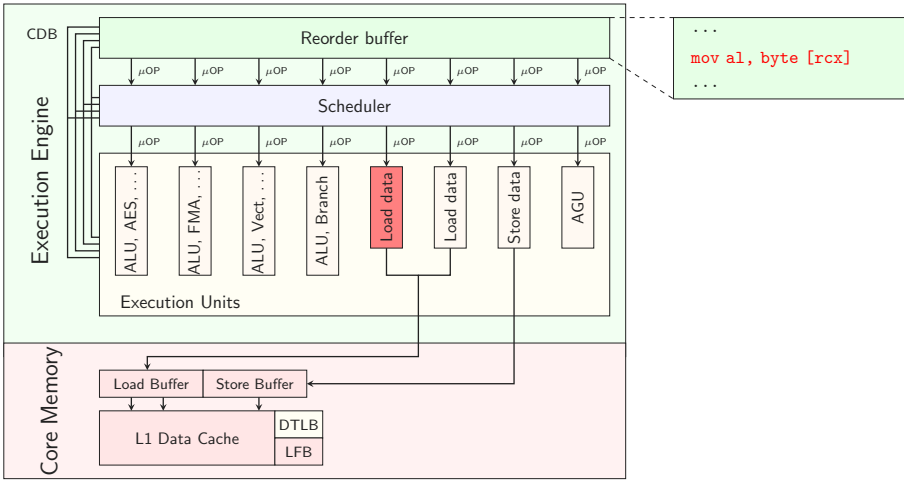


Line-fill Buffers

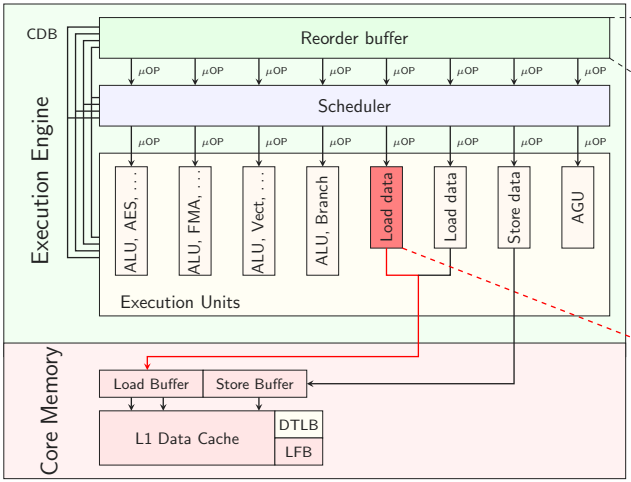



CDB
Reorder buffer
...
mov al, byte [rcx]
...
μOP
Scheduler
Execution Engine
ALU, AES, ...
ALU, FMA, ...
ALU, Vect, ...
ALU, Branch
Load data
Load data
Store data
AGU
Execution Units
Core Memory
Load Buffer
Store Buffer
L1 Data Cache
DTLB
LFB

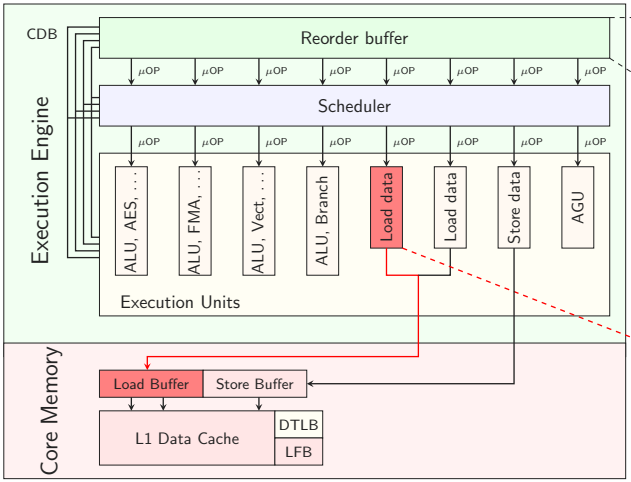
Line-fill Buffers



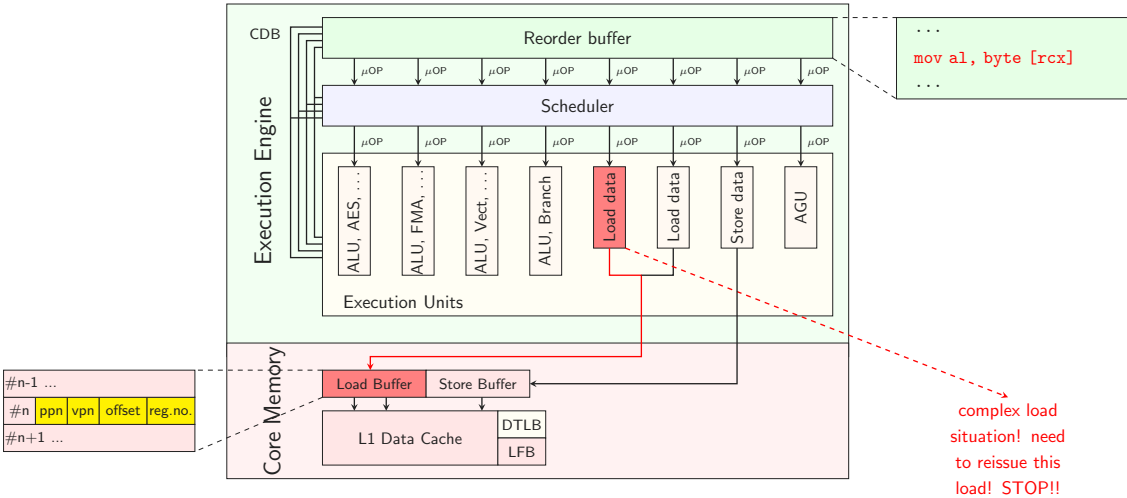
Line-fill Buffers



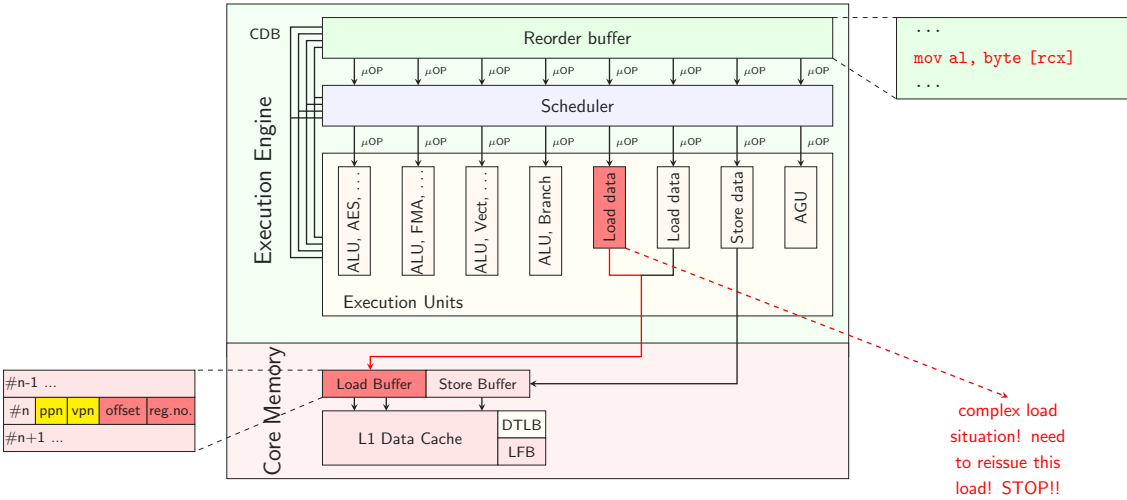
Line-fill Buffers



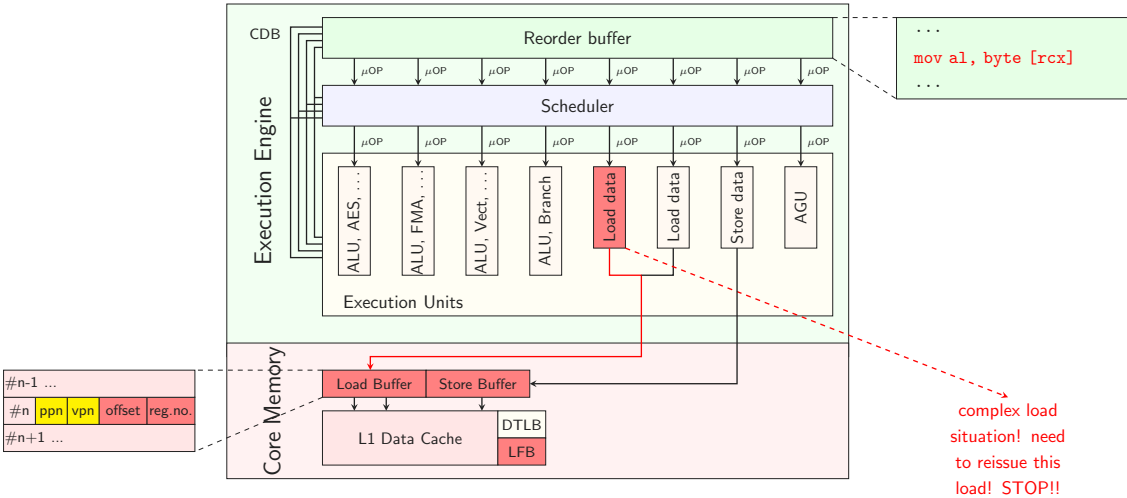
Line-fill Buffers



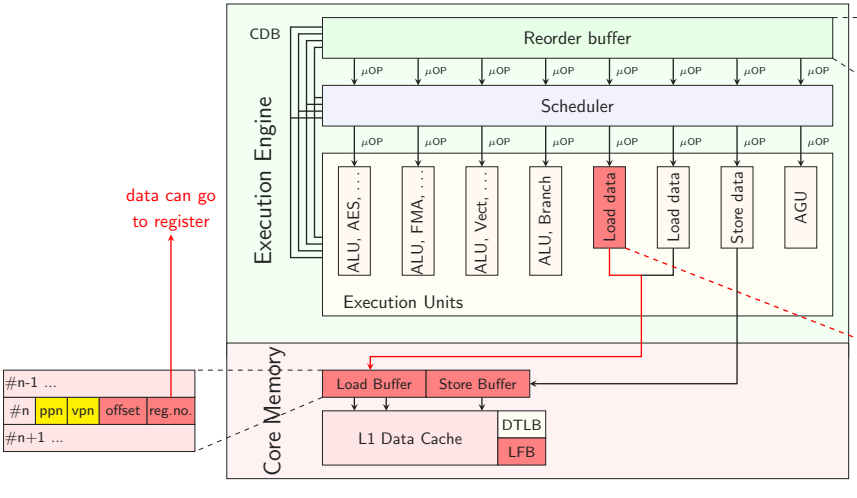
Line-fill Buffers



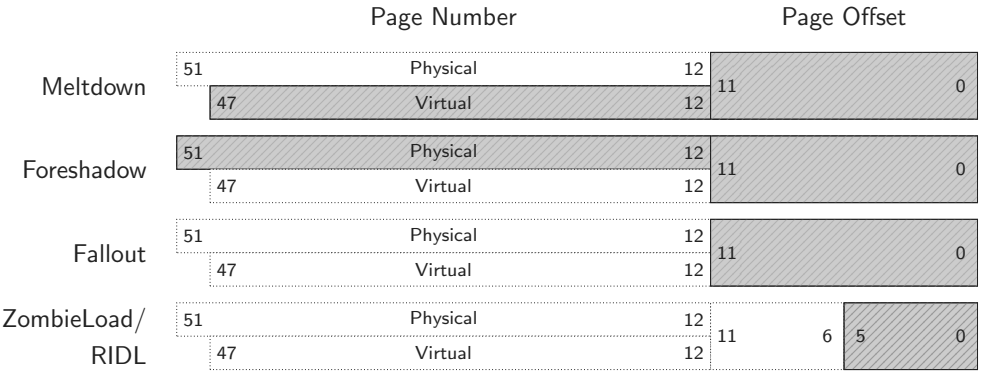
Line-fill Buffers

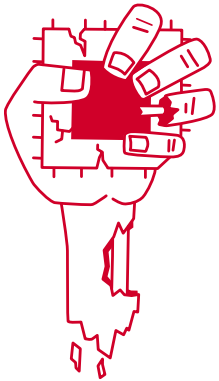


Line-fill Buffers

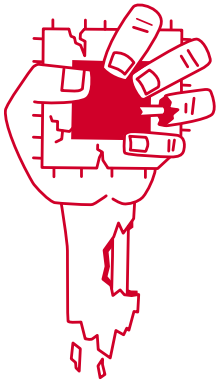


Data Sampling



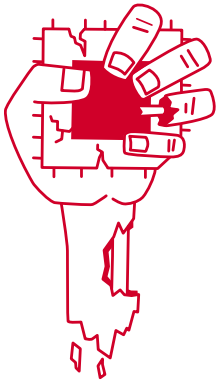


- Leaks from the fill buffer



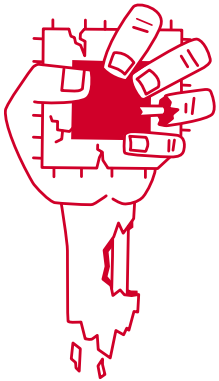
- Leaks from the fill buffer
- Crosses all privilege boundaries (Kernel, VM, SGX)

Latest Meltdown Variant: ZombieLoad



- Leaks from the fill buffer
- Crosses all privilege boundaries (Kernel, VM, SGX)
- Explored microcode assists as new type of faults

Latest Meltdown Variant: ZombieLoad



- Leaks from the fill buffer
- Crosses all privilege boundaries (Kernel, VM, SGX)
- Explored microcode assists as new type of faults
- Disadvantage: minimal control over leaked data


zombieload : zsh — Konsole <2>
File Edit View Bookmarks Settings Help
michael@hp /tmp/zombieload %
zombieload : zsh

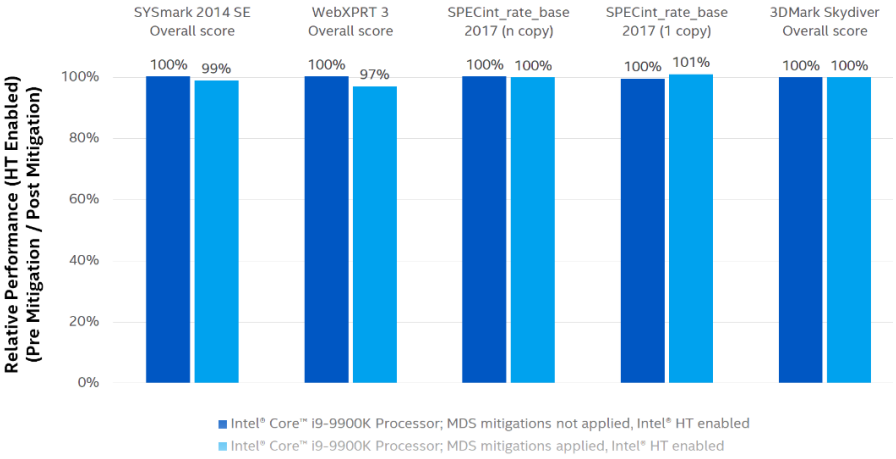
Mitigations?

- Disable HyperThreading

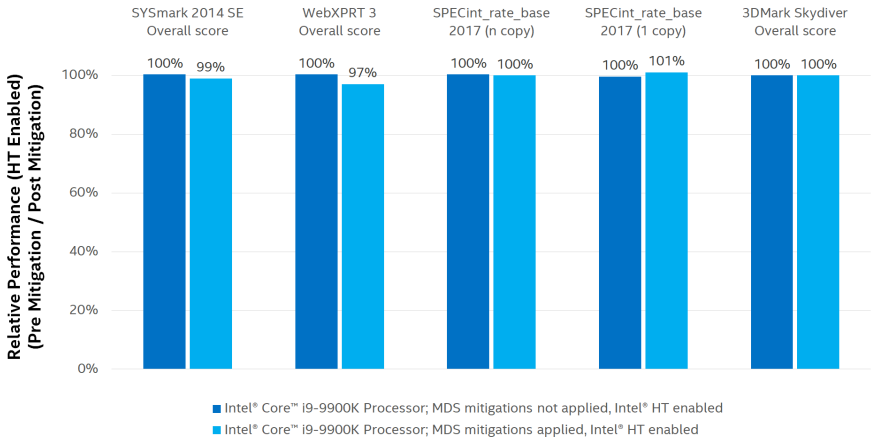
- Disable HyperThreading
- Microcode Updates (to an extent)

- Disable HyperThreading
- Microcode Updates (to an extent)
- `VERW` instruction → clear buffers on context switch

Overhead?


SYSmark 2014 SE
Overall score
WebXPRT 3
Overall score
SPECint_rate_base
2017 (n copy)
SPECint_rate_base
2017 (1 copy)
3DMark Skydiver
Overall score
100%
99%
100%
97%
100%
100%
100%
101%
100%
100%
Relative Performance (HT Enabled)
(Pre Mitigation / Post Mitigation)
100%
80%
60%
40%
20%
0%
Intel® Core™ i9-9900K Processor; MDS mitigations not applied, Intel® HT enabled
Intel® Core™ i9-9900K Processor; MDS mitigations applied, Intel® HT enabled
View Full Image

Overhead?

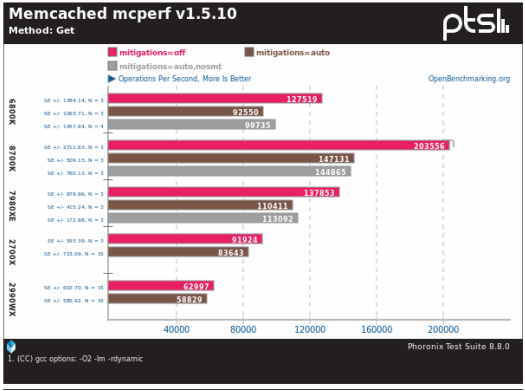


Performance results are based on testing as of May 8, 2019 and may not reflect all publicly available security updates. See configuration disclosure for details. No product or component can be absolutely secure. Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark® and MobileMark®, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information about performance and benchmark results, visit http://www.intel.com/benchmarks.

Overhead?

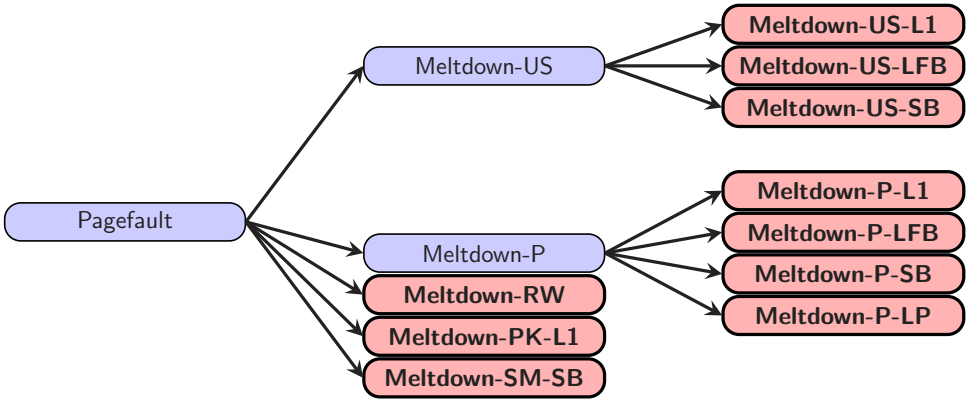
Software and workloads used in performance tests may have been optimized for the performance only on Intel microprocessors. [...] Any chance of any of those factors cause the results to vary.

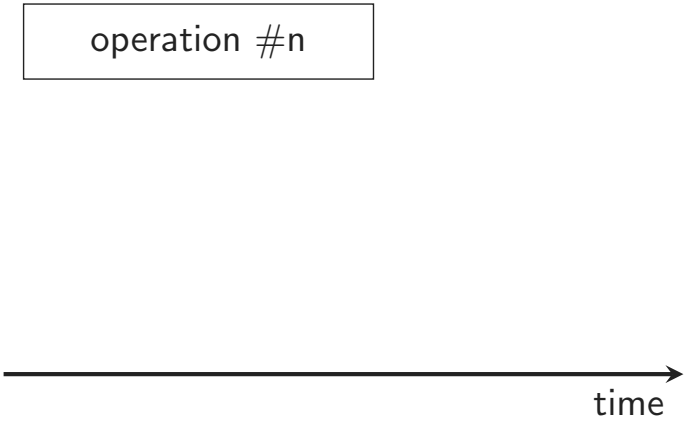
Overhead?

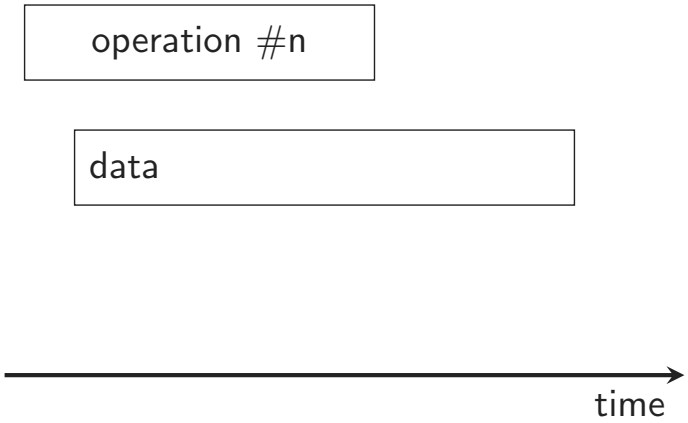


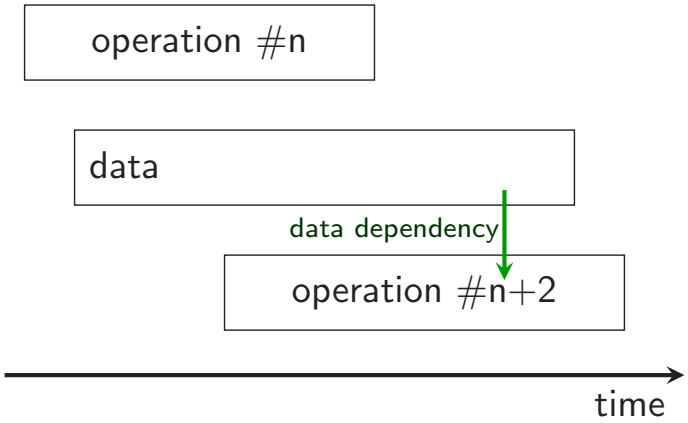
About 16% lower performance on average (phoronix.com)

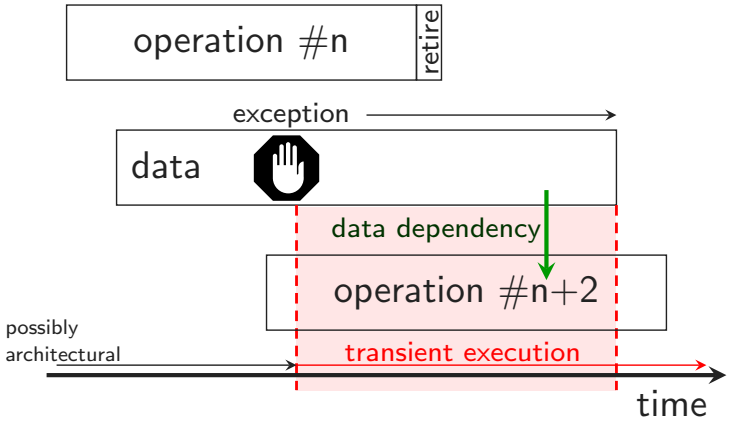
Meltdown Variants



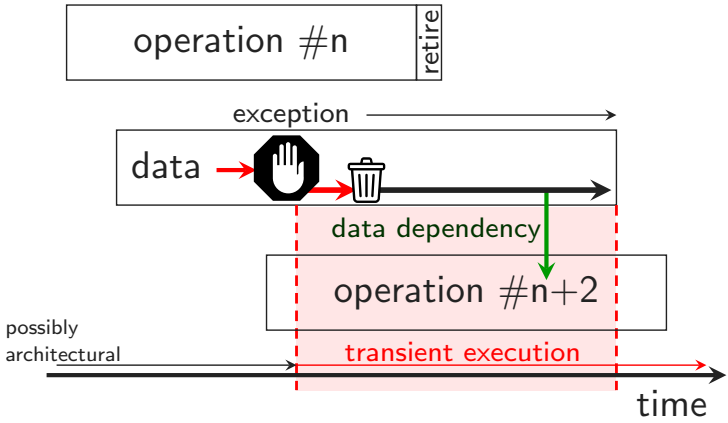

operation #n
time

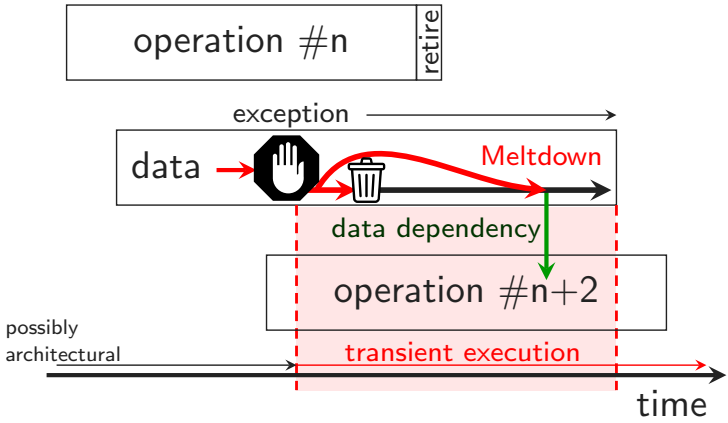

operation #n
data
time

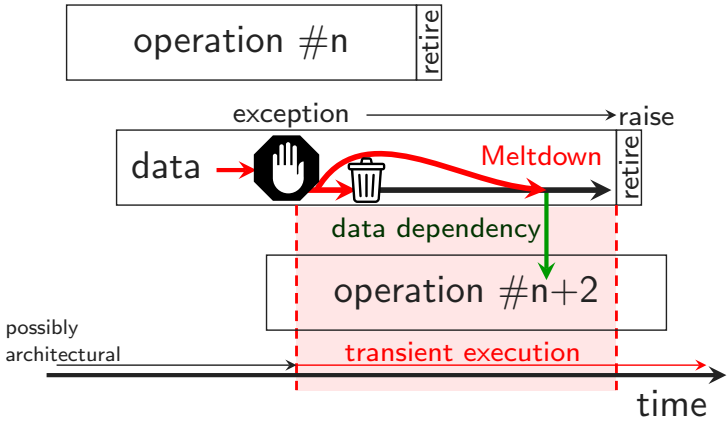

operation #n
data
data dependency
operation #n+2
time


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

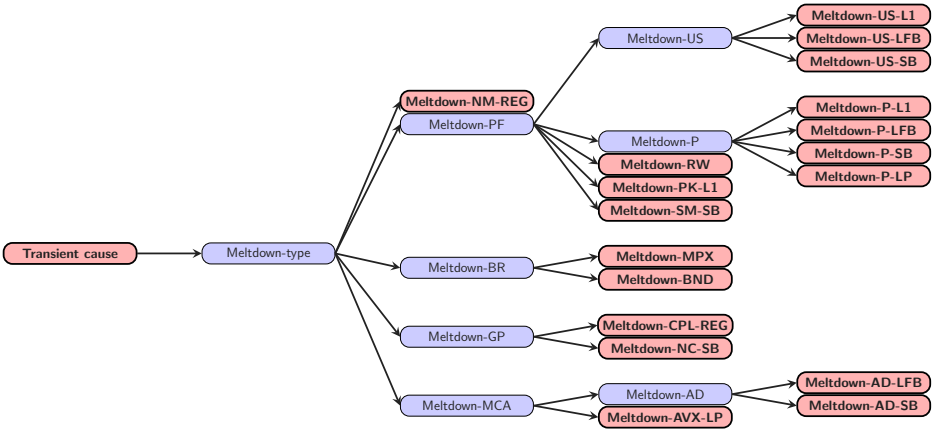
Meltdown Root Cause




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


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

Meltdown Tree



- Meltdown is not a fully solved issue

Meltdown Outlook

- Meltdown is not a fully solved issue
- The tree is extensible

Meltdown Outlook

- Meltdown is not a fully solved issue
- The tree is extensible
- More Meltdown-type issues to come

Meltdown Outlook

- Meltdown is not a fully solved issue
- The tree is extensible
- More Meltdown-type issues to come
- Silicon fixes might not be complete

- Meltdown not the only transient execution attacks

- Meltdown not the only transient execution attacks
- Spectre is a second class of transient execution attacks

- Meltdown not the only transient execution attacks
- Spectre is a second class of transient execution attacks
- Instead of faults, exploit control (or data) flow predictions

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

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

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

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

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

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

```
index = 0
```

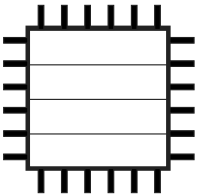
Shared Memory

	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)
```

then → `glyph[data[index]]`

else → `{}`



Memory

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

Spectre-PHT (aka Spectre Variant 1)

```
index = 0
```

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)
```

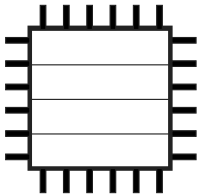
then

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

else

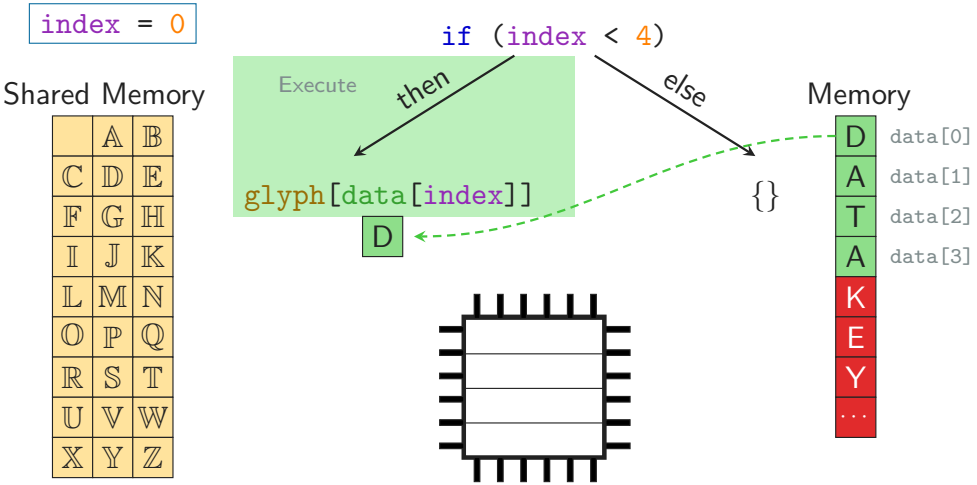
Speculate

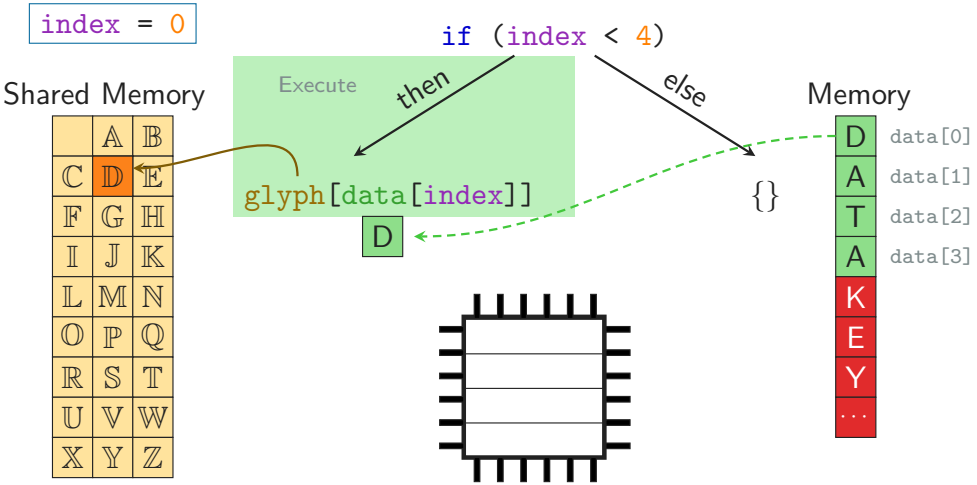
{}

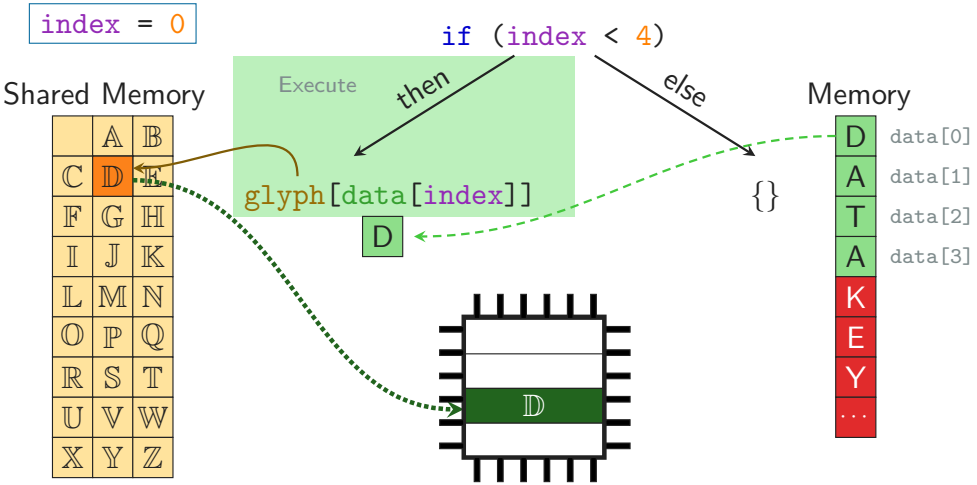


Memory

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


index = 0
if (index < 4)
then
else
Execute
glyph[data[index]]
D
{}
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
A
T
A
K
E
Y
...
data[0]
data[1]
data[2]
data[3]


index = 0
if (index < 4)
then
else
Execute
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
glyph[data[index]]
D
{}
Memory
D data[0]
A data[1]
T data[2]
A data[3]
K
E
Y
...


index = 0
if (index < 4)
then
else
Execute
Shared Memory
Memory
glyph[data[index]]
D
{}
data[0]
data[1]
data[2]
data[3]
D
A
T
A
K
E
Y
...

```
index = 1
```

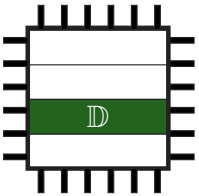
Shared Memory

	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)
```

then → `glyph[data[index]]`

else → `{}`



Memory

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

```
index = 1
```

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

Speculate

then

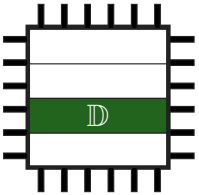
```
glyph[data[index]]
```

else

{}

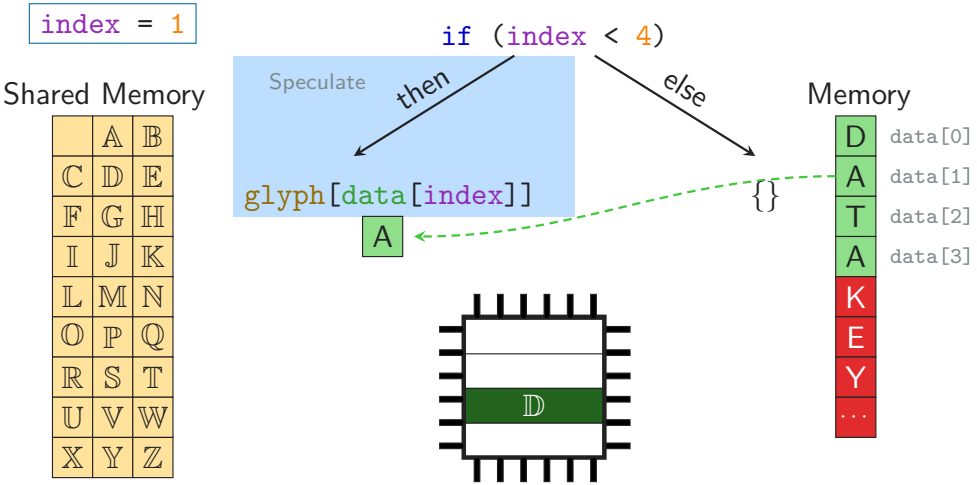
Shared Memory

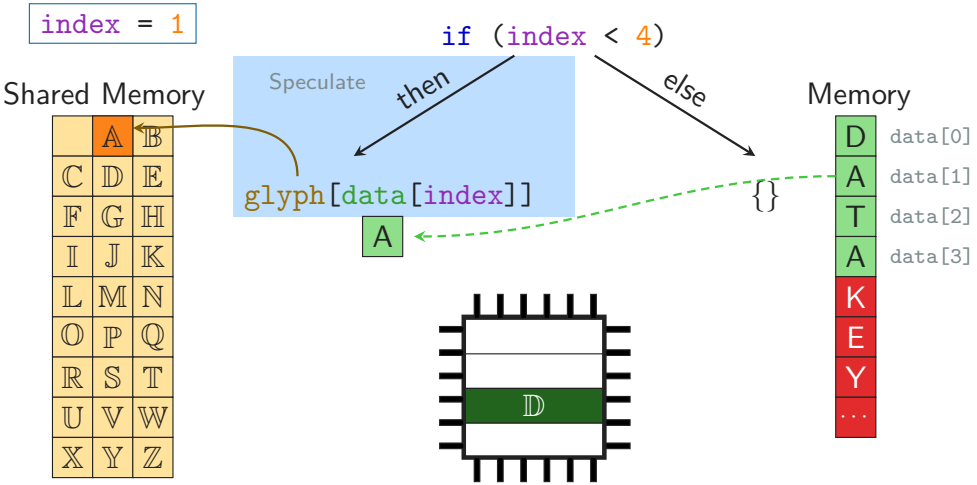
	A	B
C	D	E
F	G	H
I	J	K
L	M	N
O	P	Q
R	S	T
U	V	W
X	Y	Z

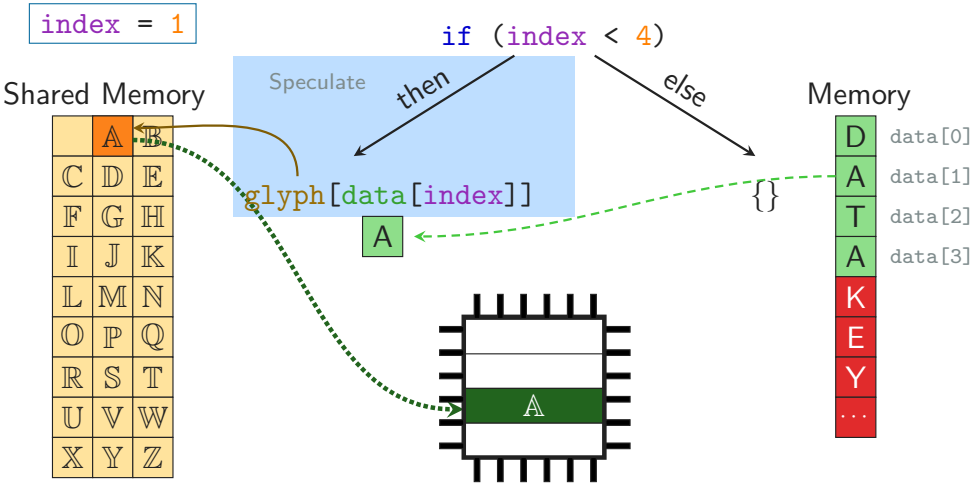


Memory

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


index = 1
if (index < 4)
Speculate
then
else
Shared Memory
Memory
glyph[data[index]]
A
{}
D
A
T
A
K
E
Y
...
data[0]
data[1]
data[2]
data[3]
D


index = 1
if (index < 4)
then
else
Speculate
Shared Memory
Memory
glyph[data[index]]
A
{}
data[0]
data[1]
data[2]
data[3]
D
A
T
A
K
E
Y
...
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
D


index = 1
if (index < 4)
Speculate
then
else
Shared Memory
Memory
glyph[data[index]]
A
{}
data[0]
data[1]
data[2]
data[3]
D
A
T
A
K
E
Y
...

```
index = 1
```

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

then

Execute

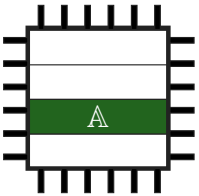
```
glyph[data[index]]
```

else

{}

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	
...	

```
index = 2
```

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

Speculate

then

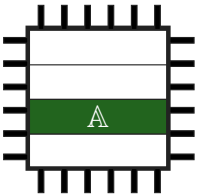
```
glyph[data[index]]
```

else

```
{}
```

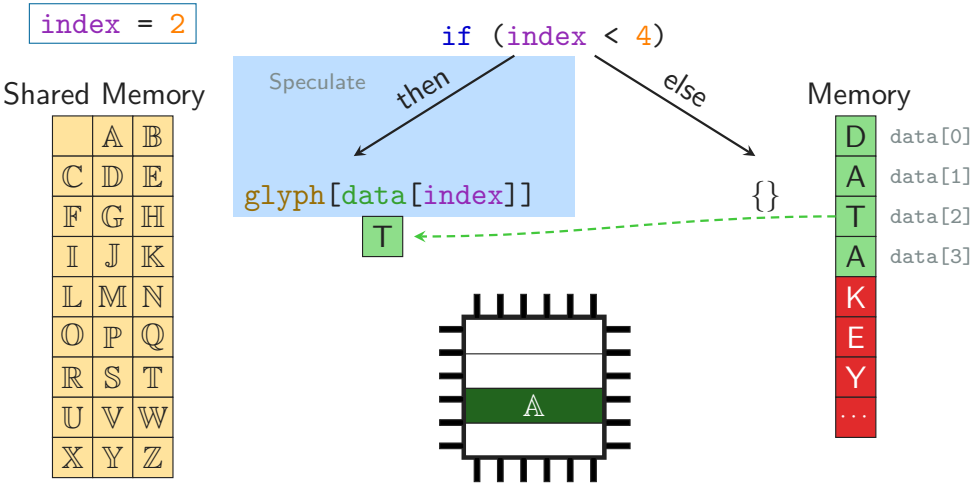
Shared Memory

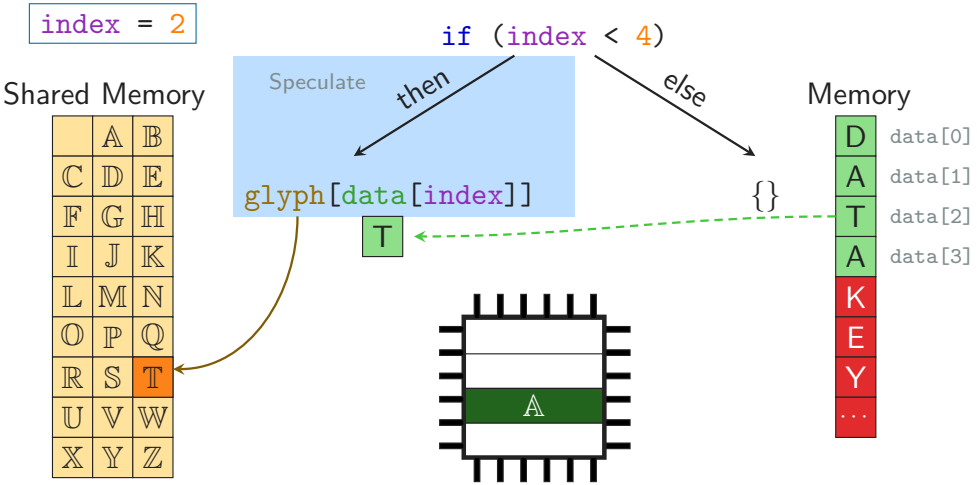
	A	B
C	D	E
F	G	H
I	J	K
L	M	N
O	P	Q
R	S	T
U	V	W
X	Y	Z

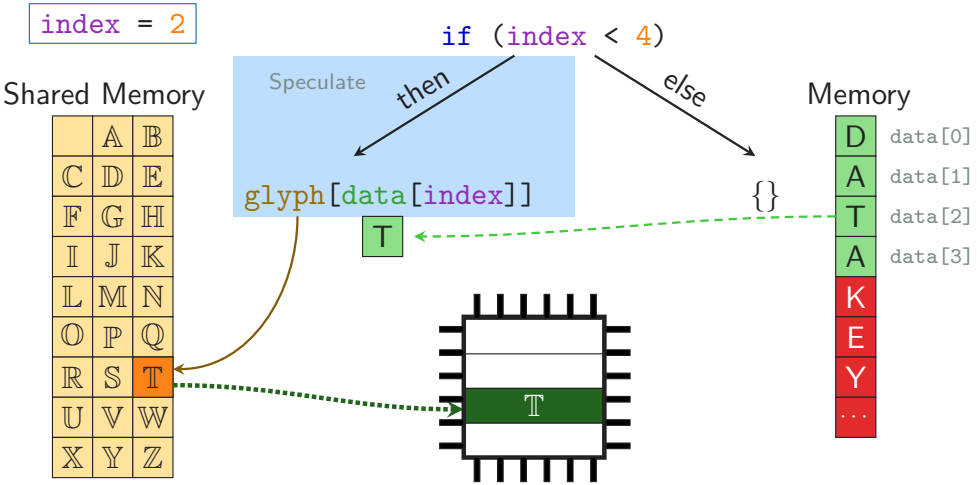


Memory

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


index = 2
Shared Memory
if (index < 4)
Speculate
then
else
glyph[data[index]]
T
{}
Memory
D
A
T
A
K
E
Y
...
data[0]
data[1]
data[2]
data[3]
A


index = 2
if (index < 4)
then
else
Speculate
glyph[data[index]]
T
{}
Shared Memory
Memory
data[0]
data[1]
data[2]
data[3]


index = 2
if (index < 4)
then
else
Speculate
glyph[data[index]]
T
{}
Shared Memory
Memory
data[0]
data[1]
data[2]
data[3]
T

```
index = 2
```

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

Execute

then

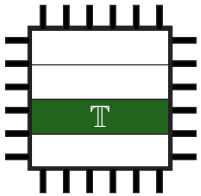
else

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

{}

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	
...	

```
index = 3
```

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)
```

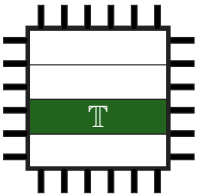
Speculate

then

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

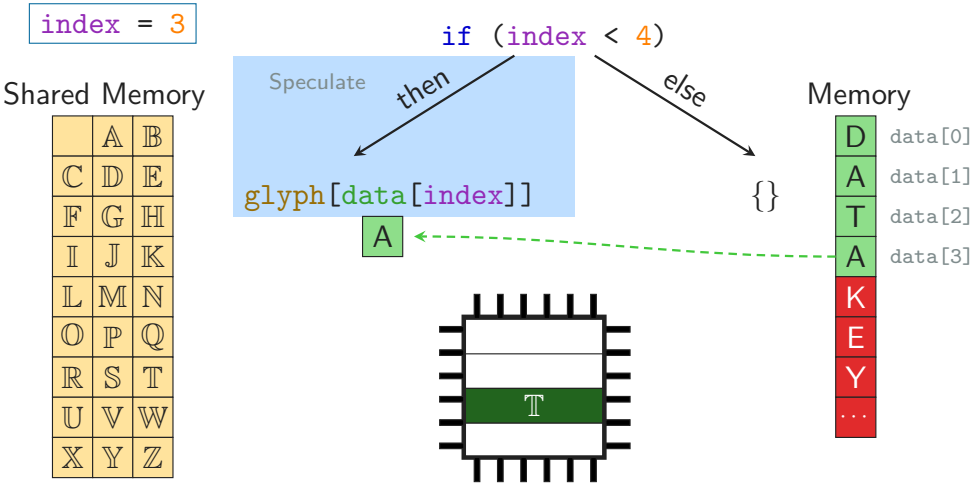
else

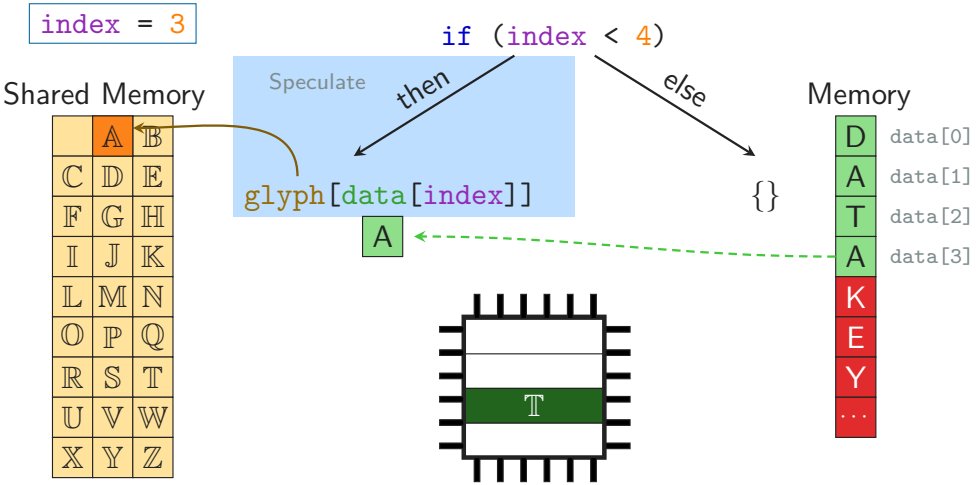
```
{}
```

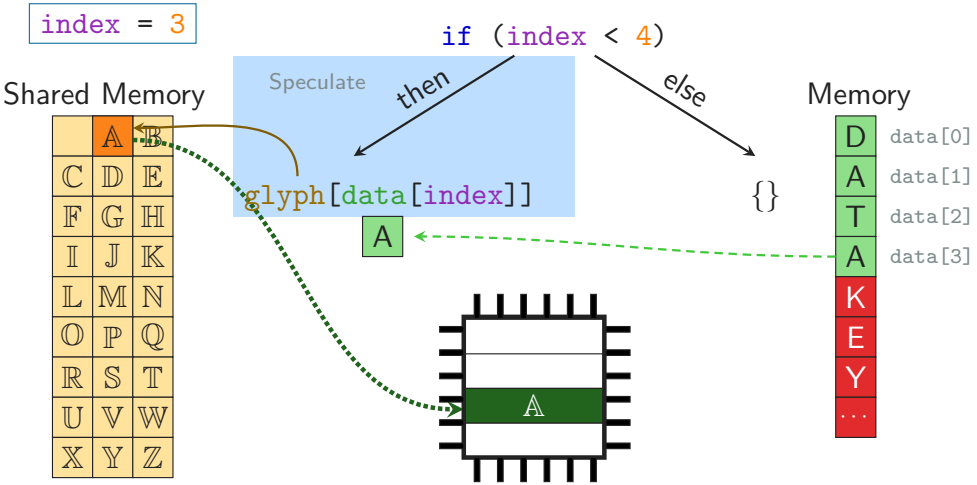


Memory

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


index = 3
Shared Memory
if (index < 4)
Speculate
then
else
glyph[data[index]]
{}
A
Memory
D
A
T
A
K
E
Y
...
data[0]
data[1]
data[2]
data[3]
T


index = 3
if (index < 4)
Speculate
then
else
Shared Memory
Memory
glyph[data[index]]
{}
A
data[0]
data[1]
data[2]
data[3]
T


index = 3
if (index < 4)
then
else
Speculate
Shared Memory
Memory
glyph[data[index]]
{}
A
data[0]
data[1]
data[2]
data[3]
D
A
T
A
K
E
Y
...

```
index = 3
```

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

then

Execute

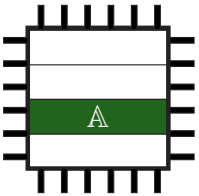
```
glyph[data[index]]
```

else

{}

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	
...	

```
index = 4
```

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)
```

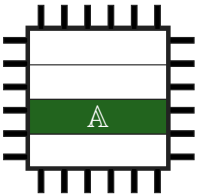
Speculate

then

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

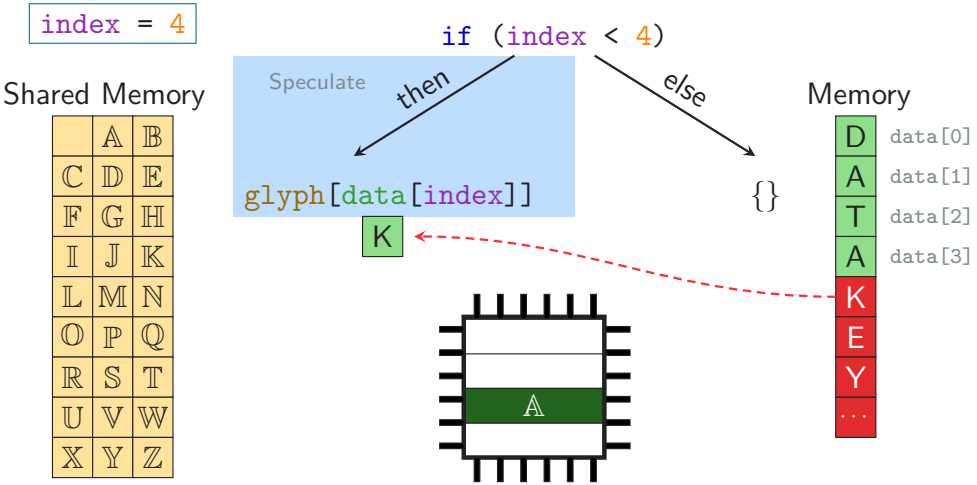
else

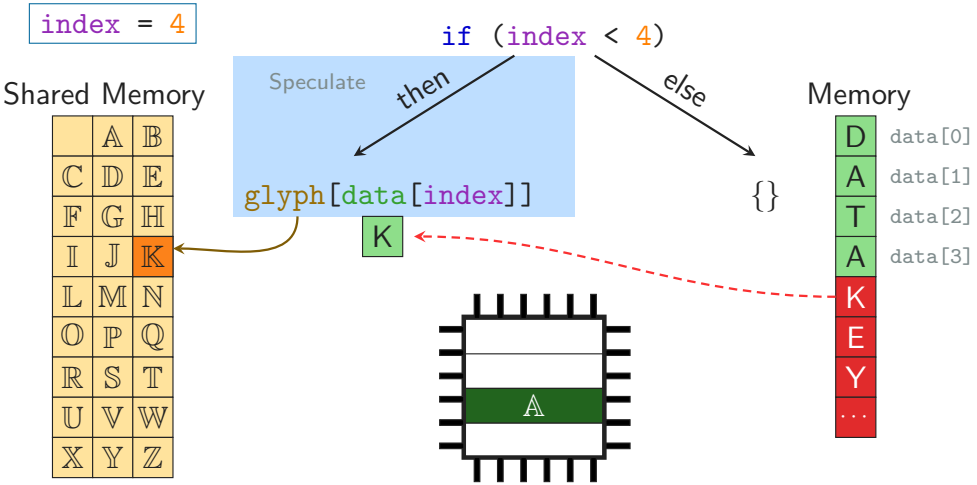
```
{}
```

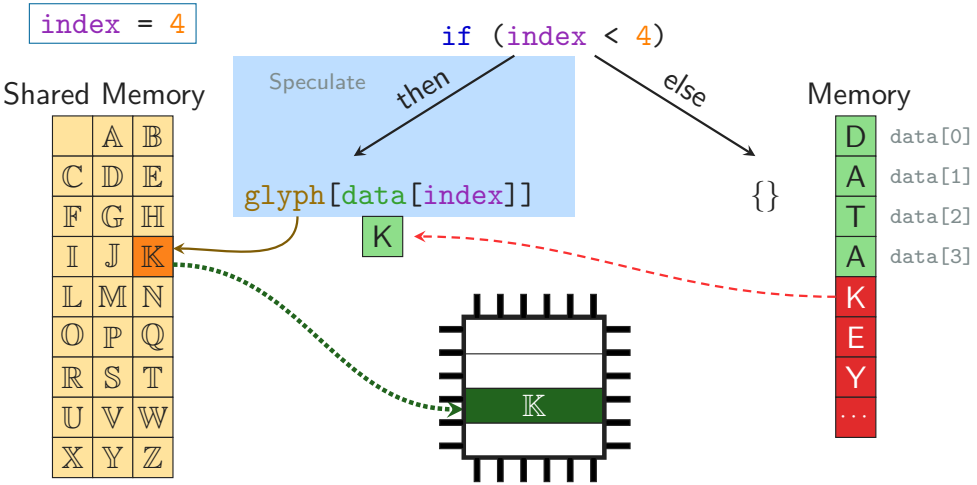


Memory

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


index = 4
if (index < 4)
Speculate
then
else
Shared Memory
Memory
glyph[data[index]]
{}
K
data[0]
data[1]
data[2]
data[3]
D
A
T
A
K
E
Y
...


index = 4
if (index < 4)
then
else
Speculate
glyph[data[index]]
K
{}
Shared Memory
Memory
data[0]
data[1]
data[2]
data[3]
D
A
T
A
K
E
Y
...
A


index = 4
if (index < 4)
Speculate
then
else
Shared Memory
glyph[data[index]]
K
{}
Memory
D
A
T
A
K
E
Y
...
data[0]
data[1]
data[2]
data[3]
K

```
index = 4
```

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

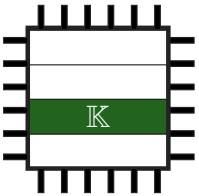
```
if (index < 4)
```

then

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

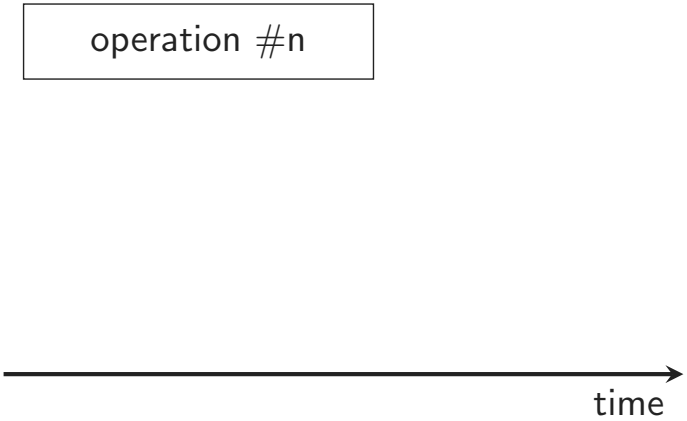
else

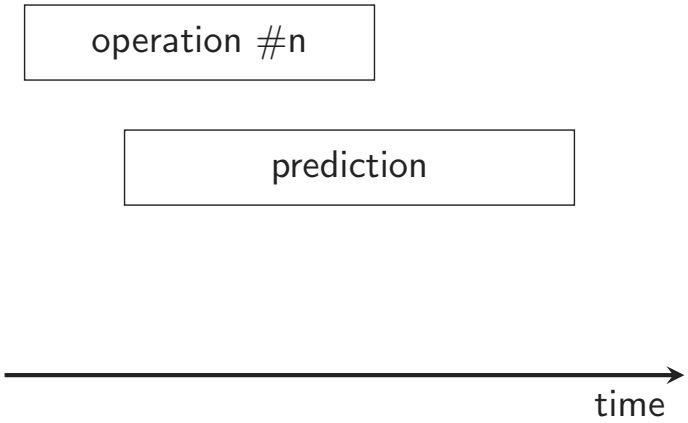
Execute {}

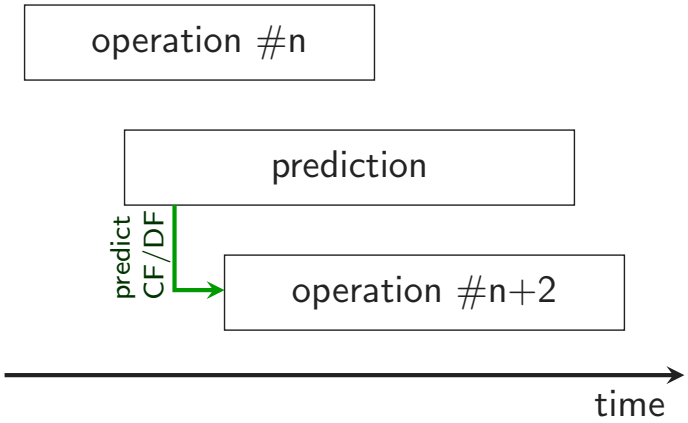


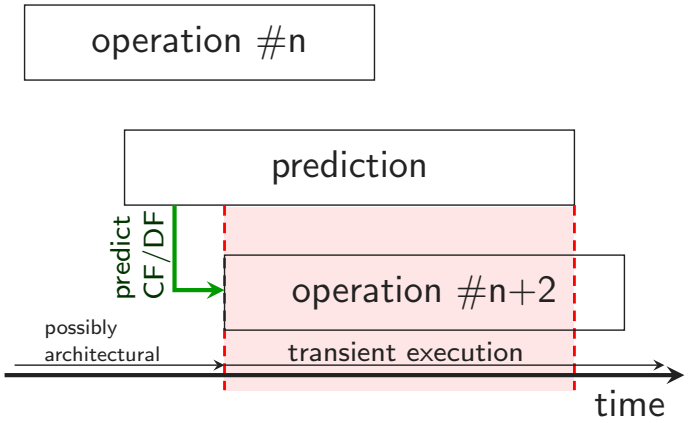
Memory

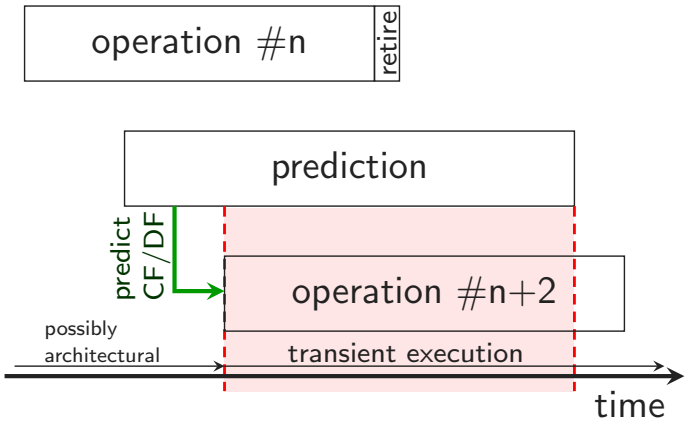
D	data[0]
A	data[1]
T	data[2]
A	data[3]
K	
E	
Y	
...	

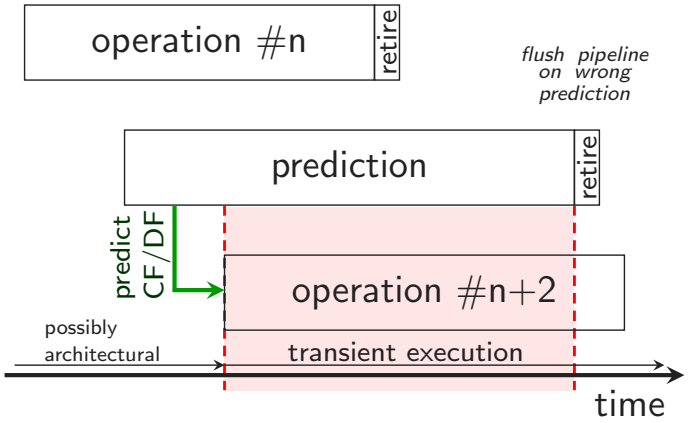

operation #n
time

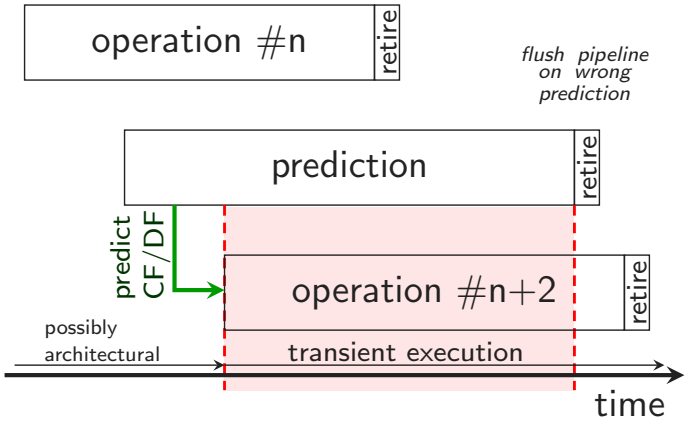

operation #n
prediction
time

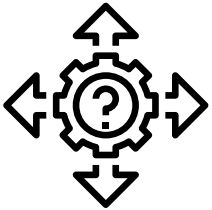

operation #n
prediction
predict
CF/DF
operation #n+2
time


operation #n
prediction
predict CF/DF
operation #n+2
possibly architectural
transient execution
time

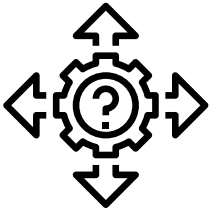

operation #n
retire
prediction
predict CF/DF
operation #n+2
possibly architectural
transient execution
time


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

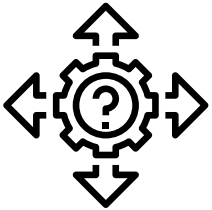

operation #n
retire
flush pipeline on wrong prediction
prediction
retire
predict CF/DF
operation #n+2
retire
possibly architectural
transient execution
time



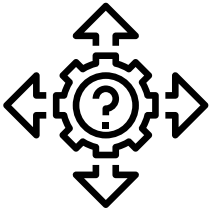
- Many predictors in modern CPUs



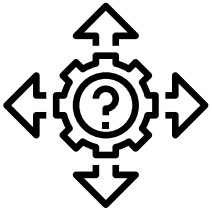
- Many predictors in modern CPUs
  - Branch taken/not taken (PHT)



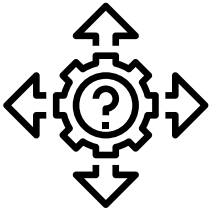
- Many predictors in modern CPUs
  - Branch taken/not taken (PHT)
  - Call/Jump destination (BTB)



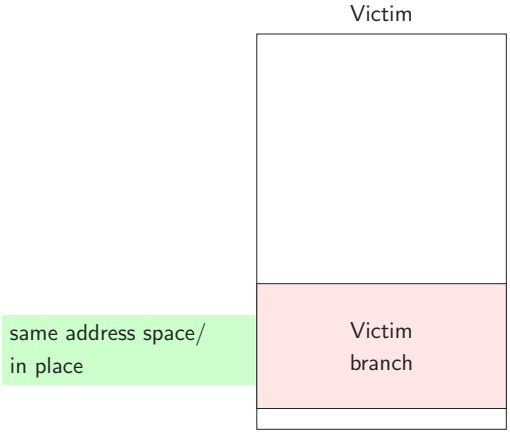
- Many predictors in modern CPUs
  - Branch taken/not taken (PHT)
  - Call/Jump destination (BTB)
  - Function return destination (RSB)

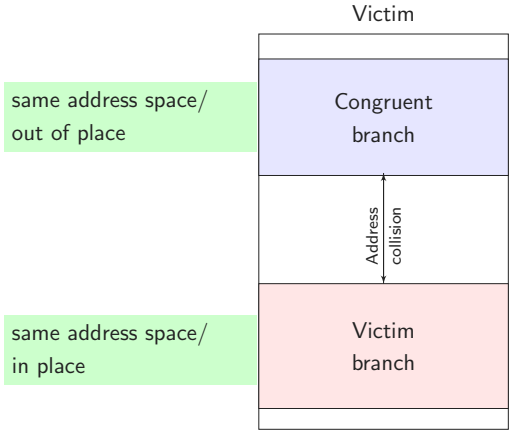


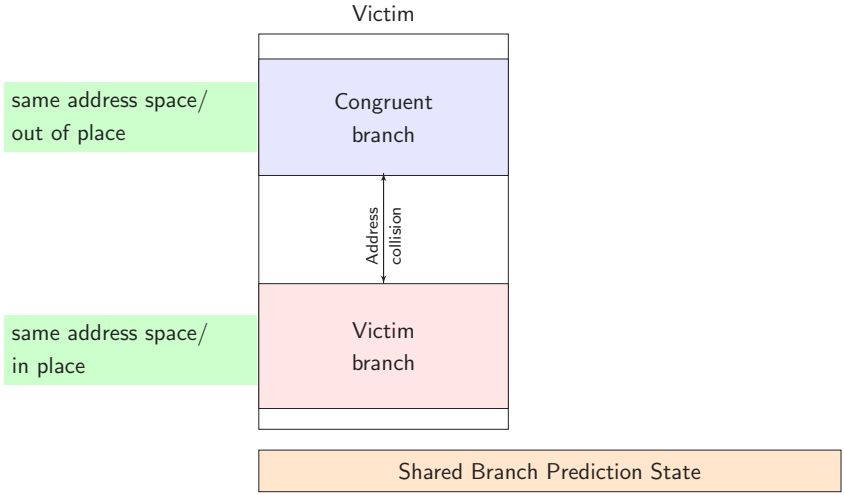
- Many predictors in modern CPUs
  - Branch taken/not taken (PHT)
  - Call/Jump destination (BTB)
  - Function return destination (RSB)
  - Load matches previous store (STL)

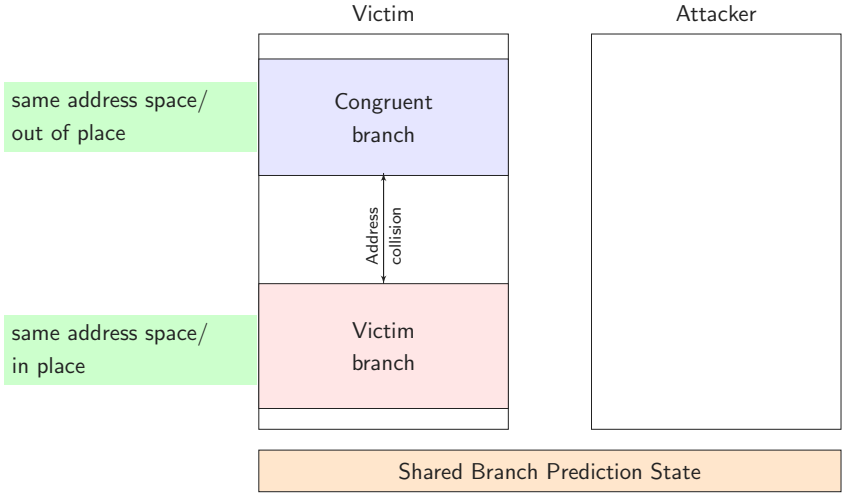


- Many predictors in modern CPUs
  - Branch taken/not taken (PHT)
  - Call/Jump destination (BTB)
  - Function return destination (RSB)
  - Load matches previous store (STL)
- Most are even shared among processes

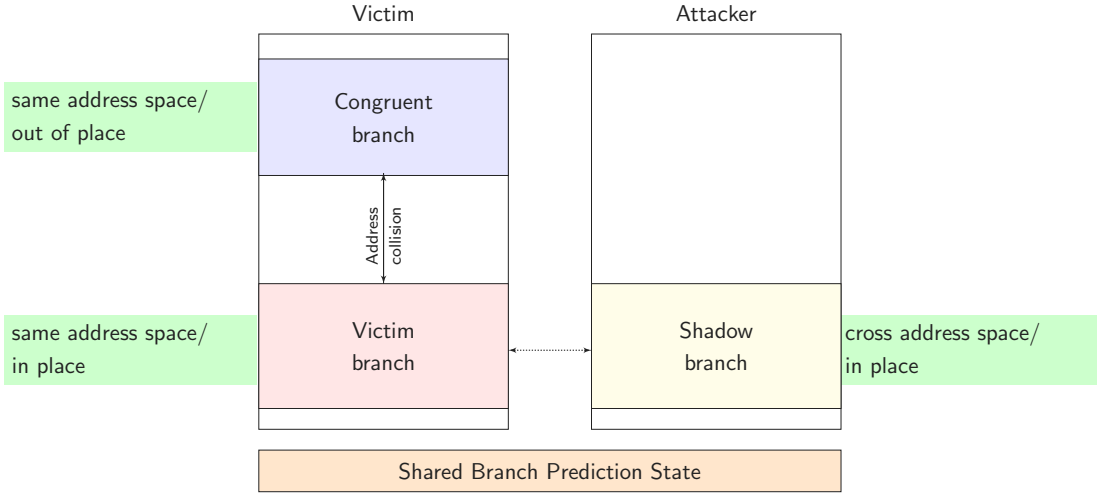

Victim
same address space/
in place
Victim
branch

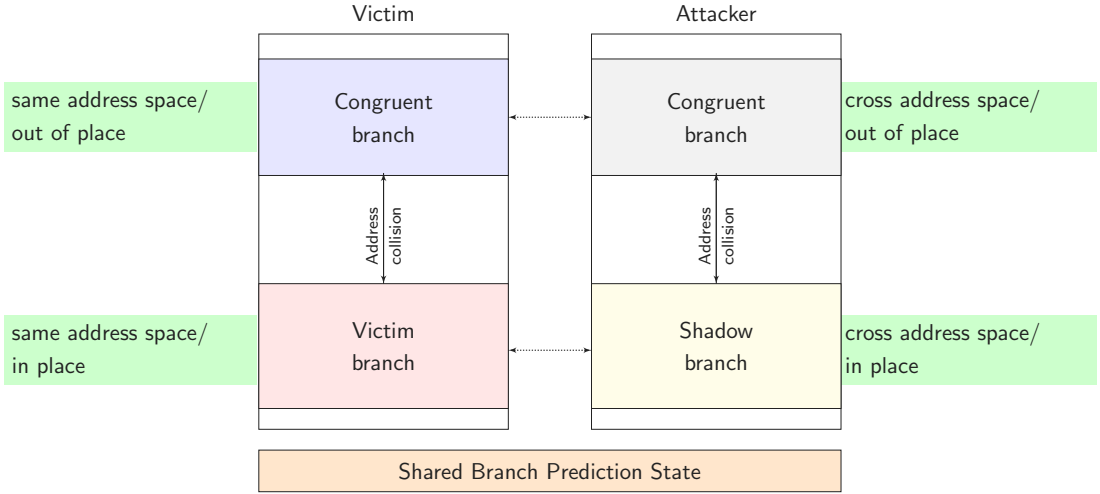

Victim
same address space/
out of place
Congruent
branch
Address
collision
same address space/
in place
Victim
branch


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


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

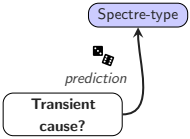
Spectre Mistraining



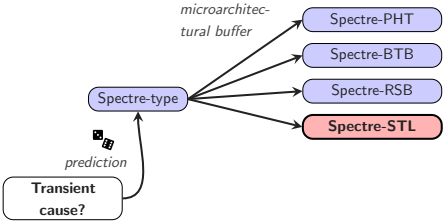

Victim
Attacker
same address space/
out of place
Congruent
branch
Congruent
branch
cross address space/
out of place
Address
collision
Address
collision
same address space/
in place
Victim
branch
Shadow
branch
cross address space/
in place
Shared Branch Prediction State

Transient cause?

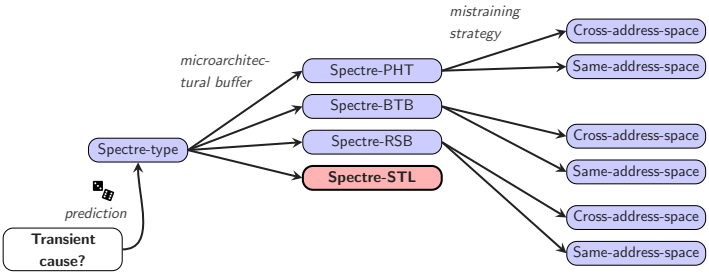
Spectre Variants



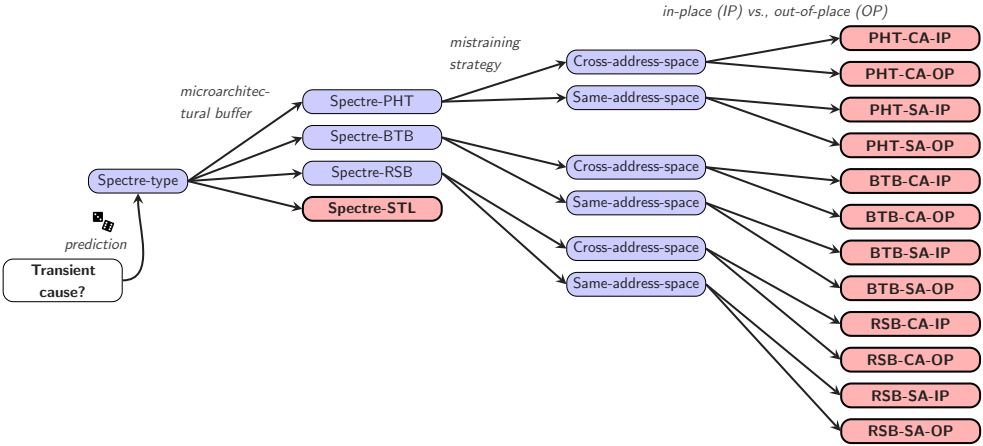
Spectre Variants

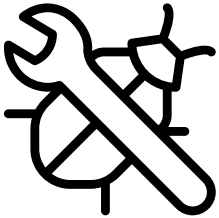


Spectre Variants

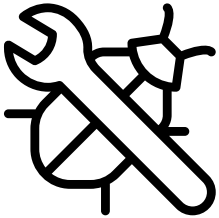


Spectre Variants

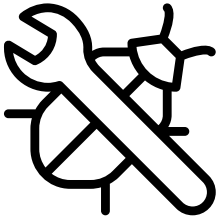




- Spectre is not a bug

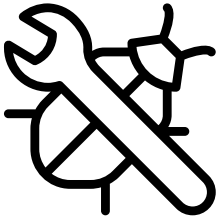


- Spectre is not a bug
- It is an useful optimization



- Spectre is not a bug
- It is an useful optimization

→ Cannot simply fix it (as with Meltdown)

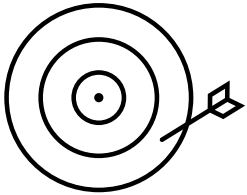


- Spectre is not a bug
- It is an useful optimization

→ Cannot simply fix it (as with Meltdown)

- Workarounds for critical code parts

Spectre defenses in 3 categories:



C1 Mitigating or reducing the accuracy of covert channels

C2 Mitigating or aborting speculation

C3 Ensuring secret data cannot be reached

	Attack \ Defense	*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																		

Attack is mitigated (●), partially mitigated (◐), not mitigated (○), theoretically mitigated (■), theoretically impeded (◧), not theoretically impeded (□), or out of scope (◇).

	Attack \ Defense	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																		●

Attack is mitigated (●), partially mitigated (◐), not mitigated (○), theoretically mitigated (■), theoretically impeded (◧), not theoretically impeded (□), or out of scope (◇).

Spectre: Defense Analysis

	Attack \ Defense	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								◐								◐		●

Attack is mitigated (●), partially mitigated (◐), not mitigated (○), theoretically mitigated (■), theoretically impeded (◧), not theoretically impeded (□), or out of scope (◇).

	Attack \ Defense	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								◐								◐		●

Attack is mitigated (●), partially mitigated (◐), not mitigated (○), theoretically mitigated (■), theoretically impeded (◧), not theoretically impeded (□), or out of scope (◇).

	Attack \ Defense	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								◐							■	◐	■	●

Attack is mitigated (●), partially mitigated (◐), not mitigated (○), theoretically mitigated (■), theoretically impeded (◧), not theoretically impeded (□), or out of scope (◇).

	Attack \ Defense	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								◐							■	◐	■	●

Attack is mitigated (●), partially mitigated (◐), not mitigated (○), theoretically mitigated (■), theoretically impeded (◧), not theoretically impeded (□), or out of scope (◇).

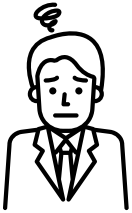
Spectre: Defense Analysis

	Attack \ Defense	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	□	□	□					◐							■	◐	■	●

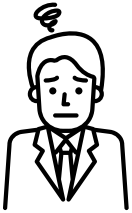
Attack is mitigated (●), partially mitigated (◐), not mitigated (○), theoretically mitigated (■), theoretically impeded (◧), not theoretically impeded (□), or out of scope (◇).

	Attack \ Defense	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	□	□	□	◇	◇	◇	◇	◐	◇	◇	◇	◇	◇	◇	■	◐	■	●

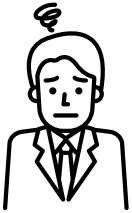
Attack is mitigated (●), partially mitigated (◐), not mitigated (○), theoretically mitigated (■), theoretically impeded (◧), not theoretically impeded (□), or out of scope (◇).



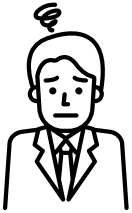
- Many countermeasures only consider the cache to get data...



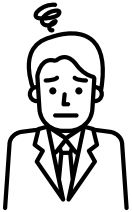
- Many countermeasures only consider the cache to get data...
- ...but there are other possibilities, e.g.,



- Many countermeasures only consider the cache to get data...
- ...but there are other possibilities, e.g.,
  - Port contention (SMoTherSpectre)



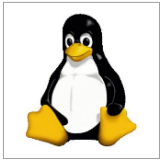
- Many countermeasures only consider the cache to get data...
- ...but there are other possibilities, e.g.,
  - Port contention (SMoTherSpectre)
  - AVX (NetSpectre)



- Many countermeasures only consider the cache to get data...
- ...but there are other possibilities, e.g.,
  - Port contention (SMoTherSpectre)
  - AVX (NetSpectre)
- Cache is just the easiest

Linux 4.19.4 & 4.14.83 Released With STIBP Code Dropped

Written by Michael Larabel in Linux Kernel on 24 November 2018 at 09:00 AM EST. 6 Comments



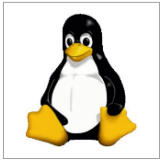
On Friday marked the release of the Linux 4.19.4 kernel as well as 4.14.83 and 4.9.139.

Greg Kroah-Hartman issued this latest round of stable point releases as basic maintenance updates. While these point releases don't tend to be too notable and generally go unmentioned on Phoronix, this round is worth pointing out since 4.19.4 and 4.14.83 are the releases that end up reverting the STIBP behavior that applied Single Thread Indirect Branch Predictors to all processes on supported systems. That is what was introduced in Linux 4.20 and then back-ported to the 4.19/4.14 LTS branches, which in turn hurt the performance a lot. So for now the code is removed.

As covered yesterday, there is improved STIBP code on the way for Linux 4.20 that by default just apply STIBP to SECCOMP threads and processes requesting it via prctl() but otherwise is off by default (that behavior can also be changed via kernel parameters).

Linux 4.19.4 & 4.14.83 Released With STIBP Code Dropped

Written by Michael Larabel in Linux Kernel on 24 November 2018 at 09:00 AM EST. 6 Comments



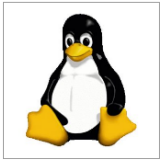
On Friday marked the release of the Linux 4.19.4 kernel as well as 4.14.83 and 4.9.139.

Greg Kroah-Hartman issued this latest round of stable point releases as basic maintenance updates. While these point releases don't tend to be too notable and generally go unmentioned on Phoronix, this round is worth pointing out since 4.19.4 and 4.14.83 are the releases that end up reverting the STIBP behavior that applied Single Thread Indirect Branch Predictors to all processes on supported systems. That is what was introduced in Linux 4.20 and then back-ported to the 4.19/4.14 LTS branches, which in turn hurt the performance a lot. So for now the code is removed.

As covered yesterday, there is improved STIBP code on the way for Linux 4.20 that by default just apply STIBP to SECCOMP threads and processes requesting it via prctl() but otherwise is off by default (that behavior can also be changed via kernel parameters).

Linux 4.19.4 & 4.14.83 Released With STIBP Code Dropped

Written by Michael Larabel in Linux Kernel on 24 November 2018 at 09:00 AM EST. 6 Comments



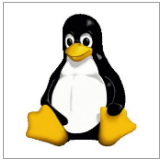
On Friday marked the release of the Linux 4.19.4 kernel as well as 4.14.83 and 4.9.139.

Greg Kroah-Hartman issued this latest round of stable point releases as basic maintenance updates. While these point releases don't tend to be too notable and generally go unmentioned on Phoronix, this round is worth pointing out since 4.19.4 and 4.14.83 are the releases that end up reverting the STIBP behavior that applied Single Thread Indirect Branch Predictors to all processes on supported systems. That is what was introduced in Linux 4.20 and then back-ported to the 4.19/4.14 LTS branches, which in turn hurt the performance a lot. So for now the code is removed.

As covered yesterday, there is improved STIBP code on the way for Linux 4.20 that by default just apply STIBP to SECCOMP threads and processes requesting it via prctl() but otherwise is off by default (that behavior can also be changed via kernel parameters).

Linux 4.19.4 & 4.14.83 Released With STIBP Code Dropped

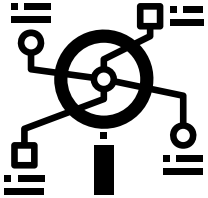
Written by Michael Larabel in Linux Kernel on 24 November 2018 at 09:00 AM EST. 6 Comments



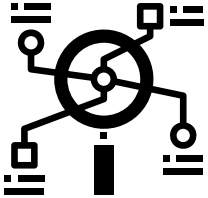
On Friday marked the release of the Linux 4.19.4 kernel as well as 4.14.83 and 4.9.139.

Greg Kroah-Hartman issued this latest round of stable point releases as basic maintenance updates. While these point releases don't tend to be too notable and generally go unmentioned on Phoronix, this round is worth pointing out since 4.19.4 and 4.14.83 are the releases that end up reverting the STIBP behavior that applied Single Thread Indirect Branch Predictors to all processes on supported systems. That is what was introduced in Linux 4.20 and then back-ported to the 4.19/4.14 LTS branches, which in turn hurt the performance a lot. So for now the code is removed.

As covered yesterday, there is improved STIBP code on the way for Linux 4.20 that by default just apply STIBP to SECCOMP threads and processes requesting it via prctl() but otherwise is off by default (that behavior can also be changed via kernel parameters).

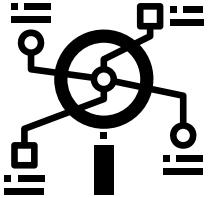


- Current mitigations are either incomplete or cost performance



- Current mitigations are either incomplete or cost performance

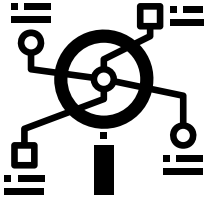
→ More research required



- Current mitigations are either incomplete or cost performance

→ More research required

- Both on attacks and defenses

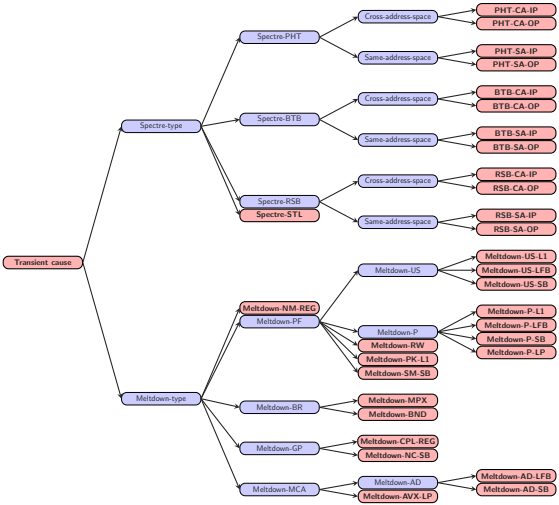


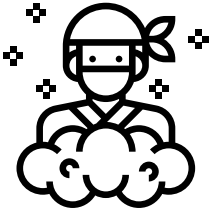
- Current mitigations are either incomplete or cost performance

→ More research required

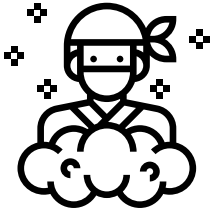
- Both on attacks and defenses

→ Efficient defenses only possible when attacks are known

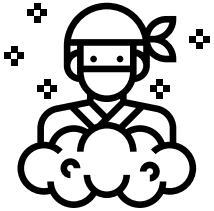

Transient cause
Spectre-type
Spectre-PHT
Cross-address-space
PHT-CA-IP
PHT-CA-OP
Same-address-space
PHT-SA-IP
PHT-SA-OP
Spectre-BTB
Cross-address-space
BTB-CA-IP
BTB-CA-OP
Same-address-space
BTB-SA-IP
BTB-SA-OP
Spectre-RSB
Spectre-STL
Cross-address-space
RSB-CA-IP
RSB-CA-OP
Same-address-space
RSB-SA-IP
RSB-SA-OP
Meltdown-type
Meltdown-US
Meltdown-US-L1
Meltdown-US-LFB
Meltdown-US-SB
Meltdown-NM-REG
Meltdown-PF
Meltdown-P
Meltdown-P-L1
Meltdown-P-LFB
Meltdown-P-SB
Meltdown-P-LP
Meltdown-RW
Meltdown-PK-L1
Meltdown-SM-SB
Meltdown-BR
Meltdown-MPX
Meltdown-BND
Meltdown-GP
Meltdown-CPL-REG
Meltdown-NC-SB
Meltdown-MCA
Meltdown-AD
Meltdown-AD-LFB
Meltdown-AD-SB
Meltdown-AVX-LP



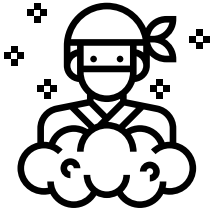
- Transient Execution Attacks are...



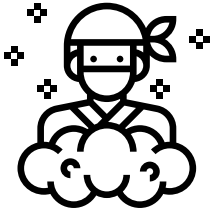
- Transient Execution Attacks are...
  - ...a novel class of attacks



- Transient Execution Attacks are...
  - ...a novel class of attacks
  - ...extremely powerful



- Transient Execution Attacks are...
  - ...a novel class of attacks
  - ...extremely powerful
  - ...only at the beginning



- Transient Execution Attacks are...
  - ...a novel class of attacks
  - ...extremely powerful
  - ...only at the beginning
- Many optimizations introduce side channels → now exploitable


BRACE YOURSELVES
MORE BUGS ARE COMING
imgflip.com

Transient Execution Attacks

Exploiting the CPU's Microarchitecture

Michael Schwarz (@misc0110), Moritz Lipp (@mlqxyz)

October 7, 2019

Graz University of Technology