Store-to-Leak Forwarding

There and Back Again

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte)

October 2, 2020

Graz University of Technology

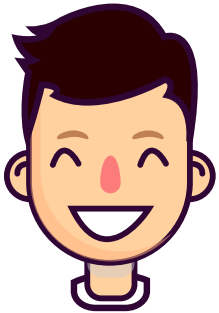
Who am I?

Claudio Canella

PhD student @ Graz University of Technology

@cc0x1f

claudio.canella@iaik.tugraz.at



Lukas Giner

PhD student @ Graz University of Technology

@redrabbyte

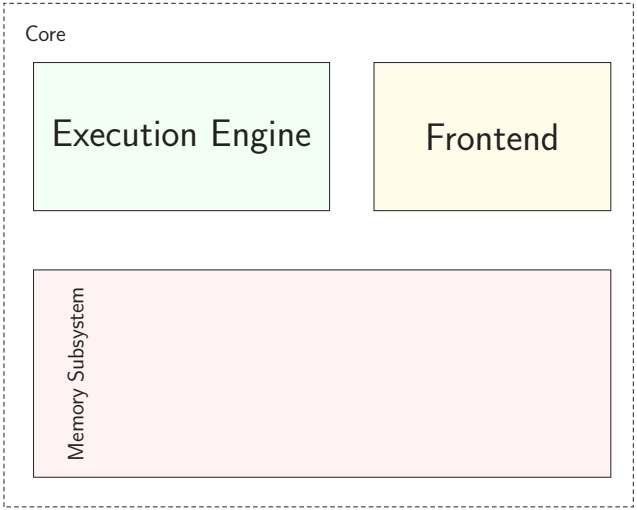
lukas.giner@iaik.tugraz.at

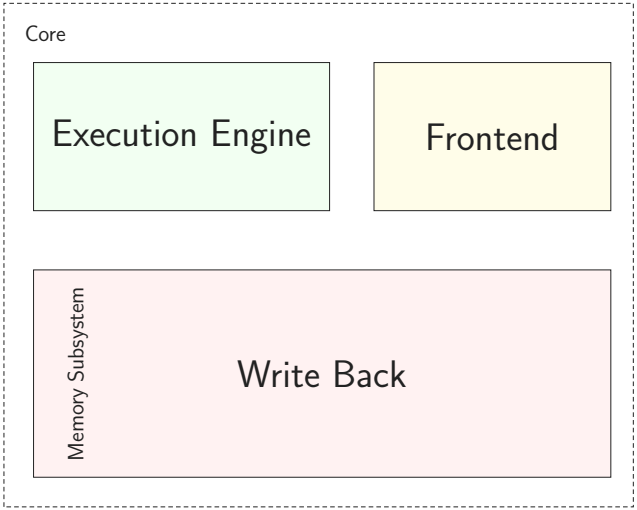
- How do loads handle previous stores?

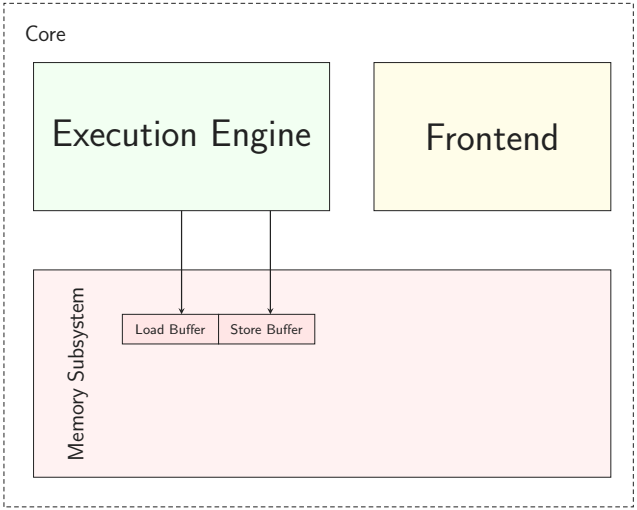
- How do loads handle previous stores?
- How is Meltdown mitigated in hardware?

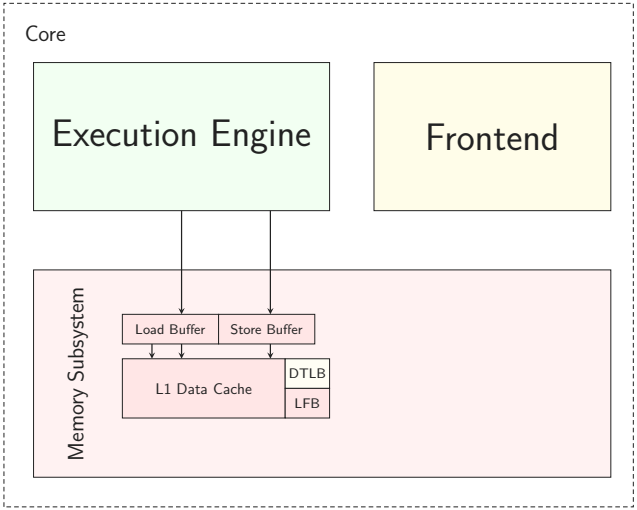
- How do loads handle previous stores?
- How is Meltdown mitigated in hardware?
- Can we abuse these mechanisms for new attacks?

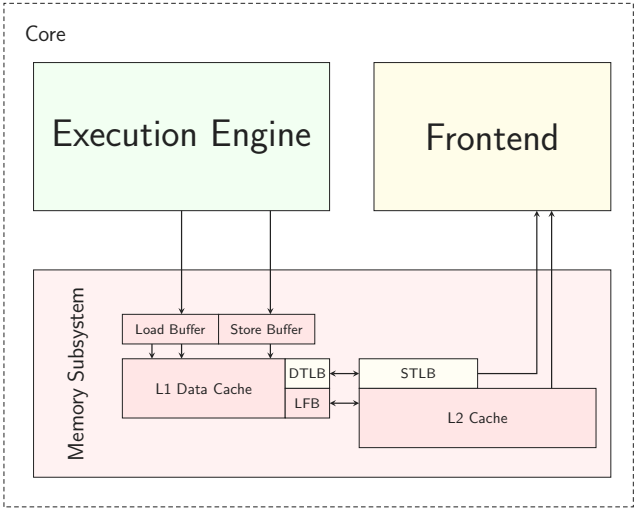
- How do loads handle previous stores?
- How is Meltdown mitigated in hardware?
- Can we abuse these mechanisms for new attacks?
- Can we mitigate such attacks efficiently?

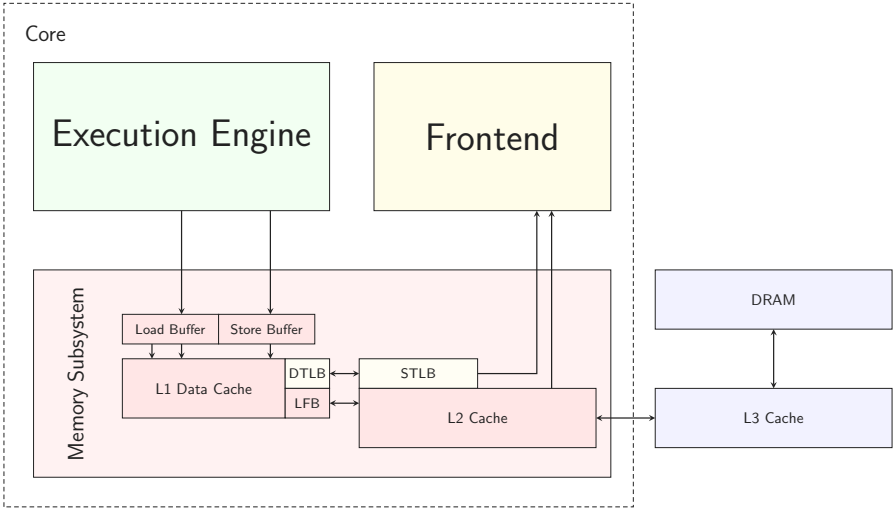

Core
Execution Engine
Frontend
Memory Subsystem


Core
Execution Engine
Frontend
Memory Subsystem
Write Back


Core
Execution Engine
Frontend
Memory Subsystem
Load Buffer
Store Buffer


Core
Execution Engine
Frontend
Memory Subsystem
Load Buffer
Store Buffer
L1 Data Cache
DTLB
LFB

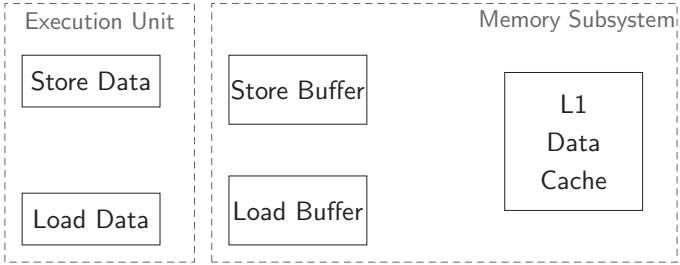

Core
Execution Engine
Frontend
Memory Subsystem
Load Buffer
Store Buffer
L1 Data Cache
DTLB
LFB
STLB
L2 Cache


Core
Execution Engine
Frontend
Memory Subsystem
Load Buffer
Store Buffer
L1 Data Cache
DTLB
LFB
STLB
L2 Cache
DRAM
L3 Cache

Store Buffer

```
mov data → [ address ]
```

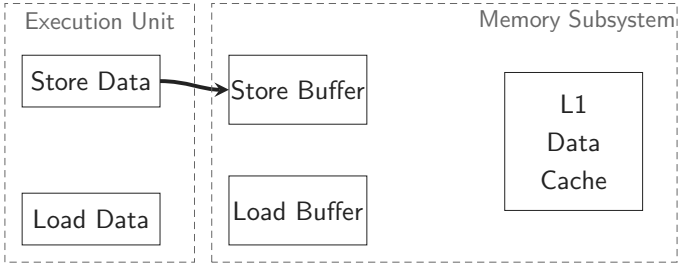
```
mov [ address ] → reg
```



Store Buffer

```
mov data → [ address ]
```

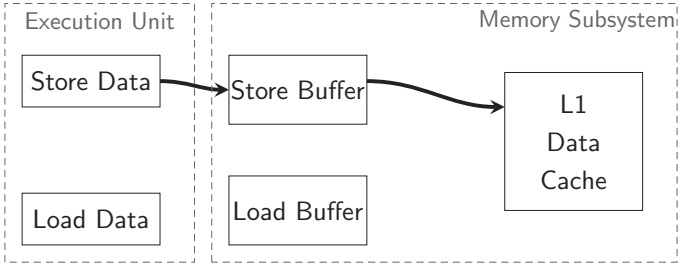
```
mov [ address ] → reg
```



Store Buffer

```
mov data → [ address ]
```

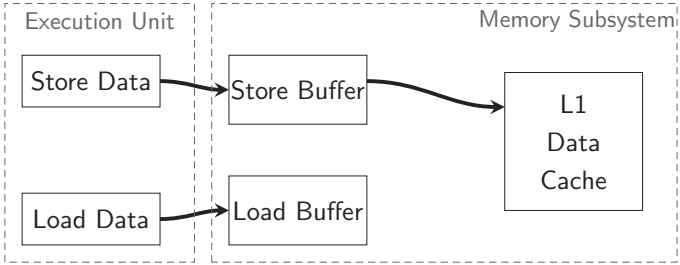
```
mov [ address ] → reg
```



Store Buffer

```
mov data → [ address ]
```

```
mov [ address ] → reg
```

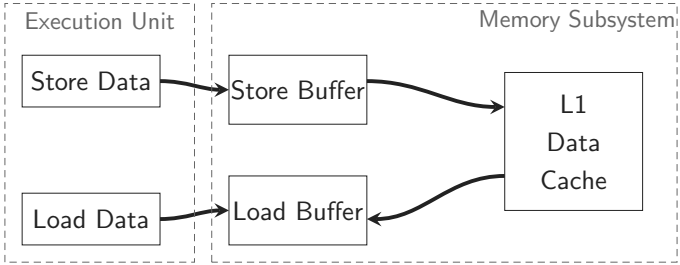


Store Buffer

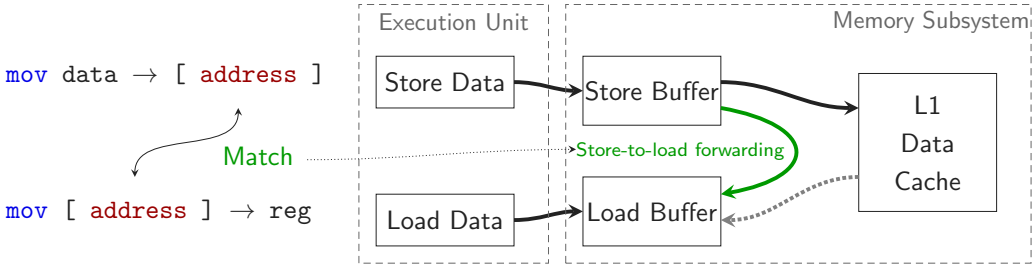
```
mov data → [ address ]
```

Compare

```
mov [ address ] → reg
```



Store Buffer



Store Buffer Match Logic

True Positive

Addresses 

Forwarding 

True Negative

Addresses 

Forwarding 

Store Buffer Match Logic

True Positive

Addresses =

Forwarding ✓

Store-to-Leak

True Negative

Addresses ≠

Forwarding ✗

Store Buffer Match Logic

	True Positive	False Positive
Addresses	=	≠
Forwarding	✓	✓

Store-to-Leak

	True Negative	False Negative
Addresses	≠	=
Forwarding	✗	✗

Store Buffer Match Logic

True Positive

Addresses =

Forwarding ✓

False Positive

Addresses ≠

Fallout ✓

Store-to-Leak

True Negative

Addresses ≠

Forwarding ✗

False Negative

Addresses =

Forwarding ✗

Store Buffer Match Logic

True Positive

Addresses =

Forwarding ✓

False Positive

Addresses ≠

Fallout ✓

Store-to-Leak

True Negative

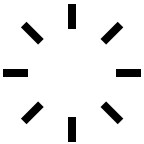
Addresses ≠

Forwarding ✗

False Negative

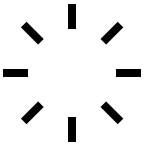
Addresses =

Spectre-STL ✗

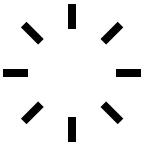


- Optimization during transient execution

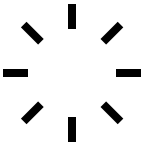
Store Buffer Match Logic



- Optimization during transient execution
- Failures not visible architecturally



- Optimization during transient execution
- Failures not visible architecturally
- Store-to-Leak relies on correct matching



- Optimization during transient execution
- Failures not visible architecturally
- Store-to-Leak relies on correct matching
- Only stores to valid addresses are forwarded

Intel Architecture Optimization Reference Manual

[The store execution phase] Fills the store buffers with linear and physical address and data.

Intel Architecture Optimization Reference Manual

[The store execution phase] Fills the store buffers with linear and physical address and data. Once store address and data are known, the store data can be forwarded to the following load operations that need it. [...]

- Virtual address that is physically backed $\rightarrow$ valid

Virtual Addresses

- Virtual address that is physically backed → valid
- Permission checks are deferred

- Virtual address that is physically backed → valid
- Permission checks are deferred

→ Exploited in Meltdown attacks

Virtual Addresses

- Virtual address that is physically backed → valid
- Permission checks are deferred

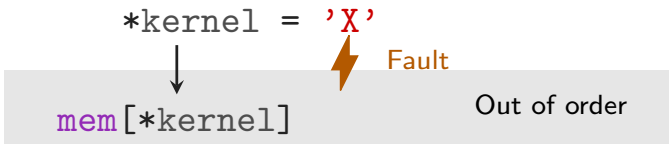
→ Exploited in Meltdown attacks

→ Store-to-load forwarding on inaccessible addresses

```
*kernel = 'X'
```

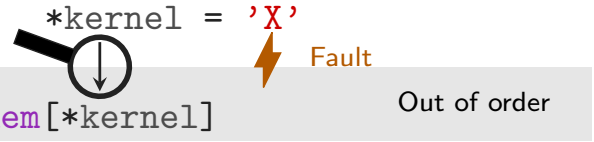
```
*kernel = 'X'
```

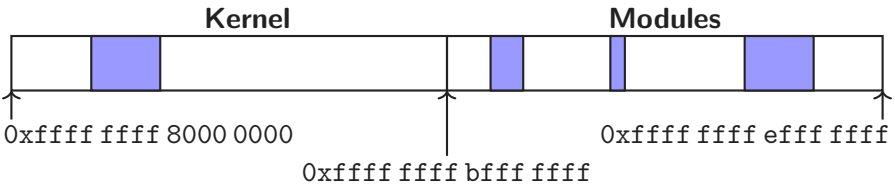
Fault

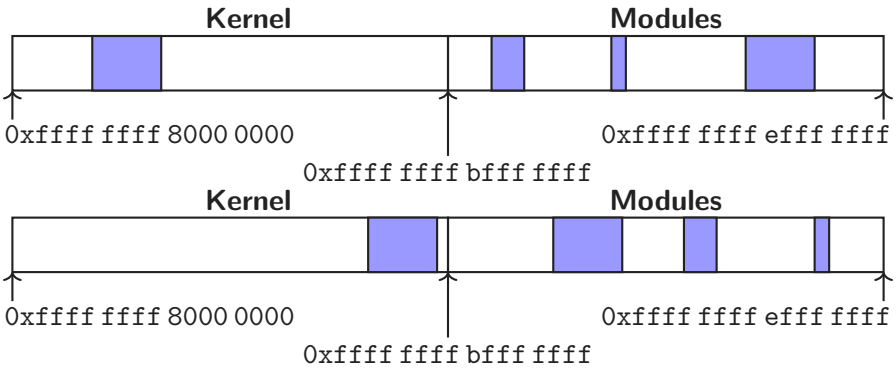

*kernel = 'X'
Fault
mem[*kernel]
Out of order

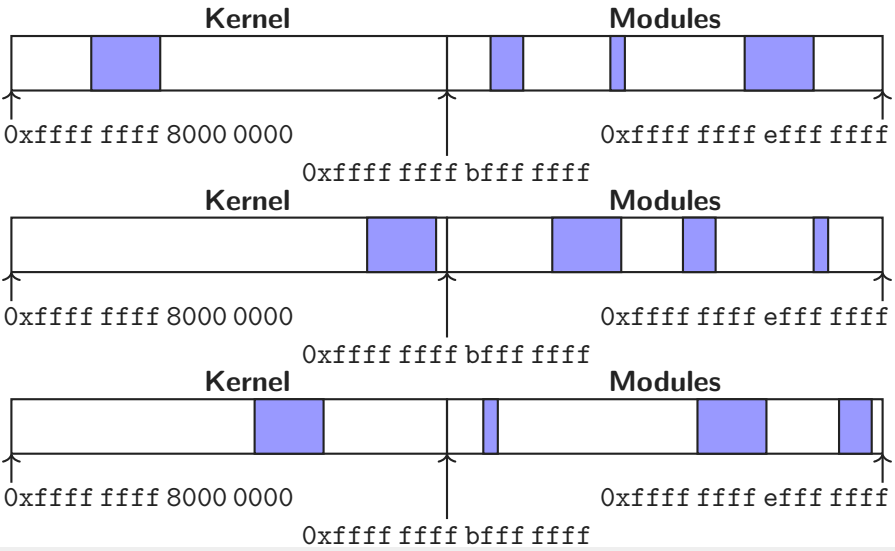
Store-to-load Forwarding on Inaccessible Addresses

Store-to-load forwarding?

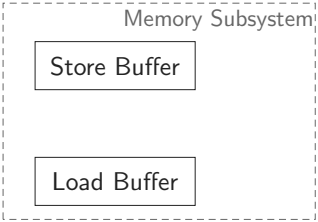



Kernel
Modules
0xffff ffff 8000 0000
0xffff ffff bfff ffff
0xffff ffff efff ffff

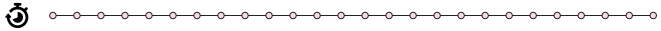

Kernel
Modules
0xffff ffff 8000 0000
0xffff ffff bfff ffff
0xffff ffff efff ffff
Kernel
Modules
0xffff ffff 8000 0000
0xffff ffff bfff ffff
0xffff ffff efff ffff

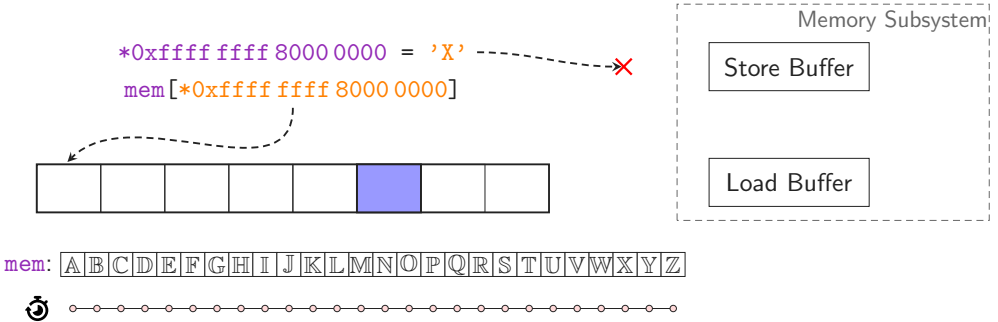

Kernel
Modules
0xffff ffff 8000 0000
0xffff ffff bfff ffff
0xffff ffff efff ffff
Kernel
Modules
0xffff ffff 8000 0000
0xffff ffff bfff ffff
0xffff ffff efff ffff
Kernel
Modules
0xffff ffff 8000 0000
0xffff ffff bfff ffff
0xffff ffff efff ffff

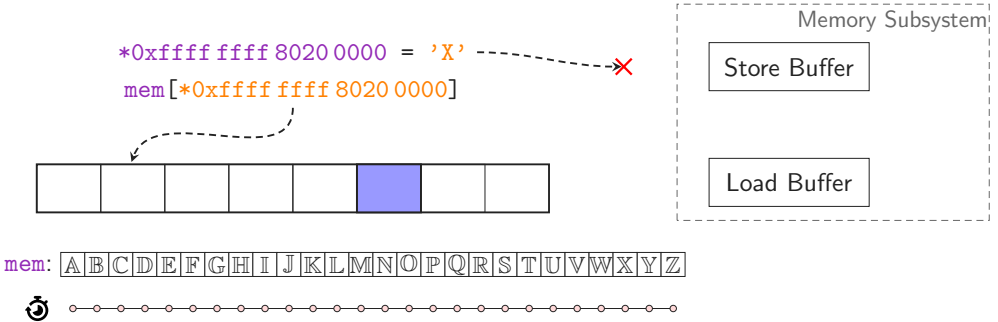
```
*0xffff ffff 8000 0000 = 'X'
mem[*0xffff ffff 8000 0000]
```

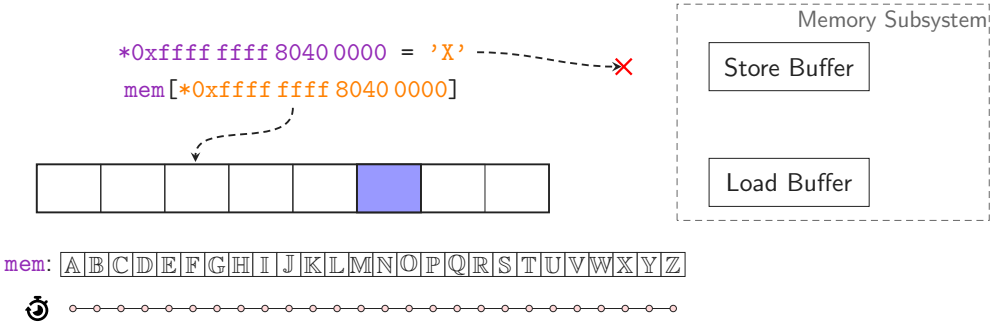


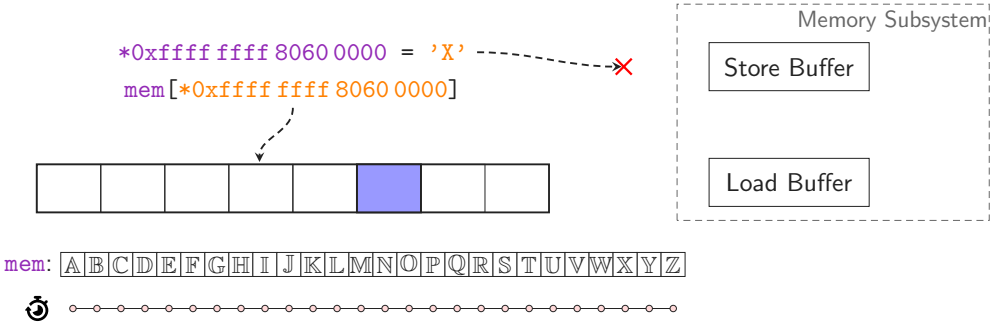
mem: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

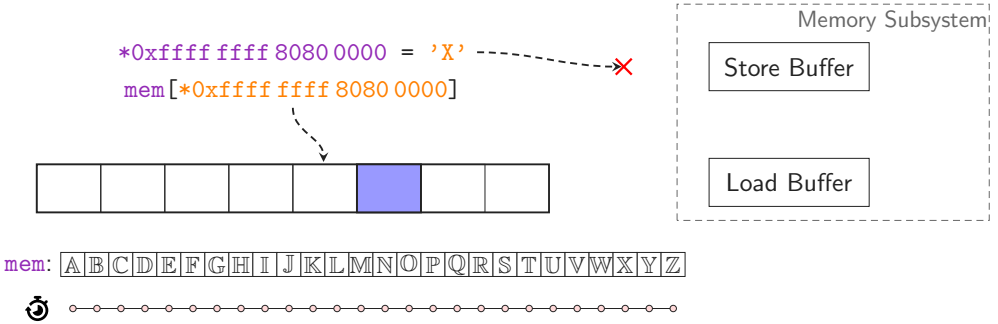


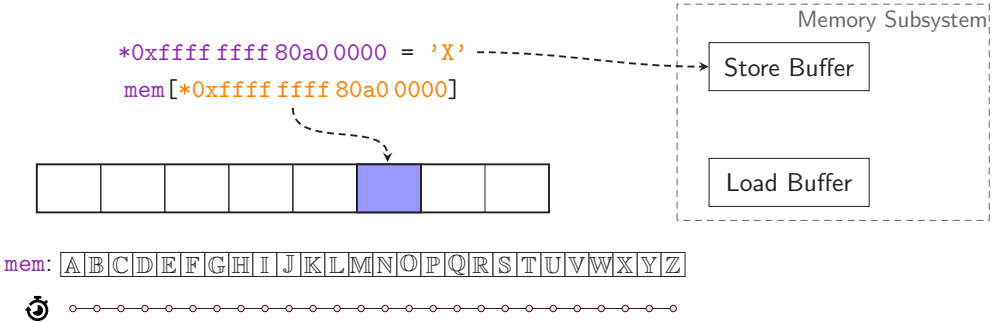

*0xffff ffff 8000 0000 = 'X'
mem[*0xffff ffff 8000 0000]
Memory Subsystem
Store Buffer
Load Buffer
mem: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z


*0xffff ffff 8020 0000 = 'X'
mem[*0xffff ffff 8020 0000]
Memory Subsystem
Store Buffer
Load Buffer
mem: 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

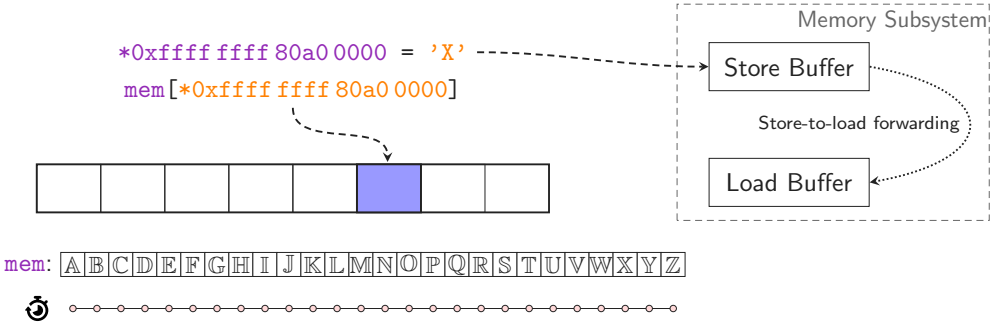

*0xffff ffff 8040 0000 = 'X'
mem[*0xffff ffff 8040 0000]
Memory Subsystem
Store Buffer
Load Buffer
mem: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

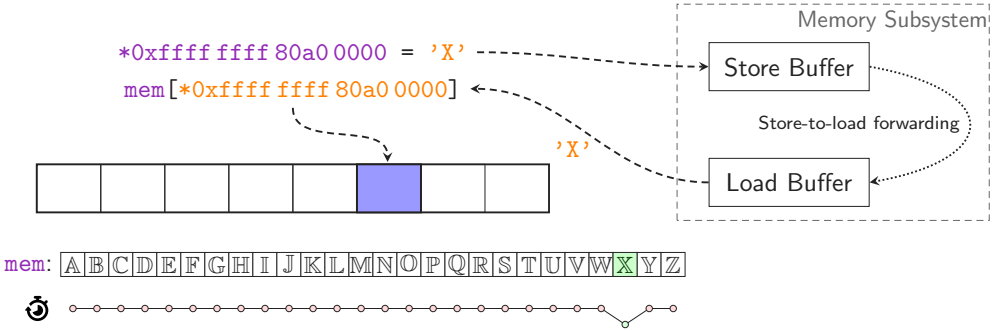

*0xffff ffff 8060 0000 = 'X'
mem[*0xffff ffff 8060 0000]
Memory Subsystem
Store Buffer
Load Buffer
mem: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

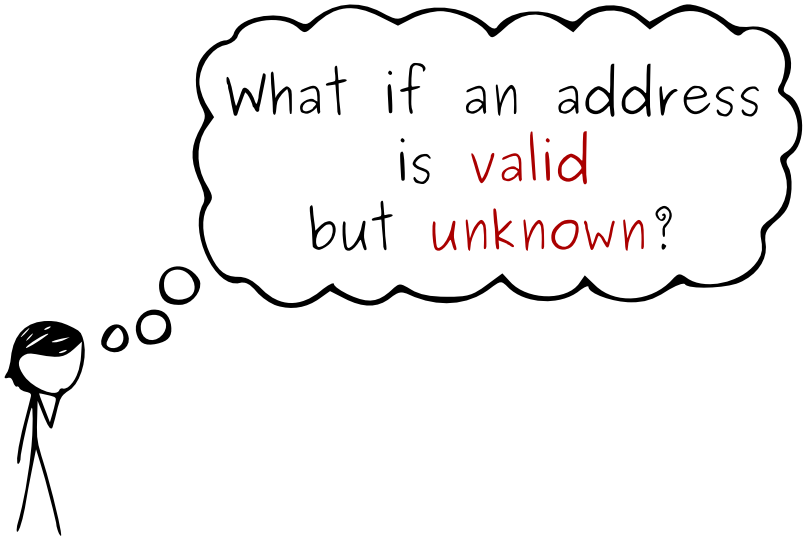

*0xffff ffff 8080 0000 = 'X'
mem[*0xffff ffff 8080 0000]
Memory Subsystem
Store Buffer
Load Buffer
mem: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

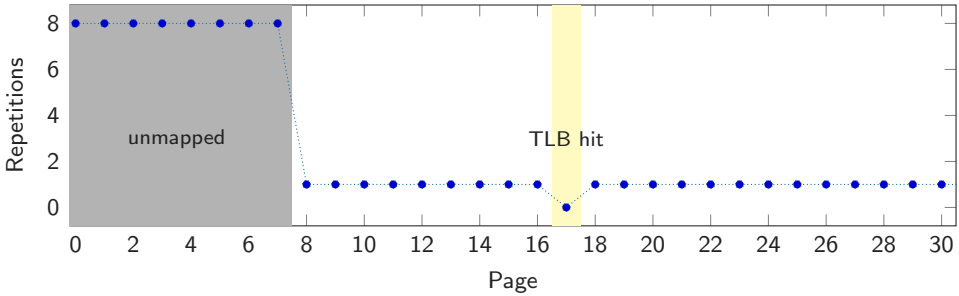

*0xffff ffff 80a0 0000 = 'X'
mem[*0xffff ffff 80a0 0000]
Memory Subsystem
Store Buffer
Load Buffer
mem: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Data Bounce

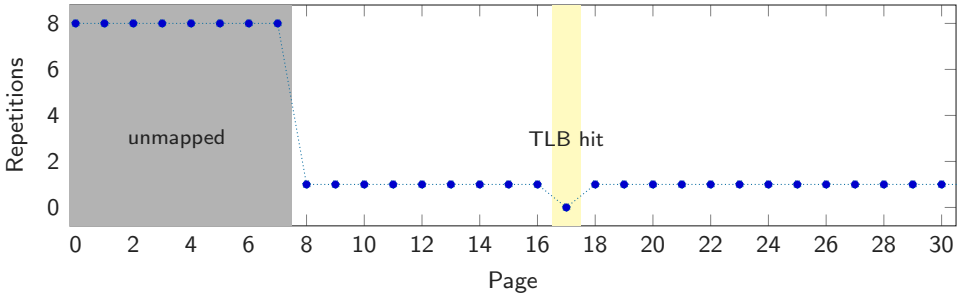

*0xffff ffff 80a0 0000 = 'X'
mem[*0xffff ffff 80a0 0000]
Memory Subsystem
Store Buffer
Store-to-load forwarding
Load Buffer
mem: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z


*0xffff ffff 80a0 0000 = 'X'
mem[*0xffff ffff 80a0 0000]
'X'
Memory Subsystem
Store Buffer
Store-to-load forwarding
Load Buffer
mem: 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

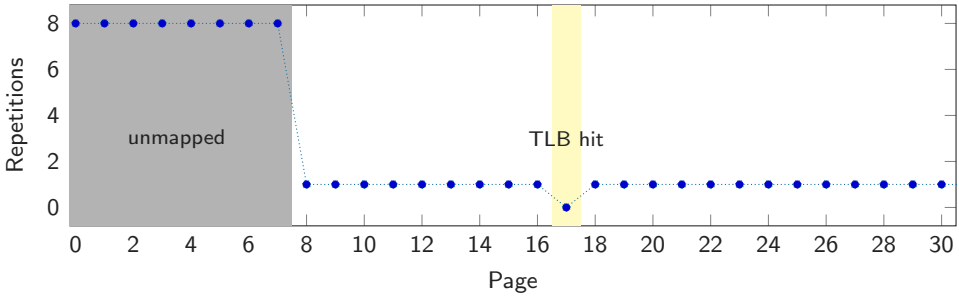

What if an address
is valid
but unknown?



- Forward on first try → in TLB

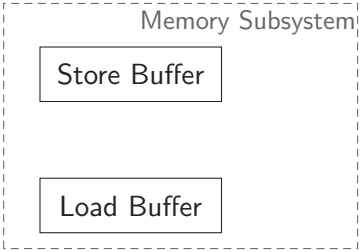


- Forward on first try → in TLB
- On second try → valid, not in TLB

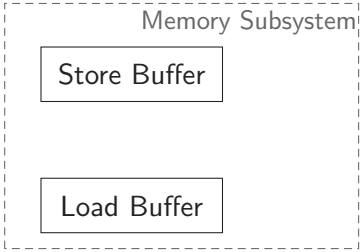


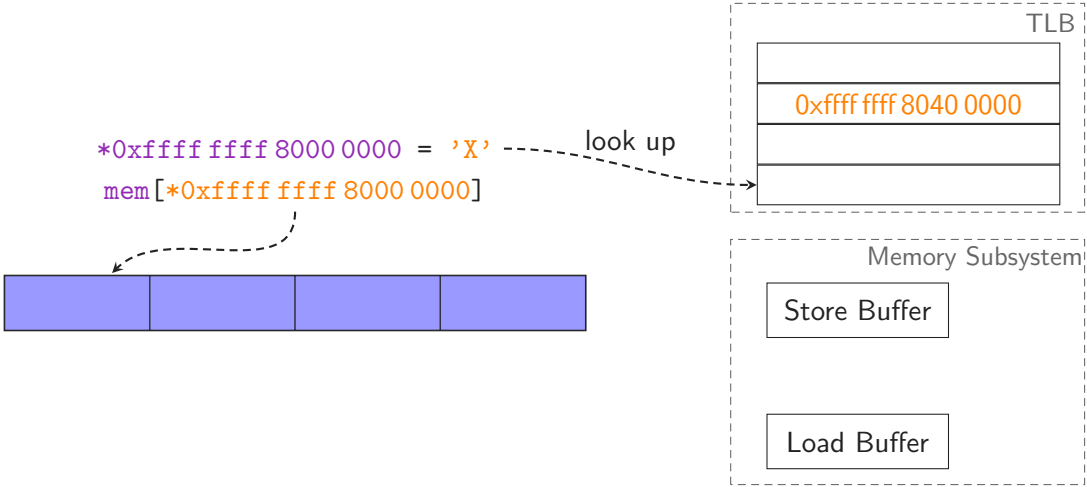
- Forward on first try → in TLB
- On second try → valid, not in TLB
- Not forwarded → unmapped

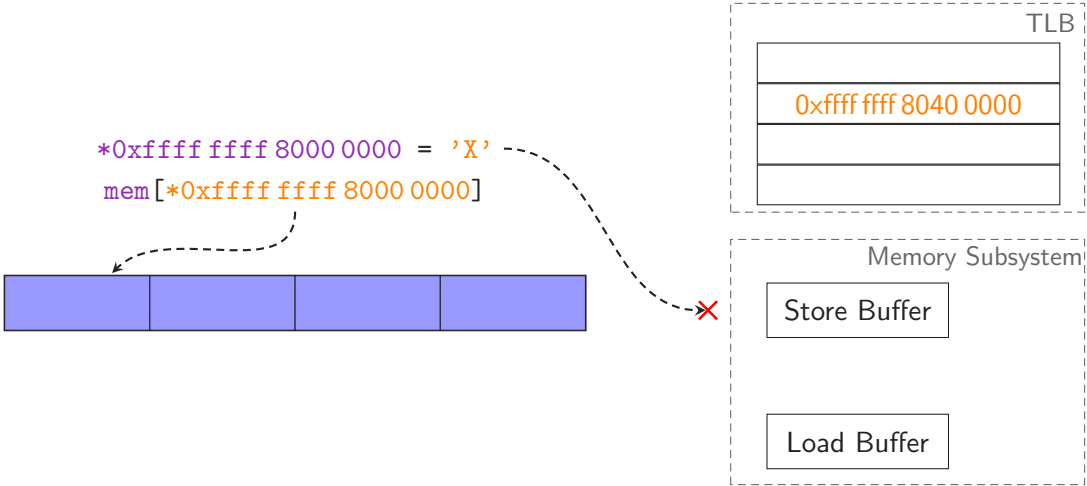
```
*0xffff ffff 8000 0000 = 'X'
mem[*0xffff ffff 8000 0000]
```



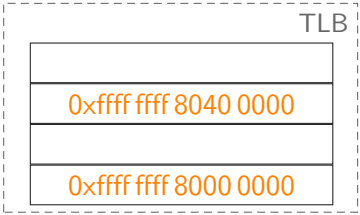
```
*0xffff ffff 8000 0000 = 'X'
mem[*0xffff ffff 8000 0000]
```

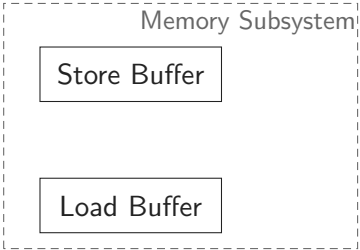



*0xffff ffff 8000 0000 = 'X'
mem[*0xffff ffff 8000 0000]
look up
TLB
0xffff ffff 8040 0000
Memory Subsystem
Store Buffer
Load Buffer

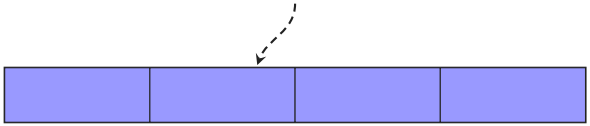

TLB
0xffff ffff 8040 0000
*0xffff ffff 8000 0000 = 'X'
mem[*0xffff ffff 8000 0000]
Memory Subsystem
Store Buffer
Load Buffer

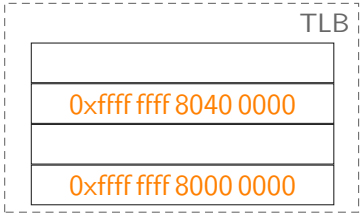
```
*0xffff ffff 8000 0000 = 'X'
mem[*0xffff ffff 8000 0000]
```

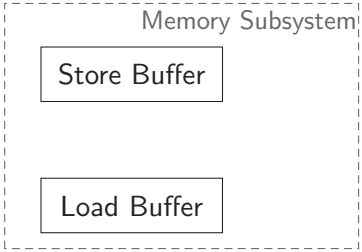


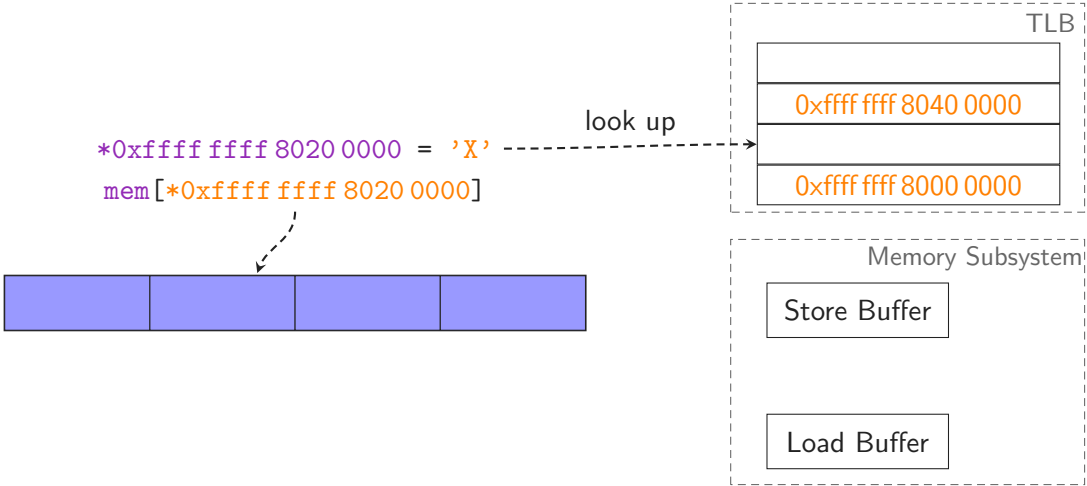


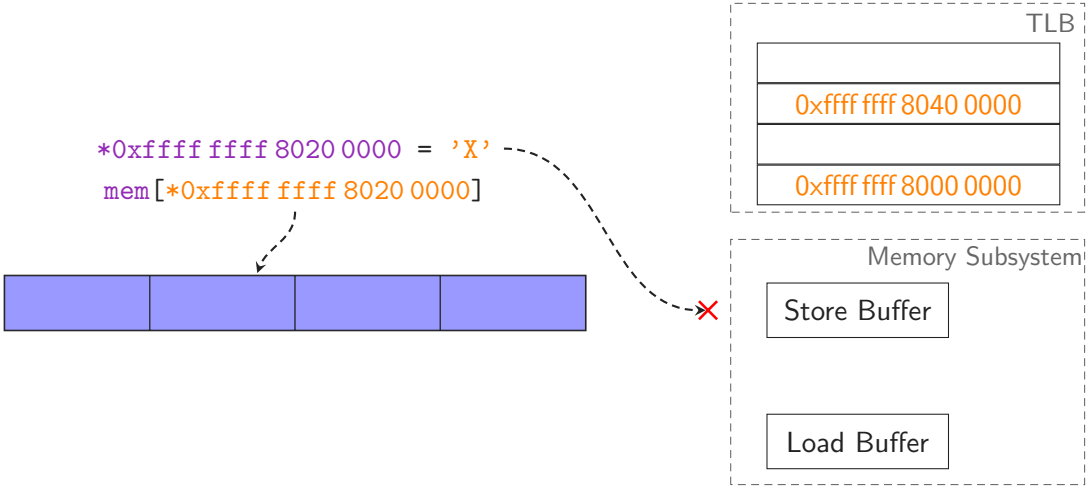
```
*0xffff ffff 8020 0000 = 'X'
mem[*0xffff ffff 8020 0000]
```



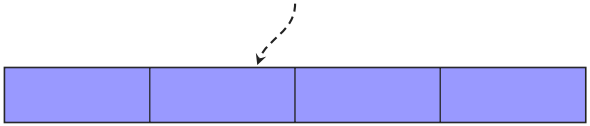


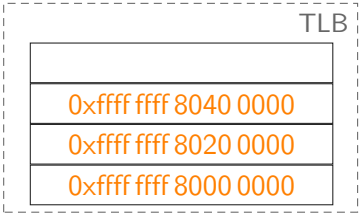


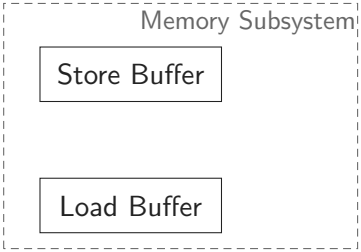

*0xffff ffff 8020 0000 = 'X'
mem[*0xffff ffff 8020 0000]
look up
TLB
0xffff ffff 8040 0000
0xffff ffff 8000 0000
Memory Subsystem
Store Buffer
Load Buffer


*0xffff ffff 8020 0000 = 'X'
mem[*0xffff ffff 8020 0000]
TLB
0xffff ffff 8040 0000
0xffff ffff 8000 0000
Memory Subsystem
Store Buffer
Load Buffer

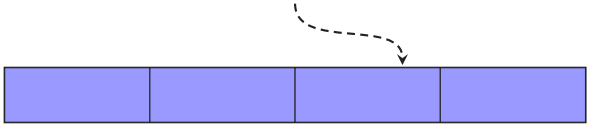
```
*0xffff ffff 8020 0000 = 'X'
mem[*0xffff ffff 8020 0000]
```

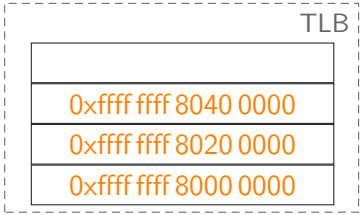


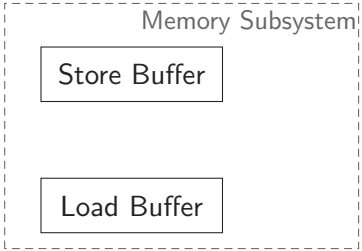


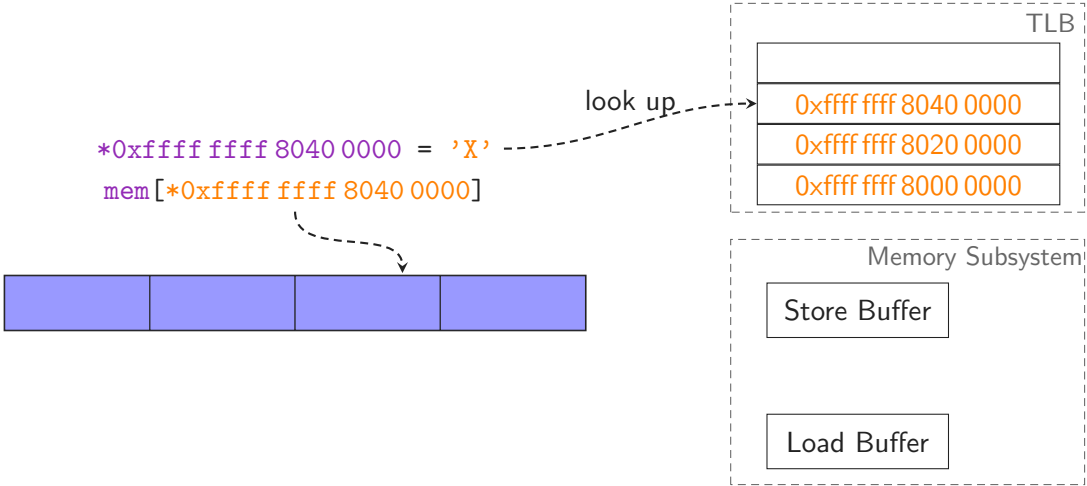


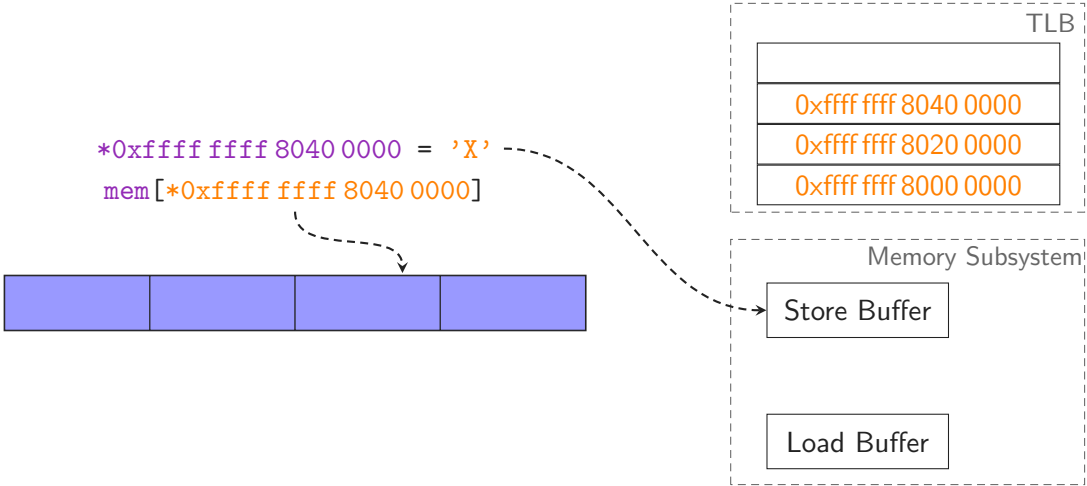
```
*0xffff ffff 8040 0000 = 'X'
mem[*0xffff ffff 8040 0000]
```

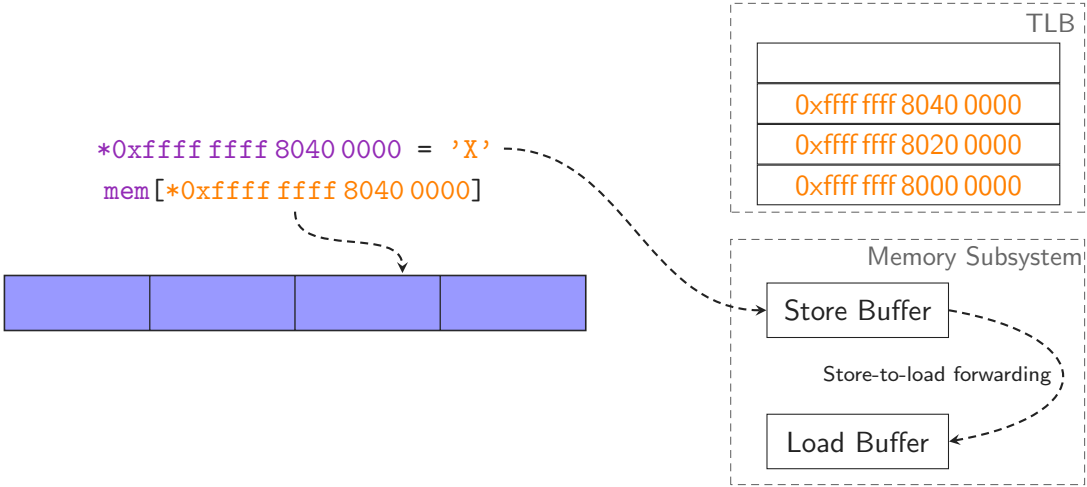


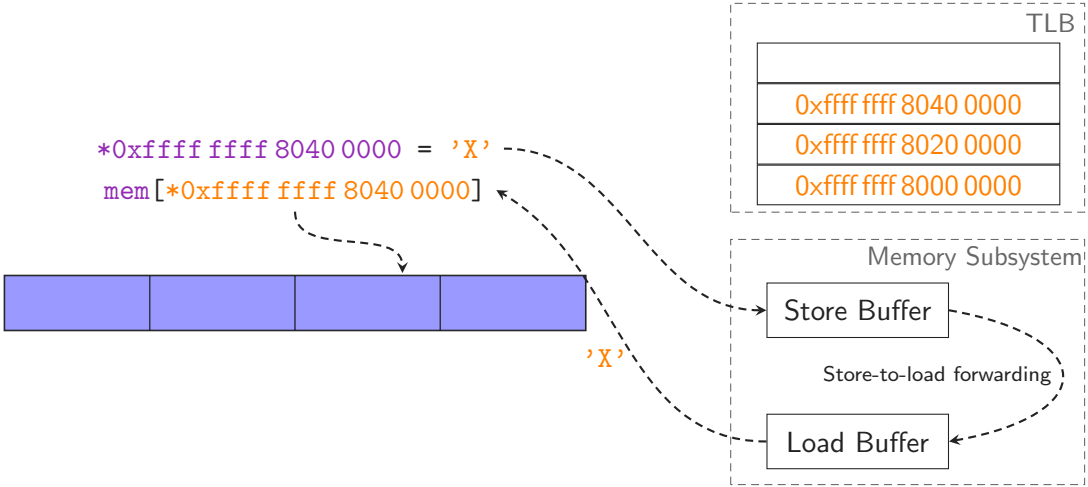




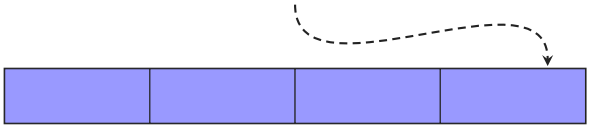

TLB
look up
*0xffff ffff 8040 0000 = 'X'
mem[*0xffff ffff 8040 0000]
0xffff ffff 8040 0000
0xffff ffff 8020 0000
0xffff ffff 8000 0000
Memory Subsystem
Store Buffer
Load Buffer

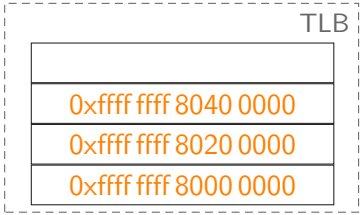

TLB
0xffff ffff 8040 0000
0xffff ffff 8020 0000
0xffff ffff 8000 0000
*0xffff ffff 8040 0000 = 'X'
mem[*0xffff ffff 8040 0000]
Memory Subsystem
Store Buffer
Load Buffer

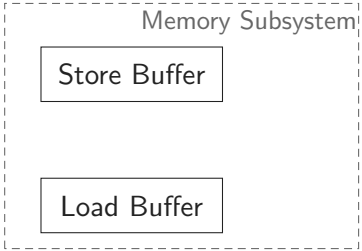

*0xffff ffff 8040 0000 = 'X'
mem[*0xffff ffff 8040 0000]
TLB
0xffff ffff 8040 0000
0xffff ffff 8020 0000
0xffff ffff 8000 0000
Memory Subsystem
Store Buffer
Store-to-load forwarding
Load Buffer

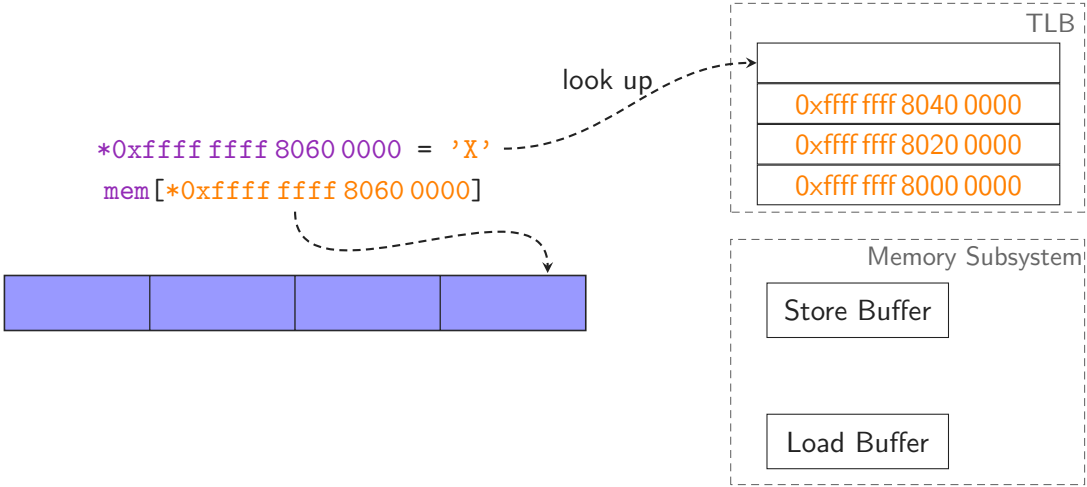

TLB
0xffff ffff 8040 0000
0xffff ffff 8020 0000
0xffff ffff 8000 0000
*0xffff ffff 8040 0000 = 'X'
mem[*0xffff ffff 8040 0000]
'X'
Memory Subsystem
Store Buffer
Store-to-load forwarding
Load Buffer

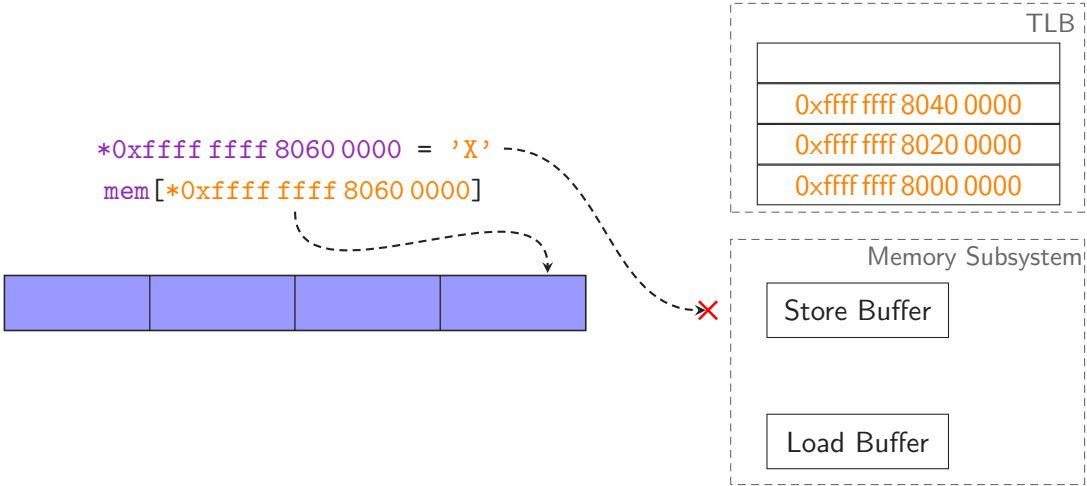
```
*0xffff ffff 8060 0000 = 'X'
mem[*0xffff ffff 8060 0000]
```



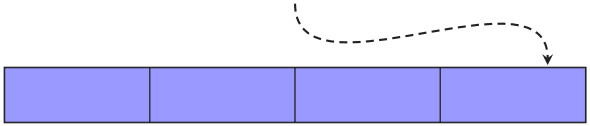


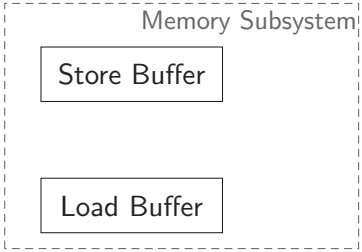


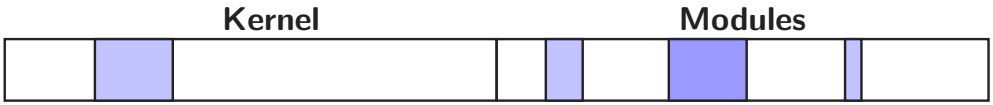

TLB
look up
*0xffff ffff 8060 0000 = 'X'
mem[*0xffff ffff 8060 0000]
0xffff ffff 8040 0000
0xffff ffff 8020 0000
0xffff ffff 8000 0000
Memory Subsystem
Store Buffer
Load Buffer

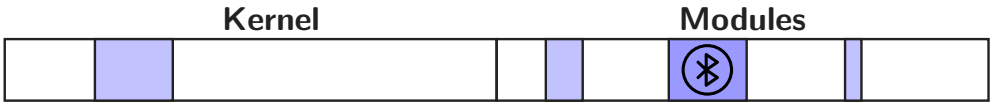

*0xffff ffff 8060 0000 = 'X'
mem[*0xffff ffff 8060 0000]
TLB
0xffff ffff 8040 0000
0xffff ffff 8020 0000
0xffff ffff 8000 0000
Memory Subsystem
Store Buffer
Load Buffer

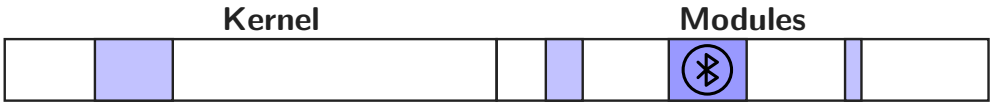
```
*0xffff ffff 8060 0000 = 'X'
mem[*0xffff ffff 8060 0000]
```

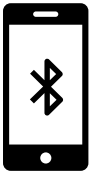


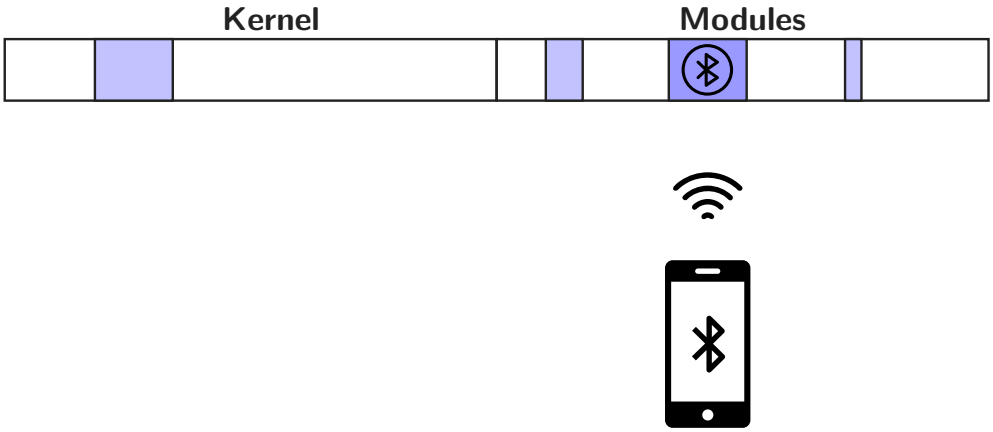


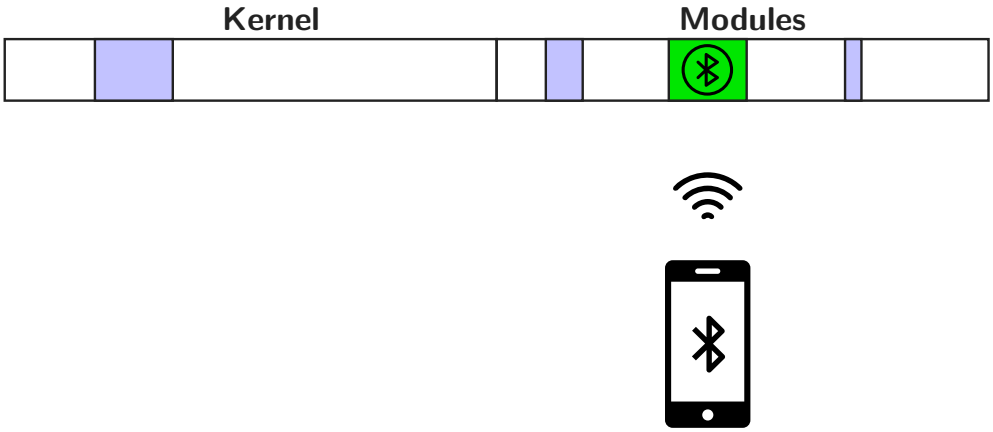

Kernel
Modules

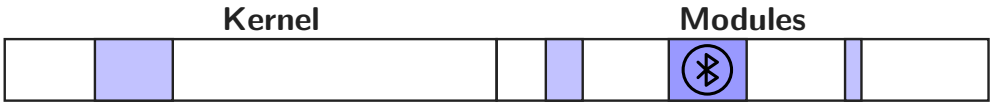

Kernel
Modules

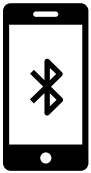

Kernel
Modules



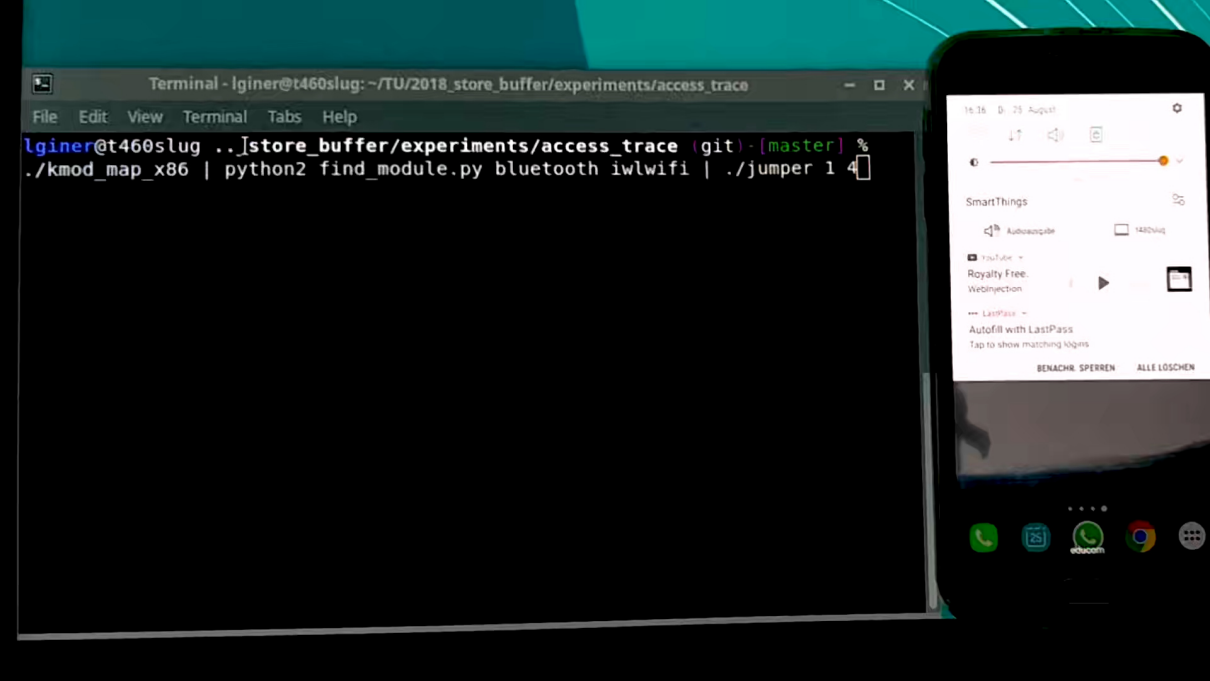

Kernel
Modules


Kernel
Modules

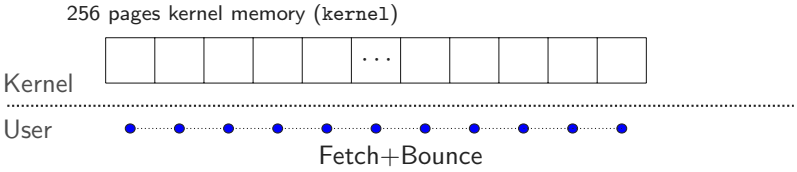

Kernel
Modules



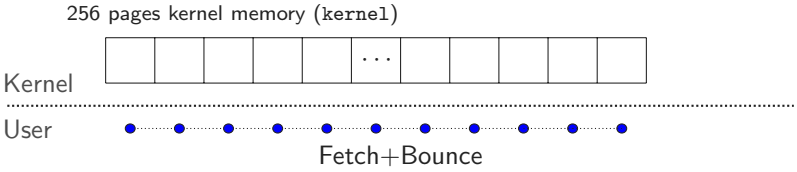
```
lginer@t460slug ../_store_buffer/experiments/access_trace (git)-[master] % ./kmod_map_x86
```

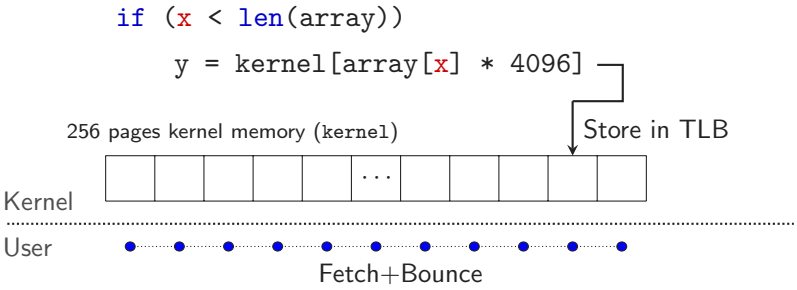

Terminal - lginer@t460slug: ~/TU/2018_store_buffer/experiments/access_trace
File Edit View Terminal Tabs Help
lginer@t460slug ..store_buffer/experiments/access_trace (git)-[master] %
./kmod_map_x86 | python2 find_module.py bluetooth iwlwifi | ./jumper 1 4
SmartThings
Royalty Free.
Webinjection
Autofill with LastPass
Tap to show matching logins
BENACHR. SPERREN
ALLE LÖSCHEN

Speculative Fetch+Bounce

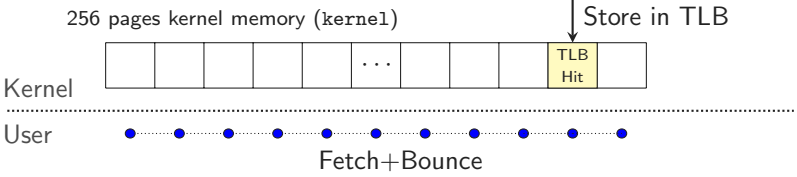


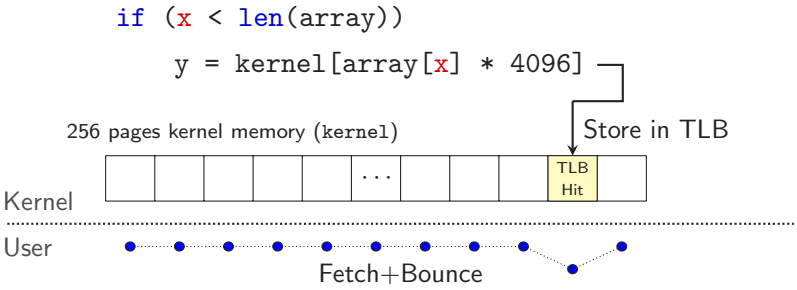
```
if (x < len(array))
    y = kernel[array[x] * 4096]
```

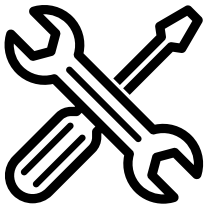



if (x < len(array))
y = kernel[array[x] * 4096]
Store in TLB
256 pages kernel memory (kernel)
Kernel
User
Fetch+Bounce

```
if (x < len(array))
    y = kernel[array[x] * 4096]
```

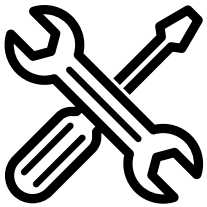



if (x < len(array))
y = kernel[array[x] * 4096]
Store in TLB
256 pages kernel memory (kernel)
TLB
Hit
Kernel
User
Fetch+Bounce



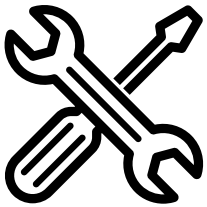
Assumptions

1. Stalling CPU might be too costly



Assumptions

1. Stalling CPU might be too costly
2. Stalling might require redesigning parts of CPU pipeline



Assumptions

1. Stalling CPU might be too costly
2. Stalling might require redesigning parts of CPU pipeline

Hypothesis

Load is executed, returned value is zeroed out on faults

Two tests:

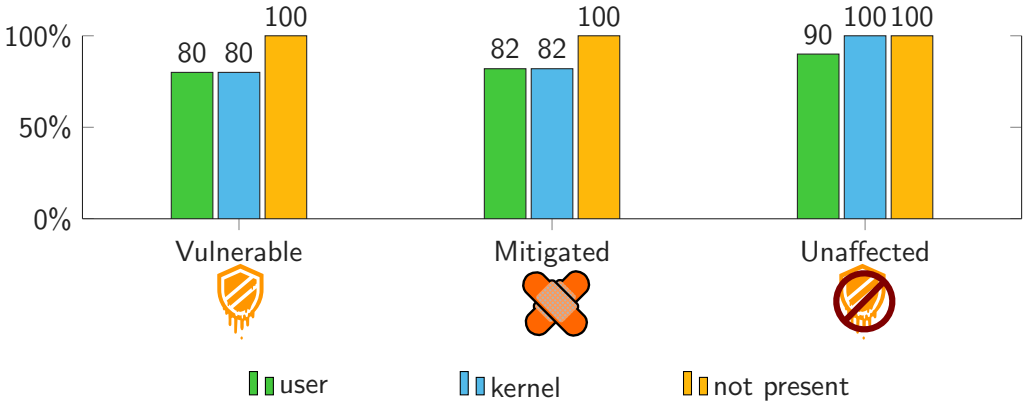
1. Perform Meltdown attack

Two tests:

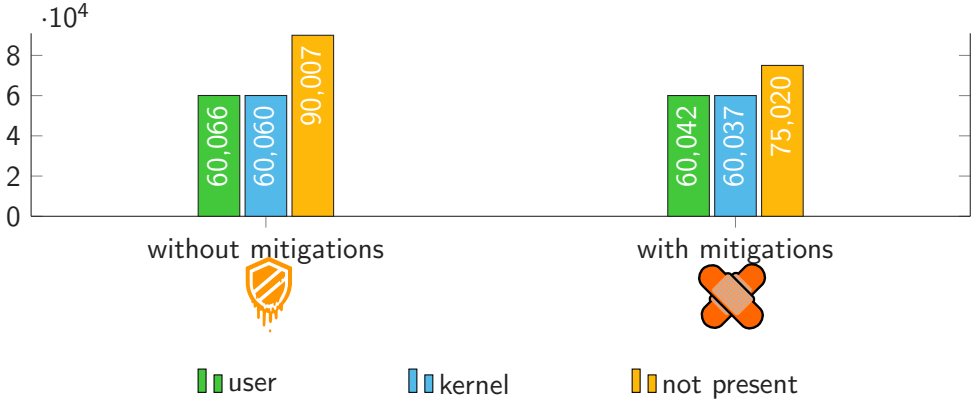
1. Perform Meltdown attack
2. Use Performance Counters

Verifying the Hypothesis

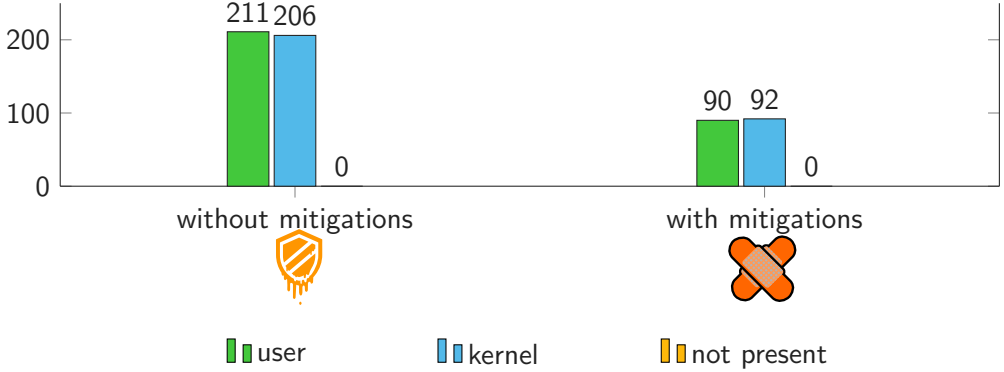
- Intel: CYCLE_ACTIVITY.STALLS_MEM_ANY
- AMD: Dispatch Stalls

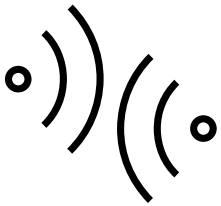


- Track number of issued μOPs on the load ports
- UOPS_DISPATCHED_PORT.PORT_2, UOPS_DISPATCHED_PORT.PORT_3

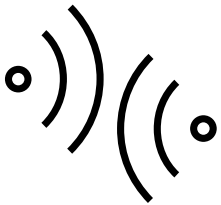


- L1D_PEND_MISS.PENDING_CYCLES

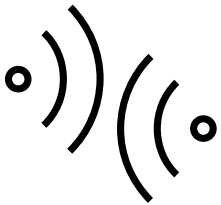




- Can we use the hardware-based mitigations for an attack?

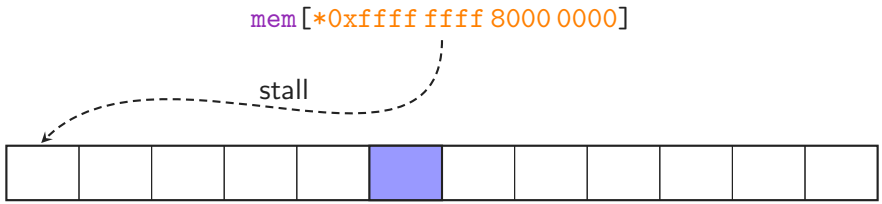


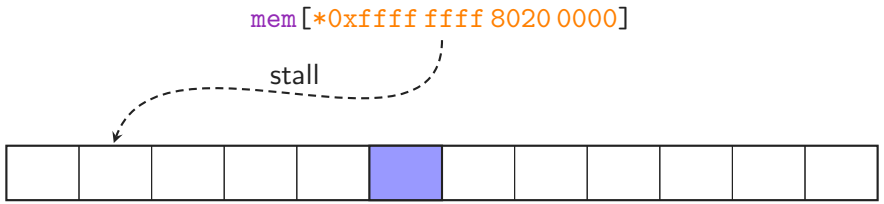
- Can we use the hardware-based mitigations for an attack?
- EchoLoad: fast and reliable KASLR break

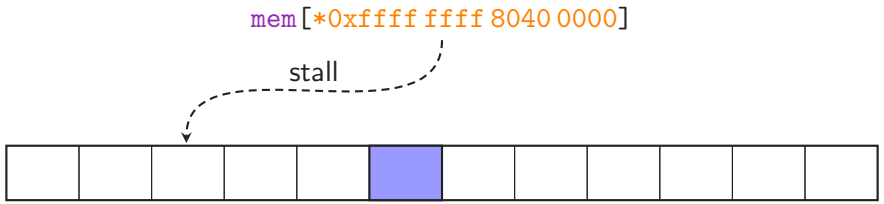


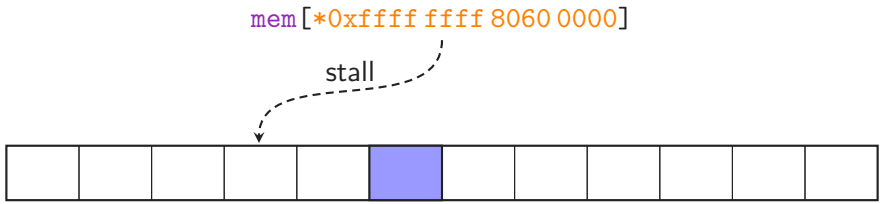
- Can we use the hardware-based mitigations for an attack?
- EchoLoad: fast and reliable KASLR break
- Encodes the returned value in the cache

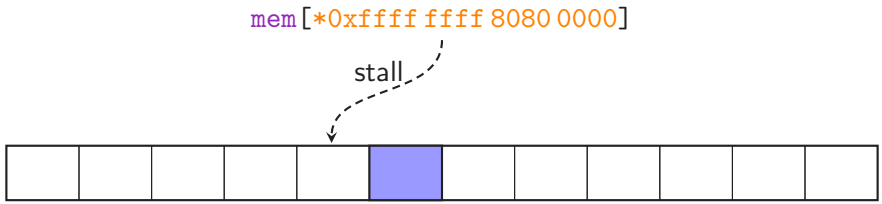
```
mem[*0xffff ffff 8000 0000]
```

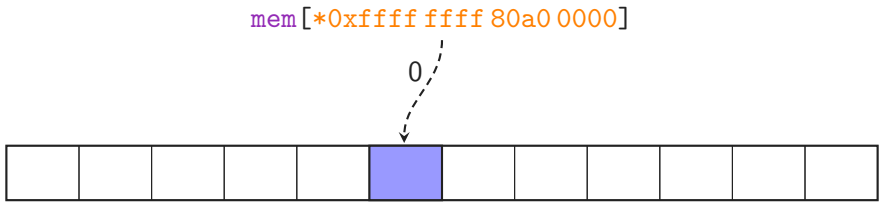

mem[*0xffff ffff 8000 0000]
stall


mem[*0xffff ffff 8020 0000]
stall


mem[*0xffff ffff 8040 0000]
stall


mem[*0xffff ffff 8060 0000]
stall

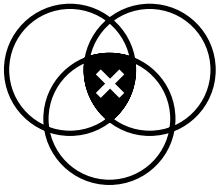

mem[*0xffff ffff 8080 0000]
stall


mem[*0xffff ffff 80a0 0000]
0

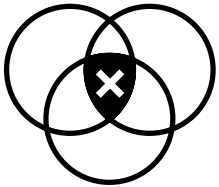
CPU		Speculation		TSX		Segfault	
i7-6700K	Time (F-Score)	63 µs	(0.999)	48 µs	(1.000)	133 µs	(1.000)
i9-9900K	Time (F-Score)	33 µs	(1.000)	29 µs	(1.000)	86 µs	(1.000)
Xeon Silver 4208	Time (F-Score)	51 µs	(0.994)	40 µs	(1.000)	127 µs	(1.000)

CPU		Speculation		TSX		Segfault	
i7-6700K	Time (F-Score)	63 µs	(0.999)	48 µs	(1.000)	133 µs	(1.000)
i9-9900K	Time (F-Score)	33 µs	(1.000)	29 µs	(1.000)	86 µs	(1.000)
Xeon Silver 4208	Time (F-Score)	51 µs	(0.994)	40 µs	(1.000)	127 µs	(1.000)

- Works in SGX and JavaScript

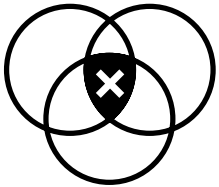


Timing difference



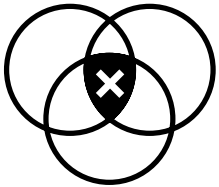
Timing difference

- between mapped and unmapped pages



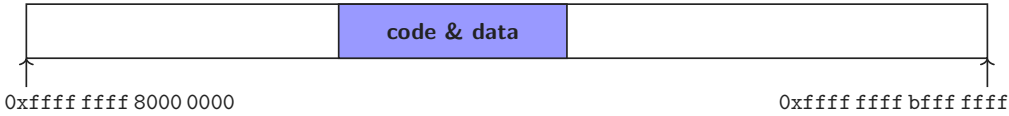
Timing difference

- between mapped and unmapped pages
- for different page sizes

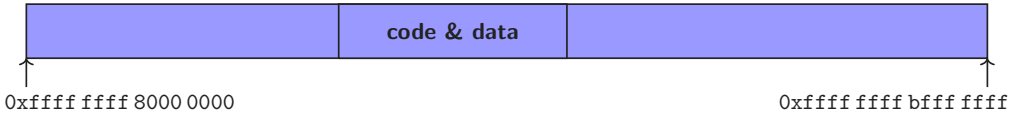


Timing difference

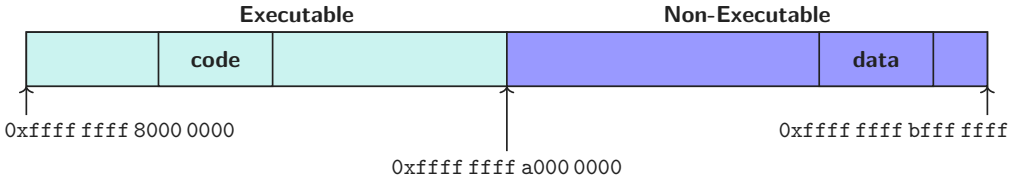
- between mapped and unmapped pages
- for different page sizes
- between executable and non-executable pages



Current Linux design



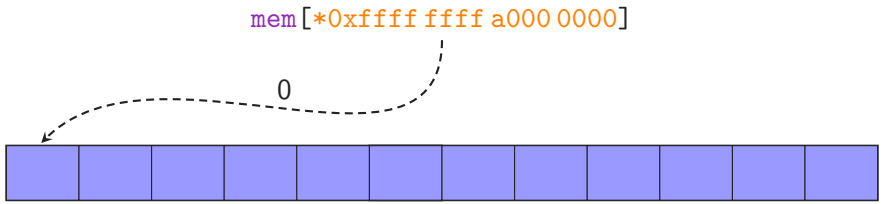
Step 1: Mitigating difference between mapped and unmapped pages

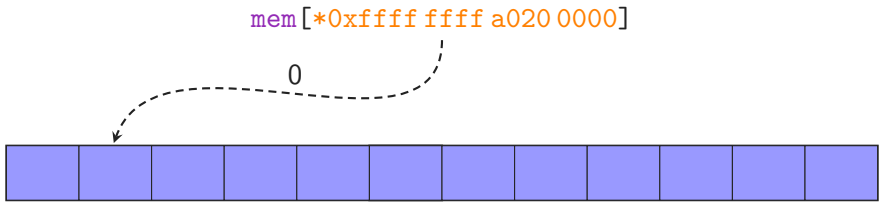


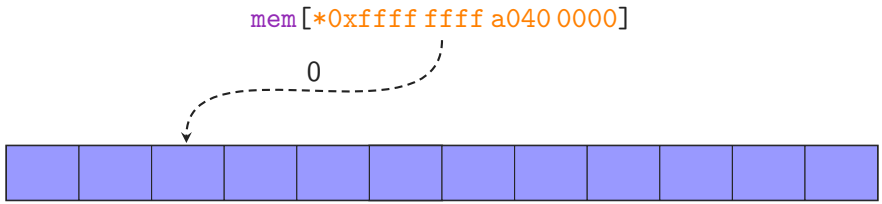
Step 2: Mitigating difference between executable and non-executable pages

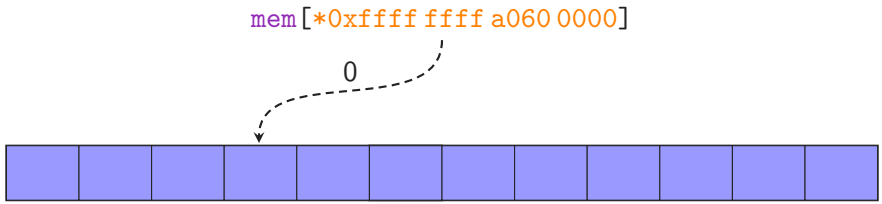
```
mem[*0xffff ffff a000 0000]
```

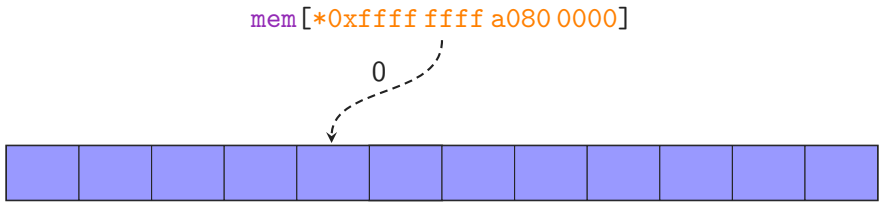
```
mem[*0xffff ffff a000 0000]
```

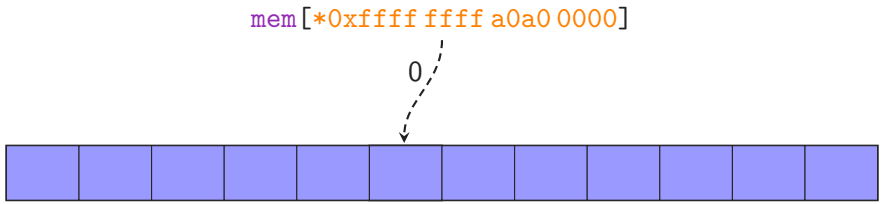

mem[*0xffff ffff a000 0000]
0

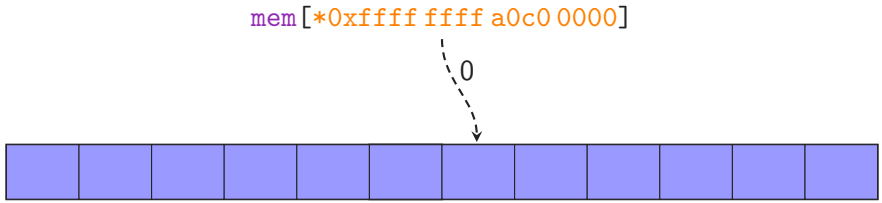

mem[*0xffff ffff a020 0000]
0

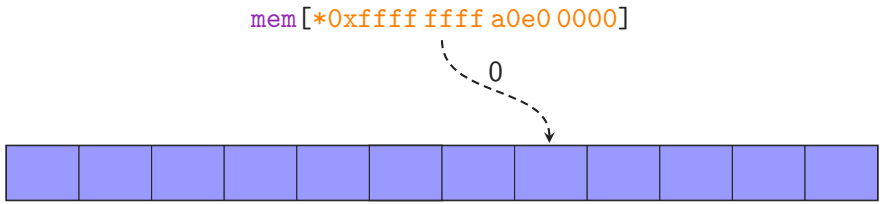

mem[*0xffff ffff a040 0000]
0

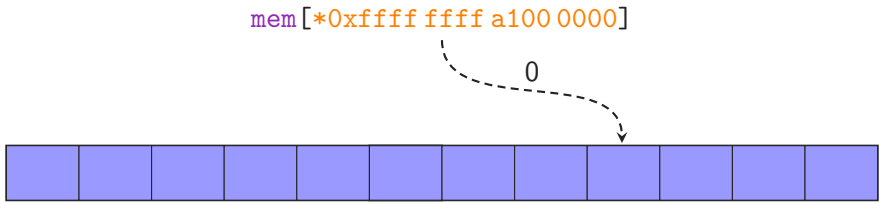

mem[*0xffff ffff a060 0000]
0

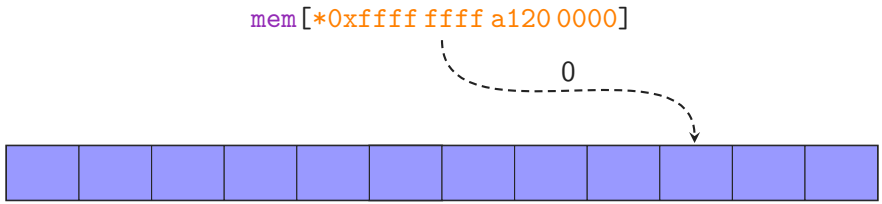

mem[*0xffff ffff a080 0000]
0

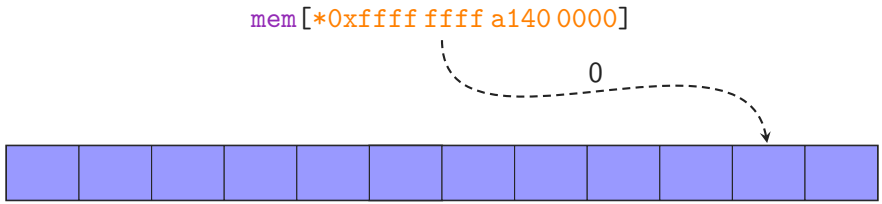

mem[*0xffff ffff a0a0 0000]
0

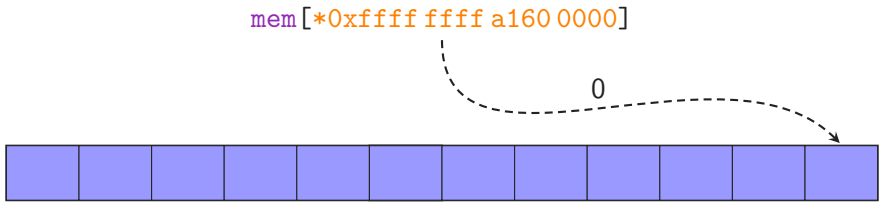

mem[*0xffff ffff a0c0 0000]
0

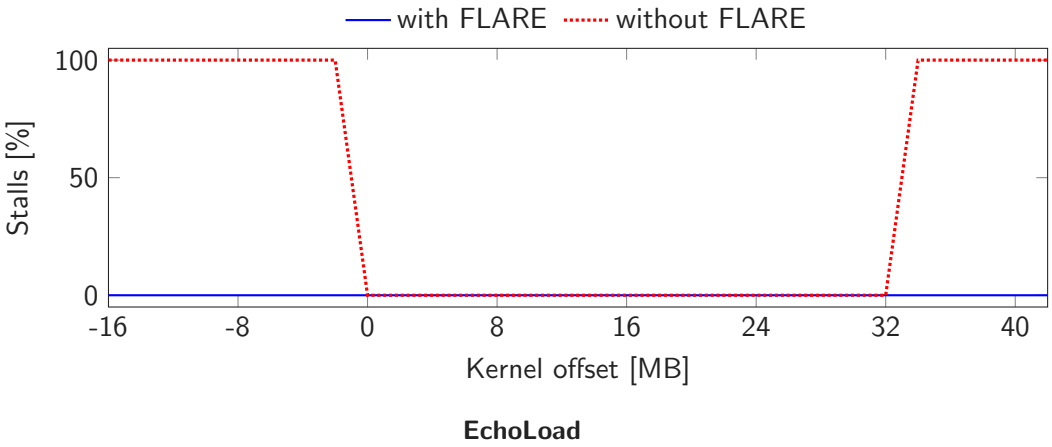

mem[*0xffff ffff a0e0 0000]
0


mem[*0xffff ffff a100 0000]
0

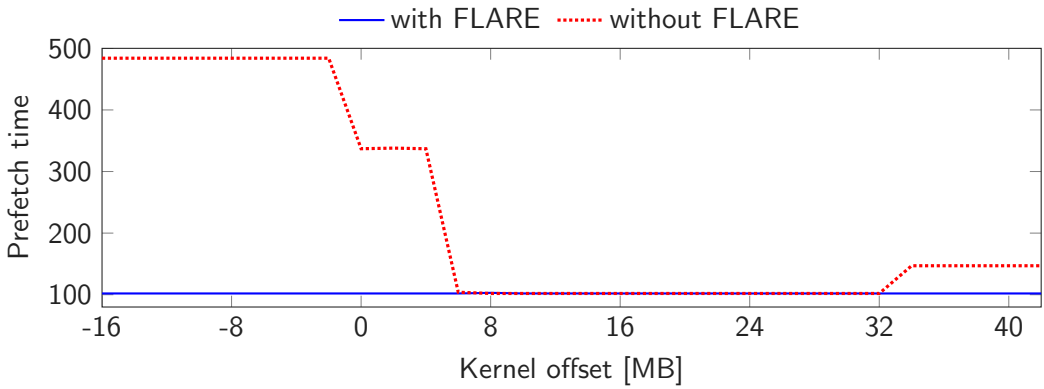

mem[*0xffff ffff a120 0000]
0


mem[*0xffff ffff a140 0000]
0

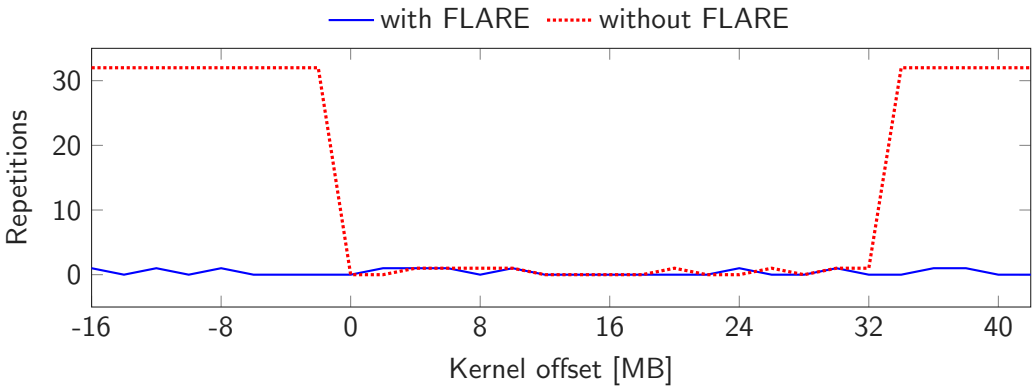

mem[*0xffff ffff a160 0000]
0


with FLARE
without FLARE
100
50
0
Stalls [%]
-16
-8
0
8
16
24
32
40
Kernel offset [MB]

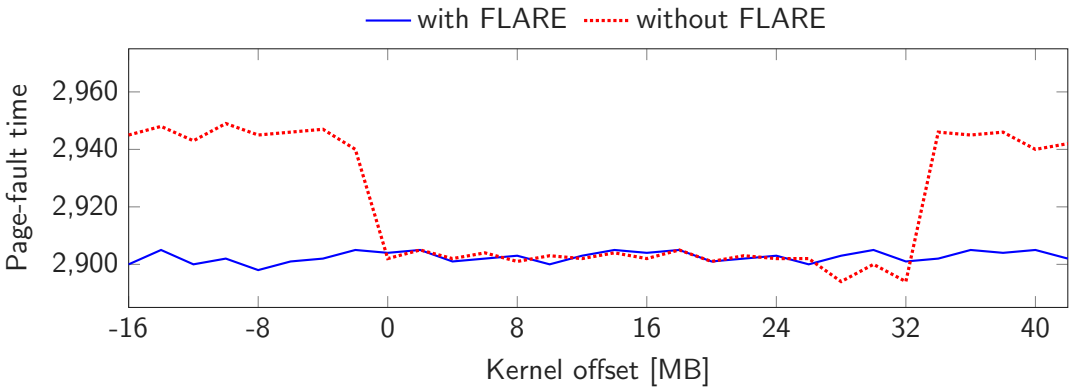
EchoLoad



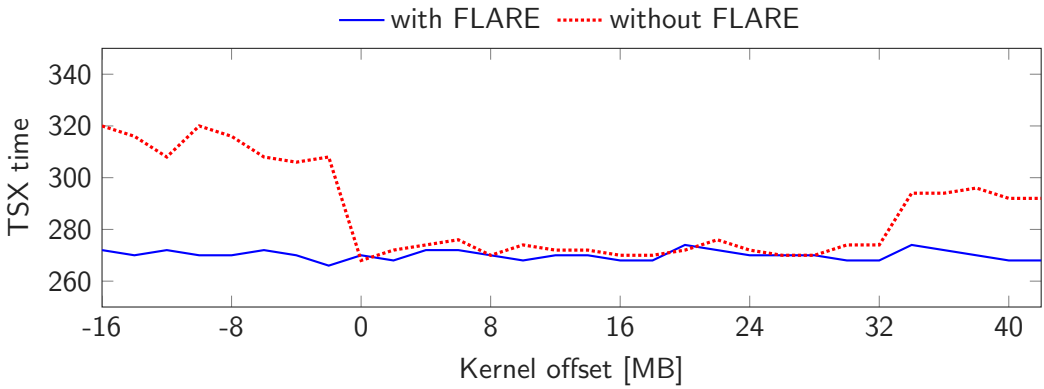
Prefetch [Gru+16]



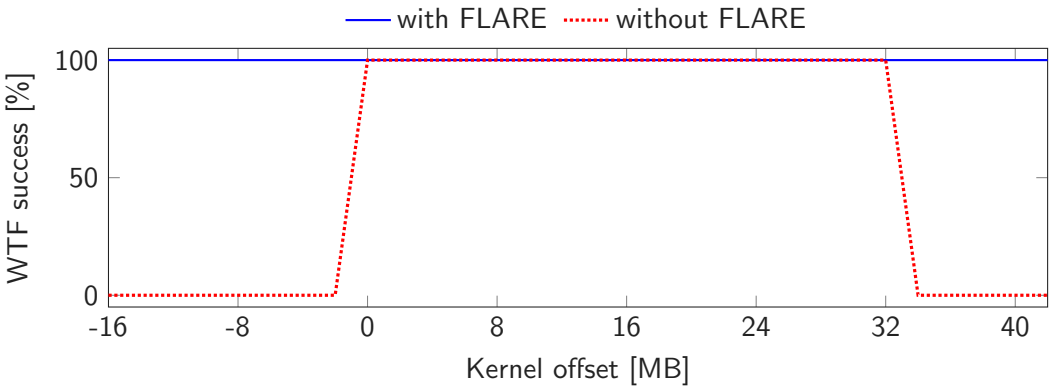
Data Bounce [Sch+19]



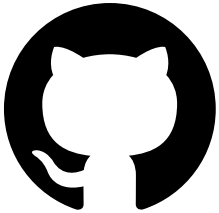
Double page fault [HWH13]


with FLARE
without FLARE
TSX time
340
320
300
280
260
-16
-8
0
8
16
24
32
40
Kernel offset [MB]

DrK [JLK16]



Fallout [Can+19]



You can find our proof-of-concept implementation of FLARE on:

- `https://github.com/IAIK/FLARE`

- Optimizations introduce security problems

- Optimizations introduce security problems
- Mitigations can overlook edge-cases

- Optimizations introduce security problems
- Mitigations can overlook edge-cases
- Once again requires software workaround

Store-to-Leak Forwarding

There and Back Again

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte)

October 2, 2020

Graz University of Technology

References

C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In: CCS. 2019.

D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In: CCS. 2016.

R. Hund, C. Willems, and T. Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In: S&P. 2013.

Y. Jang, S. Lee, and T. Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In: CCS. 2016.

M. Schwarz, C. Canella, L. Giner, and D. Gruss. Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs. In: arXiv:1905.05725 (2019).