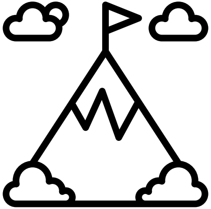
Speculative Dereferencing: Reviving Foreshadow

Martin Schwarzl (@marv0x90), Thomas Schuster, Daniel Gruss, Michael Schwarz

1st of March, 2021

Graz University of Technology

Motivation

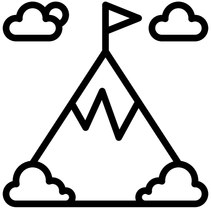


- Analysis of address-translation attack by Gruss et al. [Gru+16]

Motivation

- Analysis of address-translation attack by Gruss et al. [Gru+16]
- The effect was attributed to the prefetch instruction

Motivation



- Analysis of address-translation attack by Gruss et al. [Gru+16]
- The effect was attributed to the prefetch instruction
- Show that the actual root-cause is speculative execution in the kernel

Motivation

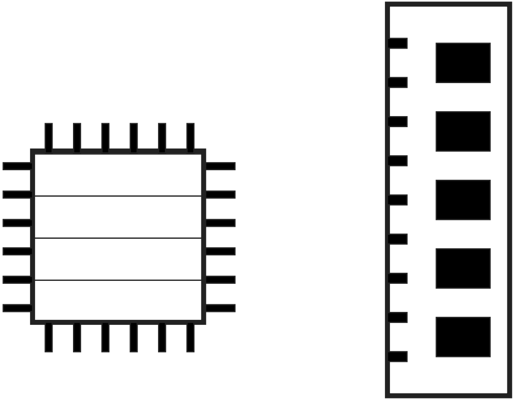
- Analysis of address-translation attack by Gruss et al. [Gru+16]
- The effect was attributed to the prefetch instruction
- Show that the actual root-cause is speculative execution in the kernel
- This misattribution led to wrong conclusions in follow-up work

Motivation

- Analysis of address-translation attack by Gruss et al. [Gru+16]
- The effect was attributed to the prefetch instruction
- Show that the actual root-cause is speculative execution in the kernel
- This misattribution led to wrong conclusions in follow-up work
- We present stronger attacks like reviving Foreshadow

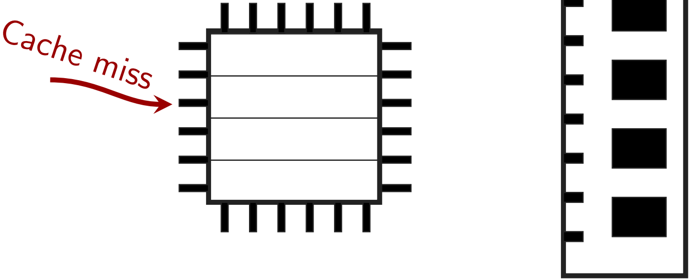
CPU Cache

```
maccess(i);

maccess(i);
```



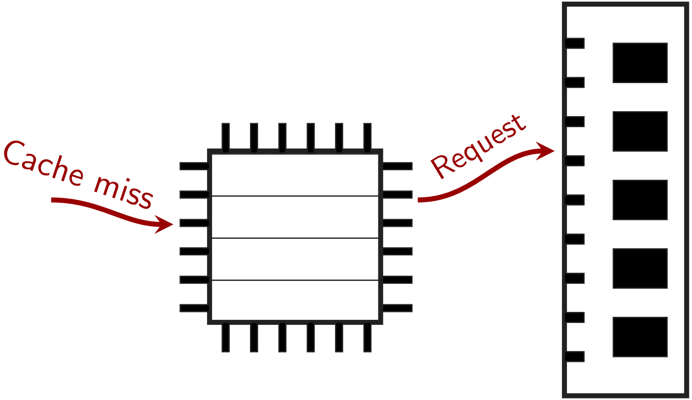
CPU Cache

```
maccess(i);

maccess(i);
```



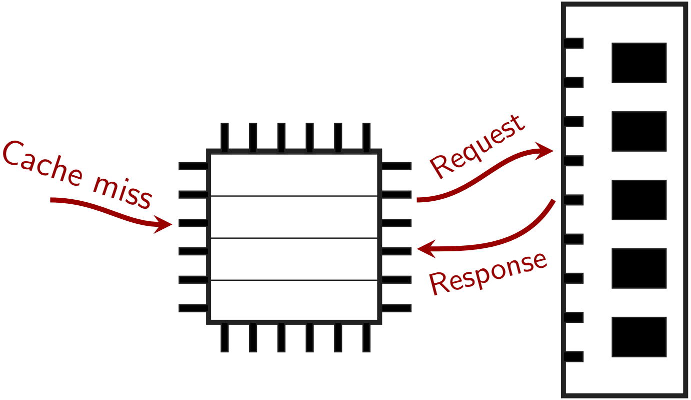
CPU Cache

```
maccess(i);

maccess(i);
```



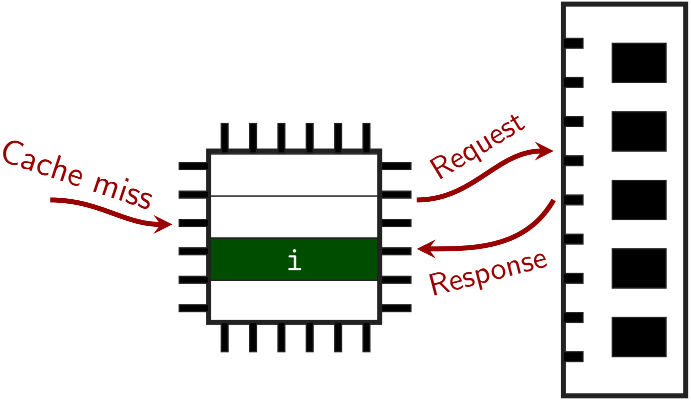
CPU Cache

```
maccess(i);

maccess(i);
```

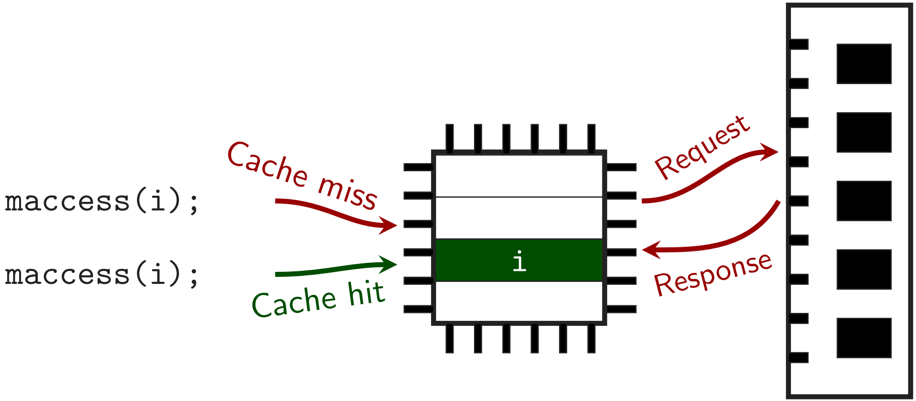


CPU Cache

```
maccess(i);

maccess(i);
```



CPU Cache

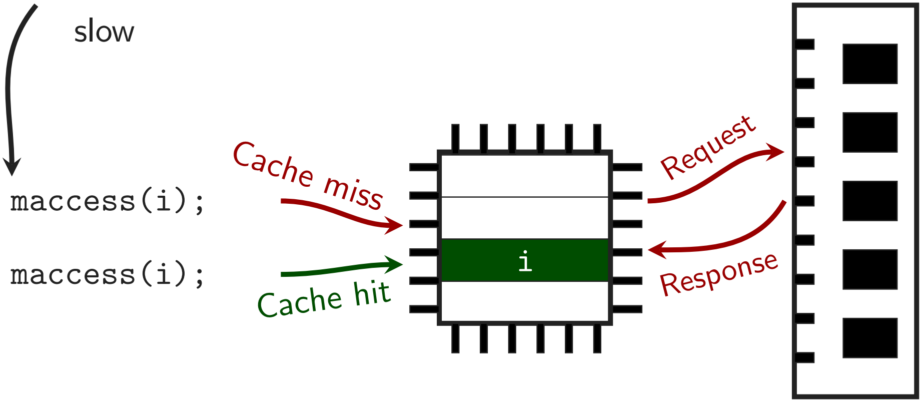


```
maccess(i);
maccess(i);
```

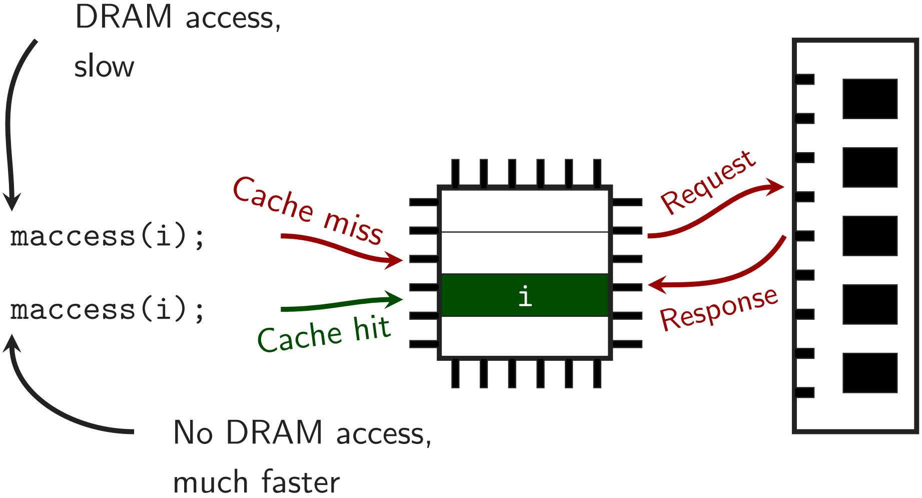
CPU Cache

DRAM access, slow

```
maccess(i);

maccess(i);
```



CPU Cache



DRAM access,
slow

```
maccess(i);
maccess(i);
```

Cache miss

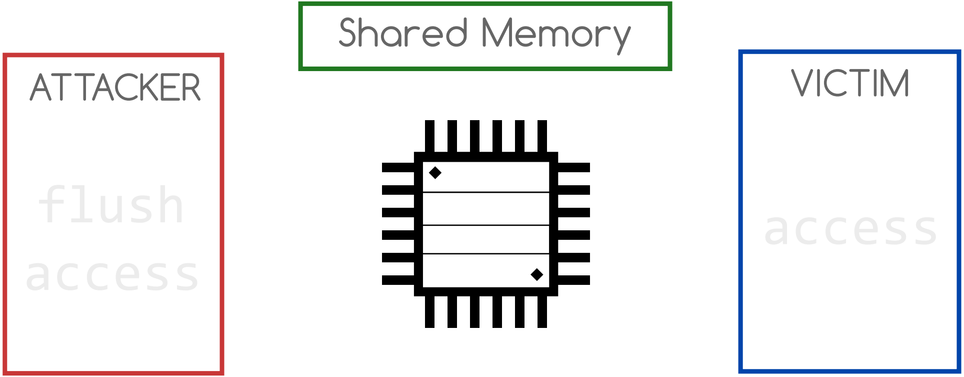
Cache hit

Request

Response

No DRAM access,
much faster

Flush+Reload



Shared Memory

ATTACKER

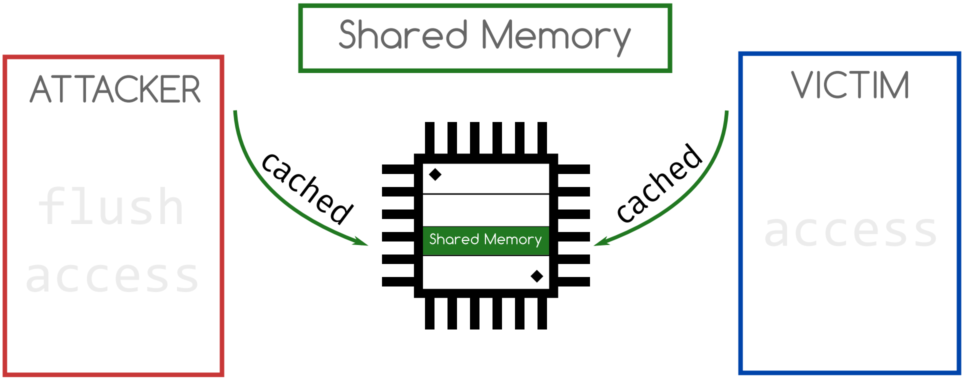
flush

access

VICTIM

access

Flush+Reload



Shared Memory

ATTACKER

flush

access

cached

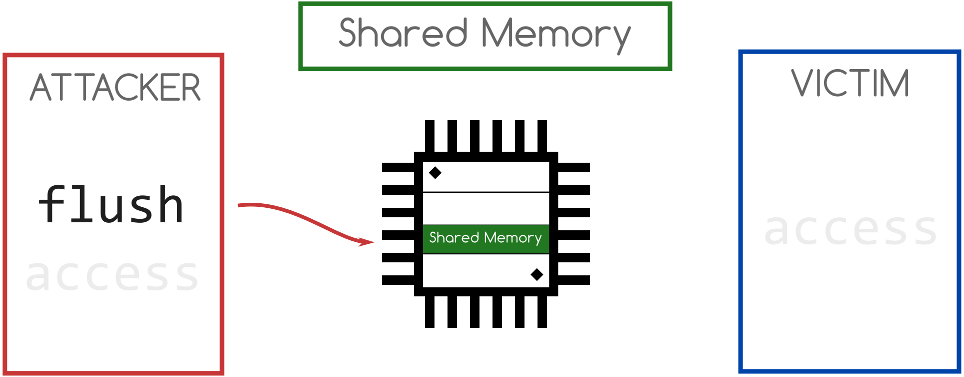
Shared Memory

cached

VICTIM

access

Flush+Reload



Shared Memory

ATTACKER

`flush`

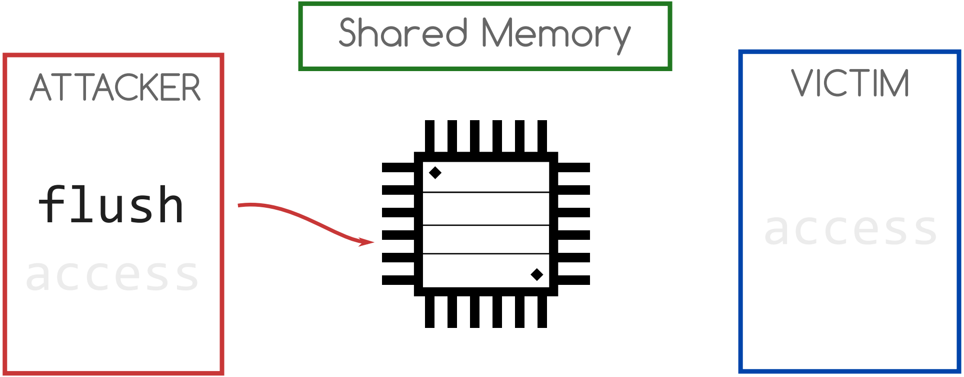
`access`

Shared Memory

VICTIM

`access`

Flush+Reload



Shared Memory

ATTACKER

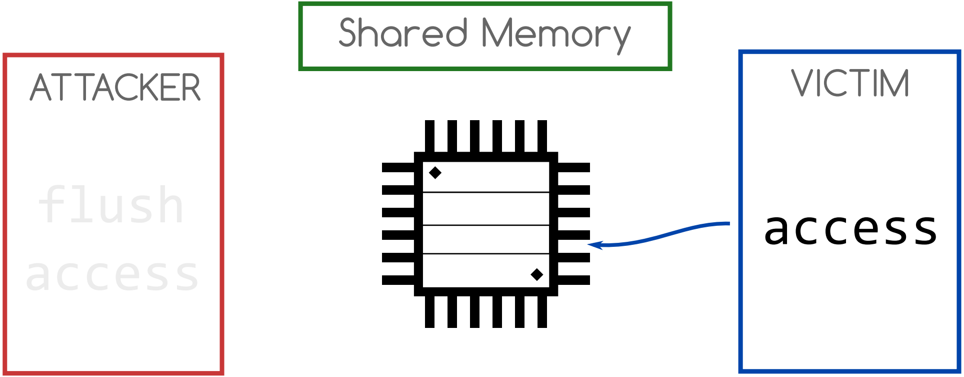
flush

access

VICTIM

access

Flush+Reload



Shared Memory

ATTACKER

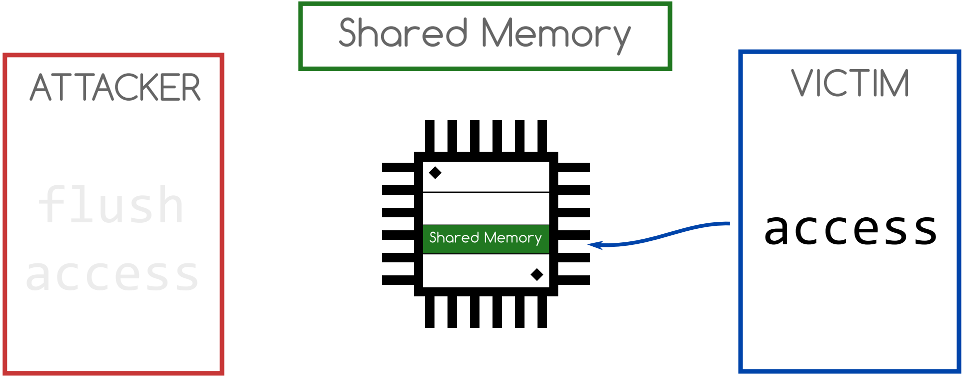
flush

access

VICTIM

access

Flush+Reload



Shared Memory

ATTACKER

flush

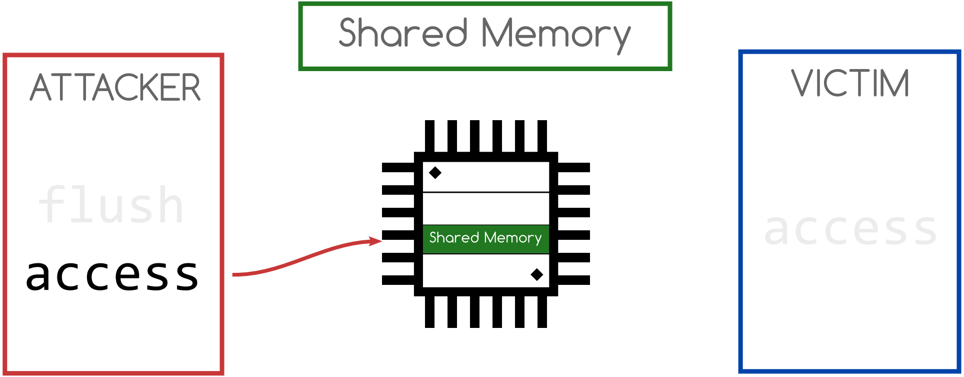
access

Shared Memory

VICTIM

access

Flush+Reload



Shared Memory

ATTACKER

flush

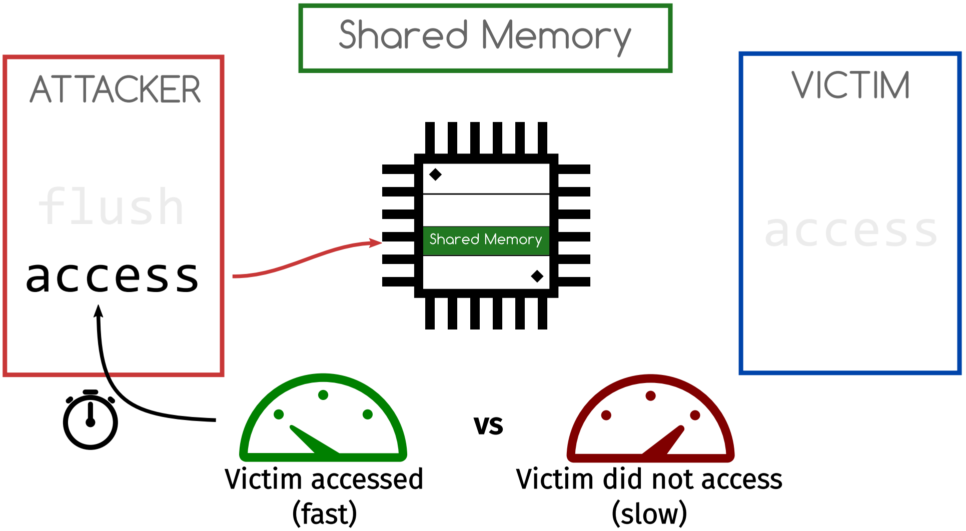
access

Shared Memory

VICTIM

access

Flush+Reload



Shared Memory

ATTACKER

flush

access

VICTIM

access

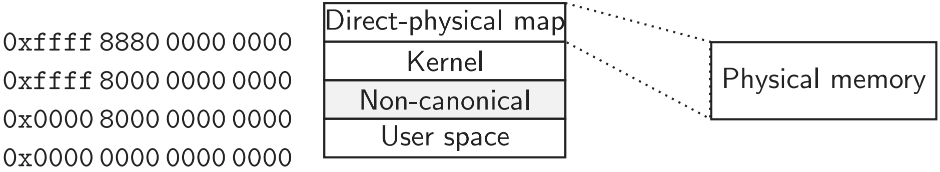
Shared Memory

vs

Victim accessed
(fast)

Victim did not access
(slow)

Virtual memory per process



0xffff 8880 0000 0000

0xffff 8000 0000 0000

0x0000 8000 0000 0000

0x0000 0000 0000 0000

Direct-physical map

Kernel

Non-canonical

User space

Physical memory

Address-Translation Attack

- Fetch kernel addresses into the cache

Address-Translation Attack

- Fetch kernel addresses into the cache
- Using this technique virtual addresses can be translated into physical addresses

Address-Translation Attack

- Fetch kernel addresses into the cache
- Using this technique virtual addresses can be translated into physical addresses
- The KAISER patch should mitigate the address-translation attack [Gru+17]

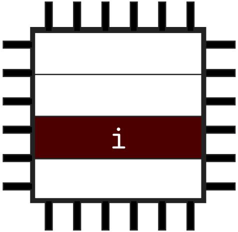
Address-Translation Attack

- Fetch kernel addresses into the cache
- Using this technique virtual addresses can be translated into physical addresses
- The KAISER patch should mitigate the address-translation attack [Gru+17]

Address-Translation Attack

i ≡ DPM-Address;

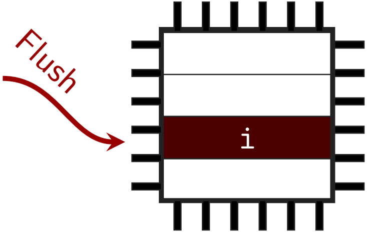
```
flush(i);

prefetch(DPM-Address);

sched_yield();

maccess(i);
```



Address-Translation Attack

i ≡ DPM-Address;

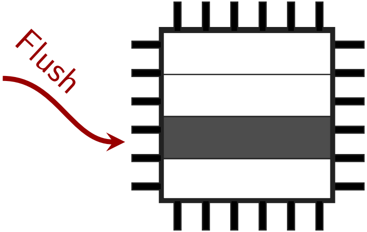
```
flush(i);

prefetch(DPM-Address);

sched_yield();

maccess(i);
```



Address-Translation Attack

i ≡ DPM-Address;

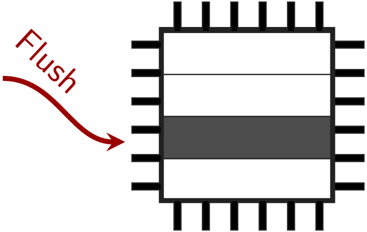
```
flush(i);

prefetch(DPM-Address);

sched_yield();

maccess(i);
```



Address-Translation Attack

i ≡ DPM-Address;

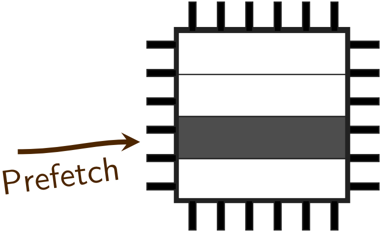
```
flush(i);

prefetch(DPM-Address);

sched_yield();

maccess(i);
```



Address-Translation Attack

i ≡ DPM-Address;

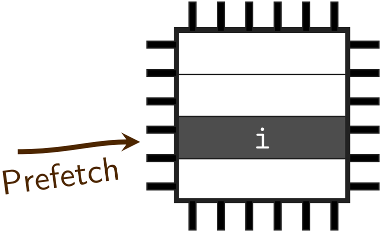
```
flush(i);

prefetch(DPM-Address);

sched_yield();

maccess(i);
```



Address-Translation Attack

i ≡ DPM-Address;

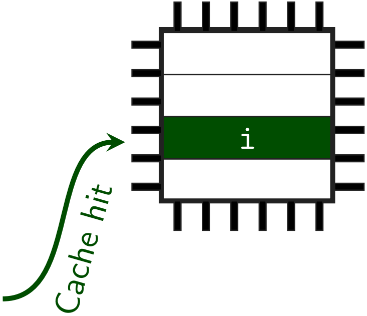
```
flush(i);

prefetch(DPM-Address);

sched_yield();

maccess(i);
```



Address-Translation Attack

i ≡ DPM-Address;

```
flush(i);

prefetch(DPM-Address);

sched_yield();

maccess(i);
```



Reproduction

- We successfully reproduced the address-translation attack

Reproduction

- We successfully reproduced the address-translation attack
- We only enable the page-table isolation and disabled all other mitigations

Reproduction

- We successfully reproduced the address-translation attack
- We only enable the page-table isolation and disabled all other mitigations
- Still cache fetches occured...

Reproduction

- We successfully reproduced the address-translation attack
- We only enable the page-table isolation and disabled all other mitigations
- Still cache fetches occured...

The attack still works even with active Meltdown mitigations?

Analysis I

- Root-cause was attributed to the `prefetch` instruction [Gru+16]

Analysis I

- Root-cause was attributed to the `prefetch` instruction [Gru+16]
- We disassembled the PoC and observed that the DPM-Address is located in a register ($r14$)

Analysis I

- Root-cause was attributed to the `prefetch` instruction [Gru+16]
- We disassembled the PoC and observed that the DPM-Address is located in a register ($r14$)
- In addition the `sched_yield` syscall is performed in the attack

Analysis I

- Root-cause was attributed to the `prefetch` instruction [Gru+16]
- We disassembled the PoC and observed that the DPM-Address is located in a register ($r14$)
- In addition the `sched_yield` syscall is performed in the attack
- By enabling all mitigations against microarchitectural the leakage disappears

Analysis I

- Root-cause was attributed to the `prefetch` instruction [Gru+16]
- We disassembled the PoC and observed that the DPM-Address is located in a register ($r14$)
- In addition the `sched_yield` syscall is performed in the attack
- By enabling all mitigations against microarchitectural the leakage disappears

Analysis II

- We `NOP`ed out the prefetch instructions

Analysis II

- We NOPed out the prefetch instructions
- Cache fetches still occured → value in register is used

Analysis II

- We `NOP`ed out the prefetch instructions
- Cache fetches still occured → value in register is used
- Up to 60 cache fetches per second

Analysis II

- We `NOPed` out the prefetch instructions
- Cache fetches still occured → value in register is used
- Up to 60 cache fetches per second
- If the `sched_yield` is removed, the leakage nearly disappears

Analysis II

- We NOPed out the prefetch instructions
- Cache fetches still occured → value in register is used
- Up to 60 cache fetches per second
- If the sched_yield is removed, the leakage nearly disappears
- If full Spectre-V2 mitigations are applied, the leakage is completely gone

Speculative Execution

- CPU tries to predict the outcome of branches

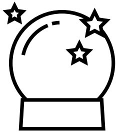
Speculative Execution

- CPU tries to predict the outcome of branches
- Predicted part gets executed speculatively

Speculative Execution

- CPU tries to predict the outcome of branches
- Predicted part gets executed speculatively
- If the prediction was correct, . . .

Speculative Execution



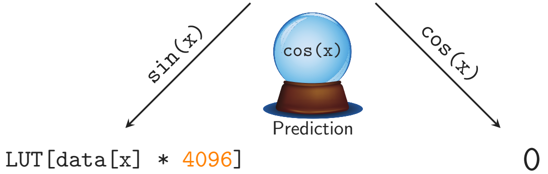
- CPU tries to predict the outcome of branches
- Predicted part gets executed speculatively
- If the prediction was correct, . . .
  - . . . very fast

Speculative Execution

- CPU tries to predict the outcome of branches
- Predicted part gets executed speculatively
- If the prediction was correct, . . .
    - . . . very fast
    - otherwise: Discard results

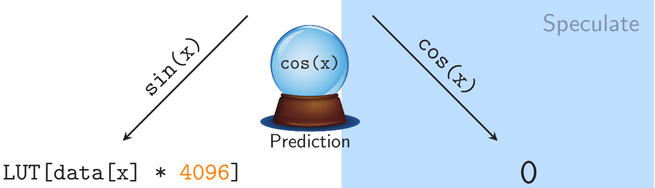
Spectre-BTB

```
float
(*math_functions[2])(float)
= {sin,cos};
fun_index = 0;

math_functions[fun_index]();
```



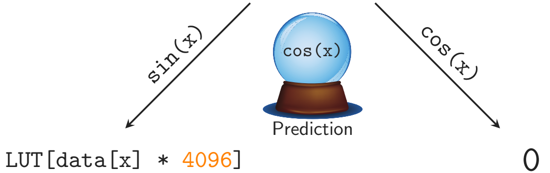
Spectre-BTB

```
float
(*math_functions[2])(float)
= {sin,cos};
fun_index = 0;

math_functions[fun_index]();
```



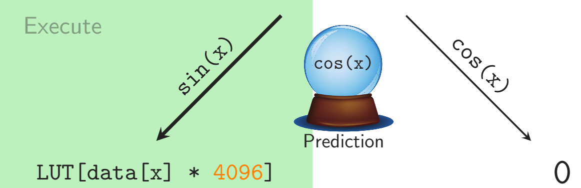
Spectre-BTB

```
float
(*math_functions[2])(float)
= {sin,cos};
fun_index = 0;

math_functions[fun_index]();
```



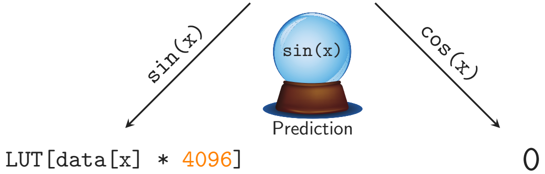
Spectre-BTB

```
float
(*math_functions[2])(float)
= {sin,cos};
fun_index = 0;

math_functions[fun_index]();
```



Spectre-BTB

```
float
(*math_functions[2])(float)
= {sin,cos};
fun_index = 0;

math_functions[fun_index]();
```



sin(x)

sin(x)

Prediction

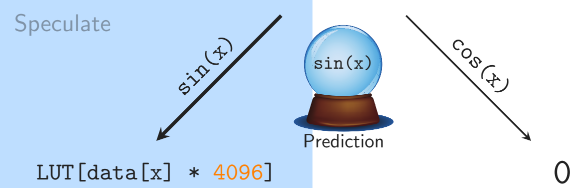
cos(x)

LUT[data[x] * 4096]

0

Spectre-BTB

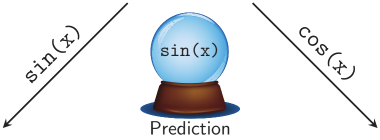
```
float
(*math_functions[2])(float)
= {sin,cos};
fun_index = 0;

math_functions[fun_index]();
```



```
LUT[data[x] * 4096]
```

Spectre-BTB

```
float
(*math_functions[2])(float)
= {sin,cos};
fun_index = 0;

math_functions[fun_index]();
```



sin(x)

sin(x)

Prediction

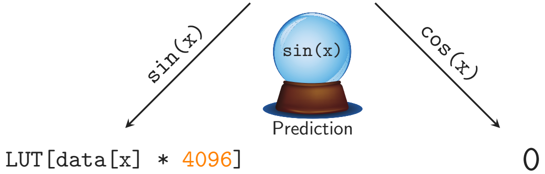
cos(x)

LUT[data[x] * 4096]

0

Spectre-BTB

```
float
(*math_functions[2])(float)
= {sin,cos};
fun_index = 1;

math_functions[fun_index]();
```

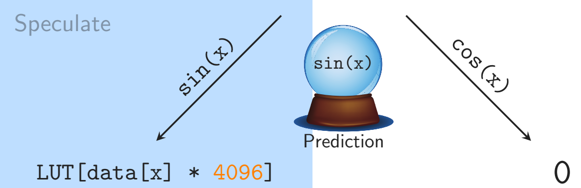


LUT[data[x] * 4096]

0

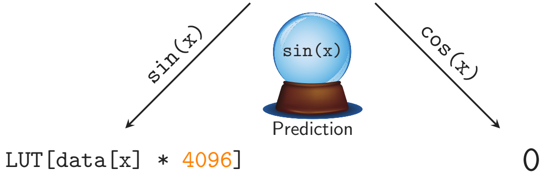
Spectre-BTB

```
float
(*math_functions[2])(float)
= {sin,cos};
fun_index = 1;

math_functions[fun_index]();
```



Spectre-BTB

```
float
(*math_functions[2])(float)
= {sin,cos};
fun_index = 1;

math_functions[fun_index]();
```



sin(x)

sin(x)

Prediction

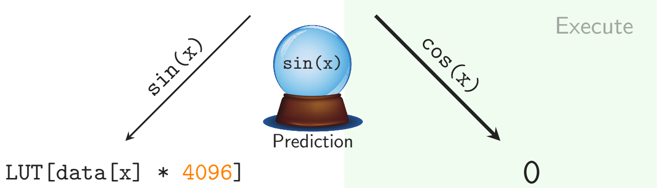
cos(x)

LUT[data[x] * 4096]

0

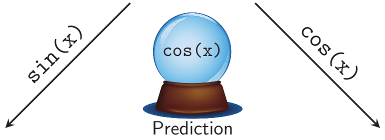
Spectre-BTB

```
float
(*math_functions[2])(float)
= {sin,cos};
fun_index = 1;

math_functions[fun_index]();
```



Spectre-BTB

```
float
(*math_functions[2])(float)
= {sin,cos};
fun_index = 1;

math_functions[fun_index]();
```



sin(x)

cos(x)

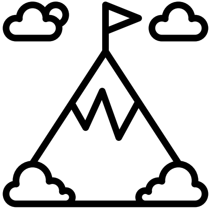
cos(x)

Prediction

```
LUT[data[x] * 4096]
```

0

Speculative execution in the kernel

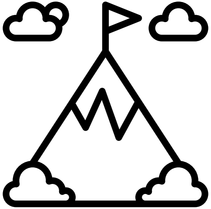


- We debugged the kernel and found a Spectre-BTB gadget in the kernel

Speculative execution in the kernel

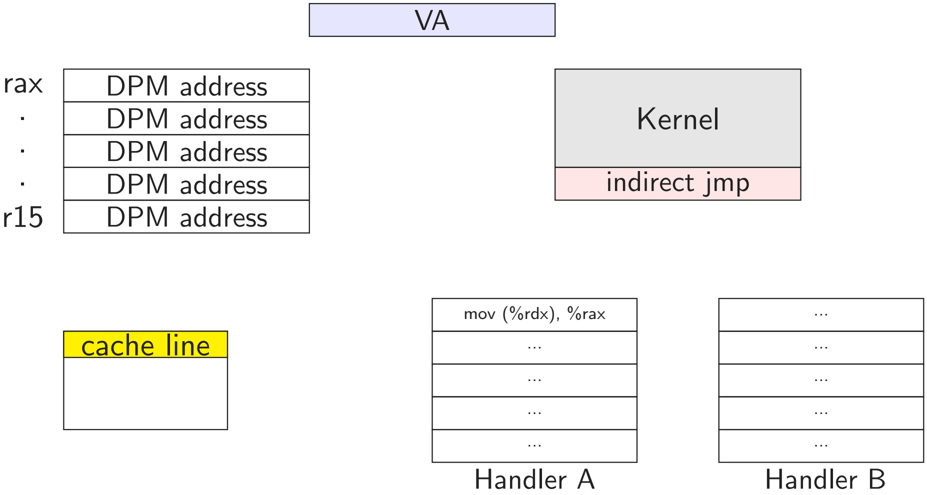
- We debugged the kernel and found a Spectre-BTB gadget in the kernel
- `put_prev_task_fair` dereferences a user-controlled register

Speculative execution in the kernel



- We debugged the kernel and found a Spectre-BTB gadget in the kernel
- `put_prev_task_fair` dereferences a user-controlled register
- There are multiple gadgets, for instance, also one triggered by NVMe interrupts

Speculative execution in the kernel



VA

rax DPM address
· DPM address
· DPM address
· DPM address
r15 DPM address

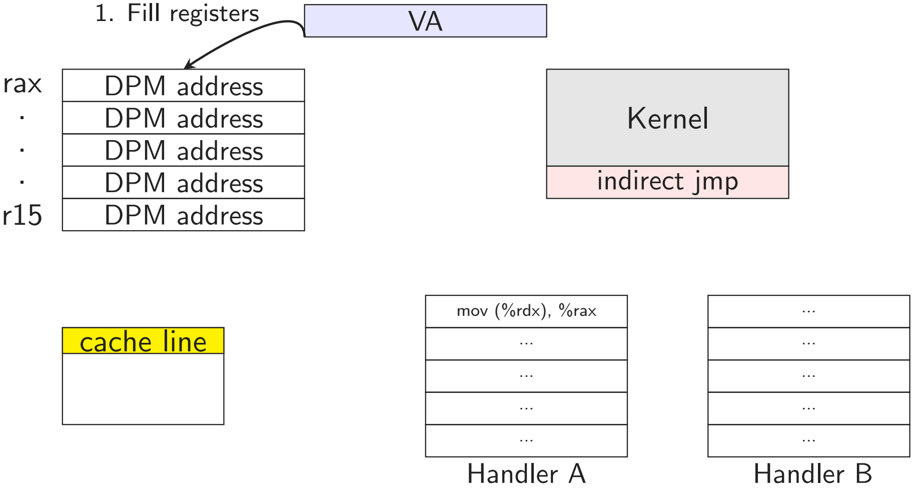
Kernel
indirect jmp

cache line

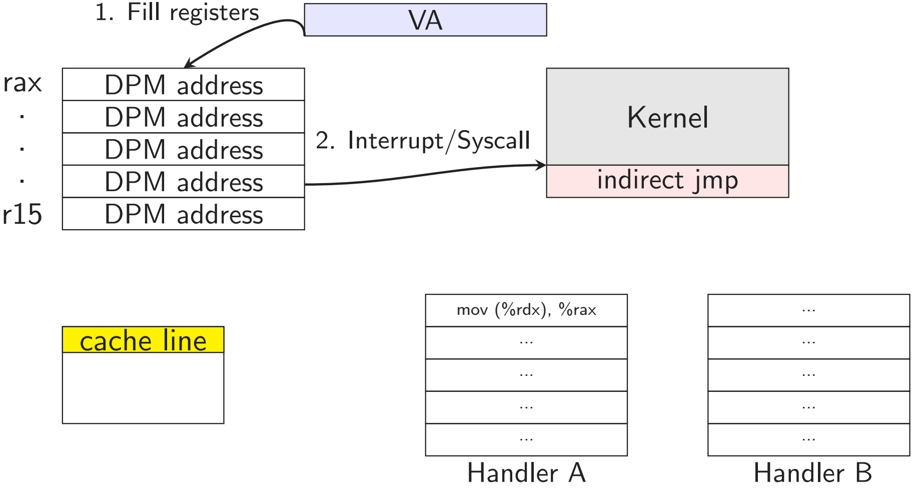
mov (%rdx), %rax
...
...
...
...
Handler A

...
...
...
...
...
Handler B

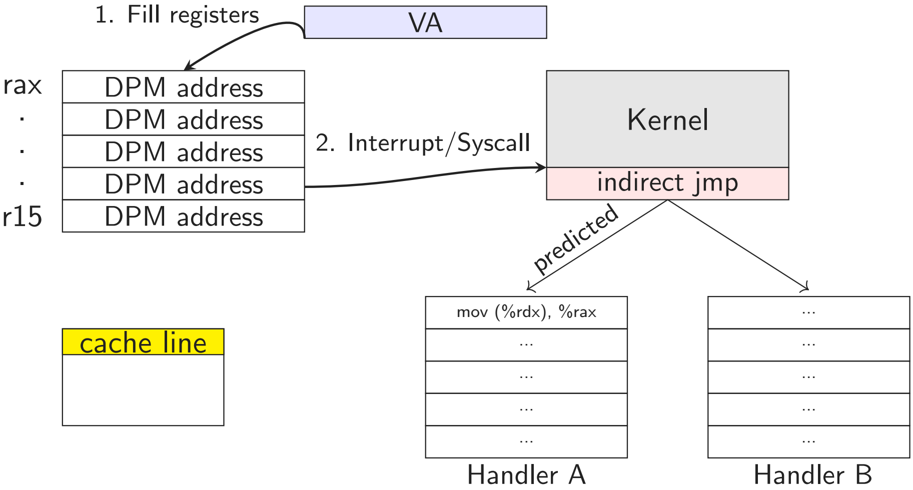
Speculative execution in the kernel



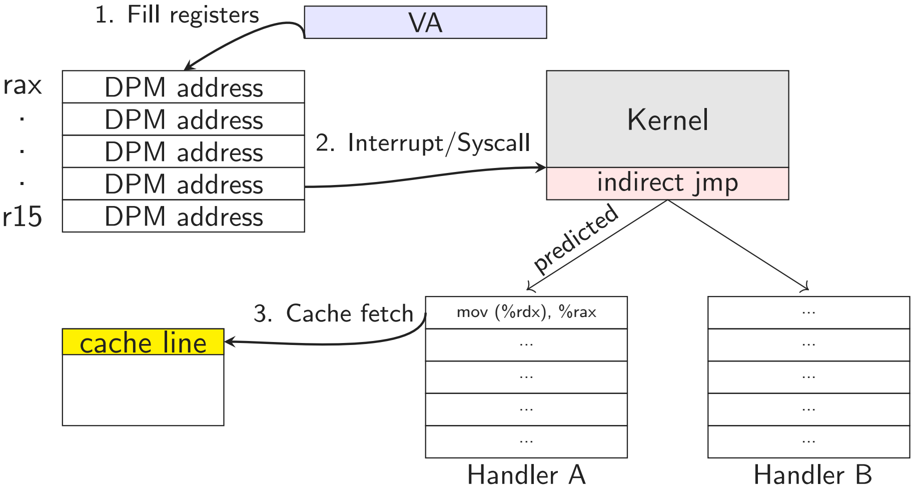
Speculative execution in the kernel



Speculative execution in the kernel



Speculative execution in the kernel



New attacks after understanding the correct root cause

Foreshadow

- Foreshadow or L1TF

Foreshadow

- Foreshadow or L1TF
- Leak data from L1 data cache

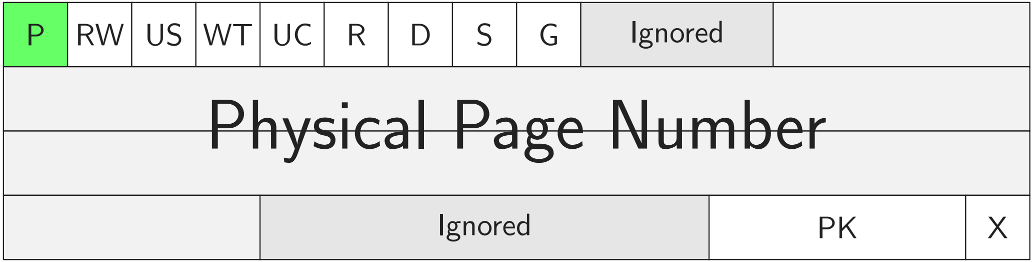
Foreshadow

- Foreshadow or L1TF
- Leak data from L1 data cache
- Affects virtual machines (VM), hypervisors (VMM), operating systems (OS) and system management mode (SMM)

Foreshadow

- Foreshadow or L1TF
- Leak data from L1 data cache
- Affects virtual machines (VM), hypervisors (VMM), operating systems (OS) and system management mode (SMM)
- Read SGX-protected memory and leak machine's private attestation key

Page Table Entry



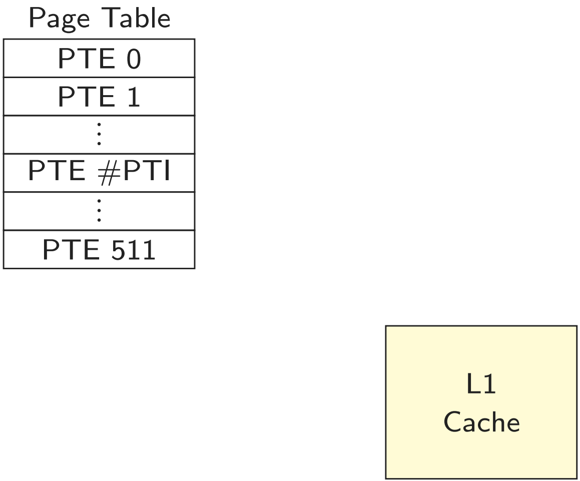
P | RW | US | WT | UC | R | D | S | G | Ignored

Physical Page Number

Ignored | PK | X

- Present bit defines whether a page is present in physical memory.

Present Bit



Page Table

PTE 0

PTE 1

⋮

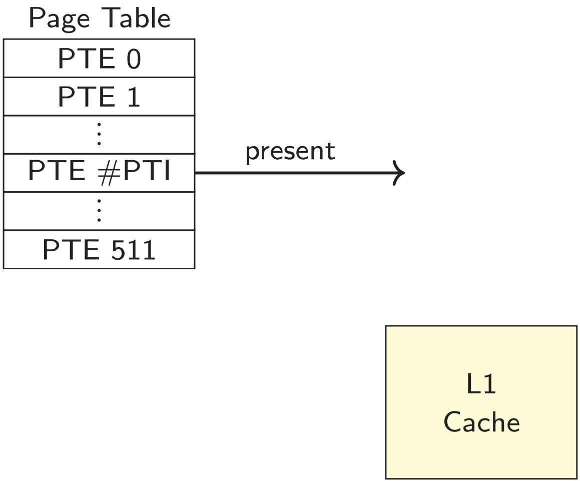
PTE #PTI

⋮

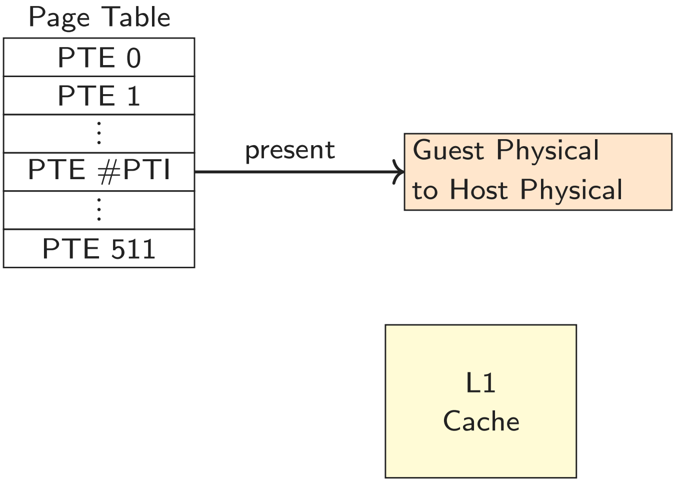
PTE 511

L1
Cache

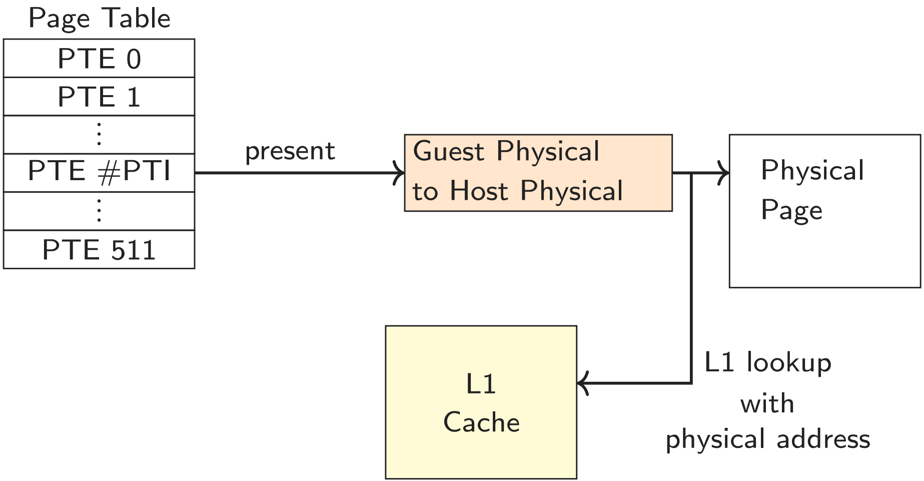
Present Bit



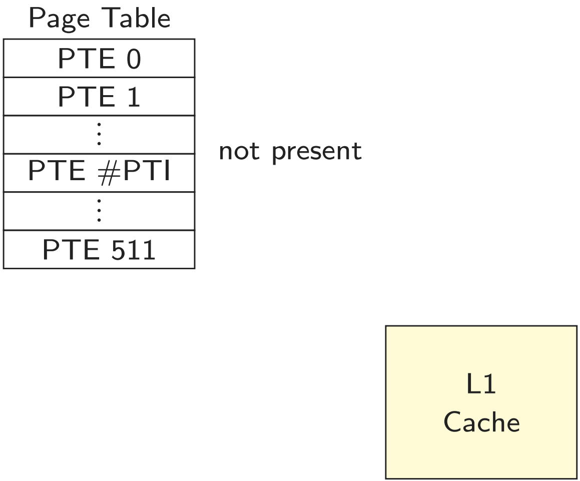
Present Bit


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

Present Bit


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

Present Bit



Page Table

PTE 0

PTE 1

⋮

PTE #PTI

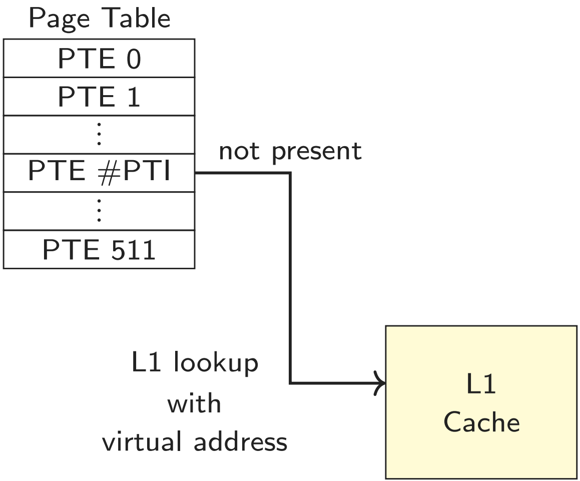
⋮

PTE 511

not present

L1 Cache

Present Bit


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

Foreshadow Revived

- Foreshadow:
  - By modifying the PTE to a host-physical virtual address

Foreshadow Revived

- Foreshadow:
  - By modifying the PTE to a host-physical virtual address
  - Suppressing the exception using TSX or exception handling

Foreshadow Revived

- Foreshadow:
  - By modifying the PTE to a host-physical virtual address
  - Suppressing the exception using TSX or exception handling
  - Leaking the content of the data via Flush+Reload

Foreshadow Revived

- Foreshadow:
  - By modifying the PTE to a host-physical virtual address
  - Suppressing the exception using TSX or exception handling
  - Leaking the content of the data via Flush+Reload
- Foreshadow is already mitigated by performing L1 flushing

Foreshadow Revived

- Foreshadow:
  - By modifying the PTE to a host-physical virtual address
  - Suppressing the exception using TSX or exception handling
  - Leaking the content of the data via Flush+Reload
- Foreshadow is already mitigated by performing L1 flushing
- Default setting for KVM is that L1 is conditionally flushed

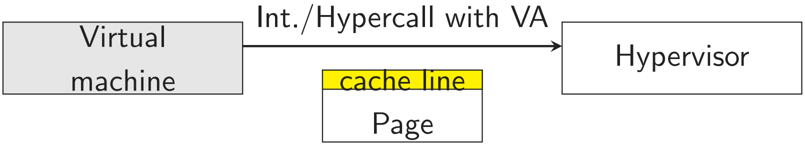
Foreshadow Revived

- Foreshadow:
  - By modifying the PTE to a host-physical virtual address
  - Suppressing the exception using TSX or exception handling
  - Leaking the content of the data via Flush+Reload
- Foreshadow is already mitigated by performing L1 flushing
- Default setting for KVM is that L1 is conditionally flushed
- Speculative Dereferencing allows it to fetch data from L3 into the cache (L1)

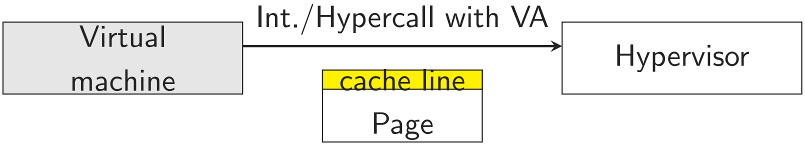
Foreshadow Revived

- Foreshadow:
  - By modifying the PTE to a host-physical virtual address
  - Suppressing the exception using TSX or exception handling
  - Leaking the content of the data via Flush+Reload
- Foreshadow is already mitigated by performing L1 flushing
- Default setting for KVM is that L1 is conditionally flushed
- Speculative Dereferencing allows it to fetch data from L3 into the cache (L1)
- Using the new insights Foreshadow is still possible on Linux KVM

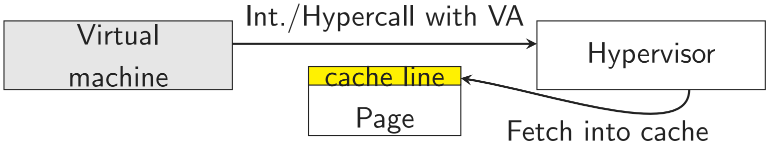
Foreshadow Revived



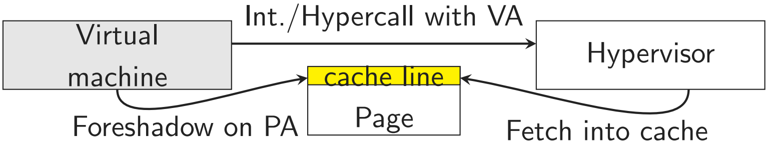
Foreshadow Revived

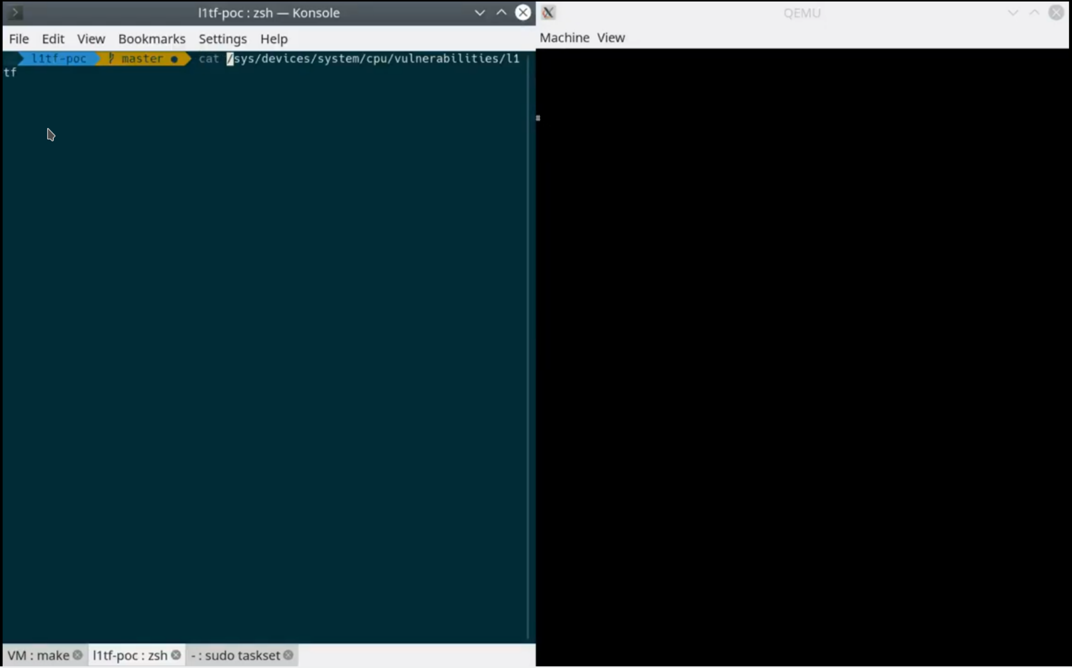


Foreshadow Revived



Foreshadow Revived





```
 l1tf-poc  master ●  cat /sys/devices/system/cpu/vulnerabilities/l1
tf
```

Dereference Trap

- With Dereference Trap we want to leak the content of registers from transient code paths

Dereference Trap

- With Dereference Trap we want to leak the content of registers from transient code paths
- We require a gadget which speculatively dereferences a register within an SGX enclave

Dereference Trap

- With Dereference Trap we want to leak the content of registers from transient code paths
- We require a gadget which speculatively dereferences a register within an SGX enclave
- The basic idea is to ensure that the entire virtual address space of the victim application is mapped

Dereference Trap

- With Dereference Trap we want to leak the content of registers from transient code paths
- We require a gadget which speculatively dereferences a register within an SGX enclave
- The basic idea is to ensure that the entire virtual address space of the victim application is mapped
- If a register containing a secret is speculatively dereferenced, the corresponding virtual address is cached

Dereference Trap

- With Dereference Trap we want to leak the content of registers from transient code paths
- We require a gadget which speculatively dereferences a register within an SGX enclave
- The basic idea is to ensure that the entire virtual address space of the victim application is mapped
- If a register containing a secret is speculatively dereferenced, the corresponding virtual address is cached
- The attacker detects whether a certain address was cached or not

Dereference Trap

- Leaking register values used in enclave(SGX)

Dereference Trap

- Leaking register values used in enclave(SGX)
- Speculative register dereferencing of memory values required (jump tables, function pointers)

Dereference Trap

- Leaking register values used in enclave(SGX)
- Speculative register dereferencing of memory values required (jump tables, function pointers)
- Due to address space limit we perform binary search by mapping the same 2 physical addresses to multiple locations

Dereference Trap

- Leaking register values used in enclave(SGX)
- Speculative register dereferencing of memory values required (jump tables, function pointers)
- Due to address space limit we perform binary search by mapping the same 2 physical addresses to multiple locations
- Split 32-bit value range into two equal sized mappings

Dereference Trap

- Leaking register values used in enclave(SGX)
- Speculative register dereferencing of memory values required (jump tables, function pointers)
- Due to address space limit we perform binary search by mapping the same 2 physical addresses to multiple locations
- Split 32-bit value range into two equal sized mappings
- One half maps physical page $p1$, the other page $p2$

Dereference Trap

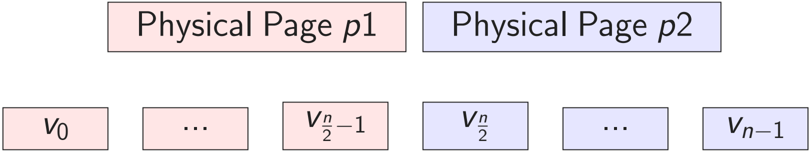
- Leaking register values used in enclave(SGX)
- Speculative register dereferencing of memory values required (jump tables, function pointers)
- Due to address space limit we perform binary search by mapping the same 2 physical addresses to multiple locations
- Split 32-bit value range into two equal sized mappings
- One half maps physical page $p1$, the other page $p2$
- Verify which physical page was cached using Flush+Reload

Dereference Trap

- Leaking register values used in enclave(SGX)
- Speculative register dereferencing of memory values required (jump tables, function pointers)
- Due to address space limit we perform binary search by mapping the same 2 physical addresses to multiple locations
- Split 32-bit value range into two equal sized mappings
- One half maps physical page $p1$, the other page $p2$
- Verify which physical page was cached using Flush+Reload
- Repeat

Dereference Trap

Flush+Reload



Physical Page $p1$

Physical Page $p2$

v_0

$\cdots$

$v_{\frac{n}{2}-1}$

$v_{\frac{n}{2}}$

$\cdots$

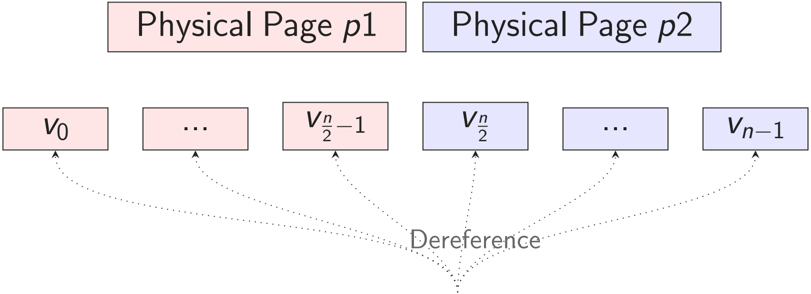
v_{n-1}

Dereference

Register Value (between v_0 and v_{n-1})

Dereference Trap

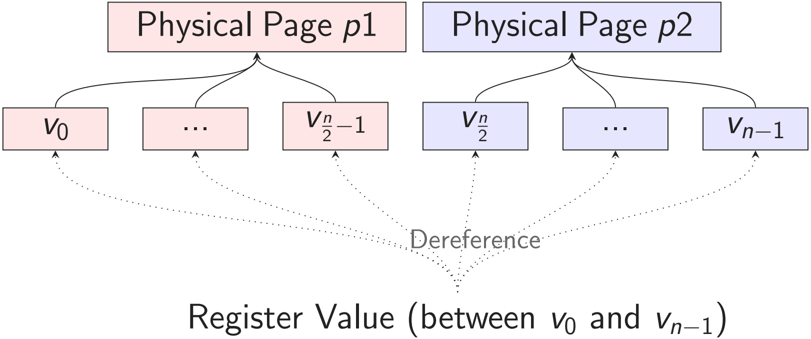
Flush+Reload



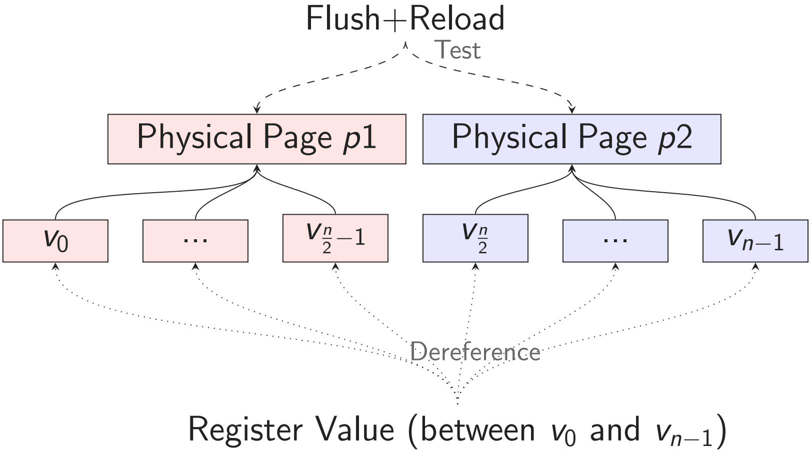
Register Value (between v_0 and v_{n-1})

Dereference Trap

Flush+Reload



Dereference Trap


Flush+Reload

Test

Physical Page $p1$

Physical Page $p2$

v_0

$\dots$

$v_{\frac{n}{2}-1}$

$v_{\frac{n}{2}}$

$\dots$

v_{n-1}

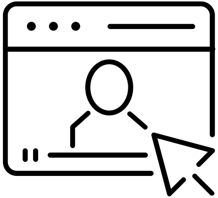
Dereference

Register Value (between v_0 and v_{n-1})

Register filling in JavaScript

- Can also be triggered in browsers

Register filling in JavaScript



- Can also be triggered in browsers
- Up to 20 cache fetches per second, if syscall would is directly triggered

Register filling in JavaScript

- Can also be triggered in browsers
- Up to 20 cache fetches per second, if syscall would is directly triggered
- On an unmodified browser 2 cache fetches per hour

Register filling in JavaScript

- Can also be triggered in browsers
- Up to 20 cache fetches per second, if syscall would is directly triggered
- On an unmodified browser 2 cache fetches per hour
- Using NVMe interrupts up to 1 cache fetch per minute

Mitigations

- KAISER [Gru+17] does not prevent the address-translation attack

Mitigations

- KAISER [Gru+17] does not prevent the address-translation attack
- EIBRS is also vulnerable (30 B/s on Ice Lake)

Mitigations

- KAISER [Gru+17] does not prevent the address-translation attack
- EIBRS is also vulnerable (30 B/s on Ice Lake)
- → Full Spectre-BTB mitigations required

Conclusion

- Root cause of `prefetch` effect was wrong

Conclusion

- Root cause of `prefetch` effect was wrong
- Real effect is speculative execution in the kernel

Conclusion

- Root cause of `prefetch` effect was wrong
- Real effect is speculative execution in the kernel
- Demonstrated that L1TF mitigations alone are not sufficient

Conclusion

- Root cause of `prefetch` effect was wrong
- Real effect is speculative execution in the kernel
- Demonstrated that L1TF mitigations alone are not sufficient
- Showed a technique to leak values from registers within SGX

Conclusion

- Root cause of `prefetch` effect was wrong
- Real effect is speculative execution in the kernel
- Demonstrated that L1TF mitigations alone are not sufficient
- Showed a technique to leak values from registers within SGX
- Demonstrated that prefetching can also be triggered in browsers

Speculative Dereferencing: Reviving Foreshadow

Martin Schwarzl (@marv0x90), Thomas Schuster, Michael Schwarz, Daniel Gruss

1st of March, 2021

Graz University of Technology

References

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

D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard. KASLR is Dead: Long Live KASLR. In: ESSoS. 2017.

Acknowledgments

We want to thank Moritz Lipp, Clementine Maurice, Anders Fogh, Xiao Yuan, Jo Van Bulck, and Frank Piessens of the original papers for reviewing and providing feedback to drafts of this work and for discussing the technical root cause with us. Furthermore, we want to thank Intel and ARM for valuable feedback on an early draft. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 681402). Additional funding was provided by generous gifts from Cloudflare, Intel and Red Hat. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.