KASLR: Break It, Fix It, Repeat

C. Canella (@cc0x1f), M. Schwarz, M. Haubenwallner, M. Schwarzl, D. Gruss

ACM AsiaCCS 2020

Graz University of Technology

- Meltdown requires hardware mitigations

Motivation

- Meltdown requires hardware mitigations
- How do these mitigations work?

C. Canella (@cc0x1f), M. Schwarz, M. Haubenwallner, M. Schwarzl, D. Gruss — Graz University of Technology

Motivation

- Meltdown requires hardware mitigations
- How do these mitigations work?
- Microarchitectural KASLR breaks are fast and reliable

Motivation

- Meltdown requires hardware mitigations
- How do these mitigations work?
- Microarchitectural KASLR breaks are fast and reliable → how do we prevent them?

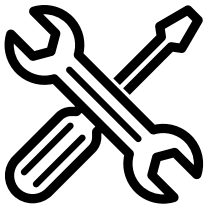
- CPU uses data in out-of-order execution before permission check

- CPU uses data in out-of-order execution before permission check
- Meltdown can read any kernel address

- CPU uses data in out-of-order execution before permission check
- Meltdown can read any kernel address
- Physical memory is usually mapped in kernel

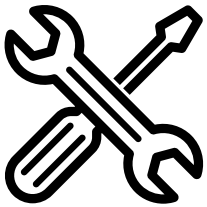
- CPU uses data in out-of-order execution before permission check
- Meltdown can read any kernel address
- Physical memory is usually mapped in kernel

→ Read arbitrary memory



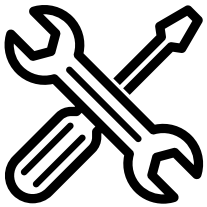
Assumptions

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

Verifying the Hypothesis

Two tests:

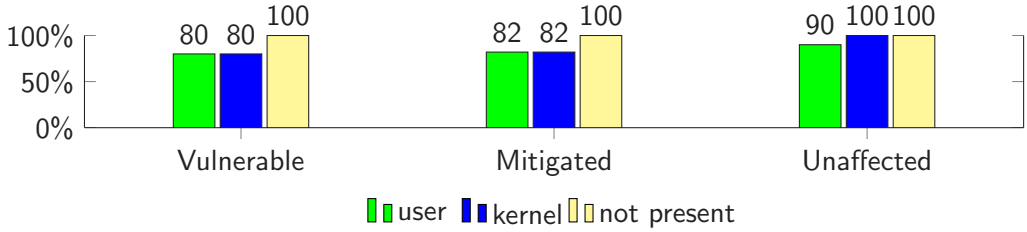
1. Perform Meltdown attack

Verifying the Hypothesis

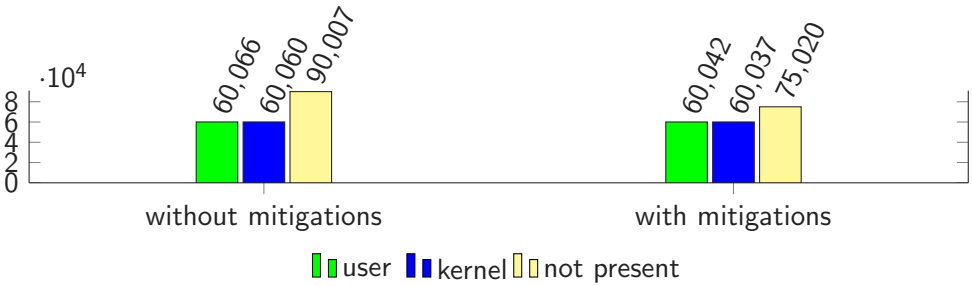
Two tests:

1. Perform Meltdown attack
2. Use Performance Counters

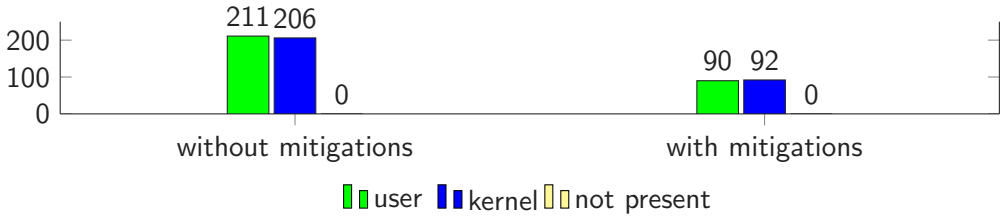
- Intel: CYCLE_ACTIVITY.STALLS_MEM_ANY
- AMD: Dispatch Stalls

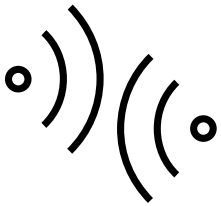


- Track number of issued μOPs on the load ports
- UOPS_DISPATCHED_PORT.PORT_2, UOPS_DISPATCHED_PORT.PORT_3

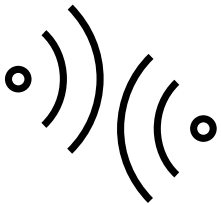


- L1D_PEND_MISS.PENDING_CYCLES

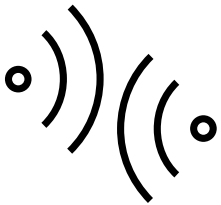




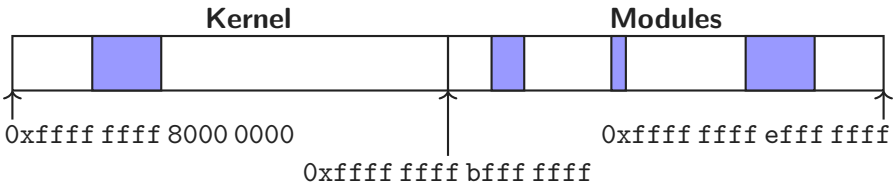
- Can we use the hardware-based mitigations for an attack?

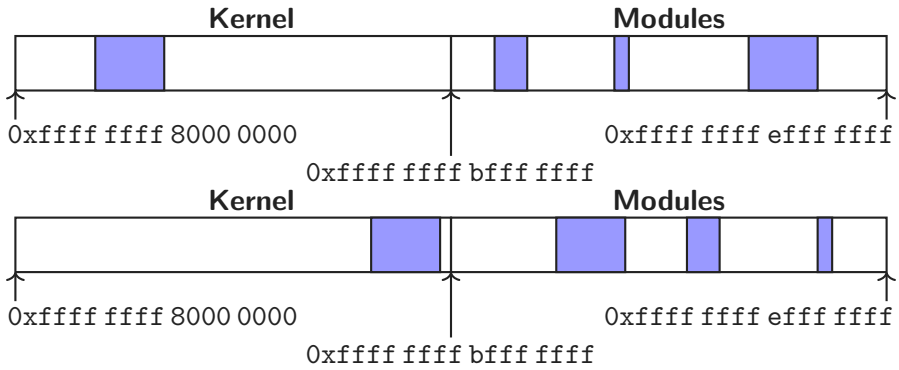


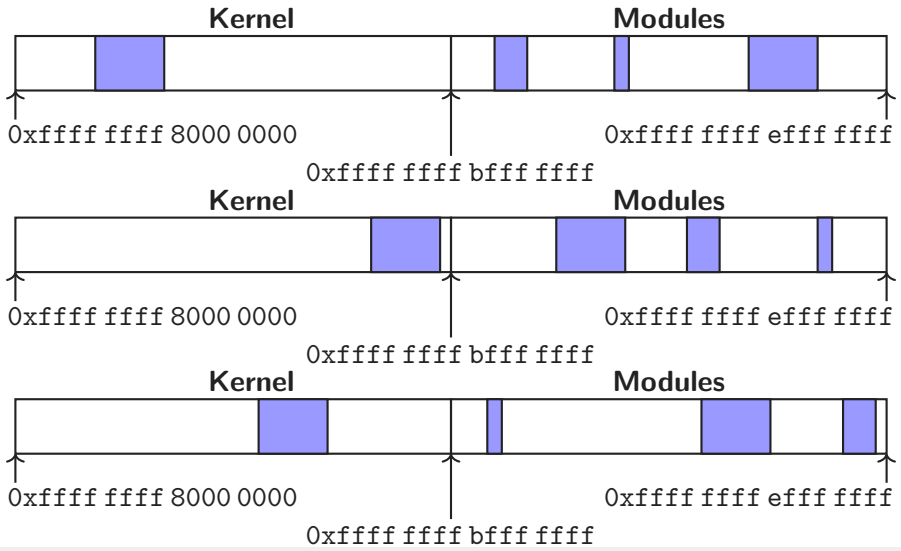
- Can we use the hardware-based mitigations for an attack?
- EchoLoad: fast and reliable KASLR break



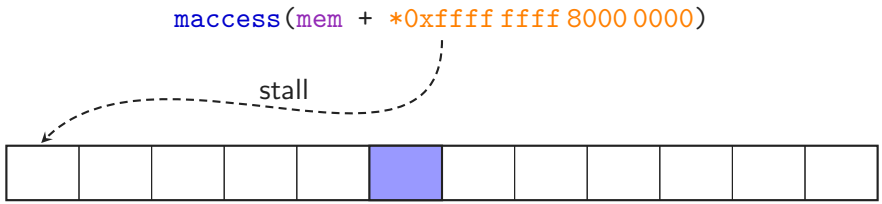
- Can we use the hardware-based mitigations for an attack?
- EchoLoad: fast and reliable KASLR break
- Encodes the returned value in the cache

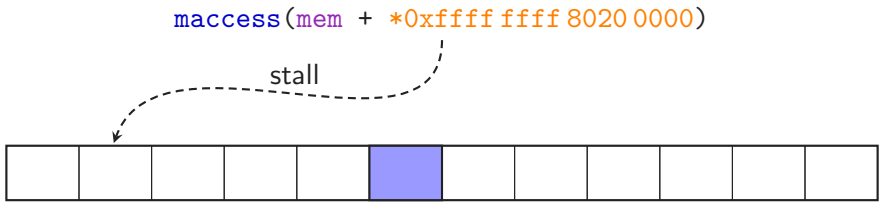

Kernel
Modules
0xffff ffff 8000 0000
0xffff ffff bfff ffff
0xffff ffff efff ffff

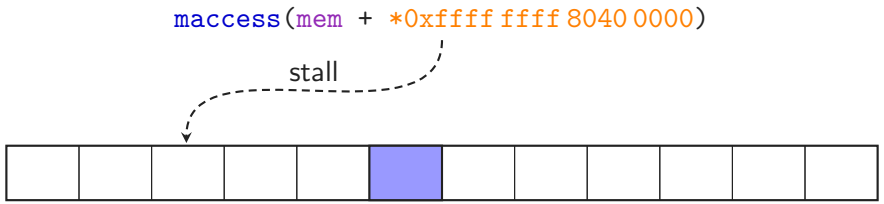

Kernel
Modules
0xffff ffff 8000 0000
0xffff ffff efff ffff
0xffff ffff bfff ffff
Kernel
Modules
0xffff ffff 8000 0000
0xffff ffff efff ffff
0xffff ffff bfff ffff

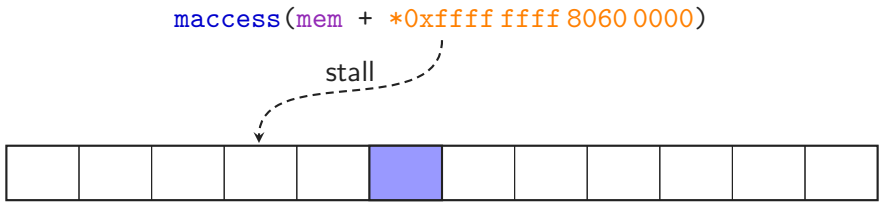

Kernel
Modules
0xffff ffff 8000 0000
0xffff ffff efff ffff
0xffff ffff bfff ffff
Kernel
Modules
0xffff ffff 8000 0000
0xffff ffff efff ffff
0xffff ffff bfff ffff
Kernel
Modules
0xffff ffff 8000 0000
0xffff ffff efff ffff
0xffff ffff bfff ffff

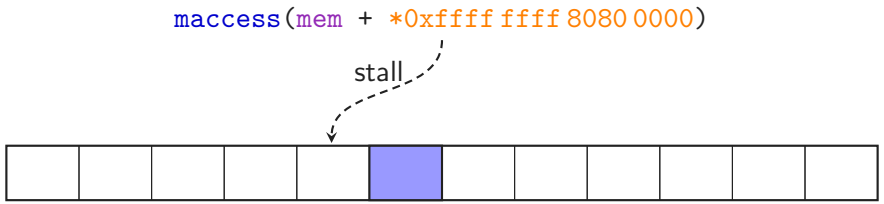
```
maccess(mem + *0xffff ffff 8000 0000)
```

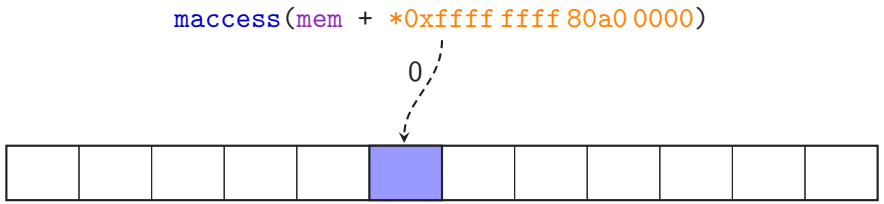

maccess(mem + *0xffff ffff 8000 0000)
stall


maccess(mem + *0xffff ffff 8020 0000)
stall


maccess(mem + *0xffff ffff 8040 0000)
stall


maccess(mem + *0xffff ffff 8060 0000)
stall


maccess(mem + *0xffff ffff 8080 0000)
stall


maccess(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)

Performance of EchoLoad

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)

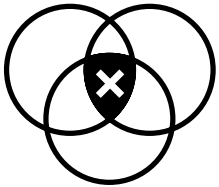
- Works in SGX and JavaScript

Performance of EchoLoad

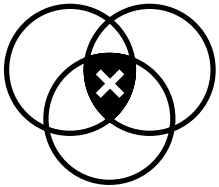
CPU		Speculation		TSX		Segfault	
i7-6700K	Time (F-Score)	63 µs	(0.999)	48 µs	(1.000)	133 µs	(1.000)
i9-9900K	Time (F-Score)	33 µs	(1.000)	29 µs	(1.000)	86 µs	(1.000)
Xeon Silver 4208	Time (F-Score)	51 µs	(0.994)	40 µs	(1.000)	127 µs	(1.000)

- Works in SGX and JavaScript
- Enables Meltdown from JavaScript on unpatched x86

C. Canella (@cc0x1f), M. Schwarz, M. Haubenwallner, M. Schwarzl, D. Gruss — Graz University of Technology

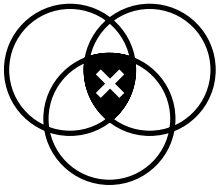


Timing difference



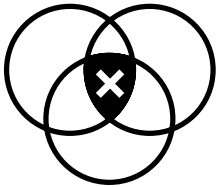
Timing difference

- between mapped and unmapped pages



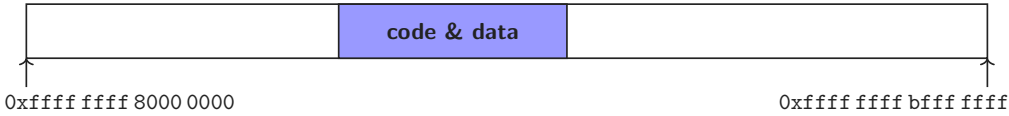
Timing difference

- between mapped and unmapped pages
- for different page sizes

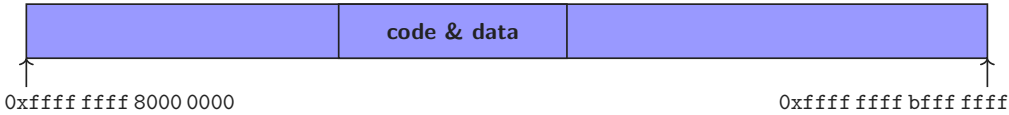


Timing difference

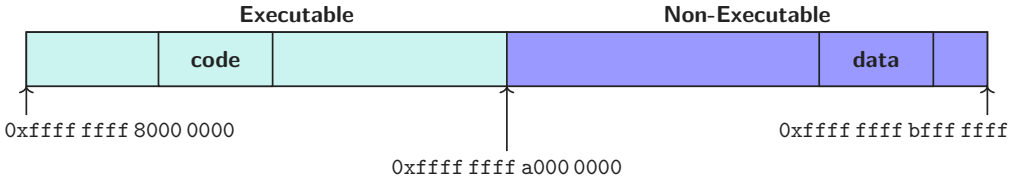
- between mapped and unmapped pages
- for different page sizes
- between executable and non-executable pages



Current Linux design



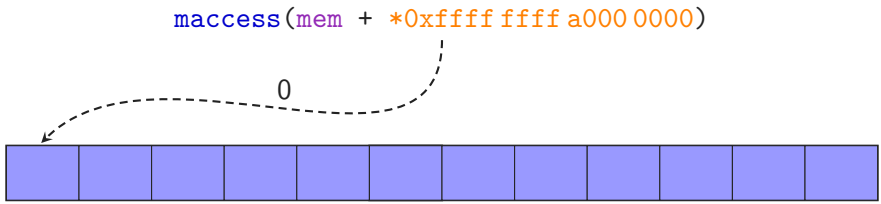
Step 1: Mitigating difference between mapped and unmapped pages

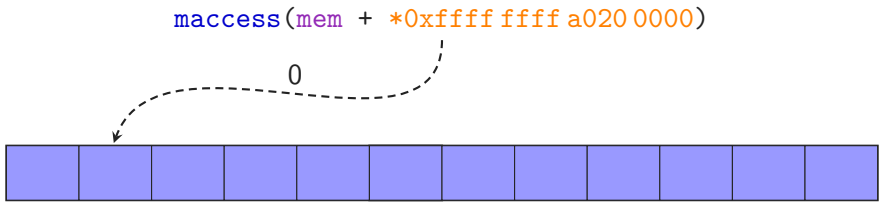


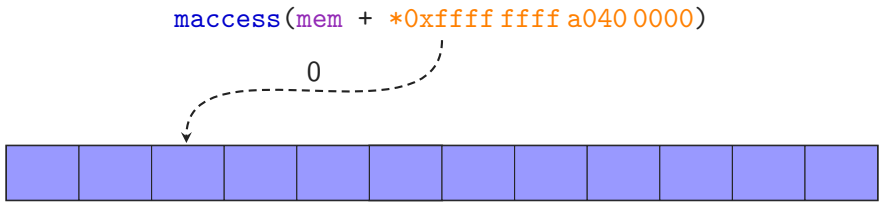
Step 2: Mitigating difference between executable and non-executable pages

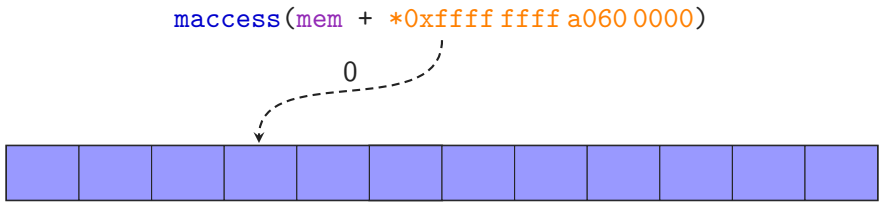
```
maccess(mem + *0xffff ffff a000 0000)
```

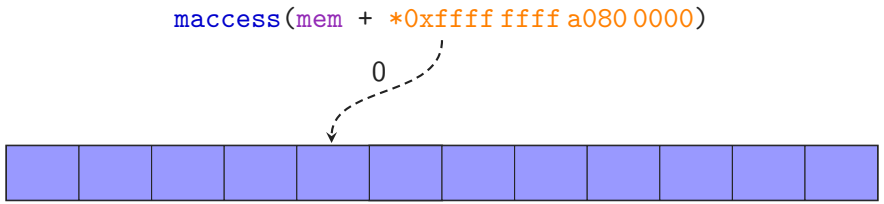
```
maccess(mem + *0xffff ffff a000 0000)
```

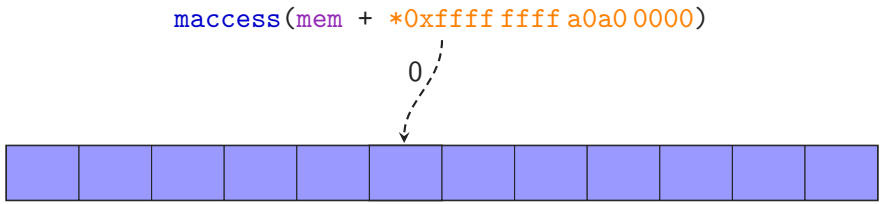

maccess(mem + *0xffff ffff a000 0000)
0

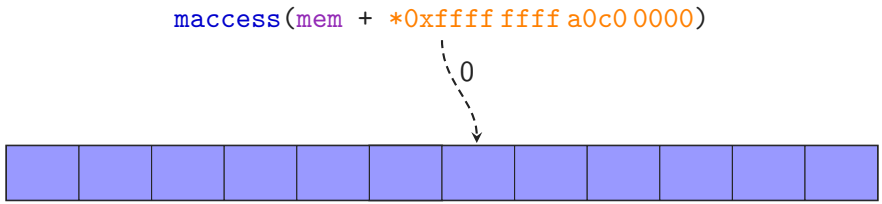

maccess(mem + *0xffff ffff a020 0000)
0

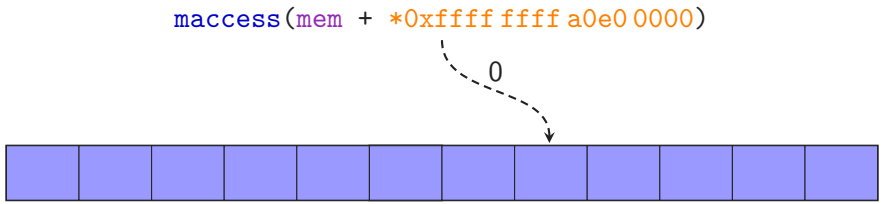

maccess(mem + *0xffff ffff a040 0000)
0

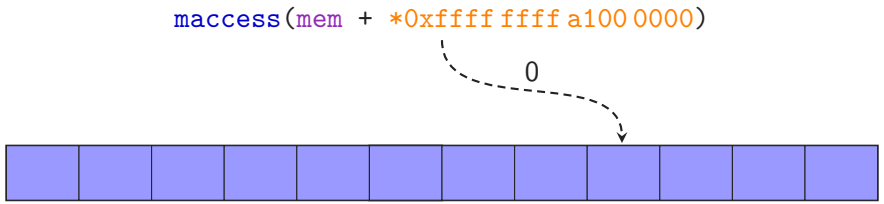

maccess(mem + *0xffff ffff a060 0000)
0

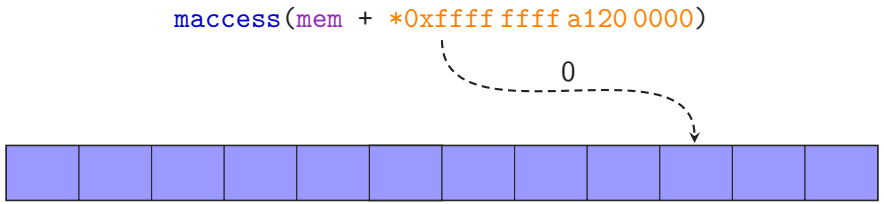

maccess(mem + *0xffff ffff a080 0000)
0

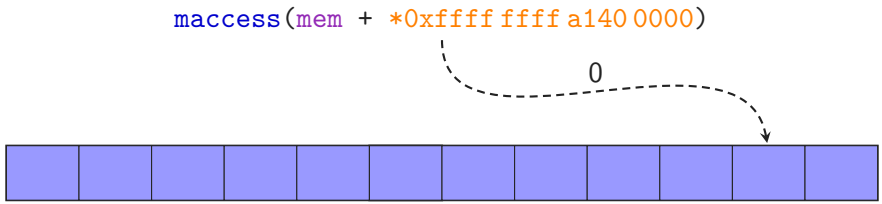

maccess(mem + *0xffff ffff a0a0 0000)
0

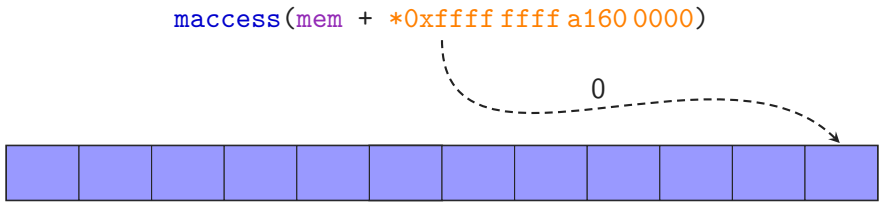

maccess(mem + *0xffff ffff a0c0 0000)
0

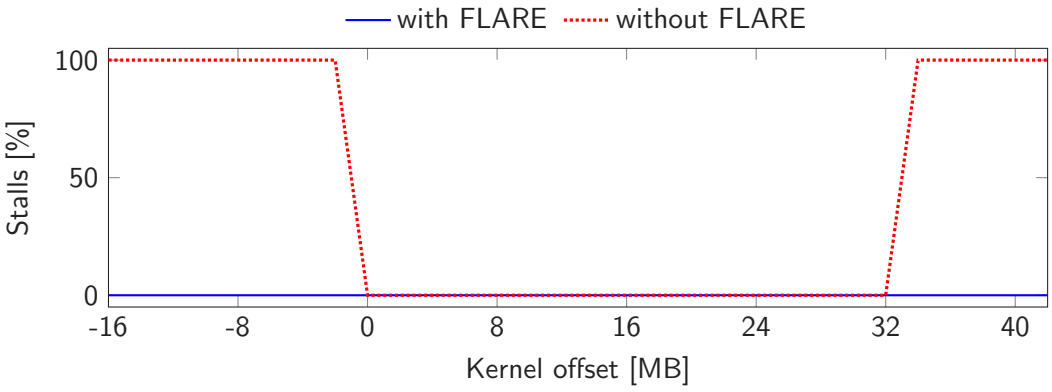

maccess(mem + *0xffff ffff a0e0 0000)
0


maccess(mem + *0xffff ffff a100 0000)
0

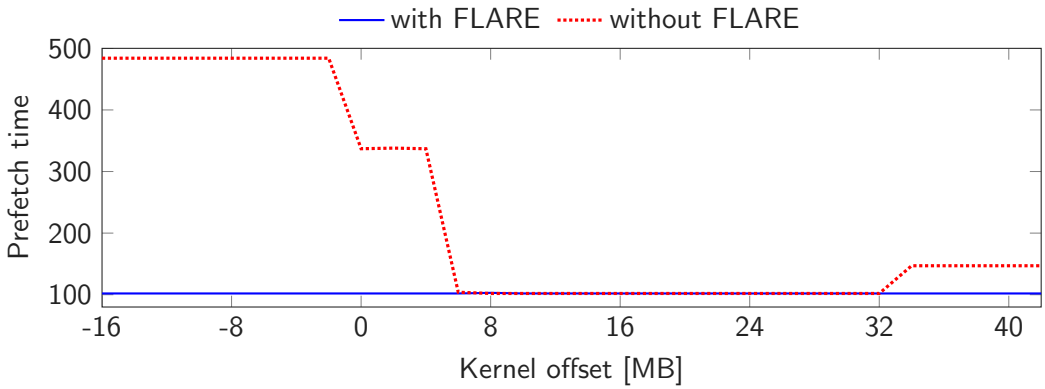

maccess(mem + *0xffff ffff a120 0000)
0


maccess(mem + *0xffff ffff a140 0000)
0

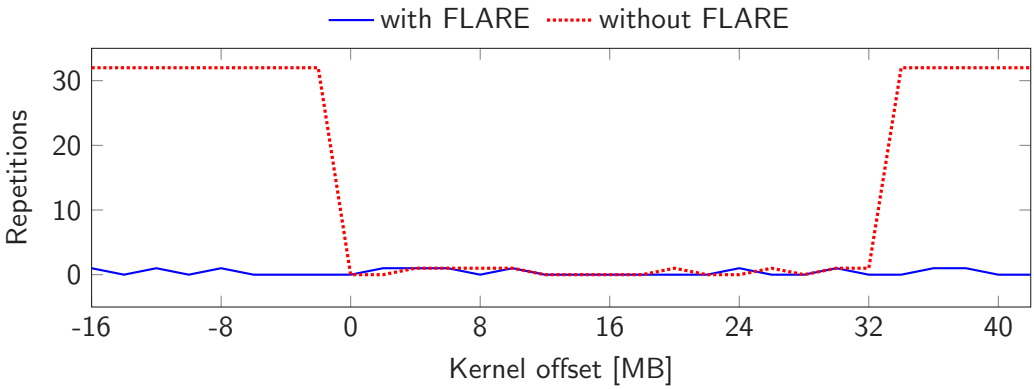

maccess(mem + *0xffff ffff a160 0000)
0


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

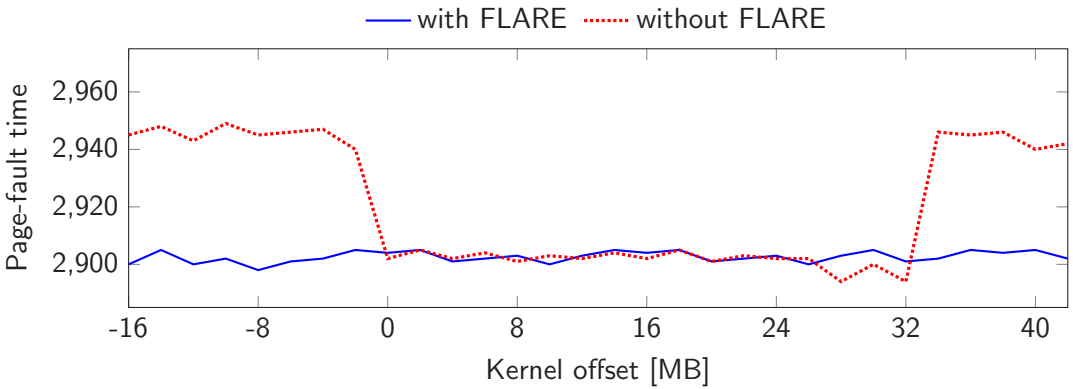
EchoLoad



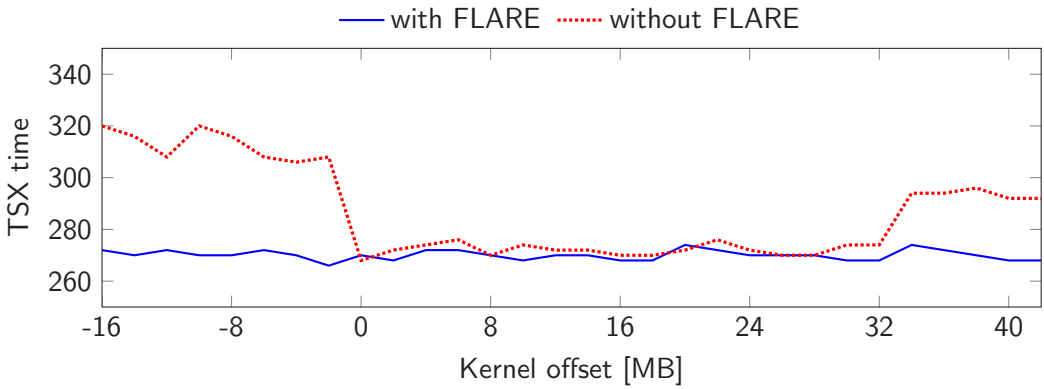
Prefetch [Gru+16]



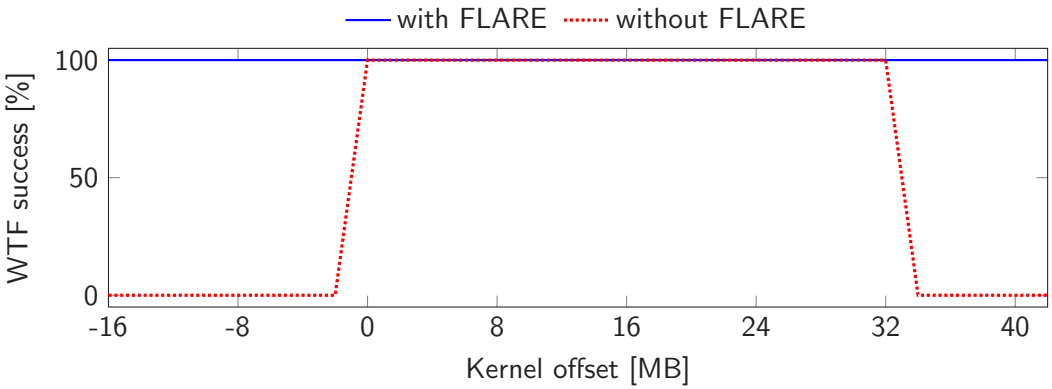
Data Bounce [Sch+19]



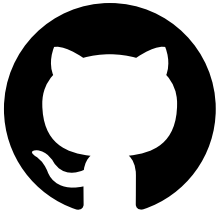
Double page fault [HWH13]


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`

More Details

More details in the paper [Can+20]

- Attacks from and on SGX
- Meltdown in JavaScript
- Kernel modules, vmalloc, ...
- ...

AsiaCCS'20

Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, Daniel Gruss.
KASLR: Break It, Fix It, Repeat.

- Reverse-engineered hardware mitigations in recent Intel CPUs

- Reverse-engineered hardware mitigations in recent Intel CPUs
- Presented a new attack based on these mitigations

- Reverse-engineered hardware mitigations in recent Intel CPUs
- Presented a new attack based on these mitigations
- Proposed a mitigation for recent microarchitectural KASLR breaks

- Reverse-engineered hardware mitigations in recent Intel CPUs
- Presented a new attack based on these mitigations
- Proposed a mitigation for recent microarchitectural KASLR breaks

→ Need to consider impact mitigations have on security

KASLR: Break It, Fix It, Repeat

Claudio Canella (@cc0x1f), Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, Daniel Gruss

ACM AsiaCCS 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.

C. Canella, M. Schwarz, M. Haubenwallner, M. Schwarzl, and D. Gruss. KASLR: Break It, Fix It, Repeat. In: AsiaCCS. 2020.

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).

Acknowledgments

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). This work has been supported by the Austrian Research Promotion Agency (FFG) via the project ESPRESSO,which is funded by the province of Styria and the Business Promotion Agencies of Styria and Carinthia. Additional funding was provided by generous gifts from Intel, ARM, and Cloudflare. 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