Meltdown

Reading Kernel Memory from User Space

Moritz Lipp¹, Michael Schwarz¹, Daniel Gruss¹, Thomas Prescher², Werner Haas², Anders Fogh³, Jann Horn⁴, Stefan Mangard¹, Paul Kocher⁵, Daniel Genkin⁶, Yuval Yarom⁷, Mike Hamburg⁸

27th USENIX Security Symposium - August 16, 2018

¹Graz University of Technology, ²Cyberus Technology GmbH, ³G-Data Advanced Analytics, ⁴Google Project Zero, ⁵Independent (www.paulkocher.com), ⁶University of Michigan, ⁷University of Adelaide & Data61, ⁸Rambus, Cryptography Research Division
Virtual Memory

Virtual Address Space

- Kernel Addresses
- Non-canonical Addresses
- User Addresses
Building the Code

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

```bash
# sudo grep linux_banner /proc/kallsyms
fffffffff81a000e0 R linux_banner
```
char data = *(char*) 0xffffffff81a000e0;
printf("%c\n", data);
Building the Code

char data = *(char*) 0xfffffffff81a000e0;
printf("%c\n", data);

• Any invalid access throws an exception → segmentation fault
• Kernel is isolated from user space
Memory Isolation

• Kernel is isolated from user space
• This isolation is a combination of hardware and software
Memory Isolation

- Kernel is isolated from user space
- This isolation is a combination of hardware and software
- User applications cannot access anything from the kernel
• CPU support **virtual address spaces** to isolate processes
• CPU support virtual address spaces to isolate processes
• Physical memory is organized in page frames
• CPU support virtual address spaces to isolate processes
• Physical memory is organized in page frames
• Virtual memory pages are mapped to page frames using page tables
Address Translation on x86-64

48-bit virtual address

CR3

PML4
- PML4E 0
- PML4E 1
- ...
- #PML4I
- PML4E 511

PDPT
- PDPTE 0
- PDPTE 1
- ...
- #PDPTI
- PDPTE 511

Page Directory
- PDE 0
- PDE 1
- ...
- PDE #PDI
- PDE 511

Page Table
- PTE 0
- PTE 1
- ...
- PTE #PTI
- PTE 511

4 KiB Page
- Byte 0
- Byte 1
- ...
- Offset
- Byte 4095

Offset (12 b)

PML4I (9 b)

PDI (9 b)

PDPTI (9 b)
### Page Table Entry

<table>
<thead>
<tr>
<th>P</th>
<th>RW</th>
<th>US</th>
<th>WT</th>
<th>UC</th>
<th>R</th>
<th>D</th>
<th>S</th>
<th>G</th>
<th>Ignored</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Physical Page Number**

<table>
<thead>
<tr>
<th>Ignored</th>
<th>X</th>
</tr>
</thead>
</table>

- User/Supervisor bit defines in which **privilege level** the page can be accessed.
• Kernel is typically mapped into every address space
• Kernel is typically mapped into every address space
• Entire physical memory is mapped in the kernel
Loading an address

```c
char data = *(char*) 0xfffffffff81a000e0;
printf("%c\n", data);
```

- We try to load an **inaccessible address**
- Permission is **checked**
• Instruction Set Architecture (ISA) is an abstract model of a computer (x86, ARMv8, SPARC, ...)
• Instruction Set Architecture (ISA) is an abstract model of a computer (x86, ARMv8, SPARC, …)
• Serves as the interface between hardware and software
• Instruction Set Architecture (ISA) is an abstract model of a computer (x86, ARMv8, SPARC, …)
• Serves as the interface between hardware and software
• Microarchitecture is an actual implementation of the ISA
• Instruction Set Architecture (ISA) is an abstract model of a computer (x86, ARMv8, SPARC, ...)
• Serves as the interface between hardware and software
• Microarchitecture is an actual implementation of the ISA
• Safe software infrastructure does not mean safe execution
Side-channel Attacks

• Safe software infrastructure does not mean safe execution
• Information leaks because of the underlying hardware
Side-channel Attacks

- Safe software infrastructure does not mean safe execution
- Information leaks because of the underlying hardware
- Exploit unintentional information leakage by side-effects
Side-channel Attacks

- Safe software infrastructure does not mean safe execution
- Information leaks because of the underlying hardware
- Exploit unintentional information leakage by side-effects

CPU Cache

printf("%d", i);
printf("%d", i);
Cache miss

printf("%d", i);
printf("%d", i);
printf("%d", i);
printf("%d", i);
printf("%d", i);
printf("%d", i);
CPU Cache

```c
printf("%d", i);
printf("%d", i);
```
CPU Cache

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

Cache miss

Cache hit

Request
Response
CPU Cache

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

Cache hit

Cache miss

Request
Response

No DRAM access, much faster
CPU Cache

DRAM access, slow

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

Cache miss

Cache hit

No DRAM access, much faster

Request
Response
Memory Access Latency

Measured access time (CPU cycles)

Number of accesses

Cache hit
Cache miss

Flush+Reload

ATTACKER

Shared Memory

VICTIM

Flush+Reload

ATTACKER

flush

access

Shared Memory

VICTIM

access
Flush+Reload

ATTACKER

flush
access

Shared Memory

VICTIM

access

Flush+Reload

ATTACKER

flush
access

Shared Memory

VICTIM

access

Victim accessed (fast) vs Victim did not access (slow)

Out-of-Order Execution

Instructions are
- fetched and decoded in the front-end
Instructions are

- fetched and decoded in the **front-end**
- dispatched to the **backend**
Out-of-Order Execution

Instructions are

• fetched and decoded in the front-end
• dispatched to the backend
• processed by individual execution units
Out-of-Order Execution

Instructions

- are executed out-of-order
Out-of-Order Execution

Instructions

- are executed out-of-order
- wait until their dependencies are ready
Instructions

- are executed **out-of-order**
- wait until their dependencies are ready
  - Later instructions might execute prior earlier instructions
Out-of-Order Execution

Instructions

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

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

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

- are executed out-of-order
- wait until their dependencies are ready
- Later instructions might execute prior earlier instructions
- retire in-order
- State becomes architecturally visible
- Exceptions are checked during retirement
- Flush pipeline and recover state
*(volatile char*) 0; // raise_exception();
array[84 * 4096] = 0;
• Flush+Reload over all pages of the array
Building the Code

• Flush+Reload over all pages of the array

• “Unreachable” code line was actually executed
• Flush+Reload over all pages of the array

• “Unreachable” code line was actually executed
• Exception was only thrown afterwards
Out-of-order instructions leave microarchitectural traces.
• Out-of-order instructions leave microarchitectural traces
  • We can see them for example in the cache
• Out-of-order instructions leave microarchitectural traces
  • We can see them for example in the cache
• Give such instructions a name: transient instructions
• Out-of-order instructions leave microarchitectural traces
  • We can see them for example in the cache
• Give such instructions a name: transient instructions
• We can indirectly observe the execution of transient instructions
Building blocks

Section 4.1
Architectural State
Transfer (Covert Channel)
Recovered Secret
Leaked
Recovery

Section 4.2
Exception Handling/Suppression
Transient Instructions
Microarchitectural State Change

Accessed
Secret

Executing transient instructions

- Transient instructions are executed all the time
Executing transient instructions

• Transient instructions are executed all the time
• Loading inaccessible addresses leads to a crash (segfault)
Transient instructions are executed all the time
Loading inaccessible addresses leads to a crash (segfault)
How to prevent the crash?
Executing transient instructions

- Transient instructions are executed all the time
- Loading inaccessible addresses leads to a crash (segfault)
- How to prevent the crash?
Building a Covert Channel

- Transfer of the microarchitectural state into an architectural state
- Transient instruction sequence is the sender
- Receiver receives the microarchitectural state change and deduces the secret from the state
Building a Covert Channel

• Leverage techniques from cache attacks: Flush+Reload
• Transmit multiple bits at once
  • 256 different byte values ⇒ access different cache line
• Not limited to the cache
• Add another *layer of indirection* to test

```c
char data = *(char*) 0xfffffffff81a000e0;
array[data * 4096] = 0;
```
• Add another layer of indirection to test

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

• Then check whether any part of array is cached
Building the Code

- Flush+Reload over all pages of the array

- Index of cache hit reveals data
• Flush+Reload over all pages of the array

• Index of cache hit reveals data

• Permission check is in some cases not fast enough
• Using out-of-order execution, we can read data at any address
• Using out-of-order execution, we can read data at any address
• Index of cache hit reveals data
Meltdown

- Using out-of-order execution, we can read data at any address
- Index of cache hit reveals data
- Permission check is in some cases not fast enough
• Using out-of-order execution, we can read data at any address
• Index of cache hit reveals data
• Permission check is in some cases not fast enough
• Entire physical memory is typically accessible through kernel space
Demo
Uncached memory

- Assumed that one can only read data \textit{stored in the L1} with Meltdown
• Assumed that one can only read data stored in the L1 with Meltdown
• Experiment where a thread flushes the value constantly and a thread on a different core reloads the value
Uncached memory

• Assumed that one can only read data stored in the L1 with Meltdown
• Experiment where a thread flushes the value constantly and a thread on a different core reloads the value
  • Target data is not in the L1 cache of the attacking core
Uncached memory

- Assumed that one can only read data stored in the L1 with Meltdown
- Experiment where a thread flushes the value constantly and a thread on a different core reloads the value
  - Target data is not in the L1 cache of the attacking core
- We can still leak the data at a lower reading rate
Uncached memory

- Assumed that one can only read data stored in the L1 with Meltdown
- Experiment where a thread flushes the value constantly and a thread on a different core reloads the value
  - Target data is not in the L1 cache of the attacking core
- We can still leak the data at a lower reading rate
- Meltdown might implicitly cache the data
• Mark pages in page tables as UC (uncachable)
Uncachable memory

- Mark pages in page tables as UC (uncachable)
  - Every read or write operation will go to main memory
Uncachable memory

- Mark pages in page tables as UC (uncachable)
  - Every read or write operation will go to main memory
- If the attacker can trigger a legitimate load (system call, ...) on the same CPU core, the data still can be leaked
Uncachable memory

- Mark pages in page tables as UC (uncachable)
  - Every read or write operation will go to main memory
- If the attacker can trigger a legitimate load (system call, ...) on the same CPU core, the data still can be leaked
- Meltdown might read the data from one of the fill buffers
Uncachable memory

• Mark pages in page tables as UC (uncachable)
  • Every read or write operation will go to main memory
• If the attacker can trigger a legitimate load (system call, ...) on the same CPU core, the data still can be leaked
• Meltdown might read the data from one of the fill buffers
  • as they are shared between threads running on the same core
Affected by Meltdown

- **Intel**: Almost every CPU

Affected by Meltdown

- **Intel**: Almost every CPU
- **AMD**: Seems not to be affected
Affected by Meltdown

- **Intel**: Almost every CPU
- **AMD**: Seems not to be affected
- **ARM**: Only the Cortex-A75
Affected by Meltdown

- **Intel**: Almost every CPU
- **AMD**: Seems not to be affected
- **ARM**: Only the Cortex-A75
- **IBM**: System Z, Power Architecture, POWER8 and POWER9
Affected by Meltdown

- **Intel**: Almost every CPU
- **AMD**: Seems not to be affected
- **ARM**: Only the Cortex-A75
- **IBM**: System Z, Power Architecture, POWER8 and POWER9
- **Apple**: All Mac and iOS devices
SAMSUNG GALAXY S7

- Exynos Mongoose M1 CPU Architecture
Samsung Galaxy S7

Samsung Galaxy S7
• Exynos Mongoose M1 CPU Architecture
  • Custom CPU core in the Exynos 8 Octa (8890)
Samsung Galaxy S7

**Samsung Galaxy S7**

- Exynos Mongoose M1 CPU Architecture
  - Custom CPU core in the Exynos 8 Octa (8890)
  - Perceptron Branch Prediction
Samsung Galaxy S7

Samsung Galaxy S7

- Exynos Mongoose M1 CPU Architecture
  - Custom CPU core in the Exynos 8 Octa (8890)
  - Perceptron Branch Prediction
  - Full Out-of-Order Instruction Execution
Samsung Galaxy S7

Samsung Galaxy S7

- Exynos Mongoose M1 CPU Architecture
  - Custom CPU core in the Exynos 8 Octa (8890)
  - Perceptron Branch Prediction
  - Full Out-of-Order Instruction Execution
    - Full Out-of-Order loads and stores
• Dumping the entire physical memory takes some time
Meltdown in Practice

- Dumping the entire physical memory takes some time
  - L1: 582 KB/s
  - L3: 12.4 KB/s
  - Uncached: 10 B/s (improved: 3.2 KB/s)
- Not very practical in most scenarios
Meltdown in Practice

• Dumping the entire physical memory takes some time
  • L1: 582 KB/s
  • L3: 12.4 KB/s
  • Uncached: 10 B/s (improved: 3.2 KB/s)

• Not very practical in most scenarios
Breaking KASLR

• De-randomize KASLR to access *internal kernel structures*

• De-randomize KASLR to access internal kernel structures
• Locate a known value inside the kernel, e.g., Linux banner
Breaking KASLR

• De-randomize KASLR to access internal kernel structures
• Locate a known value inside the kernel, e.g., Linux banner
  • Start at the default address according to the symbol table of the running kernel
• De-randomize KASLR to access internal kernel structures
• Locate a known value inside the kernel, e.g., Linux banner
  • Start at the default address according to the symbol table of the running kernel
  • Linux KASLR has an entropy of 6 bits ⇒ only 64 possible randomization offsets
Breaking KASLR

• De-randomize KASLR to access internal kernel structures
• Locate a known value inside the kernel, e.g., Linux banner
  • Start at the default address according to the symbol table of the running kernel
  • Linux KASLR has an entropy of 6 bits ⇒ only 64 possible randomization offsets
• Difference between the found address and the non-randomized base address is the randomization offset
Locating the victim process

- Linux manages all processes in a linked list
Locating the victim process

- Linux manages all processes in a linked list
- Head of the list is stored at init_task structure
Locating the victim process

- Linux manages all processes in a linked list
- **Head of the list** is stored at `init_task` structure
  - At a fixed offset that varies only among kernel builds
Locating the victim process

- Linux manages all processes in a linked list
- **Head of the list** is stored at `init_task` structure
  - At a fixed offset that varies only among kernel builds
- Each task list structure contains a **pointer to the next task** and
  - PID of the task
  - name of the task
  - Root of the multi-level page table
Dumping memory content

- Resolve physical address using paging structures
Dumping memory content

- **Resolve physical address** using paging structures
- **Read the content using the direct-physical map**
Dumping memory content

• Resolve physical address using paging structures
• Read the content using the direct-physical map
• Enumerate all mapped pages and dump entire process memory

Dumping memory content

- Resolve physical address using paging structures
- Read the content using the direct-physical map
- Enumerate all mapped pages and dump *entire process memory*
- Location of the key known, we can just dump the key directly
• Problem is rooted in hardware
Meltdown Mitigation

- Problem is rooted in hardware
- Race condition between the memory fetch and corresponding permission check
  - Serialize both of them
Meltdown Mitigation

- Problem is rooted in **hardware**
- **Race condition** between the memory fetch and corresponding permission check
  - Serialize both of them
- **Hard split** of user space and kernel space
  - New bit in control register
Meltdown Mitigation

- Problem is rooted in **hardware**
- **Race condition** between the memory fetch and corresponding permission check
  - Serialize both of them
- **Hard split** of user space and kernel space
  - New bit in control register

- **Fix the hardware** → long-term solution
Meltdown Mitigation

- Problem is rooted in **hardware**
- **Race condition** between the memory fetch and corresponding permission check
  - Serialize both of them
- **Hard split** of user space and kernel space
  - New bit in control register

- **Fix the hardware** → long-term solution
- Can we fix it in **software**?
KAISER

Userspace

Kernelspace

Applications

Operating System

Memory

• KAISER [Gru+17] has been published in May 2017 ...
• KAISER [Gru+17] has been published in May 2017 ...
• ...as a countermeasure against other side-channel attacks
• KAISER [Gru+17] has been published in May 2017 ...
• ...as a countermeasure against other side-channel attacks
• Inadvertently defeats Meltdown as well
Mitigations

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

---

Mitigations

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

- **Linux**: Kernel Page-table Isolation (KPTI)
- **Apple**: Released updates
- **Windows**: Kernel Virtual Address (KVA) Shadow
You can find our proof-of-concept implementation on:

• https://github.com/IAIK/meltdown
Conclusion

• Underestimated microarchitectural attacks for a long time
• Meltdown allows to read arbitrary kernel memory from user space
• Affecting millions of devices of various CPU manufacturers
• Countermeasures come with a performance impact
Meltdown

Reading Kernel Memory from User Space

Moritz Lipp¹, Michael Schwarz¹, Daniel Gruss¹, Thomas Prescher², Werner Haas², Anders Fogh³, Jann Horn⁴, Stefan Mangard¹, Paul Kocher⁵, Daniel Genkin⁶, Yuval Yarom⁷, Mike Hamburg⁸

27th USENIX Security Symposium - August 16, 2018

¹Graz University of Technology, ²Cyberus Technology GmbH, ³G-Data Advanced Analytics, ⁴Google Project Zero, ⁵Independent (www.paulkocher.com), ⁶University of Michigan, ⁷University of Adelaide & Data61, ⁸Rambus, Cryptography Research Division