Meltdown and Spectre

Vinod Ganapathy

March 17, 2018

Goals of this talk

- Understand the technical details of Meltdown and Spectre.
- Explore what went wrong, and examine proposed defenses.

- Let's keep the talk interactive and informal! Feel free to ...
  - … correct/question me if you find anything incorrect/unclear.
  - … add interesting tidbits that you may have heard/read about the attacks.

Why are the attacks so dangerous?

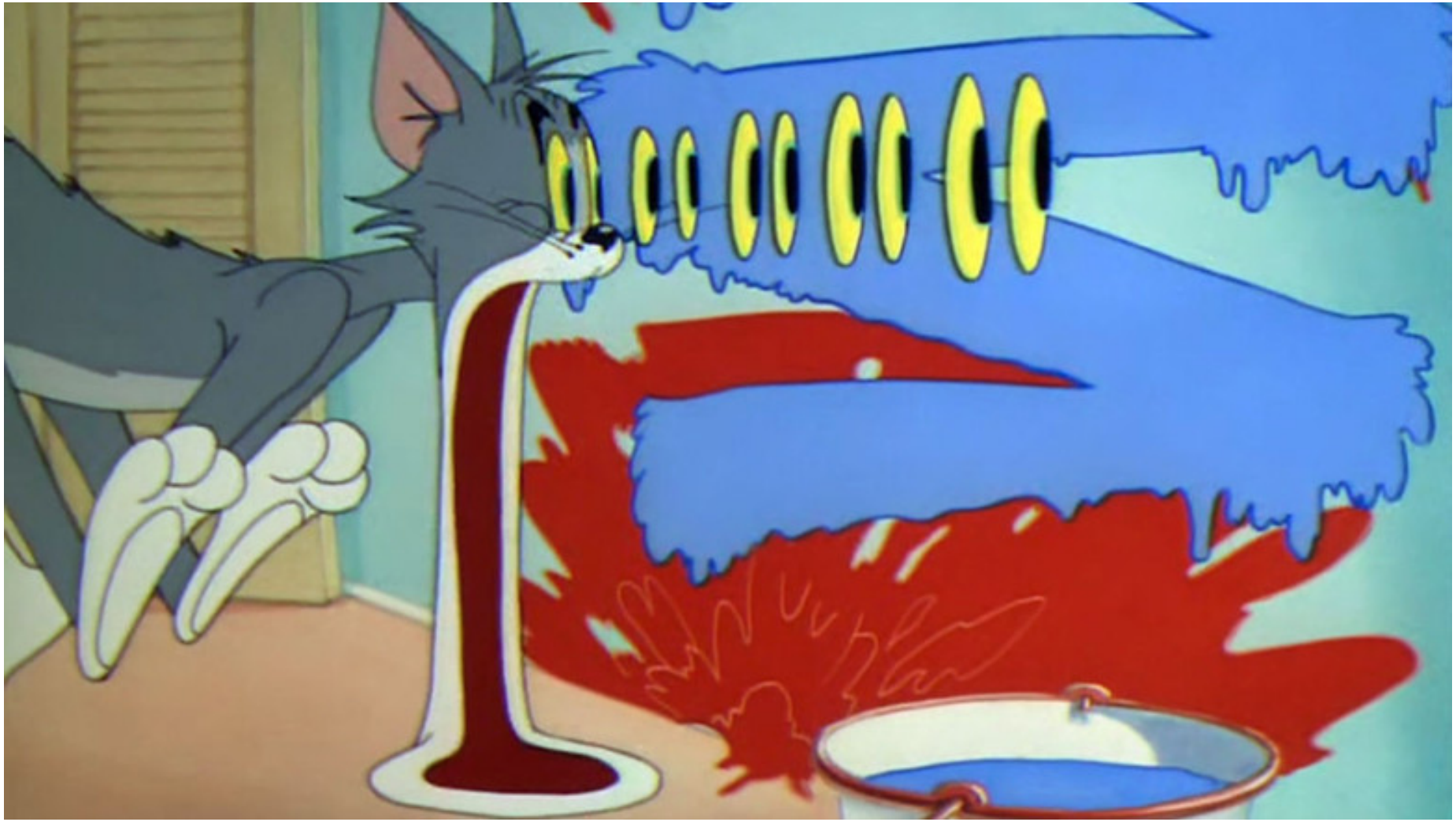
- Allow an attacker to read the contents of arbitrary memory locations on a victim computer:
  - Stored passwords, cryptographic keys, credit card numbers, etc.
- The attacks do not exploit any known software vulnerabilities (at least not in the traditional sense).
- Defence proposed only for Meltdown, not Spectre.
  - Proposed Meltdown defence has drawn criticism; is not yet reliable.

Linus Torvalds declares Intel fix for Meltdown/Spectre 'COMPLETE AND UTTER GARBAGE'

Why are the attacks so spectacular?

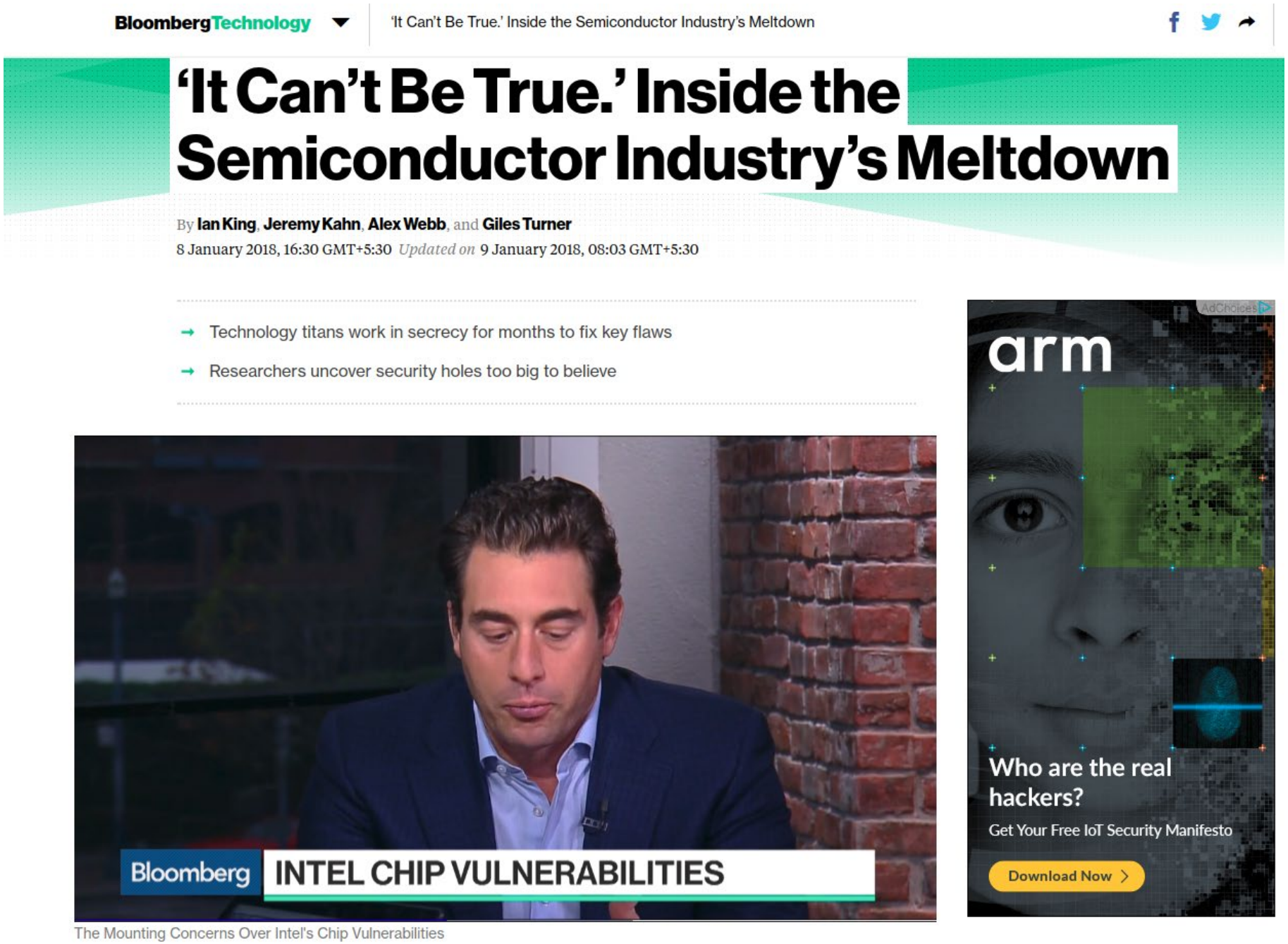
- Both attacks exploit tricks proposed in the computer architecture community decades ago!
  - Meltdown exploits out-of-order execution.
  - Spectre exploits speculative execution.
- Both tricks are completely standard and implemented by all modern processors.
  - No flaw in the proposed design/specification of these architecture tricks.
  - These tricks are now **considered an essential part** of the performance delivered by the processor.
  - Cannot simply turn them off as a defence!

Reactions:



Reactions:

Disbelief



Reactions:

Amazement

Adrian Sampson
Department of Computer Science
Cornell University

Spectacular

JANUARY 16, 2018

Spectre has nerdsniped me, hard. I've been walking into lampposts and stuff. The more I think about it, the less I understand it.

The first shocking thing is that, once you read about it, the problem is so easy to see. To summarize: predictor state is untrusted, and mispredicted execution paths can leave traces in the memory system, so malicious code can observe the behavior of "impossible" paths. It's a fundamental problem in an idea that's been architectural gospel for decades. It's one of those obvious-in-retrospect epiphanies that makes me rethink everything.

The second thing is that it's not just about speculation. We now live in a world with side channels in microarchitectures that leave no real trace in the machine's architectural state. There is already work on leaks through prefetching, where someone learns about your activity by observing how it affected a reverse-engineered prefetcher. You can imagine similar attacks on TLB state, store buffer coalescing, coherence protocols, or even replacement policies. Suddenly, the SMT side channel doesn't look so bad.

Reactions:

Scientific postmortem analyses

Andrew Myers

Security, programming languages, and computer systems

ABOUT

Meltdown, Spectre, and why hardware can be correct yet insecure

JANUARY 17, 2018 ~ ANDREW MYERS

The recent Meltdown and Spectre attacks have exposed, or at least emphasized, a fundamental problem with the conventional approach to computer security at the hardware level. Both of these attacks rely on *side channels* in conventional processor designs. By exploiting these side channels, an untrusted program can learn the contents of the operating system kernel's memory or of another process's memory, bypassing the standard hardware protection mechanisms based on virtual memory.

The initial response from Intel was apparently that Meltdown and Spectre did not reveal any bugs in their processor implementations — in other words, that their processors were implemented correctly. What's perhaps more surprising is that they have a pretty good case for making that claim. According to the commonly used definitions of correctness, the processors of Intel, AMD, and other manufacturers — all vulnerable to Spectre — are indeed correct with respect to the way they are used

Search …

Recent Posts

Meltdown, Spectre, and why hardware can be correct yet insecure

A pet peeve about hash tables

Deserialization considered harmful: the security case for persistent objects

Java vs. OCaml vs. Scala

Limits of Heroism

Reactions:

From the pioneers of speculative execution

DEPARTMENT OF
Computer Sciences
UNIVERSITY OF WISCONSIN–MADISON

AUTHENTICATED SERVICES CSL MY UW UW-MADISON

HIRING About People Education Research Connect Give

About
Location
Contacts
Events
Feature Articles
News Items
Employment

Despite current security concerns, speculative execution has powered the computing revolution

Submitted by Jennifer Smith on Wed, 01/17/2018 - 9:32am

The recently discovered Spectre and Meltdown vulnerabilities, which affect microprocessors in the majority of the world's computers, have dominated tech news in the last two weeks. Though there is no current evidence that hackers have successfully exploited these vulnerabilities, Spectre and Meltdown make it possible for bad actors to gain access to stored information. Security researchers around the world have been working on fixes.

Yet there is more to the story: speculative execution, the hardware feature that has led to this security vulnerability, also provides significant performance benefits and has been instrumental in the continued increase in microprocessor performance for the past couple of decades.

Because of their significant performance advantage, speculative execution techniques, developed by researchers at the University of Wisconsin-Madison, have been in use in billions of microprocessors worldwide for the past couple of decades.

These techniques, together with enabling approaches like branch prediction, allow a computer chip to make "educated guesses" regarding the commands it will be doing in the near future, so that it can get a head start in performing these commands. This leads to significantly increased overlap, or parallelism, in performing the commands. Resulting performance gains have made possible countless things that consumers and businesses now take for granted: fast video streaming, online payment systems, cloud computing and much more.

Following up on pioneering work on branch prediction by Jim Smith in the early 1980s, and other work by Smith and Andy Pleszkun in the mid-1980s, a model for a speculative execution microprocessor was proposed in the mid- to late 1980s in work done by Guri Sohi. This was years before the proliferation of the Internet, the early Internet worms, and the first Web browser.

It is critical to note that it is not the concept of speculative execution that creates security vulnerabilities, but rather how the approach is implemented by microprocessor designers.

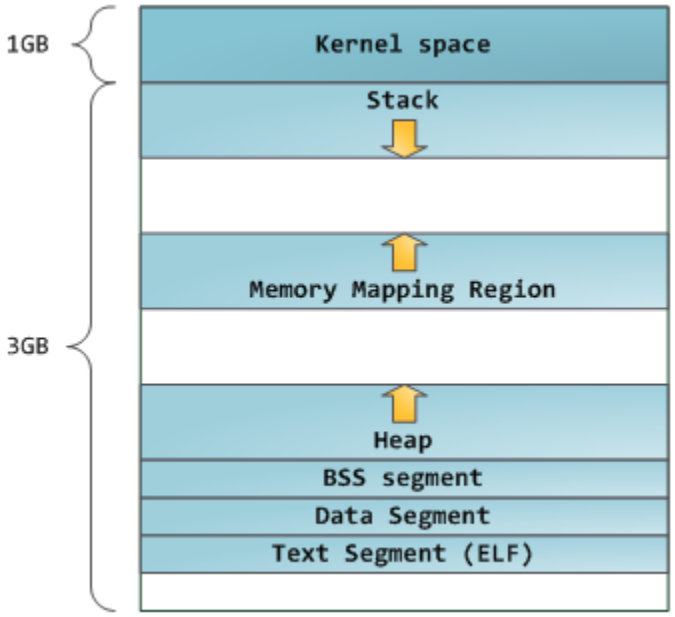
"Different implementations of the speculative execution model carry different risks," says Sohi, chair of the Computer Sciences Department (pictured at left). Sohi is also Vilas Research Professor, John P. Morgridge Professor and E. David Cronon Professor of Computer Sciences.

Because speculative execution makes guesses about what work a program needs to do, it brings in information that may ultimately not be needed. "There's information kept around as a result of speculative execution that would not normally be accessible under legitimate circumstances," says Sohi. The problem is that hackers, in very clever and indirect ways, using sophisticated

Outline

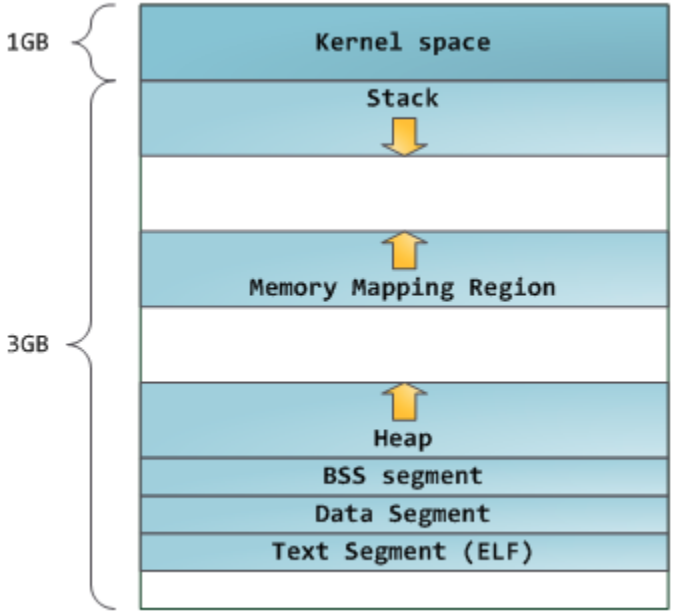
- Background
- Meltdown attack
- Spectre attack
- Open discussion: What went wrong? How can we fix it?

(Virtual) Address spaces



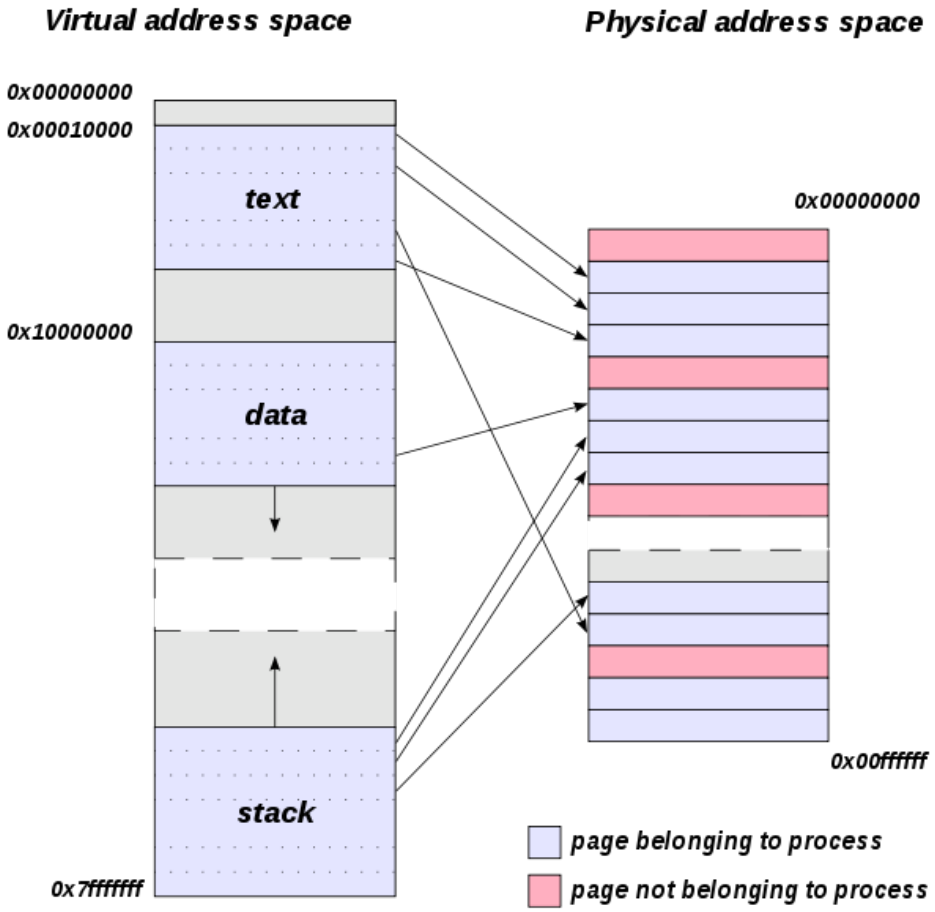
- Every process has a virtual address space: 4GB on 32-bit machines, ~281TB on 64-bit (only 48 bits currently used).
- Process code and data loaded/allocated into the address space.
- Every process's address space is isolated from every other process's address space
- BUT, kernel code and data loaded into every process's address space (e.g., top 1GB in the picture on the left, for 32-bit machines)

Isolating kernel memory from userland



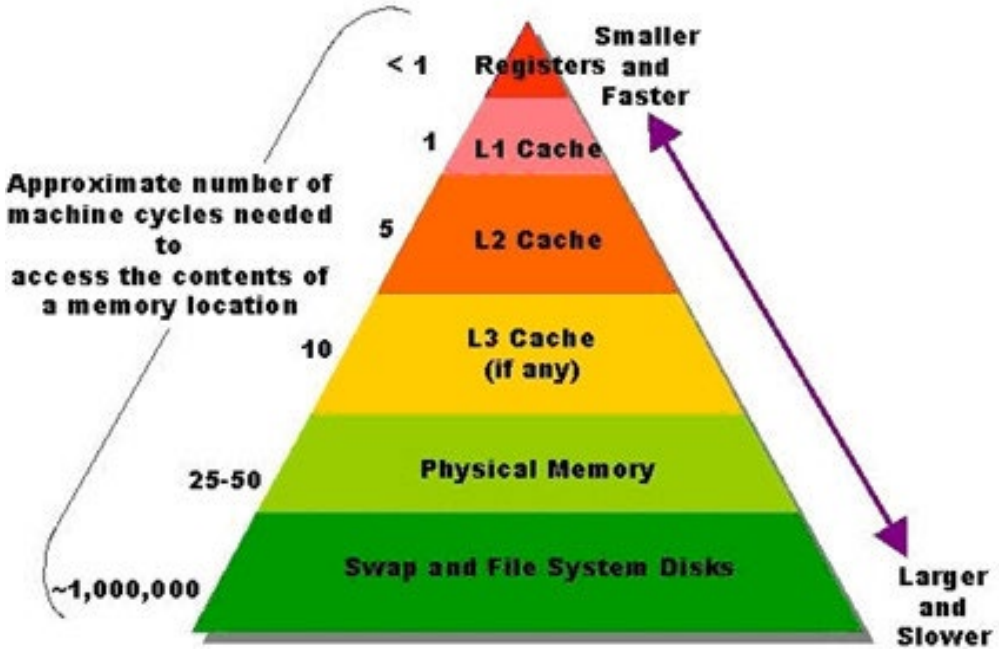
- Userland code must not directly access kernel memory.
- Kernel contains sensitive info:
  - Info about other processes.
  - Typically, all of physical memory is mapped into the kernel address space.
- If userland code attempts to directly access kernel memory, hardware triggers an exception.

Virtual and physical memory



- The OS maps each process' virtual address space to physical memory via per-process page tables.
- Pages tables for all user processes are managed by the kernel, i.e., kernel knows virtual to physical mappings for all processes.
- Kernel itself is mapped into process address space: Kernel's own virtual to physical mappings are part of the page table.

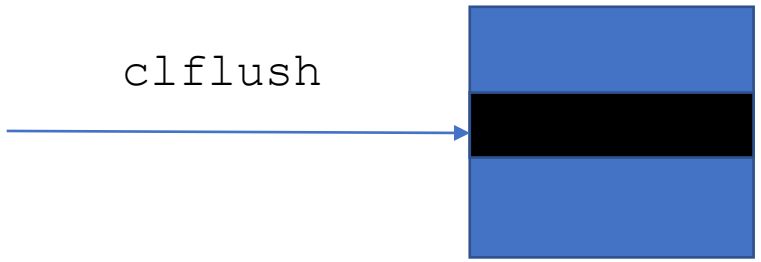
The memory hierarchy



- On-chip caches provide fast access to data.
- Typically L1 and L2 private to individual cores (on multi-core machine)
- L3 is common to all cores (also called **LLC**).
- Caches store objects in **cache-lines.** Size of cache-line varies: e.g., 64 bytes on Core-i7 for L3 caches.

Building block: Cache-based covert-channels

- **Goal**: Player A uses the faster access times of cache memory as a side-channel to communicate with Player B
- **Step 1:** Flush the cache (so it's empty)



Building block: Cache-based covert-channels

- **Goal**: Player A uses the faster access times of cache memory as a side-channel to communicate with Player B
- **Step 2:** Setup a probe array in memory. Array is set up such that each element of array occupies a different cache line

Create a Probe Array in memory

Building block: Cache-based covert-channels

- **Goal**: Player A uses the faster access times of cache memory as a side-channel to communicate with Player B
- **Step 3:** Player B wants to communicate value **X** to Player A. He does so by accesses Probe_Array[**X**]

To communicate **X**:
Access index **X**

Building block: Cache-based covert-channels

- **Goal**: Player A uses the faster access times of cache memory as a side-channel to communicate with Player B
- **Step 3: [Microarchitectural state change]** That array element is loaded into the appropriate cache line

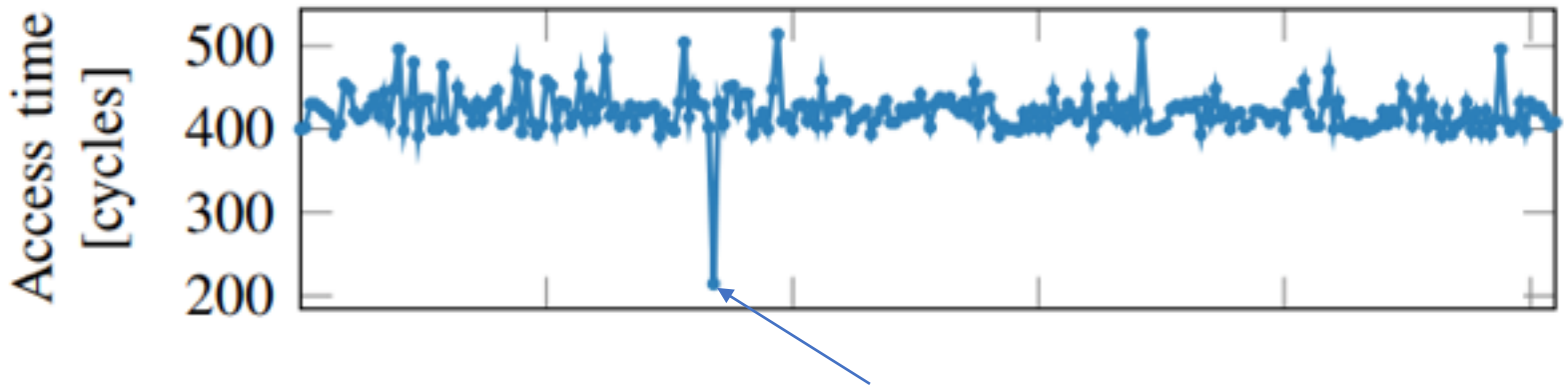
Element loaded into cache

Building block: Cache-based covert-channels

- **Goal**: Player A uses the faster access times of cache memory as a side-channel to communicate with Player B
- **Step 4:** Player A reads Probe_Array sequentially and measures the access time for each array element.

Read the probe array sequentially

Building block: Cache-based covert-channels



Smaller read time when Probe_Array[**X**] is accessed

Side channels vs. Covert channels

- In a covert channel, the attacker controls both ends of the timing channel
- Can be used by attacker to transmit bits of information from one end to the other.
- Meltdown uses such a covert channel to read kernel memory from userland.

Building block: Out-of-order execution

- Allows instructions to execute speculatively, based on data availability, rather than execute instructions in program order.
- **Key to high performance**: when program completes "in order", speculative state will have computation results ready, so cycles are not wasted along the critical path.
- **Example**:

in-order processors

`lw $3, 100($4)`	in execution, cache miss
`add $2, $3, $4`	waits until the miss is satisfied
`sub $5, $6, $7`	waits for the add

out-of-order processors

`lw $3, 100($4)`	in execution, cache miss
`sub $5, $6, $7`	can execute during the cache miss
`add $2, $3, $4`	waits until the miss is satisfied

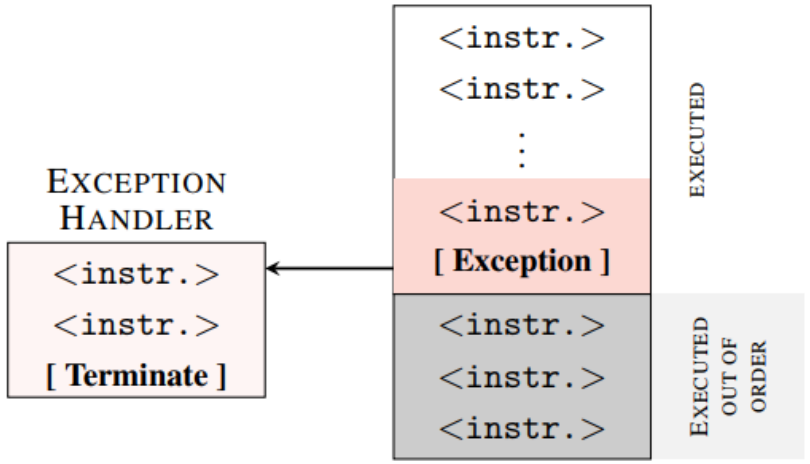
Building block: Out-of-order execution

- Above example was an instance of **dynamic scheduling.**
- Can also execute code with branches: requires **branch prediction**.
- An instance of **speculative execution**.
- Instructions execute speculatively out of order, but **commit** only when speculative state is concretized via actual execution. Instructions that have completed are said to have **retired.**
- Processors implement this using Tomasulo's algorithm.

Effects of OOO on the cache

```
1 raise_exception();
2 // the line below is never reached
3 access(probe_array[data * 4096]);
```

Listing 1: A toy example to illustrate side-effects of out-of-order execution.



Effects of OOO on the cache

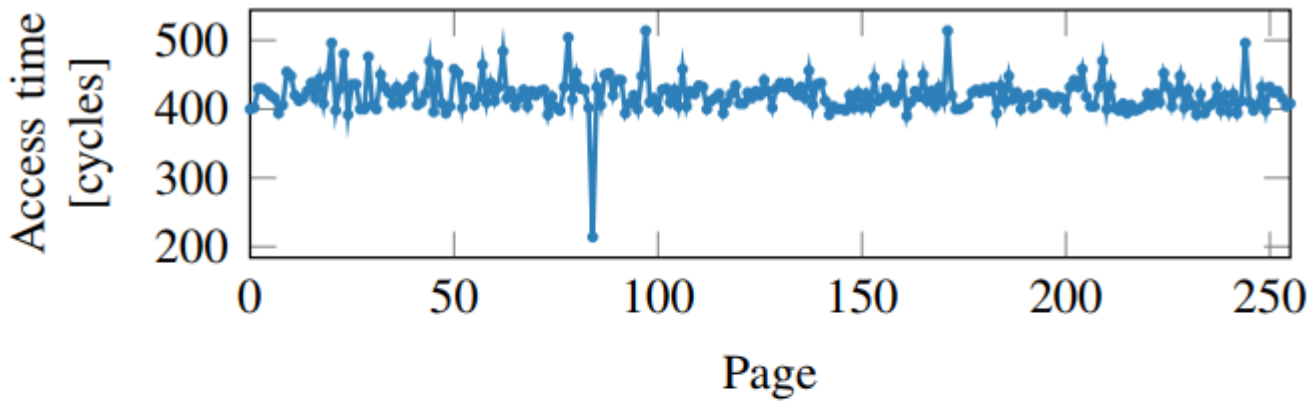


Figure 4: Even if a memory location is only accessed during out-of-order execution, it remains cached. Iterating over the 256 pages of `probe_array` shows one cache hit, exactly on the page that was accessed during the out-of-order execution.

Meltdown: The Covert Channel Setup

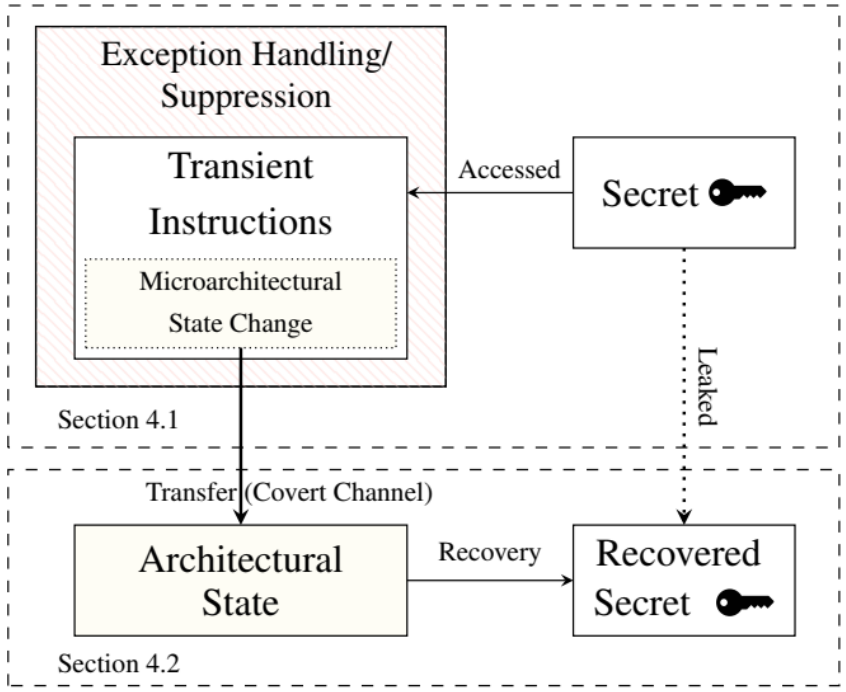
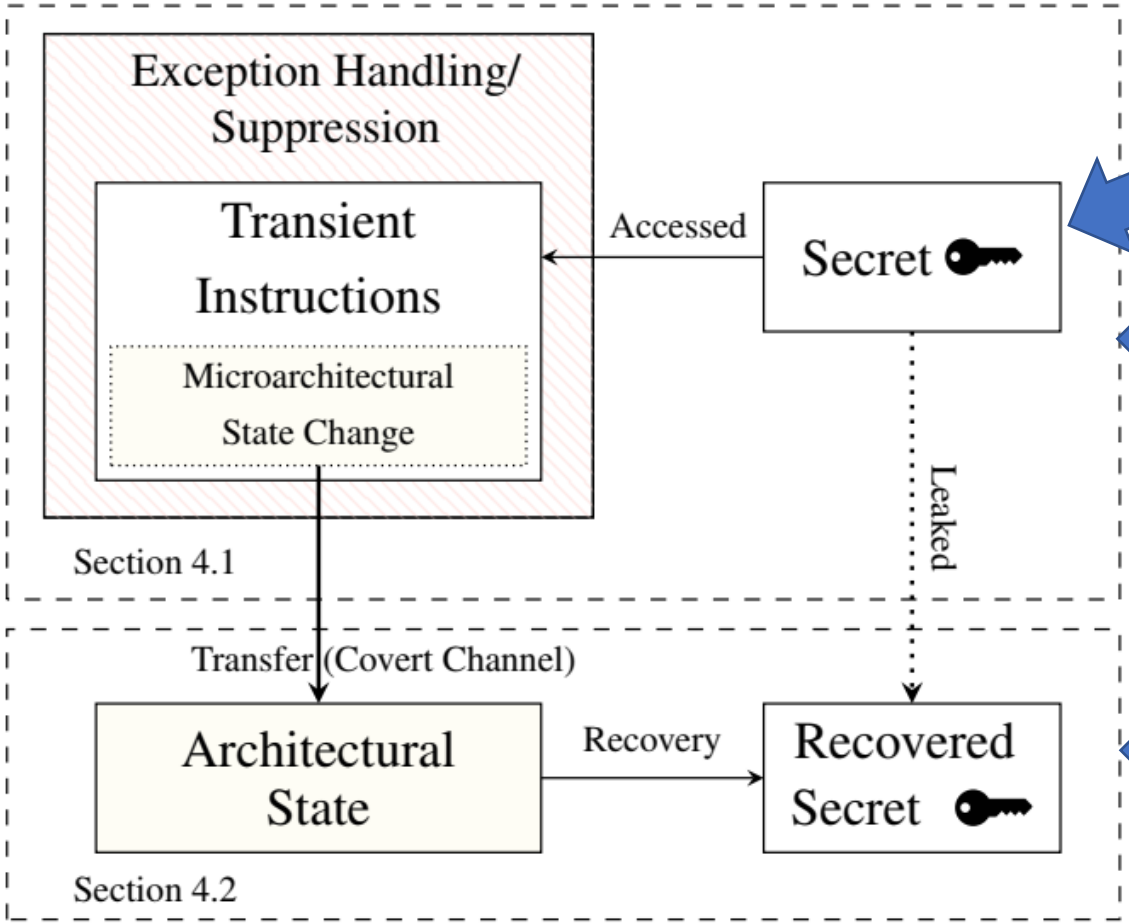


Figure 5: The Meltdown attack uses exception handling or suppression, e.g., TSX, to run a series of transient instructions. These transient instructions obtain a (persistent) secret value and change the microarchitectural state of the processor based on this secret value. This forms the sending part of a microarchitectural covert channel. The receiving side reads the microarchitectural state, making it architectural and recovering the secret

Meltdown: The Covert Channel Setup

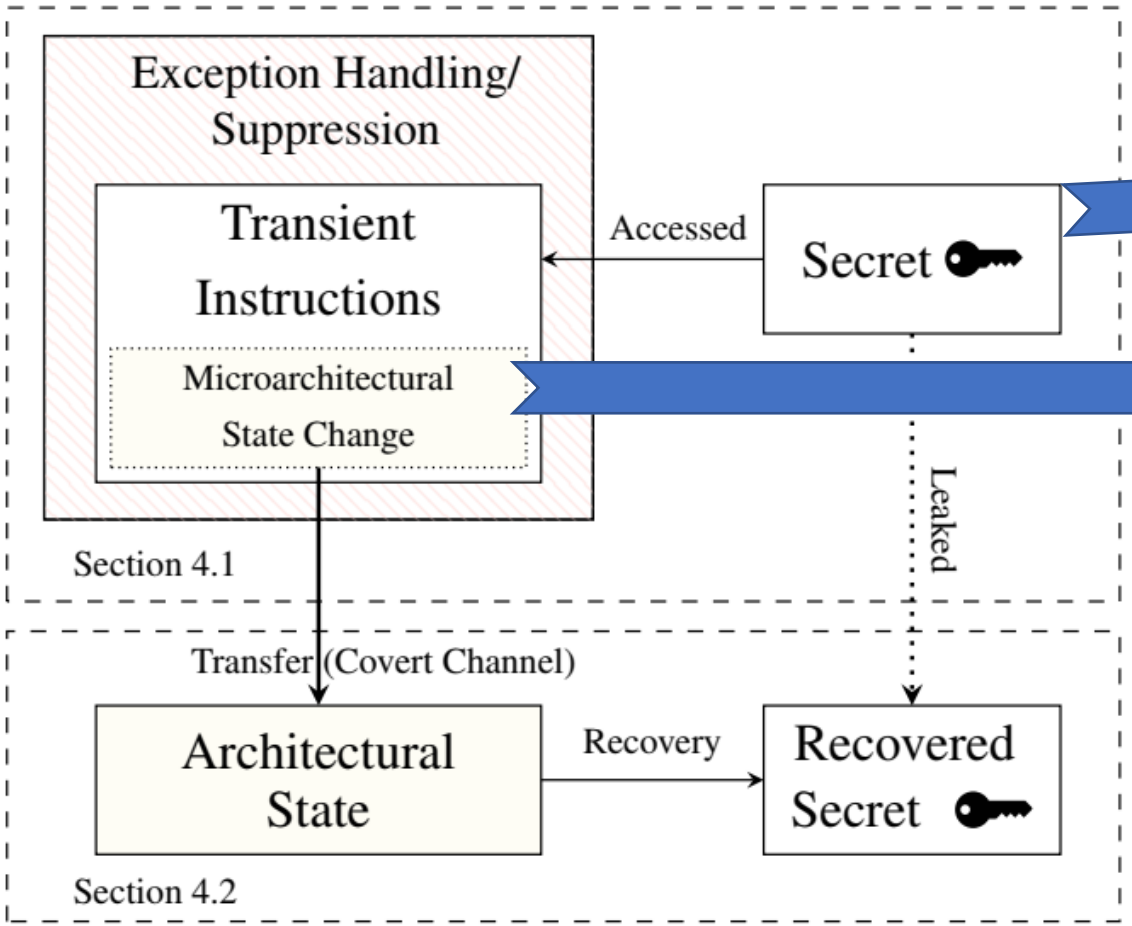


A value of interest, stored in kernel memory (typically inaccessible to user processes)

Forked child process (dies on exception, but affects cache state)

Parent process

Meltdown: Transient Instruction Sequence




```
1 ; rcx = kernel address
2 ; rbx = probe array
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

The Meltdown Attack

```
1 ; rcx = kernel address
2 ; rbx = probe array
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

- **Goal**: Attacker wants to learn the value of the byte stored at a particular kernel memory address (address in the `rcx` register).
- **Step 1: Reading the secret (Line 4) mov al, byte [rcx]**
  - Loads the byte value stored at the address RCX into AL (LSB of RAX)
  - This instruction should cause an exception if executed in userland. Subsequent instructions should never be executed.
  - **BUT,** due to OOO, subsequent instructions may already be executed speculatively.
  - Exceptions handled only when **Line 4 is retired.** By then, microarchitectural state is already affected by subsequent OOO instruction execution.

The Meltdown Attack

```
1 ; rcx = kernel address
2 ; rbx = probe array
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

- **Goal**: Attacker wants to learn the value of the byte stored at a particular kernel memory address (address in the `rcx` register).
- **Step 2: Transmitting the secret (Line 5) shl rax, 0xc**
  - Multiply the byte value **X** by the page size (4K).
  - This will be used to index into a **probe array** (base address in RBX)**.**
  - A large spatial distance ensures that neighboring locations of the probe array are not loaded into the cache (due to spatial locality optimizations).
  - Probe array is of size 256 * 4K bytes, since we only have 256 possible byte values.

The Meltdown Attack

```
1 ; rcx = kernel address
2 ; rbx = probe array
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

- **Goal**: Attacker wants to learn the value of the byte stored at a particular kernel memory address (address in the `rcx` register).
- **Step 2: Transmitting the secret (Line 7) mov rbx, qword [rbx + rax]**
  - Read Probe_Array[X] (each entry is 4K bytes long).
  - The value will be stored into the corresponding cache line
- **Step 3: Receiving the secret (Parent process)**
  - Parent process probes the cache by iterating through Probe_Array[].
  - Only the read of **Probe_Array[X]** will be a hit in the cache.
  - Attacker learns the value of **X**

The Meltdown Attack

```
1 ; rcx = kernel address
2 ; rbx = probe array
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

- What is the role of line 3 and line 6?
  - Race conditions!
  - The attacker is racing against the hardware: Must get transient instructions to execute and affect microarchitectural state before the exception for **line 4** is thrown.
  - In some machines, exception is not handled, and process crashes, but processor zeroes out registers before crashing the process.
  - If zeroing out happens faster than the operation in **line 5,** attacker will read the wrong value for **X.** So the code retries.
  - Instruction sequence called a **0-noise-bias.**

Meltdown attack application: Memory dumps

- Can iterate attack across a range of memory addresses to obtain a complete memory dump of the kernel.
- Physical memory on modern machines mapped at an offset within the kernel. So complete dump of physical memory is possible.

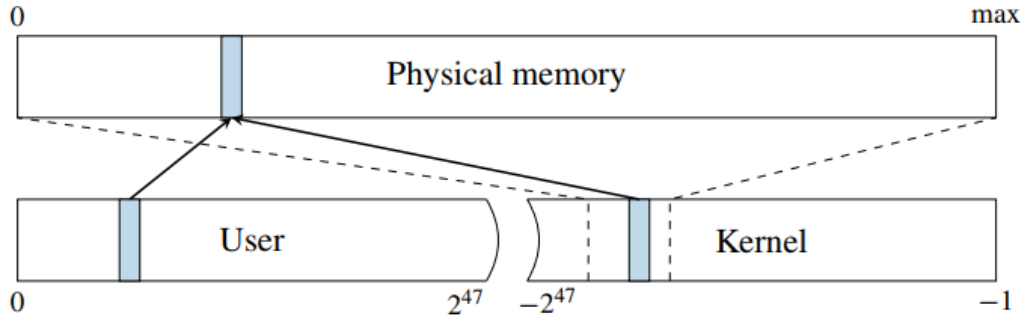


Figure 2: The physical memory is directly mapped in the kernel at a certain offset. A physical address (blue) which is mapped accessible for the user space is also mapped in the kernel space through the direct mapping.

Meltdown attack application: Memory dumps

- Attack against Firefox56 running atop a Ubunto 16.10/Linux-4.8.0 machine on Intel Corei7-6700K

```
f94b7690: e5 e5 e5 e5 e5 e5 e5 e5  e5 e5 e5 e5 e5 e5 e5 e5 |................|
f94b76a0: e5 e5 e5 e5 e5 e5 e5 e5  e5 e5 e5 e5 e5 e5 e5 e5 |................|
f94b76b0: 70 52 b8 6b 96 7f XX XX XX  XX XX XX XX XX XX XX |pR.k............|
f94b76c0: 09 XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b76d0: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b76e0: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX 81 |................|
f94b76f0: 12 XX e0 81 19 XX e0 81 44  6f 6c 70 68 69 6e 31 |........Dolphin1|
f94b7700: 38 e5 e5 e5 e5 e5 e5 e5 e5  e5 e5 e5 e5 e5 e5 e5 |8...............|
f94b7710: 70 52 b8 6b 96 7f XX XX XX  XX XX XX XX XX XX XX |pR.k............|
f94b7720: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b7730: XX XX XX XX 4a XX XX XX XX  XX XX XX XX XX XX XX |....J...........|
f94b7740: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b7750: XX XX XX XX XX XX XX XX XX  XX e0 81 69 6e 73 74 |............inst|
f94b7760: 61 5f 30 32 30 33 e5 e5 e5  e5 e5 e5 e5 e5 e5 e5 |a_0203..........|
f94b7770: 70 52 18 7d 28 7f XX XX XX  XX XX XX XX XX XX XX |pR.}(...........|
f94b7780: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b7790: XX XX XX XX 54 XX XX XX XX  XX XX XX XX XX XX XX |....T...........|
f94b77a0: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b77b0: XX XX XX XX XX XX XX XX XX  XX XX XX 73 65 63 72 |............secr|
f94b77c0: 65 74 70 77 64 30 e5 e5 e5  e5 e5 e5 e5 e5 e5 e5 |etpwd0..........|
f94b77d0: 30 b4 18 7d 28 7f XX XX XX  XX XX XX XX XX XX XX |0..}(...........|
f94b77e0: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b77f0: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b7800: e5 e5 e5 e5 e5 e5 e5 e5 e5  e5 e5 e5 e5 e5 e5 e5 |................|
f94b7810: 68 74 74 70 73 3a 2f 2f 61  64 64 6f 6e 73 2e 63 |https://addons.c/|
f94b7820: 64 6e 2e 6d 6f 7a 69 6c 6c  61 2e 6e 65 74 2f 75 |dn.mozilla.net/u|
f94b7830: 73 65 72 2d 6d 65 64 69 61  2f 61 64 64 6f 6e 5f |ser-media/addon_|
f94b7840: 69 63 6f 6e 73 2f 33 35 34  2f 33 35 34 33 39 39 |icons/354/354399|
f94b7850: 2d 36 34 2e 70 6e 67 3f 6d  6f 64 69 66 69 65 64 |-64.png?modified|
f94b7860: 3d 31 34 35 32 32 34 34 38  31 35 XX XX XX XX XX |=1452244815.....|
```

Listing 4: Memory dump of Firefox 56 on Ubuntu 16.10 on a Intel Core i7-6700K disclosing saved passwords (cf.

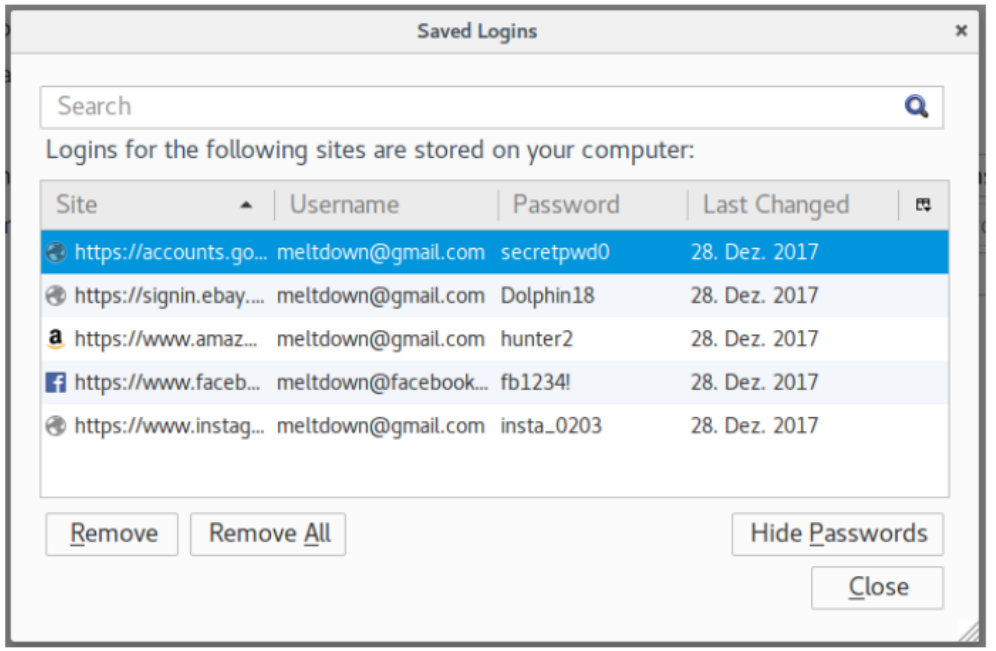
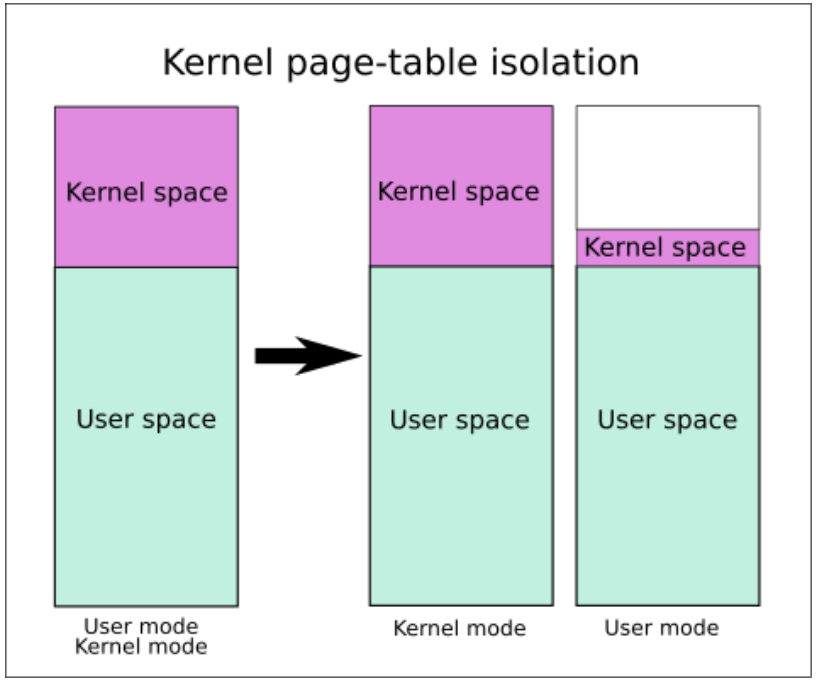


Figure 6: Firefox 56 password manager showing the stored passwords that are leaked using Meltdown in Listing 4.

Meltdown attack status

- Applied successfully on several Intel processors on various OSes (Linux-2.6.32 to 4.13.0), Windows 10, Docker, LXC, and OpenVZ.
- Proposed defense: **KPTI (Kernel Page Table Isolation).**
  - Being integrated into various OSes.
  - Long-term effectiveness is unclear.
  - Also, still seems controversial:

Linus Torvalds declares Intel fix for Meltdown/Spectre 'COMPLETE AND UTTER GARBAGE'



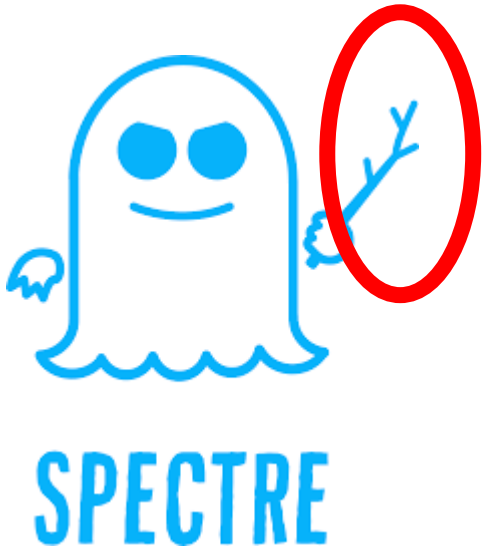
Summary of Meltdown

- Use OOO to execute an instruction that would normally raise an exception, and use it to read a secret from kernel memory.
- Transmit that secret to microarchitectural (cache) state.
- Read the cache state using a colluding process

- No “host” program required: self-contained with a parent+child attacking process.
- Only Intel processors affected.
- A fix has been proposed and is being deployed

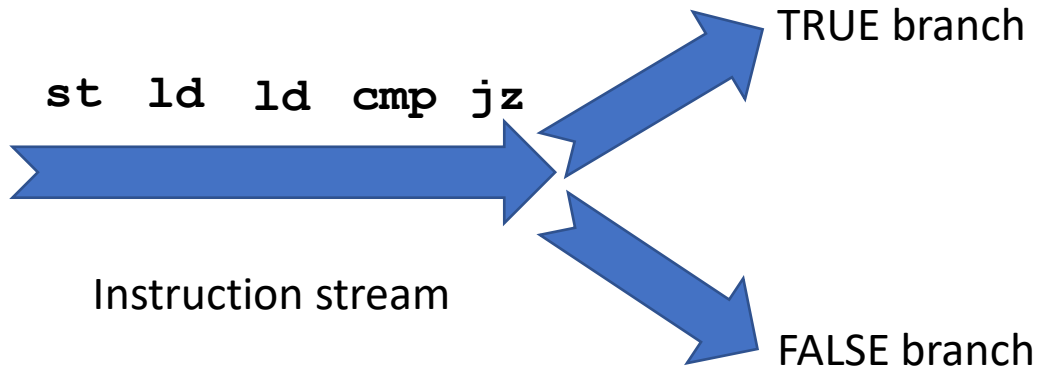
Spectre

- Affects a wide variety of processors (Intel, ARM, AMD).
- Uses another form of speculative execution: **branch prediction.**
- Slightly harder to deploy than Meltdown, in that a “host” program is required, which contains certain instruction sequences that can be misused.
- **No fixes are known to date.**



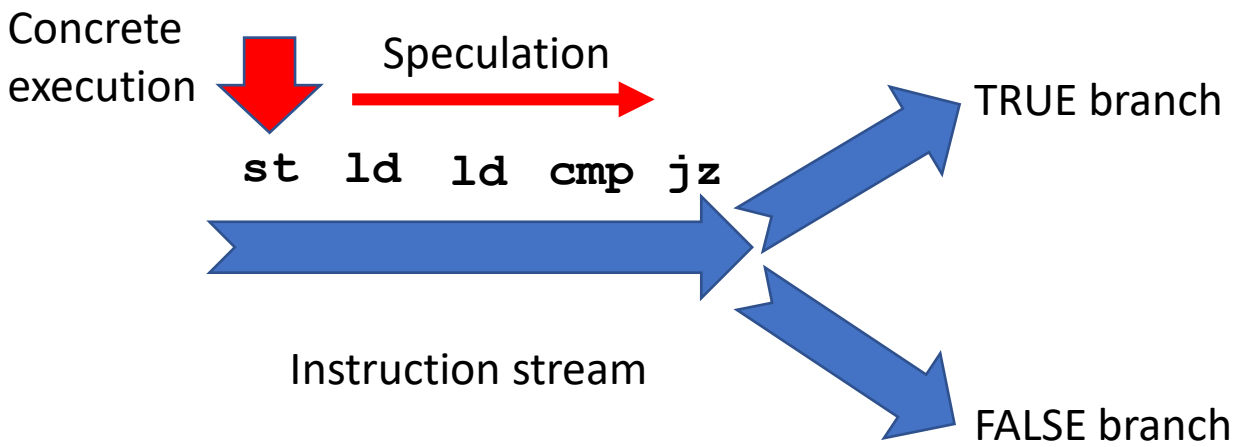
Building block: Branch prediction

- In OOO, what happens when the speculative execution engine reaches a branch?
- **Hardware branch predictor** predicts a likely outcome of the branch (based on past history), and continues to speculate along the (likely) taken branch.



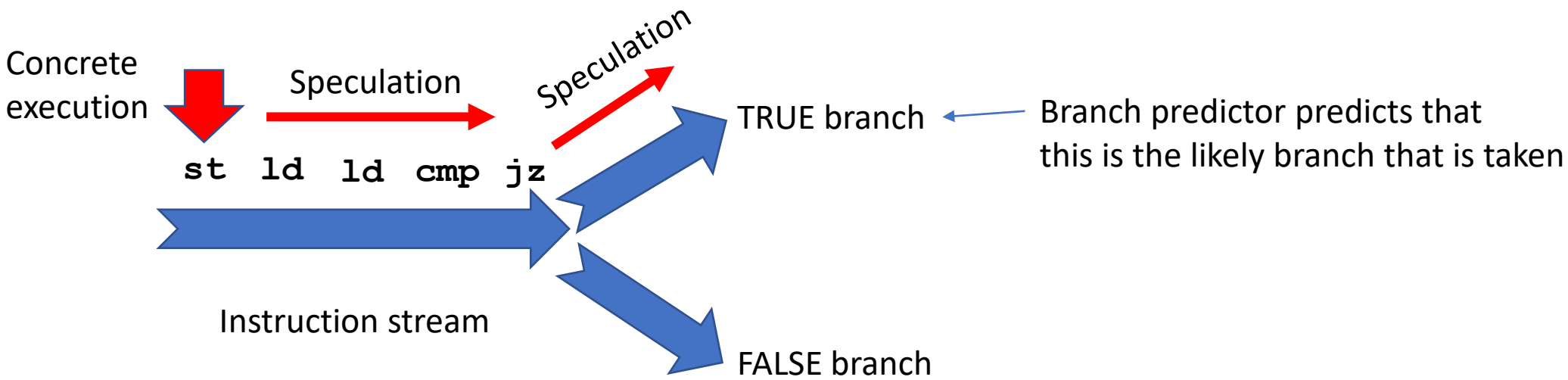
Building block: Branch prediction

- In OOO, what happens when the speculative execution engine reaches a branch?
- **Hardware branch predictor** predicts a likely outcome of the branch (based on past history), and continues to speculate along the (likely) taken branch.



Building block: Branch prediction

- In OOO, what happens when the speculative execution engine reaches a branch?
- **Hardware branch predictor** predicts a likely outcome of the branch (based on past history), and continues to speculate along the (likely) taken branch.



Basic setup of Spectre attack

- In a “host” program (the victim of the attack), find an instruction sequence with a branch.
- **Preparation**: Execute the program to train the branch predictor to go in one direction (say, TRUE)
- **Attack**: Feed it a malicious input that would cause the branch to go the other direction (i.e., FALSE), but rely on branch predictor to execute the TRUE branch. Use the speculatively executed TRUE branch to extract data from the victim program.

Consider a host program with this snippet

```
if (x < array1_size)
   y = array2[array1[x] * 256];
```

`array1` is a unsigned byte array of size `array1_size`

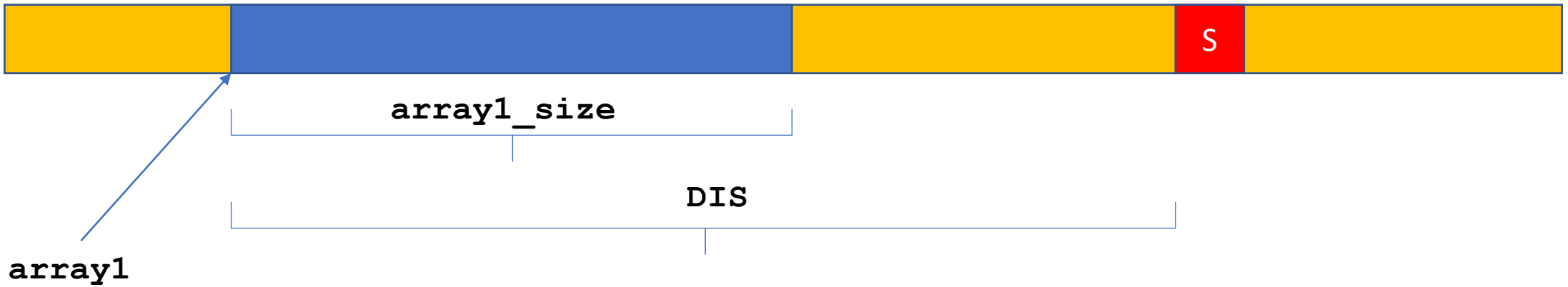
`array2` is of size 64KB (256*256)

Suppose the value of `x` is derived from user input to the program (and can therefore be controlled by attacker).

In this program, there is some secret data **S** that you wish to access

Consider a host program with this snippet

```
if (x < array1_size)
   y = array2[array1[x] * 256];
```



Observe: `array1[DIS]` obtains `S`

Attack preparation

```
if (x < array1_size)
   y = array2[array1[x] * 256];
```

1. Execute the program long enough with a number of values of x, so that the branch predictor is trained to take the true branch.
2. Arrange for cache to **not contain** `array2` and `array1_size`.
3. Arrange for cache to **contain** secret value S. How? E.g., S could be a cryptographic key you want to learn. Arrange for a cryptographic computation to happen that uses S.

Actual Spectre attack

```
if (x < array1_size)
   y = array2[array1[x] * 256];
```

Now execute the program with `x = DIS`

1. x < array1_size will lead to a cache miss. Leads to a delay in fetching array1_size. Processor speculates on branch.
2. Speculative code reads array1[DIS]. A hit in the cache (the value S)
3. Code then proceeds to read array2[S*256]. A miss in the cache.

However, array1_size may have arrived by then. Processor realizes mistake in speculation. But too late…the speculative read array2[S*256] already affects cache state

Actual Spectre attack

```
if (x < array1_size)
   y = array2[array1[x] * 256];
```

However, array1_size may have arrived by then. Processor realizes mistake in speculation. But too late…the speculative read array2[S*256] already affects cache state.

If array2 is accessible to the attacker, just probe all its elements and use cache-timing to figure out the value of S. (Many options possible here to “transmit” the microarchitectural state to the attacker).

Notes about Spectre

- Not restricted to host programs that have such a convenient code sequence built in.
  - Can search for “gadgets” (short instruction sequences) that can be “weaved” together to achieve desired effect (“Return-oriented Programming”, for those students who took my E0-256 course)
- Not restricted to conditional branches. Attack also adapted to work with indirect branches.

Spectre attack application: Breaking out of sandboxes

- JavaScript code that executes on your browser runs in a sandbox.
  - Generally does not have access to browser state, except those explicitly exported to it.
  - For example, this ensures that the JavaScript code that you get from Google when you read Gmail is unable to access your browsing history.
- Spectre can be used to create JavaScript code that “breaks” out of this sandbox to access arbitrary browser memory.
  - Same basic idea as I described.