Virtualization and Security

Daniel Sanchez
Computer Science & Artificial Intelligence Lab
M.I.T.
Evolution in Number of Users

IBM 1620
1959

Single User

Runtime loaded with program
Evolution in Number of Users

IBM 1620
1959
Single User
Runtime
loaded with
program

IBM 360
1960s
Multiple Users
OS for
sharing
resources
Evolution in Number of Users

- **IBM 1620**
  - Year: 1959
  - Use: Single User
  - Runtime: Loaded with program

- **IBM 360**
  - Year: 1960s
  - Use: Multiple Users
  - OS: For sharing resources

- **IBM PC**
  - Year: 1980s
  - Use: Single User
  - OS: For sharing resources
Evolution in Number of Users

IBM 1620 1959
Single User
Runtime loaded with program

IBM 360 1960s
Multiple Users
OS for sharing resources

IBM PC 1980s
Single User
OS for sharing resources

Cloud Servers 1990s
Multiple Users
Multiple OSs
Single-Program Machine

- Hardware executes a single program
- This program has direct and complete access to all hardware resources in the machine
Single-Program Machine

- Hardware executes a single program
- This program has direct and complete access to all hardware resources in the machine
• Hardware executes a single program
• This program has direct and complete access to all hardware resources in the machine
• The instruction set architecture (ISA) is the interface between software and hardware
Operating Systems

- Operating System (OS) goals:
  - Protection and privacy: Processes cannot access each other’s data
• Operating System (OS) goals:
  – Protection and privacy: Processes cannot access each other’s data
  – Abstraction: OS hides details of underlying hardware
    • e.g., processes open and access files instead of issuing raw commands to the disk
Operating Systems

- Operating System (OS) goals:
  - Protection and privacy: Processes cannot access each other’s data
  - Abstraction: OS hides details of underlying hardware
    - e.g., processes open and access files instead of issuing raw commands to the disk
  - Resource management: OS controls how processes share hardware (CPU, memory, disk, etc.)
Operating System Mechanisms

- The OS kernel provides a private address space to each process
  - Each process is allocated space in physical memory by the OS
  - A process is not allowed to access the memory of other processes
Operating System Mechanisms

- The OS kernel provides a **private address space** to each process
  - Each process is allocated space in physical memory by the OS
  - A process is not allowed to access the memory of other processes
The OS kernel provides a private address space to each process
- Each process is allocated space in physical memory by the OS
- A process is not allowed to access the memory of other processes

The OS kernel schedules processes into cores
- Each process is given a fraction of CPU time
- A process cannot use more CPU time than allowed
Operating System Mechanisms

- The OS kernel provides a **private address space** to each process
  - Each process is allocated space in physical memory by the OS
  - A process is not allowed to access the memory of other processes

- The OS kernel **schedules processes** into cores
  - Each process is given a fraction of CPU time
  - A process cannot use more CPU time than allowed
Operating System Mechanisms

- The OS kernel provides a private address space to each process
  - Each process is allocated space in physical memory by the OS
  - A process is not allowed to access the memory of other processes

- The OS kernel schedules processes into cores
  - Each process is given a fraction of CPU time
  - A process cannot use more CPU time than allowed

- The OS kernel lets processes invoke system services (e.g., access files or network sockets) via system calls
Virtual Machines

- The OS gives a Virtual Machine (VM) to each process
  - Each process believes it runs on its own machine...
  - ...but this machine does not exist in physical hardware
Virtual Machines

- The OS gives a **Virtual Machine (VM)** to each process
  - Each process believes it runs on its own machine...
  - ...but this machine does not exist in physical hardware
Virtual Machines

• The OS gives a **Virtual Machine (VM)** to each process
  - Each process believes it runs on its own machine...
  - ...but this machine does not exist in physical hardware
Virtual Machines

- The OS gives a Virtual Machine (VM) to each process
  - Each process believes it runs on its own machine...
  - ...but this machine does not exist in physical hardware
Virtual Machines

- The OS gives a **Virtual Machine (VM)** to each process
  - Each process believes it runs on its own machine...
  - ...but this machine does not exist in physical hardware
Virtual Machines

- A Virtual Machine (VM) is an emulation of a computer system
  - Very general concept, used beyond operating systems

```
<table>
<thead>
<tr>
<th>Process1</th>
<th>Virtual CPUs</th>
<th>Virtual Memory</th>
<th>Events</th>
<th>Files</th>
<th>Sockets</th>
<th>Syscalls</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

OS Kernel (specially privileged process)

Physical Hardware
- Processor
- Memory
- Disk
- Network card
- Display
- Keyboard
...
Virtual Machines Are Everywhere

• Example: Consider a Python program running on a Linux Virtual Machine
Virtual Machines Are Everywhere

- Example: Consider a Python program running on a Linux Virtual Machine

Python program
Virtual Machines Are Everywhere

• Example: Consider a Python program running on a Linux Virtual Machine

Python program

Python Language
Virtual Machines Are Everywhere

- Example: Consider a Python program running on a Linux Virtual Machine

Python program

Python interpreter (CPython)

Python Language
Virtual Machines Are Everywhere

• Example: Consider a Python program running on a Linux Virtual Machine

- Python program
- Python interpreter (CPython)
- Python Language
  Implements a Python VM
Virtual Machines Are Everywhere

- Example: Consider a Python program running on a Linux Virtual Machine
Virtual Machines Are Everywhere

- Example: Consider a Python program running on a Linux Virtual Machine

- Python program
- Python interpreter (CPython)
- Linux OS kernel
- Python Language
  Implements a Python VM
- Linux ABI
Virtual Machines Are Everywhere

• Example: Consider a Python program running on a Linux Virtual Machine

- Python program
- Python interpreter (CPython)
- Linux OS kernel

Python Language
  Implements a Python VM

Linux ABI
  Implements a Linux-x86 VM
Virtual Machines Are Everywhere

- Example: Consider a Python program running on a Linux Virtual Machine

  - Python program
  - Python interpreter (CPython)
    - Python Language
      - Implements a Python VM
    - Linux ABI
    - Linux OS kernel
      - Implements a Linux-x86 VM
      - x86 ISA
Virtual Machines Are Everywhere

- Example: Consider a Python program running on a Linux Virtual Machine

Python program

Python interpreter (CPython)

Linux OS kernel

VirtualBox

Python Language

Implements a Python VM

Linux ABI

Implements a Linux-x86 VM

x86 ISA
Virtual Machines Are Everywhere

- Example: Consider a Python program running on a Linux Virtual Machine

<table>
<thead>
<tr>
<th>Python program</th>
<th>Python Language</th>
<th>Implements a Python VM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python interpreter (CPython)</td>
<td>Linux ABI</td>
<td>Implements a Linux-x86 VM</td>
</tr>
<tr>
<td>Linux OS kernel</td>
<td>x86 ISA</td>
<td>Implements an x86 system VM</td>
</tr>
<tr>
<td>VirtualBox</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Virtual Machines Are Everywhere

• Example: Consider a Python program running on a Linux Virtual Machine

  - Python program
  - Python interpreter (CPython)
  - Linux OS kernel
  - VirtualBox

  Python Language
  - Implements a Python VM
  - Linux ABI
  - Implements a Linux-x86 VM
  - x86 ISA
  - Implements an x86 system VM
  - Win/Linux/MacOS/... ABI
Virtual Machines Are Everywhere

- Example: Consider a Python program running on a Linux Virtual Machine

- Python program
- Python interpreter (CPython)
- Linux OS kernel
- VirtualBox
- OS kernel (Win/Linux/MacOS/...)

- Python Language
  - Implements a Python VM
- Linux ABI
  - Implements a Linux-x86 VM
- x86 ISA
  - Implements an x86 system VM
- Win/Linux/MacOS/... ABI
Virtual Machines Are Everywhere

- Example: Consider a Python program running on a Linux Virtual Machine

- Python program
  - Python Language
    - Implements a Python VM
  - Python interpreter (CPython)
    - Linux ABI
      - Implements a Linux-x86 VM
  - Linux OS kernel
    - x86 ISA
      - Implements an x86 system VM
  - VirtualBox
    - Win/Linux/MacOS/... ABI
      - Implements an OS-x86 VM
  - OS kernel (Win/Linux/MacOS/...)

Virtual Machines Are Everywhere

- Example: Consider a Python program running on a Linux Virtual Machine

  - Python program
  - Python interpreter (CPython)
  - Linux OS kernel
  - VirtualBox
  - OS kernel (Win/Linux/MacOS/...)

  - Python Language
    - Implements a Python VM
  - Linux ABI
    - Implements a Linux-x86 VM
  - x86 ISA
    - Implements an x86 system VM
  - Win/Linux/MacOS/... ABI
    - Implements an OS-x86 VM
Virtual Machines Are Everywhere

- Example: Consider a Python program running on a Linux Virtual Machine

  - Python program
  - Python interpreter (CPython)
  - Linux OS kernel
  - VirtualBox
  - OS kernel (Win/Linux/MacOS/...)
  - Hardware (e.g., your laptop)

  - Python Language
  - Implements a Python VM
  - Linux ABI
  - Implements a Linux-x86 VM
  - x86 ISA
  - Implements an x86 system VM
  - Win/Linux/MacOS/... ABI
  - Implements an OS-x86 VM
  - x86 ISA
Virtual Machines Are Everywhere

- Example: Consider a Python program running on a Linux Virtual Machine
Implementing Virtual Machines

- Virtual machines can be implemented entirely in software, but at a performance cost
  - e.g., Python programs are 10-100x slower than native Linux programs due to Python interpreter overheads
Implementing Virtual Machines

• Virtual machines can be implemented entirely in software, but at a performance cost
  – e.g., Python programs are 10-100x slower than native Linux programs due to Python interpreter overheads

• We want to support virtual machines with minimal overheads → need hardware support!
ISA Extensions to Support OS
ISA Extensions to Support OS

- Two modes of execution: user and supervisor
ISA Extensions to Support OS

• Two modes of execution: user and supervisor
  – OS kernel runs in supervisor mode
  – All other processes run in user mode
ISA Extensions to Support OS

• Two modes of execution: user and supervisor
  – OS kernel runs in supervisor mode
  – All other processes run in user mode

• Privileged instructions and registers that are only available in supervisor mode
ISA Extensions to Support OS

- Two modes of execution: user and supervisor
  - OS kernel runs in supervisor mode
  - All other processes run in user mode
- Privileged instructions and registers that are only available in supervisor mode
- Traps (exceptions) to safely transition from user to supervisor mode
ISA Extensions to Support OS

• Two modes of execution: user and supervisor
  – OS kernel runs in supervisor mode
  – All other processes run in user mode

• Privileged instructions and registers that are only available in supervisor mode

• Traps (exceptions) to safely transition from user to supervisor mode

• Virtual memory to provide private address spaces and abstract the storage resources of the machine
Process Mode Switching

Trap, e.g., i/o read() or exception

Switch to kernel mode;
Pass arguments;
Save app state

user mode

kernel mode

Trap handler

Check arguments
Find trap handler addr

Kernel routine

Restore app state,
Return to user
Protection – Single OS

- OS Kernel
- Trap
- User Process
- User Process
Supporting Multiple OSs

<table>
<thead>
<tr>
<th>ABI</th>
<th>process_1</th>
<th>\cdots</th>
<th>process_N</th>
<th>ABI</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISA</td>
<td>OS Kernel_1</td>
<td>\cdots</td>
<td>OS Kernel_K</td>
<td>ISA</td>
</tr>
<tr>
<td></td>
<td>Virtual Machine Monitor (VMM)</td>
<td></td>
<td>Hardware</td>
<td></td>
</tr>
</tbody>
</table>
A VMM (aka Hypervisor) provides a system virtual machine to each OS.
Supporting Multiple OSs

- A VMM (aka Hypervisor) provides a system virtual machine to each OS
- VMM can run directly on hardware (as above) or on another OS
  - Precisely, VMM can be implemented against an ISA (as above) or a process-level ABI. Who knows what lays below the interface...
Motivation for Multiple OSs

Some motivations for using multiple operating systems on a single computer:

- Allows use of capabilities of multiple distinct operating systems
Motivation for Multiple OSs

Some motivations for using multiple operating systems on a single computer:

- Allows use of capabilities of multiple distinct operating systems
- Allows different users to share a system while using completely independent software stacks
Motivation for Multiple OSs

Some motivations for using multiple operating systems on a single computer:

- Allows use of capabilities of multiple distinct operating systems
- Allows different users to share a system while using completely independent software stacks
- Allows for load balancing and migration across multiple machines
Motivation for Multiple OSs

Some motivations for using multiple operating systems on a single computer:

- Allows use of capabilities of multiple distinct operating systems
- Allows different users to share a system while using completely independent software stacks
- Allows for load balancing and migration across multiple machines
- Allows operating system development without making entire machine unstable or unusable
Virtualization Nomenclature

From (Machine we are attempting to execute)

- Guest
- Client
- Foreign ISA

To (Machine that is doing the real execution)

- Host
- Target
- Native ISA
Virtual Machine Requirements
[Popek and Goldberg, 1974]

- Equivalence/Fidelity: A program running on the VMM should exhibit a behavior essentially identical to that demonstrated when running on an equivalent machine directly.

- Resource control/Safety: The VMM must be in complete control of the virtualized resources.

- Efficiency/Performance: A statistically dominant fraction of machine instructions must be executed without VMM intervention.
Virtual Machine Requirements
[Popek and Goldberg, 1974]

Classification of instructions into 3 groups:

- Privileged instructions: Instructions that trap if the processor is in user mode and do not trap if it is in a more privileged mode.

- Control-sensitive instructions: Instructions that attempt to change the configuration of resources in the system.

- Behavior-sensitive instructions: Those whose behavior depends on the configuration of resources, e.g., mode

Building an effective VMM for an architecture is possible if the set of sensitive instructions is a subset of the set of privileged instructions.
Sensitive instruction handling

Sensitive instruction

Non-VMM mode

VMM mode

Switch to VMM mode;
Pass arguments;
Save app state

VMM handler

Find handler addr

VMM routine

Restore app state,
Return to guest
Protection – Multiple OS

Diagram showing the relationship between a Virtual Machine Monitor (VMM), OS kernels, and user processes.
Virtual Memory Operations

TLB can be designed to translate guest virtual addresses (gVA) to a host physical address (hPA), but...
Virtual Memory Operations

TLB can be designed to translate guest virtual addresses (gVA) to a host physical address (hPA), but...

- TLB misses are a ‘sensitive’ operation
Virtual Memory Operations

TLB can be designed to translate guest virtual addresses (gVA) to a host physical address (hPA), but...

- TLB misses are a ‘sensitive’ operation
- TLB misses happen very very frequently
Virtual Memory Operations

TLB can be designed to translate guest virtual addresses (gVA) to a host physical address (hPA), but...

- TLB misses are a ‘sensitive’ operation
- TLB misses happen very very very frequently

- So how expensive are TLB fills?
Nested Page Tables
Nested Page Tables

Guest VA

Guest Page Table Base

Index 1

Index 2

Offset

PTP

L1 Table

PTE

L2 Table

PPN

Offset

Guest PA
Nested Page Tables

Guest VA

Guest Page Table Base

L1 Table

PTP

L2 Table

PTE

PPN Offset

Host PA == Host VA

Host Page Table Base

L1 Table

PTP

L2 Table

PTE

PPN Offset

May 13, 2021
Shadow Page Tables

Guest VA

Index 1

Guest Page Table Base

Index 2

Offset

L1 Table

PTP

L2 Table

PTE

PPN

Offset

Guest PA

May 13, 2021

MIT 6.823 Spring 2021
Shadow Page Tables
# Nested vs Shadow Paging

<table>
<thead>
<tr>
<th></th>
<th>Native</th>
<th>Nested Paging</th>
<th>Shadow Paging</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TLB Hit</strong></td>
<td>VA-&gt;PA</td>
<td>gVA-&gt;hPA</td>
<td>gVA-&gt;hPA</td>
</tr>
<tr>
<td><strong>TLB Miss (max)</strong></td>
<td>4</td>
<td>24</td>
<td>4</td>
</tr>
<tr>
<td><strong>PTE Updates</strong></td>
<td>Fast</td>
<td>Fast</td>
<td>Uses VMM</td>
</tr>
</tbody>
</table>

On x86-64
Security and Side Channels

- ISA and ABI are **timing-independent** interfaces
  - Specify *what* should happen, not *when*
Security and Side Channels

- ISA and ABI are **timing-independent** interfaces
  - Specify *what* should happen, not *when*

- Hardware isolation mechanisms like virtual memory guarantee that architectural state will not be directly exposed to other processes...
Security and Side Channels

- ISA and ABI are timing-independent interfaces
  - Specify what should happen, not when

- Hardware isolation mechanisms like virtual memory guarantee that architectural state will not be directly exposed to other processes...

- ...but timing and other implementation details (e.g., microarchitectural state, power, etc.) may be used as side channels to leak information!
Cache-Based Side Channels

- Attacker can infer shared cache behavior of victim
  - e.g., prime+probe attack: Attacker fills cache with own data, then times accesses to data to see which hit and miss, inferring which lines the victim is using
  - Leaks address-dependent information, e.g., RSA [Percival 2005] and AES keys [Osvik et al. 2005]
Cache-Based Side Channels

- Attacker can infer shared cache behavior of victim
  - e.g., prime+probe attack: Attacker fills cache with own data, then times accesses to data to see which hit and miss, inferring which lines the victim is using
  - Leaks address-dependent information, e.g., RSA [Percival 2005] and AES keys [Osvik et al. 2005]

- Microarch side channels among threads running on same SMT core?
Cache-Based Side Channels

- Attacker can infer shared cache behavior of victim
  - e.g., prime+probe attack: Attacker fills cache with own data, then times accesses to data to see which hit and miss, inferring which lines the victim is using
  - Leaks address-dependent information, e.g., RSA [Percival 2005] and AES keys [Osvik et al. 2005]
- Microarch side channels among threads running on same SMT core?
Cache-Based Side Channels

- Attacker can infer shared cache behavior of victim
  - e.g., prime+probe attack: Attacker fills cache with own data, then times accesses to data to see which hit and miss, inferring which lines the victim is using
  - Leaks address-dependent information, e.g., RSA [Percival 2005] and AES keys [Osvik et al. 2005]
- **Microarch side channels among threads running on same SMT core?**

L1/L2/L3 caches
Branch & other predictors
Cache-Based Side Channels

- Attacker can infer shared cache behavior of victim
  - e.g., prime+probe attack: Attacker fills cache with own data, then times accesses to data to see which hit and miss, inferring which lines the victim is using
  - Leaks address-dependent information, e.g., RSA [Percival 2005] and AES keys [Osvik et al. 2005]
- Microarch side channels among threads running on same SMT core?
  - L1/L2/L3 caches
  - Branch & other predictors
  - ROB/Issue/FU contention
Example: Side Channel in RSA

- Assume square-and-multiply based exponentiation

**Input**: base $b$, modulo $m$, exponent $e = (e_{n-1} \ldots e_0)_2$

**Output**: $b^e \mod m$

$r = 1$

for $i = n-1 \text{ down to } 0$ do
  $r = \sqrt{r}$
  $r = \text{mod}(r, m)$
  if $e_i == 1$ then
    $r = \text{mul}(r, b)$
    $r = \text{mod}(r, m)$
  end
end

return $r$
Example: Side Channel in RSA

- Assume square-and-multiply based exponentiation

**Input**: base $b$, modulo $m$, exponent $e = (e_{n-1} \ldots e_0)_2$

**Output**: $b^e \mod m$

$r = 1$

for $i = n-1$ down to 0 do

  $r = \sqrt{r}$

  $r = \text{mod}(r,m)$

  if $e_i == 1$ then

    $r = \text{mul}(r,b)$

    $r = \text{mod}(r,m)$

  end

end

return $r$
Example: Side Channel in RSA

- Assume square-and-multiply based exponentiation

**Input**: base $b$, modulo $m$, exponent $e = (e_{n-1} \ldots e_0)_2$

**Output**: $b^e \mod m$

$r = 1$

for $i = n-1$ down to 0 do

\[
\begin{align*}
    r &= \sqrt{r} \\
    r &= \text{mod}(r, m) \\
    \text{if } e_i \equiv 1 \text{ then} \\
    r &= \text{mul}(r, b) \\
    r &= \text{mod}(r, m)
\end{align*}
\]

end

return $r$

Secret-dependent memory accesses $\rightarrow$ transmitter
Exploiting Speculative Execution in Side-Channel Attacks

- OoO cores run instructions speculatively and out of order
- Problem: Speculative instructions can change microarchitectural state → can leak data via side channel
Exploiting Speculative Execution in Side-Channel Attacks

- OoO cores run instructions speculatively and out of order
- Problem: Speculative instructions can change microarchitectural state → can leak data via side channel
- Example: In x86, process page table can have kernel pages, but kernel pages only accessible in kernel mode
  - Avoids switching page tables on context switches
Exploiting Speculative Execution in Side-Channel Attacks

- OoO cores run instructions speculatively and out of order
- Problem: Speculative instructions can change microarchitectural state → can leak data via side channel
- Example: In x86, process page table can have kernel pages, but kernel pages only accessible in kernel mode
  - Avoids switching page tables on context switches
  - **What does the following code do when run in user mode?**

```c
val = *kernel_address;
```
Exploiting Speculative Execution in Side-Channel Attacks

- OoO cores run instructions speculatively and out of order
- Problem: Speculative instructions can change microarchitectural state → can leak data via side channel
- Example: In x86, process page table can have kernel pages, but kernel pages only accessible in kernel mode
  - Avoids switching page tables on context switches
  - What does the following code do when run in user mode?

```c
val = *kernel_address;
```

Causes a protection fault
Exploiting Speculative Execution in Side-Channel Attacks

- OoO cores run instructions speculatively and out of order
- Problem: Speculative instructions can change microarchitectural state → can leak data via side channel
- Example: In x86, process page table can have kernel pages, but kernel pages only accessible in kernel mode
  - Avoids switching page tables on context switches
  - *What does the following code do when run in user mode?*

```c
val = *kernel_address;
```

Causes a protection fault

In Intel processors, protection check happens late → Kernel data speculatively loaded into val register!
Meltdown
[Lipp et al. 2018]

1. Setup: Attacker allocates 256-line probe_array, flushes all its cache lines
Meltdown
[Lipp et al. 2018]

1. Setup: Attacker allocates 256-line probe_array, flushes all its cache lines

2. Transmit: Attacker executes

```c
uint8_t byte = *kernel_address;
probe_array[byte] = 1;
```
Meltdown
[Lipp et al. 2018]

1. Setup: Attacker allocates 256-line probe_array, flushes all its cache lines

2. Transmit: Attacker executes

```c
uint8_t byte = *kernel_address;
probe_array[byte] = 1;
```

3. Receive: After handling protection fault, attacker times accesses to all cache lines of probe_array, finds which one hits → recovers byte
Meltdown
[Lipp et al. 2018]

1. Setup: Attacker allocates 256-line `probe_array`, flushes all its cache lines

2. Transmit: Attacker executes
   ```c
   uint8_t byte = *kernel_address;
   probe_array[byte] = 1;
   ```

3. Receive: After handling protection fault, attacker times accesses to all cache lines of `probe_array`, finds which one hits → recovers `byte`

   - Result: Attacker can read arbitrary kernel data!
     - For higher performance, use transactional memory (protection fault aborts transaction on exception instead of invoking kernel)
     - Mitigation: Do not map kernel data in user page tables
General Attack Schema
[Belay, Devadas, Emer]
• Types of transmitter:
  1. Pre-existing (the victim itself leaks secret, e.g., RSA/AES keys)
  2. Programmed by attacker (e.g., Meltdown)
  3. Synthesized from existing victim code by attacker (e.g., Spectre)
Spectre variant 1 — Exploiting Conditional Branches [Kocher et al. 2018]

• Consider the following kernel code, e.g., in a system call
  
  if (x < array1_size)
      y = array2[array1[x] * 4096];
Spectre variant 1 — Exploiting Conditional Branches [Kocher et al. 2018]

- Consider the following kernel code, e.g., in a system call

```c
if (x < array1_size)
    y = array2[array1[x] * 4096];
```

1. Setup: Attacker invokes this kernel code with small values of `x` to train the branch predictor to taken
• Consider the following kernel code, e.g., in a system call

```c
if (x < array1_size)
    y = array2[array1[x] * 4096];
```

1. Setup: Attacker invokes this kernel code with small values of \(x\) to train the branch predictor to taken

2. Transmit: Attacker invokes this code with an out-of-bounds \(x\), so that \&array1[x] maps to some desired kernel address. Core mispredicts branch, fetches array2[array1[x] * 4096]’s line into the cache.
Spectre variant 1 — Exploiting Conditional Branches [Kocher et al. 2018]

- Consider the following kernel code, e.g., in a system call

  ```c
  if (x < array1_size)
      y = array2[array1[x] * 4096];
  ```

1. Setup: Attacker invokes this kernel code with small values of x to train the branch predictor to taken

2. Transmit: Attacker invokes this code with an out-of-bounds x, so that &array1[x] maps to some desired kernel address. Core mispredicts branch, fetches array2[array1[x] * 4096]’s line into the cache.

3. Receive: Attacker probes cache to infer which line of array2 was fetched, learns data at kernel address
   - array2 may or may not be accessible to attacker (can use prime+probe)
Spectre variant 2—Branch Target Injection [Kocher et al. 2018]

- Assume the BTB stores partial tags but full target PCs. How can this be exploited?
Spectre variant 2—Branch Target Injection [Kocher et al. 2018]

- Assume the BTB stores partial tags but full target PCs. How can this be exploited?
  1. Setup: Attacker chooses any jump in kernel code, mistrains BTB so that it predicts a target PC under the control of the attacker that leaks information, e.g.,
Spectre variant 2—Branch Target Injection [Kocher et al. 2018]

• Assume the BTB stores partial tags but full target PCs. How can this be exploited?
  1. Setup: Attacker chooses any jump in kernel code, mistrains BTB so that it predicts a target PC under the control of the attacker that leaks information, e.g.,

    ```c
    uint8_t byte = *kernel_address;
    probe_array[byte] = 1;
    ```
Spectre variant 2—Branch Target Injection [Kocher et al. 2018]

- Assume the BTB stores partial tags but full target PCs. How can this be exploited?
  1. Setup: Attacker chooses any jump in kernel code, mistrains BTB so that it predicts a target PC under the control of the attacker that leaks information, e.g.,

     ```c
     uint8_t byte = *kernel_address;
     probe_array[byte] = 1;
     ```
  2. Transmit & receive: Like in Spectre v1
Spectre variant 2—Branch Target Injection [Kocher et al. 2018]

- Assume the BTB stores partial tags but full target PCs. How can this be exploited?
  1. Setup: Attacker chooses any jump in kernel code, mistrains BTB so that it predicts a target PC under the control of the attacker that leaks information, e.g.,

     ```
     uint8_t byte = *kernel_address;
     probe_array[byte] = 1;
     ```

  2. Transmit & receive: Like in Spectre v1

- Most BTBs store partial tags and targets...
  - Hard to get BTB to jump from a kernel address to a far-away user address
Spectre variant 2—Branch Target Injection [Kocher et al. 2018]

• Assume the BTB stores partial tags but full target PCs. How can this be exploited?
  1. Setup: Attacker chooses any jump in kernel code, mistrains BTB so that it predicts a target PC under the control of the attacker that leaks information, e.g.,

```c
uint8_t byte = *kernel_address;
probe_array[byte] = 1;
```

  2. Transmit & receive: Like in Spectre v1

• Most BTBs store partial tags and targets...
  – Hard to get BTB to jump from a kernel address to a far-away user address

• But most cores add an indirect branch predictor that stores full targets (e.g., to predict virtual function calls)
  – Spectre v2 exploits this predictor instead
Spectre variants and mitigations

• Spectre relies on speculative execution, not late exception checks → Much harder to fix than Meltdown
Spectre variants and mitigations

- Spectre relies on speculative execution, not late exception checks → Much harder to fix than Meltdown
- Several other Spectre variants reported
  - Leveraging the speculative store buffer, return address stack, leaking privileged registers, etc.
Spectre variants and mitigations

• Spectre relies on speculative execution, not late exception checks → Much harder to fix than Meltdown

• Several other Spectre variants reported
  – Leveraging the speculative store buffer, return address stack, leaking privileged registers, etc.

• Can attack any type of VM, including OSs, VMMs, JavaScript engines in browsers, and the OS network stack (NetSpectre)
Spectre variants and mitigations

- Spectre relies on speculative execution, not late exception checks → Much harder to fix than Meltdown
- Several other Spectre variants reported
  - Leveraging the speculative store buffer, return address stack, leaking privileged registers, etc.
- Can attack any type of VM, including OSs, VMMs, JavaScript engines in browsers, and the OS network stack (NetSpectre)
- Short-term mitigations:
  - Microcode updates (disable sharing of speculative state when possible)
  - OS and compiler patches to selectively avoid speculation
Spectre variants and mitigations

• Spectre relies on speculative execution, not late exception checks → Much harder to fix than Meltdown

• Several other Spectre variants reported
  – Leveraging the speculative store buffer, return address stack, leaking privileged registers, etc.

• Can attack any type of VM, including OSs, VMMs, JavaScript engines in browsers, and the OS network stack (NetSpectre)

• Short-term mitigations:
  – Microcode updates (disable sharing of speculative state when possible)
  – OS and compiler patches to selectively avoid speculation

• Long-term mitigations:
  – Disabling speculation?
  – Closing side channels?
Thank you!