Security

*Joel Emer*
Computer Science & Artificial Intelligence Lab
M.I.T.

*With some slide credits to: Chris Fletcher and Mengia Yan*
Security and Information Leakage

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

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

• ISA and ABI are **timing-independent** interfaces, and
  – Specify *what* should happen, not *when*
Security and Information Leakage

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

- ISA and ABI are \textit{timing-independent} interfaces, and
  - Specify \textit{what} should happen, not \textit{when}

- ISA and ABS only specify \textit{architectural} updates
  - \textit{Micro-architectural changes are left unspecified}
Security and Information Leakage

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

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

- ISA and ABS only specify **architectural** updates
  - *Micro-architectural changes are left unspecified*

- ...so implementation details and timing behaviors (e.g., microarchitectural state, power, etc.) may be used as **channels** to leak information!
Simple Communication Model

Message

Transmitter

Channel

Receiver

Message
Simple Communication Model

- Transmitter accepts message
Simple Communication Model

- Transmitter accepts message
- Transmitter modulates channel
Simple Communication Model

- Transmitter accepts message
- Transmitter modulates channel
- Receiver detects modulation on channel
Simple Communication Model

- Transmitter accepts message
- Transmitter modulates channel
- Receiver detects modulation on channel
- Receiver decodes modulation as message.
Communication Model of Attacks
[Belay, Devadas, Emer]
Communication Model of Attacks
[Belay, Devadas, Emer]

- Domains – Distinct architectural domains in which architectural state is not shared.
Communication Model of Attacks
[Belay, Devadas, Emer]

- **Domains** – Distinct architectural domains in which architectural state is not shared.
- **Channel** – some “state” that can be changed, i.e., modulated, by the “transmitter” and whose modulation can be detected by the “receiver”.

Diagram:
- Domain of victim
  - Access
  - Transmitter
  - Secret
- Channel
- Receiver
  - Decode
  - Secret
- Domain of attacker
Communication Model of Attacks
[Belay, Devadas, Emer]

- **Domains** – Distinct architectural domains in which architectural state is not shared.
- **Channel** – some “state” that can be changed, i.e., modulated, by the “transmitter” and whose modulation can be detected by the “receiver”.
- **Secret** – the “message” that is transmitted on the channel and detected by the receiver.
Communication Model of Attacks
[Belay, Devadas, Emer]

- Domains – Distinct architectural domains in which architectural state is not shared.
- Channel – some “state” that can be changed, i.e., modulated, by the “transmitter” and whose modulation can be detected by the “receiver”.
- Secret – the “message” that is transmitted on the channel and detected by the receiver

Because channel is not a “direct” communication channel it is often referred to as a “side channel”
Communication Model of Attacks
[Belay, Devadas, Emer]
Communication Model of Attacks
[Belay, Devadas, Emer]

1. Transmitter “accesses” secret
Communication Model of Attacks
[Belay, Devadas, Emer]

1. Transmitter “accesses” secret
2. Transmitter modulates channel with a message based on secret
Communication Model of Attacks
[Belay, Devadas, Emer]

1. Transmitter “accesses” secret
2. Transmitter modulates channel with a message based on secret
3. Receiver detects modulation on channel
Communication Model of Attacks
[Belay, Devadas, Emer]

1. Transmitter “accesses” secret
2. Transmitter modulates channel with a message based on secret
3. Receiver detects modulation on channel
4. Receiver decodes modulation as a message containing the secret
ATM Acoustic Channels

Domain of victim

Access

Transmitter

Secret

Channel

Domain of attacker

Decode

Receiver

Secret
ATM Acoustic Channels

- Secret: Pin

Diagram showing the flow of information from the domain of the victim through the channel to the domain of the attacker, with access, channel, and decode steps.
ATM Acoustic Channels

- Secret: Pin
- Transmitter: Keypad
ATM Acoustic Channels

- Secret: Pin
- Transmitter: Keypad
- Channel: Air
ATM Acoustic Channels

- Secret: Pin
- Transmitter: Keypad
- Channel: Air
- Modulation: Acoustic waves
ATM Acoustic Channels

- Secret: Pin
- Transmitter: Keypad
- Channel: Air
- Modulation: Acoustic waves
- Receiver: Cheap Microphone
ATM Acoustic Channels

- **Secret**: Pin
- **Transmitter**: Keypad
- **Channel**: Air
- **Modulation**: Acoustic waves
- **Receiver**: Cheap Microphone
- **Decoders**: ML Model
Physical vs Timing vs uArch Channel

- What can the adversary observe?
Physical vs Timing vs uArch Channel

- What can the adversary observe?

![Diagram showing physical channels: Power, EM, sound, etc. Attack requires measurement equipment, leading to physical access.]
Physical vs Timing vs uArch Channel

• What can the adversary observe?

Physical channels

Power, EM, sound...

Attacker requires measurement equipment → physical access

Timing channels

Response time

Attacker may be remote (e.g., over an internet connection)
Physical vs Timing vs uArch Channel

- What can the adversary observe?

**Physical channels**
- Processor
  - Victim
- Power, EM, sound...
- Attacker requires measurement equipment → physical access

**Timing channels**
- Processor
  - Victim
- Response time
- Attacker may be remote (e.g., over an internet connection)

**Microarchitectural channels**
- Processor
  - Victim
  - Attacker
- Microarch events (e.g., timing, perf. counters...)
- Attacker may be remote, or be co-located
What can you do with these channels?

- Violate privilege boundaries
  - Inter-process communication
  - Infer an application’s secret
- (Semi-Invasive) application profiling
What can you do with these channels?

- Violate privilege boundaries
  - Inter-process communication
  - Infer an application’s secret

- (Semi-Invasive) application profiling

Different from traditional software or physical attacks:
- Stealthy. Sophisticated mechanisms needed to detect channel
- Usually, no permanent indication one has been exploited
A Cache-based Channel

Process 1 (Xmtr) → Cache: # sets ← Process 2 (Receiver)
A Cache-based Channel

Process 1 (Xmtr) -> Cache: # sets 

write to set

Process 2 (Receiver)
A Cache-based Channel

write to set
A Cache-based Channel

Process 1 (Xmtr)

Cache:

Process 2 (Receiver)

if (send '0')
    idle
else
    write to a set

write to set
A Cache-based Channel

Process 1 (Xmtr)

if (send '0')
    idle
else
    write to a set

Cache:

Process 2 (Receiver)

write to set
if (send ‘0’)  
   idle  
else  
   write to a set  

write to set
A Cache-based Channel

Process 1 (Xmtr)

Process 2 (Receiver)

if (send '0')
   idle
else
   write to a set

write to set

\[ t1 = \text{rdtsc()} \]
\[ t2 = \text{rdtsc()} \]
A Cache-based Channel

if (send ‘0’)
    idle
else
    write to a set

write to set

t1 = rdtsc()
read from the set
t2 = rdtsc()

if t2 - t1 > hit_time:
    decode ‘1’
else
    decode ‘0’
A Cache-based Channel

**Diagram: A Cache-based Channel**

Process 1 (Xmtr) → Cache

- If `send '0'`
  - `idle`
- Else
  - `write to a set`

Cache: # sets

Process 2 (Receiver) → Cache

- `write to set`
- `t1 = rdtsc()`
- `t2 = rdtsc()`
- If `t2 - t1 > hit_time`:
  - Decode '1'
- Else
  - Decode '0'
A Cache-based Channel

**Process 1 (Xmtr)**

**Process 2 (Receiver)**

**Cache:**

```latex
\text{if (send '0')} \\
\hspace{1em} \text{idle} \\
\text{else} \\
\hspace{1em} \text{write to a set}
```

```latex
\text{write to set} \\
\text{t1 = rdtsc()} \\
\text{t2 = rdtsc()} \\
\text{if \ t2 - t1 > hit_time:} \\
\hspace{1em} \text{decode '1'} \\
\text{else} \\
\hspace{1em} \text{decode '0'}
```
A Cache-based Channel

if (send '0')
    idle
else
    write to a set

write to set

Process 1 (Xmtr)

Cache:

Process 2 (Receiver)

t1 = rdtsc()
read from the set
t2 = rdtsc()

if t2 - t1 > hit_time:
    decode '1'
else
    decode '0'
A Cache-based Channel

if (send '0')
  idle
else
  write to a set

write to a set

\[ t1 = \text{rdtsc()} \]
\[ t2 = \text{rdtsc()} \]

if \( t2 - t1 > \text{hit\_time} \):
  decode '1'
else
  decode '0'
A Cache-based Channel

if (send '0')
    idle
else
    write to a set

write to set

Cache:

Process 1 (Xmtr)

# sets

Process 2 (Receiver)

if t2 - t1 > hit_time:
    decode '1'
else
    decode '0'

t1 = rdtsc()
read from the set
t2 = rdtsc()
A Cache-based Channel

if (send ‘0’)
   idle
else
   write to a set

write to set

\[ t1 = \text{rdtsc}() \]
\[ t2 = \text{rdtsc}() \]

if \( t2 - t1 > \text{hit\_time} \):
   decode ‘1’
else
   decode ‘0’
A Cache-based Channel

if (send '0')
    idle
else
    write to a set

write to set

write to a set

Process 1 (Xmtr)

# sets

Process 2 (Receiver)

if t2 - t1 > hit_time:
    decode '1'
else
    decode '0'

if t1 = rdtsc():
    read from the set

        t1 = rdtsc()
        t2 = rdtsc()
A Cache-based Channel

if (send '0')
   idle
else
   write to a set

write to set

$\text{t1} = \text{rdtsc()}$
$\text{t2} = \text{rdtsc()}$

if $\text{t2} - \text{t1} > \text{hit\_time}$:
   decode '1'
else
   decode '0'

Process 1 (Xmtr)

Cache:

Process 2 (Receiver)
A Cache-based Channel

if (send ‘0’)
idle
else
write to a set

write to set

if t2 – t1 > hit_time:
decode ‘1’
else
decode ‘0’

Note this requires an “active” receiver
Communication w/ Active Receiver

- **Domain of victim**
  - Access to Transmitter
  - Secret

- **Channel**

- **Domain of attacker**
  - Receiver
  - Decode
  - Secret
1. An active receiver may need to “precondition” the channel to prepare for detecting modulation
1. An active receiver may need to “precondition” the channel to prepare for detecting modulation.
1. An active receiver may need to “precondition” the channel to prepare for detecting modulation.

2. An active receiver also needs to deal with synchronization of transmission (modulation) activity with reception (demodulation) activity.
A Multi-way Cache-based Channel

Cache:

Process 1 (Xmtr)

Process 2 (Receiver)
A Multi-way Cache-based Channel

Process 1 (Xmtr)

Cache:

Process 2 (Receiver)

# sets

fill a set
A Multi-way Cache-based Channel

Cache:

Process 1 (Xmtr) -> # sets -> fill a set -> Process 2 (Receiver)
A Multi-way Cache-based Channel

if (send '0')
  idle
else
  write to a set

fill a set
A Multi-way Cache-based Channel

if (send '0')
   idle
else
   write to a set

fill a set
A Multi-way Cache-based Channel

Process 1 (Xmtr)

Process 2 (Receiver)

Cache:

if (send '0')
    idle
else
    write to a set

fill a set
A Multi-way Cache-based Channel

**Cache:**

Process 1 (Xmtr)

if (send '0')
    idle
else
    write to a set

Process 2 (Receiver)

fill a set

fill a set

fill a set

read all of the set

t1 = rdtsc()

t2 = rdtsc()
A Multi-way Cache-based Channel

Process 1 (Xmtr)

Cache:

Process 2 (Receiver)

if (send '0')
    idle
else
    write to a set

fill a set

$\text{t1} = \text{rdtsc()}$

read all of the set

$\text{t2} = \text{rdtsc()}$
A Multi-way Cache-based Channel

Cache:

Process 1 (Xmtr)

fill a set

Process 2 (Receiver)

if (send ‘0’)
    idle
else
    write to a set

\[
t_1 = \text{rdtsc()}
\]

\[
t_2 = \text{rdtsc()}
\]

if \( t_2 - t_1 > \text{hit\_time} \):
    decode ‘1’
else
    decode ‘0’
Disrupting Communication

Process 1 (Xmtr) → # sets → Process 2 (Receiver)
Disrupting Communication

Kirianski et. al. Dawg, Micro’18
Disrupting Communication

Kirianski et. al. Dawg, Micro’18
Disrupting Communication

Kirianski et. al. Dawg, Micro’18
Disrupting Communication

if (send '0')
   idle
else
   write to a set

Kirianski et. al. Dawg, Micro'18
Disrupting Communication

if (send ’0’) 
    idle
else 
    write to a set

fill a set

Kirianski et. al. Dawg, Micro’18
Disrupting Communication

Cache:

- Process 1 (Xmtr)
- Process 2 (Receiver)

# sets

if (send '0')
   idle
else
   write to a set

fill a set

Kirianski et. al. Dawg, Micro’18
Disrupting Communication

**Process 1 (Xmtr)**

if (send ‘0’)
  *idle*
else
  *write to a set*

**Process 2 (Receiver)**

**Cache:**

fill a set

```
t1 = rdtsc()
read all of the set
t2 = rdtsc()
if t2 - t1 > hit_time:
  decode ‘1’
else
  decode ‘0’
```

*Kirianski et. al. Dawg, Micro’18*
Disjoint Channels

- Domain of victim
  - Access
  - Transmitter
  - Secret

- Channel

- Domain of attacker
  - Receiver
  - Decode
  - Secret
Disjoint Channels

1. Making disjoint channels makes communication impossible.
Disjoint Channels

1. Making disjoint channels makes communication impossible.

2. Channel can be allocated by “domain” and will need to be “cleaned” as processes enter and leave running state, so next process cannot see any “modulation” on the channel.
• Adding a single hash makes it difficult for the receiver to craft an address that monitors a specific set because addresses in each process will not match one-to-one.
Obfuscating the channel (1)

• Adding a single hash makes it difficult for the receiver to craft an address that monitors a specific set because addresses in each process will not match one-to-one.
Obfuscating the channel (1)

- Adding a single hash makes it difficult for the receiver to craft an address that monitors a specific set because addresses in each process will not match one-to-one.
Obfuscating the channel (1)

- Adding a single hash makes it difficult for the receiver to craft an address that monitors a specific set because addresses in each process will not match one-to-one.
Communication with subchannels

Domain of victim

Transmitter

Secret

Access

Subchannel

Subchannel

Subchannel

Domain of attacker

Receiver

Decode

Secret
Communication with subchannels

1. Transmissions may now occur on one of many subchannels
Communication with subchannels

1. Transmissions may now occur on one of many subchannels.

2. With a single hash, analysis by the receiver can, however, figure out which subchannel will be modulated.
Obfuscating the channel (2)

- Adding a process dependent hash makes the needed cache collision probabilistic.
- Now the receiver needs an extra step to find a way to probe a variety of “channels” to detect modulation.
Obfuscating the channel (2)

- Adding a process dependent hash makes the needed cache collision probabilistic.
- Now the receiver needs an extra step to find a way to probe a variety of “channels” to detect modulation.
Obfuscating the channel (2)

- Adding a process dependent hash makes the needed cache collision probabilistic.
- Now the receiver needs an extra step to find a way to probe a variety of “channels” to detect modulation.
Obfuscating the channel (2)

- Adding a process dependent hash makes the needed cache collision probabilistic.
- Now the receiver needs an extra step to find a way to probe a variety of “channels” to detect modulation.
Receiver Calibration

Domain of victim

Transmitter

Access

Secret

Subchannel

Domain of attacker

Receiver

Subchannel

Subchannel

Decode

Secret

Calibration
1. The calibration unit determines which subchannels (addresses) the receiver needs to use to detect modulation by a transmission.
1. The calibration unit determines which subchannels (addresses) the receiver needs to use to detect modulation by a transmission

2. The receiver may just observe known transmissions by the transmitter to determine the subchannels to monitor
1. The calibration unit determines which subchannels (addresses) the receiver needs to use to detect modulation by a transmission

2. The receiver may just observe known transmissions by the transmitter to determine the subchannels to monitor

3. Or, the receiver may provoke the transmitter to make a particular transmission.
Hashing* variations

• Nature of hash
  – Well-known
  – Secret
  – Cryptographic (per machine key)

*Hash -> address to set index mapping
Hashing* variations

- **Nature of hash**
  - Well-known
  - Secret
  - Cryptographic (per machine key)

- **Hashes per core**
  - Single for all processes
  - Per process hash

*Hash -> address to set index mapping*
Hashing* variations

• Nature of hash
  – Well-known
  – Secret
  – Cryptographic (per machine key)

• Hashes per core
  – Single for all processes
  – Per process hash

• Variation with time
  – Unchanging
  – Fixed interval in accesses (all sets at once or subset of sets)
  – Random interval (all sets at once or subset of sets)

*Hash -> address to set index mapping
Hashing* variations

- Nature of hash
  - Well-known
  - Secret
  - Cryptographic (per machine key)

- Hashes per core
  - Single for all processes
  - Per process hash

- Variation with time
  - Unchanging
  - Fixed interval in accesses (all sets at once or subset of sets)
  - Random interval (all sets at once or subset of sets)

- Hashes per address
  - Single or multiple

*Hash -> address to set index mapping
Generalizes to Other Resources

Xmtr

if (send '1')
    Use resource
else
    idle

Hardware resource

$t1 = rdtsc()$

Use resource

$t2 = rdtsc()$

if ($t2 - t1 > THRESH$)
    read '1'
else
    read '0'

Receiver
# Types of State-based Channels

<table>
<thead>
<tr>
<th>Resource</th>
<th>Shared by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private cache (L1, L2)</td>
<td>Intra-core</td>
</tr>
<tr>
<td>Shared cache (LLC)</td>
<td>On-socket cross core</td>
</tr>
<tr>
<td>Cache directory</td>
<td>Cross socket</td>
</tr>
<tr>
<td>DRAM row buffer</td>
<td>Cross socket</td>
</tr>
<tr>
<td>TLB (private/shared)</td>
<td>Intra-core/Inter-core</td>
</tr>
<tr>
<td>Branch Predictor</td>
<td>Intra-core</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Simple Transmitter

```
secret = oneof(0..1)
if secret == 1:
    x = channel
```
Simple Transmitter

\[
\text{secret} = \text{oneof}(0..1)
\]

\[
\text{if secret} == 1:
\]

\[
\text{x} = \text{channel}
\]
Simple Transmitter

\[
\text{secret} = \text{oneof}(0..1)
\]

\[
\text{if secret == 1:}
\]
\[
\text{x = channel}
\]

Like an amplitude modulated (AM) radio transmission
“AM” Transmitter in RSA
[Percival 2005]

- 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 \( r \)
"AM" Transmitter in RSA
[Percival 2005]

- Assume square-and-multiply based exponentiation

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

Output: \( b^e \mod m \)

\[
\begin{align*}
r & = 1 \\
\text{for } i = n-1 \text{ down to } 0 \text{ do} \\
& \quad r = \sqrt{r} \\
& \quad r = \mod(r, m) \\
& \quad \text{if } e_i == 1 \text{ then} \\
& \qquad r = \text{mul}(r, b) \\
& \qquad r = \mod(r, m) \\
& \quad \text{end} \\
\text{end} \\
\text{return } r
\end{align*}
\]

Secret-dependent memory access → transmitter
Noise in the channel

Process 1 (Xmtr)

Cache:

# sets

Process 2 (Receiver)
Noise in the channel

Process 1 (Xmtr) -> ! sets ! -> Process 2 (Receiver)

write to set
Noise in the channel

Cache:

Process 1 (Xmtr) → # sets → Yellow Box → Process 2 (Receiver)

write to set
Noise in the channel

process 1 (Xmtr)

write to set

if (send '0')
   idle
else
   write to a set

process 2 (Receiver)

write to set
Noise in the channel

if (send '0')

idle

else

write to a set

write to set
Noise in the channel

if (send '0')
    idle
else
    write to a set

write to set
Noise in the channel

Process 1 (Xmtr)

Process 2 (Receiver)

Process 3 (Xmtr)

Cache:

if (send '0')
  idle
else
  write to a set

write to set
Noise in the channel

if (send ‘0’)
    idle
else
    write to a set

write to set

\[ t_1 = \text{rdtsc()} \]
\[ t_2 = \text{rdtsc()} \]
Noise in the channel

# sets

Cache:

Process 1 (Xmtr)

Process 3 (Xmtr)

if (send '0')
   idle
else
   write to a set

write to set

Process 2 (Receiver)

t1 = rdtsc()
read from the set
t2 = rdtsc()
Noise in the channel

Cache:

Process 1 (Xmtr)

Process 2 (Receiver)

Process 3 (Xmtr)

if (send '0')
  idle
else
  write to a set

write to set

\[
t1 = \text{rdtsc()}
\]

\[
t2 = \text{rdtsc()}
\]

if \( t2 - t1 > \text{hit\_time} \)
  decode '1'
else
  decode '0'
Noise in the channel

Process 1
(Xmtr)

Process 3
(Xmtr)

Cache:

Process 2
(Receiver)

if (send '0')
  idle
else
  write to a set

write to set

t1 = rdtsc()
read from the set
t2 = rdtsc()

if t2 - t1 > hit_time:
  decode '1'
else
  decode '0'

Receiver interprets “noise” as a signal!
Channel Noise

Domain of victim
- Transmitter
  - Access
  - Secret

Channel

Domain of attacker
- Receiver
  - Decode
  - Secret
Channel Noise

Domain of victim

Transmitter

Access

Secret

Channel

Noise

Receiver

Decode

Secret

Domain of attacker
1. Another (or the same) transmitter may introduce changes of state (noise) into the channel which will confound the receiver.
Channel Noise

1. Another (or the same) transmitter may introduce changes of state (noise) into the channel which will confound the receiver.
2. Reception now becomes probabilistic, and a stochastic analysis is needed for the receiver to decode the modulation it sees in the channel.
1. Another (or the same) transmitter may introduce changes of state (noise) into the channel which will confound the receiver.

2. Reception now becomes probabilistic, and a stochastic analysis is needed for the receiver to decode the modulation it sees in the channel.

3. Increases in reliability of reception can be improved by improved message encoding, e.g., by repeating the message.
Types of Transmitters

- Types of transmitter:
  1. Pre-existing so victim itself leaks secret, (e.g., RSA keys)
Another Transmitter

\[
\text{secret} = \text{oneof}(0..3) \\
\text{subchannel}[\text{secret}] = 1
\]
Another Transmitter

\[
\text{secret} = \text{oneof}(0..3) \\
\text{subchannel}[\text{secret}] = 1
\]

Modulation for sending 0..3
Another Transmitter

\[
\text{secret} = \text{oneof}(0..3) \\
\text{subchannel}[\text{secret}] = 1
\]

Modulation for sending 0..3

Like a frequency modulated (FM) radio transmission
Address Space

0x0  

User pages

Kernel pages

0xFF...F
Reminder: Speculative Execution

- In x86, a page table can have kernel pages which are only accessible in kernel mode:
  - This avoids switching page tables on context switches, but
Reminder: Speculative Execution

- In x86, a page table can have kernel pages which are only accessible in kernel mode:
  - This avoids switching page tables on context switches, but
  - Hardware speculatively assumes that there will not be an illegal access, so instructions following an illegal instruction are executed speculatively.
Reminder: Speculative Execution

• In x86, a page table can have kernel pages which are only accessible in kernel mode:
  − This avoids switching page tables on context switches, but
  − Hardware speculatively assumes that there will not be an illegal access, so instructions following an illegal instruction are executed speculatively.

• So what does the following code do when run in user mode:

  ```
  val = *kernel_address;
  ```
Reminder: Speculative Execution

- In x86, a page table can have kernel pages which are only accessible in kernel mode:
  - This avoids switching page tables on context switches, but
  - Hardware speculatively assumes that there will not be an illegal access, so instructions following an illegal instruction are executed speculatively.

- So what does the following code do when run in user mode?

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

- Causes a protection fault, but data at “kernel_address” is speculatively read and loaded into val!
“FM” Transmitter - Meltdown
[Lipp et al. 2018]

1. Preconditioning: Receiver allocates subchannels in subchannels[256] and flushes all its cache lines
“FM” Transmitter - Meltdown
[Lipp et al. 2018]

1. Preconditioning: Receiver allocates subchannels in subchannels[256] and flushes all its cache lines
2. Transmit: Transmitter (controlled by attacker) executes

```c
uint8_t secret = *kernel_address;
subchannels[secret] = 1;
```
“FM” Transmitter - Meltdown
[Lipp et al. 2018]

1. Preconditioning: Receiver allocates subchannels in subchannels[256] and flushes all its cache lines

2. Transmit: Transmitter (controlled by attacker) executes

   ```
   uint8_t secret = *kernel_address;
   subchannels[secret] = 1;
   ```

3. Receive: After handling protection fault, receiver times accesses to all of subchannels[256], finds the subchannel that was “modulated”, i.e., hits, and therefore has “decoded” a secret byte.
“FM” Transmitter - Meltdown
[Lipp et al. 2018]

1. Preconditioning: Receiver allocates subchannels in subchannels[256] and flushes all its cache lines

2. Transmit: Transmitter (controlled by attacker) executes

   ```c
   uint8_t secret = *kernel_address;
   subchannels[secret] = 1;
   ```

3. Receive: After handling protection fault, receiver times accesses to all of subchannels[256], finds the subchannel that was “modulated”, i.e., hits, and therefore has “decoded” a secret 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
Types of Transmitters

1. Pre-existing so victim itself leaks secret, (e.g., RSA keys)
2. Programmed and invoked by attacker (e.g., Meltdown)
Spectre variant 2
[Kocher et al. 2018]
Spectre variant 2
[Kocher et al. 2018]

• Consider a situation where there is some kernel code that looks like the following:
Spectre variant 2
[Kocher et al. 2018]

- Consider a situation where there is some kernel code that looks like the following:

```c
xmit: uint8_t index = *kernel_address;
random_array[index] = 1;
```
Spectre variant 2
[Kocher et al. 2018]

• Consider a situation where there is some kernel code that looks like the following:

```c
xmit: uint8_t index = *kernel_address;
random_array[index] = 1;
```

• Interpret that code as an FM transmitter:
Spectre variant 2
[Kocher et al. 2018]

- Consider a situation where there is some kernel code that looks like the following:

```c
xmit: uint8_t index = *(kernel_address);
random_array[index] = 1;
```

- Interpret that code as an FM transmitter:

```c
xmit: uint8_t secret = *(kernel_address);
subchannels[secret] = 1;
```
Spectre variant 2
[Kocher et al. 2018]

• Consider a situation where there is some kernel code that looks like the following:

```c
xmit: uint8_t index = *kernel_address;
random_array[index] = 1;
```

• Interpret that code as an FM transmitter:

```c
xmit: uint8_t secret = *kernel_address;
subchannels[secret] = 1;
```

• But that is kernel code that we cannot execute directly, so if only we could make the kernel jump to “xmit” we could invoke the transmitter...
Spectre variant 2
[Kocher et al. 2018]
Spectre variant 2
[Kocher et al. 2018]

- Now assume there is another bit of code in that kernel routine that we can force to be executed looks like:
• Now assume there is another bit of code in that kernel routine that we can force to be executed looks like:

• Using aliased addresses for `abc` and `xmit` train the BTB to jump to from `abc` to `xmit`
Spectre variant 2
[Kocher et al. 2018]

• Now assume there is another bit of code in that kernel routine that we can force to be executed looks like:

• Using aliased addresses for `abc` and `xmit` train the BTB to jump to from `abc` to `xmit`

• Now invoke the kernel in a way that executes `abc` and the transmitter will speculatively jump to `xmit` and execute the transmitter and send the secret....
Spectre variant 2
[Kocher et al. 2018]

• Now assume there is another bit of code in that kernel routine that we can force to be executed looks like:

• Using aliased addresses for `abc` and `xmit` train the BTB to jump to from `abc` to `xmit`

• Now invoke the kernel in a way that executes `abc` and the transmitter will speculatively jump to `xmit` and execute the transmitter and send the secret....

• Note since most BTBs store partial tags and targets it can be hard to get the BTB to jump to an arbitrary address, so Spectre uses the indirect jump predictor.
Spectre variant 2
[Kocher et al. 2018]

• Now assume there is another bit of code in that kernel routine that we can force to be executed looks like:

  abc: br xyz

• Using aliased addresses for `abc` and `xmit` train the BTB to jump to from `abc` to `xmit`

• Now invoke the kernel in a way that executes `abc` and the transmitter will speculatively jump to `xmit` and execute the transmitter and send the secret....

• Note since most BTBs store partial tags and targets it can be hard to get the BTB to jump to an arbitrary address, so Spectre uses the indirect jump predictor.
Types of Transmitters

- **Types of transmitter:**
  1. Pre-existing so victim itself leaks secret, (e.g., RSA keys)
  2. Programmed and invoked by attacker (e.g., Meltdown)
  3. Synthesized from existing victim code and invoked by attacker (e.g., Spectre V2)
• Consider the following kernel code, e.g., in a system call
  
  ```c
  if (x < array1_size)
    y = array2[array1[x] * 4096];
  ```
Spectre variant 1
[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. Precondition: Attacker invokes this kernel code with small values of $x$ to train the branch predictor to be taken
Spectre variant 1
[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. Precondition: Attacker invokes this kernel code with small values of $x$ to train the branch predictor to be 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, *speculatively* fetches $array2[array1[x] * 4096]$’s line into the cache.
Spectre variant 1
[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. Precondition: Attacker invokes this kernel code with small values of $x$ to train the branch predictor to be 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, speculatively 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 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?
Coming Spring 2022...

Learn to attack processors...

Side channel attacks
Transient/ speculative execution attacks
Row-hammer attacks

SGX Enclave Design
Hardware support for memory safety
And more!

And learn to defend them!

Take 6.888 This Spring!

Mengjia Yan
mengjia@csail.mit.edu

Graduate-Level/ AUS
12 Units (3-0-9)
MW 1:00 - 2:30
Thank you!