Security

Mengjia Yan
Computer Science & Artificial Intelligence Lab
M.I.T.
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 is a timing-independent interface, 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 is a **timing-independent** interface, and
  - Specify *what* should happen, not *when*

- ISA only specifies **architectural** updates (reg, mem, PC...)
  - *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 is a **timing-independent** interface, and
  - Specify *what* should happen, not *when*

- ISA only specifies **architectural** updates (reg, mem, PC...)
  - *Micro-architectural changes are left unspecified*

- So implementation details and timing behaviors (e.g., microarchitectural state, power, etc.) have been exploited to breach security mechanisms.
Security and Information Leakage

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

• ISA is a timing-independent interface, and
  – Specify *what* should happen, not *when*

• ISA only specifies architectural updates (reg, mem, PC...)
  – *Micro-architectural changes are left unspecified*

• So implementation details and timing behaviors (e.g., microarchitectural state, power, etc.) have been exploited to breach security mechanisms.

• In specific, they have been used as *channels* to leak information!
Standard Communication Model

- Sender
  - Message
- Recipient
Standard Communication Model

Sender

Message

Recipient

Message
1. Transmitter gets a message
Standard Communication Model

1. Transmitter gets a message
Standard Communication Model

1. Transmitter gets a message
2. Transmitter modulates channel
1. Transmitter gets a message
2. Transmitter modulates channel
3. Receiver detects modulation on channel
Standard Communication Model

1. Transmitter gets a message
2. Transmitter modulates channel
3. Receiver detects modulation on channel
4. 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.
- **Secret** – the “message” that is transmitted on the channel and detected by the receiver.
- **Channel** – some “state” that can be changed, i.e., modulated, by the “transmitter” and whose modulation can be detected by the “receiver”.

```
<table>
<thead>
<tr>
<th>Domain of victim</th>
<th>Transmitter</th>
<th>Channel</th>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Secret</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Domain of attacker</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Secret</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Decode</td>
</tr>
</tbody>
</table>
```

November 28, 2022
Communication Model of Attacks
[Belay, Devadas, Emer]

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

1. Transmitter “accesses” secret
2. Transmitter modulates channel (\textit{microarchitectural state}) 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

Transmitter

Channel

Domain of attacker

Receiver

Access

Secret

Decode

Secret
ATM Acoustic Channels

- Secret:
- Transmitter:
- Channel:
- Modulation:
- Receiver:
- Decoders:
ATM Acoustic Channels

- Secret: Pin
- Transmitter:
- Channel:
- Modulation:
- Receiver:
- Decoders:
ATM Acoustic Channels

- **Domain of victim**
  - Transmitter
  - Channel

- **Domain of attacker**
  - Receiver

- **Access**
- **Decode**
- **Secret:** Pin
- **Transmitter:** Keypad
- **Channel:**
- **Modulation:**
- **Receiver:**
- **Decoders:**
ATM Acoustic Channels

- Secret:
- Transmitter: Keypad
- Channel: Air
- Modulation:
- Receiver:
- Decoders:
ATM Acoustic Channels

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

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

- **Secret:** Pin
- **Transmitter:** Keypad, Air
- **Channel:** Acoustic waves
- **Modulation:** Cheap Microphone
- **Receiver:** ML Model
- **Decoders:**

- Domain of victim
  - Access
  - Secret
  - Transmitter
  - Channel

- Domain of attacker
  - Receiver
  - Decode
ATM Acoustic Channels

Domain of victim

- Transmitter
- Access
- Secret

Channel

- Receiver
- Decode
- Secret

Domain of attacker

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

• Types of channels

Physical channels

Power, EM, sound...

Processor

Victim

Attacker requires measurement equipment → physical access
Physical vs Timing vs uArch Channel

• Types of channels

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)
def check(input):

    size = len(passwd);

    for i in range(0,size):
        if (input [i] == password[i]):
            return ("error");

    return ("success")
Timing Channel Example

def check(input):
    size = len(passwd);
    for i in range(0, size):
        if (input[i] == password[i]):
            return ('error');
    return ('success')

Can you guess the password by monitoring the execution time?
Timing Channel Example

```python
def check(input):
    size = len(passwd);
    for i in range(0,size):
        if input[i] == password[i]:
            return "error";
    return "success"
```

Can you guess the password by monitoring the execution time?

The execution time is dependent on how many characters match between the input and the correct password. Attacker can brute-force each character.
Physical vs Timing vs uArch Channel

- **Types of channels**

  **Physical channels**
  - 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
Physical vs Timing vs uArch Channel

• Types of channels

- **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)

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

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

- (Semi-Invasive) application profiling
What can you do with uArch 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
Side Channel Attacks in 1977

• A side channel due to disk arm optimization
  – Enqueues requests by ascending cylinder number and dequeues (executes) them by the "elevator algorithm."
Side Channel Attacks in 1977

• A side channel due to disk arm optimization
  – Enqueues requests by ascending cylinder number and dequeues (executes) them by the "elevator algorithm."

• Example:
  – R issues a request to 55
  – S issues a request to either 53 or 57
  – R then issues requests to both 52 and 58
Side Channel Attacks in 1977

• A side channel due to disk arm optimization
  – Enqueues requests by ascending cylinder number and dequeues (executes) them by the "elevator algorithm."

• Example:
  – R issues a request to 55
  – S issues a request to either 53 or 57
  – R then issues requests to both 52 and 58

Q: If the request to 52 returns first, can we guess what did S issue before?
Side Channel Attacks in 1977

• A side channel due to disk arm optimization
  – Enqueues requests by ascending cylinder number and dequeues (executes) them by the "elevator algorithm."

• Example:
  – R issues a request to 55
  – S issues a request to either 53 or 57
  – R then issues requests to both 52 and 58

Q: If the request to 52 returns first, can we guess what did S issue before?

53
Side Channel Attacks in 1977

• A side channel due to disk arm optimization
  – Enqueues requests by ascending cylinder number and dequeues (executes) them by the "elevator algorithm."

• Example:
  – R issues a request to 55
  – S issues a request to either 53 or 57
  – R then issues requests to both 52 and 58

Q: If the request to 52 returns first, can we guess what did S issue before?

53

Note this requires an “active” receiver that precondition the channel
Communication w/ Active Receiver

Domain of victim

Transmitter

Access

Secret

Channel

Domain of attacker

Receiver

Decode

Secret
• An active receiver may need to “precondition” the channel to prepare for detecting modulation
An active receiver may need to "precondition" the channel to prepare for detecting modulation.
Communication w/ Active Receiver

- An active receiver may need to “precondition” the channel to prepare for detecting modulation.

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

Process 1 (Xmtr)  

Cache:

# sets

Process 2 (Receiver)
A Cache-based Channel

Cache:

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

write to set
A Cache-based Channel

![Diagram showing a cache-based channel with Process 1 (Xmtr) writing to a set in the cache, which is then read by Process 2 (Receiver). The diagram includes a note: write to set.]
A Cache-based Channel

Cache:

Process 1 (Xmtr)

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

Process 2 (Receiver)

write to set
A Cache-based Channel

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

write to set
A Cache-based Channel

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

write to set

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 = 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
```

```
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 set

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

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

Process 1 (Xmtr)

if \( \text{send '0'} \)

idle

else

write to a set

write to set

t1 = \text{rdtsc()}

read from the set
t2 = \text{rdtsc()}

if \ t2 - t1 > \text{hit_time}:
    decode '1'
else
    decode '0'

Process 2 (Receiver)
A Cache-based Channel

Process 1 (Xmtr)

if (\textbf{send} ‘0’) \texttt{idle}
else \texttt{write to a set}

write to set

Process 2 (Receiver)

t1 = \texttt{rdtsc()}
\texttt{read from the set}
t2 = \texttt{rdtsc()}

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

Cache:

Process 1 (Xmtr)

if (\texttt{send '0')} 
    \texttt{idle} 
else 
    \texttt{write to a set} 

write to set 

Process 2 (Receiver)

t1 = rdtsc() 
read from the set 
t2 = rdtsc() 
if t2 - t1 > \texttt{hit_time:} 
    decode '1' 
else 
    decode '0'
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

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

Process 1 (Xmtr)

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’

Process 2 (Receiver)
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 \bmod m
r = 1
for i = n-1 down to 0 do
    r = sqrt(r)
    r = mod(r,m)
    if e_i == 1 then
        r = mul(r,b)
        r = mod(r,m)
    end
end
return r
```
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 $r$

Secret-dependent memory access $\rightarrow$ transmitter
A Multi-way Cache-based Channel
A Multi-way Cache-based Channel

Cache:

Process 1 (Xmtr)

Process 2 (Receiver)

fill a set
A Multi-way Cache-based Channel

Cache:

Process 1 (Xmtr)

Process 2 (Receiver)

fill a set
A Multi-way Cache-based Channel

Cache:

Process 1 (Xmtr)

Process 2 (Receiver)

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

fill a set
A Multi-way Cache-based Channel

Cache:

Process 1 (Xmtr)

Process 2 (Receiver)

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

**Cache:**

Process 1 (Xmtr)

Process 2 (Receiver)

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

fill a set

t1 = rdtsc()
read all of the set
t2 = rdtsc()
A Multi-way Cache-based Channel

Cache:

Process 1 (Xmtr)  
if (send ‘0’)
  idle
else
  write to a set

Process 2 (Receiver)

# sets

fill a set

t1 = rdtsc()
read all of the set
t2 = rdtsc()
A Multi-way Cache-based Channel

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

fill a set

Process 1 (Xmtr)

Cache:

Process 2 (Receiver)

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

t1 = rdtsc()

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

Cache:

Process 1 (Xmtr)

Precondition (Prime)

Process 2 (Receiver)

Transmit

if (send ‘0’)

idle

else

write to a set

Receive (Probe)

t1 = rdtsc()

read all of the set

t2 = rdtsc()

if t2 - t1 > hit_time:

decode ‘1’

else

decode ‘0’

Decode

fill a set
Generalizes to Other Resources

if (send '1')
  Use resource
else
  idle

\[
t_1 = \text{rdtsc()}
\]
Use resource
\[
t_2 = \text{rdtsc()}
\]

if (t_2 - t_1 > \text{THRESH})
  read '1'
else
  read '0'
Generalizes to Other Resources

Any other exploitable structures?
# Channel Examples

<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>Network-on-chip</td>
<td>On-socket cross core</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Disrupting Communication

Cache:

Process 1 (Xmtr) -> # sets -> Process 2 (Receiver)
Disrupting Communication

"We found that identifying all of the sources of accurate clocks was much easier than finding all of the possible timing channels in the system. ...
... If we could make the clocks less accurate, then the effective bandwidth of all timing channels in the system would be lowered." (1991)
Secret-independent Channel Modulation

- Different from conventional communication, this is a side channel (*unintended* communication).
Secret-independent Channel Modulation

- Different from conventional communication, this is a side channel (*unintended* communication).

- One mitigation is to not use the channel.
Secret-independent Channel Modulation

- Different from conventional communication, this is a side channel (unintended communication).

- One mitigation is to not use the channel.
  -> “data-oblivious execution” or “constant-time programming”.

Domain of victim

Transmitter

Access

Secret

Domain of attacker

Channel

Decode

Receiver

Secret
Secret-independent Channel Modulation

**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

$r = \sqrt{r}$

$r = \mod(r,m)$

if $e_i == 1$ then

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

$r = \mod(r,m)$

end

end

return $r$

*How to make the code execution independent of the secret?*
Secret-independent Channel Modulation

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
  $r = \text{sqrt}(r)$
  $r = \text{mod}(r,m)$
  if $e_i == 1$ then
    $r = \text{mul}(r,b)$
    $r = \text{mod}(r,m)$
  end
end

return $r$

How to make the code execution independent of the secret?

No secret-dependent branches, memory accesses, floating point operations
Secret-independent Channel Modulation

**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
  $r = \sqrt{r}$
  $r = \mod(r,m)$
  if $e_i == 1$ then
    $r = \mul(r,b)$
    $r = \mod(r,m)$
  end
end

return $r$

**How to make the code execution independent of the secret?**

No secret-dependent branches, memory accesses, floating point operations

$p = (e_i == 1)$
$r2 = \mul(r,b)$
$r2 = \mod(r,m)$
$cmov [p] r, r2$
Secret-independent Channel Modulation

**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

$r = \text{sqrt}(r)$

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

if $e_i == 1$ then

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

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

end

end

return $r$

---

*How to make the code execution independent of the secret?*

No secret-dependent branches, memory accesses, floating point operations

*After removing the secret-dependent branch, how about code inside these functions?*
Secret-independent Channel Modulation

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

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

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

**How to make the code execution independent of the secret?**

No secret-dependent branches, memory accesses, floating point operations

**After removing the secret-dependent branch, how about code inside these functions?**

**Constant-time programming is hard**
Disrupting Communication

Cache:

Process 1 (Xmtr) -> # sets -> Process 2 (Receiver)
Disrupting Communication

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

Process 1 (Xmtr)

Cache:

Process 2 (Receiver)

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

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

# sets

Process 1 (Xmtr)

Cache:

Process 2 (Receiver)

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

fill a set

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

Cache:

Process 1 (Xmtr)

fill a set

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

Process 2 (Receiver)

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

Process 1 (Xmtr)

Cache:

Process 2 (Receiver)

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

fill a set

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

Cache:

Process 1 (Xmtr)

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

fill a set

Process 2 (Receiver)

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

November 28, 2022
Disjoint Channels

- Making disjoint channels makes communication impossible.

- 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.
Communication with subchannels

- Transmissions may now occur on one of many subchannels
Communication with subchannels

- Transmissions may now occur on one of many subchannels
- With a single hash, analysis by the receiver can, however, figure out (reverse engineer) which subchannel will be modulated.
Simple Transmitter

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

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

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

\[
\text{secret} = \text{oneof}(0..1) \\
\text{if secret} == 1: \\
\text{x} = \text{channel}
\]

Like an amplitude modulated (AM) radio transmission (RSA example)
Another Transmitter

\[
\begin{align*}
\text{secret} & = \text{oneof}(0..3) \\
\text{subchannel}[\text{secret}] & = 1
\end{align*}
\]
Another Transmitter

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

Modulation for sending 0..3

```
0  0.2  0.4  0.6  0.8  1  1.2
Xmit 0  Xmit 1  Xmit 2  Xmit 3
```

- sub ch 0
- sub ch 1
- sub ch 2
- sub ch 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
(See later Meltdown)
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

Subchannel

Subchannel

Domain of attacker

Receiver

Decode

Secret

L22-30
• The calibration unit determines which subchannels the receiver needs to use to detect modulation by a transmission.
Receiver Calibration

- The calibration unit determines which subchannels the receiver needs to use to detect modulation by a transmission.
- During calibration, the receiver may just observe known transmissions by the transmitter or 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*
Noise in the channel

Process 1 (Xmtr) → Cache: # sets → Process 2 (Receiver)
Noise in the channel

Cache:

Process 1 (Xmtr) \rightarrow \text{write to set} \rightarrow \text{Cache} \rightarrow \text{Process 2 (Receiver)}
Noise in the channel

Process 1 (Xmtr)

Cache:

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

# sets

write to set

if (send ‘0’)  
idle
else
write to a set
Noise in the channel

Process 1 (Xmtr)

Process 3 (Xmtr)

Cache:

if (send '0')

idle

else

write to a set

write to set

Process 2 (Receiver)
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
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

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

\[ t2 = \text{rdtsc}() \]
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()
Noise in the channel

Process 1 (Xmtr)

Process 2 (Receiver)

Process 3 (Xmtr)

Cache:

write to set

if (send ‘0’)

idle

else

write to a set

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

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

if $t2 - t1 > \text{hit\_time}$:

decode ‘1’

else:

decode ‘0’
Noise in the channel

Process 1 (Xmtr)

Process 2 (Receiver)

Process 3 (Xmtr)

Cache:

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

Receiver interprets “noise” as a signal!

write to set

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

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

- Another (or the same) transmitter may introduce changes of state (noise) into the channel which will confound the receiver.
Another (or the same) transmitter may introduce changes of state (noise) into the channel which will confound the receiver.

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

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

Increases in reliability of reception can be improved by improved message encoding, e.g., by repeating the message.
Noise makes signal probabilistic

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

Modulation for sending 0..3
So far...

• The communication model provides a systematic way to reason about microarchitectural side channels
So far...

- The communication model provides a systematic way to reason about microarchitectural side channels
- Different attack strategies are usually different ways of modulating channels
So far...

- The communication model provides a systematic way to reason about microarchitectural side channels.
- Different attack strategies are usually different ways of modulating channels.
- To improve channel precision, need precondition, calibration, decoding techniques, noise => all have analogies to telecommunication.
Types of Transmitters

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

Address Space

- User pages
- Kernel pages

0x0 to 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
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 do?
  ```
  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 do?
  \[
  \text{val} = *\text{kernel_address};
  \]

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

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

1. Preconditioning: Receiver allocates an array 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 an array `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” to decode the `secret`. 

"FM" Transmitter Meltdown [Lipp et al. 2018]

1. Preconditioning: Receiver allocates an array `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” to decode the `secret`.

   • Result: Attacker can read arbitrary kernel data!
"FM" Transmitter Meltdown [Lipp et al. 2018]

1. Preconditioning: Receiver allocates an array `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” to decode the `secret`.

   • Result: Attacker can read arbitrary kernel data!
     - For higher performance, use transactional memory (protection fault aborts transaction on exception instead of invoking kernel)
“FM” Transmitter Meltdown [Lipp et al. 2018]

1. Preconditioning: Receiver allocates an array `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” to decode the `secret`.

   • Result: Attacker can read arbitrary kernel data!
     – For higher performance, use transactional memory (protection fault aborts transaction on exception instead of invoking kernel)
     – Mitigation?
“FM” Transmitter Meltdown [Lipp et al. 2018]

1. Preconditioning: Receiver allocates an array `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” to decode the `secret`.

   • 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
“FM” Transmitter Meltdown [Lipp et al. 2018]

1. Preconditioning: Receiver allocates an array 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” to decode the secret.

- 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
    Return zero upon permission check failure (supporting precise exception)
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)
Spectre variant 1
[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;
uint8_t dummy = random_array[index];
```
Spectre variant 1
[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;
uint8_t dummy = random_array[index];
```

• Interpret that code as an FM transmitter:

```c
xmit: uint8_t secret = *kernel_address;
uint8_t dummy = subchannels[secret];
```
Consider a situation where there is some kernel code that looks like the following:

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

Interpret that code as an FM transmitter:

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

But this kernel code is protected by a branch. *Can we make the kernel speculatively execute “xmit”?*

```c
if (kernel_address is public_region) {
    uint8_t index = *kernel_address;
    uint8_t dummy = subchannels[index];
}
```
Spectre variant 1
[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;
  uint8_t dummy = random_array[index];
  
  xmit: uint8_t secret = *kernel_address;
  uint8_t dummy = subchannels[secret];
  
  But this kernel code is protected by a branch. Can we make the kernel speculatively execute “xmit”?
  
  if (kernel_address is public_region) {
    uint8_t index = *kernel_address;
    uint8_t dummy = subchannels[index];
  }
  ```
• Consider the following kernel code, e.g., in a system call

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

1. Precondition: Flush all the elements in `array2` from 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: Flush all the elements in `array2` from cache
2. Train: 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: Flush all the elements in `array2` from cache
2. Train: Attacker invokes this kernel code with small values of `x` to train the branch predictor to be taken
3. Transmit: Attacker invokes this code with an out-of-bounds `x`, so that `&array1[x]` points to a desired kernel address. Core mispredicts branch, **speculatively** fetches address `&array2[array1[x] * 4096]` into the cache.
Consider the following kernel code, e.g., in a system call

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

1. **Precondition:** Flush all the elements in `array2` from cache
2. **Train:** Attacker invokes this kernel code with small values of `x` to train the branch predictor to be taken
3. **Transmit:** Attacker invokes this code with an out-of-bounds `x`, so that `array1[x]` points to a desired kernel address. Core mispredicts branch, **speculatively** fetches address `array2[array1[x] * 4096]` into the cache.
4. **Receive:** Attacker probes cache to infer which line of `array2` was fetched, learns data at kernel address
Spectre variant 2
[Kocher et al. 2018]

- Can also exploit indirect branch predictor:
  - Most BTBs store partial tags for source addresses
Spectre variant 2
[Kocher et al. 2018]

- Can also exploit indirect branch predictor:
  - Most BTBs store partial tags for source addresses

```c
kernel_address = a_desired_address;
jump some_where_else
...
kernel_address = a_safe_address;
jump xmit
...
xmit: uint8_t secret = *kernel_address;
uint8_t dummy = subchannels[secret];
```
Spectre variant 2
[Kocher et al. 2018]

• Can also exploit indirect branch predictor:
  – Most BTBs store partial tags for source addresses

```
kernelp_address = a_desired_address;
jump some_where_else
...
kernelp_address = a_safe_address;
jump xmit
...
xmit: uint8_t secret = *kernelp_address;
uint8_t dummy = subchannels[secret];
```

1. Train: trigger xyz->xmit many times
2. Transmit: ‘abc’ and ‘xyz’ alias in BTB, so we can speculatively trigger abc->xmit
3. Receive: similar to Spectre v1
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)
Spectre variants and mitigations

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

- Spectre relies on speculative execution, not late exception handling → 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 handling → 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 handling → 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 handling → 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?
Summary

- ISA is a **timing-independent** interface, and
  - Specify *what* should happen, not *when*

- ISA only specifies **architectural** updates
  - *Micro-architectural changes are left unspecified*

- So implementation details (e.g., speculative execution) and timing behaviors (e.g., microarchitectural state, power, etc.) have been exploited to breach security mechanisms.

- ISA, as a software-hardware contract, is insufficient for reasoning about microarchitectural security
Coming Spring 2023: Secure Hardware Design

Learn to attack processors...
- Side channel attacks
- Transient/ speculative execution attacks
- Row-hammer attacks

And more!

SGX Enclave Design
Hardware support for memory safety

And learn to defend them!

Take 6.S983 This Spring! (ne 6.888)

Mengjia Yan
mengjia@csail.mit.edu

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

November 28, 2022
http://csg.csail.mit.edu/6.888Yan/
Thank you!