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!

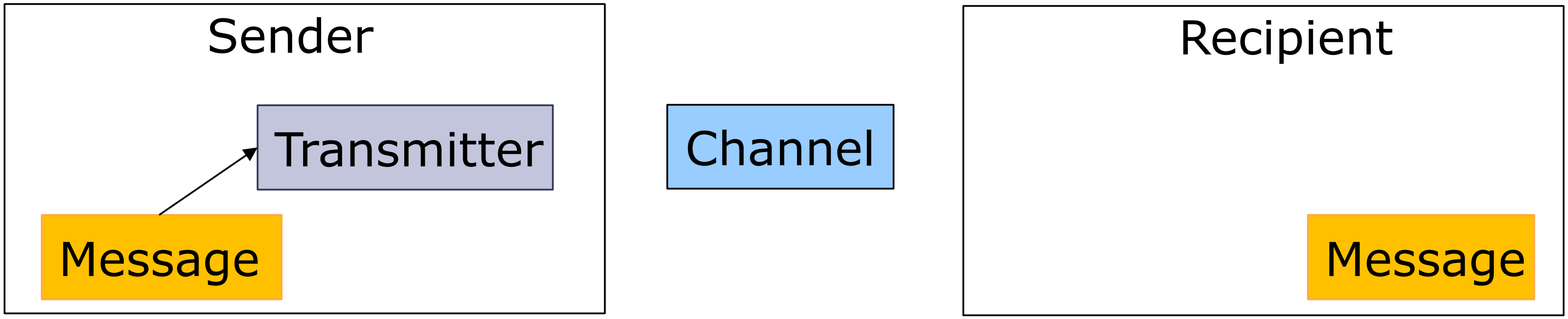
Standard Communication Model

Standard Communication Model

Standard Communication Model

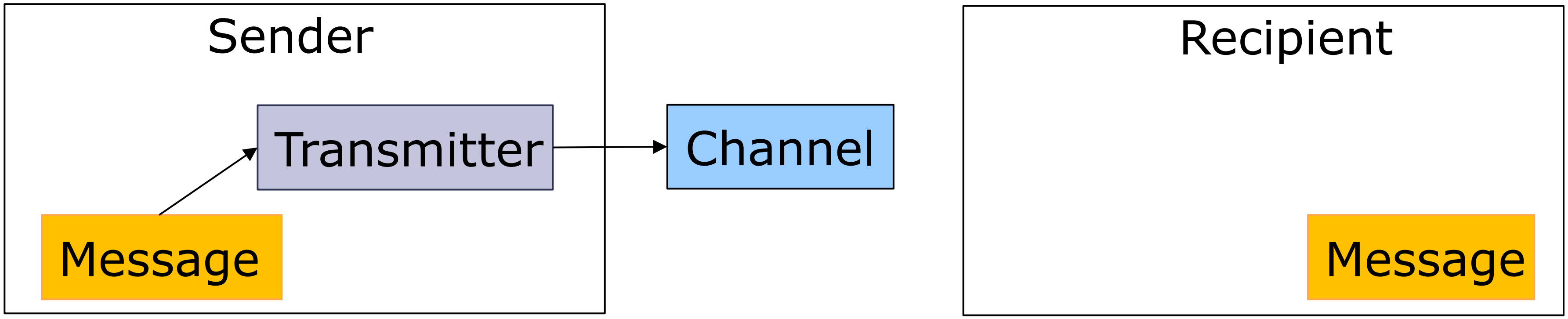
1. Transmitter gets a message

Standard Communication Model



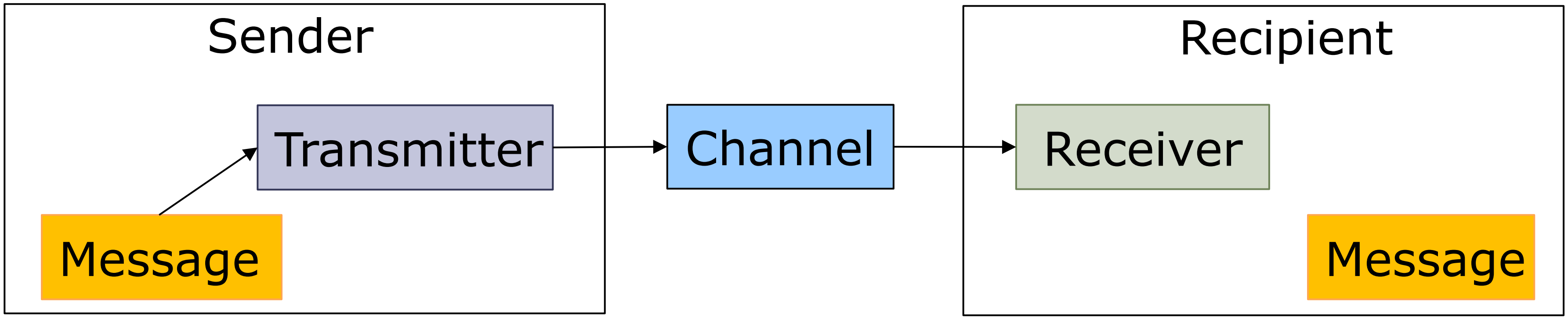
1. Transmitter gets a message

Standard Communication Model



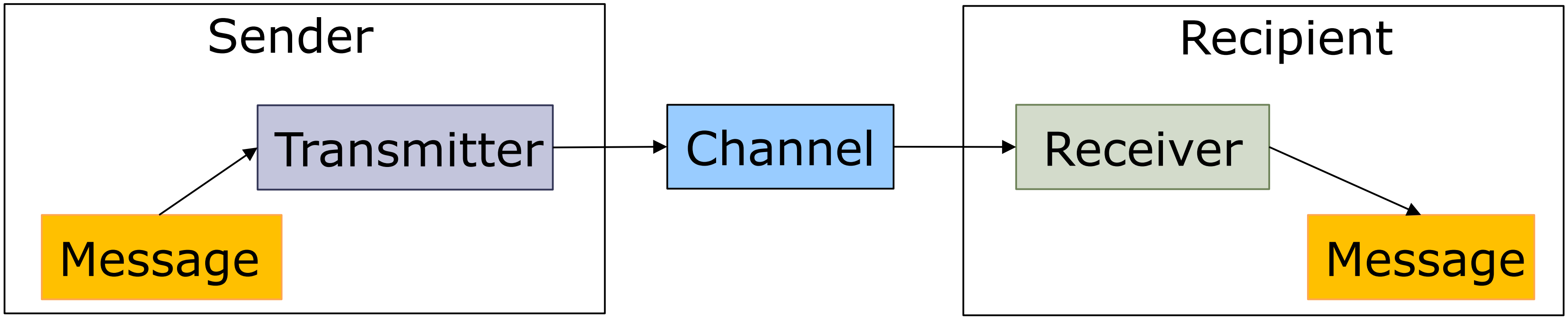
1. Transmitter gets a message
2. Transmitter modulates channel

Standard Communication Model



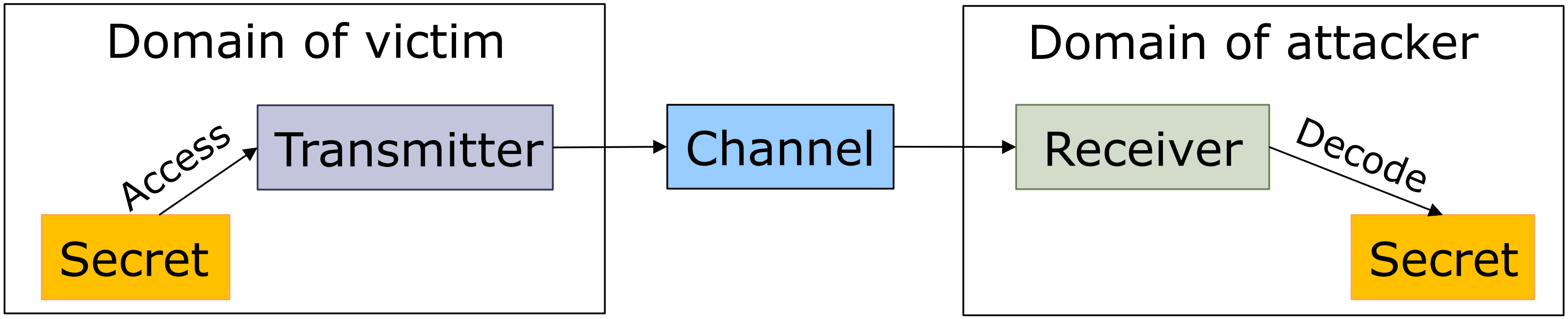
1. Transmitter gets a message
2. Transmitter modulates channel
3. Receiver detects modulation on channel

Standard Communication Model

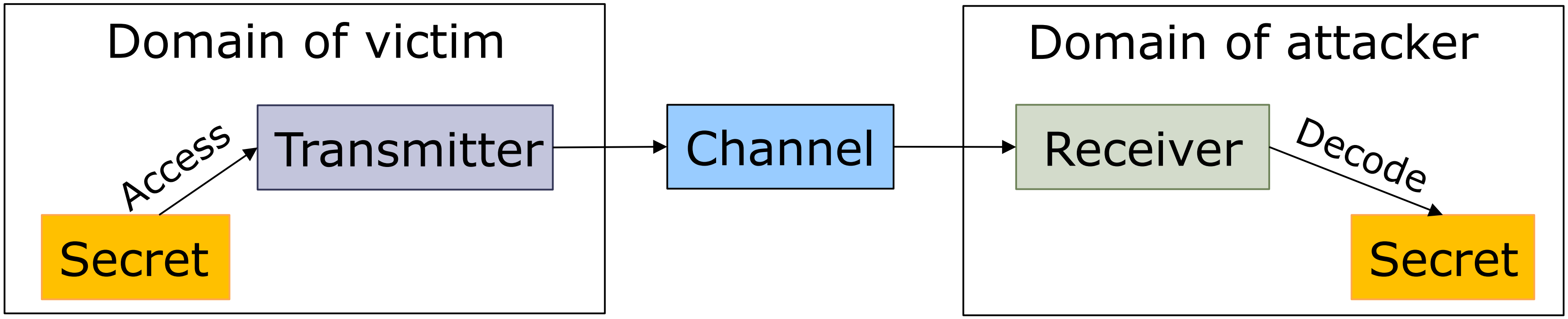


1. Transmitter gets a message
2. Transmitter modulates channel
3. Receiver detects modulation on channel
4. Receiver decodes modulation as message

Communication Model of Attacks
[Belay, Devadas, Emer]

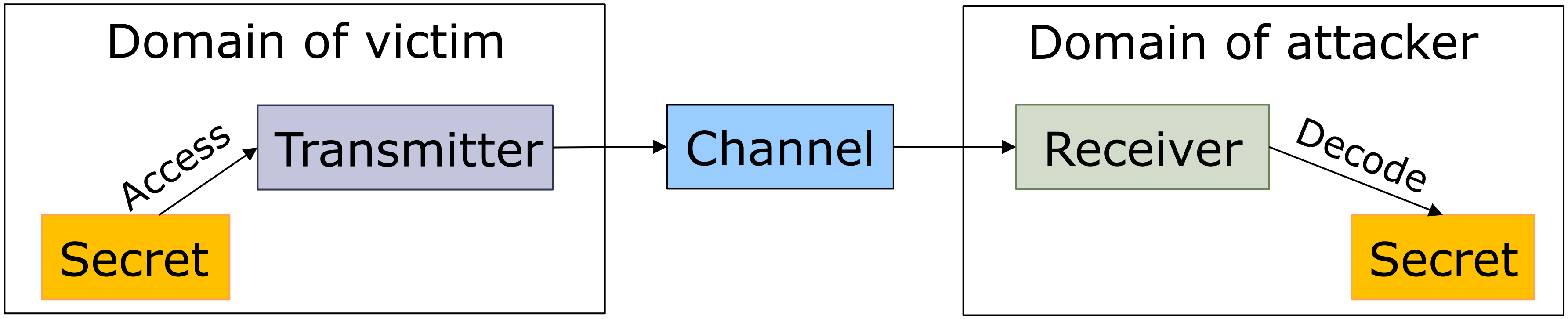


Communication Model of Attacks
[Belay, Devadas, Emer]



- 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".

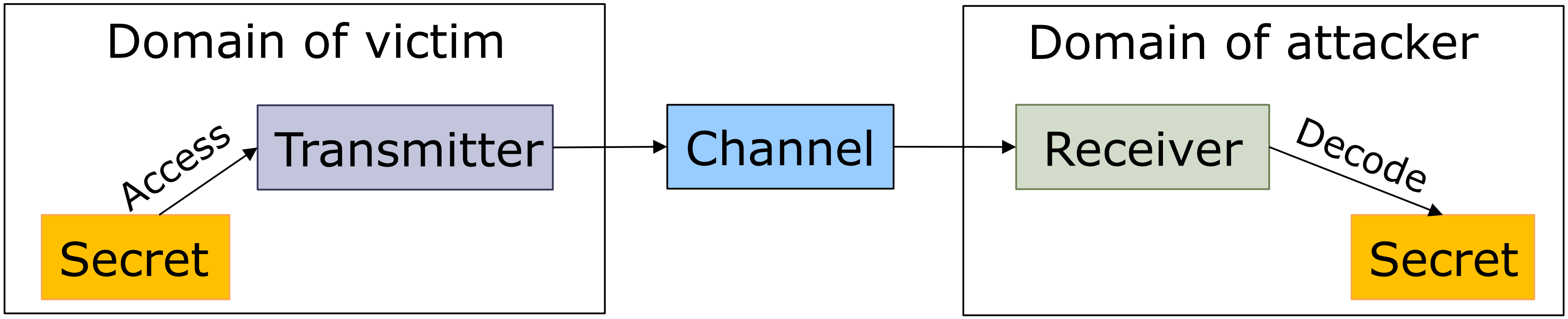
Communication Model of Attacks
[Belay, Devadas, Emer]



- 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".

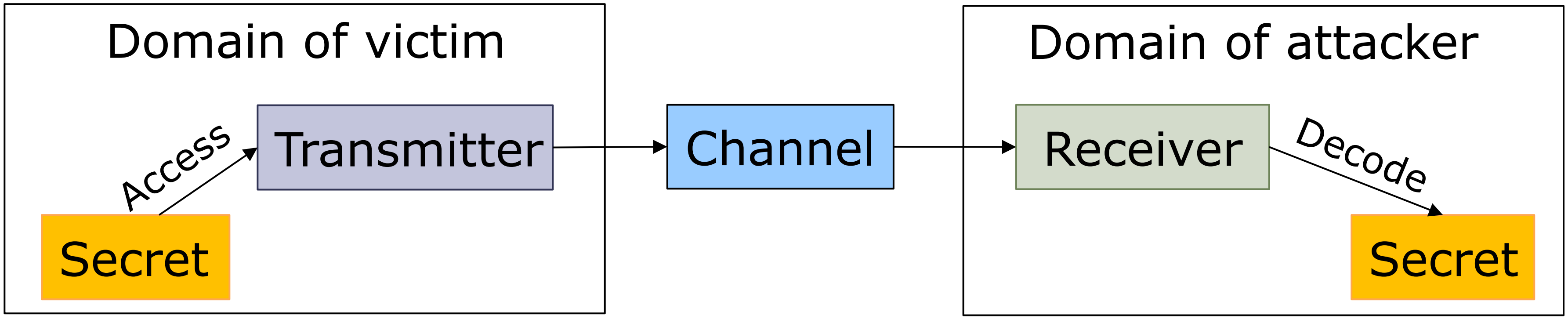
Because channel is not a "direct" communication channel, it is often referred to as a "side channel"

Communication Model of Attacks
[Belay, Devadas, Emer]

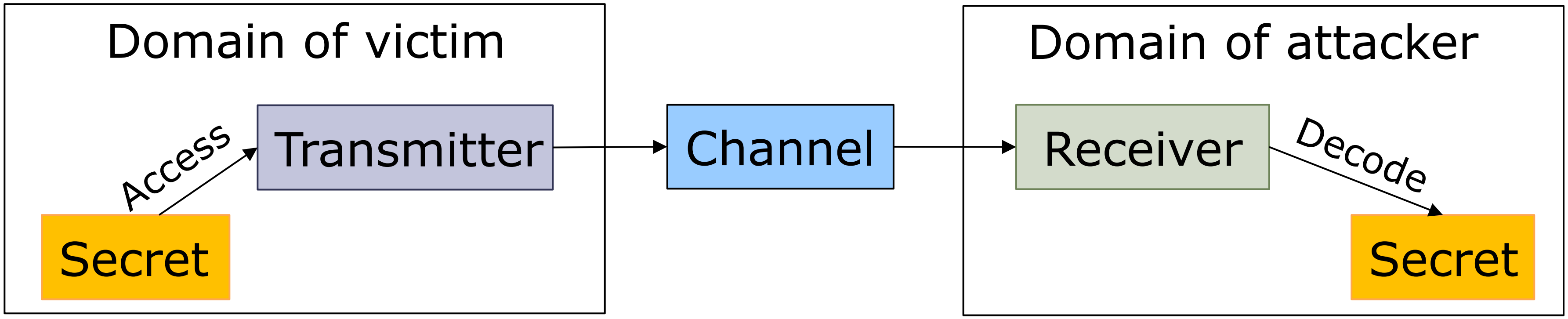


1. Transmitter “accesses” secret
2. Transmitter modulates channel *(microarchitectural state)* with a message based on secret
3. Receiver detects modulation on channel
4. Receiver decodes modulation as a message containing the secret

ATM Acoustic Channels

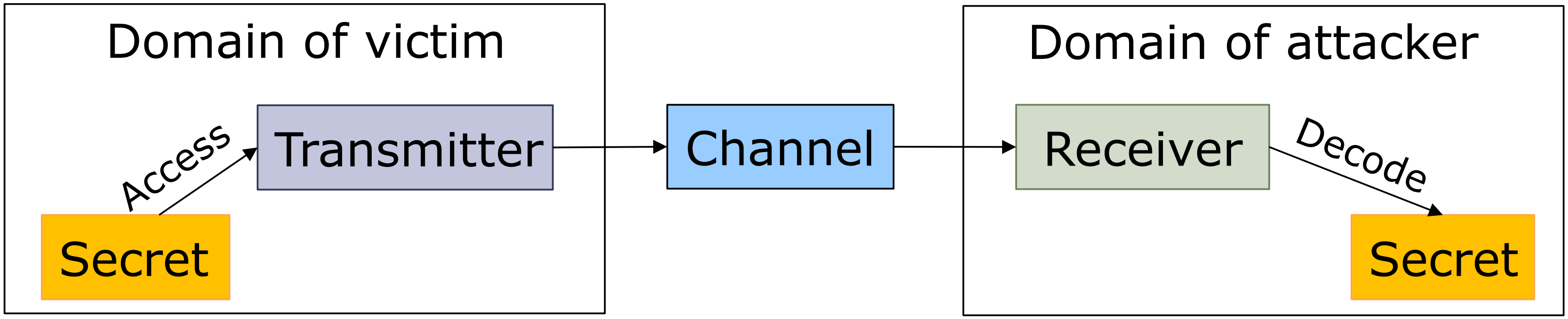


ATM Acoustic Channels



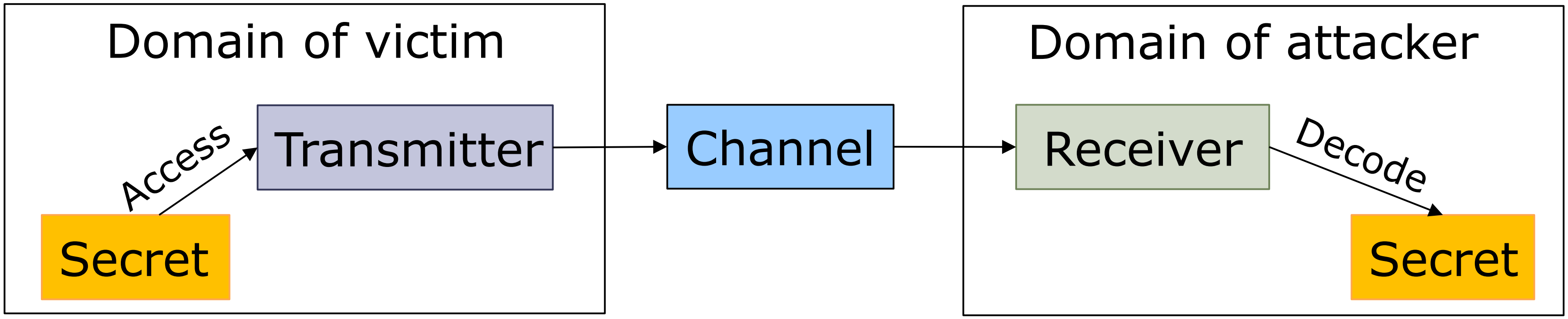
- Secret:
- Transmitter:
- Channel:
- Modulation:
- Receiver:
- Decoders:

ATM Acoustic Channels



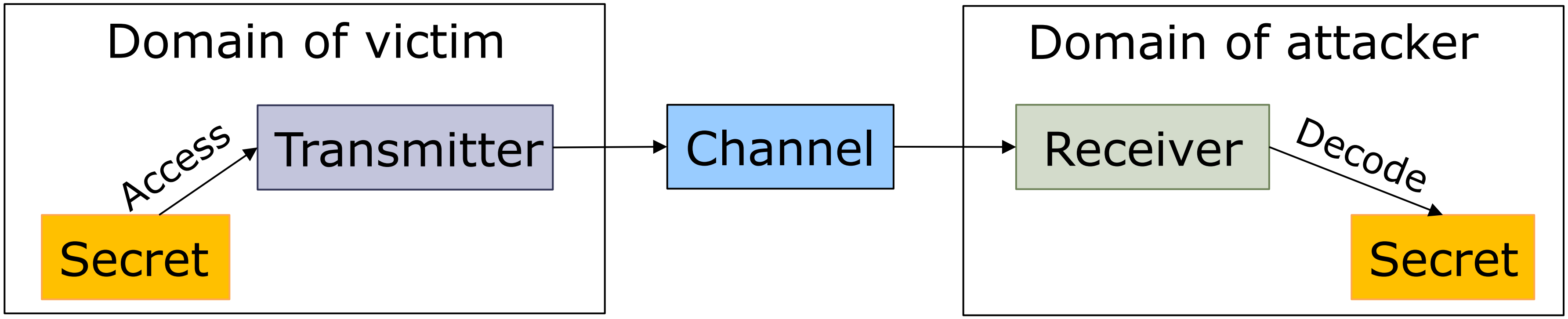
- Secret: Pin
- Transmitter:
- Channel:
- Modulation:
- Receiver:
- Decoders:

ATM Acoustic Channels



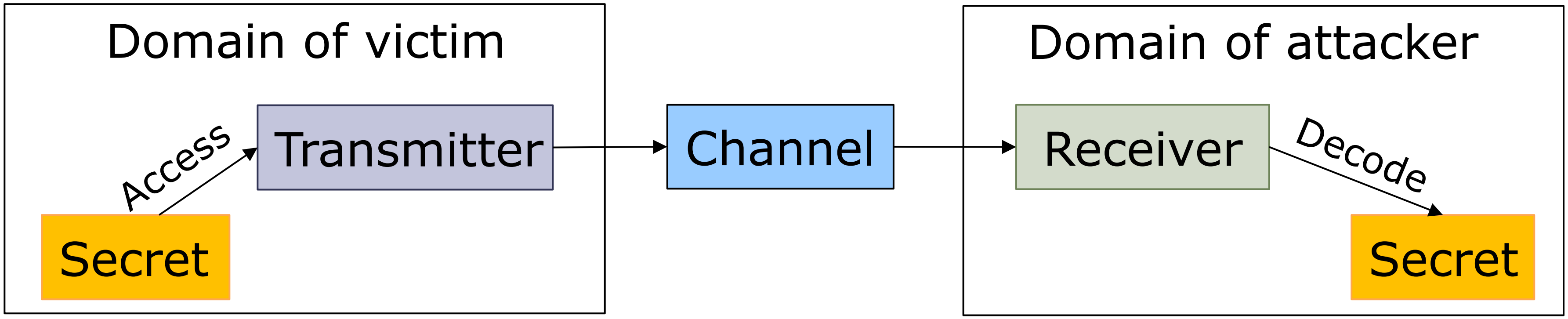
- Secret: Pin
- Transmitter: Keypad
- Channel:
- Modulation:
- Receiver:
- Decoders:

ATM Acoustic Channels



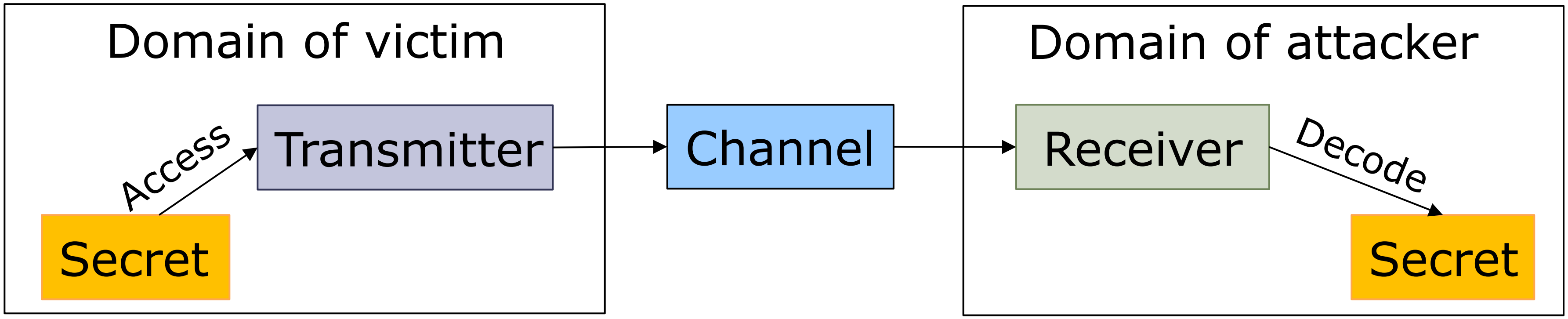
- Secret: Pin
- Transmitter: Keypad
- Channel: Air
- Modulation:
- Receiver:
- Decoders:

ATM Acoustic Channels



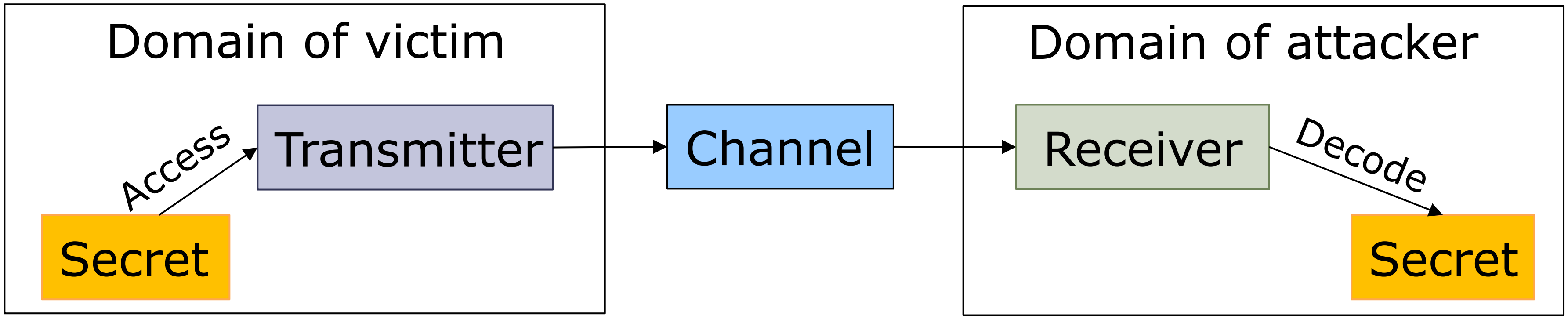
- Secret: Pin
- Transmitter: Keypad
- Channel: Air
- Modulation: Acoustic waves
- Receiver:
- Decoders:

ATM Acoustic Channels



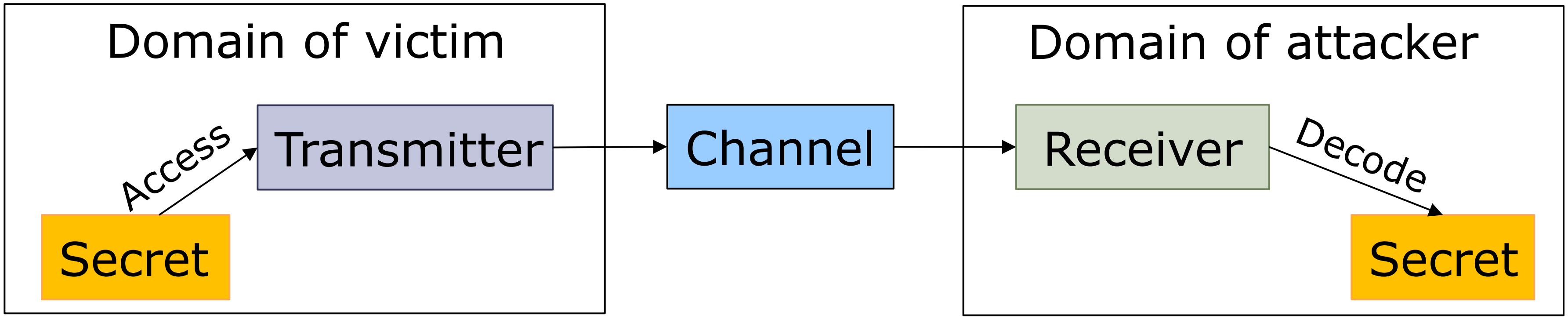
- Secret: Pin
- Transmitter: Keypad
- Channel: Air
- Modulation: Acoustic waves
- Receiver: Cheap Microphone
- Decoders:

ATM Acoustic Channels



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

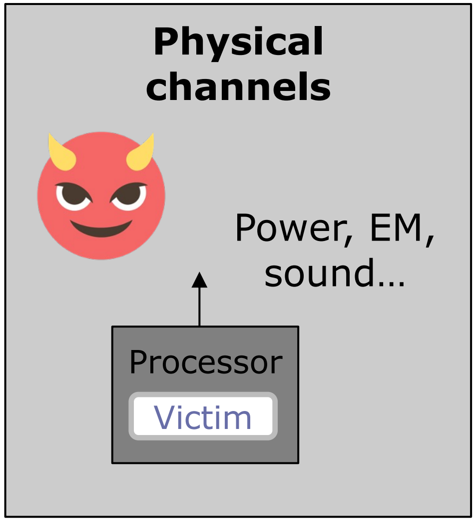
ATM Acoustic Channels



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

Physical vs Timing vs uArch Channel

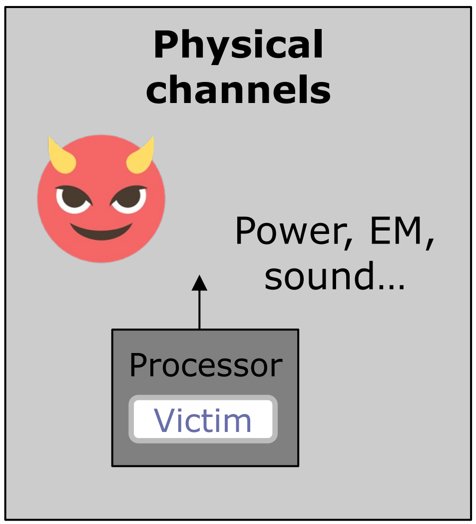
- Types of channels



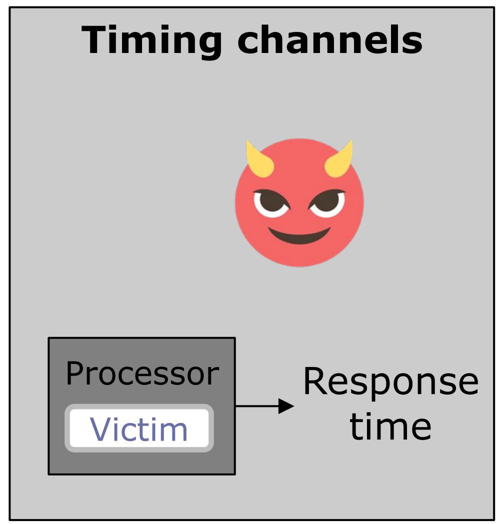
Attacker requires measurement equipment → physical access

Physical vs Timing vs uArch Channel

- Types of channels



Attacker requires measurement equipment → physical access



Attacker may be remote (e.g., over an internet connection)

Timing Channel Example

```
def check(input):

  size = len(passwd); //passwd contains 8 digits
  for i in range(0,size):
    if (input [i] != password[i]):
      return ("error");

  return (“success”)
```

Blind guess needs to maximally try: 10^8

Timing Channel Example

```
def check(input):

  size = len(passwd); //passwd contains 8 digits
  for i in range(0,size):
    if (input [i] != password[i]):
      return ("error");

  return (“success”)
```

Blind guess needs to maximally try: 10^8

Can we do better to reduce the number of trials?

Timing Channel Example

```
def check(input):

  size = len(passwd); //passwd contains 8 digits
  for i in range(0,size):
    if (input [i] != password[i]):
      return ("error");

  return (“success”)
```

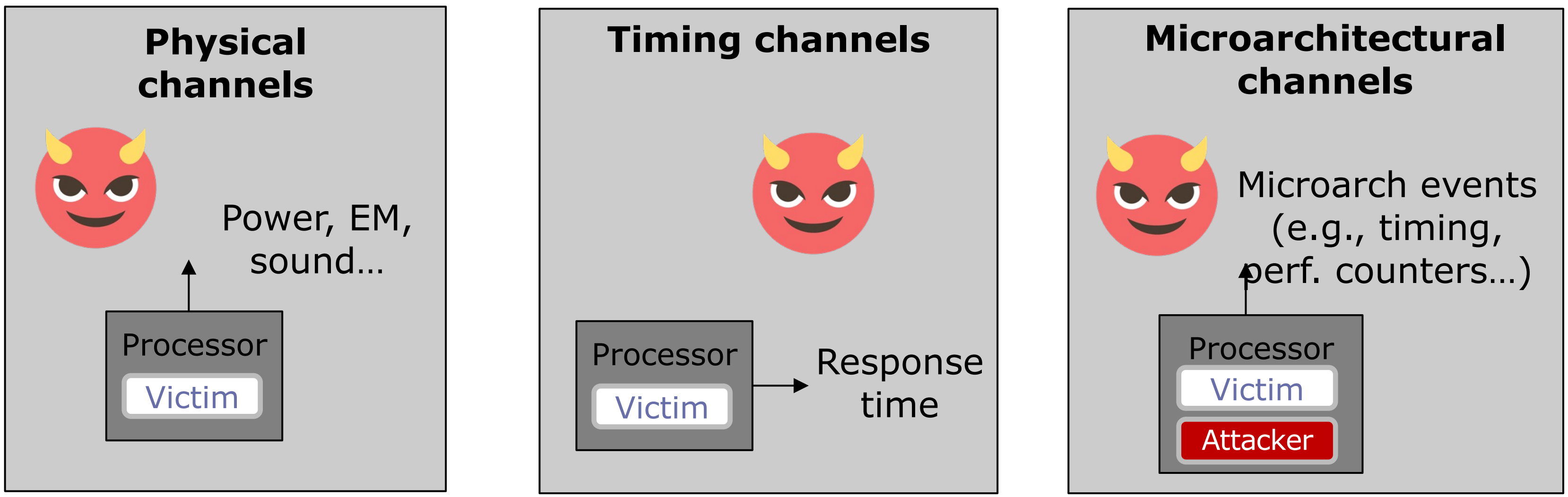
Blind guess needs to maximally try: 10^8

Can we do better to reduce the number of trials?

The execution time is dependent on how many characters match between the input and the correct password. Attacker can brute-force each character. Maximally try 10*8 times.

Physical vs Timing vs uArch Channel

- Types of channels



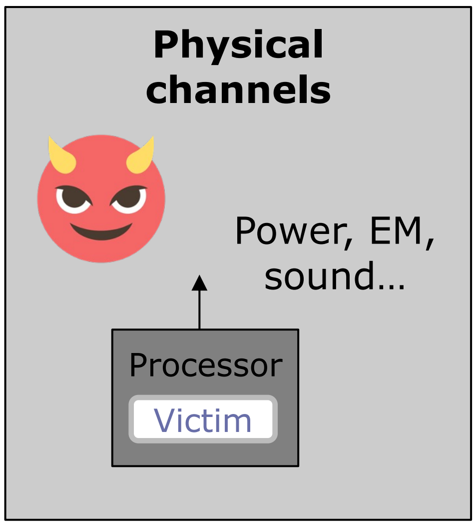
Attacker requires measurement equipment → physical access

Attacker may be remote (e.g., over an internet connection)

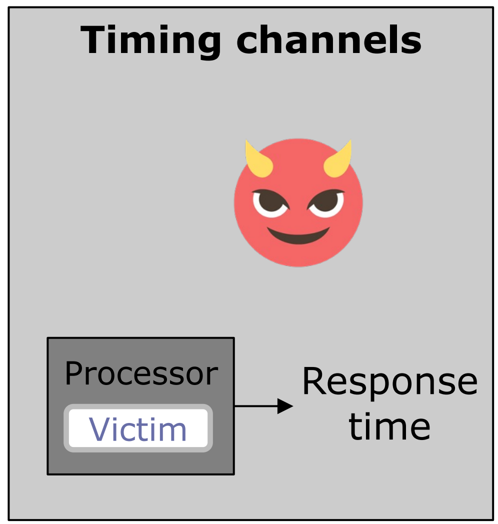
Attacker may be remote, or be co-located

Physical vs Timing vs uArch Channel

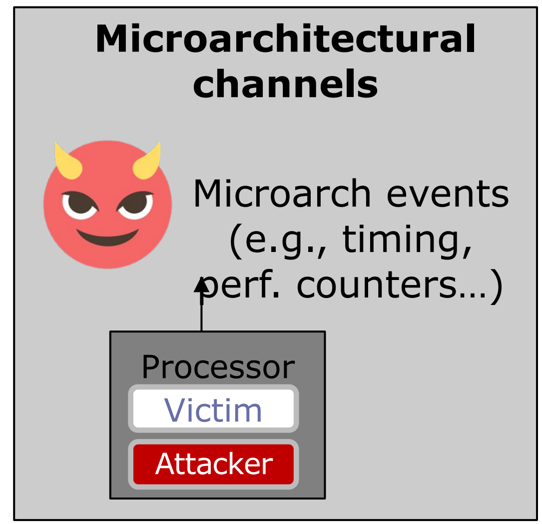
- Types of channels



Attacker requires measurement equipment → physical access



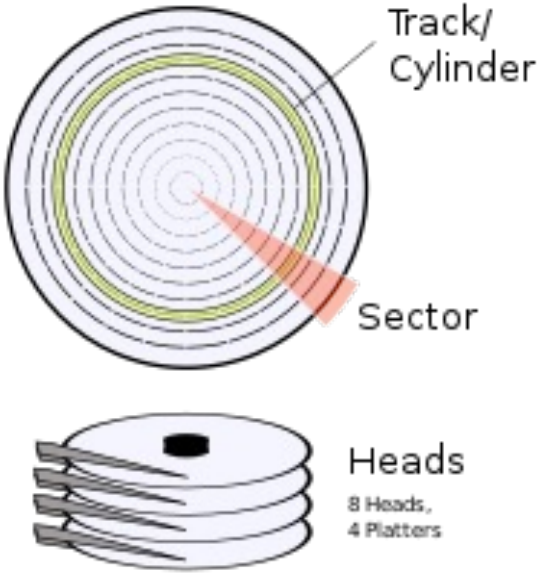
Attacker may be remote (e.g., over an internet connection)



Attacker may be remote, or be co-located

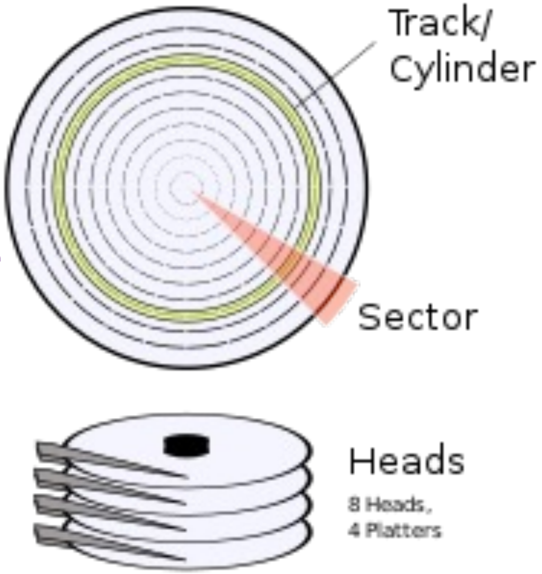
Side Channel Attacks in 1977

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



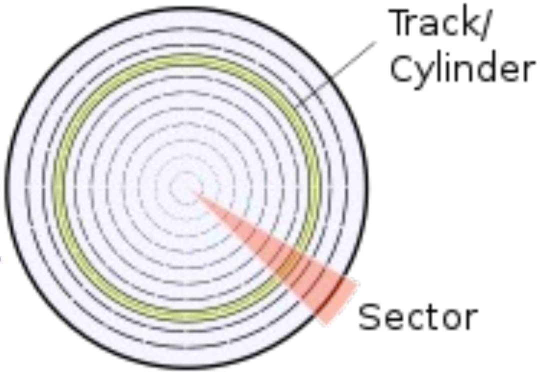
Side Channel Attacks in 1977

- A side channel due to disk arm optimization
  - Enqueues requests by ascending cylinder number and dequeues (executes) them by the "elevator algorithm."
- Example:
  1. Receiver issues a request to 55
  2. Sender issues a request to either 53 or 57
  3. Receiver then issues requests to both 52 and 58



Side Channel Attacks in 1977

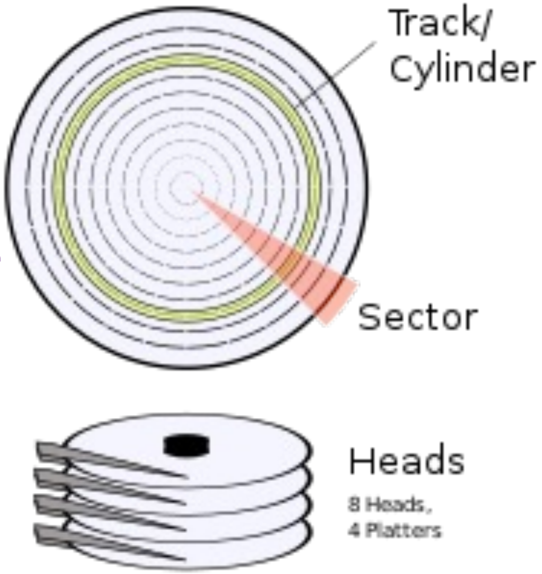
- A side channel due to disk arm optimization
  - Enqueues requests by ascending cylinder number and dequeues (executes) them by the "elevator algorithm."



- Example:
  1. Receiver issues a request to 55
  2. Sender issues a request to either 53 or 57
  3. Receiver then issues requests to both 52 and 58

Q: If the Receiver receives data for 52 first, can we guess what did Sender issue before?

Side Channel Attacks in 1977



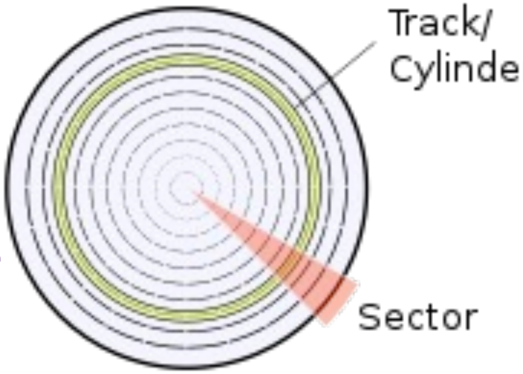
- A side channel due to disk arm optimization
  - Enqueues requests by ascending cylinder number and dequeues (executes) them by the "elevator algorithm."
- Example:
  1. Receiver issues a request to 55
  2. Sender issues a request to either 53 or 57
  3. Receiver then issues requests to both 52 and 58

Q: If the Receiver receives data for 52 first, can we guess what did Sender 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:
  1. Receiver issues a request to 55
  2. Sender issues a request to either 53 or 57
  3. Receiver then issues requests to both 52 and 58

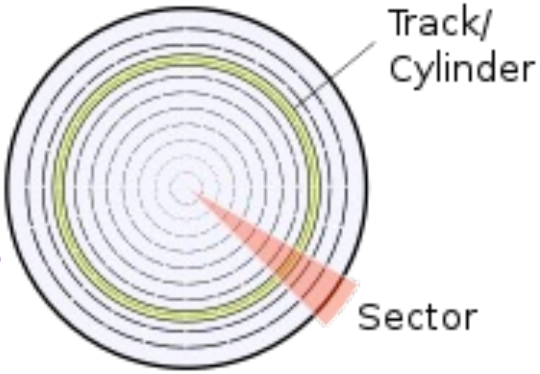
Q: If the Receiver receives data for 52 first, can we guess what did Sender issue before?

53

Q: If we remove step 1, can the attack still work?

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."



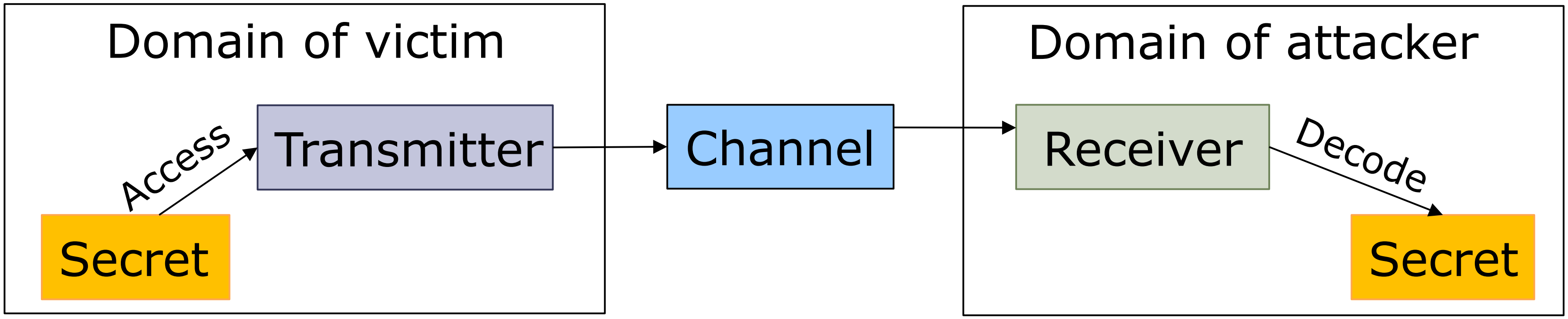
- Example:
  1. Receiver issues a request to 55
  2. Sender issues a request to either 53 or 57
  3. Receiver then issues requests to both 52 and 58

Q: If the Receiver receives data for 52 first, can we guess what did Sender issue before? 53

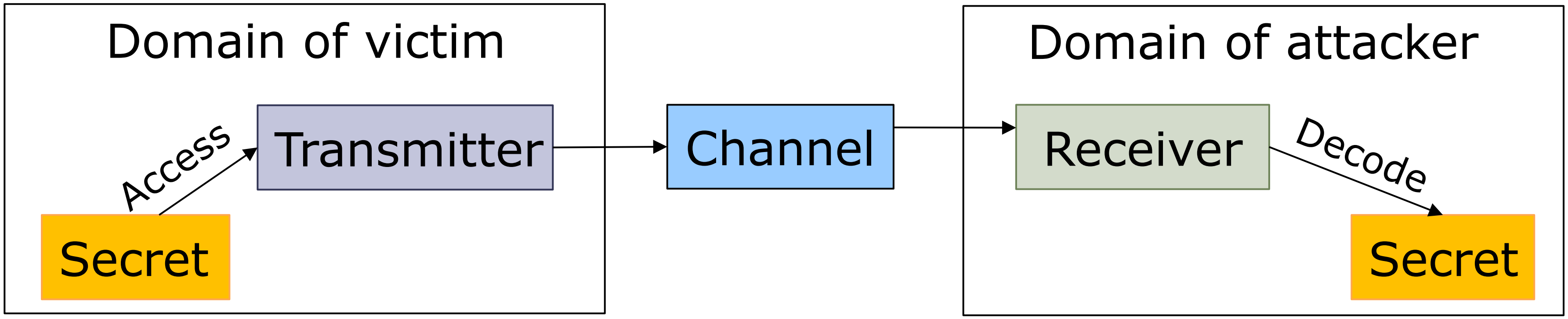
Q: If we remove step 1, can the attack still work?

Note this requires an "active" receiver that preconditions the channel

Communication w/ Active Receiver

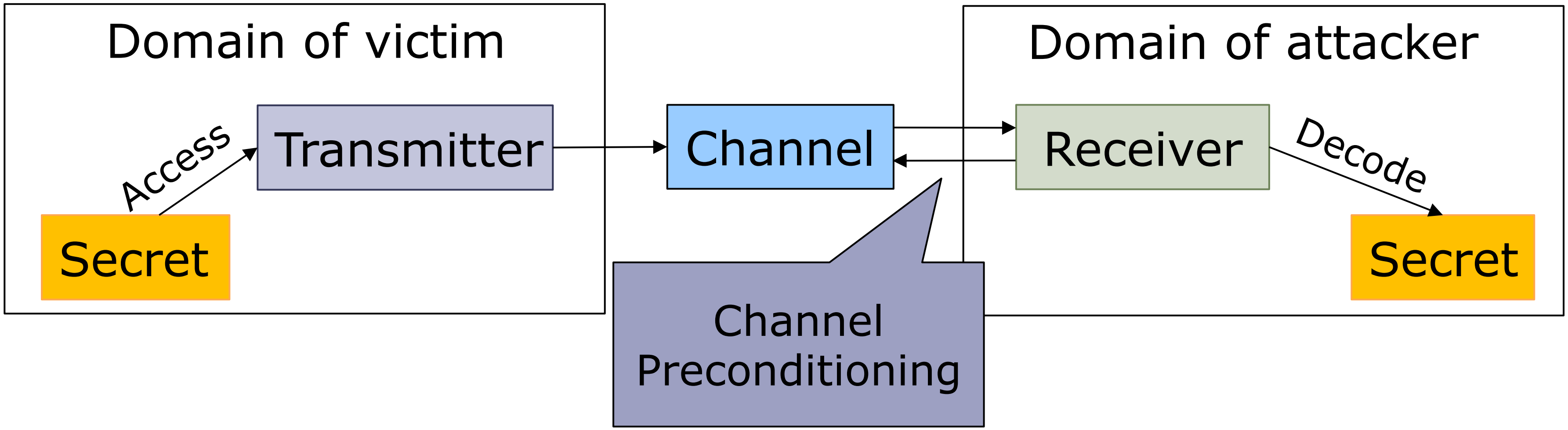


Communication w/ Active Receiver



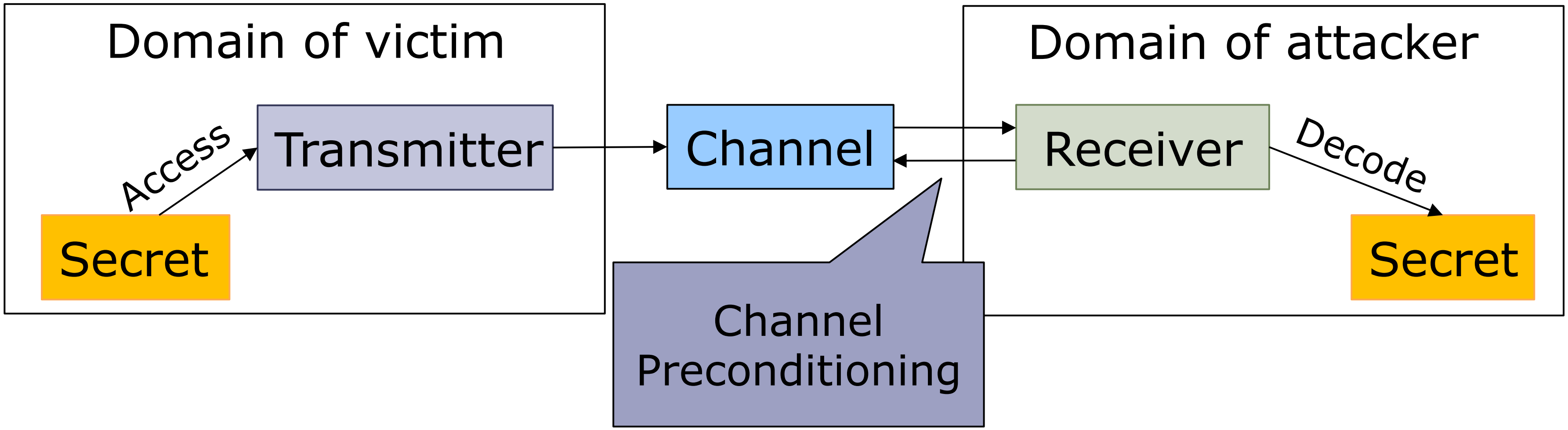
- An active receiver may need to “precondition” the channel to prepare for detecting modulation

Communication w/ Active Receiver



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

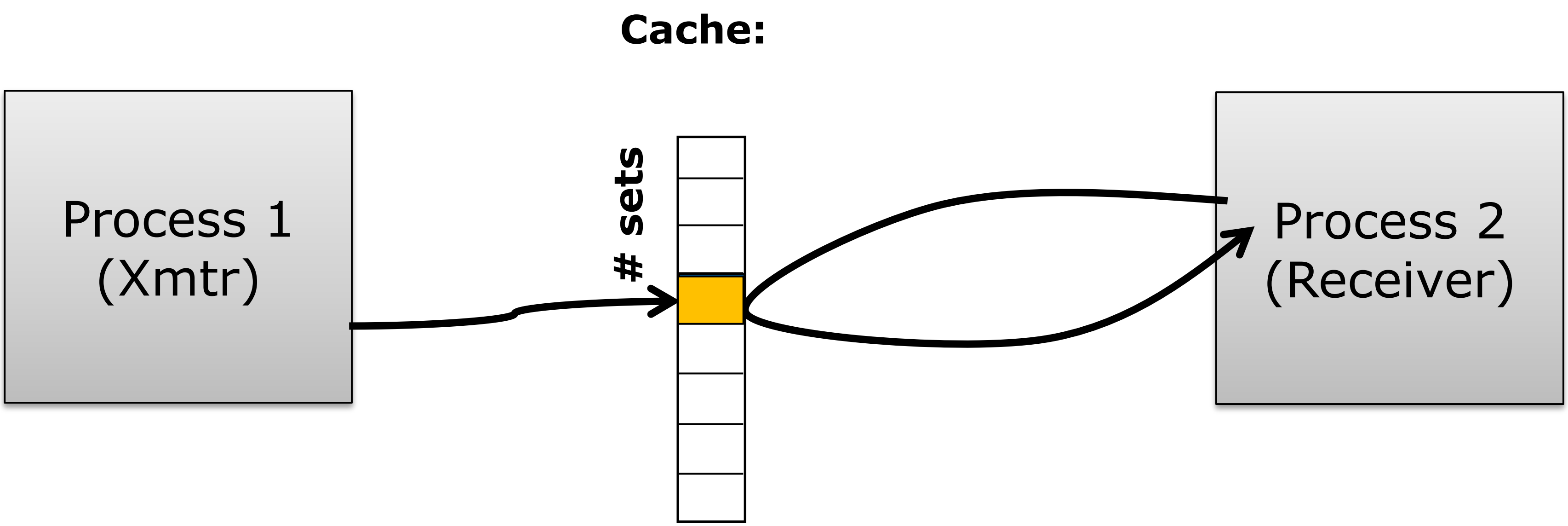
Communication w/ Active Receiver



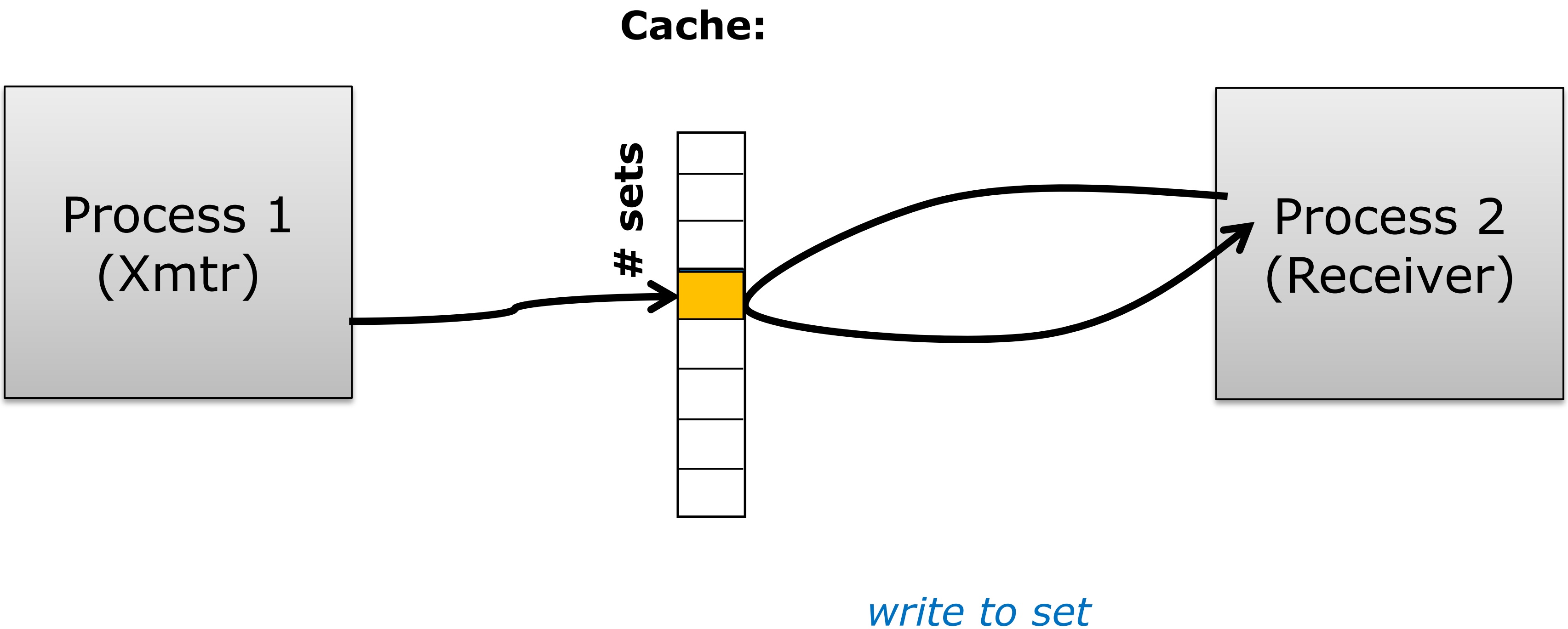
- An active receiver may need to “precondition” the channel to prepare for detecting modulation

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

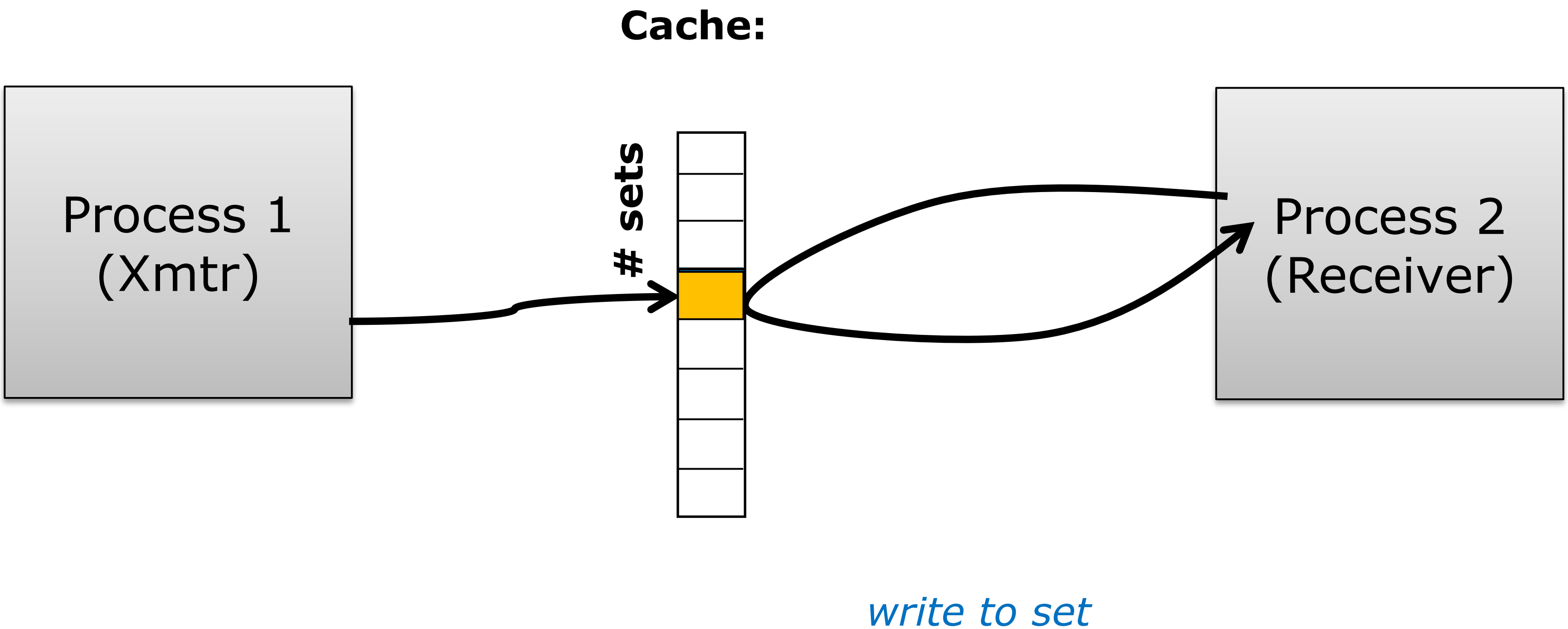
A Cache-based Channel



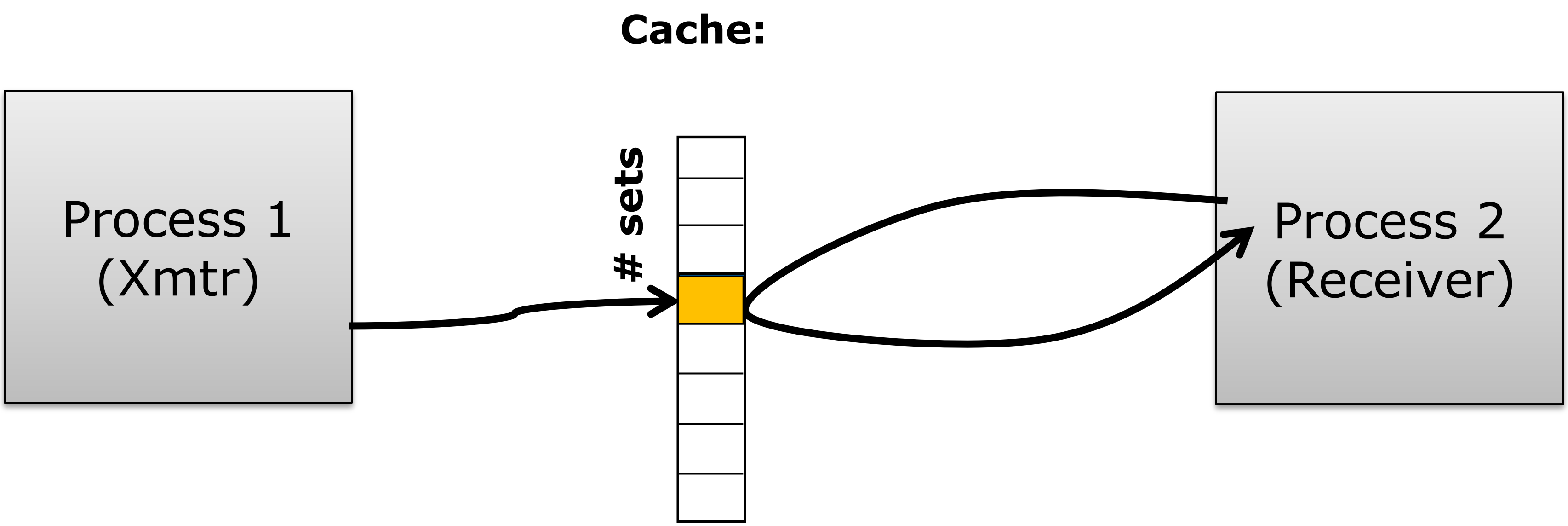
A Cache-based Channel



A Cache-based Channel



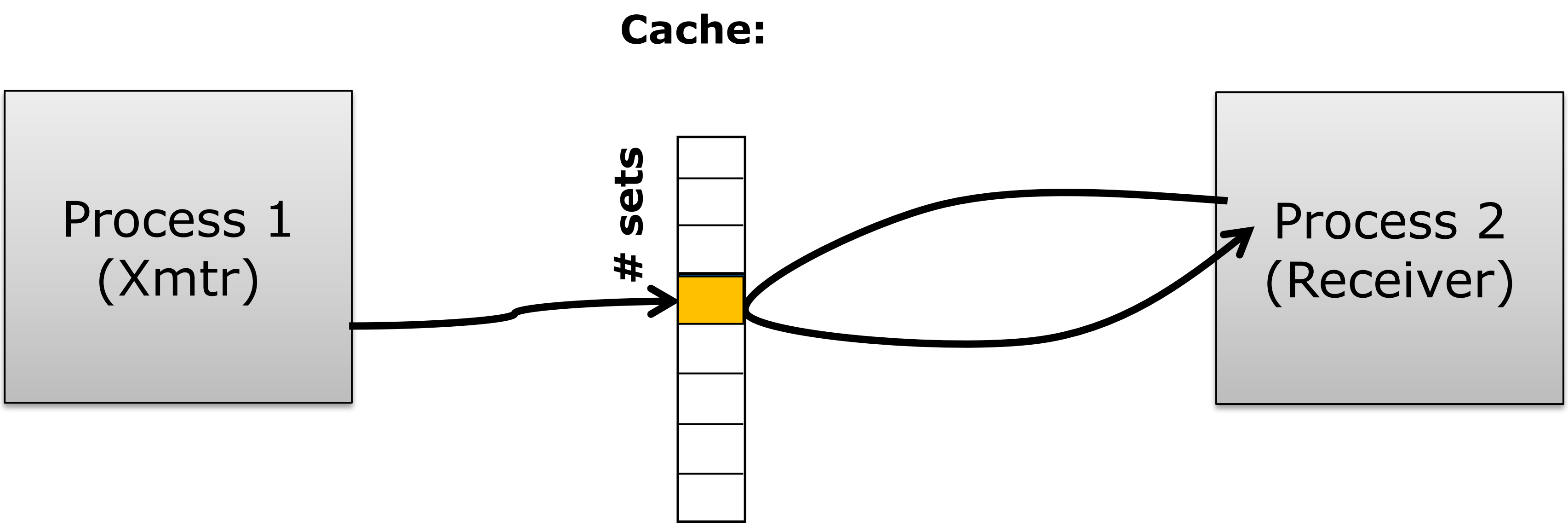
A Cache-based Channel



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

write to set

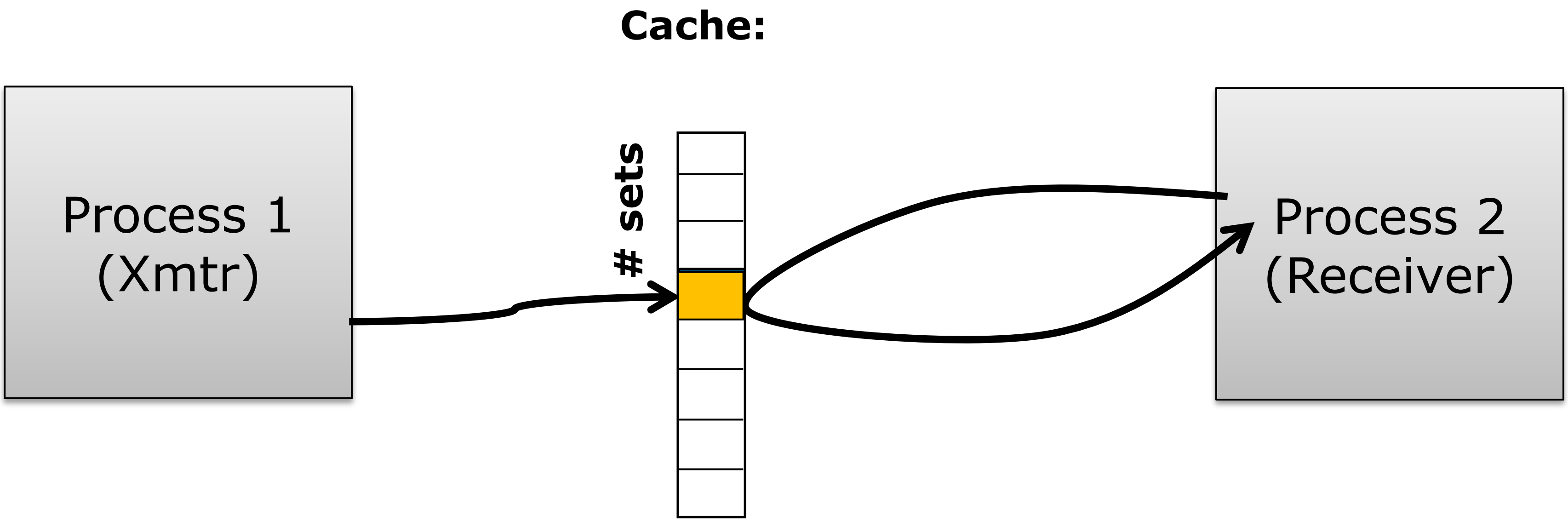
A Cache-based Channel



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

write to set

A Cache-based Channel

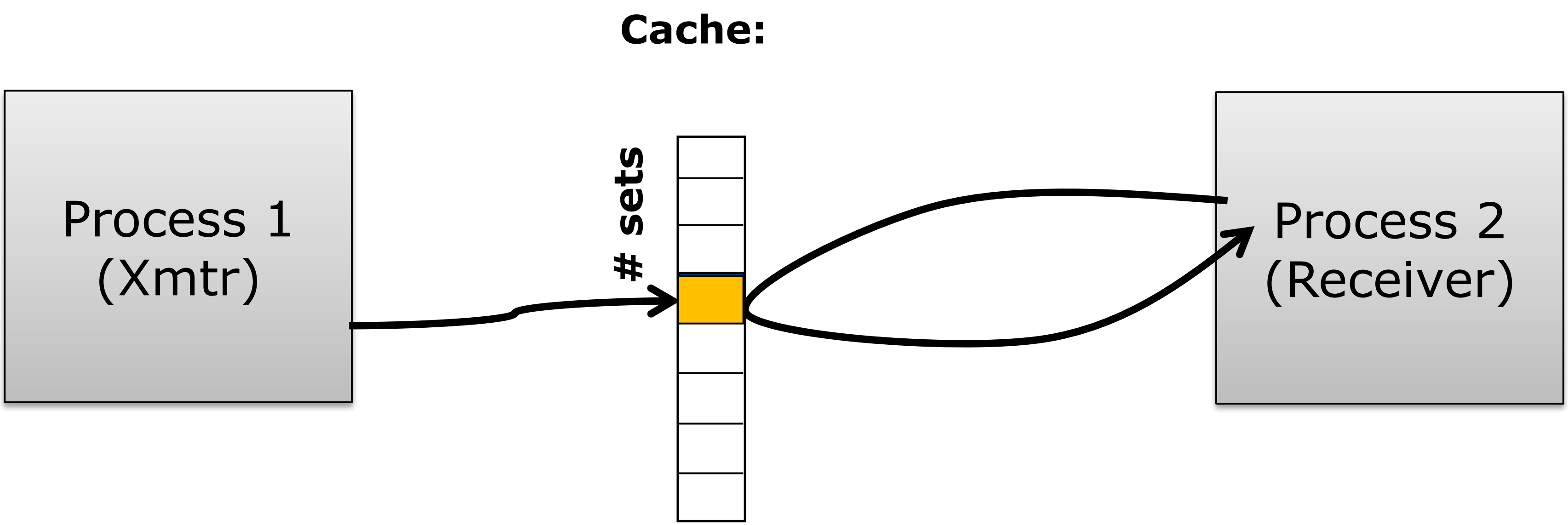


if (**send ‘0’**)
   idle
else
   write to a set

write to set

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

A Cache-based Channel



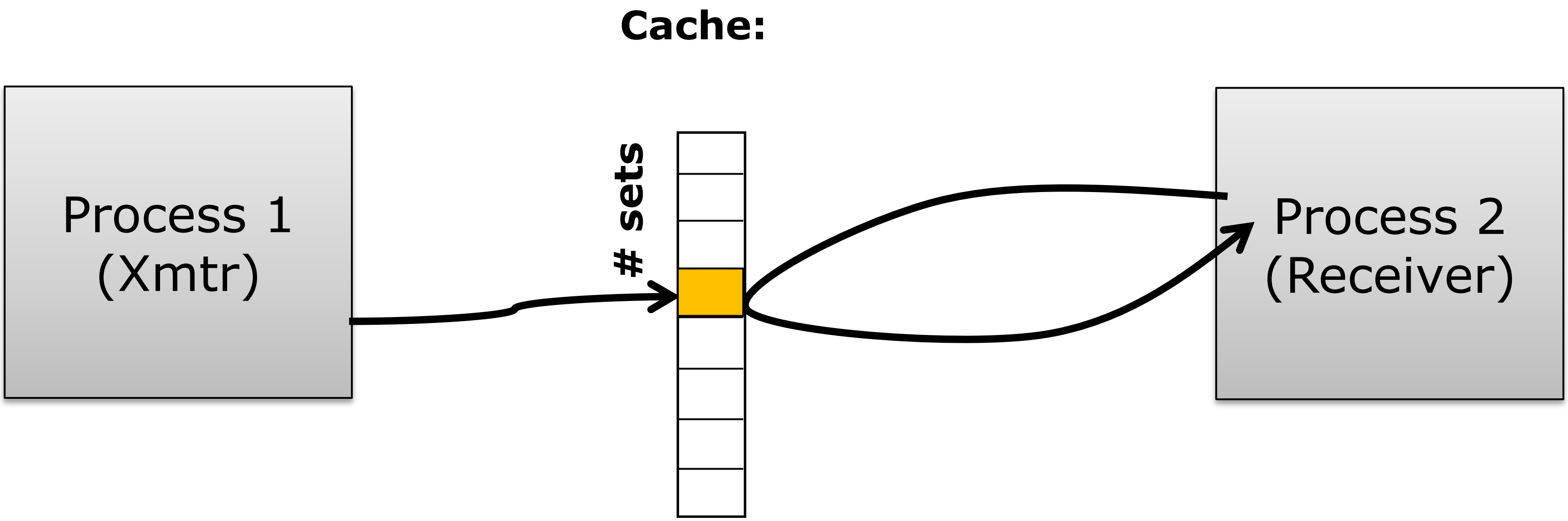
if (**send '0'**)
  idle
else
  write to a set

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



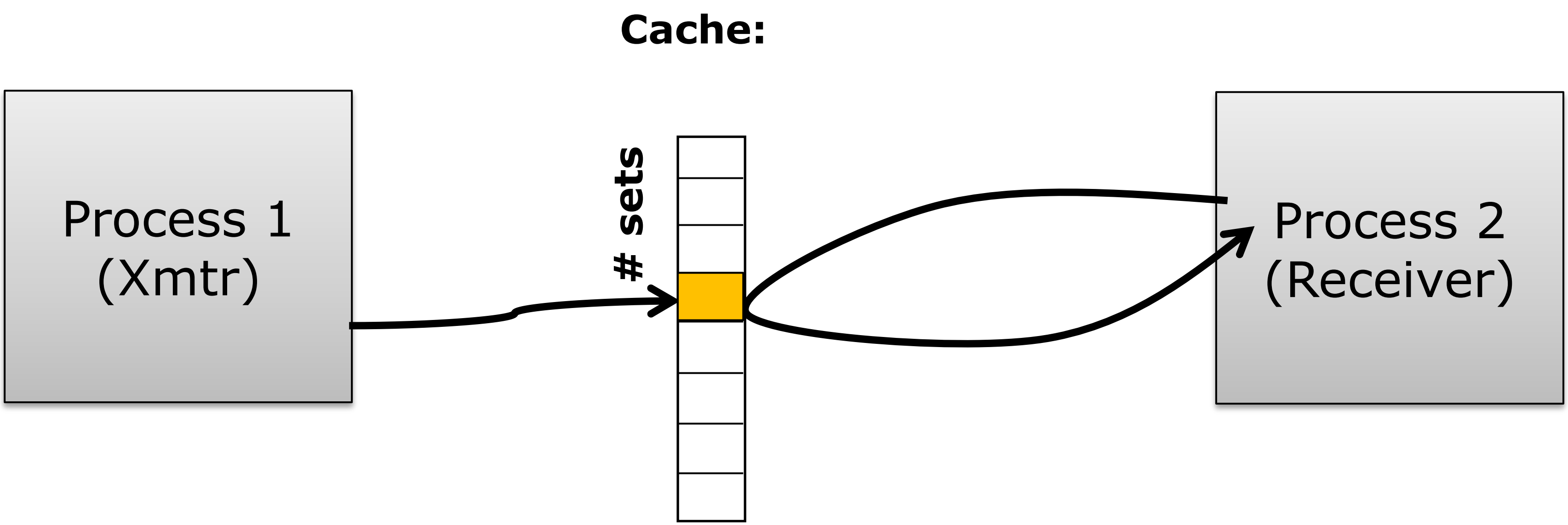
if (**send '0'**)
  idle
else
  write to a set

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



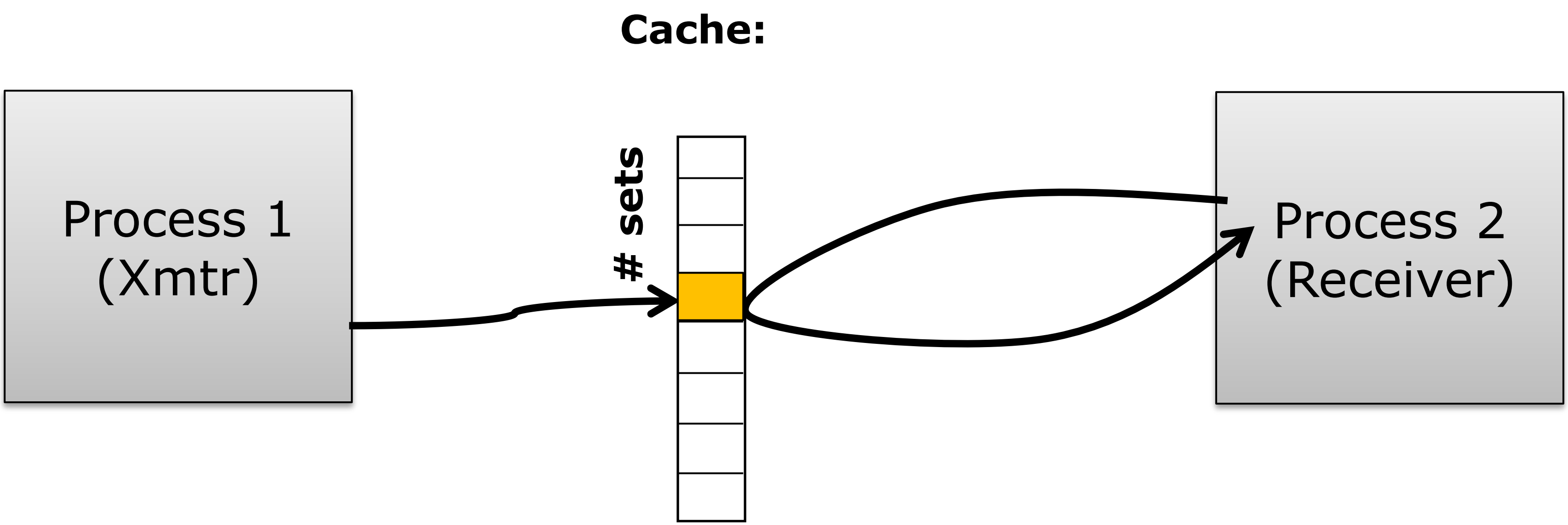
if (**send ‘0’**)
  idle
else
  write to a set

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



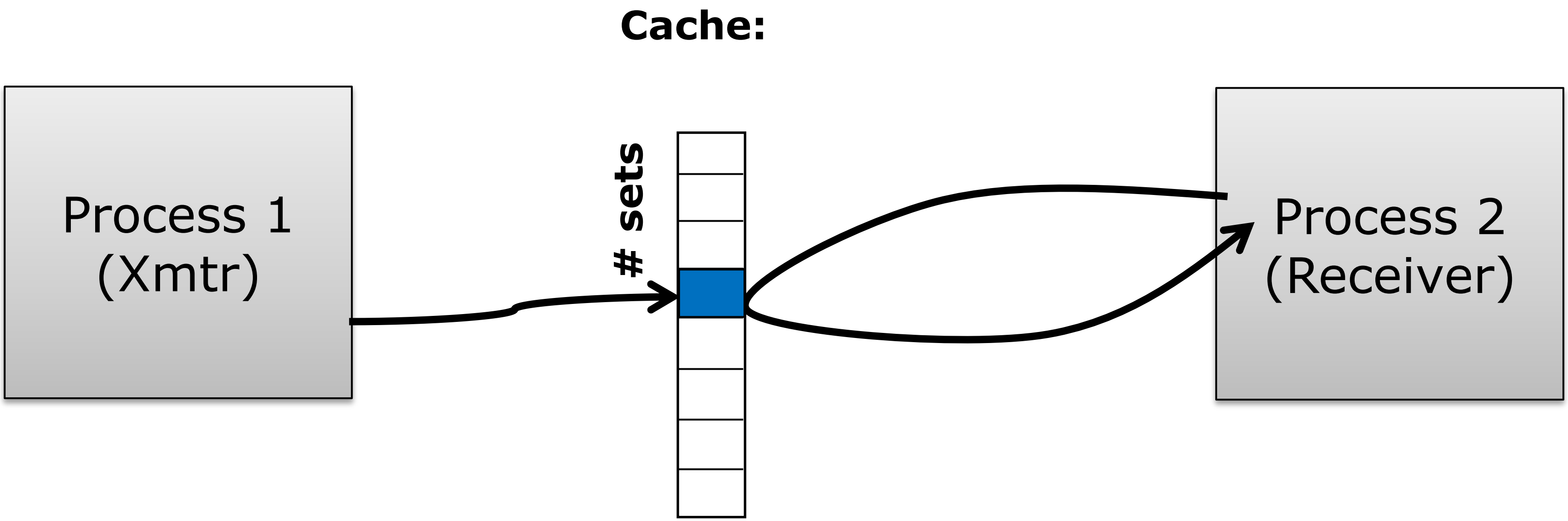
if (**send '0'**)
    idle
else
    write to a set

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



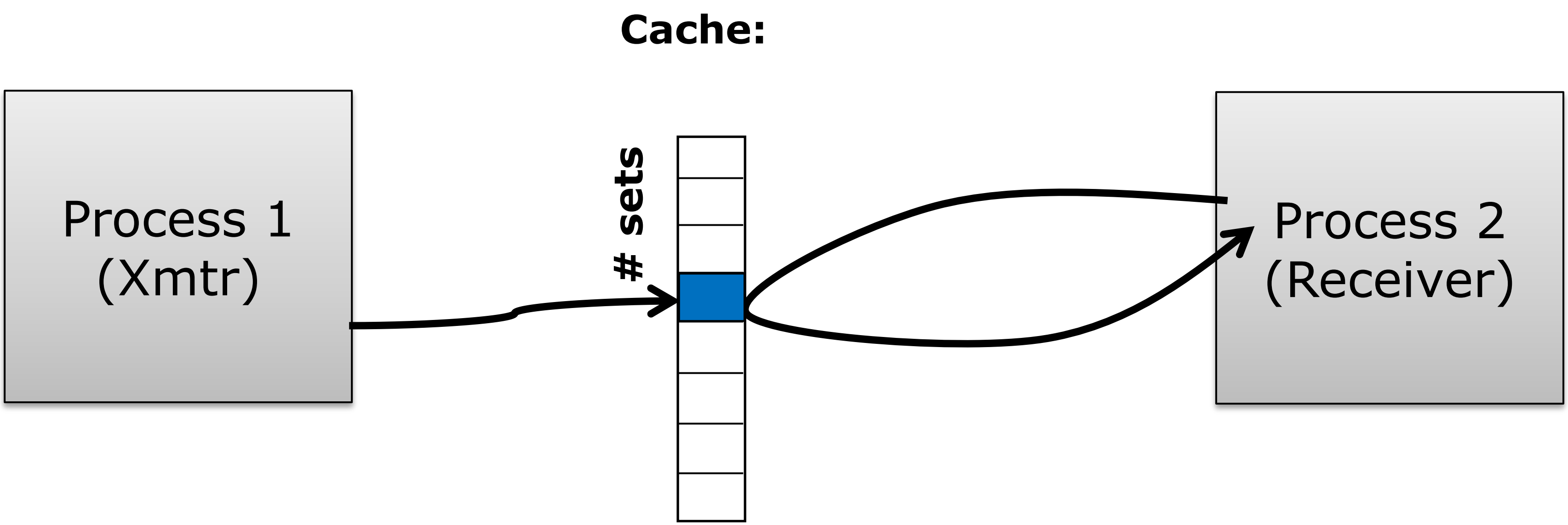
if (**send '0'**)
    idle
else
    write to a set

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



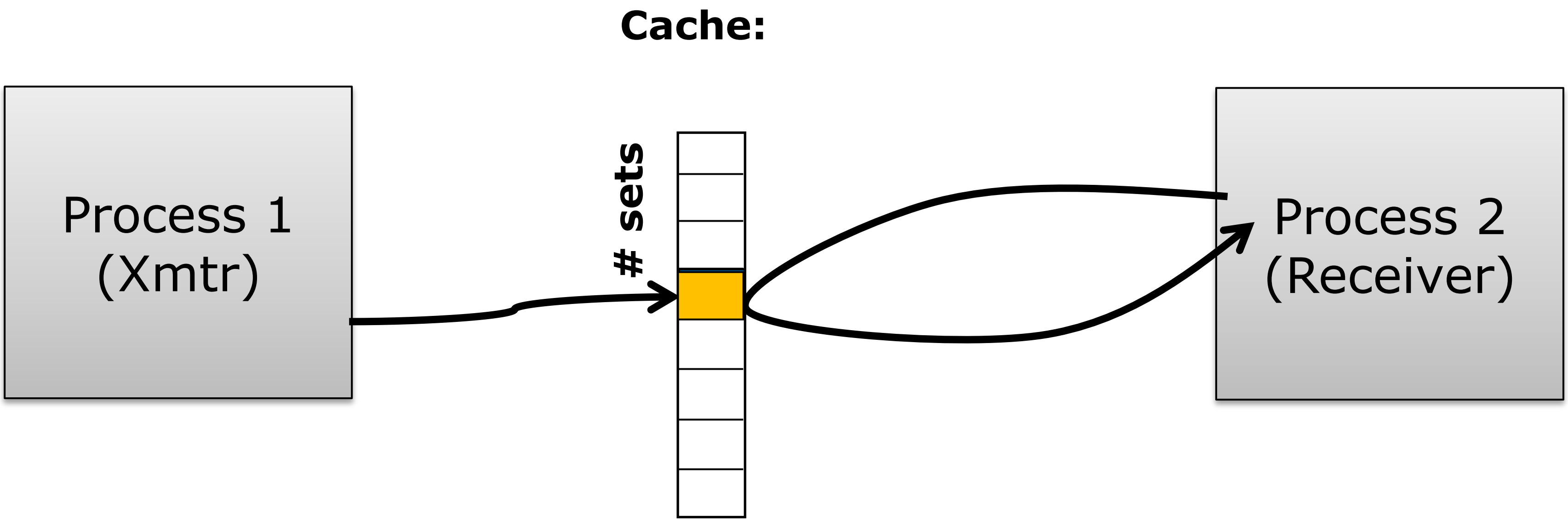
if (**send '0'**)
  idle
else
  write to a set

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’

Transmitter in RSA [Percival 2005]

- Square-and-multiply based exponentiation

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

Transmitter in RSA [Percival 2005]

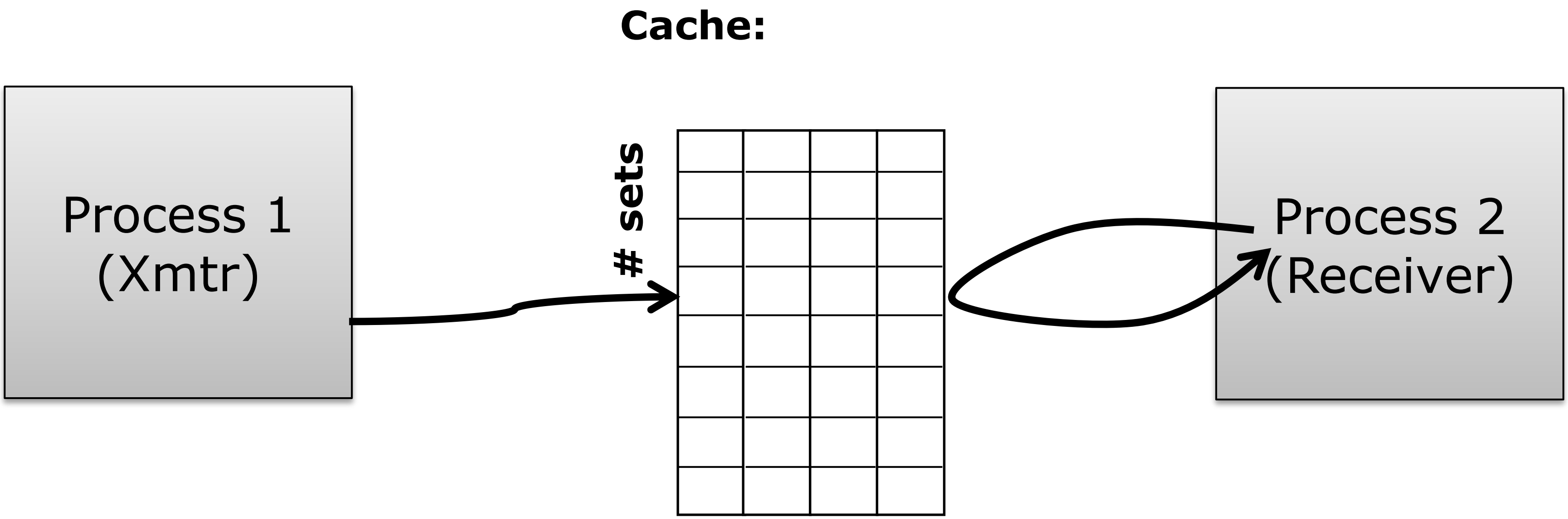
- Square-and-multiply based exponentiation

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

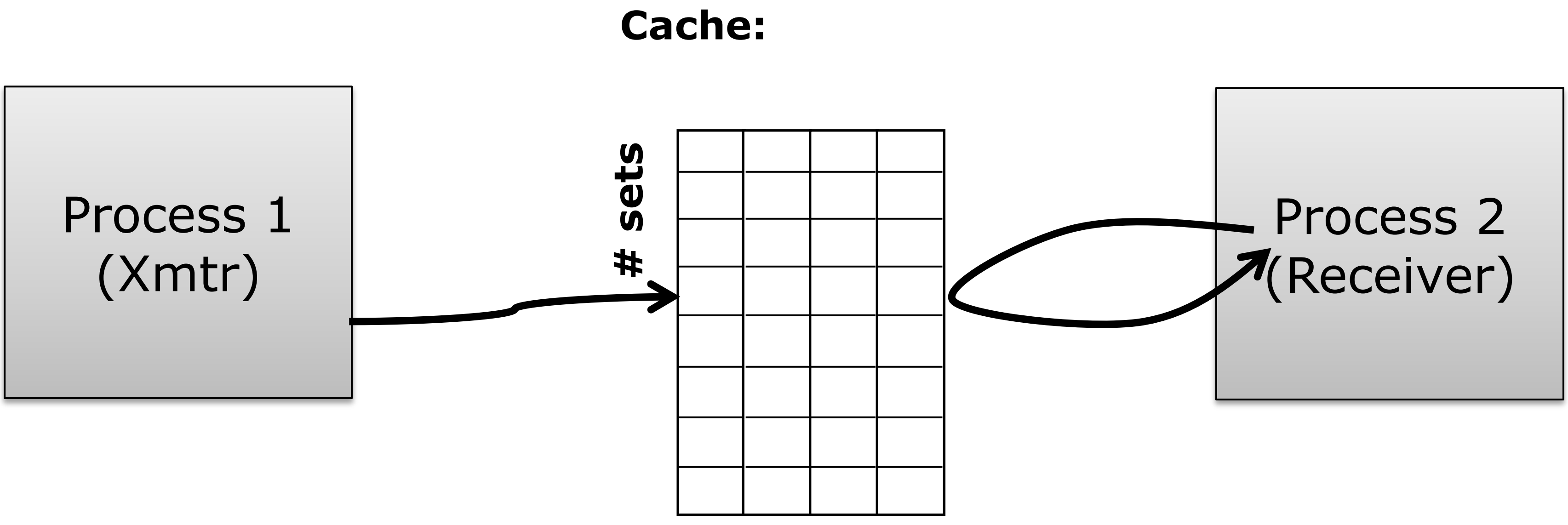
$r = 1$
for $i = n-1$ *down to* 0 do
$r = sqrt(r)$
$r = mod(r,m)$
if e_i **== 1 then**
***r* = *mul*(*r*,b)**
$r = mod(r,m)$
end
end
return r

Secret-dependent memory access → transmitter

A Multi-way Cache-based Channel

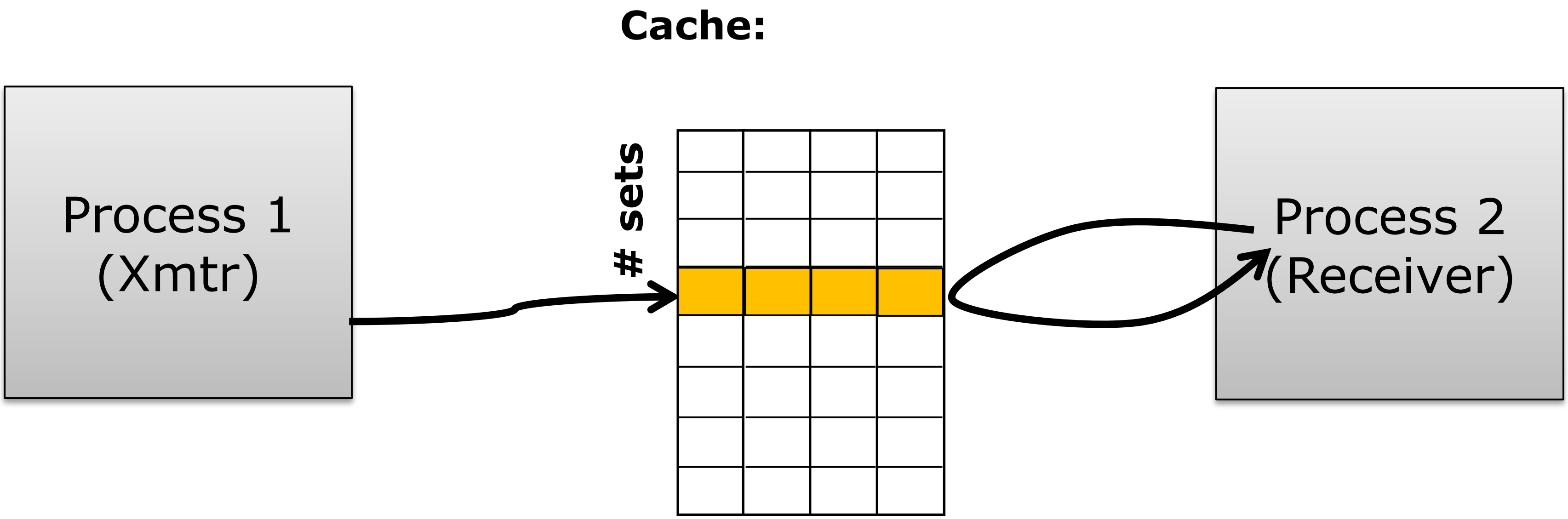


A Multi-way Cache-based Channel



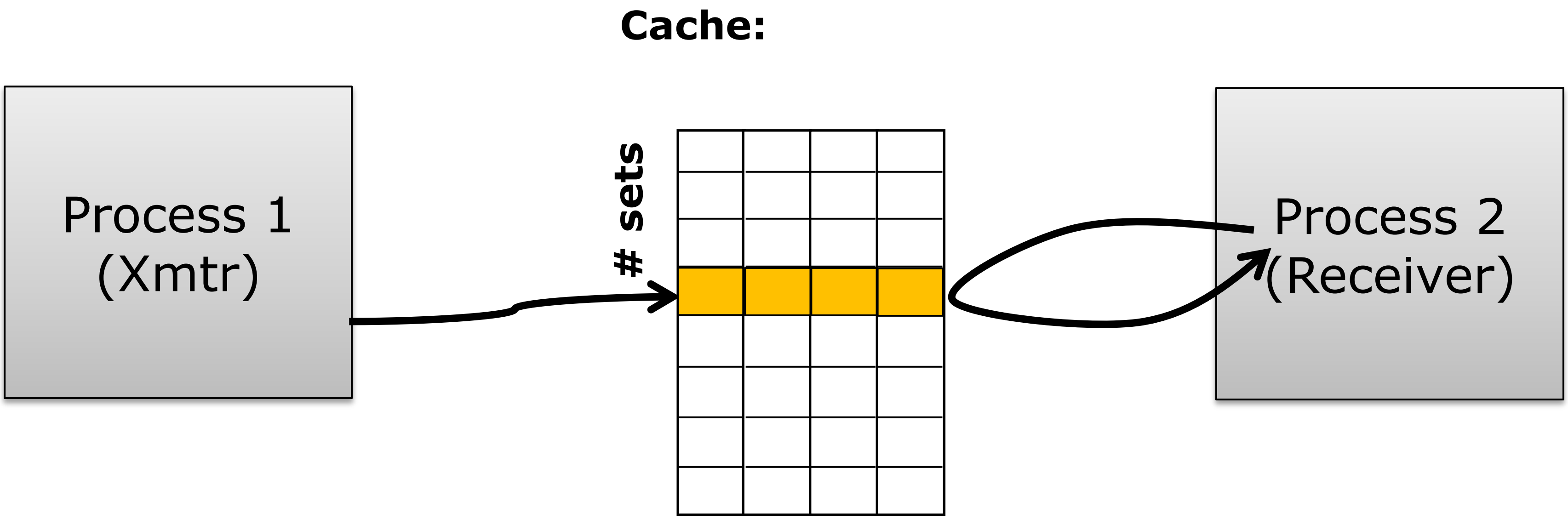
fill a set

A Multi-way Cache-based Channel



fill a set

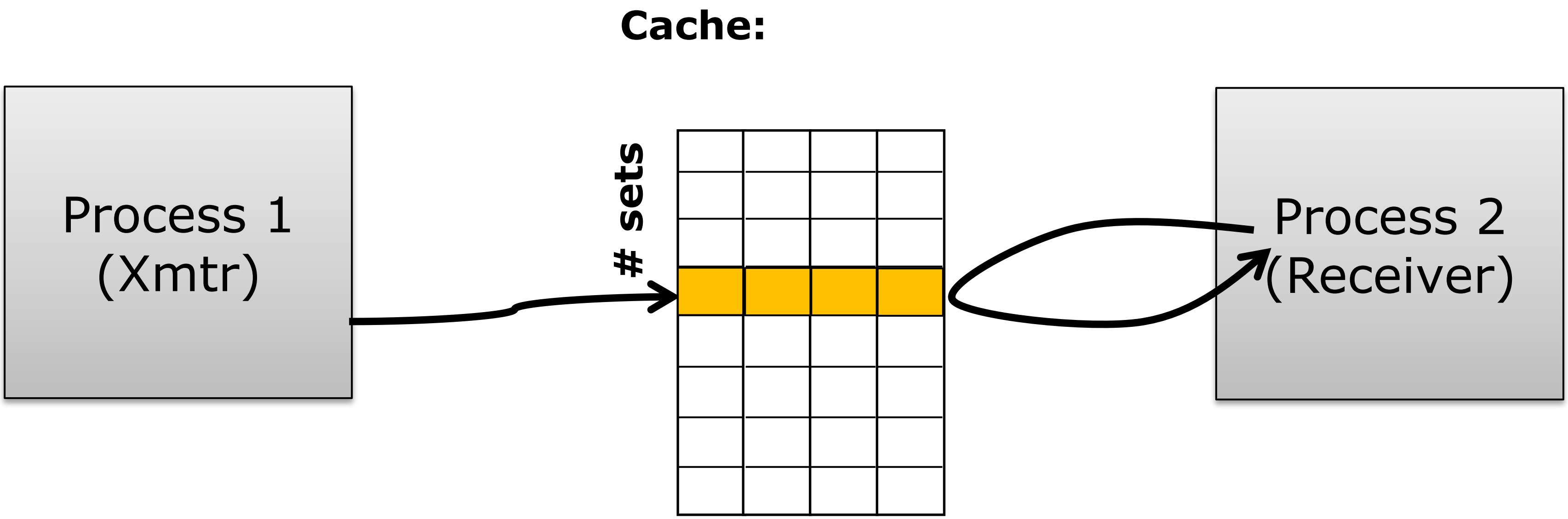
A Multi-way Cache-based Channel



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

fill a set

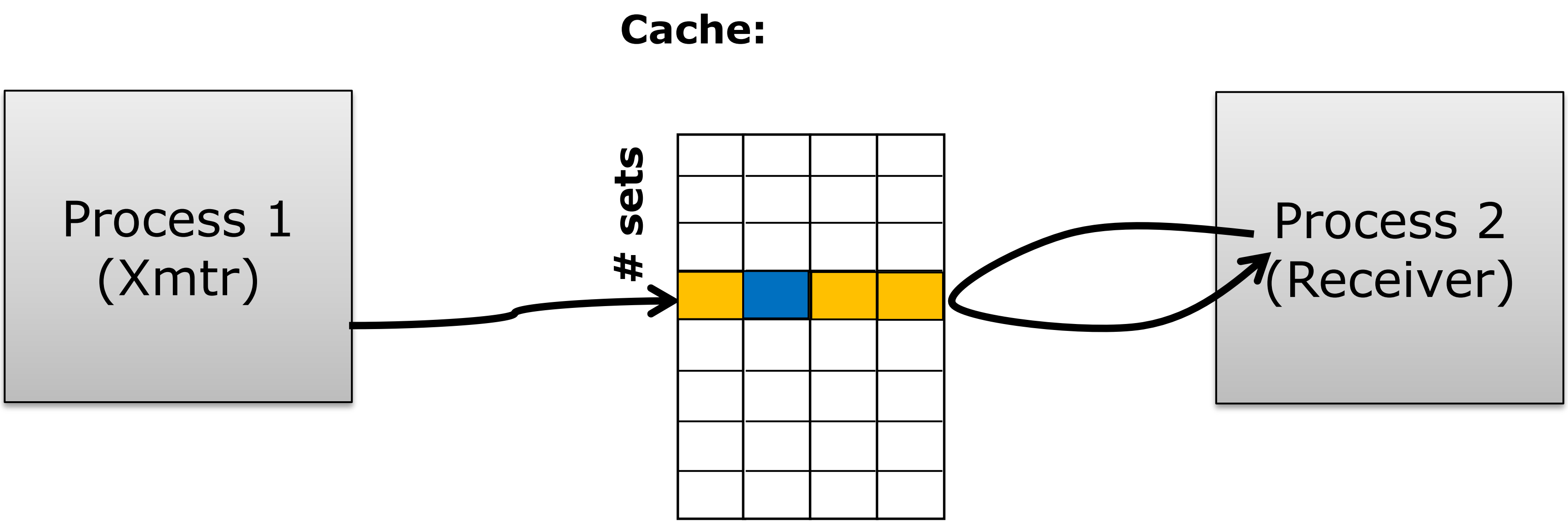
A Multi-way Cache-based Channel



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

fill a set

A Multi-way Cache-based Channel

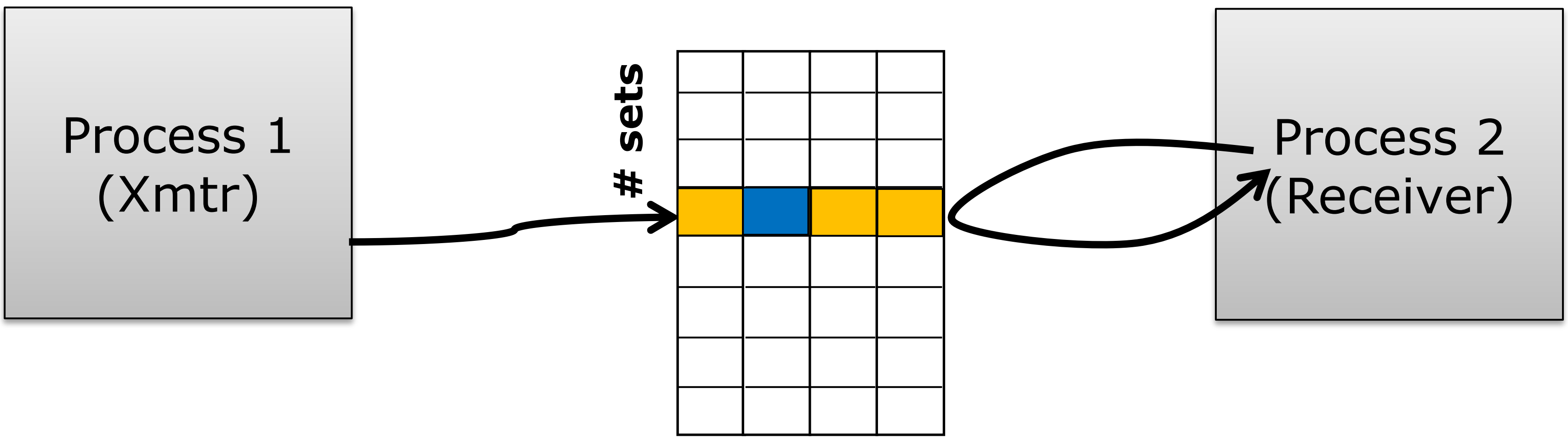


if (**send '0'**)
    idle
else
    write to a set

fill a set

A Multi-way Cache-based Channel

Cache:

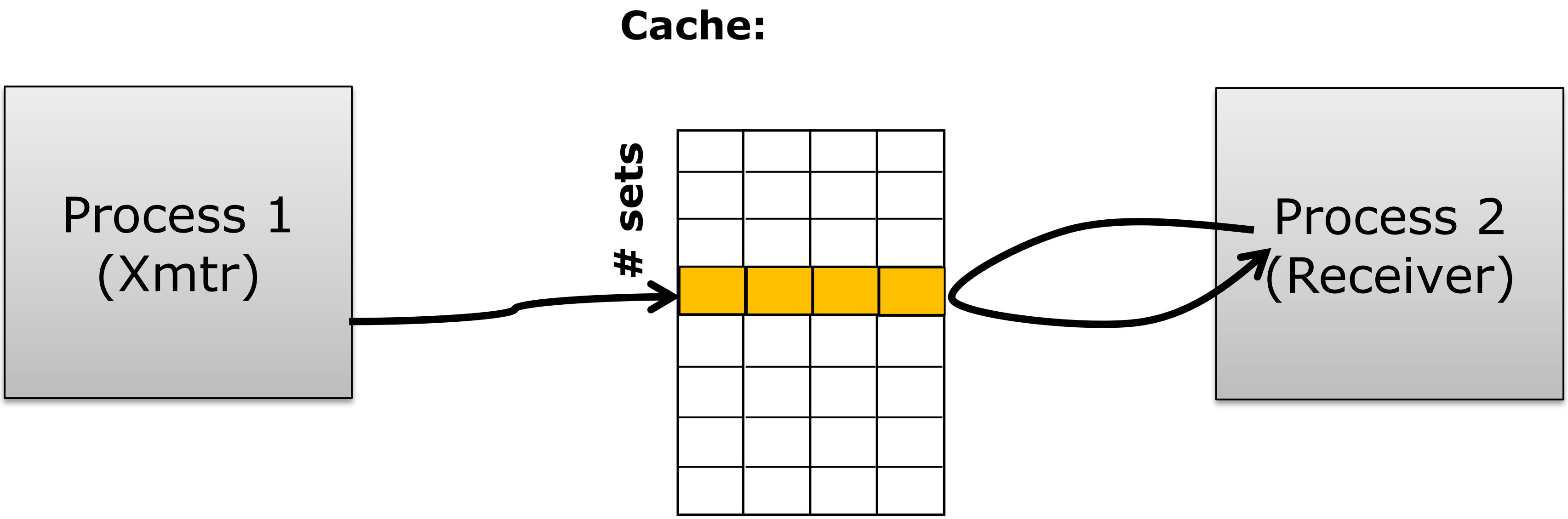


if (**send '0'**)
  idle
else
  write to a set

fill a set

t1 = rdtsc()
read all of the set
t2 = rdtsc()

A Multi-way Cache-based Channel

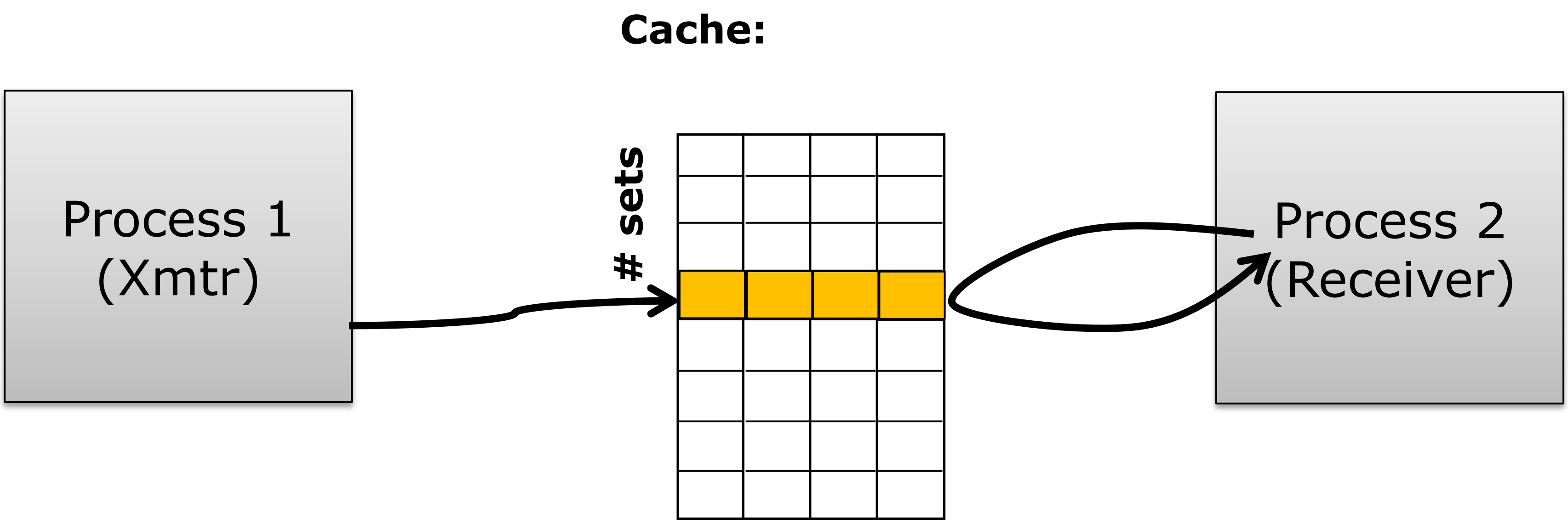


if (**send ‘0’**)
   idle
else
   write to a set

fill a set

t1 = rdtsc()
read all of the set
t2 = rdtsc()

A Multi-way Cache-based Channel



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

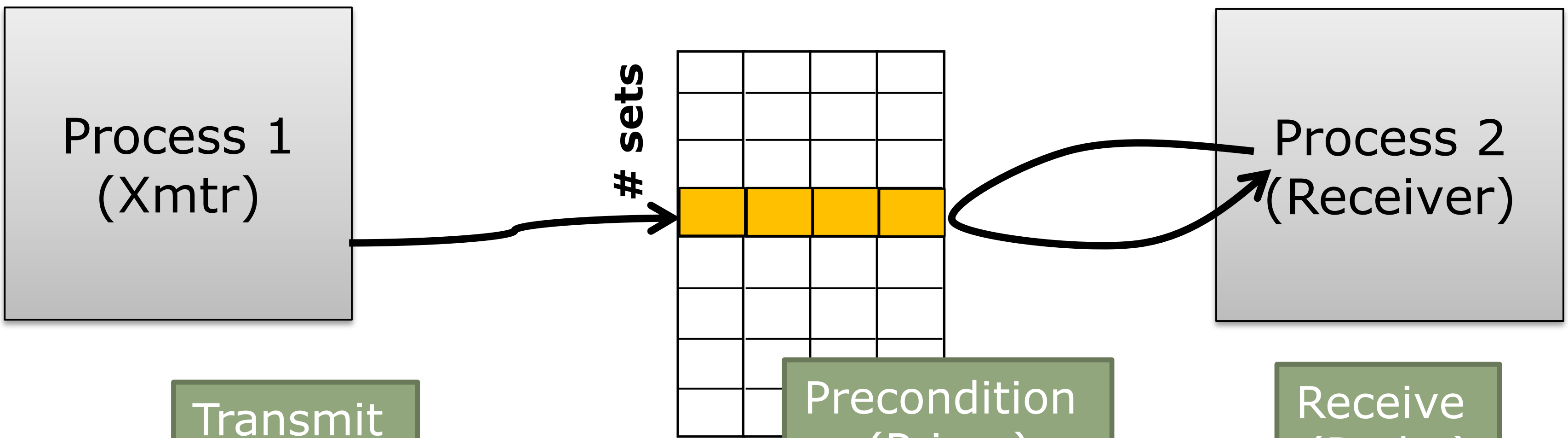
fill a set

t1 = rdtsc()
read all of the set
t2 = rdtsc()

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

A Multi-way Cache-based Channel

Cache:



Transmit

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

Precondition (Prime)

fill a set

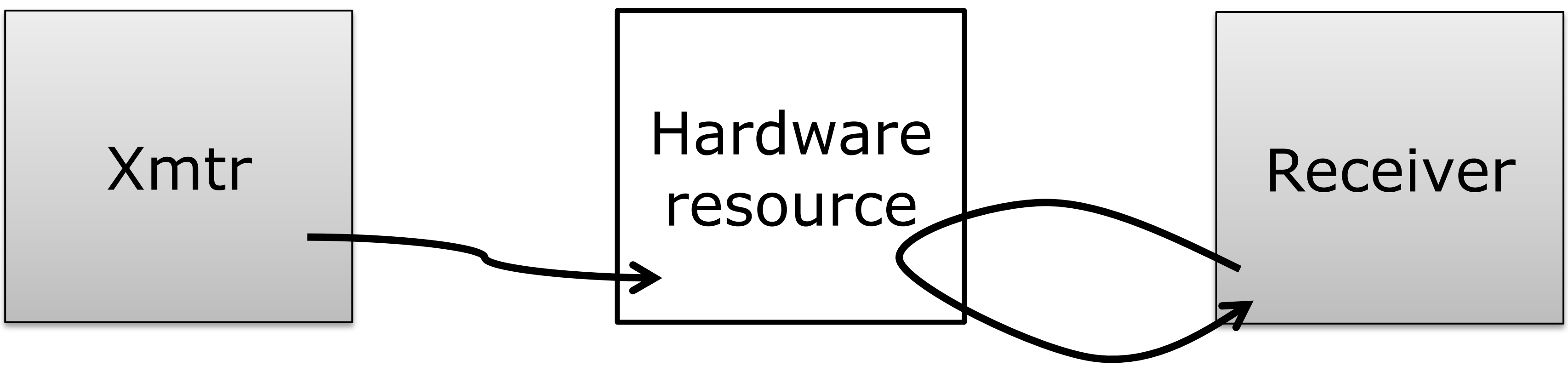
Receive (Probe)

t1 = rdtsc()
read all of the set
t2 = rdtsc()

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

Decode

Generalizes to Other Resources

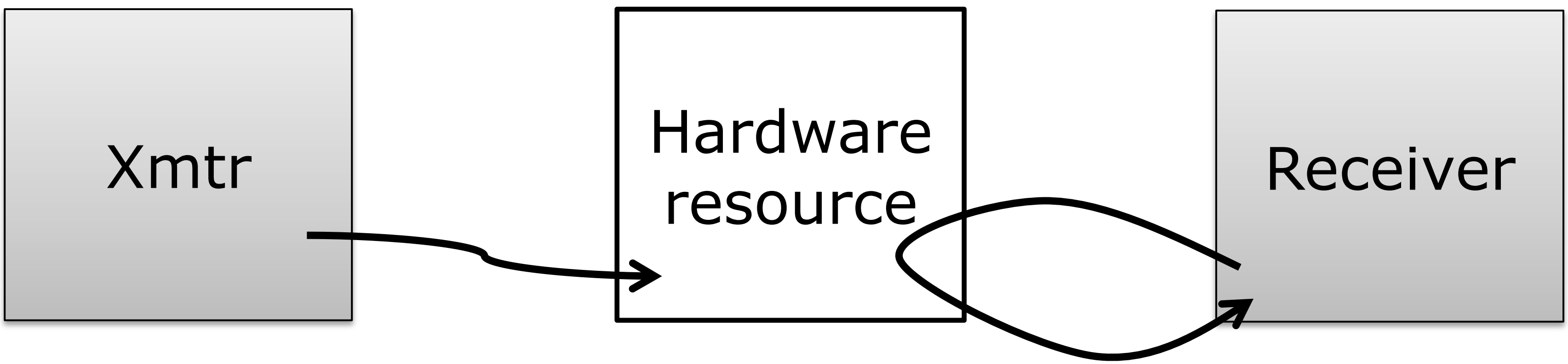


if (send ‘1’)
  Use resource
else
  idle

t1 = rdtsc()
Use resource
t2 = rdtsc()

if (t2 – t1 > THRESH)
  read ‘1’
else
  read ‘0’

Generalizes to Other Resources



if (send '1')
  Use resource
else
  idle

t1 = rdtsc()
Use resource
t2 = rdtsc()

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

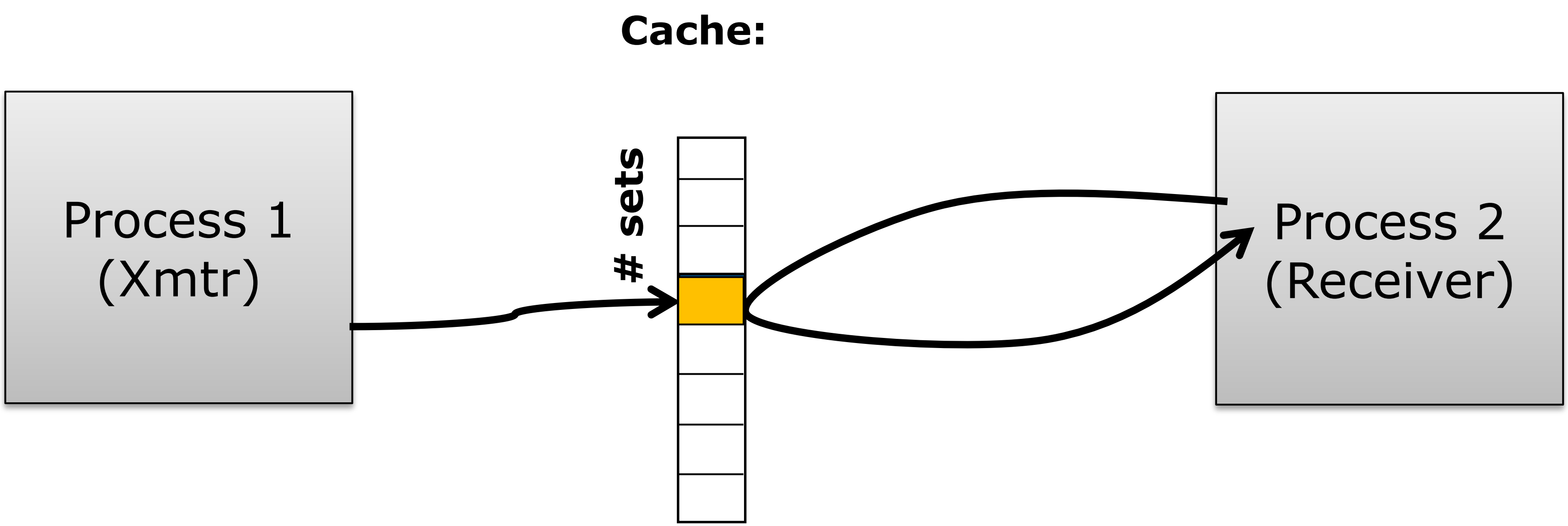
Any other exploitable structures?

Channel Examples

Resource	Shared by
Private cache (L1, L2)	Intra-core
Shared cache (LLC)	On-socket cross core
Cache directory	Cross socket
DRAM row buffer	Cross socket
TLB (private/shared)	Intra-core/Inter-core
Branch Predictor	Intra-core
Network-on-chip	On-socket cross core
…	…

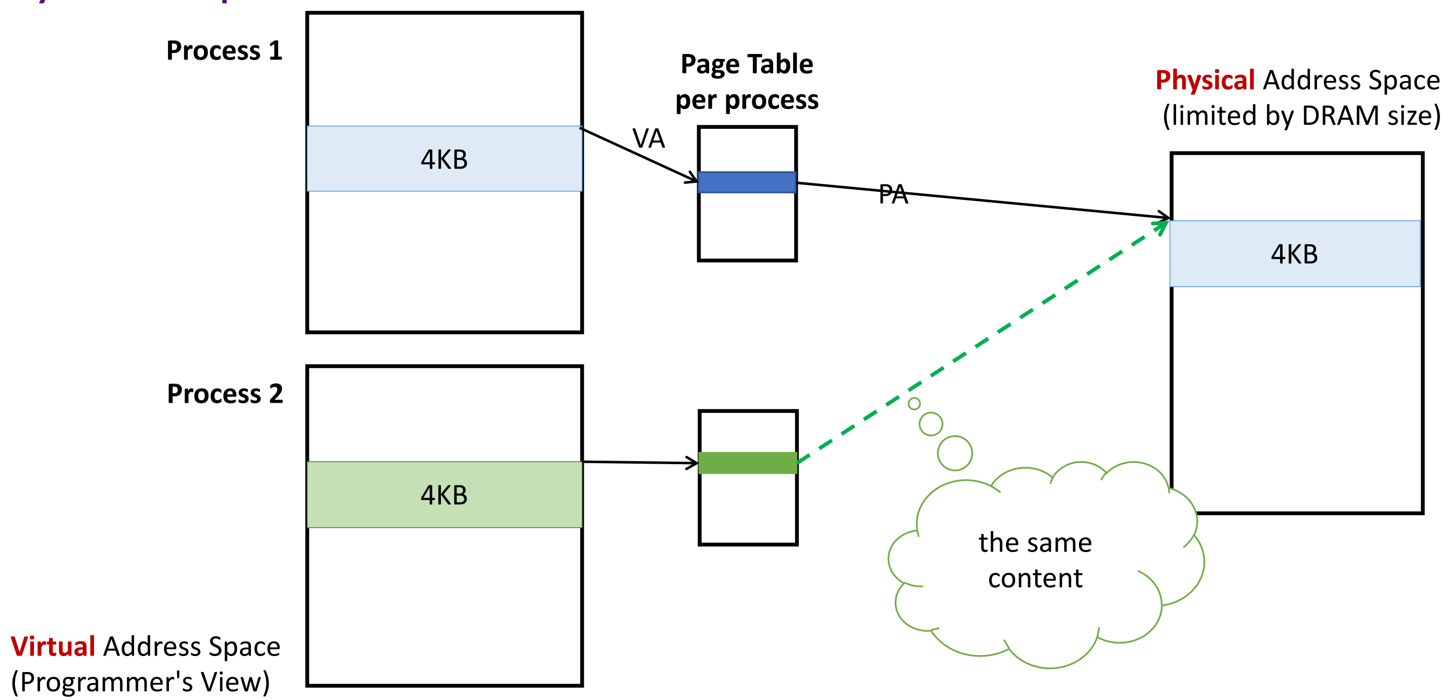
See Attack in Action: Flush+Reload

- The conceptual version
  - The sender and receiver shares addresses in a page
  - Sender repeated accesses address A or B
  - Receiver repeats:
    - flush A and B; using "clflush" -> precondition
    - wait for a few cycles; (sender does something) -> modulation
    - time how long it takes to reload A and B -> receive+decode



See Attack in Action: Page Sharing

- Virtual addresses in different processes map to the same physical address. When?
  - Lazy page allocation
  - Shared library
  - Memory de-duplication



See Attack in Action: Pseudocode

Sender:

```
buffer = mmap(4KB);
secret = getinput();

while (true){
    load buffer[secret*64];
}
```

See Attack in Action: Pseudocode

Sender:

```
buffer = mmap(4KB);
secret = getinput();

while (true){
    load buffer[secret*64];
}
```

Why *64?

See Attack in Action: Pseudocode

Sender:

```
buffer = mmap(4KB);
secret = getinput();

while (true){
    load buffer[secret*64];
}
```

Why *64?

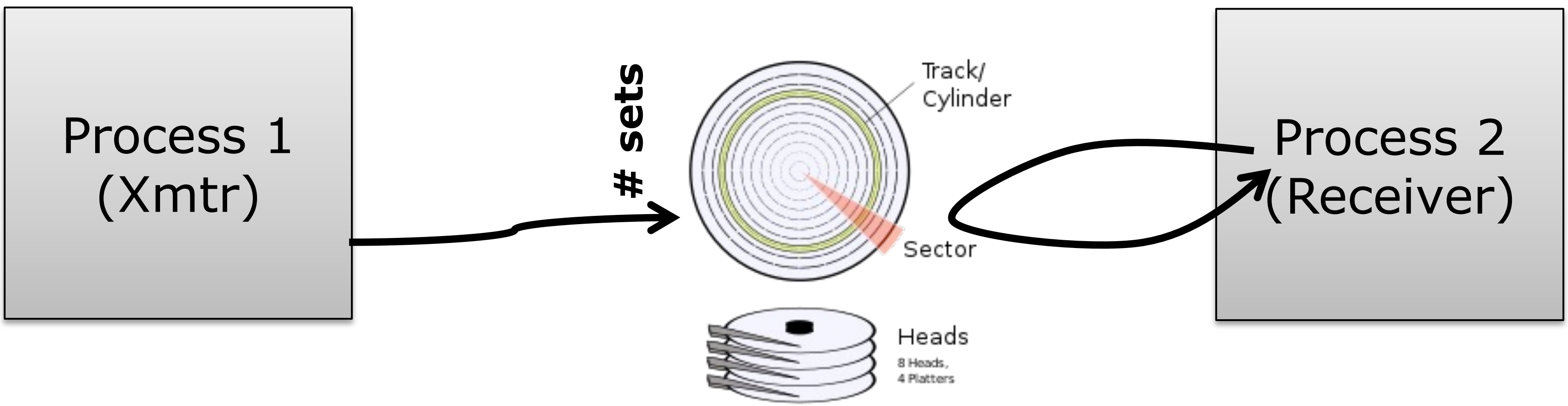
Receiver:

```
buffer = mmap(4KB);
hit_count [MAX] = 0;

for i in range(0,MAX){
    t1 = rdtsc();
    load buffer[i*64];
    t2 = rdtsc();
    if (t2-t1 > threshold){
        hit_count[i] ++;
    }
}
```

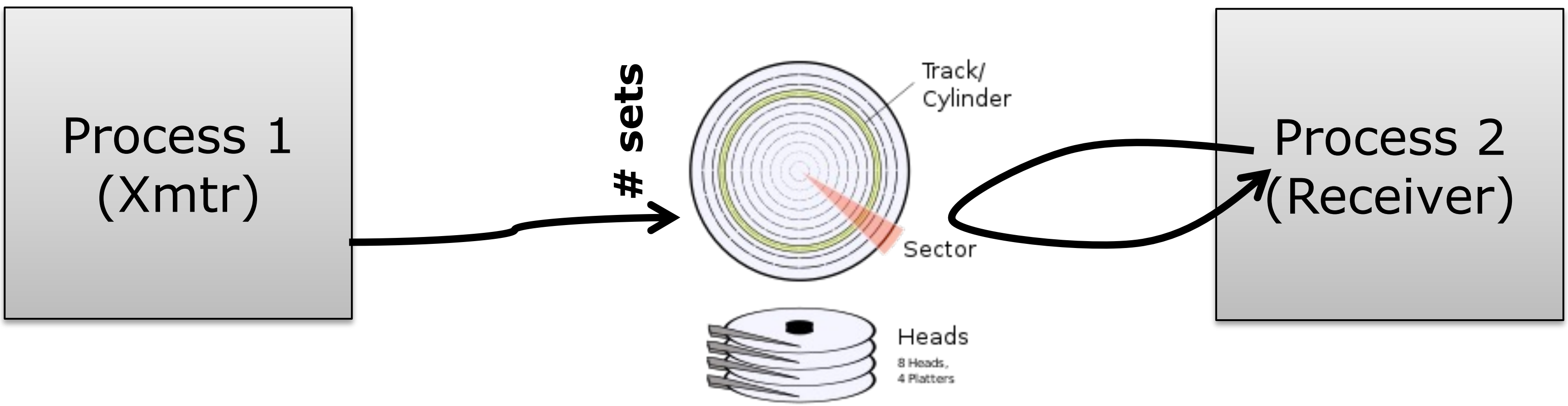
Disrupting Communication

Cache:



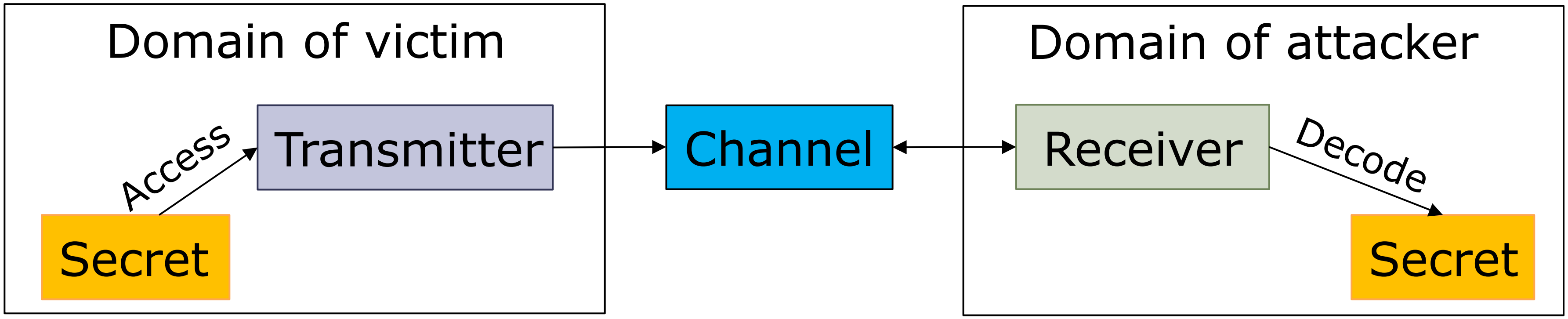
Disrupting Communication

Cache:



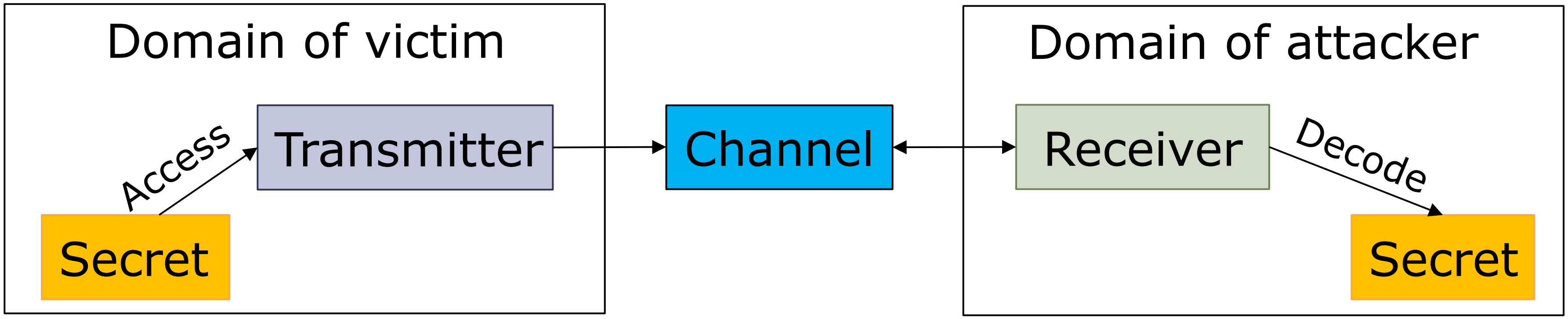
"We found that identifying all of the sources of accurate clocks was much **easier** than finding all of the possible timing channels in the system.
… If we could make the clocks less accurate, then the effective bandwidth of all timing channels in the system would be **lowered**." *(1991)*

Secret-independent Channel Modulation



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

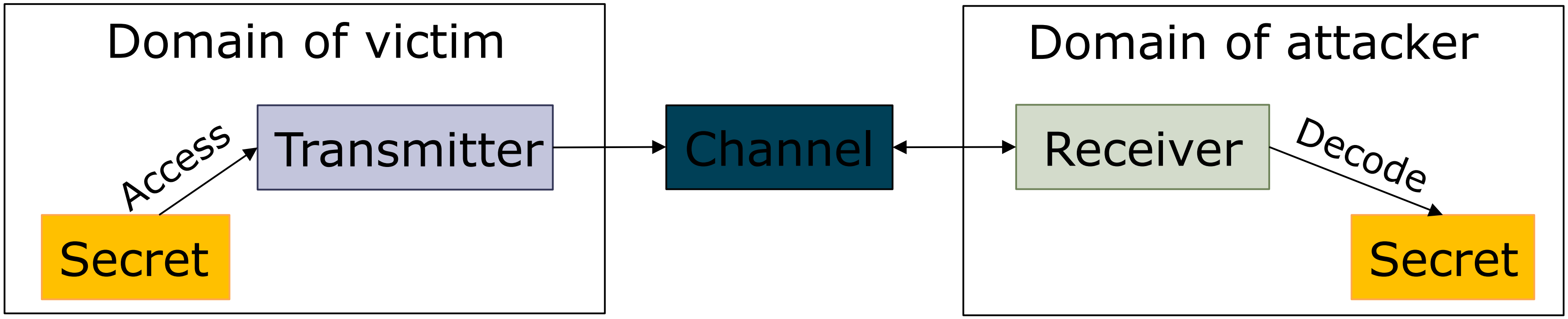
Secret-independent Channel Modulation



- 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”.

Secret-independent Channel Modulation

Input : base **b**, modulo m,
exponent $\boldsymbol{e = (e_{n-1} \ldots e_0)_2}$
Output: $b^e \bmod m$
$r = 1$
for $i = n-1$ *down to* 0 do
  $r = sqrt(r)$
  $r = mod(r,m)$
    if $\boldsymbol{e_i}$ **== 1 then**
      $r = mul(r,b)$
      $r = mod(r,m)$
    end
end
return r

How to make the code execution independent of the secret?

Secret-independent Channel Modulation

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

Secret-independent Channel Modulation

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

```
p = (ei == 1)
r2 = mul(r,b)
r2 = mod(r,m)
cmov [p] r, r2
```

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 $\boldsymbol{e = (e_{n-1} \dots e_0)_2}$
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*

```
p = (ei == 1)
r2 = mul(r,b)
r2 = mod(r,m)
cmov [p] r, r2
```

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 $\mathbf{e = (e_{n-1} \ldots e_0)_2}$
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*

```
p = (ei == 1)
r2 = mul(r,b)
r2 = mod(r,m)
cmov [p] r, r2
```

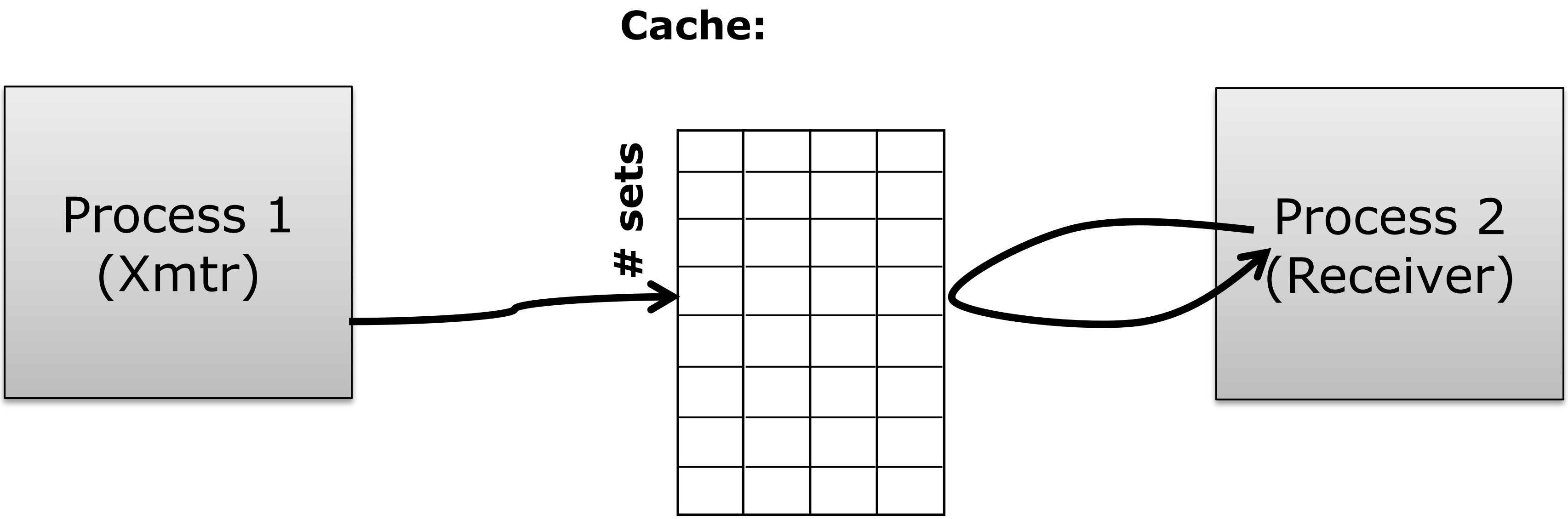
How to make the code execution independent of the secret?

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

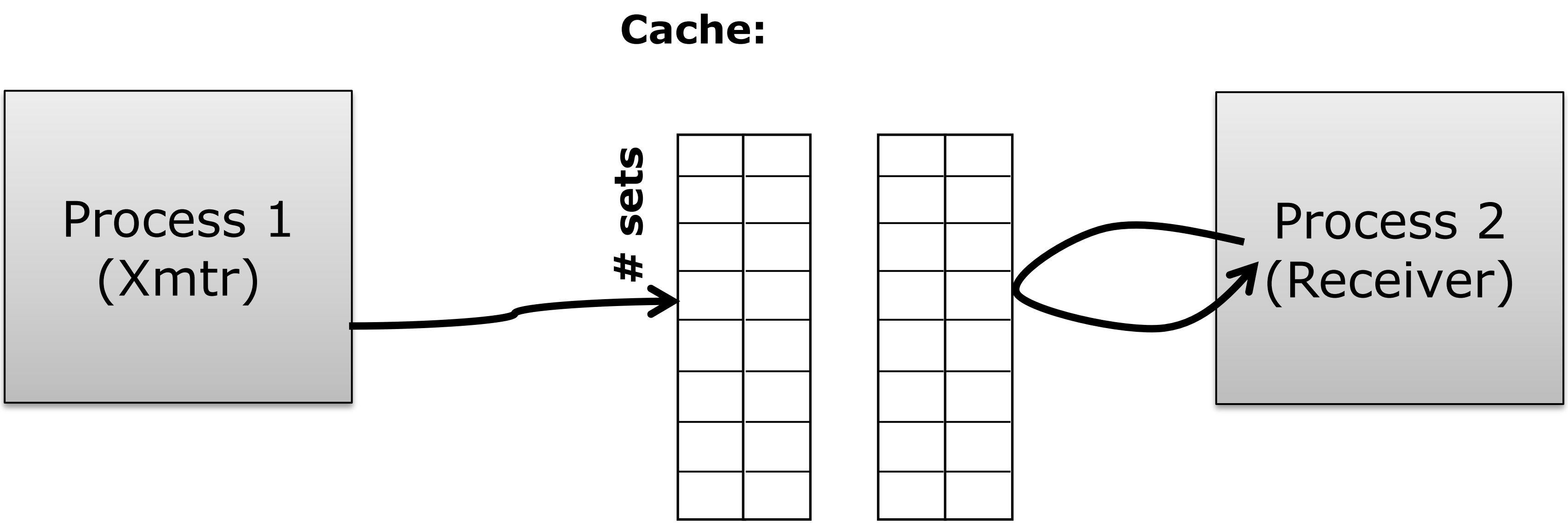
After removing the secret-dependent branch, how about code inside these functions?

Constant-time programming is hard

Disrupting Communication



Disrupting Communication



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

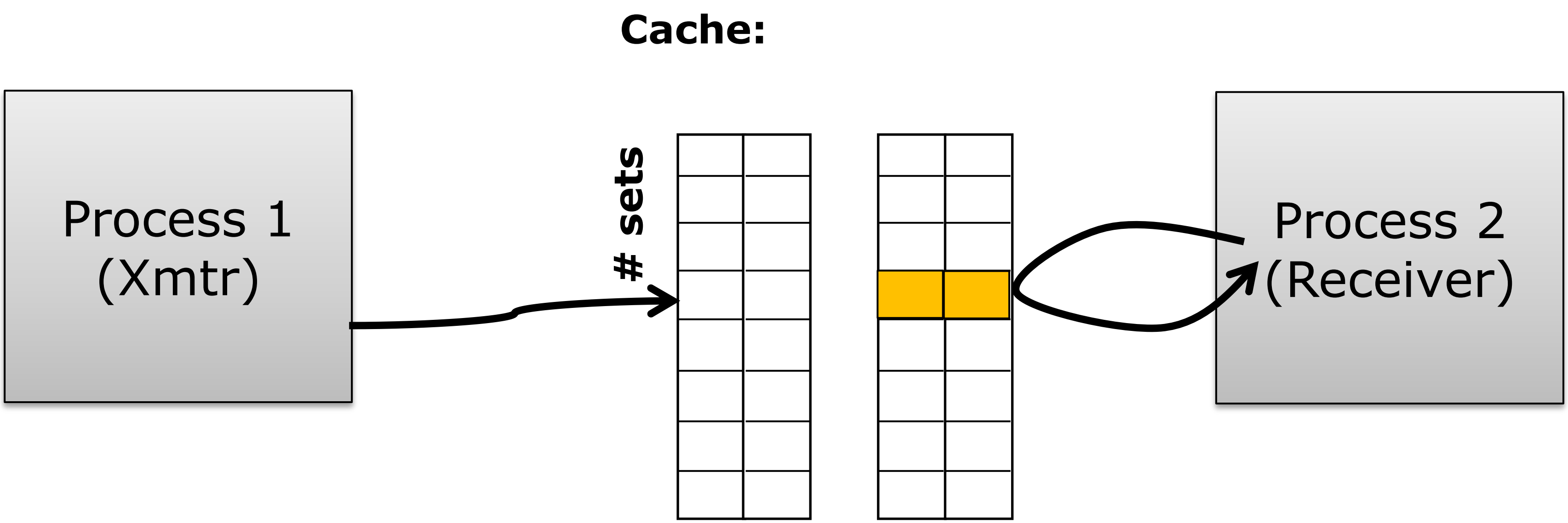
fill a set

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

Disrupting Communication



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

fill a set

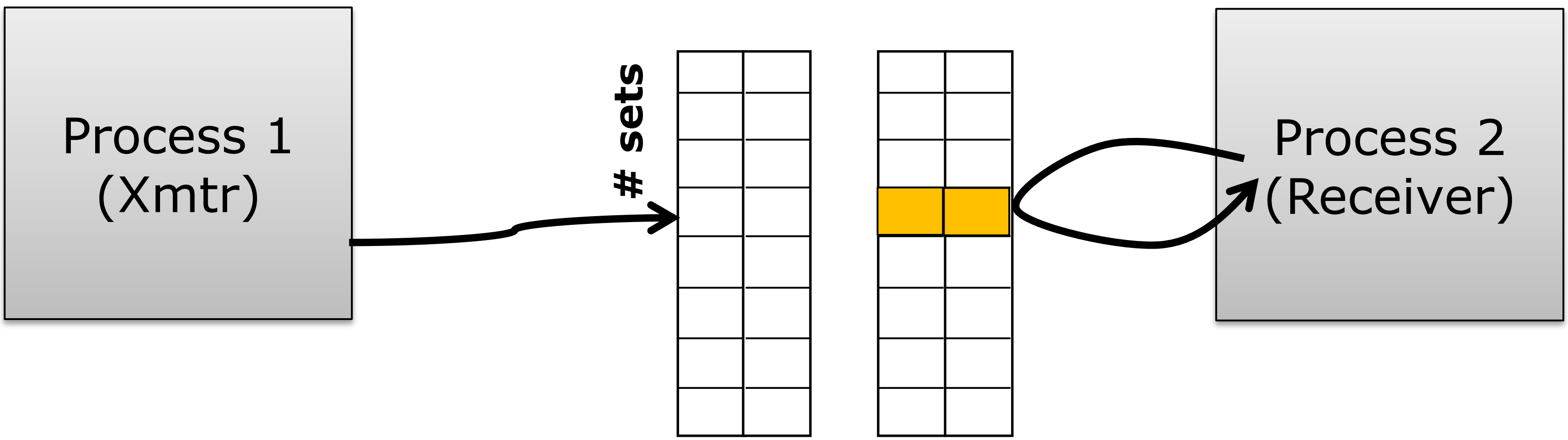
t1 = rdtsc()
read all of the set
t2 = rdtsc()

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

Kirianski et. al. Dawg, Micro'18

Disrupting Communication

Cache:



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

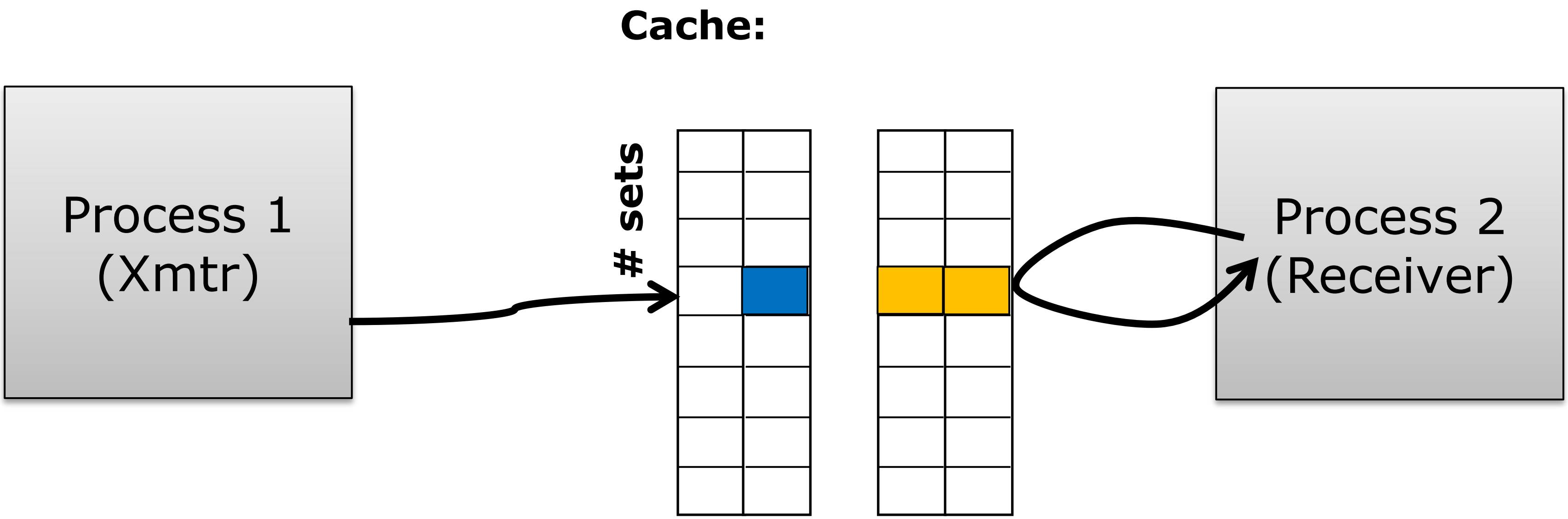
fill a set

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

Disrupting Communication



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

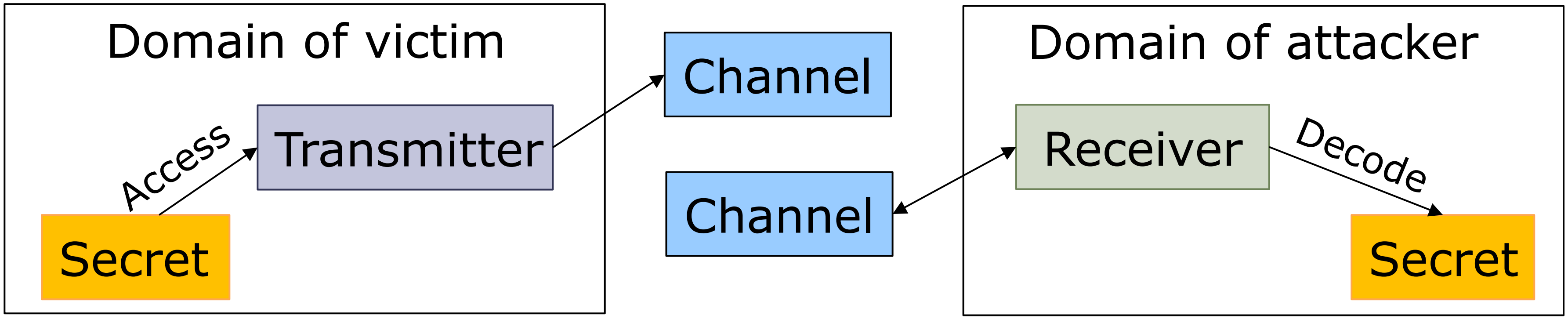
fill a set

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

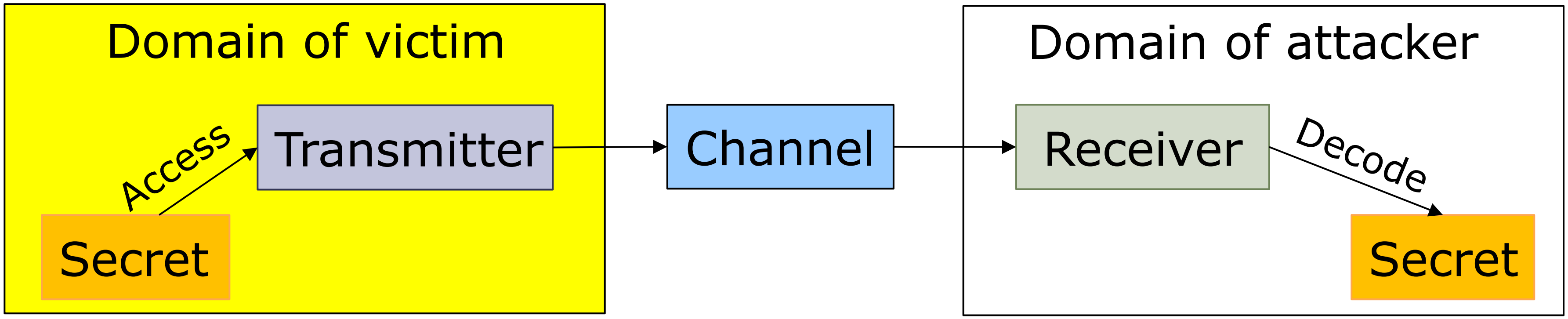
Disjoint Channels



- Making disjoint channels makes communication impossible.

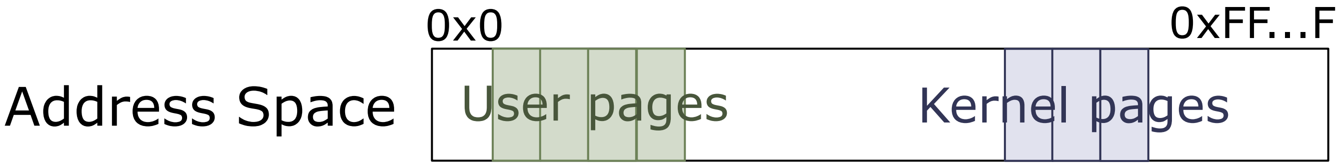
- Channel can be allocated by “domain” and will need to be “cleaned” as processes enter and leave running state, so next process cannot see any “modulation” on the channel.

Types of Transmitters

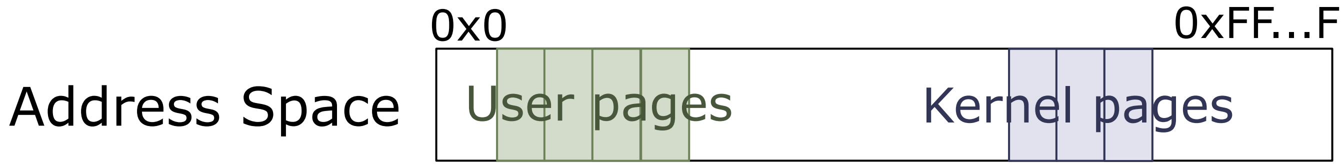


- Types of transmitter:
  1. Pre-existing so victim itself leaks secret, (e.g., RSA keys)
  2. Programmed and invoked by attacker (e.g., Meltdown)

Reminder: Speculative Execution

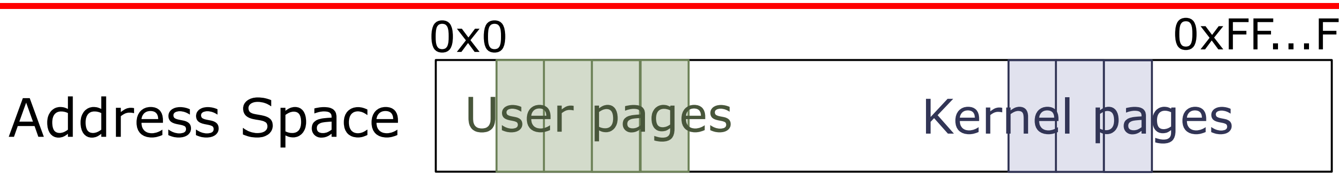


Reminder: Speculative Execution



- 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

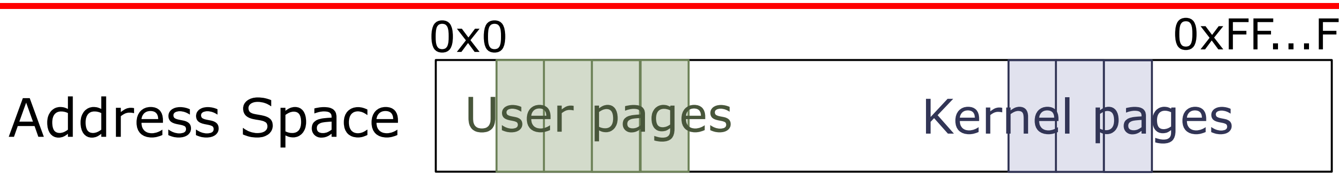
Address Space

- 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?

```
val = *kernel_address;
```

- Causes a protection fault, but data at “kernel_address” is speculatively read and loaded into val.

Meltdown [Lipp et al. 2018]

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

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;
```

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.

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!

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!
  - For higher performance, use transactional memory (protection fault aborts transaction on exception instead of invoking kernel)

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!
  - For higher performance, use transactional memory (protection fault aborts transaction on exception instead of invoking kernel)
  - Mitigation?

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!
  - 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 (KPTI)

Meltdown [Lipp et al. 2018]

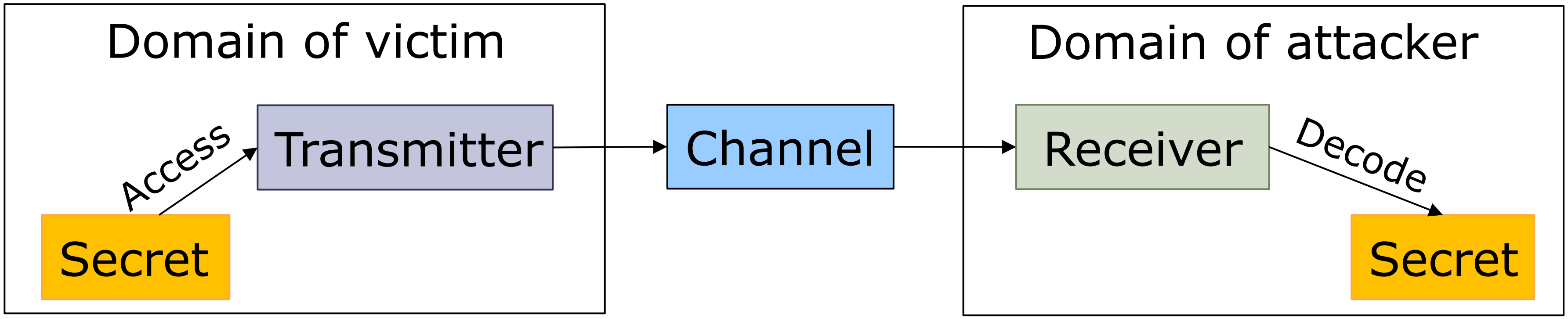
1. Preconditioning: Receiver allocates an array `subchannels[256]` and flushes all its cache lines
2. Transmit: Transmitter (controlled by attacker) executes

```
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!
  - 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 (KPTI)
    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)
    3. Synthesized from existing victim code and invoked by attacker (e.g., Spectre v2)

Spectre variant 1
[Kocher et al. 2018]

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

```
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:

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

- Interpret that code as a transmitter:

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

Spectre variant 1
[Kocher et al. 2018]

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

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

- Interpret that code as a transmitter:

```
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];
}
```

Spectre variant 1
[Kocher et al. 2018]

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

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

- Interpret that code as a transmitter:

```
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];
}
```

Conditional branch misprediction

Spectre variant 1
[Kocher et al. 2018]

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

```
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

```
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

```
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 <u>out-of-bounds</u> x, so that `&array1[x]` points to a desired kernel address. Core mispredicts branch, <u>speculatively</u> fetches address `&array2[array1[x] * 4096]` into the cache.

Spectre variant 1
[Kocher et al. 2018]

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

```
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 <u>out-of-bounds</u> x, so that `&array1[x]` points to a desired kernel address. Core mispredicts branch, <u>speculatively</u> 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

Victim_branch

```
        kernel_address = a_desired_address;
        jump some_where_else
        …
```

training_branch

```
        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

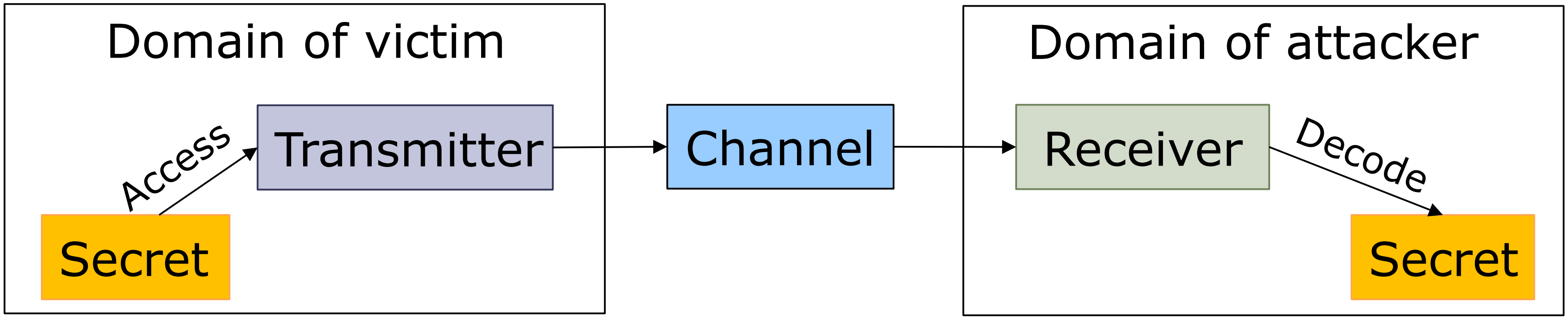
Victim_branch

training_branch

```
          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];
```

1. Train: trigger `victim_branch -> xmit` many times
2. Transmit: '`victim_branch`' and '`training_branch`' alias in BTB, so we can speculatively trigger `victim_branch -> 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*
- 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 2024: Secure Hardware Design 6.5950/1

Learn to attack processors…

Side channel attacks

Spectre, Meltdown, Foreshadow

Row-hammer attacks

Intel SGX

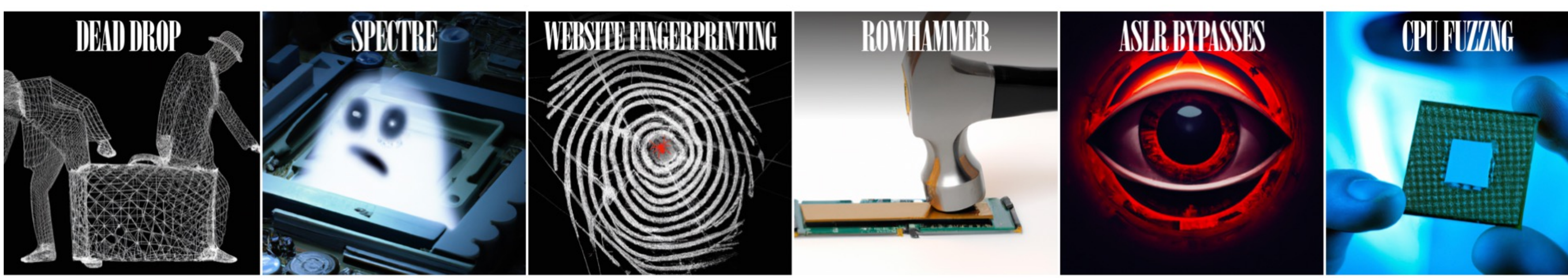
ARM TrustZone

Hardware mitigations for ROP/ JOP

And how to defend them!

Secure Hardware Design @ MIT

Making Computer Architecture Fun!



https://shd.mit.edu

Old number: 6.S983, 6.888

Thank you!