Security

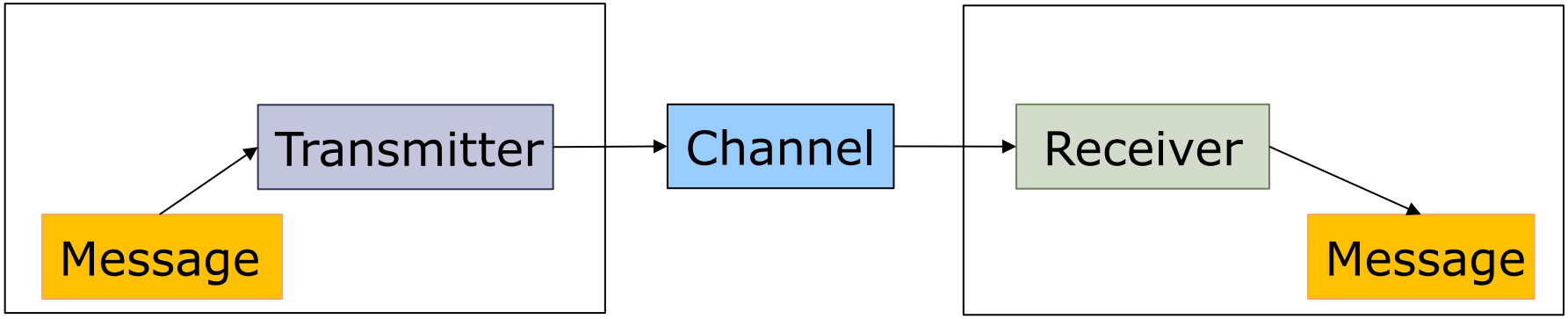
*Joel Emer**
Computer Science & Artificial Intelligence Lab
M.I.T.

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

Security and Information Leakage

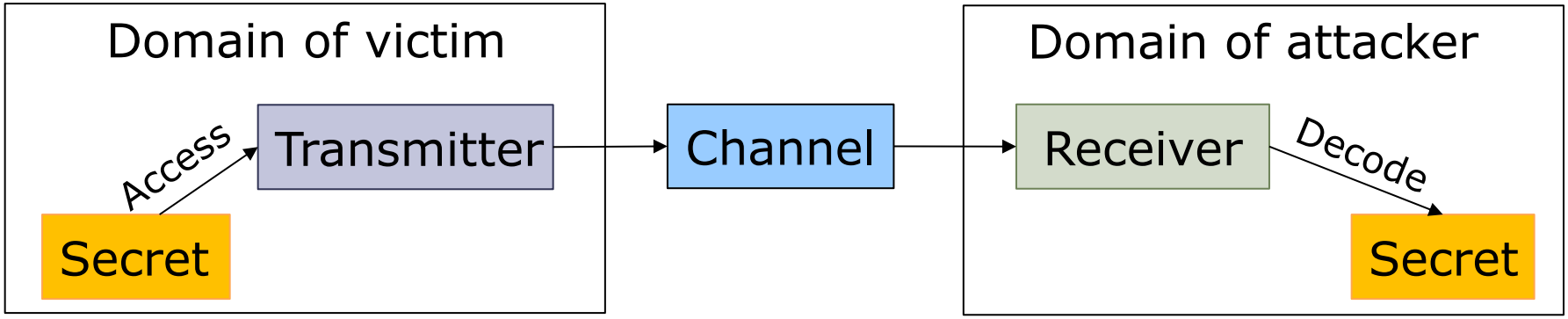
- Hardware isolation mechanisms like virtual memory guarantee that architectural state will not be directly exposed to other processes…but
- ISA and ABI are timing-independent interfaces, and
  - Specify *what* should happen, not *when*
- ISA and ABS only specify architectural updates
  - *Micro-architectural 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



- Transmitter accepts message
- Transmitter modulates channel
- Receiver detects modulation on channel
- Receiver decodes modulation as message.

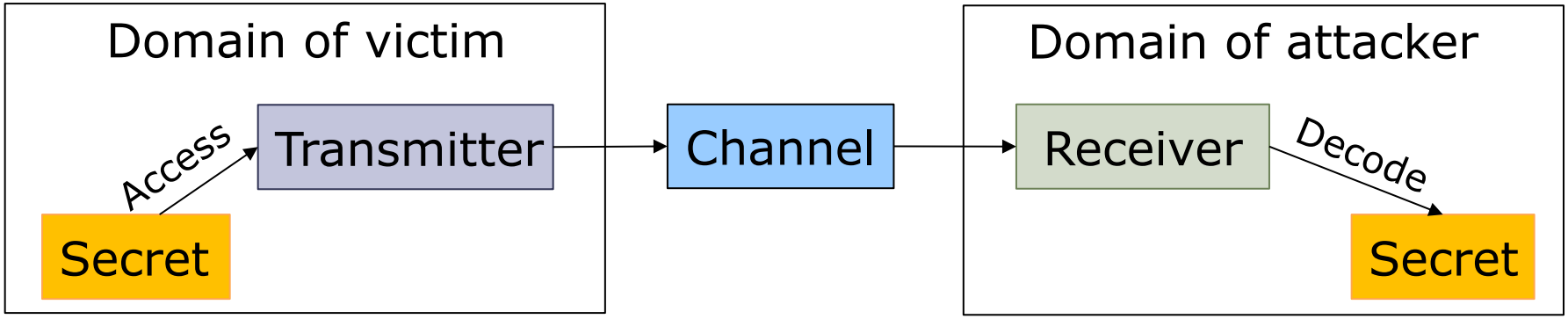
Communication Model of Attacks
[Belay, Devadas, Emer]



- 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

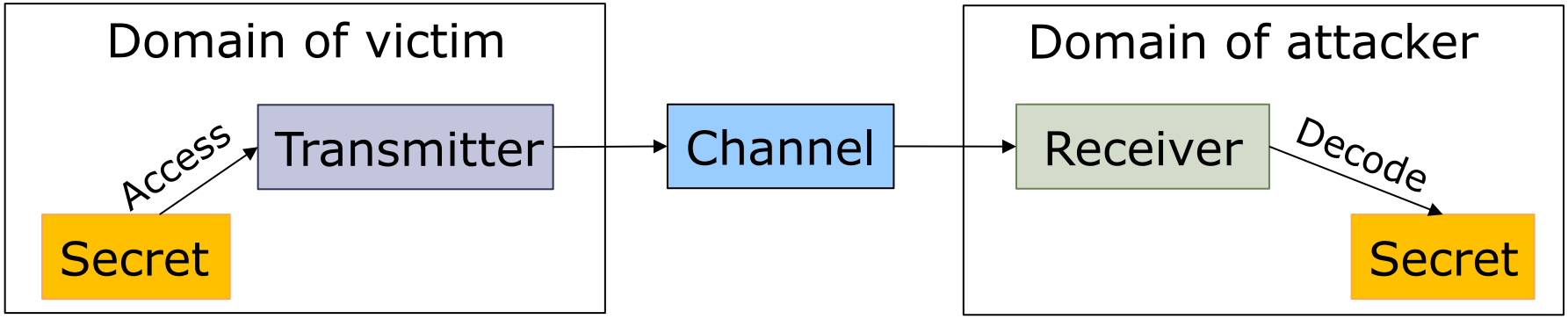
Because channel is not a "direct" communication channel it is often referred to as a "side channel"

Communication Model of Attacks
[Belay, Devadas, Emer]



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

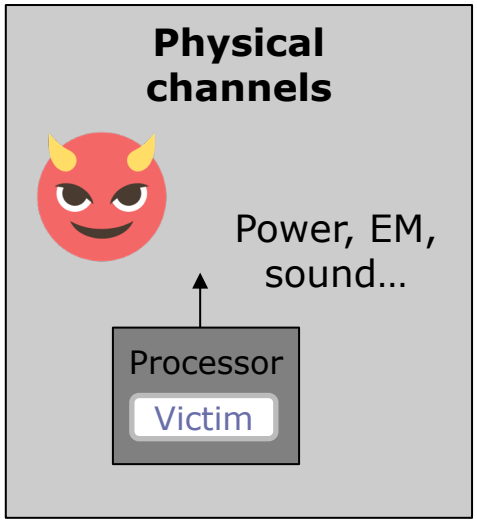
ATM Acoustic Channels



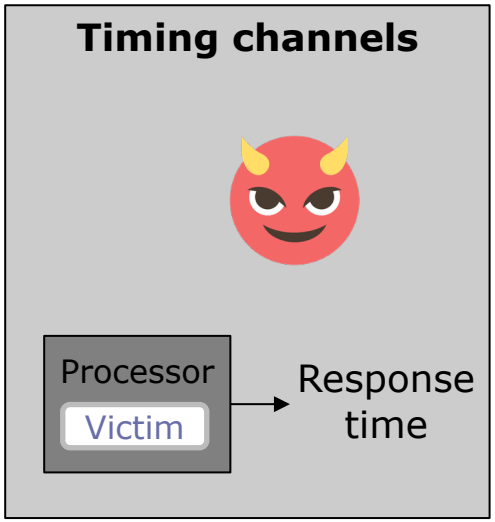
- Secret: Pin
- Transmitter: Keypad
- Channel: Air
- Modulation: Acoustic waves
- Receiver: Cheap Microphone
- Decoders: ML Model

Physical vs Timing vs uArch Channel

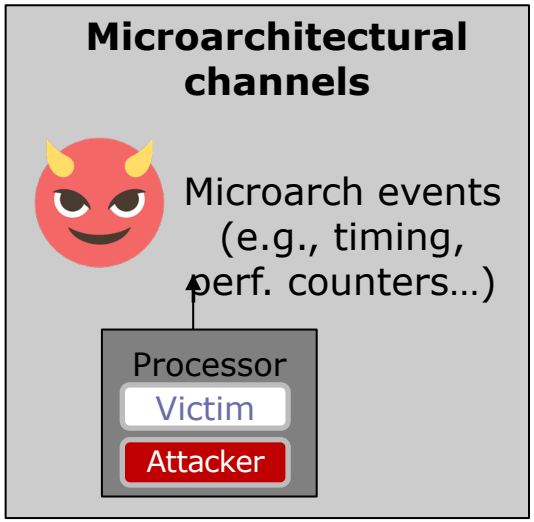
- What can the adversary observe?



Attacker requires measurement equipment → physical access



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



Attacker may be remote, or be co-located

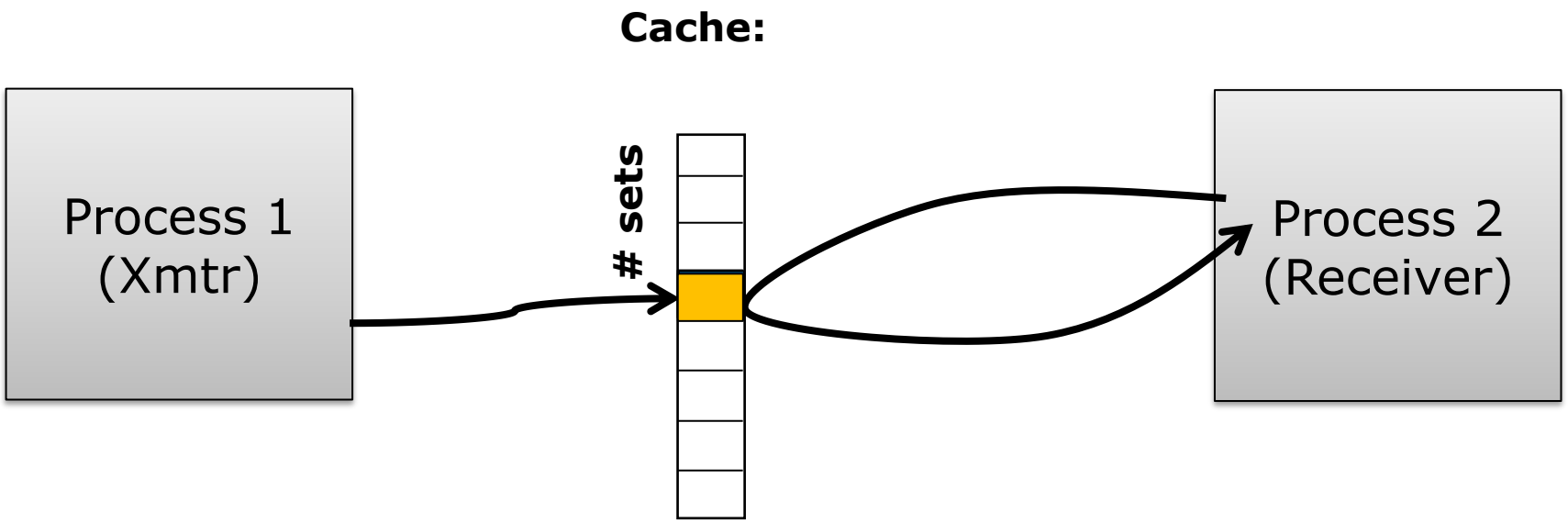
What can you do with these channels?

- Violate privilege boundaries
  - Inter-process communication
  - Infer an application's secret
- (Semi-Invasive) application profiling

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



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

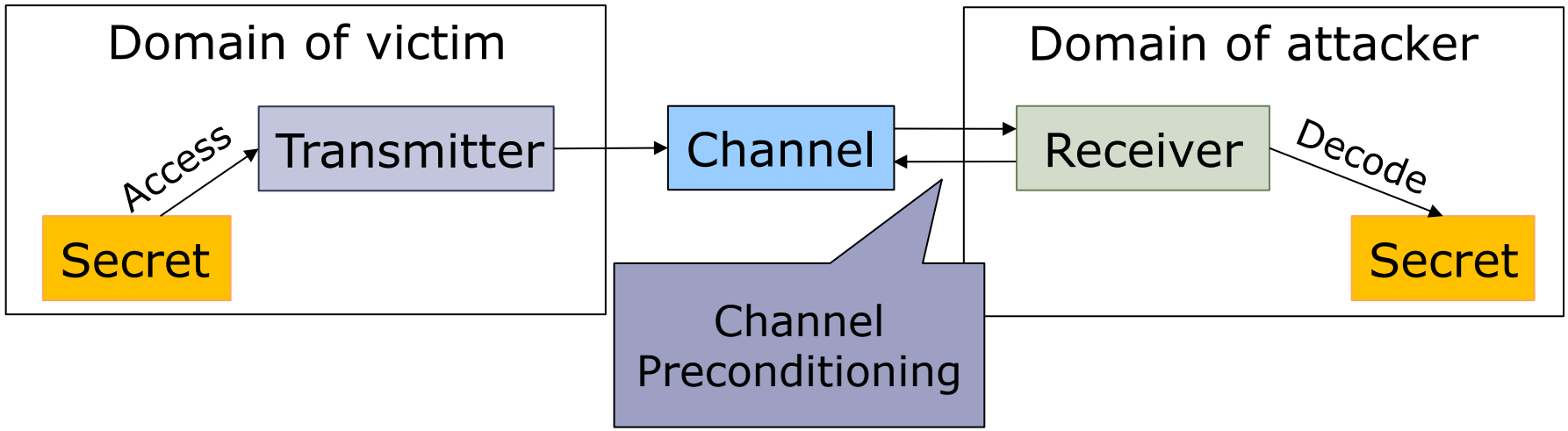
write to set

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

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

Note this requires an "active" receiver

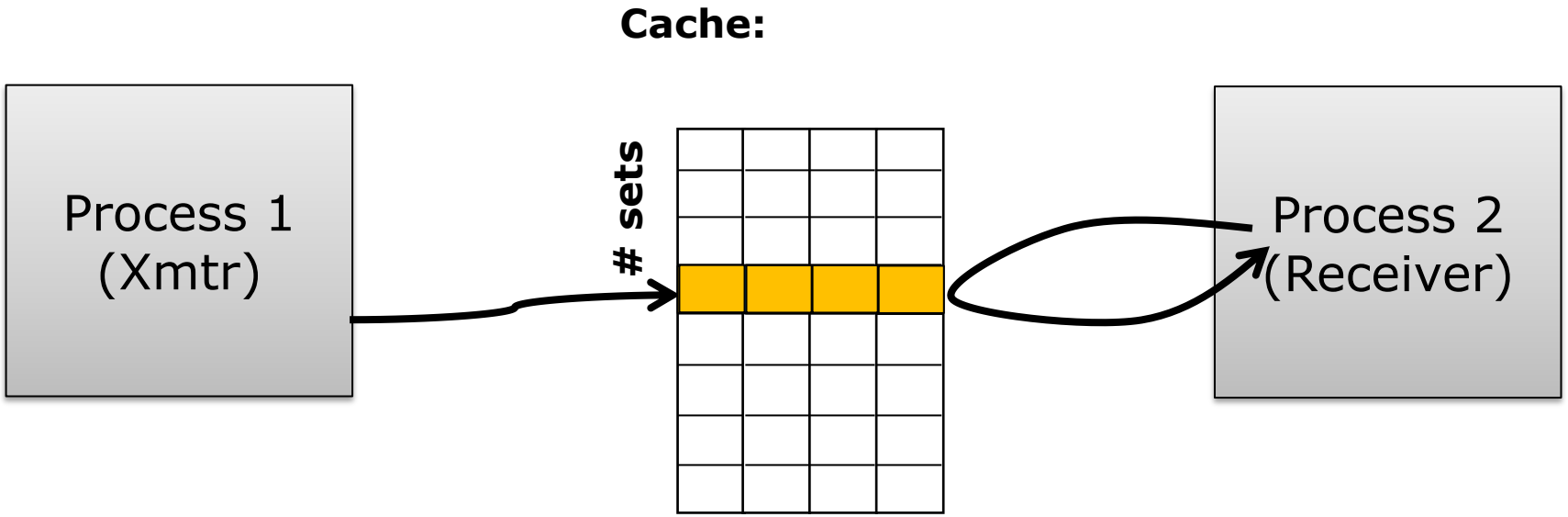
Communication w/ Active Receiver



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



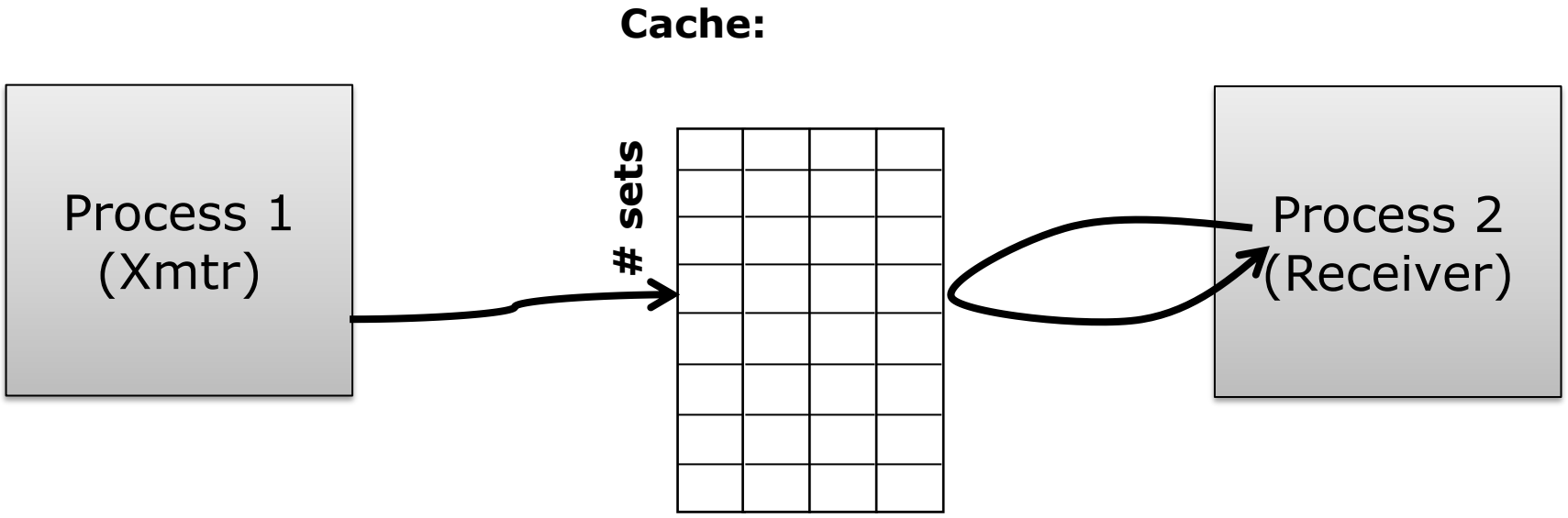
if (**send '0'**)
idle
else
write to a set

fill a set

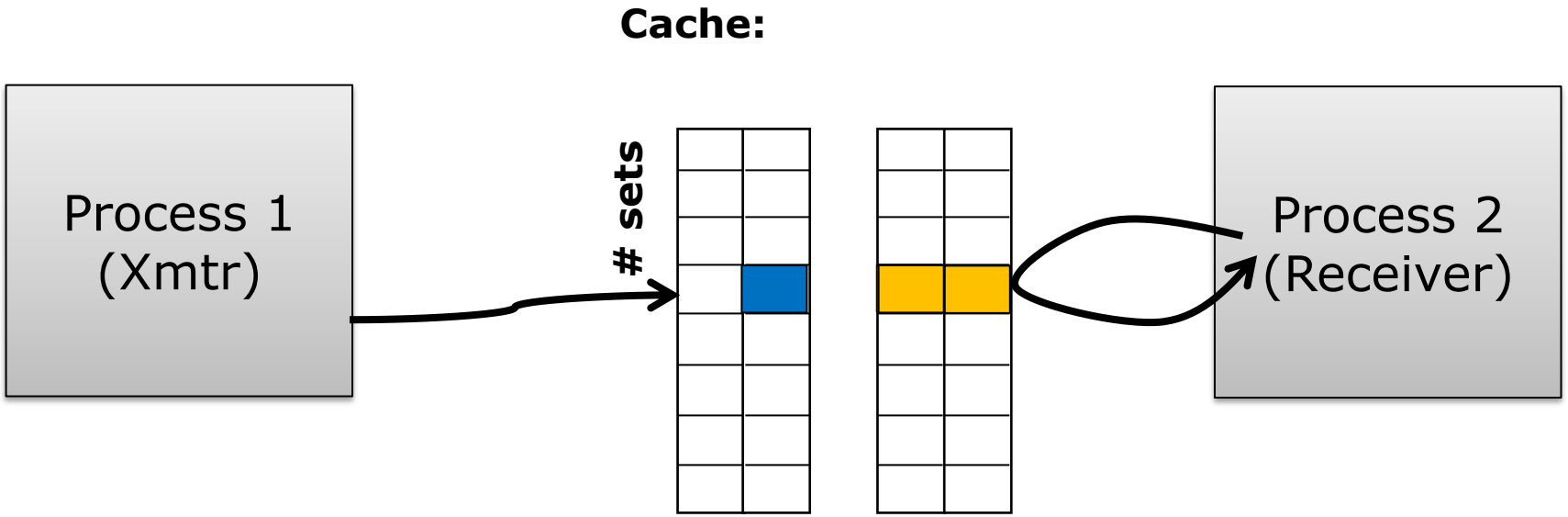
t1 = rdtsc()
read all of the set
t2 = rdtsc()

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

Disrupting Communication



Disrupting Communication



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

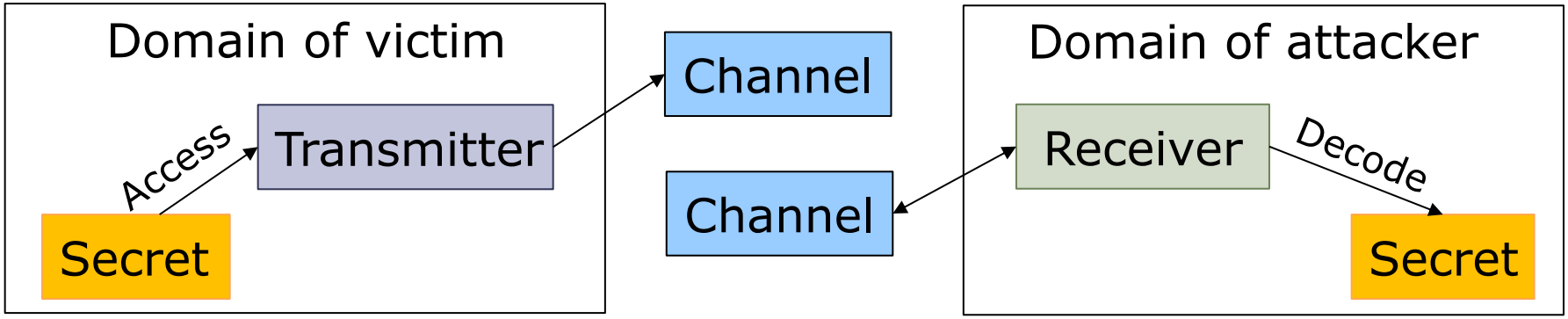
fill a set

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

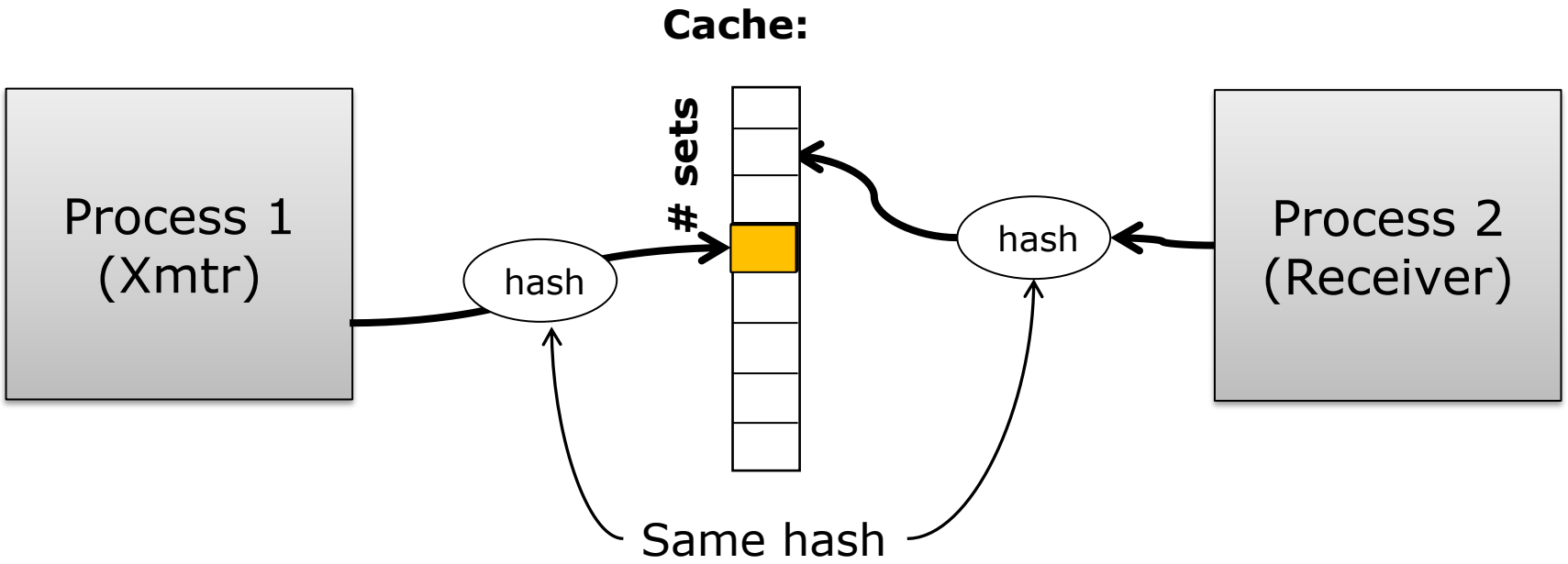
Disjoint Channels



1. Making disjoint channels makes communication impossible.

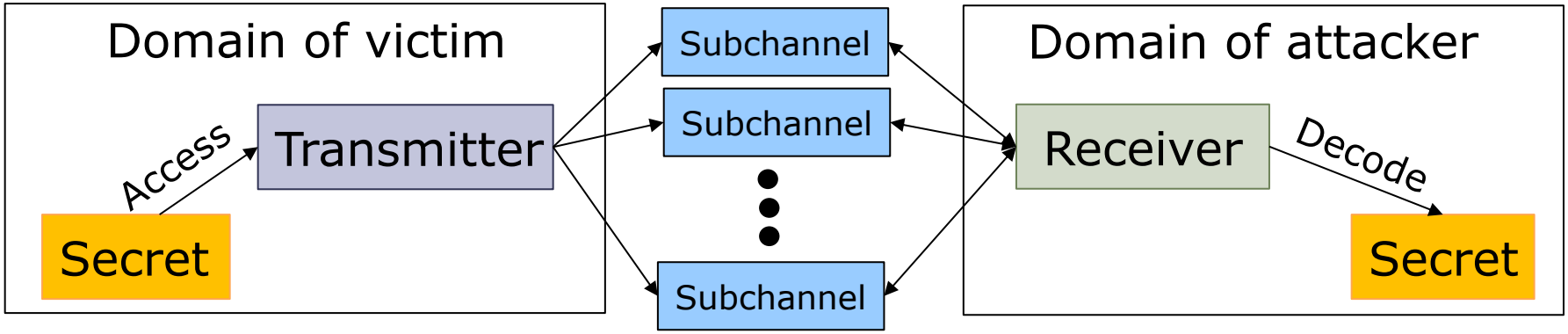
2. Channel can be allocated by "domain" and will need to be "cleaned" as processes enter and leave running state, so next process cannot see any "modulation" on the channel.

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.

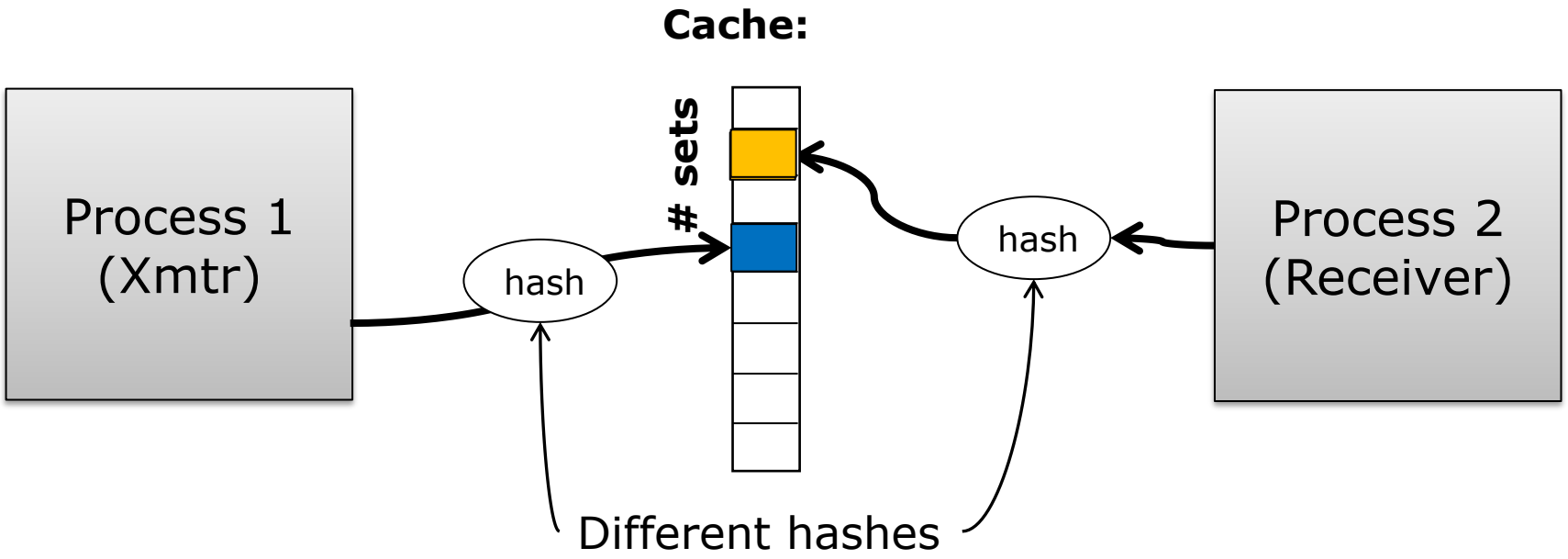
Communication with subchannels



1. Transmissions may now occur on one of many subchannels

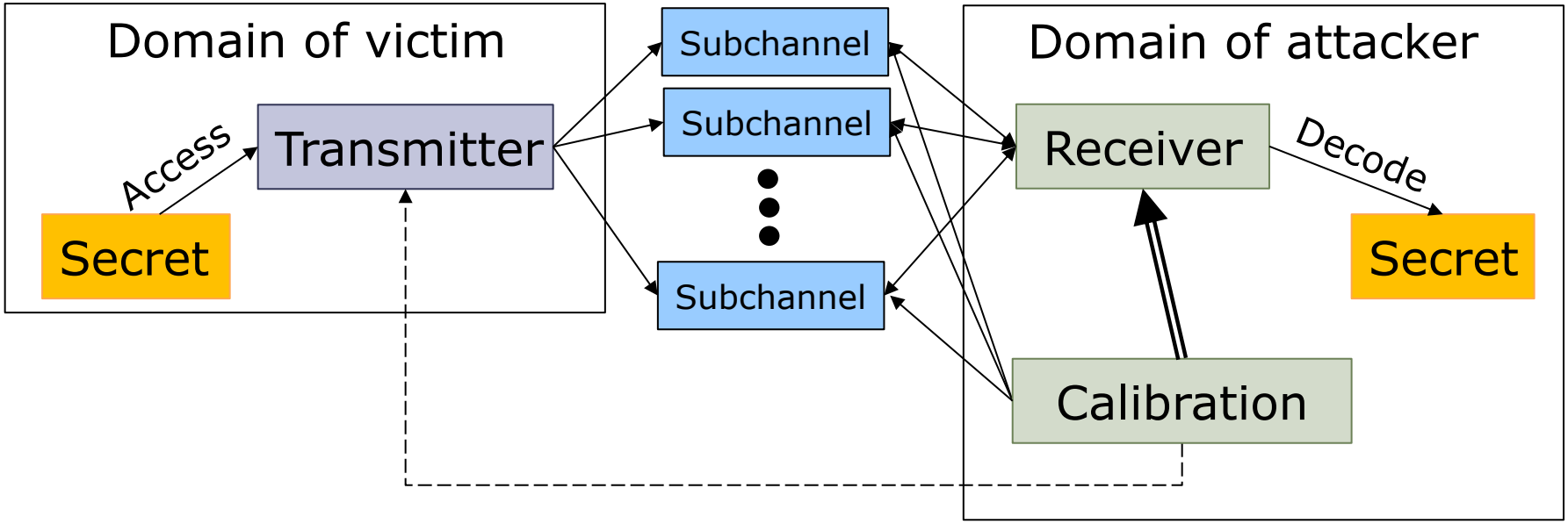
2. With a single hash, analysis by the receiver can, however, figure out which subchannel will be modulated.

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.

Receiver Calibration



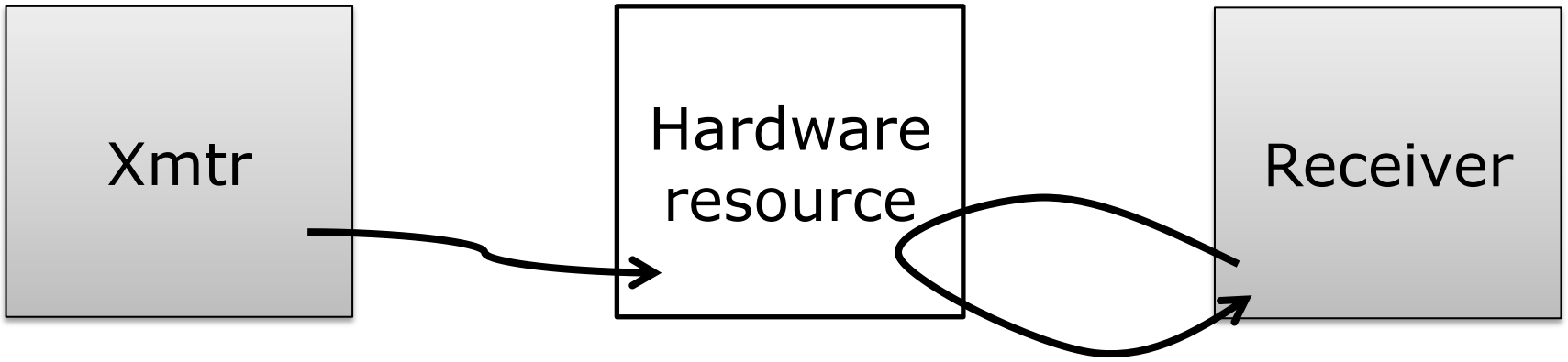
1. The calibration unit determines which subchannels (addresses) the receiver needs to use to detect modulation by a transmission
2. The receiver may just observe known transmissions by the transmitter to determine the subchannels to monitor
3. Or, the receiver may provoke the transmitter to make a particular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

Hashing* variations

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




```
if (send '1')
  Use resource
else
  idle
```

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

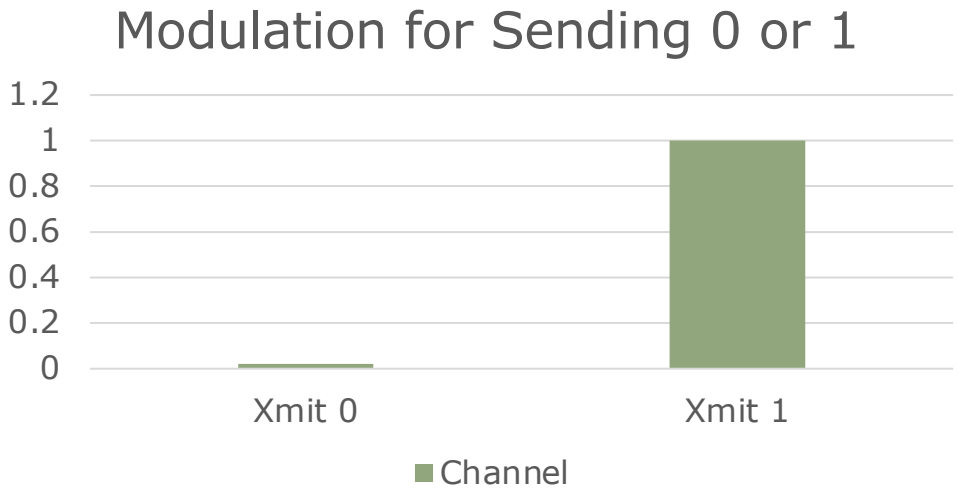
```
if (t2 – t1 > THRESH)
        read '1'
else
        read '0'
```

Types of State-based Channels

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

Simple Transmitter

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



Like an amplitude modulated (AM) radio transmission

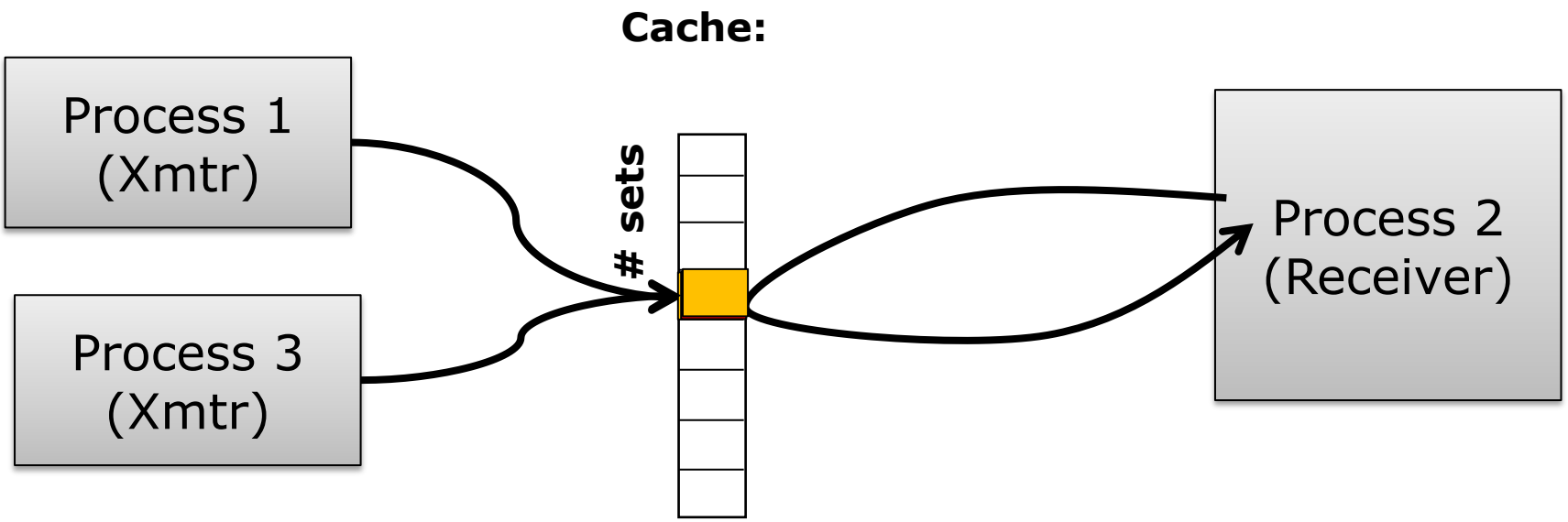
"AM" Transmitter in RSA [Percival 2005]

- Assume square-and-multiply based exponentiation

```
Input : base b, modulo m,
        exponent e = (e_{n-1} ...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
```

Secret-dependent memory access → transmitter

Noise in the channel



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

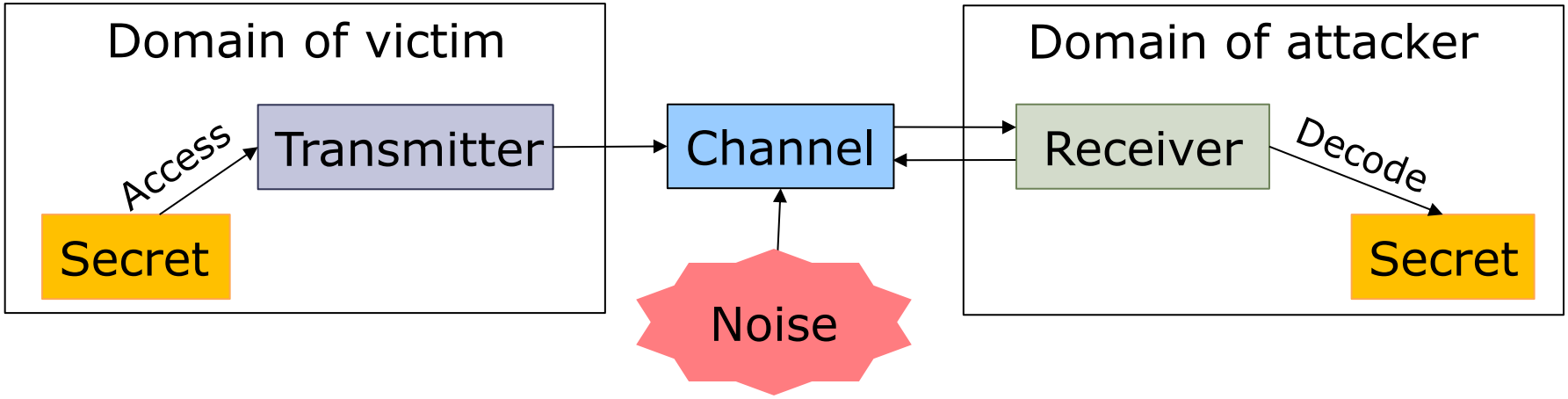
write to set

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

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

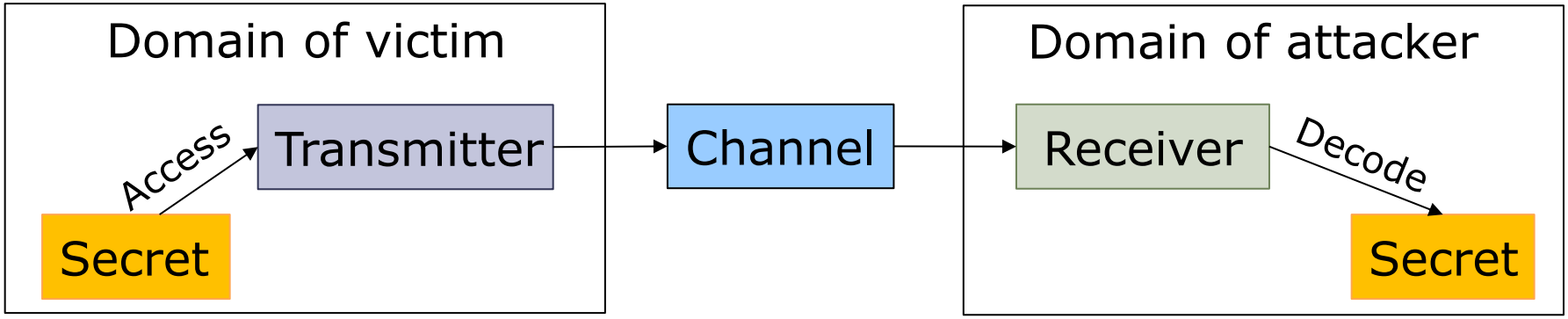
Receiver interprets "noise" as a signal!

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

Types of Transmitters

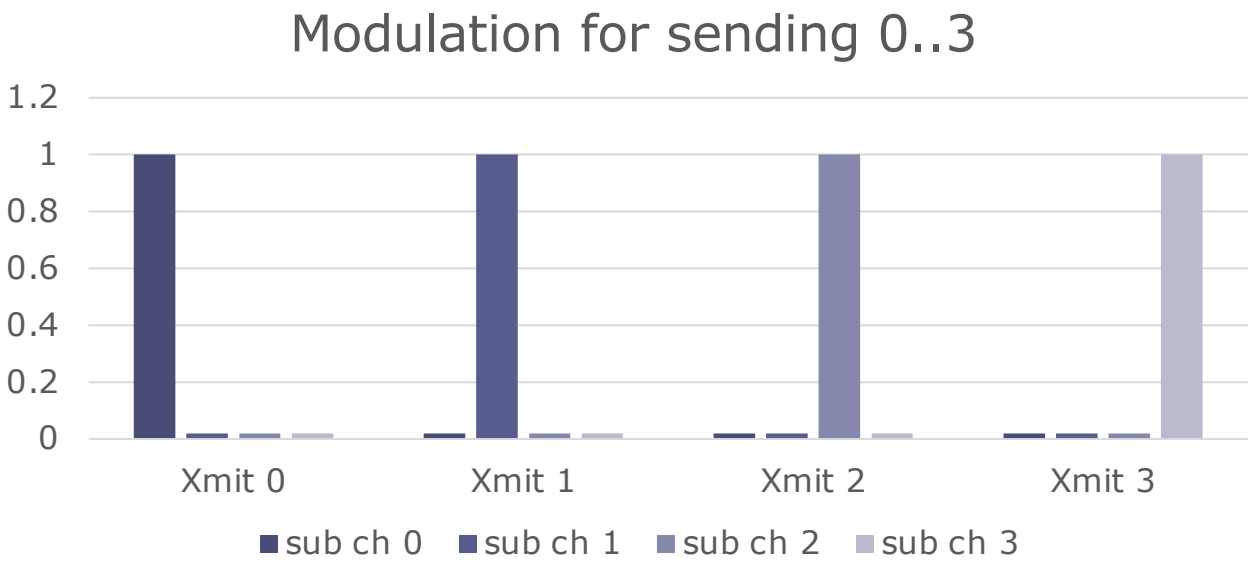


- Types of transmitter:
  1. Pre-existing so victim itself leaks secret, (e.g., RSA keys)

Another Transmitter

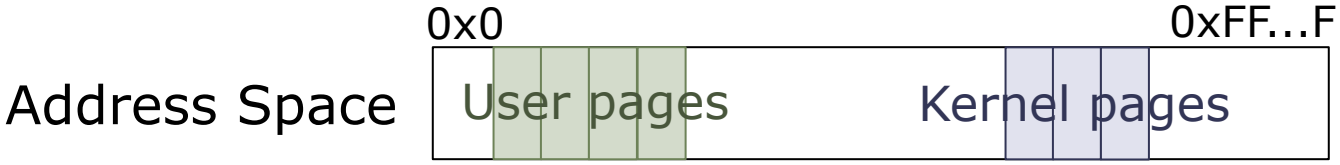
secret* = *oneof(0..3)

subchannel[***secret***] = 1



Like a frequency modulated (FM) radio transmission

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!

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

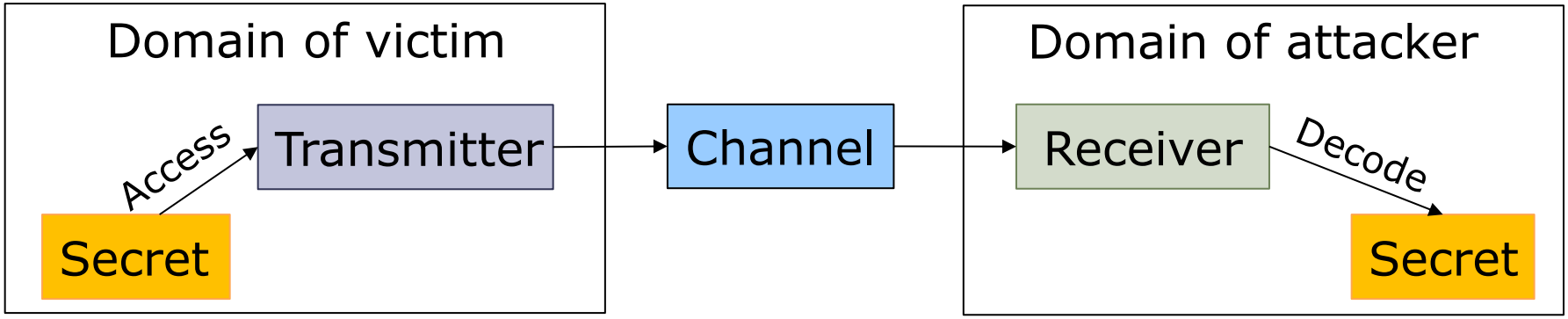
1. Preconditioning: Receiver allocates subchannels in `subchannels[256]` and flushes all its cache lines
2. Transmit: Transmitter (controlled by attacker) executes

```
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.

- 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



- 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 2
[Kocher et al. 2018]

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

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

- Interpret that code as an FM transmitter:

```
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]

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

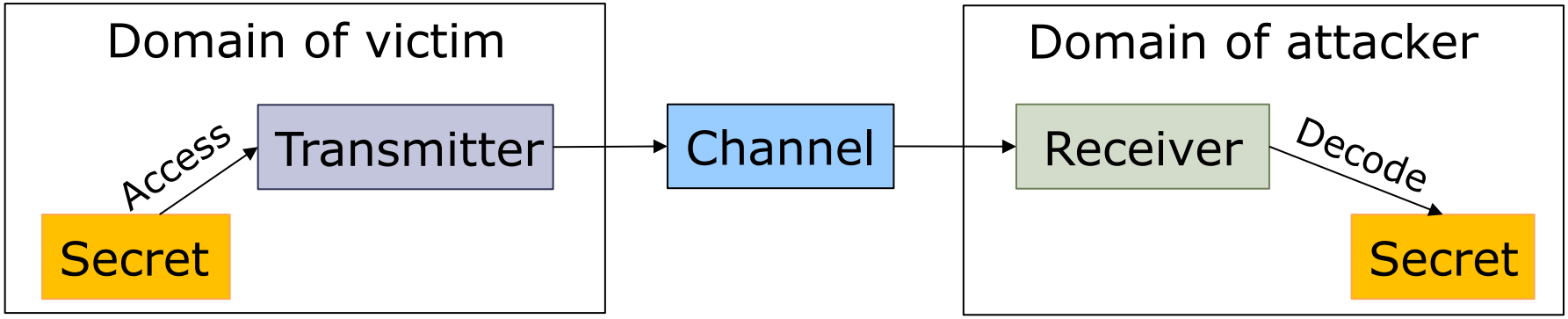
```
abc: br xyz
```

- 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)

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: 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
- 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!