Transient Side Channels

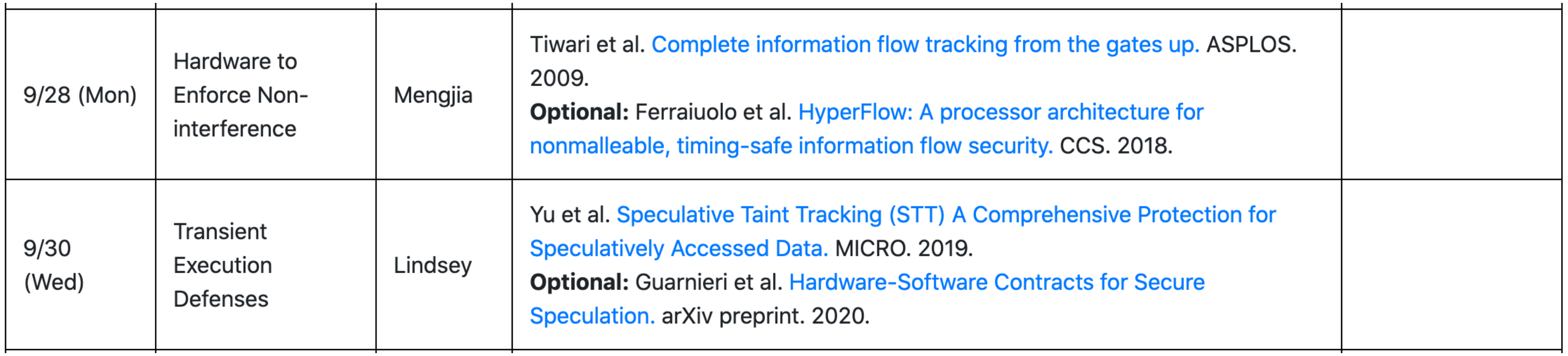
Mengjia Yan

Fall 2020

Based on slides from Christopher W. Fletcher

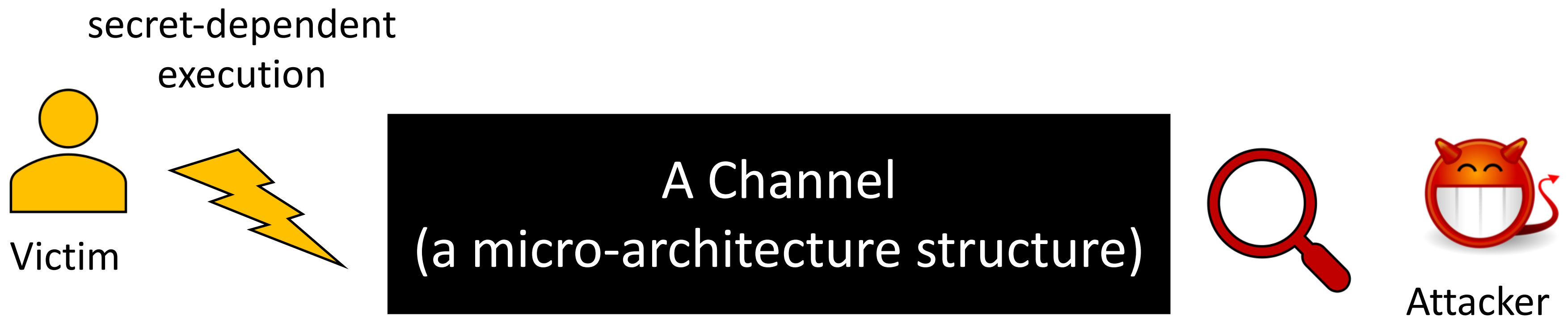
Reminder

- 1st paper review due midnight on 09/27 (before the next lecture)
- You will receive an invitation from HotCRP
  - https://mit-6888-fa20.hotcrp.com/



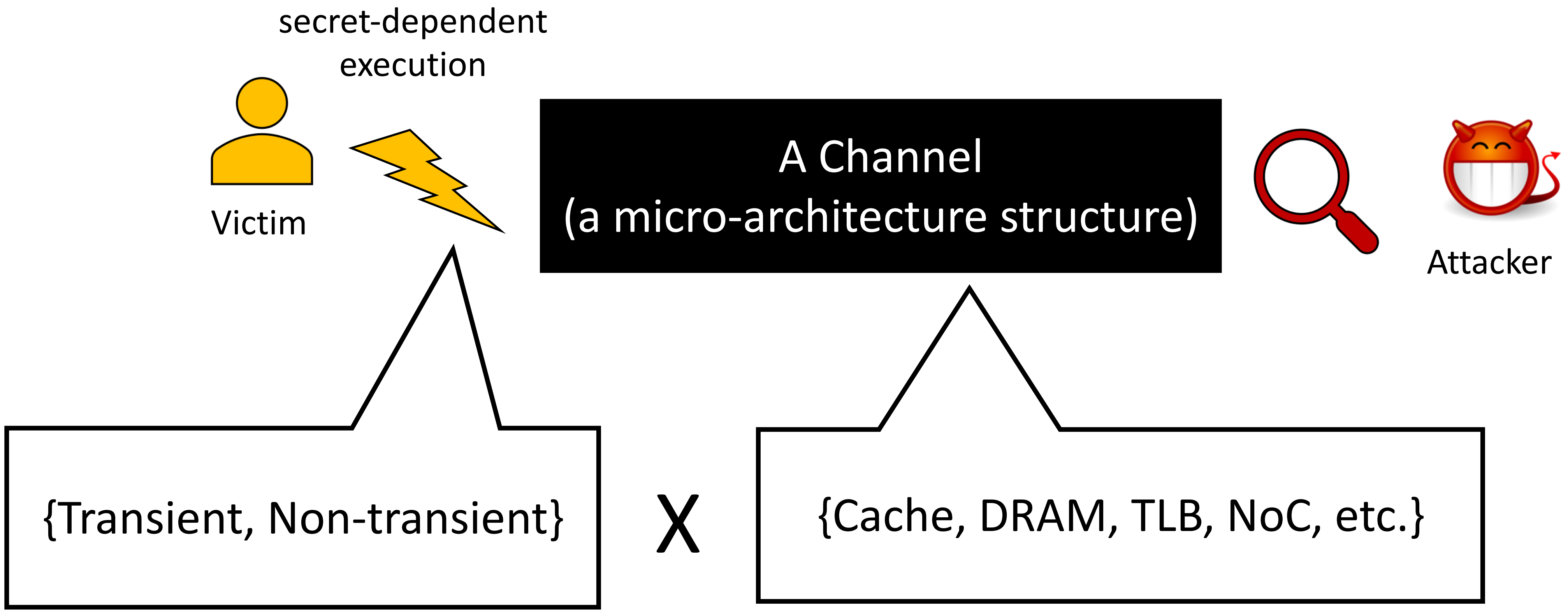
9/28 (Mon)	Hardware to Enforce Non-interference	Mengjia	Tiwari et al. Complete information flow tracking from the gates up. ASPLOS. 2009. **Optional:** Ferraiuolo et al. HyperFlow: A processor architecture for nonmalleable, timing-safe information flow security. CCS. 2018.	
9/30 (Wed)	Transient Execution Defenses	Lindsey	Yu et al. Speculative Taint Tracking (STT) A Comprehensive Protection for Speculatively Accessed Data. MICRO. 2019. **Optional:** Guarnieri et al. Hardware-Software Contracts for Secure Speculation. arXiv preprint. 2020.	

Micro-architecture Side Channels



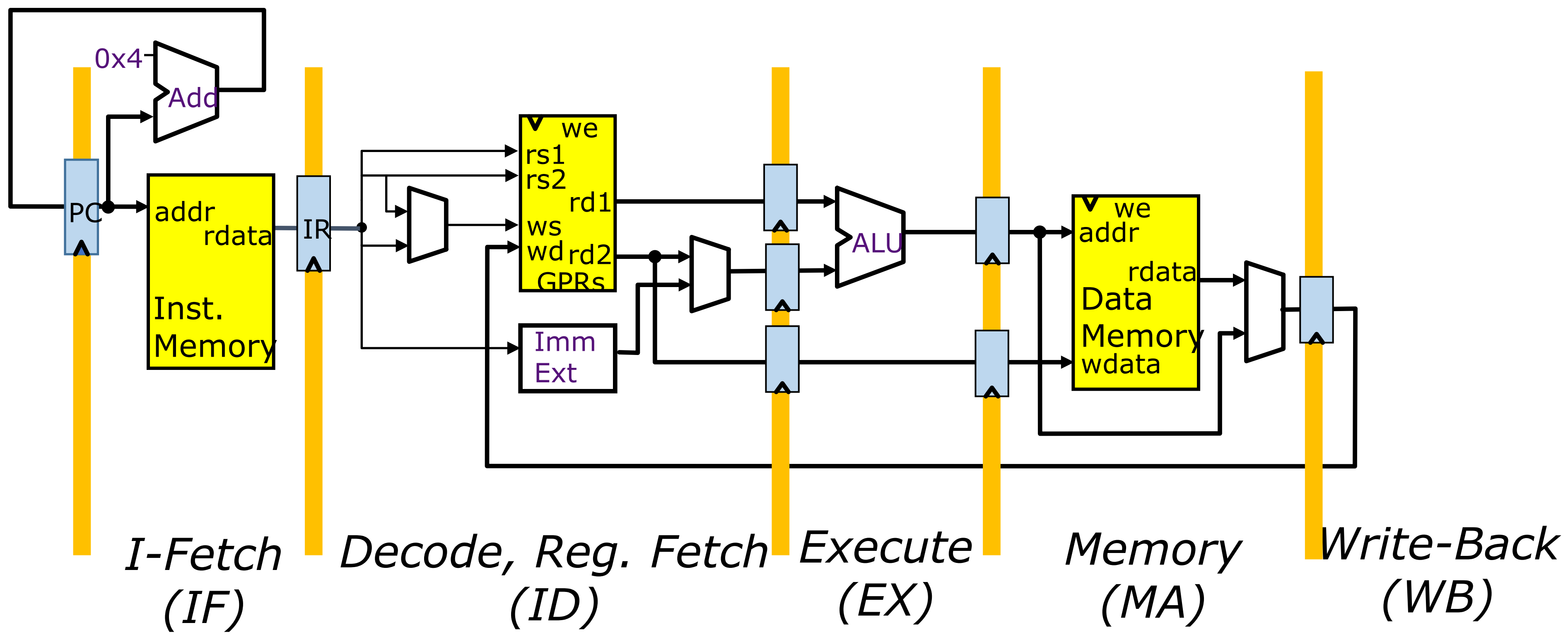
Kiriansky et al. DAWG: a defense against cache timing attacks in speculative execution processors. MICRO'18

Micro-architecture Side Channels

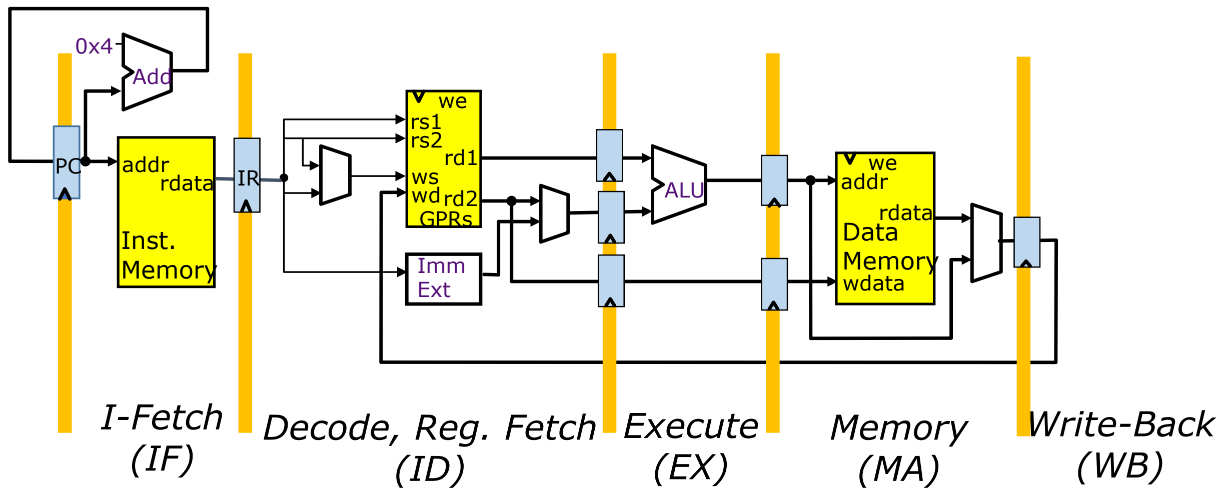

secret-dependent execution
Victim
A Channel
(a micro-architecture structure)
Attacker
{Transient, Non-transient}
X
{Cache, DRAM, TLB, NoC, etc.}

Kiriansky et al. DAWG: a defense against cache timing attacks in speculative execution processors. MICRO'18

Recap: 5-stage Pipeline

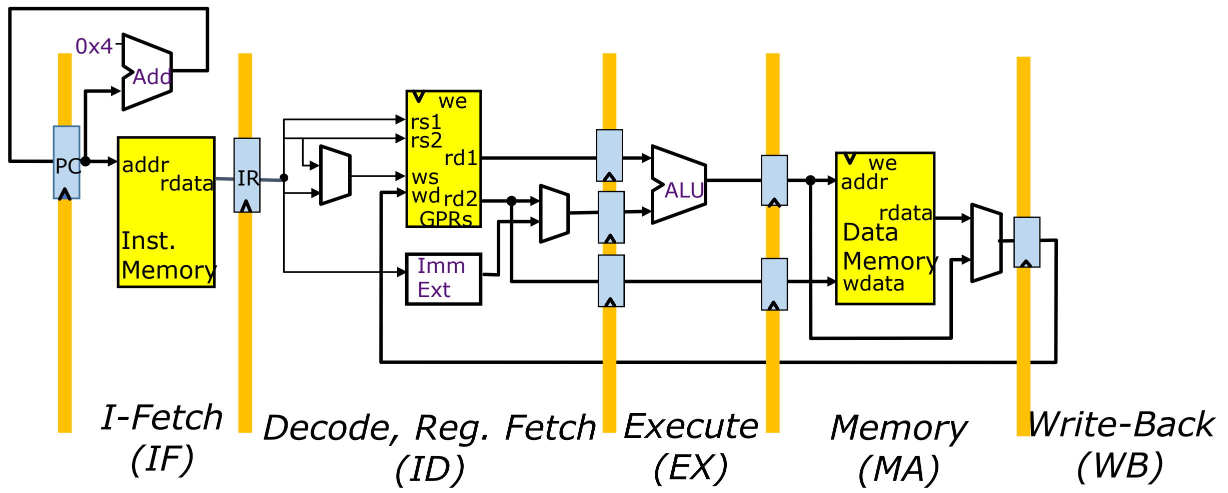


5-stage Pipeline



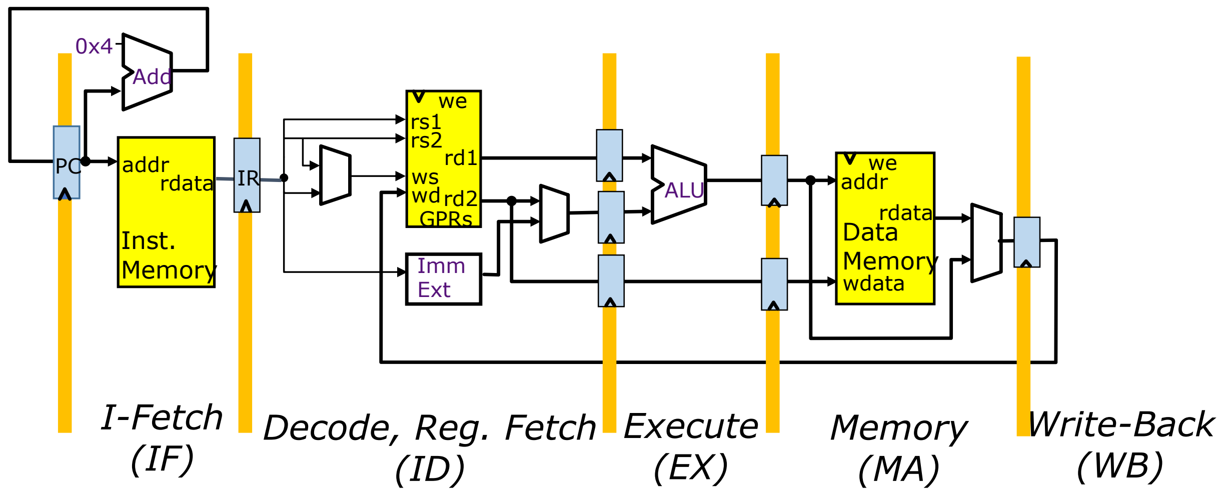
time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
instruction1	IF_1	ID_1	EX_1	MA_1	WB_1				
instruction2		IF_2	ID_2	EX_2	MA_2	WB_2			
instruction3			IF_3	ID_3	EX_3	MA_3	WB_3		
instruction4				IF_4	ID_4	EX_4	MA_4	WB_4	
instruction5					IF_5	ID_5	EX_5	MA_5	WB_5

5-stage Pipeline



time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
instruction1	IF_1	ID_1	EX_1	MA_1	WB_1				
instruction2		IF_2	ID_2	EX_2	MA_2	WB_2			
instruction3			IF_3	ID_3	EX_3	MA_3	WB_3		
instruction4				IF_4	ID_4	EX_4	MA_4	WB_4	
instruction5					IF_5	ID_5	EX_5	MA_5	WB_5

5-stage Pipeline



- In-order execution:
  - Execute instructions according to the program order

time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
instruction1	IF_1	ID_1	EX_1	MA_1	WB_1				
instruction2		IF_2	ID_2	EX_2	MA_2	WB_2			
instruction3			IF_3	ID_3	EX_3	MA_3	WB_3		
instruction4				IF_4	ID_4	EX_4	MA_4	WB_4	
instruction5					IF_5	ID_5	EX_5	MA_5	WB_5

Data Hazard and Control Hazard

	time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
Loop:	……									
	LD(R1, 0, R2)	IF_1	ID_1	EX_1	MA_1	WB_1				
	ADD(R2, 10, R3)		IF_2	ID_2	EX_2	MA_2	WB_2			
	BNE(R3, Loop)			IF_3	ID_3	EX_3	MA_3	WB_3		
	……									

Resolving Hazards

- Stall or Bypass

	time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
Loop:	……									
	LD(R1, 1, R2)	IF_1	ID_1	EX_1	MA_1	WB_1				
	ADD(R2, 10, R3)			IF_2	ID_2	EX_2	MA_2	WB_2		
	BNE(R3, Loop)				IF_3	ID_3	EX_3	MA_3	WB_3	
	……									

- Speculation (e.g., branch predictor)
  - Guess a value and continue executing anyway
  - When actual value is available, two cases
    - Guessed correctly → do nothing
    - Guessed incorrectly →restart with correct value (roll back)

Branch Predictor

- Predict Taken/Not taken
  - Not taken: PC+4
  - Taken: need to know target address

Branch Predictor

- Predict Taken/Not taken
  - Not taken: PC+4
  - Taken: need to know target address

- Predict target address
  - Branch target buffer (BTB)
  - Map <current PC, target PC>

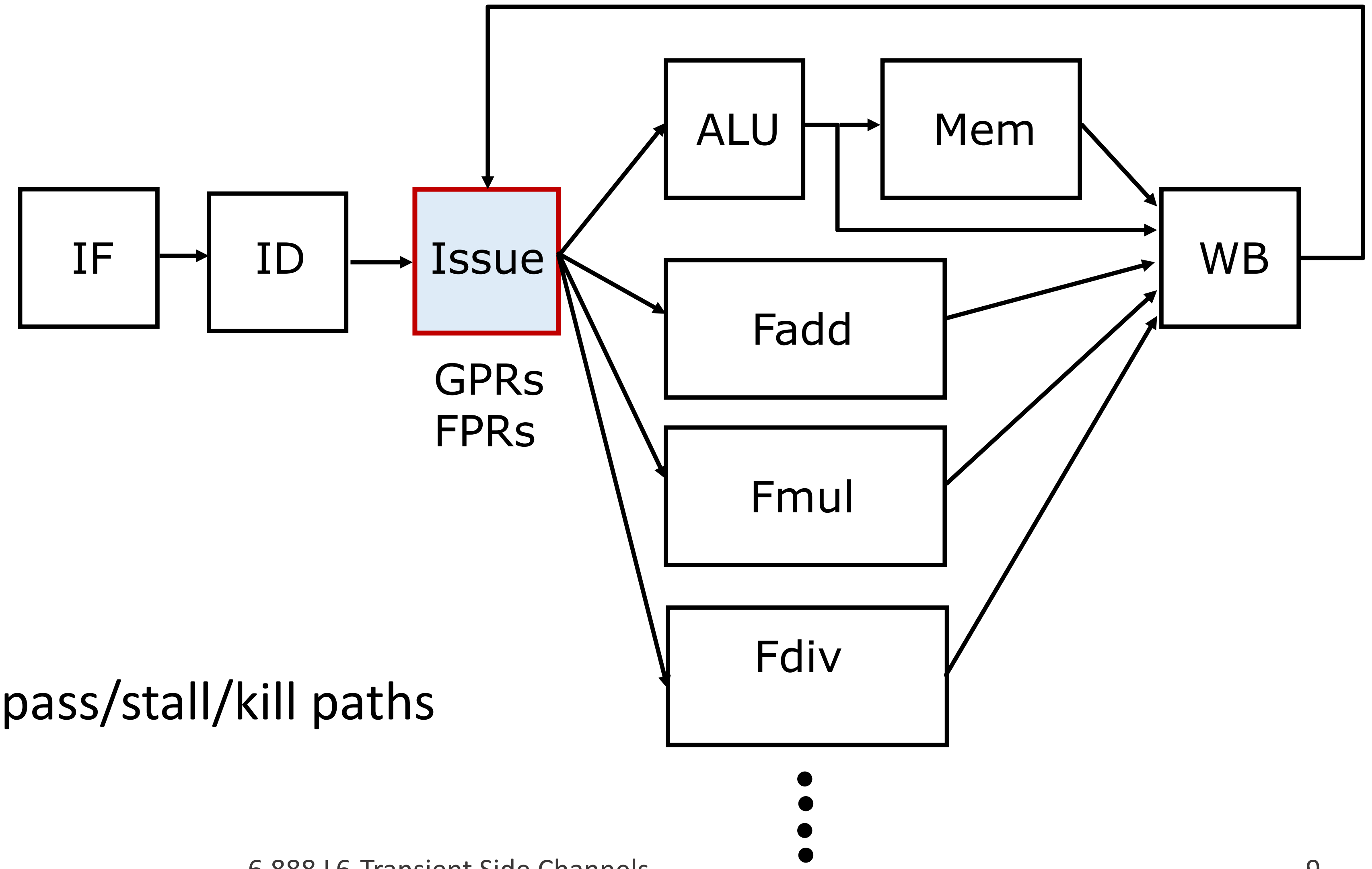
Branch Predictor

- Predict Taken/Not taken
  - Not taken: PC+4
  - Taken: need to know target address

- Predict target address
  - Branch target buffer (BTB)
  - Map <current PC, target PC>

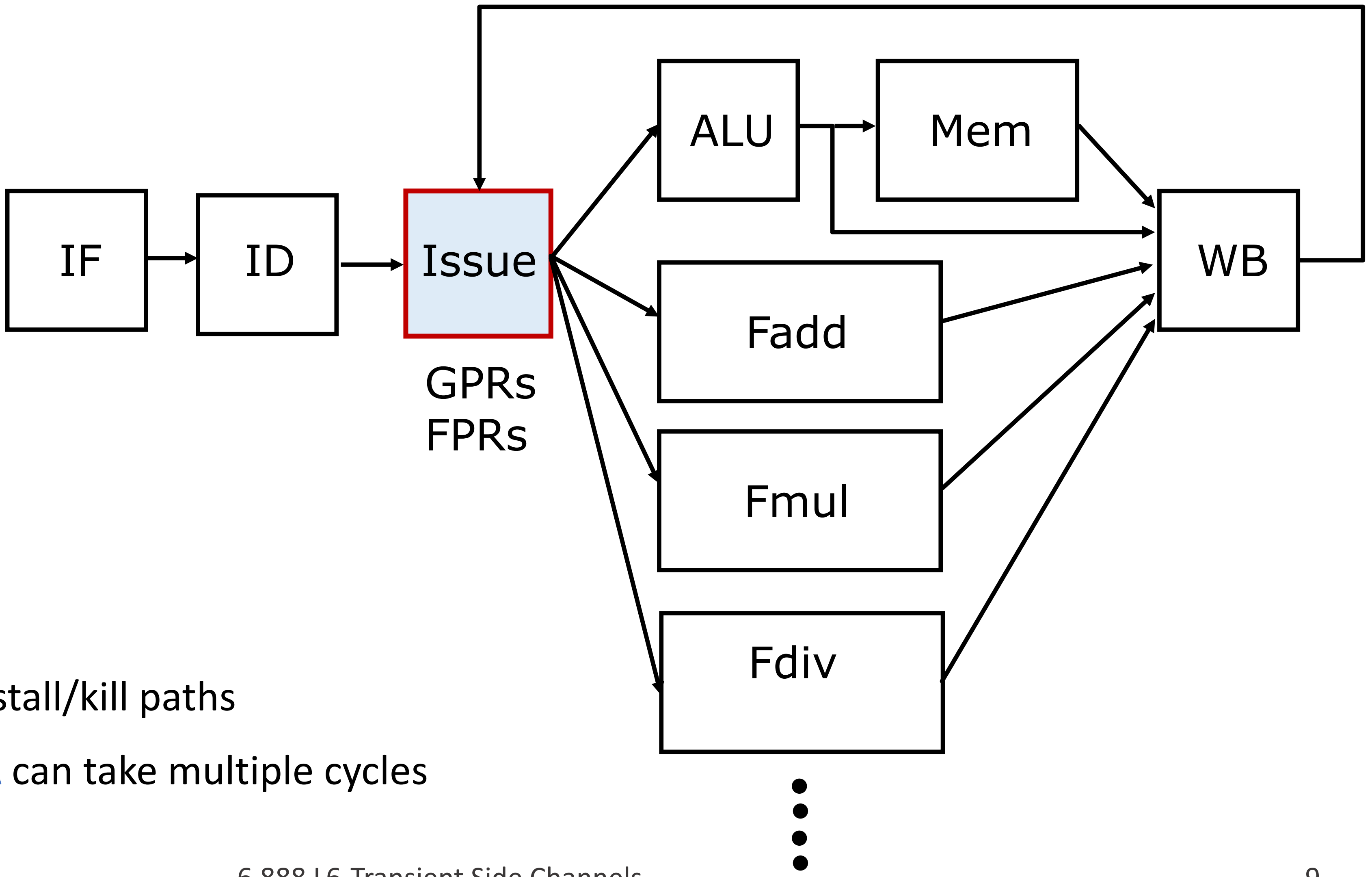
- Use history information to setup the predictor

Complex In-order Pipeline



- Need complex bypass/stall/kill paths

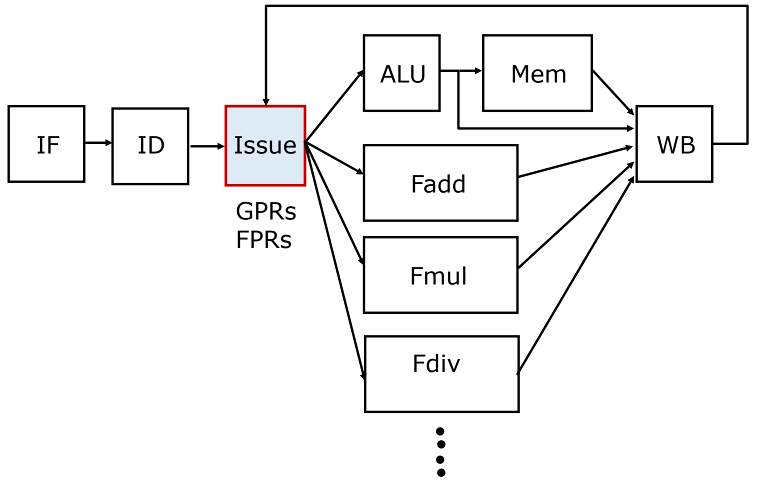
Complex In-order Pipeline



- Need complex bypass/stall/kill paths
- In real systems, **EX/MA** can take multiple cycles

Out-of-order Execution

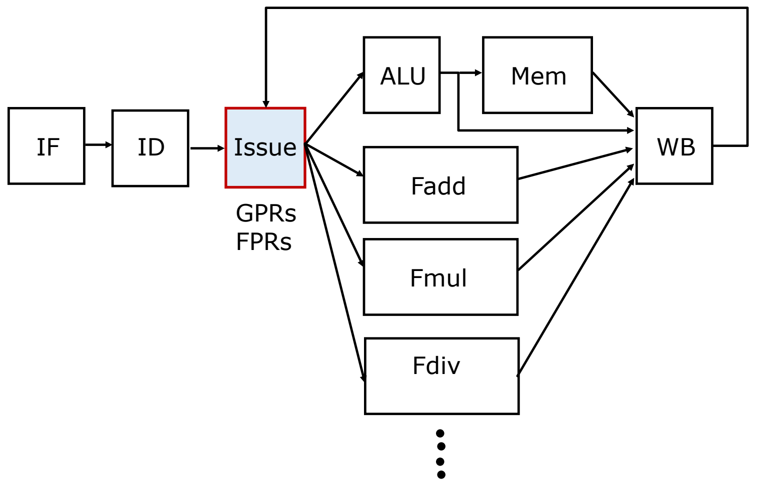
- When the pipeline is stalled, find something else to do



time t0 t1 t2 t3 t4 t5 t6 t7

Out-of-order Execution

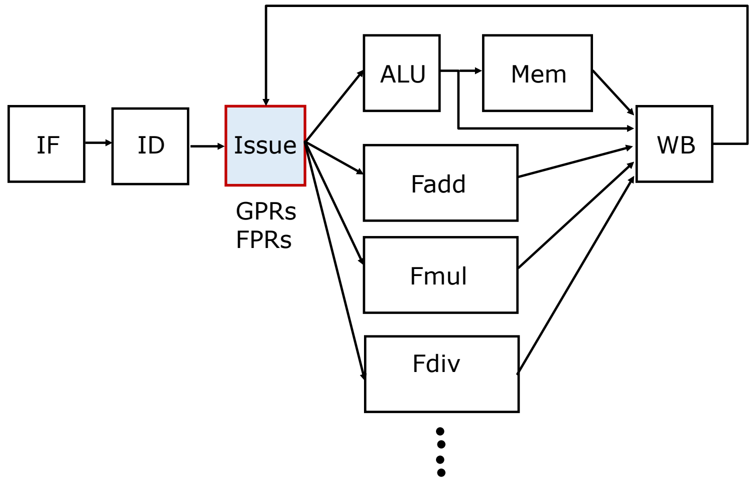
- When the pipeline is stalled, find something else to do



time	t0	t1	t2	t3	t4	t5	t6	t7
LD(R1, 1, R2)	IF_1	ID_1	EX_1	MA_1	MA_1	MA_1	MA_1	WB_1
ADD(**R3**, 10, R4)		IF_2	ID_2	EX_2	MA_2			WB_2
SUB(R4, 10, R5)			IF_3	ID_3	EX_3	MA_3		WB_3
……								

Out-of-order Execution

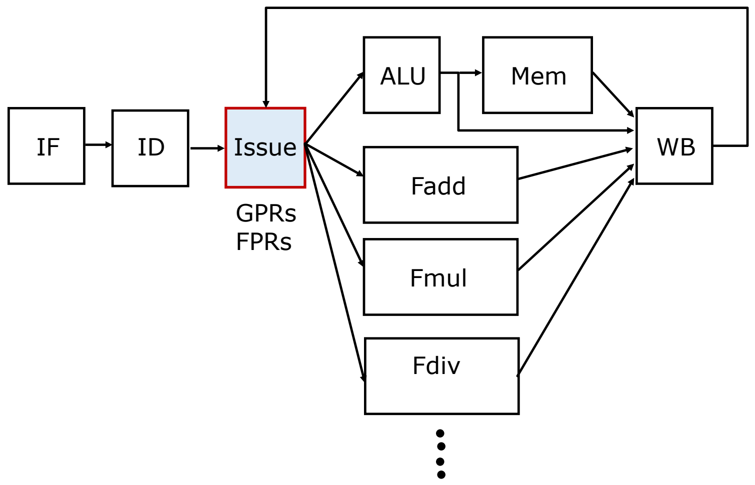
- When the pipeline is stalled, find something else to do
- When we do out-of-order execution, we are speculating that previous instructions do not cause exception



time	t0	t1	t2	t3	t4	t5	t6	t7
LD(R1, 1, R2)	IF$_1$	ID$_1$	EX$_1$	MA$_1$	MA$_1$	MA$_1$	MA$_1$	WB$_1$
ADD(**R3**, 10, R4)		IF$_2$	ID$_2$	EX$_2$	MA$_2$			WB$_2$
SUB(R4, 10, R5)			IF$_3$	ID$_3$	EX$_3$	MA$_3$		WB$_3$
……								

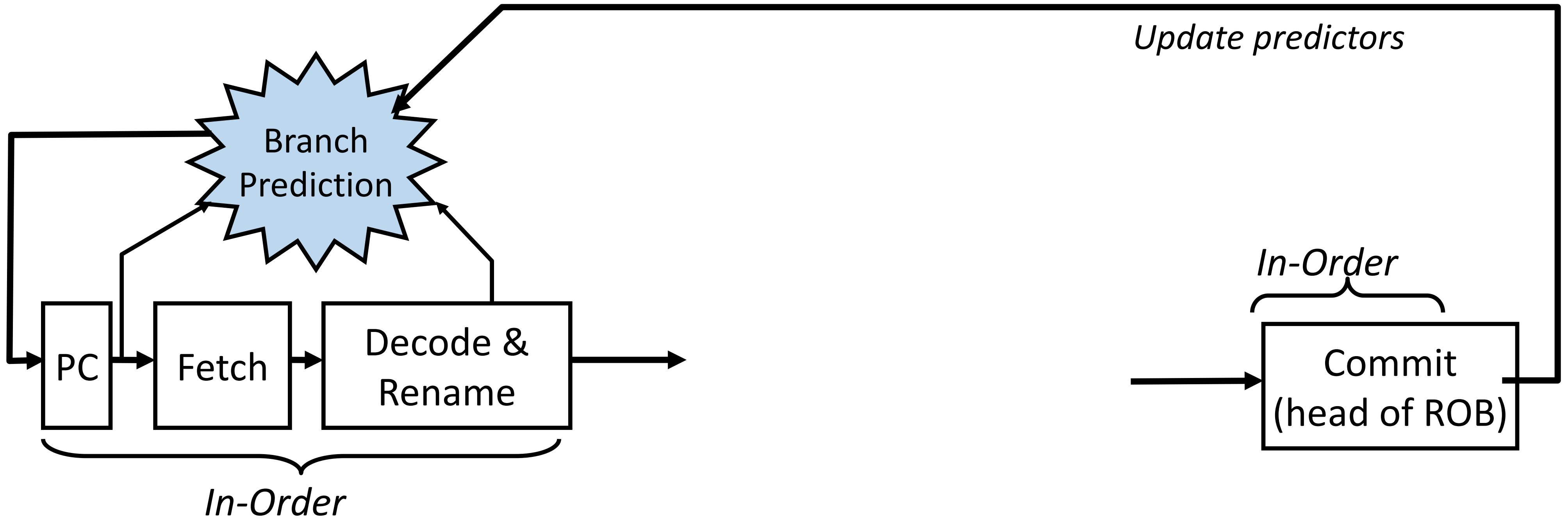
Out-of-order Execution

- When the pipeline is stalled, find something else to do
- When we do out-of-order execution, we are speculating that previous instructions do not cause exception
- If instruction *n* is speculative instruction, instruction *n+i* is also speculative

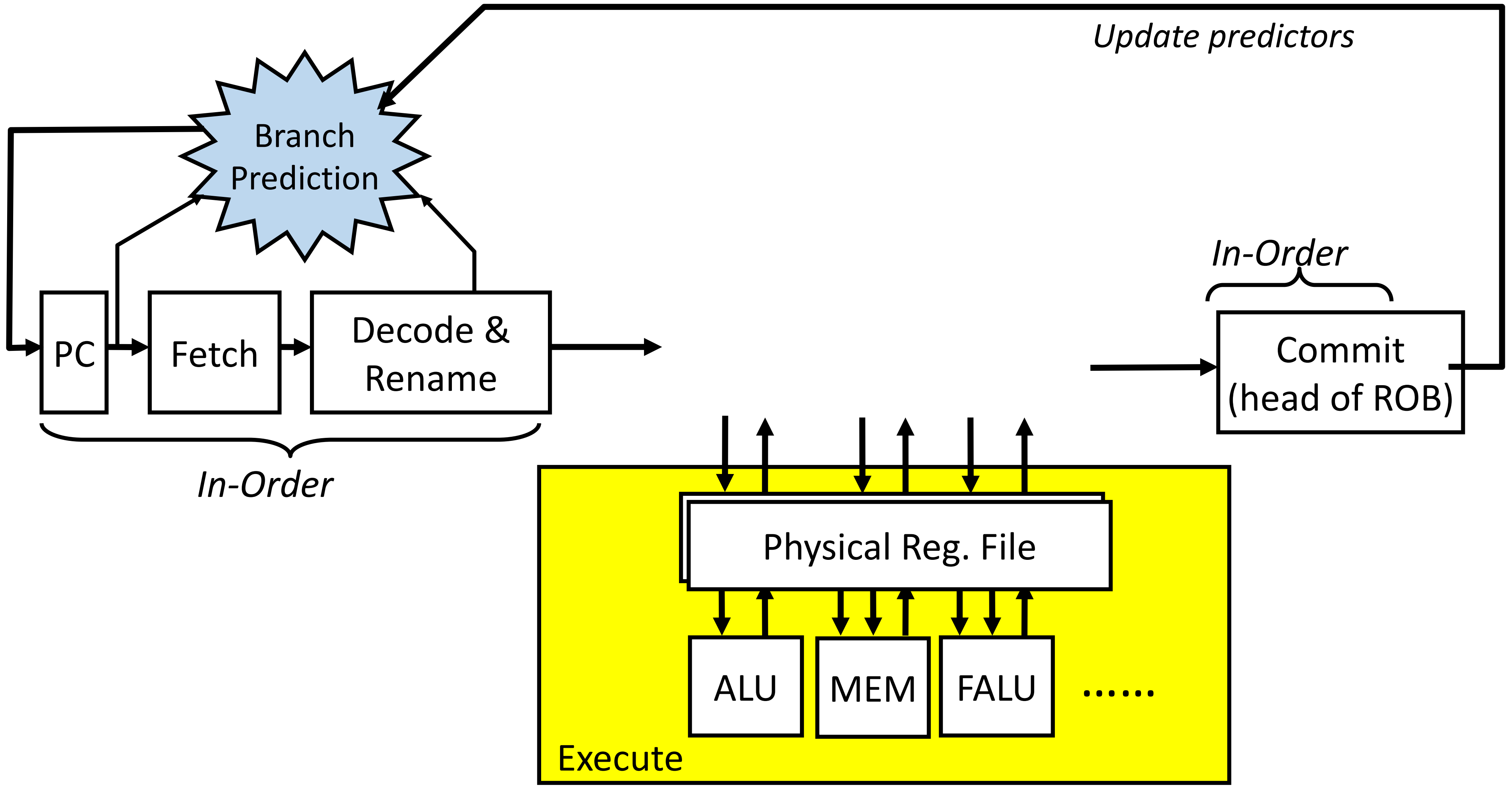


time	t0	t1	t2	t3	t4	t5	t6	t7
LD(R1, 1, R2)	IF_1	ID_1	EX_1	MA_1	MA_1	MA_1	MA_1	WB_1
ADD(**R3**, 10, R4)		IF_2	ID_2	EX_2	MA_2			WB_2
SUB(R4, 10, R5)			IF_3	ID_3	EX_3	MA_3		WB_3
……								

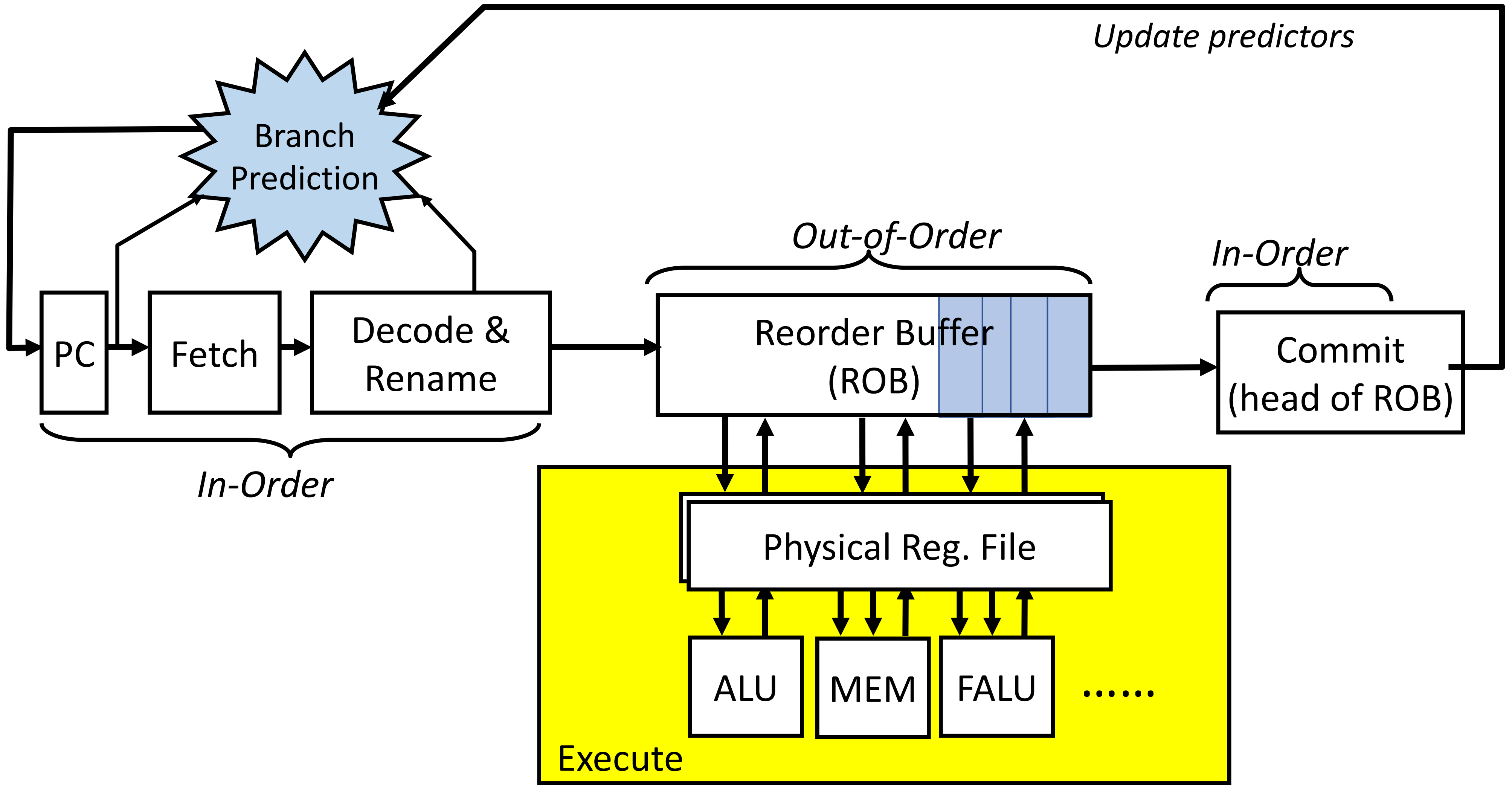
Speculative & Out-of-Order Execution



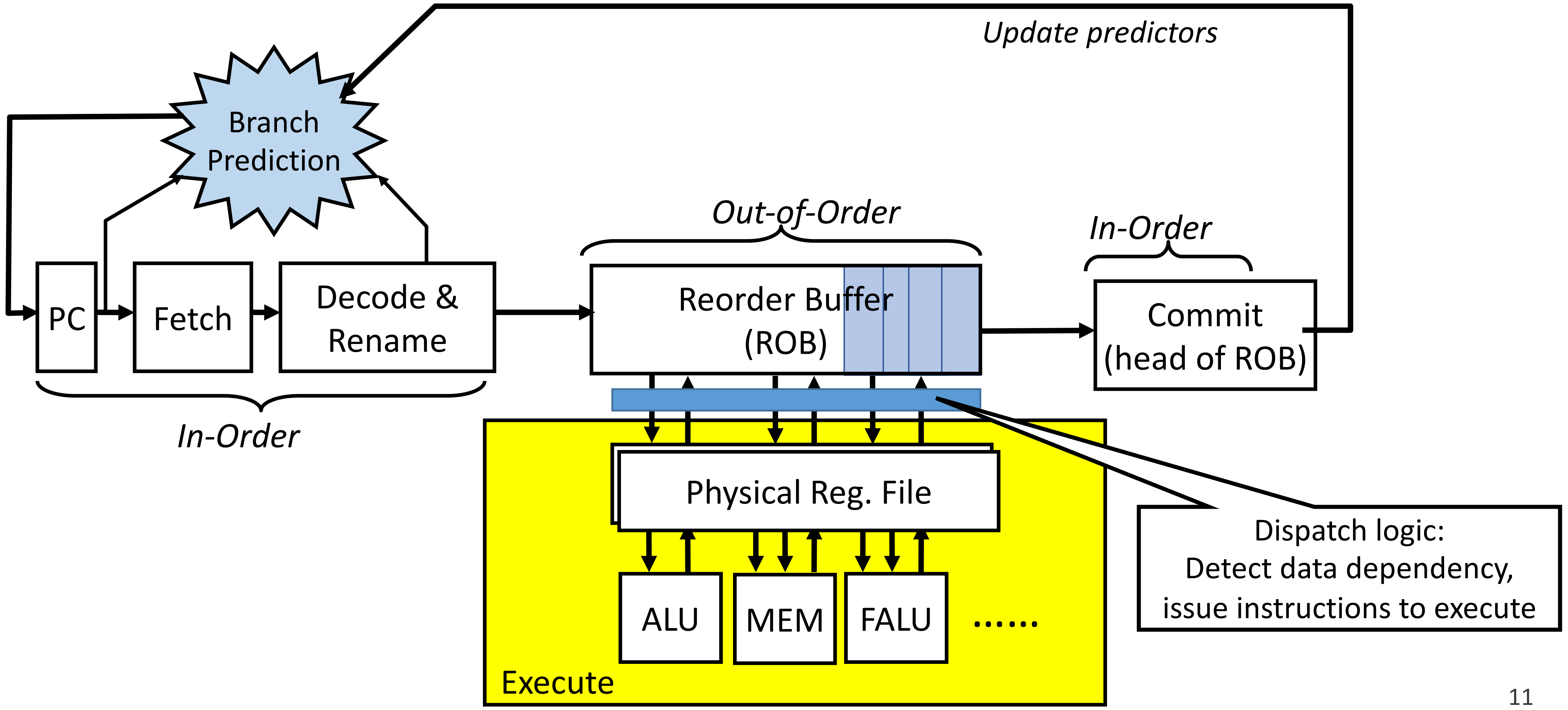
Speculative & Out-of-Order Execution



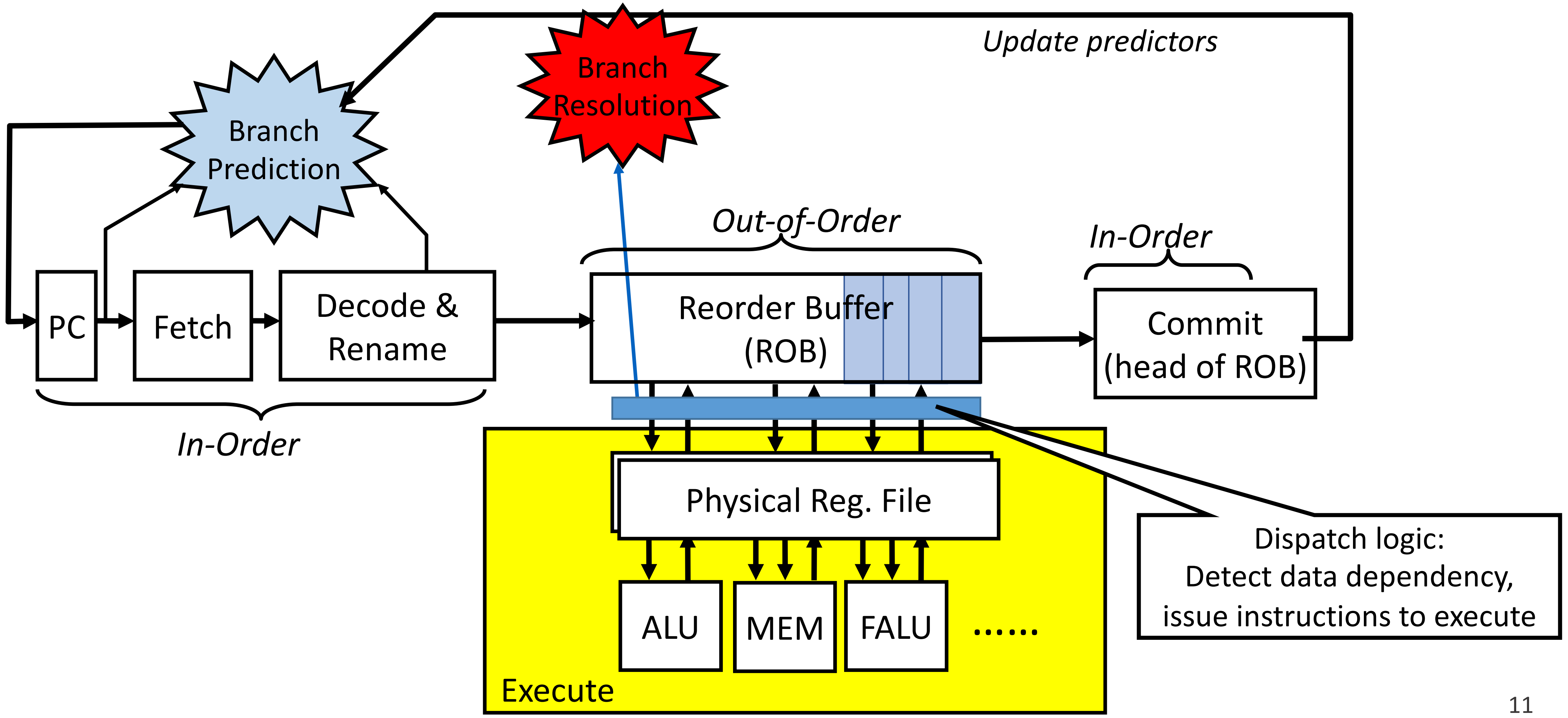
Speculative & Out-of-Order Execution



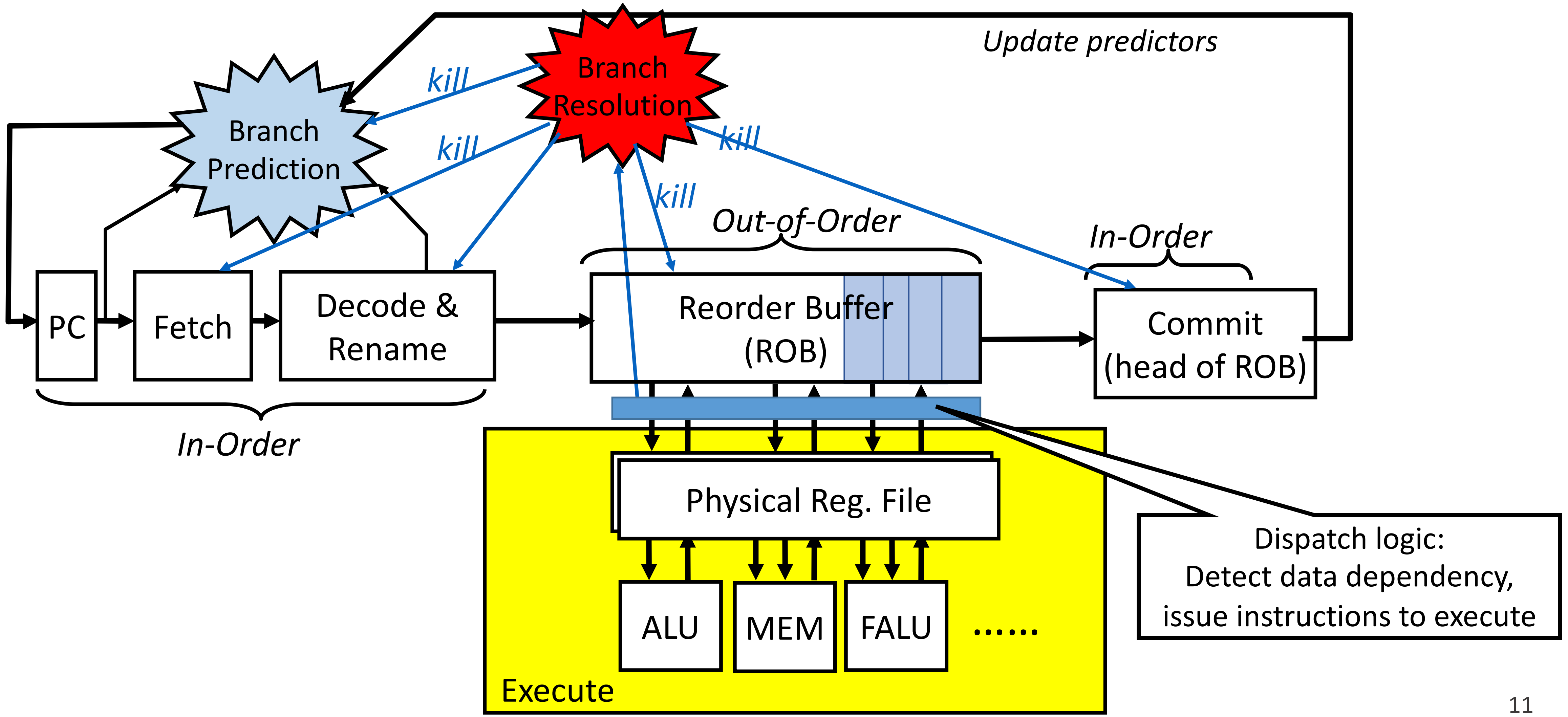
Speculative & Out-of-Order Execution



Speculative & Out-of-Order Execution



Speculative & Out-of-Order Execution



Terminology

A **speculative** instruction may squash.

- When executed, can change uArch state

Terminology

A **speculative** instruction may squash.

- When executed, can change uArch state

A **Transient** instruction *will* squash, i.e., will not commit.

A **Non-Transient** instruction will not squash, i.e., will eventually retire.

Terminology

A **speculative** instruction may squash.

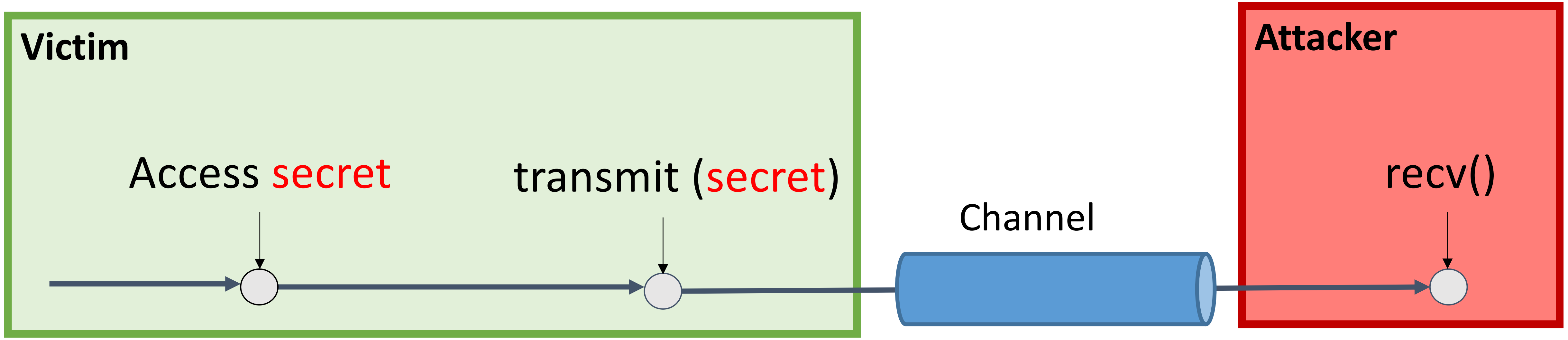
- When executed, can change uArch state

A **Transient** instruction *will* squash, i.e., will not commit.

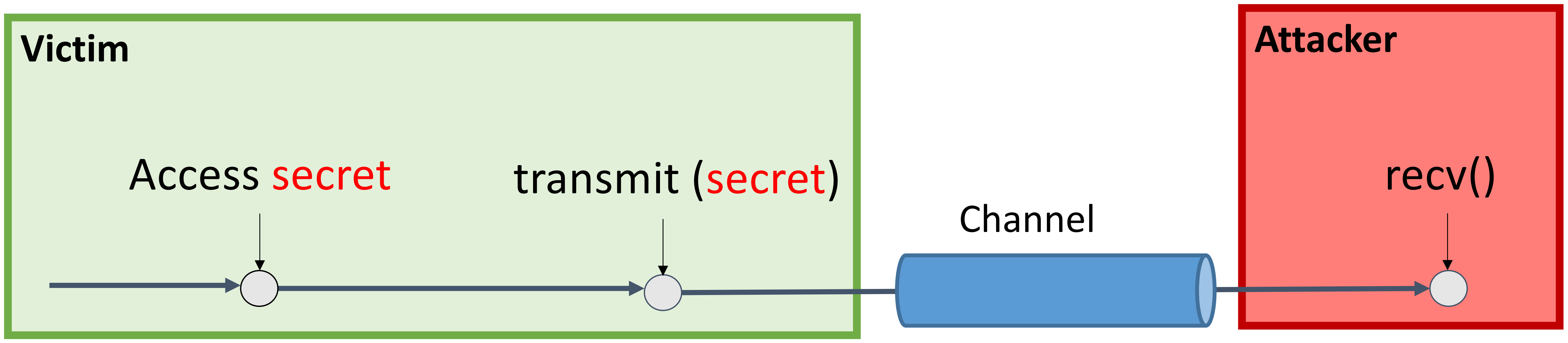
A **Non-Transient** instruction will not squash, i.e., will eventually retire.

That is, **transient instructions** are unreachable on a non-speculative microarchitecture.

General Attack Schema



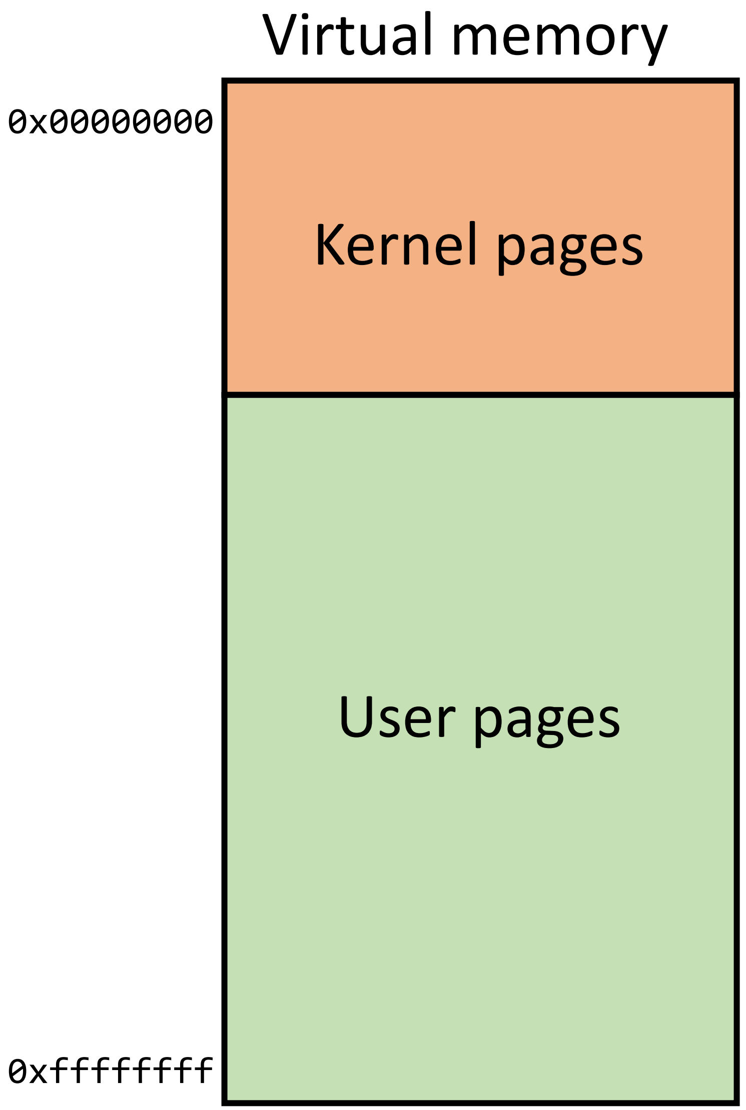
General Attack Schema



- The difference between transient and non-transient side channels
  - Whether the secret access or transmitter execution is transient

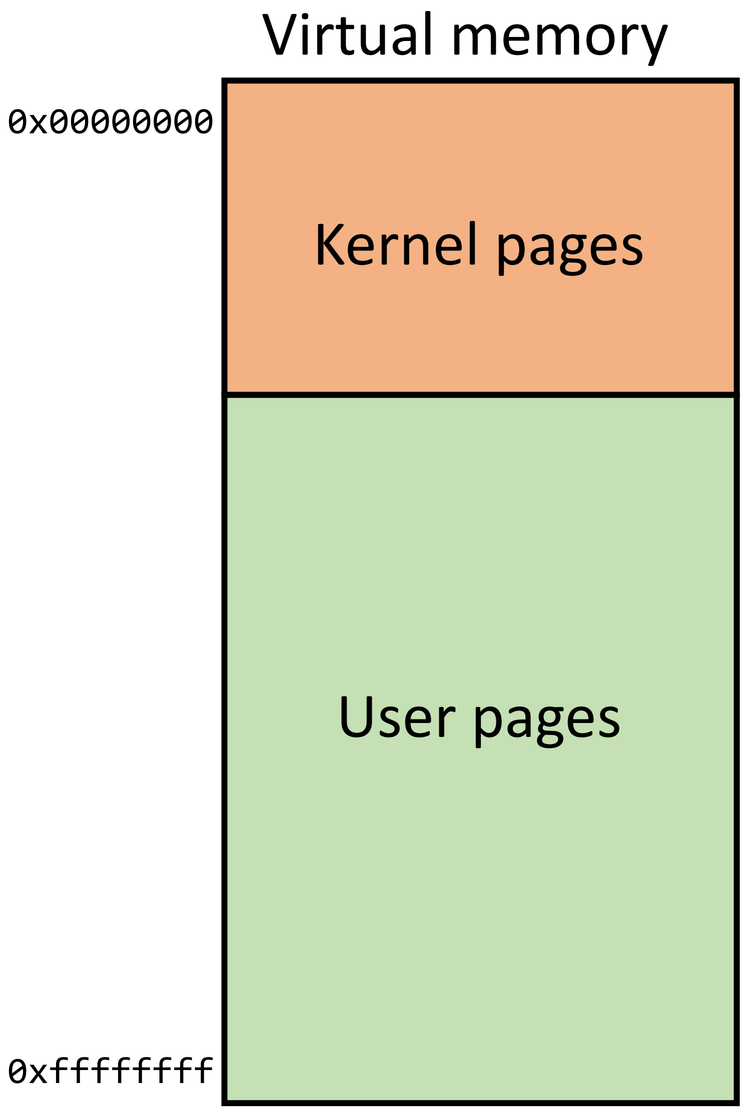
Meltdown & Spectre

Kernel/User Pages



- In x86, a process's virtual address space includes kernel pages, but kernel pages are only accessible in kernel mode
  - For performance purpose
  - Avoids switching page tables on context switches

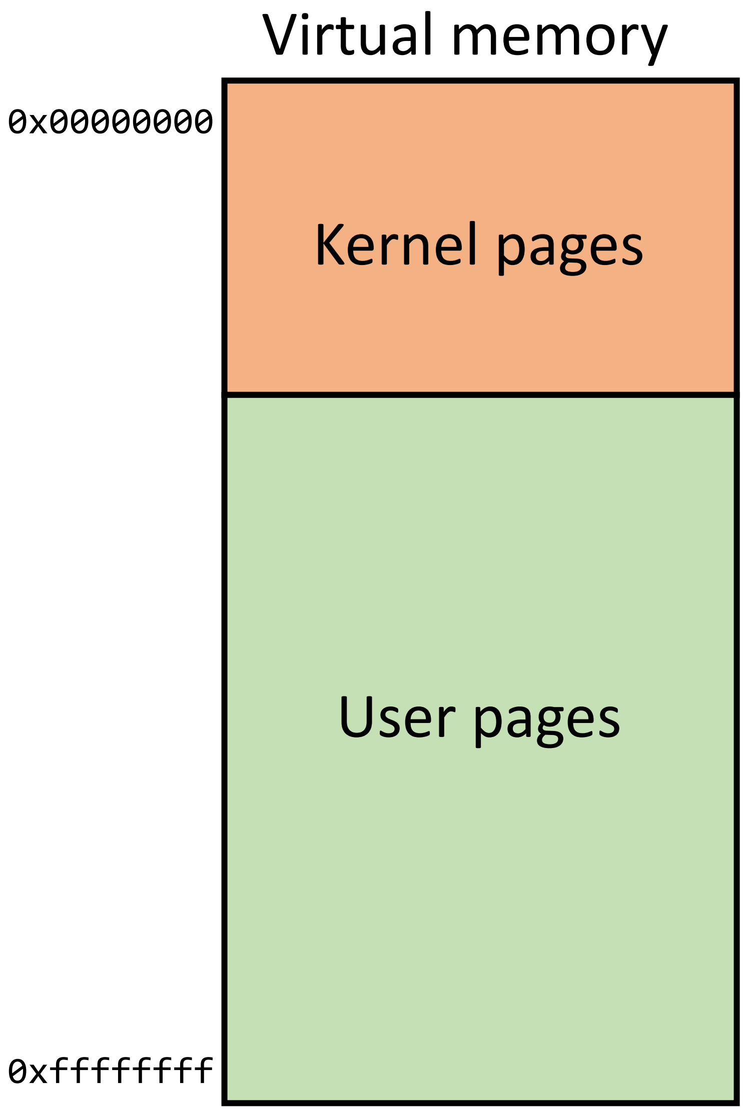
Kernel/User Pages



- In x86, a process's virtual address space includes kernel pages, but kernel pages are only accessible in kernel mode
  - For performance purpose
  - Avoids switching page tables on context switches

- What will happen if accessing kernel addresses in user mode?

Kernel/User Pages



- In x86, a process's virtual address space includes kernel pages, but kernel pages are only accessible in kernel mode
  - For performance purpose
  - Avoids switching page tables on context switches

- What will happen if accessing kernel addresses in user mode?
  - Protection fault

Meltdown

- Problem: Speculative instructions can change uArch state, e.g., cache

Meltdown

- Problem: Speculative instructions can change uArch state, e.g., cache

- Attack procedure

1. Setup: Attacker allocates probe_array, with 256 cache lines. Flushes all its cache lines
2. Transmit: Attacker executes

```
……
Ld1: uint8_t byte = *kernel_address;
Ld2: unit8_t dummy = probe_array[byte*64];
```

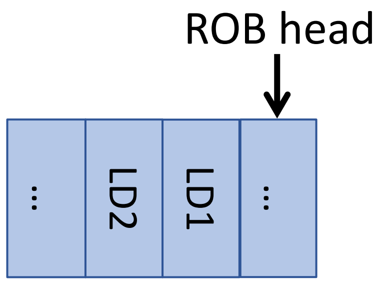
Meltdown

- Problem: Speculative instructions can change uArch state, e.g., cache

- Attack procedure

1. Setup: Attacker allocates probe_array, with 256 cache lines. Flushes all its cache lines
2. Transmit: Attacker executes

```
……
Ld1: uint8_t byte = *kernel_address;
Ld2: unit8_t dummy = probe_array[byte*64];
```



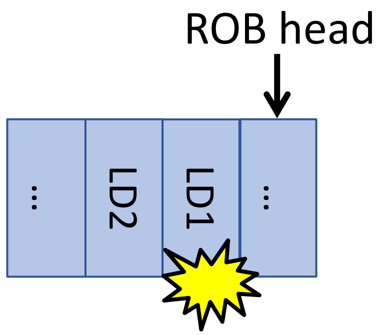
Meltdown

- Problem: Speculative instructions can change uArch state, e.g., cache

- Attack procedure

1. Setup: Attacker allocates probe_array, with 256 cache lines. Flushes all its cache lines
2. Transmit: Attacker executes

```
……
Ld1: uint8_t byte = *kernel_address;
Ld2: unit8_t dummy = probe_array[byte*64];
```



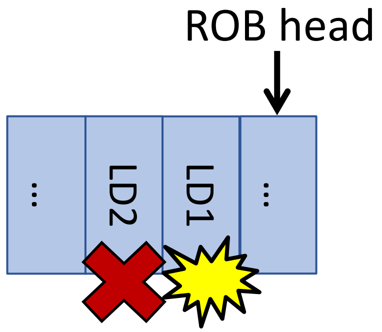
Meltdown

- Problem: Speculative instructions can change uArch state, e.g., cache

- Attack procedure

1. Setup: Attacker allocates probe_array, with 256 cache lines. Flushes all its cache lines
2. Transmit: Attacker executes

```
……
Ld1: uint8_t byte = *kernel_address;
Ld2: unit8_t dummy = probe_array[byte*64];
```



Meltdown

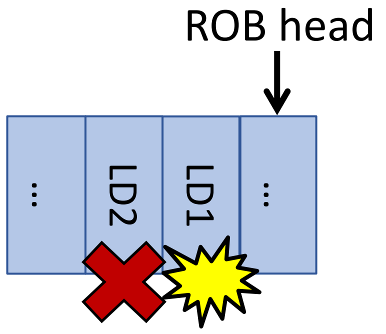
Exception handling is deferred when the instruction reaches the head of ROB.

- Problem: Speculative instructions can change uArch state, e.g., ca

- Attack procedure

1. Setup: Attacker allocates probe_array, with 256 cache lines. Flu es all its cache lines
2. Transmit: Attacker executes

```
……
Ld1: uint8_t byte = *kernel_address;
Ld2: unit8_t dummy = probe_array[byte*64];
```



Meltdown

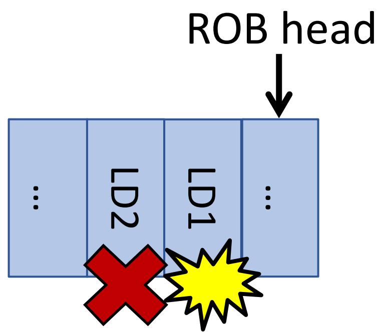
Exception handling is deferred when the instruction reaches the head of ROB.

- Problem: Speculative instructions can change uArch state, e.g., ca

- Attack procedure

1. Setup: Attacker allocates probe_array, with 256 cache lines. Flu es all its cache lines
2. Transmit: Attacker executes

```
……
Ld1: uint8_t byte = *kernel_address;
Ld2: unit8_t dummy = probe_array[byte*64];
```



3. Receive: After handling protection fault, attacker performs cache side channel attack to figure out which line of probe_array is accessed → recovers byte

Meltdown Type Attacks

- Can be used to read arbitrary memory
- Leaks across privilege levels
  - OS ←→ Application
  - SGX ←→ Application (e.g., Foreshadow)
  - Etc

Meltdown Type Attacks

- Can be used to read arbitrary memory
- Leaks across privilege levels
  - OS ←→ Application
  - SGX ←→ Application (e.g., Foreshadow)
  - Etc

- Mitigations:
  - Stall speculation
  - Register poisoning

Meltdown Type Attacks

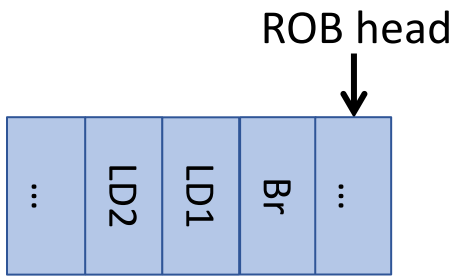
- Can be used to read arbitrary memory
- Leaks across privilege levels
  - OS ←→ Application
  - SGX ←→ Application (e.g., Foreshadow)
  - Etc

- Mitigations:
  - Stall speculation
  - Register poisoning
- We generally consider it as a design bug

Spectre Variant 1 – Exploit Branch Condition

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

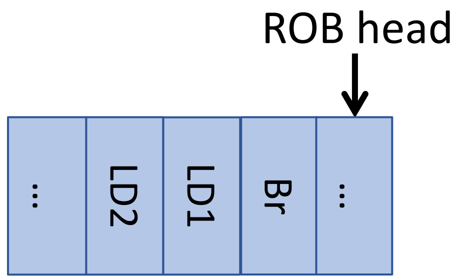
```
Br:  if (x < size_array1) {
Ld1:        secret = array1[x]*64
Ld2:        y = array2[secret]
     }
```



Spectre Variant 1 – Exploit Branch Condition

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

```
Br:  if (x < size_array1) {
Ld1:       secret = array1[x]*64
Ld2:       y = array2[secret]
     }
```



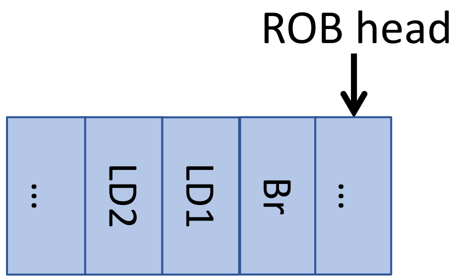
Attacker to read arbitrary memory:

1. Setup: Train branch predictor

Spectre Variant 1 – Exploit Branch Condition

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

```
Br:  if (x < size_array1) {
Ld1:       secret = array1[x]*64
Ld2:       y = array2[secret]
     }
```

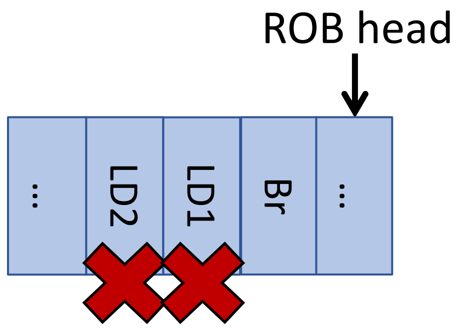


Attacker to read arbitrary memory:
1. Setup: Train branch predictor
2. Transmit: Trigger branch misprediction; *&array1[x]* maps to some desired kernel address

Spectre Variant 1 – Exploit Branch Condition

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

```
Br:  if (x < size_array1) {
Ld1:       secret = array1[x]*64
Ld2:        y = array2[secret]
     }
```

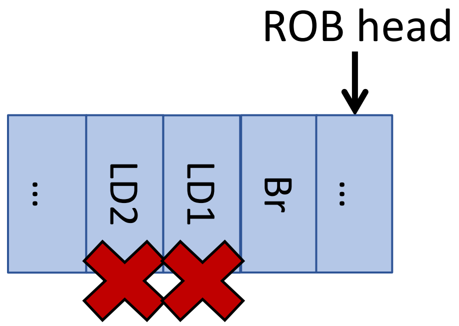


Attacker to read arbitrary memory:
1. Setup: Train branch predictor
2. Transmit: Trigger branch misprediction; *&array1[x]* maps to some desired kernel address

Spectre Variant 1 – Exploit Branch Condition

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

```
Br:  if (x < size_array1) {
Ld1:       secret = array1[x]*64
Ld2:       y = array2[secret]
     }
```



Attacker to read arbitrary memory:

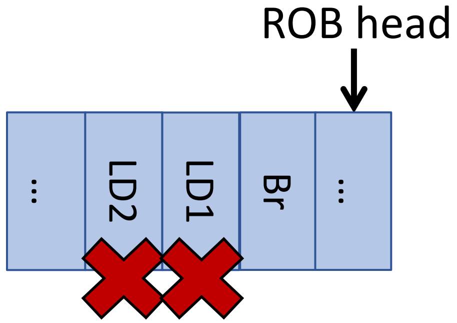
1. Setup: Train branch predictor
2. Transmit: Trigger branch misprediction; *&array1[x]* maps to some desired kernel address
3. Receive: Attacker probes cache to infer which line of *array2* was fetched

Spectre Variant 1 – Exploit Branch Condition

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

```
Br:  if (x < size_array1) {
Ld1:       secret = array1[x]*64
Ld2:       y = array2[secret]
     }
```

Always malicious?



Attacker to read arbitrary memory:
1. Setup: Train branch predictor
2. Transmit: Trigger branch misprediction; `&array1[x]` maps to some desired kernel address
3. Receive: Attacker probes cache to infer which line of `array2` was fetched

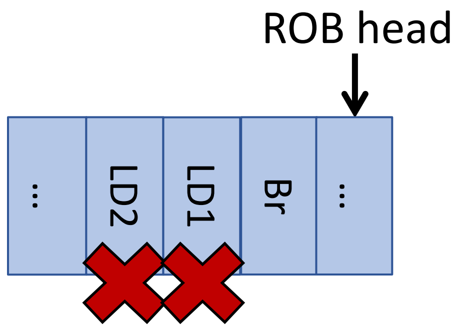
Spectre Variant 1 – Exploit Branch Condition

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

```
Br:  if (x < size_array1) {
Ld1:       secret = array1[x]*64
Ld2:        y = array2[secret]
     }
```

Always malicious?

No. It may be a benign misprediction.



Attacker to read arbitrary memory:

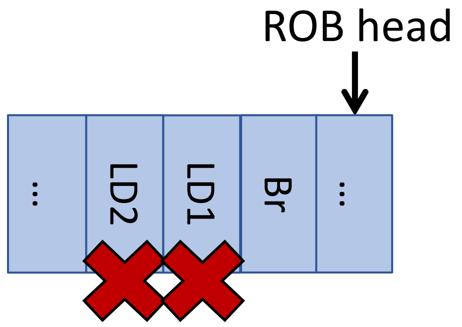
1. Setup: Train branch predictor
2. Transmit: Trigger branch misprediction; *&array1[x]* maps to some desired kernel address
3. Receive: Attacker probes cache to infer which line of *array2* was fetched

Spectre Variant 1 – Exploit Branch Condition

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

```
Br:  if (x < size_array1) {
Ld1:       secret = array1[x]*64
Ld2:        y = array2[secret]
     }
```

Always malicious?
No. It may be a benign misprediction.
We do not consider Spectre as a bug.



Attacker to read arbitrary memory:
1. Setup: Train branch predictor
2. Transmit: Trigger branch misprediction; `&array1[x]` maps to some desired kernel address
3. Receive: Attacker probes cache to infer which line of `array2` was fetched

Spectre Variant 2 – Exploit Branch Target

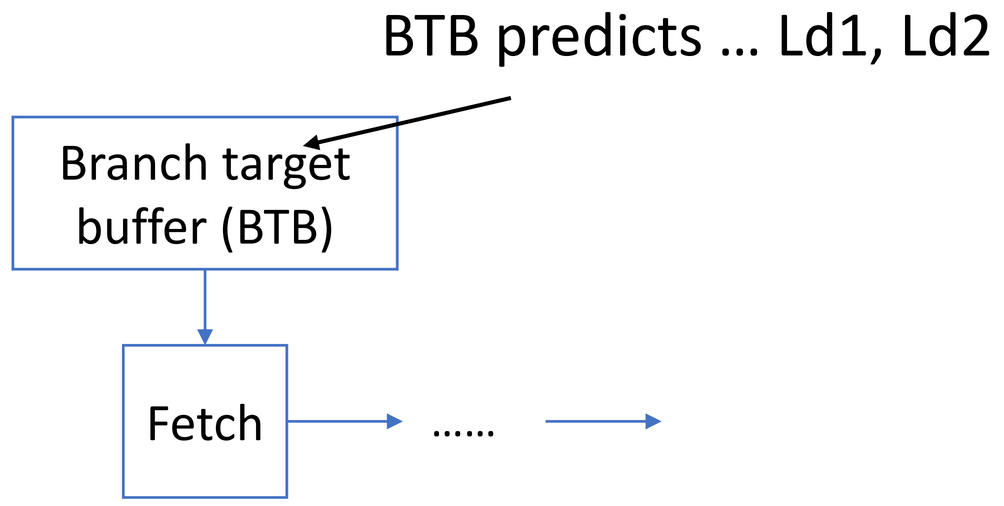
- Most BTBs store partial tags and targets…
    - <last n bits of current PC, target PC>

```
oxfff110   Br: if (…) {
           …       }

           …

oxfff234   Ld1: secret = array1[x]*4096
           Ld2: y = array2[secret]
```

Spectre Variant 2 – Exploit Branch Target

- Most BTBs store partial tags and targets…
  - <last n bits of current PC, target PC>

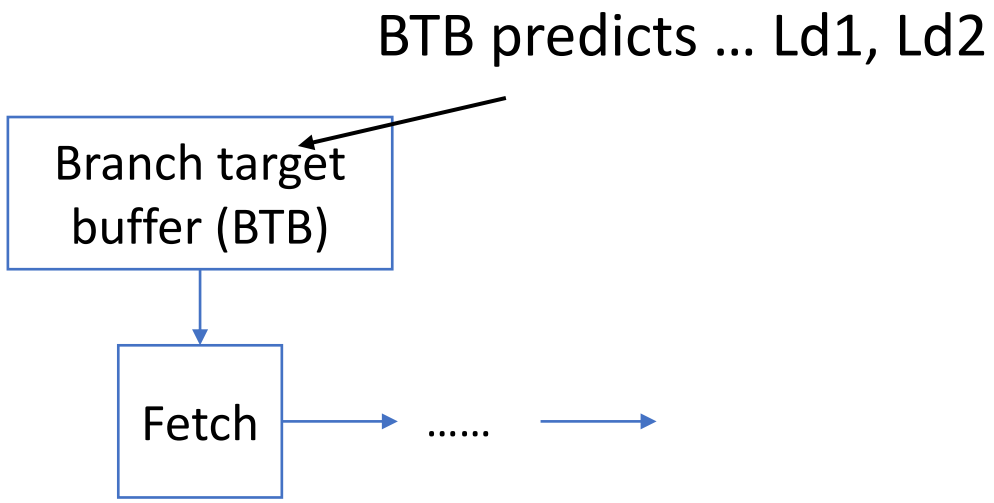
```
oxfff110   Br: if (…) {
           …       }

           …

oxfff234   Ld1: secret = array1[x]*4096

           Ld2: y = array2[secret]
```



Spectre Variant 2 – Exploit Branch Target

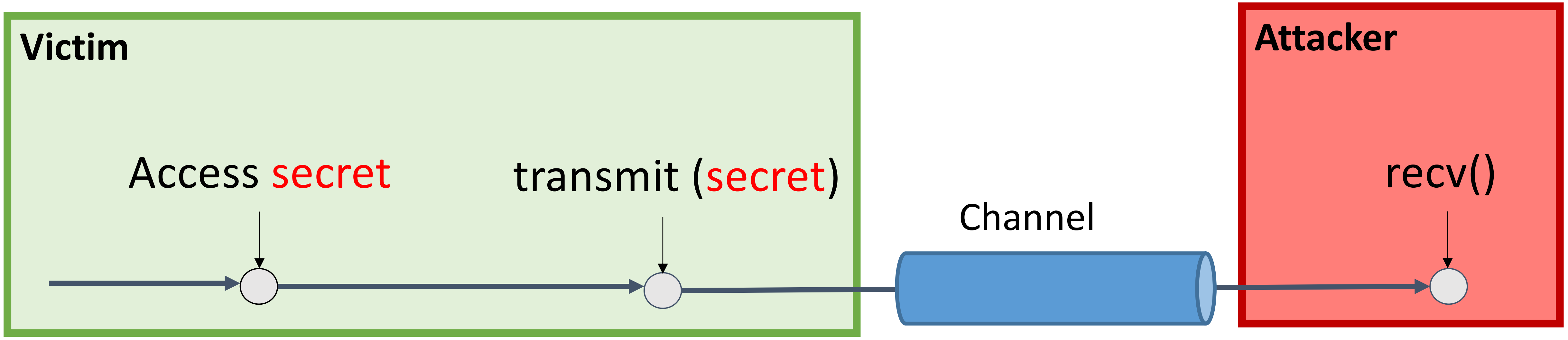
- Most BTBs store partial tags and targets…
    - <last n bits of current PC, target PC>

```
oxfff110   Br: if (…) {
           …       }

           …

oxfff234   Ld1: secret = array1[x]*4096
           Ld2: y = array2[secret]
```



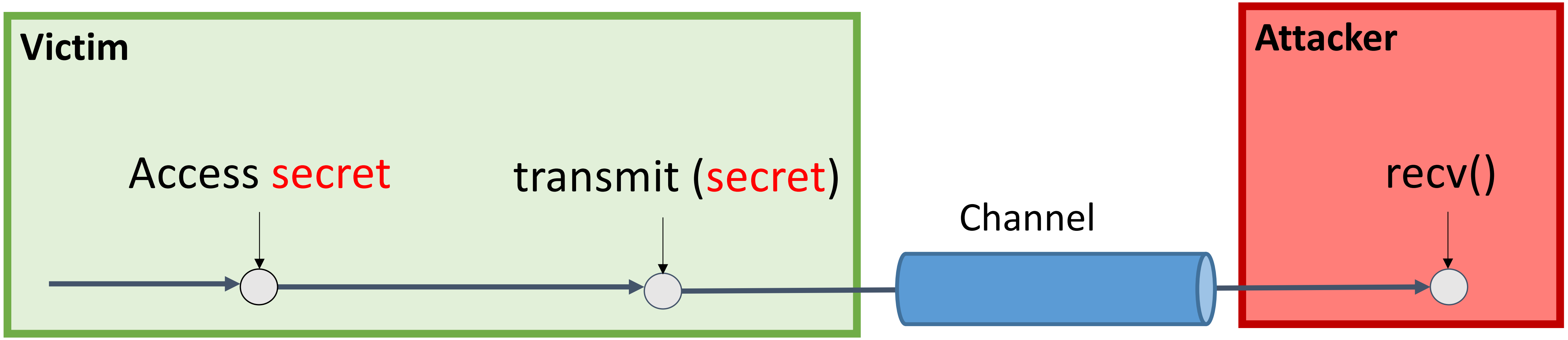
Train BTB properly → Execute arbitrary gadgets speculatively

General Attack Schema



- Traditional (non-transient) attacks
  - Data-dependent program behavior
- Transient attacks
  - Meltdown = transient execution + deferred exception handling
  - Spectre = transient execution on wrong paths

General Attack Schema

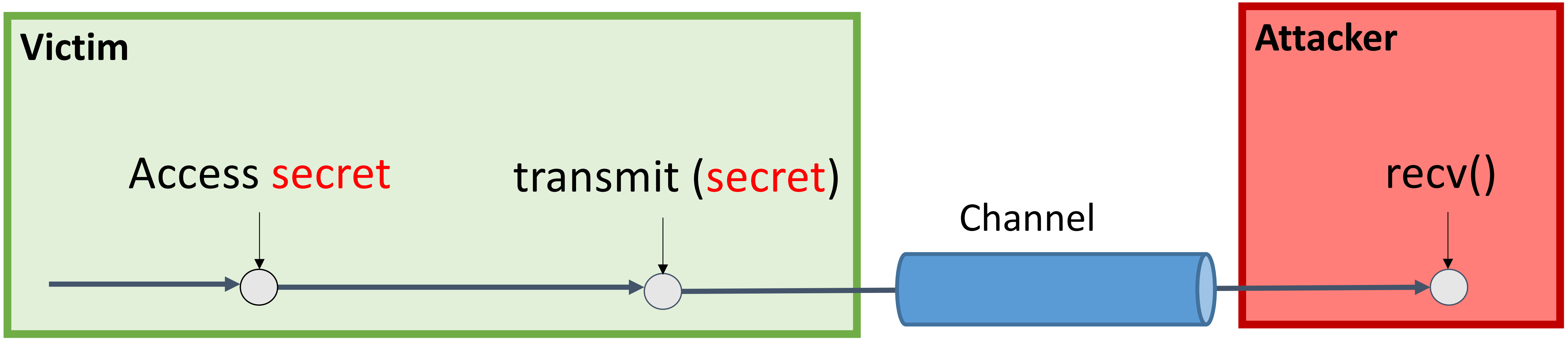


- Traditional (non-transient) attacks
  - Data-dependent program behavior

Hard to fix

- Transient attacks
  - Meltdown = transient execution + deferred exception handling
  - Spectre = transient execution on wrong paths

General Attack Schema



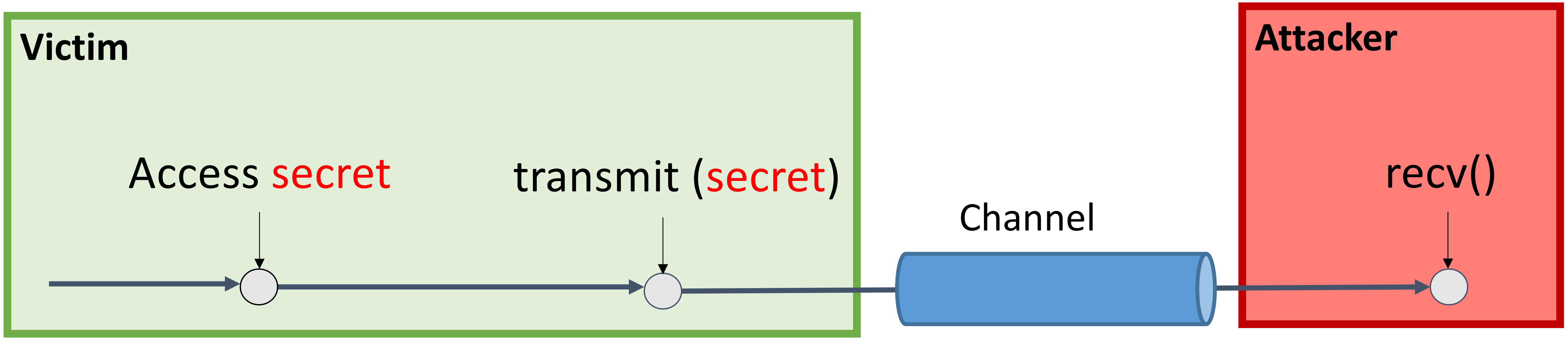
- Traditional (non-transient) attacks
  - Data-dependent program behavior

Hard to fix

- Transient attacks
  - Meltdown = transient execution + deferred exception handling
  - Spectre = transient execution on wrong paths

"Easy" to fix

General Attack Schema



- Traditional (non-transient) attacks
  - Data-dependent program behavior
- Transient attacks
  - Meltdown = transient execution + deferred exception handling
  - Spectre = transient execution on wrong paths

Hard to fix

"Easy" to fix

Hard to fix

Takeaways

Transient execution attacks *use* (not “are”) side/covert channels.

Takeaways

Transient execution attacks *use* (not “are”) side/covert channels.

“Spectre” (wrong-path execution) is **fundamental**.
Speculation/prediction is not perfect.

Takeaways

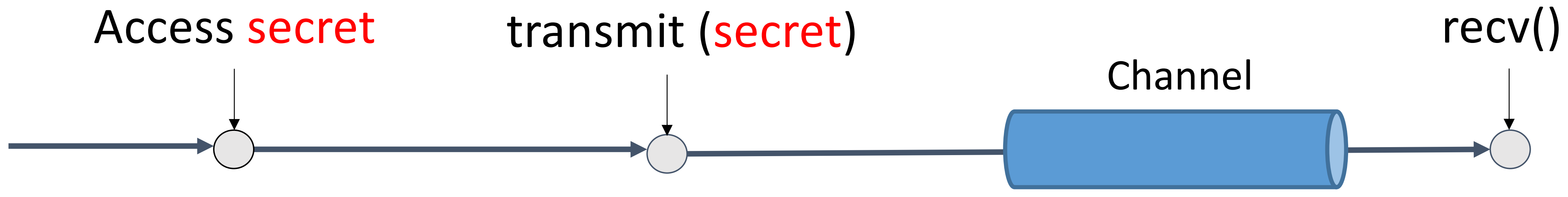
Transient execution attacks *use* (not “are”) side/covert channels.

“Spectre” (wrong-path execution) is **fundamental**.
Speculation/prediction is not perfect.

“Meltdown” (deferred exceptions) is **not fundamental**.

Transient v.s. Non-transient

Classification



{Transient, Non-transient} secret x {Transient, Non-transient} transmitter

Secret accessed	Transmitter	Classification
Non-transient	Non-transient	Traditional side channels
Transient	Non-transient	Not possible on today’s machines?
Non-transient	Transient	Spectre
Transient	Transient	Spectre

Non-transient secret + Non-transient transmitter

What can leak?

A subset of committed architectural state, at each point in the program's dynamic execution.

Non-transient secret + Non-transient transmitter

What can leak?

A subset of committed architectural state, at each point in the program's dynamic execution.

```
secret <- load(0x5)
secret <- secret + 1
secret -> store(0x5)
```

Non-transient secret + Non-transient transmitter

What can leak?

A subset of committed architectural state, at each point in the program's dynamic execution.

```
secret <- load(0x5)
secret <- secret + 1
secret -> store(0x5)
```

secret does not leak
(assume '+' data independent)

Non-transient secret + Non-transient transmitter

What can leak?

A subset of committed architectural state, at each point in the program's dynamic execution.

```
secret <- load(0x5)
secret <- secret + 1
secret -> store(0x5)
```

secret does not leak
(assume '+' data independent)

```
secret <- load(0x5)
Dummy<- load(secret)
```

secret leaks

Non-transient secret + Non-transient transmitter

What can leak?

A subset of committed architectural state, at each point in the program's dynamic execution.

```
secret <- load(0x5)
secret <- secret + 1
secret -> store(0x5)
```

secret does not leak
(assume '+' data independent)

```
secret <- load(0x5)
Dummy<- load(secret)
```

secret leaks

```
secret <- load(0x5)
if (false)
  Dummy<-load(secret)
```

secret does not leak

Non-transient secret + {Transient, Non-transient} transmitter

```
secret <- load(0x5)
secret <- secret + 1
secret -> store(0x5)
```

```
secret <- load(0x5)
Dummy<- load(secret)
```

```
secret <- load(0x5)
if (false)
  Dummy<-load(secret)
```

Non-transient secret + Non-transient transmitter:

secret does not leak

secret leaks

secret does not leak

Non-transient secret + {Transient, Non-transient} transmitter

```
secret <- load(0x5)
secret <- secret + 1
secret -> store(0x5)
```

```
secret <- load(0x5)
Dummy<- load(secret)
```

```
secret <- load(0x5)
if (false)
  Dummy<-load(secret)
```

Non-transient secret + Non-transient transmitter:

secret does not leak

secret leaks

secret does not leak

Non-transient secret + Transient secret :

Non-transient secret + {Transient, Non-transient} transmitter

```
secret <- load(0x5)
secret <- secret + 1
secret -> store(0x5)
```

```
secret <- load(0x5)
Dummy<- load(secret)
```

```
secret <- load(0x5)
if (false)
  Dummy<-load(secret)
```

Non-transient secret + Non-transient transmitter:

secret does not leak | secret leaks | secret does not leak

Non-transient secret + Transient secret : ||

secret does not leak | secret leaks

Non-transient secret + {Transient, Non-transient} transmitter

```
secret <- load(0x5)
secret <- secret + 1
secret -> store(0x5)
```

```
secret <- load(0x5)
Dummy<- load(secret)
```

```
secret <- load(0x5)
if (false)
  Dummy<-load(secret)
```

Non-transient secret + Non-transient transmitter:

secret does not leak | secret leaks | secret does not leak

$$\parallel \qquad \cancel{\parallel}$$

Non-transient secret + Transient secret :

secret does not leak | secret leaks | secret leaks (!)

Leakage Summary

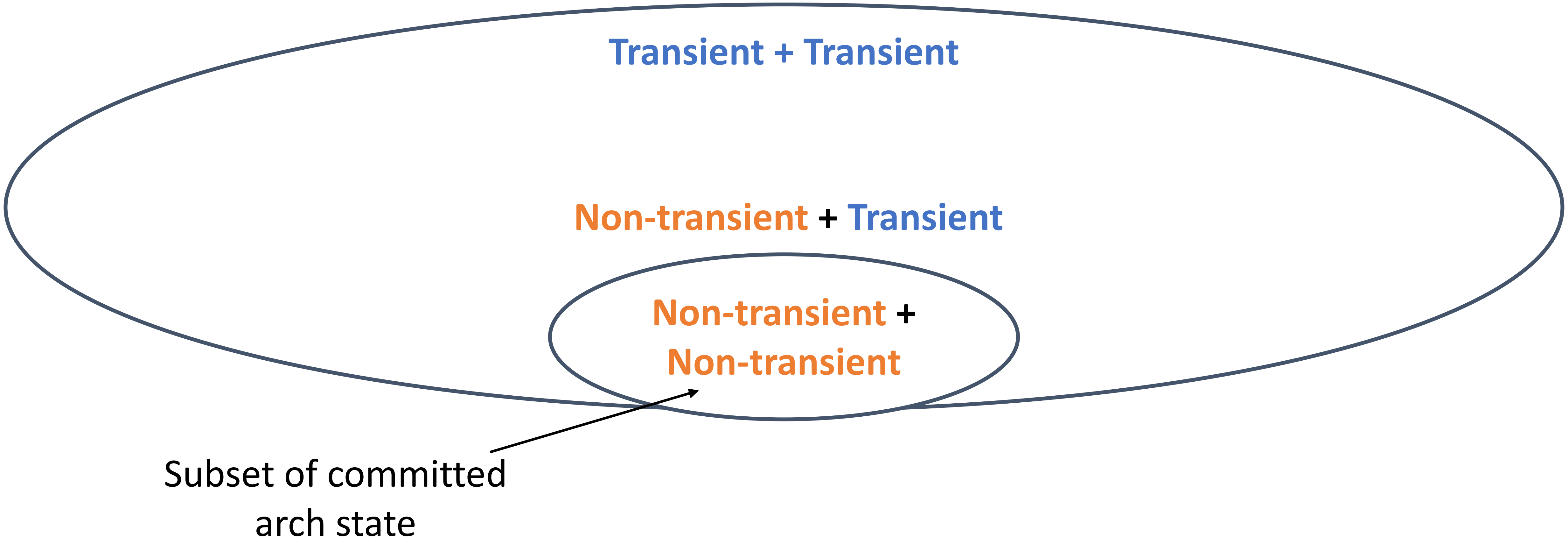
{Transient, Non-transient} secret x {Transient, Non-transient} transmitter

Transient + Transient

Non-transient + Transient

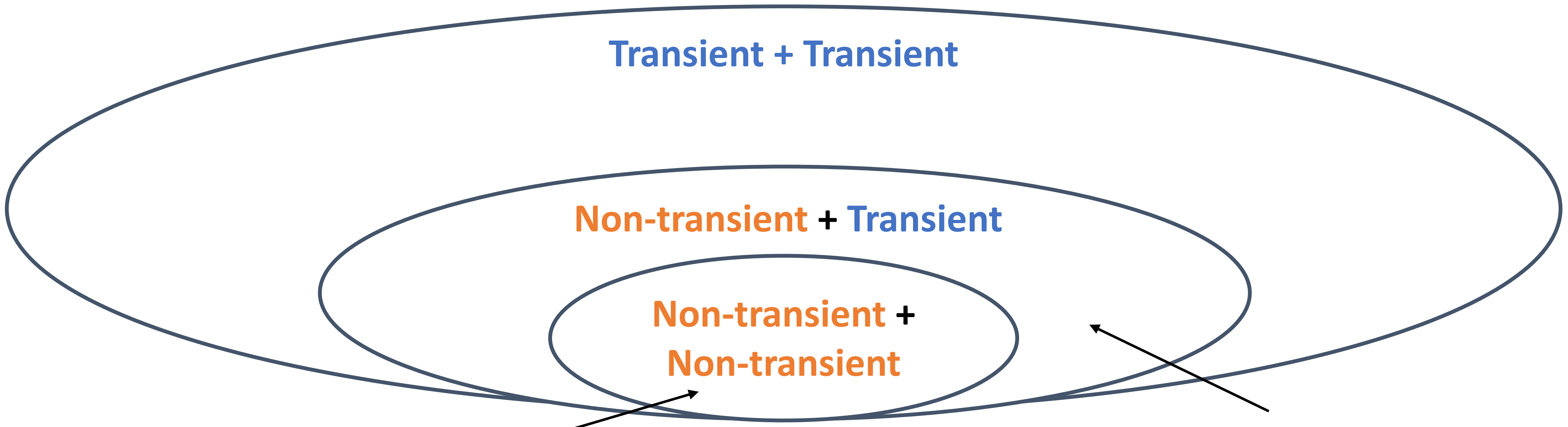
Leakage Summary

{Transient, Non-transient} secret x {Transient, Non-transient} transmitter


Transient + Transient
Non-transient + Transient
Non-transient + Non-transient
Subset of committed arch state

Leakage Summary

{Transient, Non-transient} secret x {Transient, Non-transient} transmitter

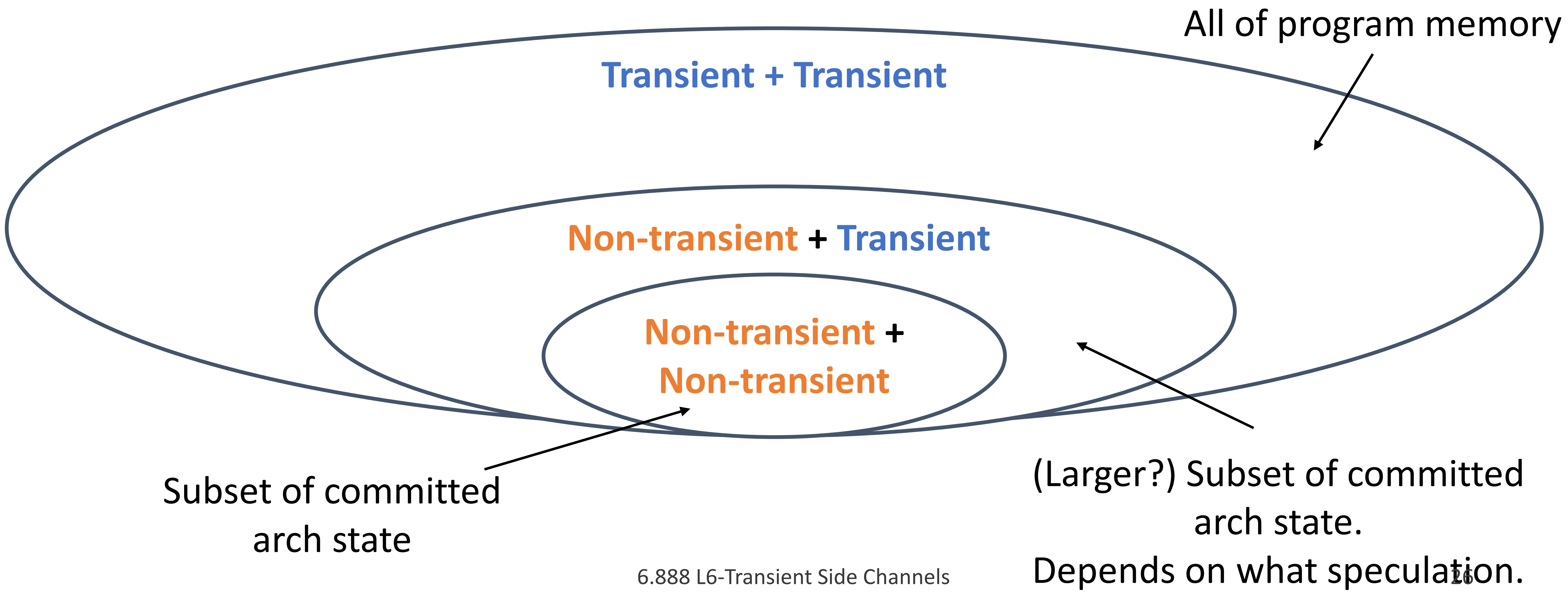


Subset of committed arch state

(Larger?) Subset of committed arch state. Depends on what speculation.

Leakage Summary

{Transient, Non-transient} secret x {Transient, Non-transient} transmitter


All of program memory
Transient + Transient
Non-transient + Transient
Non-transient + Non-transient
Subset of committed arch state
(Larger?) Subset of committed arch state.
Depends on what speculation.

Next Lecture:

Tiwari et al. Complete information flow tracking from the gates up. ASPLOS. 2009.