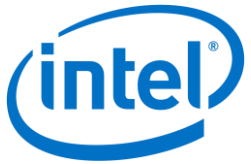


Principled Secure Processor Design

CHRISTOPHER W. FLETCHER

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Thank you

To my students and collaborators ☺

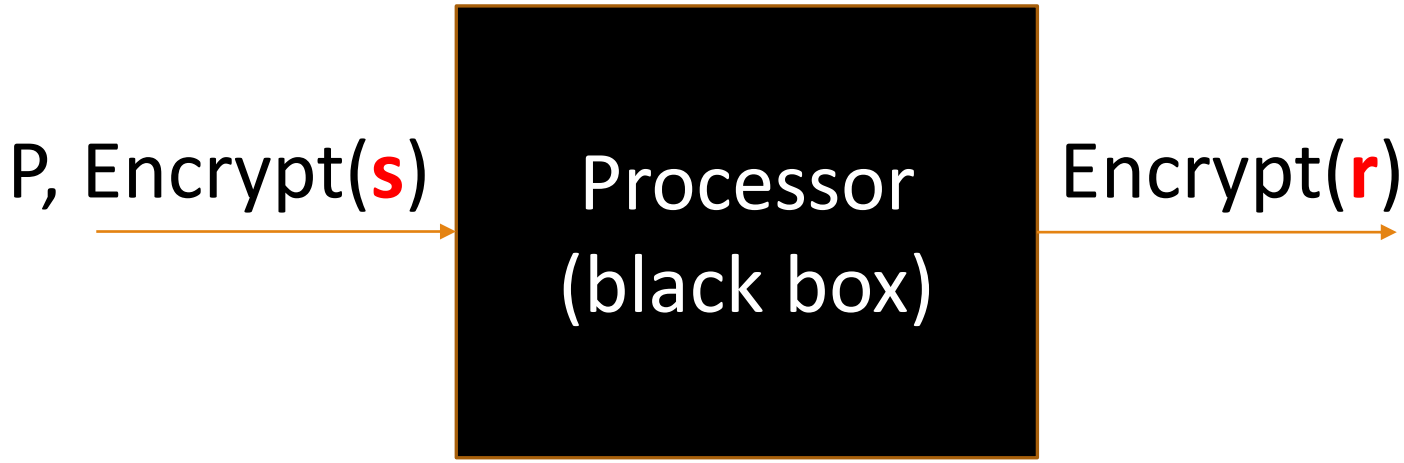
Jiyong Yu, Mohamad El Hajj, Lucas Hsiung, Mengjia Yan, Namrata Mantri, Artem Khyzha, Adam Morrison, Josep Torrellas, Po-An Tsai, Andres Sanchez, Daniel Sanchez

How good is process isolation nowadays, anyway?

Computing Abstractions Today (ideal)

r = P(**s**)

secrets are red



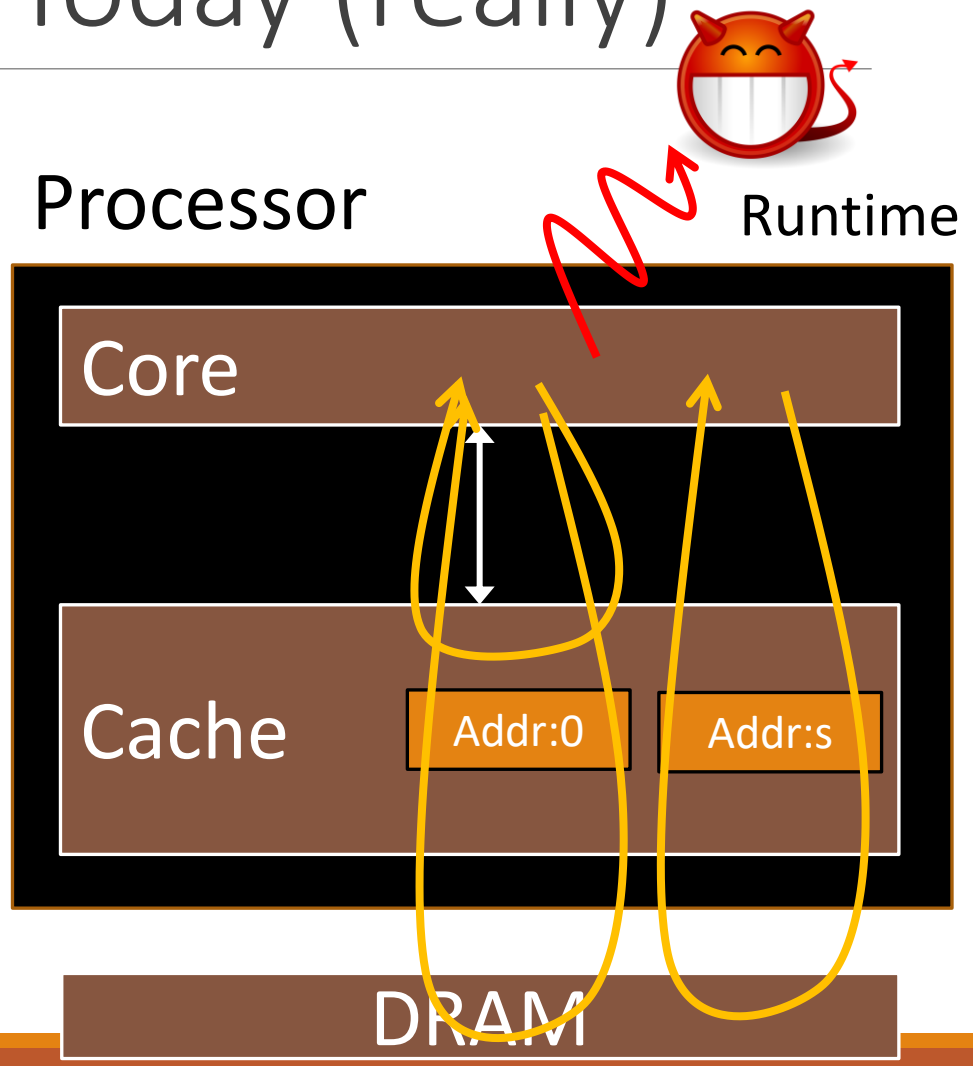
OS: protects computation P(**s**)

Crypto: protects data in transit (**s, r**)

→ No unauthorized party learns **s** ☺

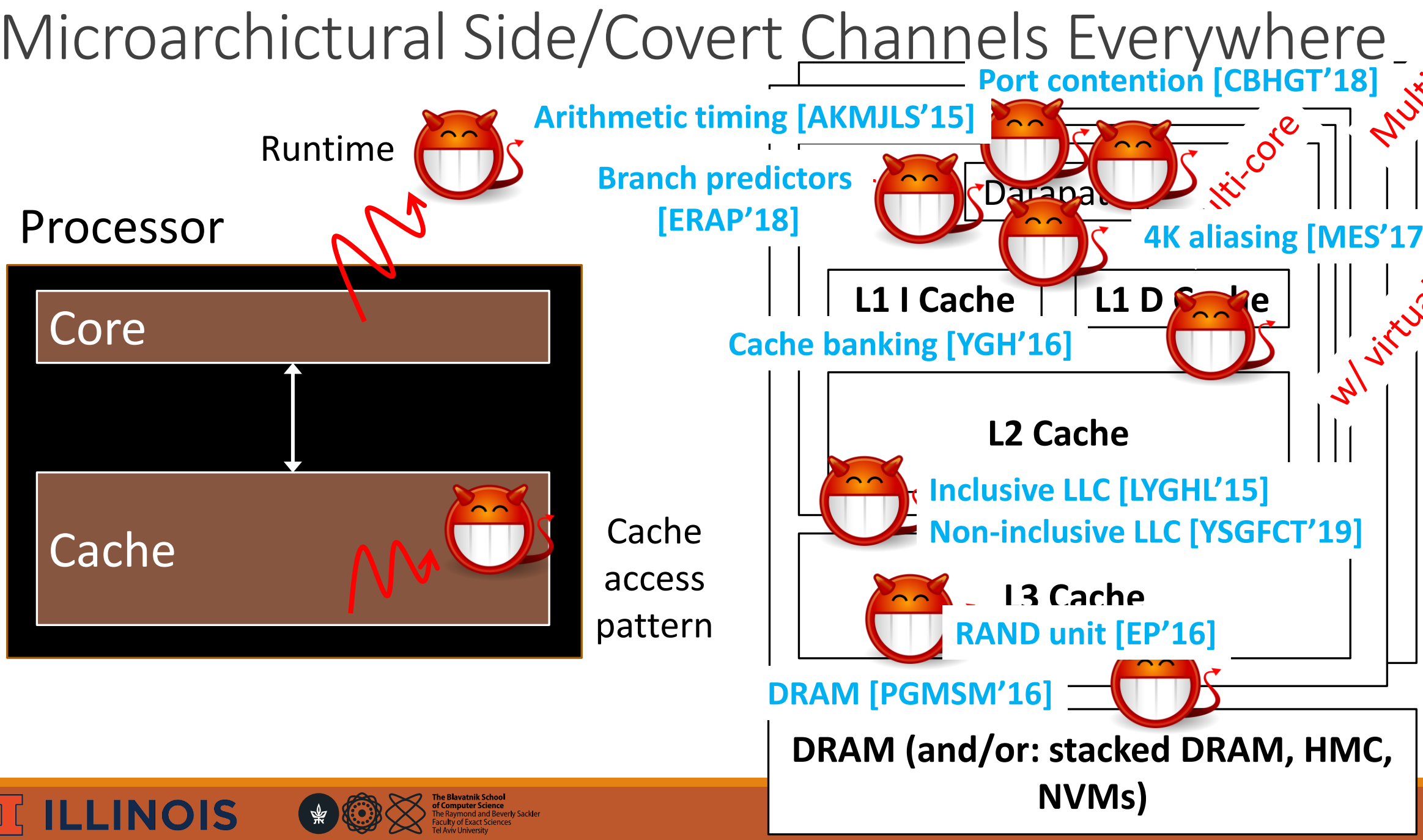
Computing Abstractions Today (really)

```
// s = bool
void P(secret s) {
   load(0);
   load(s*BLOCK_SZ);
}
```



	Case 1: s = 0	**Case 2: s = 1**
Cache:	miss, hit	miss, miss
	→ P is fast	→ P is slow

Microarchictural Side/Covert Channels Everywhere



Worse: attacks can enable “read gadgets”

Unsafe:

```
void P(secret s) {
  load(0);
  load(s*BLOCK_SZ); }
```

Safe:

```
void P(secret s) {
  load(0);
  load(0*BLOCK_SZ);
  load(1*BLOCK_SZ); }
```

Read gadget:

`bool ← read(addr a)`

- Attacker controls a
- Leaks P’s memory bit by bit

Mcilroy et al.; Spectre is here to stay: An analysis of side-channels and speculative execution, 2019.

This talk:

Principled, low-overhead defenses against microarchitectural attacks**

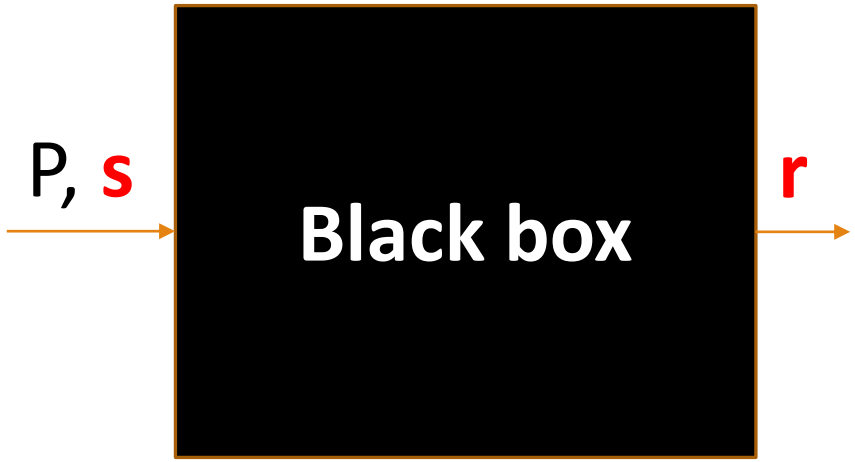
** FOCUSING TODAY ON SPECULATIVE EXECUTION ATTACKS

Principled, Low-overhead

Many uarch side/covert channel (cache, predictors, etc.)

Want: some clean security definition
E.g.,

$$\mathbf{r} = P(\mathbf{s})$$



i.e., secure given any uarch side/covert channels

Principled, Low-overhead

Obviously.

But not at the expense of clean security.

Classified  Unclassified

High  Low

A lattice model of secure information flow; Dorothy E. Denning, CACM 1976

Security + Low-overhead

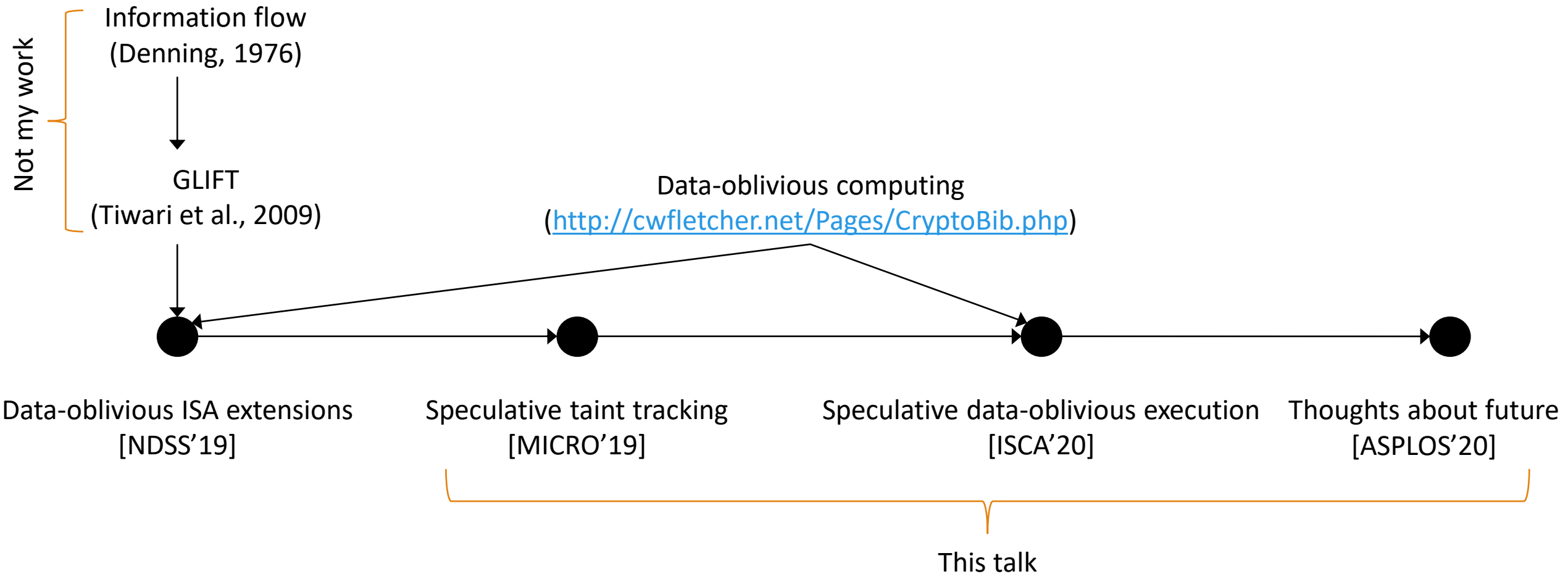
Classified  Unclassified

Classified ⟹ Classified 

Unclassified ⟹ Classified 

Unclassified ⟹ Unclassified 

This talk


Not my work
Information flow
(Denning, 1976)
GLIFT
(Tiwari et al., 2009)
Data-oblivious computing
(http://cwfletcher.net/Pages/CryptoBib.php)
Data-oblivious ISA extensions
[NDSS'19]
Speculative taint tracking
[MICRO'19]
Speculative data-oblivious execution
[ISCA'20]
Thoughts about future
[ASPLOS'20]
This talk

Part 1: Speculative Taint Tracking (STT)

COMPREHENSIVE PROTECTION FOR SPECULATIVE DATA

Speculative Execution Attacks*

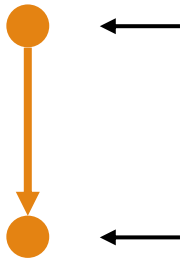
● ← Speculation starts

```
// Spectre Variant 1
if (addr < N) {    // speculation
    // access instruction
    spec_val = load [addr];

    // covert channel
    load [spec_val];

}
```

*: Kocher et al.; “Spectre Attacks: Exploiting Speculative Execution”, SP’19.

Speculative Execution Attacks*

```
// Spectre Variant 1

if (addr < N) {     // speculation
    // access instruction
    spec_val = load [addr];

    // covert channel
    load [spec_val];

}
```



Speculation starts

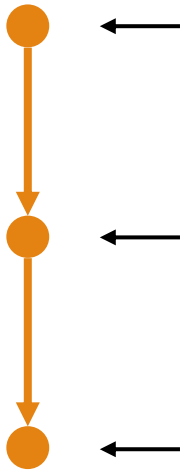
Speculative access instruction**
accesses secret

*: Kocher et al.; “Spectre Attacks: Exploiting Speculative Execution”, SP’19.

**: Kiriansky, Vladimir, et al.; "DAWG: A defense against cache timing attacks in speculative execution processors." MICRO’18.

Speculative Execution Attacks*

```
// Spectre Variant 1

if (addr < N) {     // speculation

    // access instruction
    spec_val = load [addr];

    // covert channel
    load [spec_val];

}
```



Speculation starts

Speculative access instruction**
accesses secret

Creates a **covert channel**
to leak secret

*: Kocher et al.; “Spectre Attacks: Exploiting Speculative Execution”, SP’19.

**: Kiriansky, Vladimir, et al.; "DAWG: A defense against cache timing attacks in speculative execution processors." MICRO’18.

Speculative Execution Attacks*

```
// Spectre Variant 1
if (addr < N) {    // speculation
    // access instruction
    spec_val = load [addr];
    // covert channel
    load [spec_val];
}
```

addr = N+1

- Speculation starts
- **Speculative access instruction**** accesses secret
- Creates a **covert channel** to leak secret
- Speculation ends - misspeculation!

*: Kocher et al.; “Spectre Attacks: Exploiting Speculative Execution”, SP’19.

**: Kiriansky, Vladimir, et al.; "DAWG: A defense against cache timing attacks in speculative execution processors." MICRO’18.

Speculative Execution Attacks*

```
// Spectre Variant 1
if (addr < N) {    // speculation
    // access instruction
    spec_val = load [addr];
    // covert channel
    load [spec_val];
}
```

Speculation starts

Speculative access instruction**
accesses secret

Creates a **covert channel**
to leak secret

Speculation ends -
misspeculation!

*: Kocher et al.; “Spectre Attacks: Exploiting Speculative Execution”, SP’19.

**: Kiriansky, Vladimir, et al.; "DAWG: A defense against cache timing attacks in speculative execution processors." MICRO’18.

Speculative Execution Attacks

```
// Spectre Variant 1

if (addr < N) {     // speculation

    // access instruction
    spec_val = load [addr];

    // covert channel
    load [spec_val];

}
```

Read gadget ☹

bool ← read(addr)

Main Insight of STT

Main Insight of STT

“Sufficient for security: prevent secrets from reaching covert channels”

Main Insight of STT

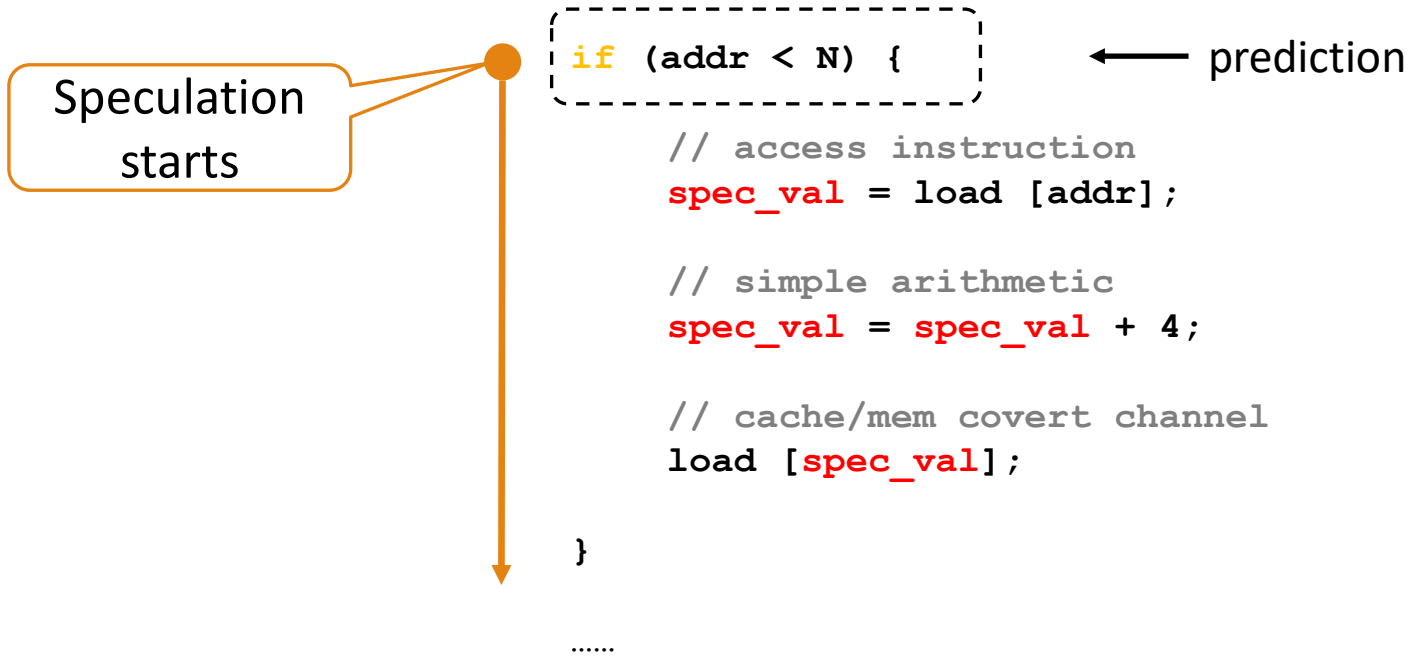
“Sufficient for security: prevent secrets from reaching covert channels”

```
if (addr < N) {

    // access instruction
    spec_val = load [addr];

    // simple arithmetic
    spec_val = spec_val + 4;

    // cache/mem covert channel
    load [spec_val];

}

.....
```

Creates a covert channel?	Input operand is a secret?	Requires protection?

Main Insight of STT

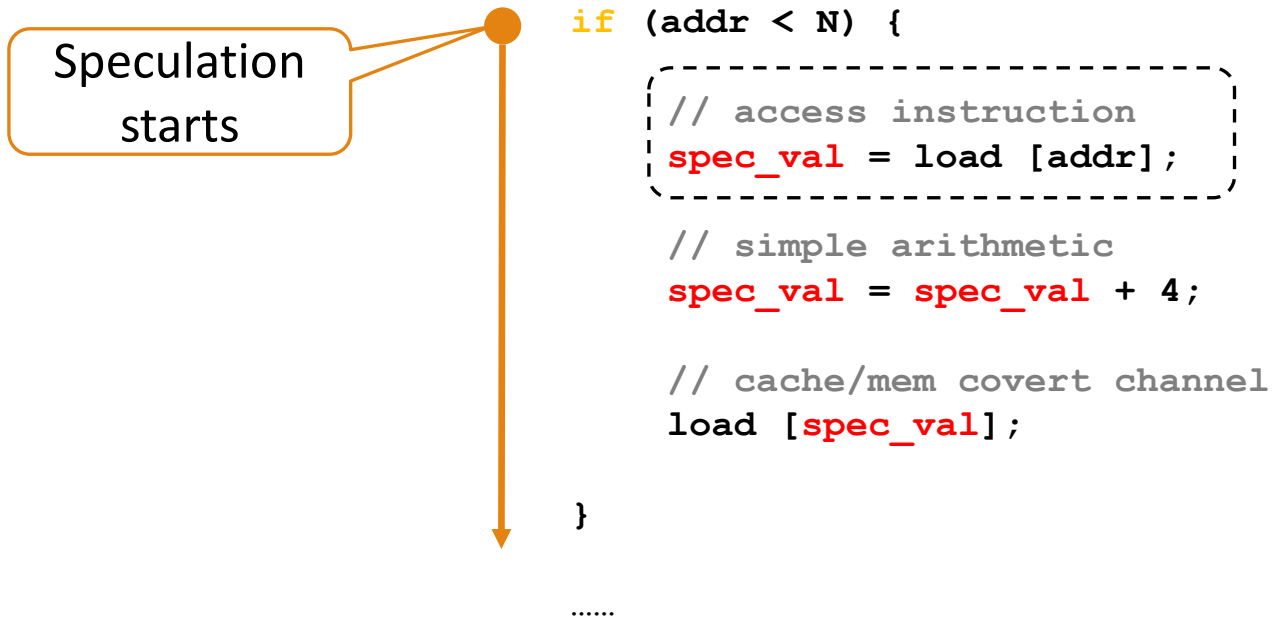
“Sufficient for security: prevent secrets from reaching covert channels”



Creates a covert channel?	Input operand is a secret?	Requires protection?

Main Insight of STT

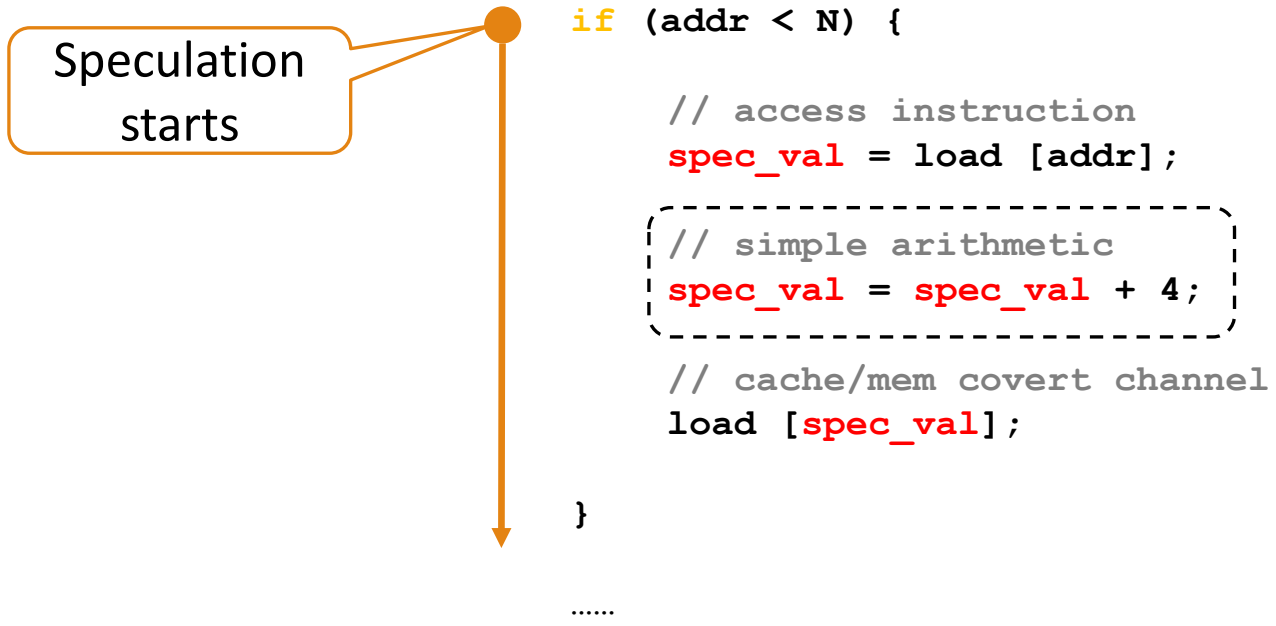
“Sufficient for security: prevent secrets from reaching covert channels”



Creates a covert channel?	Input operand is a secret?	Requires protection?
Yes	No	No

Main Insight of STT

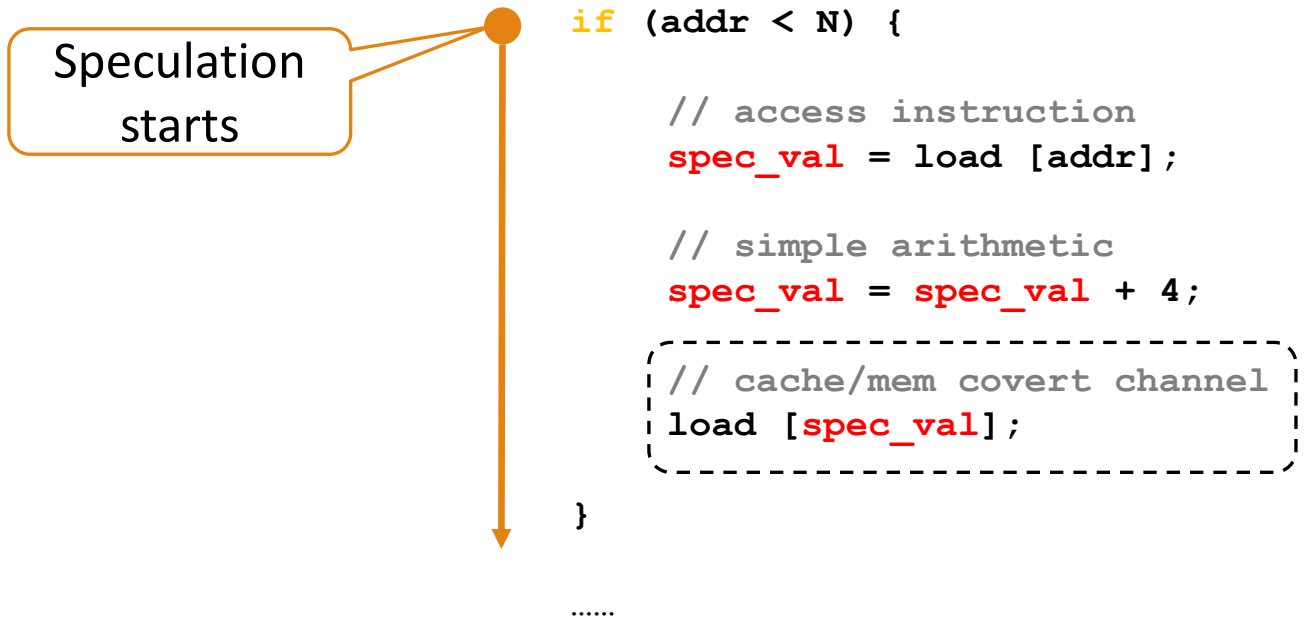
“Sufficient for security: prevent secrets from reaching covert channels”



Creates a covert channel?	Input operand is a secret?	Requires protection?
Yes	No	No
No	Yes	No

Main Insight of STT

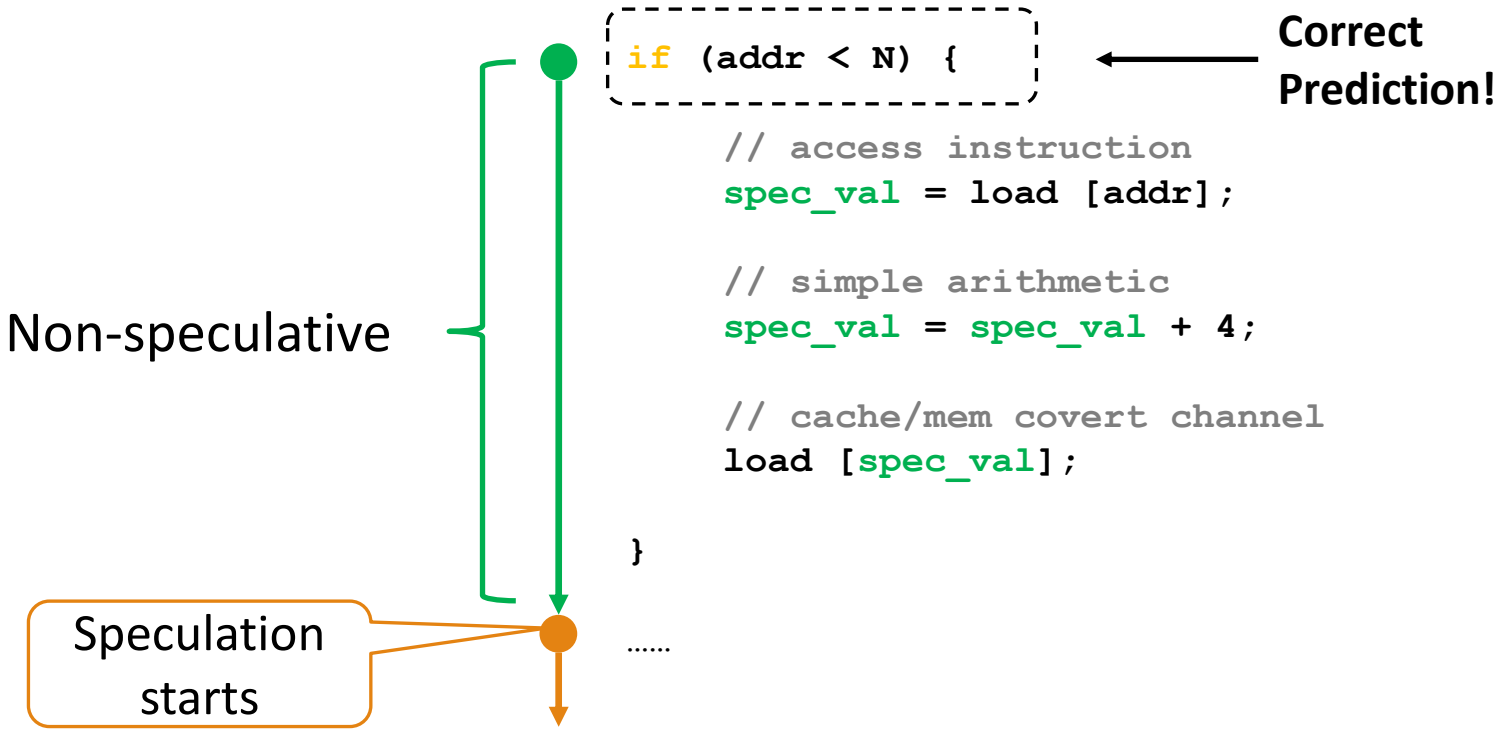
“Sufficient for security: prevent secrets from reaching covert channels”



Creates a covert channel?	Input operand is a secret?	Requires protection?
Yes	No	No
No	Yes	No
Yes	Yes	Yes

Main Insight of STT

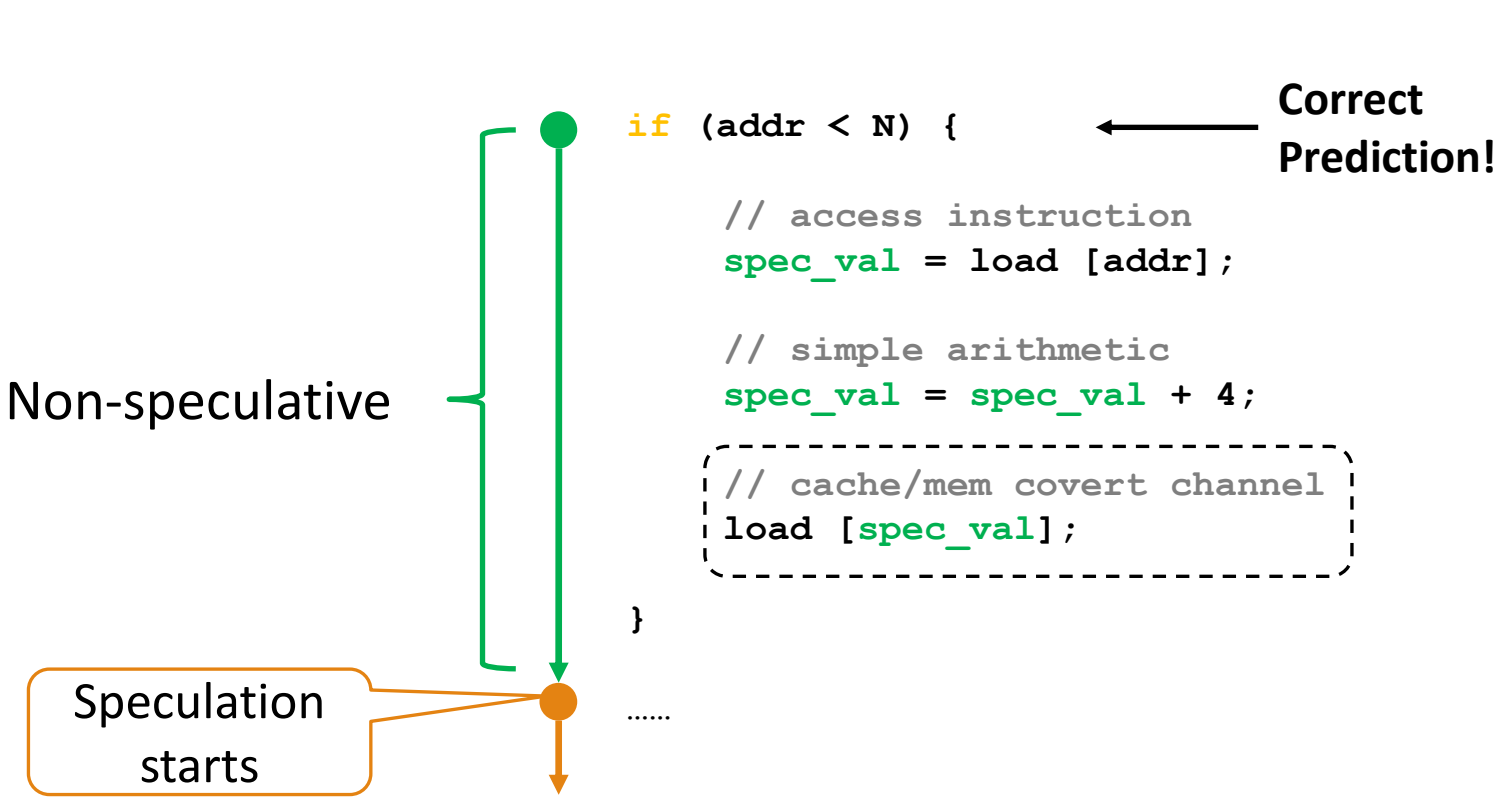
“Sufficient for security: prevent secrets from reaching covert channels”



Creates a covert channel?	Input operand is a secret?	Requires protection?
Yes	No	No
No	No	No
Yes	No	No

Main Insight of STT

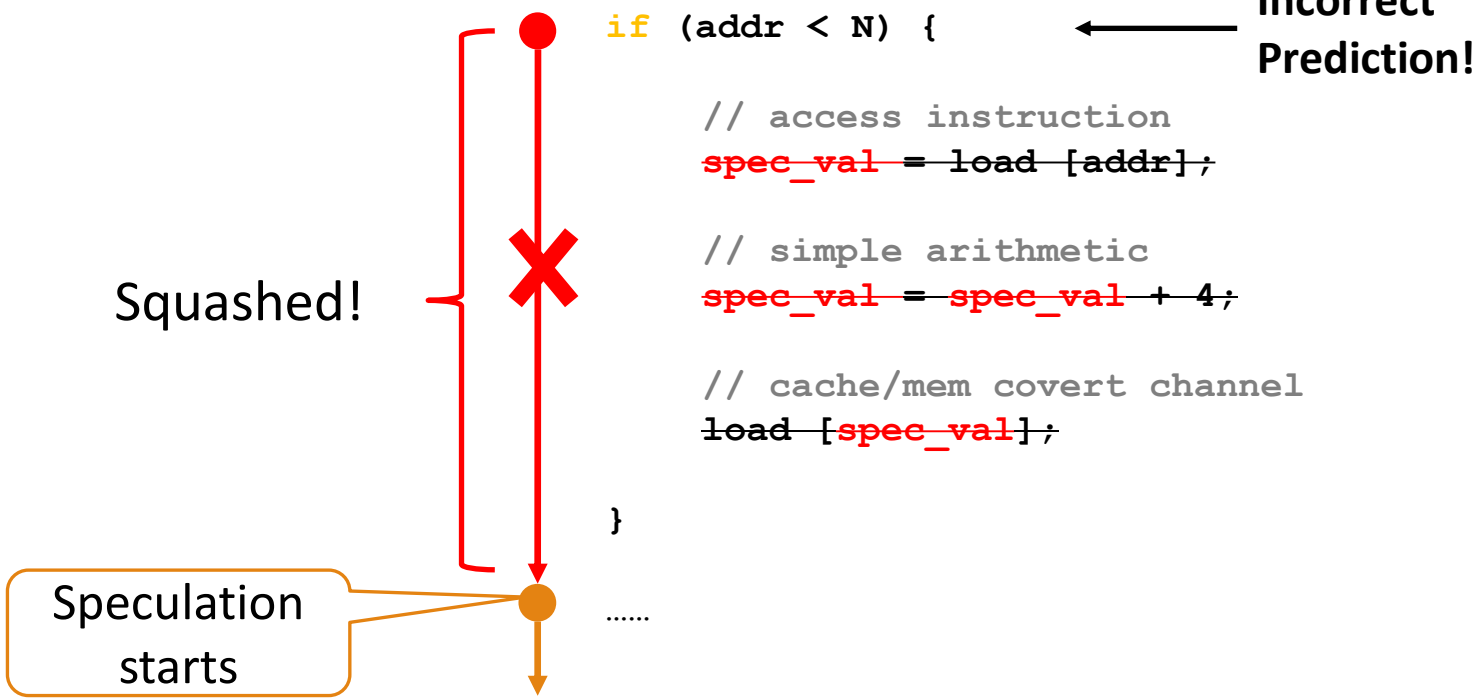
“Sufficient for security: prevent secrets from reaching covert channels”



Creates a covert channel?	Input operand is a secret?	Requires protection?
Yes	No	No
No	No	No
Yes	No	No

Main Insight of STT

“Sufficient for security: prevent secrets from reaching covert channels”



Creates a covert channel?	Input operand is a secret?	Requires protection?
Yes	No	No
No	Yes	No
Yes	Yes	Yes

Speculative Taint Tracking

Secret
(speculatively accessed data)

Covert channels

Speculative Taint Tracking

Secret
(speculatively accessed data)

Covert channels

Security definition:

Arbitrary speculative execution can only leak retired register file state.

~~Read~~ ~~gadget~~

Speculative Taint Tracking

Secret
(speculatively accessed data)

Covert channels

Speculative Taint Tracking

Secret
(speculatively accessed data)

Covert channels

What are the covert channels?

Speculative Taint Tracking

Secret
(speculatively accessed data)

Covert channels

What are the covert channels?

A new classification to understand covert channels in speculative machines

Speculative Taint Tracking

Secret
(speculatively accessed data)

Covert channels

What are the covert channels?

A new classification to understand covert channels in speculative machines

How to identify all the secrets?

Speculative Taint Tracking

Secret
(speculatively accessed data)

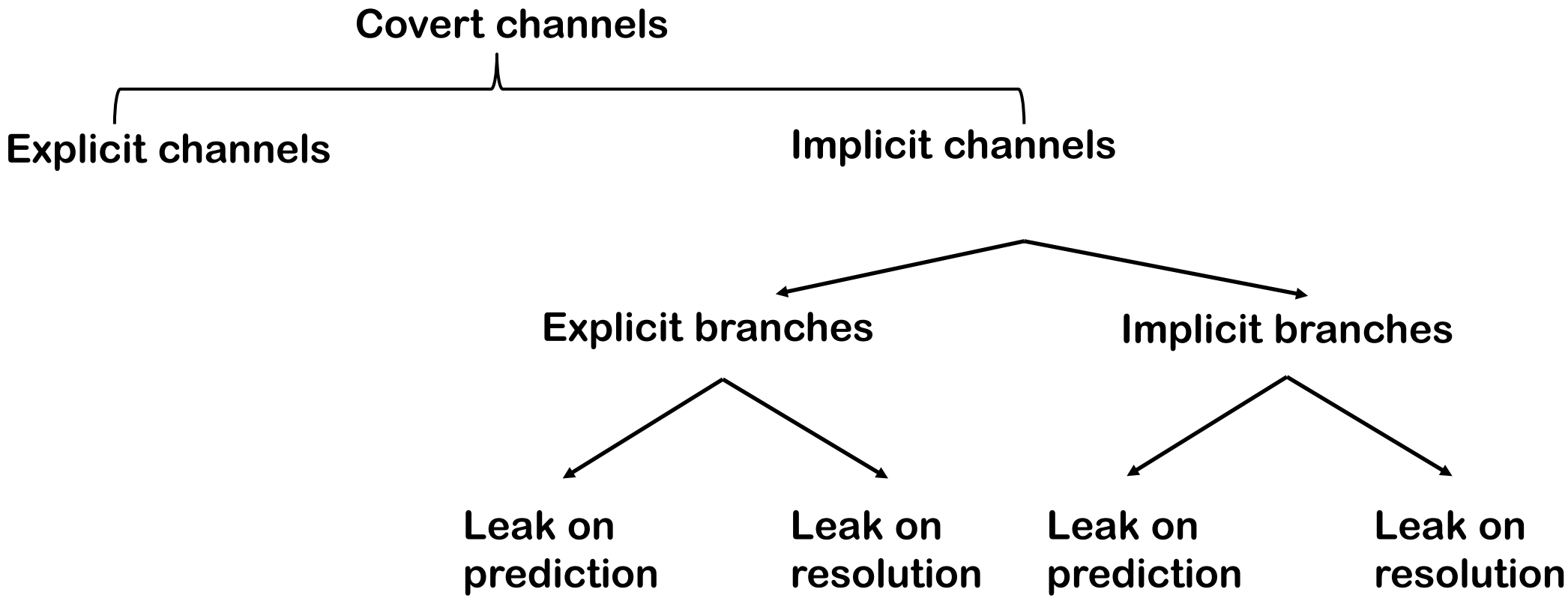
Covert channels

What are the covert channels? → **A new classification to understand covert channels in speculative machines**

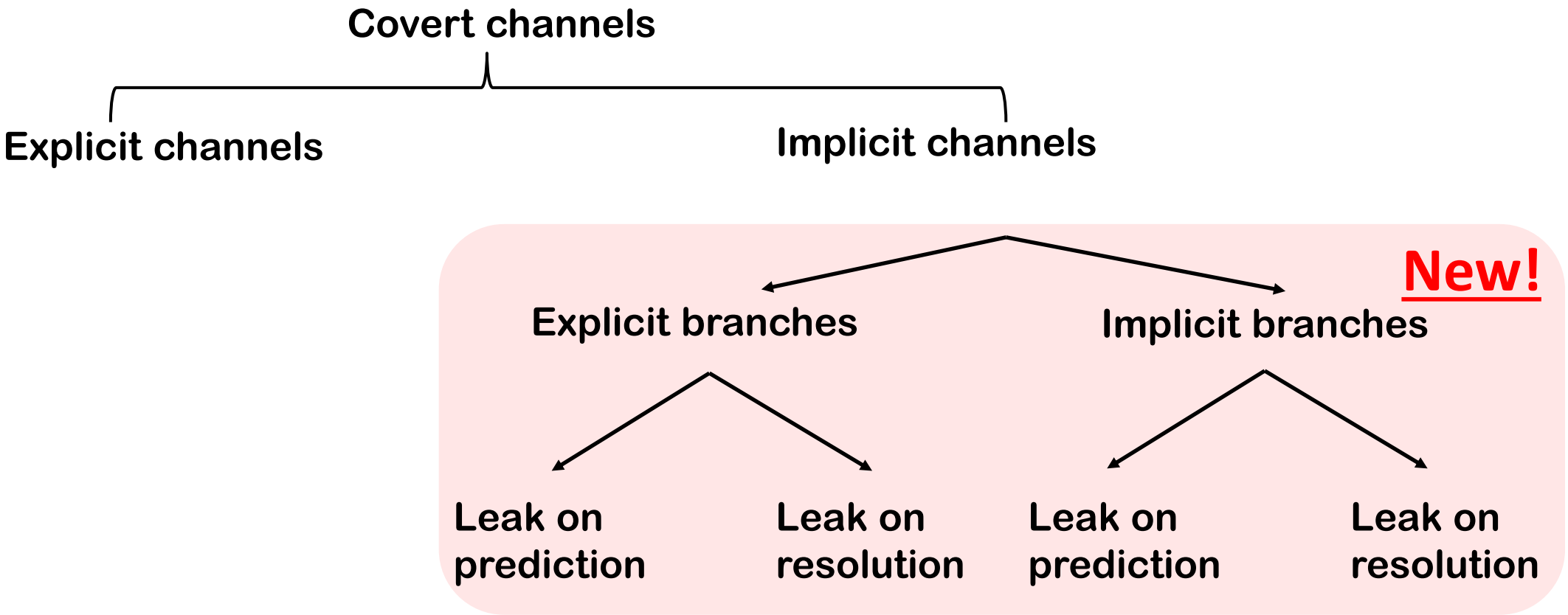
How to identify all the secrets? → **A new taint/untaint mechanism to track secrets in hardware**

A Classification of Covert Channels in HW

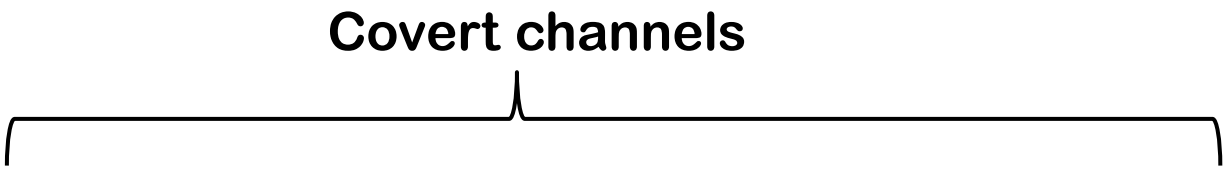
Classification of Covert Channels



Classification of Covert Channels



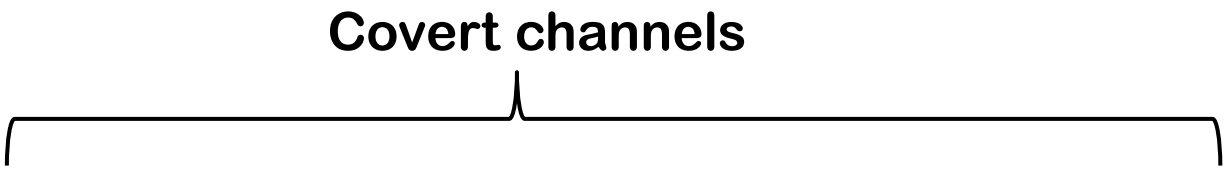
Classification of Covert Channels



Explicit channels:
Secret inputs are directly leaked by *operand-dependent hardware resource usage*

```
load [secret];
```

Classification of Covert Channels



Explicit channels:
Secret inputs are directly leaked by *operand-dependent hardware resource usage*

Examples:
- memory loads
- data-dependent arithmetic

Classification of Covert Channels

Covert channels

Explicit channels:
Secret inputs are directly leaked by *operand-dependent hardware resource usage*

Examples:
- memory loads
- data-dependent arithmetic

Implicit channels:
Secret inputs are indirectly leaked by *how (or that) one or several instructions execute*

```
secret = load [addr];
if (secret == 1)
    load [0x00];
```

Classification of Covert Channels

Covert channels

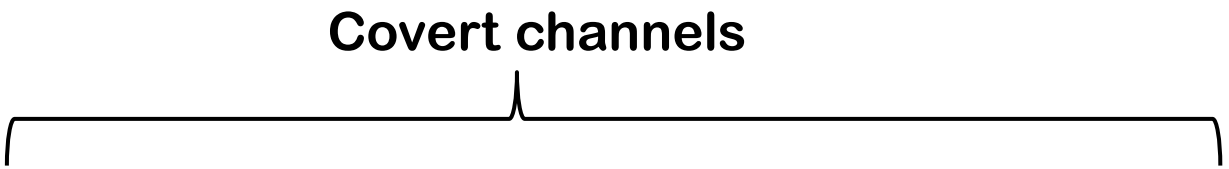
Explicit channels:
Secret inputs are directly leaked by *operand-dependent hardware resource usage*

Examples:
- memory loads
- data-dependent arithmetic

Implicit channels:
Secret inputs are indirectly leaked by *how (or that) one or several instructions execute*

```
secret = load [addr];
if (secret == 1)
    load [0x00];
```

Classification of Covert Channels



Explicit channels:
Secret inputs are directly leaked by *operand-dependent hardware resource usage*

Examples:
- memory loads
- data-dependent arithmetic

Implicit channels:
Secret inputs are indirectly leaked by *how (or that) one or several instructions execute*

Examples:
- branch/jump instructions

Classification of Covert Channels

Covert channels

Explicit channels:
Secret inputs are directly leaked by *operand-dependent hardware resource usage*

Examples:
- memory loads
- data-dependent arithmetic

Implicit channels:
Secret inputs are indirectly leaked by *how (or that) one or several instructions execute*

New!

Explicit branches
Examples:
- Branch/jump instructions

Classification of Covert Channels

Covert channels

Explicit channels:
Secret inputs are directly leaked by *operand-dependent hardware resource usage*

Examples:
- memory loads
- data-dependent arithmetic

Implicit channels:
Secret inputs are indirectly leaked by *how (or that) one or several instructions execute*

New!

Explicit branches
Examples:
- Branch/jump instructions

Leak on prediction

Leak on resolution

Secrets are red

Non-secrets are green

Attacker can see sequence of memory accesses (to L1 cache)

```
secret = load [0x00];
if (secret == 1)
   load [0x01];
else
  load [0xFF];
```

Case 1 (`secret == 1`): Attacker sees [`0x00, 0x01`]

Case 2 (`secret == 0`): Attacker sees [`0x00, 0xFF`]

Explicit Branches @ Prediction

```
… …
… …
if ( secret )
… …
… …
if ( public )
    load [0x00];
else
    load [0x10];
```

Explicit Branches @ Prediction

Cause:

The predictor state becomes a function of secret

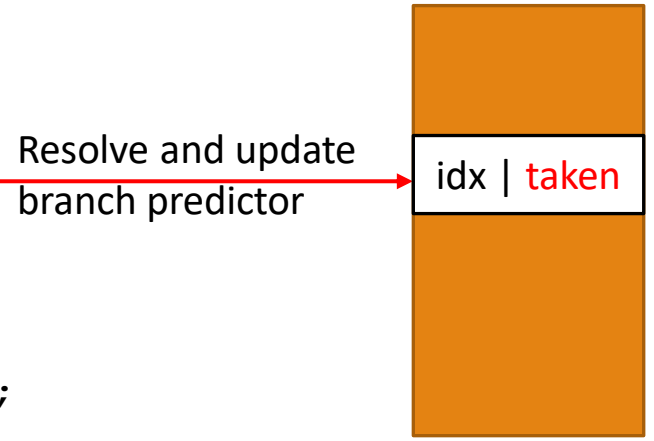
```
… …
… …
if ( secret )
… …
… …
if ( public )
    load [0x00];
else
    load [0x10];
```

Explicit Branches @ Prediction

Cause:

The predictor state becomes a function of secret

```
… …
… …
if ( secret )
… …
… …
if ( public )
    load [0x00];
else
    load [0x10];
```

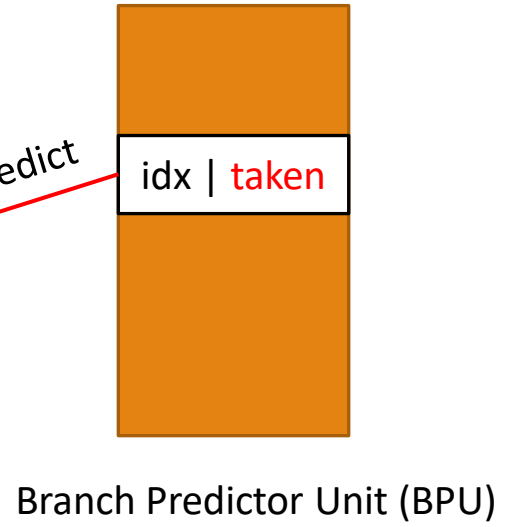


Explicit Branches @ Prediction

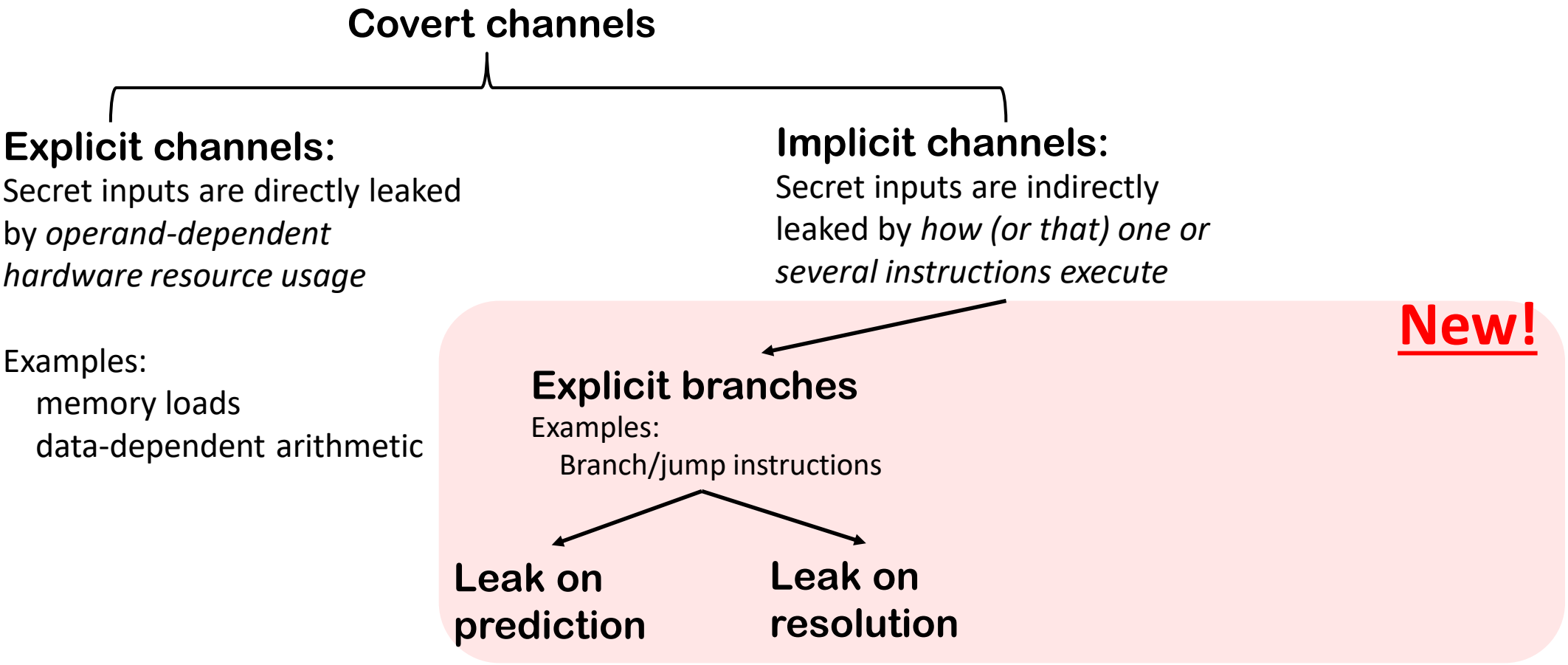
Cause:

The predictor state becomes a function of secret

```
… …
… …
if ( secret )
… …
… …
if ( public )
    load [0x00];
else
    load [0x10];
```



Classification of Covert Channels



Explicit Branches @ Resolution

```
if (secret) {
    y++;
}
z = load [0x00]
```

Explicit Branches @ Resolution

Cause:

The resolution of a mis-speculation triggers a pipeline squash and alternation of control flow

```
if (secret) {
    y++;
}
z = load [0x00]
```

Explicit Branches @ Resolution

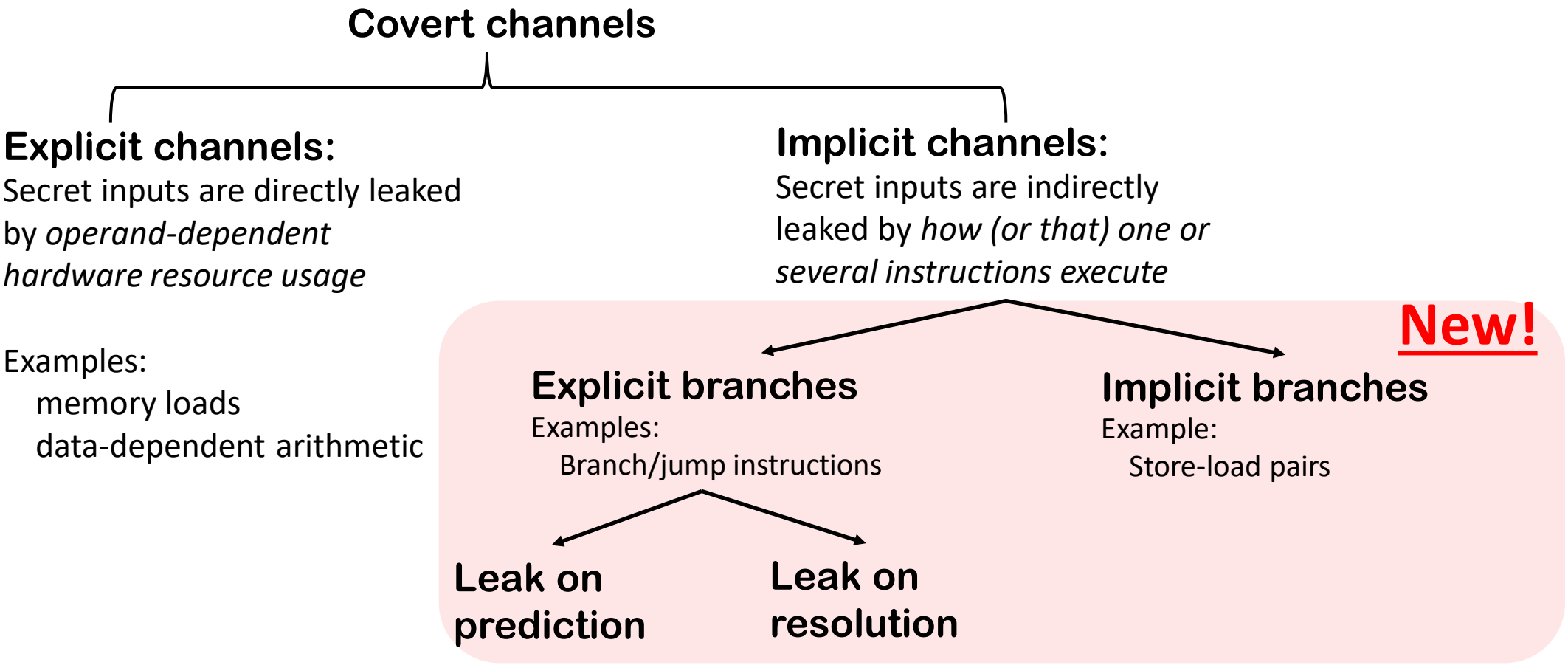
Cause:

The resolution of a mis-speculation triggers a pipeline squash and alternation of control flow

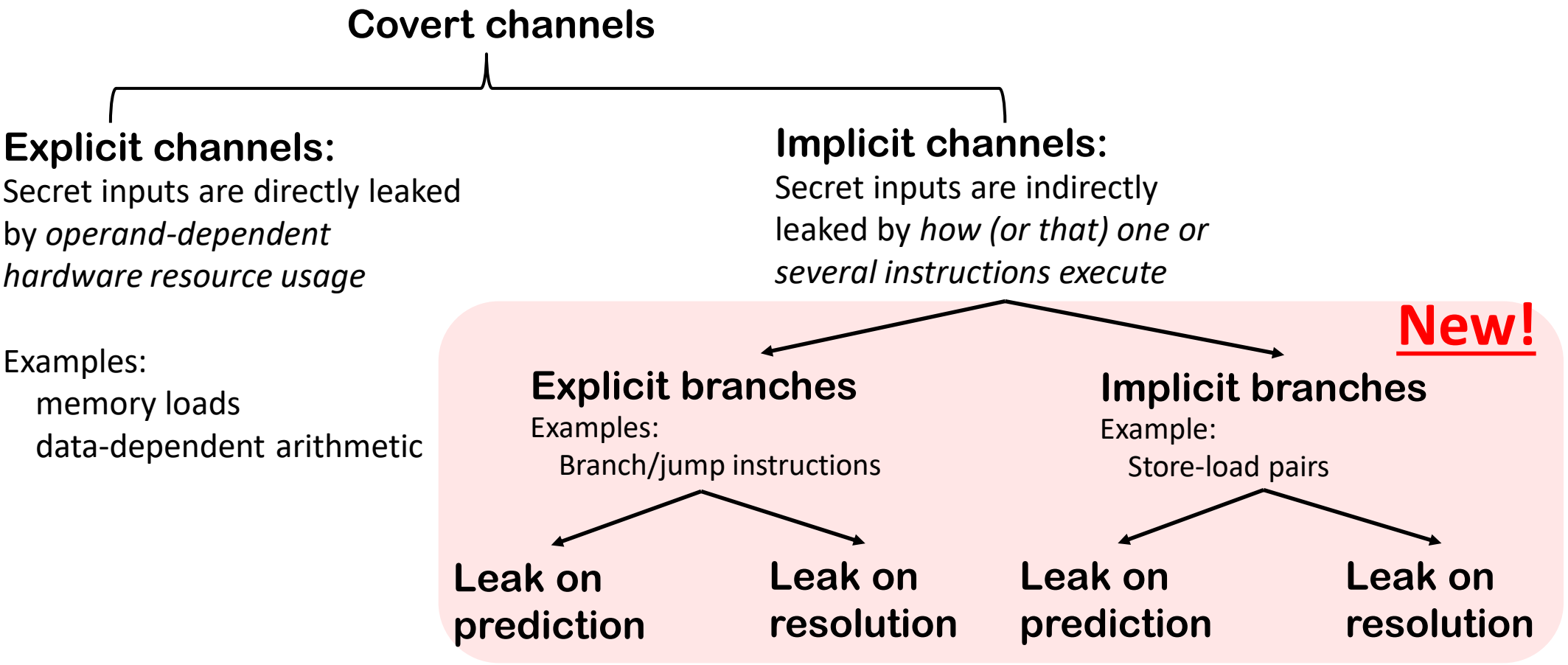
```
if (secret) {
    y++;
}
z = load [0x00]
```

secret != prediction
→ squash
→ load executes twice!

Classification of Covert Channels



Classification of Covert Channels



Implicit Branches

```
store [secret] = foo;

bar = load [0x00];
```

Implicit Branches

Cause:

Non-control flow instructions create branch-like behaviors.

```
store [secret] = foo;

bar = load [0x00];
```

Implicit Branches

Cause:

Non-control flow instructions create branch-like behaviors.

```
store [secret] = foo;
```

```
bar = load [0x00];
```

Can be thought as:

```
if (secret == 0x00) {
    forward from store queue
}
else {
    cache_load [0x00]
}
```

Identifying Secrets using Tainting/Untainting

Identifying Secrets using Tainting/Untainting

Basic idea: taint all the secrets

- Speculatively accessed data (secrets by definition)
- And their dependents

Identifying Secrets using Tainting/Untainting

Basic idea: taint all the secrets

- Speculatively accessed data (secrets by definition)
- And their dependents

```
if (addr < N) {

    // access instruction
    a = load [addr];

    // simple arithmetic
    b = a + 4;

    // cache/mem covert channel
    load [b];

}

.....
.....
.....
```

speculative

Identifying Secrets using Tainting/Untainting

Basic idea: taint all the secrets

- Speculatively accessed data (secrets by definition)
- And their dependents

STT ***taints***:

1) Output of speculative access instructions (a)

```
if (addr < N) {

    // access instruction
    a = load [addr];

    // simple arithmetic
    b = a + 4;

    // cache/mem covert channel
    load [b];

}

.....
.....
.....
```

speculative

Identifying Secrets using Tainting/Untainting

Basic idea: taint all the secrets

- Speculatively accessed data (secrets by definition)
- And their dependents

STT ***taints***:

1) Output of speculative access instructions (a)
2) Output of instructions with tainted inputs (b)

```
if (addr < N) {

    // access instruction
    a = load [addr];

    // simple arithmetic
    b = a + 4;

    // cache/mem covert channel
    load [b];

}

.....
.....
.....
```

speculative

Identifying Secrets using Tainting/Untainting

Basic idea: taint all the secrets

- Speculatively accessed data (secrets by definition)
- And their dependents

STT ***taints***:

1) Output of speculative access instructions (a)
2) Output of instructions with tainted inputs (b)

```
if (addr < N) {        ⟵ Resolved!

    // access instruction
    a = load [addr];

    // simple arithmetic
    b = a + 4;

    // cache/mem covert channel
    load [b];

}

.....
.....
.....
```

speculative

Identifying Secrets using Tainting/Untainting

Basic idea: taint all the secrets

- Speculatively accessed data (secrets by definition)
- And their dependents

STT ***taints***:

1) Output of speculative access instructions (a)
2) Output of instructions with tainted inputs (b)

STT ***untaints*** *when:*

1) A speculative access instruction becomes non-speculative (a)

```
if (addr < N) {

    // access instruction
    a = load [addr];

    // simple arithmetic
    b = a + 4;

    // cache/mem covert channel
    load [b];

}

.....
.....
.....
```

← **Resolved**!

speculative

Identifying Secrets using Tainting/Untainting

Basic idea: taint all the secrets

- Speculatively accessed data (secrets by definition)
- And their dependents

STT ***taints***:

1) Output of speculative access instructions (a)
2) Output of instructions with tainted inputs (b)

STT ***untaints*** *when:*

1) A speculative access instruction becomes non-speculative (a)
2) An instruction has all its input untainted (b)

```
if (addr < N) {

    // access instruction
    a = load [addr];

    // simple arithmetic
    b = a + 4;

    // cache/mem covert channel
    load [b];

}

.....
.....
.....
```

Resolved!

speculative

Identifying Secrets using Tainting/Untainting

Basic idea: taint all the secrets

- Speculatively accessed data (secrets by definition)

```
if (addr < N) {        ← Resolved!
```

Data is tainted → Data is speculative (not necessarily other way around)

```
}

.....
.....
.....
```

speculative

STT ***untaints*** *when:*

1) A speculative access instruction becomes non-speculative (a)
2) An instruction has all its input untainted (b)

Microarchitect Identifies …

Instructions forming **explicit channels**

- E.g. load, data-dependent arithmetic

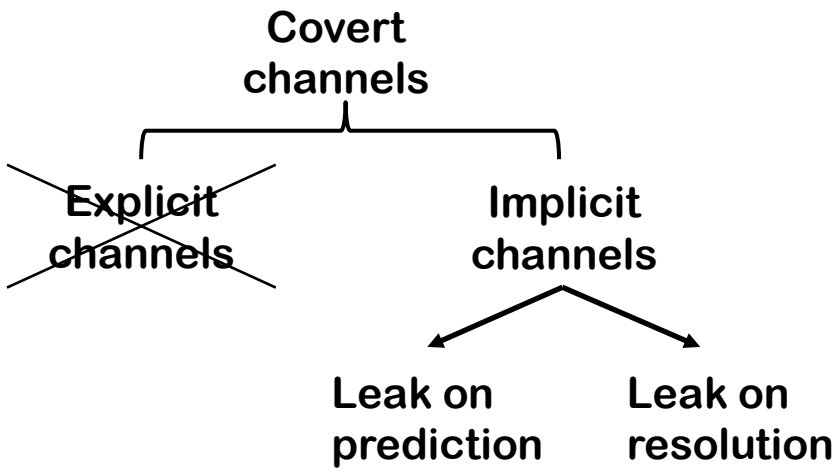
Instructions forming **implicit channels**

- E.g. control-flow instructions, store-load pairs

Blocking Covert Channels

Explicit channels:

- Delay execution until operands **untainted** (e.g., load address)



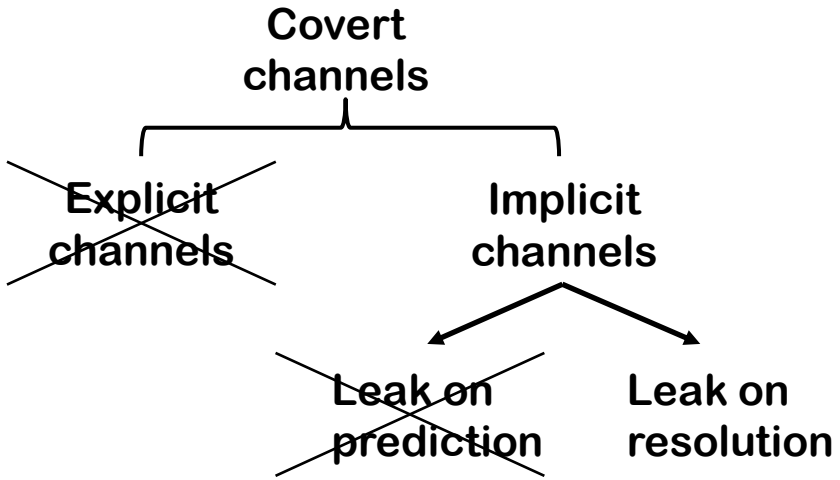
Blocking Covert Channels

Explicit channels:

- Delay execution until operands **untainted** (e.g., load address)

Implicit channels:

- Delay predictor update until branch predicate **untainted**



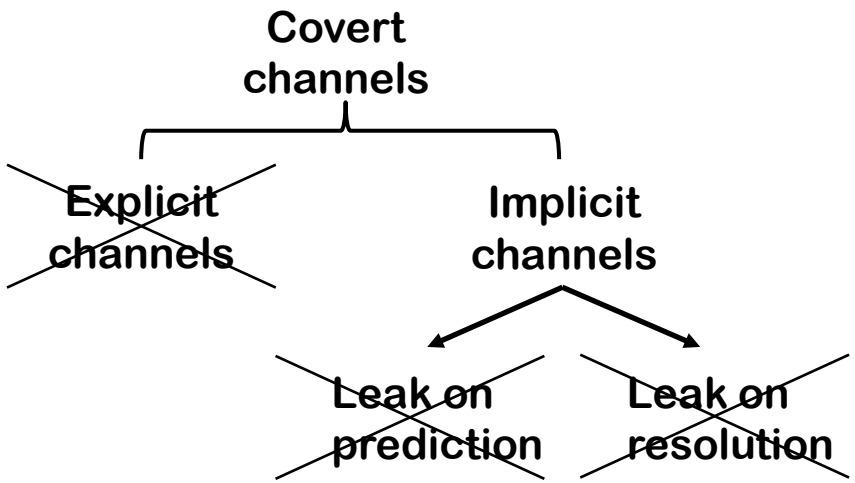
Blocking Covert Channels

Explicit channels:

- Delay execution until operands **untainted** (e.g., load address)

Implicit channels:

- Delay predictor update until branch predicate **untainted**
- Delay resolution until branch predicate **untainted**



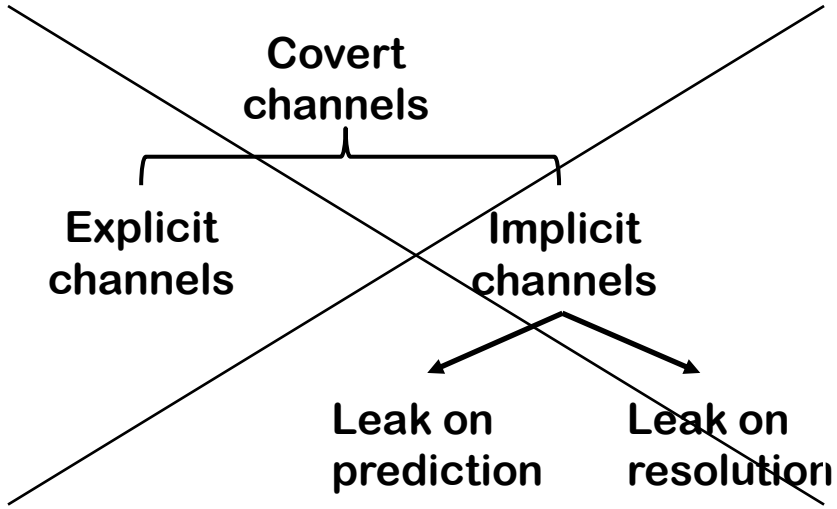
Blocking Covert Channels

Explicit channels:

- Delay execution until operands **untainted** (e.g., load address)

Implicit channels:

- Delay predictor update until branch predicate **untainted**
- Delay resolution until branch predicate **untainted**



What speculative work can we safely do?

Safe to execute all instructions w/ untainted operands

Safe to execute safe (no explicit channel) instructions w/ tainted operands

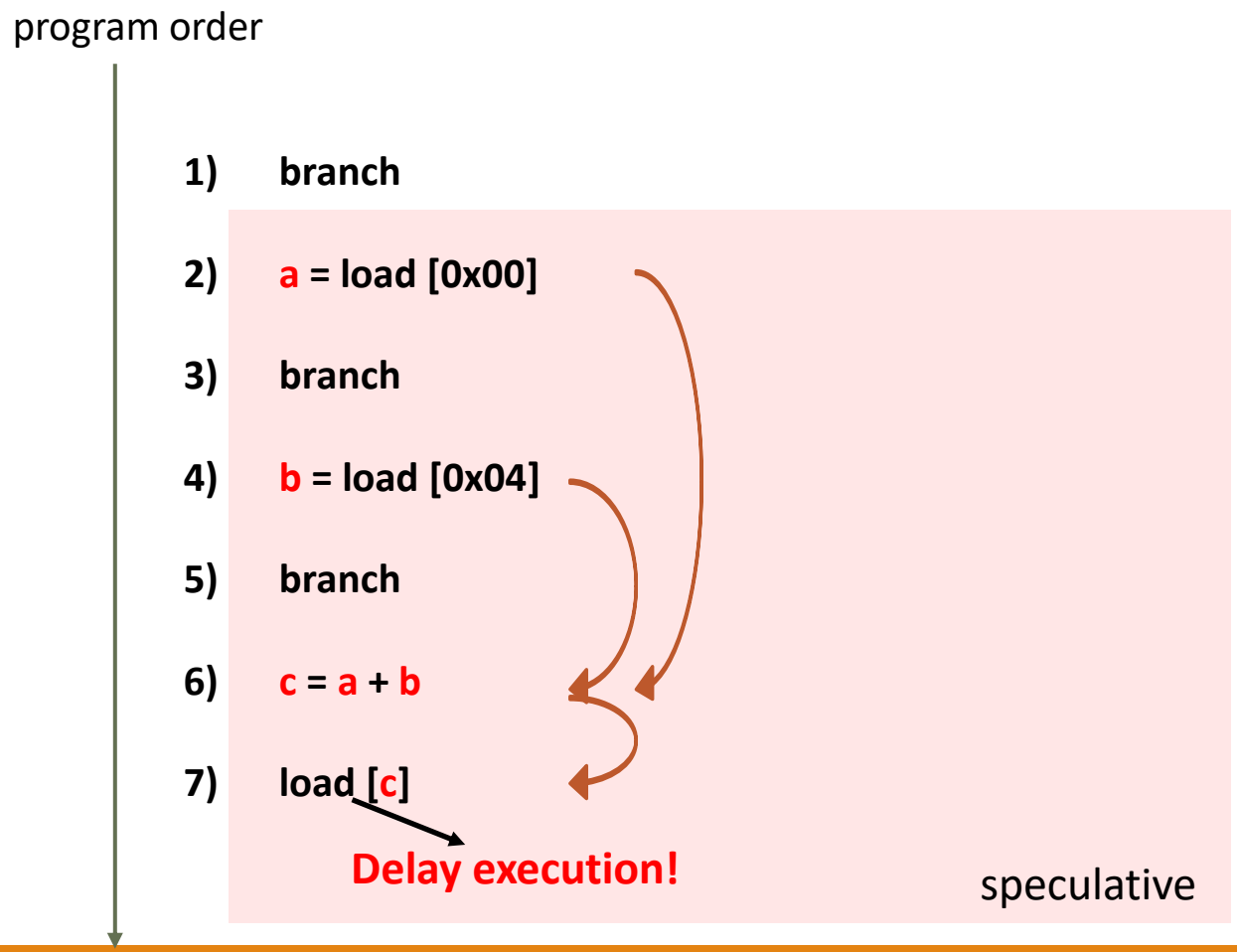
Safe to predict on implicit/explicit branches w/ tainted predicates
Note: predictors have high accuracy.

```
a = 0
if (secret) a+=CACHE_LN_SZ
load(a) // covert channel
```

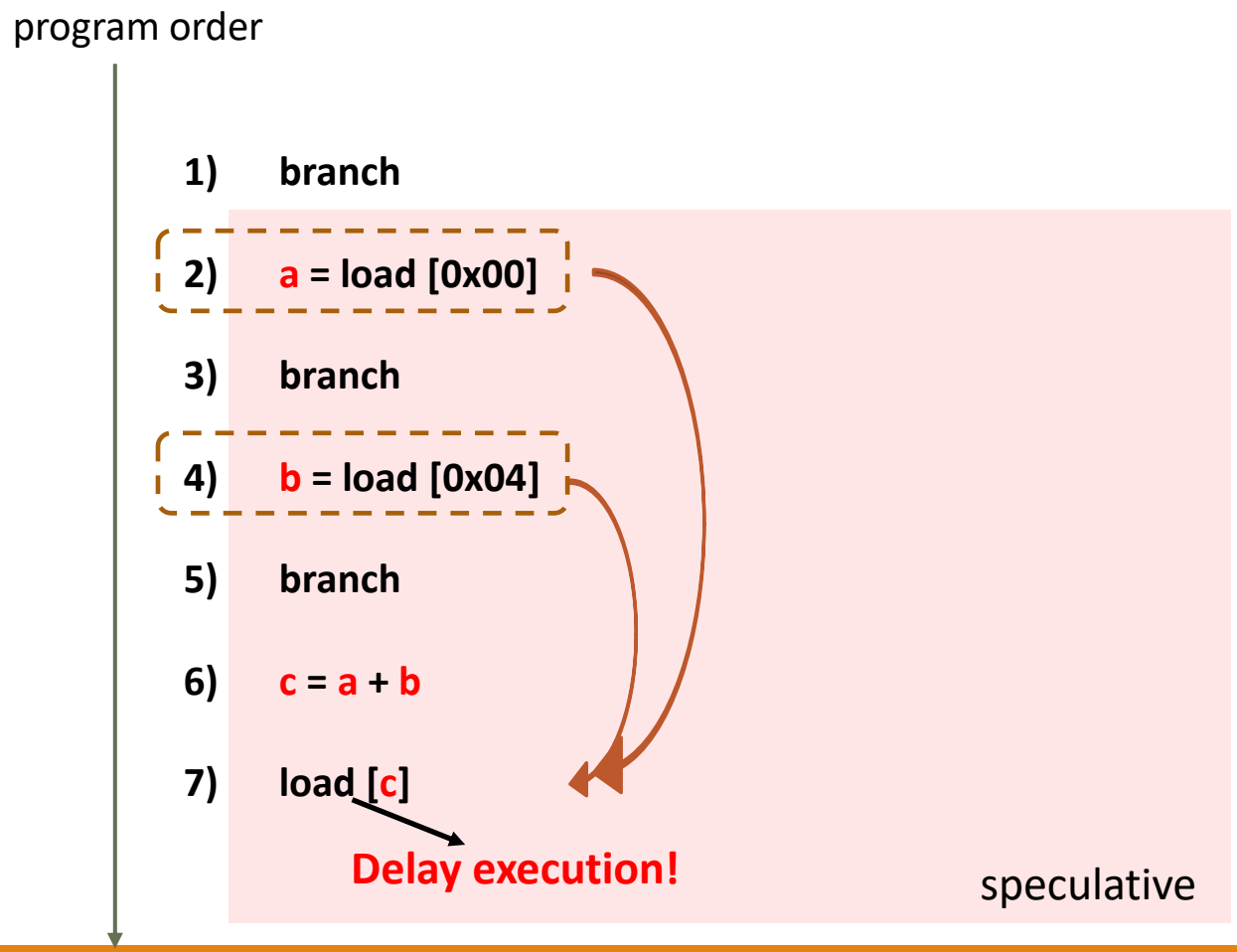
PC = non-sensitive
→ Predictor state = non-sensitive
→ Safe to predict on branch ☺

Hardware Implementation of STT

Efficient Implementation of Tainting/Untainting Logic

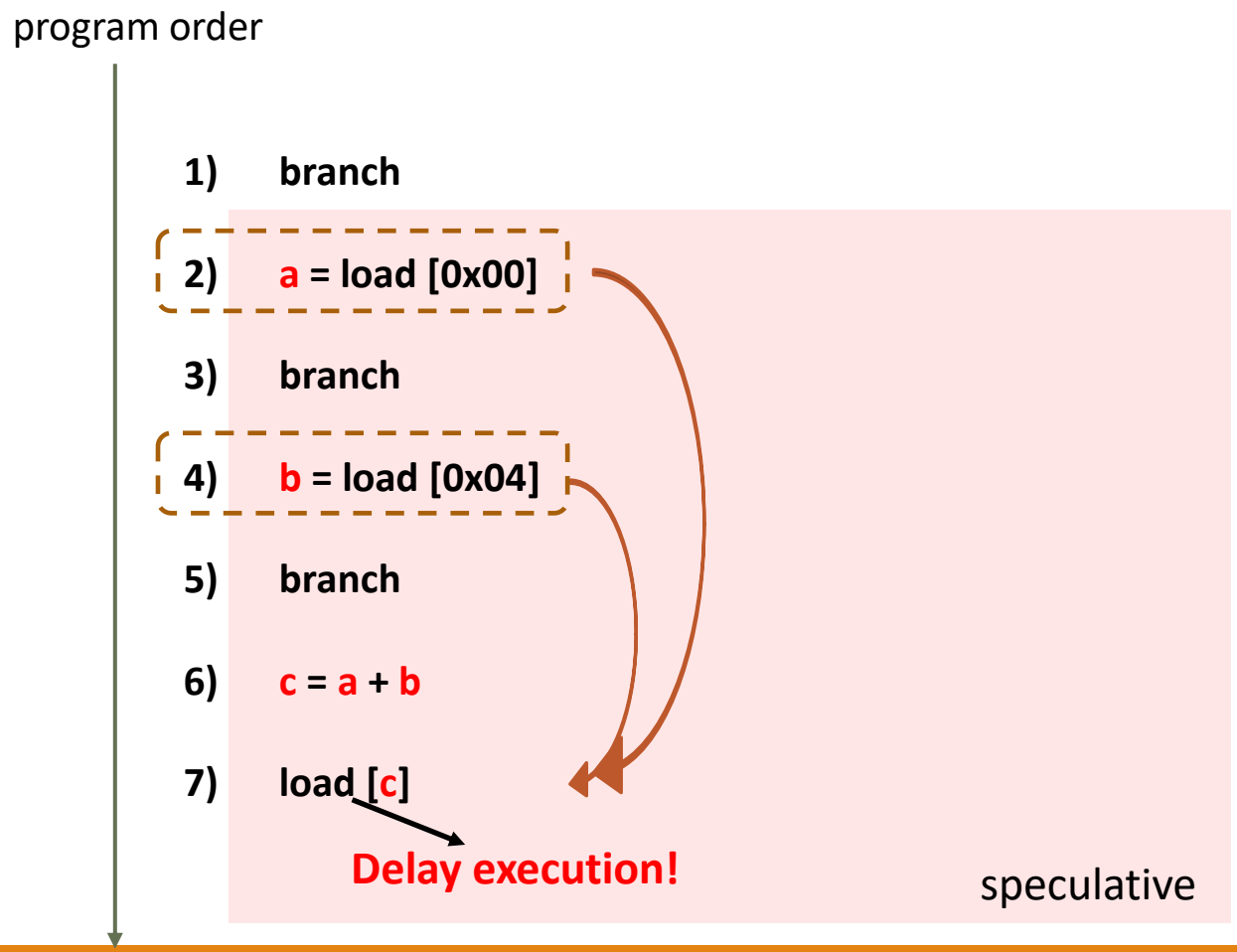


Efficient Implementation of Tainting/Untainting Logic



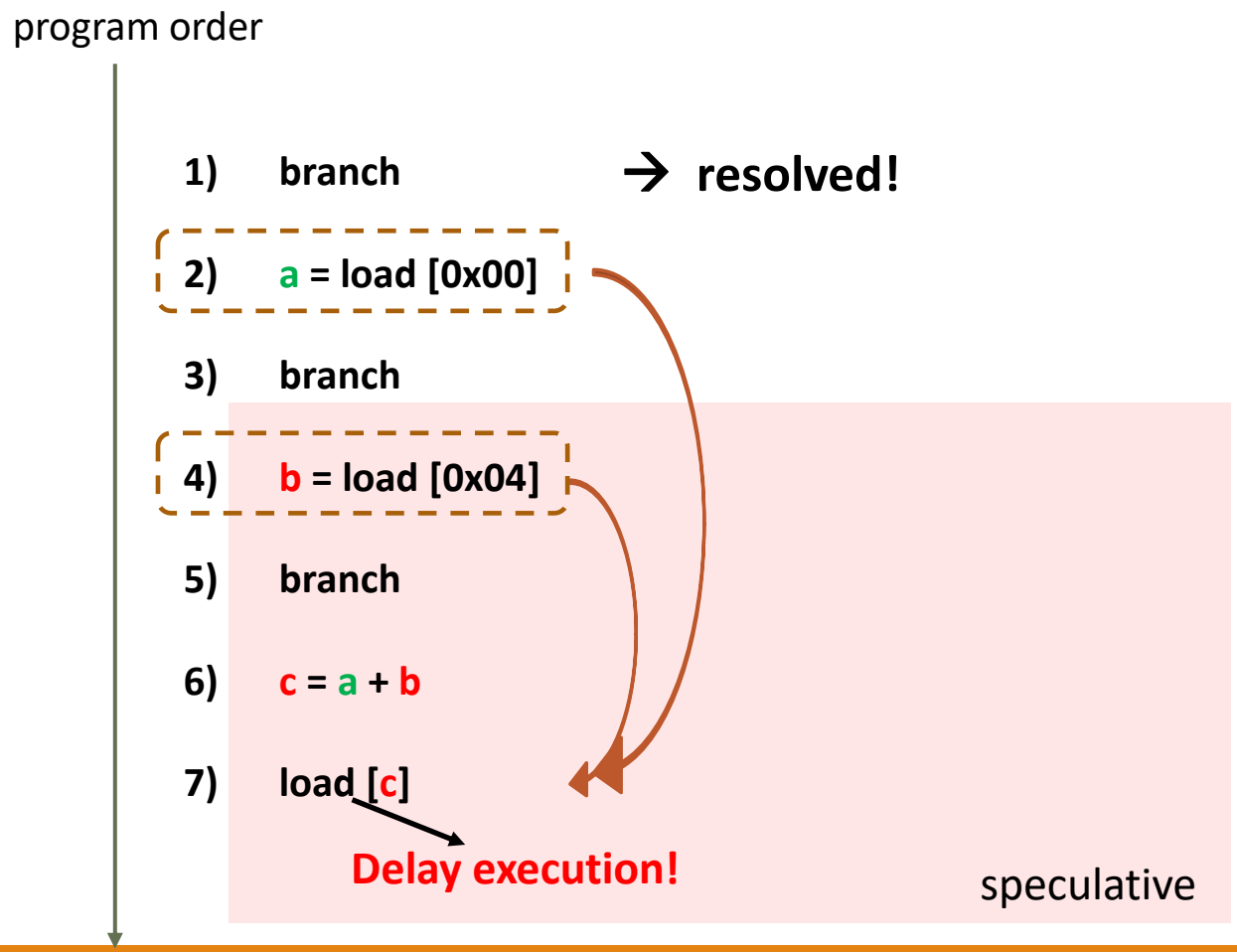
Efficient Implementation of Tainting/Untainting Logic

Observation: All instructions turn non-speculative in-order



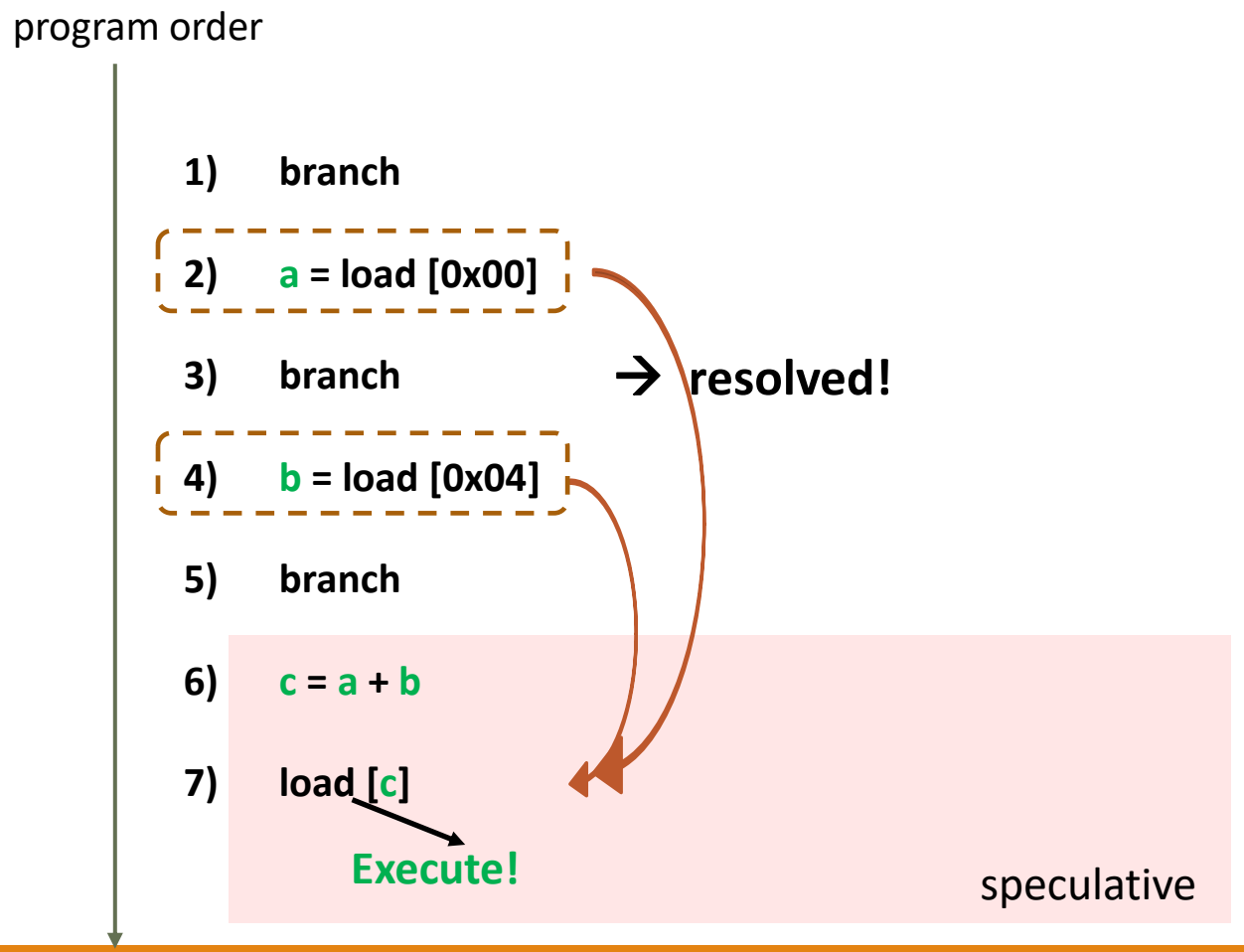
Efficient Implementation of Tainting/Untainting Logic

Observation: All instructions turn non-speculative in-order



Efficient Implementation of Tainting/Untainting Logic

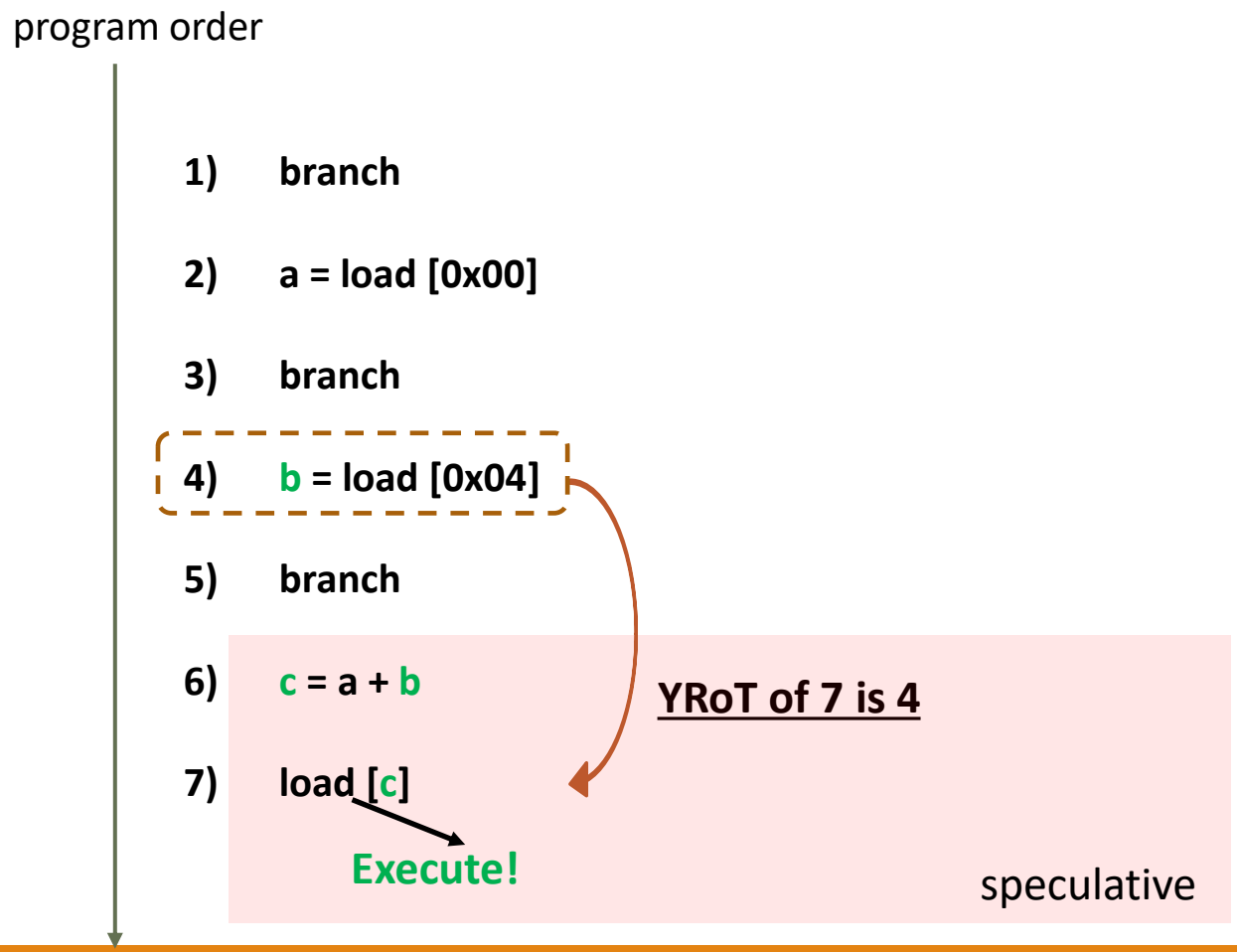
Observation: All instructions turn non-speculative in-order



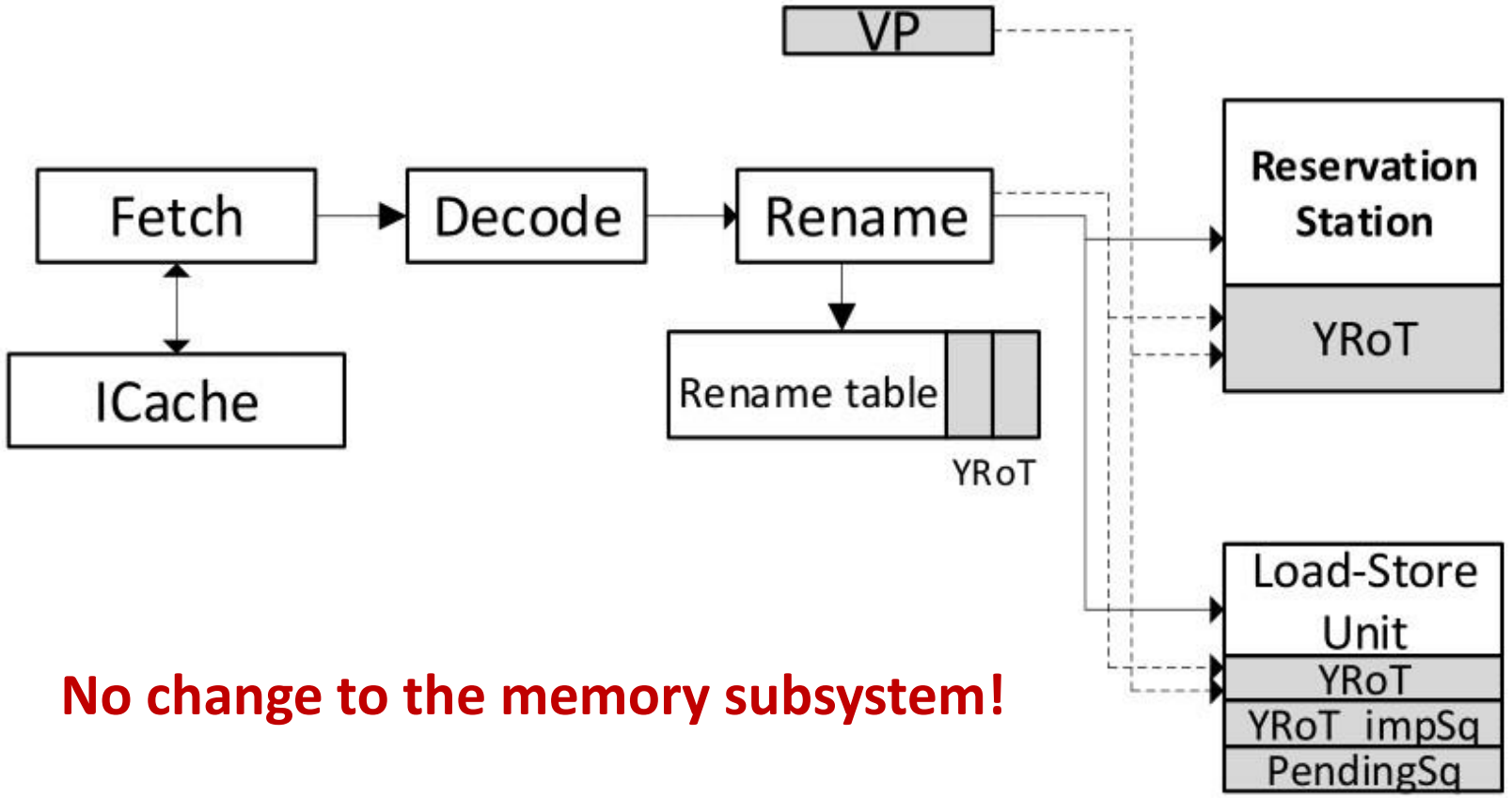
Efficient Implementation of Tainting/Untainting Logic

Observation: All instructions turn non-speculative in-order

Each instruction tracks the "youngest access instruction" it depends on -- ***"Youngest Root of Taint" (YRoT)***



Efficient Implementation of Tainting/Untainting Logic



No change to the memory subsystem!

Security Evaluation

Security definition:

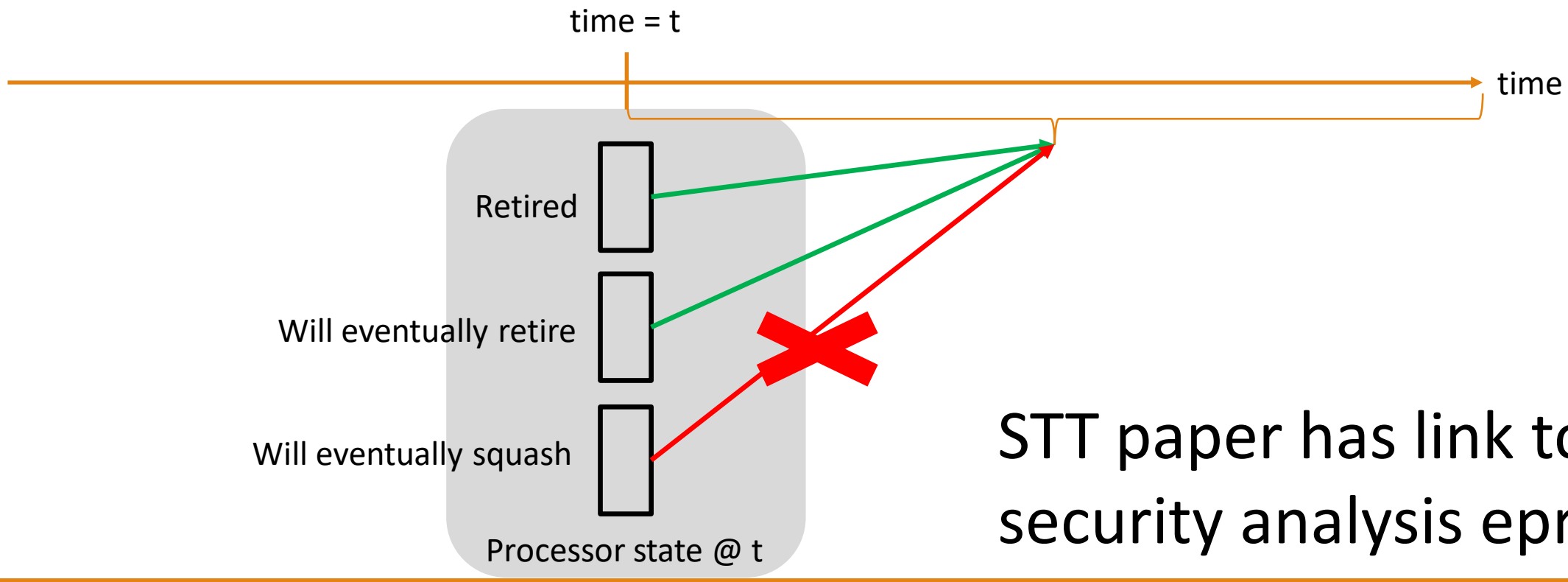
Arbitrary speculative execution can only leak

retired register file state (not arbitrary program memory)

No read gadgets!

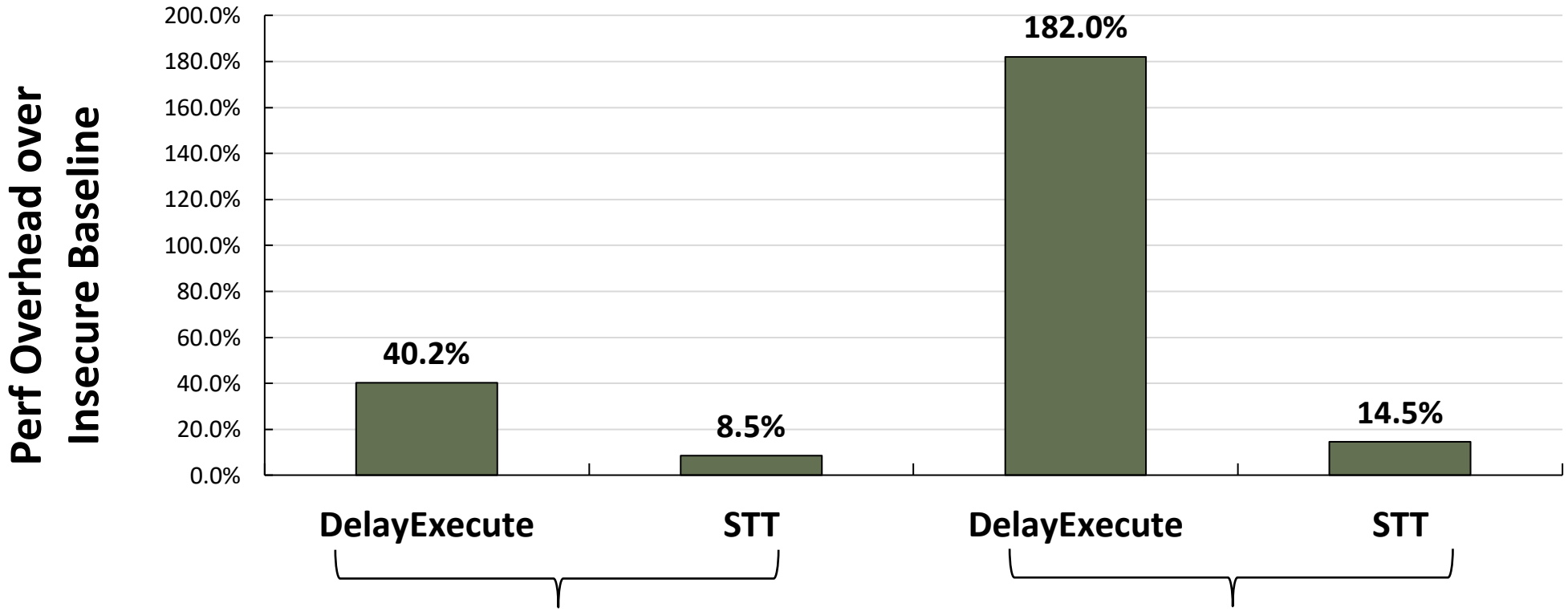
Security Evaluation

STT enforces a non-interference property w.r.t speculatively accessed data:



STT paper has link to security analysis eprint

Performance Evaluation on SPEC2006



Summary

STT Blocks leakage of speculatively accessed data over any uarch covert channels with:

1) High performance

2) Provable security protection

3) No software change; No memory subsystem change

Part 2: Speculative Data-Oblivious Execution (SDO)

PERFORMANCE OPTIMIZATION FRAMEWORK FOR STT

Where does overhead come from in STT?

Explicit channels (a.k.a. transmit instructions):

- Delay execution until operands **untainted** (e.g., load address)

>90% of overhead

Implicit channels:

- Delay predictor update until branch predicate **untainted**
- Delay resolution until branch predicate **untainted**

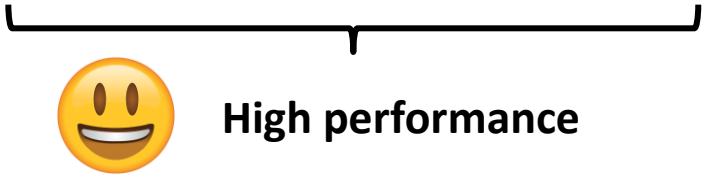
```
if (addr < N) {   // speculation

    // access instruction
    secret = load [addr];

    // transmit instruction
    transmit secret;

}
```

E.g., loads, floating point, …
→ Delay execution

Speculative Data Oblivious (SDO): Executive Summary

Speculative Data Oblivious (SDO): Executive Summary

Idea 1. Execute `transmit secret`



Speculative Data Oblivious (SDO): Executive Summary

Idea 1. Execute `transmit secret` by eliminating operand-dependent hardware usage (being data oblivious)

Speculative Data Oblivious (SDO): Executive Summary

Idea 1. Execute `transmit secret` by eliminating operand-dependent hardware usage (being data oblivious)

Idea 2. Predict how the execution should be performed

Speculative Data Oblivious (SDO): Executive Summary

Idea 1. Execute `transmit secret` by eliminating operand-dependent hardware usage (being data oblivious)

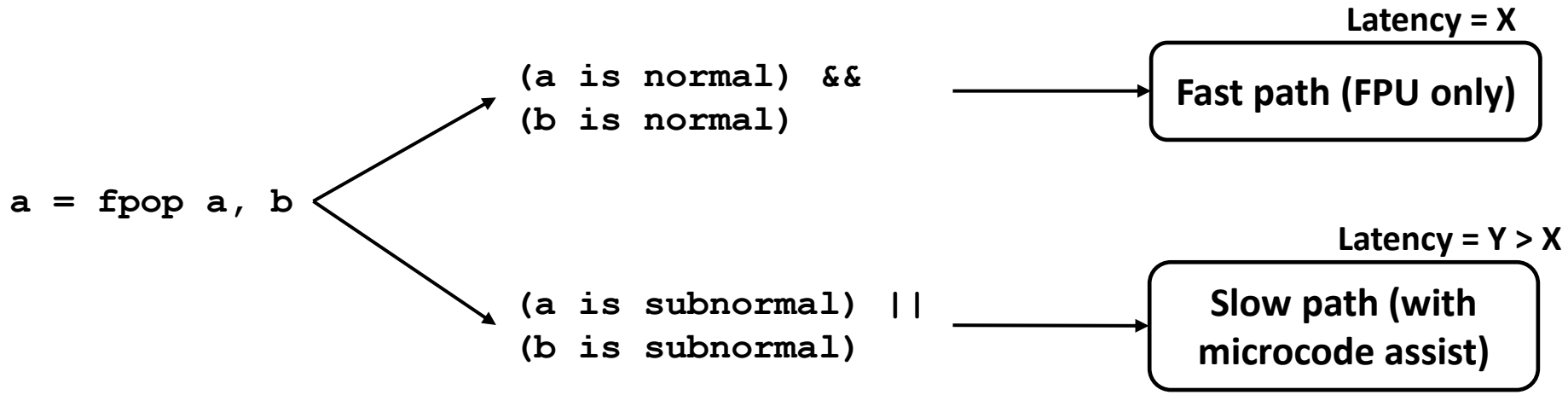
Idea 2. Predict how the execution should be performed

Problem: combining idea 1 & 2 creates security problems

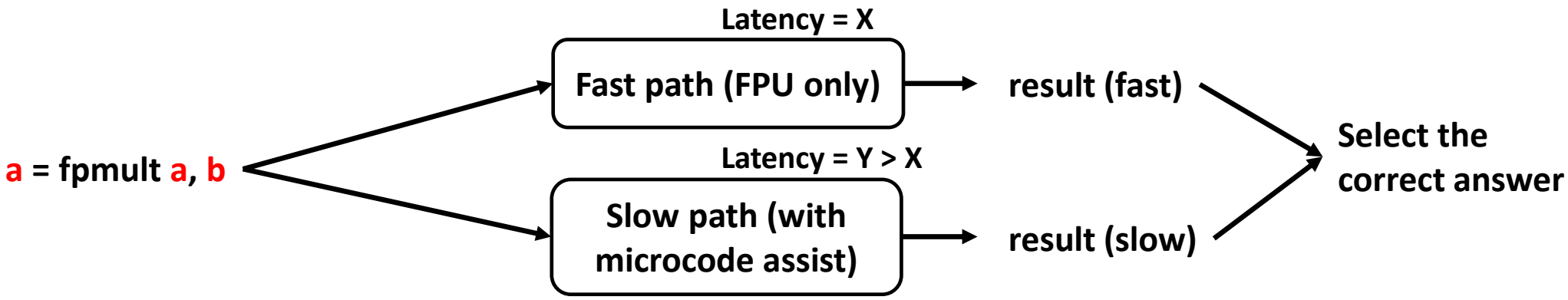
Solution: build on top of Speculative Taint Tracking (STT)

Example: Subnormal Floating-point Operation

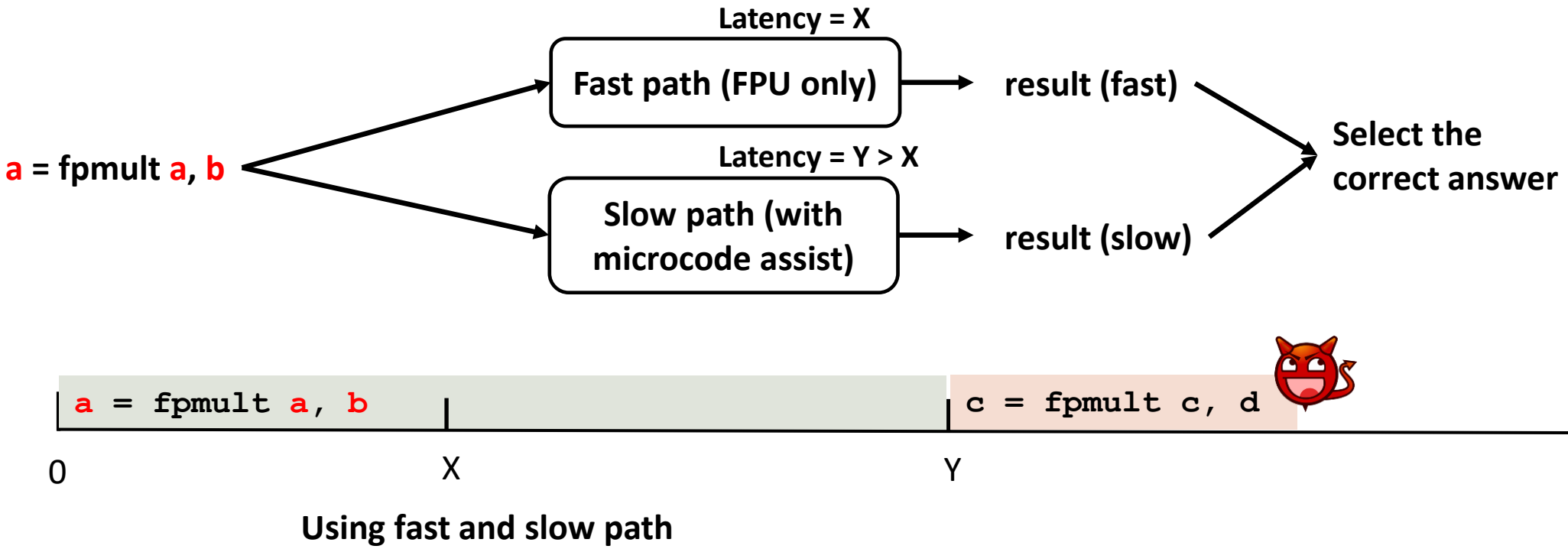
- Double-precision floating point
  - Normal input: (2.23e−308, 1.79e308), processed by Floating-Point Unit (FPU)
  - Subnormal input: (4.9e−324, 2.23e−308), requiring microcode assist



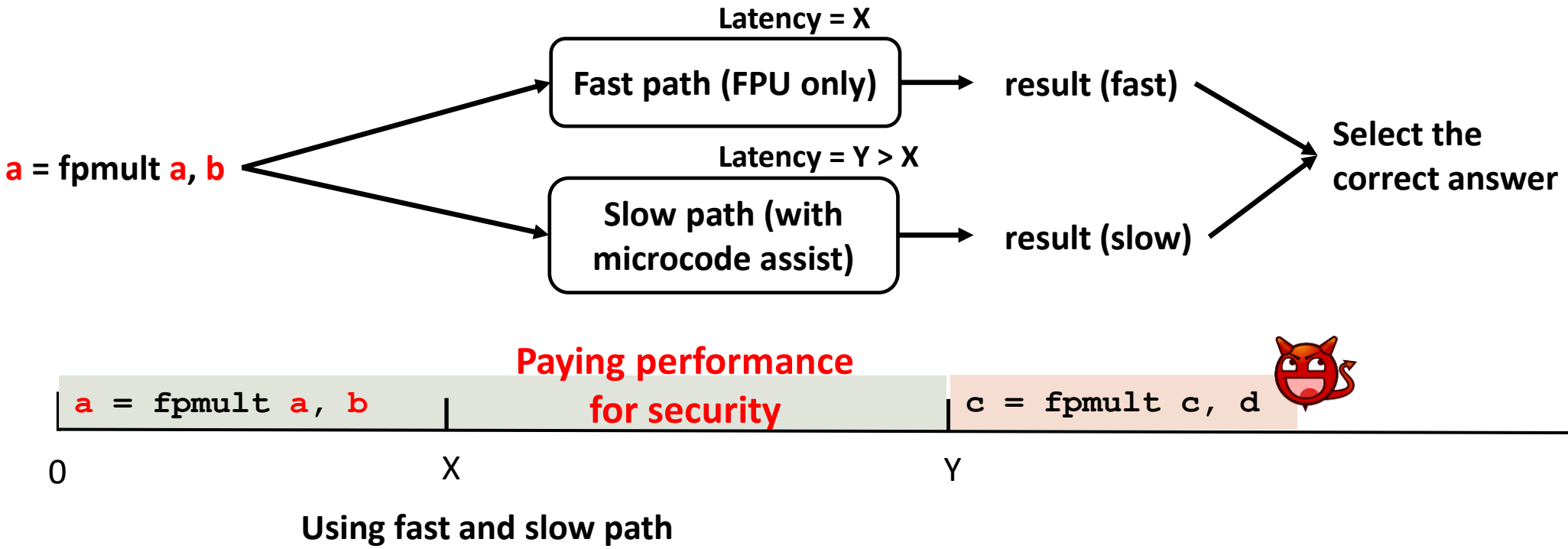
Idea 1: Being Data Oblivious



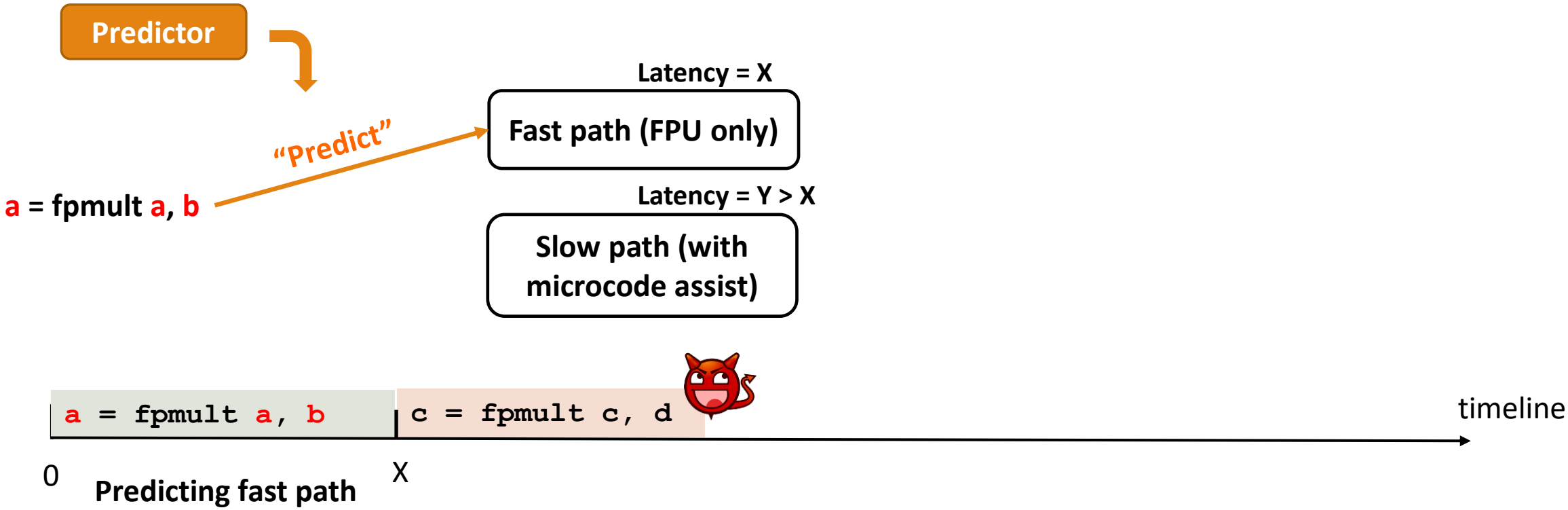
Idea 1: Being Data Oblivious



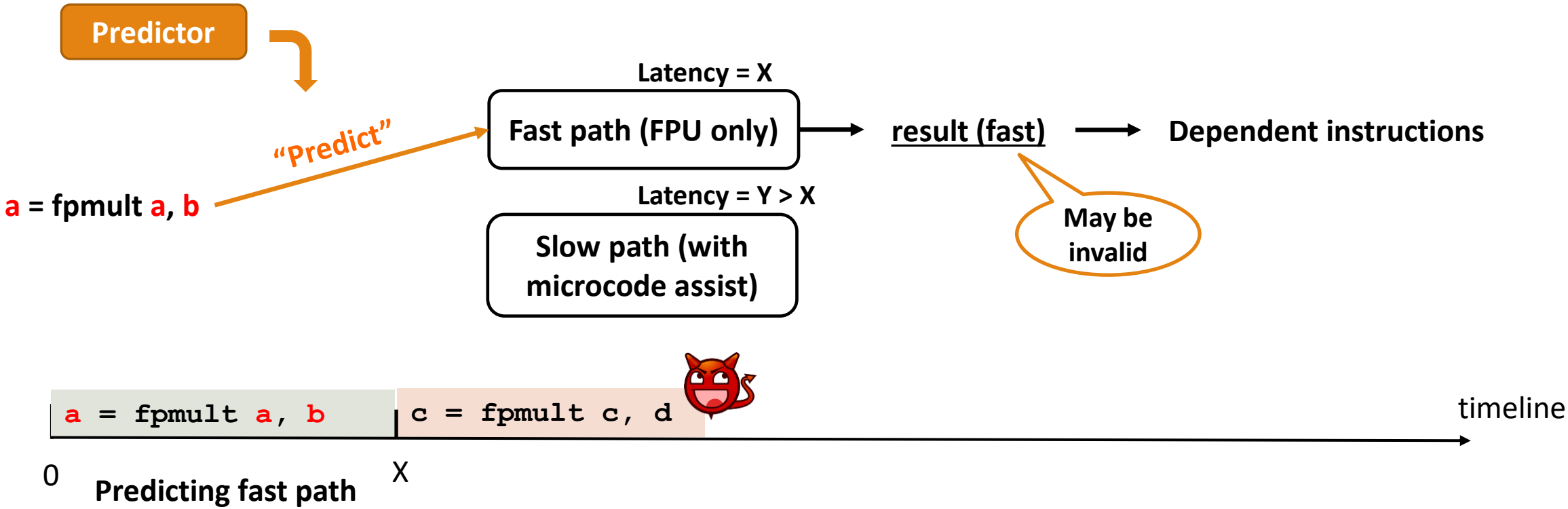
Idea 1: Being Data Oblivious



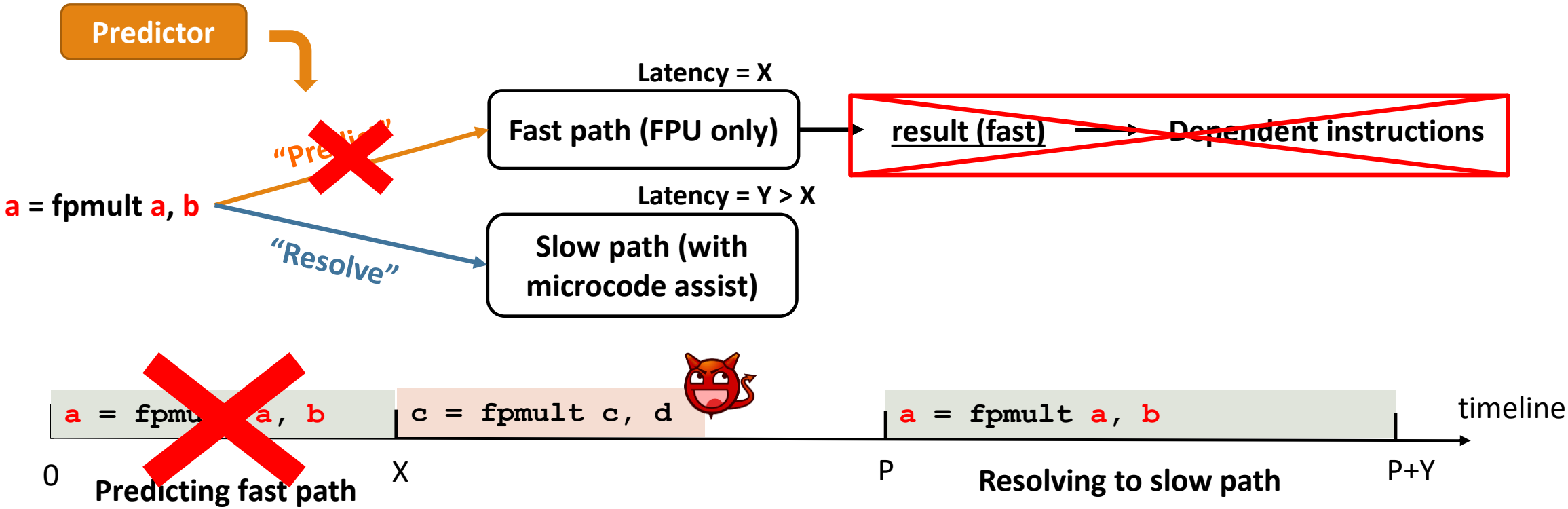
Idea 2: “Predicting” Execution to Perform



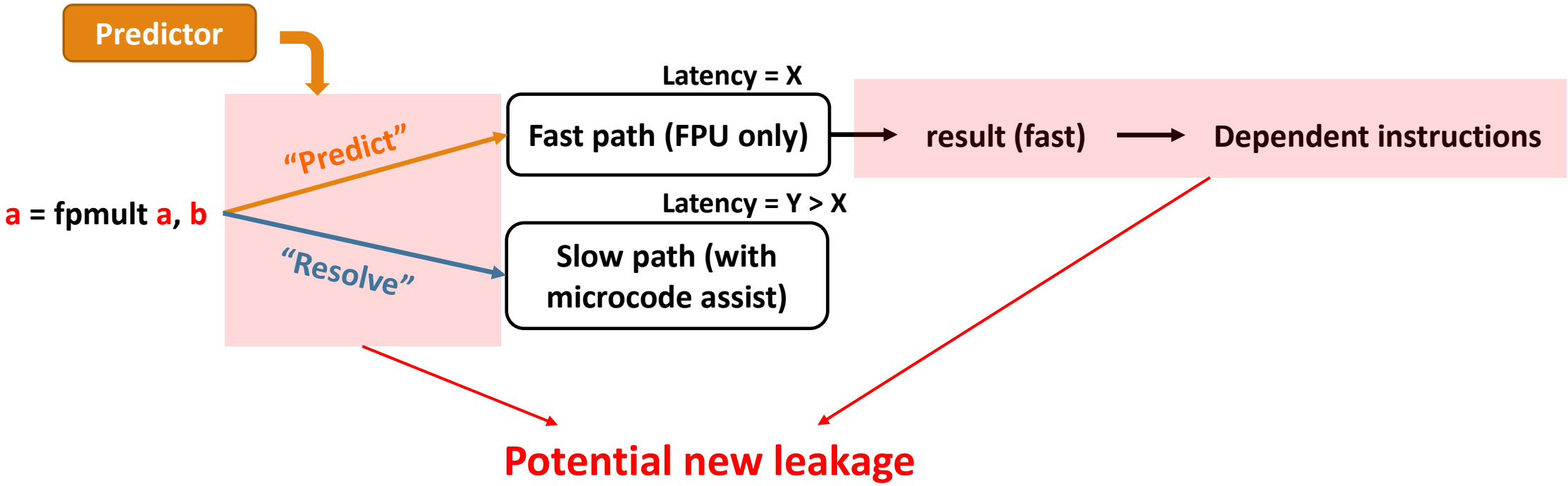
Idea 2: “Predicting” Execution to Perform



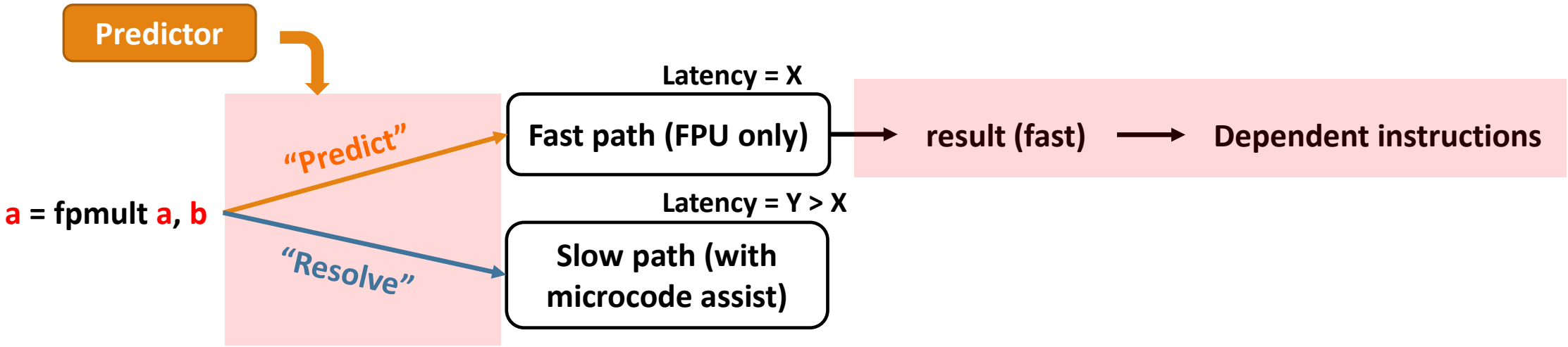
Idea 2: “Predicting” Execution to Perform



Idea 2: “Predicting” Execution to Perform

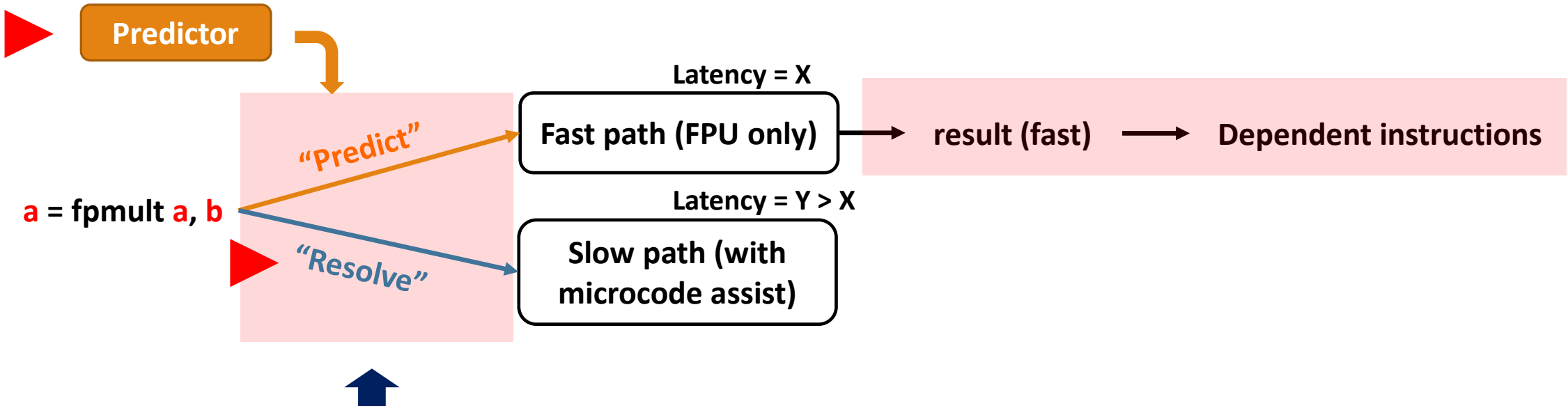


Applying STT for Security

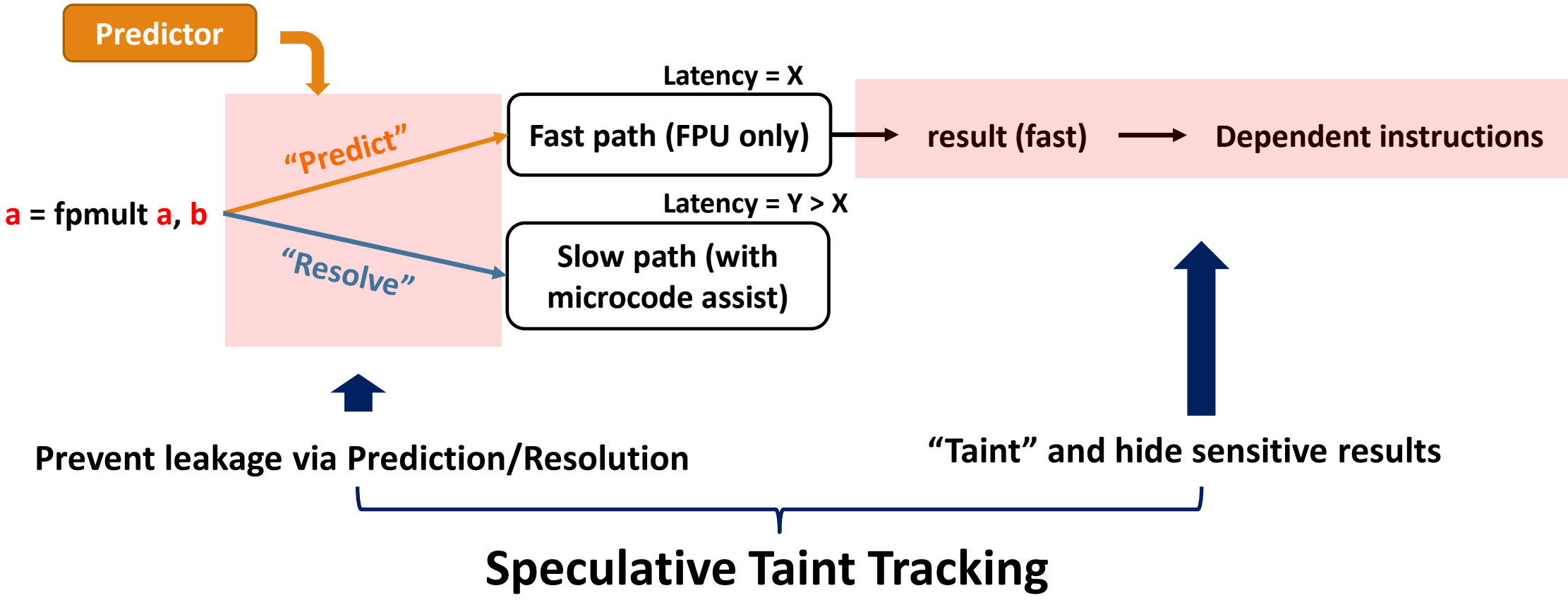


Speculative Taint Tracking

Applying STT for Security



Applying STT for Security



Applying STT for Security

How STT "**prevents leakage via prediction/resolution**":

- Never update predictors with any secret information
- Delay resolution until safe

Applying STT for Security

How STT "**prevents leakage via prediction/resolution**":

- Never update predictors with any secret information
- Delay resolution until safe

How STT "**taints and hides sensitive results**":

- Sensitive data is marked tainted
- Taint propagates through program dataflow
- Transmitters with tainted arguments are handled safely

Applying STT for Security

How STT "**prevents leakage via prediction/resolution**"

STT Makes Prediction SAFE Again!

We build predictors to reduce defense overhead

- Taint propagates through program dataflow
- Transmitters with tainted arguments are handled safely

Speculative Data Oblivious Execution (SDO)

Idea 1. Safely execute transmitters in a data-oblivious (DO) manner

Idea 2. Predict how the execution should be performed

Data Oblivious variants + Predicting which variant

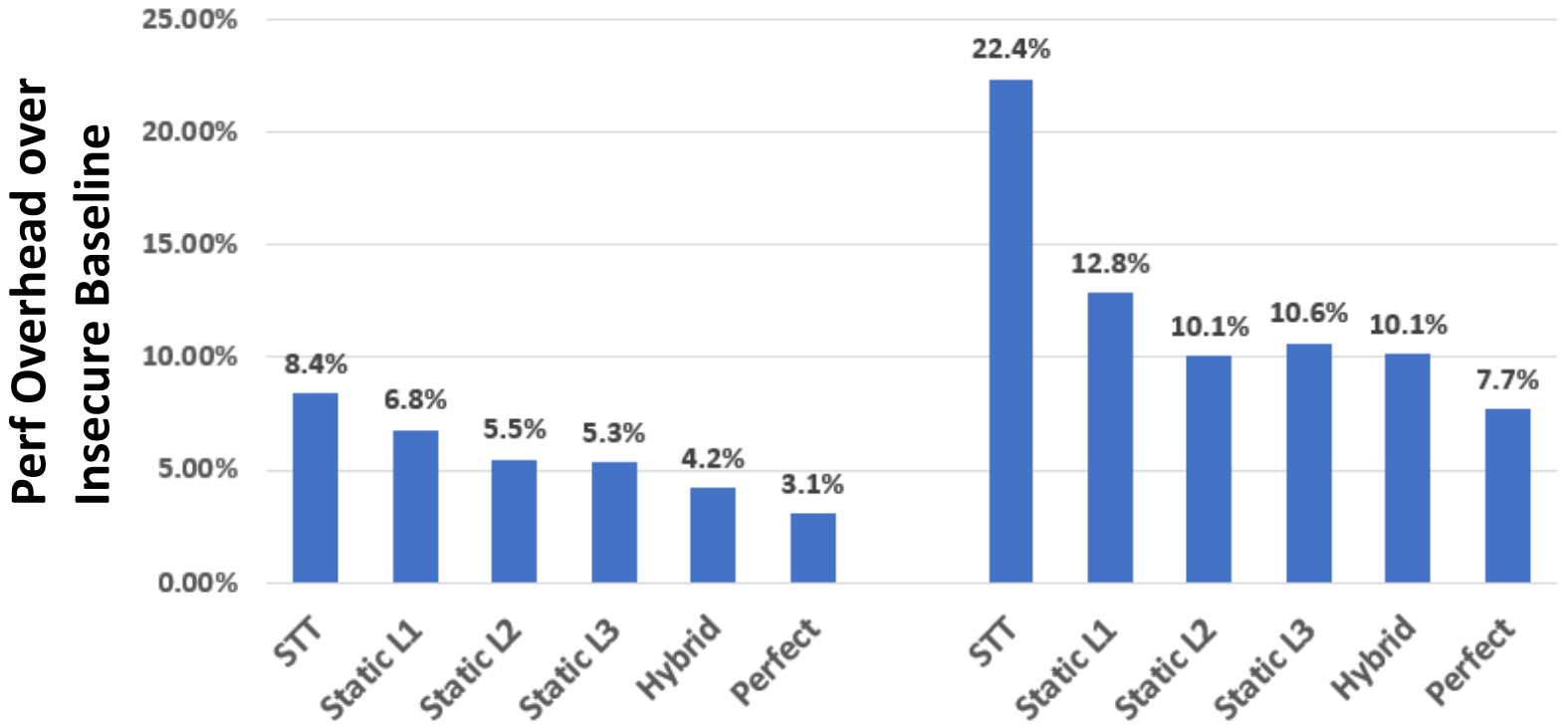
+ Safe Prediction with STT

=

SDO

Net result: execute unsafe transmitters early and safely

Performance Evaluation on SPEC2017



"Spectre" attack model
Consider control-flow speculation

"Futuristic" attack model
Consider all types of speculation

Transmitters:

- **Load**
- **Floating-point multiplication**
- **Floating-point division**

Static L1: always predicting DO-ld_{L1}
Static L2: always predicting DO-ld_{L2}
Static L3: always predicting DO-ld_{L3}
Hybrid: using the hybrid predictor
Perfect: prediction is accurate and precise

Conclusion

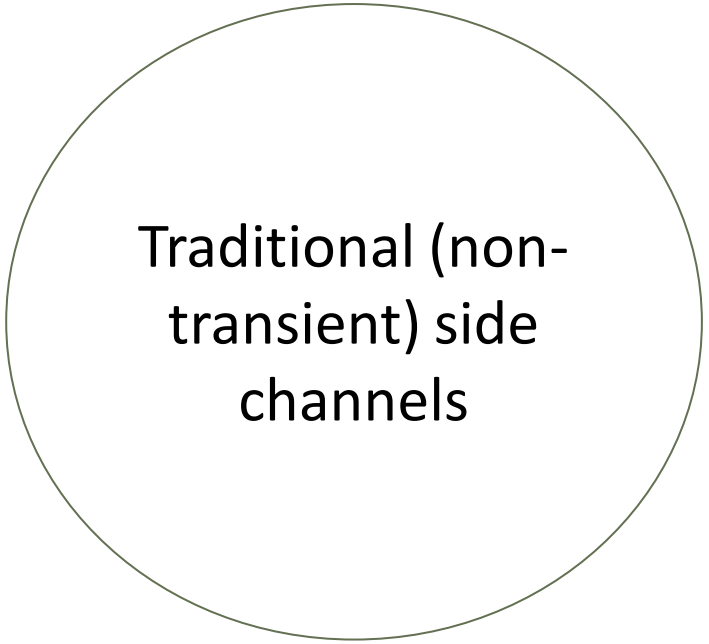
Data Oblivious variants + Predicting which variant + Safe Prediction with STT

=

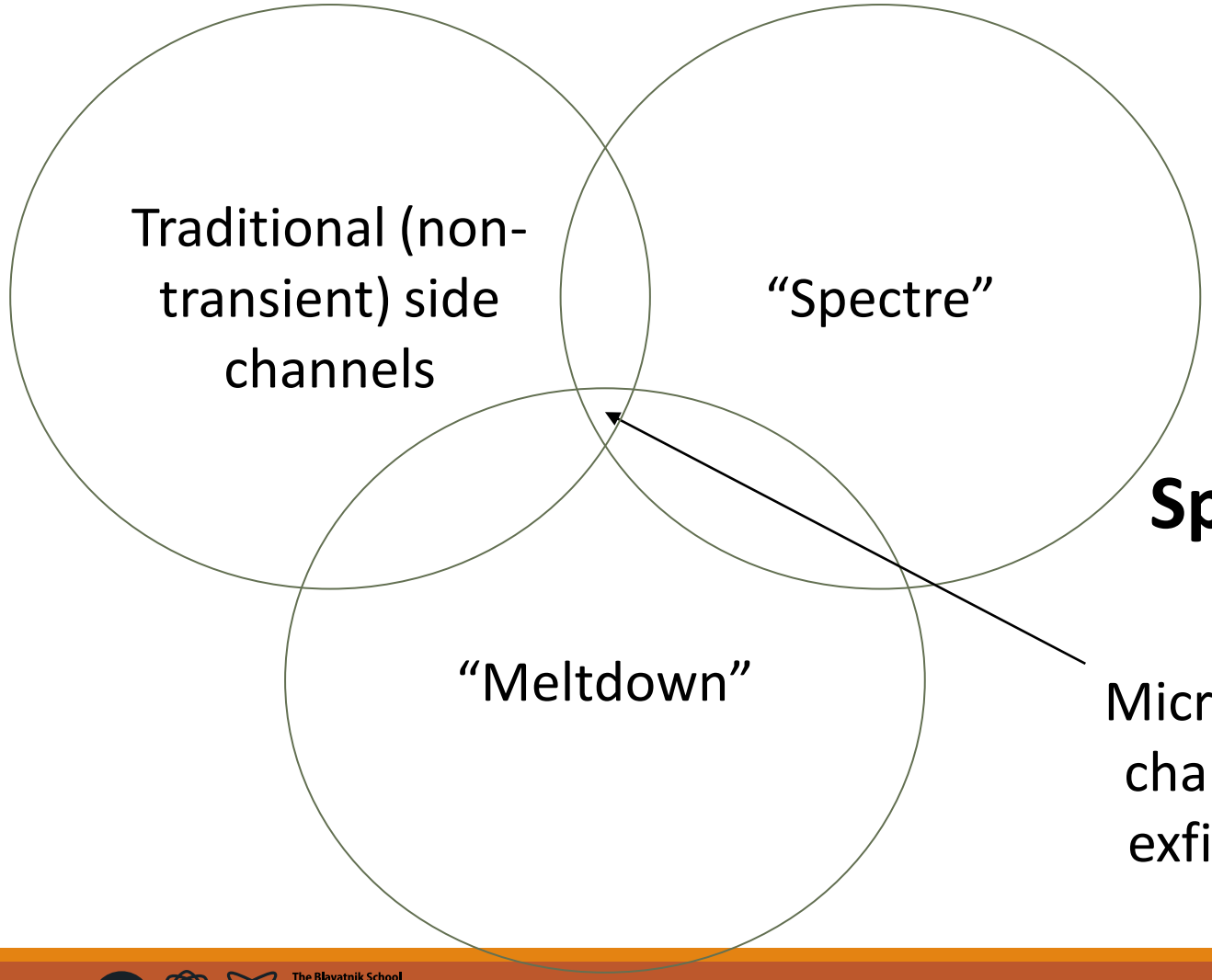
Safe, early execution of transmitters

Part 3: Where things are going (my view) + some new read gadgets

Pre 2018



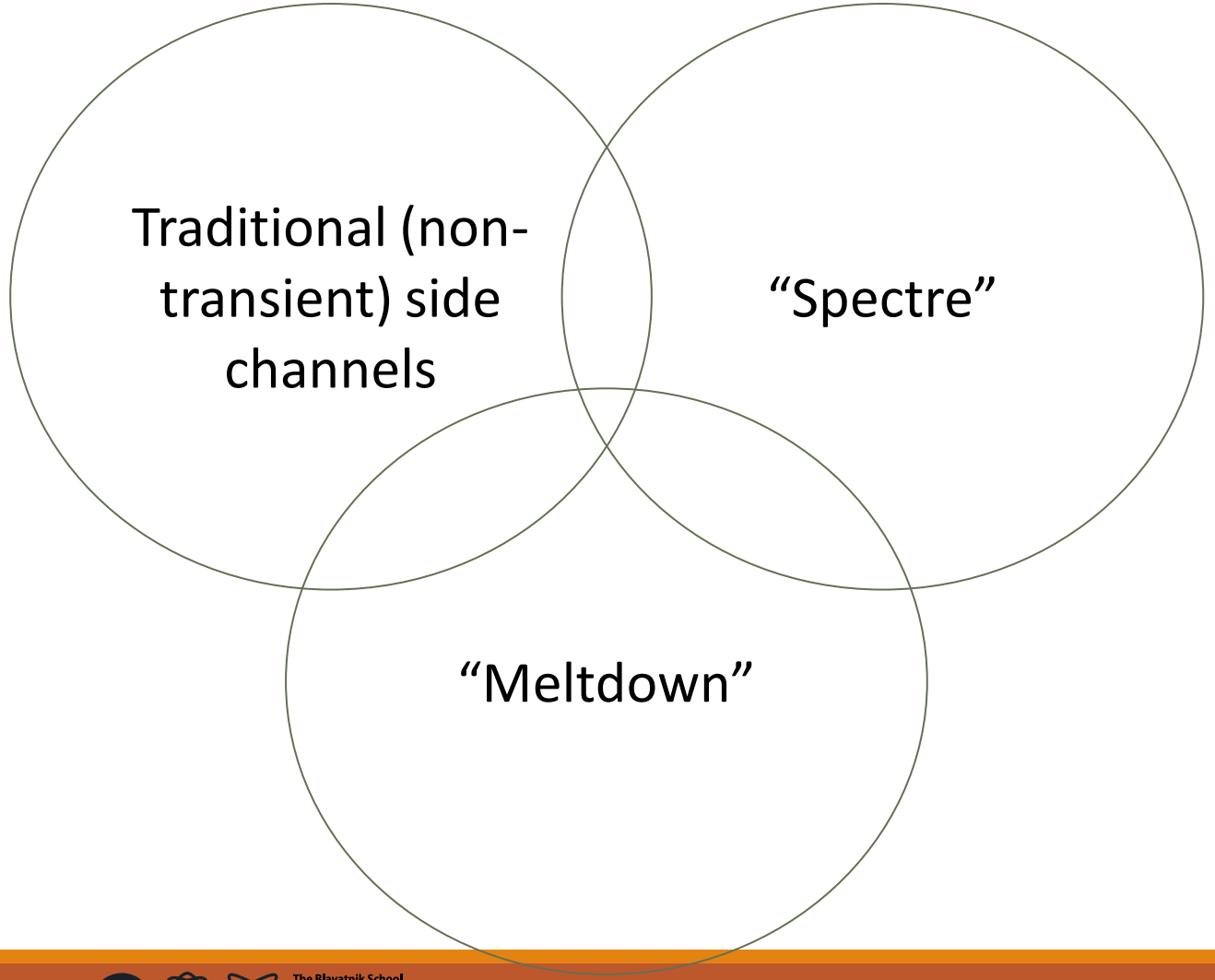
Post 2018



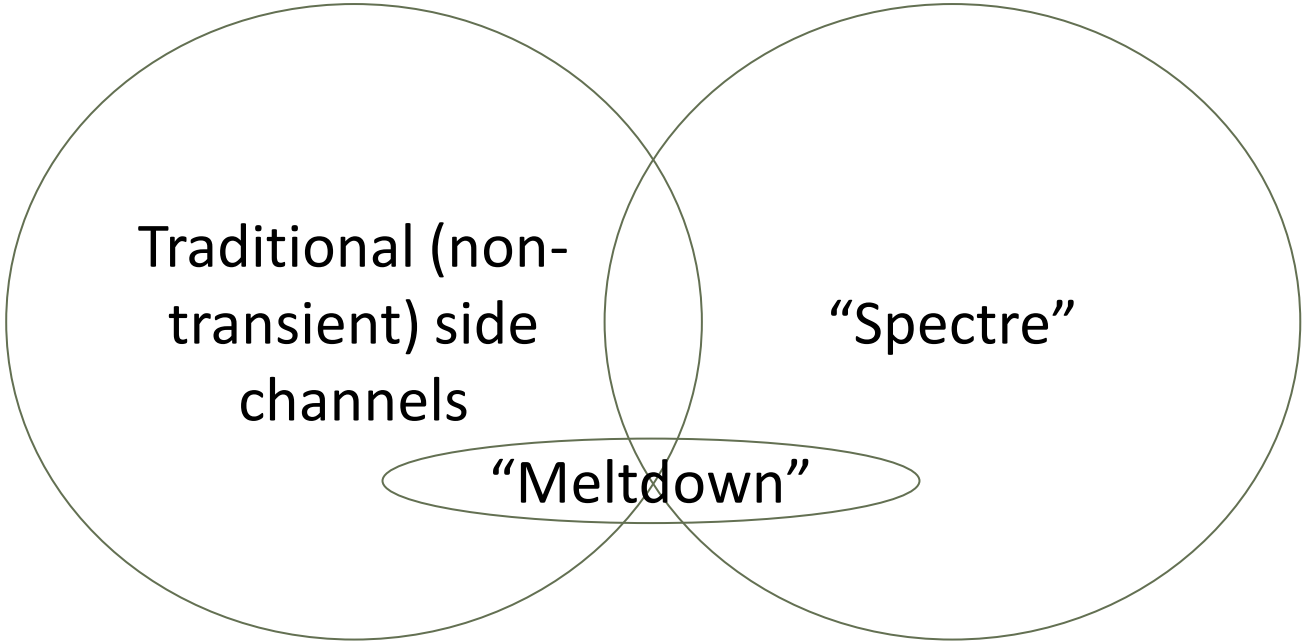
Speculative/Transient execution attacks

Microarchitectural channels used to exfiltrate secrets

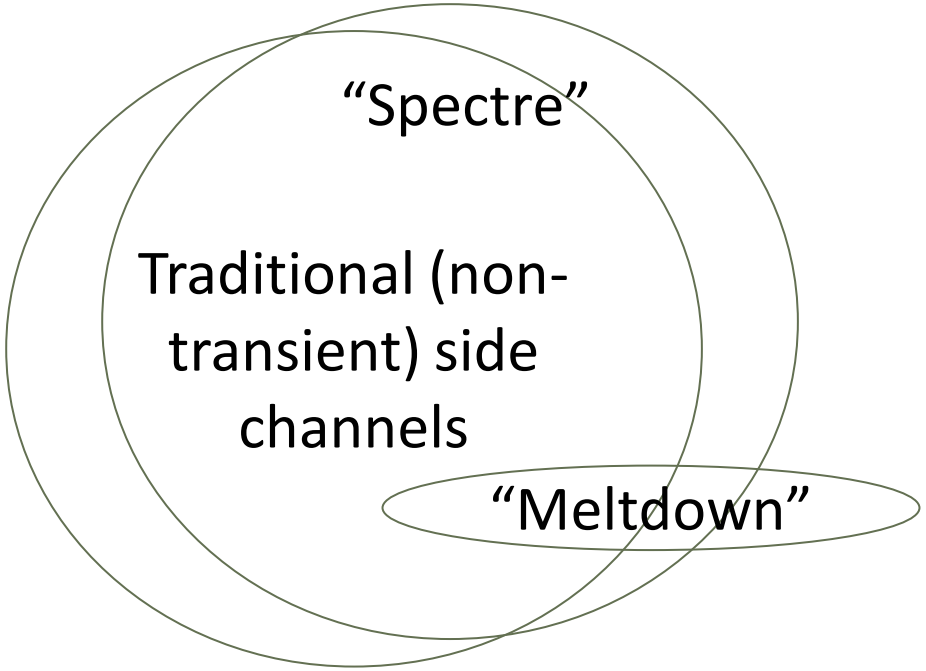
Post 2018


Traditional (non-transient) side channels
“Spectre”
“Meltdown”

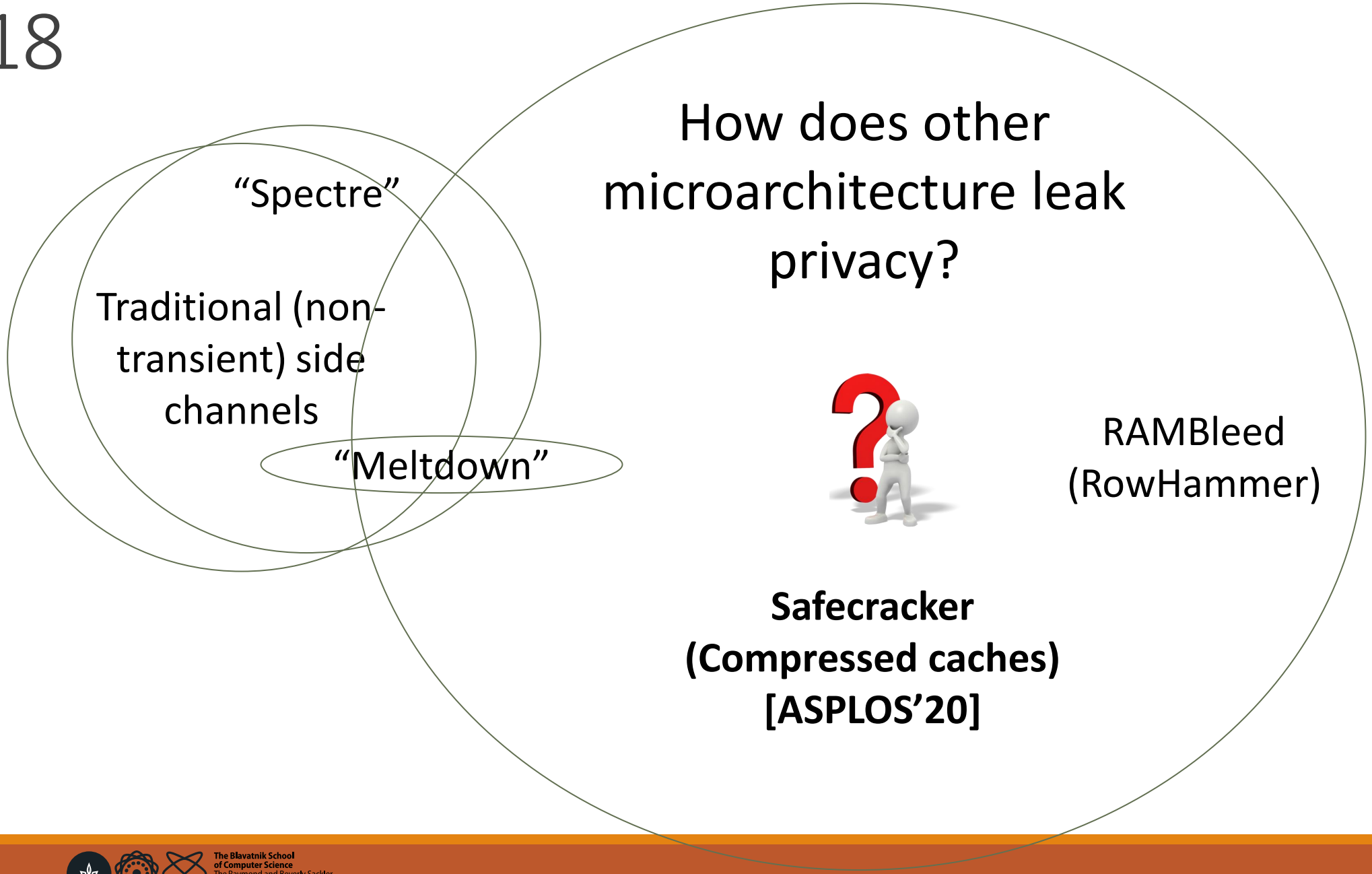
Post 2018


Traditional (non-transient) side channels
“Spectre”
“Meltdown”

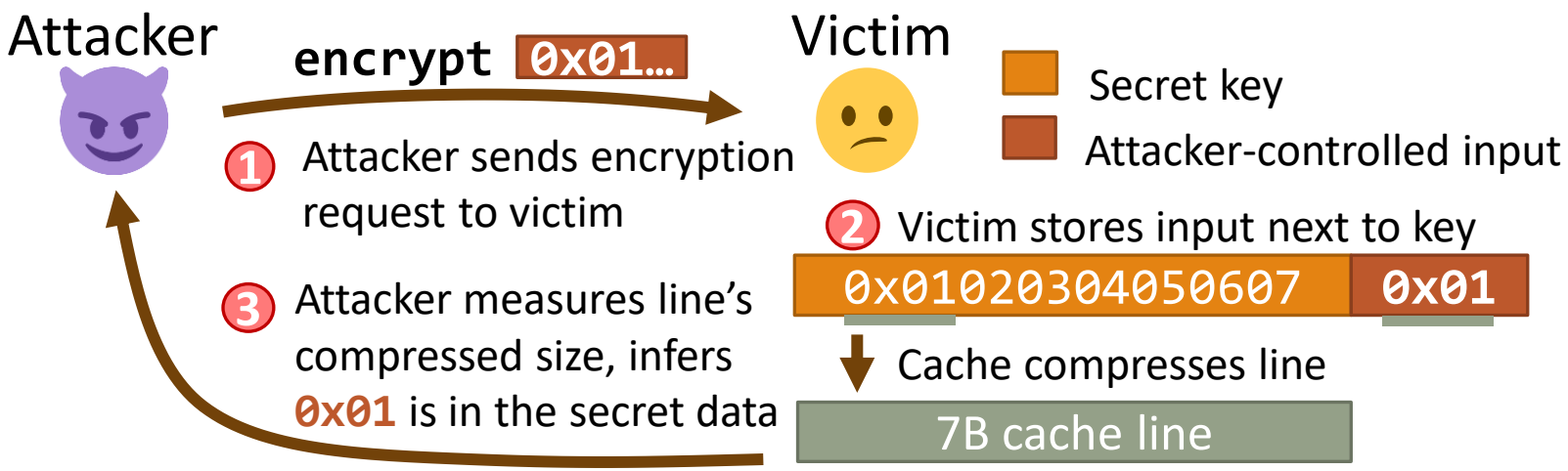
Post 2018


“Spectre”
Traditional (non-transient) side channels
“Meltdown”

Post 2018


How does other microarchitecture leak privacy?
“Spectre”
Traditional (non-transient) side channels
“Meltdown”
RAMBleed (RowHammer)
Safecracker (Compressed caches) [ASPLOS’20]

Compressed Cache Attacks



Read Gadgets from Compressed Cache

Co-locate attacker data w/ secret data → leak secret data
Numerous ways to co-locate data.

HEARTBLEED-LIKE

```
p = malloc(SZ);
memcpy(p, usr_data, SZ);
```

BROP-LIKE

Given:
* re-startable service
* “buffer overflow”

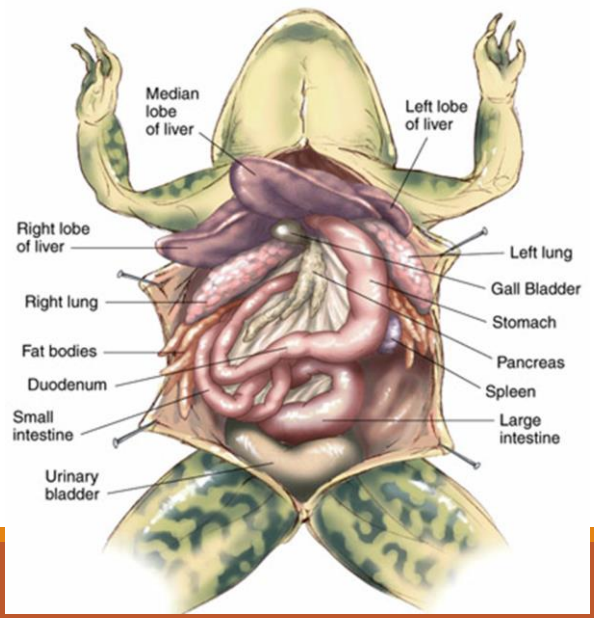
1.) Overflow buffer to size N, guess byte N+1
2.) Repeat (1) until byte N+1 leaked
3.) N++; Goto (1).

Conclusion

In crypto/info flow, we usually ask: do *any* secret bits leak?

In HW, need to ask when *all* bits can leak.

Need new abstractions/defenses to reason about leakage.



≠