# Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data

JIYONG YU, MENGJIA YAN, ARTEM KHYZHA*,
ADAM MORRISON*, JOSEP TORRELLAS, CHRISTOPHER W. FLETCHER
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN *TEL AVIV UNIVERSITY

# Processors are Insecure

# Processors are Insecure

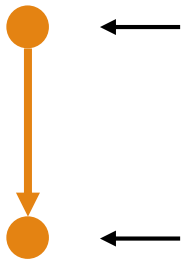

# Speculative Execution Attacks

Speculation starts

```
// Spectre Variant 1

if (addr < N) {    // speculation

    // access instruction
    spec_val = load [addr];

    // covert channel
    load [spec_val];

}
```

# Speculative Execution Attacks

```
// Spectre Variant 1

if (addr < N) {    // speculation

    // access instruction
    spec_val = load [addr];

    // covert channel
    load [spec_val];

}
```



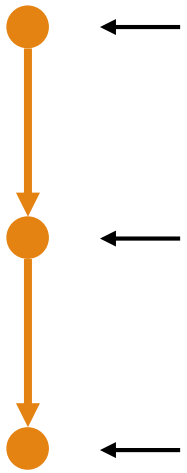
Speculation starts

**Speculative access instruction***
accesses secret

*: Kiriansky, Vladimir, et al. "DAWG: A defense against cache timing attacks in speculative execution processors." *MICRO-51*, 2018.

# 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

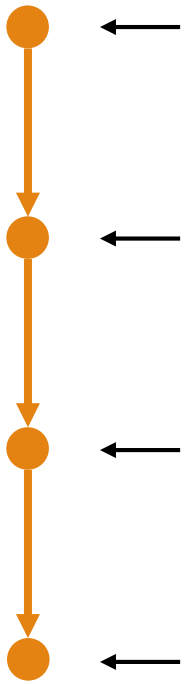
Creates a **covert channel**
to leak secret

*: Kiriansky, Vladimir, et al. "DAWG: A defense against cache timing attacks in speculative execution processors." *MICRO-51*, 2018.

# 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

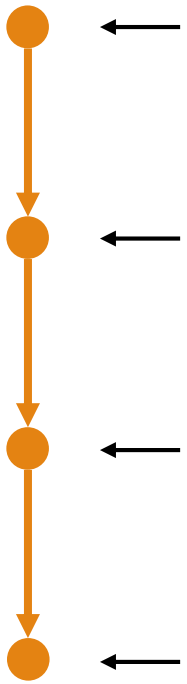
Creates a **covert channel**
to leak secret

Speculation ends -
misspeculation!

*: Kiriansky, Vladimir, et al. "DAWG: A defense against cache timing attacks in speculative execution processors." *MICRO-51*, 2018.

# 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!

*: Kiriansky, Vladimir, et al. "DAWG: A defense against cache timing attacks in speculative execution processors." *MICRO-51*, 2018.

# 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"**

Speculation starts

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

    // simple arithmetic
    spec_val = spec_val + 4;

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

}

......
```

← prediction

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

# Main Insight of STT

**"Sufficient for security: prevent secrets from reaching covert channels"**

Speculation starts

```
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? |
|---|---|---|
| Yes | No | No |
| | | |
| | | |

# Main Insight of STT

**"Sufficient for security: prevent secrets from reaching covert channels"**

Speculation starts

```
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? |
|---|---|---|
| Yes | No | No |
| No | Yes | No |
| | | |

# Main Insight of STT

**"Sufficient for security: prevent secrets from reaching covert channels"**

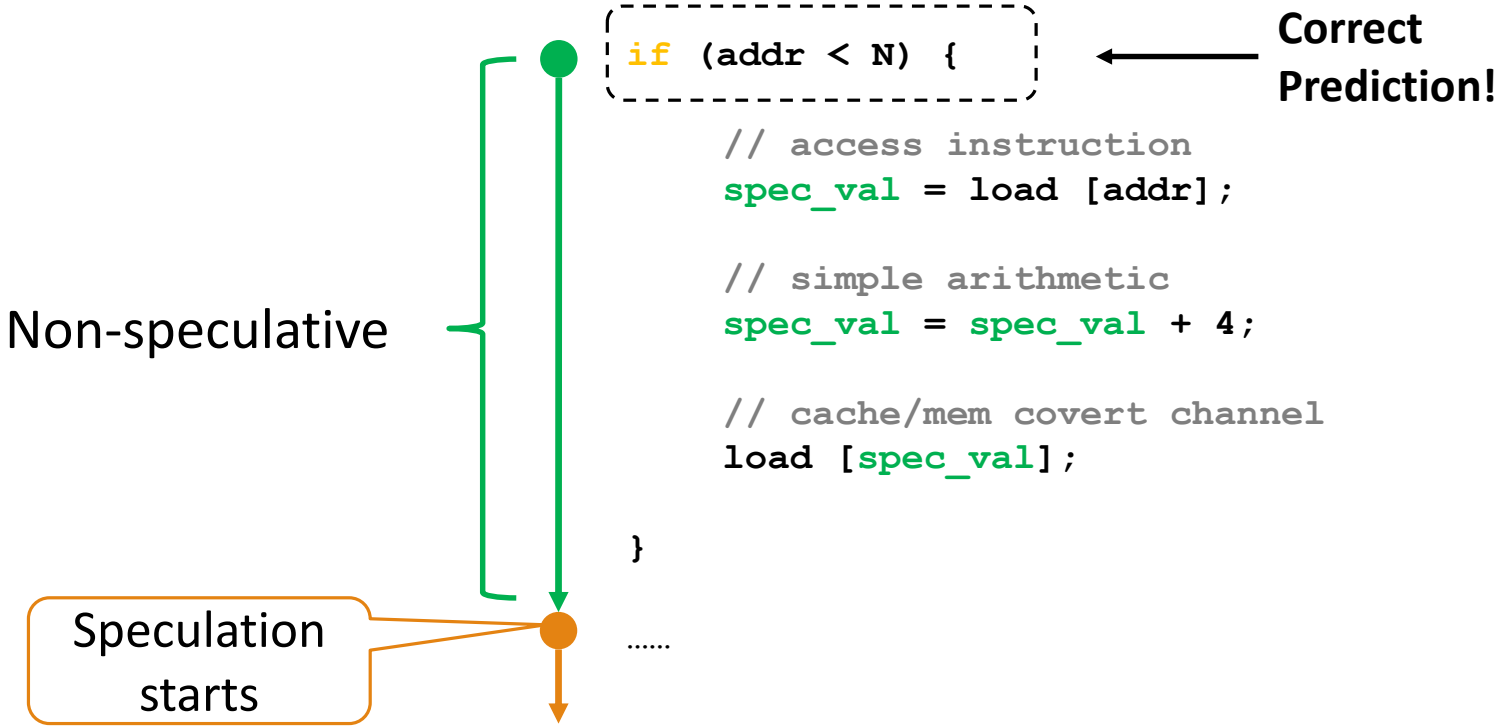
Speculation starts

```
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? |
|---|---|---|
| Yes | No | No |
| No | Yes | No |
| Yes | Yes | Yes |

# Main Insight of STT

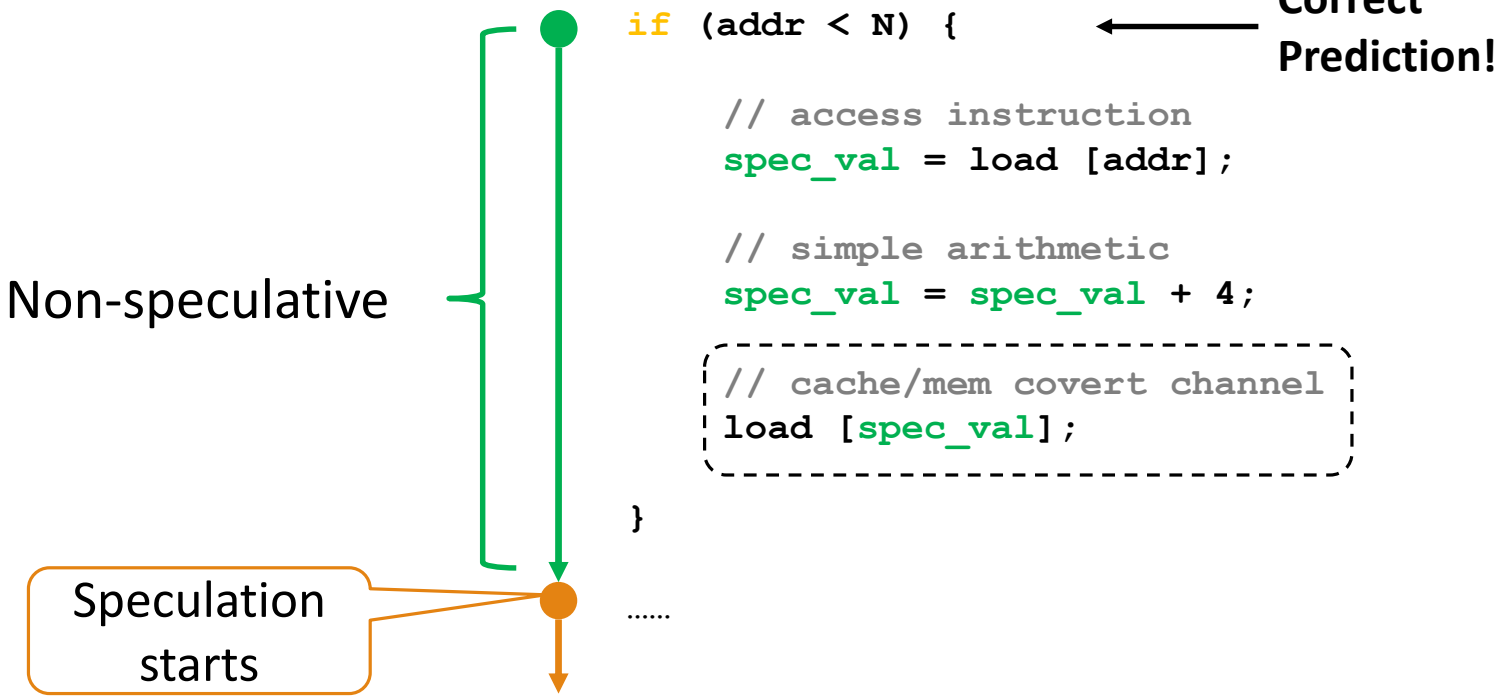
**"Sufficient for security: prevent secrets from reaching covert channels"**




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

# Main Insight of STT

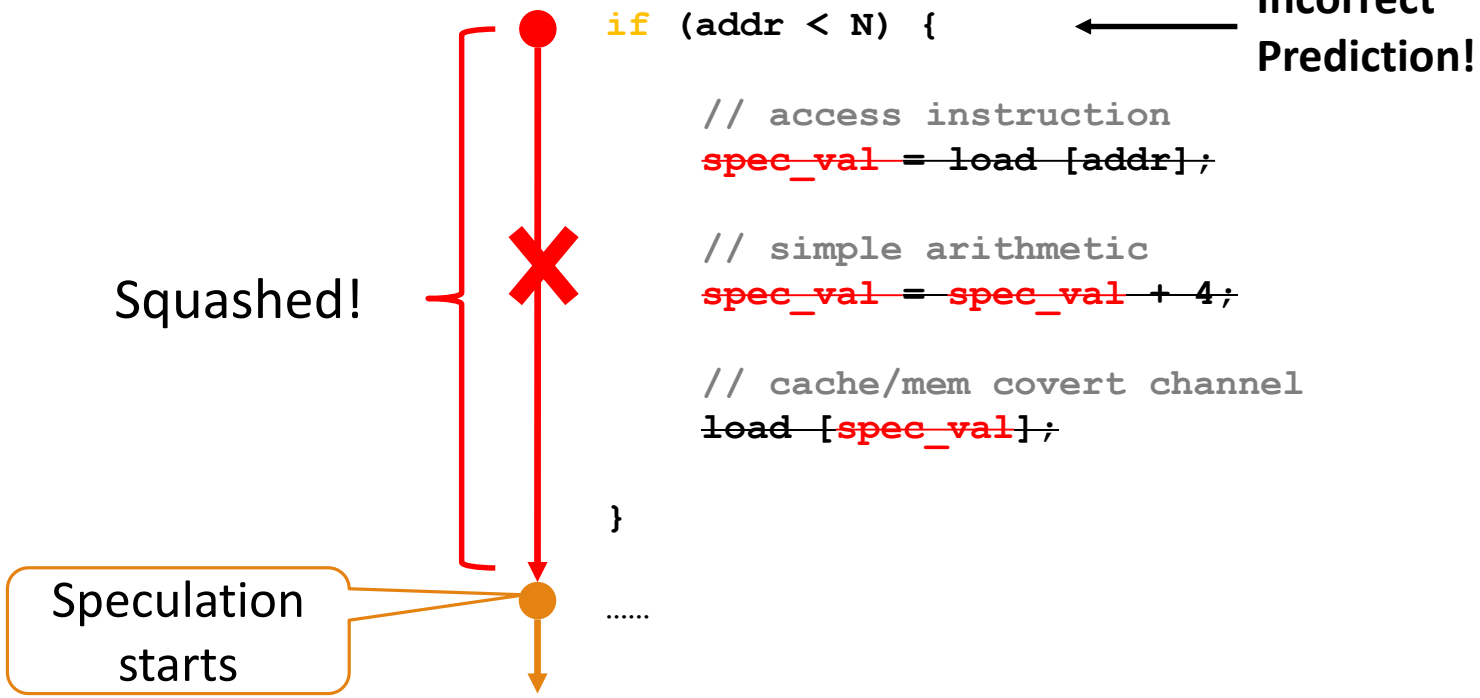
**"Sufficient for security: prevent secrets from reaching covert channels"**




| 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

# 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)

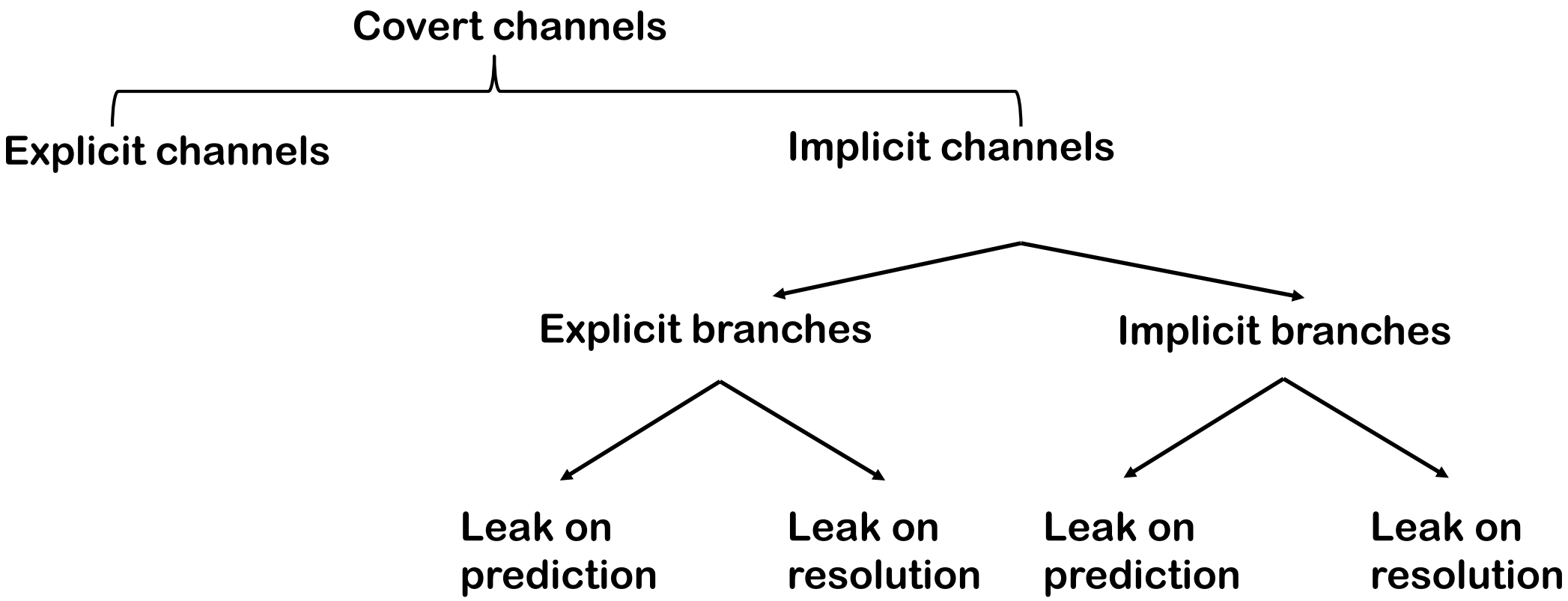
Covert channels

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

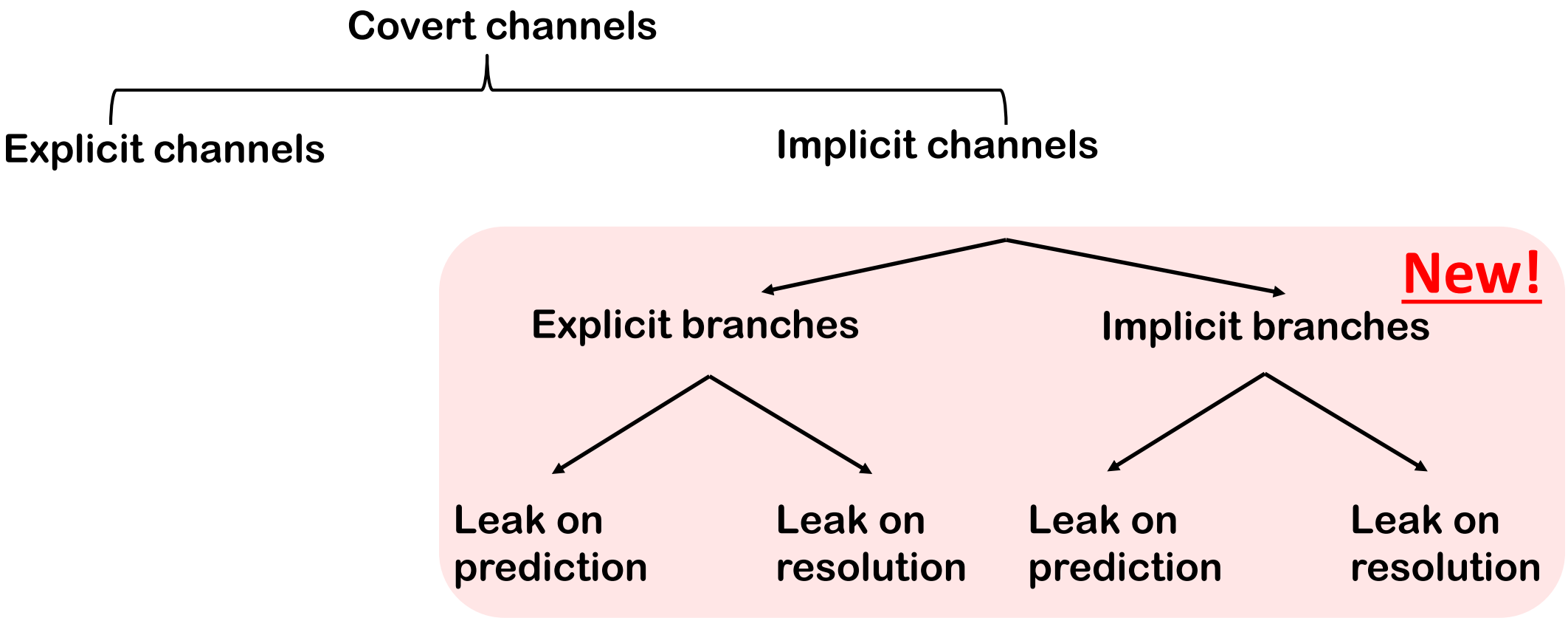
**How to identify all the secrets?** → **A new taint/untaint mechanism to track secrets in hardware**

# A Classification of Covert Channels in HW

# Classification of Covert Channels



# Classification of Covert Channels



# Classification of Covert Channels

**Covert channels**

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

```
load [secret];
```

# 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

# 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*

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**

# Explicit Branches @ Prediction

Cause:

The predictor state becomes a function of secret

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

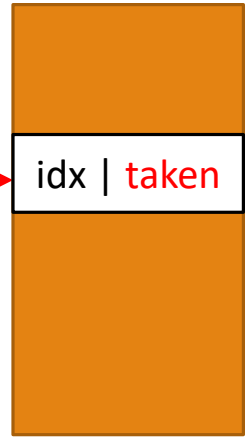
# Explicit Branches @ Prediction

Cause:

The predictor state becomes a function of secret

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

Resolve and update branch predictor



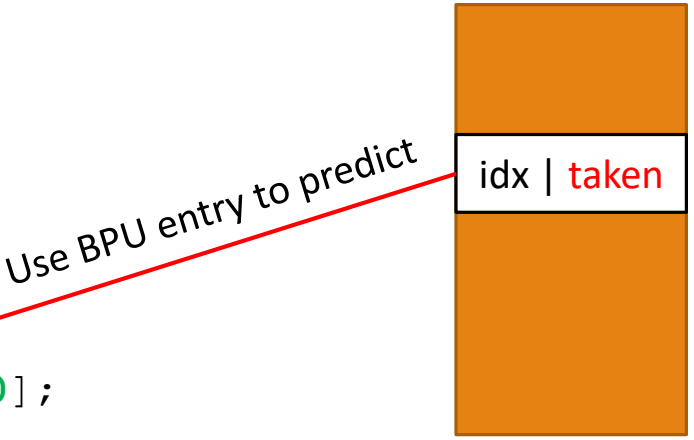

Branch Predictor Unit (BPU)

# Explicit Branches @ Prediction

Cause:

The predictor state becomes a function of secret

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




Branch Predictor Unit (BPU)

# 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**

# 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

**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**

**Implicit branches**
Example:
- Store-load pairs

# 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**

**Implicit branches**
Example:
- Store-load pairs

**Leak on prediction**

**Leak on resolution**

# Implicit Branches

```
store [secret] = foo;

bar = load [0x00];
```

Cause:

Non-control flow instructions create branch-like behaviors.

# 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) {

    // 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)

```
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

# Microarchitect Identifies …

Instructions forming **explicit channels**

- E.g. load, data-dependent arithmetic

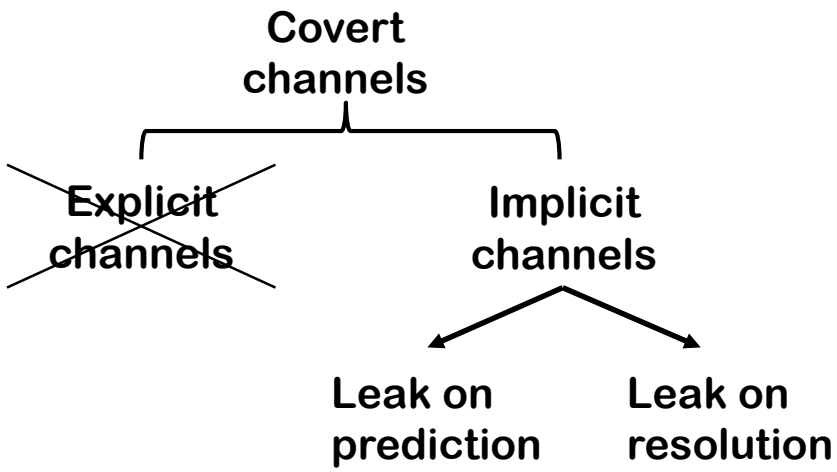
Instructions forming **implicit channels**

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

# Blocking Covert Channels

**Explicit channels:**

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



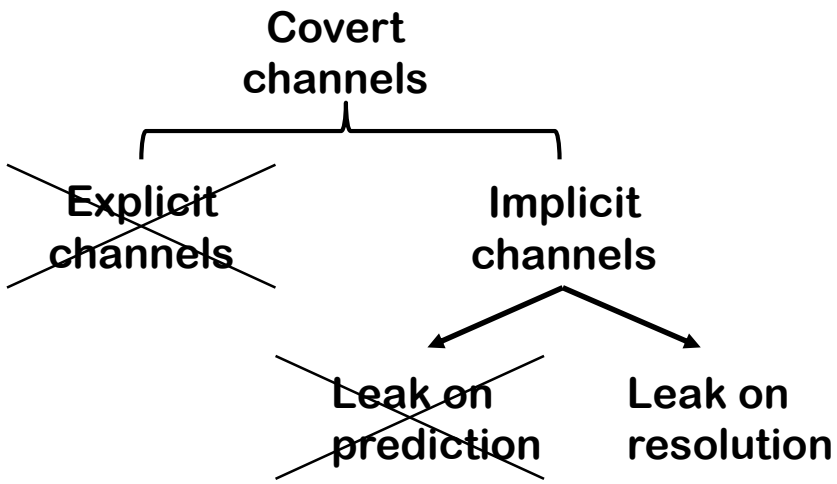
# Blocking Covert Channels

**Explicit channels:**

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

**Implicit channels:**

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



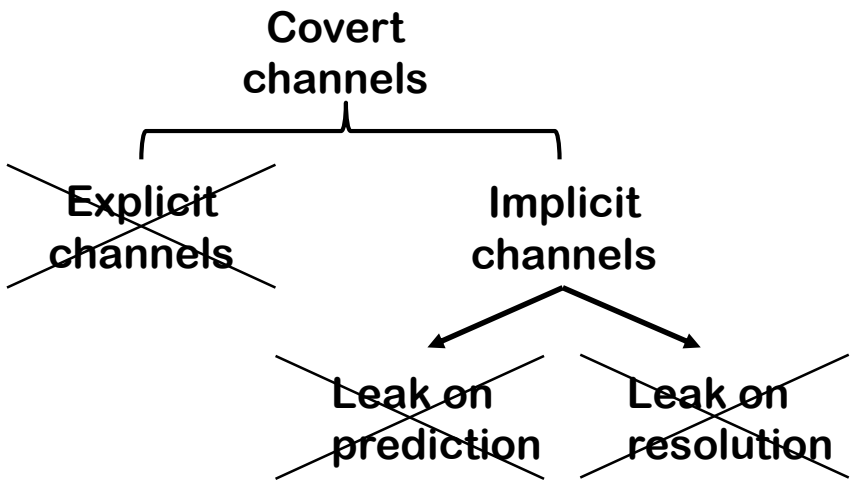
# Blocking Covert Channels

**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**



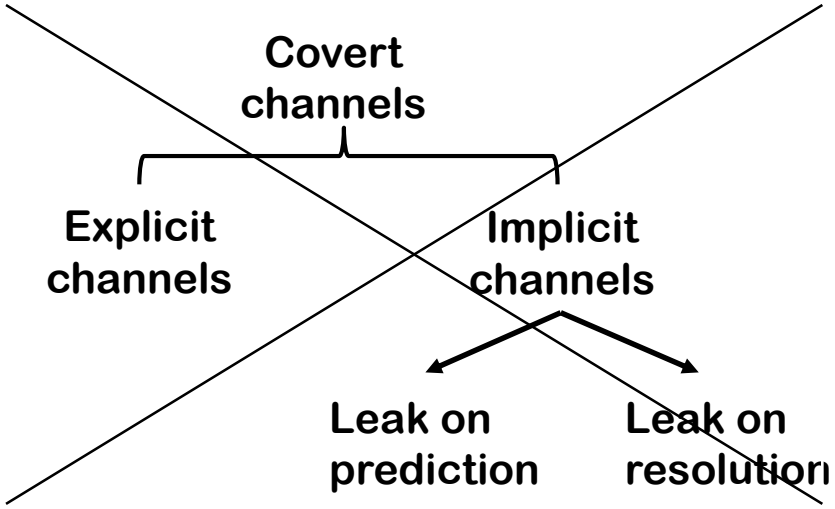
# Blocking Covert Channels

**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**



# Hardware Implementation of STT

# Efficient Implementation of Tainting/Untainting Logic

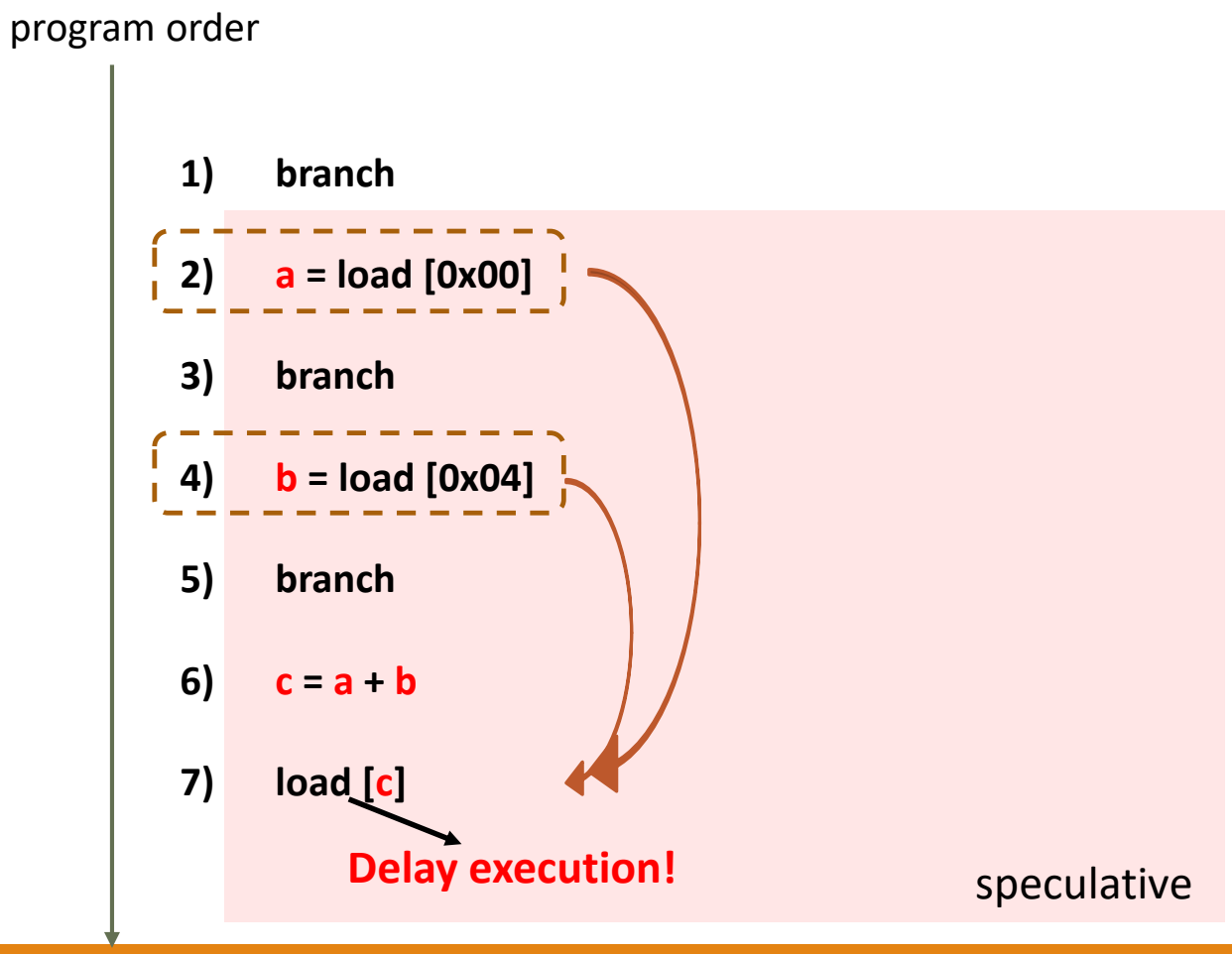
program order

1) branch
2) a = load [0x00]
3) branch
4) b = load [0x04]
5) branch
6) c = a + b
7) load [c]

Delay execution!

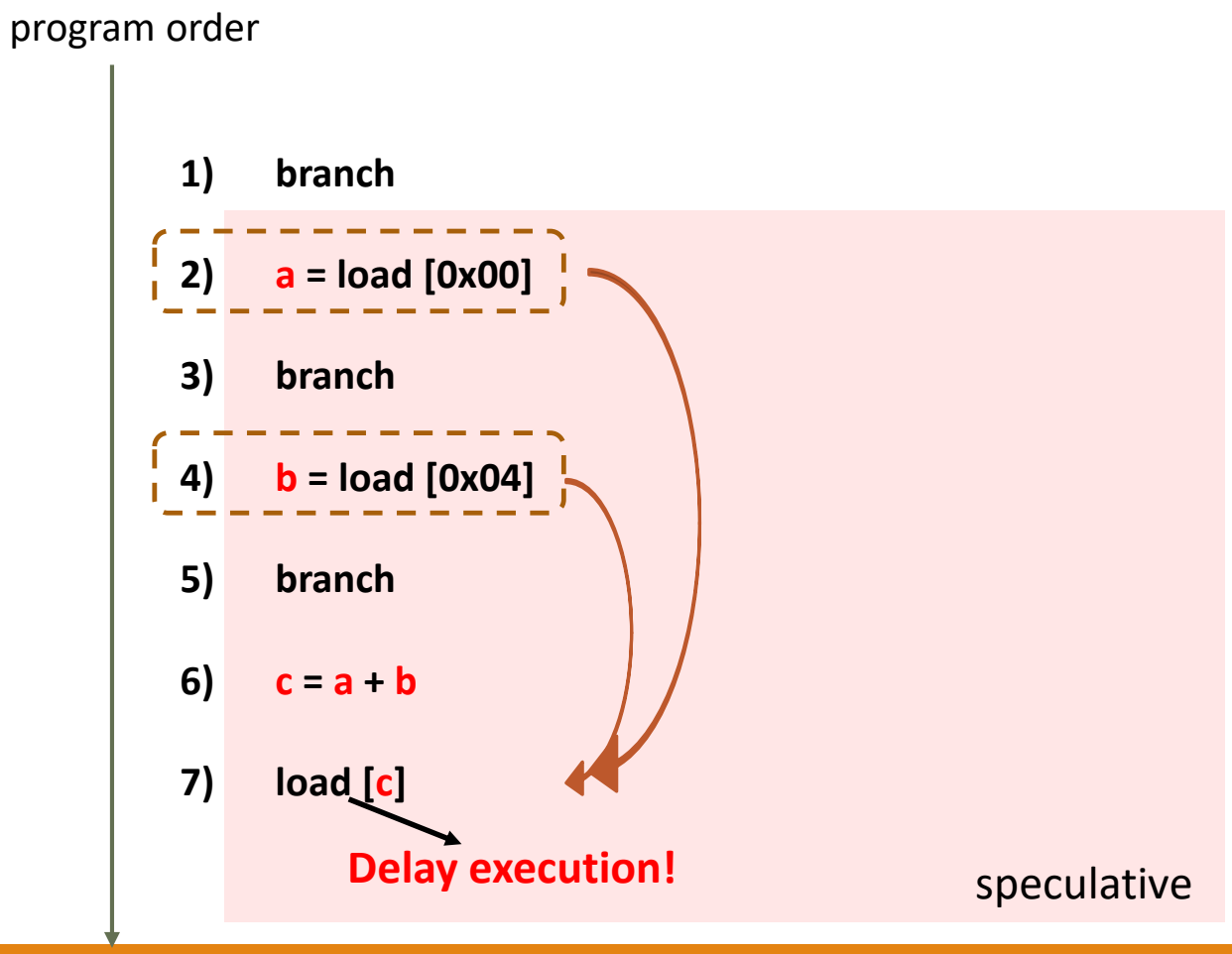
speculative

# Efficient Implementation of Tainting/Untainting Logic



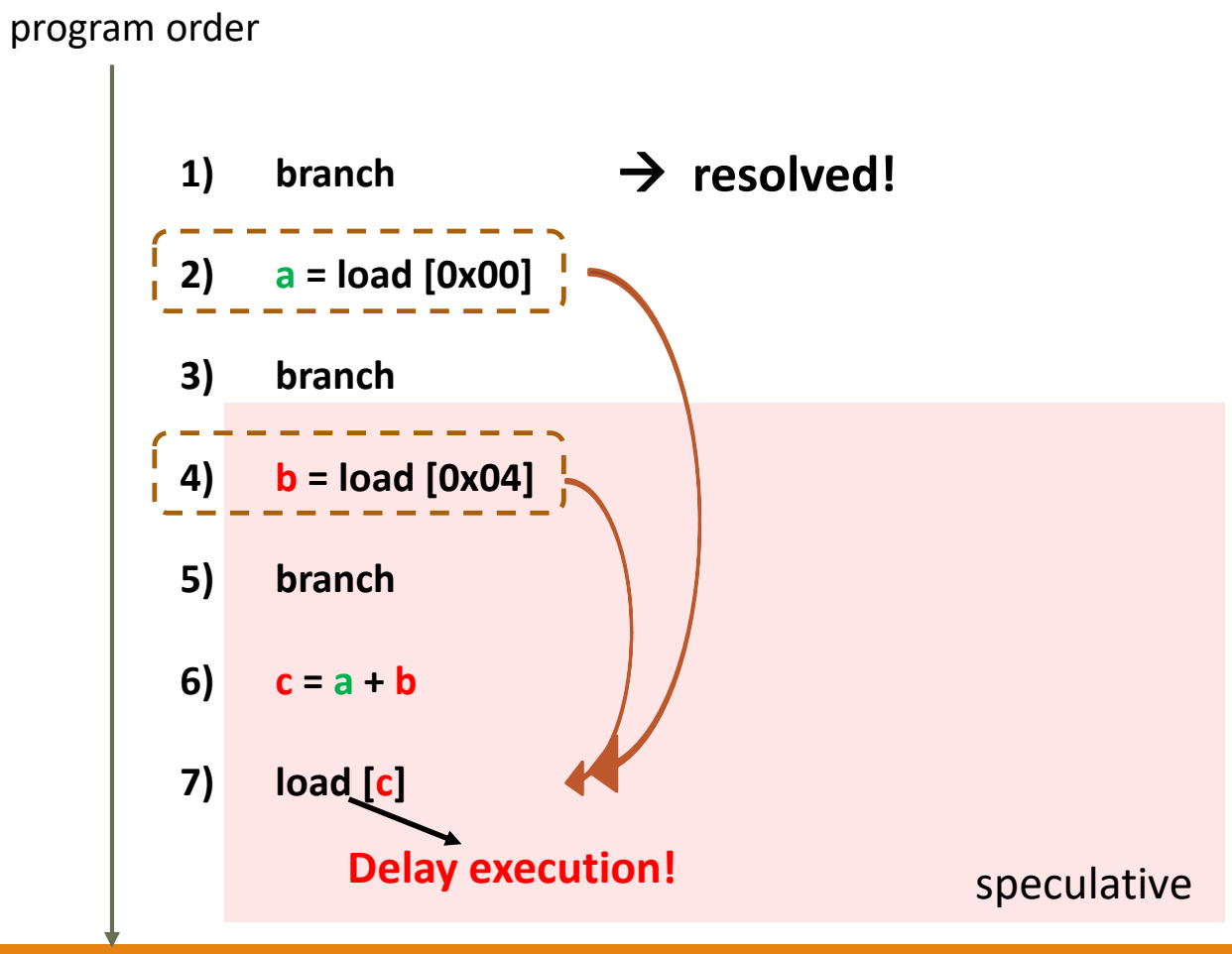
# Efficient Implementation of Tainting/Untainting Logic

Observation: All instructions turn non-speculative in-order



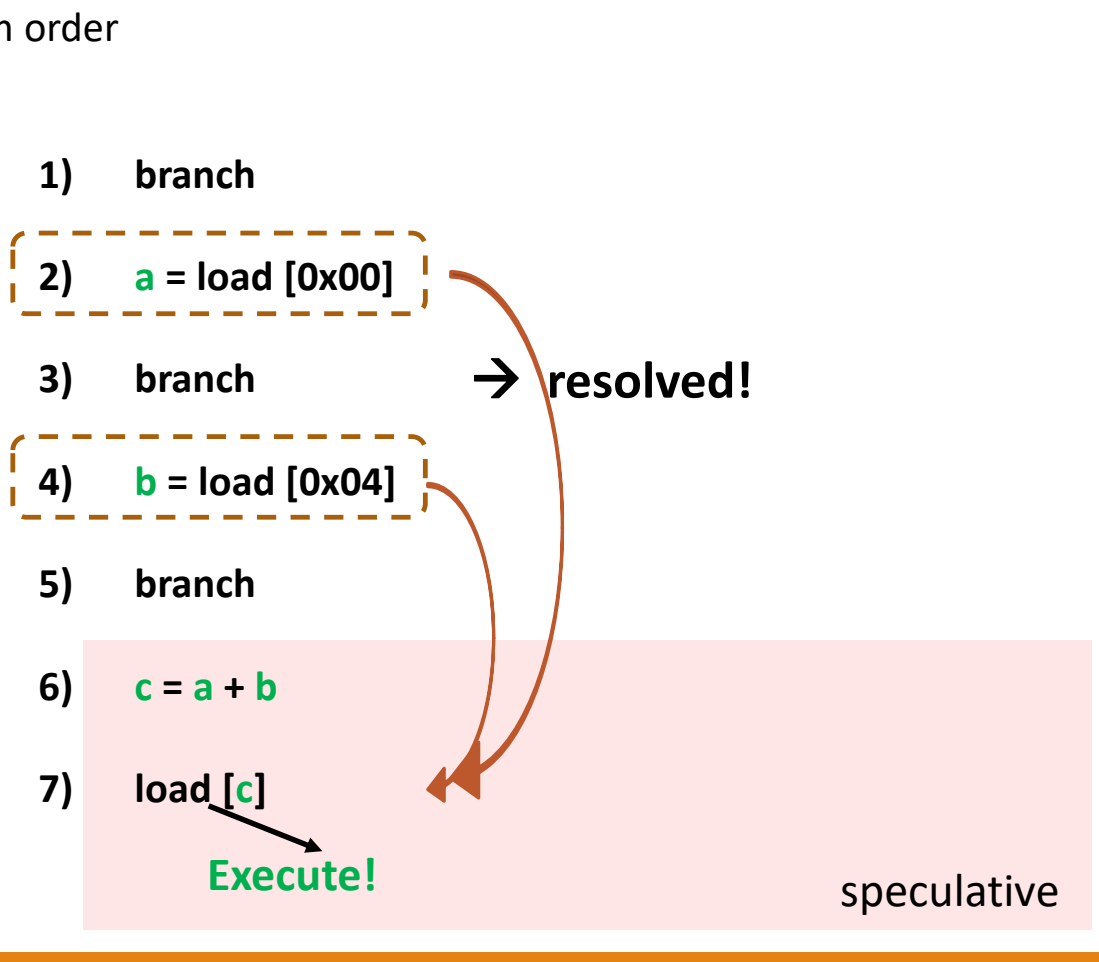
# Efficient Implementation of Tainting/Untainting Logic

Observation: All instructions turn non-speculative in-order



# Efficient Implementation of Tainting/Untainting Logic

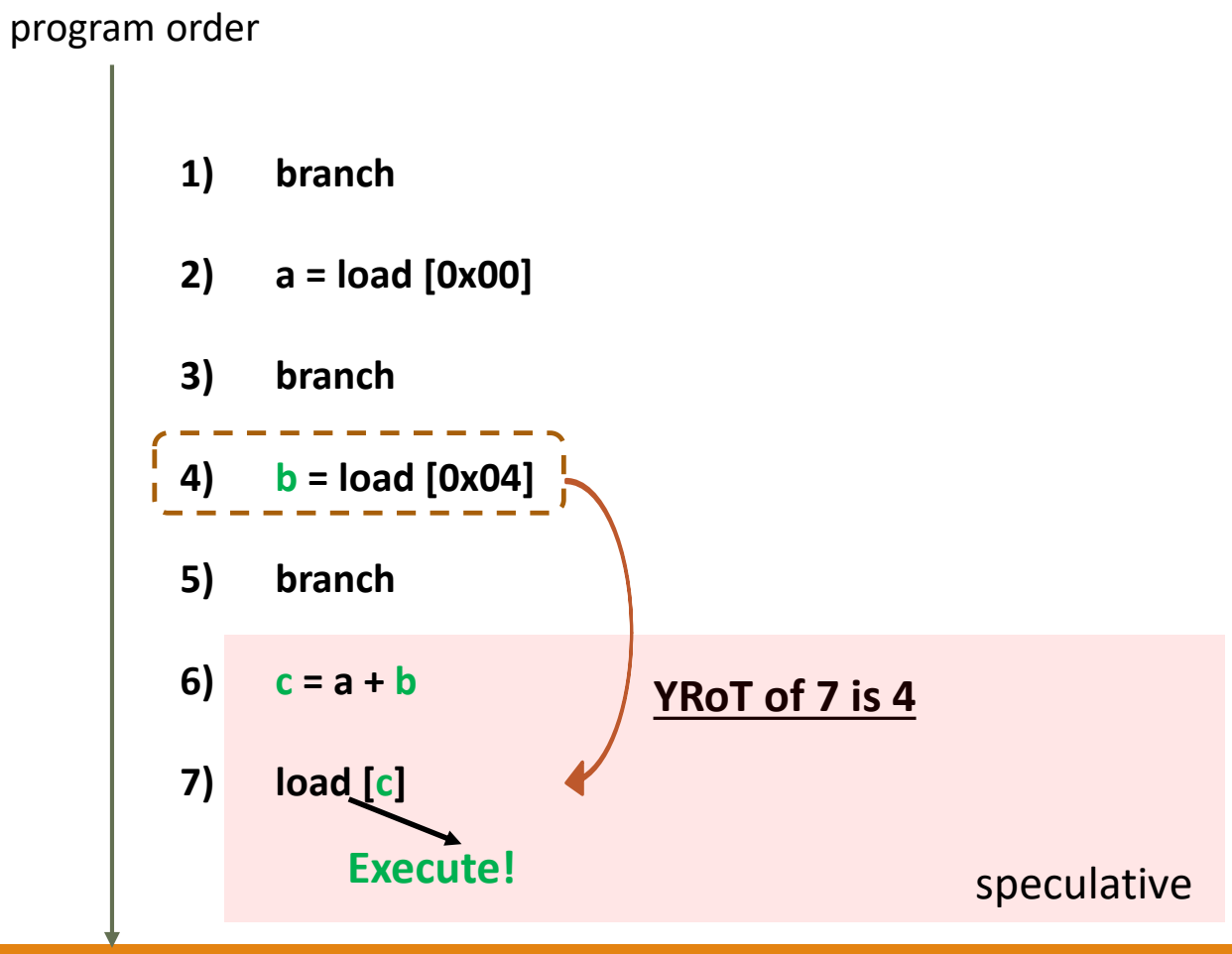
Observation: All instructions turn non-speculative in-order



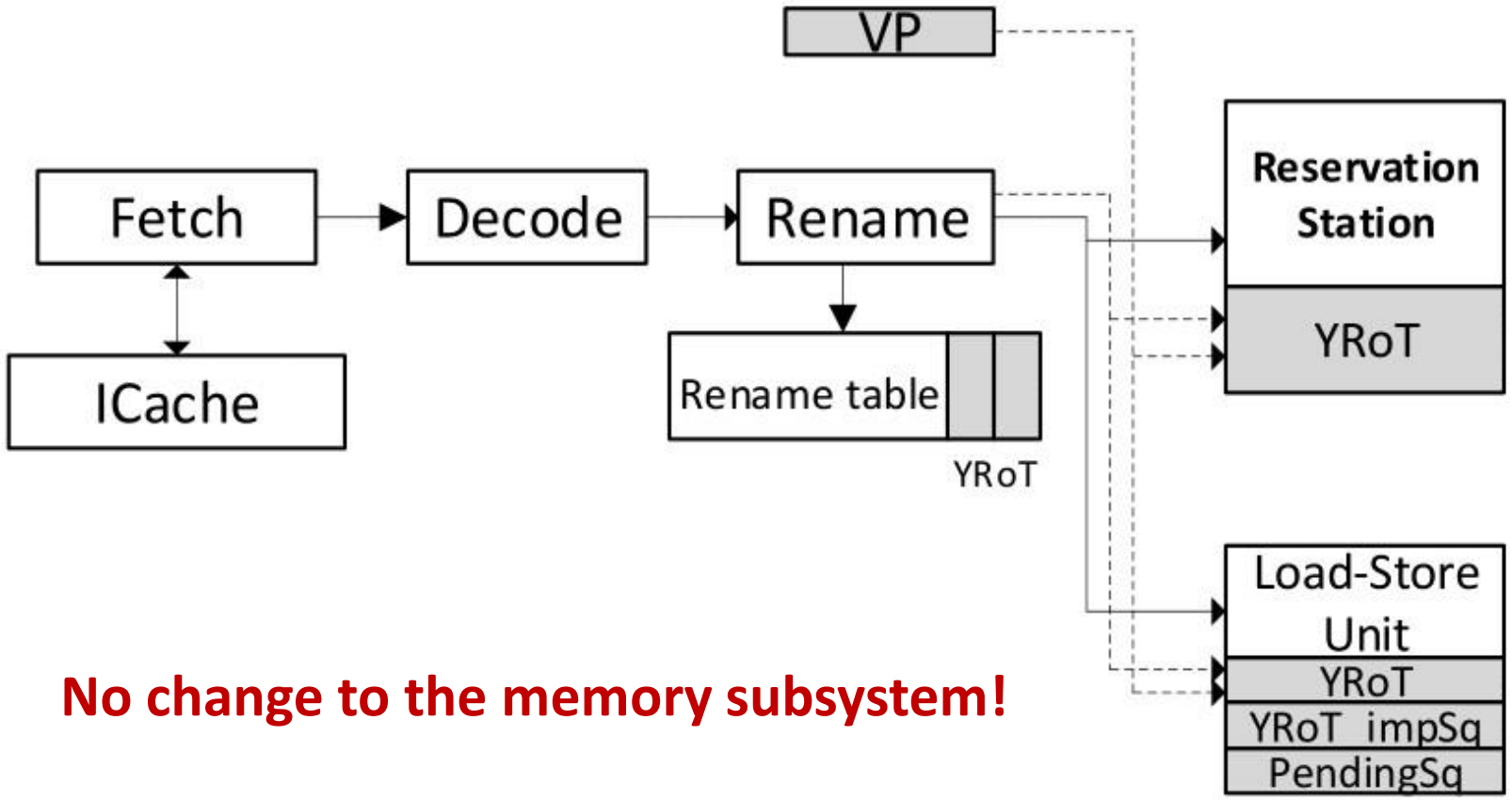
# Efficient Implementation of Tainting/Untainting Logic

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

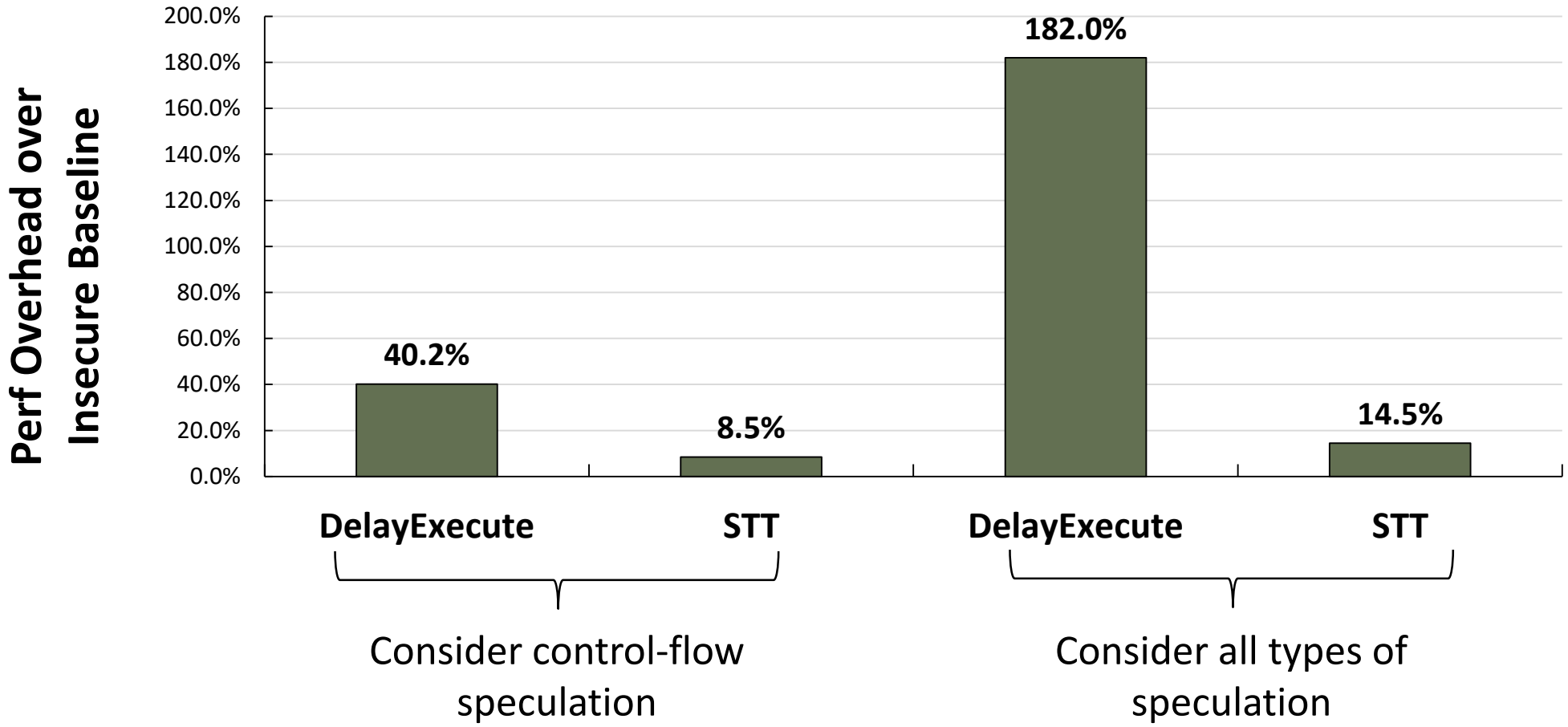
Security definition:

**Arbitrary speculative execution can only leak retired register file state (not arbitrary program memory)**

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

The link to the **detailed formal analysis and security proof** is in the paper

# Performance Evaluation on SPEC2006



# Conclusion

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

# Conclusion

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

**Questions?**

# Backup slides

# Threat Model

A powerful attacker who:

- Monitors covert channels
  - Cache, SIMD units, or any shared hardware resources
- From everywhere
  - Within same thread
  - Adjacent SMT context
  - Cross core
- At cycle granularity

# TODO: compare with NDA and SpecShield

# TODO: STT threat model vs. STT+ threat model

# Outline

Introduction

Threat Model

Speculative Taint Tracking

Evaluation

Conclusion

# Implementation of STT

A new classification of covert channels in HW

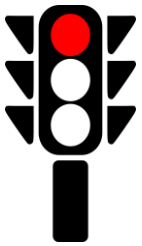
➔ Specify instructions with explicit or implicit channels

# Putting it together

Executes all instructions w/ untainted inputs

Executes non-transmit instructions w/ tainted inputs

Predicts explicit/implicit branches w/ tainted predicates

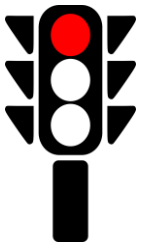


# Putting it together

Executes all instructions w/ untainted inputs

Executes non-transmit instructions w/ tainted inputs

Predicts explicit/implicit branches w/ tainted predicates



Delay executing transmit instructions w/ tainted inputs → Block explicit channel

Delay resolution/predictor updates of explicit/implicit branches w/ tainted predicates

# Putting it together

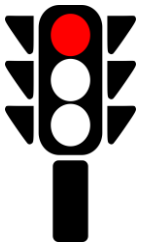
Executes all instructions w/ untainted inputs

TODO: spend a little bit more time on this slide?

Executes non-transmit instructions w/ tainted inputs

Predicts explicit/implicit branches w/ tainted predicates

TODO: what is transmit instruction?



Delay executing transmit instructions w/ tainted inputs → Block explicit channel

Delay resolution/predictor updates of explicit/implicit branches w/ tainted predicates → Block implicit channel

# Microarchitecture Design of STT

- Each instruction has its "*Youngest Root of Taint*" (YRoT)

For each (transmit) instruction:

Input is secret

⇔ YRoT is still speculative

⇔ Visibility point is ahead of YRoT

program order

1) branch
2) a = load [0x00] // YRoT = -1
3) branch_1
4) b = load [0x04] // YRoT = -1
5) branch
6) **c** = a + b // YRoT = max(2, 4) = <u>4 > 1</u>
7) d = load [**c**] // YRoT = <u>4 > 1</u>

# Microarchitecture Design of STT

- Each instruction has its "*Youngest Root of Taint*" (YRoT)

For each (transmit) instruction:

Input is secret

⇔ YRoT is still speculative

⇔ Visibility point is ahead of YRoT

program order

1) branch_0

2) a = load [0x00] // YRoT = -1

3) branch_1

4) b = load [0x04] // YRoT = -1

5) branch_2

6) **c** = a + b // YRoT = max(2, 4) = <u>4 > 3</u>

7) d = load [**c**] // YRoT = <u>4 > 3</u>

# Microarchitecture Design of STT

- Each instruction has its "*Youngest Root of Taint*" (YRoT)

For each (transmit) instruction:

Input is secret

⇔ YRoT is still speculative

⇔ Visibility point is ahead of YRoT

program order

```
1)  branch_0
2)  a = load [0x00]     // YRoT = -1
3)  branch_1
4)  b = load [0x04]     // YRoT = -1
5)  branch_2                          ← VP
6)  c = a + b           // YRoT = max(2, 4) = 4 < 5
7)  d = load [c]        // YRoT = 4 < 5
```

# 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)***

For each instruction:

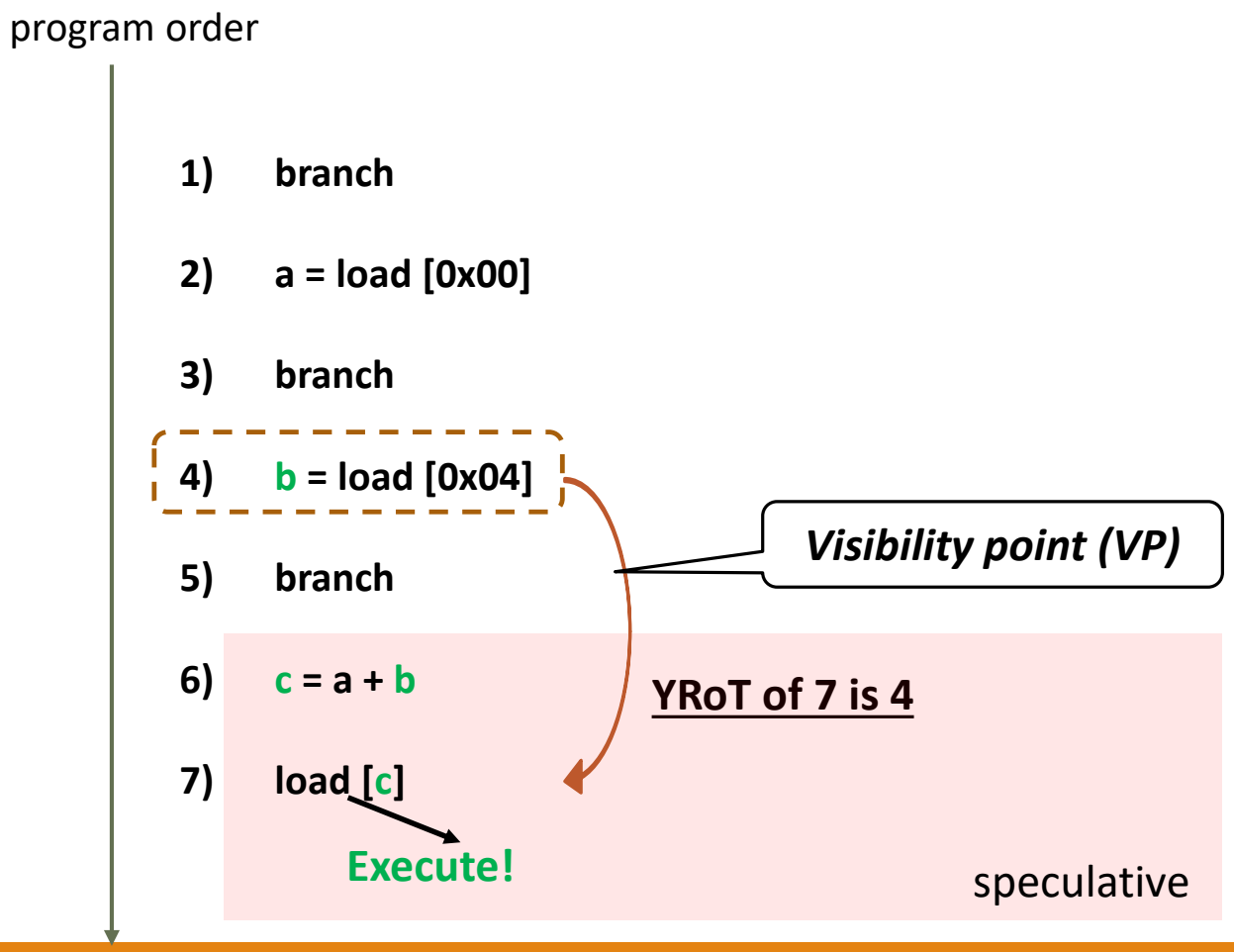
Input is tainted

TODO: remove this

⇔ Input depends on some speculative access instruction

⇔ YRoT is still speculative

⇔ Visibility point is ahead of YRoT



# Security Evaluation

Arbitrary speculative execution can only leak retired register file state (not arbitrary program memory)

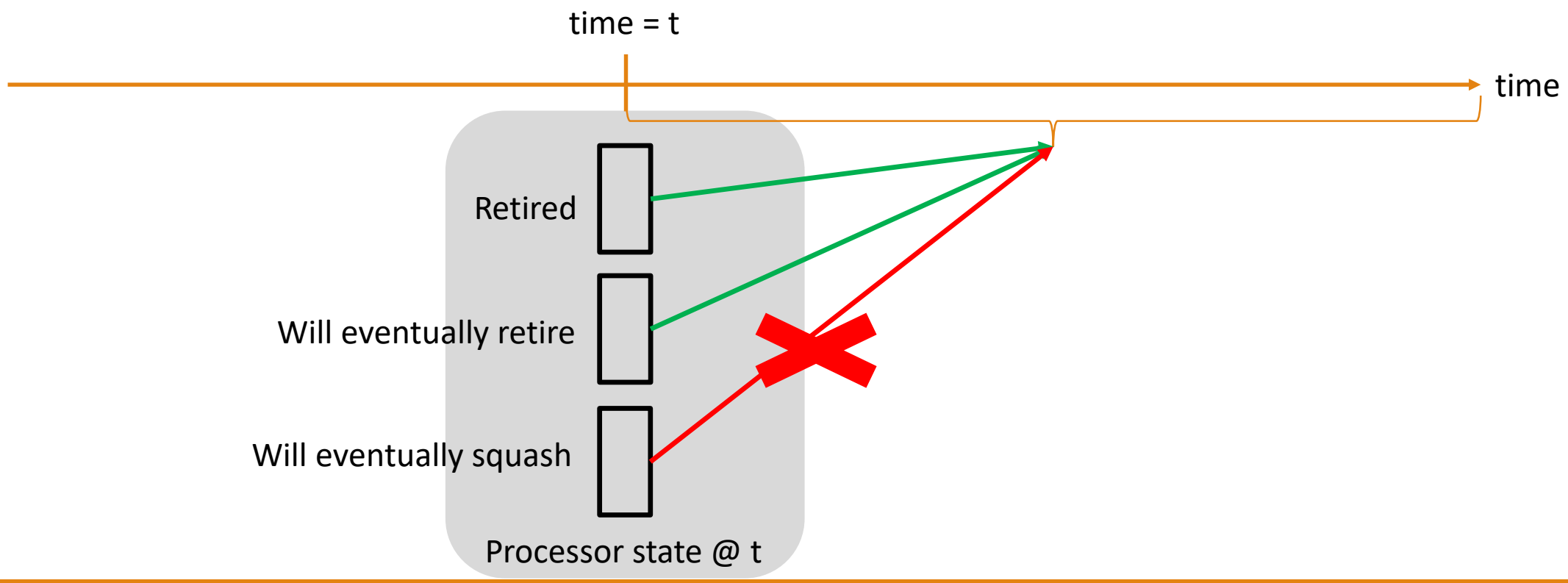
# Security Evaluation

Arbitrary speculative execution can only leak retired register file state (not arbitrary program memory)


**The Universal Read Gadget** == many Spectre variants (1, 3, 4, ..), MDS attacks, Meltdown, etc.

# Security Evaluation

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



# Security Evaluation

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