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)

\[ r = P(s) \]

OS: protects computation \( P(s) \)
Crypto: protects data in transit \((s, r)\)
\[ \rightarrow \text{No unauthorized party learns } s \]
// 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
Microarchitectural Side/Covert Channels Everywhere

Runtime

Processor

Core

Cache

Core

Cache access pattern

Cache banking [YGH’16]

L1 I Cache

L1 D Cache

L2 Cache

Inclusive LLC [LYGHL’15]

Non-inclusive LLC [YSGFCT’19]

L3 Cache

RAND unit [EP’16]

DRAM [PGMSM’16]

DRAM (and/or: stacked DRAM, HMC, NVMs)

Arithmetic timing [AKMJLS’15]

Branch predictors [ERAP’18]

Port contention [CBHGT’18]

4K aliasing [MES’17]

Inclusive LLC [LYGHL’15]

Non-inclusive LLC [YSGFCT’19]

DRAM [PGMSM’16]
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.,

\[ r = P(s) \]

Black box

i.e., secure given any uarch side/covert channels
Principled, Low-overhead

Obviously.

But not at the expense of clean security.
A lattice model of secure information flow; Dorothy E. Denning, CACM 1976
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*

// Spectre Variant 1

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

    // covert channel
    load [spec_val];
}

Speculative Execution Attacks*

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

Speculative Execution Attacks*

```c
// 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

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!

Speculative Execution Attacks*

// 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];
}

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”

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

    // simple arithmetic
    spec_val = spec_val + 4;

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

<table>
<thead>
<tr>
<th>Creates a covert channel?</th>
<th>Input operand is a secret?</th>
<th>Requires protection?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

......
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];
}
```

<table>
<thead>
<tr>
<th>Creates a covert channel?</th>
<th>Input operand is a secret?</th>
<th>Requires protection?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Speculation starts
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];
}
```

<table>
<thead>
<tr>
<th>Creates a covert channel?</th>
<th>Input operand is a secret?</th>
<th>Requires protection?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Speculation starts
Main Insight of STT

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

```c
if (addr < N) {
    // access instruction
    spec_val = load [addr];
    // simple arithmetic
    spec_val = spec_val + 4;
    // cache/mem covert channel
    load [spec_val];
}
```

<table>
<thead>
<tr>
<th>Creates a covert channel?</th>
<th>Input operand is a secret?</th>
<th>Requires protection?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

……
Main Insight of STT

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

```c
if (addr < N) {
    // access instruction
    spec_val = load [addr];
    // simple arithmetic
    spec_val = spec_val + 4;
    // cache/mem covert channel
    load [spec_val];
}
```

<table>
<thead>
<tr>
<th>Creates a covert channel?</th>
<th>Input operand is a secret?</th>
<th>Requires protection?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
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];
}

Correct Prediction!

<table>
<thead>
<tr>
<th>Creates a covert channel?</th>
<th>Input operand is a secret?</th>
<th>Requires protection?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
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];
}

Correct Prediction!

<table>
<thead>
<tr>
<th>Creates a covert channel?</th>
<th>Input operand is a secret?</th>
<th>Requires protection?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
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];
}
```

Incorrect Prediction!

<table>
<thead>
<tr>
<th>Creates a covert channel?</th>
<th>Input operand is a secret?</th>
<th>Requires protection?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Speculation starts

Squashed!
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.
Speculative Taint Tracking

Secret
(speculatively accessed data) \(\rightarrow\) 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

What are the covert channels?

How to identify all the secrets?

Secret (speculatively accessed data)

Covert channels

A new classification to understand covert channels in speculative machines

A new taint/untaint mechanism to track secrets in hardware
A Classification of Covert Channels in HW
Classification of Covert Channels

Covert channels

Explicit channels

Explicit branches

Leak on prediction

Implicit branches

Implicit branches

Leak on resolution

Leak on
prediction

Leak on resolution
Classification of Covert Channels

Explicit channels

Explicit branches
- Leak on prediction

Implicit branches
- Leak on resolution

Implicit channels
- Leak on prediction
- Leak on resolution

New!
Classification of 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

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

```plaintext
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

```plaintext
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

Explicit branches
Examples:
- Branch/jump instructions

New!
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.

- 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];
```
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

```c
... ...
... ...
if ( secret )
... ...
... ...
if ( public )
  load [0x00];
else
  load [0x10];
```
Explicit Branches @ Prediction

Cause:
The predictor state becomes a function of secret

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

Use BPU entry to predict

Branch Predictor Unit (BPU)
Classification of Covert Channels

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

Examples:
- memory loads
- data-dependent arithmetic

Explicit branches
Examples:
- Branch/jump instructions

Leak on prediction
Leak on resolution

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

New!
Explicit Branches @ Resolution

```java
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.

```c
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

```c
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

Implicit branches
Example:
- Store-load pairs

Leak on resolution
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:
- Explicit branches: Branch/jump instructions
- Implicit branches: Store-load pairs

Leak on prediction
Leak on resolution

New!
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

```java
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)

```c
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)

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

    // simple arithmetic
    b = a + 4;

    // cache/mem covert channel
    load [b];
}
```
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)

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

    // simple arithmetic
    b = a + 4;

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

```
……
……
……
```

Resolved!
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)

```c
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)

**STT untaints when:**
1) A speculative access instruction becomes non-speculative (a)
2) An instruction has all its input untainted (b)

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

    // simple arithmetic
    b = a + 4;

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

……
……
……
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)

Data is tainted $\rightarrow$ Data is speculative
(not necessarily other way around)

if (addr < N) {
    // access instruction
    a = load [addr];
    // simple arithmetic
    b = a + 4;
    // cache/mem covert channel
    load [b];

    .....  \[speculative\]
}

Resolved!
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 un tainted (e.g., load address)

Implicit channels:
- Delay predictor update until branch predicate un tainted
- Delay resolution until branch predicate un tainted
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.

```plaintext
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

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

(program order)

Delay execution!

(speculative)
Efficient Implementation of Tainting/Untainting Logic

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

Delay execution!

Speculative
Efficient Implementation of Tainting/Uncainting Logic

Observation: All instructions turn non-speculative in-order

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

Delay execution!
Efficient Implementation of Tainting/Untainting Logic

Observation: All instructions turn non-speculative in-order

- 1) branch
- 2) $a = \text{load}[0x00]$ → resolved!
- 3) branch
- 4) $b = \text{load}[0x04]$ → speculative
- 5) branch
- 6) $c = a + b$
- 7) load[$c$]

Delay execution!
Efficient Implementation of Tainting/Untainting Logic

Observation: All instructions turn non-speculative in-order

1) branch
2) $a = \text{load } [0x00]$→speculative
3) branch
4) $b = \text{load } [0x04]$→resolved!
5) branch
6) $c = a + b$
7) $\text{load } [c]$→ Execute!

Program 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)

Program order:

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

YRoT of 7 is 4
speculative
Execute!
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:

Processor state @ t

- Retired
- Will eventually retire
- Will eventually squash

STT paper has link to security analysis eprint
Performance Evaluation on SPEC2006

Consider control-flow speculation

Consider all types of speculation

<table>
<thead>
<tr>
<th></th>
<th>Perf Overhead over Insecure Baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td>DelayExecute</td>
<td>40.2%</td>
</tr>
<tr>
<td>STT</td>
<td>8.5%</td>
</tr>
<tr>
<td>DelayExecute</td>
<td>182.0%</td>
</tr>
<tr>
<td>STT</td>
<td>14.5%</td>
</tr>
</tbody>
</table>
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)

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

>90% of overhead
Speculative Data Oblivious (SDO): Executive Summary
Speculative Data Oblivious (SDO): Executive Summary

Idea 1. Execute transmit secret

High performance
Speculative Data Oblivious (SDO): Executive Summary

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

- High performance
- High security, **low performance**
Speculative Data Oblivious (SDO): Executive Summary

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

- High performance

Idea 2. Predict how the execution should be performed

- High security, low performance
Speculative Data Oblivious (SDO): Executive Summary

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

High performance

Idea 2. Predict how the execution should be performed

High security, low performance

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

\[
a = \text{fpop } a, b
\]

- \((a \text{ is normal}) \&\& (b \text{ is normal})\):
  - Latency = \(X\)
  - Fast path (FPU only)

- \((a \text{ is subnormal}) \text{ || } (b \text{ is subnormal})\):
  - Latency = \(Y > X\)
  - Slow path (with microcode assist)
Idea 1: Being Data Oblivious

\[ a = \text{fpmult} \ a, \ b \]

- **Fast path (FPU only)**
  - Latency = \( X \)
  - Result (fast)

- **Slow path (with microcode assist)**
  - Latency = \( Y > X \)
  - Result (slow)

Select the correct answer if \( Y > X \).
Idea 1: Being Data Oblivious

Using fast and slow path

Latency = X
Fast path (FPU only) → result (fast)
Latency = Y > X
Slow path (with microcode assist) → result (slow)

Select the correct answer

Timeline:

0 X Y

Using fast and slow path

\[ a = \text{fpmult} \ a, \ b \]

\[ c = \text{fpmult} \ c, \ d \]
Idea 1: Being Data Oblivious

Using fast and slow path

Fast path (FPU only)
Latency = X
result (fast)

Slow path (with microcode assist)
Latency = Y > X
result (slow)

Select the correct answer

Paying performance for security

a = fpmult a, b
b = c, d

0 X Y
Idea 2: “Predicting” Execution to Perform

a = fpmult a, b

Fast path (FPU only)

Latency = X

"Predict"

Slow path (with microcode assist)

Latency = Y > X

c = fpmult c, d

Predicting fast path
Idea 2: “Predicting” Execution to Perform

- Fast path (FPU only)
  - Latency = \( X \)
  - \( a = \text{fpmult} \ a, \ b \)
- Slow path (with microcode assist)
  - Latency = \( Y > X \)
- Dependent instructions
  - May be invalid

Prediction timeline:

- Predicting fast path
- \( a = \text{fpmult} \ a, \ b \) at time 0
- Result (fast) at time \( X \)
- \( c = \text{fpmult} \ c, \ d \)
Idea 2: “Predicting” Execution to Perform

Fast path (FPU only)
Latency = X

Slow path (with microcode assist)
Latency = Y > X

result (fast)

Dependent instructions

Predictor

a = fpmult a, b

"predict"

"Resolve"

a = fpmult a, b

Predicting fast path

0
X

P
P+Y

Resolving to slow path

timeline
Idea 2: “Predicting” Execution to Perform

\[ a = \text{fpmult} \ a, \ b \]

**Fast path (FPU only)**
- Latency = \( X \)
- Result (fast)

**Slow path (with microcode assist)**
- Latency = \( Y > X \)

Dependent instructions

Potential new leakage
Applying STT for Security

Speculative Taint Tracking
Applying STT for Security

Predictor

```
a = fpmult a, b
```

"Predict"

Latency = X

Fast path (FPU only)

"Resolve"

Latency = Y > X

Slow path (with microcode assist)

result (fast) → Dependent instructions

Prevent leakage via Prediction/Resolution

Speculative Taint Tracking
Applying STT for Security

\[ a = \text{fpmult} \ a, \ b \]

**Fast path (FPU only)**

Latency = \( X \)

result (fast) \( \rightarrow \) Dependent instructions

**Slow path (with microcode assist)**

Latency = \( Y > X \)

**“Predict”**

**“Resolve”**

**Speculative Taint Tracking**

Prevent leakage via Prediction/Resolution

“Taint” and hide sensitive results
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!**

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

We build predictors to reduce defense overhead
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

Transmitters:
- Load
- Floating-point multiplication
- Floating-point division

Static L1: always predicting DO–1d_{L1}
Static L2: always predicting DO–1d_{L2}
Static L3: always predicting DO–1d_{L3}
Hybrid: using the hybrid predictor
Perfect: prediction is accurate and precise

“Spectre” attack model
Consider control-flow speculation

“Futuristic” attack model
Consider all types of speculation

<table>
<thead>
<tr>
<th></th>
<th>Perf Overhead over Insecure Baseline</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0.0%</td>
</tr>
<tr>
<td>STT</td>
<td>8.4%</td>
</tr>
<tr>
<td>Static L1</td>
<td>6.8%</td>
</tr>
<tr>
<td>Static L2</td>
<td>5.5%</td>
</tr>
<tr>
<td>Static L3</td>
<td>5.3%</td>
</tr>
<tr>
<td>Hybrid</td>
<td>4.2%</td>
</tr>
<tr>
<td>Perfect</td>
<td>3.1%</td>
</tr>
</tbody>
</table>

22.4%
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

Traditional (non-transient) side channels
Post 2018

Traditional (non-transient) side channels

“Spectre”

“Meltdown”

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

Traditional (non-transient) side channels

“Spectre”

“Meltdown”

How does other microarchitecture leak privacy?

Safecracker (Compressed caches) [ASPLOS’20]

RAMBleed (RowHammer)
Compressed Cache Attacks

1. Attacker sends encryption request to victim
2. Victim stores input next to key
3. Attacker measures line’s compressed size, infers $0x01$ is in the secret data
Read Gadgets from Compressed Cache

Co-locate attacker data with secret data $\rightarrow$ leak secret data

Numerous ways to co-locate data.

HEARTBLEED-LIKE

$$p = \text{malloc}(SZ);$$
$$\text{memcpy}(p, \text{usr\_data}, SZ);$$

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.