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

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

⚠️ CAUTION
SIDE-CHANNEL ATTACKS
Processors are Insecure

Introduction
Speculative Taint Tracking
Evaluation
Conclusion
Speculative Execution Attacks

// Spectre Variant 1

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

    // covert channel
    load [spec_val];
}

Speculation starts
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];
}

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];
}
```

Speculation starts

Speculative access instruction* accesses secret

Creates a covert channel to leak secret

Speculation ends - misspeculation!

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>
</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];
}
```

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>
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>

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>
<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”
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>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];
}
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) ➔ X ➔ Covert channels

What are the covert channels?
Speculative Taint Tracking

Secret (speculatively accessed data) \rightarrow \text{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

- Explicit channels
  - Explicit branches
    - Leak on prediction
  - Implicit branches
    - Leak on resolution
- Implicit channels
  - Leak on prediction
  - Leak on resolution
Classification of Covert Channels

Covert channels

Explicit channels

Explicit branches

Leak on prediction

Implicit channels

Implicit branches

Leak on resolution

Leak on prediction

Leak on resolution

New!
Classification of Covert Channels

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

```c
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

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

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

Leak on prediction
Leak on resolution

New!
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];

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

```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

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

Implicit branches
Example:
- Store-load pairs

New!

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

New!

Leak on prediction
Leak on resolution
Leak on prediction
Leak on resolution
Implicit Branches

Cause:
Non-control flow instructions create branch-like behaviors.

\[
\text{store } [\text{secret}] = \text{foo}; \\
\text{bar} = \text{load } [0x00];
\]
Implicit Branches

Cause:
Non-control flow instructions create branch-like behaviors.

```c
store [secret] = foo;

bar = load [0x00];
```

Can be thought as:

```c
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

```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)

```
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];
}
```

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)
2) An instruction has all its input untainted (b)

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

    // simple arithmetic
    b = a + 4;

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

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 \textit{untainted} (e.g., load address)

Implicit channels:
◦ Delay predictor update until branch predicate \textit{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

Introduction
Speculative Taint Tracking
Evaluation
Conclusion

Program order:

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/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

Observation: All instructions turn non-speculative in-order
Efficient Implementation of Tainting/Untainting Logic

Observation: All instructions turn non-speculative in-order

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

→ resolved!
```

Delay execution!
Efficient Implementation of Tainting/Untainting Logic

Observation: All instructions turn non-speculative in-order

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

resolved!
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

Consider control-flow 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>
</tbody>
</table>

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>182.0%</td>
</tr>
<tr>
<td>STT</td>
<td>14.5%</td>
</tr>
</tbody>
</table>
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?