Speculative Data-Oblivious Execution: Mobilizing Safe Prediction For Safe and Efficient Speculative Execution

JIYONG YU, NAMRATA MANTRI,
JOSEP TORRELLAS, ADAM MORRISON*, CHRISTOPHER W. FLETCHER
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

*TEL AVIV UNIVERSITY
Speculative Execution Attacks

- Attacker exploits speculative execution to leak data through hardware usage
Speculative Execution Attacks

- Attacker exploits **speculative execution** to leak data through hardware usage

```c
if (addr < N) {
    // speculation
    secret = load [addr];

    // transmit instruction
    transmit secret;
}
```

Speculation starts

```
time
```

(Continued...)
Speculative Execution Attacks

- Attacker exploits **speculative execution** to leak data through hardware usage

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

    // transmit instruction
    transmit secret;
}
```

Speculation starts

Speculative `secret` is accessed

**time**
Speculative Execution Attacks

- Attacker exploits **speculative execution** to leak data through hardware usage

```plaintext
if (addr < N) { // speculation
    // access instruction
    secret = load [addr];
    // transmit instruction
    transmit secret;
}
```

- Speculation starts
- Speculative `secret` is accessed
- Speculative `secret` is transmitted via hardware usage
Speculative Execution Attacks

- Attacker exploits **speculative execution** to leak data through hardware usage

```c
if (addr < N) {
    // speculation
    // access instruction
    secret = load [addr];
    // transmit instruction
    transmit secret;
}
```

- Speculation starts
- Speculative `secret` is accessed
- Speculative `secret` is transmitted via hardware usage

**Shared hardware**

---

**Speculative Data-Oblivious Framework**

- SDO for Loads
- Evaluation
- Conclusion
Speculative Execution Attacks

- Attacker exploits **speculative execution** to leak data through hardware usage

```c
if (addr < N) { // speculation
    // access instruction
    secret = load [addr];
    // transmit instruction
    transmit secret;
}
```

- Speculation starts
- Speculative `secret` is accessed
- Speculative `secret` is transmitted via hardware usage

**Shared hardware**
Speculative Execution Attacks

- Attacker exploits **speculative execution** to leak data through hardware usage

```plaintext
if (addr < N) {
    // speculation
    // access instruction
    secret = load [addr];
    // transmit instruction
    transmit secret;
}
```

- Speculation starts
- Speculative `secret` is accessed
- Speculative `secret` is transmitted via hardware usage

**Shared hardware**
Speculative Execution Attacks

- Attacker exploits **speculative execution** to leak data through hardware usage

```c
if (addr < N) {  // speculation
    // access instruction
    secret = load [addr];
    // transmit instruction
    transmit secret;
}
```

**Speculation starts**

Speculative `secret` is accessed

Speculative `secret` is transmitted via hardware usage

Attacker infers `secret` via hardware state
Existing Mitigations

- How to deal with transmit secret?
Existing Mitigations

- How to deal with **transmit secret**?
- Solution: Delayed Execution
  - Prior works: SpecShield [PACT’19], NDA [MICRO’19], STT [MICRO’19]

<table>
<thead>
<tr>
<th>Transmit instruction</th>
<th>Hardware vulnerability</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>Cache side channel</td>
</tr>
<tr>
<td>Floating point</td>
<td>Subnormal floating point</td>
</tr>
<tr>
<td>operations</td>
<td></td>
</tr>
<tr>
<td>......</td>
<td>......</td>
</tr>
</tbody>
</table>
Existing Mitigations

- How to deal with \texttt{transmit secret}? 
- Solution: Delayed Execution
  - Prior works: SpecShield [PACT’19], NDA [MICRO’19], STT [MICRO’19]

<table>
<thead>
<tr>
<th>Transmit instruction</th>
<th>Hardware vulnerability</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>Cache side channel</td>
</tr>
<tr>
<td>Floating point</td>
<td>Subnormal floating point</td>
</tr>
<tr>
<td>operations</td>
<td></td>
</tr>
<tr>
<td>......</td>
<td>......</td>
</tr>
</tbody>
</table>

\begin{verbatim}
if (addr < N) {
  // speculation
  // access instruction
  secret = load [addr];

  // transmit instruction
  transmit secret;

  Delaying execution
}
\end{verbatim}
Existing Mitigations

- How to deal with transmit secret?
- Solution: Delayed Execution
  - Prior works: SpecShield [PACT’19], NDA [MICRO’19], STT [MICRO’19]
- Strong security guarantee
- High performance overhead

<table>
<thead>
<tr>
<th>Transmit instruction</th>
<th>Hardware vulnerability</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>Cache side channel</td>
</tr>
<tr>
<td>Floating point operations</td>
<td>Subnormal floating point</td>
</tr>
<tr>
<td>......</td>
<td>......</td>
</tr>
</tbody>
</table>
Existing Mitigations

▪ How to deal with transmit secret?

Improve the performance of Delayed Execution and Maintain its security guarantee.
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: \((2.23\times10^{-308}, 1.79\times10^{308})\), processed by Floating-Point Unit (FPU)
  - Subnormal input: \((4.9\times10^{-324}, 2.23\times10^{-308})\), requiring microcode assist

```
a = fpop a, b
```

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

- (a is subnormal) || (b is subnormal)
  - Latency = \(Y > X\)
  - Slow path (with microcode assist)
Problem: Leaking Whether Input is Normal/Subnormal

// owned by victim
a = fpmult a, b

// owned by attacker
c = fpmult c, d
Problem: Leaking Whether Input is Normal/Subnormal

Both $a$ and $b$ are normal

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

Fast path (FPU only)

Latency = $X$

Slow path (with microcode assist)

Latency = $Y > X$

// owned by victim
$a = \text{fpmult} \ a, \ b$

// owned by attacker
$c = \text{fpmult} \ c, \ d$

Timeline:

Using fast path | $c = \text{fpmult} \ c, \ d$
Problem: Leaking Whether Input is Normal/Subnormal

Fast path (FPU only)
Latency = X

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

// owned by attacker
\[ c = fpmult c, d \]

// owned by victim
\[ a = fpmult a, b \]

Both a and b are normal
Using fast path \( X \)

a or b is subnormal
Using slow path \( Y \)

Using fast path:
\[ a = fpmult a, b \]
\[ c = fpmult c, d \]

Using slow path:
\[ a = fpmult a, b \]
\[ c = fpmult c, d \]
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
Idea 1: Being Data Oblivious

```
a = 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

```
c = fpmult c, d
```

Using fast and slow path

Timeline:

```
0  X  Y
```

Using fast and slow path
Idea 1: Being Data Oblivious

Latency = X

Fast path (FPU only) -> result (fast)

Latency = Y > X

Slow path (with microcode assist) -> result (slow)

Select the correct answer

Using fast and slow path

Paying performance for security

a = fpmult a, b

c = fpmult c, d
Idea 2: “Predicting” Execution to Perform

Latency = X

Fast path (FPU only)

Latency = Y > X

Slow path (with microcode assist)

a = fpmult a, b

c = fpmult c, d

Predicting fast path

Predictor

“Predict”

timeline
Idea 2: “Predicting” Execution to Perform

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

May be invalid

Predictor

0  Predicting fast path  X

Timeline

Idea 2: “Predicting” Execution to Perform

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

May be invalid

Predictor

0  Predicting fast path  X

Timeline

Idea 2: “Predicting” Execution to Perform

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

May be invalid

Predictor

0  Predicting fast path  X

Timeline
Idea 2: “Predicting” Execution to Perform

```
a = fpmult a, b
Latency = X
result (fast) ➔ Dependent instructions

P + Y

Predicting fast path
```

```
a = fpmult a, b
Latency = Y > X

Slow path (with microcode assist)
```

```
c = fpmult c, d
```

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

```
P
Resolving to slow path
```
Idea 2: “Predicting” Execution to Perform

- Fast path (FPU only)
  - Latency = $X$
  - Result (fast)
  - Dependent instructions
- Slow path (with microcode assist)
  - Latency = $Y > X$

Potential new leakage

$\text{a} = \text{fpmult a, b}$
Applying STT for Security

Speculative Taint Tracking
Applying STT for Security

Predictor

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

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

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

Dependent instructions

Prevent leakage via Prediction/Resolution

Speculative Taint Tracking
Applying STT for Security

Speculative Taint Tracking

Predictor

Latency = X
Fast path (FPU only)

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

a = fpmult a, b

“Predict”

“Resolve”

“Taint” and hide sensitive results

Prevent leakage via Prediction/Resolution

Dependent instructions

result (fast)
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 Great (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*
Speculative Data Oblivious Execution (SDO)

What’s Next:

- Generic SDO Framework
- Implementing SDO for load instructions
- Evaluation
- Conclusion
SDO Framework

- Step 1: Define data-oblivious (DO) variants for unsafe transmitters
SDO Framework: Step 1: Define Data-oblivious (DO) Variants

<table>
<thead>
<tr>
<th>Transmit instruction</th>
<th>dest &lt;- op args</th>
</tr>
</thead>
<tbody>
<tr>
<td>DO variants</td>
<td>DO-op_1</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>DO-op_N</td>
</tr>
<tr>
<td>Execution of DO variants</td>
<td>(dest_1, success_1) &lt;- DO-op_1 args</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>(dest_N, success_N) &lt;- DO-op_N args</td>
</tr>
</tbody>
</table>
SDO Framework: Step 1: Define Data-oblivious (DO) Variants

**Transmit instruction**

dest <- op args

**DO variants**

- DO-op_1
- ...
- DO-op_N

**Execution of DO variants**

- (dest_1, success_1) <- DO-op_1 args
- ...
- (dest_N, success_N) <- DO-op_N args

**Fast path (FPU only)**

dest = fpmult args

**Slow path (with microcode assist)**

fast_path

slow_path
SDO Framework: Step 1: Define Data-oblivious (DO) Variants

<table>
<thead>
<tr>
<th>Transmit instruction</th>
<th>dest &lt;- op args</th>
</tr>
</thead>
<tbody>
<tr>
<td>DO variants</td>
<td>DO-op₁, …, DO-opₙ</td>
</tr>
<tr>
<td>Execution of DO variants</td>
<td>(dest₁, success₁) &lt;- DO-op₁ args, …, (destₙ, successₙ) &lt;- DO-opₙ args</td>
</tr>
</tbody>
</table>

Fast path (FPU only):

dest = fpmult args

Slow path (with microcode assist):

dest_{fast}, success_{fast} <- fast_path args (success_{fast} = TRUE if args is normal)
dest_{slow}, success_{slow} <- slow_path args (success_{slow} = TRUE if args is subnormal)
SDO Framework: Step 2: Predict Which DO Variant to Use

<table>
<thead>
<tr>
<th>Transmit instruction</th>
<th>dest &lt;- op args</th>
</tr>
</thead>
<tbody>
<tr>
<td>DO variants</td>
<td>DO-op₁, ... DO-opₙ</td>
</tr>
<tr>
<td>Predictor</td>
<td>Pred</td>
</tr>
<tr>
<td>Predicting DO variant</td>
<td>i &lt;- Pred.predict (public_input) (destᵢ, successᵢ) &lt;- DO-opᵢ args</td>
</tr>
</tbody>
</table>
SDO Framework: Step 2: Predict Which DO Variant to Use

<table>
<thead>
<tr>
<th>Transmit instruction</th>
<th>dest &lt;- op args</th>
</tr>
</thead>
<tbody>
<tr>
<td>DO variants</td>
<td>DO-op₁</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>DO-opₙ</td>
</tr>
<tr>
<td>Predictor</td>
<td>Pred</td>
</tr>
<tr>
<td>Predicting DO variant</td>
<td>i &lt;- Pred.predict (public_input)</td>
</tr>
<tr>
<td></td>
<td>(destᵢ, successᵢ) &lt;- DO-opᵢ args</td>
</tr>
</tbody>
</table>

**Dependent instructions**

\[ \text{dest}_{\text{fast}} \]

**Fast path (FPU only)**

\[ \text{dest} = \text{fpmult} \text{ args} \]

**Slow path (with microcode assist)**

**Static Predictor: always predicting “Fast path”**

\[ \text{dest}_{\text{fast}}, \text{success}_{\text{fast}} \leftarrow \text{fast path args} \]

\[ (\text{success}_{\text{fast}} = \text{TRUE if args is normal}) \]

\[ (\text{success}_{\text{fast}} = \text{FALSE if args is subnormal}) \]
SDO Framework: Step 3: Resolve Prediction when safe

<table>
<thead>
<tr>
<th>Transmit instruction</th>
<th>dest \leftarrow op \text{ args}</th>
</tr>
</thead>
<tbody>
<tr>
<td>DO variants</td>
<td>DO-op_1</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>DO-op_N</td>
</tr>
<tr>
<td>Predictor</td>
<td>Pred</td>
</tr>
<tr>
<td>Predicting DO variant</td>
<td>i \leftarrow \text{Pred.predict} (\text{public_input})</td>
</tr>
<tr>
<td></td>
<td>(dest_i, success_i) \leftarrow DO-op_i \text{ args}</td>
</tr>
<tr>
<td>Resolving when safe</td>
<td>Pred.update(...)</td>
</tr>
<tr>
<td></td>
<td>if (!success_i)</td>
</tr>
<tr>
<td></td>
<td>squash from &quot;dest \leftarrow op \text{ args}&quot;</td>
</tr>
</tbody>
</table>

- **Fast path (FPU only)**: `dest_{fast} = \text{fpmult args}`
- **Slow path (with microcode assist)**:
SDO Framework: Step 3: Resolve Prediction when safe

Transmit instruction: \(\text{dest} \leftarrow \text{op} \text{ args}\)

DO variants:
- \(\text{DO-op}_1\)
- ...
- \(\text{DO-op}_N\)

Predictor: \(\text{Pred}\)

Predicting DO variant:
- \(i \leftarrow \text{Pred.predict (public_input)}\)
- \((\text{dest}_i, \text{success}_i) \leftarrow \text{DO-op}_i \text{ args}\)

Resolving when safe:
- \(\text{Pred.update(...)}\)
- \(\text{if (!success}_i)\)
- \(\text{squash from "dest} \leftarrow \text{op args"}\)

Fast path (FPU only):
- \(\text{dest} = \text{fpmult args}\)

Slow path (with microcode assist):
- \(\text{success}_\text{fast} = \text{FALSE}\)
Designing SDO for Loads

- Load is the vital motivation and challenge for SDO
  - The execution of loads is complicated, susceptible to various attacks
  - Most performance overhead comes from loads
Step 1: Define DO Variants for Loads

- **DO variants**
  - **DO-\texttt{ld}_L1**: only accessing L1
  - **DO-\texttt{ld}_L2**: only accessing L1 and L2 sequentially
  - **DO-\texttt{ld}_L3**: only accessing L1, L2 and L3 sequentially
  - **DO-\texttt{ld}_\texttt{Mem}**: accessing L1, L2, L3 and DRAM sequentially

- \((\texttt{dest}_{xx}, \texttt{success}_{xx}) \leftarrow \text{DO-\texttt{ld}_{xx} \texttt{addr}} \quad // \texttt{dest}_{xx} = 1 \text{ if } \texttt{success}_{xx} = \text{FALSE}\)
Step 1: Define DO Variants for Loads

- **DO variants**
  - DO-1d_{L1} : only accessing L1
  - DO-1d_{L2} : only accessing L1 and L2 sequentially
  - DO-1d_{L3} : only accessing L1, L2 and L3 sequentially
  - DO-1d_{Mem} : accessing L1, L2, L3 and DRAM sequentially

- \((\text{dest}_{xx}, \text{success}_{xx}) \leftarrow \text{DO-1d}_{xx} \text{ addr} \quad // \text{dest}_{xx} = 1 \text{ if success}_{xx} == \text{FALSE}\)

- DO variants \((\text{DO-1d}_{L1})\) must be free of adversary-observable hardware resource usage
  - Cannot modify cache state (tag, data, LRU bits, etc.)
  - Cannot incur address-dependent latency (e.g., free of bank conflict, port contention)
  - ......
Step 1: Define DO Variants for Loads

- **DO variants**
  - **DO-ld\_L1**: only accessing L1
  - **DO-ld\_L2**: only accessing L1 and L2 sequentially
  - **DO-ld\_L3**: only accessing L1, L2 and L3 sequentially
  - **DO-ld\_Mem**: accessing L1, L2, L3 and DRAM sequentially

- $(\text{dest}_{XX}, \text{success}_{XX}) \leftarrow \text{DO-ld}_{XX} \text{ addr} \quad // \quad \text{dest}_{XX} = 1 \text{ if } \text{success}_{XX} = \bot$

- DO variants $(\text{DO-ld}_{L1})$ must be free of adversary-observable hardware resource usage
  - Cannot modify cache state (tag, data, LRU bits, etc.)
  - Cannot incur address-dependent latency (e.g., free of bank conflict, port contention)
  - ......

- For more details (e.g., load re-ordering, performance optimizations) about DO variants, please see the paper
Step 2: Predict Which DO Variant to Use

- **Goal: accurate and precise** cache level prediction
  - Suppose a load requires data from cache level \( i \) and the predictor predicts level \( j \)
  - “accurate and precise”: \( i == j \)
  - “accurate but imprecise”: \( i < j \) -> redundant cache access -> unnecessary load latency
  - “inaccurate”: \( i > j \) -> cache miss -> writeback \( \bot \) to dependents -> squash

<table>
<thead>
<tr>
<th>Predicted level</th>
<th>DO Variant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (L1)</td>
<td>DO–1d_{L1}</td>
</tr>
<tr>
<td>2 (L2)</td>
<td>DO–1d_{L2}</td>
</tr>
<tr>
<td>3 (L3)</td>
<td>DO–1d_{L3}</td>
</tr>
<tr>
<td>4 (Memory)</td>
<td>DO–1d_{Mem}</td>
</tr>
</tbody>
</table>
Step 2: Predict Which DO Variant to Use

- **Goal:** accurate and precise cache level prediction
  - Suppose a load requires data from cache level \( i \) and the predictor predicts level \( j \)
  - “accurate and precise”: \( i == j \)
  - “accurate but imprecise”: \( i < j \) \implies\ redundant cache access \implies\ unnecessary load latency
  - “inaccurate”: \( i > j \) \implies\ cache miss \implies\ writeback \perp\ to dependents \implies\ squash

- **Hybrid predictor:**
  - “Greedy” (for loads with irregular access pattern):
    Maintain a history, and pick the lowest level among history
  - “Loop” (for loads with regular access pattern)
    Learn the recurring pattern, and predict based on the pattern

<table>
<thead>
<tr>
<th>Predicted level</th>
<th>DO Variant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (L1)</td>
<td>DO-1d_L1</td>
</tr>
<tr>
<td>2 (L2)</td>
<td>DO-1d_L2</td>
</tr>
<tr>
<td>3 (L3)</td>
<td>DO-1d_L3</td>
</tr>
<tr>
<td>4 (Memory)</td>
<td>DO-1d_Mem</td>
</tr>
</tbody>
</table>
Step 3: Resolve When Load is Safe

- Update the predictor

- Squash if `success == FALSE`

- In a multi-processor:
  - `DO-ld_{lx}` cannot modify cache state
  - Data fetched by `DO-ld_{lx}` may not be cached in L1
  - May missing cache invalidation

- Solution: send a second load request to validate if a cache invalidation was missed
  - We adopt the validation infrastructure proposed in InvisiSpec [MICRO’18]
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−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
Conclusion

- SDO serves as a new speculative execution attack mitigation with high-performance and high-security

- The proposed SDO framework augments STT with significant speedup without compromising security

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

=  

**Safe, early execution of transmitters**
Applying STT for Security

STT: prediction and resolution never depend on sensitive data

We can build new predictors to get more performance

Prevent leakage via Prediction/Resolution

“Taint” and hide sensitive results

Speculative Taint Tracking