Hardware-Software Contracts for Secure Speculation

Jan Reineke @ Universität des Saarlandes

Joint work with
Marco Guarnieri, Pepe Vila @ IMDEA Software, Madrid
Boris Köpf @ Microsoft Research, Cambridge, UK

Supported by Intel Strategic Research Alliance (ISRA)
“Information Flow Tracking across the Hardware-Software Boundary”
Exploits **speculative execution** to leak sensitive information.

Almost all modern processors are affected.

---

Countermeasures

Software-level countermeasures
✓ *Example*: insert “fences” to selectively terminate speculative execution
✓ Implemented in major compilers (Microsoft Visual C++, Intel ICC, Clang)

Hardware-level countermeasures
✓ Disabling speculation
✓ Delaying speculative loads
✓ Taint tracking of speculative data: STT & NDA
Hardware-level countermeasures

InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy

Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison
University of Illinois at Urbana-Champaign
{myan8, jchoi42, skarlat2}@illinois.edu

Invisible in the Cache Hierarchy

Kevin Loughlin
University of Michigan

Efficient Invisible Speculative Execution through Selective Delay and Value Prediction

Stefanos Kaxiras
Uppsala University
stefanos.kaxiras@it.uu.se

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

Jiyong Yu
University of Illinois at Urbana-Champaign
myan8@illinois.edu

Mengjia Yan
University of Illinois at Urbana-Champaign
myan8@illinois.edu

Josep Torrellas
University of Illinois at Urbana-Champaign
torrella@illinois.edu

Artem Khyzha
Tel Aviv University
artkhyzha@mail.tau.ac.il

Christopher W. Fletcher
University of Illinois at Urbana-Champaign
cwfletcher@illinois.edu

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

Moinuddin K. Qureshi
Georgia Institute of Technology
moin@gatech.edu

NDoS: Approach to Safe Speculation

Ofir Weisse
University of Michigan

Thomas F. Wenisch
University of Michigan
Goals

1. Capture hardware-level countermeasures in a unifying framework

2. Introduce hardware-software contracts to capture their security guarantees

3. Requirements for secure programming based on hardware-software contracts
Outline

1. Speculative Execution Attacks
2. Hardware-level Countermeasures
3. Hardware-Software Contracts
4. Requirements for Secure Programming
1. Speculative Execution Attacks
Microarchitecture 101

**Cache**
Fast but small memory storing recently accessed data across cores

Die shot of AMD “Barcelona” Quad Core CPU
Background: Caches
if \( x < A\_size \)
\[ y = B[A[x]] \]

\[ c \leftarrow x < A\_size \]
\[ \text{beqz } c, \ END \]
\[ L1: \text{load } t, A + x \]
\[ \text{load } y, B + t \]
\[ END: \]

μAssembly = our “toy” assembly language
Background: Speculative execution

• Predict instructions’ outcomes and speculatively continue execution

• Rollback changes if speculation was wrong

Only architectural (ISA, “logical”) state, **not** microarchitectural state
Background: Branch prediction

```
c ← x < A.size
beqz c, END HELP
L1: load t, A + x
load y, B + t
END:
```

Size of array A

Predictions based on branch history & program structure
Background: Reorder buffer

- Key hardware data structure for out-of-order and speculative execution
- Keeps track of “in-flight instructions”

Example:

<table>
<thead>
<tr>
<th>Entry</th>
<th>Instruction</th>
<th>Control Dep.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>( c \leftarrow x &lt; A_size )</td>
<td>-</td>
</tr>
<tr>
<td>1:</td>
<td>beqz ( c ), END</td>
<td>-</td>
</tr>
<tr>
<td>2:</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3:</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Speculative Instruction Fetch

<table>
<thead>
<tr>
<th>Entry</th>
<th>Instruction</th>
<th>Control Dep.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>( c \leftarrow x &lt; A_size )</td>
<td>-</td>
</tr>
<tr>
<td>1:</td>
<td>beqz ( c ), END</td>
<td>-</td>
</tr>
<tr>
<td>2:</td>
<td>load ( t ), ( A + x )</td>
<td>2</td>
</tr>
<tr>
<td>3:</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

\[ \begin{align*}
c & \leftarrow x < A\_size \\
\text{beqz } c, \text{ END} \\
L1: & \text{load } t, \ A + x \\
& \text{load } y, \ B + t \\
\text{END:} & \end{align*} \]
Background: Reorder buffer

- Key hardware data structure for out-of-order and speculative execution
- Keeps track of “in-flight instructions”

Example:

<table>
<thead>
<tr>
<th>Entry</th>
<th>Instruction</th>
<th>Control Dep.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( c \leftarrow x &lt; A_size )</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>beqz ( c ), END</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>load ( t, A + x )</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Speculative Instruction Fetch</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: beqz ( c ), END</td>
</tr>
<tr>
<td>2: load ( t, A + x )</td>
</tr>
<tr>
<td>3: load ( y, B + t )</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Entry</th>
<th>Instruction</th>
<th>Control Dep.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( c \leftarrow x &lt; A_size )</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>beqz ( c ), END</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>load ( t, A + x )</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

| END: |

\( L1: \) load \( t, A + x \) 
load \( y, B + t \)
## Background: Reorder buffer

- Key hardware data structure for out-of-order and speculative execution
- Keeps track of “in-flight instructions”

### Example:

<table>
<thead>
<tr>
<th>Entry</th>
<th>Instruction</th>
<th>Control Dep.</th>
<th>Control Dep.</th>
<th>Control Dep.</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>c ← x &lt; A_size</td>
<td>-</td>
<td>x</td>
<td>&lt; A_size</td>
<td>c ← x &lt; A_size</td>
</tr>
<tr>
<td>1</td>
<td>beqz c, END</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>beqz c, END</td>
</tr>
<tr>
<td>2</td>
<td>load t, A + x</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>load t, A + x</td>
</tr>
<tr>
<td>3</td>
<td>load y, B + t</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>load y, B + t</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Background: Reorder buffer

- Key hardware data structure for out-of-order and speculative execution
- Keeps track of “in-flight instructions”

Example:

<table>
<thead>
<tr>
<th>Entry</th>
<th>Instruction</th>
<th>Control Dep.</th>
<th>Rollback mis-speculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>c ← 0</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1:</td>
<td>beqz 0, END</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>2:</td>
<td>load t, A + x</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3:</td>
<td>load y, B + t</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Entry</th>
<th>Instruction</th>
<th>Control Dep.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>c ← x &lt; A_size</td>
<td>-</td>
</tr>
<tr>
<td>1:</td>
<td>beqz c, END</td>
<td>-</td>
</tr>
<tr>
<td>L1:</td>
<td>load t, A + x</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>load y, B + t</td>
<td>-</td>
</tr>
<tr>
<td>END:</td>
<td></td>
<td>-</td>
</tr>
</tbody>
</table>

Rollback mis-speculation:
## Background: Reorder buffer

- **Key hardware data structure for out-of-order and speculative execution**
- **Keeps track of “in-flight instructions”**

**Example:**

<table>
<thead>
<tr>
<th>Entry</th>
<th>Instruction</th>
<th>Control Dep.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>c ← 0</td>
<td>-</td>
</tr>
<tr>
<td>1:</td>
<td>beqz 0, \textit{END}</td>
<td>-</td>
</tr>
<tr>
<td>2:</td>
<td>load t, A + x</td>
<td>2</td>
</tr>
<tr>
<td>3:</td>
<td>load y, B + t</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Example continued:**

<table>
<thead>
<tr>
<th>Entry</th>
<th>Instruction</th>
<th>Control Dep.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>beqz c, \textit{END}</td>
<td>-</td>
</tr>
<tr>
<td>1:</td>
<td>load t, A + x</td>
<td>-</td>
</tr>
<tr>
<td>2:</td>
<td>load y, B + t</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

\( c \leftarrow x < A\text{\_size} \)

\textit{L1: load t, A + x}

\textit{load y, B + t}

\textit{END:}
void f(int x):
    c ← x < A_size
    beqz c, END
L1: load t, A + x
    load y, B + t
END:

What is in A[128]?

Spectre V1

A_size=16

B
B[0] B[1] ...
### Spectre V1

**What is in \( A[128] \)?**

1. **Training**
   - \( f(0); f(1); f(2); \ldots \)

2. **Prepare cache ("Prime")**
   - 2) Run \( f(128) \)

3. **Extract from cache ("Probe")**

---

**Cache state**

\[
\begin{array}{c}
A_{\text{size}} = 16 \\
B_{[0]} B_{[1]} B_{[A[128]]} \\
\end{array}
\]

**Address depends on** \( A[128] \)

**Persistent beyond rollback**

\[ \text{void } f(\text{int } x) : \]

\[
\begin{align*}
  c &\leftarrow x < A_{\text{size}} \\
  \text{beqz } c, \text{ END} \\
  L1: \text{ load } t, A + x \\
  \text{ load } y, B + t \\
\end{align*}
\]

**END:**
2. Hardware-level Countermeasures
A parametric speculative, out-of-order processor

- Hardware-level countermeasures restrict speculative execution
- Intended to work for arbitrary
  - (compliant) scheduler
  - memory hierarchy
  - branch predictor

We introduce a **processor model** that is **parametric** in
- scheduler
- caches
- branch predictor
A parametric speculative, out-of-order processor ... formalized as binary relation on hardware states

\[ \langle m, a, buf, cs, bp, sc \rangle \]
A parametric speculative, out-of-order processor ...
... formalized as binary relation on hardware states

\[ \langle m, a, buf, cs, bp \rangle \overset{d}{\Rightarrow} \langle m', a', buf', cs', bp' \rangle \]

\[ d = next(sc) \quad sc' = update(sc, buf' \downarrow) \]

\[ \langle m, a, buf, cs, bp, sc \rangle \Rightarrow \langle m', a', buf', cs', bp', sc' \rangle \]

directive from scheduler, fetch, execute i, retire

predecessor state

successor state
Rules capture effect of directives
Example: Fetch

Applying reorder buffer to register state

**Fetch-Branch-Hit**

\[
\begin{align*}
    a' &= apl(buf, a) \\
    |buf| &< w \\
    p(a'(pc)) &= beqz x, \ell \\
    \ell' &= predict(bp, a'(pc)) \\
    access(cs, a'(pc)) &= Hit \\
    update(cs, a'(pc)) &= cs' \\
    \langle m, a, buf, cs, bp \rangle &\xrightarrow{\text{fetch}} \langle m, a, buf \cdot pc \leftarrow \ell' @ a'(pc), cs', bp \rangle
\end{align*}
\]

Reorder buffer is not full
Branch predictor says \( l' \) is next

Instruction is a branch
Add change of pc to reorder buffer
Updating cache state
Rules capture effect of directive

result of instruction at head of reorder buffer is resolved

\[ \text{RETIRED-ASSIGNMENT} \]

\[
\text{buf} = x \leftarrow v \cdot \epsilon \cdot \text{buf}'
\]

\[
v \in Vals
\]

\[
\langle m, a, \text{buf}, cs, bp \rangle \xrightarrow{\text{retire}} \langle m, a[x \mapsto v], \text{buf}', cs, bp \rangle
\]

apply change to registers

and remove entry from reorder buffer
Capturing countermeasures

- Countermeasures restrict freedom of scheduler
- Defined for arbitrary cache and branch predictor

We consider three approaches:

1. Disabling speculation
2. Delaying speculative loads
3. Taint tracking of speculative values
Disabling speculative execution

Constrain scheduler to

1. **fetch**
2. **execute 1**
3. **retire**
4. go to 1.

- Obviously eliminates all speculative-execution attacks.
- Really slow.
Delaying speculative loads (Sakalis et al., ISCA 2019)

**Step-Others**

\[
\langle m, a, buf, cs, bp \rangle^d \Rightarrow \langle m', a', buf', cs', bp' \rangle \\
\text{if } d = \text{next}(sc) \quad \text{then } sc' = \text{update}(sc, buf') \\
\text{and } d \in \{\text{fetch}, \text{retire}\} \vee (d = \text{execute } i \land \text{buf}|i \neq \text{load } x, e) \\
\langle m, a, buf, cs, bp, sc \rangle \Rightarrow \text{loadDelay} \langle m', a', buf', cs', bp', sc' \rangle
\]

**Step-Eager-Delay**

\[
\langle m, a, buf, cs, bp \rangle^d \Rightarrow \langle m', a', buf', cs', bp' \rangle \\
\text{if } d = \text{next}(sc) \quad \text{then } sc' = \text{update}(sc, buf') \\
\text{if } buf|i = \text{load } x, e \quad \forall pc \leftarrow \ell @ \ell' \in \text{buf}[0..i - 1]. \ell' = \epsilon \\
\langle m, a, buf, cs, bp, sc \rangle \Rightarrow \text{loadDelay} \langle m', a', buf', cs', bp', sc' \rangle
\]

**Question:** Does this eliminate all speculative-execution attacks?
Delaying speculative loads (Sakalis et al., ISCA 2019)

Spectre v1 Example

\[
\begin{align*}
c & \leftarrow x < A_{\text{size}} \\
\text{beqz } c & , \text{ END} \\
L1: \quad \text{load } t & , A + x \\
\text{load } y & , B + t \\
\text{END:} & \\
\end{align*}
\]

Will only be performed non-speculatively.

$\rightarrow$ Problem solved.
Delaying speculative loads (Sakalis et al., ISCA 2019)

Variant of Spectre v1 Example

t = A[x]
if (x < A_size)
  if (B[t])
    ...

\[
c \leftarrow x < A\_size
\]
load t, A + x
beqz c, END
L1: beqz t, L2
END:

Unlike entirely non-speculative execution, leaks whether A[x] is 0!

Question: How to capture its security guarantees?
Taint-tracking speculative values
(STT, Yu et al., MICRO 2019, NDA, Weisse et al. MICRO 2019)

- Allow speculative loads, but make sure the loaded values do not leak

Difference:

- STT: Prevent any “transmit” instruction on data derived from speculative loads
- NDD: Prevent any propagation of speculatively loaded data
  (more conservative than STT)
Taint-tracking speculative values
(STT, Yu et al., MICRO 2019, NDA, Weisse et al. MICRO 2019)

**Step**

\[
\begin{align*}
    d &= \text{next}(sc) \\
    \langle m, a, \textcolor{blue}{buf_{ul}}, cs, bp \rangle &\xrightarrow{d} \langle m', a', \textcolor{blue}{buf'_{ul}}, cs', bp' \rangle \\
    sc' &= \text{update}(sc, \textcolor{red}{buf'}_{\downarrow}) \\
    \textcolor{blue}{buf'} &= \text{lbl}(\textcolor{blue}{buf'_{ul}}, buf, d) \\
    \langle m, a, \textcolor{blue}{buf}, cs, bp, sc \rangle &\xrightarrow{\text{tt}} \langle m', a'\textcolor{blue}{buf'}, cs', bp', sc' \rangle
\end{align*}
\]

Hiding data that should not leak

“Richer” reorder buffer state captures “taint”

Allows for more speculative and out-of-order execution.
But how secure is it?
Taint-tracking speculative values
(STT, Yu et al., MICRO 2019, NDA, Weisse et al. MICRO 2019)

\[
t = \mathbf{A}[x]
\]
\[
\text{if (} x < \mathbf{A}_{\text{size}} \text{)}
\]
\[
y = \mathbf{B}[t]
\]

“semantically equivalent”
to Spectre v1 example

Potentially out-of-array-bounds
Leaks \( \mathbf{A}[x] \) under both STT and NDA
3. Hardware-Software Contracts
Hardware-software contracts

Goals:

• Capture security guarantees in a **simple, mechanism-independent** manner

• Carve out **differences between existing countermeasures**

• Serve as a **basis for secure programming**
Meaning of contracts

Definition 1 ({{·}} ⊨ [·]). A hardware semantics {{·}} satisfies a contract [·] if, for all programs \( p \) and all initial arch. states \( \sigma, \sigma' \), if \( [p](\sigma) = [p](\sigma') \), then \( \{p\}(\sigma) = \{p\}(\sigma') \).

Contract assigns sequences of observations to architectural states

States \( \sigma, \sigma' \) are contract-indistinguishable.

States \( \sigma, \sigma' \) are HW-indistinguishable.
Structure of contracts

**Contract** =

Execution Mode · Observer Mode

“How are programs executed”

“What is visible about the execution?”
Two execution modes

**Sequential (seq)** = non-speculative execution

**Speculative (spec)** = each branch is mispredicted and speculatively executed
Observer modes

**Program counter (pc)** = observer can see addresses of
  - instruction fetches

**Constant time (ct)** = observer can see addresses of
  - loads
  - stores
  - instruction fetches

**Architectural (arch)** = ct + observer can see values of loads
A Lattice of Contracts

“Leaks all data accessed non-speculatively”

“Leaks addresses of non-spec. loads/stores/instruction fetches”
Security Guarantees

- \[ \cdot \cdot \cdot \text{seq-arch} \text{ct} \cdot \cdot \cdot \] seq
- \[ \cdot \cdot \cdot \text{spec-arch} \text{ct} \cdot \cdot \cdot \] spec
- \[ \cdot \cdot \cdot \text{loadDelay} \cdot \cdot \cdot \] ct

\{ \cdot \cdot \cdot \} seq = disabling speculation
\{ \cdot \cdot \cdot \} loadDelay = delaying speculative loads
\{ \cdot \cdot \cdot \} tt = taint tracking
4. Requirements for Secure Programming
What is secure programming?

**Definition 3** \((p \vdash NI(\pi, [\cdot]))\). Program \(p\) is non-interferent w.r.t. contract \([\cdot]\) and policy \(\pi\) if for all initial arch. states \(\sigma, \sigma: \sigma \simeq_L \sigma' \Rightarrow [p](\sigma) = [p](\sigma').\)

"States agree on public data"

"Executions are indistinguishable under contract"

**Proposition 2.** \(If p \vdash NI(\pi, [\cdot]) \) and \(\{\cdot\} \vdash [\cdot], \) then \(p \vdash NI(\pi, \{\cdot\}).\)
What is secure programming?

Two variants

“Sandboxing”
e.g. Javascript in the browser

“Constant-time programming”
e.g. cryptographic code
Sandboxing

“Traditional” sandboxing mechanisms ensure that malicious code may not access secret data non-speculatively.

Appropriate bounds check

\[
\text{if } (x < \text{A_size}) \\
y = B[A[x]]
\]

Ensures non-interference w.r.t. \( [\cdot]_{\text{arch}}^{\text{seq}} \).

HW-level countermeasures are adequate for sandboxing.
**Constant-time programming**

“Traditional” constant-time programming ensures non-interference w.r.t. $\lbrack \cdot \rbrack_{ct}^{\text{seq}}$.

- HW-level countermeasures are not fully adequate for traditional constant-time programming.
- Need to apply additional SW-level countermeasures.
Thank you for your attention!

Find out more in the paper: