Speculation Invariance (InvarSpec): Faster Safe Execution Through Program Analysis

Zirui Neil Zhao, Houxiang Ji, Mengjia Yan, Jiyong Yu, Christopher Fletcher, Adam Morrison, Darko Marinov, Josep Torrellas
Univ. of Illinois, Tel Aviv Univ.
zirui6@illinois.edu

International Symposium on Microarchitecture (MICRO) October 2020
Speculative Execution Attacks

Modern microprocessors are threatened by speculative execution side-channel attacks

**Transient instruction**: instruction bound to squash

**Speculative execution attack**: uses transient instructions to leak secrets

```c
if (x < array1_size) { // mispredicted
    uint8 secret = array1[x];   // transient
    uint8 y = array2[secret * 64]; // transient
}
```
How Most Existing Hardware Defenses Work

- Special hardware mechanism that protects instructions while they are speculative
- When the instruction reaches **Visibility Point (VP)**, the protection is lifted

**VP:** Point when the instruction can execute safely without protection

Depends on the threat model
- **Spectre:** when all older branches are resolved
- **Comprehensive**\(^*\): when the instruction cannot be squashed anymore

\* Yan et al., “InvisiSpec: Making Speculative Execution Invisible in the Cache” (MICRO’18)
Squashing and Transmitter Instructions

Squashing instruction: can cause squashes that may lead to security violations (defined by the threat model)

Transmitter instruction: execution can create operand-dependent microarchitectural resource usage that reveals a secret

* Sakalis et al., “Efficient invisible speculative execution through selective delay and value prediction” (ISCA’19)
Observation: Hardware is Over-Protecting

$qd x$ is neither data nor control dependent on the branch

$qd x$ will always read from the same value of $x$ and commit

*No need to protect $qd x$*
Why? Hardware Only Has the View of Dynamic Execution Path

Wish: ld x can execute before turning non-speculative

Need software support so the HW can identify this case
Insight: **Speculation Invariance**

Intuition: A speculative instruction can become *Speculation Invariant* at some point before turning non-speculative.
Insight: **Speculation Invariance**

Consider *loads* as squashing instructions (due to exceptions and memory consistency violations)

![Diagram showing the difference between speculation invariant and not speculation invariant](image)

Data Dependence (DD)
Contribution

- **InvarSpec**: A software-hardware framework that
  - Identifies when a speculative instruction becomes Speculation Invariant
  - Allows the execution of the speculative instruction without protection

- **Components of InvarSpec:**
  1) Static analysis pass of the program
  2) Core microarchitecture
Contribution

- **InvarSpec**: A software-hardware framework that
  - Identifies when a speculative instruction becomes Speculation Invariant
  - Allows the execution of the speculative instruction without protection

- **Components of InvarSpec**:
  1. Static analysis pass of the program
  2. Core microarchitecture

- **Reduction in the execution overhead of existing hardware defense schemes**
  - Fence while speculative: 195.3% → 108.2%
  - Delay on Miss: 39.5% → 24.4%
  - InvisiSpec: 15.4% → 10.9%
A speculative instruction $i$ becomes Speculation Invariant (SI) when
(1) whether $i$ will execute, and
(2) the values of $i$’s source operands
do not depend on speculative state

We say that $i$ reaches its Execution-Safe Point (ESP) when:
(1) it is speculation invariant, and
(2) its source operands are ready

Proposal: Lift the protection mechanism as soon as a speculative instruction reaches its
ESP and execute without protection

Intuition: The instruction is guaranteed to eventually commit using the exact same
operands, despite any future squashes
What We Gain

Transmitter is ready to be *speculatively* executed

Protection engaged

Transmitter reaches ESP

Transmitter reaches VP

Protection lifted

Transmitter retires
What We Gain

Transmitter is ready to be *speculatively* executed

Protection engaged

Transmission reaches ESP

Protection lifted

Transmission reaches VP

Executes earlier: improves performance

Transmission retires
Safe Set (SS) of an Instruction

**Safe Set (SS) for an instruction** $i$: set of **older** squashing instructions that cannot prevent $i$ from becoming Speculation Invariant.

When is the HW sure that instruction $i$ has become Speculation Invariant?

- **If HW does not know $i$’s SS**: all of its older squashing instructions have produced their final results
- **If HW knows $i$’s SS**: all of its older squashing instructions that are not in $SS(i)$ have produced their final results

Diagram:

- **SS($ld\ x$) = \{branch\}
- **SS($ld\ x$) = \{ld\ y\}
- **z = ld\ y**
InvarSpec's Threat Model

InvarSpec inherits the transmitters and the threat model from the hardware defense scheme that it augments.

Rule out attacks based on the exact timing of when speculative instructions execute.

Why? Because the underlying hardware schemes that InvarSpec augments do not consider them.

In the paper:
- Squashing instructions: branches & loads
- Transmitter instructions: loads
InvarSpec Framework

- **Software**: analysis pass that generates SS for the instructions
  - **Baseline**: Populates SS with instructions that are safe for all the execution paths
  - **Enhanced**: Also places in SS some instructions not safe for some execution paths

- **Hardware**: microarchitecture in the processor core
  - Brings SS of the instruction being executed to the pipeline
  - Computes when the instruction becomes Speculation Invariant and can execute
InvarSpec Analysis: Baseline

Instruction Dependence Graph (IDG) of instruction $i$: a graph that contains all instructions within the same function that may affect whether $i$ executes or the values of $i$'s source operands.

```
b = ld c; // squashing
if (a) {
  // squashing
  x = ld y; // squashing
}
ld x;    // transmitter
```

$SS(ld \ x) = \{ld \ c\}$
InvarSpec Analysis: Enhanced

**Insight:** Some dependencies only exist on certain execution paths to $i$

**Technique:** Remove some of the edges from the IDG($i$) and place more squashing instructions in the SS($i$)

**Benefit:** Better performance while still secure

More details in the paper
Hardware Support: Storing SS

Store SSs in SS pages
- Fixed VA offset between code and SS pages
- Fixed offset between instruction and its SS
Core has a small **SS Cache** with recently-used SSs.
When instruction is decoded, the SS Cache is checked:
- If **Hit**: provide the SS
- If **Miss**: request it when safe
Hardware Support: Computing Speculation Invariance

IFB records all the inflight transmitters and squashing instructions

PC of the instruction

Bitmask that contains SS information

Is it Speculation Invariant?

Has it produced final results?
Hardware Support: Computing Speculation Invariance

PCs from SS cache: $SS(i) = \{0x12F, 0x96\}$

Incoming transmitter $i$

| 0x110 | SS Bitmask | SI | Final |

$=$

| 0x12F | SS Bitmask | SI | Final |
| 0x108 | SS Bitmask | SI | Final |
| 0x96  | SS Bitmask | SI | Final |
| 0x90  | SS Bitmask | SI | Final |

Head

Inflight Buffer (IFB)
Hardware Support: Computing Speculation Invariance

PCs from SS cache: \( SS(i) = \{0x12F, 0x96\} \)

Incoming transmitter \( i \)

<table>
<thead>
<tr>
<th></th>
<th>0x110</th>
<th>0x12F</th>
<th>0x108</th>
<th>0x96</th>
<th>0x90</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS Bitmask</td>
<td>SI Final</td>
<td>SS Bitmask</td>
<td>SS Bitmask</td>
<td>SS Bitmask</td>
<td>SS Bitmask</td>
</tr>
<tr>
<td>SI Final</td>
<td>SI Final</td>
<td>SI Final</td>
<td>SI Final</td>
<td>SI Final</td>
<td>SI Final</td>
</tr>
</tbody>
</table>

Inflight Buffer (IFB)

Head
Hardware Support: Computing Speculation Invariance

PCs from SS cache: SS(i) = {0x12F, 0x96}
Hardware Support: Computing Speculation Invariance

PCs from SS cache: SS(i) = \{0x12F, 0x96\}

Inflight Buffer (IFB)
Hardware Support: Computing Speculation Invariance

PCs from SS cache: \( SS(i) = \{0x12F, 0x96\} \)
Hardware Support: Computing Speculation Invariance

PCs from SS cache: $SS(i) = \{0x12F, 0x96\}$

Inflight Buffer (IFB)
Evaluation: Execution Overhead (SPEC 2017)

Vanilla: Defense scheme without InvarSpec
SS: Defense scheme with InvarSpec Baseline
SS++: Defense scheme with InvarSpec Enhanced

Average execution overhead of InvarSpec over conventional (unsafe) core

**Fence**
Place fences before speculative loads

- Vanilla: 195.3%
- SS: 169.5%
- SS++: 108.2%

**Delay-On-Miss (DOM)**
Allow speculative loads to access only L1; stall if miss on L1

- Vanilla: 39.5%
- SS: 38.4%
- SS++: 24.4%

**InvisiSpec**
Invisibly issue speculative loads and follow up with a second access

- Vanilla: 15.4%
- SS: 14.9%
- SS++: 10.9%

InvarSpec delivers substantial reductions in the execution overhead of defense schemes
Conclusion: InvarSpec

- Defense scheme against speculative execution attacks that combines cooperative compiler and hardware mechanisms
- Can augment many existing hardware-only defense schemes
- Substantially reduces the overhead of defense schemes:
  - Fence: 195.3% → 108.2%
  - DOM: 39.5% → 24.4%
  - InvisiSpec: 15.4% → 10.9%

Available at: http://iacoma.cs.uiuc.edu/iacoma-papers/micro20_1.pdf
Speculation Invariance (InvarSpec): Faster Safe Execution Through Program Analysis

Zirui Neil Zhao, Houxiang Ji, Mengjia Yan, Jiyong Yu, Christopher Fletcher, Adam Morrison, Darko Marinov, Josep Torrellas
Univ. of Illinois, Tel Aviv Univ.
ziruiz6@illinois.edu

International Symposium on Microarchitecture (MICRO) October 2020