InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy

Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison*, Christopher W. Fletcher, and Josep Torrellas

University of Illinois at Urbana-Champaign  *Tel Aviv University
Speculative Execution Attacks

- Exploit the side effects of instructions on incorrectly-speculated paths (instructions to be squashed)

- Exploit a key feature of current ooo processors: speculative execution

- Any meaningful defense needs hardware support

- Defenses will be easier to implement in
  - Simple cores
  - Cores with “little legacy”
Existing Speculative Execution Attacks

- Exploit the side-effects of **transient instructions** (speculatively-executed instructions that are doomed to be squashed)

```plaintext
1: if (x < array1_size) {
2:     val = array1[x]
3:     ld array2[val]
4: }
```

### Existing Attack Sources of Transient Instructions

<table>
<thead>
<tr>
<th>Existing Attack</th>
<th>Sources of Transient Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spectre</td>
<td>Control-flow misprediction</td>
</tr>
<tr>
<td>Meltdown</td>
<td>Virtual memory exception</td>
</tr>
<tr>
<td>L1 Terminal Fault</td>
<td></td>
</tr>
<tr>
<td>Speculative Store Bypass</td>
<td>Address alias between a load and an earlier store</td>
</tr>
</tbody>
</table>
Proposal: **Comprehensive Speculative Attack Model**

- Any speculative load (i.e., load not at the head of ROB) is vulnerable

<table>
<thead>
<tr>
<th>Attack Model</th>
<th>Sources of Transient Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comprehensive</td>
<td>Various events, such as: • Exceptions • Control-flow mispredictions • Address alias between a load and an earlier store • Address alias between two loads • Memory consistency model violations • Interrupts</td>
</tr>
</tbody>
</table>

- New speculative execution attacks may be found in the future
Lifetime of a load instruction

Load is issued to memory

Load is speculative

All prior branches are resolved

Load reaches head of ROB

• Exceptions
• Control-flow mispredictions
• Address alias load and store
• Address alias load and load
• Memory consistency model violations
• Interrupts

Visibility Point

The load becomes unsquashable
How do we Defend against Speculative Attacks?
Trivial Approach

- Delay issuing the load until its Visibility Point

Load could be issued to memory

Visibility Point

Issue the load

Load reaches head of ROB

 Speculative loads are issued later than in a conventional processor

To defend against the comprehensive attack model → the processor becomes an in-order processor
Better Approach

- The load probes the local L1/L2
- If hit, get the data and use it, but do not change the state (i.e., cache replacement bits)
- Otherwise, delay until the Visibility Point

Load could be issued to memory

- Probe L1/L2 and hit
- Visibility Point
- Issue the load

Load reaches head of ROB

- Change replacement bits
- Visibility Point
- Delay
- Load reaches head of ROB
Advanced Approach

- The load probes the local L1/L2
  - If hit, get the data and use it, but do not change the state
  - Otherwise
    - Use Value Prediction to return a value to the pipeline
    - At the Visibility Point: Issue the load and compare the return value to the predicted one
      - Squash if they are different

Load could be issued to memory

Probe L1/L2. If miss, predict value

Visibility Point

Issue the load

Compare

Load reaches head of ROB
More Advanced Approach

- Speculative Taint Tracking: Smart delay-based

Load could be issued to memory

Untainted: Issue without delay

Visibility Point

Tainted: Delay until untainted

Issue the load

Load reaches head of ROB

Visibility Point

Load reaches head of ROB
Super Advanced Approach:

InvisiSpec: Issue the Load Invisibly
InvisiSpec

- Issue the load immediately, invisibly (i.e., no change to the cache hierarchy)
- At the Visibility Point: HW re-issues the load and changes state of caches

Speculative loads are issued as early as in a conventional machine and can pass data to dependent instructions immediately

Add an “invisible transaction” that does not change the states of the caches
Making Unsafe Loads Invisible

- Invisible load request
  - No modification to cache states, including
    - Cache occupancy
    - Replacement information
    - Coherence information
    - TLB state
  - Bring the data to Speculative Buffer (SB) (in addition to the register)
Making Loads Visible at Visibility Point:

- Make the load visible: HW issues a normal request (which changes the caches)
- While in the Window of Invisibility, processor does not receive invalidations

Risk of memory consistency violations

Load is issued to memory

Visibility Point

Make the load visible in cache

Load reaches head of ROB

Issue an invisible load request

Use the value

Window of Invisibility
An Example of Memory Consistency Violation

P1 does not receive an invalidation!

P1 sees the lock is free, but has read old counter.
How to Make Load Visible while Maintaining Consistency?

At Visibility Point: HW issues the request that changes state of caches ("Validation")

Data goes into L1. Compare the incoming data and the one in the SB

If mismatch, squash the load as it violates memory consistency model

- Problem: validations may cause stall at ROB head

```
P1
```

![Diagram showing the process of making load visible while maintaining consistency](image)
How to Reduce the Cost of Validations?

- For loads with no risk of consistency violation: **Issue an Exposure, not a Validation**
  - HW issues the request at the Visibility Point
  - Load does not need to wait for response to retire
  - Data goes into cache. No need to compare data
- High performance: does not cause a stall

There are many cases where a load can use Exposures
Pros & Cons of InvisiSpec

- **Security**
  - Successfully prevent attacks
- **Highest performance**
  - Speculative loads are issued as early as in a conventional machine
- **Applicability**
  - Handle multi-threaded issues
- **No software changes**

- **Add invisible coherence transaction**
- **May stall due to Validations**

**BUT:**
- Most validations are converted to Exposures
Discussion

- There is a range of HW solutions with various tradeoffs
- Ideas ready to be implemented in a simple processor
- Keen to see the impact of a provably (more) secure processor
Conclusion

- InvisiSpec is the first comprehensive defense mechanism against speculative execution attacks in the cache hierarchy.
- We published the code of our architecture simulator: [https://github.com/mjyan0720/InvisiSpec-1.0](https://github.com/mjyan0720/InvisiSpec-1.0)
More in the paper

- Security analysis
- Details of when to use validations and exposures, and overlapping of them
- Details of implementation
- Detailed performance and area overhead evaluation results
Speculative Execution Attacks

- An example of Spectre Variant 1 (array bound checking attack).

Victim code

1: if (x < array1_size) {
2:   val = array1[x]
3:   ld array2[val]
4: }

Attack to read arbitrary memory:

1) Train branch predictor
2) Trigger branch misprediction
3) Side channel

Speculative execution attacks exploit side effects of instructions on paths that will be squashed.