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

†Authors contributed equally to this work.
Motivation: Speculative Execution Attacks

- Hardware speculative execution offers a big attack surface for covert and side channels

- Speculative execution attacks exploit the side effects of instructions on incorrect speculative paths (squashed)

- It is crucial to fix this vulnerability efficiently

Compilers and programmers can not reason about it.
Outline

- Background
- A comprehensive threat model
- Invisible Speculation (InvisiSpec): the first holistic defense mechanism
- Evaluation results and current work
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.
Generalization of Speculative Execution Attacks

- Transient instructions: speculatively-executed instructions that are destined to be squashed.
- Speculative execution attack exploits side effects of transient instructions.

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

<table>
<thead>
<tr>
<th>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>Address alias between a load and an earlier store</td>
</tr>
<tr>
<td>Speculative Store Bypass</td>
<td></td>
</tr>
</tbody>
</table>
Futuristic Speculative Attack Model

- Futuristic speculative attack model
  - An attacker can exploit *any* speculative load (load not at the head of ROB).
  - It includes all existing attacks and future speculative execution attacks

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

- **Load is issued to memory**
- **Load is speculative**
  - All prior branches are resolved
  - Visibility Point
- **Load reaches head of ROB**

**Spectre attack model**
- Unsafe
  - Naïve solution
    - Delay issuing the load until its visibility point
  - The load becomes unsquashable

**Futuristic attack model**
- Unsafe
  - safe
InvisiSpec

- The first holistic defense mechanism against speculative execution attacks
- Key idea:
  - Make unsafe loads invisible in the cache hierarchy

Speculative loads are issued as early as in a conventional machine
Making Unsafe Loads Invisible

- Invisible load request
  - No modification to cache states, including
    - Cache occupancy
    - Replacement information
    - Coherence information
    - TLB state
  - Bring the data, store in Speculative Buffer (SB) and in register
Making Safe Loads Visible

- Make the load visible: HW issues a normal request (which changes the caches)
- While in the window of invisibility, processor does not receive invalidations
An Example of Memory Consistency Violation

- An example under TSO

P1 will not receive an invalidation!

P1 will see the lock is free, but has read old counter.
Maintaining Memory Consistency

- Need to issue another access at Visibility Point: Validation
  1) HW issues a request (which changes caches)
  2) Data goes into cache. Compare the incoming data and the one in the SB
  3) If mismatch, squash the load as it violates memory consistency model
- Problem: validations may cause a processor stall

![Diagram showing memory consistency and validation process]
Maintaining Memory Consistency (II)

- For loads with no risk of memory consistency violation: Exposure
  - HW issues a 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

See the paper for the many cases where a load can use exposures
Pros & Cons of InvisiSpec

- **Security**
  - Successfully prevent attacks in both Spectre and Futuristic attack models
- **High performance**
  - Speculative loads are issued as early as in a conventional machine
- **Applicability**
  - Handle multi-threaded issues
- **No software changes**

- **Performance overhead**
  - Double accesses
  - May stall due to validations

BUT:
- Many can be converted to exposures
- Most hit in L1, and return very quickly
Average Execution Time for SPEC and PARSEC (Normalized)

- Conventional insecure baseline: 0.00
- Delay load until visibility point (Spectre attack model): 1.74
- InvisiSpec: 1.21
- Delay load until visibility point (Futuristic attack model): 3.08
- InvisiSpec: 1.72
Current Work

- Reduce performance overhead: selectively enable InvisiSpec
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
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