Formal Verification of Combined Spectre Attacks

Xaver Fabian1  Koby Chan2  Marco Guarnieri3  Marco Patrignani1

1CISPA Helmholtz Center for Information Security  2Stanford  3Imdea Software Institute

Speculative execution allows CPUs to improve performance by using prediction mechanisms that predict the outcome of branching and other instructions without waiting for the correct result. When the prediction is wrong, the CPU rolls back the effects of the speculatively-executed instructions on the architectural state (i.e., memory, registers). However, the side effects on the microarchitectural state, which includes the cache and buffers, are not rolled back and thus can leak possible confidential data that was speculatively accessed (Listing 1). Spectre attacks [1-4] demonstrate that most modern CPUs are affected by this speculation-based vulnerability.

Listing 1: Standard Spectre v1 example. For \( y \geq size_A \), \( A[y] \) can be speculatively read and is leaked into the cache via an access to array \( B \).

```c
if (y < size_A)
    x = A[y];
    temp &= B[x * 512];
```

Listing 2: Spectre v4 example. If \( y \geq size_A \), \( A[y] \) can be speculatively read and is leaked into the cache via an access to array \( A \).

```c
if (y < size_A)
    x = A[y];
    temp &= B[x * 512];
```

Listing 3: Combining Spectre v1 and v4. Misprediction of the branching instruction in Line 4 and the missed matching \( store \) in Line 3 for the \( load \) of \( p \) in Line 5 leads to leaking the secret value into the cache in Line 5.

```c
if (x != 0)
    temp &= A[*p];
```

In this paper, we propose two new semantics: one for Spectre v4 and one for a combination of Spectre v1 and Spectre v4. Then, we extend the Spectre v1 verification tool Spectector [5] with our new semantics.

We exploit properties of the semantics of Spectre v1 and Spectre v4 to merge them into a new combined semantics we call Spectre v14. This combined semantics detects the vulnerability in the combined semantics and proves that (1) the combined semantics are strictly stronger than its parts and (2) that we can recover the analysis of the parts from the combined semantics. In addition, we provide insights into how countermeasure against one Spectre variant affects the vulnerabilities relying on multiple Spectre variants.

We validate our extension of Spectector on a benchmark for Spectre v4 proposed by Daniel et al. [6] while for the combined v14 semantics, we use our novel code snippets.

For the future, we want to create a semantics for Spectre v5 to investigate how our compositional approach scales to new versions such as v15, v45 and v145.

I. Semantics Foundations

We build on the semantics for Spectre v1 in Guarnieri et al. [5] and devise an always mispredict semantics to model
v4 speculation. The model similarities make it easier to combine the existing v1 semantics with the new v4 one into v14. Additionally, the always mispredict semantics is deterministic, which makes for efficient verification.

We formalise our semantics for a μASM language with the expected assembly-like instructions. Its operational non-speculative semantics makes a program $p$ with configurations $σ$ (which consists of the memory and the register assignments) step while producing observable actions $τ$ (reads, writes, pc locations). This semantics is denoted with $p, σ \rightarrow p, σ′$.

The states of our v4 semantics are $Φ_4$. i.e., stacks of speculative instances $Φ_4$, where the topmost instance of the state is used to execute the instruction. States are stacks because speculation pushes a new state on top of the stack, which is popped when speculation ends. Each instance contains the program $p$, a counter $ctr$ that uniquely identifies the speculation instance, a configuration $σ$, and the speculation window $ω$, describing the amount of instructions possible during speculation. Speculation is modeled by mispredicting every store instruction, i.e., it is skipped (Rule AM-Store).

The rule pushes a new instance with configuration $σ''$, that skips the store instruction: this models speculatively skipping a store, because it models the effect of the memory disambiguator mispredicting the matching address between this store instruction and future load instructions, which results in loading stale values. When speculation ends, the instance used for speculation is popped and execution continues with the old instance $σ'$, which is calculated according to the non-speculative semantics. The behaviour of a program $p$ is:

$$Beh^4(p) = \{ τ \mid \forall σ \in InitConf. (p, σ) \xrightarrow{τ} Φ_4 \}$$

where, with a slight abuse of notation, $(p, σ) \xrightarrow{τ} Φ_4$ indicates the execution of program $p$ until completion, while generating the trace $τ$ (i.e., a list of observable actions).

To create the combined semantics v14, we define its states as the union of the states of the SPECTRE V1 and SPECTRE V4 semantics. We extend our trace model with tags $t = \{ v1, v4 \}$ for the start id, and rollback id, observations to mark the origin of the speculative transaction they were generated from.

We define projection functions $\uparrow^1: Φ_14 \rightarrow Φ_4$ and $\uparrow^4: Φ_14 \rightarrow Φ_4$ that extract the corresponding state from the combined state. We overload them to work on traces as well:

$$\varepsilon^1 = \varepsilon \quad (\uparrow^1 \cdot τ)^1 = τ \cdot (\uparrow^4)^1$$

The projection on traces deletes all speculative transactions (marked by start id and rollback id) that are not generated by the corresponding semantics that we project to.

The rules of the combined semantics (Rule AM-v1-step AM-v4-step) use the projection functions to extract the corresponding state and delegate to the semantics of $Φ_14$ and $Φ_4$ to make a step. This delegation allows us to reuse proofs about $Φ_1$ and $Φ_4$ in the proofs for the combined semantics.

We note that the combined v14 semantics is technically not deterministic, but is confluent for single steps. We now have everything to relate the combined v14 semantics to its parts:

**Theorem 1.** Let $p$ be a program and $ω$ be a speculation window. Then $Beh^4(p) = Beh^1(p)^1$.

**Theorem 2.** Let $p$ be a program and $ω$ be a speculation window. Then $Beh^1(p) = Beh^4(p)^1$.

We use Speculative Non-Interference (SNI) as the security condition to show security of programs, and we prove that SNI of the combined semantics implies SNI of the individual ones. Note that the inverse is not true, SNI of the individual semantics (V1 and V4) does not imply SNI of the combined ones (v14). An example is our snippet Listing 3 which is SNI under SPECTRE V1 and SPECTRE V4 in isolation, but is not SNI (and it is not secure) under SPECTRE V14.

**II. Implementation**

We implemented our semantics in SPECTECTOR by Guarnieri et al. [5] and validated our extension on the test suite for SPECTRE V4 by Daniel et al. [6] (Table 1).

<table>
<thead>
<tr>
<th>Test case</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>case01 (-)</td>
<td>✓</td>
</tr>
<tr>
<td>case02 (-)</td>
<td>✓</td>
</tr>
<tr>
<td>case03 (+)</td>
<td>✓</td>
</tr>
<tr>
<td>case04 (-)</td>
<td>✓</td>
</tr>
<tr>
<td>case05 (-)</td>
<td>✓</td>
</tr>
<tr>
<td>case06 (-)</td>
<td>✓</td>
</tr>
<tr>
<td>case07 (-)</td>
<td>✓</td>
</tr>
<tr>
<td>case08 (-)</td>
<td>✓</td>
</tr>
<tr>
<td>case09* (+)</td>
<td>✓</td>
</tr>
<tr>
<td>case10 (-)</td>
<td>✓</td>
</tr>
<tr>
<td>case11 (-)</td>
<td>✓</td>
</tr>
<tr>
<td>case12 (+)</td>
<td>✓</td>
</tr>
<tr>
<td>case13(-)</td>
<td>✓</td>
</tr>
</tbody>
</table>

TABLE 1: Result of the litmus test cases for SPECTRE V4. The expected results are SAFE (+) or UNSAFE (-) w.r.t SPECTRE V4. A ✓ represents that our tool correctly classifies the test case. The * represents that the speculation window $ω$ was reduced, because of state explosion. For the combined attacks, there are no existing benchmarks we can use. That is why we analyse our snippet with the combined v14 semantics (last row).

The results show that our semantics correctly classifies all the test cases.

Following the discussion of tests2 and Ponce-de León and Kinder 7 approach, we marked this test case as unsafe, because we assume all initial values to be secret
REFERENCES


