Formal Verification of Combined Spectre Attacks

Xaver Fabian[1] Koby Chan[2] Marco Guarnieri[3] Marco Patrignani[1]

[1]CISPA Helmholz Center for Information Security [2]Stanford [3]Imdea 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.

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

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

The scientific community came up with different tools to verify that a program is resistant against Spectre attacks. One branch of these solutions focuses on operational semantics to build program verification tools. However, these verification tools concentrate on specific Spectre variants in isolation, while in practice, processors are using different prediction mechanisms in combination at the same time. Unfortunately, there is no way to verify the absence of Spectre attacks that use multiple predictors at the same time. Furthermore, to scale the verification of such combined Spectre attacks, it is desirable to compose existing solutions together into the combined approach, instead of devising ad-hoc semantics (and tools) for specific variants. A combined approach would allow maximising the reuse of the verification (and proofs) of the already existing solution for the verification (and proofs) of the new combined attacks.

In addition, there are not many semantics for verification of programs vulnerable to Spectre variants different from Spectre v1. For example, one of the Spectre attacks we want to focus on is Spectre v4 [4], because it abuses a different prediction mechanism. Spectre v4 exploits the prediction mechanism behind store-to-load forwarding. To gain speedup, modern processors employ a store queue and they will put `store` instructions into the store queue until they are committed to the main memory. When a `load` instruction is executed, the processor will look first in the store queue for a matching `store`. This lookup can be slow, so the CPU employs a memory disambiguation predictor that guesses if the address of the `load` matches with the address of the `store`. In Listing 2, a misprediction for the `load` instruction in Line 3 causes it to take its value from the stale `store` instruction in Line 1. The speculative access of the memory is then leaked into the microarchitectural state by the array access into `B`.

```
1 p = &secret;
2 p = &public;
3 temp = B[*p * 512];
```

Listing 2: Spectre v4 example.

Now with Spectre v4, we can give an example of a novel attack (Listing 3) that exploits the prediction mechanisms behind Spectre v1 and Spectre v4 combined.

```
1 x = 0;
2 p = &secret;
3 p = &public;
4 if (x != 0)
5     temp &= A[*p];
```

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.

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 Listing 3 and we prove that (1) the combined semantics is 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 $p, \sigma \xrightarrow{\tau} p, \sigma'$.

The states of our v4 semantics are $\overline{\Phi}_4$, 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).

$$\boxed{\text{V4 semantics judgement: } \Phi_4 \overset{\tau}{\rightsquigarrow}_4 \overline{\Phi}'_4}$$

(AM-Store)

$$\frac{\begin{array}{c} p(\sigma(\mathbf{pc})) = \mathbf{store}\ x, e \qquad (p,\sigma) \xrightarrow{\tau} (p,\sigma') \qquad j = min(\omega, n) \\ \sigma'' = \sigma[\mathbf{pc} \mapsto \sigma(\mathbf{pc}) + 1] \qquad \tau' = \tau \cdot \texttt{skip}\ ctr \cdot \texttt{start}\ ctr \end{array}}{\langle p, ctr, \sigma, n+1 \rangle \overset{\tau'}{\rightsquigarrow}_4 \langle p, ctr, \sigma', n \rangle \cdot \langle p, ctr+1, \sigma'', j \rangle}$$

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:

$$Beh^4(p) = \{\overline{\tau} \mid \forall \sigma \in InitConf.\ (p,\sigma) \overset{\overline{\tau}}{\rightsquigarrow}_4 \_\}$$

where, with a slight abuse of notation, $(p,\sigma) \overset{\overline{\tau}}{\rightsquigarrow}_4 \_$ indicates the execution of program p until completion, while generating the trace $\overline{\tau}$ (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_t and `rollback` id_t observations to mark the origin of the speculative transaction they were generated from.

We define projection functions $\restriction^4 \colon \Phi_{14} \rightarrow \Phi_4$ and $\restriction^1 \colon \Phi_{14} \rightarrow \Phi_1$ that extract the corresponding state from the combined state. We overload them to work on traces as well:

$$\varepsilon\restriction^4 = \varepsilon \qquad (\tau \cdot \overline{\tau})\restriction^4 = \tau \cdot (\overline{\tau})\restriction^4$$
$$\texttt{start}_{v1}\ id \cdots \texttt{rollback}_{v1}\ id \cdot \overline{\tau}\restriction^4 = \overline{\tau}\restriction^4$$

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 $\rightsquigarrow_1$ and $\rightsquigarrow_4$ to make a step. This delegation allows us to reuse proofs about $\rightsquigarrow_1$ and $\rightsquigarrow_4$ in the proofs for the combined semantics.

$$\boxed{\text{V14 semantics judgement: } \Phi_{14} \overset{\tau}{\rightsquigarrow}_{14} \overline{\Phi}'_{14}}$$

(AM-v4-step) (AM-v1-step)

$$\frac{\Phi_{14}\restriction^4 \overset{\tau}{\rightsquigarrow}_4 \overline{\Phi}'_{14}\restriction^4}{\Phi_{14} \overset{\tau}{\rightsquigarrow}_{14} \overline{\Phi}'_{14}} \qquad \frac{\Phi_{14}\restriction^1 \overset{\tau}{\rightsquigarrow}_1 \overline{\Phi}'_{14}\restriction^1}{\Phi_{14} \overset{\tau}{\rightsquigarrow}_{14} \overline{\Phi}'_{14}}$$

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 $Beh^4(p) = Beh^{14}(p)\restriction^4$.*

Theorem 2. *Let p be a program and ω be a speculation window. Then $Beh^1(p) = Beh^{14}(p)\restriction^1$.*

We use Speculative Non-Interference (SNI [5]) 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 I).

Test case	Correct
case01 (-)	✓
case02 (-)	✓
case03 (+)	✓
case04 (-)	✓
case05 (-)	✓
case06 (-)	✓
case07 (-)	✓
case08 (-)	✓
case09* (+)	✓
case10 (-)	✓
case11 (-)	✓
case12 (+)	✓
case13[1] (-)	✓
Listing 3 (-)	✓

TABLE I: 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.

[1] Following the discussion of https://github.com/binsec/haunted_bench/issues/2 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

[1] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[2] G. Maisuradze and C. Rossow, "Ret2spec: Speculative execution using return stack buffers," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 2109–2122. [Online]. Available: https://doi.org/10.1145/3243734.3243761

[3] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. Baltimore, MD: USENIX Association, Aug. 2018. [Online]. Available: https://www.usenix.org/conference/woot18/presentation/koruyeh

[4] J. Horn, "Speculative execution, variant 4: Speculative store bypass," https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2018, accessed: 2021-04-11.

[5] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "Spectector: Principled detection of speculative information flows," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1–19.

[6] L. Daniel, S. Bardin, and T. Rezk, "Hunting the haunter - efficient relational symbolic execution for spectre with haunted relse," in *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/hunting-the-haunter-efficient-relational-symbolic-execution-for-spectre-with-haunted-relse/

[7] H. Ponce-de León and J. Kinder, "Cats vs. spectre: An axiomatic approach to modeling speculative execution attacks," 2021.