# Internship on a Type System for Formally Verified Spectre Mitigations

Advisors: Catalin Hritcu and Jonathan Baumann, MPI-SP, Bochum, Germany

November 6, 2026

## Introduction

The Spectre speculative side-channel attacks are a formidable threat for the security of software systems [5, 14]. Without any defenses, attackers can easily construct "universal read gadgets" that leak a sensitive program's entire memory [14, 17]. Yet strong and efficient Spectre defenses are notoriously challenging to implement. Even the Linux kernel, for which many developers have spent years to implement effective Spectre defenses, still sometime falls pray to Spectre attacks, including universal read gadgets [22, 23], and the situation seems worse when developers have fewer resources and rely entirely on the insufficient defenses currently provided by the OSes and compilers. Some of these attacks exploit new Spectre variants that current defenses don't deal with [11, 12, 24], but other attacks simply exploit implementation mistakes, bad security-efficiency tradeoffs, and unclear and overly optimistic assumptions that developers often make when implementing Spectre defenses, which often involve a complex combination of software in compilers and OSes as well as hardware features and models, which are all very difficult to get right.

We believe that formally secure compilation can play a central role in building watertight Spectre defenses that are strong and fit well together. This will raise the security bar and go beyond increasing attacker effort to achieving an end-to-end security notion with respect to mathematically specified attacker models, which guarantees the elimination of entire classes of Spectre attacks. Our recent work [2, 3] has made advances in this direction, providing the first machine-verified proofs for SLH [6] and Ultimate SLH [27], and designing new mitigations with formal security guarantees. However, we suspect that our current proofs are more complicated than necessary, and that a specially designed type system could serve as a useful abstraction layer to simplify both existing and future proofs.

**Background** Essentially all modern processors execute instructions out-of-order and in parallel. This is essential to their performance, since instructions can stall for a long time while waiting for e.g. data loaded from memory. Additionally, they also employ a variety of *prediction mechanisms* for control flow instructions such as conditional branches and indirect calls. These predict the outcome, allowing the processor to *speculatively* begin executing instructions before the control flow instruction has fully resolved. If the prediction then turns out to be incorrect, the *speculative* instructions will be squashed and should not affect the sequential behaviour of the program [19]. The Spectre class of attacks arises because speculatively executed instructions still leave traces in the microarchitecture, most notably the cache state, which are not reset when the misspeculation is detected [5, 14]. The resulting microarchitectural state can then be extracted by attackers, for example via cache attacks [25, 26]. The resulting attacks are very powerful, as they can bypass sequential safeguards [13, 14] or even completely hijack control flow [4, 14, 15, 16].

We are mainly interested in mitigations based on *Speculative Load Hardening (SLH)* [6], which follow a general approach of detecting misspeculation by accumulating control flow conditions in a special register called the *misspeculation flag*, and using this register to conditionally mask the operands of vulnerable instructions (those which leave detectable traces in the microarchitectural state, most notably load and

store instructions). Important variants include *Selective SLH* [20, 21], which leverages security annotations in cryptographic code to introduce masking only if the leaked value is secret, and *Ultimate SLH* [27], which takes a much more thorough approach to masking that also includes branch conditions and variable-time instructions. Our own work, *Flexible SLH* [2], is a generalization of these two approaches with machine-checked security proofs.

Previous work has taken a variety of approaches to defining formal security properties for programs under speculative execution [8]. The most relevant ones are *speculative constant-time*, which requires a program to leak no secrets during *sequential or speculative* execution and is commonly used for cryptographic code [1, 7], and *speculative non-interference*, which requires a program to not leak *more* during speculative execution than it does sequentially [9, 10]. The latter definition is usually better suited to verifying general-purpose mitigations, as it is generally difficult to enforce the absence of sequential leakage. In contrast, our own work uses a different notion, which is especially suited to compiler mitigations: *Relative security* [2, 3], which requires that the *compiled* program does not leak more (sequentially or speculatively) than the *source* program does under sequential semantics. This definition mainly rules out the pathological case of mitigations introducing additional sequential leakage in order to achieve speculative non-interference.

To simplify proofs for compiler-based countermeasures, Patrignani and Guarnieri [18] developed *speculative safety*, a single-trace property that overapproximates speculative non-interference. It relies on dynamic taint tracking in an instrumented semantics, treating values as *safe* if they would leak sequentially, and *unsafe* otherwise. The advantage of this technique is that preservation of safety properties is generally easier to prove than preservation of hyperproperties, however, this technique has not generally caught on. We nevertheless believe this technique deserves another look: Our hyperproperty proofs for FlexSLH and Triosecuris turned out laborious, yet the optimizations we want to verify are justified by facts about single executions (a value has already been masked) or by static types (a value is public). Such facts are exactly what taint tracking records, and the ideal semantics in our proof of FlexSLH already tracks information-flow labels dynamically. The relational reasoning is then needed only once, to prove that speculative safety implies speculative non-interference.

Shivakumar et al. [21] have developed a type system specifically designed to verify the implementation of selective speculative load hardening in Jasmin. Two core ideas of this type system is that it tracks sequential and speculative types separately, and that it can track whether the misspeculation flag accurately represents the whether execution is currently misspeculating. Unfortunately however, this type system imposes a restriction that the speculative taint of a value may never be lower than the sequential one, which makes this type system unsuitable for some proofs we would like to conduct in the future.

## Internship topic

The goal of this internship is to design a type system for speculative security that tracks sequential and speculative taints separately, to use it to enforce speculative safety, and to prove in Rocq that it guarantees speculative non-interference. Such a type system should make the security proofs of compiler mitigations simpler than our current proofs, since preserving a safety property is easier than preserving a hyperproperty. It should also cover optimizations that the existing formal models do not capture, such as LLVM's reuse of already masked values in LLVM SLH and Ultimate SLH, where a value may hold a secret sequentially yet be safe to leak speculatively. The internship builds on the existing Rocq development of FlexSLH [2], which provides the speculative semantics, the hardening and its security proofs. It has the following tasks.

**Task 1:** Lift the above-mentioned restriction of the type system of Shivakumar et al. [21], use it to enforce speculative safety, and prove (in Rocq) that speculative safety implies speculative noninterference. Specifically, we will define speculative safety semantically (in the style of Patrignani and Guarnieri), and then design the type system to enforce it. This will allow us to refine the type system iteratively without moving targets, and without needing to adapt the proof of speculative noninterference.

We expect memory to be the delicate part, since a type system must approximate which locations hold masked values; the array-based language of FlexSLH keeps this manageable.

Optionally, also add a *declassify* primitive, which Shivakumar et al. [21] omitted from their proofs.

**Task 2:** Use the type system to prove speculative safety of FlexSLH [2], thus obtaining speculative non-interference, which can be combined with a simple sequential security proof to obtain relative security. This exercises the type system on a hardening whose security is already proved, before applying it to the optimizations of Tasks 3 and 4.

**Task 3:** Use speculative safety to prove the security of an optimization implemented in practice by LLVM SLH and Ultimate SLH [27], but not yet formally verified: While the existing formal models [2, 3] always mask all operands of every vulnerable instruction, the LLVM implementation tracks which values have been masked before, and reuses previously masked values instead of masking them again. This optimization is the main motivator for this type system, as the masked values may sequentially contain secrets, but be safe to leak during speculative execution (as the value will have been masked).

**Task 4:** Implement in Rocq and prove secure this same optimization for FlexSLH, making full use of the ability to track both sequential and speculative taints.

**Task 5:** Use this type system to design further optimizations or other hardening strategies. For example, the type system could check a program that the developer has partially hardened by hand, so that programs can be optimized manually without compromising formal security guarantees.

*Required skills:* Rocq, functional programming, programming language semantics, type systems.
*Acquired skills:* Spectre attacks and defenses, speculative load hardening, information-flow type systems, machine-checked proofs of security properties.

## References


[1] G. Barthe, S. Cauligi, B. Grégoire, A. Koutsos, K. Liao, T. Oliveira, S. Priya, T. Rezk, and P. Schwabe. High-assurance cryptography in the Spectre era. *IEEE SP*. 2021.

[2] J. Baumann, R. Blanco, L. Ducruet, S. Harwig, and C. Hritcu. FSLH: flexible mechanized speculative load hardening. *CSF*. 2025.

[3] J. Baumann, Y. Kim, Y. Farba, C. Hritcu, and J. Leatherman-Brooks. Triosecuris: Formally verified protection against speculative control-flow hijacking. CSF'26, 2026.

[4] A. Bhattacharyya, A. Sánchez, E. M. Koruyeh, N. B. Abu-Ghazaleh, C. Song, and M. Payer. SpecROP: Speculative exploitation of ROP chains. *RAID*. 2020.

[5] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. A systematic evaluation of transient execution attacks and defenses. *USENIX Security*. 2019.

[6] C. Carruth. Speculative Load Hardening: A Spectre variant #1 mitigation technique. LLVM Reference, 2018.

[7] S. Cauligi, C. Disselkoen, K. von Gleissenthall, D. M. Tullsen, D. Stefan, T. Rezk, and G. Barthe. Constant-time foundations for the new Spectre era. *PLDI*. 2020.

[8] S. Cauligi, C. Disselkoen, D. Moghimi, G. Barthe, and D. Stefan. SoK: Practical foundations for software Spectre defenses. *IEEE SP*. 2022.

[9] K. Cheang, C. Rasmussen, S. A. Seshia, and P. Subramanyan. A formal approach to secure speculation. *CSF*. 2019.

[10] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez. Spectector: Principled detection of speculative information flows. *IEEE SP*. 2020.

[11] J. Kim, J. Chuang, and D. Genkin. FLOP: Breaking the Apple M3 CPU via false load output predictions. To appear in USENIX Security, 2025.

[12] J. Kim, D. Genkin, and Y. Yarom. SLAP: data speculation attacks via load address prediction on Apple silicon. *IEEE SP*. 2025.

[13] V. Kiriansky and C. A. Waldspurger. Speculative buffer overflows: Attacks and defenses. *CoRR*, abs/1807.03757, 2018.

[14] 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. *IEEE SP*. 2019.

[15] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. B. Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. *WOOT*. 2018.

[16] G. Maisuradze and C. Rossow. ret2spec: Speculative execution using return stack buffers. *CCS*. 2018.

[17] R. McIlroy, J. Sevcík, T. Tebbi, B. L. Titzer, and T. Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *CoRR*, abs/1902.05178, 2019.

[18] M. Patrignani and M. Guarnieri. Exorcising Spectres with secure compilers. *CCS*. 2021.

[19] F. Piessens. Transient execution attacks. *IEEE Secur. Priv.*, 21(5), 2023.

[20] B. A. Shivakumar, J. Barnes, G. Barthe, S. Cauligi, C. Chuengsatiansup, D. Genkin, S. O'Connell, P. Schwabe, R. Q. Sim, and Y. Yarom. Spectre declassified: Reading from the right place at the wrong time. *IEEE SP*. 2023.

[21] B. A. Shivakumar, G. Barthe, B. Grégoire, V. Laporte, T. Oliveira, S. Priya, P. Schwabe, and L. Tabary-Maujean. Typing high-speed cryptography against Spectre v1. *IEEE SP*. 2023.

[22] S. Wiebing and C. Giuffrida. Training Solo: On the limitations of domain isolation against Spectre-v2 attacks. *IEEE SP*. 2025.

[23] S. Wiebing, A. de Faveri Tron, H. Bos, and C. Giuffrida. InSpectre Gadget: Inspecting the residual attack surface of cross-privilege Spectre v2. *USENIX Security*. 2024.

[24] J. Wikner, D. Trujillo, and K. Razavi. Phantom: Exploiting decoder-detectable mispredictions. *MICRO*. 2023.

[25] Y. Yarom. *Microarchitectural Attacks*. Wiley-ISTE, 2025.

[26] Y. Yarom and K. Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache Side-Channel attack. *USENIX Security*. 2014.

[27] Z. Zhang, G. Barthe, C. Chuengsatiansup, P. Schwabe, and Y. Yarom. Ultimate SLH: Taking speculative load hardening to the next level. *USENIX Security*. 2023.