# FSLH: Flexible Mechanized Speculative Load Hardening

Jonathan Baumann[1,2] Roberto Blanco[1,3] Léon Ducruet[1,4,5] Sebastian Harwig[1,6] Cătălin Hriţcu[1]

[1]MPI-SP, Bochum [2]ENS Paris-Saclay [3]TU Eindhoven [4]ENS Lyon [5]Aarhus University [6]Ruhr University Bochum

***Keywords:*** side-channel defenses, speculative execution, Spectre, secure compilation, speculative load hardening, relative security, formal verification, Rocq, Coq

## 1 Background

Speculative side-channel attacks such as Spectre pose formidable threats for the security of computer systems [5, 12]. For instance, in typical Spectre v1 attacks [11], misspeculated array bounds checks cause out-of-bounds memory accesses to load and reveal secrets via timing variations. SLH [6] is a software countermeasure against such attacks that dynamically tracks whether execution is in a mispredicted branch using a misspeculation flag register.

It is, however, challenging to build software protections that are both efficient and that provide formal end-to-end security guarantees against precisely specified, speculative side-channel attacker models [7]. Cryptography researchers are leading the way in this space, with defenses such as *selective* SLH efficiently achieving speculative constant-time guarantees against Spectre v1 for cryptographic code with typical overheads under 1% [14, 15]. This work is, however, specialized to only cryptographic code and often also to domain-specific languages for cryptography, such as Jasmin [1].

It is more difficult to properly protect arbitrary programs written in general-purpose languages, which in particular do not obey the cryptographic constant-time discipline [2, 8]. SLH was not strong enough for protecting such non-cryptographic code [13], leading to the introduction of Ultimate SLH [13, 16], which uses the misspeculation flag to mask not only the values loaded from memory, but also all branch conditions, all memory addresses, etc. While this should in principle be strong enough to achieve a relative security notion [7, 13, 16], it also brings ~150% overhead on the SPEC benchmarks [16], which seems unacceptable for many practical scenarios.

## 2 Contribution

In recent work [4], we therefore introduce FSLH, a flexible SLH notion that achieves the best of both worlds by generalizing both Selective and Ultimate SLH. Like Selective SLH, FSLH keeps track of which program inputs are secret and which ones not and only protects those memory operations that could potentially leak secret inputs when ran speculatively. Like Ultimate SLH, FSLH also provides protection for non-cryptographic code that does not respect the constant-time discipline, but does this by only masking those observable expressions (e. g. branch conditions) that are potentially influenced by secrets.

We give a suitable definition of relative security for transformations protecting arbitrary programs, which states that the transformed program running with speculation should not leak more than what the source program leaks sequentially. We formally prove in Rocq (previously knows as Coq) that FSLH enforces this relative security notion. Finally, we also prove that FSLH can be instantiated to recover both Selective and Ultimate SLH, therefore obtaining mechanized security proofs for both versions.

## 3 The FSLH Transformation

The FSLH mitigation builds on the design of Selective SLH [14], which in turn builds on SLH [6], a program transformation combining

1. a mechanism to keep track of misspeculation in a special *misspeculation flag*, and
2. a means of preventing leakage by masking based on this misspeculation flag.

The former mechanism is shared between Selective SLH, Ultimate SLH and FSLH and relies on using branchless conditional expressions after every branch to update the misspeculation flag with the condition used for the branch, so that even if the branch is mispredicted, the flag will be updated correctly. The latter can take several forms, including *address hardening*, where the memory addresses for load instructions are masked to zero if the misspeculation flag is set, and *value hardening*, where the loaded value will be masked to zero instead.

Selective SLH [14] is a refinement of SLH that applies only to code satisfying the *cryptographic constant-time (CCT) discipline*, which is standard practice in the field of cryptography [2, 3, 8, 9] and requires that

1. program inputs are identified as public or secret,
2. control flow and memory addresses are not allowed to depend on secrets.

This discipline is easily enforceable with a type system, which Selective SLH [14] leverages to *selectively* apply value hardening only to loads to *public* variables, as the CCT discipline already guarantees that there is no leakage from *secret* variables.

For FSLH, we substitute the CCT type system with a sound and fully permissive *flow-sensitive* static analysis similar to the algorithmic version of Hunt and Sands's [10] flow-sensitive type system. This analysis only tracks which expressions and variables contain secrets, but does not restrict the use of secret-dependent expressions in e. g. branch conditions. While our analysis is sound, it is necessarily imprecise, as information on control flow is not available statically, so the analysis safely overapproximates which variables may contain secrets after branches and loops. Importantly however, this analysis accepts *arbitrary* programs, and imposes no restrictions.

Unlike Selective SLH, which only protects loads to public variables, FSLH thus has to deal with more situations that may produce leakage:

1. Branch conditions may now be secret, and therefore require masking to prevent leaking secrets via control flow, like in Ultimate SLH [16]. This is achieved by modifying affected branch conditions to additionally check that the misspeculation flag is not set, therefore ensuring that these conditions will evaluate to `false` during misspeculated execution without affecting sequential execution. Unlike Ultimate SLH, we only apply this masking if the branch condition is secret.
2. Memory addresses can now also be secret, in which case we apply address hardening instead of value hardening, so that the address is also protected.

## 4 Machine-checked Relative Security Proof

We prove in Rocq that FSLH ensures a notion of *Relative Security*, guaranteeing that the hardened program does not leak *more* during speculative execution than the source program does sequentially. More formally, it states that for any pair of initial states which agree on the values of all public variables and for which the attacker-visible observations produced during sequential execution of the source program are equal, the observations produced by the hardened program during speculative execution will also be equal.

Taking inspiration from Shivakumar et al. [14], the security proof decomposes into a compiler correctness proof for an *ideal semantics* and a separate relative security proof for this semantics. Our ideal semantics operates on *annotated programs*, where expressions are marked as either public or secret according to the static analysis described above. Like the target semantics, the ideal semantics features speculative execution, however, it masks values directly in the semantics according to the annotations. Therefore, the *source* program in the ideal semantics behaves the exact same way as the *target* program in the speculative semantics, producing the same attacker-visible observations. We formally establish this using a backwards compiler correctness proof.

Thus, the proof of relative security is reduced to proving relative security between the ideal semantics and the sequential semantics. For this, we further instrument the ideal semantics with *sound* dynamic information flow tracking. Relative security then follows from the following three facts:

1. Execution along the correct path in the ideal semantics is the same as sequential execution, and produces the same attacker-visible observations.
2. Execution in the ideal semantics preserves agreement on public values, i. e. if we execute the same program on two initial states that agree on all variables that are initially considered public, then the resulting states will agree both on the labelings computed by the dynamic tracking, and on the values of all variables labeled public.
3. During mispredicted execution, all observable expressions that are not public are masked. [1]

In order to prove the last point, we introduce a *well-labeledness* predicate that describes whether annotations in a program are sound with respect to an initial and a final labeling. It does not require the labels to be precise, i. e. an expression which is actually public is allowed to be labeled as secret, but not the other way round. Importantly, this means that well-labeledness is preserved when *strengthening* the initial labeling (changing a label from secret to public) and when *weakening* the final labeling (changing a label from public to secret), which allows us to prove that execution in the ideal semantics preserves well-labeledness, in the sense that after executing a step of a well-labeled program in the ideal semantics, the remaining program is well labeled for the initial labeling *updated by the dynamic tracking* and the original final labeling.

Since the ideal semantics preserves well-labeledness, we know that during mispredicted execution, all secret values will be masked, so the attacker-visible observations depend only on the values of public variables. Since the ideal semanics further preserves agreement on public variables, we obtain that all observations produced during misspeculated execution must be the same for both executions. Since the non-misspeculated parts of the ideal executions correspond exactly to the sequential executions, which produce equal observations, we obtain relative security of the ideal semantics with respect to the sequential semantics.

## 5 Conclusion

We have introduced FSLH, a mitigation which generalizes Selective SLH and Ultimate SLH to achieve the best of both worlds: Like Ultimate SLH, it achieves relative security to all programs, without restrictions to any particular discipline; like Selective SLH, it is efficient by only inserting protections where needed. We formally prove in Rocq that FSLH ensures relative security. Since FSLH generalizes Selective SLH and Ultimate SLH, we also obtain machine-checked security proofs for both these mitigations as corollaries.

[1] This is highly similar to a lemma in the security proof of Ultimate SLH [16], for which the observations only depend on the structure of the program.

## References

[1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-Assurance and High-Speed Cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 1807–1823. doi:10.1145/3133956.3134078

[2] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *25th USENIX Security Symposium*, Thorsten Holz and Stefan Savage (Eds.). USENIX Association, 53–70. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida

[3] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Luna, and David Pichardie. 2020. System-Level Non-interference of Constant-Time Cryptography. Part II: Verified Static Analysis and Stealth Memory. *J. Autom. Reason.* 64, 8 (2020), 1685–1729. doi:10.1007/S10817-020-09548-X

[4] Jonathan Baumann, Roberto Blanco, Léon Ducruet, Sebastian Harwig, and Catalin Hritcu. 2025. FSLH: Flexible Mechanized Speculative Load Hardening. In *38th IEEE Computer Security Foundations Symposium, CSF 2025, Santa Cruz, CA, USA, June 16-20, 2025.* IEEE, 569–584. doi:10.1109/CSF64896.2025.00023

[5] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *28th USENIX Security Symposium*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 249–266. https://www.usenix.org/conference/usenixsecurity19/presentation/canella

[6] Chandler Carruth. 2018. Speculative Load Hardening: A Spectre Variant #1 Mitigation Technique. LLVM Reference. https://llvm.org/docs/SpeculativeLoadHardening.html

[7] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. 2022. SoK: Practical Foundations for Software Spectre Defenses. In *43rd IEEE Symposium on Security and Privacy, SP 2022.* IEEE, 666–680. doi:10.1109/SP46214.2022.9833707

[8] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. 2019. FaCT: a DSL for timing-sensitive computation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 174–189. doi:10.1145/3314221.3314605

[9] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2023. Binsec/Rel: Symbolic Binary Analyzer for Security with Applications to Constant-Time and Secret-Erasure. *ACM Trans. Priv. Secur.* 26, 2 (2023), 11:1–11:42. doi:10.1145/3563037

[10] Sebastian Hunt and David Sands. 2006. On flow-sensitive security types. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, J. Gregory Morrisett and Simon L. Peyton Jones (Eds.). ACM, 79–90. doi:10.1145/1111037.1111045

[11] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy, SP.* IEEE, 1–19. doi:10.1109/SP.2019.00002

[12] Ross McIlroy, Jaroslav Sevcík, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. 2019. Spectre is here to stay: An analysis of side-channels and speculative execution. *CoRR* abs/1902.05178 (2019). arXiv:1902.05178 http://arxiv.org/abs/1902.05178

[13] Marco Patrignani and Marco Guarnieri. 2021. Exorcising Spectres with Secure Compilers. In *2021 ACM SIGSAC Conference on Computer and Communications Security, CCS*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 445–461. doi:10.1145/3460120.3484534

[14] Basavesh Ammanaghatta Shivakumar, Jack Barnes, Gilles Barthe, Sunjay Cauligi, Chitchanok Chuengsatiansup, Daniel Genkin, Sioli O'Connell, Peter Schwabe, Rui Qi Sim, and Yuval Yarom. 2023. Spectre Declassified: Reading from the Right Place at the Wrong Time. In *44th IEEE Symposium on Security and Privacy, SP.* IEEE, 1753–1770. doi:10.1109/SP46215.2023.10179355

[15] Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Swarn Priya, Peter Schwabe, and Lucas Tabary-Maujean. 2023. Typing High-Speed Cryptography against Spectre v1. In *44th IEEE Symposium on Security and Privacy, SP.* IEEE, 1094–1111. doi:10.1109/SP46215.2023.10179418

[16] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. 2023. Ultimate SLH: Taking Speculative Load Hardening to the Next Level. In *32nd USENIX Security Symposium*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association, 7125–7142. https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-zhiyuan-slh