Internships on Property-Based Testing of LLVM for Side-Channel Leakage

Advisors: Catalin Hritcu and Yonghyun Kim, MPI-SP, Bochum, Germany

November 25, 2026

Summary

We will build RS-Test-LLVM, the first framework for rigorously testing whether compilation introduces timing leakage, against a formal leakage semantics, for arbitrary C/C++ and Rust programs. A compiler that preserves every output can still introduce such leakage, and LLVM does so in practice, turning branchless idioms into branches. Yet LLVM is not checked for it. We will test LLVM's optimization pipelines, from its IR passes down to its x86 backend, for *relative security*: the translated program leaks no more than the original. Most testing will run programs in executable formal semantics. For LLVM IR this is Vellvm, a Rocq semantics that we are extending with leakage. For the x86 machine IR of LLVM's backend, we will build a leakage semantics in the Lean proof assistant. For the Spectre defenses we are adding to LLVM, the translated program must leak no more when run speculatively than the original does when run sequentially, which we will test on the compiled x86 binaries with the Revizor testing tool. The project will deliver a leakage semantics for each IR, a catalogue of leakage-introducing LLVM passes, a validation of our Spectre defenses, and case studies testing LLVM's pipelines on crypto and systems code. By catching leaks before they reach production, RS-Test-LLVM is also a step towards a variant of LLVM that introduces no leakage of its own and whose Spectre defenses protect arbitrary programs, cryptographic and non-cryptographic.

Introduction

Verified compilation usually means preserving the source program's *functional* behavior. For code that handles secrets this is not enough: a compiler can preserve every output and still introduce *timing leakage* through branches, memory addresses, or variable-time operations that depend on secrets, and such leaks are practical to exploit, even remotely [10]. Cryptographic libraries avoid all three by the constant-time (CT) discipline, and a few specialized compilers are proved to preserve CT, such as a variant of the verified CompCert C compiler [3] and the Jasmin compiler [4]. But CT is a discipline for a small body of expert crypto code; other code that handles secrets cannot expect a translation to make it CT, but it can ask that the translation introduce no new leakage.

Translations in mainstream compilers do introduce leakage though. Recent work showed LLVM's middle end rewriting branchless idioms (such as (m & a) | (~m & b), for a mask m of all ones or all zeros) into select instructions (c ? a : b) that some backends then lower to branches [11, 17, 26, 32]. LLVM's x86 backend was also seen turning conditional-move instructions (cmov) back into branches [11]. Recently, LLVM was also found to compile the CT mask arithmetic of the Rust cmov crate into a branch on 32-bit ARM [25]. These leaks were found in compiled code after the fact, not by systematically testing the compiler's transformations for the leaks they introduce. Speculative execution adds leakage of its own: without defenses, Spectre attacks can leak a program's entire memory [16, 20]. Defenses, often inserted by compilers, are notoriously hard to get right, so that even the Linux kernel, after years of work on its defenses, still falls prey to such attacks [30, 31].

We propose to make the absence of introduced timing leakage, *for every source program*, a security re-

quirement on translations, and to test it on LLVM and on our Spectre defenses. Our formal requirement is *relative security*: *the translated program should leak no more than the original*. We write $\mathcal{L}_P(i)$ for the possibly infinite sequence of leakage events, such as branch conditions and memory addresses, that P produces on input i. A translation of S into T is relatively secure if, for all inputs i_1, i_2 on which S has no undefined behavior, whenever $\mathcal{L}_S(i_1)$ and $\mathcal{L}_S(i_2)$ are equal or one is a prefix of the other, so are $\mathcal{L}_T(i_1)$ and $\mathcal{L}_T(i_2)$.[1] As in our earlier work [6, 7], traces may be infinite, so nonterminating runs count. For a Spectre defense, $\mathcal{L}_T(i)$ also includes what T leaks under speculative execution, with the same attacker-chosen mispredictions in the runs on i_1 and i_2. Relative security puts no CT constraints on the source and declares no secret in advance: whatever S's leakage does not reveal, T's must not reveal either. We previously proved two Spectre defenses relatively secure in Rocq, for simplified languages. The first, flexible speculative load hardening [6], defends against mispredicted conditional branches and received a Distinguished Paper Award at CSF'25. The second, Triosecuris [7], also covers indirect branches and returns, and we are now implementing it in LLVM's x86 backend for Intel processors.

Our long-term goal is the first realistic compiler that is relatively secure end to end for arbitrary C/C++ and Rust programs. This compiler will be a variant of LLVM whose passes introduce no leakage of their own and whose Spectre defenses ensure that compiled programs running with speculation leak no more than the source program does sequentially. On the way there, RS-Test-LLVM will run programs before and after each of LLVM's IR optimizations, on pairs of inputs, in Vellvm [8], an executable Rocq semantics of LLVM IR that we are extending with leakage. For LLVM's x86 backend, we will build a semantics of its machine IR in Lean, using the VeIR compiler infrastructure [27, 29] (Topic 3). The internship topics below check LLVM at several levels: its IR passes (Topics 1 and 2), its x86 backend (Topic 3), and the binaries it produces (Topic 4). Topic 4 tests these binaries for relative security under speculation contracts, which give x86 processors a leakage semantics that includes speculation [13, 22].

RS-Test-LLVM project (internship context)

RS-Test-LLVM (codename) will be a property-based testing framework for relative security of LLVM transformations, with generated programs, generated input pairs, a Rocq definition of relative security as oracle, and as executor Vellvm [8], which covers a large subset of LLVM IR. Alive2 [19] checks LLVM transformations for functional correctness with an SMT solver; RS-Test-LLVM will be its counterpart for relative security, using rigorous testing instead, which scales to whole pipelines and to programs with loops.

For RS-Test-LLVM's leakage model, we are currently adding to Vellvm leakage events for branch conditions, memory addresses, and the operands of variable-time instructions such as division. The model is parameterized by a *backend leakage contract*: like a speculation contract for a processor, it states what the lowering of each IR construct leaks. We instantiate the contract twice, differing only on the select instruction. The conservative contract makes a select leak its condition [32], catching the known class of leaks at the IR level. The optimistic contract treats as constant time the kinds of selects that the backend should lower without leaking their condition, catching instead a select in S turned into a branch in T. The conservative contract suits backends that may lower an integer select to a branch, as on small 32-bit ARM microcontrollers such as the Cortex-M0, where the cmov crate broke [25]. The optimistic contract is more precise for backends that usually lower integer selects to conditional moves, as x86-64 and AArch64 do.

For each generated program S and input i_1, RS-Test-LLVM's dynamic taint tracker marks the parts of i_1, such as arguments and memory bytes, that reach leakage events. RS-Test-LLVM's input-pair generator then fuzzes the unmarked parts, proposing inputs i_2 that S should not distinguish from i_1, as in the contract-driven input generation of the Revizor testing tool [23]. We would also like to try fuzzing the marked parts

[1] For a pass that changes where objects lie in memory, we run S under the layout that T gives its objects, placing each object of S where T places the object it becomes. Otherwise a program that branches on the bits of an address could, in both runs of T, take a branch whose body leaks the input, and skip it in both runs of S: no pass that moves objects would be relatively secure.

sparingly, since a mark means only that S's leakage depends on a part, not that S leaks all of it. RS-Test-LLVM then runs S and T on i_1 and i_2 for a bounded number of steps. It reports a violation when S's traces are equal or one a prefix of the other but T's are neither, provided that S terminates on both inputs within the bound and has no undefined behavior. Since these runs of S check that it does not distinguish the reported pair, the taint tracker need not be trusted. This procedure will be a checker in our property-based testing tool for Rocq, QuickChick [24]. Every counterexample it reports violates the definition of relative security, though violations that need longer runs go unreported.

We have already built prototypes of the taint tracker and input-pair generator for programs generated by GenLLVM [8], Vellvm's QuickChick generator, which produces IR without undefined behavior. However, our tests of the tracker on such programs failed to detect many of the mutants that we seeded into it by dropping taint propagation at specific operations [14]. We traced these misses to GenLLVM: its constraints for avoiding undefined behavior were overly conservative, and its generators rarely let input values flow through memory or function returns. The Vellvm developers agreed with this diagnosis, and we have started generalizing these generators, since a useful test program needs inputs that flow, without causing undefined behavior, into computation that a pass rewrites, where T could newly leak them. With the improved generators, these tests already detect additional memory-related mutants, but connecting these data flows to leakage events, such as branch conditions, remains a problem we are currently working on. The leakage model, the checker and the generalized generators should mostly be in place by early 2027. An intern will work together with Yonghyun Kim, the postdoc in our group leading this project, on a topic along the lines of those below.

Internship topics

Topic 1: Finding leakage-introducing LLVM passes. We will extend the generalized program generator with templates, small IR fragments harvested from the tests LLVM ships for each pass and from the branchless idioms of the crypto libraries BearSSL and BoringSSL. We will also steer the generator toward programs that chosen passes optimize, by adjusting how often each template is used [18]. We will measure how well the generator finds leaks by seeding leakage-introducing mutants into the transformations under test, such as a branchless idiom rewritten into a select or a select into a branch, which the checker must catch [14]. To test pipelines that vectorize loops, we will also add to Vellvm the vector intrinsics it still lacks, for masked loads and stores and for reductions. We expect to find leaks introduced by LLVM that prior work could not: it examined only constant-time libraries, whereas relative security, on generated source programs that are not necessarily constant time, can catch other IR-level rewrites that reveal more than the source. When RS-Test-LLVM finds a violation in a pipeline, it will bisect the pipeline to find the pass that introduced the leak, and shrink the program and input pair to a small counterexample. The main result will be a catalogue of leakage-introducing passes, each with a small counterexample labeled with the contracts under which it leaks. It will be evidence for the ongoing effort to give LLVM a constant-time select that no pass may turn into a branch [1], and for extending such guarantees to relative security.

Required skills: Rocq, functional programming, C/C++, compilers.
Acquired skills: QuickChick, Vellvm, LLVM optimizations, compiler testing, constant-time programming.

Topic 2: Case studies on real code. We will test real code at the level of LLVM IR, one function at a time. For each function, we will write a harness that keeps both inputs of a pair valid and small: consistent pointers and lengths, and short moduli for public-key code, so that each run stays short in Vellvm's interpreter. We will first test LLVM's pipelines on functions of BearSSL and BoringSSL, with their unoptimized IR as S and the result of an optimization pipeline on it as T. We will check that RS-Test-LLVM rediscovers the IR-level leaks prior work found in BearSSL [11, 17, 26], and look for new ones in the parts of these libraries that are not constant time. We will then test the same pipelines on non-cryptographic code that handles secrets, such as sudo's password handling. No secret is declared: T is allowed to leak the password's length, as S

does by reading up to the newline, but not its characters.

Such real code can fall outside what Vellvm can currently execute, through vector intrinsics, assembly code, precompiled libraries, and system calls. We will extend Vellvm where possible and otherwise work around these gaps, for instance by using the libraries' builds without assembly, building library IR from source, and stubbing system calls.

Required skills: Rocq, C, compilers.
Acquired skills: LLVM optimizations, constant-time programming.

Topic 3: Testing LLVM's x86 backend on a Lean semantics of machine IR. On x86-64, many of the leaks observed in practice arise in LLVM's backend, below the IR that Vellvm models. For instance, a pass turns conditional moves back into branches [11], and instruction selection lowers a select of two loaded values to a single load whose address reveals the condition [26]. We will give an executable semantics to the x86 machine IR that LLVM's backend uses between instruction selection and register allocation, in which programs consist of x86 instructions over virtual registers. We will build it in Lean, as a new dialect (intermediate language) of VeIR [27, 29], a compiler infrastructure written in Lean. VeIR already has such a dialect for RISC-V instructions over virtual registers, with an executable semantics. As a first milestone, we will add leakage events to that dialect and test LLVM's RISC-V instruction selection with it, before the x86 dialect exists. The x86 semantics will have leakage events for branches, addresses, and variable-time operands, and will leave out floating-point and vector instructions. For individual instructions, the semantics can build on, or be tested against, the x86 model written in the Sail ISA specification language [2, 12]. Building on it would first need its translation to Lean, as was done for the Sail model of RISC-V on which VeIR's instruction semantics builds [28]. With the x86 semantics as a second executor, we will then use RS-Test-LLVM to test x86 instruction selection, running S in Vellvm and T in the machine-IR semantics, and so check whether instruction selection honors each contract. The two executors exchange only inputs, memory layouts and traces with the checker, so the second one need not be written in Rocq. We will also test the backend passes that run between instruction selection and register allocation, such as the conversion of conditional moves into branches, running both S and T in the machine-IR semantics. A main result will be the x86 semantics, validated by checking its outputs against native runs of the compiled code on the same inputs. As a stretch goal, we will extend it to BearSSL's RSA code compiled without vectorization, to rediscover the leak that the conversion of conditional moves into branches was seen introducing there [11]. The x86 backend can also be tested as a whole without this semantics, by running S in Vellvm and the compiled binary sequentially in Revizor's model (Topic 4). We will use such tests to find leaks early. Beyond this internship, the semantics is a first step towards verifying backend passes, and, once extended with speculative execution, towards testing and verifying the load hardening of the Spectre defenses that Topic 4 tests.

Required skills: Lean or Rocq, programming language semantics, compilers.
Acquired skills: Lean, VeIR, LLVM's backends, x86 assembly.

Topic 4: Testing our Spectre defenses in LLVM with Revizor. We are implementing several variants of Triosecuris [7] in LLVM's x86 backend, for Intel processors. Triosecuris defends against the Spectre attacks that mistrain the processor's predictors for conditional branches (PHT), indirect branches (BTB), and returns (RSB). Triosecuris combines speculative load hardening, which masks loads and branch conditions under misspeculation, with hardware control-flow integrity in the style of Intel's CET [15], which restricts the targets of indirect branches and returns. We proved Triosecuris relatively secure in Rocq for a simplified language. This topic will validate our implementation of Triosecuris with Revizor [22], a tool that tests x86 processors against speculation contracts by comparing what programs leak through the caches with what an executable model of the contract predicts. Here S is a program compiled without the defense and run sequentially, and T the same program compiled with it and run speculatively. RS-Test-LLVM will propose many input pairs, Revizor's model will keep those on which S's traces agree, and will check that T's traces agree too under speculation contracts. We will use Revizor 2.0 [21], whose new contract model,

built on the DynamoRIO binary instrumentation framework [9], proved more efficient and easier to use in our initial experiments than LmSpec/LmTest [5], the earlier variant of Revizor for testing software. Revizor's contracts model misspeculation of conditional branches and of store-to-load forwarding, but not of indirect branches and returns; store-to-load forwarding is outside Triosecuris's scope. This topic will add contracts for indirect branches and returns and, since our checks rely on them, validate them on Intel processors. These contracts should stay tractable because CET restricts where indirect branches can misspeculate to, and because Triosecuris requires setting a processor flag that keeps the BTB from predicting returns [7]. As with RS-Test-LLVM, we will seed realistic bugs into the defense's implementation [14]: the tests should catch them, and a violation reported for an unmodified variant points to a real bug. Beyond generated programs, we will test the defense on real code, such as BearSSL's functions, whose indirect calls exercise the protection of indirect branches. S could also be the program's LLVM IR, run in Vellvm, or, once Topic 3's semantics exists, the machine IR right before the defense, run in that semantics. After this internship, once Topic 3's semantics is extended beyond register allocation and with speculative execution, the same tests could also blame a violation on a single backend pass.

Required skills: compilers, C++, x86 assembly.
Acquired skills: Spectre attacks and defenses, speculation contracts, Revizor, LLVM's x86 backend.

References

[1] J. Alexandre. Introducing constant-time support for LLVM to protect cryptographic code. Trail of Bits blog, 2025.

[2] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, and P. Sewell. ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS. *PACMPL*, 3(POPL), 2019.

[3] G. Barthe, S. Blazy, B. Grégoire, R. Hutin, V. Laporte, D. Pichardie, and A. Trieu. Formal verification of a constant-time preserving C compiler. *PACMPL*, 4(POPL), 2020.

[4] G. Barthe, B. Grégoire, V. Laporte, and S. Priya. Structured leakage and applications to cryptographic constant-time and cost. *CCS*. 2021.

[5] G. Barthe, M. Böhme, S. Cauligi, C. Chuengsatiansup, D. Genkin, M. Guarnieri, D. M. Romero, P. Schwabe, D. Wu, and Y. Yarom. Testing side-channel security of cryptographic implementations against future microarchitectures. *CCS*. 2024.

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

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

[8] C. Beck, H. Chen, and S. Zdancewic. Vellvm: Formalizing the informal LLVM. *NFM*. 2025.

[9] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2004.

[10] D. Brumley and D. Boneh. Remote timing attacks are practical. *Comput. Networks*, 48(5), 2005.

[11] A. Geimer and C. Maurice. Fun with flags: How compilers break and fix constant-time code. *CoRR*, abs/2507.06112, 2025. Presented at PriSC 2026.

[12] S. Goel, W. A. Hunt, Jr., and M. Kaufmann. Engineering a formal, executable x86 ISA simulator for software verification. In M. Hinchey, J. P. Bowen, and E. Olderog, editors, *Provably Correct Systems*, NASA Monographs in Systems and Software Engineering, pages 173–209. Springer, 2017.

[13] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila. Hardware-software contracts for secure speculation. *IEEE SP*. 2021.

[14] C. Hriţcu, L. Lampropoulos, A. Spector-Zabusky, A. Azevedo de Amorim, M. Dénès, J. Hughes, B. C. Pierce, and D. Vytiniotis. Testing noninterference, quickly. *JFP*, 26, 2016.

[15] Intel Corporation. Intel® 64 and IA-32 architectures software developer's manual: Volume 1: Basic architecture: Chapter 18 Control-Flow Enforcement Technology (CET), 2025.

[16] 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.

[17] J. Liu, L. Martínez Gavier, Z. Zhang, M. Böhme, and G. Barthe. Analyzing compiler-induced constant-time violations with compiler at scale. Cryptology ePrint Archive, Paper 2026/2653; to appear at IEEE S&P 2027, 2026.

[18] V. Livinskii, D. Babokin, and J. Regehr. Random testing for C and C++ compilers with YARPGen. *PACMPL*, 4(OOPSLA), 2020.

[19] N. P. Lopes, J. Lee, C. Hur, Z. Liu, and J. Regehr. Alive2: bounded translation validation for LLVM. *PLDI*. 2021.

[20] 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.

[21] Microsoft. Revizor 2.0 documentation: the DynamoRIO model backend, 2026.
[22] O. Oleksenko, C. Fetzer, B. Köpf, and M. Silberstein. Revizor: testing black-box CPUs against speculation contracts. *ASPLOS*. 2022.
[23] O. Oleksenko, M. Guarnieri, B. Köpf, and M. Silberstein. Hide and seek with Spectres: Efficient discovery of speculative information leaks with random testing. *IEEE SP*. 2023.
[24] Z. Paraskevopoulou, C. Hriţcu, M. Dénès, L. Lampropoulos, and B. C. Pierce. Foundational property-based testing. *ITP*. 2015.
[25] RustSec Advisory Database. RUSTSEC-2026-0003: Non-constant-time code generation on ARM32 targets (CVE-2026-23519, `cmov` crate), 2026.
[26] M. Schneider, D. Lain, I. Puddu, N. Dutly, and S. Capkun. Breaking bad: How compilers break constant-time implementations. To appear at AsiaCCS, 2025.
[27] L. Stefanesco, M. Fehr, G. Rennie, and T. Grosser. Fast and correct pointer-based data structures in ITPs. HAL preprint hal-05647613, 2026.
[28] The riscv-lean developers. riscv-lean: Bitblastable semantics for RISC-V in Lean 4, proven equivalent to the Sail RISC-V model. Software repository, 2026.
[29] The VeIR developers. VeIR: Verified Intermediate Representation. Software repository, 2026.
[30] S. Wiebing and C. Giuffrida. Training Solo: On the limitations of domain isolation against Spectre-v2 attacks. *IEEE SP*. 2025.
[31] 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.
[32] Z. Zhang and G. Barthe. CT-LLVM: Automatic large-scale constant-time analysis. Cryptology ePrint Archive, Paper 2025/338, 2025.