

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

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

October 9, 2026

Summary

We will build RelSecLLVM, the first framework for rigorously testing whether a code translation introduces timing leakage, against a formal leakage semantics, for arbitrary Rust and C/C++ programs. A translation that preserves every output can still introduce such leakage, and LLVM does so in practice, turning branchless idioms into branches. Yet neither LLVM nor C-to-Rust translators are checked for it. We will test LLVM's optimization pipelines and C-to-Rust translations, including LLVM-based ones, for *relative security*: the translated program leaks no more than the original. The project will deliver an executable leakage semantics of LLVM IR, a catalogue of leakage-introducing LLVM passes reported upstream, and case studies, from small functions to whole libraries, showing how often and why C-to-Rust translations of crypto and systems code leak more than the C original. RelSecLLVM can prevent such leaks from reaching production.

Introduction

Verified code translation usually means preserving the original's *functional* behavior. For code that handles secrets this is not enough: a translation can preserve every output and still introduce *timing leakage* through branches, memory addresses, or variable-time operations that depend on secrets. Cryptographic libraries avoid all three by the constant-time (CT) discipline, and only a few compilers are proved to preserve CT, such as a variant of the verified CompCert C compiler [1] and the Jasmin compiler [2]. But CT is a discipline for a small body of expert crypto code; most code that handles secrets is not CT, and these proofs cover only CT code. Code that is not CT cannot expect a translation to make it CT, but it can ask that the translation introduce no new leakage.

Translations do introduce leakage though: recent work showed LLVM's middle end rewriting branchless idioms such as $(m \& a) \mid (\sim m \& b)$ into select instructions $(c ? a : b)$ that the backend then lowers to branches [13]; and in January 2026 LLVM was found to compile the CT mask arithmetic of the Rust `cmov` crate into a branch on 32-bit ARM [12]. These leaks were found in existing code after the fact, not by testing each transformation for introduced leaks. Nor are C-to-Rust translators [7] checked for leakage, though they can introduce it too.

We propose to make the absence of introduced timing leakage, *for every program*, a criterion of translation correctness, and to test it on LLVM passes and pipelines and C-to-Rust translations. Our criterion is *relative security*: *the translated program should leak no more than the original*; we previously proved it in Rocq for two Spectre defenses [3, 4]. Writing $\mathcal{L}_P(i)$ for the sequence of leakage events of running P 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, $\mathcal{L}_S(i_1) = \mathcal{L}_S(i_2) \implies \mathcal{L}_T(i_1) = \mathcal{L}_T(i_2)$. Relative security constrains the translation of every program, with no CT constraints and no secret declared in advance.

Internship topics

RelSecLLVM will be a property-based testing framework for relative security of LLVM transformations and C-to-Rust translations, with generated programs and input pairs, a Rocq definition of relative security as oracle, and Vellvm as executor. Vellvm [5] gives a large subset of LLVM IR an executable Rocq semantics, which we will extend with leakage events. S and T will run as IR programs in this semantics, so all testing will happen at the level of LLVM IR, with the backend's leakage approximated by a contract. RelSecLLVM will be to relative security what Alive2 [9] is to functional correctness, but will use rigorous testing instead of SMT solving, so it scales to whole pipelines, loops included.

We have built a prototype taint tracker and input-pair generator (Topic 1) for programs generated by GenLLVM [5], Vellvm's QuickChick generator. Bugs we seeded into the taint tracker went unnoticed, because GenLLVM's instruction generators rarely use inputs as operands: phi nodes arise only for a few loop shapes and binary operations apply only to constants, so inputs rarely reach any instruction, let alone a leakage event. The Vellvm team agreed with the diagnosis, and we have started generalizing these generators. Each intern will work on one topic together with Yonghyun Kim, a postdoc in our group.

Topic 1: Mechanized leakage model and relational testing. We will define a leakage model of LLVM IR by adding leakage events for branch conditions, memory addresses, and variable-time operands to Vellvm. Inputs are the entry function's arguments, the memory they point to, and the data that system calls return, with no part declared secret. We take the trace $\mathcal{L}_P(i)$ from runs of Vellvm's interpreter, which resolves LLVM IR's nondeterminism, so the trace is a function of i . The model is parameterized by a *backend leakage contract* stating what the lowering of each IR construct leaks. We instantiate the contract twice, differing only on select: the conservative contract makes it leak its condition, as in [13], catching the known class of leaks at the IR level; the optimistic contract treats it as constant time, catching instead a select of S turned into a branch in T , as jump threading does.

For each generated program S and input i_1 , RelSecLLVM runs S once under our prototype taint tracker, which marks the parts of i_1 that reach leakage events under the contract in use; the input-pair generator then proposes inputs i_2 that S should not distinguish from i_1 , fuzzing the rest freely and, since marks over-approximate what S reveals, even marked parts sparingly. RelSecLLVM then runs T on i_1 and i_2 : a difference between the two traces is a violation when runs of S on both inputs confirm that S 's traces agree and neither input triggers undefined behavior, so the taint tracker need not be trusted. The checker implementing this test, written in our QuickChick tool [11], will be proved sound and complete against the Rocq definition of relative security on terminating runs, so the trusted base is this definition, the contract, and the instrumented Vellvm.

Required skills: Rocq, functional programming, programming language semantics, compilers.

Acquired skills: LLVM IR, Vellvm, timing side channels, property-based testing with QuickChick.

Topic 2: Generating programs with data flow from inputs. GenLLVM produces IR without undefined behavior, but its instruction generators follow fixed patterns, so inputs rarely reach any computation. A test program is useful when some of its inputs are not determined by S 's leakage yet reach computation that a pass rewrites, since those are the inputs that T could newly leak. We will produce such programs by generalizing GenLLVM's instruction generators to let inputs flow into computation.

Following *Testing Noninterference, Quickly* [6], we will measure test adequacy by seeding two kinds of mutants. Leakage-introducing mutants go into the transformation under test: the identity transformation ($T = S$) and, optionally, pipelines found clean under the contract in use; examples are a branchless idiom rewritten into a select under the conservative contract and a select into a branch under the optimistic one; such a mutant counts as found when the checker fails with it and passes without it. Taint-dropping mutants go into the taint tracker, with $T = S$, so that the input-pair generator freely varies an input part that S leaks; such a mutant counts as found when more proposed pairs fail confirmation than without it.

Required skills: functional programming, compilers.

Acquired skills: Rocq, QuickChick, Vellvm, random program generation, mutation testing, LLVM IR.

Topic 3: Finding leakage-introducing LLVM passes. We will extend the program generator of Topic 2 with IR templates harvested from LLVM’s regression tests and from the branchless idioms of the C crypto libraries BearSSL and BoringSSL. The generator will instantiate these templates, connect them to inputs and to each other, and keep some inputs clear of S ’s leakage events; like YARPGen [8], it will steer the mix of templates toward chosen passes. Optionally, we will also test real code, with the unoptimized IR of BearSSL and BoringSSL as S and their optimized IR as T . When RelSecLLVM 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 to report upstream, using QuickChick’s shrinking and LLVM’s test-case reducer.

Required skills: functional programming, C, compilers.

Acquired skills: Rocq, QuickChick, LLVM IR and optimizations, compiler testing, constant-time programming.

Topic 4: C-to-Rust translation case studies. Here S is a C function and T its Rust translation with the same C interface, both compiled to unoptimized IR; we will use c2rust for literal translations and LLM-based translators for idiomatic ones. We will start with constant-time functions from BearSSL and BoringSSL and with non-cryptographic code that handles secrets: the password handling of OpenSSH and sudo, and data-oblivious algorithms from the enclave literature [10]. No secret is declared: a Rust version of sudo’s password reader may leak the password’s length, as the C code does, but not its characters. At scale, and optionally for this internship, we will test c2rust and LLM translations of all of BearSSL, and LLM translations of a sample of CRUST-Bench projects [7], measuring how often idiomatic translation adds leakage.

Required skills: C, Rust, compilers.

Acquired skills: C-to-Rust translation with c2rust and LLMs, constant-time programming.

References

- [1] 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.
- [2] G. Barthe, B. Grégoire, V. Laporte, and S. Priya. [Structured leakage and applications to cryptographic constant-time and cost](#). *CCS*. 2021.
- [3] J. Baumann, R. Blanco, L. Ducruet, S. Harwig, and C. Hritcu. [FSLH: flexible mechanized speculative load hardening](#). *CSF*. 2025.
- [4] J. Baumann, Y. Kim, Y. Farba, C. Hritcu, and J. Leatherman-Brooks. [Triosecuris: Formally verified protection against speculative control-flow hijacking](#). *CSF’26*, 2026.
- [5] C. Beck, H. Chen, and S. Zdancewic. [Vellvm: Formalizing the informal LLVM](#). *NFM*. 2025.
- [6] C. Hrițcu, 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.
- [7] A. Khatry, R. Zhang, J. Pan, Z. Wang, Q. Chen, G. Durrett, and I. Dillig. [CRUST-Bench: A comprehensive benchmark for C-to-safe-Rust transpilation](#). *COLM*, 2025.
- [8] V. Livinskii, D. Babokin, and J. Regehr. [Random testing for C and C++ compilers with YARPGen](#). *PACMPL*, 4(OOPSLA), 2020.
- [9] N. P. Lopes, J. Lee, C. Hur, Z. Liu, and J. Regehr. [Alive2: bounded translation validation for LLVM](#). *PLDI*. 2021.
- [10] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa. Oblivious multi-party machine learning on trusted processors. *USENIX Security*, 2016.
- [11] Z. Paraskevopoulou, C. Hrițcu, M. Dénès, L. Lampropoulos, and B. C. Pierce. [Foundational property-based testing](#). *ITP*. 2015.
- [12] RustSec Advisory Database. [RUSTSEC-2026-0003: Non-constant-time code generation on ARM32 targets \(CVE-2026-23519, cmov crate\)](#), 2026.
- [13] Z. Zhang and G. Barthe. [CT-LLVM: Automatic large-scale constant-time analysis](#). *Cryptology ePrint Archive*, Paper 2025/338, 2025.