# LLVM Translation Validation Automated with Large Language Models and Lean

Chunhao Liao
University of Waterloo
Canada
chunhao.liao@uwaterloo.ca

Hongxu Xu
University of Waterloo
Canada
hongxu.xu@uwaterloo.ca

Xintong Zhou
University of Waterloo
Canada
x27zhou@uwaterloo.ca

Yizhou Zhang
University of Waterloo
Canada
yizhou@uwaterloo.ca

Chengnian Sun
University of Waterloo
Canada
cnsun@uwaterloo.ca



## Abstract

LLVM is the cornerstone of modern compilers, but its subtle intermediate representation (IR) semantics make transformations error-prone and necessitate formal verification. Alive2, a state-of-the-art translation validator based on satisfiability modulo theories, has achieved substantial success in automating the validation of LLVM transformations. However, it still faces scalability limitations, does not support symbolic bitwidths, and offers only bounded guarantees for loops. In contrast, interactive theorem provers such as Lean can address these cases but require substantial proof engineering.

In this paper, we present Trivet, a framework combining large language models (LLMs) and Lean for automated translation validation of LLVM transformations. Trivet generates structured proof scaffolds based on source and target functions, automatically discharges obligations amenable to deterministic reasoning, and delegates transformation-specific obligations to LLMs. It produces refinement proofs or counterexample-based refutations, with every successful verdict checked by the Lean kernel. On 148 LLVM transformations, Trivet verifies or refutes 147, leaving one invalid case unresolved. Successful cases include 60 loop-free transformations with symbolic bitwidths, 27 cases from a restricted class of loop-containing transformations, and 10 complex valid fixed-bitwidth cases on which Alive2 times out. Compared with an unscaffolded baseline, scaffolding enables 26 additional proofs. On cases solved by both configurations, it reduces mean proof time by 75.9% and mean monetary cost by 88%.

***Keywords:*** Compilers, Translation Validation, Large Language Models, Formal Verification

## 1 Introduction

The LLVM compiler infrastructure [16] is a cornerstone of modern software development in industry and academia. At its core is the LLVM Intermediate Representation (IR), a strongly typed language that bridges high-level source languages and diverse target architectures. To enable aggressive optimization, the IR provides highly expressive semantics, including Undefined Behavior (UB) [25]. This expressiveness comes at a cost: combined with the sophisticated reasoning required by optimization passes, it makes correct implementation notoriously difficult—developers must account for every edge case by hand, and a single overlooked one can silently produce a miscompilation. Indeed, optimization passes account for a significant portion of reported bugs in LLVM [17, 51, 60], making reliable optimizer development a persistent challenge.

***Translation Validation.*** To provide formal guarantees that compiler transformations preserve program semantics, modern compiler development increasingly relies on translation validation [44, 49]. This technique compares a source function (pre-transformation program) with its corresponding target function (post-transformation program) using a formal semantic model, ultimately either proving that the target refines the source (see theorem 2.1) or generating a counterexample if the transformation is invalid. A prominent instance of this methodology is Alive2 [38], a state-of-the-art, SMT-based [3] *bounded* translation validator specifically designed for LLVM IR. Alive2 has been widely adopted by the LLVM community to rigorously vet compiler transformations before they are accepted [24].

Despite its practical effectiveness, Alive2 has three inherent limitations:

1. **Solver Scalability.** As an SMT-based tool, Alive2 can time out on large bitwidths, long instruction sequences, and complex formulas [45].
2. **Fixed-Bitwidth Reasoning.** Alive2 only supports validation of fixed-bitwidth transformation instances, limiting its ability to verify optimizations that hold across symbolic bitwidths [21, 43].
3. **Bounded Loop Unrolling.** Alive2 validates loops by unrolling them up to a fixed bound, leaving subsequent iterations unchecked.

These limitations motivate a complementary approach to formally validating LLVM transformations.

Interactive theorem provers (ITPs) offer a promising alternative for translation validation. Compared to SMT-based bounded translation validation, an ITP can efficiently express and validate transformations involving symbolic bitwidths, large bitwidths, long instruction sequences, and loops. While these benefits are compelling in principle, the practical adoption of such provers has historically been hindered by the prohibitive cost of proof engineering, which often requires an order of magnitude more proof code than implementation code [8, 9, 18]. However, recent advances in Large Language Models (LLMs) make ITP-based workflows increasingly viable. LLMs can iteratively synthesize and repair proof scripts, while the ITP kernel rigorously guarantees the correctness of the final proof [40, 53, 55].

***Trivet.*** To this end, this paper presents Trivet[1], a framework that pioneers the application of LLMs and Lean [10, 11] to the automated translation validation of LLVM transformations. By leveraging domain knowledge of compiler optimizations, Trivet automatically extracts a Lean refinement theorem and structured proof scaffolds directly from the source and target functions. These scaffolds not only minimize expensive LLM queries, but also systematically guide the LLM through both verification and refutation. For verification, the scaffold leaves focused typed holes for transformation-specific proof obligations, which an LLM can solve, using Lean's diagnostic feedback to iteratively repair proofs when necessary. For refutation, a structured prompt guides the LLM to propose a concrete counterexample. Trivet uses a deterministic scaffold to check the candidate counterexample and, upon successful validation, construct a machine-checked proof of non-refinement. In both cases, Lean remains the trusted checker of the final verdict.

We evaluate Trivet on a dataset of 148 LLVM transformations. The dataset includes a loop-free subset (60 valid, 60 invalid) and a loop-containing subset (14 valid, 14 invalid). Both subsets cover fixed-bitwidth and symbolic-bitwidth cases. Overall, Trivet successfully validates all 74 valid transformations and refutes 73 invalid ones, leaving only one invalid transformation unresolved. Moreover, compared to an unscaffolded baseline, scaffolding reduces mean proof time by 75.9% and mean monetary cost by 88%, while enabling 26 additional proofs in total. In contrast, Alive2 validates 20 out of 30 loop-free, fixed-bitwidth LLVM transformations, but cannot validate symbolic-bitwidth cases or provide unbounded guarantees for loop-containing transformations. These results demonstrate Trivet's effectiveness in automating the translation validation of LLVM transformations.

***Contributions.*** Our main contributions are listed below.

- We introduce Trivet, the first framework applying LLMs and Lean to the automated translation validation of LLVM transformations: the LLM synthesizes refinement proofs or counterexamples, while the Lean kernel rigorously certifies every verdict.
- We develop an automated, non-LLM generator that extracts a Lean refinement theorem and structured proof scaffolds from the source and target functions. These scaffolds encode compiler domain knowledge to guide the LLM, reducing validation to focused, transformation-specific holes and converting LLM-proposed counterexamples into machine-checked proofs.
- We evaluate Trivet on 148 LLVM transformations, successfully validating all 74 valid cases and refuting 73 invalid ones, leaving only one invalid transformation unresolved. Trivet validates symbolic-bitwidth and loop-containing transformations beyond Alive2's scope. Scaffolding reduces mean proof time by 75.9% and mean monetary cost by 88% compared to an unscaffolded baseline, enabling 26 additional proofs in total.

[1]Trivet vets LLVM transformations based on the integration of three components: Lean, LLMs, and LLVM.

## 2 Background

This section provides necessary background on LLVM transformations and LLVM translation validation.

### 2.1 LLVM Transformations

LLVM transformations rewrite LLVM IR while preserving the program's observable behavior. They range from local instruction-level rewrites to optimizations involving multiple basic blocks or loops. Following prior work [43], we represent a transformation by the following judgment:

$$PRE \vDash LHS \Rightarrow RHS$$

where *LHS* (left-hand side) denotes the pre-transformation fragment, *RHS* (right-hand side) denotes its replacement, and *PRE* is a logical predicate over the operands in *LHS* that defines when the transformation is valid. Specifically, this judgment states that replacing *LHS* with *RHS* is semantics-preserving when *PRE* holds. When no explicit precondition is specified, *PRE* defaults to true, indicating that the transformation is always valid and imposing no constraints on the operands.

In practice, LLVM IR translation validation instantiates this judgment as a pair of LLVM functions [24]: a *src* function, which encodes *PRE* and *LHS*, and a *tgt* function, which encodes *RHS*. The *PRE* is expressed within the *src* function through the `llvm.assume` intrinsic, which restricts the inputs to those satisfying *PRE* [28].

***Undefined Behavior Semantics.*** LLVM has two forms of Undefined Behavior (UB): *immediate* UB and *deferred* UB [25]. Immediate UB, such as division by zero, permits arbitrary program behavior. Deferred UB, such as integer overflow, instead produces an ill-defined value that propagates through downstream computations. These ill-defined values take two

forms in LLVM: *poison* [29] and *undef* [32]. Our work supports *poison*, which taints dependent computations and triggers immediate UB when used in a way that requires a well-defined value, such as a branch condition. In contrast, an *undef* value nondeterministically takes any value of its type at each use, which has long complicated reasoning about LLVM semantics. LLVM has deprecated undef [37], and thus our work does not support it.

***Refinement Relation.*** Since optimizers routinely exploit UB, correctness cannot be formulated as equivalence. Following Alive2 [26, 38], we formalize transformation correctness in terms of refinement.

**Definition 2.1** (Refinement)**.** A target function *tgt refines* a source function *src*, denoted $src \sqsupseteq tgt$, if for every input:

- if *src* triggers immediate UB, *tgt* may exhibit any behavior [27];
- if *src* returns poison, *tgt* may return poison or any other value, but must not trigger immediate UB [29];
- if *src* returns a well-defined value, *tgt* must return the same value and must not trigger immediate UB [33].

In the absence of UB, this relation degenerates to equivalence.

### 2.2 LLVM Translation Validation

Translation validation checks each transformation result independently. Given source and target functions *src* and *tgt*, the validator determines whether $src \sqsupseteq tgt$. A successful check verifies the transformation; otherwise, it reports a counterexample showing that *tgt* does not refine *src*.

#### 2.2.1 Bounded SMT-Based Translation Validation.

Bounded SMT-based translation validation encodes the semantics of the source and target functions as logical constraints over bounded domains, such as bit-vectors with fixed-bitwidth and finitely unrolled loops. An SMT solver then searches for an input on which *tgt* does not refine *src*; if no such input exists, the transformation is verified within the encoded bounds [20, 38]. Alive2, the state-of-the-art LLVM translation validator [24, 38], follows this approach.

***Inherent Limitations of Alive2.*** Despite its widespread adoption in the LLVM community, Alive2 inherits fundamental limitations from its underlying bounded SMT-based design, particularly concerning solver scalability, fixed-bitwidth reasoning, and bounded loop unrolling.

*Limitation: Solver Scalability.* Alive2 suffers from scalability bottlenecks due to its whole-function formulation of refinement checking and its reliance on bit-blasting in the underlying SMT solver. It encodes complete source and target functions without decomposition, causing solver complexity to grow rapidly with instruction count and bitwidth [45]. Furthermore, bit-blasting lowers fixed-width arithmetic to Boolean circuits, which can become particularly expensive for transformations involving nonlinear bit-vector operations (*e.g.* `bvudiv`) or generalized rewrites [21, 43].

*Limitation: Fixed-Bitwidth Reasoning.* Since standard SMT bit-vector logics, such as the quantifier-free logic over fixed-size bit-vectors (QF_BV), reason about fixed bitwidths [2], Alive2 [38] cannot validate transformations over symbolic bitwidths. Prior work therefore commonly bounds validation to selected bitwidths (*e.g.* 64 bits) [39, 43]. Consequently, validation at selected bitwidths leaves every other width unverified. Moreover, transformations with multiple bitwidths require validating each width combination separately, substantially increasing the validation overhead.

*Limitation: Bounded Loop Unrolling.* Alive2 handles transformations involving loops by unrolling them to a fixed depth [38]. Choosing this depth poses a dilemma: shallow unrolling misses bugs that manifest only at later iterations, while deeper bounds make the generated queries intractable. Consequently, bounded loop validation guarantees correctness only within the selected bound, leaving subsequent iterations unchecked.

#### 2.2.2 ITP-Based Translation Validation.

Interactive theorem provers (ITPs) such as Rocq [52] and Lean [10, 11] provide a rigorous alternative to SMT-based translation validation. In an ITP, the semantics of the compiler IR and the target refinement relation are formalized in higher-order logic, with correctness proofs mechanically verified by a small, trusted kernel. For example, Vellvm [5, 62, 64, 65] formalizes LLVM IR semantics in Rocq, while Lean-MLIR [6] provides a Lean framework to validate transformations across multiple MLIR dialects, including LLVM.

While ITP-based validation naturally handles symbolic bitwidths and unbounded loops, it fundamentally relies on manual proof construction—a labor-intensive process requiring specialized formal methods expertise. Recent advances in applying Large Language Models (LLMs) to automated theorem proving offer a viable path to mitigate this burden. In this work, we combine the foundational correctness guarantees of ITPs with the automation of LLMs to synthesize formal proofs for LLVM transformations.

## 3 Methodology

This section presents the design of Trivet, a framework that integrates an LLM with Lean to automate translation validation for LLVM transformations.

Figure 1 shows Trivet's workflow. For each transformation, Trivet launches two branches *in parallel*, each producing a proof for the Lean kernel to check:

- **Refinement branch.** Trivet first emits a refinement scaffold that discharges routine obligations and leaves typed holes for the transformation-specific proof obligations; the LLM later fills the holes.

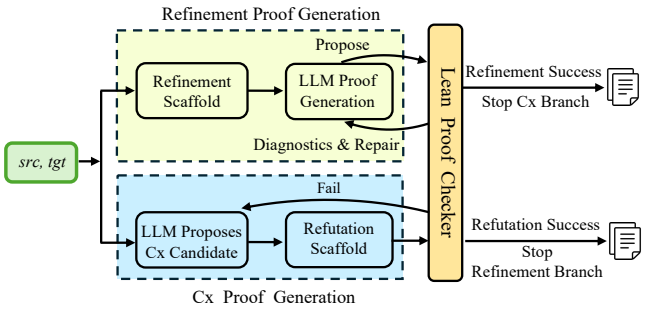


**Figure 1.** The workflow of Trivet. Cx denotes counterexample.

- **Counterexample branch.** The LLM first proposes a counterexample candidate; Trivet then emits a refutation scaffold that turns the candidate into a complete Lean proof of non-refinement.

Lean checks each proof, with its diagnostics guiding the LLM to revise the refinement proof or propose a new counterexample. Each branch repeats generation and checking until a proof is accepted or the time budget expires. The first accepted proof establishes validity or invalidity; Trivet stops the other branch and reports the result. If neither succeeds, Trivet leaves the transformation unresolved.

This section is organized as follows. §§ 3.1 and 3.2 present the refinement and counterexample branches for loop-free, single-block transformations. § 3.3 extends these workflows to a restricted class of loop-containing transformations. § 3.4 describes the implementation and limitations.

### 3.1 Refinement Proof Generation

Generating complete refinement proofs with an LLM is costly, particularly across diverse UB cases. To address this, we design a scaffold construction strategy that delegates only transformation-specific reasoning to the LLM: it decomposes the refinement goal ($src \sqsupseteq tgt$) via UB-triggering conditions, automatically discharges deterministic obligations, and leaves only typed proof holes for the LLM to complete.

To guide scaffold construction, we examine Lean-MLIR's LLVM dialect [6] and identify nine cases in which an operator or instruction flag may produce UB. We call such operators and flags *UB-sensitive constructs* and associate each with its precise *UB-triggering circumstance*, the condition under which each construct triggers UB, such as signed overflow for `nsw` or a zero divisor for `udiv`.

Trivet constructs the scaffold by splitting cases on these conditions and automatically discharging proof obligations whenever the source yields UB. For well-defined source cases, Trivet generates typed proof holes—establishing value equivalence and absence of target UB—to be completed by the LLM under the corresponding branch hypotheses. To guarantee soundness, Trivet ensures that the generated proof passes Lean kernel type-checking without open goals, strictly preserves the original definitions and theorem statement, and relies exclusively on standard axioms [54].

#### 3.1.1 Illustrative Example.

Figure 2 illustrates refinement proof generation for a real-world LLVM transformation [23]. Let $w$ be a positive bitwidth, and $a$ and $b$ be $w$-bit inputs. The transformation is:

$$\underbrace{(y <_{\mathrm{s}} 0) = (T <_{\mathrm{s}} 0) \;\vDash\; y <_{\mathrm{u}} T}_{src(w)} \implies \underbrace{a +_{\mathsf{nuw}} b <_{\mathrm{u}} 1}_{tgt(w)},$$

$$\text{where } T = w \times w,\; y = (b +_{\mathsf{nuw}} (T-1)) +_{\mathsf{nuw}} a.$$

Here, all arithmetic uses $w$-bit values. Multiplication and subtraction wrap modulo $2^w$, while `nuw` additions produce poison on unsigned overflow. The operators $<_{\mathrm{s}}$ and $<_{\mathrm{u}}$ denote signed and unsigned comparisons, respectively. $src(w)$ and $tgt(w)$ denote the source and target functions instantiated with input bitwidth $w$. The refinement goal for the functions *src* and *tgt* is:

$$\forall w \in \mathbb{N},\; w > 0 : \quad src(w) \sqsupseteq tgt(w).$$

As shown in Figure 2 (i), Trivet identifies five UB-sensitive constructs: poison inputs (C1), unsigned overflow in two source additions (C2, C3), an assumption violation (C4), and unsigned overflow in the target addition (C5). While C1–C4 stem from the source, C5 is target-specific. Because C5 could cause the target to evaluate to poison for a well-defined source execution (violating refinement), it must be proven unreachable and cannot be discharged automatically.

In Figure 2 (ii), Trivet performs case analysis on C1–C4 to cover all source outcomes. If any of C1–C3 holds, the source evaluates to poison, and a pre-existing lemma discharges the obligation since the target produces no immediate UB. If C4 holds, the source triggers immediate UB, vacuously satisfying refinement.

Otherwise, the source is well-defined. Trivet then generates two proof holes: a value equivalence hole $(y <_{\mathrm{u}} T) = (z <_{\mathrm{u}} 1)$, and a target-UB hole ensuring that $a + b$ does not overflow (C5). In Figure 2 (iii), the LLM fills these two holes under the accumulated branch hypotheses (which rule out C1–C4). Finally, the Lean kernel type-checks the assembled proof within the scaffold.

#### 3.1.2 Proof Scaffold Generation.

Trivet first identifies UB-sensitive constructs across both functions and systematically splits on three categories: ① the source's UB-triggering circumstances, ordered by when values are first demanded; ② target-specific circumstances that can trigger immediate UB; and ③ value-selection conditions (e.g., in `select` instructions [31]), which guides the LLM to prove each branch separately. These splits produce proof obligations under distinct branch hypotheses, such that the original refinement goal holds if and only if all resulting obligations are discharged.

Each obligation is handled according to the source outcome. For immediate UB, Trivet automatically discharges

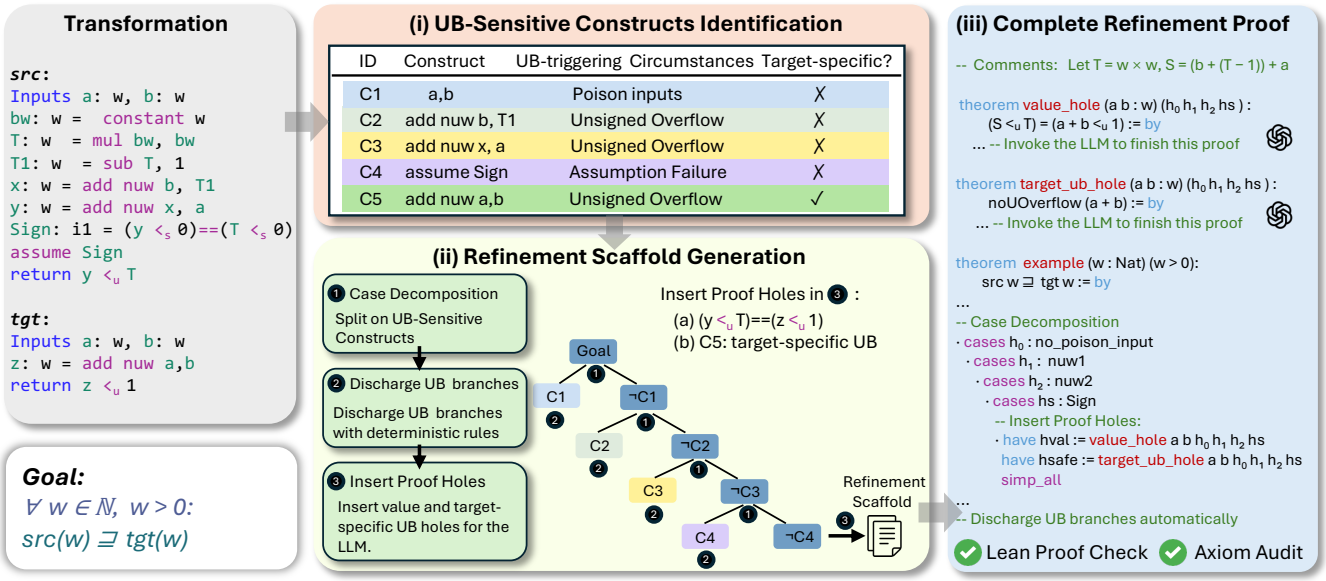


**Figure 2.** An illustrative example of refinement proof generation in Trivet.

the obligation because any target behavior is permitted [26]. For poison, it discharges the obligation once target immediate UB is ruled out, as the target may yield either poison or a well-defined value [30]. For a well-defined value, Trivet leaves one hole for value equality and separate holes to rule out each target-specific UB-triggering circumstance. Finally, Trivet validates the scaffold's syntactic and type correctness using Lean.

**3.1.3 LLM-Driven Proof Completion.** Trivet asks the LLM to fill the remaining typed holes with proofs of value equivalence and the absence of target-specific UB. A candidate proof is accepted when it satisfies the following conditions:

1. the completed proof passes kernel checking;
2. the proof contains neither `sorry` nor `admit`;
3. the proof depends exclusively on the standard axioms [54]; and
4. the source and target definitions and the theorem statement remain unchanged.

***Prompt Design.*** The prompt template comprises three core elements.

- **System and Task.** The prompt assigns the LLM the role of a Lean theorem prover and identifies the target proof file. It instructs the LLM to finish the proof by replacing the `sorry` placeholders with valid proofs.
- **Instructions.** The prompt instructs the LLM to inspect proof goals and retrieve diagnostics. The prompt also specifies that the LLM may add auxiliary lemmas and raise resource limits, such as `maxHeartbeats` and `maxRecDepth`, when necessary.
- **Hard Rules.** To preserve proof integrity, the prompt explicitly forbids changes to the refinement theorem statement or the source and target definitions.

## 3.2 Counterexample Proof Generation

The counterexample branch refutes invalid transformations by turning a candidate counterexample into a proof of non-refinement. Because refinement quantifies universally over all admissible bitwidths and inputs, a single counterexample suffices to disprove it. The LLM is restricted solely to proposing a candidate counterexample with concrete bitwidths and input values, after which a deterministic scaffold constructs the complete Lean proof. Because this scaffold introduces no additional axiomatic dependencies, we omit the axiom audit from § 3.1.3. The proof is accepted once it passes Lean kernel type-checking.

**3.2.1 Illustrative Example.** Figure 3 illustrates the counterexample proof generation in Trivet using an incorrect generalization of a real-world LLVM transformation [22, 43]. This transformation involves two symbolic bitwidths satisfying $0 < w_1 < w_2$, where inputs $x$ and $C_2$ are of width $w_1$, whereas $C_1$ is of width $w_2$:

$$\underbrace{C_2 = w_1 - 1 \;\vDash\; (0 - \text{zext}(x \gg_u C_2)) \;\&\; (C_1 - \text{sext}(x))}_{src(w_1, w_2)}$$
$$\Longrightarrow \underbrace{\left(x <_s (1 \ll (C_2 + 1))\right) \;?\; (C_1 - \text{sext}(x)) : 0.}_{tgt(w_1, w_2)}$$

The source constructs a mask from the sign bit of $x$, whereas the target replaces this mask with a signed comparison and a conditional selection. The counterexample goal is to establish that there exist admissible bitwidths for which the source is

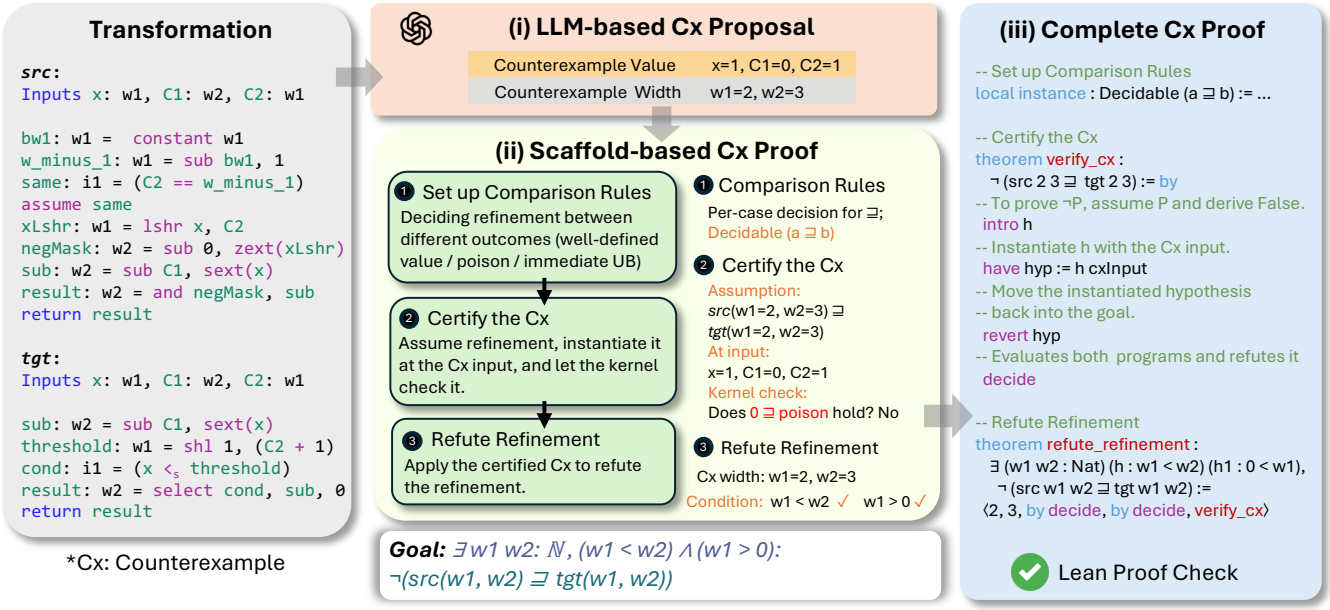


**Figure 3.** An illustrative example of counterexample proof generation in Trivet.

not refined by the target:

$$\exists w_1, w_2 \in \mathbb{N},\ 0 < w_1 < w_2 : \neg\big(src(w_1, w_2) \sqsupseteq tgt(w_1, w_2)\big)$$

The counterexample proof process comprises three stages. First, the LLM proposes a candidate with bitwidths $w_1 = 2$ and $w_2 = 3$ and input values $x = 1$, $C_1 = 0$, and $C_2 = 1$, as shown in Figure 3 (i). Second, the scaffold constructs a formal proof of non-refinement from this candidate (Figure 3 (ii)). To enable proof by computation, the scaffold provides a computable `Decidable` instance for $a \sqsupseteq b$, covering well-defined values, poison, and immediate UB. This instance allows the Lean kernel to check that the concrete source and target outcomes violate refinement, thereby certifying the counterexample. To establish the contradiction, the scaffold assumes $src(2, 3) \sqsupseteq tgt(2, 3)$, specializes this assumption to the proposed counterexample inputs, and invokes kernel evaluation. The source evaluates to a well-defined 0, whereas the target evaluates to poison under LLVM semantics [29] because `shl 1, 2` shifts a two-bit value by its bitwidth. The kernel therefore refutes $0 \sqsupseteq poison$. Together with proofs of the instantiated width constraints $w_1 < w_2$ (*i.e.*, $2 < 3$) and $0 < w_1$ (*i.e.*, $0 < 2$), this fixed-width contradiction refutes the refinement theorem. Finally, Trivet accepts the proof once it passes Lean kernel type-checking.

**3.2.2 LLM-Driven Counterexample Proposal.** Trivet asks the LLM to propose a candidate counterexample with concrete bitwidths and input values. If Trivet fails to refute the refinement theorem using the candidate, it appends the Lean diagnostics to the prompt and asks the LLM to generate a new candidate. This process repeats until Lean verifies a proof of non-refinement or five attempts have been made.

***Prompt Design.*** The prompt template follows the same structure as the refinement prompt in § 3.1.3.

- **System and Task.** The prompt assigns the LLM the role of a counterexample generator. It asks the LLM to find a counterexample that violates the refinement claim.
- **Instructions.** The prompt instructs the LLM to propose a counterexample by assigning concrete values to the symbolic bitwidths and inputs. If a candidate cannot be certified, the next prompt includes the Lean diagnostics and asks the LLM to propose a new counterexample candidate.
- **Hard Rules.** The LLM must save one candidate to the designated file as JSON, with complete `bitwidths` and `inputs` objects and no extra text. Before returning, it must check all constraints and confirm that the instantiated target function does not refine its source function.

**3.2.3 Scaffold-Based Proof Construction.** Given a candidate counterexample proposed by the LLM, Trivet deterministically synthesizes a proof of non-refinement in two steps:

***Contradiction via Decidable Evaluation.*** The scaffold assumes the claimed refinement theorem for contradiction and specializes it with the proposed bitwidths and input values. To evaluate the resulting concrete proposition, Trivet leverages a `Decidable` instance of the refinement relation ($a \sqsupseteq b$) that formalizes comparisons across well-defined values, poison, and immediate UB. Kernel evaluation reduces the source and target expressions to concrete semantic outcomes; because these outcomes violate refinement (*e.g.*, deriving

$0 \sqsupseteq \textit{poison}$), the decision procedure reduces the proposition to `False`.

***Precondition Discharge and Verification.*** The scaffold then discharges the concrete bitwidth constraints (*e.g.*, $0 < w_1 < w_2$) via computation, assembling the counterexample and the contradiction into a complete proof of the negated refinement theorem. The final proof contains no remaining holes and is validated by Lean kernel type-checking.

### 3.3 Loop-Containing Transformations

For loop-free transformations, refinement can be established by directly comparing the source and target behaviors for all possible inputs. For transformations involving loops, however, the proof must also account for all feasible loop iteration counts, since a loop may execute an arbitrary number of iterations depending on the inputs. To address this challenge, Alive2 applies bounded loop unrolling, checking refinement only up to a selected unrolling bound and thereby potentially missing counterexamples that require more iterations [38]. By contrast, Trivet establishes refinement by induction, yielding a single correctness proof that covers all feasible loop iteration counts without relying on a fixed unrolling bound.

In this paper, Trivet provides preliminary support for a restricted class of loop-containing transformations, from a source function containing LLVM's canonical loop form [35, 36] to a loop-free target function with a single basic block. Specifically, the source function consists of three blocks: an entry preheader, a single self-looping body block with one back edge, and a dedicated exit block that returns immediately. This fixed topology enables Trivet to reuse a fixed proof scaffold and is grounded in practice: canonical loop form is enforced before the loop-pass execution, and the single-block, single-back-edge form is targeted by several loop idiom recognizers [34, 35].

For general loop-containing transformations, however, the proof scaffold varies with the source function's control-flow graph (CFG). This variation arises because inductive proofs require every cycle in the CFG to pass through an invariant-bearing cutpoint [12], resulting in CFG-specific invariants and induction obligations. Accordingly, we leave support for general loop structures to future work.

#### 3.3.1 Loop-Aware Refinement Scaffold and Proof Generation.

Following the loop-free workflow in § 3.1, Trivet first constructs a loop-aware scaffold by extending the loop-free scaffold in § 3.1.2 to loop-containing transformations and then invokes the LLM to fill the proof holes and complete the proof.

For loop-containing transformations, Trivet reuses the loop-free scaffold's case decomposition, refinement rules, and target-specific UB holes. In the source entry block, it splits cases and automatically applies the refinement rules to discharge obligations for poison and immediate UB outcomes. For the target function, Trivet generates target-specific UB

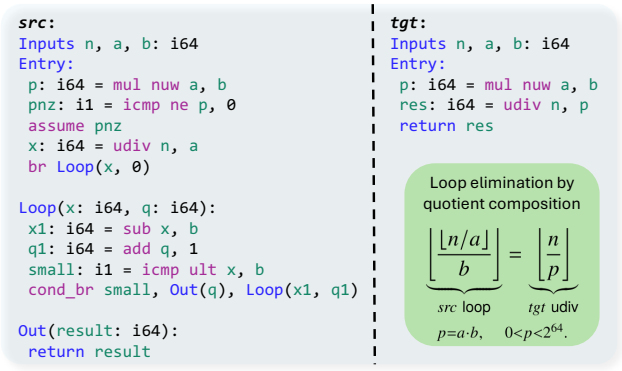


**Figure 4.** An illustrative example for a loop-containing transformation

holes to rule out UB in the target when the source returns a well-defined value. For well-defined source outcomes, Trivet's loop-free scaffold uses a single value hole requiring the source and target functions to return the same value. In contrast, for transformations involving loops, Trivet replaces this single hole with loop-specific holes that jointly specify the proof obligations for refinement over an arbitrary number of iterations. Table 1 provides an overview of the general and loop-specific holes in the refinement scaffold. Finally, Trivet invokes the LLM to provide the loop invariant and fill the proof holes in the scaffold and checks the completed proof following the same procedure in § 3.1.3.

***Illustrative Example.*** Figure 4 shows an illustrative loop-containing transformation that replaces a quotient computation by repeated subtraction with an unsigned division. The source loop implements Euclidean division [7]: for $x, b \in \mathbb{N}$ with $b > 0$, the unique quotient–remainder pair $(q, r)$ satisfies $x = q \cdot b + r$ and $0 \le r < b$, where $q = \lfloor x/b \rfloor$ counts the subtractions. Starting from `x = udiv n, a` and $q = 0$, the source repeatedly updates $(x, q)$ to $(x - b, q + 1)$ while $x \ge b$ and then returns $q$. It thus computes $\lfloor \lfloor n/a \rfloor / b \rfloor = \lfloor n/(a \cdot b) \rfloor$, which the target computes directly using `udiv n, p`, where $p = a \cdot b$.

***Refinement Proof for the Example.*** Trivet's refinement proof scaffold first applies the case decomposition process to discharge the source entry block's poison and immediate UB cases (*e.g.* $p = 0$ violates `assume pnz`). For the remaining cases, the scaffold leaves target-specific UB obligations and three loop-specific holes:

1. *Target-specific UB.* Proofs that the target's `mul nuw` cannot overflow and that $p \neq 0$ for `udiv n, p`.
2. *Loop invariant.* A predicate $I(n, a, b, x, q)$ relating the *tgt* inputs ($n$, $a$, and $b$) to the *src* loop state ($x$, and $q$).
3. *Inductive step.* A proof that $I$ is preserved by $(x, q) \mapsto (x - b, q + 1)$ when $x \ge b$ and implies $q = \lfloor n/p \rfloor$ when $x < b$.

**Table 1.** General and loop-specific holes in the preliminary loop-containing scaffold.

| Type | Hole and Role |
|---|---|
| General | • *Target-specific UB*: rules out each relevant target-side UB condition whenever the source is defined. |
| Loop-specific | • *Loop invariant*: summarizes the reachable source-loop state and function inputs. |
| | • *Inductive step*: preserves the invariant and proves equality with the target result at loop exit. |
| | • *Final assembly*: establishes the invariant at loop entry, then uses the inductive proof to establish refinement for any number of loop iterations. |

4. *Final assembly.* A proof establishing $I$ at loop entry, where $(x, q) = (\lfloor n/a \rfloor, 0)$, and using the inductive step to show that the source loop's result equals $\lfloor n/p \rfloor$.

The LLM completes the proof by filling these holes as follows:

1. *Target-specific UB.* It uses $a \cdot b < 2^{64}$ and $p \neq 0$ from the source entry conditions to exclude overflow in `mul nuw` and division by zero in `udiv n, p`.
2. *Loop invariant.* It defines $I$ to require well-defined inputs and loop values, $0 < a \cdot b < 2^{64}$, and $q + \lfloor x/b \rfloor = \lfloor n/(a \cdot b) \rfloor$. Here, $q$ and $\lfloor x/b \rfloor$ count completed and remaining subtractions, respectively.
3. *Inductive step.* On the back edge ($x \geq b$), it derives $q + 1 < 2^{64}$ from $I$, so neither $x - b$ nor $q + 1$ wraps. It then uses $\lfloor x/b \rfloor = 1 + \lfloor (x-b)/b \rfloor$ to prove that $(x, q) \mapsto (x-b, q+1)$ preserves $I$. At exit ($x < b$), $\lfloor x/b \rfloor = 0$ gives $q = \lfloor n/p \rfloor$, matching the target result.
4. *Final assembly.* At loop entry, $x = \lfloor n/a \rfloor$ and $q = 0$. The source entry conditions and the identity $\lfloor \lfloor n/a \rfloor / b \rfloor = \lfloor n/(a \cdot b) \rfloor$ establish $I$ for this initial state. The LLM combines this initialization proof with the inductive step to prove refinement for any number of loop iterations.

Trivet checks the completed proof following § 3.1.3.

**3.3.2 Loop-Aware Counterexample Certification.** The counterexample branch extends the loop-free workflow in § 3.2. The LLM proposes a counterexample candidate with concrete bitwidths, inputs, and an execution bound that limits the loop iteration count and ensures evaluation terminates. The scaffold accepts a counterexample candidate only if the loop execution completes within this bound. Otherwise, the scaffold rejects the candidate and requests a new one. Given this candidate, Trivet reuses the deterministic stages from § 3.2.3: decidable comparison of execution outcomes, certification of the concrete candidate, and construction of a proof of non-refinement from the certified counterexample. Following the loop-free workflow in § 3.2, Trivet allows at most five counterexample generation attempts per transformation.

### 3.4 Implementation and Limitations

We build Trivet on top of the LLVM dialect in Lean-MLIR [6], a Lean 4 framework that formalizes the semantics of MLIR programs: it parses MLIR code into well-typed Lean terms with precise per-operation semantics, so that properties of MLIR programs become Lean theorems. Using MLIR's translation mechanisms [42], LLVM IR converts bidirectionally to and from the LLVM dialect without loss of semantic information [41].

However, Lean-MLIR originally modeled only a simplified subset of the LLVM dialect. To enable translation validation of real-world transformations, we extended it along three dimensions: ① refining its semantic foundations to support immediate UB and transformation preconditions, ② adding missing bit-manipulation operators (*e.g.* `ctpop`, `cttz`, `ctlz`), and ③ generalizing the framework to multiple symbolic-bitwidth parameters and multi-block transformations.

In Trivet, the LLM operates as an agent: we use the Codex CLI [46] with the `gpt-5.5` model [48] to perform proof completion in the refinement branch and counterexample proposal in the counterexample branch. Trivet itself remains a deterministic harness, accepting the agent's output only after the Lean kernel checks the resulting proof.

Our formalization currently targets integer instructions, and our workflows support loop-free, single-block transformations plus a restricted class of loop-containing ones. Extending the formalization to memory operations, floating-point arithmetic, and richer loop classes is left as future work.

## 4 Evaluation

This section presents our extensive evaluation of Trivet, demonstrating its effectiveness in automating translation validation for LLVM transformations. We organize our evaluation around four research questions:

- **RQ1:** Can Trivet effectively validate real-world valid LLVM transformations?
- **RQ2:** Can Trivet effectively refute real-world invalid LLVM transformations?
- **RQ3:** Can Trivet effectively validate and refute loop-containing LLVM transformations derived from real-world cases?
- **RQ4:** What is the impact of Trivet's scaffold on refinement and counterexample proof generation?

We intentionally separate loop-free (RQ1 and RQ2) and loop-containing transformations (RQ3) because Alive2 offers different guarantees. For loop-free transformations, Alive2 can validate them under its modeled semantics, while for loop-containing transformations, it only provides bounded guarantees due to its finite loop unrolling [38].

***Setup.*** All experiments were conducted on an Ubuntu 22.04 (64-bit) server equipped with a 32-core Intel Xeon Gold 5217 CPU @ 3.00GHz and 376 GB of RAM. We applied a time limit of two hours per transformation across all RQs. Since

**Table 2.** Dataset Composition

| Loop Structure | RQ | Validity | Bitwidth | | Total |
|---|---|---|---|---|---|
| | | | Fixed | Symbolic | |
| Loop-free | RQ1 | Valid | 30 | 30 | 60 |
| | RQ2 | Invalid | 30 | 30 | 60 |
| Loop-containing | RQ3 | Valid | 7 | 7 | 14 |
| | RQ3 | Invalid | 7 | 7 | 14 |
| Overall | – | – | 74 | 74 | 148 |

Trivet does not support `undef` for the reasons discussed in § 2.1, we ran Alive2 with `-disable-undef-input`; this configuration ensures a fair comparison because our dataset has no other mechanism for introducing `undef`. As the underlying LLM, Trivet uses `gpt-5.5` with extra-high reasoning effort.

### 4.1 Dataset

Our dataset comprises 148 real-world LLVM transformation instances from the LLVM GitHub repository. Table 2 summarizes their distribution by loop structure, validity, bitwidth, and research question.

***Loop-Free Dataset.*** The loop-free dataset contains fixed- and symbolic-bitwidth cases.

In the fixed-bitwidth portion, Alive2 proves the 15 valid cases and refutes the 15 invalid cases within its standard 10 s time limit. The remaining 15 valid cases and 15 invalid cases involve nonlinear bit-vector operations, interdependent preconditions, and relatively large but common concrete bitwidths (*e.g.* `i32` and `i64`). These structures make it challenging for SMT-based methods such as Alive2 to finish within the standard 10 s time limit [45]. We include these cases to assess whether Trivet can validate or refute transformations that are challenging for SMT-based methods.

In the symbolic-bitwidth portion, we derive each valid case by applying the transformation-generalization techniques developed in prior work [21, 43] to its fixed-bitwidth counterpart. We manually construct each invalid case from its fixed-bitwidth counterpart by replacing the concrete integer bitwidth with a symbolic one, because these techniques do not support the generalization of invalid transformations.

***Loop-Containing Dataset.*** The loop-containing dataset comprises 14 valid and 14 invalid transformations. Each group contains equal numbers of fixed- and symbolic-bitwidth cases. Each transformation replaces a three-basic-block source function with a loop-free target function with a single basic block. We derive each symbolic-bitwidth case from its fixed-bitwidth counterpart by manually replacing the concrete integer bitwidth with a symbolic one because prior transformation-generalization techniques [21, 43] do not support loop-containing transformations.

**Table 3.** Trivet versus Alive2 on 60 valid transformations.

| Stratum | $N$ | Trivet | | Alive2 | |
|---|---|---|---|---|---|
| | | Proved | Mean (s) | Proved | Mean (s) |
| Symbolic bitwidth | 30 | 30 | 722 | n/a | n/a |
| Fixed bitwidth | 30 | 30 | 647 | 20 | 206 |
| Total | 60 | 60 | 684 | 20 | 206 |

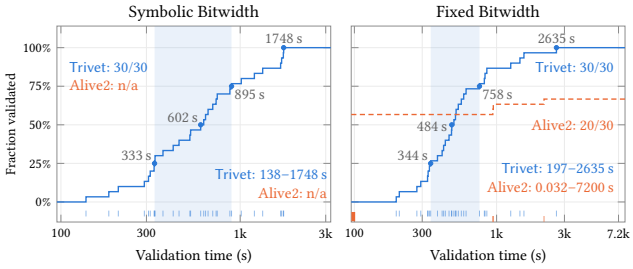


**Figure 5.** Validation time distributions for RQ1 (log scale).

### 4.2 RQ1: Validating Transformations

In RQ1, we investigate whether Trivet can establish refinement for real-world LLVM transformations and compare its validation capability with Alive2. Table 3 summarizes the aggregate results for both systems on the same 60 transformations and Figure 5 shows the validation-time distributions.

**4.2.1 Experimental Results.** Trivet validates all 60 transformations, with median and mean validation times of 522 s and 684 s, respectively. Alive2 applies only to the 30 fixed-bitwidth transformations and validates 20; the remaining 10 runs do not complete within the time limit.

For the symbolic-bitwidth cases, Trivet validates all 30 cases in 138–1748 s, with a median of 614 s and a mean of 722 s. Each generated theorem is parametric in the bitwidth and establishes refinement for every width satisfying the required constraints (*e.g.* $w \bmod 2 = 0$). Since Alive2 does not support transformations with symbolic bitwidths, Table 3 reports "n/a" for these cases.

For the fixed-bitwidth cases, Alive2 takes less than 1 s to validate each of the 15 cases, whereas Trivet validates the same cases with a median of 463 s and a mean of 487 s. In the remaining 15 challenging cases, Alive2 validates only 5 within the time limit of two hours; its successful runs have a median of 941 s and a mean of 823 s. In contrast, Trivet validates all 15 challenging cases in 207–1257 s, with a median of 492 s. Overall, Alive2 is faster on the 15 cases but validates only a subset of the 15 challenging cases, whereas Trivet validates all fixed-bitwidth cases.

**4.2.2 Result Analysis.** The symbolic-bitwidth results show that a single Trivet proof establishes refinement for all admissible widths rather than for one concrete instantiation, providing a width-independent guarantee outside Alive2's

**Table 4.** Trivet versus Alive2 on 60 invalid transformations.

| | | Trivet | | Alive2 | |
|---|---|---|---|---|---|
| **Stratum** | *N* | **Refuted** | **Mean (s)** | **Refuted** | **Mean (s)** |
| Symbolic bitwidth | 30 | 30 | 66 | n/a | n/a |
| Fixed bitwidth | 30 | 30 | 56 | 29 | 47 |
| Total | 60 | 60 | 61 | 29 | 47 |

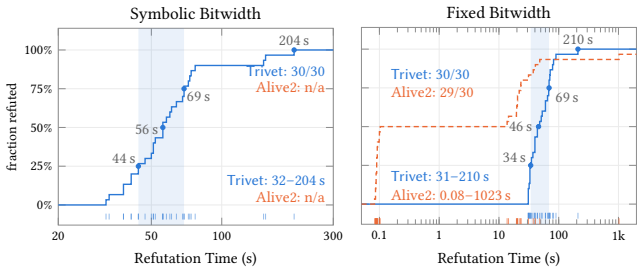


**Figure 6.** Refutation time distributions for RQ2 (log scale).

supported scope. For fixed bitwidths, Alive2 exhibits a performance cliff as proof difficulty increases. Similarly, the original Alive2 study found that extending the timeout from one to five minutes yielded less than 5% additional results [38]. By contrast, Trivet exhibits more consistent runtimes across the fixed-bitwidth cases.

The performance cliff stems primarily from two factors: Alive2's whole-function refinement formulation and the bit-blasting in the SMT solver. Alive2 encodes the complete source and target functions rather than decomposing validation into smaller independent obligations, producing increasingly difficult queries as the transformations grow more complex. Moreover, this difficulty is exacerbated by the bit-blasting mechanism, especially for transformations involving nonlinear operators (*e.g.* div, mul, and urem). Although bit-blasting enables automated reasoning via SAT, expanding $w$-bit operations into Boolean circuits obscures their algebraic structure and makes particular arithmetic operations expensive to solve [4, 14, 45]. In contrast, Trivet leverages the LLM's algebraic reasoning capabilities to decompose complex proof obligations, completing the proof step by step, thereby avoiding this performance cliff.

Prior work shows that bit-blasting can be extremely expensive at common bitwidths such as 32 bits [45]; whole-function refinement checking can further amplify this cost. For example, consider a simple transformation $\big((x/_u y) \cdot y \le x\big) \implies \text{true}$, where $/_u$ denotes unsigned integer division. Although this transformation is clearly valid under LLVM semantics, Alive2 times out on its `i32` instance after two hours, whereas Trivet validates the same `i32` instance in 339 s.

### 4.3 RQ2: Refuting Invalid Transformations

In RQ2, we investigate whether Trivet can refute real-world invalid LLVM transformations and compare its refutation capability with Alive2. Table 4 summarizes the aggregate results, while Figure 6 shows the corresponding refutation-time distributions.

**4.3.1 Experimental Results.** Trivet produces certified refutations for all 60 invalid transformations, with a mean refutation time of 61 s. Candidate generation accounts for 47 s (77%), while scaffold-based certification accounts for 14 s (23%). For the symbolic-bitwidth transformations, Trivet refutes all 30 cases in 32–204 s, with a median of 56 s and a mean of 66 s. For each case, the LLM proposes a counterexample candidate containing concrete bitwidths and input values, which the scaffold automatically certifies. For the fixed-bitwidth transformations, Trivet refutes all 30 cases in 31–210 s, with a median of 46 s and a mean of 56 s. In contrast, Alive2 refutes 29 and times out on the remaining one. Successful runs take 0.08–1023 s, with a median of 0.103 s and a mean of 47 s.

**4.3.2 Result Analysis.** The symbolic-bitwidth results show that Trivet can refute a width-parametric transformation by certifying a counterexample that instantiates both the bitwidth and inputs, extending coverage beyond Alive2's scope. For fixed bitwidths, Alive2 is faster on most cases but still exhibits a pronounced long tail and leaves one case unresolved after two hours, consistent with its behavior on valid transformations in § 4.2. In contrast, Trivet has more concentrated runtimes and refutes every case.

### 4.4 RQ3: Loop-Containing Transformations

Table 5 summarizes the experimental results for the 28 instances. Trivet provides refinement proofs for all valid cases at both fixed and symbolic bitwidths. The symbolic proofs establish correctness for all widths and inputs, with median and mean proof-generation times of 1471.2 s and 1673.5 s. The fixed-width runs have corresponding times of 1741.7 s and 1821.7 s. Fixed-width refutation certifies 6/7 invalid cases, with a median time of 211.1 s. Symbolic-width refutation certifies all 7 invalid cases, with a median of 87.5 s. Each symbolic counterexample specifies concrete inputs and a bitwidth. We omit Alive2 because it does not support symbolic bitwidths and establishes correctness only within a user-configurable unrolling bound, rather than proving the transformation correct for all loop iteration counts [38].

For the unresolved invalid transformation, Trivet exhausted its counterexample candidate attempts after 306 s without finding a certified counterexample. With $x, n, a_i$ : `i32`, the transformation replaces a loop with a multiplication:

$$a_0 = 0, \quad a_{i+1} = a_i + x, \quad \text{return } a_n \longrightarrow \text{return}(x \times n).$$

**Table 5.** RQ3 results for loop-containing transformations.

| Bitwidth | Task | $N$ | Succeeded | Median (s) | Mean (s) |
|---|---|---|---|---|---|
| Symbolic | Validation | 7 | 7 | 1471.2 | 1673.5 |
| Fixed | Validation | 7 | 7 | 1741.7 | 1821.7 |
| Symbolic | Refutation | 7 | 7 | 87.5 | 88.6 |
| Fixed | Refutation | 7 | 6 | 211.1 | 207.8 |

**Table 6.** Scaffold ablation results. Mean runtimes are in seconds for successful cases in both configurations. The column Δ is the cost reduction from Trivet$^-$ to Trivet.

| | | Successful | | Mean Runtime (s) | | | Mean Cost (USD) | | |
|---|---|---|---|---|---|---|---|---|---|
| Bitwidth | $N$ | Trivet$^-$ | Trivet | Trivet$^-$ | Trivet | Speedup | Trivet$^-$ | Trivet | Δ |
| *Refinement proofs (loop-free)* | | | | | | | | | |
| Symbolic | 30 | 24 | 30 | 3265.9 | 760.6 | 4.3× | $13.93 | $1.48 | 89.4% |
| Fixed | 30 | 23 | 30 | 2791.6 | 634.3 | 4.4× | $12.49 | $1.45 | 88.4% |
| Total | 60 | 47 | 60 | 3033.8 | 698.8 | 4.3× | $13.23 | $1.47 | 88.9% |
| *Counterexample proofs (loop-free)* | | | | | | | | | |
| Symbolic | 30 | 25 | 30 | 907.4 | 64.2 | 14.1× | $2.80 | $0.08 | 97.1% |
| Fixed | 30 | 24 | 30 | 934 | 58 | 16.1× | $3.52 | $0.08 | 97.7% |
| Total | 60 | 49 | 60 | 920.4 | 61.1 | 15.1× | $3.15 | $0.08 | 97.5% |
| *Refinement proofs (loop-containing)* | | | | | | | | | |
| Symbolic | 7 | 6 | 7 | 2784.5 | 1446.6 | 1.9 × | $12.06 | $2.76 | 77.1% |
| Fixed | 7 | 6 | 7 | 3042.1 | 1603.4 | 1.9 × | $13.35 | $3.10 | 76.8% |
| Total | 14 | 12 | 14 | 2913.3 | 1525 | 1.9 × | $12.7 | $2.93 | 76.9% |
| *Counterexample proofs (loop-containing)* | | | | | | | | | |
| Symbolic | 7 | 7 | 7 | 697.2 | 88.6 | 7.9 × | $2.20 | $0.12 | 94.5% |
| Fixed | 7 | 6 | 6 | 842.4 | 207.8 | 4.1 × | $2.40 | $0.37 | 84.6% |
| Total | 14 | 13 | 13 | 764.22 | 143.62 | 5.3 × | $2.29 | $0.24 | 89.7% |

For `x=poison`, `n=0`, the source executes no loop iterations, leaving the accumulator at its initial value of zero. The target instead evaluates `x * n`, which propagates poison. Although the arithmetic identity holds for defined operands, the rewrite introduces a dependence on `x` into the source's otherwise constant return value when `n=0`.

### 4.5 RQ4: Ablation Study

In RQ4, we assess the contribution of the scaffold by comparing Trivet with its unscaffolded variant Trivet$^-$, which requires the LLM to generate complete Lean proofs from scratch. Keeping all other experimental settings fixed and evaluating on the RQ1–RQ3 datasets, we measure proof success rate, runtime, and monetary cost in the ablation study.

**4.5.1 Success Rate and Runtime.** For loop-free transformations, Table 6 shows that Trivet completes all refinement and counterexample proofs. In contrast, Trivet$^-$ succeeds on 47 of 60 refinement cases and 49 of 60 counterexample cases. Among cases successfully completed by both configurations, Trivet is faster in every case. The speedups in mean runtime are 4.3× for refinement and 15.1× for counterexamples. For loop-containing transformations, Trivet$^-$ and Trivet prove 12 and 14 of 14 valid instances, respectively, and refute 13 and 13 of 14 invalid instances. Among cases solved by both configurations, the ratios of baseline to scaffolded mean runtime are 1.9 × for refinement and 5.3 × for counterexamples.

These results are consistent with scaffolding reducing proof generation overhead. We observe that generating proofs from scratch can be inefficient due to extensive tactic exploration and repeated proof attempts, with loose proof structure and diverse UB cases adding further complexity. In contrast, the scaffold guides the LLM through well-defined subgoals, yielding proofs with a clearer logical structure.

**4.5.2 Monetary Cost.** We report per-transformation proof-generation costs based on OpenAI's pricing as of September 2026 [47]. For loop-free transformations, mean costs decrease from $13.23 to $1.47 for the 47 valid cases and from $3.15 to $0.08 for the 49 invalid cases, corresponding to reductions of 88.9% and 97.5%, respectively. For loop-containing transformations with complete usage records, mean costs decrease from $12.7 to $2.93 for refinement proofs and from $2.29 to $0.24 for counterexample proofs.

## 5 Related Work

We review the following three areas related to Trivet.

***Translation Validation.*** Translation validation checks individual compiler runs or transformations rather than verifying the compiler implementation [1, 44, 49]. Equality-based approaches use equality saturation or normalized value graphs to check LLVM and MLIR transformations [50, 56, 61]. Alive [39] and Alive2 [38] provide SMT-based refinement checking for LLVM. AliveInLean follows this design by verifying its condition generator in Lean but still relies on an SMT solver, whereas Crellvm checks compiler-emitted proofs with a Coq-verified validator [15, 19]. Trivet instead uses an LLM to synthesize per-transformation Lean proofs checked by the kernel.

***LLMs for Compilers.*** Recent work applies LLMs to compiler optimization and debugging. For optimization development, LPO [58] discovers formally verified peephole optimizations, while LPG [21] generalizes concrete optimizations into reusable rules. Recent studies evaluate agents implementing LLVM optimizations [13, 57]. For compiler debugging, LPR and subsequent agentic work minimize bug-triggering programs [63, 66]. By contrast, Trivet uses an LLM to construct correctness or counterexample proofs for a given transformation, with Lean checking the result.

***LLMs for Interactive Theorem Proving.*** LLMs have been used to automate proof construction in interactive theorem provers. LeanDojo and Rango use retrieval augmentation for Lean and Rocq, respectively [55, 59]. COPRA combines stateful search with prover feedback, while PALM iteratively repairs generated proofs [40, 53]. Trivet applies LLM-based proof synthesis to LLVM translation validation, using a domain-specific scaffold to produce machine-checked proofs in Lean.

## 6 Conclusion

This paper presents Trivet, a framework integrating LLMs and Lean for automated LLVM translation validation. By combining deterministic scaffolds, LLM-guided proof synthesis, and Lean kernel checking, Trivet produces refinement and counterexample proofs for fixed and symbolic bitwidths, covering loop-free transformations and a restricted class of loop-containing ones. Our evaluation shows that Trivet validates transformations beyond the practical reach of Alive2, the state-of-the-art SMT-based validator, while scaffolding improves proof completion and efficiency.

## References


[1] Clark Barrett, Yi Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli, and Lenore Zuck. 2005. TVOC: a translation validator for optimizing compilers. In *Proceedings of the 17th International Conference on Computer Aided Verification* (Edinburgh, Scotland, UK) *(CAV'05)*. Springer-Verlag, Berlin, Heidelberg, 291–295. doi:10.1007/11513988_29

[2] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2025. *The SMT-LIB Standard: Version 2.7*. Technical Report. Department of Computer Science, The University of Iowa. http://www.SMT-LIB.org

[3] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. 2021. Satisfiability Modulo Theories. In *Handbook of Satisfiability* (second ed.), Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (Eds.). IOS Press, Chapter 33, 1267–1329.

[4] Max Barth, Matthias Heizmann, and Jochen Hoenicke. 2026. A lazy and modular approach to int-blasting. *Acta Inf.* 63, 2 (April 2026), 33 pages. doi:10.1007/s00236-026-00529-y

[5] Calvin Beck, Irene Yoon, Hanxi Chen, Yannick Zakowski, and Steve Zdancewic. 2024. A Two-Phase Infinite/Finite Low-Level Memory Model: Reconciling Integer–Pointer Casts, Finite Space, and undef at the LLVM IR Level of Abstraction. *Proc. ACM Program. Lang.* 8, ICFP, Article 263 (Aug. 2024), 29 pages. doi:10.1145/3674652

[6] Siddharth Bhat, Alex Keizer, Chris Hughes, Andrés Goens, and Tobias Grosser. 2024. Verifying Peephole Rewriting in SSA Compiler IRs. In *15th International Conference on Interactive Theorem Proving (ITP 2024) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 309)*, Yves Bertot, Temur Kutsia, and Michael Norrish (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 9:1–9:20. doi:10.4230/LIPIcs.ITP.2024.9

[7] Raymond T. Boute. 1992. The Euclidean definition of the functions div and mod. *ACM Trans. Program. Lang. Syst.* 14, 2 (April 1992), 127–144. doi:10.1145/128861.128862

[8] Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. 2021. GoJournal: a verified, concurrent, crash-safe journaling system. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 423–439. https://www.usenix.org/conference/osdi21/presentation/chajed

[9] Yun-Sheng Chang, Ralf Jung, Upamanyu Sharma, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2023. Verifying vMVCC, a high-performance transaction library using multi-version concurrency control. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 871–886. https://www.usenix.org/conference/osdi23/presentation/chang

[10] Leonardo De Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. 2015. The lean theorem prover (system description). In *International conference on automated deduction*. Springer, 378–388.

[11] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 625–635. doi:10.1007/978-3-030-79876-5_37

[12] Robert W Floyd. 1993. Assigning meanings to programs. In *Program Verification: Fundamental Issues in Computer Science*. Springer, 65–81.

[13] Batu Guan, Zirui Wang, and Shaohua Li. 2026. Understanding Agent-Based Patching of Compiler Missed Optimizations. arXiv:2607.02370 [cs.SE] https://arxiv.org/abs/2607.02370

[14] Liana Hadarean, Kshitij Bansal, Dejan Jovanović, Clark Barrett, and Cesare Tinelli. 2014. A Tale of Two Solvers: Eager and Lazy Approaches to Bit-Vectors. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 680–695.

[15] Jeehoon Kang, Yoonseung Kim, Youngju Song, Juneyoung Lee, Sanghoon Park, Mark Dongyeon Shin, Yonghyun Kim, Sungkeun Cho, Joonwon Choi, Chung-Kil Hur, and Kwangkeun Yi. 2018. Crellvm: verified credible compilation for LLVM. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 631–645. doi:10.1145/3192366.3192377

[16] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 75.

[17] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 216–226. doi:10.1145/2594291.2594334

[18] Hayley Leblanc, Nathan Taylor, James Bornholt, and Vijay Chidambaram. 2025. SquirrelFS: Using the Rust Compiler to Check File-System Crash Consistency. *ACM Trans. Storage* 21, 4, Article 27 (Nov. 2025), 39 pages. doi:10.1145/3769109

[19] Juneyoung Lee, Chung-Kil Hur, and Nuno P. Lopes. 2019. AliveInLean: A Verified LLVM Peephole Optimization Verifier. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 445–455.

[20] Juneyoung Lee, Dongjoo Kim, Chung-Kil Hur, and Nuno P. Lopes. 2021. An SMT Encoding of LLVM's Memory Model for Bounded Translation Validation. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II*. Springer-Verlag, Berlin, Heidelberg, 752–776. doi:10.1007/978-3-030-81688-9_35

[21] Chunhao Liao, Hongxu Xu, Xintong Zhou, Zhenyang Xu, and Chengnian Sun. 2026. Leveraging Large Language Models for Generalizing Peephole Optimizations. arXiv:2603.18477 [cs.PL] https://arxiv.org/abs/2603.18477

[22] LLVM Project. 2022. Missed optimization for -(foo < 0) in Clang when foo was previously sign-extended and in LLVM for all cases. https://github.com/llvm/llvm-project/issues/57381

[23] LLVM Project. 2025. Missed Optimization: Fold a Same-Sign Unsigned Comparison into a Zero-Sum Test. https://github.com/llvm/llvm-project/issues/166890

[24] LLVM Project. 2026. InstCombine Contributor Guide. https://llvm.org/docs/InstCombineContributorGuide.html. Accessed: 2026-04-30.

[25] LLVM Project. 2026. LLVM IR Undefined Behavior (UB) Manual. https://llvm.org/docs/UndefinedBehavior.html. Accessed: 2026-07-13.

[26] LLVM Project. 2026. LLVM IR Undefined Behavior (UB) Manual: Summary. https://llvm.org/docs/UndefinedBehavior.html#summary. Accessed: 2026-07-25.

[27] LLVM Project. 2026. LLVM Language Reference Manual: Immediate Undefined Behavior. https://llvm.org/docs/UndefinedBehavior.html#immediate-ub. Accessed: 2026-07-13.

[28] LLVM Project. 2026. LLVM Language Reference Manual: 'llvm.assume' Intrinsic. https://llvm.org/docs/LangRef.html#llvm-assume-intrinsic. Accessed: 2026-07-26.

[29] LLVM Project. 2026. LLVM Language Reference Manual: Poison Values. https://llvm.org/docs/UndefinedBehavior.html#poison-values. Accessed: 2026-07-13.

[30] LLVM Project. 2026. LLVM Language Reference Manual: Poison Values. https://llvm.org/docs/LangRef.html#poison-values. Accessed: 2026-07-25.

[31] LLVM Project. 2026. LLVM Language Reference Manual: 'select' Instruction. https://llvm.org/docs/LangRef.html#select-instruction. Accessed: 2026-09-09.

[32] LLVM Project. 2026. LLVM Language Reference Manual: Undefined Values. https://llvm.org/docs/UndefinedBehavior.html#undef-values. Accessed: 2026-07-13.

[33] LLVM Project. 2026. LLVM Language Reference Manual: Well-Defined Values. https://llvm.org/docs/LangRef.html#well-defined-values. Accessed: 2026-09-08.

[34] LLVM Project. 2026. LLVM Loop Idiom Recognize Pass Source Code. https://github.com/llvm/llvm-project/blob/main/llvm/lib/Transforms/Scalar/LoopIdiomRecognize.cpp Idiom recognizers gated on `getNumBlocks() == 1` and `getNumBackEdges() == 1`.

[35] LLVM Project. 2026. LLVM Loop Pass Manager Source Code. https://github.com/llvm/llvm-project/blob/main/llvm/include/llvm/Transforms/Scalar/LoopPassManager.h `FunctionToLoopPassAdaptor` runs `LoopSimplify` and `LCSSA` before every loop pass.

[36] LLVM Project. 2026. LLVM Loop Terminology (and Canonical Forms). https://llvm.org/docs/LoopTerminology.html#loop-simplify-form Accessed: 2026-09-03.

[37] Pedro Lobo, John McIver, George Mitenkov, Juneyoung Lee, Kirshanthan Sundararajah, and Nuno P. Lopes. 2026. Towards Removing Undef Values from LLVM IR. *Proc. ACM Program. Lang.* 10, PLDI, Article 172 (June 2026), 24 pages. doi:10.1145/3808250

[38] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 65–79. doi:10.1145/3453483.3454030

[39] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. 2015. Provably correct peephole optimizations with alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 22–32. doi:10.1145/2737924.2737965

[40] Minghai Lu, Benjamin Delaware, and Tianyi Zhang. 2024. Proof Automation with Large Language Models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 1509–1520. doi:10.1145/3691620.3695521

[41] MLIR Project. 2026. 'llvm' Dialect. https://mlir.llvm.org/docs/Dialects/LLVM/. Accessed: 2026-05-03.

[42] MLIR Project. 2026. LLVM IR Target. https://mlir.llvm.org/docs/TargetLLVMIR/. Accessed: 2026-07-22.

[43] Manasij Mukherjee and John Regehr. 2024. Hydra: Generalizing Peephole Optimizations with Program Synthesis. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 120 (April 2024), 29 pages. doi:10.1145/3649837

[44] George C. Necula. 2000. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. Association for Computing Machinery, New York, NY, USA, 83–94. doi:10.1145/349299.349314

[45] Aina Niemetz, Mathias Preiner, and Yoni Zohar. 2024. Scalable Bit-Blasting with Abstractions. In *Computer Aided Verification: 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24–27, 2024, Proceedings, Part I* (Montreal, QC, Canada). Springer-Verlag, Berlin, Heidelberg, 178–200. doi:10.1007/978-3-031-65627-9_9

[46] OpenAI. 2025. Codex CLI. https://github.com/openai/codex Accessed: 2026-09-10.

[47] OpenAI. 2026. API Pricing. https://developers.openai.com/api/docs/pricing Accessed: 2026-09-06.

[48] OpenAI. 2026. GPT-5.5 Model. https://developers.openai.com/api/docs/models/gpt-5.5 Accessed: 2026-09-06.

[49] Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation Validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '98)*. Springer-Verlag, Berlin, Heidelberg, 151–166.

[50] Michael Stepp, Ross Tate, and Sorin Lerner. 2011. Equality-based translation validator for LLVM. In *Proceedings of the 23rd International Conference on Computer Aided Verification* (Snowbird, UT) *(CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 737–742.

[51] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward understanding compiler bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 294–305. doi:10.1145/2931037.2931074

[52] The Rocq Development Team. 2025. *The Rocq Prover*. doi:10.5281/zenodo.15149629

[53] Amitayush Thakur, George Tsoukalas, Yeming Wen, Jimmy Xin, and Swarat Chaudhuri. 2024. An In-Context Learning Agent for Formal Theorem-Proving. In *First Conference on Language Modeling*. https://openreview.net/forum?id=V7HRrxXUhN

[54] The Lean Project. [n. d.]. standard-axioms. The Lean Language Reference. https://lean-lang.org/doc/reference/latest/Axioms/#standard-axioms Accessed 2026-07-28.

[55] Kyle Thompson, Nuno Saavedra, Pedro Carrott, Kevin Fisher, Alex Sanchez-Stern, Yuriy Brun, João F. Ferreira, Sorin Lerner, and Emily First. 2025. Rango: Adaptive Retrieval-Augmented Proving for Automated Software Verification. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (Ottawa, Ontario, Canada) *(ICSE '25)*. IEEE Press, 347–359. doi:10.1109/ICSE55347.2025.00161

[56] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. 2011. Evaluating value-graph translation validation for LLVM. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 295–305. doi:10.1145/1993498.1993533

[57] Hongxu Xu, Chunhao Liao, Xintong Zhou, and Chengnian Sun. 2026. Can Coding Agents Implement Missed Compiler Optimizations? Evaluating LLM Agents on LLVM Peephole Optimizations. arXiv:2607.02684 [cs.SE] https://arxiv.org/abs/2607.02684

[58] Zhenyang Xu, Hongxu Xu, Yongqiang Tian, Xintong Zhou, and Chengnian Sun. 2026. LPO: Discovering Missed Peephole Optimizations with Large Language Models. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (USA) *(ASPLOS '26)*. Association for Computing Machinery, New York, NY, USA, 1136–1150. doi:10.1145/3779212.3790184

[59] Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J Prenger, and Animashree Anandkumar. 2023. LeanDojo: Theorem Proving with Retrieval-Augmented Language Models. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 21573–21612.

doi:10.52202/075280-0944

[60] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. doi:10.1145/1993498.1993532

[61] Jiaqi Yin, Zhan Song, Nicolas Bohm Agostini, Antonino Tumeo, and Cunxi Yu. 2025. HEC: equivalence verification checking for code transformation via equality saturation. In *Proceedings of the 2025 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '25)*. USENIX Association, USA, Article 69, 16 pages.

[62] Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. 2021. Modular, compositional, and executable formal semantics for LLVM IR. *Proc. ACM Program. Lang.* 5, ICFP, Article 67 (Aug. 2021), 30 pages. doi:10.1145/3473572

[63] Mengxiao Zhang, Yongqiang Tian, Zhenyang Xu, Yiwen Dong, Shin Hwei Tan, and Chengnian Sun. 2024. LPR: Large Language Models-Aided Program Reduction. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 261–273. doi:10.1145/3650212.3652126

[64] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 427–440. doi:10.1145/2103656.2103709

[65] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2013. Formal verification of SSA-based optimizations for LLVM. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 175–186. doi:10.1145/2491956.2462164

[66] Xintong Zhou, Hongxu Xu, Chunhao Liao, Puzhuo Liu, Yongqiang Tian, and Chengnian Sun. 2026. Semantic-aware and Self-improving Program Reduction via Agentic Large Language Models. arXiv:2607.03766 [cs.SE] https://arxiv.org/abs/2607.03766