

Semantic Reification: A New Paradigm for Random Program Generation

KAVYA CHOPRA*[†], ETH Zurich, Switzerland

CONG LI[†], ETH Zurich, Switzerland

THODORIS SOTIROPOULOS, ETH Zurich, Switzerland

ZHENDONG SU, ETH Zurich, Switzerland

We introduce *semantic reification*, a novel paradigm for random program generation that centers on program semantics rather than syntax. Our key insight is to reformulate random program generation to capture two types of program semantics: (1) compile-time semantics (what a program can do), represented by the control flow graph (CFG), and (2) runtime semantics (what a program actually does), represented by execution paths within the CFG. For any CFG and any execution path on it, semantic reification constructs a program guaranteed to be *well-behaved* with respect to a specific input and output. This means that when executed with this input, the program deterministically follows the designated execution path to produce the expected output. This paradigm differs from existing work by supporting arbitrary control flow such as unbounded loops and irreducible regions, while still ensuring that the generated programs are semantically correct and terminating. We develop a practical realization of this paradigm. First, we introduce *symbolic function reification* that integrates a lightweight form of symbolic execution into the generation process to generate an individual, leaf function (i.e., a function that is free of function calls). Each leaf function satisfies the constraints of a given CFG and a selected execution path. Second, we compose multiple leaf functions into a larger, more complex program via semantics-preserving peephole rewriting, guided by an arbitrary call graph. Over five months, our implementation for C compilers, REIFY, has uncovered 59 bugs in GCC and LLVM (57 confirmed, 27 fixed), 24 of which are long-latent. Among them, 36 are wrong-code bugs, many are high-priority issues, and most of them involve semantic characteristics overlooked by existing tools. We believe semantic reification opens new directions for research beyond compilers, such as validating debuggers, analyzers, and verifiers.

CCS Concepts: • **Software and its engineering** → **Compilers; Software testing and debugging**.

Additional Key Words and Phrases: random program generation, compiler testing, compilers

ACM Reference Format:

Kavya Chopra, Cong Li, Thodoris Sotiropoulos, and Zhendong Su. 2026. Semantic Reification: A New Paradigm for Random Program Generation. *Proc. ACM Program. Lang.* 10, PLDI, Article 190 (June 2026), 30 pages. <https://doi.org/10.1145/3808268>

1 Introduction

Compilers are among the most critical pieces of systems software. All applications, regardless of their purpose or complexity, rely on compilers to function correctly. Therefore, compilers must be efficient, effective, and reliable. Over the years, compiler engineers and researchers have devoted

*Kavya initiated this work during her internship at ETH Zurich and as part of her degree program at IIT Delhi.

[†]Both authors contributed equally.

Authors' Contact Information: [Kavya Chopra](mailto:kavya.chopra.iitd@gmail.com), ETH Zurich, Zurich, Switzerland, kavya.chopra.iitd@gmail.com; [Cong Li](mailto:cong.li@inf.ethz.ch), ETH Zurich, Zurich, Switzerland, cong.li@inf.ethz.ch; [Thodoris Sotiropoulos](mailto:thodoros.sotiropoulos@inf.ethz.ch), ETH Zurich, Zurich, Switzerland, thodoros.sotiropoulos@inf.ethz.ch; [Zhendong Su](mailto:zhendong.su@inf.ethz.ch), ETH Zurich, Zurich, Switzerland, zhendong.su@inf.ethz.ch.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 2475-1421/2026/6-ART190

<https://doi.org/10.1145/3808268>

substantial effort to achieving these goals, with *random program generation* emerging as a prominent approach to improving compiler reliability. State-of-the-art random program generators (or RPGs), such as Csmith [44], YARPGen [23, 24], and others [2, 9, 11, 28, 41], have achieved remarkable success and gained widespread adoption in both academia and industry.

However, despite more than a decade of testing with existing RPGs, real-world compilers (e.g., GCC and LLVM) still harbor numerous undiscovered bugs. One primary reason is that existing RPGs follow a common paradigm, which we call *syntactic reification*. Syntactic reification applies production rules from a context-free grammar to ensure that the generated programs are valid and structurally diverse. Yet, to maintain semantic correctness in practice, it must restrict the structures of programs it generates. For example, to avoid undefined behavior (UB, e.g., signed overflows) [4], Csmith eliminates potentially unsafe operations from loops via conservative static analyses, and inserts runtime wrappers such as $x!=0 ? z/x : z/1$ into the generated programs. YARPGen enforces that all iterations of a loop exhibit identical runtime semantics, that is, execute identically. Neither tool allows modifying loop induction variables or loop bounds inside the loop body, as such modification makes it challenging to guarantee termination and to avoid UB [23, 24, 28, 44]. These restrictions limit the generation of rich semantic behaviors, such as those involving unbounded loops (i.e., loops without statically known bounds such as literal, constant, or compiler-inferred bounds), irreducible loops (i.e., loops without a single entry point that controls all paths into the loop), or other complex control-flow structures. Program mutators such as Creal [21] can partially mitigate this by injecting real-world code, but they offer no semantic guarantees, and thereby many of the generated programs are still not UB-free.

Semantic reification. To address these limitations, we introduce *semantic reification*, a new paradigm for random program generation. Unlike syntactic reification, which operates primarily on syntax, semantic reification centers on program semantics. We distinguish between two kinds of semantics: compile-time semantics (what a program can do) and runtime semantics (what a program actually does). The key insight is reformulating random program generation to capture both forms of semantics as follows. Given an *arbitrary* control flow graph (CFG) skeleton g to capture compile-time semantics and an *arbitrary* entry-to-exit path π (that we call execution path or EP) within g to capture runtime semantics, it produces a program P , input i , and output o , satisfying the following criteria: (1) P is both syntactically and semantically correct for i ; (2) g corresponds to the CFG of P ; and (3) executing $P(i)$ deterministically follows π and produces o .

This reformulation is motivated by two reasons. First, although runtime semantics are fixed for a given input, compilers must reason about all possible executions when performing optimizations. Semantic reification thus exposes compiler bugs that arise from incorrect reasoning about control flow or data flow, while still guaranteeing that every generated program behaves deterministically and remains free of UB for the given input. Second, allowing arbitrary CFGs and EPs results in more complex data flows. This further enriches the semantic behaviors available for compiler optimizations. In summary, semantic reification differs from existing RPGs in three ways: (1) it inherently supports arbitrary control flow, including unbounded loops and irreducible regions; (2) in the presence of them, it further ensures that a generated program is both well-defined and guaranteed to terminate under a generated input; and (3) by producing an expected output, it also enables direct validation of compiler correctness without relying on pseudo-oracles [3, 27].

Practical realization. How to realize semantic reification in practice? Given a CFG g and an EP π , our realization first populates each basic block in g with random statements and conditional/unconditional jumps that respect the structure of g . It then leverages *symbolic execution* [15] to derive a path condition and compute an input i that forces the program to follow the selected path π to produce o . Importantly, this symbolic execution step explores only the single EP π , rather than all

possible EPs in P . Therefore, it does not cause the classic path explosion problem encountered in traditional symbolic execution.

Nevertheless, the aforementioned algorithmic sketch still incurs substantial overhead when generating large programs with numerous functions and function calls, because symbolic execution creates a separate copy of each function for every invocation. To have a practical realization, we separate individual function generation, which we call *symbolic function reification*, from whole program generation: the former generates compact, *leaf functions* that do not call other functions [7], while the latter rewrites these functions with inter-procedural function calls which combine them into more complex programs using arbitrary call graphs (CGs).

Symbolic function reification. To produce a leaf function f , we begin by generating a random CFG sketch g and sampling a random EP π from g . Each basic block in g is then populated with a sequence of statements that are free of function calls. Each block ends with a jump that connects it to its successors, ensuring consistency with the structure of g . Notably, every statement initially contains *symbols* (such as $s1$ in statement $z = s1*x + s2/y - s3$;) whose values are not yet determined. To assign concrete values, our approach performs symbolic execution along the selected path π , collecting constraints such as path constraints (to enforce the terminating traversal of π) and well-definedness constraints (to ensure the absence of UB). In this way, f 's control flow and data flow are reified through the symbols and their associated constraints. Finally, the collected constraints are solved using an SMT solver to obtain the concrete input i and output o , resulting in a function where all symbols are replaced by concrete values consistent with g and π .

Whole-program generation. The above process yields a set of independent functions. To construct a complete program P , we first generate a random CG that defines caller-callee relationships among some of these functions. For each caller-callee pair in the CG, the caller function is then modified to invoke the callee f with the generated input i and output o through semantics-preserving peephole rewriting [18, 21]: an arbitrary constant literal c in the caller is replaced with the expression $f(i) + (c - o)$. This ensures that the rewriting expression evaluates to the same value as the original constant while establishing an inter-procedural call.

Experimental results. We developed REIFY, an RPG for C. Over the past five months, we used REIFY to validate GCC and Clang/LLVM. We reported a total of 59 bugs, with 57 confirmed and 27 already fixed. Among these, 24 bugs have remained undetected for more than three major versions. 36 bugs are wrong-code bugs that violate REIFY's oracle of guaranteed termination or expected output. In particular, these wrong-code bugs manifest as execution hangs, runtime failures such as segmentation faults, or incorrect output. Many of these are high-priority issues, with 17/39 confirmed GCC bugs ranked P1–P2 and 15/18 LLVM bugs resolved within 15 days.

Contributions. We make the following major contributions:

- We introduce a novel paradigm for random program generation called *semantic reification*, which centers the generation process around program semantics by design.
- We develop a practical and effective realization of semantic reification: *symbolic function reification* is used to generate compact individual functions, while peephole rewriting is employed to compose large programs.
- We implement a tool REIFY for C, which led to the discovery of 59 bugs in GCC and LLVM. Notably, 50 out of the 59 bugs discovered by REIFY could not have been detected by existing random program generators, as they are triggered by programs exhibiting unbounded loops or irreducible regions, which lie outside the generation scope of existing tools. REIFY is publicly available on GitHub: <https://github.com/conngli/Reify>.

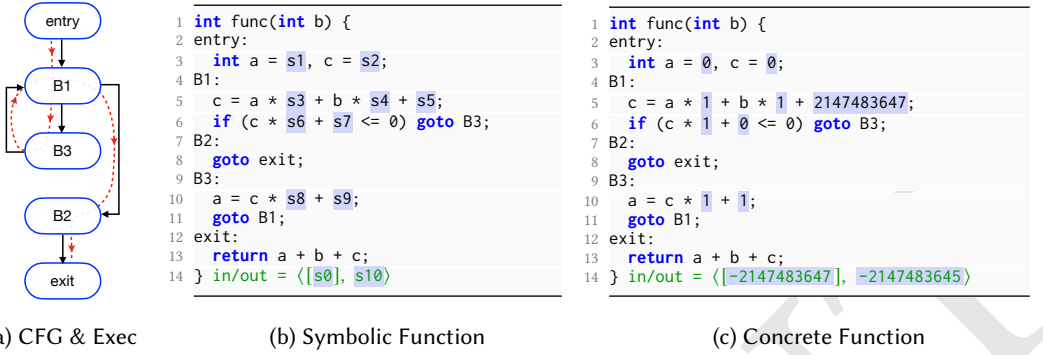


Fig. 1. GCC-119690: A program generated by our approach that triggers a long-standing bug in GCC. It illustrates how the approach incrementally constructs the final program: from generating a CFG and an execution path (EP, in red) to producing a concrete program that structurally follows this CFG and the EP.

2 Illustrative Example and Overview

We motivate our approach by walking through a concrete bug found in GCC.

GCC example. Consider the C program in Figure 1c and its CFG shown in Figure 1a. The code defines a function `func`, which takes a parameter `b`, and consists of several basic blocks connected via `goto` statements; two local variables `a` and `c` are defined in `entry`. Basic blocks `B1` and `B3` form a loop (lines 6 and 11) that terminates once the condition on line 6 becomes false. When calling `func(-2147483647)`, the execution follows `entry` \rightarrow `B1` \rightarrow `B3` \rightarrow `B1` \rightarrow `B2` \rightarrow `exit` (indicated by a red directed dashed line in Figure 1a) and terminates with `a=1`, `b=-2147483647`, and `c=1`.

When compiled with GCC 15 using the `-O3` option (i.e., at the highest optimization level), the function produces `-2147483646`, which is *different* from the expected result `-2147483645` given by the sum of `a`, `b`, and `c`. This discrepancy occurs because, in the optimized program, the state of `a` in the `exit` block of `func` is incorrectly calculated as `a=0` rather than `a=1`. The issue arises from a long-standing loop optimization bug in GCC introduced nearly *eight* years ago, which causes the compiler to generate incorrect code when using the `-O3` optimization level: the SCEV (Scalar Evolution) pass computes incorrect evolution results for PHI values of the loop `B1` \leftrightarrow `B3`, causing the IVOPT (Induction Variable Optimization) pass to select an incorrect loop induction variable.

2.1 Challenges

Why is it difficult for existing RPGs to produce programs like Figure 1c and to discover the bug?

C1: Arbitrary control flow. A critical factor that triggers the bug in Figure 1c is the presence of an *unbounded loop* `B1` \leftrightarrow `B3`, where (1) the loop bound `c` gets involved in the loop body (line 5), and (2) `c` is also the loop induction variable. Therefore, the number of loop iterations cannot be determined in advance by the compiler. Existing RPGs adopt a conservative approach, producing *only* loops with a fixed or predictable bound, such as `for (i=0; i<j; i++)` where the induction variable `i` and the bound `j` are *not* updated in the loop body [23, 24, 28, 44]. This restricts the diversity of the semantic behaviors generated, including data and control flow. Consequently, they fail to explore unbounded and irreducible loops, and thus insufficiently exercise complex loop optimizations.

C2: Guaranteed semantics and termination. One might wonder why existing RPGs do not simply allow arbitrary control flow including unbounded loops. The reason is that once this restriction is lifted, the generated programs often fail to terminate, making it impossible to observe their outputs and verify them against the expected behavior. Even when termination is ensured,

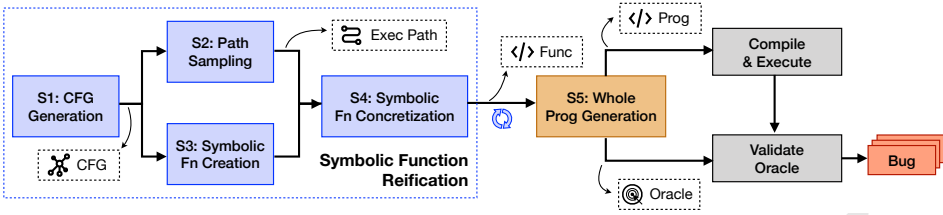


Fig. 2. An overview of our practical realization for finding compiler bugs via semantic reification.

RPGs must also guarantee that generated programs are well-defined, such as free of undefined behavior (UB). For example, Figure 1c includes arithmetic expressions consisting of additions and multiplications of 32-bit integers (e.g., line 5), which are prone to the signed overflow UB. When UB presents, any result is considered acceptable, and thus, there is no ground truth to check the program’s output against. To avoid UB, existing RPGs follow various conservative strategies as mentioned previously [23, 24, 28, 44], limiting the expressiveness of the generated programs and may prevent optimization opportunities. In fact, wrapping the arithmetic expressions of Figure 1c (lines 5, 6, 10) with safe guards (as done by Csmith) hinders the manifestation of the GCC bug.

C3: Established test oracle. Even with an RPG capable of producing code with arbitrary control flow and well-defined semantics, the test oracle problem [42] remains. A test oracle defines the expected behavior of a program against which its actual output is compared. Existing approaches address this through differential testing [27], where the behavior of a program is validated against that produced by another equivalent compiler (e.g., Clang). While effective, this approach is limited to languages with multiple implementations. Other approaches restrict to loop-free programs to establish the expected output while generating programs [28].

2.2 Overview

The goal of *semantic reification* (formalized in Section 3) is to address these challenges by generating programs with *arbitrary control flow* and potentially unbounded and irreducible loops, while (1) ensuring they are still *well-behaved*, i.e., free of UB and guaranteed to terminate on given inputs, and (2) defining an *oracle* that captures their expected behavior. Figure 2 presents an overview of our realization for semantic reification. We illustrate it using the GCC example (Figure 1c) and expand the technical details in Section 4. It consists of the following steps.

S1: CFG generation. The approach begins by generating a random control flow graph (CFG) with five basic blocks: Figure 1a. The CFG includes a cycle $B1 \leftrightarrow B3$ that corresponds to a reducible loop.

S2: Path sampling. Our approach then samples an execution path (or EP) $\text{entry} \rightarrow B1 \rightarrow B3 \rightarrow B1 \rightarrow B2 \rightarrow \text{exit}$ (indicated with dashed red lines) from the CFG. This path corresponds to a concrete EP, starting at its entry node and ending at its exit node. Notably, the path includes repeated nodes, i.e., node B1. This indicates that the loop $B1 \leftrightarrow B3$ is executed with two iterations.

S3: Symbolic function creation. The next step populates the CFG into a *symbolic function* displayed in Figure 1b. It is a leaf function that does not contain any function calls. Each basic block consists of symbolic statements such as variable declarations (line 3) and assignments (e.g., line 5), followed by a jump statement (e.g., lines 6 and 8) connecting it to subsequent blocks. These statements use *symbols* (highlighted in light blue), which represent concrete values that are not yet known or determined. The symbols s_0 and s_{10} denote the input and output of `func`, respectively.

S4: Symbolic function concretization. This step concretizes the symbolic function into the concrete function in Figure 1c by assigning each symbol with a concrete value (highlighted in light

blue). Our approach leverages *symbolic execution* to encode path conditions, computation, and well-definedness of all operations as constraints along the sampled EP. Through symbolic execution, we obtain the input-output pair for the function: when executed with $s_0 = -2147483647$, it produces $s_{10} = -2147483645$. Section A in our supplementary material lists all encoded constraints.

S5: Whole-program generation. Steps 1–4 constitute the process of *symbolic function reification*, which generates an individual leaf function. The process is independently applied multiple times to generate several functions. We then combine them into a whole program by creating a random call graph (CG) and applying semantics-preserving peephole rewriting according to the generated input and output. For example, a constant -2147483647 in a generated caller function can be rewritten by $\text{func}(-2147483647) - 2$, as $\text{func}(-2147483647)$ returns -2147483645 . This rewrite establishes an inter-procedural call relationship, while preserving the value of the original constant.

Compiler validation. Once our approach generates a concrete program, it compiles and executes it using the generated input. It then validates whether the program output matches the generated output. In the example of Figure 1c, compiling the program with -03 and running it yields -2147483646 , which differs from the expected value -2147483645 . When such mismatches occur, our approach reports a potential compiler bug.

3 Semantic Reification

The main idea of semantic reification to capture program semantics through CFGs and EPs. Although alternative representations (e.g., abstract syntax trees, or sea-of-nodes representations) exist, CFGs provide the most natural representation of control flow, and are the foundation for many data-flow analyses. Moreover, CFGs are the predominant intermediate representation (IR) in modern optimizing compilers. For example, both GCC and LLVM employ CFG-based IRs, where each function comprises a sequence of basic blocks [6, 25], and V8’s end-tier optimizing compiler has reverted to a CFG-based IR from a sea-of-nodes IR [39].

Definition 3.1 (Control Flow Graph). A CFG $g = \langle \alpha, \omega, B, J \rangle$ of a program P is a directed graph, where: $\alpha \in B$ is the entry block; $\omega \in B$ is the exit block; B is a set of basic blocks; and J is a set of jumps $J \subseteq B \times B \times T$, with $T = \{f \mid f \in P\} \cup \{\text{call}, \text{ret}\}$ a set of labels that denote the type of the jump. An edge $b_1 \xrightarrow{f} b_2 \in J$ represents a conditional or unconditional intra-procedural transfer of control from $b_1 \in B$ to $b_2 \in B$ within function $f \in P$, whereas the labels *call* and *ret* indicate inter-procedural function call and return, respectively.

According to Definition 3.1, a CFG in this paper is a skeletal structure that does not contain any associated program statements. It also requires a program to have one entry and one exit.

Definition 3.2 (Execution Path). Let $g = \langle \alpha, \omega, B, J \rangle$ be a CFG. An execution path (or execution or EP) $\pi = [b_1, b_2, \dots, b_{|\pi|}]$ on the CFG g is a path from the entry block to the exit block, where (1) $b_1 = \alpha$, (2) $b_{|\pi|} = \omega$, and (3) $\forall i \in \{1, 2, \dots, |\pi| - 1\}. b_i \xrightarrow{t} b_{i+1} \in J$, for $t \in T$.

According to Definition 3.2, a path on a CFG is considered an EP only if it starts at the entry node and transitively reaches its exit node.

Definition 3.3 (Semantic Reification). Let G be a grammar, and let g be an arbitrary CFG along with an EP π within g . Semantic reification, denoted as $\mathbb{S}\mathbb{R}_G$, (1) produces a *syntactically* and *semantically* valid program P , (2) an input $i \in \text{dom}(P)$, and (3) an output o such that:

$$\langle P, i, o \rangle \in \mathbb{S}\mathbb{R}_G(g, \pi) \rightarrow (\text{valid}(P, G) \wedge \text{cfg}(P) = g \wedge \text{exec}(P, i) = \langle o, \pi \rangle)$$

where (1) $\text{valid}(P, G)$ indicates that P is derived from G , (2) $\text{cfg}(P) = g$ implies that g is P 's CFG, and (3) $\text{exec}(P, i)$ executes P with i to obtain the output and the corresponding EP, i.e., the sequence of basic blocks visited during the execution.

Based on Definition 3.3, semantic reification starts with a CFG g , which encodes the program's compile-time semantics, and reifies the runtime semantics of a specific EP π as follows. First, it generates a program P such that its control flow matches the structure of g . Then, semantic reification determines an input i and an output o such that, when P is executed with i , it follows the exact path π within its CFG and produces the output o .

Design principles. We design semantic reification as a *complement* of the existing program generation techniques that rely on syntactic reification [13, 23, 24, 44]. Semantic reification does *not* mean to cover the entire design space of semantics-aware test generation. Instead, it describes a specific paradigm: semantic properties are made explicit and treated as first-class generation targets, and concrete programs are then constructed to reify those properties. Specifically, semantic reification addresses the challenges and limitations of syntactic reification (Section 2.1), as follows.

- *Arbitrary control flow.* We address this challenge by allowing arbitrary CFGs. In particular, $\mathbb{S}\mathbb{R}_G$ ensures that the generated program P can exhibit any control-flow shape permitted by a given CFG g , including unbounded loops and irreducible regions. As a result, programs produced through $\mathbb{S}\mathbb{R}_G$ can exercise compiler optimizations more thoroughly.
- *Guaranteed semantics and termination.* We address this challenge by introducing the EP π . $\mathbb{S}\mathbb{R}_G$ guarantees that executing P with input i *deterministically* terminates while following the EP π . This requires that P is well defined and free of UB on i , while π ensures that P contains no non-terminating behavior. Consequently, although a single π is specified, compilers must account for all possible EPs.
- *Established test oracle.* We address this challenge by providing the oracle o along with P and i . This enables the direct detection of wrong-code bugs, without relying on pseudo-oracles that are error prone to UB, such as differential or metamorphic testing.

Realizing semantic reification. An $\mathbb{S}\mathbb{R}_G$ -based RPG can be viewed as solving three coupled subproblems: structural reification, i.e., constructing P matching g ; path reification, i.e., ensuring execution follows π ; and behavioral reification, i.e., realizing valid i and o . This decomposition involves several implementation choices depending on the order in which these subproblems are resolved. For example, one may apply programming-by-example [12] techniques, such as enumerative [10] or template-based [35] synthesis, to search for programs consistent with both g and $\langle i, o \rangle$. However, such approaches typically incur prohibitive synthesis cost. Alternatively, one may begin from a concrete input i and incrementally instantiate concrete statements along π , updating program state so that path feasibility is guaranteed by construction; however, early concrete input choices can render later branch conditions (especially loop conditions) unsatisfiable, requiring nontrivial backtracking to ensure the feasibility of path π . Therefore, our realization adopts a practical, middle ground through single-path symbolic execution.

4 Program Generation by Semantic Reification

At a high level, our realization proceeds as follows. First, we generate a random CFG and sample an EP within it. Next, using the generated CFG, we construct a symbolic program that corresponds to the generated CFG. This is done by populating every basic block in a CFG using symbolic statements and expressions by expanding production rules of a grammar. These symbols indicate that the values in the program are not yet known. Finally, using a lightweight form of *symbolic execution*, we assign concrete values to these symbols according to the constraints imposed by the

```

1 int f4(int d) {
2   if (d > 0) { return d - 1; }
3   else { return d + 1; }
4 }
5 int f3(int c) { return f4(c); }
6 int f2(int b) { return f4(b); }
7 void f1(int a) {
8   f2(s0); f3(s1);
9 }

```

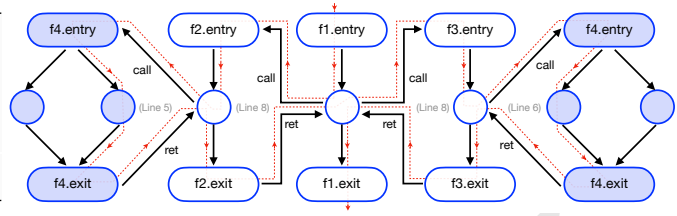


Fig. 3. An example symbolic program with multiple function calls and its inter-procedural CFG. At lines 5 and 6, the intra-procedural CFG of f_4 is inlined once at each invocation (see highlighted nodes).

selected path. This symbolic execution step yields (1) input values that cause the program to follow the chosen path, and (2) produces the corresponding output when executed under those inputs.

CFG granularity. A critical factor to determine while implementing the aforementioned high-level process is the *granularity* of CFG. One straightforward approach is to construct a single *inter-procedural* CFG (per Definition 3.1) and sample an EP through it. However, such an approach is inefficient: the sampled path will include nodes of *intra-procedural* CFGs of callee functions, which must be *cloned* and *inlined* into the caller’s CFG at each call site. This leads to an increase in constraint size during symbolic execution, since the constraints of each inlined function must be encoded once per invocation. For example, in Figure 3, the function f_4 is invoked twice along a single path (in red directed dashed lines), which requires two distinct encodings of its constraints. This is because each invocation of f_4 receives different inputs and produces different outputs (lines 5, 6, 8). The constraints generated for one invocation at a given call site *cannot* be reused for another, leading to rapid growth in the overall constraint size. Notably, this overhead significantly increases when dealing with programs that contain loops.

To address this challenge, we separate individual function generation from the generation of whole programs that include inter-procedural function calls. We first operate on multiple individual *intra-procedural* CFGs to generate leaf functions, a process that we call *semantic function reification* (Section 4.2). Then we link the individual functions via a call graph (CG) through rewriting leaf functions with inter-procedural calls (Section 4.3). This leads to efficiency and flexibility as the constraints of each generated function are encoded and solved at most once.

4.1 The SYMLANG Language & Preliminary Definitions

Before presenting the technical details, we first introduce a CFG-based symbolic language called SYMLANG. Similar to the widely used WHILE language [30], SYMLANG is intentionally minimal and designed to illustrate the key concepts of our approach. However, unlike WHILE, SYMLANG is explicitly CFG-based, with all loops implemented using `goto` statements rather than `while` constructs. While `goto` is discouraged stylistically in some contexts, `goto`-like control-flow patterns are *not* rare from a compiler’s perspective: they arise frequently in systems code (e.g., error-handling/cleanup paths), and more importantly, are fundamental in compiler IRs and machine code, where arbitrary CFGs are the norm. Compilers must perform correct analysis and transformation based on them.

SYMLANG. Figure 4a shows SYMLANG’s context-free grammar in extended BNF. SYMLANG is based on symbols and symbolic entities. A symbol $s \in Symbol$ represents any potential constant, whose value is not yet known. Symbolic entities are code constructs that involve symbols, including symbolic expressions, symbolic statements, and symbolic functions. Function f corresponds to a single symbolic leaf function. This reflects our strategy (Section 4.2): instead of generating directly a whole program that contains inter-procedural calls, we first generate individual leaf functions

<i>Function</i>	$f \rightarrow b^+$	$\llbracket s \rrbracket = M(s), s \in M$	<i>Symbol</i>
<i>Basic block</i>	$b \rightarrow L : \sigma^+$	$\llbracket s \rrbracket = \text{any const}, s \notin M$	
<i>Statement</i>	$\sigma \rightarrow v := \varepsilon \mid \text{goto } L \mid$ $\text{if}(\varepsilon) \text{ goto } L_1 \text{ else goto } L_2$	$\llbracket v \otimes s \rrbracket = v \otimes \llbracket s \rrbracket$	<i>Expr Term</i>
<i>Expression</i>	$\varepsilon \rightarrow \varepsilon \oplus \tau \mid \tau$	$\llbracket \varepsilon \oplus \tau \rrbracket = \llbracket \varepsilon \rrbracket \oplus \llbracket \tau \rrbracket$	<i>Expression</i>
<i>Expr Term</i>	$\tau \rightarrow v \otimes s \mid s$	$\llbracket v := s \rrbracket = v := \llbracket s \rrbracket$	<i>Statement</i>
<i>Symbol</i>	$s \rightarrow$ the set of all symbols	$\llbracket \text{goto } L \rrbracket = \text{goto } L$	
<i>Variable</i>	$v \rightarrow$ the set of variable names	$\llbracket \text{if}(\varepsilon) \text{ goto } L_1$	$= \text{if}(\llbracket \varepsilon \rrbracket \neq 0) \text{ goto } L_1$
<i>Label</i>	$L \rightarrow$ the set of label names	$\text{else goto } L_2 \rrbracket = \text{else goto } L_2$	

(a) Extended BNF Grammar

(b) Concretization Rules

Fig. 4. Grammar and concretization rules for our symbolic language SYMLANG. \oplus denotes additive operators “+” or “−”, \otimes extends \oplus to include multiplicative operators “*”, “/”, or “%”. On the right side, M represents the model; φ is an SMT formula encoding the constraints over symbols; $\llbracket \cdot \rrbracket$ is short for the concretization function $\llbracket \cdot \rrbracket_{\varphi}^M$. Concretizing a program, function, and basic block is straightforward and thus is omitted here.

independent of each other. Each symbolic function includes a sequence of basic blocks. A basic block has a unique label L followed by a sequence of symbolic statements, such as variable declarations and assignments, unconditional **goto** statements, or **goto** statements guarded by conditions. **goto** statements are used to transfer control to other basic blocks, while an assignment statement $v := \varepsilon$ assigns the side-effect-free symbolic expression ε to the variable v . Expressions in SYMLANG are either binary operations (denoted by the \oplus symbol) or terms, that is, variables, symbols, and their combination. For a term $v \otimes s$, it is also possible to replace s with random concrete values or any variable, leaving only the input parameters of f as symbolic. However, this design greatly restricts the solution space, often resulting in nonlinear and unsatisfiable constraints.

Concretization. We concretize a symbolic entity x into a concrete entity x^* by a concretization function denoted as $\llbracket x \rrbracket_{\varphi}^M$, with the following meaning. Given (1) a logical formula φ that encodes the constraints over the symbols defined in x , and (2) a model M that assigns concrete values to symbols such that M satisfies φ , denoted as $M \models \varphi$, the concretization function $\llbracket x \rrbracket_{\varphi}^M$ produces its concrete version x^* in which every occurrence of a symbol s is replaced by its concrete value $M(s)$ given by the model M . Starting from a symbolic function, $\llbracket \cdot \rrbracket_{\varphi}^M$ is recursively applied to all blocks, statements, expressions, and terms, through the respective rules listed in Figure 4b.

4.2 Symbolic Function Reification

Symbolic function reification generates leaf functions that contain no calls to other functions. In this process, a symbolic function f is constructed from a given intra-procedural CFG of arbitrary shape. The desired semantic properties of f are encoded as a formula φ – a set of constraints – over symbols. Solving these constraints yields a model M , which is then used to concretize f . This process leads to (1) a concrete function $f^* = \llbracket f \rrbracket_{\varphi}^M$, (2) an input i , and (3) an expected output o .

S1: CFG generation. A valid intra-procedural CFG $g = \langle \alpha, \omega, B, J \rangle$ is constructed as follows.

- (1) *Create basic blocks.* Generate N (configured by the user) basic blocks and add them to the set B .
- (2) *Add control-flow edges.* Connect the blocks in B by creating random directed edges with the same label f and add them to the set J . Each block must have at most two successors, i.e., $\forall b \in B. \text{outdeg}(b) \leq 2$, representing the true and false branches of a conditional jump.
- (3) *Create the entry block.* Create a unique entry block α and connect it to a basic block $b \in B$ chosen at random. Add the corresponding f -labeled edges to J .

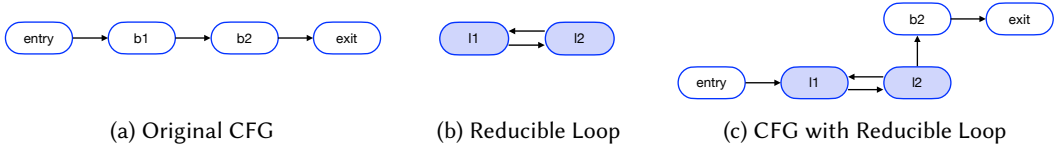


Fig. 5. A minimal example demonstrating the process of injecting reducible loops into a CFG at basic block b_1 . Highlighted nodes correspond to the inserted nodes that form a reducible loop.

- (4) *Create the exit block and identify all blocks reachable from α .* Create a unique exit block ω . Then identify all blocks that are reachable from the entry, including the entry block itself. Keep these reachable blocks in a set $R = \{\text{reachable}(\alpha, b) \mid b \in B\} \cup \{\alpha\}$.
 - (5) *Connect to the exit.* Randomly select one or more blocks from R and connect each of them to ω , except those already having two successors. Add the corresponding f -labeled edges to J .
- The resulting CFG (1) contains a unique entry and exit (steps 3 and 4), (2) is intra-procedural with a single function f and **call**- and **ret**-free (steps 2, 3, and 5), (3) ensures each basic block has at most two successors (step 2), and (4) guarantees the exit is reachable from the entry (steps 3 and 5).

Reducible loops. Following the above procedure, most of the generated CFGs contain irreducible loops, since the generation process does not restrict the dominance relationship between basic blocks. To introduce more structured and diverse control flows, our approach also creates reducible loops. This is achieved by randomly replacing a basic block $b_\ell \in B$, $b_\ell \notin \{\alpha, \omega\} \wedge \text{outdeg}(b_\ell) \leq 1$ with a reducible loop ℓ . To construct ℓ , our approach generates an additional random CFG $g_\ell = \langle \alpha_\ell, \omega_\ell, B_\ell, J_\ell \rangle$ using the previous process. It then adds a jump from the exit node back to the entry ($\omega_\ell \xrightarrow{f} \alpha_\ell$), ensuring that the resulting CFG is a loop. In this loop, α_ℓ is the loop header dominating all other basic blocks in the body, and ω_ℓ is the loop latch. Finally, g_ℓ is integrated into the original CFG by redirecting all predecessors of b_ℓ to α_ℓ and connecting ω_ℓ to all successors of b_ℓ .

The above process is exemplified by Figure 5. Starting with an original CFG (Figure 5a), a reducible loop (Figure 5b) is generated and inserted, leading to a CFG with a reducible loop (Figure 5c). The basic block b_1 in Figure 5a is replaced by the highlighted nodes in Figure 5c.

S2: Path sampling. To sample a random EP π that starts at the entry node and ends at the exit node of a generated CFG g , our approach performs a random walk on g starting from the entry node α . For each visited basic block, our approach then uniformly selects a successor node $b \in B$ to visit next by considering *only* those nodes that directly or transitively reach the exit node ω . To allow arbitrary paths, including those that traverse loops, this step imposes no restriction on revisiting nodes during the random walk, meaning nodes may be revisited multiple times. The sampling process ends when ω is reached or the length of path ($|\pi|$) exceeds a predefined threshold. If the threshold is reached and the resulting path $\pi = [\alpha, \dots, b]$ is still incomplete, that is, $b \neq \omega$, π is patched by appending the shortest path (no cycles) from b to ω . This patch ensures that the selected path is an EP (per Definition 3.2), i.e., it ends at the exit node ω .

Consider again the CFG of Figure 1a. The path sampling starts at the entry node, which leads to B_1 . B_1 has two successors, B_2 and B_3 , both reaching exit. Suppose B_3 is chosen next. From B_3 , the only successor is B_1 , which is revisited. Then B_1 selects B_2 , leading to exit.

S3: Symbolic function creation. Like syntactic reification, this step is directed by the grammar of SYMLANG. We randomly select and expand the production rules of SYMLANG (Figure 4a). For the entry block, we declare a set of local variables initialized by symbols. For the exit block, a **return** statement is emitted to compute the checksum on *all* defined local variables. For all intermediate blocks, the procedure randomly generates a set of assignments and symbolic expressions using the

Table 1. Rules for constraint generation. INT_MIN and INT_MAX indicate the smallest and largest signed integer value, respectively. sext1(\cdot) extends the given value by one bit, e.g., sext1(0b01) = 0b001. round0(\cdot) casts a float value into an integer by rounding it toward zero. next(π) gives the next basic block to be visited along path π . $\varphi[\![\cdot]\!]$ converts a symbolic entity to its SMT formula.

Rule	Grammar	Computation	Definedness	Termination
DIV-TERM	v_i / s_j	$\text{round0}(\varphi[\![v_i]\!] \div \varphi[\![s_j]\!])$	$\varphi[\![s_j]\!] \neq 0$ $\varphi[\![v_i]\!] \neq \text{INT_MIN} \vee \varphi[\![s_j]\!] \neq -1$ $\varphi[\![v_i]\!] \div \varphi[\![s_j]\!] \times \varphi[\![s_j]\!] = \varphi[\![v_i]\!]$	-
ADD-EXPR	$\varepsilon_i + \tau_j$	$\varphi[\![\varepsilon_i]\!] + \varphi[\![\tau_j]\!]$	$\text{sext1}(\varphi[\![\varepsilon_i]\!]) + \text{sext1}(\varphi[\![\tau_j]\!])$ $\in [\text{INT_MIN}, \text{INT_MAX}]$	-
ASSIGN	$v_i := \varepsilon_j$	$\varphi[\![v_i]\!] = \varphi[\![\varepsilon_j]\!]$	-	-
CON-JMP	if(ε) goto L_1 else goto L_2	-	-	$\varphi[\![\varepsilon]\!] \neq 0$ if $L_1 = \text{next}(\pi)$ $\varphi[\![\varepsilon]\!] = 0$ if $L_2 = \text{next}(\pi)$

variables *already declared* in the entry block. This guarantees that every variable usage along the sampled EP is guaranteed to be defined earlier on that path, preventing uninitialized variables by construction. Finally, jump statements are inserted according to the structure of the CFG: (1) blocks with no successors contain no jumps, (2) blocks with one successor contain an unconditional jump, and (3) blocks with two successors contain a conditional jump guarded by a symbolic expression. Figure 1b presents an example symbolic function produced by this process (the corresponding CFG is shown in Figure 1a).

Note that this step naturally enforces data dependencies. Each generated statement in intermediate blocks either consumes previously defined variables or redefines variables that may be used later. Additionally, by returning a checksum over *all* defined variables, every statement along the selected path effectively contributes to the final checksum.

S4: Symbolic function concretization. Having created a symbolic function f from a given CFG, the next step is to concretize it by replacing all symbol occurrences with concrete values. To do so, we encode constraints that capture the semantic properties of each statement in f . In particular, we generate the following types of constraints.

- **Definedness.** All operations in f are well defined, free of undefined behavior.
- **Computation.** Each statement respects the operational semantics of SYMLANG, meaning that all computations behave consistently with the language's execution rules.
- **Termination.** When executed, f 's control flows along the sampled EP π and reaches the exit node without divergence.

To generate the aforementioned constraints, we perform symbolic execution, encoding the relevant semantic properties as SMT constraints denoted as φ . Instead of executing f with its branches and loops, we execute a *sequential SSA* (static single-assignment) variant of f . This variant is obtained by flattening the basic blocks in f according to the EP π selected in previous steps. For basic blocks executed multiple times, statements are inlined, and new SSA versions are introduced for each assigned variable. Table 1 presents the rules for constraint generation for a subset of production rules in SYMLANG. The remaining rules are straightforward and are provided in the supplementary material (Section B). The rules use $\varphi[\![\cdot]\!]$ to denote the translation function from a symbolic entity into an SMT logical formula.

Consider again Figure 1b. Visiting the entry block generates constraints $a_{\cdot 0} = s_1$, $c_{\cdot 0} = s_2$, and $b_{\cdot 0} = s_0$, where $x_{\cdot 0}$ are the initial SSA versions of a , b , and c , while s_i are their initial values. To ensure UB freedom, range constraints such as $\text{INT_MIN} \leq s_i \leq \text{INT_MAX}$ are also generated. For this example, the full SSA and constraints are given in our supplementary material's Section A.

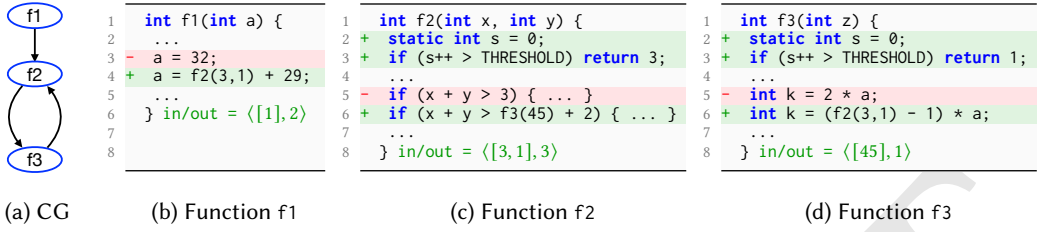


Fig. 6. An example illustrating how individual functions f_1 , f_2 , and f_3 generated by symbolic function reification are combined according to the caller-callee relationships defined by a randomly generated call graph (Figure 6a). The resulting program starts with f_1 and then calls f_2 and f_3 . Highlighted lines are the peephole rewrites that introduce these relationships while preserving termination.

Concretization. After generating the SMT constraints φ , we use an SMT solver to obtain an assignment model $M \models \varphi$. With M , the symbolic function f is converted into a concrete function f^* according to the concretization function $f^* = \llbracket f \rrbracket_{\varphi}^M$. Thanks to the SSA transformation, all internal program states are embedded in M , including the input i and output o where $o = f^*(i)$. If the solver fails (unsat or unknown), the approach resamples a new EP (Step 2) and retries. The process ends after a predefined number of unsuccessful attempts. The resulting f^* , i , and o already suffice to validate compiler correctness, even in the absence of whole-program generation, as many compiler optimization passes work at the function level [8, 26].

4.3 Whole-Program Generation

To create more complex programs that involve inter-procedural calls, we first apply symbolic function reification (Section 4.2) to generate a number of independent functions. To link these functions together, we proceed with two steps: (1) call graph generation and (2) peephole rewrites.

Call graph generation. This step establishes random caller-callee relationships among the generated functions. The goal is to construct a call graph (CG) of arbitrary shape, potentially including cycles indicating recursive calls. The process first creates a graph with N nodes corresponding to N random functions previously produced. It then adds edges between nodes at random to define function calls, and finally designates one function as the program's entry point.

Peephole rewrites. To respect the generated CG, our approach applies *peephole rewriting* [18, 21]. Consider two functions f_1^* and f_2^* where f_1^* calls f_2^* . Symbolic function reification associates f_k^* with a deterministic input-output pair $\langle i_k, o_k \rangle$, meaning that $f_k^*(i_k) = o_k$. The key idea of our peephole rewriting is to replace a constant c (i.e., a concretized symbol) in the caller f_1^* with a call to the callee that preserves the same value as c : $f_2^*(i_2) + (c - o_2)$, where $c - o_2$ is guaranteed to be well-defined. This rewrite links two functions semantically with inter-procedural relations.

Recursion and termination. The generated CG may include mutually recursive calls. To ensure termination, we introduce a short-circuit termination mechanism. Our approach rewrites the entry block of each callee function f_k^* by directly returning its known output o_k . To preserve effective execution before short-circuiting, we additionally apply a call-count-based strategy: the first several (user controlled) invocations execute the selected path normally, and only subsequent calls short-circuit to the expected return value. This guarantees that each function is effectively exercised while still ensuring termination with the correct output.

Figure 6 illustrates whole-program generation with three functions. Random caller-callee relationships are established in a cyclic CG (Figure 6a). Function calls are introduced via peephole rewrites (lines 3–4 of Figure 6b and lines 5–6 of Figures 6c and 6d). Callee's preambles are modified

to short-circuit to their known output once exceeding call-count thresholds (lines 2–3 in Figures 6c and 6d), ensuring effective execution and termination when recursion is present (i.e., $f2 \leftrightarrow f3$).

4.4 Undefined, Implementation-Defined, and Unspecified Behavior

Beyond undefined behavior, C distinguishes two more categories of problematic behaviors : (1) *implementation-defined behavior*, where the result depends on the compiler implementation but must be documented, and (2) *unspecified behavior*, where the C standard permits multiple outcomes without requiring documentation. Our approach addresses these three types of “bad” behaviors through a combination of three strategies: (1) restricting SYMLANG to a subset of C in which implementation-defined and unspecified behaviors are either eliminated or made irrelevant, (2) modeling behavior avoidance as SMT constraints, and (3) excluding specific programming patterns (e.g., calling functions with arbitrary expressions) from the generation logic.

Undefined behavior. As already discussed (Section 4.2), our approach eliminates UB by modeling its avoidance as an SMT problem. This is enforced during symbolic execution, where each language construct is associated with definedness constraints (e.g., no division by zero, no overflow). However, in some cases, preventing UB also relies on earlier or later steps. For example, ensuring that variables are defined before use depends on the *symbolic function creation* step (Section 4.2), while avoiding infinite recursion requires the short-circuit mechanism of *whole program generation* (Section 4.3). Below we summarize UBs that may arise and how they are handled by our approach.

UB	SYMLANG	S2: Path Samp.	S3: Sym. Fn. Cre.	S4: Sym. Fn. Con.	S5: Wh. Pro. Gen.
<i>Use of uninitialized variables</i>	Define variables only in the entry	–	Initialize variables symbolically	Add computation constraints	–
<i>Invalid scalar values</i>	No type casts	–	–	–	–
<i>Signed overflow</i>	No type casts	–	–	Add definedness constraints	Add definedness constraints
<i>Division by zero</i>	–	–	–	Add definedness constraints	–
<i>Out-of-bounds array access</i>	–	–	–	Add definedness constraints	–
<i>Unseq. modificat. and accesses</i>	Use side-effect-free expressions	–	–	–	–
<i>Infinite loops w/o side effect</i>	–	Use sampled, fixed execution paths	–	Add termination constraints	–
<i>Infinite recursion</i>	No function calls	Use sampled, fixed execution paths	–	Add termination constraints	Use a short-circuit mechanism

Implementation-defined behavior. We address implementation-defined behavior in two ways. First, SYMLANG currently excludes constructs that may expose implementation-defined behavior. For example, SYMLANG does not support bitwise operations on signed integers and type casts; for struct and pointer support, C-style bit-fields in structs, pointer arithmetic, and pointer casting should remain out of consideration. Second, SYMLANG employs fixed-width integer semantics, avoiding dependence on implementation- and platform-specific integer widths.

Unspecified behavior. C leaves certain behaviors unspecified, such as the evaluation order of subexpressions and function arguments. SYMLANG makes such behaviors irrelevant by restricting to a subset of C where evaluation order cannot affect program outcomes. Specifically, SYMLANG enforces side-effect-free expressions by excluding operators such as ++, --, and compound assignments like +=, and by treating assignments as statements rather than expressions. Also, SYMLANG itself contains no function-call expressions. Function calls are only introduced during *whole-program generation* (Section 4.3), where their arguments are constant literals produced by our approach

and *not* arbitrary expressions. Combined with the guarantee that all expressions are UB-free, these restrictions ensure that the evaluation of subexpressions and arguments is independent of order.

4.5 Implementation and Discussion

We implemented our approach in a tool called REIFY consisting of ~7K lines of C++ code, using the Z3 SMT solver for constraint solving. Our goal is to validate the feasibility and effectiveness of our approach with minimal yet representative language features. Therefore, REIFY currently targets a restricted core when generating C programs: 32-bit integers and 32-bit integer arrays. However, even with this limited feature support, the results are promising (Section 5).

Assumptions and limitations. REIFY assumes that the target compilers faithfully implement the C standard on the restricted subset. For example, they provide support for 32-bit integers with two's complement representation, and conform to C99 for division and remainder. To our knowledge, all mainstream C implementations (e.g., GCC and LLVM) satisfy these assumptions.

SMT solving is the main bottleneck that affects REIFY. To address this, we purposely decouple function generation from program generation: the former produces small, independent functions within acceptable time limits, while the latter efficiently constructs large programs by combining these functions. REIFY's speed can be further improved through parallelization. Section 5.2 provides empirical evidence on the scalability and performance of REIFY.

Semantic reification's formulation does not address type-related semantic problems, such as type validity or the well-formedness of declarations involving advanced language features, particularly those of object-oriented languages, such as C++ or Java (e.g., inheritance, overloading, type specialization, or type erasure) [2, 36, 40]. We leave it as an open challenge whether semantic reification can inspire corresponding approaches that guarantee type validity. Similarly, it does not address concurrency-related mechanisms such as synchronization. As with existing work [23, 24, 44], our focus is on generating single-threaded programs. REIFY also avoids generating long-running programs, i.e., programs that run for a long time but eventually terminate. However, this limitation mainly matters when testing JIT compilers, which typically optimize only after code becomes hot. This is less relevant for static compilers such as GCC and LLVM.

Extensibility. Supporting additional primitive or aggregate types in REIFY is possible. Implementing a new feature needs to (1) update SYMLANG's grammar (Figure 4a) and the type system if necessary (e.g., necessary for structs and pointers), (2) update *symbolic function concretization* step of our approach (Figure 2) to encode the definedness, computation, and termination constraints for the new feature, and (3) then update the *symbolic function creation* step (Figure 2) to generate instructions involving the new feature. To support heap-allocated objects and pointers, REIFY could adopt memory models similar to those used in traditional symbolic execution [14]. We expect the associated overhead to be significantly smaller, since REIFY solves constraints only along a single path, thereby reducing the search space. Adding support for heap objects and pointer manipulation would further expand the space of generated programs, potentially uncovering bugs in memory-related optimizations. We leave this for future work.

Avoiding pointer arithmetic and restricting expressions to be side-effect free are our deliberate and foundational design principles for SYMLANG for dealing with problematic behaviors (Section 4.4). While extensions such as the ++ operator are possible in principle, supporting such features would require the *symbolic function creation* step to rule out read-write and write-write conflicts between sequence points, for instance by linearizing conflicting subexpressions into separate statements. Similarly for pointer arithmetic, although feasible, it would require the *symbolic function concretization* step to adopt an object-based memory model, represent each pointer as a base object and an

offset, and generate definedness constraints ensuring that every pointer update remains within the bounds of the same object; otherwise, it may lead to undefined behavior.

Generality. Semantic reification is language-agnostic. It is also possible to extend our implementation REIFY to target other languages. Supporting a new language requires (1) implementing a backend that lowers SYMLANG programs to the target language, and (2) adapting the constraint-generation rules (Table 1) to match the language’s semantics and avoid language-specific problematic behavior. As demonstration, we ported REIFY to generate Java bytecode and WebAssembly. In a one-week testing campaign on the OpenJ9 JVM, we uncovered three wrong-code JIT compiler bugs (one new and two previously known), all promptly confirmed and fixed by developers.

Granularity. One may ask what is lost by applying semantic function reification to generate individual functions (Section 4.2), and only then composing them into whole programs (Section 4.3), instead of operating directly on inter-procedural CFGs. The sacrifice comes from the fact that in the current approach, function arguments are instantiated with specific constants that correspond to input values for which the output is known. This ensures correctness and termination, but restricts argument values to solver-determined constants. In contrast, directly generating and solving over inter-procedural CFGs would allow function calls to receive arbitrary symbolic values (arguments), but at the cost of prohibitively large constraints (Figure 3). Although this may lose opportunities for finding bugs within inter-procedural dataflow optimization passes, it increases chances for inlining, argument elimination, function specialization, etc.

5 Evaluation

We evaluate REIFY based on the following research questions:

RQ1 Can REIFY discover new bugs in C compilers? (Section 5.1)

RQ2 What is the performance and scalability of REIFY? (Section 5.2)

RQ3 What is the code coverage achieved by REIFY? (Section 5.3)

Summary of findings. Over the past five months, we have used REIFY to validate GCC and Clang/LLVM on an x86-64 Linux system with AMD processors. We have selected the latest development versions of both compilers to avoid reporting previously known bugs. Although REIFY is architecture-independent, we have not yet conducted testing campaigns on other platforms. Here is a summary of our opportunistic bug-finding efforts (Section 5.1):

- *Strong bug-finding capability.* REIFY discovers 59 bugs, with 57 confirmed and 27 fixed. More than 40% (24/59) of the bugs remain undetected for more than three major versions.¹
- *High incidence of wrong-code bugs.* Around 63% (36/57) of the confirmed bugs are wrong-code bugs that violate our oracle, exhibiting diverse symptoms such as execution failures, execution hangs, and incorrect outputs.
- *High incidence of high-priority bugs.* Nearly half (17/39) of the confirmed GCC bugs are classified as top-priority (P1–P2), and almost all LLVM bugs (15/18) are fixed within 15 days.
- *Diverse semantic characteristics.* Even after reduction, more than 80% (50/59) of the bugs still require non-trivial control flow to manifest, such as unbounded loops and irreducible regions. This explains why these bugs are missed by prior approaches; they cannot produce programs with such complex control flow.

Comparison with baselines. We further evaluate REIFY regarding RQ2 and RQ3. We compare our results with existing random program generators (or RPGs): Csmith [44] and YARPGen [24]. We choose them because they are actively maintained, widely adopted by production-grade C compilers. Moreover, recent work (2023 onwards) on mutation-based and hybrid compiler testing [5, 21, 31, 40]

¹For a version $x.y.z$, we consider x as the major version.

Table 2. Statistics of the reported issues (left) and their affected compiler optimizations (the top-six; right). Note: *These optimizations are loop-relevant.

GCC LLVM Total				GCC LLVM					
Reported	41	18	59	GCC			LLVM		
				Optimization	End	#	Optimization	End	#
Confirmed	39	18	57	Straight Line Strength Red.	middle	9	SLP Vectorization*	middle	8
Fixed	10	17	27	Value Range Propagation	middle	8	Loop Invariant Motion*	middle	4
Duplicate	5	0	5	RTL Optimization	back	7	Loop Vectorization*	middle	3
Wrong code	32	4	36	Scalar Evolution*	middle	5	Function Specialization	middle	1
Crashing	5	11	16	Induction Variable Opt.*	middle	5	Constraint Elimination	middle	1
Performance	4	3	7	Loop Invariant Motion*	middle	2	IR Generation	front	1

continues relying on them as core RPGs, indicating that they remain representative of the state of the art. Our comparisons show that REIFY is a promising complement to Csmith and YARPGen: (1) REIFY achieves practical generation throughput comparable to them (Section 5.2), and (2) REIFY covers code regions that they fail to reach due to our support for arbitrary control flow (Section 5.3). At the same, REIFY is capable of detecting at least 50 bugs that are out of reach of the existing RPGs, due to the generation of nontrivial control flow, which is the main contribution of REIFY.

5.1 RQ1: Bug-Finding Results

Table 2 presents the bug-finding results of our approach. We report a total of 59 bugs in GCC and LLVM, of which 41 are found in GCC and 18 in LLVM. Of these, 57 have been confirmed, and 27 have been fixed to date. A bug is considered “Confirmed” if the compiler developers tag it as such. Despite our efforts to avoid reporting duplicates, five issue reports are classified as known bugs.

Types of bugs. We categorize the reported bugs into three types based on their symptoms. The majority (61%) manifest as *wrong-code bugs*, where the compiler emits incorrect code that ultimately causes the program to fail at runtime. Importantly, all of these wrong-code bugs are detected directly by our oracle, without relying on pseudo-oracles such as differential testing. GCC accounts for 78% of wrong-code bugs, exhibiting a wide range of runtime failures, including illegal instructions (SIGILL), floating-point exceptions (SIGFPE),² segmentation faults (SIGSEGV), execution hangs, and incorrect outputs relative to our oracle (as in the example in Figure 1).

In contrast, most LLVM bugs (>61%) are *crashes*, where the compiler unexpectedly terminates during compilation. We attribute this to LLVM’s extensive use of assertions, which likely prevents many wrong-code cases by failing early. Among LLVM’s wrong-code bugs, one out of four triggers a floating-point exception (SIGFPE), with the remainder producing outputs that violate the oracle.

Finally, REIFY uncovers seven *performance bugs* (four in GCC and three in LLVM), where the compiler fails to compile the input program within a reasonable time or hangs.

Importance of bugs. For GCC, 44% (17/39) of the confirmed bugs are marked as high priority (P1 or P2), including four P1 bugs (the most severe category) and 13 P2 bugs. Although LLVM’s issue tracker does not assign severity or priority labels, their fix timeline suggests their importance: 15 of the 18 LLVM bugs have been resolved within 15 days, and 8 of those have been addressed within just two days after our reports.

We measure how many compiler versions each bug affects. Overall, 24/57 confirmed bugs remain undetected for more than three major versions. On GCC, bugs affect approximately three major

²All reported SIGFPEs are division-by-zero exceptions. These arise when GCC’s/LLVM’s erroneous optimizations transform REIFY-generated UB-free code into code with UB. REIFY successfully detects these as compiler bugs.

releases on average with more than half (20/39) impact at least four major versions. The oldest bug we find dates back to GCC 6.1, released nine years ago. For LLVM, bugs affect an average of two major versions, with the longest-surviving bug persisting across nine major releases.

Affected compiler optimizations. We also examine the root causes of the discovered bugs by identifying the specific optimization passes in which they occur. To do so, for each bug, we inspect developer’s fixes and, when necessary, the corresponding discussion among compiler developers. When the exact pass cannot be determined, we assign a broader category (e.g., IR generation for LLVM or RTL optimization for GCC). Overall, the bugs discovered by REIFY span more than 15 distinct optimization passes in GCC and LLVM, including the front end, middle end, and back end. The six most frequently affected passes for each compiler are summarized in Table 2.

Most of the bugs lie in the middle end, which is the component where optimization passes operate on intermediate representations (GIMPLE for GCC and LLVM IR for LLVM). In GCC, the most error-prone pass is SLSR (straight-line strength reduction), followed by VRP (value-range propagation). In LLVM, the most frequently affected passes are SLP (superword-level parallelization) vectorization and LICM (loop-invariant code motion).

Interestingly, many bugs detected by REIFY lie in loop-related optimizations. Unlike prior work [18, 21, 23, 24, 44], where most bugs arise in peephole optimizations confined to single instructions or basic blocks (e.g., instruction combining), the bugs discovered by REIFY are predominantly in inter-block (e.g., VRP, scalar evolution, vectorization) or inter-procedural optimizations (e.g., function specialization). This difference stems from semantic reification’s focus on modeling arbitrary control and data flow rather than on producing a large variety of statements within individual basic blocks. These results indicate that semantic reification is a strong complement to existing program-generation techniques.

Comparative analysis. To better understand why certain bugs are exclusively discovered by REIFY, we manually analyze the minimized bug-triggering programs. Our analysis is based on developer-reduced code when available, or our own reduction using Creduce [32]. Our analysis reveals that 50 out of the 59 bugs discovered by REIFY involve non-trivial control flow that existing RPGs (namely, Csmith [44] and YARPGen [24]) fail to generate for preserving semantic correctness, such as avoidance of UB (Section 5.4 presents concrete examples of such bugs). In particular:

- *Unbounded loops*: 50 bugs contain unbounded loops where induction variables or loop bounds are modified within the loop body: 30 wrong-code bugs, 13 crashes, and 7 performance issues. The example in Figure 1 involves unbounded loops. These are out of reach of existing RPGs.
- *Irreducible control flow*: 28 bugs involve irreducible regions lacking a dominating basic block for the loop body: 12 wrong-code bugs, 9 crashes, and 7 performance bugs. These are out of reach of existing RPGs, requiring the use of `goto`. Figure 8a presents an example. The remaining bugs are triggered by reducible control flow that does not require `goto`.
- Of the remaining bugs, four involve bounded loops, and five are loop-free. Therefore, in theory, they could be detected by existing RPGs.

This comparative analysis demonstrates that REIFY substantially enhances and complements existing RPGs by uncovering bugs that they fail to detect.

5.2 RQ2: Performance and Scalability

REIFY includes configurable parameters that influence the size of the generated programs as well as the performance of the generation process. The configuration consists of six parameters, where “default” denotes the default value of each parameter used by REIFY:

- #BBL: Number of basic blocks b generated per leaf function f (default: 15).
- #VAR: Number of variables v defined in the entry block α per function f (default: 8).

Table 3. Performance results for each component of REIFY, compared to Csmith and YARPGen. Each of them are independently executed 1,000 times (with different random seeds) to obtain an average.

Tool	SUCC	TMO	Throughput	Overhead	#Token	#Block	#Jump
Csmith	100%	0%	0.63/s	1.583s	35,690	427	854
YARPGen	100%	0%	13.81/s	0.072s	13,204	245	490
REIFY [◦]	82%	5%	2.97/s	0.336s	1,658	45	96
REIFY [•]	100%	0%	275.48/s	0.003s	42,844	456	913

- #ASS: Number of assignment statements σ generated per basic block b (default: 2).
- #TERA: Number of expression terms τ used in an expression ε of an assignment σ (default: 2).
- #TERC: Number of expression terms τ used in each conditional expression ε (default: 3).
- #FUN: Number of functions f to generate per program (default: 10).

Setup. To better understand the impact of the aforementioned parameters on REIFY’s performance and scalability, we conduct two experiments. In both experiments, we employ the following metrics:

- *Success rate*: The fraction of functions or programs successfully generated. A generation attempt is considered successful only if REIFY is able to concretize every symbol in the program under the given constraints.
- *Timeout rate*: The fraction of cases where the generation exceeds the time limit. In our experiments, we set a 3-second timeout for generating a single function or program.
- *Throughput*: The number of functions or programs successfully generated per second.
- *Overhead*: The number of seconds used to generate a single function or program.
- *Code size*: The size of the generated functions or programs, measured in the number of tokens, basic blocks, and block jumps.

We run the experiments on an AMD machine with an EPYC 9654 96-core CPU. We enable only one process for each generation attempt.

Performance. We use REIFY’s default configuration to generate 1,000 individual functions via symbolic function reification (Section 4.2), and an additional 1,000 programs through whole-program generation (Section 4.3). For simplicity, we refer to the individual-function generation component as REIFY[◦] and the whole-program generation component as REIFY[•]. For each function generated by REIFY[◦] and each program generated by REIFY[•], we compute the performance metrics described above. For comparison, we also use existing RPGs, namely Csmith [44] and YARPGen [24], to generate 1,000 programs each and measure their generation performance.

Table 3 summarizes the performance results. We separate generation cost (i.e., “Throughput” and “Overhead”) from test structure (i.e., CFG complexity via “#Block” and “#Jump”). It shows that REIFY achieves practical generation throughput comparable to these established RPGs at the default configuration. In particular, REIFY[◦] intentionally generates smaller functions (about 5–10× fewer tokens/blocks/jumps), while REIFY[•] combines them into programs of comparable overall size to Csmith and YARPGen with more than 10× lower generation overhead (i.e., approximately 3ms). At the same time, REIFY’s functions are about 10× denser in terms of CFG structure, and program-level density is comparable. The main bottleneck of REIFY is symbolic function reification (i.e., REIFY[◦]), due to the cost of constraint solving. On average, REIFY[◦] requires about 350ms to generate a function of approximately 1,600 tokens. This overhead lies between that of YARPGen and Csmith. Scaling REIFY[◦] to larger functions would increase the number of semantic constraints (see “Scalability” below). This will pose a challenge for the underlying SMT solver, which accounts for about 85% of REIFY[◦]’s total overhead. However, we believe this function size is acceptable for compiler testing, as most bug-triggering programs are small [1, 38]. REIFY[◦] experiences a 5% timeout

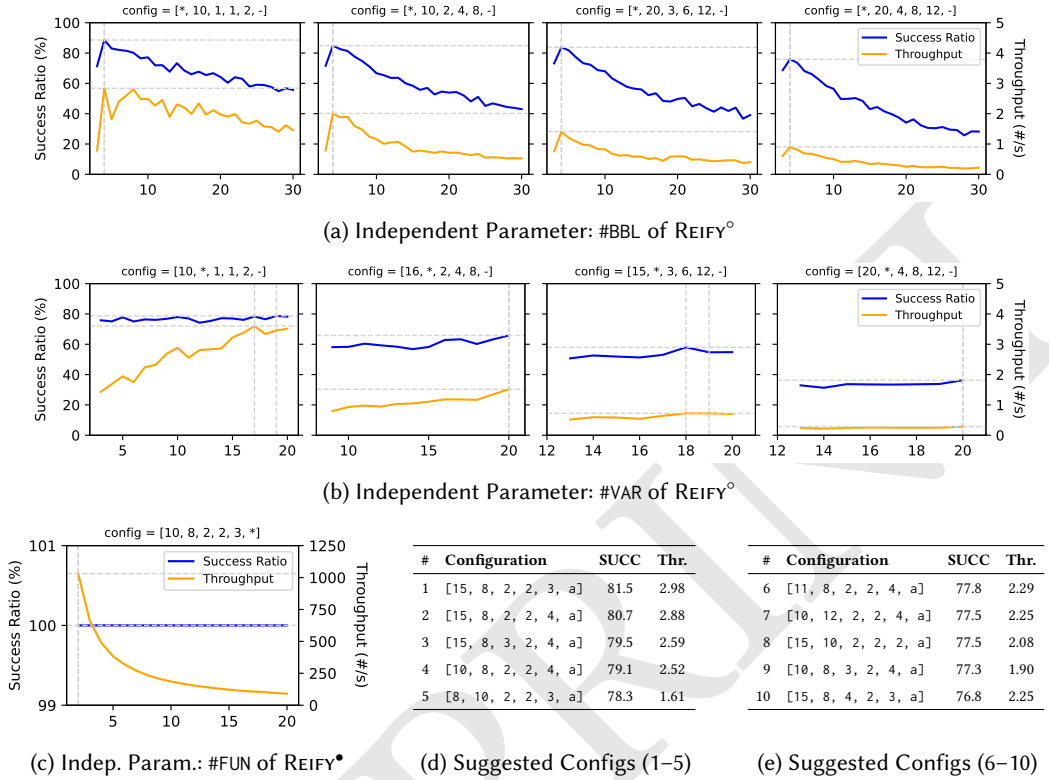


Fig. 7. Scalability of REIFY with respect to #BBL, #VAR, and #FUN. In each plot, the title indicates the configuration of the six parameters in the format [#BBL, #VAR, #ASS, #TERA, #TERC, #FUN], where “*” denotes the independent parameter, “-” indicates a non-applicable parameter, and numeric values represent controlled parameters. The last two tables present our suggested configurations for REIFY, where “a” means any value can be taken.

rate (i.e., the SMT solver exceeds the 3-second limit) and a 13% failure rate due to unsatisfiable constraints or cases where the solver returns unknown.

Scalability. To evaluate scalability, namely, how different parameters affect REIFY’s performance, we conduct a controlled experiment. Among the six parameters discussed earlier, we focus on three of them: #BBL, #VAR, and #FUN. These are chosen because they significantly influence the control-flow and data-flow complexity of the generated code. For each parameter, we vary it independently while keeping the remaining five parameters fixed. The fixed parameters are set according to some representative configurations. For example, when evaluating the impact of #BBL, we vary the number of basic blocks while keeping the other parameters (e.g., #VAR, #FUN) fixed across four predefined configurations. We repeat this process for #VAR and #FUN in a similar manner.

Figure 7 presents the results of our experiments. Limited by space, we present only a subset of the evaluated configurations. The full results, including those for the remaining three parameters (#ASS, #TERA, and #TERC) are provided in the supplementary material (Section C). For REIFY°, the effects of #BBL and #VAR exhibit distinct trends. As shown in Figure 7a, increasing #BBL negatively impacts both the success rate and throughput. This behavior is expected: a larger number of basic blocks increases the size and structural complexity of the CFG, as well as the complexity of the EP. In practice, using fewer than 10 basic blocks strikes a good balance. It achieves over 60% success rate

Table 4. Code coverage improvement on GCC and LLVM: Augmenting Csmith- or YARPGen-generated programs with REIFY^o-generated functions and then REIFY^{*}-generated programs.

	Setting	Csmith + REIFY			YARPGen + REIFY		
		Lines	Functions	Branches	Lines	Functions	Branches
GCC	100K (existing)	36.54%	39.55%	26.93%	32.99%	36.39%	23.89%
	+100K (REIFY ^o)	36.73%	39.65%	27.11%	34.78%	38.32%	25.31%
	rel. change (%)	+0.19%	+0.10%	+0.18%	+1.79%	+1.93%	+1.42%
	abs. change (#)	+1324	+88	+1434	+12772	+1625	+11254
	+100K (REIFY [*])	36.94%	39.74%	27.40%	35.19%	38.70%	25.80%
	rel. change (%)	+0.21%	+0.09%	+0.29%	+0.41%	+0.38%	+0.49%
LLVM	100K (existing)	15.89%	18.98%	13.60%	14.71%	17.94%	12.62%
	+100K (REIFY ^o)	15.97%	19.02%	13.69%	15.16%	18.37%	13.00%
	rel. change (%)	+0.08%	+0.04%	+0.09%	+0.45%	+0.43%	+0.38%
	abs. change (#)	+1112	+108	+1516	+6681	+1062	+7075
	+100K (REIFY [*])	16.05%	19.09%	13.76%	15.35%	18.57%	13.16%
	rel. change (%)	+0.08%	+0.07%	+0.07%	+0.19%	+0.20%	+0.16%
	abs. change (#)	+1145	+170	+1320	+2747	+507	+2940

and about 0.5 functions per second across the four configurations shown in Figure 7a. In contrast, increasing #VAR slightly improves performance (Figure 7b). This is because a larger variable pool gives REIFY^o more flexibility when constructing expressions: it results in fewer symbols and a sparser set of constraints, which is more friendly to the SMT solver.

Finally, increasing #FUN reduces throughput (Figure 7c), since each additional function increases both parsing cost and the number of semantic constraints. However, even at 20 functions per program, REIFY^{*} maintains high generation throughput of approximately 90 programs per second. Notably, for this experiment, we report one representative configuration, as parsing cost for small functions is negligible across scales.

Top configurations. Figure 7d and Figure 7e show the ten configurations that achieve the best performance. Rather than relying on a single configuration, our experience testing GCC and LLVM suggests generating a diverse set of individual functions under multiple configurations, and then composing them into whole programs with varied numbers and shapes of functions. Additional recommended configurations, suitable for different testing goals and resource budgets, are provided in the supplementary material (Section C).

5.3 RQ3: Code Coverage

We evaluate the ability of REIFY to reach code areas that Csmith and YARPGen fail to exercise. Specifically, for each existing RPG (Csmith or YARPGen) and each compiler (GCC or LLVM), we generate 100,000 programs. Next, we employ REIFY^o to generate 100,000 individual functions, which are then used to produce 100,000 programs with REIFY^{*}. To quantify REIFY's contribution, we measure its incremental code coverage, that is, the additional lines, branches, and functions in the compiler that are executed exclusively by REIFY programs, beyond those covered by Csmith or YARPGen. We run all RPGs using their default configurations.

Table 4 presents the code coverage results. Neither REIFY^o nor REIFY^{*} leads to a substantial increase in overall code coverage (less than 2%). The total code coverage for both compilers remains relatively low, below 40%. A potential reason for this is REIFY's limited support for the C grammar.

<pre> 1 int a, b, *c = &b; 2 void g(int i) { 3 int f[] = {a}; a = 0; 4 ... 5 h: ... 6 if (b) { 7 if (f[3]) goto k; 8 goto h; 9 } 10 ... 11 while (1) { 12 k: if (f[1]) break; 13 } 14 } </pre>	<pre> 1 int main() { 2 int c = -2147483647, 3 d = -2147483647, 4 e = 2147483647; 5 if (0) 6 f: e = d + e - 2; 7 g: ... 8 if (d - c - e) { 9 ... 10 if (d - c) goto g; 11 } 12 if (e) goto f; 13 ... 14 } </pre>	<pre> 1 @a = global i32 0 2 @c = global i32 0 3 define i32 @main() { 4 k: ... 5 s: ... 6 %2 = load i32, ptr @a; 7 %3 = load i32, ptr @c; 8 ... 9 %c8 = icmp sgt i32 %m7, -1 10 br i1 %c8, %t, %k 11 t: ... 12 br i1 %c4, %s, %e 13 o: ... e: ... 14 } </pre>	<pre> 1 define i32 @f(i32 %x) { 2 start: ... 3 %c = icmp eq i32 %x, 1 4 br i1 %c, %exit, %start 5 exit: ret i32 0 6 } 7 define void @g() { 8 start: ... 9 %res_0 = call i32 @f(i32 0) 10 %c = icmp eq i32 %res_0, 0 11 br i1 %c, %ret, %unreach 12 unreachable: 13 %uc = call i32 @f(i32 2) 14 } </pre>
(a) GCC-121756	(b) GCC-121370	(c) LLVM-138509	(d) LLVM-153295

Fig. 8. A set of bugs found by REIFY. The presented code snippets have been simplified for the sake of clarity.

However, this result is also consistent with the findings reported by existing RPGs [23, 24, 44]. We observe that REIFY’s improvement over YARPGen is greater than that over Csmith, and the improvement on GCC usually exceeds that on LLVM. This is expected, as Csmith supports a wider range of C language features, whereas YARPGen and REIFY are more general in scope.

Both REIFY^o and REIFY^{*} reach more than 80 new functions missed by existing tests for both compilers. Most functions covered *exclusively* by REIFY lie in the implementation of inter-procedural optimizations and control-flow graph transformations, rather than peephole optimizations. This indicates that REIFY is a promising complement to existing RPGs. More details about these functions exclusively covered by REIFY can be found in the supplementary material’s Section C.

5.4 Examples of Bug-Triggering Programs

We discuss a set of bugs discovered *exclusively* by REIFY. These bugs are beyond the capabilities of existing RPGs, as their manifestation requires unbounded loops, potentially irreducible.

Figure 8a: Code sinking violates memory SSA rules. GCC’s code-sinking optimization attempts to move a memory write ($a = 0$, line 3) into an irreducible block (k , line 12) that has multiple incoming control-flow paths. GCC uses memory SSA to track how memory values change across control-flow paths. During code sinking, the compiler incorrectly updates these memory dependencies when moving $a = 0$ into an irreducible block with multiple incoming edges. This leaves the memory SSA in an inconsistent state, violating the compiler’s assumptions and causing a crash.

Figure 8b: SCEV causes erroneous loop optimization. This long-latent bug, which is present since GCC 9.1, originates in GCC’s Scalar Evolution (SCEV) analysis, which reasons about how variables change within loops. In this example, SCEV incorrectly concludes that the conditional on line 8 could trigger a signed overflow UB. Because code having UB can be transformed arbitrarily [4], the compiler treats the entire loop (lines 7–10) as unreachable. During optimization, GCC then converts the unbounded loop into an infinite loop, completely unrolls it, and finally eliminates it as “dead code.” The generated assembly is inserted the `ud2` instruction, which is used by GCC to mark unreachable code paths. It causes a SIGILL (illegal instruction) at runtime.

Figure 8c: LICM cycles loop unswitching. LLVM’s Loop Invariant Code Motion (LICM) optimization incorrectly interacts with Loop Unswitching, leading to a non-terminating behavior during compilation. In particular, LICM moves the memory loads of $@a$ (line 6) and $@c$ (line 7) outside the unbounded loop ($s \rightarrow t \rightarrow s$) into its preheader, i.e., the section of code that executes before the loop begins. Afterward, loop unswitching duplicates the loop based on the conditional $\%c8$ (line 9), creating specialized loop versions for particular values of $\%c8$. Within these duplicated loops, new

expressions are introduced, which LICM again identifies as movable and hoists them outside. This repeated interaction between LICM and loop unswitching makes the compiler hang. The bug was initially mitigated by disabling loop unswitching under certain conditions, but that workaround later proved too conservative, as it interfered with other optimizations, such as vectorization. The issue was subsequently reopened and properly fixed.

Figure 8d: Function specialization invalidates SCCP. LLVM's Sparse Conditional Constant Propagation (SCCP) optimization first analyzes the function $@f()$ and determines that the basic block labeled `unreach` (line 13) is unreachable. This is because the call $@f(0)$ always returns the constant value 0 (line 9). Later, the function specialization optimization creates a new, optimized version of $@f()$ for the specific argument $@f(0)$, and removes the original $@f()$ definition. However, this invalidates the earlier SCCP's result because the definition of the callee function is updated. As a result of this change, LLVM now believes that `unreach` is now reachable. It then crashes upon encountering a call to $@f(2)$ during the analysis of the basic block `unreach` (line 13), since the original function $@f()$ is deleted but is still referenced.

6 Related Work

Generation-based techniques. These techniques generate programs from scratch, typically following syntactic reification. Orange [28], Csmith [44], and YARPGen [23, 24] are three representative ones closely related to ours. To ensure semantic correctness, they take conservative strategies that limit the expressiveness, as we discussed previously. REIFY^o belongs to this group of techniques, but it supports arbitrary control flow, including unbounded loops and irreducible regions. Beyond C compilers, there are RPGs focusing on other programming languages, such as Rust [34, 41], Java [2, 11], JavaScript [9, 13], or even deep learning compilers [22].

Mutation-based techniques. In addition to random seed mutation [5, 31, 33] or mutant enumeration [45], much of the work [19, 21, 37] in this group follows EMI (Equivalence Modulo Input) [18] to perform semantics-preserving transformations. REIFY^{*} similarly follows EMI principles to apply peephole rewrites, but it does not rely on static analysis or profiling and provides guarantees on semantic correctness. It is also promising to realize semantic reification through a mutation-based approach as we discussed before (Section 3).

Validation and verification. CompCert [20] is a verified optimizing compiler that operates on a subset of the C language. However, applying it to the whole, production-grade compilers such as GCC and LLVM remains infeasible. Translation validation (TV) [29] such as Alive and other tools [16, 17, 43] offers a practical alternative by verifying the code before and after translation, typically by comparing their runtime semantics. Similar to these efforts, REIFY models the program semantics formally to ensure correctness.

7 Conclusion

We introduced semantic reification which centers program generation on program semantics, allowing diverse and complex control- and data-flow behaviors. We implemented REIFY and applied it to two production C compilers, GCC and Clang/LLVM, leading to 59 discovered bugs, including many critical ones, demonstrating its effectiveness. We believe that the semantic reification paradigm and techniques are general and can be applied in other contexts, opening opportunities for validating static analyzers, debuggers, and verifiers.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback on an earlier version of this paper.

Data Availability

The artifacts supporting this paper, including source code and experimental results, are publicly available on Zenodo: <https://doi.org/10.5281/zenodo.18984610>. The repository also contains instructions and experimental scripts for reproducing the experiments.

References

- [1] Stefanos Chaliasos, Thodoris Sotiropoulos, Georgios-Petros Drosos, Charalambos Mitropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. 2021. Well-Typed Programs Can Go Wrong: A Study of Typing-Related Bugs in JVM Compilers. *Proc. ACM Program. Lang.* 5, OOPSLA (2021).
- [2] Stefanos Chaliasos, Thodoris Sotiropoulos, Diomidis Spinellis, Arthur Gervais, Benjamin Livshits, and Dimitris Mitropoulos. 2022. Finding Typing Compiler Bugs. In *Proceedings of the 2022 ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22)*.
- [3] Tsong Yueh Chen, Shing Chi Cheung, and Shiu Ming Yiu. 1998. Metamorphic Testing: A New Approach for Generating Next Test Cases. *Department of Computer Science, The Hong Kong University of Science and Technology, Tech. Rep. HKUST-CS98-01* (1998).
- [4] cppreference. 2025. *Undefined Behavior (C)*. <https://en.cppreference.com/w/c/language/behavior.html>
- [5] Karine Even-Mendoza, Arindam Sharma, Alastair F. Donaldson, and Cristian Cadar. 2023. GrayC: Greybox Fuzzing of Compilers and Analysers for C. In *Proceedings of the 2023 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23)*.
- [6] GCC. 2025. *GIMPLE*. <https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html>
- [7] GCC. 2025. *Leaf Functions*. <https://gcc.gnu.org/onlinedocs/gccint/Leaf-Functions.html>
- [8] GCC. 2025. *Passes and Files of the Compiler*. <https://gcc.gnu.org/onlinedocs/gccint/Passes.html>
- [9] Samuel Groß. 2018. *FuzzIL: Coverage Guided Fuzzing for JavaScript Engines*. Master's thesis. Karlsruhe Institute of Technology.
- [10] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets using Input-Output Examples. In *Proceedings of the 2011 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*.
- [11] Mohammad R. Haghghat, Dmitry Khukhro, Andrey Yakovlev, Nina Rinskaya, and Ivan Popov. 2018. *JavaFuzzer*. <https://github.com/AzulSystems/JavaFuzzer>
- [12] Daniel Conrad Halbert. 1984. *Programming by Example*. University of California, Berkeley.
- [13] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the 2012 USENIX Conference on Security Symposium (Security '12)*.
- [14] Timotej Kapus and Cristian Cadar. 2019. A Segmented Memory Model for Symbolic Execution. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19)*.
- [15] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19 (1976).
- [16] Jaeseong Kwon, Bongjun Jang, Juneyoung Lee, and Kihong Heo. 2025. Optimization-Directed Compiler Fuzzing for Continuous Translation Validation. *Proc. ACM Program. Lang.* 9, PLDI (2025).
- [17] Seungwan Kwon, Jaeseong Kwon, Wooseok Kang, Juneyoung Lee, and Kihong Heo. 2024. Translation Validation for JIT Compiler in the V8 JavaScript Engine. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*.
- [18] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence modulo Inputs. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*.
- [19] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '15)*.
- [20] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52 (2009).
- [21] Shaohua Li, Theodoros Theodoridis, and Zhendong Su. 2024. Boosting Compiler Testing by Injecting Real-World Code. *Proc. ACM Program. Lang.* 8, PLDI (2024).
- [22] Jiawei Liu, Yuxiang Wei, Sen Yang, Yinlin Deng, and Lingming Zhang. 2022. Coverage-Guided Tensor Compiler Fuzzing with Joint IR-Pass Mutation. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022).
- [23] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random Testing for C and C++ Compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA (2020).
- [24] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2023. Fuzzing Loop Optimizations in Compilers for C++ and Data-Parallel Languages. *Proc. ACM Program. Lang.* PLDI (2023).
- [25] LLVM. 2025. *LLVM Language Reference Manual*. <https://llvm.org/docs/LangRef.html>
- [26] LLVM. 2025. *LLVM's Analysis and Transform Passes*. <https://llvm.org/docs/Passes.html>

- [27] William M. McKeeman. 1998. Differential Testing for Software. *Digit. Tech. J.* 10, 1 (1998).
- [28] Eriko Nagai, Atsushi Hashimoto, and Nagisa Ishiura. 2014. Reinforcing Random Testing of Arithmetic Optimization of C Compilers by Scaling up Size and Number of Expressions. *IPSJ Trans. Syst. LSI Des. Methodol.* 7 (2014).
- [29] George C. Necula. 2000. Translation Validation for An Optimizing Compiler. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00)*.
- [30] Flemming Nielson, Hanne R Nielson, and Chris Hankin. 2004. *Principles of Program Analysis*. Springer Science & Business Media.
- [31] Xianfei Ou, Cong Li, Yanyan Jiang, and Chang Xu. 2024. The Mutators Reloaded: Fuzzing Compilers with Large Language Model Generated Mutation Operators. In *Proceedings of the 2024 ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '24)*.
- [32] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-Case Reduction for C Compiler Bugs. In *Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*.
- [33] Yuyang Rong, Zhanghan Yu, Zhenkai Weng, Stephen Neuendorffer, and Hao Chen. 2025. IRFuzzer: Specialized Fuzzing for LLVM Backend Code Generation. In *Proceedings of the 2025 IEEE/ACM International Conference on Software Engineering (ICSE '25)*.
- [34] Mayank Sharma, Pingshi Yu, and Alastair F. Donaldson. 2023. RustSmith: Random Differential Compiler Testing for Rust. In *Proceedings of the 2023 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23)*.
- [35] Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph. D. Dissertation. Advisor(s) Bodik, Rastislav.
- [36] Thodoris Sotiropoulos, Stefanos Chaliasos, and Zhendong Su. 2024. API-Driven Program Synthesis for Testing Static Typing Implementations. *Proc. ACM Program. Lang.* 8, POPL (2024).
- [37] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding Compiler Bugs via Live Code Mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '16)*.
- [38] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward Understanding Compiler Bugs in GCC and LLVM. In *Proceedings of the 2016 International Symposium on Software Testing and Analysis (ISSTA '16)*.
- [39] V8. 2025. *Land Ahoy: Leaving the Sea of Nodes*. <https://v8.dev/blog/leaving-the-sea-of-nodes>
- [40] Bo Wang, Chong Chen, Ming Deng, Junjie Chen, Xing Zhang, Youfang Lin, Dan Hao, and Jun Sun. 2025. Fuzzing C++ Compilers via Type-Driven Mutation. *Proc. ACM Program. Lang.* 9, OOPSLA2 (2025).
- [41] Qian Wang and Ralf Jung. 2024. Rustlantis: Randomized Differential Testing of the Rust Compiler. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024).
- [42] Elaine J. Weyuker. 1982. On Testing Non-Testable Programs. *Comput. J.* 25 (1982).
- [43] Zhiyuan Wu. 2025. *Translation Validation for the HotSpot C2 Just-in-Time Compiler*. Master's thesis. KTH Royal Institute of Technology.
- [44] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*.
- [45] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal Program Enumeration for Rigorous Compiler Testing. In *Proceedings of the 2017 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '17)*.

A The Illustrating Example: SSA and Constraints

The following table corresponds to Figure 1. The “Execution” column lists the execution path “entry \rightarrow B1 \rightarrow B3 \rightarrow B1 \rightarrow B2 \rightarrow exit”. The “SSA Statement” column shows the sequential SSA representation of Figure 1b. The remaining columns are the constraints collected to ensure the generated function is well-defined and deterministically terminates according to the given execution path. In this table, all computations involved in constraints are mathematical and do not overflow or underflow — the only UB considered in this example besides uninitialized scalars.

Execution	SSA Statement	Computation	Definedness	Control Flow
entry	int a_0 = s1, c_0 = s2;	$b_0 = s_0$ $a_0 = s_1$ $c_0 = s_2$	$\text{INT_MIN} \leq s_0 \leq \text{INT_MAX}$ $\text{INT_MIN} \leq s_1 \leq \text{INT_MAX}$ $\text{INT_MIN} \leq s_2 \leq \text{INT_MAX}$	–
B1	$c_1 = a_0 * s3 + b_0 * s4 + s5;$	$t_1 = a_0 \times s_3$ $t_2 = b_0 \times s_4$ $t_3 = t_1 + t_2$ $c_1 = t_3 + s_5$	$\text{INT_MIN} \leq s_3 \leq \text{INT_MAX}$ $\text{INT_MIN} \leq s_4 \leq \text{INT_MAX}$ $\text{INT_MIN} \leq s_5 \leq \text{INT_MAX}$ $\text{INT_MIN} \leq t_1 \leq \text{INT_MAX}$ $\text{INT_MIN} \leq t_2 \leq \text{INT_MAX}$ $\text{INT_MIN} \leq t_3 \leq \text{INT_MAX}$ $\text{INT_MIN} \leq c_1 \leq \text{INT_MAX}$	–
	if (c_1 * s6 + s7 <= 0) goto B3;	$t_4 = c_1 \times s_6$ $t_5 = t_4 + s_7$	$\text{INT_MIN} \leq s_6 \leq \text{INT_MAX}$ $\text{INT_MIN} \leq s_7 \leq \text{INT_MAX}$ $\text{INT_MIN} \leq t_4 \leq \text{INT_MAX}$ $\text{INT_MIN} \leq t_5 \leq \text{INT_MAX}$	$t_5 \leq 0$
B3	$a_1 = c_1 * s8 + s9;$	$t_6 = c_1 \times s_8$ $a_1 = t_6 + s_9$	$\text{INT_MIN} \leq s_8 \leq \text{INT_MAX}$ $\text{INT_MIN} \leq s_9 \leq \text{INT_MAX}$ $\text{INT_MIN} \leq t_6 \leq \text{INT_MAX}$ $\text{INT_MIN} \leq a_1 \leq \text{INT_MAX}$	–
	goto B1;	–	–	–
B1	$c_2 = a_1 * s3 + b_0 * s4 + s5;$	$t_7 = a_1 \times s_3$ $t_8 = t_7 + t_2$ $c_2 = t_8 + s_5$	$\text{INT_MIN} \leq t_7 \leq \text{INT_MAX}$ $\text{INT_MIN} \leq t_8 \leq \text{INT_MAX}$ $\text{INT_MIN} \leq c_2 \leq \text{INT_MAX}$	–
	if (c_2 * s6 + s7 <= 0) goto B3;	$t_9 = c_2 \times s_6$ $t_{10} = t_9 + s_7$	$\text{INT_MIN} \leq t_9 \leq \text{INT_MAX}$ $\text{INT_MIN} \leq t_{10} \leq \text{INT_MAX}$	$t_{10} > 0$
B2	goto exit;	–	–	–
exit	return a_1 + b_0 + c_2;	$t_{11} = a_1 + b_0$ $s_{10} = t_{11} + c_2$	$\text{INT_MIN} \leq t_{11} \leq \text{INT_MAX}$ $\text{INT_MIN} \leq s_{10} \leq \text{INT_MAX}$	–

B Full Semantic Encoding Rules

The following table extends Table 1 to support the entire grammar of SYMLANG. In this table: `INT_MIN` and `INT_MAX` indicate the smallest and largest signed integer value, respectively; `i32_const(·)` creates a 32-bit integer free variable with a given name; `sext1(·)` extends the given value by one bit, e.g., `sext1(0b01) = 0b001`; `round0(·)` casts a float value into an integer by rounding it toward zero; `next(π)` gives the next basic block to be visited along path π ; $\varphi[\![\cdot]\!]$ converts a symbolic entity to its SMT formula. Regarding the `RETURN` rule, the constraints are encoded based on the checksum function employed; the table uses a simple addition as an example, but in practice, `CRC32` is used. Furthermore, although we classified rules into three categories, some computation and control-flow rules also imply definedness. For example: the computation rules of `DECLARE` indicate that all scalars are initialized before use; the computation rules of `*-EXPR` imply a well-defined, left-to-right evaluation order; the grammar itself prevents invalid scalars; the terminating control flow ensures that there are no infinite loops without side effects, etc.

Rule	Grammar	Computation	Definedness	Termination
SYMBOL	s_i	$\varphi[\![s_i]\!] = \text{i32_const}(\text{name}:s_i)$	$\varphi[\![s_i]\!] \in [\text{INT_MIN}, \text{INT_MAX}]$	–
LOC-VAR	v_i	$\varphi[\![v_i]\!] = \text{i32_const}(\text{name}:v_i)$	$\varphi[\![v_i]\!] \in [\text{INT_MIN}, \text{INT_MAX}]$	–
DECLARE	$v_i := s_j$	$\varphi[\![v_i]\!] = \varphi[\![s_j]\!]$	–	–
ADD-TERM	$v_i + s_j$	$\varphi[\![v_i]\!] + \varphi[\![s_j]\!]$	$\text{sext1}(\varphi[\![v_i]\!]) + \text{sext1}(\varphi[\![s_j]\!])$ $\in [\text{INT_MIN}, \text{INT_MAX}]$	–
SUB-TERM	$v_i - s_j$	$\varphi[\![v_i]\!] - \varphi[\![s_j]\!]$	$\text{sext1}(\varphi[\![v_i]\!]) - \text{sext1}(\varphi[\![s_j]\!])$ $\in [\text{INT_MIN}, \text{INT_MAX}]$	–
MUL-TERM	$v_i * s_j$	$\varphi[\![v_i]\!] \times \varphi[\![s_j]\!]$	$\text{sext1}(\varphi[\![v_i]\!]) \times \text{sext1}(\varphi[\![s_j]\!])$ $\in [\text{INT_MIN}, \text{INT_MAX}]$	–
DIV-TERM	v_i / s_j	$\text{round0}(\varphi[\![v_i]\!] \div \varphi[\![s_j]\!])$	$\varphi[\![s_j]\!] \neq 0$ $\varphi[\![v_i]\!] \neq \text{INT_MIN} \vee \varphi[\![s_j]\!] \neq -1$ $\varphi[\![v_i]\!] \div \varphi[\![s_j]\!] \times \varphi[\![s_j]\!] = \varphi[\![v_i]\!]$	–
REM-TERM	$v_i \% s_j$	$\varphi[\![v_i]\!] - \varphi[\![v_i/s_j]\!] \times \varphi[\![s_j]\!]$	$\varphi[\![s_j]\!] \neq 0$ $\varphi[\![v_i]\!] \neq \text{INT_MIN} \vee \varphi[\![s_j]\!] \neq -1$ $\varphi[\![v_i]\!] \div \varphi[\![s_j]\!] \times \varphi[\![s_j]\!] = \varphi[\![v_i]\!]$	–
ADD-EXPR	$\varepsilon_i + \tau_j$	$\varphi[\![\varepsilon_i]\!] + \varphi[\![\tau_j]\!]$	$\text{sext1}(\varphi[\![\varepsilon_i]\!]) + \text{sext1}(\varphi[\![\tau_j]\!])$ $\in [\text{INT_MIN}, \text{INT_MAX}]$	–
SUB-EXPR	$\varepsilon_i - \tau_j$	$\varphi[\![\varepsilon_i]\!] - \varphi[\![\tau_j]\!]$	$\text{sext1}(\varphi[\![\varepsilon_i]\!]) - \text{sext1}(\varphi[\![\tau_j]\!])$ $\in [\text{INT_MIN}, \text{INT_MAX}]$	–
ASSIGN	$v_i := \varepsilon_j$	$\varphi[\![v_i]\!] = \varphi[\![\varepsilon_j]\!]$	–	–
ABS-JMP	<code>goto L</code>	–	–	–
CON-JMP	<code>if(ε) goto L_1</code> <code>else goto L_2</code>	–	–	$\varphi[\![\varepsilon]\!] \neq 0$ if $L_1 = \text{next}(\pi)$ $\varphi[\![\varepsilon]\!] = 0$ if $L_2 = \text{next}(\pi)$
RETURN	<code>return $\sum_{i=0}^n v_i$</code>	$\varphi[\![o]\!] = \sum_{i=0}^n \varphi[\![v_i]\!]$	$\forall j \in [1, n].$ $\sum_{i=0}^j \text{sext1}(\varphi[\![v_i]\!])$ $\in [\text{INT_MIN}, \text{INT_MAX}]$	–

C More Experimental Data

This section provides more data for our evaluation, such as the scalability (Figure 7) or the coverage experiment (Table 4). For the former, we include more independent parameters and more configurations for each independent parameter, as well as more suggested configurations. For the latter, we present the optimization-related GCC or LLVM files where our coverage is better.

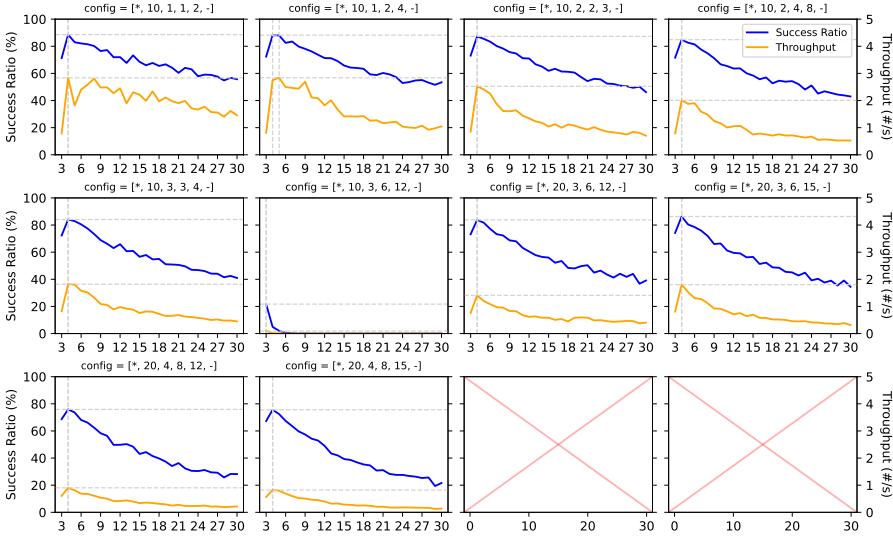


Fig. 9. Independent Parameter: #BBL of REIFY°. The configuration $[*, 10, 3, 6, 12, -]$ is invalid, as it violates the requirement that the number of variables used per condition (12) must not exceed the total number of variables (10), rendering the plot incorrect.

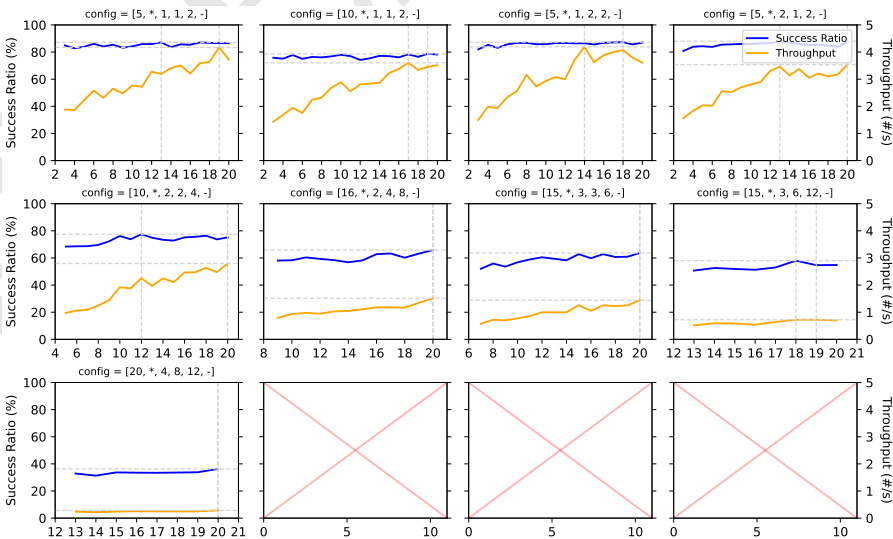


Fig. 10. Independent Parameter: #VAR of REIFY°

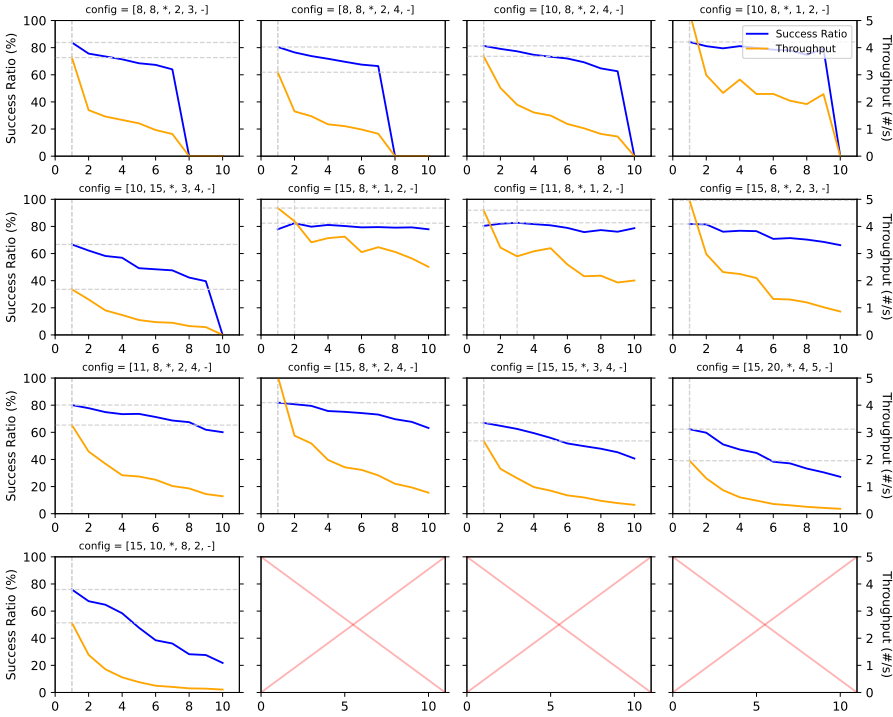


Fig. 11. Independent Parameter: #ASS of REIFY^o. This plot shows that #ASS is another major parameter influencing the scalability and performance of REIFY^o, alongside #BBL. This is expected, as an increase in assignments per basic block raises the number of constraints by at least the number of basic blocks. For any configuration, once the number of assignments exceeds a certain threshold, the success ratio drops sharply. This is because the additional assignments generate a dense set of constraints over a limited set of variables, increasing the likelihood of being unsatisfiable. According to it, choosing 2–5 assignments per basic block leads to a better balance among all evaluated configurations.

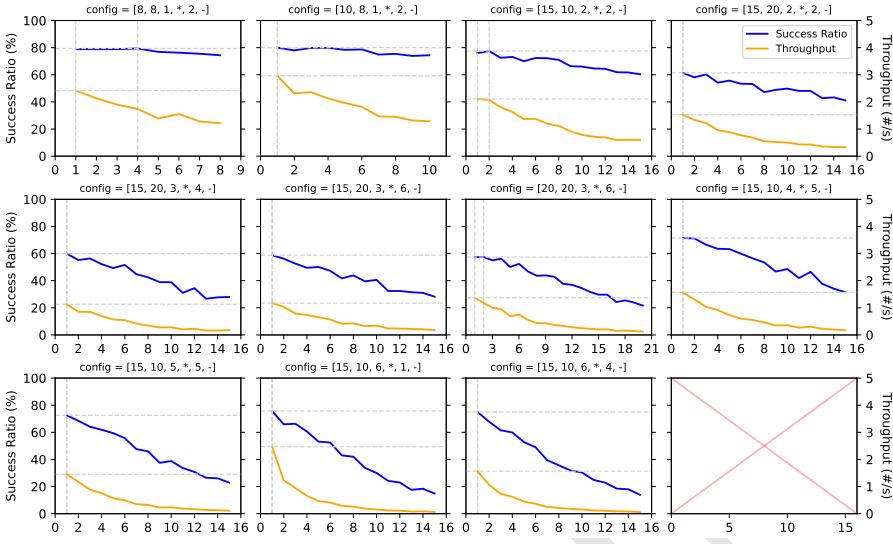


Fig. 12. Independent Parameter: #TERA of REIFY^o. Increasing the number of variables used in each assignment strengthens the correlations between variables and constrains the solution space, leading to negative influence.

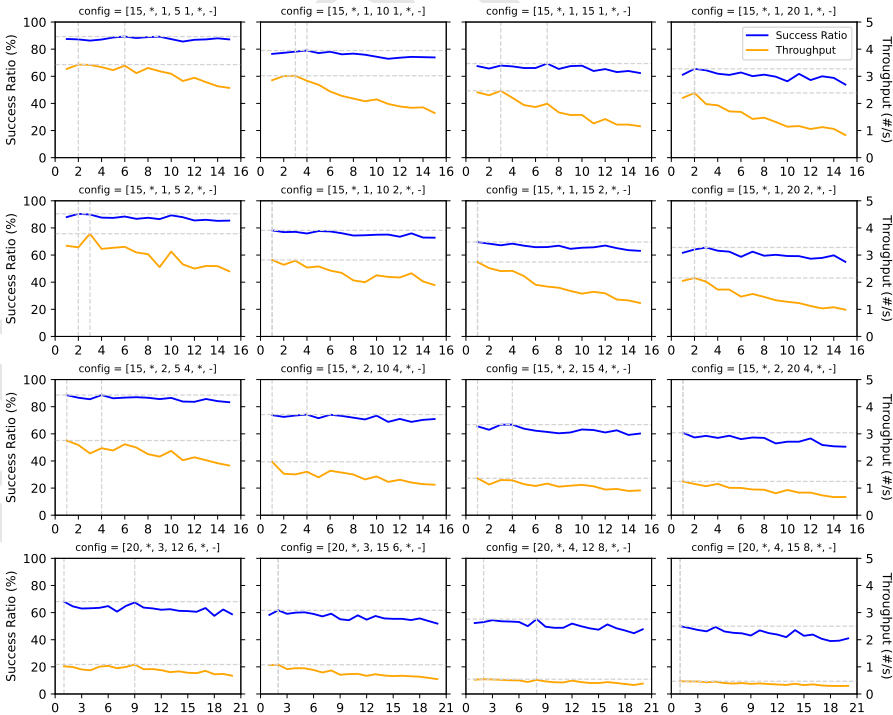


Fig. 13. Independent Parameter: #TERC of REIFY^o. Its influence stems from the same mechanism as #TERA, but #TERC has a smaller effect, since the former impacts all assignments while the latter only affects control flow.

Table 5. Configurations and their success rate and throughput. The data is sorted via success ratio first, then throughput. For a configuration, “*” denotes the independent parameter; “-” indicates a non-applicable parameter; numeric values represent controlled parameters; “a” means any value.

(a) Configs #1–#25				(b) Configs #26–#50				(c) Configs #51–#70			
#	Configuration	SUCC	Thr.	#	Configuration	SUCC	Thr.	#	Configuration	SUCC	Thr.
1	[15,8,2,2,3,a]	81.5	2.98	26	[10,10,2,2,3,a]	74.7	1.64	51	[10,8,6,2,4,a]	72.0	1.19
2	[15,8,2,2,4,a]	80.7	2.88	27	[10,8,4,2,4,a]	74.7	1.61	52	[8,20,3,6,15,a]	72.0	1.12
3	[15,8,3,2,4,a]	79.5	2.59	28	[15,8,6,2,4,a]	74.2	1.62	53	[15,10,2,4,8,a]	71.9	1.50
4	[10,8,2,2,4,a]	79.1	2.52	29	[15,10,2,4,4,a]	74.2	1.60	54	[8,8,4,2,4,a]	71.8	1.17
5	[8,10,2,2,3,a]	78.3	1.61	30	[15,10,2,4,6,a]	74.1	1.64	55	[15,10,2,4,5,a]	71.5	1.40
6	[11,8,2,2,4,a]	77.8	2.29	31	[10,11,2,2,4,a]	73.8	1.88	56	[8,8,4,2,3,a]	71.4	1.34
7	[10,12,2,2,4,a]	77.5	2.25	32	[8,8,3,2,4,a]	73.8	1.47	57	[15,8,7,2,3,a]	71.4	1.31
8	[15,10,2,2,2,a]	77.5	2.08	33	[10,19,2,2,4,a]	73.7	2.47	58	[11,8,6,2,4,a]	71.3	1.25
9	[10,8,3,2,4,a]	77.3	1.90	34	[11,8,5,2,4,a]	73.6	1.37	59	[11,10,2,2,3,a]	71.2	1.44
10	[15,8,4,2,3,a]	76.8	2.25	35	[15,10,2,4,3,a]	73.5	1.50	60	[9,10,2,4,8,a]	71.2	1.24
11	[15,8,5,2,3,a]	76.5	2.10	36	[8,8,3,2,3,a]	73.5	1.46	61	[15,10,4,2,5,a]	71.1	1.31
12	[8,8,2,2,4,a]	76.5	1.65	37	[10,14,2,2,4,a]	73.4	2.25	62	[12,10,2,2,3,a]	71.0	1.34
13	[10,18,2,2,4,a]	76.4	2.64	38	[15,10,2,4,10,a]	73.4	1.43	63	[15,10,2,4,12,a]	71.0	1.31
14	[10,10,2,2,4,a]	76.2	1.92	39	[11,8,4,2,4,a]	73.4	1.42	64	[15,10,2,8,2,a]	71.0	1.11
15	[15,8,3,2,3,a]	76.1	2.32	40	[8,10,3,3,4,a]	73.4	1.32	65	[15,10,2,4,15,a]	70.9	1.12
16	[15,8,4,2,4,a]	75.7	1.98	41	[10,8,5,2,4,a]	73.3	1.50	66	[15,8,6,2,3,a]	70.8	1.33
17	[9,10,2,2,3,a]	75.7	1.60	42	[15,10,2,4,2,a]	73.2	1.63	67	[15,10,2,4,9,a]	70.6	1.32
18	[10,17,2,2,4,a]	75.6	2.47	43	[15,10,2,4,7,a]	73.2	1.57	68	[15,8,8,2,3,a]	70.3	1.20
19	[8,8,2,2,3,a]	75.6	1.70	44	[15,8,7,2,4,a]	73.1	1.41	69	[15,10,2,4,14,a]	70.3	1.14
20	[10,20,2,2,4,a]	75.2	2.80	45	[10,15,2,2,4,a]	72.8	2.11	70	[15,10,2,5,2,a]	70.0	1.36
21	[10,16,2,2,4,a]	75.2	2.47	46	[15,10,2,3,2,a]	72.5	1.80				
22	[15,8,5,2,4,a]	75.1	1.71	47	[15,10,2,4,2,a]	72.5	1.53				
23	[10,13,2,2,4,a]	74.9	1.97	48	[15,10,2,6,2,a]	72.4	1.38				
24	[11,8,3,2,4,a]	74.9	1.84	49	[10,9,2,2,4,a]	72.3	1.44				
25	[8,10,2,4,8,a]	74.8	1.48	50	[15,10,2,7,2,a]	72.2	1.20				

Table 6. The top 10 optimization-related files where REIFY covers more lines of code (left) or exclusively covers more functions (right). In the left table, the column “%” indicates REIFY’s improvement in line coverage compared with either Csmith or YARPGen, and in the right table, the column “#” denotes the number of exclusively covered functions by REIFY compared with either Csmith or YARPGen.

(a) Better Coverage				(b) Exclusive Coverage			
GCC	%	LLVM	%	GCC	#	LLVM	#
ipa-cp.cc	37.6	DivRemPairs.cpp	39.5	ipa-cp.cc	85	SROA.cpp	44
tree-sra.cc	36.1	CFG.cpp	1.4	tree-sra.cc	57	FunctionSpecializ.cpp	22
ipa-param-manip.cc	37.2	MemoryDepAnalysis.cpp	4.2	ipa-prop.cc	39	DeadStoreElimination.cpp	15
ipa-sra.cc	20.7	DeadArgElimination.cpp	4.5	ipa-param-manip.cc	37	InlineCost.cpp	15
ipa-fnsummary.cc	11.8	AliasAnalysis.cpp	5.2	cfgexpand.cc	30	CFG.cpp	14
tree-inline.cc	9	MemCpyOptimizer.cpp	1.7	ipa-sra.cc	21	ArgumentPromotion.cpp	13
ipa-split.cc	29.3	EarlyCSE.cpp	1.8	tree-eh.cc	16	SCCPSolver.cpp	11
ipa-modref.cc	8.1	LoopInfo.cpp	2.1	cgraphclones.cc	14	MemCpyOptimizer.cpp	11
ipa-cp.cc	6.3	BranchProbInfo.cpp	1.8	df-problems.cc	11	GlobalOpt.cpp	11
ipa-inline.cc	11.6	SROA.cpp	22.5	cgraph.cc	11	BranchProbInfo.cpp	10

Received 2025-11-14; accepted 2026-04-03