Optimization-Directed Compiler Fuzzing for Continuous Translation Validation

JAESEONG KWON, KAIST, Korea
BONGJUN JANG, KAIST, Korea
JUNEYOUNG LEE, AWS, USA
KIHONG HEO, KAIST, Korea

Incorrect compiler optimizations can lead to unintended program behavior and security vulnerabilities. However, the enormous size and complexity of modern compilers make it challenging to ensure the correctness of optimizations. The problem becomes more severe as compiler engineers continuously add new optimizations to improve performance and support new language features. In this paper, we propose Optimuzz, a framework to effectively detect incorrect optimization bugs in such continuously changing compilers. The key idea is to combine two complementary techniques: directed grey-box fuzzing and translation validation. We design a novel optimization-directed fuzzing framework that efficiently generates input programs to trigger specific compiler optimizations. Optimuzz then uses existing translation validation tools to verify the correctness of the optimizations on the input programs. We instantiate our approach for two major compilers, LLVM and TurboFan. The results show that Optimuzz can effectively detect miscompilation bugs in these compilers, outperforming state-of-the-art tools. We also applied Optimuzz to the latest version of LLVM and discovered 55 new miscompilation bugs.

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

Additional Key Words and Phrases: Compiler Testing, Directed Fuzzing, Translation Validation, Compiler Optimization, Miscompilation

ACM Reference Format:
Jaeseong Kwon, Bongjun Jang, Juneyoung Lee, and Kihong Heo. 2025. Optimization-Directed Compiler Fuzzing for Continuous Translation Validation. *Proc. ACM Program. Lang.* 9, PLDI, Article 172 (June 2025), 24 pages. https://doi.org/10.1145/3729275

1 Introduction

Real-world compilers are large, complex, and constantly evolving. Developers frequently add new features, fix bugs, and improve the performance of compilers. This makes reasoning about the correctness of compilers a challenging task. For example, the LLVM compiler infrastructure [24] comprises over 1.5M lines of code and has over 30K commits by 1.8K contributors from Jan 2024 to Sep 2024. In particular, reasoning about compiler optimization passes is more challenging because they are frequently updated and represent one of the most complex and error-prone parts of the compiler [68]. Any bug in compiler optimization passes that silently changes the behavior of the compiled program may result in critical problems such as security vulnerabilities [5–10].

Recent studies have demonstrated that translation validation (TV) [23, 57, 61] is a promising approach to ensure the correctness of real-world compilers. Given a program, TV tools such as

Authors' Contact Information: Jaeseong Kwon, KAIST, Daejeon, Korea, jaeseong.kwon@kaist.ac.kr; Bongjun Jang, KAIST, Daejeon, Korea, bongjun.jang@kaist.ac.kr; Juneyoung Lee, AWS, Austin, USA, lebjuney@amazon.com; Kihong Heo, KAIST, Daejeon, Korea, kihong.heo@kaist.ac.kr.

ACM 2475-1421/2025/6-ART172
https://doi.org/10.1145/3729275

Alive2 [57] check whether a compiler optimization preserves the semantics of the input program. They show semantics preservation by checking the refinement relation between the two programs before and after each individual optimization [61]. Combined with an automated theorem prover such as SMT solver, compiler engineers and users can use TV without having domain knowledge in formal verification. TV tools may not validate the compilation of large input programs due to the limitations of the automated theorem prover in practice. However, working on small input programs is still helpful for *increased* assurance of compiler correctness. For example, LLVM uses Alive2 as a regression testing tool for its unit test suite when optimizations are added or modified [57].

However, the effectiveness of TV depends on a provided set of input programs (e.g., LLVM's test suite). This is because they only prove the correctness of the compiler on the provided input programs. This is a fundamental problem and cannot be theoretically resolved because there are an infinite number of possible programs. To practically address this limitation, one can combine TV tools with a fuzzer to generate new input programs that trigger compiler optimizations [13, 23, 28]. However, this simple combination may not be effective when optimization conditions are complex as compiler optimizations are only triggered when input programs satisfy specific constraints. This in turn hinders the effectiveness of TV tools in *proactively* detecting miscompilation bugs.

In this paper, we present Optimuzz, a framework to continuously validate the correctness of compilers. Given an update of the compiler source code, Optimuzz systematically selects a target location (i.e., source line) to validate the updated optimizations. Then, Optimuzz leverages *directed fuzzing* [2, 4, 11, 20, 21] to generate input programs that are likely to reach the target. Directed fuzzing is an emerging technique that generates inputs to reach specific target locations in the program under test. This technique has been successfully applied to a variety of domains, such as patch testing, crash reproduction, and static analysis alarm inspection [2, 4, 21]. In our context, this fuzzer aims to generate input programs that trigger specific compiler optimizations at the target location. Optimuzz then uses existing TV tools to check the correctness of the optimizations on those programs. In addition to such *continuous* validation, our idea can also be generalized for *batch* mode validation which targets all the optimizations in the latest version of the compiler.

There are two key challenges in designing a fuzzer for compiler validation. First, we must have an *effective guidance strategy* for the fuzzer to trigger the target optimizations. Conventional grey-box fuzzers typically rely on coverage feedback to guide the generation of inputs. When a newly generated input covers an unvisited program point, fuzzers consider the input as promising and keep it for further mutation. However, since compilers are typically large and complex, feedback from points unrelated to the target optimization can negatively impact the effectiveness of the fuzzer. To address this challenge, we design a *selective-coverage-guided search strategy* that focuses exclusively on the relevant parts of the compiler code that are likely to lead to the target optimization.

Second, the fuzzer must have an *effective mutation strategy* to generate input programs to quickly trigger the target optimizations. Unlike conventional fuzzing which handles input as a sequence of bytes or formatted documents (e.g., XML) [14], we must generate programs that satisfy certain semantic constraints (i.e., optimization conditions). This challenge makes naive fuzzing with random mutation highly ineffective. We address this challenge by designing a *data-flow-based targeted mutation strategy*. Our mutation strategy is inspired by the common structures of code transformation rules in compiler optimizations. Such rules typically check the existence of certain code patterns where each of the matched code snippets is closely related to one another along the data-flow. Based on this observation, Optimuzz prioritizes mutating code snippets related to the key statements involved in the target optimizations.

We instantiate Optimuzz with two real-world compilers, LLVM and TurboFan. LLVM is a widely used compiler infrastructure for many programming languages such as C, C++, and Rust. TurboFan is the optimizing JIT (just-in-time) compiler in the V8 JavaScript engine [16]. Our

experiments show that OptiMuzz effectively detects miscompilation bugs in these compilers. In particular, OptiMuzz was able to reproduce 23 bugs in LLVM and 4 bugs in TurboFan significantly faster than state-of-the-art fuzzing techniques. Moreover, OptiMuzz was able to detect 55 new miscompilation bugs in the latest version of LLVM.

We summarize our contributions below:

- We present a novel approach, OptiMuzz, to continuously validate the correctness of compiler optimizations.
- We design an optimization-directed fuzzing technique that effectively generates input programs to trigger specific compiler optimizations.
- We demonstrate the effectiveness of OptiMuzz by applying it to two real-world compilers, LLVM and TurboFan. OptiMuzz reproduced known miscompilation bugs in LLVM and TurboFan faster than state-of-the-art tools and discovered 55 new miscompilation bugs in LLVM.
- Our tool and data are available at our website (https://prosys.kaist.ac.kr/optimuzz) and Zenodo (https://doi.org/10.5281/zenodo.15037303).

2 Problem Definition

We formalize compiler update and its relation to TV. We found no prior work explaining this topic.

2.1 Formal Definition of Compiler Update

We can define the semantics of a compiler as a function from a source program to a target program, $C : \mathcal{P}_{src} \to \mathcal{P}_{tgt}$, where $\mathcal{P}_{src}$ and $\mathcal{P}_{tgt}$ are sets of all syntactically valid source and target programs respectively. We define that two compilers C and C' are equal if for any source program $P_{src} \in \mathcal{P}_{src}$, $C(P_{src}) = C'(P_{src})$ where $=$ is the syntactic equality between the two target programs.

If compilers C and C' are not equal, there must exist a set of source programs that makes the two compilers have diverging behavior, i.e., $\{P \mid C(P) \neq C'(P)\}$. We will call this program set a difference between two compilers and use the notation $D(C, C')$. A typical subset of $D(C, C')$ in the real world is a set of unit test programs that are attached to every compiler commit. For example, in the LLVM compiler community, it is strongly suggested that each commit must include at least one new unit test program that will compile into a different target program with the commit.

Ideally, we can prove a sequence of compiler updates $C_0, C_1, \cdots, C_k$ to be correct by using of D and TV. We can show that C_k is correct under the assumption that C_0 is also correct as follows:

$$\forall P \in D(C_i, C_{i+1}).\ \Omega(P, C_{i+1}(P)) = \text{Correct} \qquad \text{for } 0 \leq i < k$$

where Ω is a TV tool that checks the correctness of the compilation result. However, performing this in practice is hard because $D(C_i, C_{i+1})$ will contain an infinite number of programs. For example, if C_{i+1} is C_i with a new arithmetic optimization "$(x + y) - y \Longrightarrow x$" added, $D(C_i, C_{i+1})$ will be an infinite set of programs where each program contains "$(x + y) - y$" as a subexpression.

To address this problem, we define the *kernel* of the set D. The kernel $D_K(C_i, C_{i+1})$ is a subset of $D(C_i, C_{i+1})$ such that every $P \in D_K(C_i, C_{i+1})$ is *1-minimal* [67]. A program P is 1-minimal if it no longer activates the optimization if any element from P is removed. The concept of program reduction allows us to define the partial ordering between programs; if P can be reduced from Q, we denote it as $P \sqsubseteq Q$. Then, the kernel is the set of programs that are minimal in $\sqsubseteq$. For example, consider an optimization that rewrites the expression "$x + y - y$" into "x". Programs "$x + y - y$" and "$x + y - y - z$" triggers this optimization. However, since we can reduce "$x + y - y - z$" to "$x + y - y$", the kernel is the singleton set {"$x + y - y$"}. This property guarantees the uniqueness of the kernel.

Thus, if $C_{i+1}(P)$ is correct by TV for every $P \in D_K$, then $C_{i+1}(P)$ for $P \in D$ is also correct. For example, checking only one unit test program containing the expression "$(x + y) - y$" using TV will suffice to prove the correctness of the compiler update that adds the optimization "$(x+y)-y \Longrightarrow x$".

2.2 Approximating $D_K(C, C')$ via Directed Fuzzing

Even though the kernel $D_K(C, C')$ compactly represents the difference between two compilers, it may contain infinitely many programs, which makes running a TV tool on all of them impractical. For example, a compiler developer may generalize optimization "$(x+y)-y \implies x$" to "$(x+y)-y' \implies x$" if y and y' are known to be equal using data flow analysis. Note that the kernel of this update is an infinite set of programs where y and y' are syntactically different but semantically equal.

In addition, finding $D_K(C, C')$ is hard in practice. Constructing the set $D(C, C')$ will require the formally defined semantics of C and C', which are large software written in full-fledged programming languages like C++. Even if D could be successfully constructed, getting its kernel will require a precise understanding of the compiler update. This can be challenging when advanced compiler analysis is involved.

Therefore, rather than manually finding the exact kernel for every compiler update, we choose an automatic approach that finds an approximation of the kernel. For automation, we will utilize a *directed fuzzer* to generate random programs to compile. Directed fuzzing is a technique that generates input to reach a specific target location in a program under test. In our problem, the target location is the source code location that is affected by the compiler update and the set of generated input programs will be an approximation of the kernel set.

For a better approximation, we propose two strategies. First, we start the fuzzing with unit test programs attached to a compiler update. Compiler developers are typically encouraged to include 1-minimal unit tests which activate the target optimization. This means that ideally the unit tests are already included in the kernel. When the updated optimization is incorrect, the attached passing unit tests and bug-triggering programs are also in the kernel. Thus, we search for miscompilation bugs in the kernel by starting with the unit tests. Second, we look into the actual compiler updates and properly use their line numbers as the target locations for the directed fuzzer. Compiling any program in the kernel must arrive at the target source location, otherwise, the result of the compilation would not have been affected by the update.

2.3 D_K, Rewrite Rules and Target Location of Directed Fuzzer

Conceptually, a compiler is a big set of rewrite rules whose application order is carefully orchestrated. In this perspective, a compiler update can be categorized into three types: **(D1)** adding a new rewrite rule, **(D2)** removing a rewrite rule, or **(D3)** updating an existing rule. Even if there are other kinds of compiler updates such as extending its intermediate representation or restructuring the directory hierarchy, the three rewrite-rule updates will consist significant portion of the compiler updates.

If a compiler update adds a new rewrite rule **(D1)**, the kernel of the update D_K is a set of programs whose compilation will invoke the new rewrite rule. If an update removes an existing rewrite rule **(D2)**, its kernel is a set of programs that invoked the deleted rewrite rule in the past. If a compiler update is a modification of an already existing rewrite rule **(D3)**, the kernel D_K is a set of programs that do not fire the old rewrite rule but fire the new rewrite rule, or vice versa.

When the rewrite rules are implemented in the compiler source code, they often have a common code structure in practice. For example, peephole optimizations of LLVM are typically implemented as if-statements whose condition matches the input pattern (left-hand side of the rule) and the body of the conditional describes its output expression (right-hand side of the rule). The following example of an optimization in LLVM replaces a remainder operation with a bitwise-and operation:

$$(X \ \% \ 2^N) < 0 \ ? \ (X \ \% \ 2^N) + 2^N : (X \ \% \ 2^N) \implies X \ \& \ (2^N - 1).$$

This rule is implemented in LLVM as a function shown in Fig. 1. Note that this function consists of a series of condition checks and the body of the main if statement (Line 15) corresponds to the right-hand side of the rule.

```
1  static Instruction *foldSelectWithSRem(SelectInst &SI, InstCombinerImpl &IC, IRBuilderBase &Builder) {
2    Value *Cond, *Tr, *Fl = SI.getCondition(), SI.getTrueValue(), SI.getFalseValue();
3    ICmpInst::Predicate Pred;
4    Value *Op, *RemRes, *Remainder;
5    ...
6    if (!(match(Cond, m_ICmp(Pred, m_Value(RemRes), m_APInt(C))) &&
7          isSignBitCheck(Pred, *C, TrueIfSigned)))
8      return nullptr;
9
10 - if (match(Tr, m_Add(m_Value(RemRes), m_Value(Remainder))) &&
11 + if (match(Tr, m_Add(m_Specific(RemRes), m_Value(Remainder))) &&
12       match(RemRes, m_SRem(m_Value(Op), m_Specific(Remainder))) &&
13       IC.isKnownToBeAPowerOfTwo(Remainder, true) &&
14       Fl == RemRes)
15     return FoldToBitwiseAnd(Remainder); // code transformation
16   ...
17   return nullptr;
18 }
```

Fig. 1. Miscompilation bug in LLVM (Issue 89516 [52]) and its patch

```
define i8 @src(i8 %x, i8 %y) {
  %n = shl i8 1, %x
  %c = icmp slt i8 %y, 0
  %r = srem i8 1, %n
  %t = add i8 %r, %n
  %s = select i1 %c, i8 %t, i8 %r
  ret i8 %s
}
; before optimization: @src(0, 255) = 1
```

(a)

```
define i8 @tgt(i8 %x, i8 %y) {
  %a = shl i8 1, %x
  %b = add i8 %a, -1
  %c = and i8 1, %b
  ret i8 %c
}

; after optimization: @tgt(0, 255) = 0
```

(b)

Fig. 2. LLVM IR code before (a) and after (b) the incorrect optimization in Fig. 1.

We can rely on this practice to effectively decide which program locations a directed fuzzer must target. For a compiler update that adds a new optimization **(D1)** whose code structure follows the previously mentioned pattern, we can pick the body of the corresponding if-branch as a target of the fuzzer. This means that the fuzzer aims to generate programs that trigger the newly added optimization. For the removal of an optimization **(D2)**, a fuzzer must generate a program that enters the if-body before the compiler update. For a patch that modifies an existing optimization **(D3)**, a fuzzer must generate two types of programs: (1) a program that did not enter the if-body but enters the body after the update, or (2) a program did enter the if-body but does not enter after the update.

Note that when an existing optimization is modified **(D3)**, one may design a fuzzer that takes two target locations in two different compilers. However, implementing such a fuzzer may require a significant amount of effort because many existing well-written fuzzing frameworks work only for a single program [17]. Instead, we can simply consider the patch as a new introduction of the optimization (**D1**) and use a single-target directed fuzzer. This can generate programs not in the kernel which does enter the if-body both before and after the update. Nonetheless, this design is still useful in practice, as the optimization may contain a bug before the update unless formally verified. The details of our criteria for selecting the target location will be explained in §4.4.

3 Overview

3.1 Motivating Examples

We illustrate our approach with a miscompilation bug in LLVM shown in Fig. 1. This optimization was introduced on Sep 16, 2023. The remainder of the Euclidean division by a power of 2 is typically optimized with a bitwise operation: $X \ \% \ 2^N \implies X \ \& \ (2^N - 1)$ where X is an integer and N is a positive integer. The optimization in Fig. 1 was to extend the basic optimization to the case when

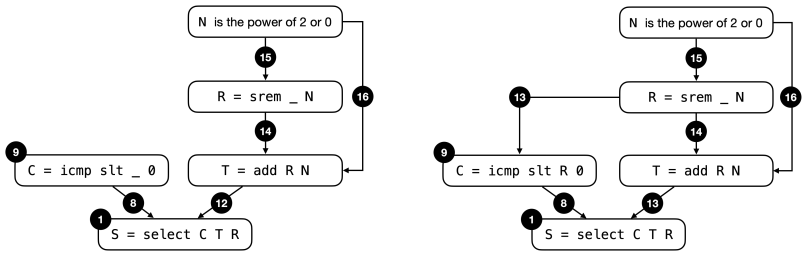


Fig. 3. Incorrect (a) and correct (b) optimization patterns. Each node and edge represent matching statements and their data dependencies, respectively. The numbers in circles represent the corresponding line numbers in the source code. Underscores indicate "don't-care".

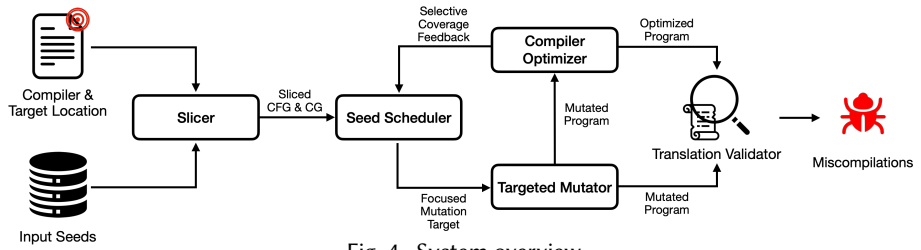


Fig. 4. System overview

the select instruction chooses between two values: $(X \ \% \ 2^N) < 0 \ ? \ (X \ \% \ 2^N) + 2^N : (X \ \% \ 2^N)$. When $X \ \% \ 2^N$ is negative, $(X \ \% \ 2^N) + 2^N$ is equivalent to $X \ \& \ (2^N - 1)$.

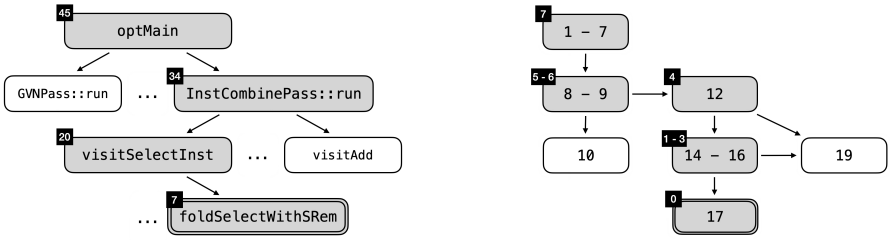
However, the code in Fig. 1 does not correctly implement the intended optimization. Fig. 2 shows an example of the miscompilation bug where the source and target programs produce different results. When the optimization was added, two test cases checked its correctness but failed to detect the bug. The bug was revealed seven months later and subsequently patched as shown in Fig. 1.

Compilers typically implement optimizations as rewrite rules that match a specific pattern in the input program. This miscompilation bug is caused by a subtle error in the pattern matching. Fig. 3(a) shows the incorrectly implemented pattern. Each number corresponds to the relevant line in the source code that implements the pattern-matching process. For example, the function (Line 1) is called when there exists a `select` instruction in the input program. The function then examines whether the condition of the `select` statement is an `icmp` instruction (Line 6) that checks if a value is less than zero (Line 7). If the input program satisfies all the conditions in the pattern, the compiler applies the optimization (Line 15). However, the implementation misses a data dependency between the `icmp` and `srem` instructions in the pattern. The correct pattern is shown in Fig. 3(b).

Notice that it is difficult to detect such miscompilation bugs because one should find a specific input program that matches all the conditions. State-of-the-art fuzzing tools for compilers are not effective in finding such bugs because they do not focus on a given target optimization. In our experience, state-of-the-art fuzzing tools for LLVM, such as FLUX [28] and Alive-Mutate [13], are not able to find this miscompilation bug within 1 hour.

3.2 Our approach: Optimuzz

We describe Optimuzz, a novel framework for finding miscompilation bugs. The system overview is shown in Fig. 4. Optimuzz takes as input a target source location in the compiler which corresponds to a specific optimization. The location can be either manually identified by developers or heuristically inferred by Optimuzz. We observed that each compiler optimization rule is typically



(a) Call graph of the LLVM optimization module (b) Control-flow graph of the `foldSelectWithSRem`

Fig. 5. Each node in the call graph (CG) represents a function within the optimizer, while each node in the control flow graph (CFG) corresponds to a line number in the target function. In the CG, shaded nodes indicate transitive callers of the target function. In the CFG, shaded nodes represent statements that can lead to the target optimization point. Double-edged nodes denote the target function and the target optimization point, respectively. Numbers at the top left of each node represent the distance to the target point.

implemented as an if-statement which contains the corresponding transformation like Line 15 in Fig. 1. Based on this insight, one can easily identify the target location for an optimization rule.

OPTIMUZZ can effectively generate input programs that trigger the miscompilation bug within only 36 minutes. The core idea is to employ directed fuzzing to generate input programs and use TV tools to detect miscompilation bugs. The rest of this section describes the key components of OPTIMUZZ each of which addresses a specific challenge in optimization-directed fuzzing.

3.2.1 Guided Search Strategy. We designed a guided search strategy to effectively generate input programs that trigger the target optimization. The strategy consists of two components: *selective coverage feedback* and *distance metric*.

OPTIMUZZ receives coverage feedback exclusively from relevant parts to the target optimization. Conventional grey-box fuzzers check the coverage of each execution and add new inputs to the seed pool if they cover new program locations. However, in our context, this uniform coverage feedback can promote unrelated inputs to the target optimization as seeds. This degrades the performance of fuzzing as the fuzzer spends time mutating irrelevant seeds.

OPTIMUZZ first analyzes the source code structure of the compiler and slices relevant parts to the target optimization. We note that compiler optimizations have simple hierarchical structures. Fig. 5(a) shows the call graph of the optimization module in LLVM. The main function of the optimizer, `optMain` calls each optimization pass such as `InstCombinePass`. Each pass then visits all the instructions in the input program and applies the corresponding optimization rules. Therefore, only the shaded nodes in the call graph can potentially reach the target function `foldSelectWithSRem` which contains the target optimization point. Similarly, the control-flow graph of the target function is shown in Fig. 5(b). The original version of the function contains a sequence of three nested conditional statements. Among them, only the shaded nodes can lead to the target optimization point. We observe that this structural pattern is common in other optimization passes in LLVM and even in other compilers like TURBOFAN, a JavaScript JIT compiler.

During the fuzzing phase, OPTIMUZZ keeps mutants that discover new coverage along the sliced paths as new seeds for further mutation. This process allows OPTIMUZZ to focus on the relevant conditions and discard seeds that are irrelevant to the target optimization.

Inspired other directed fuzzers [2, 11, 20, 21], OPTIMUZZ also employs a similar distance metric to guide the fuzzer toward the target optimization. The intuition is to measure the distance of each input program to the target optimization. If one program has a smaller distance value to the target than others, we expect that the program is more likely to trigger the target optimization.

```
define i32 @f(i32 %x, i32 %y) {
  %c = icmp slt i32 %x, 0
  %s = select %c, i32 -1, i32 %y
  ret i32 %s
}
```

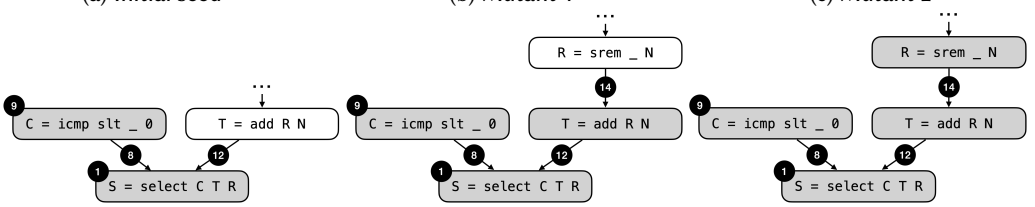
(a) Initial seed

```
define i32 @f(i32 %x, i32 %y) {
  %c = icmp slt i32 %x, 0
  %t = add i32 %x, 16
  %s = select %c, i32 %t, i32 %y
  ret i32 %s
}
```

(b) Mutant 1

```
define i32 @f(i32 %x, i32 %y) {
  %c = icmp slt i32 %x, 0
  %r = srem i32 %y, %x
  %t = add i32 %r, 16
  %s = select %c, i32 %t, i32 %y
  ret i32 %s
}
```

(c) Mutant 2



(d) Matched pattern by initial seed (e) Matched pattern by mutant 1 (f) Matched pattern by mutant 2

Fig. 6. Targeted mutation strategy of Optimuzz

Therefore, Optimuzz mutates those input programs more frequently than the others to generate more promising seeds.

The distance of an input program is computed based on the distance of each node covered by the input. The distance of a node in the target function is defined as the shortest length to the target optimization point. Fig. 5(b) shows the distance of each node in the target function `foldSelectWithSRem`. The distance of the target node 15 is defined as 0. The other distances are computed based on the shortest length to the target point. For inter-procedural paths, we set the weight of an edge between functions to 10 following the design of AFLGo [2]. Fig. 5(a) shows the distance of the entry point of each function in the optimizer. In the end, the distance of an input program is computed as the average distance to all nodes covered by the input.

Optimuzz then decides the number of mutants to generate from each seed based on its distance. Therefore, this distance metric allows the fuzzer to generate more promising input to trigger the target optimization at a higher rate.

3.2.2 Targeted Mutation Strategy. Optimuzz employs a targeted mutation strategy to effectively generate programs that trigger the target optimization. The intuition is also based on the observation of the typical pattern matching process in the compiler. We illustrate our strategy using the initial seed program in Fig. 6(a), which is available in LLVM's unit test suite [32]. Since this program contains the `select` and `icmp` instructions, it partially matches the target optimization pattern as shown in Fig. 6(d). This corresponds to the code coverage of Lines 1–7 in Fig. 1.

Initially, the fuzzer randomly transforms the seed program to derive mutant programs. Suppose the fuzzer adds an `add` instruction to define the second operand of the `select` instruction as shown in Fig. 6(b). This change results in a new matching pattern as shown in Fig. 6(e), which is closer to the target optimization pattern. Optimuzz captures this progress by observing the new code coverage of Line 10 in Fig. 1.

Now Optimuzz infers that the `add` instruction is a key component to satisfy more conditions for the optimization under test and applies a targeted mutation strategy around the instruction. Notice that compiler optimization patterns often involve a sequence of instructions that are data-dependent as shown in the motivating example. Thus, Optimuzz focuses on the data dependencies of the `add` instruction such as its operands and users. Especially, Optimuzz mutates the inferred relevant parts more frequently than other instructions. This in turn results in efficiently generating programs that match more sub-patterns for the target optimization like the mutant shown in Fig. 6(c).

Algorithm 1: OPTIMUZZ(C, t, Σ_0, Ω), where C is the compiler under test, t is the target location, Σ_0 is the initial seed pool, and Ω is the translation validator.

```
1  S ← Slice(C, t)                                                  // §4.2.1
2  Σ ← SelectSeed(Σ_0, S)                                           // §4.2.3
3  ⟨Candidate, Bug⟩ ← ⟨∅, ∅⟩
4  while not timeout do
5      ⟨s, dist, δ⟩ ← Choose(Σ)
6      e ← AssignEnergy(dist)                                       // §4.2.4
7      for e times do
8          ⟨s', δ'⟩ ← Mutate(s, δ)
9          cov ← MeasureCov(S, s')
10         dist' ← ComputeDist(C, t, s')                            // §4.2.2
11         if cov has any gain then
12             Σ ← Σ ∪ {⟨s', dist', δ'⟩}
13         if s' covers t then
14             Candidate ← Candidate ∪ {s'}
15 for s ∈ Candidate do
16     if Ω(s, C(s)) reports a miscompilation then
17         Bug ← Bug ∪ {s}                                          // Miscompilation bugs
18 return Bug
```

We demonstrate that this idea is generally applicable to various optimization rules (e.g., InstCombine, GVN) and even to other compilers for different languages. In the evaluation, we show that OPTIMUZZ can effectively find miscompilation bugs in TURBOFAN, a JavaScript JIT compiler.

4 Framework

This section describes OPTIMUZZ, a general optimization-directed fuzzing framework for compilers. The overall process is described in Algorithm 1. OPTIMUZZ is parameterized by the compiler under test C with the target location t, the initial seed pool Σ_0, and the translation validator Ω. Given the target location t, OPTIMUZZ first slices the compiler source code to extract the relevant code for the target location (Line 1). Next, OPTIMUZZ selects promising seeds from the initial seed pool Σ_0 based on their distance to the target location (Line 2). Then, OPTIMUZZ iteratively selects a seed from the pool and assigns energy to the seed based on the distance to the target location (Lines 5–6). The energy determines the number of times the seed should be mutated (Line 8). For each mutation, OPTIMUZZ measures the code coverage of the mutated seed (Line 9) and computes the distance to the target location (Line 10). If the mutated seed newly covers nodes in the sliced graph, OPTIMUZZ adds the seed to the pool (Line 12). If the mutated seed covers the target location, OPTIMUZZ adds the mutated seed to the candidate set (Line 14). All the seeds in the candidate set are then validated by Ω (Line 17). Finally, OPTIMUZZ returns the set of seeds that trigger miscompilations.

In the rest of this section, we describe the key components of OPTIMUZZ, including the guided search and targeted mutation. We also describe our deployment model for testing real-world compilers and the criteria for selecting target locations in the compiler source code.

4.1 Preliminaries

We assume each function in the compiler is represented as a control-flow graph (CFG). The CFG of function f is a directed graph $(\mathbb{C}_f, \rightarrow_f)$ that represents the control flow within the function where

$\mathbb{C}_f$ is the set of program points within the function and $(\rightarrow_f) \subseteq \mathbb{C}_f \times \mathbb{C}_f$ is the set of control-flow edges between program points. We denote the CFG of f as CFG_f and a node in CFG_f as n_f. We define the inter-procedural CFG (ICFG) as a directed graph $(\mathbb{C}, \rightarrow)$ that represents the control-flow relations of the entire compiler where $\mathbb{C} = \bigcup_{f \in \mathbb{F}} \mathbb{C}_f$ is the set of program points in the entire compiler and $(\rightarrow) \subseteq \mathbb{C} \times \mathbb{C}$ is the set of control flow edges between program points. We define a call graph (CG) as a directed graph $(\mathbb{F}, \rightarrow_{\mathbb{F}})$ that represents the call relations between functions in the compiler where $\mathbb{F}$ is the set of functions in the compiler and $(\rightarrow_{\mathbb{F}}) \subseteq \mathbb{F} \times \mathbb{F}$ is the call relations.

4.2 Guided Search Strategy

This section describes how OPTIMUZZ guides the fuzzing process to reach the target location. We first slice the compiler source code to extract the relevant code for the target location (§4.2.1). The extracted code is then instrumented to provide selective coverage feedback during the fuzzing process and used to compute the distance of seeds to the target location (§4.2.2). In following sections, we describe how OPTIMUZZ selects seeds (§4.2.3) and assigns energy to seeds based on their distance to the target location (§4.2.4).

4.2.1 Slicing. We first define the slice SF on the call graph which is the set of functions that are relevant to the target location t: $SF = \{f \in \mathbb{F} \mid f \rightarrow_{\mathbb{F}}^* f_t\}$. Each f_t is the function that contains the target location t. OPTIMUZZ considers transitive callers of the target function f_t as relevant to the target location. The intuition behind this design choice is based on the common structure of compiler optimization processes as shown in Fig. 5(a). The optimization process is typically implemented as a sequence of passes such as InstCombine or GVN. Each pass also consists of a sequence of rewrite rules each of which is responsible for a specific optimization such as foldSelectWithSRem. One of such rewrite rules is typically the target optimization that we aim to trigger. Our slicing strategy is designed to capture such a hierarchical structure of compilers and guide the fuzzing process to reach the target optimization efficiently without being deviated by irrelevant code.

One may wonder what happens if the target optimization is triggered only after another specific optimization is applied. In this case, our slicing strategy misses the dependencies between optimizations. However, we observed that this is rare in practice. Even if such dependencies exist, OPTIMUZZ still has a chance to generate an input program that triggers both the preceding and target optimizations, similar to conventional unguided fuzzers.

We define the slice S on ICFG as the set of all relevant program points to the target location t, formally, $S = \{c_f \in \mathbb{C} \mid c_f \rightarrow^* t \wedge f \in SF_t\}$. Based on the same intuition as above, we exploit the hierarchical structure of the compiler optimization process within a function as shown in Fig. 5(b). In practice, each function in the optimizer (e.g., foldSelectWithSRem) corresponds to a specific rewrite rule that is responsible for a specific optimization. Thus, this strategy effectively guides the search to the target optimization within the function.

4.2.2 Seed Distance. OPTIMUZZ computes seed distance to estimate how close a seed is to reach the designated target locations in the compiler (Line 10). This distance is defined based on the nodes covered by the seed within the slice. This metric helps prioritize seeds that are more likely to reach the target efficiently during the fuzzing process.

First, we define the weight of an edge (n_1, n_2) in ICFG based on the slice S as follows:

$$\text{Weight}_S(n_1, n_2) = \begin{cases} 1 & \text{if } n_1 \rightarrow n_2 \text{ is an intraprocedural edge} \wedge n_1, n_2 \in S \\ 10 & \text{if } n_1 \rightarrow n_2 \text{ is an interprocedural edge} \wedge n_1, n_2 \in S \\ 0 & \text{otherwise} \end{cases}$$

Similar to the previous work [2], we assign different weights to edges based on whether they are intraprocedural or interprocedural. This differentiation ensures that paths crossing function boundaries are appropriately penalized, making intraprocedural exploration more favorable. However, we assign weights to edges only if both the source and destination nodes are in the slice. Notice that if an edge is not in the slice, the weight is set to 0, effectively excluding the edge from the distance calculation.

The distance of a node n to the target location t is defined as the shortest length from n to t:

$$\mathsf{Dist}_{S,t}(n) = \sum_{n_1 \to n_2 \in SP} \mathsf{Weight}_S(n_1, n_2)$$

where SP is the shortest path from n to t based on the weights assigned to the edges.

Finally we define the distance of a seed s to a specific target t. The distance is defined as the average of the distances from all sliced nodes covered by the seed to the target location:

$$\mathsf{Dist}_{S,t}(s) = \frac{1}{|\mathbb{C}_s|} \sum_{n \in \mathbb{C}_s} \mathsf{Dist}_{S,t}(n)$$

where $\mathbb{C}_s$ represents the set of sliced nodes in the ICFG that are covered by seed s: (i.e., $\mathbb{C}_s = \mathbb{C} \cap S$). This average distance provides a quantitative measure of how well the seed is exploring regions of the program close to the target. Seeds with smaller average distances are considered closer to reaching the target and are therefore more valuable for guiding the fuzzing process.

4.2.3 Initial Seed Selection. Given an initial seed pool Σ_0, OptiMuzz selects seeds that are more promising for reaching the target locations (Line 2) The seed selection is based on the distance of each seed to the target locations. We select seeds that are the closest to the target locations, as these are the most promising candidates for further exploration. This in turn increases the likelihood of quickly reaching the target locations and triggering the desired optimizations.

4.2.4 Energy Assignment. Inspired by existing grey-box fuzzers [2–4, 20, 21], OptiMuzz defines the energy of a seed to guide the fuzzing process (Line 6). The energy of a seed determines the number of mutants generated from it. OptiMuzz assigns energy to seeds based on their distance from the target locations. Based on this energy assignment, OptiMuzz generates more mutants from seeds that are closer to the target locations. This increases the likelihood of reaching the target locations and triggering the desired optimizations.

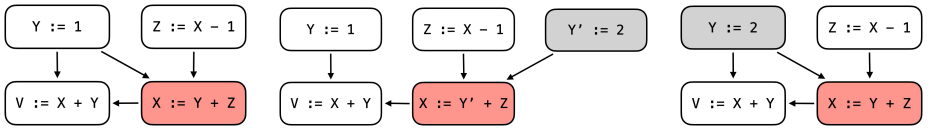
The energy $E(s)$ of a seed s is defined using the distance to the target locations: $\lceil \frac{10}{1+\mathsf{Dist}_{S,t}(s)} \rceil$. The formula is designed to ensure that (1) seeds closer to the target locations receive more energy, and (2) the energy is bounded (between 1 to 10) to prevent it from becoming too large.

4.3 Targeted Mutation Strategy

This section describes a generic framework for the targeted mutation strategy. Our framework is parameterized by a set of basic mutation operators. Among them, our targeted mutation strategy is applied to mutation operators that preserve the control-flow of input programs. Note that this strategy is generally applicable to different types of programming languages and compilers. In the evaluation, we demonstrate the effectiveness of this strategy on LLVM and JavaScript programs.

In the rest of this section, we first formalize the representation of input programs (§4.3.1). Then, we describe the basic mutation operators and a naive fuzzing strategy (§4.3.2). Finally, we present our targeted mutation strategy (§4.3.3).

4.3.1 Input Program Representation. An input program P is represented as a control-flow graph (CFG) whose node is a program point. We assume that each program point is associated with a



(a) Def-Use graph (b) InsertCmd (c) ChangeExp

Fig. 7. Def-Use graph of and mutants of a program `Y := 1; Z := X - 1; X := Y + Z; V := X + Y`. The red node is the last mutated node of the input program and the grey nodes are newly mutated nodes.

command. For brevity, we consider the following simple imperative language.

$$C \to x := E \qquad E \to n \mid x \mid E\ Op\ E \qquad Op \to + \mid - \mid \cdots$$

4.3.2 Basic Mutation Operators. Optimuzz considers two basic mutation operators:

- $\mathsf{InsertCmd}(c, x, c', x', E)$ inserts a new node c' before node c where c' defines a new variable x' with expression E and x is replaced with x' in c.
- $\mathsf{ChangeExp}(c, E)$ changes the expression in the command at node c to expression E.

These operators cover a wide range of possible mutations that can be applied to input programs in various languages. While the first operator inserts new nodes, it does not introduce conditionals or loops, ensuring that the control flow remains unchanged.

A naive fuzzer would randomly apply these mutations to the input program. That is, it would randomly select a node and apply one of the mutation operators with randomly generated expressions. However, this approach does not effectively guide the fuzzing process toward the target optimization because of the huge search space of possible mutations.

4.3.3 Targeted Mutation Strategy. Our targeted mutation strategy is based on the def-use graph (DUG) of input programs. We use the standard def-use relations between commands as follows:

$$c_1 \overset{x}{\hookrightarrow} c_2 \iff \text{variable } x \text{ is defined in } c_1 \text{ and used in } c_2.$$

Such def-use relations can be simply obtained by the SSA form of input programs derived by most modern compilers. During the fuzzing process, Optimuzz keeps track of the last mutated node δ of each mutant if the mutant covers new nodes in the slice (Line 8 and 12 in Algorithm 1). When the mutant is chosen for further mutation, Optimuzz prioritizes the next node to mutate based on the def-use relations between the lastly mutated node δ and the nodes in the mutant (Line 5 and 8).

Given the lastly mutated node δ of a mutant, Optimuzz instantiates the following mutation operators based on the def-use relations of δ:

$$\{\mathsf{InsertCmd}(\delta, x, c, x', E) \mid x' \text{ is a new variable, } E \text{ is a randomly generated expression}\}$$
$$\cup\ \{\mathsf{ChangeExp}(c, E) \mid c \overset{x}{\hookrightarrow} \delta,\ E \text{ is a randomly generated expression}\}$$
$$\cup\ \{\mathsf{ChangeExp}(c, E) \mid \delta \overset{x}{\hookrightarrow} c,\ E \text{ is a randomly generated expression that uses } x\}$$

The first set of mutations inserts new nodes that define new variables and replace existing variables in the last mutated node δ. The second set of mutations changes the expressions in the nodes that define variables used in δ. Similarly, the third set of mutations changes the expressions in the nodes that use variables defined in δ.

Fig. 7 illustrates an example of our targeted mutation strategy. Suppose the last mutated command δ of the input program is $X := Y + Z$. Based on the def-use relations of δ, Optimuzz can generate a mutant shown in Fig. 7(b) by using mutation $\mathsf{InsertCmd}(\delta, y, c, y', 2)$ where c is the grey node. This mutation inserts a new node $Y' := 2$ before node δ and replaces Y with Y' in δ. Similarly, Optimuzz generates a mutant in Fig. 7(c) by using mutation $\mathsf{ChangeExp}(c_1, 2)$ where c_1 is the grey node.

```
1 Instruction *Optimize() {
2 +   if (A)
3 +       return Opt1(); // Target
4
5     if (C)
6         return Opt2();
7     return nullptr;
8 }
```

(a) New optimization added

```
1 Instruction *Optimize() {
2     if (A)
3 +       if (B)
4             return nullptr;
5     if (C)
6         return Opt1(); // Target
7     return nullptr;
8 }
```

(b) Condition weakened

```
1 Instruction *Optimize() {
2     if (A)
3 +       if (B)
4             return Opt1();// Target
5     if (C)
6         return Opt2();   // Target
7     return nullptr;
8 }
```

(c) Condition strengthened

Fig. 8. Target selection criteria

4.4 Deployment Model and Target Selection Criteria

We designed two deployment models for OPTIMUZZ: *continuous* and *batch* mode. In continuous mode, OPTIMUZZ aims to validate recently updated optimizations. The goal of the fuzzer is to generate an approximated kernel for each update. We also apply OPTIMUZZ in batch mode where OPTIMUZZ aims to validate the whole set of optimizations. This batch mode is also meaningful in practice unless the underlying optimizations are not formally verified. In the evaluation, we demonstrate both modes of OPTIMUZZ are effective in discovering new miscompilations in real-world compilers.

For continuous mode, we use simple yet effective criteria to select the target location in the compiler code. The principle is that we select the body of an if-statement that handles more input programs in the new compiler than the previous version. Intuitively, if an updated rewrite rule handles more cases than before, we validate the correctness of the optimization for the new cases.

Our criteria are designed to cover three common update patterns of real-world compilers such as LLVM and TURBOFAN. First, Fig. 8(a) illustrates an update that introduces a new rewrite rule. In this case, all programs that match the condition `A` are optimized by the new optimization `Opt1`. Among them, the programs that also match the condition `C` are not optimized by `Opt2` anymore. Thus, we set the target location to Line 3 for fuzzing. Second, we consider a condition loosening of an existing rewrite rule as shown in Fig. 8(b). The optimization condition for `Opt1` is updated from $\neg A \wedge C$ to $\neg(A \wedge B) \wedge C$. Such changes may introduce an optimization opportunity that was not available before. Thus, we set the target locations to the bodies of all subsequent if-statements in the function such as Line 6 to validate the optimization with the loosened condition. If there is no if-statement after the changed line, we set the target location to the return statement of the function. Lastly, we consider a condition strengthening of an existing rewrite rule as shown in Fig. 8(c). The optimization condition for `Opt1` is updated from A to $A \wedge B$. This pattern is an exception to the general principle as it reduces the optimization opportunities for `Opt1`. However, this pattern is still meaningful in practice because it represents common bug fixes in rewrite rules. In this case, we set the target location to Line 4. We also set the target location to Line 6 based on the second pattern.

Among the selected branches using the criteria, we only consider if-branches with interesting return values. In LLVM and TURBOFAN, optimization functions typically return `true` or newly allocated objects if the optimization is applied. On the other hand, they return `false` or `nullptr` when the optimization is not applied. Therefore, OPTIMUZZ filters out such non-interesting branches.

For batch mode, we basically select all branches as target locations if they return interesting values. We further discard branches that immediately return the result of other helper functions like `if (Res = Helper()) return Res;` because meaningful branches in `Helper` are already targeted. If there are too many potential target locations, we can heuristically select a subset of them such as the branches in more error-prone files according to the commit history.

5 Evaluation

We evaluate the performance of OPTIMUZZ to answer the following research questions:

RQ1 How effective is OPTIMUZZ in terms of reproducing target miscompilation bugs?

RQ2 How do the guided search and targeted mutation affect the performance of Optimuzz?
RQ3 How do different distance metrics affect the performance of Optimuzz?
RQ4 How effective is Optimuzz in revealing unknown miscompilation bugs?

5.1 Experimental Setup

We instantiated Optimuzz for LLVM and TurboFan. LLVM is a popular compiler infrastructure for various programming languages [24]. TurboFan is a JIT compiler for the V8 JavaScript engine [16]. All the experiments are performed on Linux machines with 512GB RAM and Intel Xeon 2.90GHz.

For LLVM, we implemented a custom fuzzer for LLVM IR using a set of simple mutation operators such as instruction insertion, opcode mutation, operand mutation, and type mutation. Our fuzzer handles middle-end optimizations, such as instcombine, slp-vectorization, and GVN. We compare Optimuzz with two state-of-the-art compiler fuzzers for LLVM IR: FLUX [28] and Alive-Mutate [13]. We chose those tools because they generate or mutate LLVM IR to detect miscompilation bugs in LLVM. For initial seeds, we used the LLVM unit test suite published by FLUX [29]. We employed Alive2 [57] for translation validation of LLVM IR.

For TurboFan, we implemented Optimuzz on top of Fuzzilli [17], a state-of-the-art fuzzer for JavaScript. Note that the vanilla Fuzzilli failed to generate any input program to trigger the target optimizations in our benchmark within 48 hours. Therefore, we configured Fuzzilli using the settings published by TurboTV [23]. This restricts Fuzzilli to only generate programs with semantics encoded in TurboTV and always generates functions for JIT compiling. Additionally, we limited the constants used by the fuzzer and updated the probability of function arguments being used within the function body. We used this configured Fuzzilli as our baseline and implemented our strategies atop it. For initial seeds, we used mjsunit [15] unit test suite used by V8. We employed TurboTV for translation validation of TurboFan IR.

We evaluate the performance of Optimuzz using known miscompilation bugs in LLVM and TurboFan. For LLVM, we collected *all* the miscompilation bugs over the past three years if they can be detected by Alive2. As a result, we collected 24 miscompilation bugs, excluding crash bugs and assertion errors which are out of the scope of Alive2. These bugs involve 7 optimization passes including instcombine, vector-combine, and slp-vectorizer. For TurboFan, we used the miscompilation bugs addressed in the TurboTV paper [23]. Among the 9 bugs in the paper, we excluded 3 bugs because Fuzzilli was not compatible with the buggy versions of TurboFan.

5.2 RQ1. Reproducing Target Miscompilation Bugs

We evaluate the performance of Optimuzz in reproducing known miscompilation bugs in LLVM and TurboFan. We compare Optimuzz with FLUX and Alive-Mutate for LLVM, and with Fuzzilli for TurboFan. Due to the randomness of fuzzing, we measure the median time to trigger each target bug over 10 repetitions with the time limits of 1 hour for LLVM and 6 hours for TurboFan. We then validated the generated input programs using TV tools if they triggered the target optimizations.

Table 1 shows the effectiveness of Optimuzz in reproducing known miscompilation bugs of LLVM. Optimuzz successfully reproduced 23 bugs while existing non-directed tools failed to reproduce most of them. FLUX did not reproduce any bugs more than four times. Alive-Mutate only reproduced Issue 55291 in 53 minutes whereas Optimuzz detected the bug in only 14 minutes. These results indicate that our directed approach is more effective in reproducing target miscompilation bugs than FLUX and Alive-Mutate.

Although Optimuzz is effective in most cases, there is one failing case: Issue 84025. Fig. 9 illustrates this miscompilation bug. The function `src` contains bitwise operations $((x \ll 32) \mid 65) \ll 64$ that generate an `i128` type value which is then converted to a vector-type value $\langle 0, 0, 65, x \rangle$ of length 4. LLVM optimizes them into an `insertelement` operation that locates the pre-calculated

Table 1. Comparison of Optimuzz with baseline tools in reproducing known miscompilation bugs in LLVM. We report the median time of reproductions over 10 repetitions in minutes. Dashes (-) indicate that the tool failed to reproduce the bug for more than half of the repeated experiments. The numbers in parentheses indicate the number of successful reproductions within the time limit. Column **Pass** represents the optimization pass that caused the miscompilation such as InstCombine (IC), Reassociate (R), InstSimplify (IS), Correlated-Propagation (CP), and Slp-Vectorization (SV).

Issue	**Pass**	FLUX	Alive-Mutate	Optimuzz	**Issue**	**Pass**	FLUX	Alive-Mutate	Optimuzz
55291 [33]	IC	- (0)	53 (6)	14 (10)	70470 [45]	IC	- (0)	- (0)	24 (8)
57357 [34]	IC	- (0)	- (0)	16 (10)	72911 [46]	IC	- (0)	- (1)	16 (10)
57683 [35]	R	- (0)	- (0)	14 (10)	74890 [47]	IC	- (0)	- (0)	14 (10)
58977 [36]	IS	- (0)	- (0)	19 (9)	75437 [48]	SV	- (0)	- (0)	21 (10)
59279 [37]	IC	- (0)	- (0)	15 (10)	76441 [49]	IC	- (0)	- (0)	56 (6)
59301 [38]	CP	- (4)	- (2)	15 (10)	84025 [50]	IC	- (0)	- (0)	- (0)
59836 [39]	IC	- (0)	- (1)	15 (10)	89390 [51]	VC	- (0)	- (0)	14 (10)
61312 [40]	IC	- (0)	- (0)	18 (10)	89516 [52]	IC	- (0)	- (0)	36 (7)
62401 [41]	IC	- (0)	- (0)	20 (10)	89669 [53]	IC	- (0)	- (0)	24 (9)
62901 [42]	CP	- (0)	- (0)	15 (10)	91417 [54]	R	- (0)	- (0)	14 (10)
63327 [43]	IC	- (0)	- (0)	14 (10)	98753 [55]	SV	- (0)	- (0)	22 (10)
64339 [44]	IC	- (0)	- (0)	34 (7)	98838 [56]	IC	- (0)	- (0)	24 (10)

```
define <4xi32> @src(i32 %X){
  %a = zext i32 %X to i64
  %b = shl i64 %a, 32
  %c = or i64 %b, 65
  %d = zext i64 %c to i128
  %e = shl i128 %d, 64
  %f = bitcast %e, <4xi32>
  ret <4xi32> %f
}
; @src(1) = <0, 0, 65, 1>
```

(a)

```
define <4xi32> @tgt(i32 %X){
  %a = insertelement <0,0,0,poison>, %X, 3
  ret <4xi32> %a
}
; @tgt(1) = <0, 0, 0, 1>
```

(b)

```
define <2xi32> @mut(i32 %X){
  %a = zext i32 %X to i64
  %b = shl i64 %a, 32
  %c = or i64 %b, %X
  %d = bitcast %c, <2xi32>
  ret <2xi32> %d
}
```

(c)

Fig. 9. Miscompilation bug in LLVM (Issue 84025). Program (a) is incorrectly optimized to program (b) by LLVM. Optimuzz generates program (c) to trigger the optimization but fails to reproduce the bug.

values at specific indices of the vector. However, the optimization fails to locate the value 65 in the vector as shown in Fig. 9(b). This bug can occur when the resulting vector is longer than 2.

Optimuzz is not effective in this case because the optimization rule depends on a long sequence of multiple instructions with complex relationships among them. This rule is generally applicable to any N-bit integer-type value and generates an n-bit vector-type value with N/n elements. To derive a diverse set of such input programs, Optimuzz needs to not only mutate the length of the resulting vector but also generate the preceding sequence of instructions each of which is consistent with the vector. For example, Optimuzz was able to generate an input program that triggers this optimization with a vector of length 2, as shown in Fig. 9(c). To trigger the bug, Optimuzz must mutate the length of the vector to at least 3 and generate the preceding instructions (`zext`, `shl` and `or`) that correctly populate the vector. However, our strategies failed to further generate input programs with this complex constraint.

Next, we evaluate the effectiveness of Optimuzz in TurboFan. We also iterated 10 times for each fuzzer and for each bug. Note that TurboFan is a JIT compiler for JavaScript, which makes the fuzzing process more complex than that of LLVM. Thus, we set the time limit to 6 hours. Similar to the LLVM experiments, we measured the median time to trigger the target bug.

Table 2. Comparison of Optimuzz with the baseline tools in reproducing known miscompilation bugs in TurboFan. **Phase** means the optimization phase where the bug exists.

Issue	Phase	Fuzzilli	Optimuzz
1195650 [5]	simplified-lowering	- (0)	- (0)
1198705 [6]	simplified-lowering	- (0)	3h 26m (10)
1199345 [7]	simplified-lowering	- (0)	3h 57m (9)
1200490 [8]	simplified-lowering	- (0)	- (0)
1234764 [9]	machine-operator-reducer	- (0)	4h 35m (6)
1234770 [10]	machine-operator-reducer	- (0)	2h 39m (10)

Table 2 shows the experimental results. The baseline undirected fuzzer, Fuzzilli reproduced no bugs within 6 hours, while Optimuzz successfully reproduced 4 bugs. Note that it is more challenging to generate optimization-triggering input programs for TurboFan than for LLVM. LLVM takes an input program in LLVM IR and performs same optimizations on the same IR. This consistency allows the fuzzer to directly mutate the input program based on the coverage feedback. However, TurboFan takes a JavaScript program as an input and translates it to a TurboFan IR program. Furthermore, TurboFan IR consists of high-level (JS layer), middle-level (Simplified layer), and low-level (Machine layer) IR. The compiler sequentially translates the input program from a high-level IR to a low-level IR and performs different optimizations on each layer. This complexity makes it difficult to directly check if the target optimization is triggered based on the coverage feedback. Nevertheless, Optimuzz successfully reproduced the majority of bugs. This demonstrates that Optimuzz is effective for complex compilers like TurboFan.

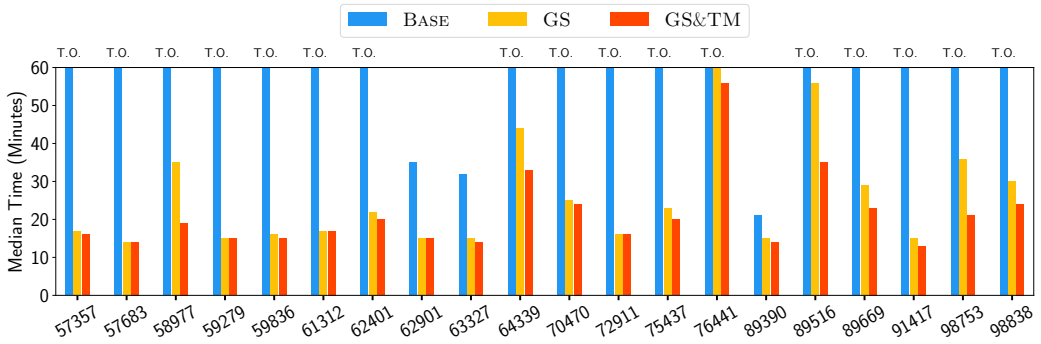
Lastly, we measure the success rate of our mutation operators of Optimuzz in LLVM to assess the overall quality of the mutation. Overall, 72.99% of the generated programs are compilable over 10 repetitions for all bugs. In contrast to other differential testing tools for LLVM [66], Optimuzz is not concerned about undefined behaviors (UB) in the generated programs. UB can lead to different optimization results. While this divergence is correct according to the language specification, it may produce false positives in conventional differential testing. On the other hand, we use a TV tool to detect miscompilation bugs even if UBs are involved by checking refinement relations. Therefore we accept every compilable program as a valid input program. For TurboFan, we used the existing mutation operators of Fuzzilli. They are designed to always generate syntactically correct JavaScript programs [17] and JavaScript does not have UBs like LLVM.

Overall, the results indicate that our ideas are effective and generally applicable to different types of compilers. Existing non-directed fuzzers do not perform well in triggering target optimizations. However, Optimuzz has successfully reproduced the majority of bugs and shown effectiveness both in AOT (ahead-of-time) and JIT compilers of different languages, LLVM and TurboFan.

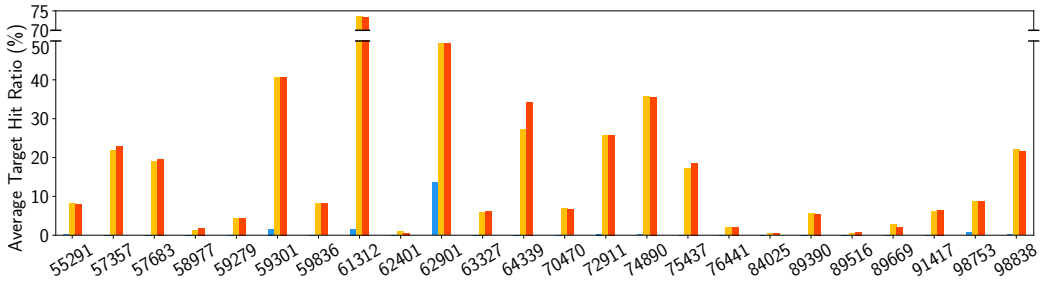
5.3 RQ2. Impact of Guided Search Strategy & Targeted Mutation Strategy

We evaluate the impact of the guided search strategy and targeted mutation strategy of Optimuzz in reproducing known miscompilation bugs and the target hit ratio. We instantiated Optimuzz in 3 different configurations: Base, GS, and GS&TM. Base is a basic grey-box undirected fuzzer without any strategy for directed fuzzing. GS employs the guided search strategy, and GS&TM combines the guided search strategy with the targeted mutation strategy. We ran each configuration 10 times for each bug and measured the median time to trigger the target bugs.

5.3.1 Impact on LLVM. Fig. 10(a) shows the impact of each strategy for LLVM. For each variant of Optimuzz, we measured the median time to trigger the target bug. Base only reproduced 6 bugs within the time limit. The guided search strategy significantly improved the performance, allowing



(a) Time to reproduce target miscompilation bugs. Cases where all variants failed to reproduce within one hour or succeeded in reproducing within 20 minutes are excluded.



(b) The average ratio of unique target-reaching programs to all uniquely generated programs.

Fig. 10. Impact of the guided search and targeted mutation strategies on LLVM.

GS to reproduce 22 bugs. For example, the guided search strategy is particularly effective when the path to the target optimization points is long and optimization conditions are complex. In Issue 59279, there are four call points to the target function, and 13 conditions within the function must be met for optimization. Base fails to reproduce the bug within the time limit, but GS succeeds in only 15 minutes. The targeted mutation strategy further boosts performance, especially when the input program being mutated is large. For Issue 76441, to reproduce the bug, an input program must either contain 7 instructions, as stated in the previous bug report, or 4 instructions using vector-type values with `undef` elements, as Optimuzz discovered. Base and GS failed to discover the bug within the time limit but GS&TM succeeded. In total, combined with the targeted mutation strategy, GS&TM reproduced 23 bugs.

GS&TM significantly improved the performance over GS for Issues 58977 and 98753. This improvement shows that the targeted mutation strategy effectively guides the fuzzer when the code slice is large. For those cases, the guided search strategy selected 1043 and 1062 basic blocks as relevant program points to the target optimizations, respectively. However, the guided search strategy selected only 255 basic blocks on average for each bug. Such large code slices can hinder the effectiveness of fuzzing when only the selective coverage feedback is applied. This suggests that both strategies work synergistically to enhance the performance of Optimuzz.

Next, Fig. 10(b) shows the average target hit ratio of each variant of Optimuzz achieved within the time limit of one hour. GS and GS&TM significantly improved the target hit ratio for all bugs compared to Base. This demonstrates that the guided search strategy effectively directs the fuzzer to generate optimization-triggering input programs. Note that for many cases, the hit ratio of GS&TM is not improved compared to GS even though the time to reproduce the bug is highly reduced such as Issue 98753. This is because GS often mutates inessential parts of the input programs once the target is reached. Such programs still trigger the target optimizations but do not diversify

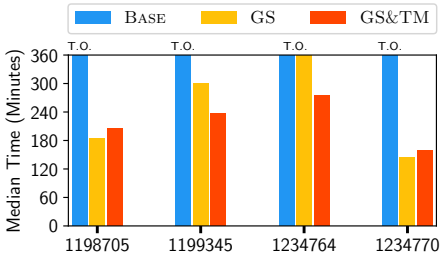


Fig. 11. Impact of guided search strategy and targeted mutation strategy on TurboFan. Issues where all tools failed to reproduce within the time limit are excluded.

Table 3. Comparison of Optimuzz configurations in producing unique input programs that trigger TurboFan's target optimizations within 6 hours.

Issue	Base	GS	GS&TM
1195650 [5]	87	446	549
1198705 [6]	320	1,019	1,008
1199345 [7]	164	3,836	3,810
1200490 [8]	17	798	851
1234764 [9]	0	120	146
1234770 [10]	15	312	514

the optimization-triggering conditions. This result indicates that both strategies are essential to effectively find an approximation of the kernel of a compiler update, as described in §2.

5.3.2 Impact on TurboFan. Fig. 11 shows the performance of Optimuzz with different strategies for reproducing bugs. GS successfully reproduced 3 bugs while Base failed to reproduce any bugs. GS&TM reproduced 4 bugs, including Issue 1234764 which GS failed to reproduce. The optimization condition of the issue requires a specific structure with 4 or more instructions. The targeted mutation strategy helps the fuzzer effectively select instructions in the input program that need to be modified We observe that GS&TM did not consistently outperform GS. The main reason is the gap between the input language (JavaScript) and the layered intermediate representation (TurboFan IR) as described in §5.2. Nonetheless, the targeted mutation strategy is still effective and significantly reduces the time spent by more than 1 hour for Issue 1199345 and 1234764.

Table 3 shows the average number of input programs that reach the target locations. Base was able to generate only a few programs that reach the target optimization. On the other hand, GS and GS&TM generate between 3 to 50 times more input programs that trigger the target optimization compared to Base. Notably, while Optimuzz could not successfully reproduce Issues 1195650 and 1200490, it generated more than 500 input programs that reached the target optimizations.

The overall results indicate that our strategies effectively create approximated kernels for testing target optimizations. Base without any strategies generates only a limited number of input programs that trigger the target optimization, often resulting in incomplete bug reproduction. Our strategies are generally effective for both LLVM and TurboFan. GS&TM, with all strategies applied, demonstrates the best performance.

5.4 RQ3. Impact of Different Distance Metrics

This section evaluates the impact of our distance metric on the performance of Optimuzz. We instantiated $\text{Optimuzz}_{\text{AFLGo}}$ which employs the distance metric of AFLGo [2], a widely used distance metric in the fuzzing community. AFLGo computes the distance from a source basic block n in function f_n to the target basic block n^* in f_{n^*} in two stages: (1) intra-procedural path from n to a call site to a function g that transitively calls f_{n^*}, and (2) inter-procedural call chain from g to f_{n^*}. In contrast, our metric computes the shortest paths in the ICFG from the source to the target, including all basic blocks and call edges along the paths. This makes our distance more sensitive to all optimization conditions along the entire paths. We ran $\text{Optimuzz}_{\text{AFLGo}}$ for all 24 cases in the LLVM benchmark and measured the time to trigger the target bugs.

Fig. 12 shows the experimental results for Issue 58977, 61312, 64339, and 98753. We focus on these cases because $\text{Optimuzz}_{\text{AFLGo}}$ and Optimuzz assign significantly different distance values to basic blocks in these cases. In other cases, they assign similar distance values to basic blocks and thus no performance differences are observed. We observed that Optimuzz significantly

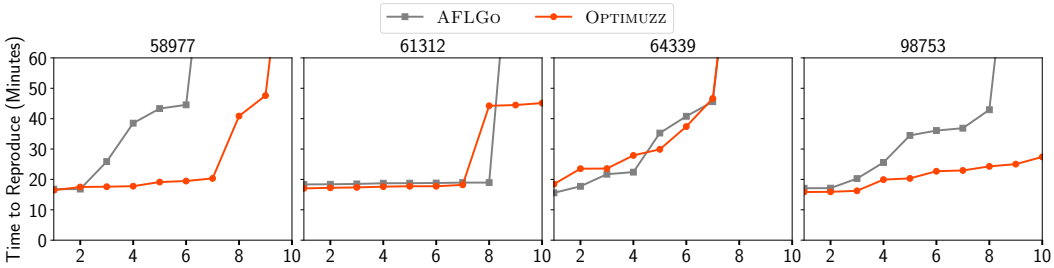


Fig. 12. Impact of different distance metrics. The x-axis represents repetitions and the y-axis represents the sorted time to reproduce the target bugs in ascending order.

Table 4. New bugs detected in each optimization pass of the LLVM latest version.

Mode	InstCombine	InstSimplify	Correlated-Propatation	Slp-Vectorization	GVN	VectorCombine
Continuous	3	0	3	0	0	1
Batch	39	2	0	4	2	1

outperformed Optimuzz$_{\text{AFLGo}}$ for Issue 58977 and 98753. This is mainly because our distance metric more effectively penalizes input programs that do not satisfy the target optimization conditions than AFLGo. We observed that those optimizations require much longer sequences of instructions to reach the target optimization. In such cases, Optimuzz assigns higher distance values to inputs that do not reach the target optimization than Optimuzz$_{\text{AFLGo}}$. Thus promising inputs that are closer to the target optimization are more emphasized in Optimuzz.

Next, we do not observe significant performance differences in Issue 61312 and 64339. The main reason is the poor quality of the initial seed programs. For those cases, the seed programs already satisfy the target optimization conditions but contain many irrelevant instructions. This introduces nonessential mutations that do not affect the target optimization. Therefore, the performance is less sensitive to the guidance by the distance metric but rather to the impact of mutation operations.

The overall results demonstrate the effectiveness of our distance metric. Our distance metric can effectively guide the fuzzer to generate target-reaching seeds. This is especially useful when the target optimization requires a long sequence of instructions to reach the target point.

5.5 RQ4. Effectiveness of Revealing Unknown Miscompilation Bugs

We evaluate the effectiveness of Optimuzz in revealing unknown miscompilation bugs in the latest version of LLVM. We instantiated Optimuzz in two deployment modes: continuous and batch, as described in § 4.4. For continuous mode, we collected 41 commits of LLVM optimizer updates from the past year. For batch mode, we collected all branches in files implementing optimization passes from the latest version of LLVM. Specifically, we selected 7 optimization passes of `instcombine`, `instsimplify`, `reassociate`, `correlated-propagation`, `gvn`, `slp-vectorizer`, and `vectorcombine`. For example, we examined 16 error-prone files according to the history including `InstructionSimplify.cpp`, `InstCombineAndOrXor.cpp`, `InstCombineSelect.cpp`, `Reassociate.cpp`, `GVN.cpp`, `VectorCombine.cpp`, and `SLPVectorizer.cpp`. Overall, we specified 79 and 4,527 target locations for continuous mode and batch mode, respectively. We set the time limit to 1 hour for each target.

Table 4 shows the overall results. In continuous mode, Optimuzz discovered 7 miscompilation bugs. In batch mode, Optimuzz discovered 48 miscompilation. We reported all bugs to the LLVM developers and 13 of them have been patched as of the writing of this paper.

Fig. 13 illustrates an example of a miscompilation bug discovered from a recent commit in InstCombine. To test this commit, we targeted Line 12 as described in § 4.4, because the addition of

```
1  Instruction *foldSelectIntoOp(SelectInst &S, Value *T, Value *F) {
2      ...
3      if (isa<FPMathOperator>(&SI) &&
4  +       !computeKnownFPClass(FalseVal, FMF, fcNan, &SI).isKnownNeverNaN()) // Incomplete Update
5          return nullptr;
6      ...
7      NewSel->takeName(TVI);
8      BinaryOperator *BO = BinaryOperator::Create(TVI->getOpcode(), FalseVal, NewSel);
9      BO->copyIRFlags(TVI);
10 ++  if (isa<FPMathOperator>(&SI)) // Patch for a detected bug
11 ++      BO->andIRFlags(NewSel);
12     return BO; // Target Location
13 }
```

Fig. 13. Miscompilation bug in the latest LLVM and its patch

Line 4 changes the condition of the subsequent optimizations. The developer aimed to mitigate a miscompilation issue within this optimization but was unable to fully address all miscompiling cases. Within 1 hour, Optimuzz detected a new miscompilation bug in this optimization. We reported this bug to the LLVM developers, and they added a patch at Line 10. Note that this bug was not detected by Alive2 with 8 unit tests written by the developers at the time of the commit. On the other hand, Optimuzz successfully discovered such edge cases.

The overall results demonstrate that Optimuzz is effective at discovering miscompilation bugs in real-world compilers. Especially for LLVM, our results indicate that only 1 hour of fuzzing per location provides sufficient testing coverage. This capability to automatically generate unit tests for corner cases, which may be difficult to identify manually, offers a significant improvement in reliability for maintaining a frequently updated compiler.

6 Discussion

Optimuzz's effectiveness mainly relies on the underlying TV tools. For example, for LLVM, we use Alive2 which supports bounded loop unrolling and is specialized for intraprocedural optimizations. Thus, Optimuzz supports loop optimizations up to a configured loop unrolling bound and excludes interprocedural optimizations for LLVM.

Our guidance and mutation strategies are motivated by common code patterns implementing rewrite rules. Thus, Optimuzz is more effective for optimizations that are explicitly expressed as rewrite rules such as LLVM's InstCombine. However, we observe Optimuzz can also effectively detect other types of optimizations. For example, Optimuzz discovered unknown bugs in GVN and SLPVectorizer optimization passes in LLVM. Although such optimizations are not explicitly expressed as rewrite rules in the source code, they also share the common structure of compiler optimization passes. They check optimization conditions and apply transformations through nested conditional statements. We observed such patterns across various compilers such as TurboFan, GCC and recent deep-learning compilers. Thus, we conjecture that Optimuzz can be effective for a wide range of optimizations across different compilers.

Currently, our targeted mutations only support mutations that preserve the control flow of input programs. This enables us to design an effective mutation strategy based on the def-use relations of the input program. To support more general mutations, we need to develop a more general targeted mutation strategy. We leave this as future work.

7 Related Work

Our system effectively applies translation validation (TV) [58, 61] to check the correctness of compiler optimizations. TV ensures the correctness of a given compilation by checking the refinement relation between the source and target programs. Recently, with significantly increasing interest, several tools have been developed for real-world compilers such as Alive2 [57] for LLVM and

TurboTV [23] for TurboFan. However, TV tools can only check the correctness of the observed compilations. This in turn limits the effectiveness of these tools in proactively detecting bugs. Optimuzz addresses this limitation by employing directed fuzzing to generate programs that trigger the target optimization. We demonstrated the effectiveness of this approach on real-world compilers, uncovering many miscompilation bugs that were previously undetected by existing approaches.

Optimuzz provides an effective way to test compilers for miscompilation bugs. In recent years, there has been a large body of work on compiler testing for widely used languages such as C/C++ [12, 13, 26, 28, 30, 31, 66] or JavaScript [1, 18, 19, 25, 59, 60, 63]. Most of these approaches employ mutation-based fuzzing and observe erroneous behavior such as crashes. Since this may miss latent bugs such as miscompilations, some approaches use differential testing that compares outputs of programs compiled from different compilers [1, 63, 64]. However, it still requires specific inputs and UB-free programs to be effective. To overcome this, recent methods combine fuzzing with TV [13, 28], since TV can detect miscompilation bugs even for UB programs using refinement relations. However, such simple combinations may not be effective in large compilers. We empirically demonstrated that Optimuzz is more effective than existing approaches for detecting bugs in real-world compilers.

Recently researchers have proposed a new approach to test compilers using large language models (LLMs) [65]. They proposed an LLM agent that analyzes compiler source codes and generates input programs that trigger optimizations. While general and effective, LLMs are known to be computationally expensive. They require specialized computing devices (e.g., GPUs) or cloud services (e.g., GPT-4) to generate input programs. In contrast, Optimuzz provides a more efficient solution by using lightweight coverage feedback and targeted mutation.

Optimuzz is orthogonal to generation-based compiler testing tools [27, 30, 62, 66]. They generate new random programs from scratch or by using existing real-world programs. While these tools aim to trigger various behaviors across the entire compiler, Optimuzz focuses on validating specific target optimizations. Additionally, their generated programs can be integrated with Optimuzz to enrich the initial seed pool for directed fuzzing.

Optimuzz is the first approach to apply directed fuzzing to validate compiler optimizations. Directed fuzzing [2, 11, 20, 21] is an emerging technique that focuses on generating inputs that cover a specific target location in a program. This approach is particularly effective for testing large and complex programs with a small number of suspicious locations such as recently modified code or potential errors identified by static analysis. All previous directed fuzzing approaches handle inputs as sequences of bytes or simple formatted documents (e.g., XML). However, testing compiler optimizers requires the fuzzer to generate input programs with specific optimization patterns. Inspired by the success of directed fuzzing in other domains, we propose a novel directed fuzzing approach for compiler optimizers. Our experiments show that the proposed strategies significantly improve the effectiveness of fuzzing for compiler optimizations.

8 Conclusion

We proposed Optimuzz, a novel solution for continuously validating the correctness of compilers. We designed strategies—*guided search strategy* and *targeted mutation strategy*—that enable the fuzzer to effectively generate input programs for testing target optimizations. We applied these ideas to LLVM and TurboFan, and demonstrated that Optimuzz could reproduce 23 more bugs more efficiently than FLUX and Alive-Mutate, and detect four additional bugs faster than Fuzzilli. Additionally, our experiments on the latest version of LLVM revealed 55 new miscompilation bugs. These results highlight Optimuzz's effectiveness in continuously validating compiler optimizations and ensuring the detection of latent miscompilation bugs. We anticipate that this approach will effectively validate the correctness of continuously updated compiler optimizations and provide a practical solution for maintaining reliability in modern compilers.

Data-Availability Statement

Optimuzz is available as an archived version on Zenodo [22]. This provides comprehensive instructions and scripts for reproducibility of experiments reported in the paper. The initial seed programs for reproducing the experiments are also included. Furthermore, a recently updated version of Optimuzz is open-sourced on our website (https://prosys.kaist.ac.kr/optimuzz). We provide Optimuzz toolchain compatible with the latest version of LLVM to support reusability.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (RS-2021-NR060080), and an Amazon Research Award Fall 2023. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of Amazon.

References

[1] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. 2022. Jit-Picking: Differential Fuzzing of JavaScript Engines. In *ACM Conference on Computer and Communications Security (CCS)*.
[2] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[3] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-Based Greybox Fuzzing as Markov Chain. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[4] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a Desired Directed Grey-Box Fuzzer. In *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
[5] Chromium. 2021. Issue 1195650. https://bugs.chromium.org/p/chromium/issues/detail?id=1195650
[6] Chromium. 2021. Issue 1198705. https://bugs.chromium.org/p/chromium/issues/detail?id=1198705
[7] Chromium. 2021. Issue 1199345. https://bugs.chromium.org/p/chromium/issues/detail?id=1199345
[8] Chromium. 2021. Issue 1200490. https://bugs.chromium.org/p/chromium/issues/detail?id=1200490
[9] Chromium. 2021. Issue 1234764. https://bugs.chromium.org/p/chromium/issues/detail?id=1234764
[10] Chromium. 2021. Issue 1234770. https://bugs.chromium.org/p/chromium/issues/detail?id=1234770
[11] Zhengjie Du, Yuekang Li, Yang Liu, and Bing Mao. 2022. Windranger: A Directed Greybox Fuzzer Driven by Deviation Basic Blocks. In *Proceedings of the International Conference on Software Engineering (ICSE)*.
[12] Karine Even-Mendoza, Arindam Sharma, Alastair F. Donaldson, and Cristian Cadar. 2023. GrayC: Greybox Fuzzing of Compilers and Analysers for C. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.
[13] Yuyou Fan and John Regehr. 2024. High-Throughput, Formal-Methods-Assisted Fuzzing for LLVM. *Proceedings of the International Symposium on Code Generation and Optimization (CGO)* (2024).
[14] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. 2020. WEIZZ: Automatic Grey-Box Fuzzing for Structured Binary Formats. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*.
[15] Google. 2008. Mjsunit. https://chromium.googlesource.com/v8/v8/+/master/test/mjsunit/
[16] Google. 2008. V8. https://v8.dev
[17] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. 2023. FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities. In *Network and Distributed System Security Symposium (NDSS)*.
[18] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *Network and Distributed System Security Symposium (NDSS)*.
[19] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the USENIX Security Symposium (Security)*.
[20] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2022. Beacon: Directed Grey-Box Fuzzing with Provable Path Pruning. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*.
[21] Tae Eun Kim, Jaeseung Choi, Kihong Heo, and Sang Kil Cha. 2023. DAFL: Directed Grey-Box Fuzzing Guided by Data Dependency. In *Proceedings of the USENIX Security Symposium (Security)*.
[22] Jaeseong Kwon, Bongjun Jang, Juneyoung Lee, and Kihong Heo. 2025. Optimization-Directed Compiler Fuzzing for Continuous Translation Validation. (2025). https://doi.org/10.5281/zenodo.15037303
[23] 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 International Conference on Software Engineering (ICSE)*.

[24] C. Lattner and V. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization*.
[25] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. 2020. Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer. In *Proceedings of the USENIX Security Symposium (Security)*, Srdjan Capkun and Franziska Roesner (Eds.).
[26] Shaohua Li and Zhendong Su. 2024. UBFuzz: Finding Bugs in Sanitizer Implementations. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
[27] Shaohua Li, Theodoros Theodoridis, and Zhendong Su. 2024. Boosting Compiler Testing by Injecting Real-World Code. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*.
[28] Eric Liu, Shengjie Xu, and David Lie. 2023. FLUX: Finding Bugs with LLVM IR Based Unit Test Crossovers. In *International Conference on Automated Software Engineering (ASE)*.
[29] Liu, Eric. 2024. FLUX Unit Test Suite. https://github.com/ericliuu/flux/blob/main/scripts/collect_unittest_functions.py
[30] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random Testing for C and C++ Compilers with YARPGen. *Proceedings of the ACM on Programming Languages* (2020).
[31] 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).
[32] LLVM. 2016. LLVM InstCombine Unit Test Suite. https://github.com/llvm/llvm-project/tree/main/llvm/test/Transforms/InstCombine
[33] LLVM. 2022. Issue 55291. https://github.com/llvm/llvm-project/issues/55291
[34] LLVM. 2022. Issue 57357. https://github.com/llvm/llvm-project/issues/57357
[35] LLVM. 2022. Issue 57683. https://github.com/llvm/llvm-project/issues/57683
[36] LLVM. 2022. Issue 58977. https://github.com/llvm/llvm-project/issues/58977
[37] LLVM. 2022. Issue 59279. https://github.com/llvm/llvm-project/issues/59279
[38] LLVM. 2022. Issue 59301. https://github.com/llvm/llvm-project/issues/59301
[39] LLVM. 2023. Issue 59836. https://github.com/llvm/llvm-project/issues/59301
[40] LLVM. 2023. Issue 61312. https://github.com/llvm/llvm-project/issues/61312
[41] LLVM. 2023. Issue 62401. https://github.com/llvm/llvm-project/issues/62401
[42] LLVM. 2023. Issue 62901. https://github.com/llvm/llvm-project/issues/62901
[43] LLVM. 2023. Issue 63327. https://github.com/llvm/llvm-project/issues/63327
[44] LLVM. 2023. Issue 64339. https://github.com/llvm/llvm-project/issues/64339
[45] LLVM. 2023. Issue 70470. https://github.com/llvm/llvm-project/issues/70470
[46] LLVM. 2023. Issue 72911. https://github.com/llvm/llvm-project/issues/72911
[47] LLVM. 2023. Issue 74890. https://github.com/llvm/llvm-project/issues/74890
[48] LLVM. 2023. Issue 75437. https://github.com/llvm/llvm-project/issues/75437
[49] LLVM. 2023. Issue 76441. https://github.com/llvm/llvm-project/issues/76441
[50] LLVM. 2024. Issue 84025. https://github.com/llvm/llvm-project/issues/84025
[51] LLVM. 2024. Issue 89390. https://github.com/llvm/llvm-project/issues/89390
[52] LLVM. 2024. Issue 89516. https://github.com/llvm/llvm-project/issues/89519
[53] LLVM. 2024. Issue 89669. https://github.com/llvm/llvm-project/issues/89669
[54] LLVM. 2024. Issue 91417. https://github.com/llvm/llvm-project/issues/91417
[55] LLVM. 2024. Issue 98753. https://github.com/llvm/llvm-project/issues/98753
[56] LLVM. 2024. Issue 98838. https://github.com/llvm/llvm-project/issues/98838
[57] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: Bounded Translation Validation for LLVM. In *ACM SIGPLAN International Conference on ProgrammingLanguage Design and Implementation (PLDI)*.
[58] George C. Necula. 2000. Translation Validation for an Optimizing Compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
[59] Jihyeok Park, Seungmin An, Dongjun Youn, Gyeongwon Kim, and Sukyoung Ryu. 2021. JEST: N+1-Version Differential Testing of Both JavaScript Engines and Specification *(Proceedings of the 43rd International Conference on Software Engineering (ICSE))*.
[60] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. 2020. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *IEEE Symposium on Security and Privacy (SP)*. IEEE.
[61] A. Pnueli, M. Siegel, and E. Singerman. 1998. Translation Validation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.
[62] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding Compiler Bugs via Live Code Mutation. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[63] Junjie Wang, Xiaoning Du Zhiyi Zhang, Shuang Liu, and Junjie Chen. 2023. FuzzJIT: Oracle-enhanced Fuzzing for JavaScript Engine JIT Compiler. In *USENIX Security Symposium (Security)*.
[64] Mingyuan Wu, Minghai Lu, Heming Cui, Junjie Chen, Yuqun Zhang, and Lingming Zhang. 2023. JITfuzz: Coverage-guided Fuzzing for JVM Just-in-Time Compilers. In *Proceedings of the International Conference on Software Engineering (ICSE)*.
[65] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. 2024. WhiteFox: White-Box Compiler Fuzzing Empowered by Large Language Models. *Proceedings of the ACM SIGPLAN Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2024).
[66] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *ACM SIGPLAN International Conference on ProgrammingLanguage Design and Implementation (PLDI)*.
[67] A. Zeller and R. Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering* (2002).
[68] Zhide Zhou, Zhilei Ren, Guojun Gao, and He Jiang. 2021. An Empirical Study of Optimization Bugs in GCC and LLVM. *Journal of Systems and Software* 174 (2021).

Received 2024-11-15; accepted 2025-03-06