

Differential Testing SSA Optimization Passes in Go Compilers Using LLM Generated Code Mutators

Luiz Terres

Universidade Federal da Fronteira Sul
Chapecó, Brazil
luiz.terres@estudante.uffs.edu.br

Denio Duarte

Universidade Federal da Fronteira Sul
Chapecó, Brazil
duarte@uffs.edu.br

Andrei Braga

Universidade Federal da Fronteira Sul
Chapecó, Brazil
andrei.braga@uffs.edu.br

Samuel Feitosa

Universidade Federal da Fronteira Sul
Chapecó, Brazil
samuel.feitosa@uffs.edu.br

Abstract

Compilers are fundamental tools for software infrastructure, yet their complex implementation makes them prone to bugs and hard to formally verify. The literature presents a range of techniques that aim at random testing (fuzzing) compilers for maintaining acceptable levels of correctness. Among traditional tools for compiler fuzzing, the complex implementation and maintenance of code generators and mutators remain an open challenge. Recent efforts aim to address this challenge by employing Large Language Models (LLMs) for test case generation. This paper presents Goopfuzz, a fuzzer that leverages Gemini 3.1 Pro and targets SSA passes in Go compilers in a differential testing, mutation-based, and coverage-guided approach. By leveraging the Gemini model to generate code mutators and removing the LLM from the fuzzing loop, the Goopfuzz hybrid approach mitigates the complex implementation challenge associated with traditional methods and the low throughput of LLM-pure solutions. During Goopfuzz experiments, Gemini 3.1 Pro synthesized 76.25% of all 157 working mutators on the first try. Synthesized mutators produced a total of 165,321 unique Go programs to serve as test cases during the fuzzing campaign, exercising 15.75% of the SSA implementation portion of the Go compiler. The differential testing oracle evaluated the compilation results for those programs against the last two stable releases of the official Go compiler (1.25.10 and 1.26.3) in different optimization levels.

Keywords

Compiler, Differential, Testing, Fuzzing, LLM, Code, Mutation, Coverage, Go, SSA, Optimization

1 Introduction

The first goals for what would come to be known as the Go programming language were conceived by Robert Griesemer, Rob Pike and Ken Thompson at Google on September 21, 2007 [15]. Its design aims to offer a good balance between performance and syntax simplicity. The following years molded ideas and prototype compilers for the Go lang, until on November 10, 2009, it became a publicly open-source project [15]. In 2026, Go is consolidated on the software market, offering native concurrency and garbage collection as a compiled programming language.

A compiler for a programming language is expected to map the source program into another language, keeping all semantic

qualities from the original program. Besides efforts to formally verify this class of computer programs [5], modern optimizing compilers are often considered too complex to be fully verified and depend on dynamic testing (fuzzing) for assuring software quality. Additionally to the fact that software developers rely on the proper working of compilers, their failures can expose security exploits. In his Turing Award lecture, Ken Thompson reinforces the central role of compilers in software development and security concerns associated with it by discussing how malicious code injected into a compiler implementation compromises security on applications compiled with it [18]. In a comparatively recent example, Bauer et al. exploit a naturally occurring bug in a production compiler by modifying a Linux utility to trigger it, allowing a user to gain privileges inside the machine [1].

In view of its importance, many compiler testing (fuzzing) approaches have been proposed to address the need to ensure proper compiler operation and security. Fuzzing efforts in the literature differentiate themselves by the technique they employ on the generation of test cases and the assessment of potential failures [6]. Traditionally, fuzzing techniques rely on grammar-based program generators or mutators for generating valid and diverse compiler test suites that spot miscompilations or compiler crashes. Those tools pose a barrier to non-expert users for their complex implementation and complex needed maintenance as the programming language evolves. Recent efforts indicate a special attention to the employment of Large Language Models (LLMs) into the testing process [4, 12, 19]. Many LLM-pure solutions successfully employ LLMs as test case generators, addressing challenges associated with the complexity of crafting those grammar-based programs, at the cost of lower throughput caused by the computational overhead of those models.

This work presents an approach called Goopfuzz, which proposes a solution that circumvents challenges associated with both the complexity of crafting program mutators and the high computational overhead of LLMs, using a hybrid approach. By leveraging the LLM as a synthesizer of mutator programs, the fuzzing loop of Goopfuzz occurs as in a traditional non-LLM solution, without throughput reduction. Technically, this work presents a solution that leverages code generation and understanding capabilities of the LLM Gemini 3.1 Pro to synthesize Go AST mutators to trigger the Static Single Assignment (SSA) portion of the Go compiler, given the Go compiler SSA passes implementation. The rationale

for targeting SSA passes is motivated by their highly structured representation. The mapping from Go language constructs into SSA patterns is well-defined. This regularity could help the LLM to recognize these patterns and synthesize code mutators that increase the complexity of regions in the seed programs that exercise the later SSA optimization passes on those mappings.

The LLM-synthesized code mutators are applied to Go program seeds to generate a Go compiler test suite targeting SSA optimization passes. The generated test suite undergoes code coverage metric capture to assess their capability of targeting expected portions of the compiler. The Goofuzz differential testing oracle approach process the generated mutants using different stable versions of the official Go compiler under different optimization levels, watching for variable results. This work implements Goofuzz and runs a 15-batch experiment that lasted for 13 hours and 20 minutes, collecting results. The authors manually analyze the experimental results to answer the following research questions:

- RQ1:** Are LLMs capable of synthesizing code mutators?
- RQ2:** Are LLM-synthesized AST code mutators a viable choice for compiler fuzzing based on mutation operations?
- RQ3:** Is Goofuzz effective in exercising code paths from the SSA passes in the Go compiler?
- RQ4:** Does Goofuzz discover divergent outputs from generated mutants?

The following sections of this paper provide an overview of a theoretical background on the Go compiler and compiler testing tasks (Section 2). Discusses recent efforts in the literature on the direction of the employment of LLMs for testing compilers (Section 3). Presents the Goofuzz method (Section 4) and experimental setup (Section 5). Answer the proposed research questions by analyzing the Goofuzz experimental results (Section 6). And finally, it concludes the paper (Section 7).

2 Background

2.1 The Go Compiler

The compilation process for the official Go compiler can be logically separated into parsing, type checking and intermediary representation (IR) construction, middle-end optimizations, and machine-code generation [16]. The parser outputs an Abstract Syntax Tree (AST) that represents the original code. After validation passes on the AST, the compiler generates a node representation from the AST and performs machine-independent optimizations over it. This IR is then mapped into Static Single Assignment (SSA) form, where the compiler applies advanced optimization passes. The machine code generation phase involves lowering the abstraction from the machine-independent SSA form and applying optimizations to architecture-specific forms before generating the executable binary.

Static Single Assignment is an IR form where each variable is assigned exactly once. For the Go compiler, the SSA form has Values, Blocks and Functions. A variable in Go code translates into a unique SSA Value. This Value is a set of information that defines the state of the variable in a given time. If that variable in Go code is reassigned (its content changes), this new state of the variable translates into a new unique SSA Value. The old and the new content of the code variable were translated to two different SSA Values. Basically every operation on the Go code translates into values, unless it's

a block or a function. An if block translates into an SSA Block which represents the flow of the code. An SSA Block explicitly defines the flow of the program using successor Blocks and to which Value each Block should return after it finishes. Code functions translate into SSA Functions which contain operations (SSA Values) and Blocks. The conceptually simple form of SSA is frequently employed by compilers as a robust representation of code, useful to explicitly show the dataflow of the program and to run optimization algorithms on. In Go, since version 1.7, the compiler features an SSA-based backend¹, applying optimization passes over the SSA representation of input code to generate optimized output binaries [17].

2.2 Compiler Testing

Besides compilers being programs that software developers rarely doubt on, literature shows they are not free from bugs that should not be ignored [1, 7, 13]. Those failures can cause problems for their main user (developers) who rely on a proper interaction with the compiler, as well as to downstream software users (casual computer users) from using programs built with compilers containing security threats. Most intuitive compiler failures include an incorrect mapping from the source code (miscompilation), unexpected termination of the compiler with no generated executable output (crash), or incorrect application of compiler options (flags). Subtler forms of failure include logical manipulation that exposes security threats (e.g., [18]).

For the infinite input space and complex implementation of modern compilers, many approaches for compiler testing rely on dynamic testing (also addressed as fuzzing). Dynamic testing is essentially disposing of the system under test with inputs it is expected to handle without compromising the system's integrity. Compilers accept as inputs programming language programs and options (flags) that alter the compilation process. This dual input nature of compilers allows for two different approaches for testing. Option fuzzing applied to compilers focuses on testing the harness of the compilation process by stressing different compilation options (flags) given the same code file input [20]. The other and more traditional approach, input fuzzing applied to compilers, consists of feeding random programs to the compiler implementation and depending on a test oracle to spot inconsistencies. This process is related to known challenges addressed by previous literature [2, 14] that will be discussed in the following subsections.

2.3 Test Suite Generation for Compilers

The generation of test cases is at the core of any dynamic testing effort. The quality of input determines the success in finding errors in the system under test. Applying completely random mutation on programs, although close to the initial rationale for fuzz testing [9], is considered not optimal for compiler testing, for programs that do not follow strict syntax rules often fall short in reaching deep components of a modern compiler [8]. Among the challenges associated with the generation of the compiler test suite are:

- **Validity:** Valid programs are vital for tackling deeper compiler components, since malformed inputs would throw an

¹Go 1.7 release notes: <https://go.dev/doc/go1.7>

error already on the compiler’s frontend. However, constructing valid programs is non-trivial due to language restrictions.

- **Undefined behavior:** A test program containing behavior undefined by the language specification undermines the correctness assessment of the output.
- **Diversity:** Increasing diversity of language constructs in test programs augments the tested portion of the compiler. However, the increase in the diversity of generated programs is directly related to the increase of the complexity of the program generator itself.

Traditionally, test suite generation strategies can be divided into generation and mutation. Generation strategies rely on the grammar of the programming language for constructing valid programs from scratch. Using a grammar to rule or to aid the process of generating a test seed guarantees or augments the generation of syntactically valid programs that potentially reach deeper components during compilation. Mutation strategies require an initial pool of seed containing valid programs in which mutators apply modification operations in semantic-preserving or non-preserving approaches, aiming to expand the diversity of the corpus. While those techniques aim to address the challenges of generating valid, undefined behavior-free, and diverse code, they also present associated challenges themselves. The complexity of constructing program generators and mutators is widely acknowledged by the literature [2, 3, 11].

The diversity challenge of a test suite is particularly commonly addressed by the literature via instrumenting the compiler to capture code coverage statistics. This compiler instrumentation allows capturing which blocks of the compiler execute for a given input program. It allows for a fine-grained feedback of what compiler internals are being tested. Most approaches that leverage this internal compiler information often involve prioritizing some inputs over the others. The rationale behind prioritizing test cases by their result in the code coverage metric is straightforward. It aims to test the most lines of code of the compiler with the least number of test cases and achieve this by rewarding programs that explore new portions of the compiler.

2.4 Test Oracles for Compiler Testing

Another key step of the compiler fuzzing process is defining a test oracle. This oracle is concerned with identifying when a bug has occurred, which is precisely the main challenge associated with this step. If random programs are the input, the expected output is also unknown. A basic form of test oracle is executing the compiler, watching for crashes or hangs. However, this oracle falls short in identifying more subtle compiler failures as miscompilations. Differential testing is among the most popular approaches for test oracles [8]. It consists of comparing the output of two or more comparable systems. If any of the outputs present inconsistent behavior, one of them is assumed to have a bug. Among the strategies adopted for differential testing in compiler testing are:

- **Cross-Compiler:** Compares the execution results from different compilers given the same input program. Requires that the programming language be supported by at least two distinct compiler implementations.

- **Cross-Optimization:** Compares the execution results from the same compiler under different optimization levels. Only possible if the compiler offers options to do so (often via flags).
- **Cross-Version:** Compares the execution results from the same compiler under different implementation versions. Requires more than one version of the same compiler.

The differential testing approach provides a partial solution to the test oracle problem by highlighting varying results. Once the test oracle flags a test case as a potential bug trigger, the test case often needs to be further refined. Post-processing results of a compiler fuzzing campaign is another important step for dealing with the noisy results associated with random compiler testing. The literature brings test case reduction and duplicate identification approaches to sanitize bug-triggering programs before issuing compiler developers.

3 Related Work

As the development of software testing using Large Language Models (LLMs) evolved, solutions employing those models for testing compilers also arose [19]. The rationale behind applying LLMs for compiler testing is straightforward. As models capable of synthesizing code with valid syntax and multiple language constructions, they are natural test program generators. Their ease of use addresses challenges associated with implementing traditional code generators as discussed in the previous section. Fuzz4all [21] is one LLM-pure solution that employs LLMs directly into the test case generation process inside the fuzzing loop. Its Autoprompt technique makes the solution language-agnostic, provided sufficient context is available for the language model. In bug-finding experiments, Fuzz4all compares above-baseline testing tools among multiple systems that take formal language as inputs, beyond just compilers.

Similar techniques have a narrower scope and focus their testing effort on compiler components. Whitefox [22] is one LLM-pure solution that targets testing optimization operations on deep learning compilers. It employs the LLMs for both code understanding and test program generation, also addressing the lack of guidance of black- and grey-box methods for fuzzing. From the source code of deep learning optimizations, an LLM generates requirements to trigger optimizations, and another LLM generates test cases based on the requirements. Besides being primarily focused on deep learning compilers, Whitefox’s solution can be expanded to programming language compilers, and the authors provide a successful generality case study on LLVM compilers. Apart from successfully finding compiler failures, those techniques are associated with exhibiting lower throughput on the test program generation when compared to non-LLM for the computational overhead of those models [10, 12, 21].

A trendline of works addresses the high computational cost of LLM-pure compiler fuzzing solutions by removing the language model from the test generation task of the fuzzing loop. Those solutions still address the challenges associated with the complexity of traditional code generators by applying LLMs indirectly into the test case generator task. Among them, Metamut [11] presents a

solution that employs an LLM to synthesize code mutators for compiler testing with help and feedback from expert users. It proposes a framework that systematically implements code mutators that modify seeds via AST manipulation to serve as test case generators on the fuzzing loop. Another example is Legofuzz [10] which employs the language models in a first phase to generate functions as standalone pieces. A non-LLM technique glues those functions together on the generation loop to create complex test cases for fuzzing.

Goopfuzz is conceptually situated between Whitefox and Metamut solutions. While Metamut synthesizes code mutators that manipulate the AST of input programs using LLMs with help from expert users, and Whitefox employs the code understanding and code generation capabilities of those models to generate test cases targeting specific paths of the compiler’s implementation, Goopfuzz leverages their core ideas together. Goopfuzz employs the LLM to understand code portions of the Go compiler and to generate specific code mutators that modify the AST of input programs targeting to trigger SSA optimization passes on the Go compiler, allowing to remove the LLM from the fuzzing campaign. Goopfuzz provides a more targeted and non-assisted solution when compared to Metamut and aims to address the throughput constraints by the computational overhead from the LLM-on-loop of the Whitefox solution.

4 Methodology

Goopfuzz proposes a method for testing Go compiler addressing challenges from both traditional fuzzing and LLM-based approaches. For that matter, its methodology is divided into synthesis and execution phases. The synthesis phase synthesizes code mutators and seed Go programs via a non-expert user interaction with an LLM. The generation of these artifacts is essential for the posterior execution phase, in which the fuzzing campaign occurs.

4.1 Synthesis Phase

Figure 1 overviews the workflow of an iteration of the synthesis phase, whose purpose is to synthesize mutators and seeds. For this phase, the user provides the Large Language Model (LLM) with the implementation for each SSA optimization of the Go compiler. The user then prompts the model to generate descriptions of mutation operations and function seeds that trigger those SSA passes. As a result of this first interaction, the user will get two partial artifacts for each SSA optimization pass, a Function Seed ② that contains specific language constructs which triggers the pass and a list of Mutation Descriptions ① for that pass.

Those generated partial artifacts (①, ②) serve as context for a new interaction between user and model to synthesize the final artifacts (working code mutators and seeds). This second interaction produces actual Mutators ③ implemented in the Go language that take in go programs, modify their structure via AST manipulation and outputs a mutant. A mutant is precisely a Go program based on the input Go program (also called the mutant’s seed) with mutation operations applied. A third user-LLM interaction happens to generate an actual Seed Program ④ from the seed function ②. Given the seed function, the LLM is prompted to generate calls

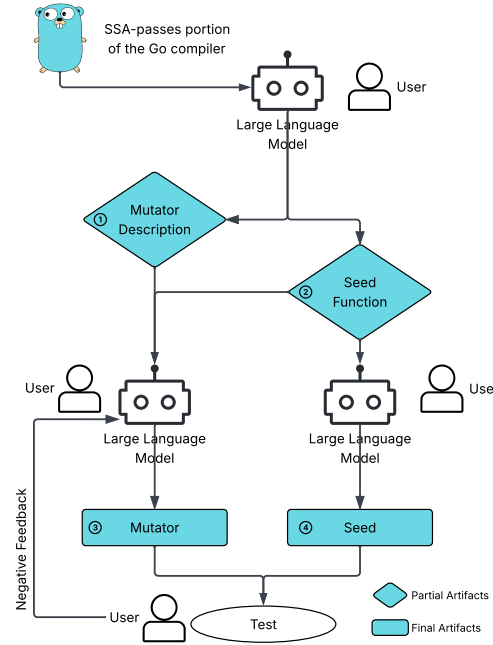


Figure 1: Synthesis phase of Goopfuzz.

for that function inside a main block and adjust imports, function arguments, and return values so that the final seed compiles.

With mutators and seeds synthesized, a final step of evaluation of those code mutators is performed to maintain a certain level of correctness on their implementation. In a simple testing framework, each code mutator is applied to its respective seed in order to produce a mutant. A mutator is considered a bad mutator if it does not compile, if it breaks during the mutation operation, or if the mutant generated does not compile (e.g., syntax error). In either bad mutator scenario, a negative feedback is given to the LLM containing the mutator and error description. The model is then prompted to generate a more robust version of the mutator, which is tested again and re-prompted if necessary.

4.2 Execution Phase

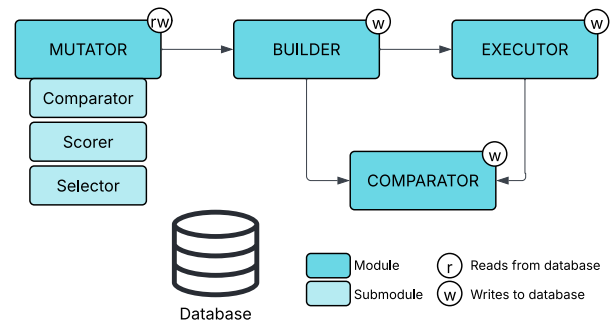


Figure 2: Execution phase of Goopfuzz.

The execution phase of the Goopfuzz methodology consists of a differential fuzzing campaign loop of the Go compiler leveraging the final artifacts from the synthesis phase. The core idea of the execution phase is to continuously apply mutation operations on seeds, producing candidate test cases to stress the SSA optimization passes portion of the Go compiler. Our proposed architecture consists of modules that work together as described by Figure 2.

The mutator module holds the code mutators generated by the execution phase. It reads the seeds from the database to apply AST manipulations via those code mutators. This is the only module that depends on submodules. The Comparator submodule takes as input the mutant and the seed from which it was generated and compares the contents on their AST. Mutants that present differences on AST when compared to their seed’s AST are considered Unique (otherwise they are considered Duplicated). Since our code mutators are limited in the language constructs they modify in the AST of a seed, there are seeds in which certain mutators will never produce any change, generating a duplicate. Mutants that differ from their seed and do not cause errors during mutation operation are scored by the Scorer submodule. Scoring consists of capturing the code coverage metric for the mutant. As previously discussed in the Background Section (2.2), seed prioritization plays an important role in the effectiveness of mutation-based fuzzing. To prioritize which seed to mutate, Goopfuzz doesn’t apply greedy code coverage guidance. This strategy is further discussed in a later section that discusses the implementation of Goopfuzz (5.4). All information produced by the mutant module is persisted to a database using the Selector submodule.

All unique mutants are processed using the remaining modules. The Builder module compiles the mutant across different versions and optimization options (e.g., disable optimization flag) of the Go compiler. It produces and stores artifacts of compilation errors or warnings and successful executable binaries. Successful compilations have their binaries executed by the Executor module, which captures and stores outputs and errors. The Comparator module is the last module that compares compilation artifacts among different versions of the compiler for the same mutant and execution artifacts. The post-processing of the results for the analysis of the potential bug-triggering programs are performed manually by the authors.

5 Experimental Setup

The experiment for Goopfuzz includes the synthesis and execution phase of the proposed methodology. The execution phase was set to generate 15 batches of programs. This section presents the technical setup for the experiment.

5.1 Generation of Code Mutators and Seeds

All code mutators and seeds were generated via interaction of the authors with the online interface of Google’s LLM Gemini version 3.1 Pro. The Go compiler source code files were extracted from the list of SSA passes available on the GitHub mirror of the Go project². These Go source files were passed as context for the LLM on the generation of the partial artifacts as stated in the methodology for

this work. In all LLM-authors interactions, the prompt instructions were trivial orders (e.g., synthesize a code mutator for this mutator description). Mutator corrections contained as context were the mutator’s name and its output error, followed by a trivial order to correct the mutator. The final mutators act as standalone programs for each SSA pass that take in Go programs and output mutants. All mutators were compiled using the last stable release of the Go compiler (Go 1.26.3).

5.2 Environment

The execution phase of Goopfuzz was conducted on an Intel i7-10700K machine with 24GB RAM and a SATA III SSD running Fedora Linux 43.

5.3 Implementation of the Fuzz Driver

Goopfuzz was implemented in python. Each module from Figure 2 run as a separated process and read and write data from and to a main SQLite database. The generation of mutants happens in a generational-like strategy, called batches. In the beginning of a batch, the Mutator module reads a set of interesting seeds (Subsection 5.4) from the database and runs all mutators on all seeds. For each seed, a mutant is generated, built by 4 different compilers (Subsection 5.5), executed, and has its builds and executions outputs compared. All operations have an execution timeout of 60 seconds to avoid loop hangs. The experiment was set to run for 15 batches.

5.4 Seed Prioritization

A Go compiler instrumented for code coverage captures blocks of code exercised by each non-duplicated mutant under the directory that applies SSA optimization passes on the compiler implementation. The Mutator module keeps track of the global set of blocks all mutants execute. Mutants that execute new blocks are considered interesting to serve as seeds for posterior batches. The seeds for the first batch are all LLM-generated seeds. Those seeds are called Adams, for they are crafted by the LLM, each for a specific SSA pass. The pool of seeds for the next batch is composed by all the mutants from the last batch that explore new coverage, and a compilable mutant descendant from each Adam. The rationale behind the Adam is to keep at least one representative for each SSA pass across all batches and avoid an SSA pass going underexplored if none of its representatives generate new coverage on a single batch.

5.5 Go Compilers

The last two stable releases³ to the date of this work of the official Go compiler were targeted on the experiment (Go 1.26.3 and Go 1.25.10). For the cross-optimization differential testing, the flags `-gcflags=all=-N -l` (disabling optimizations) were set to both compilers, totaling 4 different compilers under test.

6 Results

This section analyzes the results from the 15-batch experiment run of the execution phase of Goopfuzz. This operation took 13 hours and 20 minutes.

²Go SSA passes: <https://github.com/golang/go/blob/master/src/cmd/compile/internal/ssa/compile.go>

³All Go releases: <https://go.dev/dl/>

6.1 RQ1: LLM-Generated Mutators

Table 1: Generation of Mutators for Passes.

	SSA Passes	Code Mutators		
Working	26	122		
Discarded	1	3		
Failing	20	BDE*	GBM**	MNC***
		12	20	3
Total	47	160		
Final Corpus	46	157		

* BDE: Broke During Execution

** GBM: Generated Broken Mutant

*** MNC: Mutator did Not Compile

Table 1 summarizes the result of generating Go AST manipulators code mutators using Gemini 3.1 Pro. Out of 47 SSA passes extracted from the Go compiler implementation, 122 mutators (76.25%) for 26 passes (55.32%) succeed in the preliminary test against their own crafted seed on the first try. This high percentage of valid mutators suggests a positive outcome for the LLM as mutator synthesizers. For 1 SSA pass (2.13%), the model concluded the mutation in code from the mutator description could never tackle the expected portion of the compiler due to a frontend limitation. Gemini suggested adjusting the compiler frontend to bypass this limitation, but it would break the test template for the Goopfuzz solution, so it was discarded. The remaining 20 SSA passes (42.55%) presented at least one failing mutator, totaling 35 code mutators (21.88%) failing against their own crafted seed on the first try. However, only 2 out of these 35 failing mutators needed 2 negative feedbacks, while 33 mutators met the preliminary requirements with 1 error feedback. A total of 83 prompts were needed to generate 157 working code mutators (46 synthesis prompts and 37 negative feedback prompts) without technical intervention from the user on their implementation, relying solely on the language model’s understanding of the implementation and feedback messages.

6.2 RQ2: Generated Mutants

To assess the viability of using large language models to synthesize code mutators, the authors analyze the mutants they produce. Figure 3 shows the number of generated unique mutants, duplicated mutants, and buildable unique mutants relative to one another across batches 1 to 15. The graph suggests a positive outcome for the mutators. Across all batches, the number of compilable mutants kept close to or higher relative to the duplicated mutants, demonstrating the capability of the mutators of augmenting seed complexity without breaking language syntax. The whole experiment generated 165,321 mutants, with 97,179 (58.78%) of them unique mutants and 68,142 (41.22%) duplicated. Duplicated mutants were automatically discarded during the fuzzing campaign since the build and execution results for them were already captured. Out of the unique mutants, 80,286 of them (82.62%) successfully compiled, generating an executable binary. No code mutator broke during

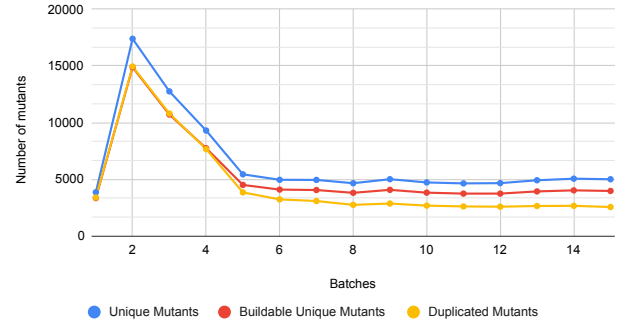


Figure 3: Generated mutants per batch.

the execution of 165,321 mutation operations. All mutation operations outputted either a modified unique mutant or an unmodified duplicated mutant, which suggests a robust implementation and edge case handling by the LLM when dealing with the program’s AST. Figure 4 showcases an example of code mutation performed by the code mutator “InjectExplicitIndexOffset” that targets the “AddressingModes” SSA pass and introduces a redundant index transformation to the original code. Lines in green were added by the mutator into the mutant, while lines in red were removed from the seed.

```

for i := 0; i < len(data); i++ {
    idxMut_1 := i + 4
    sum += data[i]
    sum += data[idxMut_1-4]
}

```

Figure 4: Mutation example

6.3 RQ3: SSA Passes Coverage

The metric adopted to measure the effectiveness of Goopfuzz in generating mutants that exercise the compiler implementation is the code coverage capture. The Go compiler supports instrumentation to capture which blocks under the SSA portion of the compiler are executed, which allows for refined coverage feedback since we are targeting SSA passes. Table 2 presents for each batch the cumulative sum of blocks executed up until the batch, the number of newly executed blocks during the batch, and the percentage of Total Blocks considering the entire SSA implementation. From the initial 11.63% portion of the compiler executed by LLM-generated seeds, the fuzzing loop via mutations expanded in +4.11% of the explored blocks in 15 batches. Besides some batches presenting modest new discoveries, they didn’t undermine the possibility of posterior discovery of new execution paths. It suggests the potential of the Goopfuzz to keep finding new coverage in seeds from beyond the limited 15 batches of the experiment.

Table 2: Block coverage metrics per batch

Batch	Total Blocks	New Blocks	Total Percentage
0	12276	*	11.63%
1	14821	+ 2545	14.04%
2	16166	+ 1345	15.31%
3	16402	+ 236	15.53%
4	16454	+ 52	15.58%
5	16465	+ 11	15.59%
6	16475	+ 10	15.60%
7	16497	+ 22	15.62%
8	16522	+ 25	15.65%
9	16568	+ 46	15.69%
10	16569	+ 1	15.69%
11	16569	+ 0	15.69%
12	16578	+ 9	15.70%
13	16584	+ 6	15.71%
14	16589	+ 5	15.71%
15	16616	+ 27	15.74%

This table presents coverage data exercised by the mutants during the experiment for blocks of code under the SSA portion of the Go compiler implementation.

6.4 RQ4: Variable Outputs

Numerous inconsistencies were spotted by the differential testing oracle for the stable versions of the Go compiler under test. The analysis (post-processing) for bug finding was conducted manually by the authors from the results of the differential testing oracle. The differential testing oracle spotted a total of 5725 differing results among 12.38% build errors, 1.62% execution outputs, and 85.99% execution errors:

709 differing build errors were flagged due to what appears to be an improvement in the diagnostics of the newer version of the Go compiler for operations with mismatched types. Figure 5 shows the difference in feedback message between versions 1.25 and 1.26 for the same bad usage of types inside the mutant caused. Mismatched type error appears across batches 1 to 15 of the experiment, in variants from 44 different Adams.

93 differing execution outputs caused by reading garbage from memory when using unsafe pointer arithmetic. During the batches 13, 14, and 15, a total of 93 variants of the same Adam spanned presenting a line of code that reads data from garbage, stores it, and prints it.

4923 differing execution errors presenting expected runtime panics and errors. Those errors occur across all 1 to 15 batches, caused by variants of all Adams. Among the errors here are bad interface conversions (1988), induced panics (1373), index out of range operations (1133), bounds out of range operations (284), invalid memory address or nil pointer dereference (125), and stack overflows (20).

7 Conclusion and Future Work

This paper presented Goopfuzz, an LLM-hybrid approach for compiler fuzzing that targets SSA optimization passes of the Go compiler via coverage-guided test case generation using code mutators

```
# Go 1.25.10 enabled optimization:
S3779M10.go:17:32: cannot convert 2 (untyped int constant)
to type chan int
# Go 1.26.3 enabled optimization:
S3779M10.go:17:18: invalid operation: ch_03c33a78 * 2
(mismatched types chan int and untyped int)
```

Figure 5: Go compiler diagnostics changes on mismatched type errors.

synthesized by an LLM. Goopfuzz successfully addresses the cost and computational overhead of modern LLM-based approaches for compiler testing by removing the language model from the fuzzing loop. It also addresses the need for expert developers to implement complex test case generators by leveraging Gemini 3.1 Pro to synthesize AST code mutators via non-expert user prompts.

The experimental evaluation from the synthesized code mutators using Gemini 3.1 Pro highlights positive results in ease of implementation and quality of the mutators and their efficiency. The model was capable of understanding Go’s implementation of SSA passes and synthesized 157 AST code mutators (with 76.25% on the first try) for 46 different SSA passes, needing 83 prompts total. When applied to initial seeds, the mutators expanded the code coverage of the Go compiler’s SSA implementation by +4.11% over 15 batches. The number of buildable mutants surpassed the number of duplicated mutants across all batches, pointing to the efficiency of those test case generators. By the end of the experiment, a total of 165,321 mutants were generated without breaking the execution of mutation operations, demonstrating a robust implementation of code mutators by the LLM.

As future work, feedback from the code coverage captured during the execution of Goopfuzz could guide the LLM to refine generated code mutators, potentially addressing current underexplored code paths. The ease of generating mutators allows for future SSA passes implemented by the Go authors to be trivially synthesized and added to the mutators corpus of this fuzzer.

Artifact Availability

All prompt structures, synthesized mutants and seeds and the python implementation of Goopfuzz are available on the GitHub repository: <https://anonymous.4open.science/r/Goopfuzz-6B5B>.

References

- [1] Scott Bauer, Pascal Cuoq, and John Regehr. 2015. Deniable backdoors using compiler bugs. *International Journal of PoC/GTFO* 08 (2015), 7–9.
- [2] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2021. A Survey of Compiler Testing. *ACM Comput. Surv.* 53, 1 (2021), 4:1–4:36. doi:10.1145/3363562
- [3] Davide Italiano and Chris Cummins. 2025. Finding Missed Code Size Optimizations in Compilers using Large Language Models. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction* (Las Vegas, NV, USA) (CC ’25). Association for Computing Machinery, New York, NY, USA, 81–91. doi:10.1145/3708493.3712686
- [4] Yu Jiang, Jie Liang, Fuchen Ma, Yuanliang Chen, Chijin Zhou, Yuheng Shen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Shanshan Li, and Quan Zhang. 2024. When Fuzzing Meets LLMs: Challenges and Opportunities. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (Porto de Galinhas, Brazil) (FSE 2024). Association for Computing Machinery, New York, NY, USA, 492–496. doi:10.1145/3663529.3663784

- [5] Daniel Kästner, Xavier Leroy, Sandrine Blazy, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. *CompCert - A Formally Verified Optimizing Compiler*.
- [6] Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 47, 11 (2021), 2312–2331. doi:10.1109/TSE.2019.2946563
- [7] Michaël Marcozzi, Qiyi Tang, Alastair F. Donaldson, and Cristian Cadar. 2019. Compiler fuzzing: how much does it matter? *Proc. ACM Program. Lang.* 3, OOPSLA, Article 155 (Oct. 2019), 29 pages. doi:10.1145/3360581
- [8] William M. McKeeman. 1998. Differential Testing for Software. *Digit. Tech. J.* 10 (1998), 100–107. <https://api.semanticscholar.org/CorpusID:14018070>
- [9] Barton P. Miller, Lars Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33, 12 (Dec. 1990), 32–44. doi:10.1145/96267.96279
- [10] Yunbo Ni. 2025. LegoFuzz: Interleaving Large Language Models for Compiler Testing. In *Companion Proceedings of the 2025 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (Singapore, Singapore) (SPLASH Companion '25)*. Association for Computing Machinery, New York, NY, USA, 37–39. doi:10.1145/3758316.3763251
- [11] Xianfei Ou, Cong Li, Yanyan Jiang, and Chang Xu. 2025. The Mutators Reloaded: Fuzzing Compilers with Large Language Model Generated Mutation Operators. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (Hilton La Jolla Torrey Pines, La Jolla, CA, USA) (ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 298–312. doi:10.1145/3622781.3674171
- [12] Gabriel Ricardo, Natanael Santos Junior, Flavio Figueiredo, and Fernando Pereira. 2025. On the Practicality of LLM-Based Compiler Fuzzing. In *Anais do XXIX Simpósio Brasileiro de Linguagens de Programação (Recife/PE)*. SBC, Porto Alegre, RS, Brasil, 59–66. doi:10.5753/sblp.2025.12264
- [13] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward understanding compiler bugs in GCC and LLVM. *Proceedings of the 25th International Symposium on Software Testing and Analysis* (2016). <https://api.semanticscholar.org/CorpusID:8339241>
- [14] Yixuan Tang, Zhilei Ren, Weiqiang Kong, and He Jiang. 2020. Compiler testing: a systematic literature analysis. *Frontiers of Computer Science* 14, 1 (2020), 1–20. doi:10.1007/s11704-019-8231-0
- [15] The Go Authors. 2026. The Go Project - The Go Programming Language. <https://go.dev/project> Accessed: 2026-05-30.
- [16] The Go Authors. 2026. Introduction to the Go compiler - The Go Programming Language. <https://go.dev/src/cmd/compile/README> Accessed: 2026-05-30.
- [17] The Go Authors. n.d.. *cmd/compile/internal/ssa README*. <https://go.dev/src/cmd/compile/internal/ssa/README>. Accessed: 2026-06-01.
- [18] Ken Thompson. 1984. Reflections on trusting trust. *Commun. ACM* 27, 8 (Aug. 1984), 761–763. doi:10.1145/358198.358210
- [19] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software Testing With Large Language Models: Survey, Landscape, and Vision. *IEEE Trans. Softw. Eng.* 50, 4 (April 2024), 911–936. doi:10.1109/TSE.2024.3368208
- [20] Taiyan Wang, Ruipeng Wang, Yu Chen, Lu Yu, Zulie Pan, Min Zhang, Huimin Ma, and Jinghua Zheng. 2024. Enhancing Black-box Compiler Option Fuzzing with LLM through Command Feedback. In *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*. 319–330. doi:10.1109/ISSRE62328.2024.00039
- [21] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4All: Universal Fuzzing with Large Language Models. In *Proceedings of the 46th International Conference on Software Engineering (ICSE '24)*.
- [22] 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. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 296 (Oct. 2024), 27 pages. doi:10.1145/3689736