Interleaving Large Language Models for Compiler Testing

YUNBO NI, The Chinese University of Hong Kong, Hong Kong
SHAOHUA LI*, The Chinese University of Hong Kong, Hong Kong

Testing compilers with AI models, especially large language models (LLMs), has shown great promise. However, current approaches struggle with two key problems: The generated programs for testing compilers are often too simple, and extensive testing with the LLMs is computationally expensive. In this paper, we propose a novel compiler testing framework that decouples the testing process into two distinct phases: an offline phase and an online phase. In the offline phase, we use LLMs to generate a collection of small but feature-rich code pieces. In the online phase, we reuse these code pieces by strategically combining them to build high-quality and valid test programs, which are then used to test compilers.

We implement this idea in a tool, LegoFuzz, for testing C compilers. The results are striking: we found 66 bugs in GCC and LLVM, the most widely used C compilers. Almost half of the bugs are miscompilation bugs, which are serious and hard-to-find bugs that none of the existing LLM-based tools could find. We believe this efficient design opens up new possibilities for using AI models in software testing beyond just C compilers.

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

Additional Key Words and Phrases: Compilers, Testing, Reliability

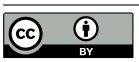
ACM Reference Format:
Yunbo Ni and Shaohua Li*. 2025. Interleaving Large Language Models for Compiler Testing. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 301 (October 2025), 27 pages. https://doi.org/10.1145/3763079

1 Introduction

Compilers play a fundamental role in the modern software ecosystem. However, despite significant efforts to enhance their reliability, they remain prone to bugs [3]. Therefore, large-scale compiler testing is essential for identifying and eliminating these bugs. Various approaches have been proposed, such as random program generation [21, 22, 40] and mutation-based testing [16, 33]. With the advancement of Large Language Models (LLMs), new tools have emerged to enhance compiler testing. A notable example is Fuzz4All [38], a universal fuzzer designed for multiple purposes, including compiler testing. It leverages LLM-generated prompts within the testing loop to iteratively generate test programs, improving the efficiency and diversity of fuzzing. Another tool, WhiteFox [39], is specifically designed for deep learning compilers and features a multi-agent framework. It generates prompts using relevant documentation and example code to guide LLMs in producing targeted and effective test programs.

Although current LLM-based compiler testing has demonstrated great potential, its practical deployment comes with notable challenges. In real-world applications, two major concerns arise: (1) the difficulty of ensuring the quality and validity of generated test cases and (2) the high computational cost of integrating LLMs into large-scale testing workflows. Below, we elaborate on the two core challenges.

*Corresponding author.
Authors' Contact Information: Yunbo Ni, The Chinese University of Hong Kong, Shatin, Hong Kong, ybni@cse.cuhk.edu.hk; Shaohua Li*, The Chinese University of Hong Kong, Shatin, Hong Kong, shaohuali@cse.cuhk.edu.hk.



ACM 2475-1421/2025/10-ART301
https://doi.org/10.1145/3763079

```
1 // missing <inttypes.h>
2 typedef struct {
3   int32_t id;
4 } U;
5 void showU(const U* u) {
6   printf("%"PRId32"\n", u->id);
7 }
```

line 6:error: expected ')' before 'PRId32'

(a) LLM-generated code with syntax error.

```
1 // ...
2 int main() {
3   char username[6];
4   char name[] = "Jonathan";
5   strcpy(username, name);
6   return 0;
7 }
```

line 5:error: AddressSanitizer: stack-buffer-overflow

(b) LLM-generated code with semantics error.

Fig. 1. Examples of LLM-generated erroneous code.

➤ **Challenge 1: Low quality of testing programs.** In compiler testing, we typically need to generate complex and valid programs, both *syntactically* and *semantically*. However, the programs generated by the current LLM-based tools do not always meet these requirements. Unlike natural languages, programs contain rich grammar and semantic constraints, which LLMs are not fully aware of [15]. Thus, LLMs are fundamentally limited in that they are not guaranteed to generate valid programs. For example, Figure 1 (a) shows the C program generated by the model used in Fuzz4All, which contains a syntax error and cannot be compiled by GCC. Figure 1 (b) shows another LLM-generated program, which can be compiled but contains a semantic error, *i.e.*, a buffer overflow in line 5. Such invalid programs can only be used to find crash bugs, not miscompilation bugs. In fact, neither Fuzz4All nor WhiteFox found any miscompilation bugs in C compilers. All GCC bugs found by Fuzz4All are related to compiler front-end crashes rather than core compiler optimizations. WhiteFox identified only two bugs in LLVM, one involving a compiler backend crash and another related to a crash in error diagnostics. According to its own reports, Fuzz4All generates only 37.26% valid C programs for GCC. The average length of its generated programs is just 18 lines, while WhiteFox produces slightly longer programs with an average of 21 lines based on our evaluation. Hence, how to generate complex (*e.g.*, thousands of lines) yet valid programs with LLMs for compiler testing is a challenging problem.

➤ **Challenge 2: High computational cost.** Currently, using LLMs for fuzzing is expensive, both computationally and financially. An effective fuzzing process is usually required to generate a large volume of test inputs in a short period [43]. However, current LLM-based fuzzers are highly constrained by the sheer amount of computational cost of LLMs. For example, Fuzz4All can generate only around 16,000 valid C programs in 24 hours. Extending these LLM-based fuzzers to produce hundreds of millions of programs would require an extremely large amount of computational or financial resources. In contrast, traditional fuzzers can easily achieve such volumes of data on modern hardware, as there is some code analysis checking the changes or the generated code to some degree. For example, Csmith generates around 1 million test programs overnight, with 99.96% validity rate [10, 11, 34].

➤ **Our core idea.** This paper addresses the challenges mentioned above by innovatively decoupling the entire compiler testing process into two distinct phases: *an offline phase* and *an online phase*. Rather than relying solely on LLMs throughout the entire pipeline, we integrate both LLMs and traditional program generation/mutation techniques, each assigned to the phase where they are most effective. In the offline phase, we utilize LLMs to generate small yet feature-rich and valid programs, which are then ***reused as building blocks*** for the online phase. In the online phase, we iteratively use the building blocks to generate increasingly complex yet valid programs for compiler testing. This hybrid approach balances the strengths of both LLMs and traditional methods, optimizing both the quality and the cost of LLM-based compiler testing.

➤ **Our approach: LegoFuzz.** We propose LegoFuzz, a novel compiler testing framework that strategically decouples LLM usage into two synergistic phases: ***an offline phase*** for collecting high-quality code building blocks through LLM-guided transformation of real-world code, and ***an online phase*** for iterative program synthesis that efficiently reuses these building blocks to generate complex test cases without further LLM invocations. Our goal in the first offline phase is to generate programs with measurable complexity while maintaining syntactic and semantic validity. An observation is that LLM-generated code often exhibits recurring structural patterns that correlate with the models' pre-training data [9]. This observation suggests potential limitations in the diversity of generated programs, which could affect their effectiveness for compiler testing. To address this limitation, we leverage existing open-source code as templates to guide LLMs in generating or transforming code snippets. We then validate their syntactic and semantic validity and only keep the valid ones. All code snippets are organized as single functions to facilitate the validation as well as the future synthesis process. Using this approach, we construct a code database containing *over half a million* functions from 146 open-source projects, including system software (*e.g.*, Linux), databases (*e.g.*, Redis), and web servers (*e.g.*, Nginx).

The online phase uses these functions as building blocks to synthesize larger, more complex programs. A crucial step in this phase is to establish dependencies between the functions. Without these dependencies, *i.e.*, simply putting multiple functions in one program, compiling the program would be mostly equivalent to compiling each function separately, thus limiting the coherence of the generated program. For example, the left snippet shows two independent functions, while the right one introduces a call from `a` to `b`, forming a dependency. With such dependencies, we cover 3,726 more lines in LLVM and trigger more analysis and optimizations like inline and instcombine.

```
int b(int y) { return y * 2; }
int a(int x) {
  if (x > 10) return 1;
  else return 0;
}
```

```
int b(int y) { return y * 2; }
int a(int x) {
  if (x > b(5)) return 1;
  else return 0;
}
```

We implement two specific mechanisms to build dependencies between functions: (1) function call insertion, where we insert calls between functions based on type compatibility and semantic constraints, and (2) global variable share, where we introduce shared variables to create verifiable data dependencies between functions.

To evaluate the effectiveness of LegoFuzz, we select two widely used modern C compilers, GCC and LLVM, as our fuzzing targets. By stress-testing their latest versions, LegoFuzz successfully uncovered 66 compiler bugs, among which 30 were miscompilation bugs, and 56 were already fixed by the developers. In conclusion, this paper makes the following contributions:

- We present LegoFuzz, an LLM-based compiler testing framework that decouples the program generation process into offline and online phases, enabling the effective generation of high-quality test programs with LLMs.
- We design a novel real-world code-aligned prompting method to guide LLMs in generating code snippets with a diverse range of features.
- We propose a novel iterative program synthesis method that strategically combines multiple code snippets to complex, feature-rich yet semantically valid programs.
- We implement LegoFuzz for C compiler testing and construct a comprehensive database consisting of over half a million functions generated by LLMs. We conduct an extensive fuzzing campaign with LegoFuzz, discovering 66 unique bugs in widely used modern C compilers, *i.e.*, GCC and LLVM.

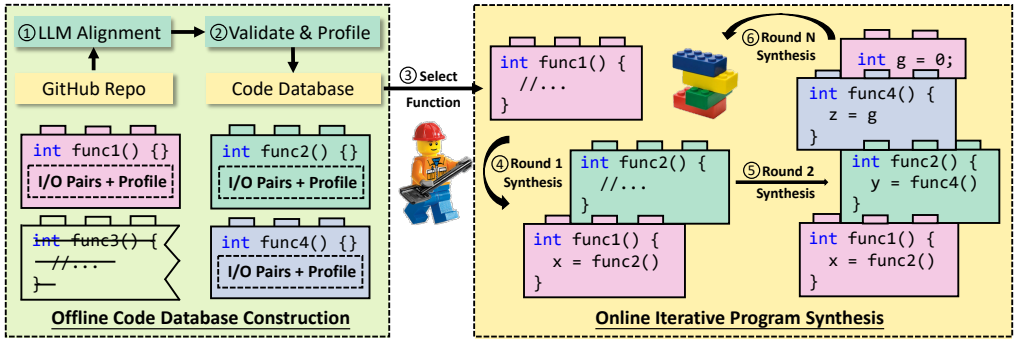


Fig. 2. Design overview of LegoFuzz. The left part is the offline code database construction phase, which provides the building blocks for the online iterative program synthesis on the right.

LegoFuzz has been open-sourced at https://github.com/cuhk-s3/LegoFuzz. We believe that this fuzzing framework offers a promising direction for applying LLMs to real-world compiler testing scenarios. Its modular and extensible design philosophy allows easy adaptation to other systems under test, broadening its applicability beyond C compiler testing.

2 Design Overview

This section outlines the high-level design of our proposed framework LegoFuzz. The core idea of LegoFuzz is to separate the whole testing process into offline and online phases. Figure 2 shows the high-level workflow of LegoFuzz. The offline phase queries an LLM to collect valid code snippets, which enables us to control the quality of code as well as the cost of LLM invocations. In contrast, the online phase eliminates the dependency on LLMs by **reusing** these pre-generated code snippets. Through our proposed *iterative program synthesis*, the online phase constructs increasingly complex yet valid programs, which can be used to test compilers. Below, we provide a more detailed overview of our LegoFuzz:

(1) **Phase 1: Offline Code Database Construction with LLMs.** Firstly, we utilize an LLM to generate a diverse set of code snippets. At a high level, we propose to use real-world code snippets from GitHub repositories as the guiding templates to enable LLMs to generate diverse code snippets. As illustrated in the left part of Figure 2, the first step is to collect code via LLM-based alignment. In this example, four functions, *i.e.*, `func1`, `func2`, `func3`, and `func4`, are collected. In the second step, we analyze each function to determine its validity and get its run-time profiling information. Invalid functions with syntactic or semantic errors are discarded. In this example, the function `func3` is excluded. All collected code snippets, along with their profiling information, are stored in the code database.

(2) **Phase 2: Online Iterative Program Synthesis.** Although the code produced by the LLM contains diverse features, they are still simple, *e.g.*, averagely 30 lines, and we could not find any interesting compiler bugs according to our evaluation. In this paper, we propose an iterative program synthesis method to generate complex programs with these LLM-generated code snippets as building blocks. For example, in the right part of Figure 2, we select the function `func1` as the seed function in Step ③. During Round 1 of synthesis in Step ④, `func1` is modified by replacing the value of `x` with a call to `func2()`. LegoFuzz ensures that this replacement introduces no side effects by leveraging the pre-collected profiling results. Similarly, in the following two iterations of synthesis in Step ⑤, a call to `func4` is inserted into `func2`, and a read from the global variable `g` is inserted into `func4`. The

```
1  unsigned long a;
2  int b() {
3    unsigned long e = a;
4    int c = e;
5    int d = c < 100 ? c : 0;
6    if (d + (int)e & 608) {
7      while (e & 608) {
8        e <<= 1;
9      }
10   }
11   return 0;
12 }
```

Fig. 3. Code that triggers a crash in GCC.

```
1  int a(int b, int c[], int d, int e) {
2    _Bool f = (e == b); int g = 1;
3    if (!f) {
4      g = 0;
5      for (int h = 0; h < d; h++) {
6        if (c[h] == e) {
7          g = 1; break;
8        }
9      }
10   }
11   return g;
12 }
```

Fig. 4. An LLM-generated program with potential overflow.

example program can undergo multiple synthesis iterations in Step ⑥, allowing for the **reuse** of additional building blocks. After synthesis, we generate a driver main function that calls all used functions such as `func1`, `func2`, and `func4`, and operates on global variables like `g` to perform mutations and print checksums for fuzzing. We detail this process in Section 4. Through this *iterative program synthesis* process, the final synthesized program can grow significantly in complexity, ultimately reaching **tens of thousands of lines** in length. Because this online phase directly uses the established code database, it can quickly produce a large number of complex programs without querying LLMs. In some sense, we are maximizing the utility of LLMs' outputs with our iterative synthesis while eliminating the need for continual LLM usage to significantly reduce costs.

LegoFuzz aims at efficiently and effectively generating complex yet valid programs using LLMs. This approach enables the seamless integration of LLM-based fuzzing into real-world testing pipelines. The framework's modular architecture provides great flexibility, allowing developers to easily adapt and extend its capabilities.

3 LegoFuzz

In this section, we describe the technical details of our LegoFuzz framework. Section 3.1 details the offline code collection with large language models, while Section 3.2 introduces the online iterative program synthesis.

3.1 Offline Code Database Construction with Large Language Models

The goal of the offline phase is to collect a set of code snippets using LLMs, which will serve as the building blocks for the later online phase. We use an LLM to first generate these code snippets. Then, we validate them and carefully analyze how they work. By the end of this offline phase, we will have a code database $\mathcal{D}_F$, where each entry $E_i \in \mathcal{D}_F$ contains two essential parts, *i.e.*, $E_i = \{F_i, prof_i\}$: (1) F_i is a code snippet existing as a function, and (2) $prof_i$ is a detailed profile that records how the function F_i behaves when run, such as variable types and variable values. Specifically, each F_i meets the following two criteria:

- **Expressive:** Compilers must handle a wide variety of program structures and features. For effective compiler testing, our generated programs should contain complex and expressive elements such as non-trivial control flows and intricate program semantics. Figure 3 shows one of our reported crash bugs in GCC. This bug is triggered by the *complex, nested, and unbounded loop structure* in lines 6–10. Such sophisticated code patterns are difficult for LLMs to generate through standard natural language prompting. To overcome this limitation,

```c
struct Buffer {
  long nalloc;
  long len;
  char* body;
};

// Issue1: External function
int realloc_body(Buffer*);

// Issue2: Non-numeric I/O
void buf_write(Buffer *b, char c) {
  if (b->nalloc == (b->len + 1)) {
    realloc_body(b);
  }
  b->body[b->len++] = c;
}
```

(a) A simplified program from real-world.

```c
struct Buffer {
  long nalloc;
  long len;
  char* body;
};
int buf_write(int nalloc, int len,
              char *body, char c) {
  Buffer b; b.nalloc = nalloc;
  b.len = len; b.body = body;
  if (b.nalloc == (b.len + 1)) {
    b.nalloc *= 2;
    b.body = realloc(b.body, b.nalloc);
  }
  b.body[b.len++] = c;
  return b.len;
}
```

(b) A program transformed by an LLM.

Fig. 5. The program in (a) is from an open-source project. LLMs transform it to the function in (b).

we introduce real-world code-aligned prompting, a technique that uses real-world code as templates to guide LLM code generation. We explain this approach in detail in Section 3.1.1.

- **Valid:** All generated code must be both syntactically and semantically valid, adhering to the language specification. For C programs, this means passing compiler grammar checks and containing no undefined behaviors at runtime. This property is crucial because *compilers are only designed to correctly compile valid code* [6], making it impossible to detect genuine miscompilation bugs using invalid code. In our design, we ensure validity through rigorous validation and profiling (see details in Section 3.1.2). Figure 4 illustrates a function transformed by an LLM from the open-sourced `Fastsocket` [12] project. This function is only valid when the input variable `d` is less than the length of array `c[]`. For instance, when `d=1` and `len(c)=2`, the function works correctly. However, if `d=3`, a stack buffer-overflow occurs at line 6, making the function invalid. To prevent such issues, for each function, we validate it with diverse randomized inputs and only keep inputs that exhibit no undefined behaviors in the function. With these inputs, we ensure each function can be safely used in our online program synthesis phase. We detail this process in Section 3.1.2.

3.1.1 *Real-world Code-aligned Code Generation.*

LLMs have shown a superior ability in code generation, but it remains challenging to *instruct* LLMs to produce *a large volume of feature-rich code* suitable for compiler testing. Take the code in Figure 5 (a) as an example. It contains a non-primitive struct and manipulates a struct variable inside the `buf_write` function. *How can we instruct LLMs to generate such code?* Precisely describing such code features in natural language is hard, let alone the automated generation of millions of such code. Fortunately, developers have already written such code in real-world projects. The large volume of open-sourced projects exercise a wide range of language features and code patterns, such as pointer manipulation, byte-level operations, non-trivial control flows, *etc.* Our idea is to use these real-world codes as the template to guide LLMs. In fact, the code in Figure 5 (a) is from an open-source project `8cc`[35], and the code in Figure 5 (b) is the transformed version by an LLM that can be used in our LegoFuzz framework. Below, we detail two critical designs of our real-world code-aligned code generation approach.

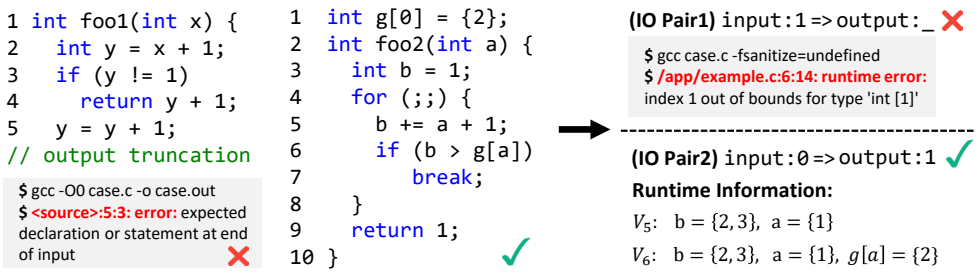


Fig. 6. Process of code database construction.

➤ **Where and how to get real-world code snippets?** In our framework, real-world code snippets can be sourced from a variety of origins with varying levels of granularity. These snippets can range from entire programs to individual functions, as LLMs possess a fundamental ability to understand the surrounding context of a given snippet. In order to control the length of our prompts and facilitate the profiling process, we extract functions from open-source projects as well as the necessary context, such as type definitions and global variables. These real-world functions will then be used as the template to guide LLMs. We will justify our choice of having functions as the granularity in Section 6.

➤ **How to prompt LLMs to generate real-world code-aligned code?** To preserve the expressiveness of LLM-generated code, we introduce *real-world code-aligned prompting*, a technique that guides LLMs in transforming real-world code snippets. To ensure proper transformation, we specifically design the prompt with instructions that enforce two levels of alignment:

- **Syntax-level alignment.** To facilitate the input generation of later profiling process, LegoFuzz requires the transformed function (1) to be numeric, *i.e.*, both its input and output have only numeric types or pointers to numeric types, and (2) to be the sole function with no additional function definitions present. For example, the transformed function in Figure 5 (b) satisfies this requirement. This requirement ensures that we can easily model a function's semantics with its numeric input and output, which eases our implementation of both the later profiling and online program synthesis.
- **Semantics-level alignment.** One might think that LegoFuzz cannot transform real-world code having complex types (*e.g.*, `struct Buffer` in Figure 5 (a)) or multiple functions (*e.g.*, `realloc_body` and `buf_write` in Figure 5 (a)). However, we guide the LLMs to preserve the full semantics of the original program. For complex types like `struct`, initialization can be deferred to the function body. For programs with multiple functions, the transformed output aims to inline all logic into a single function while maintaining the original program's behavior. For example, in Figure 5 (b), the `struct` definition is moved inside the function body, and the struct input is replaced with its numeric fields. Furthermore, LLMs can deduce that `realloc_body` likely reallocates memory for the `body` field of `Buffer`. Consequently, it can be directly implemented using the standard C memory management function `realloc`. This inference achieves the semantics-level alignment with the original program.

Since LLMs do not always follow users' instructions, even if we instruct LLMs to satisfy the above alignments, the transformed function may still violate the above requirements. Thus, we validate and filter these transformed functions in the next step.

3.1.2 *Code Database Construction.*

In this part, we describe how we validate the transformed functions and profile the valid ones. The profiling process is inspired by Hermes [33] and Creal [19].

Although we have instructed the LLM to align the generated functions with the original programs, the output may still exhibit several issues that impact their validity. For instance, current LLMs are constrained by the `max_completion_tokens` parameter [28], which limits the number of output tokens. As a result, generated programs are prone to truncation, leaving them incomplete and cannot be compiled. For example, the first function `foo1` in Figure 6 is incomplete due to truncation, resulting in a compilation error. This function is thus excluded from the code database.

For syntactically valid functions, it is crucial to ensure their runtime behaviors are also correct. Since runtime behaviors are associated with input, different inputs may lead to different behaviors. For instance, the second function `foo2` in Figure 6 is syntactically valid, but it may lead to runtime errors when the input `a` is greater than the length of the global array `g[]`. To ensure runtime validity, we construct a main program that invokes the candidate function with a randomly generated input based on its input types, and then use multiple sanitizers [31, 32]—including ASan, UBSan, MSan, and TypeSan—together with CompCert [5] to cross check if the input triggers runtime errors, a common practice in checking the runtime validity of a program [16, 19]. Since sanitizers can miss certain undefined behaviors in theory [18], we did not encounter such cases during our evaluation. We believe that this is due to our combined use of multiple tools, which makes it highly unlikely that false alarms will occur. If the input does not trigger runtime errors, we then use this input to profile the function and collect the intermediate runtime information. The top right part of Figure 6 shows the first generated input (*i.e.*, `1`) for the function `foo2`, which triggers a stack buffer-overflow. Thus, we discard this input and proceed to generate another input. The second input (*i.e.*, `0`) is valid, so we use it to profile the function and collect its runtime profile $prof$. For each valid function F_i, its profile $prof_i$ contains the following information:

- **input:** The input value of the function, such as `0`.
- **output:** The output of executing the function with the input, such as `1`.
- **expression values at each line:** The values of expressions evaluated at each line of the function, such as the values of `b`, `a`, and `g[a]` at lines 5 and 6, as shown in the bottom right part of Figure 6. We focus on basic expressions that hold values, including variables, array accesses, pointer dereferences, and field accesses.

In practice, LegoFuzz can generate multiple valid inputs for one function, which means one F_i may have multiple profiles $prof_i$. These functions, together with their profiles, are stored in the code database $\mathcal{D}_F$: (1) each function F_i is syntactically valid, and (2) invoking F_i with any input from the corresponding $prof_i$ is semantically valid.

A natural question arises: ***can these individual functions alone detect compiler bugs?*** Our evaluation in Section 5.2 shows that the answer is negative. This finding directly motivates our approach to synthesize complex test programs by combining multiple functions with rich inter-dependencies rather than relying on individual functions alone.

3.2 Online Iterative Program Synthesis

As has been discussed before, effective compiler testing often requires complex programs with rich features. Given the code database, our target is to synthesize complex yet valid programs. One may argue that we can simply put multiple functions into one file to get a complex program, as shown in Figure 7 (a), where we put two individual functions, `func1` and `func2`, into one file. However, since there are no dependencies between these two functions, compilers will deal with them separately at the module level. Thus, it is almost equivalent to testing the compilers with each function individually.

```c
1  int g = 4;
2  // Input: {1, 0} => Output: 2
3  int func1(int a, unsigned int b){
4    // prof: a = {1}
5    b = a + 1;
6    return b;
7  }
8  // Input: {1, 2} => Output: 5
9  struct S { int c; };
10 int func2(char d, int e) {
11   struct S s;
12   // prof: d = {1}, e = {2}
13   s.c = d;
14   for (;;) {
15     // prof: e = {2}
16     d = e + s.c;
17     if (s.c > e) break;
18     s.c ++;
19   }
20   return d;
21 }
22 int main() { func2(1, 2); }
```

(a) Example at the start of the synthesis process.

Two rounds of iterative synthesis

```c
1  int g = 4;
2
3  int func1(int a, unsigned int b){
4    // prof: a = {1}
5    b = a + 1; g = g + a - 1;
6    return b;
7  }
8
9  struct S { int c; };
10 int func2(char d, int e) {
11   struct S s;
12   // prof: d = {1}, e = {2}
13   s.c = func1(d, e - 2) - 1;
14   for (;;) {
15     // prof: e = {2}
16     d = g - 2 + s.c;
17     if (s.c > e) break;
18     s.c ++;
19   }
20   return d;
21 }
22 int main() { func2(1, 2); }
```

(b) Example program after iterative synthesis.

Fig. 7. The example in (a) is the beginning of one synthesis iteration. One possible synthesis result after two iterations is shown in (b). The synthesized part is highlighted in `gray` in (b).

Our core idea is to couple multiple functions by building **complex dependencies** among them. We build the dependencies in two ways: (1) function call insertion and (2) global variable share. For example, the program in Figure 7 (b) shows the resulting program from Figure 7 (a) by building dependencies with LegoFuzz. In Figure 7 (b) at line 13, we replace the expression `d` with a call to `func1` to build dependency; The global variable `g` is written in `func1` at line 5 and read by `func2` at line 16, further building dependencies between them.

Algorithm 1 provides the algorithmic sketch of our iterative program synthesis approach. We use the example in Figure 7 to illustrate the synthesis process. Given a code database $\mathcal{D}_F$ and a predefined iteration number $\mathcal{N}$, our generator operates as follows:

Step 1. *Generate Global Variables* (line 2): Similar to Csmith and other generative black-box tools [21, 22, 40], we first generate a set of global variables $\mathcal{G}$ with random numeric types and values. For example, the global variable `g` in Figure 7 (b) is generated with a random value of 4.

Step 2. *Prepare Seed Program* (lines 3-5): A function is randomly selected from $\mathcal{D}_F$ as the seed function. Next, we generate the driver program $\mathcal{P}$ to invoke it with an input. The seed function is then added to the list of used functions *F_list*. For example, `func2()` is the selected seed function in Figure 7 (b), and we synthesized the driver main function to invoke it with an input selected from its profile, *i.e.*, {1, 2}.

Step 3. *Select Target Function* (lines 7-9): For each iteration, the process begins by selecting a target function, denoted as *Target*, which will be used in the future synthesis. A set of matched expressions $\mathcal{E}$ is then extracted from its stored profiling result *prof*. Since there is only one function in the first iteration, *i.e.*, `func2()` in Figure 7 (a), the target function is `func2()`. The matched expressions are `d` at line 13, `e` and `s.c` at lines 16 and 17. The high-level guideline of selecting these matched expressions is to choose the ones that can be replaced by other

Algorithm 1: Iterative Program Synthesis

```
1  procedure Synthesis(Code Database D_F, Iteration Number N):
2      G ← GenerateGlobalVars()
3      Seed ← SelectFunction(D_F)
4      P ← SynthDriverProgram(Seed)
5      F_list ← [Seed]
6      repeat
7          Target ← GetFunction(F_list)
8          prof ← GetProfile(Target)
9          E ← GetMatchedExpr(prof)
10         foreach expr ∈ E do
               // randomly decide if to sythesize expr
11             if FlipCoin() then
                   // randomly choose function call insertion or global variable share
12                 if FlipCoin() then
13                     F ← SelectFunction(D_F)
14                     if F ∉ F_list then
15                         expr' ← SynFuncCall(expr, F, prof)
16                         P ← InsertFunc(P, expr, expr', Target)
17                         F_list.append(F)
18                 else
19                     g ← GetGlobalVar(G)
20                     expr' ← SynGlobal(expr, g, prof)
21                     P ← InsertGlobalVar(P, expr, expr', Target)
22     until N times
23     return P
```

expressions with the same runtime values. Note that the runtime values of these expressions under the input are available and stored in *prof*. For example, the runtime value of `d` is 1 at line 13, as annotated in line 12 in Figure 7 (a).

Step 4. *Build Dependency by Function Call Insertion* (lines 13-17): If the expression is selected to be synthesized with a function call, the generator first selects a function F from $\mathcal{D}_f$. To avoid potential stack overflow from recursive or cyclic calls, the generator checks if F is already in the *F_list*, *i.e.*, whether F has been used or not. If F is not in *F_list*, a new expression $expr'$ with a call to F is synthesized to replace the original expression $expr$, and F is added to *F_list*. For example, from Figure 7 (a) to (b), the expression `d` at line 13 is replaced by a function call to `func1()`. We will discuss the details of function call insertion in Section 3.2.1.

Step 5. *Build Dependency by Global Variable Share* (lines 19-21): If the expression is selected to be synthesized with a global variable, the generator randomly selects a global variable g from $\mathcal{G}$. It then synthesizes a read or write operation for g with $expr$. For example, from Figure 7 (a) to (b), the expression `e` is replaced by `g - 2` in line 16. Another usage of the global variable `g` is also shown at line 5, where we write `g` with `a` in the next iteration. The read or write from/to the same global variable builds dependencies between functions `func1()` and `func2()`. We will discuss the details of global variable share in Section 3.2.2.

Algorithm 2: Function Call Insertion.

```
1  procedure SynFuncCall(Target Expression expr, Function F, Profile prof):
2      [inp_1, inp_2, ···, inp_m] ← F.input
3      FC = " F_i.name (<para>_1, <para>_2, ···, <para>_m)"
4      foreach k ∈ [1...m] do
5          V ← GetStableVariables(prof)
6          para ← SynthesizeExpression (V, inp_k)
7          FC.Substitute(<para>_k, para)
8      if IsStable(expr) then
9          val ← GetValue(expr)
10         expr' ← SynthesizeExpression(FC, val)
11     else
12         expr' ← expr + SynthesizeExpression(FC, 0)
13     return expr'
```

As shown in the algorithm, the synthesis iterates over all matched expressions, indicating that multiple insertions can occur in a single iteration. Consequently, the finally synthesized program may contain more than one function call. Additionally, by using a predefined iteration number $\mathcal{N}$, the size of the synthesized program remains controllable. The details of building dependencies in Steps 5 and 6 are discussed in Section 3.2.1 and Section 3.2.2. The key to building dependencies is to ensure that the synthesized program has the same semantics as the seed program. For example, the final synthesized program in Figure 7 (b) has the same output as the seed program in Figure 7 (a). This semantic preservation relies on the valid profiling results obtained in the offline phase, which guide the safe dependency construction. Since all profiling is completed offline, it does not impact the efficiency of the online synthesis process.

3.2.1 *Build Dependency by Function Call Insertion.*

Algorithm 2 outlines the procedure for synthesizing an expression using a function call. It begins by extracting the inputs of F stored in the code database and constructs a function call with parameter placeholders that match the number of inputs (lines 2-3). For each placeholder, the set of stable variables V—those that exhibit only one run-time value at a given location—is collected from the stored profiling results *prof* (line 5). The placeholder is then substituted with a synthesized parameter *para* that aligns with the value (lines 6-7). The expression *expr* is then handled based on its stability in the following two cases: (1) *expr* is stable (lines 8-10): The corresponding stable value *val* is retrieved, and a new expression *expr′* is synthesized by combining the return value of F with *val*. (2) *expr* is unstable (lines 11-12): A synthesized expression synthesized with *FC* equal to 0 is concatenated with the original expression to form the new expression *expr′*.

Example. As shown in Figure 7(a), the target expression d in line 13 is stable with a single value of 1. Our target is to synthesize a new expression with a call to `func1()` that also evaluates to 1:

- (Line 2 in Algorithm 2) We extract the input list for `func1()` as $\{1, 0\}$.
- (Line 3 in Algorithm 2) We construct the function call `FC` as "`func1(<para>`$_1$`, <para>`$_2$`)`".
- (Lines 5-6 in Algorithm 2) We first retrieve the set of stable variables $V = \{d, e\}$, all of which have only one runtime value. For each input, we generate $\texttt{para}_1$ as the expression d, which evaluates to 1, and $\texttt{para}_2$ as $e - 2$, which evaluates to `0`. Therefore, `FC` is updated with the new parameters, resulting in "`func1(`d`, `$e-2$`)`", which is semantically equivalent to "`func1(1, 0)`".

Algorithm 3: Global Variable Share.

```
1  procedure SynGlobal(Target Expression expr, Global Variable G, Profile prof):
2      if IsStable(expr) then
3          val ← GetValue(expr)
4          if FlipCoin() then
5              expr′ ← SynthesizeExpression(G, val)
6          else
7              stmt_write = "G = G + (expr − val);"
8              InsertStatementAfter(stmt_wirte)
9      else
10         expr′ ← expr + SynthesizeExpression(G, 0)
11     return expr′
```

- (Lines 8-10 in Algorithm 2) Since the target expression d is stable, we retrieve its value, *i.e.*, $val = 1$. Since the return value of `func1()` is 2, we then synthesize the new expression $expr'$ as `func1(`d`,` $e - 2$`)` -1, which evaluates to 1 and equals to the runtime value of the target expression d. The "-1" after the function call is to massage the return value of `func1()` to the same value as the target expression d. In our implementation, we used several operators, such as "+" and "-", to achieve this.

3.2.2 *Build Dependency by Global Variable Share.*

Algorithm 3 describes the process of synthesizing global variable share. It breaks down into synthesizing read and write operations to the global variables. The construction of a read expression is similar to a function call insertion. Specifically, the new expression $expr'$ is synthesized based on the stability of the target expression $expr$. If $expr$ is stable, the stable value val is used to construct a new expression $expr'$ with the same value (lines 4-5). For unstable expressions, we synthesize a new expression that equals 0 and concatenate it with $expr$ (lines 9-10). When synthesizing a write operation, we need to ensure that the generated expression does not alter the program semantics unexpectedly. The reason is that global variables are shared by multiple functions, and their values need to be statically known so that our generator can precisely control the program semantics. We achieve this by only using stable variables for global variable writes. A write statement is constructed by combining g and the expression $expr - val$, which is 0 as val is the runtime value of $expr$. This ensures that the synthesized write operation does not change the runtime value of g (line 7). This write operation is then inserted into the program at an appropriate location (line 8).

Example. We show how to synthesize the read and write operations for the global variable g from Figure 7 (a) to (b) with Algorithm 3:

- (Lines 2-3 in Algorithm 3) Suppose the target function is `func2()` and the target expression is e at line 16 in the first iteration. Since e is stable, we can generate either read or write operations for the global variable g. We retrieve the value of e and assign val as 2.
- (Lines 4-5 in Algorithm 3) We synthesize the new expression $expr'$ as $g - 2$, which evaluates to the same value 2 as the original expression e. Then, we replace the original expression e with $expr'$ in the program.
- (Line 6-8 in Algorithm 3) In the next iteration, suppose we select `func1()` as the target function and a at line 5 as the target expression, we generate the write statement "$g = g + a - 1$;" and

insert it after this location. Because $a - 1$ is 0, the write operation does not change the runtime value of g.

3.3 Beyond Creal: Design Innovations in LegoFuzz

LegoFuzz not only addresses the fundamental challenges faced by existing LLM-based compiler testing tools, but also introduces key innovations that go beyond the capabilities of traditional fuzzing frameworks. Creal [19] serves as a representative approach in this space, combining real-world code with Csmith-generated seeds to construct test programs. Thus, Creal's effectiveness relies heavily on Csmith seeds, while LegoFuzz is Csmith-free and only uses LLM-generated code through iterative synthesis. From this perspective, LegoFuzz can potentially extend to other languages where a mature Csmith-like generator is not available. In addition, as detailed below, LegoFuzz diverges from Creal in both the construction of its test database and the synthesis of complete programs, offering a wider applicability and greater flexibility.

3.3.1 Enhancing Code Diversity via Code-Aligned Prompting. For the database construction, the core contribution of LegoFuzz is the *real-world code-aligned prompting* in Section 3.1.1, which addresses the challenge of getting diverse and valid tests in LLM-based testing, which neither Creal nor other LLM-based compiler testing approaches can solve. For example, Creal is unable to transform the program shown in Figure 5 (a), as it cannot handle non-numeric types in the `buf_write` function syntactically, nor can it resolve the external function `realloc_body` semantically. As a result, it achieves only a 5% valid extraction rate. In contrast, LegoFuzz successfully processes such cases, achieving a valid rate of over 50%.

3.3.2 Generating Complex Programs through Iterative Synthesis. For connecting program fragments, LegoFuzz presents a novel iterative synthesis methodology that connects different fragments through function call insertion and global variable sharing, as detailed in Section 3.2. The function call insertion is indeed similar to Creal, but Creal's design only supports **one-time** insertion with limited types, whereas LegoFuzz supports **unlimited insertion times** with richer type support. Other components, like global variable sharing and whole program construction, are **all unique** in LegoFuzz. This limitation prevents Creal from covering more lines of code or detecting certain classes of bugs. We will demonstrate this in Section 5 through both coverage analysis and a case study highlighting LegoFuzz's ability to uncover complex bug patterns beyond Creal's reach.

4 Implementation

This section details the implementation of real-world code-aligned prompting and explains how the synthesizer integrates into the overall fuzzing process.

4.1 Prompt Design

Given a piece of code extracted from a real-world project, LegoFuzz uses a pre-defined prompt to guide the LLM to transform the code into a new version. Figure 8 shows the pre-defined prompt. It begins by defining the role of the LLM as an expert in writing programs and describing the overall task. The transformation process is guided by the *chain-of-thought prompt* strategy [37], which decomposes the task into ordered steps: understanding the code first, followed by generating the function, and finally verifying the result. To ensure real-world code alignment, detailed instructions are provided to enforce explicit numeric I/O, preserve logical structures, maintain compatibility, and avoid undefined behaviors. We also apply *few-shot in-context learning* [2], offering examples of both correct and incorrect transformations as well as explanations. The prompt concludes by specifying output format constraints to ensure that the transformed code can be extracted effectively from the response. By using this prompt with LLMs, we successfully transform `{provided_code_snippet}`

1. System and Task

System: You are an expert in converting complex C programs into a single, standalone C function. Your role is to analyze the given C program, understand its logic in depth, and transform it into one self-contained C function that preserves the original behavior while adhering to all specified constraints.

Task: Convert the given C program to one single compilable C function that takes integer inputs and returns an integer: {provided_code_snippet}

2. Steps and Instructions

Steps:
1. Extract definitions
2. Extract type declarations
3. Process variables and functions
4. Merge logic of all involved functions
5. Final verification for the function

(Note: each step is attached with explanation.)

Instructions:
1. Ensure explicit integer-only I/O
2. Generate exactly one function
3. Preserve all logic
4. Ensure compatibility
5. No standard I/O functions
6. Avoid undefined behavior

(Note: each point is attached with explanation.)

3. Adversial Examples

Example of incorrect trasnformation:
- Original: {original_code_snippet_A}
- Transformed: {incorrect_code_snippet}

(With the issues of the incorrect transformation)

Example of correct transformation:
- Original: {original_code_snippet_B}
- Transformed: {correct_code_snippet}

(With explanation of the correct transformation)

4. Output Requirement

Requirement: Provide only the correctly transformed function without any explanations or comments. Ensure the output adheres strictly to all the above instructions.

Fig. 8. Pre-defined prompt guiding LLMs to transform real-world code.

into the formatted code. An example of this transformation has been shown in Figure 5. Note that other prompt designs may also work well. Our prompt here provides a useful example that is used in our implementation. Designing better prompting methods is interesting, but orthogonal to our work.

4.2 Fuzzing Execution

Detecting crash bugs is straightforward. If a compiler crashes when compiling a test program generated by LegoFuzz, we find a crash bug. In order to find miscompilation bugs, like CSmith [40] and other tools derived from it [19], LegoFuzz also employs randomized differential testing. We compute and print the checksum of the return values from the used functions and global variables to illustrate the behavior of the generated program. Specifically, we mutate a global variable by combining the return values of functions with the values of collected global variables. This mutation process is encapsulated within the main function, and a print statement is included to output the checksum of the modified global variable. For example, in Figure 7, we will add more print statements in the main function to output the checksum of the global variable `g` and the return values of both `func1()` and `func2()`. Finally, we compare the checksum values generated by different compilers and across various optimization levels to identify discrepant outputs. Discrepant outputs indicate the presence of a miscompilation bug in one of the compilers. We then reduce the generated program to a minimal example to manually decide which compiler is buggy and where to report the bug.

5 Evaluation

In this section, we evaluate the effectiveness and design choices of LegoFuzz through the following research questions:

- **RQ1 (Bug finding and coverage analysis).** *Is LegoFuzz effective in finding crash and miscompilation bugs in C compilers, and achieving high coverage?*
- **RQ2 (Ablation analysis).** *How important are the key components of LegoFuzz, including the code database, the choice of LLM, and the iteration number?*
- **RQ3 (Comparison with existing tools).** *How does LegoFuzz compare with existing state-of-the-art LLM-based testing tools such as Fuzz4All, WhiteFox, and Creal?*
- **RQ4 (Case study).** *What types of bugs can LegoFuzz uncover, and how do they demonstrate the tool's unique strengths?*

5.1 Experiment Setup

Compiler Targets. Our study primarily focuses on the two most widely used and mature C compilers, *i.e.*, GCC (from `9366940` to `eb26b66`) and LLVM (from `b1560bd` to `029cb8a`). To ensure up-to-date evaluations, we update both compilers daily and utilize their latest versions for continuous testing. We apply five standard optimization levels, *i.e.*, `-O0`, `-O1`, `-Os`, `-O2`, and `-O3`, across both compilers.

Large Language Models. We utilized ChatGPT-4o-mini, a fast and cost-effective lightweight model for LLM-based transformations. Specifically, we employ the `gpt-4o-mini-2024-07-18` checkpoint with a `max_token` limit of 512 and a `temperature` setting of 0.7. Note that LegoFuzz is not limited to this specific model and also supports locally deployed LLMs. We will evaluate the potential impacts of different models in Section 5.3.

Code Database. Our toolchain for database construction is partially built upon Creal [19]. To further enhance the function extraction capability from real-world projects, we extend its capabilities to support additional C features, such as structs, typedefs, and more. Instead of directly crawling open-source projects from GitHub, we selected AnghaBench [8], which provides a vast collection of over 1,040,000 functions extracted from 146 open-source projects on GitHub. These projects include widely used software such as Linux, Redis, and Nginx, among others. After the offline code collection described in Section 3.1, we constructed a database with 553,246 functions, around 53% of all functions in AnghaBench. Most discarded functions are due to the LLM not extracting an invalid function from them. Although the LLM does not always follow our instructions, our filtering process ensures that only valid code is preserved. Among these discarded functions, the most common reason is the violation of the first instruction in Figure 8, with an invalid I/O function rate of 14.2%. The whole database construction costs us $394 in invoking ChatGPT-4o-mini. Figure 9 shows the distribution of lines of code and branches in our database. The majority of functions contain fewer than 60 lines of code and fewer than 10 branches. On average, each function consists of approximately 30 lines of code and 3 branches. As has been discussed in Section 3.1.2, these functions alone can not find any compiler bugs.

Environment. We conducted all our evaluations on one Linux server running Ubuntu 20.04 LTS. It is equipped with an AMD EPYC 7742 64-core CPU and 256GB RAM.

Testing Process. We conducted the fuzzing process continuously on a dedicated server to ensure thorough and uninterrupted testing. In each round, LegoFuzz selects a single function as the seed and generates 10 mutant versions of it. For each synthesized test case, we compile and execute the program using both GCC and LLVM, leveraging their outputs as the test oracle, as discussed in Section 4.2. Each fuzzing process is constrained to 1 GB of memory and a 200-second compilation timeout to prevent resource exhaustion and ensure fairness. If a miscompilation or runtime crash is detected, we employ C-Reduce [7] to minimize the faulty program, isolating the root cause of

Table 1. Status of Reported Bugs

Status	GCC	LLVM	Total
Confirmed	0	2	2
Fixed	18	38	56
Duplicate	5	3	8
Total	23	43	66

Table 2. Symptoms of Reported Bugs

Symptom	GCC	LLVM	Total
Crash	7	29	36
Miscompilation	16	14	30
Total	23	43	66

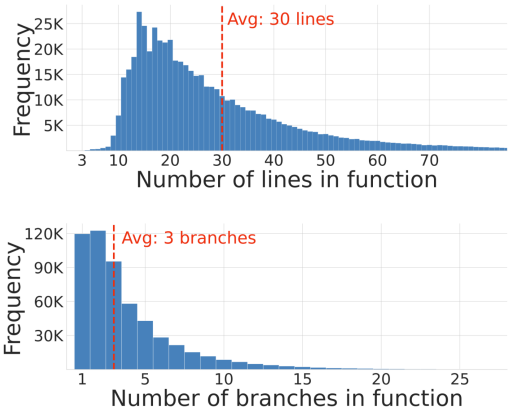


Fig. 9. Distributions of functions in our database.

the issue. Finally, we submit a detailed bug report, including the reduced test case, to the respective compiler developers for further diagnosis.

5.2 RQ1: Bug Finding and Coverage Analysis

Bug-finding capability and program coverage are fundamental metrics for evaluating the effectiveness of a testing approach. We first present the bug-finding results of LegoFuzz, followed by an analysis of the coverage achieved by its synthesized programs.

Baseline: individual function testing. We conducted an experiment using functions directly from our database to test the latest versions of GCC and LLVM. The results were conclusive — *no compiler bugs were detected.* This outcome is not surprising. While our database collectively contains diverse program features, each individual function implements only a limited subset of these features. Modern compiler optimizations operate on complex interactions between program elements, often requiring specific combinations of features to trigger bugs. Programs that expose compiler bugs typically contain intricate control flows, data dependencies, and feature interactions that simple, isolated functions lack.

Number of Bugs. Table 1 provides the status summary of all reported bugs identified by LegoFuzz. In total, LegoFuzz has reported 66 bugs; 58 of them (88%) have been confirmed as previously unknown and new bugs, and 56 bugs (85%) have already been fixed. Specifically, GCC developers have fixed 100% (18/18) of the reported bugs, while LLVM developers have fixed 95.0% (38/40) of them. This highlights the effectiveness of LegoFuzz in identifying critical issues and the willingness of both GCC and LLVM developers to address our reported bugs. Since compiler maintainers, users, and testers are also testing compilers, LegoFuzz reported 8 duplicate bugs that were concurrently identified by them. *Nevertheless, the significant number of new bugs identified by LegoFuzz demonstrates its strong bug-finding capability.*

Symptoms of Bugs. Table 2 summarizes the symptoms of our reported bugs. These bugs can be categorized into two main types: (1) *Crash*: This type of bug occurs when the compiler encounters an internal error during the compilation process, typically due to assertion failures or runtime failures. (2) *Miscompilation*: In this case, the compiler implicitly generates incorrect code without any observable consequences. Such bugs are hard to detect and are the most concerning type of bugs in compilers [3]. As shown in the table, *nearly half of the bugs are miscompilation bugs*, demonstrating the strong capability of LegoFuzz in detecting hard-to-detect bugs. Detecting such

Developer Discussion

GCC Maintainer:
"I am kinda of shock that smtgcc didn't find this earlier."

smtgcc Maintainer:
"I'll add support for abort. Will be fun to see if we find more ancient bugs!"

Fig. 10. Developers' discussions on bug report https://gcc.gnu.org/bugzilla/show_bug.cgi?id=118915.

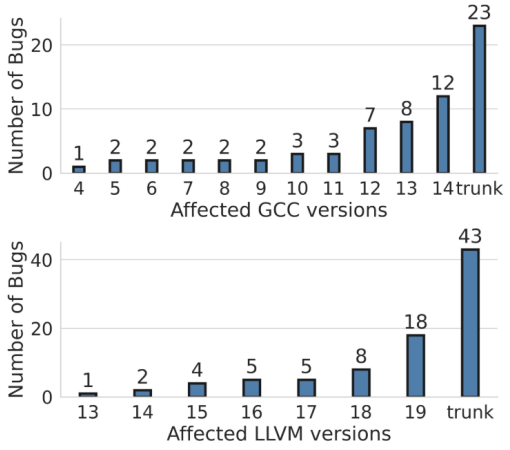


Fig. 11. Stable compiler versions affected by our reported bugs.

Table 3. Affected LLVM components.

Component	#Bugs
Loop Transformations	15
Vectorization Optimization	8
Peephole Optimizations	4
SLP Vectorization	4
Backend	2
Selection DAG	2
Scalar Evolution Analysis	2
Dead Store Elimination	1
Dominance based Optimizations	1
Induction Variable Transformations	1
Loop Invariant Code Motion	1

Table 4. Affected GCC components.

Component	#Bugs
Peephole Optimizations	8
Backend	3
CFG Transformations	2
Constant Propagation	2
Loop Invariant Motion	2
IPA constant propagation	1
Number of Iterations Analysis	1
Predictive Commoning	1
Value Numbering	1
Value Range Analysis	1
Vectorization	1

miscompilation bugs requires valid and feature-rich testing programs. As a comparison, none of the existing LLM-based compiler testing tools [38, 39] can find any miscompilation bugs in either GCC or LLVM.

Importance of bugs. To assess the impact of the discovered bugs, we tested each bug-triggering program to see if they can trigger crashes or miscompilations on earlier stable compiler versions. Figure 11 shows the number of bugs affecting different compiler versions. The result indicates that LegoFuzz is highly effective at uncovering *long-latent* bugs — issues that have persisted undetected for years despite extensive compiler testing efforts. Notably, we identified 7 bugs that affected GCC versions predating GCC-12 and 8 bugs that impacted LLVM versions released before LLVM-18. Given that these compiler versions have been released from 1 to 10 years ago, the longevity of these undetected bugs underscores their criticality. Remarkably, one particular GCC miscompilation bug was traced back to a code change that has existed since at least 2006. This demonstrates that the bug remained unnoticed for nearly two decades, highlighting the limitations of existing testing methodologies. Since there are numerous compiler testing efforts in both academia and industry, *the fact that these long-latent bugs have evaded all previous testing techniques further highlights the exceptional bug-finding capability of LegoFuzz.*

Compiler	Generator	FC	LC	BC
GCC	Seeds (1,000)	31.9%	27.0%	16.0%
	Functions (10,000)	36.0% (+3,575)	33.3% (+55,916)	20.7% (+51,419)
	LegoFuzz (10,000)	**38.9% (+6,103)**	**39.5% (+110,945)**	**25.9% (+108,308)**
LLVM	Seeds (1,000)	24.3%	31.7%	16.0%
	Functions (10,000)	26.4% (+1,979)	34.6% (+47,522)	20.3% (+30,008)
	LegoFuzz (10,000)	**27.6% (+3,110)**	**36.6% (+80,295)**	**23.8% (+54,433)**

Table 5. Line coverage (LC), function coverage (FC), and branch coverage (BC) of GCC and LLVM.

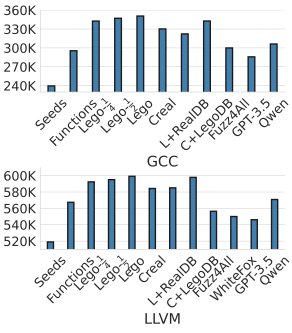


Fig. 12. Line coverage of different variants

Beyond exposing long-hidden issues, some of our discovered bugs also provide valuable insights for the improvement of future compilers or related tools. For example, one miscompilation bug[1] was found to have implications for future, yet-to-be-developed versions of GCC. Furthermore, discussions regarding the root cause of a long-latent GCC miscompilation bug led to unexpected reactions from developers. Figure 10 shows the comments we received for one of our reported bugs. A GCC maintainer expressed the surprise that our reported bug had not been detected by smtgcc [36], which is designed to guarantee the detection of the reported bug. In response, the smtgcc maintainer acknowledged the limitation and stated that he would introduce additional support to detect similar *ancient* bugs in future analyses.

Affected compiler components. Table 4 and Table 3 summarize the compiler components affected by the bugs we identified in GCC and LLVM, respectively. These components were determined based on the diagnostic information and fix messages provided by compiler maintainers. As shown in the results, LegoFuzz is capable of uncovering a diverse range of bugs. In both GCC and LLVM, many of the bugs are related to loop transformations and peephole optimizations, which is consistent with findings from prior empirical studies [42]. To offer a more comprehensive understanding, we will present representative bug cases in Section 5.5, demonstrating the distinctive characteristics of our synthesized programs and their effectiveness in compiler testing.

Coverage and generation speed. We perform the coverage analysis and track the generation time throughout the fuzzing process to assess whether LegoFuzz can effectively explore more parts of the compilers and how efficient it is. Starting from 1,000 randomly selected seed functions, we let LegoFuzz generate 10 programs for each seed, resulting in 10,000 programs produced by LegoFuzz. We then measure function coverage, line coverage, and branch coverage of these programs across GCC and LLVM.

Table 5 and Figure 12 present the results of our coverage analysis. The "Seed" rows in Table 5 and bars in Figure 12 show the coverage of the 1,000 seed functions that LegoFuzz starts from. The "Functions" rows and bars show the coverage of 10,000 randomly selected functions from our database. That is, we directly use these functions individually as testing programs. The "LegoFuzz" rows and bars show the coverage of the 10,000 programs generated by LegoFuzz. It is not surprising that LegoFuzz significantly enhances coverage compared to the seed functions. Specifically, LegoFuzz increases line coverage in GCC by 12.5%, corresponding to 110,945 additional lines, and

[1] https://gcc.gnu.org/bugzilla/show_bug.cgi?id=118638

in LLVM by 2.9%, covering 80,295 more lines. Similar trends are observed for function coverage and branch coverage. Compared to "Functions", LegoFuzz achieves substantially higher coverage, indicating that iterative synthesis plays a crucial role in generating more complex programs that engage a wider range of compiler features.

We also logged the time used to generate the 10,000 programs by LegoFuzz. In total, LegoFuzz generated these 10,000 programs in 193 seconds, averaging 0.02 seconds per program. This high efficiency indicates that program generation is not a bottleneck for LegoFuzz. In fact, we observed that most of the time during compiler testing is spent on compiling the generated programs, which can often take several seconds per program.

5.3 RQ2: Ablation Analysis

We study the impact of the code database, the choice of LLM, and the iteration number on the effectiveness of LegoFuzz, using coverage as the primary metric.

Importance of code database. The effectiveness of LegoFuzz is closely related to the quality of the code database. A consequent question is *whether the quality and size of the code database contribute to the effectiveness of LegoFuzz*. To answer this question, we conduct a coverage analysis on different variants of the database:

- *LegoFuzz-*$\frac{1}{2}$: We reduce the size of the code database used in LegoFuzz by half. Specifically, we randomly select half of the available functions and use this smaller subset during generation of 10,000 cases.
- *LegoFuzz-*$\frac{1}{4}$: We further halve the database, reducing it to a quarter of its original size to generate 10,000 cases.
- *LegoFuzz-Creal*: We apply Creal's original database to guide LegoFuzz in generating 10,000 test cases. The resulting coverage is shown in the "L+RealDB" bar in Figure 12.
- *Creal-LegoFuzz*: We use LegoFuzz's code database to generate 10,000 test cases under the Creal framework. The "C+LegoDB" bar in Figure 12 shows the resulting coverage.

The third to fifth bars in Figure 12 show the results of using different size of LegoFuzz's code database. We can observe a clear trend: as the code database size increases, so does the achieved line coverage. LegoFuzz outperforms LegoFuzz-$\frac{1}{2}$ and LegoFuzz-$\frac{1}{2}$ outperforms LegoFuzz-$\frac{1}{4}$ in both GCC and LLVM, indicating that a larger code database provides more diverse features. This also suggests that future work on establishing a larger code database can potentially further improve LegoFuzz. We observe that applying Creal's database to guide LegoFuzz ("L+RealDB") leads to reduced coverage compared to LegoFuzz with its own database, highlighting the importance of our LLM-driven offline phase and real-world code-aligned prompting strategy. Meanwhile, when Creal is guided by our database ("C+LegoDB"), it achieves substantially higher coverage than using its original database ("Creal"), as shown in Figure 12. This demonstrates that our code database is also broadly effective across different frameworks.

Alternative Large Language Models. In our current implementation, we utilize ChatGPT-4o-mini as the underlying LLM. Although the choice of LLM is orthogonal to the design of our framework, we aim to explore the adaptability and generalizability of our framework across different models. We select ChatGPT-3.5-turbo [29] and Qwen2.5 Coder 32B Instruct [30] for evaluation. Since GPT-3.5-turbo is a legacy model, we aim to evaluate whether our framework can effectively adapt to a less powerful model. Since Qwen Coder is a well-known code-specific LLM, we aim to evaluate whether a specialized model can generate a more expressive code database.

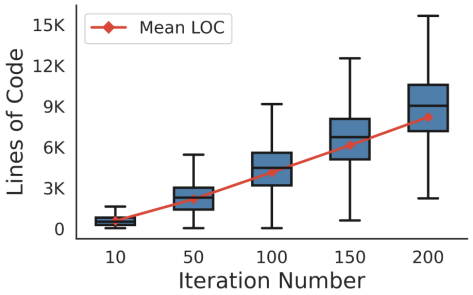


Fig. 13. Relationship between iteration number and lines of code.

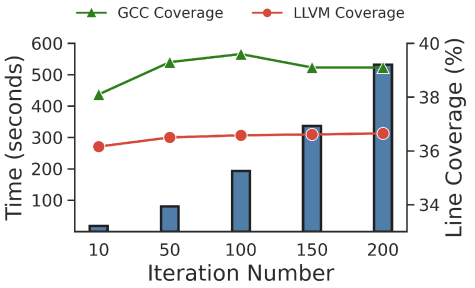


Fig. 14. Relationship between iteration number, generation time, and line coverage.

The "Functions" rows in Table 5 show the coverage of the 10,000 functions in the database used by LegoFuzz. These functions are generated by GPT-4o-mini by transforming 10,000 real-world code snippets. With the same pipeline, we ask each of the two new LLMs to generate 10,000 functions. Finally, we evaluate the coverage of these two sets of new functions.

The "GPT-3.5" and "Qwen" bars in Figure 12 show the coverage of the 10,000 functions generated by GPT-3.5-turbo and Qwen Coder, respectively. As expected, GPT-3.5-turbo achieves lower coverage compared to the more advanced GPT-4o-mini ("Functions"). Qwen Coder, on the other hand, achieves higher coverage compared to GPT-4o-mini, indicating that a specialized model can generate a more expressive code database and potentially improve the performance of LegoFuzz. Nevertheless, our design of LegoFuzz is loosely coupled with the choice of LLM, and the prompt can be easily adapted to better suit specific LLMs. Our core contribution is the design of a novel testing framework, which is orthogonal to the choice of LLM.

Impact of iteration number. In Algorithm 1, LegoFuzz uses the iteration number $\mathcal{N}$ to control the number of synthesis iterations during the online phase. To understand the impact of $\mathcal{N}$, we set $\mathcal{N}$ to 10, 50, 100, 150, and 200. For each $\mathcal{N}$, we generate 10,000 programs. We then compute the average lines of code (LOC) across the programs, the time of generating 10,000 programs, and the line coverage of these programs.

Figure 13 shows the trend in LOC as $\mathcal{N}$ increases. There is a clear upward trend in LOC, with the maximum LOC exceeding 15,000 when $\mathcal{N}$ reaches 200. This is expected as a higher $\mathcal{N}$ leads to more functions being selected for synthesis. Figure 14 shows that more iterations require more time to generate programs, but the coverage does not increase much after $\mathcal{N}$ reaches 100. The main reason is that additional functions introduced by more iterations do not contribute meaningfully to the overall coverage, as most of the relevant code paths have already been explored. They may even hinder compiler optimization, as evidenced by the noticeable decline in GCC coverage. Therefore, for practical real-world fuzzing applications, setting $\mathcal{N}$ to 100 offers a good balance between a reasonable generation time and a high coverage.

5.4 RQ3: Comparison with Existing Tools

We compare LegoFuzz with Fuzz4All [38], WhiteFox [39], and Creal [19], three state-of-the-art compiler testing frameworks. While Fuzz4All and WhiteFox represent recent LLM-based approaches, Creal integrates real-world code with traditional fuzzing via Csmith.

Bug-finding analysis. We manually collected and analyzed all the reported GCC and LLVM bugs by Fuzz4All and WhiteFox. The results show that *neither Fuzz4All nor WhiteFox has discovered any miscompilation bugs*. In particular, Fuzz4All focused on g++, GCC's C++ compiler. In total, Fuzz4All discovered only eight confirmed bugs with half of them being fixed. Notably, all of them

are related to g++ frontend issues, rather than compiler optimizations. WhiteFox, on the other hand, has identified two bugs in LLVM — one is a backend-related crash, and another is a crash in error diagnostics. These findings suggest that both tools have very limited effectiveness in testing C compilers, as their discovered bugs do not touch the core compiler optimizations. Since Creal has extensively tested the GCC and LLVM, the fact that LegoFuzz discovered many long-latent bugs demonstrates the strong complementary bug-finding capability of Creal. We will discuss such cases in Section 5.5 to show the distinctive characteristics of discovered bugs.

Coverage analysis. We generate 10,000 test cases using Fuzz4All, WhiteFox and Creal for evaluation. Since WhiteFox is designed to generate C++ code, which is then converted into LLVM IR instead of directly targeting C, we only measure its LLVM coverage.

For LLM-based tools, as shown in Figure 12, Fuzz4All achieves significantly lower line coverage in GCC compared to LegoFuzz, with a coverage gap of over 50,000 lines of code. In LLVM, both tools exhibit even lower coverage than the raw function-level inputs used by LegoFuzz prior to the synthesis process, highlighting their limited effectiveness in covering compiler optimizations. For Creal ("Creal"), LegoFuzz still outperforms it substantially, demonstrating the advantage of our proposed unique methods. We further observe that using the same code database ("L+RealDB"), LegoFuzz still achieves higher coverage than Creal, highlighting the effectiveness of its iterative synthesis strategy.

Generation speed analysis. The time cost of generating 10,000 test programs for Fuzz4All and WhiteFox is 12,121 and 28,284 seconds, respectively. Compared to LegoFuzz's default configuration with $\mathcal{N} = 100$ (193 seconds), Fuzz4All and WhiteFox are considerably slower, taking 62 times and 146 times longer, respectively. Although all tools rely on LLMs, the offline and online decoupling design of LegoFuzz allows it to generate test programs at a much faster speed. We also include Creal in this comparison, which takes 13,997 seconds to generate the same number of programs. LegoFuzz eliminates the dependency on Csmith-generated seeds by directly composing functions from code database, which enables not only faster generation but also greater portability to languages without mature seed generators.

5.5 RQ4: Case Study

This section presents four cases to demonstrate that LegoFuzz can discover deep bugs in compilers. Note that all programs are reduced and simplified from the original ones for better readability.

Figure 15 (a): The original bug-triggering program generated by LegoFuzz contains 4,945 lines and 62 functions. This reduced program triggers a crash in GCC with the -O3 optimization flag. The issue occurs because GCC mistakenly classifies a reduction pattern in the outer loop when, in fact, the variable is not used outside the loop. Specifically, in line 16, the variable `c` is incremented within the inner loop, which GCC misinterprets as a reduction operation that spans both loops. This leads to an assertion failure during the vectorization pass, resulting in a compiler crash.

Figure 15 (b): The original bug-triggering program generated by LegoFuzz contains 6,697 lines and 91 functions. This reduced program triggers a miscompilation in LLVM due to an incorrect transformation performed by the InstCombine pass. The issue arises from an invalid optimization applied immediately after inlining the function `l()` into `m()`. Specifically, in line 5, the function `f()` computes a pointer offset using `g + *k`, where `g` is an array and `k` is an integer pointer. However, after inlining, LLVM incorrectly simplifies the `sext` operation, leading to an incorrect `getelementptr` (GEP) offset calculation in line 10. This results in a miscomputed pointer access, which causes undefined behavior when dereferencing `d` in line 6. This miscompilation causes the compiled binary to return 143 instead of 0. This program features a long and semantically meaningful function call chain starting from `m()`, which is rarely observed in Creal's bug cases. Such

```c
int a;
char b;
long c, d, e;
unsigned long f;
long g() {
  if (a <= 0)
    return 1;
  for (; d; d++) {
    e = 0;
    for (; e < a; e++) {
      unsigned long h = 0;
      switch (b)
      case 2:
        if (e)
          h = 5;
      c += h;
    }
  }
  c /= f;
}
```

(a) GCC -O3 crashes on this code. It ICEs on the unexpected stmt/SLP node arrangement.

```c
int printf(const char *, ...);
char a, b; int c; char *e = &b;
int f(char *g, int *k) {
  char *d = g + *k;
  for (; *d && *d <= ' '; d++) ;
  if (*d) return 0;
  return 1;
}
int l(int g) {
  char h[] = {a, a, a};
  int i[] = {g};
  int j = f(h, i);
  return j;
}
long m() {
  *e = 255;
  for (; l(b + 1);) return 0;
  for (;;) ;
}
int main() {  m(); printf("%d\n", c);}
```

(b) LLVM at -O3 miscompiles this code. The compiled binary returns 143 instead of 0.

```c
char a;
struct b {
  short c;
  char d;
  long e;
  int f;
} static g;
int h;
void i(struct b j) {
  char k;
  int l;
  for (; j.d; --j.d) {
    l = g.c == 0 ? 0 : 4294967295U % g.c;
    k = l >= 2 || a >> l ? 0 : l;
    h = k;
  }
}
void m() { i(g); }
```

(c) LLVM at -O2/3 crashes on this code. It triggers assertion failure in LoopVectorize.

```c
int printf(const char *, ...);
int a, c;
long b;
short d;
long e(long f, long h, long i) {
  for (long g = f; g <= h; g += i)
    b += g;
  return b;
}

int main() {
  c = 1;
  for (; c >= 0; c--)
    ;
  for (; e(d + 40, d + 76, c + 51) < 4;)
    ;
  printf("%X\n", a);
}
```

(d) GCC at -O2/3 miscompiles this code. The compiled binary times out instead of returning 0.

Fig. 15. Sample reduced programs that trigger compiler bugs.

patterns emerge naturally from LegoFuzz's unique iterative program synthesis, where complex interactions between function calls and global variables are progressively constructed.

Figure 15 (c): The original bug-triggering program generated by LegoFuzz contains 3,989 lines and 107 functions. This reduced program triggers a crash in LLVM. The bug occurs due to a discrepancy between the new VPlan cost model and the legacy cost model. Specifically, in line 12, the loop iterates when `j.d` is nonzero, decrementing `j.d` in each iteration. Inside the loop, in line 13, the expression `4294967295U % g.c` is computed, which results in an undefined behavior if `g.c` is zero. LLVM's loop vectorization planner attempts to determine the best vectorization factor, but during cost analysis, an inconsistency arises between the two cost models. This results in an assertion failure in `LoopVectorize` due to a mismatch in the expected vectorization behavior.

Figure 15 (d) The original bug-triggering program generated by LegoFuzz contains 6,485 lines and 24 functions. This reduced program triggers a miscompilation bug in GCC. The root cause is in the loop iteration analysis. Specifically, in line 7, the function `e()` accumulates values into the global variable `b` using a loop in steps of `i`. However, in line 15, the second loop in `main()` calls `e(d + 40, d + 76, c + 51)`, with `c` being decremented to `-1` in the previous loop. Since `c` influences the step size (`i` in `e()`), the compiler performs an invalid transformation of the loop bound, leading to an incorrect evaluation of the loop exit condition. GCC incorrectly optimizes the loop termination check, replacing a less-than comparison (`g <= h`) with a non-equal comparison (`g != h`), which leads to an unintended infinite loop depending on the initial values.

6 Discussion

➤ **Alternative prompting methods.** LegoFuzz adopts a flexible design that allows for prompt customization, enabling it to better align with specific models. As Section 5.3 indicates, the current prompt is particularly effective for models that excel at processing natural language instructions. However, it may not be as suitable for task-specific models, such as those specialized in code generation. By adjusting the prompt, we can optimize the transformation process to accommodate different types of language models and enhance their performance in generating diverse code structures. Better prompt design and better LLM models can improve the quality of the constructed code database, further enhancing the effectiveness of LegoFuzz.

➤ **More function signatures.** Our current approach leverages LLMs to generate numeric functions only. The core reason is to simplify the iterative synthesis process. Using numeric values for both function inputs and outputs and global variables eases our engineering effort in connecting different codes together. Supporting more types is theoretically doable, but it would require additional engineering efforts. Furthermore, even for numeric functions, the function bodies contain a much divergent range of types, such as strings and user-defined structs, and thus, the overall expressiveness of the generated programs is not affected.

➤ **Alternative sources of code snippets.** LegoFuzz shows, for the first time, that LLMs can be used to find deep compiler bugs. Given a constructed code database, the online synthesis component of LegoFuzz can generate testing programs. It is theoretically possible to use methods other than LLMs to construct the code database. For example, we can use historical bug-triggering programs [41] or real-world functions directly [19]. However, we argue that these sources are not as extensive as LLMs. Compared to real-world programs, historical bug-triggering programs are far less diverse. Directly using real-world programs is also constrained by their complexity and uncertainty. For example, Creal [19] only manages to collect fifty thousand functions from one million real-world programs. In comparison, we can generate over half a million functions. Therefore, coupled with LLMs, LegoFuzz can be more effective in discovering deep bugs.

➤ **Beyond function level synthesis.** LegoFuzz chains multiple functions together to form a testing program. One may wonder whether working at the granularity of function would lead to a loss of diversity in the generated programs. We argue that these programs are expressive enough to cover a wide range of compiler behaviors: (1) Our bug-finding results in Section 5 have already shown that LegoFuzz can discover deep bugs in various compiler optimizations, and (2) most compiler optimizations work at the function level [13, 23], ensuring that our testing programs can exercise a wide range of compiler optimizations. Extending LegoFuzz to support other levels of program synthesis is an interesting direction for future work and does not affect the core contribution of LegoFuzz. For example, when building dependencies between two functions, instead of generating a function call, we can directly merge one function body into another to form a larger function.

However, this is similar to the existing practice of inlining functions during compilation. Adding "__attribute__((always_inline))" directive to function definitions can also achieve a similar effect.

➤ **Beyond C compiler testing.** LegoFuzz describes a new paradigm of compiler testing, *i.e.*, using LLMs to generate building blocks first and thus use them to synthesize testing programs. This paper provides the first proof-of-concept implementation of this new paradigm on testing C compilers. Implementing LegoFuzz takes much less engineering effort than writing a program generator from scratch. This opens up many exciting opportunities for testing compilers for various programming languages, such as Rust, where a reliable and efficient program generator is not yet available.

7 Related Work

7.1 Generation-based compiler testing.

Significant efforts have been dedicated to developing automated program generators for compiler testing. Csmith [40] is one of the most widely used program generators for detecting C/C++ compiler bugs. It can generate a large number of test programs covering a broad subset of the C language while avoiding undefined behaviors, making it an essential tool for compiler validation. CLsmith [20] is inspired by Csmith, which is a program generator for OpenCL compilers and has six modes for generation. Morrisset *et al.* [24] extended Csmith by incorporating support for mutexes, atomic variables, and system calls for locking and unlocking mutexes, enabling the detection of C/C++ concurrency bugs. YARPGen [21] and its successor, YARPGen v2 [22], are modern program generators specifically designed to test scalar and loop optimization bugs.

While these tools have been effective in discovering numerous compiler bugs, they are rule-driven and may eventually reach a saturation point [1], where they struggle to uncover new bugs in a given compiler. This limitation arises from the predefined constraints embedded in these generators, which restrict their ability to explore certain aspects of compiler behavior. LegoFuzz addresses this limitation by leveraging real-world programs instead of relying solely on predefined generation rules. By adopting a data-driven rather than a rule-driven approach, LegoFuzz benefits from the rich expressiveness of real-world code, significantly expanding the search space and enhancing its ability to explore deeper compiler behaviors.

7.2 Mutation-based compiler testing.

Instead of generating a complete program from scratch, another line of research is to mutate parts of an existing test program. Some of the most effective tools maintain semantic equivalence during mutation by leveraging the concept of equivalence modulo inputs (EMI) [16]. For instance, Orion [16] and Athena [17] employ random and guided mutation strategies, respectively, primarily by inserting or deleting dead code blocks. In contrast, Hermes [33] focuses on mutating live code — sections that are actually executed. Meanwhile, GrayC [11] applies coverage-guided mutations to seed programs, which proves effective in detecting crash bugs, though it has not been successful in uncovering miscompilation bugs. Beyond semantics-preserving mutations, some approaches modify programs without maintaining their original semantics. For example, classfuzz [4] mutates class files using a diverse set of mutation operations to test JVM implementations. Similarly, LangFuzz [14] adopts a two-phase fuzzing strategy, *i.e.*, learning and mutation, to discover bugs in JavaScript interpreters. It reuses syntax-valid code fragments from seed programs, but performs static, grammar-based substitutions. In contrast, LegoFuzz applies LLM-based, context-aware transformations, enabling more expressive and adaptive mutations. Negai *et al.* propose a program generator [26] for random arithmetic expressions, which employs non-semantics-preserving mutations on the generated expressions. This work builds upon their earlier approach [25], which preserved semantics to avoid

introducing undefined behavior. In the later work, they enhance the generator by incorporating heuristics to produce more diverse expressions.

Similar to random test case generators, mutation-based compiler testing is also constrained by predefined mutation rules, which limit the diversity of generated test cases. These approaches often struggle to explore complex or unexpected program behaviors that may trigger deeper compiler bugs. To overcome this limitation, LegoFuzz incorporates iterative synthesis and LLM-based transformations to enhance mutation diversity. Instead of relying solely on fixed mutation rules, our framework leverages real-world programs as a foundation for mutation, ensuring a broader and more expressive test space. By utilizing LLMs, we can intelligently transform code at the syntax level while preserving its structural integrity. Through iterative rounds of mutation and synthesis, LegoFuzz continuously refines and expands the mutation space, significantly improving its ability to uncover deep-seated compiler bugs.

7.3 LLM-based Compiler Testing

With the rapid advancement of large language models (LLMs), leveraging LLMs for test case generation has emerged as a promising direction in compiler testing. Fuzz4All [38] is the first universal fuzzer that utilizes LLMs as both an input generator and a mutation engine, enabling the testing of widely used systems, including compilers. It employs a large *distillation LLM* to sample multiple candidate prompts, which are then passed to a *generation LLM* to produce diverse test cases. WhiteFox [39] is the first white-box compiler fuzzer that integrates LLMs with source-code analysis to test deep learning (DL) compiler optimizations. It adopts a multi-agent framework comprising three key components: *Requirement Summarization*, *Test Generation*, and *Feedback Loop*. Given the optimization pass source code from DL compilers, it analyzes the requirements necessary to trigger optimizations, generates corresponding test cases, and feeds valid cases back into the loop for continuous refinement.

Despite their effectiveness, these existing tools heavily center around LLMs, leading to multiple interactions per test case generation round, which significantly increases computational cost and makes them time-consuming compared to traditional testing approaches. In contrast, LegoFuzz adopts a hybrid offline/online mode to enhance reuse efficiency, reducing both the computational and financial burden. The generated test cases undergo rigorous validation to ensure high quality, making LegoFuzz a more efficient and scalable solution. Evaluation results in Section 5.4 demonstrate that LegoFuzz outperforms existing LLM-based frameworks.

8 Conclusion

We present LegoFuzz, an LLM-based compiler testing framework that decouples the compiler testing process into two synergistic phases: an offline phase and an online phase. In the offline phase, we leverage real-world code-aligned prompting to guide LLMs in generating small, feature-rich, and valid program snippets. These building blocks are then reused in the online phase to synthesize large and complex test programs through iterative program synthesis. Our evaluation demonstrates the effectiveness of this approach. LegoFuzz has uncovered 66 bugs in GCC and LLVM, most of which have already been fixed by compiler developers. Notably, nearly half of the reported bugs are miscompilation bugs, which cannot be detected by previous LLM-based tools. We believe that LegoFuzz opens up a new paradigm of compiler testing, and we are excited to see more research in this direction.

Data-Availability Statement

LegoFuzz is open-sourced at https://github.com/cuhk-s3/LegoFuzz. All the source code and data for reproducing the experimental results in this paper are available here [27].

References

[1] Domenico Amalfitano, Nicola Amatucci, Anna Rita Fasolino, Porfirio Tramontana, Emily Kowalczyk, and Atif M. Memon. 2015. Exploiting the saturation effect in automatic random testing of Android applications. In *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft '15)*. 33–43. doi:10.1109/MobileSoft.2015.11

[2] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS '20)*. Article 159. doi:10.18653/v1/2022.emnlp-main.90

[3] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *ACM Comput. Surv.* 53, 1, Article 4 (2020). doi:10.1145/3363562

[4] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. In *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI' 16)*. 85–99. doi:10.1145/2980983.2908095

[5] CompCert. 2025. The CompCert project. https://compcert.org/ Accessed: 2025-07-17.

[6] CppRference. 2025. C Language: Undefined Behavior. https://en.cppreference.com/w/c/language/behavior Accessed: 2025-03-17.

[7] Csmith-project. 2025. C-Reduce, a C and C++ program reducer. https://github.com/csmith-project/creduce Accessed: 2025-02-22.

[8] Anderson Faustino da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimarães, and Fernando Magno Quintão Pereira. 2021. AnghaBench: A Suite with One Million Compilable C Benchmarks for Code-Size Reduction. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO '21)*. 378–390. doi:10.1109/CGO51591.2021.9370322

[9] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2024. Large Language Models are Edge-Case Generators: Crafting Unusual Programs for Fuzzing Deep Learning Libraries. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Article 70. doi:10.1145/3597503.3623343

[10] Karine Even-Mendoza, Cristian Cadar, and Alastair F. Donaldson. 2022. CsmithEdge: more effective compiler testing by handling undefined behaviour less conservatively. *Empirical Softw. Engg.* 27, 6 (2022), 35 pages. doi:10.1007/s10664-022-10146-1

[11] Karine Even-Mendoza, Arindam Sharma, Alastair F. Donaldson, and Cristian Cadar. 2023. GrayC: Greybox Fuzzing of Compilers and Analysers for C. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23)*. 1219–1231. doi:10.1145/3597926.3598130

[12] Fastos. 2025. Fastsocket: a highly scalable socket and its underlying networking implementation of Linux kernel. https://github.com/fastos/fastsocket Accessed: 2025-03-17.

[13] GNU GCC. 2025. Passes and Files of the Compiler. https://gcc.gnu.org/onlinedocs/gccint/Passes.html.

[14] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium (Security'12)*. 38.

[15] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A Survey on Large Language Models for Code Generation. arXiv:2406.00515 [cs.CL] https://arxiv.org/abs/2406.00515

[16] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI' 14)*. 216–226. doi:10.1145/2666356.2594334

[17] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA' 15)*. 386–399. doi:10.1145/2858965.2814319

[18] Shaohua Li and Zhendong Su. 2024. UBFuzz: Finding Bugs in Sanitizer Implementations. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '24)*. 435–449. doi:10.1145/3617232.3624874

[19] Shaohua Li, Theodoros Theodoridis, and Zhendong Su. 2024. Boosting Compiler Testing by Injecting Real-World Code. *Proc. ACM Program. Lang.* 8, PLDI, Article 156 (2024). doi:10.1145/3656386

[20] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F Donaldson. 2015. Many-core compiler fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. 65–76. doi:10.1145/2813885.2737986

[21] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 196 (2020). doi:10.1145/3428264
[22] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2023. Fuzzing Loop Optimizations in Compilers for C++ and Data-Parallel Languages. *Proc. ACM Program. Lang.* 7, PLDI, Article 181 (2023). doi:10.1145/3591295
[23] LLVM. 2025. LLVM's Analysis and Transform Passes. https://llvm.org/docs/Passes.html.
[24] Robin Morisset, Pankaj Pawan, and Francesco Zappa Nardelli. 2013. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. 187–196. doi:10.1145/2491956.2491967
[25] Eriko Nagai, Hironobu Awazu, Nagisa Ishiura, and Naoya Takeda. 2012. Random testing of C compilers targeting arithmetic optimization. In *Workshop on Synthesis And System Integration of Mixed Information Technologies (SASIMI 2012)*. 48–53. https://api.semanticscholar.org/CorpusID:16947207
[26] Eriko Nagai, Atsushi Hashimoto, and Nagisa Ishiura. 2014. Reinforcing Random Testing of Arithmetic Optimization of C Compilers by Scaling up Size and Number of Expressions. *IPSJ Transactions on System and LSI Design Methodology* 7 (2014), 91–100. doi:10.2197/ipsjtsldm.7.91
[27] Yunbo Ni. 2025. Artifact for LegoFuzz – Interleaving Large Language Models for Compiler Testing. doi:10.5281/zenodo.15761520
[28] OpenAI. 2025. OpenAI API Reference: max_completion_tokens. https://platform.openai.com/docs/api-reference/runs/createRun#runs-createrun-max_completion_tokens Accessed: 2025-03-19.
[29] OpenRouter. 2025. GPT-3.5 Turbo. https://openrouter.ai/openai/gpt-3.5-turbo Accessed: 2025-03-12.
[30] OpenRouter. 2025. Qwen2.5 Coder 32B Instruct. https://openrouter.ai/qwen/qwen-2.5-coder-32b-instruct Accessed: 2025-03-12.
[31] Kosta Serebryany. 2016. Continuous Fuzzing with libFuzzer and AddressSanitizer. In *2016 IEEE Cybersecurity Development (SecDev)*. 157–157. doi:10.1109/SecDev.2016.043
[32] Kostya Serebryany. 2016. Sanitize, Fuzz, and Harden Your C++ Code.
[33] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN international conference on object-oriented programming, systems, languages, and applications (OOPSLA' 16)*. 849–863. doi:10.1145/3022671.2984038
[34] Haoxin Tu, Jiang He, Xiaochen Li, Zhilei Ren, Zhide Zhou, and Lingxiao Jiang. 2022. Remgen: Remanufacturing a Random Program Generator for Compiler Testing. In *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE '22)*. 529–540. doi:10.1109/ISSRE55969.2022.00057
[35] Rui Ueyama. 2025. 8cc: A Tiny C Compiler. https://github.com/rui314/8cc Accessed: 2025-02-17.
[36] Krister Walfridsson. 2025. smtgcc: Some experiments with SMT solvers and GIMPLE IR. https://github.com/kristerw/smtgcc/ Accessed: 2025-03-13.
[37] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems (NeurIPS '22)*. Article 1800.
[38] 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 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. 1–13. doi:10.1145/3597503.3639121
[39] 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 (2024). doi:10.1145/3689736
[40] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. 283–294. doi:10.1145/1993498.1993532
[41] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal program enumeration for rigorous compiler testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '17)*. 347–361. doi:10.1145/3140587.3062379
[42] 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), 110884. doi:10.1016/j.jss.2020.110884
[43] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: A Survey for Roadmap. *ACM Comput. Surv.* 54, 11s, Article 230 (2022). doi:10.1145/3512345

Received 2025-03-25; accepted 2025-08-12