The Mutators Reloaded: Fuzzing Compilers with Large Language Model Generated Mutation Operators

Xianfei Ou
State Key Laboratory for
Novel Software Technology,
Nanjing University
Nanjing, China
ouxianfei@smail.nju.edu.cn

Cong Li
Ant Group & School of
Cyber Science and Tech.,
Zhejiang University
Hangzhou, China
chifei.lc@antgroup.com

Yanyan Jiang
State Key Laboratory for
Novel Software Technology,
Nanjing University
Nanjing, China
jyy@nju.edu.cn

Chang Xu
State Key Laboratory for
Novel Software Technology,
Nanjing University
Nanjing, China
changxu@nju.edu.cn

Abstract

Crafting high-quality mutators–the core of mutation-based fuzzing that shapes the search space–is challenging. It requires human expertise and creativity, and their implementation demands knowledge of compiler internals. This paper presents MetaMut framework for developing new, useful mutators for compiler fuzzing. It integrates our compiler-domain knowledge into prompts and processes that can best harness the capabilities of a large language model. With MetaMut, we have successfully created 118 semantic-aware mutators at approximately $0.5 each, with only moderate human effort. With these mutators, our fuzzer uncovered 131 bugs in GCC and Clang, 129 of which were confirmed or fixed. The success of MetaMut suggests that the integration of AI into software and system engineering tasks traditionally thought to require expert human intervention could be a promising research direction.

ACM Reference Format:
Xianfei Ou, Cong Li, Yanyan Jiang, and Chang Xu. 2024. The Mutators Reloaded: Fuzzing Compilers with Large Language Model Generated Mutation Operators. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3622781.3674171

1 Introduction

Compiler fuzzing has gained significant attention over the past decades [9–11, 32, 35, 36, 45, 53, 56]. By generating a multitude of test programs, these fuzzers uncovered thousands of critical bugs in production compilers such as GCC and LLVM, substantially enhancing their quality [10, 11].

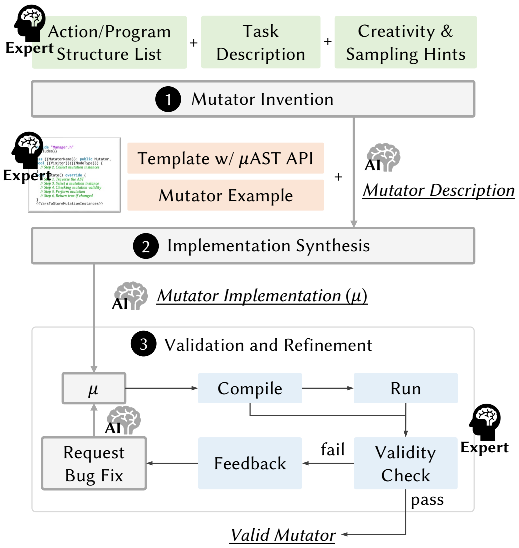

Figure 1. The METAMUT framework. We provided the prompts, code templates, auxiliary libraries, and validation scripts (colored boxes). An LLM takes care of all rest tedious work of mutator invention and implementation synthesis (gray boxes in bold). Underlined items are created by LLM.

Mutation-based search is a mainstream approach to compiler fuzzing. In mutation-based fuzzing, mutation operators, or *mutators*, define the shape of the search space[1] and thus determine the potential and limitations of the technique. Effective mutators must be aware of both the program's syntax and semantic structure to produce valid and diverse mutants that can reach deeper behaviors of compiler components, such as IR generation, program analysis, and optimization.

However, crafting high-quality mutators is challenging. Design and implementation of effective mutators require

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0391-1/24/04...$15.00
https://doi.org/10.1145/3622781.3674171

[1] Mutators determine each test program's all possible descendants (mutants). Beginning with a set of seed programs, applying mutators broadens this set into an expanded array of test programs, thereby creating a search space (a graph where each vertex corresponds to a test program). Fuzzers adopt a breadth-first-like search strategy to sample test programs in this graph.

expertise of compiler internals. As a result, the creation of semantic-aware mutators is a pivotal factor in generating substantial high-quality compiler test cases [18], and typically, existing techniques only present a few new mutators. Therefore, the quest is:

How to systematically design and implement a spectrum of useful program mutators with moderate, or even negligible, cost?

This paper provides the first positive response to this question, leveraging recent advancements in large language models (LLMs) which are shown to be effective in a range of programming-related tasks, including program comprehension, refactoring, and synthesis.

The challenge is that inventing new mutators and synthesizing mutator implementations end-to-end are beyond the capability of today's LLMs. These tasks are laborious even for experienced researchers and developers with expertise in compiler internals and compiler fuzzing.

The MetaMut Framework. This paper introduces MetaMut for developing new, useful mutators for compiler fuzzing. We break down the mutator generation problem into three stages as depicted in Figure 1, and we integrate our compiler domain knowledge into prompts and processes that can best harness an LLM's capabilities:

1. Mutator invention, where we guide an LLM to generate natural-language mutator names and descriptions, based on a list of actions and program structures.
2. Implementation synthesis, where the LLM fills a carefully crafted mutator template (with auxiliary instructions and an in-context learning example) in one shot to produce a tentative mutator implementation.
3. Validation and refinement, where any synthesized mutator is compiled, applied to a set of LLM-generated unit tests, and checked for validity. Error messages are fed back to the LLM for correction and refinement.

Following this workflow, we obtained 118 valid[2] semantic-aware mutators. The first set of 68 "supervised" mutators resulted from approximately two weeks of manual prompt engineering and refinement. After consolidating all prompts, a second set of 50 "unsupervised" mutators were generated by invoking MetaMut 100 times without any human intervention. Beyond our manual efforts, each mutator costs ~$0.5 for invoking GPT-4 APIs.

When integrating these mutators into a simple coverage-guided fuzzer, both set of mutators (supervised and unsupervised) outperform state-of-the-art fuzzers AFL++ [20], Csmith [53], YARPGen [35, 36], and GrayC [18] in terms of both code coverage and unique crashes. In a broader span of experiments, we uncovered 131 bugs in the latest versions of GCC and Clang, where 129 were already confirmed and 83 were found in the middle-end or back-end compiler modules.

Novelty and Contributions. This paper adopts a fundamentally different approach from existing LLM-empowered fuzzers such as Fuzz4All [50], FuzzGPT [15], and White-Fox [52], which underutilize the power of LLMs by only employing them as test case generators. We demonstrate that LLMs, being correctly instructed, can be effective in synthesizing non-trivial software artifacts like mutators. This offers a new, distinct perspective in compiler fuzzing that complements existing work [15, 18, 50, 52]. In summary, this paper makes the following contributions:

- We introduce MetaMut, a framework that leverages large language models for automatically devising non-trivial mutator designs and implementations.
- We create 118 executable semantic-aware mutators, at a cost of ~$0.5 each and with moderate human effort.
- We demonstrate the effectiveness of both MetaMut and the generated mutators by uncovering 131 GCC/Clang bugs, with 129 confirmed or fixed.

MetaMut and the generated mutators (descriptions and implementations) are available via https://icsnju.github.io/MetaMut/.

2 MetaMut and Illustrative Example

MetaMut decomposes the challenging mutator generation problem into three sub-tasks (steps) as illustrated in Figure 1: mutator invention, implementation synthesis, and validation and refinement. This section provides a walkthrough of the mutator generation process, during which we illustrate a previously unknown bug in Clang-17 uncovered by MetaMut-invented mutator.

❶ **Mutator Invention.** Every mutator, denoting a small-step modification of the source code, naturally has a brief natural-language description. Conversely, in theory, one could enumerate all conceivable natural language sentences that describe mutators (within a length bound), and for each description *synthesize* a corresponding mutator implementation. To approximate this procedure, we implicitly define a *probability space* of potentially useful natural-language mutator descriptions, aiming to perform effective mutations to a program's control- and data-flow structures, via the following prompt to LLMs:

> Give me the name and a brief description of a semantic-aware mutation operator that performs **[Action]** on **[Program Structure]**, where both the action and the program structure are selected from the list below[3]:

Actions	Add, Modify, Copy, Swap, Inline, Destruct, Group, Combine, Lift, Switch, Inverse...
Program Struct.	BinaryOperator, LogicalExpr, CharLiteral, IfStmt, Attribute, Builtins, ArrayDimension...

[2] A mutator is considered valid if it consistently conforms to its name and description across all (unit) test cases.

```
#include "Mutator.h"
#include "Manager.h"
{{Includes}}

class {{MutatorName}}: public Mutator, public ASTVisitor {
  bool {{Visitor}}({{NodeType}}) {
    // Step 2, Collect mutation instances
  }
  bool mutate() override {
    // Step 1, Traverse the AST
    // Step 3, Select a mutation instance
    // Step 4, Check mutation validity
    // Step 5, Perform mutation
    // Step 6, Return true if changed
  }
  {{VarsToStoreMutationInstances}}
};

static RegisterMutator<{{MutatorName}}>
  M("{{MutatorName}}", "{{MutatorDescription}}")
```

Figure 2. The mutator template. The steps and the "`{{...}}`"-wrapped tokens are to be completed by LLM.

MetaMut invents new mutators by sampling from this probability space. The complete list of actions and program structures is included directly in the prompt. Repeatedly invoking an LLM with a high temperature setting effectively performs a beam-search-like procedure [26], sampling possible mutator descriptions within the probability space. An example of a mutator invented by GPT-4 is:

> **ModifyFunctionReturnTypeToVoid**: *Change a function's return type to void, remove all return statements, and replace all uses of the function's result with a default value.*

The emergent and associative capabilities of LLMs enable the creation of mutators that are not confined to the predefined list of actions and program structures. For instance, "return type of void" is a relevant mutator description that was not included in our list, yet GPT-4 appears to have "invented" it.

❷ **Implementation Synthesis.** LLMs can generate plausible mutator names and descriptions like "*ModifyFunctionReturnTypeToVoid*" (or *Ret2V*). Yet, manually implementing these mutators is notoriously challenging due to the complexity of AST traversal and modification, particularly when interacting with the intricate compiler-internal APIs, e.g., the Clang AST APIs. For example, *Ret2V* involves traversing and removing all `return` sites within the function. Even though manual implementations work, it is difficult to scale: creating hundreds of mutators would be extremely labor-intensive.

To automate the synthesis process, we again utilize LLMs. Directly synthesizing mutators like *Ret2V* is currently beyond the capabilities of today's LLMs. Alternatively, we provide a sketched program template and step-by-step instructions to the LLM using the chain-of-thought prompting method [30, 48]. Given the mutator description and template as shown in Figure 2, GPT-4 generates a tentative implementation of *Ret2V*, depicted in Figure 3. This implementation randomly selects a function with a non-void return type and changes its return type to `void`.

```
class ModifyFunctionReturnTypeToVoid : ... {
  std::vector<ReturnStmt *> TheReturns;
  std::vector<CallExpr *> TheCalls;
};

bool ModifyFunctionReturnTypeToVoid::mutate() {
  TraverseAST(getASTContext());
  if (TheFunctions.empty()) return false;

  FunctionDecl *func = randElement(TheFunctions);

  // Change the return type to void
  QualType voidType = getASTContext().VoidTy;
  std::string voidTypeStr = formatAsDecl(voidType, "");

  SourceRange typeRange = func->
    getReturnTypeSourceRange();
  getRewriter().ReplaceText(typeRange, voidTypeStr);

  // Remove all return statements
  for (ReturnStmt *retStmt : TheReturns) {
    getRewriter().ReplaceText(
      retStmt->getSourceRange(), "");
  }
  return true;
}
```

Figure 3. GPT-4's first implementation of *Ret2V*. It appears "good" and compiles.

❸ **Validation and Refinement.** LLMs do not always produce valid mutators. Each synthesized mutator is validated against an LLM-generated test suite of compilable and executable C programs P. MetaMut asserts that any mutant of the test suite, like $P' = Ret2V(P)$, should be compilable. Any compilation errors in P', or crashes/hangs during mutation are automatically reported back to the LLM for bug-fixing until all issues in the mutator like *Ret2V* are resolved.

The initial *Ret2V* implementation in Figure 3 fails to rewrite function calls to `func`, e.g., `int x = foo();`. After changing `foo` to a void function, the assignment statement would no longer compile. Figure 4 shows GPT-4's successful correction after two rounds of feedback.

Mutators and Compiler Fuzzing. After establishing the workflow in Figure 1, we manually interacted with GPT-4 to generate mutators, analyzed and debugged the generated mutants, and refined these prompts. Over approximately two weeks, we obtained 68 valid (consistent with generated natural-language descriptions) "supervised" mutators. Subsequently, with all prompts consolidated, we initiated an independent, fully automated mutator generation process that invoked the workflow 100 times, resulting in 50 valid "unsupervised" mutants. The cost of generating each mutant using the LLM is approximately \$0.50.

These mutators define a new search space for compiler validation. Armed with these mutators, we successfully identified 129 bugs in GCC and Clang. One such bug is uncovered by applying *Ret2V* to the GCC test suite #20001226-1. The

[3] The prompt and the list have been simplified for brevity. The complete prompt, actions, and program structures are provided in our repository.

```
1     class ModifyFunctionReturnTypeToVoid : ... {
2   -   vector<ReturnStmt *> TheReturns;
3   -   vector<CallExpr *> TheCalls;
4   +   map<FunctionDecl *, vector<ReturnStmt *>> FuncReturns;
5   +   map<FunctionDecl *, vector<CallExpr *>> FuncCalls;
6     };
7
8     bool ModifyFunctionReturnTypeToVoid::mutate() {
9       TraverseAST(getASTContext());
10      if (TheFunctions.empty()) return false;
11
12      FunctionDecl *func = randElement(TheFunctions);
13
14      // Change the return type to void
15      QualType voidType = getASTContext().VoidTy;
16      std::string voidTypeStr = formatAsDecl(voidType, "");
17
18      SourceRange typeRange =
19        func->getReturnTypeSourceRange();
20      getRewriter().ReplaceText(typeRange, voidTypeStr);
21
22      // Remove all return statements
23  -   for (ReturnStmt *retStmt : TheReturns) {
24  +   for (ReturnStmt *retStmt : FuncReturns[func]) {
25        getRewriter().ReplaceText(
26          retStmt->getSourceRange(), "");
27      }
28
29  +   // Replace all calls
30  +   for (CallExpr *call : FuncCalls[func]) {
31  +     QualType returnType = func->getReturnType();
32  +     std::string replaceText =
33  +       returnType->isFloatingType() ? "0.0" : "0";
34  +     getRewriter().ReplaceText(
35  +       call->getSourceRange(), replaceText);
36  +   }
37      return true;
38    }
```

Figure 4. GPT-4 fixed version of *Ret2V*. "Replace all calls" in Lines 29–36 rewrites all calls to `func` with a constant. Even if the function is no longer called at runtime after *Ret2V*, `func` is still compiled and can uncover potential compiler bugs.

```
- unsigned foo(int x[64], int y[64]) {
+ void foo(int x[64], int y[64]) {
    C4096(x,y); // 4,096 jumps
-   return 0x01234567;
+   ;
  gt:
-   return 0x12345678;
+   ;
  lt:
-   return 0xF0123456;
+   ;
  }
```

Figure 5. Clang #63762: A previously unknown bug in Clang-17 discovered by MetaMut. Applying the *Ret2V* mutator to GCC test suite #20001226-1 changes the function return-type to void and removes all return statements.

mutant in Figure 5 triggers a Clang internal assertion error affecting Clang ≥ 17. This error is triggered when there is no computational logic between `C4096(x,y)` and the labels `gt` and `lt`. Removing all `return` statements creates the precise condition for manifesting this bug.

Discussions. MetaMut could be regarded as a successful showcase of "*AI-expert co-design*". While compiler testing experts have recognized the significance of effective mutators in bug-finding [18, 32, 34, 53], a massive implementation of them was impractical before the advent of LLMs. MetaMut united such domain-expertise with the logical reasoning and program synthesis capabilities of LLMs, facilitating both the automated design and creation of mutators.

Given the extensive design space of *all possible mutators*, we acknowledge that MetaMut is neither sound (guaranteed to synthesize correct implementations conforming to descriptions) nor complete (guaranteed to generate all possible mutators). Yet, it is extremely *useful*. LLMs, when effectively prompted and refined via an expert-designed process, can be powerful in assisting, or even taking place of human experts in handling domain-specific programming tasks.

3 MetaMut and Fuzzer Implementation

This section expands Section 2 with implementation details.

3.1 Mutator Invention

Recognizing that mutators essentially revises the structure of ASTs, we crafted the following prompt template to focus the probability space for mutator invention:

> A semantic-aware mutation operator that performs **[Action]** on **[Program Structure]**.

To obtain useful natural-language descriptions that conform to this template, we further prompt the LLM with clear and specific instructions:

1. *Task description.* We provide a direct list of actions and program structures within the prompt. The **[Action]** list is derived from the member functions of the Clang AST and IR APIs. The **[Program Structure]** list covers all Clang AST node types. LLMs can then select an action and a program structure from the list to create a mutator. For example, a mutator that adjusts array dimensions could be created by pairing the action "Modify" with the program structure "ArrayDimension".
2. *Creativity hints.* We encourage LLMs to explore actions and program structures that are related to, but not limited to, those listed. *ModifyReturnTypeToVoid* in Section 2 is an example of such "creative thinking".
3. *Sampling hints.* We include a list of previously generated mutator names and descriptions in the prompt and instruct the LLM to avoid creating duplicates. This approach biases the search towards new mutators, leading to a more efficient sampling process.

The complete prompt and the LLM's responses are provided in our repository.

3.2 Implementation Synthesis

Despite being with a precise natural-language description, mutator implementation is challenging, even for developers experienced with Clang's internals. End-to-end mutator synthesis is currently beyond the capabilities of today's LLMs.

```
class Mutator {
  // ---------- Query APIs ----------
  template<T> StrRef getSourceText(T *node); // Extracts the source code of a tree node for replication at new locations
  SrcLoc findStrLocFrom(SrcLoc loc, StrRef target); // Locates the position of a string starting from a specified location
  SrcRng findBracesRange(SrcLoc from); // Identifies the range of the latest pair of enclosed braces
  template<T> T &randElement(vector<T> &elements); // Choose random element

  // ---------- Rewriting APIs ----------
  bool removeParmFromFuncDecl(ParmVarDecl *PV); // Removes a parameter from function declaration
  bool removeArgFromExpr(Expr *E); // Removes an argument from a function invocation

  // ---------- Semantic checking APIs ----------
  bool checkBinop(int op, Expr *lhs, Expr *rhs); // Checks if operator op can be applied to lhs and rhs
  bool checkAssignment(SrcLoc loc, Type lhsTy, Type rhsTy); // Checks if an expression can be replaced by another

  // ---------- Helpers ----------
  Str generateUniqueName(Str baseName); // Generates a unique identifier for a new variable/function/type
  Str formatAsDecl(Type ty, Str placeholder); // Formats a given type and identifier as a variable declaration
};
```

Figure 6. μAST API examples.

We reduce the synthesis difficulty for the LLM with two complementary strategies:

1. We encapsulate Clang AST APIs into a set of simplified APIs, which we refer to as μAST. These APIs have more straightforward arguments, are more naturally readable, and are thus more easily managed by language models.
2. We restrict program synthesis to completing a code *template*, where the mutator implementation is broken down into discrete steps.

μAST APIs. To simplify the synthesis task for LLMs, we encapsulated Clang AST APIs into higher-level μAST APIs, with *readability* as a primary design goal, as shown in Figure 6. μAST APIs provide mechanisms for AST traversal, node retrieval and rewriting, and semantic checks. For example, simply removing a function parameter's declaration node is insufficient to fully eliminate the parameter; one must also remove the trailing comma. μAST APIs provide `removeParamFromFuncDecl` for this purpose. Some mutations may not be universally applicable; for instance, changing an addition operation (+) to a multiplication (*) could result in an error if the operand types do not support multiplication. μAST APIs include a function like `checkBinop` to verify the validity of such mutations. Additionally, μAST nodes offer functions like `randElement` for randomly selecting AST elements of a specified type, which would otherwise require extensive coding using Clang AST APIs.

All μAST APIs are encapsulated within the `Mutator` class, which is the parent class for all synthesized mutators. MetaMut employs the few-shot in-context learning paradigm [4] to instruct LLMs on how to progressively complete the template. This is achieved through a complete mutator example, including its name, description, and implementation, which adheres to the template as per its description. The declarations of the `Mutator` and `ASTVisitor` classes, along with helper functions, are provided in `Mutator.h` and are included in the prompt to LLMs.

Template-Based Synthesis. We guide LLMs to synthesize mutator implementations following a predefined code template, as depicted in Figure 2. The template includes placeholders such as `{{Includes}}` and `{{Visitor}}`, along with instructions for completing sub-tasks (notated as "Step x" in the comments). Although the synthesis process is still one-shot, this chain-of-thought prompting method [48] provides clear spatial cues about the structure of the synthesized mutator, resulting in a high success rate: nearly half of the mutators are correct on the first attempt, and many others can be automatically corrected during the refinement loop.

3.3 Validation and Refinement

There is no guarantee that a mutator implementation by an LLM (or even a human expert) is correct. MetaMut includes a validation step to identify errors in the implementation, provides explanations to the LLM, and requests corrections.

Validation Goals. For a mutator μ and a test program P, we apply μ to produce a mutant $P' = \mu(P)$. First, μ must be compilable. Then, executing $\mu(P)$ should neither hang nor crash. Finally, P' should be a valid program, and whenever the program structure targeted by μ is present in P, it should be appropriately modified in P'. To this end, MetaMut prompts the LLM to generate test programs P that specifically contain the program structure targeted by the mutator μ:

> Generate test cases for which the mutator **[Name]** and **[Description]** can be applied.

We found that LLMs are capable of generating compilable code snippets that include the specified program structure.

Refinement Loop. Using these generated test programs, MetaMut implements a domain-specific, chain-of-thought process for self-driven refinement. The refinement procedure breaks down the validation goal into sub-goals, ranging from the simplest (#1) to the most complex (#6):

#	Validation Goal	Unmet Feedback
1	μ compiles	*<error message>*
2	μ terminates (not hang)	P *<stack trace>*
3	μ returns (not crash)	*<stack trace>*
4	μ outputs something	P
5	μ changes something	P
6	μ creates compilable mutant	P P' *<error message>*

Given a mutator implementation μ, MetaMut:

Algorithm 1: The μCFuzz micro fuzzer.

```
function $\mu$CFuzz(Seed Programs $\mathcal{S}$, Mutators $M$, Compiler $C$)
    $\mathcal{P} \leftarrow \mathcal{S}$
    repeat
        $P \leftarrow$ random_choice($\mathcal{P}$)
        $M' \leftarrow$ random_shuffle($M$)
        for $\mu \in M'$ do
            $P' \leftarrow \mu(P)$ // apply mutator $\mu$ to obtain a mutant
            if br_cover$_C(P') \not\subseteq \bigcup_{P \in \mathcal{P}}$ br_cover$_C(P)$ then
                $\mathcal{P} \leftarrow \mathcal{P} \cup \{P'\}$ // $P'$ covers a new branch
                break
    until timeout
```

1. Verifies whether all goals are met by compiling μ using Clang, running μ and obtaining $P' = \mu(P)$, and checking P', from #1 to #6.
2. Provides the LLM with feedback on the unmet, simplest goal to obtain a corrected implementation; or returns this likely valid mutator[4] if it satisfies all goals.

For mutators that cannot be automatically corrected after many iterations, MetaMut halts the automated process and (optionally) requests human intervention. An expert can then either diagnose the root cause of the failure and guide the LLM towards a resolution, or provide a direct fix.

3.4 Micro and Macro Coverage-Guided Fuzzers

All mutators are integrated into a micro coverage-guided C fuzzer, μCFuzz. As illustrated in Algorithm 1, given a set of seed programs, μCFuzz in each iteration attempts to apply a mutator μ (selected in random order) to an existing program P. If $P' = \mu(P)$ uncovers new code paths, it is added back into the pool for further mutations. Despite lacking advanced optimizations such as Havoc, mopt, fork-based execution, or seed pool culling found in mature fuzzers, μCFuzz significantly outperforms AFL++ [20], GrayC [18], Csmith [53], and YARPGen [35, 36] in terms of both code coverage and unique crashes, as demonstrated in Section 5.2.

In addition to μCFuzz, we also developed a macro fuzzer, on the same basis of Algorithm 1, which incorporates several engineering improvements specifically tailored for long-term bug-hunting. Major enhancements include:

1. Random sampling of compiler command-line arguments.
2. Implementing the Havoc strategy [49], which involves multiple rounds of mutation to enhance the diversity of the generated mutants;
3. Maintaining a shared coverage map across processes to facilitate parallel coverage-guided fuzzing;
4. Limiting resource usages, thus preventing system-wide performance issues, such as memory shortages caused by out-of-memory conditions due to compiler bugs.

4 Generated Mutators

We generate mutators following the workflow of Figure 1:

1. Supervised mutators M_s: An author interacted with GPT-4 using a set of tentative prompts. These prompts were refined as needed to generate compelling mutator descriptions and valid mutator implementations. If the LLM failed to correct a mutator during the validation-refinement step, the author manually debugged, corrected it, and identified limitations to refine the prompts further. Additionally, the author addressed bugs and made minor revisions to the design of the μAST APIs.
2. Unsupervised mutators M_u: With the prompts and μAST implementation consolidated, the fully automatic MetaMut was executed 100 times without human intervention. There was no information leakage between the supervised and unsupervised cycles, except for the prompts. We consider both system errors (such as API throttle errors) and validation failures as unsuccessful cases.

Two authors of this paper independently verified all generated mutators. A mutator was considered valid and was added to M_s or M_u only if its implementation performs as described across all test cases (the authors added new test cases whenever necessary). Any conflicting case was reviewed until a consensus was reached. All generated mutators are available in our repository.

4.1 Generated Mutators

Overview. The process described above yielded $|M_\text{s}| = 68$ supervised mutators and $|M_\text{u}| = 50$ unsupervised mutators. All mutators in M_s are confirmed as valid, as they were manually analyzed and corrected as necessary. During the 100 MetaMut invocations for generating M_u, 24 failed due to unexpected issues such as API throttling or timeouts. Of the remaining 76 mutators, $|M_\text{u}| = 50$ (65.8%) are valid. Additionally, with the improved stability of APIs and the enhanced capabilities of language models, we expect our framework to produce an increased number of valid mutators.

The 118 valid mutators can be classified into five categories based on their target structures: Variable (16), Expression (50), Statement (27), Function (19), and Type (6) Mutators. Expression Mutators were the most common, accounting for 42%, while Type Mutators were the least common at 5%.

It is also interesting that there is only limited overlap between the mutators generated under supervised and unsupervised settings. We identified only six pairs of mutators (approximately 10%) perform similar actions on similar program structures, such as "*ModifyIntegerLiteral*" and "*ReplaceLiteralWithRandomValue*". This suggests that LLMs have the potential to generate a broad range of mutators.

[4] The validation goals are necessary but not sufficient conditions which ensure the mutator to perform the described behavior on P. Therefore, we further manually checked all likely valid mutators.

Examples. Below provide some examples, with names and descriptions copied verbatim from the LLM's outputs:

> **DuplicateBranch** (M_s): "*This mutator finds an IfStmt, duplicates one of its branches (then or else), and replaces the other branch with the duplicated one*".

> **SwitchInitExpr** (M_s): "*This mutator randomly selects a VarDecl and swaps its init expression with the init expression of another randomly selected VarDecl in the same scope, while ensuring the types of the variables are compatible*".

> **InverseUnaryOperator** (M_s): "*This mutator selects a unary operation (like unary minus or logical not) and inverses it. For instance,* `-a` *would become* `-(-a)` *and* `!a` *would become* `!!a`".

Sometimes, LLMs can also provide a relevant mutator that is not strictly adhere to the template "perform [Action] on [Program Structure]". GPT-4 drove MetaMut to generate 33 such "creative" mutators, such as *Ret2V* (as aforementioned), *SimpleUninliner*, and *TransformSwitchToIfElse*:

> **SimpleUninliner** (M_s): "*Turn a block of code into a function call*".

> **TransformSwitchToIfElse** (M_u): "*This mutator identifies a 'switch' statement in the code and transforms it into an equivalent series of 'if-else' statements, effectively altering the control flow structure*".

In addition to mutating a single program structure,such mutators typically involve multiple distinct program structures and should often maintain certain semantic relationships before and after the mutation.

Automatic Validation and Refinement. 27 out of 50 (54%) mutators in M_u were invalid prior to refinement. The LLM fixed a total of 107 bugs in these tentative mutator implementations, and the refinement loop managed to fix an average of 3.96 bugs for each valid mutator. As shown in Table 1, the most common bugs were "mutator not compiling" (51.4%), followed by "creating compile-error mutants" (33.6%). The mutator generation logs, including the chat history between MetaMut and GPT-4, are available in our repository.

We have also categorized the failure cases for the 26 invalid mutators, whose implementations are not included in M_u. Specifically, 6 generated mutators (23.1%) did not survive the refinement loop (validation goals #1–#6). For the remaining 20 likely correct mutators[5], the causes of failure are:

- *Mismatched implementation.* 7 mutators (26.9%) do not align with their descriptions. Notably, the synthesized *InverseUnaryOperator* implementation incorrectly transforms `-a` into `-(--a)`. We considered it invalid, although it might still be useful in compiler fuzzing.
- *Unthorough test cases.* Although mutants may survive the automatic refinement loop, 10 generated mutators (38.5%) produced compile-error mutants when tested against more complex tests crafted by the authors.
- *Duplicate.* Despite instructions to the LLM to avoid creating duplicates, 3 mutators (11.5%) were still duplicates of previously generated ones.

Table 1. Classification of bugs fixed by MetaMut's validation-refinement loop for M_u. Bug categories are the same as the validation goals #1–#6 in Section 3.

#	Validation Goal's Violations	Fixed (#)
1	μ not compile	55
2	μ hangs	0
3	μ crashes	4
4	μ outputs nothing	11
5	μ does not rewrite	1
6	μ creates compile-error mutant	36

Although it is unfortunate that the LLM failed to provide a correct *InverseUnaryOperator* implementation, rectifying this should not be a significant challenge for a compiler expert. We remain optimistic that future LLMs will be more effective in both generating correct implementations of mutators and thorough test cases.

4.2 Human Labor and Generation Costs

Our manual efforts in implementing MetaMut for C include:

- *Supervision.* It took approximately two weeks for one of the authors to refine the prompts used in generating M_s and to correct any invalid mutator implementations.
- *μAST APIs.* The implementation of μAST APIs on top of Clang AST APIs took about two days for an author.

We believe that this manual effort is relatively moderate compared to manually implementing hundreds of mutators from scratch. We anticipate that the manual labor involved in adapting MetaMut to a new programming language will not be significantly greater than this.

The LLM token costs of MetaMut are summarized in Table 2. On average, our framework consumed 8,600 tokens with 6 QA rounds to generate each semantic-aware mutator, corresponding to ~US$0.5 when using OpenAI's ChatCompletion APIs (or ~$0.17 using `gpt-4-turbo`). This process involved two fixed QA rounds for mutator invention and implementation, costing ~1,200 and ~2,500 tokens, respectively. To validate the executable mutator, the bug-fixing loop required ~4 QA rounds and approximately 4,900 tokens on average. It took MetaMut < 6 minutes to generate a compilable and valid semantic-aware mutator, with about 50% of mutators generated within 3.5 minutes. The generation time ranged from less than 1.5 minutes to ~30 minutes.

[5] Although we excluded them from M_u, they may still be useful in generating mutants for compiler fuzzing.

Table 2. Generation cost of one mutator: "Tokens" is the consumed tokens; "QA" is the number of rounds that MetaMut interacted with GPT-4; "Time" is in seconds.

Metrics	Steps	Min	Max	Median	Mean
Tokens	Invention	359	2,240	1,130	1,158
	Implementation	372	3,870	2,488	2,501
	Bug-Fixing	335	30,923	2,077	4,935
	Total	3,214	35,312	6,054	8,595
QA	Bug-Fixing	1	23	2.0	4.0
	Total	3	25	4.0	6.0
Time	Invention	11	21	15	15
	Implementation	14	101	49	49
	Bug-Fixing	29	1,876	130	281
	Total	83	1,949	189	346

Table 3. Request/response time of a single mutator.

	Min	Max	Median	Mean
Wait for Response (s)	11	123	46	43
Prepare for Request (s)	0	69	9	17

Given MetaMut's workflow, it is also understandable that the majority of the time (81.2%) was spent on bug fixing.

Table 3 shows the time spent awaiting a response and preparing a request during mutator generation. The mean wait time is about 43 seconds. Request preparation, which involves mutator compilation and execution, and feedback information collection, averaged around 17 seconds.

Compared with developing a mutator from scratch by a human expert, we believe that such a cost is quite affordable and practical. We are also optimistic that the rapid development of faster and more capable LLMs will further improve and reduce the cost of mutator generation.

5 Evaluation

We explore the following two research questions:

RQ1 How does a compiler fuzzer, equipped with MetaMut-generated semantic-aware mutators, compare to state-of-the-art compiler fuzzers?

RQ2 Can a compiler fuzzer using MetaMut's semantic-aware mutators uncover new bugs in widely-used, production-level compilers?

5.1 Evaluation Setups

Answering RQ1. We integrate the supervised and unsupervised mutators, M_s and M_u, into μCFuzz, resulting in two variants: μCFuzz.s and μCFuzz.u. We compare their performance against four state-of-the-art techniques: the generation-based Csmith [53] and YARPGen [35, 36], the mutation-based GrayC [18], and the generic fuzzer AFL++ [20]. All fuzzers, except for Csmith and YARPGen which operate without seeds, are initialized with the same set of 1,839 seed inputs derived from the test suites of GCC and Clang–two extensively tested, production C compilers. We evaluate all six fuzzers on GCC-14 and Clang-18, all with the `-O2` option. The following experiments are conducted:

Coverage and Crashes: The first experiment focuses on two critical metrics for evaluating a fuzzer: code coverage and unique crashes. For each fuzzer, we deploy 60 parallel instances, each assigned to a separate CPU, to run for 24 hours. This setup results in fuzzing in a total of

$$2 \text{ (compilers)} \times 6 \text{ (fuzzers)} \times 60 \text{ (CPUs)} = 720$$

CPU days (or 17,280 CPU hours). We specifically measure the branch coverage. A crash is uniquely identified by its top two stack frames (including program counter), with helper functions like `llvm::report_error` being excluded.

Compilable Mutants: The other experiment compares the ratio of compilable mutants generated by each fuzzer to study the semantic awareness of our mutators. We run all six fuzzers for 24 hours and repeat the process ten times. All generated mutants are recorded, and the average ratio of compilable mutants is calculated.

Answering RQ2. Over an eight-month period, we conduct a field experiment to assess the bug-finding capabilities of our macro fuzzer, which integrates both M_s and M_u. We test the most recent releases of GCC and Clang: GCC-12, GCC-14, Clang-17, and Clang-18. We use the same seed set as in RQ1 to bootstrap the fuzzer.

Configurations. Both supervised and unsupervised mutator generation employ GPT-4 as the LLM, with a temperature setting of 0.8 and a top percentage of 0.95. The automatic fix procedure is terminated if it could not provide a validation-passing mutator after 27 repair attempts. All experiments are conducted on a DELL PowerEdge R6515 Server running Ubuntu 22.04, equipped with a 64-core (128-thread) AMD EPYC 7713P processor and 128 GiB of memory.

5.2 RQ1: Comparison with Existing Fuzzers

Code Coverage. Figure 7 illustrates the average coverage trends of μCFuzz, Csmith, YARPGen, GrayC, and AFL++ across GCC and Clang. Overall, supervised μCFuzz.s outperforms unsupervised μCFuzz.u by a margin of 2% in coverage improvements. μCFuzz.u shows coverage enhancements of 5.4% and 6.1% over the best results from Csmith, YARPGen, GrayC, and AFL++ on GCC and Clang, respectively. GrayC outperforms AFL++, Csmith, and YARPGen for both GCC and Clang, achieved with only five carefully designed semantic-aware mutators[6]. These observations may indicate that, when appropriately prompted, large language models (LLMs) have the potential to generate mutators with a diversity that match those crafted by human experts.

[6] We obtained the number via `./grayc --list-mutations`.

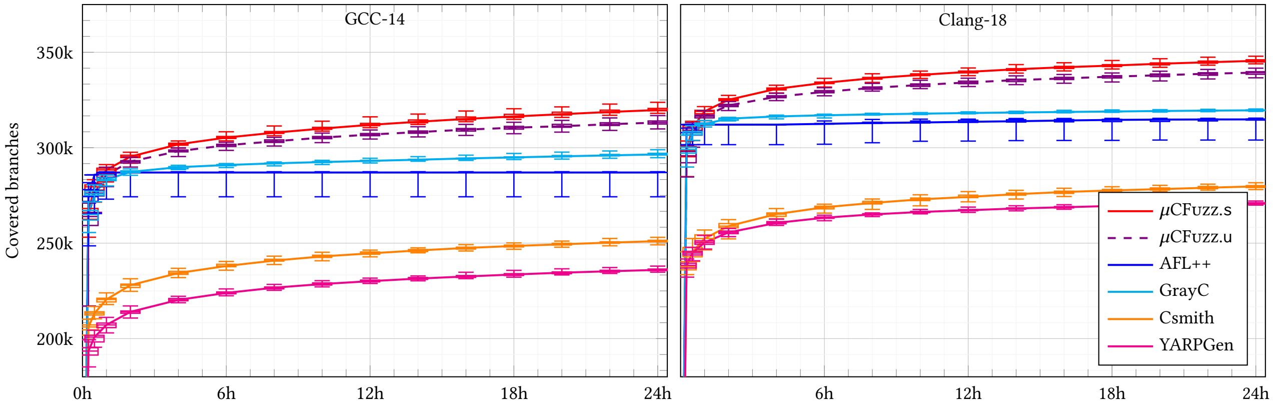

Figure 7. Coverage trends of each fuzzer for GCC-14 and Clang-18. Both μCFuzz.s and μCFuzz.u outperform existing fuzzers, with supervised mutators perform better.

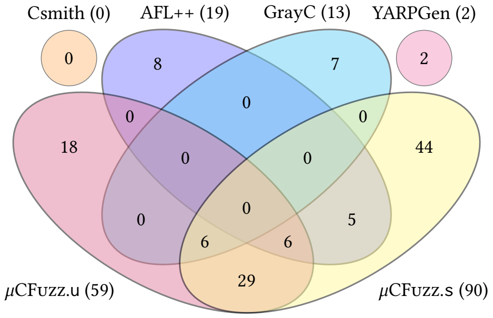

Figure 8. Venn diagram of discovered unique crashes.

AFL++ also significantly outperforms Csmith and YARPGen in terms of code coverage, likely due to its extensive engineering and implementation-level optimizations, including techniques like Havoc. Furthermore, AFL++ produced a substantial number of non-compilable mutants, which are expected to cover more error-handling code within the compiler front-end module.

Unique Crashes. The evaluated techniques have identified a total of 125 unique crashes, as illustrated in Figure 8. Both μCFuzz.s and μCFuzz.u significantly outperform the best results of GrayC, AFL++, Csmith, and YARPGen, detecting three times more number of crashes.

Of these crashes, μCFuzz.s and μCFuzz.u accounted for 86.4% (108 crashes), while AFL++, GrayC, Csmith, and YARPGen identified 15.2% (19 crashes), 10.4% (13 crashes), 0% (0 crashes), and 1.6% (2 crashes) respectively. Notably, μCFuzz exclusively reported 72.8% (91 out of 125) of the unique crashes, which is 2.6 times the number reported by the combined efforts of the other four techniques (34 out of 125).

μCFuzz.s, benefiting from human expertise[7], identified 53% more unique crashes than μCFuzz.u. AFL++ and GrayC each detected at least 13 unique crashes, whereas Csmith did not found any crashes on GCC-14 and Clang-18, despite being allocated 1,440 CPU hours. This outcome aligns with findings from [35], which suggest that Csmith may have reached a saturation point in testing production compilers. YARPGen found two unique crashes in GCC and Clang. This can be attributed to YARPGen's specific design focus on exploring loop misoptimizations [36], rather than targeting general compiler bugs.

Trends of unique crash counts over time for both GCC and Clang are plotted in Figure 9. Interestingly, although GrayC surpasses AFL++ in terms of code coverage for GCC, AFL++ identified significantly more unique crashes (15 versus 2). This is likely because the generic fuzzers like AFL++ explores more front-end program paths. Most (11 of 15) GCC crashes found by AFL++ are from the front-end.

Crash Distributions. We further classify all unique crashes by their appearing compiler components. As displayed in Table 4, both μCFuzz.u and μCFuzz.s were able to trigger compiler crashes from compiler front-end to IR generation, optimization, and compiler back-end, and both of them discovered more unique crashes in all compiler components compared to the other tools.

None of the six tools except for GrayC, YARPGen and μCFuzz could cause compiler's optimization or back-end modules to crash, where μCFuzz.s and μCFuzz.u detected 2.6 and 5 times more unique crashes in these modules, respectively. μCFuzz is also the only tool that were able to make compiler back-end crash. We believe that this highlights the semantic-awareness and effectiveness of MetaMut-generated mutators. Though AFL++ found more unique crashes than GrayC, 79% were in the compiler front-end.

Crash Cases. The semantic-aware mutators generated by MetaMut are capable of steering the mutant to exhibit

[7] Recall that we manually fixed any errors in the mutator implementations in generating M_S.

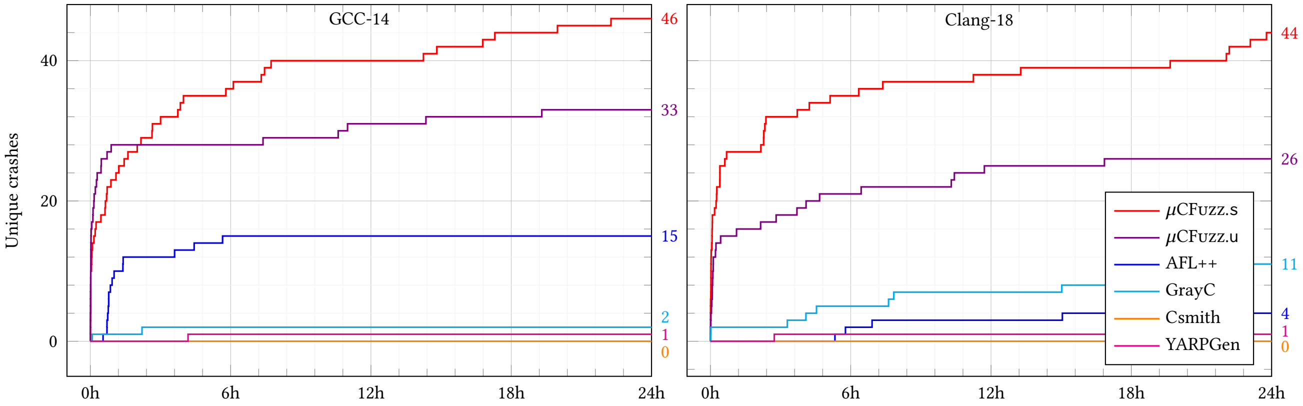

Figure 9. Unique crashes found by the evaluated fuzzer in GCC-14 and Clang-18. For each fuzzer, the plot reports each unique crash's earliest discovery time.

Table 4. Overview of the found unique crashes, where "IR" stands for IR generation and "Opt" represents Optimization.

	Front-End	IR	Opt	Back-End	Total
AFL++	15	4	0	0	19
GrayC	5	3	5	0	13
Csmith	0	0	0	0	0
YARPGen	0	0	2	0	0
μCFuzz.u	15	26	10	8	59
μCFuzz.s	24	31	24	11	90

Table 5. The number of compilable test programs generated within the 24-hour fuzzing run (averaged over 10 runs).

Tool	Compilable (#)	Total (#)	Ratio (%)
AFL++	76,075	2,154,621	3.53
GrayC	973,178	983,078	98.99
Csmith	31,338	31,381	99.86
YARPGen	75,658	75,785	99.83
μCFuzz.u	770,658	1,070,368	72.00
μCFuzz.s	723,776	972,002	74.46

non-trivial control and data flows, thereby revealing compiler bugs. An example of a mutant identified exclusively by μCFuzz.s, and not by the other tools, is:

```
  ...
- static char buffer[32];
+ char const volatile buffer[32];
- int test4() { return sprintf(buffer,"%s","bar"); }
+ int test4() { return sprintf(buffer,"%s",buffer); }
  void main_test (void) {
    ...
    memset (buffer, 'A', 32);
    if (test4 () != 3) abort ();
    ...
  }
```

This mutant activates GCC's `strlen` optimization, which optimizes the return value `sprintf(...)` to `strlen(...)`. However, `buffer` is not NULL-terminated, triggering the compiler to create an invalid memory range, which is subsequently captured by an assertion failure in `verify_range`.

Our semantic-aware mutators *ChangeVarDeclQualifier* and *CopyExpr* were consecutively applied to mark `buffer` as `const` and to substitute `"bar"` with `buffer`. AFL++, Csmith, and YARPGen are not likely to generate this kind of programs, which require mutating a seed to define a `const char` array and `sprintf` it to itself. For GrayC, we did not find any of its mutators having the ability to modify neither qualifiers nor function arguments.

Finally, we also noticed that μCFuzz.s and μCFuzz.u did not detect 17 (13.6%) unique crashes identified by either Csmith, YARPGen, GrayC, or AFL++. These cases can be attributed primarily to two factors: (1) MetaMut is designed for generating semantic-aware mutators, which allocates less focus to the compiler front-end compared to AFL++; (2) Certain mutators from GrayC, such as *InjectControlFlow*, are out of the probability space defined by MetaMut, i.e., "perform [Action] on [Program Structure]." Despite these limitations, μCFuzz successfully identified 27 front-end crashes and outperform the other tools in detecting more crashes across all evaluated subjects and compiler modules.

Compilable Mutants. As shown in Table 5, Csmith, YARPGen, and GrayC generated the highest proportion of compilable mutants among these tools. This is expected because they strictly followed the syntax and semantics of C during the development of their semantic-aware mutators. In contrast, a majority of the mutants produced by AFL++ fail to compile because AFL++ treats programs as byte arrays without semantic awareness. Over 70% of the mutants generated by MetaMut are compilable. This is ratio approximately 30% more compared to existing approaches that leverage LLMs for end-to-end compiler test program generation [50].

Table 6. Overview of the reported compiler bugs.

	Clang	GCC	Total
Reported	81	50	131
	Numbers of reported compiler bugs		
Confirmed	81	48	129
Fixed	18	17	35
Duplicate	5	8	13
	Numbers of affected compiler modules		
Front-End	32	16	48
IR Generation	27	18	45
Optimization	8	14	22
Back-End	14	2	16
	Numbers of bugs by their consequences		
Segmentation Fault	3	6	9
Assertion Failure	71	40	111
Hang	7	4	11

In terms of throughput, μCFuzz achieved a rate of approximately 11 mutants per second, comparable to the rates of GrayC and AFL++, and it significantly outperforms Csmith and YARPGen. This throughput is also substantially higher than that achieved by end-to-end generation using LLMs.

5.3 RQ2: Bug-Hunting Capability

Bugs Found. Finally, we conducted an eight-month-long field experiment by integrating all 118 mutators into our macro fuzzer to fuzz the latest versions of GCC and Clang.

With M_s and M_u, our macro fuzzer uncovered a total of 131 bugs within the two compilers, of which 129 have been confirmed or fixed (Table 6). Specifically, a bug is categorized as "Confirmed" if the compiler developers are able to reproduce it in their environments. If not, it remains in "Reported", even if we have an evident crash log for reproduction and diagnosis. We also identified 13 bugs that were duplicates of previously reported issues, suggesting that our mutators that the others may encounter. These were "Duplicate."

Among the 48 confirmed bugs reported to GCC, which follows GNU's development workflow for bug diagnosis and repair, 19 (39.6%) are assigned priority ≥ P2 by the developers, including three at P1 (non-duplicate and fixed). Additionally, 19 bugs affect two or more major versions, with 14 impacting three or more. Fixed and duplicated bugs constitute 52.1% of all confirmed bugs in GCC. For Clang, there are over 20,000 open non-question issues on GitHub; the progress in fixing these issues is generally slower than that in GCC. Yet, we observe that developers are progressively addressing reported bugs in both GCC and Clang.

The 131 bugs affect a broad spectrum of compiler modules, with front-end being the most impacted (36.6%). Notably, over 63.4% of them passed the front-end modules which perform syntax and semantic checks, highlighting the semantic awareness of our mutators. 16 bugs are uncovered in the back-end modules, indicating that certain mutants produced by our mutators are capable of reaching deep compiler modules and activating rarely exercised code paths.

The majority (85%) of the bugs triggered the compiler's internal assertion violations, with 7% causing segmentation faults and 8% leading to hangs. These assertion violations, which manifest as "crashes," indicate internal inconsistencies and are often attributable to logical errors in the compiler rather than simple programming errors like buffer overflow. Below are a few such bug cases:

GCC #111820. GCC-14 hangs when compiling the following mutant with the `-O3 -fno-tree-vrp` options:

```
- int r[6];
+ int r;
+ int r_0;

- void f(int n) {
+ void f() {
+   int n = 0;
    while (--n) {
-     r[0] += r[5];
+     r_0 += r;
-     r[1] += r[0]; r[2] += r[1]; r[3] += r[2];
-     r[4] += r[3]; r[5] += r[4];
+     r += r; r += r; r += r; r += r; r += r;
    }
  }
```

This mutant results from applying three mutators:

- *ChangeParamScope*, for moving the parameter `n` from the parameter scope to the local scope of function `f` and initializing `n` with 0.
- *AggregateMemberToScalarVariable*, for transforming the first array subscript expression `r[0]` into a scalar variable `r_0` and adding a declaration for it.
- *ReduceArrayDimension*, for simplifying array `r[6]` into a zero-dimension scalar `r` and updating its references.

This bug led to extensive discussions on GCC's Bugzilla, and the developers eventually localized the bug in the loop vectorizer. *ChangeParamScope* is critical for this bug. By transferring the scope of `n`, the subsequent **`while`** loop changes from an indefinite, non-evaluable state to a definite, evaluable state at compile time. The other two mutations create the necessary conditions for the loop vectorizer to be enabled. However, the loop vectorizer freezes because it miscalculates the number of iterations for the loop that starts at zero and decreases indefinitely towards negative infinity.

GCC #111819. GCC-14 crashes on assertion failure when compiling below mutant with the default GCC options:

```
  _Complex double x;
+ long long combinedVar_1;
  int *bar(void) {
-   return (int *)&__imag x;
+   return (int *)&__imag (*(_Complex double *)(
+     (char *)&combinedVar_1 + 16));
  }
```

The bug is triggered by sequentially applying the `__imag` and `&` operators to a variable cast as `_Complex` **`double`**, involving approximately 16 rounds of mutations during our fuzzing campaign. We eventually narrowed it down to two critical mutations–*CombineVariable* and *DecaySmallStruct*–identified as required to trigger the assertion failure[8]. The *DecaySmallStruct* mutator casts the struct into a `long long` variable and changes all references to `x` into a pointer arithmetic between the `long long` variable and some offsets. This creates an expression that applies `&` and `__imag` to a `_Complex double` -cast variable, triggering the crash. The bug root cause is developer's oversight in not addressing specific cases within `fold_offsetof`.

Clang #69213. Clang-18 crashes when compiling the following mutant with the default Clang options:

```
  struct s2 {...};
- foo (struct s2 *ptr) {
+ foo (int *ptr) {
-   *ptr = (struct s2) {{}, 0};
+   *ptr = (int) {{}, 0};
  }
```

The test case has been minimized to include only the essential code and mutation sites necessary to trigger the bug. This bug was discovered by the *StructToInt* mutator, which changes the type `struct s2` to `int`, causing Clang to access a non-exist AST node.

All these bug cases demonstrate the value and significance of the MetaMut framework and its generated mutators.

5.4 Discussions

Implications. In summary, MetaMut has successfully generated 118 executable and valid semantic-aware mutators at a reasonable and practical generation cost (Section 4.2). These mutators enabled our micro compiler fuzzer, μCFuzz, to achieve higher code coverage and to find more unique crashes compared to state-of-the-art fuzzers, while maintaining comparable throughput to GrayC and AFL++ (Section 5.2). The effectiveness of these mutators is also evident given the 131 bugs uncovered in GCC and Clang (Section 5.3).

Therefore, we believe that this paper presents a new approach to integrating human domain-expert knowledge into the complex task of mutator design and implementation. This is why we consider our framework to operate at a "meta" level: instead of directly asking LLMs to generate mutants–which is costly and time-consuming[9], MetaMut redefines a novel search space shaped by the LLM-generated mutators.

Finally, MetaMut "amplifies" the capability of LLMs in generating mutators, which are complex software artifacts that even compiler experts find challenging to implement. Given that current LLMs are not yet capable of autonomously inventing and synthesizing mutators end-to-end, this development suggests a promising future where LLMs could significantly reduce human effort in the extensive engineering of complex systems and software artifacts.

Porting MetaMut to Other Languages. For individuals aiming to port MetaMut to a new programming language, it is feasible to directly use our refined prompts. The porting also requires the following major implementation efforts concerning μAST APIs (Figure 6):

- *AST Parsing and Rewriting*: The most widely used parser for C/C++ today, capable of handling GNU C/C++ extensions, remains the Clang AST APIs. For other programming languages, such as Java, developers can employ tools like ANTLR [42] and tree-sitter [5], which offer simpler APIs compared to those of Clang AST.
- *Semantic Queries and Checks* (e.g., retrieving the callee of a function call): Developers can repurpose existing analyses (e.g., indexers and linters) implemented using the Language Server Protocol [6]. Furthermore, these analyses do not need to be perfectly precise–a conservative approach may still yield useful mutators, albeit at the expense of generating more invalid mutants.

Technical challenges may also involve dealing with language-specific features like annotations, macros, and type systems.

Limitations. Our mutator description template constrains that only one action is performed on a program structure, which most invented mutators adhere to. However, human-designed mutators [18] could potentially be more "creative". Exploring the design space of our template further is a promising direction for future research. We also identify three primary limitations of MetaMut:

1. The context length of LLMs may be a bottleneck, potentially limiting the synthesis and refinement capabilities.
2. LLMs fall short in providing correct fixes for complex bugs, such as those causing "Mutator Hangs", as observed in the invalid mutators.
3. Our test-based validation goals are not sound and cannot guarantee validity for all generated mutators.

However, these limitations are tied to the current capabilities of LLMs. Given the rapid advancement of LLMs, we expect these limitations to become less significant over time.

Threats to Validity. The first threat to validity is the potential bias in the manual validation of generated mutators. To mitigate this, the two authors worked independently and only considered a mutator valid if they reached a consensus. Another concern is the heavy reliance of both μCFuzz and MetaMut on libclang, a library potentially familiar to LLMs due to its presence in training data. Nevertheless, we do not consider this a major threat because one can fine-tune LLMs to enhance its capability on handling a specific library.

6 Related Work

Generation-Based Fuzzing. These techniques generate programs from scratch [3, 7, 11, 17, 22, 37]. Csmith randomly selects a production rule and recursively expands the rule to

[8] We present a simplified mutant in this paper. Merely applying the two mutators to the original seed will not cause a crash. Prior rounds of mutations create necessary preconditions for these two mutators to come into play.

[9] We estimate that a similar scale of fuzzing campaign would cost $1,000,000 using GPT-4 API for end-to-end test input generation.

create a new program, with a special focus on avoiding undefined behaviors [53]. CCG produces chaotic and random C programs for compiler crashes [39]. YARPGen addressed the saturation issue faced by Csmith and other program generators [35]. It built different generation policies for different compiler components like the loop optimizer [36]. There are also program generators for Java compilers [8] and JVMs [46], Rust compilers [43], JavaScript engines [25], FPGA synthesis tools [24], and polyglot generators [23].

All of them have demonstrated notable effectiveness in fuzzing compilers. Different from them, we generate new programs by mutating an existing pool of real-world, seed programs which is known to be rich of language features that a generation-based fuzzer must model.

Mutation-Based Fuzzing. This is also a mainstream approach to compiler validation. Orion introduces EMI, which randomly eliminates dead branches to generate programs with identical runtime behavior [32]. Athena removes dead branches or inserts code into unexecuted regions via Bayesian optimization [33]. Hermes injects dead code snippets into live code regions [45]. SPE focuses on enumerating variable usage patterns in skeleton-based test synthesis [56]. GrayC designs five semantic-aware mutators specifically for creating more compilable programs [18].

Mutation-based fuzzing is also widely applied to other (just-in-time) compilers. Classfuzz mutates Java class files randomly [12], while dexfuzz targets Android dex files [31]. Classming disrupts the control and data flow of live bytecode [13]. JAttack extracts program skeletons and populates them with synthesized expressions [55]. Artemis develops semantic-aware mutators to influence JVM interpretation and compilation behaviors [34]. CodeAlchemist disassembles JavaScript programs into code bricks and reassembles them to create new mutants [21].

All these heavily depend on manually crafted mutators, requiring significant human expertise, creativity, and are labor-intensive. In contrast, our AI-expert co-designed framework, MetaMut, can automatically generate a wide range of mutators with moderate costs (~$0.5) and human effort.

Mutation Testing. Mutation testing can be used to assess the thoroughness of a test suite by perturbing (mutating) the program under test and examining if the suite can detect these mutations [28, 41]. Mutators in mutation testing mainly focus on exploring boundary cases (e.g., boundary or faulty values [1, 14]) that may not be covered by the test suite. These mutators typically only involve simple single-point code modifications. In contrast, mutators for compiler fuzzing explores into deeper aspects of programming language specifications, features, and even program semantics, often requiring multi-point modifications. Such mutators are useful in triggering different IR-generation/optimization behaviors but may not be as effective in mutation testing because the resulting mutants are likely to be killed by even trivial test cases. MetaMut may also be potentially useful in mutation testing by generating mutators that explore boundary program behaviors.

LLM-Empowered Testing and Validation. LLMs have recently gained prominence for their versatility, finding applications across various research domains [2, 19, 38, 47, 51, 54]. In the context of compiler fuzzing, LLMs have shown significant potential due to their capability to understand and synthesize programs [4, 27]. WhiteFox exploits LLM to analyze the source code of a compiler and produce corresponding requirements for testing programs; it then leverages LLM to generate compiler test cases over previously analyzed requirements, which are encoded in the prompt [52]. The auto-prompting methodology proposed by Fuzz4All enables LLMs to generate a variety of test programs for various programming languages automatically from their documents and specifications [50]. In the broader context of testing and validation, TitanFuzz leverages LLMs to generate API calls [15]; FuzzGPT primes LLMs to synthesize atypical programs from historical bug-triggering programs [16]; SLNET focuses on generating Simulink models [44]; LIBRO generates test programs from bug reports [29]; LLM4VV validates OpenACC by LLM-generated programs [40]. Unlike these techniques that employ LLMs as test generators, MetaMut operates at a meta-level. Once the mutators are established, no further neural network inference is required, making our approach more efficient, controllable, and economical.

7 Conclusion

This paper introduces MetaMut, a framework that harnesses both human expertise and the power of LLMs to enhance compiler fuzzing through the streamlined generation of useful semantic-aware mutators. The encouraging experimental results highlight MetaMut as a useful tool for improving compiler reliability, and suggests that integrating AI into software and system engineering is feasible, even for tasks once believed to require human expertise exclusively.

Acknowledgments

We are grateful to the anonymous reviewers and our shepherd, Yongle Zhang, for their valuable feedback on earlier versions of this paper. This work was supported in part by National Key R&D Program (Grant #2022YFB4501801) of China, National Natural Science Foundation of China (Grants #61932021, #62025202, #62272218), the Leading-edge Technology Program of Jiangsu Natural Science Foundation (Grant #BK20202001), Cooperation Fund of Huawei-Nanjing University Next Generation Programming Innovation Lab (YBN2019105178SW22), and the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China. Yanyan Jiang (jyy@nju.edu.cn), the corresponding author, was supported by the Xiaomi Foundation.

A Artifact Appendix

A.1 Abstract

This artifact consists of:

1. The scripts that drive the mutator invention, implementation synthesis, and the validation and refinement loop.
2. The coverage-guided fuzzers that leverage the generated mutators to produce descendants.
3. A full list of generated mutators and uncovered bug cases.

A.2 Artifact check-list (meta-information)

- **Runtime Environment:** Linux with Docker.
- **Hardware:** x86-64 PC or server.
- **Output:** A set of semantic-aware mutators, which can be used by fuzzers to uncover compiler bugs. The fuzzers are also included in the artifact.
- **Publicly available?:** Yes.

A.3 Description

A.3.1 How to access.

MetaMut is available at:

https://icsnju.github.io/MetaMut/.

We also provide a Docker image that can reproduce our experimental results, available at https://zenodo.org/records/11473356. This includes generating mutators (Section 4), the comparative experiment (Section 5.2), and field bug hunting (Section 5.3).

A.4 Installation

The Docker image does not require installation.

A.5 Evaluation and expected results

Please refer to the website (or archive) for more information.

References

[1] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2016.

[2] Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models. *CoRR*, abs/2108.07732, 2021.

[3] Abdulazeez S Boujarwah and Kassem Saleh. Compiler test case generation methods: A survey and assessment. *Information and Software Technology*, 39, 1997.

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D 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 Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, NeurIPS '20, 2020.

[5] Max Brunsfeld. Tree-sitter, 2021. URL https://tree-sitter.github.io/tree-sitter/.

[6] Hendrik Bünder. Decoupling language and editor-the impact of the language server protocol on textual domain-specific languages. In *International Conference on Model-Based Software and Systems Engineering*, MODELSWARD '19, 2019.

[7] Augusto Celentano, S Crespi Reghizzi, P Della Vigna, Carlo Ghezzi, G Granata, and Florencia Savoretti. Compiler testing using a sentence generator. *Software: Practice and Experience*, 10, 1980.

[8] Stefanos Chaliasos, Thodoris Sotiropoulos, Diomidis Spinellis, Arthur Gervais, Benjamin Livshits, and Dimitris Mitropoulos. Finding typing compiler bugs. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI '22, 2022.

[9] Junjie Chen and Chenyao Suo. Boosting compiler testing via compiler optimization exploration. *ACM Transactions on Software Engineering and Methodology*, 31, 2022. doi: 10.1145/3508362.

[10] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. An empirical comparison of compiler testing techniques. In *International Conference on Software Engineering*, ICSE '16, 2016. doi: 10.1145/2884781.2884878.

[11] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. A survey of compiler testing. *ACM Computing Surveys*, 53, 2020. doi: 10.1145/3363562.

[12] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. Coverage-directed differential testing of JVM implementations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, 2016. doi: 10.1145/2908080.2908095.

[13] Yuting Chen, Ting Su, and Zhendong Su. Deep differential testing of jvm implementations. In *IEEE/ACM International Conference on Software Engineering*, ICSE '19, 2019. doi: 10.1109/ICSE.2019.00127.

[14] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. PIT: A practical mutation testing tool for Java. In *International Symposium on Software Testing and Analysis*, ISSTA '16, 2016.

[15] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '23, 2023. doi: 10.1145/3597926.3598067.

[16] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. Large language models are edge-case fuzzers: Testing deep learning libraries via FuzzGPT. In *IEEE/ACM International Conference on Software Engineering*, ICSE '24, 2024.

[17] Arthur G Duncan and John S Hutchison. Using attributed grammars to test designs and implementations. In *International Conference on Software Engineering*, ICSE '81, 1981.

[18] Karine Even-Mendoza, Arindam Sharma, Alastair F. Donaldson, and Cristian Cadar. GrayC: Greybox fuzzing of compilers and analysers for C. In *International Symposium on Software Testing and Analysis*, ISSTA '23, 2023. doi: 10.1145/3597926.3598130.

[19] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. Automated repair of programs from large language models. In *IEEE/ACM International Conference on Software Engineering*, ICSE '23, 2023.

[20] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *USENIX Workshop on Offensive Technologies*, WOOT '20, 2020.

[21] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. CodeAlchemist: Semantics-aware code generation to find vulnerabilities in JavaScript engines. In *Annual Network and Distributed System Security Symposium*, NDSS '19, 2019.

[22] Kenneth V. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 9, 1970.

[23] William Hatch, Pierce Darragh, Sorawee Porncharoenwase, Guy Watson, and Eric Eide. Generating conforming programs with Xsmith. In *ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE '23, 2023.

[24] Yann Herklotz and John Wickerson. Finding and understanding bugs in FPGA synthesis tools. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA ’20, 2020. doi: 10.1145/3373087.3375310.

[25] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *USENIX Security Symposium*, SEC ’12, 2012.

[26] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *International Conference on Learning Representations*, ICLR ’20, 2020.

[27] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. Jigsaw: Large language models meet program synthesis. In *International Conference on Software Engineering*, ICSE ’22, 2022. doi: 10.1145/3510003.3510203.

[28] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37, 2011. doi: 10.1109/TSE.2010.62.

[29] Sungmin Kang, Juyeon Yoon, and Shin Yoo. Large language models are few-shot testers: Exploring LLM-based general bug reproduction. In *International Conference on Software Engineering*, ICSE ’23, 2023.

[30] Takeshi Kojima, Shixiang (Shane) Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. In *Conference on Neural Information Processing Systems*, NeurIPS ’22, 2022.

[31] Stephen Kyle, Hugh Leather, Björn Franke, Dave Butcher, and Stuart Monteith. Application of domain-aware binary fuzzing to aid android virtual machine testing. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE ’15, 2015. doi: 10.1145/2731186.2731198.

[32] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *ACM-SIGPLAN Symposium on Programming Language Design and Implementation*, PLDI ’14, 2014. doi: 10.1145/2594291.2594334.

[33] Vu Le, Chengnian Sun, and Zhendong Su. Finding deep compiler bugs via guided stochastic program mutation. In *ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA ’15, 2015. doi: 10.1145/2858965.2814319.

[34] Cong Li, Yanyan Jiang, Chang Xu, and Zhendong Su. Validating JIT compilers via compilation space exploration. In *Symposium on Operating Systems Principles*, SOSP ’23, 2023. doi: 10.1145/3600006.3613140.

[35] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random testing for C and C++ compilers with YARPGen. *ACM on Programming Languages*, 4(OOPSLA), 2020. doi: 10.1145/3428264.

[36] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Fuzzing loop optimizations in compilers for C++ and data-parallel languages. *ACM on Programming Languages*, 7(PLDI), 2023. doi: 10.1145/3591295.

[37] Peter M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, 7, 1990.

[38] Bonan Min, Hayley Ross, Elior Sulem, Amir Pouran Ben Veyseh, Thien Huu Nguyen, Oscar Sainz, Eneko Agirre, Ilana Heintz, and Dan Roth. Recent advances in natural language processing via large pre-trained language models: A survey. *ACM Computing Surveys*, 56, 2023. doi: 10.1145/3605943.

[39] Mrktn. CCG, 2013. URL https://github.com/Mrktn/ccg/.

[40] Christian Munley, Aaron Jarmusch, and Sunita Chandrasekaran. LLM4VV: Developing LLM-driven testsuite for compiler validation. *CoRR*, abs/2310.04963, 2023. doi: 10.48550/ARXIV.2310.04963.

[41] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: An analysis and survey. volume 112 of *Advances in Computers*. 2019. doi: https://doi.org/10.1016/bs.adcom.2018.03.015.

[42] Terence J. Parr and Russell W. Quong. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25, 1995.

[43] Mayank Sharma, Pingshi Yu, and Alastair F Donaldson. Rustsmith: Random differential compiler testing for rust. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA ’23, 2023.

[44] Sohil Lal Shrestha. Harnessing large language models for simulink toolchain testing and developing diverse open-source corpora of simulink models for metric and evolution analysis. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA ’23, 2023. doi: 10.1145/3597926.3605233.

[45] Chengnian Sun, Vu Le, and Zhendong Su. Finding compiler bugs via live code mutation. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA ’16, 2016. doi: 10.1145/2983990.2984038.

[46] Azul Systems. JavaFuzzer, 2013. URL https://github.com/AzulSystems/JavaFuzzer/.

[47] Arun James Thirunavukarasu, Darren Shu Jeng Ting, Kabilan Elangovan, Laura Gutierrez, Ting Fang Tan, and Daniel Shu Wei Ting. Large language models in medicine. *Nature Medicine*, 29(8), 2023. doi: 10.1038/s41591-023-02448-8.

[48] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Conference on Neural Information Processing Systems*, NeurIPS ’22, 2022.

[49] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming Cui, Lingming Zhang, and Yuqun Zhang. One fuzzing strategy to rule them all. In *International Conference on Software Engineering*, ICSE ’22, 2022. doi: 10.1145/3510003.3510174.

[50] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4All: Universal fuzzing with large language models. In *IEEE/ACM International Conference on Software Engineering*, ICSE ’24, 2024. doi: 10.1145/3597503.3639121.

[51] Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. *CoRR*, abs/2309.03409, 2023. doi: 10.48550/ARXIV.2309.03409.

[52] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. White-box compiler fuzzing empowered by large language models. *CoRR*, abs/2310.15991, 2023. doi: 10.48550/ARXIV.2310.15991.

[53] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’11, 2011. doi: 10.1145/1993498.1993532.

[54] Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Yongji Wang, and Jian-Guang Lou. Large language models meet NL2Code: A survey. In *Annual Meeting of the Association for Computational Linguistics*, ACL ’23, 2023. doi: 10.18653/V1/2023.ACL-LONG.411.

[55] Zhiqiang Zang, Nathan Wiatrek, Milos Gligoric, and August Shi. Compiler testing using template Java programs. In *IEEE/ACM International Conference on Automated Software Engineering*, ASE ’22, 2023. doi: 10.1145/3551349.3556958.

[56] Qirun Zhang, Chengnian Sun, and Zhendong Su. Skeletal program enumeration for rigorous compiler testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’17, 2017. doi: 10.1145/3062341.3062379.