Learning Compiler Fuzzing Mutators from Historical Bugs

Lingjun Liu
North Carolina State University
Raleigh, NC, USA
lliu39@ncsu.edu

Feiran (Alex) Qin
North Carolina State University
Raleigh, NC, USA
fqin2@ncsu.edu

Owolabi Legunsen
Cornell University
Ithaca, NY, USA
legunsen@cornell.edu

Marcelo d'Amorim
North Carolina State University
Raleigh, NC, USA
mdamori@ncsu.edu

Abstract

Bugs in compilers, which are critical infrastructure today, can have outsized negative impacts. Mutational fuzzers aid compiler bug detection by systematically mutating compiler inputs, i.e., programs. Their effectiveness depends on the quality of the mutators used. Yet, no prior work used compiler bug histories as a source of mutators.

We propose IssueMut, the first approach for extracting compiler fuzzing mutators from bug histories. Our insight is that bug reports contain hints about program elements that induced compiler bugs; they can guide fuzzers towards similar bugs. IssueMut uses an automated method to mine mutators from bug reports and retrofit such mutators into existing mutational compiler fuzzers.

Using IssueMut, we mine 587 mutators from int(1457 + 303) GCC and LLVM bug reports. Then, we run IssueMut on these compilers, with all their test inputs as seed corpora. We find that "bug history" mutators are effective: they find new bugs that a state-of-the-art mutational compiler fuzzer misses—**25** in GCC and **27** in LLVM. Out of the int(37 + 28) bugs we reported, 60 were confirmed or fixed, validating our idea that bug histories have rich information that compiler fuzzers should leverage.

CCS Concepts

• **Software and its engineering** → **Software verification and validation**; **Functionality**.

Keywords

compiler testing, bug histories, mutational fuzzing

ACM Reference Format:
Lingjun Liu, Feiran (Alex) Qin, Owolabi Legunsen, and Marcelo d'Amorim. 2026. Learning Compiler Fuzzing Mutators from Historical Bugs. In *23rd International Conference on Mining Software Repositories (MSR '26), April 13–14, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3793302.3793374

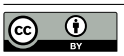
1 Introduction

Compilers are broadly used in software development. Ensuring their correctness is therefore critical. An impressive number of fuzzers were proposed to find compiler bugs [7, 10, 33, 34]. Mutational fuzzers [1, 9, 26, 44, 68, 70] were recently shown to be effective. But, it is challenging to explore the state space of a mutational fuzzing campaign—expressible as a graph with input programs as nodes and all possible mutations of those inputs as vertices. So, high-quality mutators [10, 44] that can produce syntactically-valid inputs (i.e., self-contained code snippets) to trigger deep compiler behaviors are needed. We observe that existing mutational fuzzers do not leverage an important ingredient in creating mutators: the history of previous bugs. We conjecture that bug reports contain important hints to drive a fuzzer to areas of the search space that are likely to reveal bugs. *This paper evaluates this conjecture.*

Leveraging bug histories for bug finding was explored in other contexts [8, 45, 53, 55, 72], including compiler fuzzing [11, 50, 68, 73]. But, no prior work derived mutators for mutational fuzzing from bug histories. Recently, MetaMut [44] used large language models (LLMs) to obtain mutators for a mutational fuzzer, but MetaMut does not leverage bug reports.

Our study is based on **IssueMut**, an approach we propose to extract mutators from bug histories and fuzz compilers with them. IssueMut has two components: (1) *a mining component* that semi-automatically reverse engineers mutators from bug reports, and (2) *an enhanced fuzzer framework* that retrofits the resulting mutators into existing mutational fuzzers. We next briefly introduce these two components (§3 has details).

IssueMut's *mining component* takes as input past reports of fixed bugs, and outputs mutators that can drive a mutational fuzzer towards similar bugs. Mining is done in four steps. First, IssueMut automatically extracts bug-triggering input programs–henceforth called *positive inputs*–from bug reports. Bug reports with (i) no inputs or with (ii) inputs that crash the compiler are discarded; the former are not useful and the latter indicate that a compiler bug is still present. Second, IssueMut automatically obtains *negative inputs*: slight modifications of positive inputs that do not trigger the reported bugs. Third, we manually confirm that the negative input does not trigger the bug as the positive input does. Fourth, IssueMut uses LLM agents to automatically generate mutators (i.e., programs that make simple semantic-changing code transformations) from the positive and negative inputs. A mutator should transform negative inputs into positive ones. IssueMut prevents from generating overfitting mutators by validating them against additional test cases generated from the mutation description.



This work is licensed under a Creative Commons Attribution 4.0 International License.
MSR '26, Rio de Janeiro, Brazil
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2474-9/2026/04
https://doi.org/10.1145/3793302.3793374

IssueMut's *enhanced fuzzer framework* retrofits the "bug history" mutators into mutational fuzzers (i.e., MetaMut [44] and Kitten [66]). MetaMut is a state-of-the-art (SoTA) mutational fuzzer that was shown to outperform leading generational and mutational fuzzers (AFL++ [12], Csmith [67], YARPGen [27], and GrayC [10]) with respect to coverage and unique crashes triggered. Kitten is a SoTA grammar-based mutational fuzzer that is language-agnostic and supports the C programming language. Prior work showed that Kitten outperforms the grammar-based mutational fuzzer Grammarinator and the language-agnostic fuzzer Fuzz4All in terms of the number of discovered crashes.

We evaluate IssueMut in two ways. First, we carry out a mining campaign on bug histories of GCC and LLVM, two widely used C compilers. We obtain **587** *code mutators* from int(1457 + 303) reports (1457 in GCC and 303 in LLVM). Second, we integrate mined mutators into SoTA mutational fuzzers (i.e., MetaMut and Kitten) and conduct compiler fuzzing campaigns using GCC and LLVM regression tests as seed corpora. We evaluate IssueMut across four dimensions:

1. How do IssueMut mutators improve mutational fuzzers?
2. How do the mutators from IssueMut and MetaMut differ?
3. How beneficial is it to run fuzzing campaigns with only "successful" crash-revealing mutators?
4. How useful are the bugs reported with IssueMut?

Considering the *first dimension*, we assess the impact of IssueMut mutators on SoTA mutational fuzzers (i.e., Kitten and MetaMut). We build variants of these fuzzers—MetaMut-i and Kitten-i—by augmenting them with IssueMut mutators. Across multiple fuzzing campaigns, MetaMut-i and Kitten-i discover a significantly higher number of crashes than their respective baselines.

Considering the *second dimension*, we find that IssueMut and MetaMut mutators have important similarities and differences. For example, we find that, for both sets, most successful mutators only reveal one bug and that most bugs are triggered by inputs obtained from one mutation. But, the sets of mutators are also inherently different–IssueMut mutators focus on finding bugs that use elements from previously reported ones, while MetaMut mutators relies on what the LLM learns. A concrete manifestation of this difference is that IssueMut can mine successful mutators by transforming program elements from the recent C23 specification [21], as recent bugs are related to those elements [14, 30–32]. We also find that IssueMut and MetaMut mutators are effective. Both sets lead to several compiler crashes within the 24 hours campaigns with IssueMut triggering **64** unique crashes that MetaMut did not find (See Figure 6b).

Considering the *third dimension*, fuzzing with only successful mutators triggers more new crashes and triggers them relatively faster, compared to running separate campaigns that run the original, (larger) sets of mutators for longer. Intuitively, fuzzing using only successful mutators enables more exhaustive coverage of the search space with mutators that are more likely to help find bugs. We find, e.g., that a more focused fuzzing campaign triggers **20** crashes that are not triggered by longer-running campaigns with the full set of IssueMut or MetaMut mutators.

Considering the *fourth dimension*, we find that GCC and LLVM developers reacted positively. They confirmed **60** out of **int(37 + 28)** bugs that we report from the crashes triggered by IssueMut mutators.

```
static int __sync_lock_test_and_set_4 (volatile int*, int);
int v;
int sync_lock_test_and_set (int a)
{ return __sync_lock_test_and_set(&v, a); }
```

Listing 1: Bug present in Clang since version 12; still present in version 20. This bug was revealed from two mutations applied to a seed file; they introduce a declaration for the function call `__sync_lock_test_and_set` **and replace the** `extern` **keyword (not visible) with the** `static` **keyword.**

Lastly, we compare IssueMut with Fuzz4All [65], a recent generational LLM-based fuzzer for systems like compilers. We use Fuzz4All in a fuzzing campaign with a 24-hour time budget for the fuzzing loop. We also use a similar GPU setup as in the Fuzz4All paper, but we fuzz more recent compiler versions to facilitate our comparison. We repeat this experiment 7 times, generating a similar number of inputs as in the Fuzz4All paper (~450,000 inputs from 14 runs; 7 for GCC and 7 for LLVM), but these are much fewer than those generated by IssueMut in 24 hours (~6,700,000). Fuzz4All triggers 11 crashes, two of which are duplicate. Of the 9 unique crashes, 4 are also triggered by IssueMut.

This paper makes the following contributions:

- ⋆ **Idea.** We propose the IssueMut approach for "mining" data from bug histories into simple, yet effective fuzzing mutators. We focus on C compilers, but there is no fundamental reason why "bug history" mutators should not apply more broadly.
- ⋆ **Mutators.** We curate a set of 587 C code mutators, which reflect salient code features that helped find previously reported bugs.
- ⋆ **Evaluation.** We comprehensively evaluate IssueMut against several baselines (e.g., Kitten, MetaMut, and Fuzz4All). We find int(37 + 28) bugs, many of which are unique to IssueMut.
- ⋆ **Lessons.** We present several lessons learned from our study (§ 5.4). For example, we find that mutators exploring recently-proposed compiler features –e.g., those proposed in the C23 standard [21]– are promising for finding bugs and encourage fuzzers to test more files with those features.

2 Example

We present a bug-revealing input that IssueMut obtains using "bug history" mutators and provide an overview of IssueMut's steps.

2.1 Clang bug #120083

The code snippet below is part of GCC's test suite [58]:

```
int v;
int sync_lock_test_and_set (int a)
{ return __sync_lock_test_and_set (&v, a); }
```

Listing 1 shows a variant of this snippet that triggers a crash in Clang version 20 (revision 9bdf683). That crash occurs during LLVM's IR code generation for method `sync_lock_test_and_set`. LLVM developers confirmed our bug report, saying that the root cause of the crash has been in the codebase since Clang version 12.0.0, which was released on April 14, 2021.[1] Two mutations yield the code in Listing 1. The first mutation, **M17** [39], adds a missing declaration to a function used in the original input; that declaration uses the `extern` qualifier. The second mutation, **M3** [39], replaces the `extern` keyword –introduced by M17– with the `static` keyword.

[1] Playground reproduction: https://godbolt.org/z/v3zEGx1xn

We apply each mutation separately and find that both are needed to reproduce the bug. The effects of mutations M3 and M17 in Listing 1 are shown in yellow and green, respectively.

2.2 Overview

2.2.1 Positive and Negative Examples. IssueMut uses a semi-automated approach to obtain positive and negative examples from bug reports (see § 3). We next summarize how IssueMut obtains these examples for mutators M17 and M3.

M17. We use the following input from GCC bug report #108777 [16] to obtain a *positive test* for M17:

```
extern void *memcpy(void *, const void *, __SIZE_TYPE__);
extern void *memmove(void *, const void *, __SIZE_TYPE__);
extern void *memset(void *, int, __SIZE_TYPE__);
void foo(void *p, void *q, int s) { memcpy(p, q, s); }
void bar(void *p, void *q, int s) { memmove(p, q, s); }
void baz(void *p, int c, int s) { memset(p, c, s); }
```

The discussion in the report explains that when explicit external function declarations (e.g., `extern void *memcpy(...)`) are present in code, GCC treats them as regular external function references rather than built-in functions. That treatment prevents GCC from applying built-in function optimizations and transformations. IssueMut replaces these explicit `extern` function declarations with standard headers to obtain a *negative test case*, so that the compiler recognizes these functions as built-ins and apply instrumentation. To sum up, from these examples, IssueMut creates a mutator (M17) that analyzes the code to determine the signatures of functions with a missing declaration, then it adds a function declaration with an `extern` modifier.

M3. We use the following input from GCC bug report #108449 [15] to obtain a *positive test* for M3:

```
static int vfork(); void f() { vfork(); }
```

The bug report explains that this input manifests a regression in the compiler. GCC-13 raises an Internal Compiler Error (ICE) when compiling code with a static and undefined function declaration. In that case, the compiler internally attempts to convert the `static` declaration to `extern` declaration, but it fails. The corresponding negative input behaves similarly, i.e., the corresponding mutator replaces the `static` modifier in a function declaration with `extern` to simulate the problematic scenario reported in this regression. To sum up, from this pair of examples, IssueMut creates a mutator (M3) that replaces the keyword `extern` with the keyword `static`.

2.2.2 Mutator Creation. The following simplified code snippet implements mutator M17 as an Clang AST visitor [60]:

```
// Mutator M17
class AddExternDeclaration ... {
  std::set<const FunctionDecl *> FunctionDecls;
  // collect signatures of function calls
  bool VisitCallExpr(CallExpr *Call) { ... }
  // manipulate AST to introduce function declaration
  // with extern keyword at the beginning of the file.
  bool mutate() { ... } }
```

M17 is a context-sensitive mutator that needs to collect information before mutating the code. But, in most bug reports that we analyze, we find that mutators do *not* require contextual information. In those cases, a sed-like approach is sufficient to create mutators. Mutator M3 is an example of a context-insensitive mutator. It is implemented with this generic bash script, passing "`extern`" and "`static`" as parameters `PATTERN` and `REPLACEMENT`, respectively:

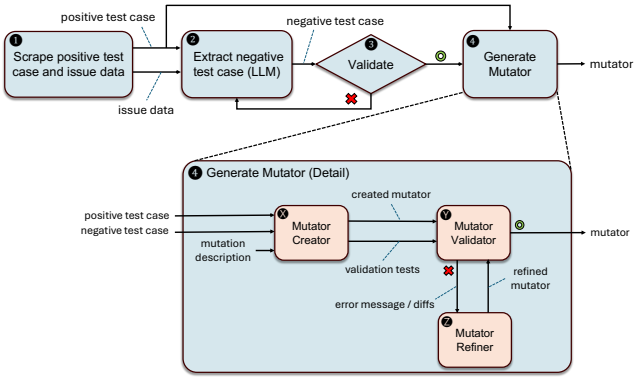


Figure 1: The workflow of IssueMut's mutator miner.

```
#!/bin/bash
FILE=$1
PATTERN=$2
REPLACEMENT=$3 ...
# Replace a string matching regex $PATTERN with
# string $REPLACEMENT at line ln in file $FILE
sed -i -E "${ln}s/$PATTERN/$REPLACEMENT/" $FILE
```

2.2.3 Summary. IssueMut mines 587 mutators from GCC and LLVM bug histories. We configured the SoTA mutational compiler fuzzer MetaMut [44] with these mutators and systematically applied them using 35,472 seed test cases drawn from GCC and LLVM. IssueMut mutators discovered 78 crashes, 64 of which MetaMut mutators miss.

3 Approach

We describe each component of IssueMut in more detail. §3.1 describes IssueMut's automated miner that obtains mutators from bug reports. §3.2 describes IssueMut's fuzzing framework.

3.1 Mutator Mining

Figure 1 shows the workflow of IssueMut's mutator mining. We describe each of the steps in the following.

3.1.1 Scrape positive test case and issue data (❶). We write scripts to scrape test cases from fixed and closed issues created in the period from January 2023 to October 2024 (22 months). We chose this window to prioritize recency, but IssueMut is not tied to this period and can be applied to longer histories. A total of 1457 and 303 candidate issues from GCC and Clang, respectively, satisfy these criteria. For each candidate issue, we automatically execute the associated test on a recent version of the corresponding compiler and discard tests that still trigger crashes. This step ensures that any newly observed crashes during fuzzing can be attributed to our mutators rather than known regressions. If the original test already triggers a crash in a recent compiler version, mutating that test could trivially rediscover the same failure and artificially inflate our measurement of mutator effectiveness. If no crash is detected, this step reports a pair of issue and associated test case as output. We say a test case is "positive" because it triggers a bug in a previous compiler version; the presence of an issue associated with the test demonstrates that ability. We say a test case is "negative", otherwise. Figure 2a shows a fragment of a GCC bug report, including the affected compiler component (tree-optimization), the bug consequence (ice-on-valid-code), the bug-revealing test file

```
Product: gcc
Component: tree-optimization
Keywords: ice-on-valid-code ...
$ cat z1.c
static int vfork();              ### FILE ###
void f() { vfork(); }            ### FILE ###
$ gcc-13-20230115 -c z1.c -O2
during GIMPLE pass: cddce
z1.c: In function 'f':
z1.c:2:1: internal compiler error: in \
  eliminate_unnecessary_stmts, at \
  tree-ssa-dce.cc:1512
    2 | void f() { vfork(); }
      | ^~~
0xe0f0dc eliminate_unnecessary_stmts
    ../../gcc/tree-ssa-dce.cc:1512...
0xe11e55 execute
    ../../gcc/tree-ssa-dce.cc:2069
```

(a) Issue metadata and stack trace. A valid input (z1.c) reveals an Internal Compiler Error (ICE) during an optimization step in GCC.

```
When we have static declaration without definition we diagnose that
and turn it into an extern declaration. That can alter the outcome
of maybe_special_function_p here and there is really no point in
doing that, so do not.
```

(b) Relevant fragment of the issue discussion.

```
# positive test case          # negative test case
static int vfork();           extern int vfork();
void f() { vfork(); }         void f() { vfork(); }
```

(c) Positive and negative test cases.

Figure 2: Fragment of bug report [15] and corresponding positive and negative test cases.

(see "### FILE###"), and a stack trace fragment. Figure 2b shows a portion of the discussion explaining the bug.

3.1.2 Extract negative test case (❷). This step produces a negative test case from a given pair of issue and a positive test case that is mined from ❶. For example, if the issue indicates that adding `__attribute__((noipa))` reveals a bug when used in a function, the negation of that positive test case removes that attribute. We use an LLM to solve this task, due to its ability to handle text and code. We use GPT-4o mini [42] because we find it to offer a reasonable trade-off between accuracy and cost. To optimize the prompt engineering process, we employed PromptPerfect [17], an AI prompt optimization tool designed to enhance the quality of LLM outputs. The optimized prompt we obtain was:

```
You are an experienced C developer. Your task is to read a bug
report and corresponding bug-revealing input and produce a similar
input that does not manifest the bug. The response should only
include the description of the mutation and the mutated C code.
###Bug Report: ...
###Bug-revealing Input: ...
```

Figure 2c shows an example of a negative test case (simplified for clarity) for the corresponding issue and positive test case.

3.1.3 Validate negative test case (❸). We manually check if the negative test case compiles and if it is consistent with the issue description. If the negative test case does not reflect the issue, we repeat the query in ❷ with another LLM– Claude or o1-mini. We discard the issue if we cannot validate the test case. A test case can have multiple negative variants, similar in different ways to the positive test cases and not bug triggering. IssueMut only requires one that reproduces the issue. Indeed, different negative tests can produce different mutators; quantifying the sensitivity of to this choice is left as future work.

3.1.4 Generate mutators (❹). This step creates mutators for a pair of positive and negative test cases. A mutator is a function that identifies target locations and changes the code at one or more of those locations. We empirically find that most mutators are context-insensitive, i.e., they check preconditions at the change location (see §2.2.2). Based on this observation, we developed an automated pipeline that generates mutators by deriving sed-like transformations from positive and negative test case pairs. More precisely, IssueMut automatically creates mutators that transform a negative test case into the corresponding positive one. We use a LangChain-based [6] agentic architecture powered by the Gemini 2.5 Pro model. The architecture is based on three agents:

Mutator Creator. This agent performs three tasks to create a script for a mutator: *Generalize Mutation*, *Generate Tests*, and *Create Script*. First, the task *Generalize Mutation* reverses the description of how the LLM mutates a positive test case into a negative test case, as obtained in step ❷, and removes the specific parts from the description (e.g., literals). For example, given LLVM issue #113692 [29], the agent first reverses the mutation description "Changed the inline assembly constraint from `+f` to `+x` to use SSE registers instead of the x87 floating point stack...", and then generalizes it to obtain the description: "Change an inline assembly constraint from one that uses an SSE register to one that uses the x87 floating point stack." Second, the task *Generate Tests* generates a given number (3 default) of test cases (i.e., C code input-output pairs) to validate the mutation description, i.e., to ensure the mutator works beyond the original negative-positive input pair. This step is crucial to create generalizable mutators and prevent "overfitting". For the issue above, an example test case pair is:

```
void func(float a) { __asm__("fsqrt" : "+x"(a)); } // input
void func(float a) { __asm__("fsqrt" : "+f"(a)); } // output
```

Third, the task *Create Script* synthesizes a generic sed-based bash script from a mutation description. We use the template command `sed -i -E's/<r>/<s>/g' <file>`, which replaces all strings matching the regular expression `<r>` with the string `<s>` in the input `<file>`. Our artifacts contain the prompt [48] that this agent uses to obtain a bash script.

Mutator Validator. This agent validates the mutator script by running it against all test cases, i.e., the original test case (§3.1.1 and §3.1.2) and the additional test cases (task "Generate Tests"). Recall that a test artifact is an input-output pair of C code. So, testing consists of checking whether running the script on the input produces the expected output. The validator agent produces one of four outcomes for a mutator: (1) *Correct* - all 4 test cases pass, resulting in a finalized mutator; (2) *Partially Correct* - 3 out of 4 test cases pass; (3) *Error* - the execution of the test harness –not a test– fails; (4) *Wrong* - passes in fewer than 3 test cases. We report a mutator if the agent labels it as "Correct" or "Partially Correct". We trigger a refinement step when observing the "Error" and "Wrong" outcomes. More precisely, we send the corresponding error message and test output diffs to the "Mutator Refiner" agent along with the request to refine the mutator.

Mutator Refiner. This agent refines likely invalid mutator scripts using the error messages obtained during the validation step. The refiner agent iterates until the script can pass on the majority of

the tests or until it reaches five iterations. Our artifacts include the prompt [49] that the Mutator Refiner agent uses to refine a bash script. For the LLVM issue #113692 [29], the final mutator script is:

```
#!/bin/bash
FILE=$1
PATTERN='"\+x"(\s*\([^)]*\))'
REPLACEMENT='"\+f"\1'
sed -i -E 's/$PATTERN/$REPLACEMENT/' $FILE
```

For the few cases where contextual information is necessary to transform the code, we use Clang's AST visitors [60], like MetaMut. §2.2.2 shows a simplified version of a bash script and AST visitors that we use to implement these mutators. It is worth noting that syntactic duplicates can arise when creating mutators. We look for duplicates by searching for hash collisions in a hash table. We compute a hash for a mutator by applying the mutator on a random sample of our seed inputs representative of the entire seed corpus with 99% confidence and 1% margin of error [57]. We apply a mutator on each of these inputs and use the `sha256sum` UNIX command to obtain an aggregate hash from the outputs. We find a total of 16 duplicate mutators (14 from GCC and 2 from LLVM). We exclude these 16 from those that we mine. Of the **587** (=603-16) mutators that remain, 412 are from GCC and 175 are from LLVM. There is no fundamental reason for this imbalance; it only reflects that we started our experiments with GCC. In the rest of this paper, we use **bhis** (bug history) to refer to this set of 587 mutators.

3.2 Enhanced Mutational Fuzzer Framework

Generational fuzzers create inputs anew, whereas mutational fuzzers modify existing inputs [70]. We focus on mutational fuzzing due to its ability to efficiently create variations of existing inputs. In compiler testing, inputs are self-contained code snippets. A mutational fuzzer randomly applies mutations to files present in a queue, initialized from a seed corpus. The seed corpus that we use includes all test cases from GCC [58] and LLVM [61]. Fuzzing stops after a time budget. It is worth noting that many recent mutational fuzzers use coverage to drive fuzzing. Coverage-guided fuzzers (CGF) save mutations that uncover new branches [59] of a program under test in its queue for additional fuzzing.

We integrate our mined mutators into recently-proposed SoTA mutational fuzzers, MetaMut [44] and Kitten [66]. MetaMut is a coverage-guided mutational fuzzer of C programs that revealed bugs in GCC and LLVM. Kitten is a language-agnostic mutational fuzzer that has been shown to reveal bugs in C compilers.

MetaMut mutators use abstract syntax tree (AST) visitors [47] to transform C files. As of now, MetaMut only checks for crashes, but there is no fundamental reason MetaMut could not be adapted to detect other kinds of bugs, such as miscompilations through differential testing [35]. We use the stacktrace-based procedure provided in MetaMut to de-duplicate alarms raised during our fuzzing campaigns. That procedure considers the top two stack frames, including the program counter and ignores helper functions.

4 Evaluation

Research Questions. Our evaluation addresses four questions:

RQ1. How do IssueMut mutators improve SoTA mutational fuzzers?

RQ2. How do IssueMut and MetaMut mutators differ?

RQ3. How beneficial is fuzzing with *only successful* mutators, compared to fuzzing with all mutators?

RQ4. How useful are the bug reports of IssueMut?

The first question investigates the impact of IssueMut mutators on SoTA mutational fuzzers (e.g., MetaMut and Kitten) in terms of coverage and ability to reveal bugs. The second question makes an in-depth comparison between IssueMut and MetaMut mutators quantitatively (e.g., crashes over time) and qualitatively (e.g., static and dynamic characteristics). The third question evaluates the benefits of restricting the set of mutators in a fuzzing campaign to only include mutators that previously triggered crashes. Intuitively, using fewer mutators can lead to more efficient exploration. The fourth question evaluates the usefulness of the bug reports we submitted as a result of using IssueMut mutators for fuzzing.

4.1 Baselines

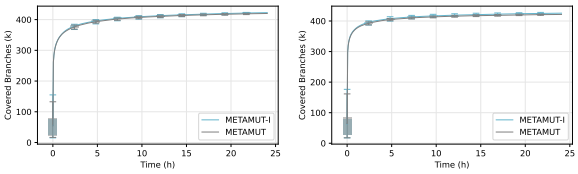
Our primary comparison baseline is **MetaMut** [44], a recently-proposed SoTA mutational fuzzer that is *extensible* with user-provided mutators; it (i) achieves 5.4% and 6.1% higher code coverage than leading generational and mutational compiler fuzzers, respectively, (AFL++ [12], Csmith [67], YARPGen [27], and GrayC [10]) and (2) detects three times more unique crashes than these tools. MetaMut uses LLMs to create mutators within a space of possible actions (e.g., Add) and program elements (e.g., Expression). MetaMut provides two sets of mutators that differ based on whether human supervision is used to obtain them: **mu.s** and **mu.u**. The set mu.s (supervised) includes 68 mutators obtained through interactions with GPT-4 [43]. MetaMut authors created these mutators by analyzing, debugging, and validating the generated AST-based mutators (§2.2.2). They refined LLM prompts throughout this process, and considered prompts to be fully refined after roughly two weeks. After that, they ran their workflow *without* human supervision for 100 iterations using the same GPT-4 model and obtain another set of 50 mutators, referred to a mu.u (unsupervised). We compare these two sets of MetaMut mutators with IssueMut mutators (see RQ2, §4.4).

Our secondary comparison baseline is **Kitten** [66], a SoTA language-agnostic grammar-based mutational fuzzer with support for C. We consider Kitten as a secondary baseline because its generic design makes it less directly comparable to IssueMut than MetaMut. Our mined mutators derive from C compiler bug histories and target C-specific constructs, aligning them more closely with MetaMut, which is also a mutational fuzzer specialized for C. Kitten performs tree-level mutations (splicing, replacement, deletion, and repetition) on parse trees and performs token-level mutations (insertion, deletion, and replacement) on token sequences from seed programs to generate syntactically valid and invalid test programs for compiler testing.

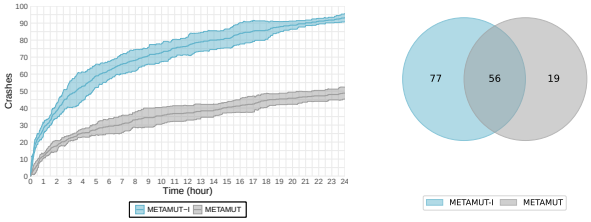
Section 5 also reports on a lightweight comparison against **Fuzz4All** [65]. It is worth noting that Fuzz4All requires GPUs to generate files whereas MetaMut and Kitten only require CPUs.

4.2 Setup

We run experiments on an Ubuntu 22.04.5 LTS DELL PowerEdge R6625 server, equipped with two 96-core AMD EPYC 9684X processors and 755 GB of memory. As in prior work on compiler

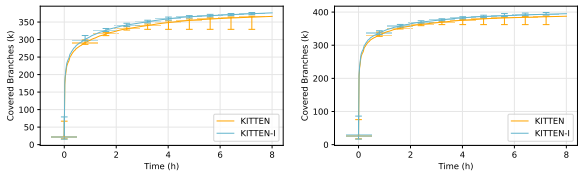


(a) Coverage: GCC (left) and LLVM (right).

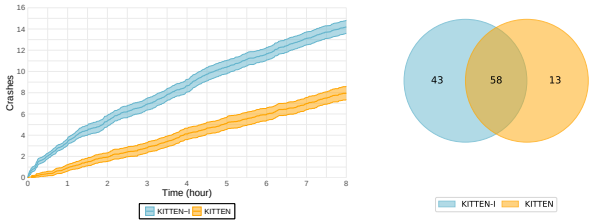


(b) Crashes: progress (left) and uniqueness (right)

Figure 3: Comparison of MetaMut and MetaMut-i.



(a) Coverage: GCC (left) and LLVM (right).



(b) Crashes: progress (left) and uniqueness (right)

Figure 4: Comparison of Kitten and Kitten-i.

fuzzing [44], we focus on the −O2 optimization level. For the sake of reproducibility, we run experiments on two fixed recent compiler revisions: GCC 15 (87492fb) and Clang 20 (9bdf683). But, we report bugs in the most recent revisions of these compilers. To account for inherent nondeterminism, we run each experiment multiple times using fixed random-number generation seeds. Applying mutators to seed programs can generate invalid programs; we consider a program valid if it compiles successfully, and across our evaluation runs the average compilation success rate is 71.2% (MetaMut reports a compilation success rate of 72%). Since production compilers should reject invalid inputs gracefully with diagnostics rather than crash, we treat crashes on invalid inputs as robustness failures.

4.3 Answering RQ1

We report the impact of IssueMut mutators on two SoTA mutational fuzzers, Kitten and MetaMut. We augment MetaMut and Kitten with mutators mined by IssueMut, and call the resulting variants MetaMut-i and Kitten-i, respectively. For fair comparison, we use the same seed corpora across all techniques. For crash analysis, we use MetaMut's crash deduplicator to identify unique crashes across all techniques.

We present the results for MetaMut-i in §4.3.1 and for Kitten-i in §4.3.2. We report MetaMut and Kitten separately because they exhibit intrinsic differences in input generation efficiency due to their fundamentally different fuzzing strategies. MetaMut's coverage-guided approach improves its exploration efficiency but limits the number of files it generates. But, Kitten's grammar-based approach generates more inputs as it does not rely on coverage feedback. Separate evaluation allows us to isolate and study the impact of IssueMut mutators on these two distinct fuzzing paradigms.

4.3.1 MetaMut vs. MetaMut-i. We augment MetaMut with all 587 mutators mined by IssueMut, resulting in a variant that we call MetaMut-i. For fair comparison, we allocate 30 physical cores each to MetaMut and MetaMut-i. We use MetaMut's parallel setting to run 30 agents in parallel (-j 30), and run MetaMut and MetaMut-i five times each, with each run taking 24 hours.

Figure 3 compares MetaMut and MetaMut-i, showing how IssueMut's mutators impact coverage and bug-discovery effectiveness. In Figure 3a, we see that the *branch coverage* difference between MetaMut and MetaMut-i is negligible. This is expected: both variants are coverage-guided. In contrast, Figure 3b shows that the bug-discovery effectiveness of both variants differs substantially. There, MetaMut-i reveals an average of 93.2 crashes after 24 hours, whereas MetaMut reveals only 49.0 crashes – a difference of up to 1.9×. The Venn diagram in Figure 3b also shows that MetaMut-i discovers 77 unique crashes, which far exceeds the 19 unique crashes found by MetaMut.

4.3.2 Kitten vs. Kitten-i. Kitten-i augments Kitten with 20 mutators from IssueMut (out of 587). Unlike MetaMut, which directly supports retrofitting mutators, Kitten lacks such infrastructure, and its AST representation differs from Clang's AST used by IssueMut mutators and MetaMut. Incorporating all mined 587 mutators into Kitten would therefore require substantially more engineering effort than it took us for MetaMut. To make the comparison feasible, we randomly select and implement 20 mutators into Kitten. We use the default Kitten setting, and run both Kitten and Kitten-i on a physical CPU core. Also, we run Kitten and Kitten-i 50 times each, with each run taking 8 hours.

Figure 4 compares Kitten and Kitten-i, showing how IssueMut's mutators impact coverage and bug-discovery effectiveness. Figure 4a shows that Kitten-i achieves a marginally higher *branch coverage* than Kitten for GCC and LLVM. By augmenting Kitten with 20 IssueMut mutators, Kitten-i exposes additional execution paths, as reflected in its higher branch coverage. The progress plot in Figure 4b shows that after 8 hours, Kitten-i reveals an average of 14.22 crashes, compared to only 7.98 for Kitten, corresponding to an improvement of up to 1.78×. The Venn diagram in Figure 4b also shows the advantage of Kitten-i: it discovers 43 unique crashes that Kitten did not, more than three times the 13 crashes that Kitten (but not Kitten-i) finds. A paired t-test confirms that the improvement is significant ($p < 0.001$), with a 95% confidence interval of [5.45, 7.03] crashes. The effect size (Cohen's $d = 2.23$) also shows a large practical impact of IssueMut's mutators in Kitten-i.

RQ1 (Enhancing SoTA)
MetaMut-i and Kitten-i outperform their baseline SoTA variants (MetaMut and Kitten) w.r.t. number of unique bugs found. After 24 hours, MetaMut-i finds 58 more crashes than MetaMut (77 vs. 19). Similarly, after 8 hours, Kitten-i finds 30 more crashes than Kitten (43 vs. 13). More importantly, MetaMut-i and Kitten-i find many crashes that these baseline variants miss.

4.4 Answering RQ2

We assess how IssueMut and MetaMut mutators compare from different angles: static (§4.4.1), dynamic (§4.4.2), crashes over time (§4.4.3), crash uniqueness (§4.4.3), and crash diversity (§4.4.4).

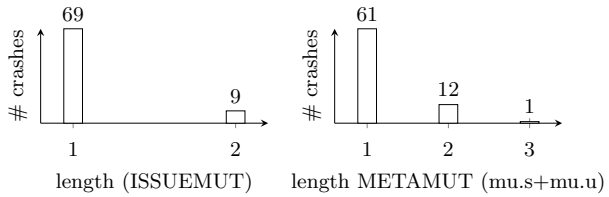
We compare IssueMut and MetaMut mutators by running fuzzing campaigns with MetaMut configured under three mutator sets: MetaMut mutators (**mu.s** and **mu.u**) and IssueMut mutators (**bhis**). For this experiment, we used the setup from Section 4.3.1, i.e., we allocated 30 physical cores to each fuzzing technique, and ran five times for 24 hours.

4.4.1 Static Characteristics. We focus on two characteristics of a mutator: the kind of change it makes (e.g., code addition) and the program elements it manipulates (e.g., expression).

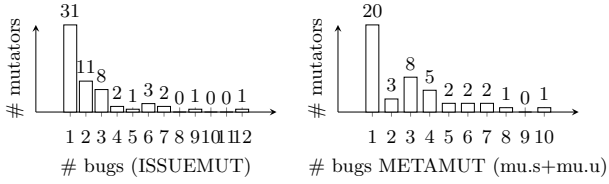
Our artifacts include a list of IssueMut mutators that contributed to expose at least one crash [39]. Table 1 shows a representative list of ten IssueMut mutators that contributed to revealing at least three bugs. Column "Id" is the identifier of a mutator, "Src." shows the compiler targeted by the mutator (G=GCC or L=LLVM), "Action" and "Program Element" indicate, respectively, the transformation and the target program element for that transformation, and "Description" is a short description of the mutator. The symbol ⋆ denotes mutators related to the recent C23 standard [21].

Figures in our artifacts [37, 38] show, respectively, the distributions of transformations and target program elements for *successful* IssueMut and MetaMut mutators. Figure [37] indicates a discrepancy in the proportion of actions between these two mutator sets. IssueMut mutators most often add, modify, or delete program elements. In contrast, most MetaMut mutators perform modifications. Figure [38] further shows that some groups of program elements are targeted by only one of the techniques, i.e., their corresponding mutator sets. We attribute these structural differences to how MetaMut and IssueMut create mutators. IssueMut derives mutators from bug histories, while MetaMut relies on an LLM and focuses on specific AST elements to reduce task definition scope in prompts. Consequently, the two approaches produce mutators with distinct structural characteristics. Syntactically, IssueMut mutators often involve multiple edits, reflecting changes in positive-negative pairs, e.g., one mutator may introduce a new variable declaration and then assigns a type-compatible value to that variable. In contrast, MetaMut mutators perform single edits as they are generated from an LLM prompt with only one mutation verb and one AST target. As a result, IssueMut and MetaMut mutators differ both in the actions they perform and in the program elements they manipulate.

Beyond these syntactic differences, we also observe semantic differences between the two approaches. Because IssueMut is sourced from real-world bug histories, its mutators reflect the recency of language evolution; for example, several successful IssueMut mutators manipulate constructs introduced in the recent C23 standard (e.g., **M8** and **M9** in Table 1). The generality of LLMs sometimes causes them fail to accurately prioritize recency.



(a) Histogram #bugs × #mutators.



(b) Histogram length mutation sequence × # crashes.

Figure 5: Dynamic characterization.

4.4.2 Dynamic Characteristics. We further compare IssueMut with MetaMut by considering the number of crashes they reveal and examining the lengths of the mutation chains associated with the detected crashes. The histograms from Figure 5a show the number of mutators (y-axis) that reveal a given number of crashes (x-axis). For example, eight of IssueMut's mutators reveal three crashes. Results indicate that, a higher number of mutators mined with IssueMut reveals more than one crash compared to MetaMut (mu.s and mu.u combined).

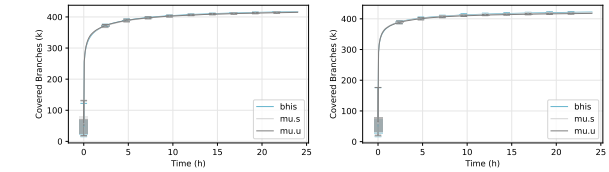
Figure 5b compares the lengths of IssueMut and MetaMut mutation sequences that produce crash-triggering inputs. It is worth noting that to measure the length of such chains we minimize mutation sequences aposteriori. Results indicate that the vast majority of crash-triggering inputs (int(100*(69 / 78))% in IssueMut and int(100*(61 / 74))% in MetaMut) are produced by a single mutation. The presence of crashes triggered using inputs resulting from multiple mutations indicate that (i) one mutation increased compiler coverage so the file was added to the queue; and (ii) the other mutator changed that file in a way that triggered a crash. Note that all three mutator sets (two from MetaMut and one from IssueMut) perform *one* mutation at a time, like in first-order mutants in mutation testing [23]. But, the fuzzing campaign can apply multiple mutations in subsequent steps, like in high-order mutants [22]. Overall, results indicate that IssueMut and MetaMut mutators have similar dynamic behaviors: most successful mutators trigger a single crash and most crash-triggering inputs result from one mutation.

4.4.3 Coverage and Crashes Triggered over Time. Figures 6a and 6b show the trends in coverage and crashes detected over time for each set of mutators, namely **bhis**, **mu.s**, and **mu.u**. From Figure 6a, we

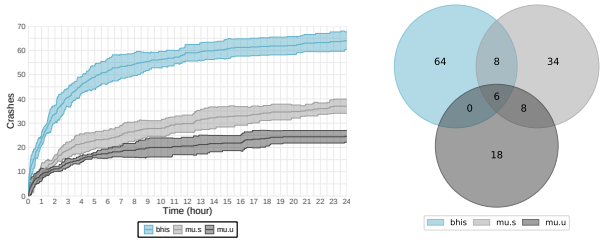
Table 1: Representative list of mutators revealed more than three crashes from IssueMut. ⋆ =C23 standard

At=Attribute BF=Builtin function Uo=Unary Operator FD=Function Declaration L=Literal I=Initialization P=Parameter E=Expression Sc=Storage Class Specifier S=Statement T=Type VD=Variable Declaration C=Character

Id	Src.	Action	Pgm. Elements	Description
M1	G	Add	BF E	Adds built-in function `__builtin_assoc_barrier()` around the return expression.
M2	G	Swap	E	Swaps two arguments of function calls.
M3	G	Modify	Sc	Replaces `extern` storage class specifier with `static`.
M4	G	Remove	Uo	Removes a type cast operator.
M5	G	Modify	P T	Replaces const pointer parameters with non-const non-pointer parameters.
M6	G	Modify	E	Replaces a variable reference with a call expression.
M7	L	Remove	VD	Removes the size expression from an array variable declaration.
M8	L	Modify	L	⋆ Replaces integer literals with a large numeric value using a new integer literal suffix. (e.g., changing `123` to `66666...wb` (The `wb` suffix is a bit-precise integer literal suffix introduced in the C23 standard)).
M9	L	Modify	L	⋆ Replaces integer zero literals into binary zero literals, which is introduced in the C23 standard (e.g., 0b0).
M10	L	Add	At FD	Adds `__attribute__((target_clones("default,avx")))` to function declarators.



(a) GCC and LLVM coverage.



(b) Differences in bugs revealed by techniques.

Figure 6: Progress of crash detection over time (left) and differences of crashes each mutator set detects (right).

find that differences in *branch coverage* are negligible, indicating that the mutator sets explore comparable code regions. However, Figure 6b (right) shows that a substantial number of unique crashes were discovered by IssueMut mutators, suggesting that IssueMut mutators guide the fuzzer toward distinct bug-revealing execution paths. The shaded areas in the progress plot on Figure 6b (left) represent the 95% confidence interval of the distributions of crashes at a given point in time. It is worth noting that crashes are deduplicated on each run to avoid inflation of results. On average, IssueMut's mutators reveal more crashes (**64.0**, on average) over time when compared to MetaMut's mutator sets (**37.0** and **24.6**, on average). We analyze the number of distinct crashes that each set of mutators finds at the end of fuzzing campaigns. The venn diagram in Figure 6b (right) shows the relationships among crashes found by each mutator set. The diagram merges results by taking the union of the observed crashes across the various runs of the fuzzer for a given mutator set. To sum up, under 24 hours fuzzing budgets, fuzzing with **bhis** (IssueMut) mutators reveals several crashes that mu.s and mu.u (MetaMut) miss **64** unique crashes in the union of bhis-induced crashes. So, there is evidence that "bug history" mutators complement and improve the bug-finding effectiveness of SoTA mutators.

4.4.4 Diversity. Table 2 shows the kinds of crashes that IssueMut and MetaMut find and their distributions across compiler modules.

Table 2: Kinds of crashes and their distribution across modules.

	IssueMut		MetaMut	
	Clang	GCC	Clang	GCC
Kind				
Segmentation Fault	5	7	4	2
Assertion Failure	33	14	41	7
Hang	0	0	1	0
Internal Compiler Error	2	17	3	16
Affected compiler modules				
Front-End	14	8	20	10
IR Generation	16	17	20	5
Optimization	6	9	2	0
Back-End	4	4	7	10

Assertion Failures and Internal Compiler Errors (ICE) are more common than hangs or segmentation faults. The proportions of these classes of crashes are similar for IssueMut and MetaMut. Also, crashes in the Front-End and in IR Generation are more common, compared to others. IssueMut and MetaMut find bugs deeper in the compiler stack, but they do so in lower numbers.

IssueMut and MetaMut trigger crashes of different kinds and in different modules. Overall, "bug history" mutators are distinct compared to the set of mutators from MetaMut.

RQ2 (Distinction from SoTA mutators)
IssueMut and MetaMut mutators differ in various ways. They trigger different crash types at different modules, showing that bug-history mutators complement and improve the bug-finding effectiveness of other mutator sets.

4.5 Answering RQ3

We evaluate the benefits of restricting the size of the set of mutators used in a fuzzing campaign by including only "successful" ones: those that previously triggered crashes. Intuitively, using fewer mutators can cover more of the search space within a time budget. Prior work explores strategies to calibrate exploitation-exploration before [2, 46, 69]. We report on a limit study to exploit such successful mutators. Such set includes 60 mutators from IssueMut (listed in our artifacts [39]) and 44 mutators from MetaMut.

Figure 7 shows the number of distinct crashes obtained after merging runs across multiple seeds with union. The venn diagram shows the differences in crashes found by each mutator set: **bhis**, **mu.s**, **mu.u**, and **successful** (104 mutators from all three sets). The fuzzing campaigns with "successful" mutators triggers **20** crashes

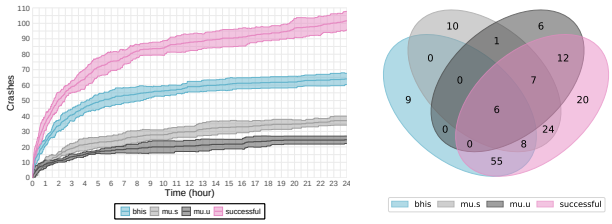


Figure 7: Comparison between fuzzing with bhis , mu.s , mu.u mutators and fuzzing with successful mutators (104).

that are *not* triggered by any other campaign. Of these, 6 crashes are due to IssueMut's mutators *alone*: 5 in LLVM (149368, 149369, 149371, 149372, one does not crash on trunk), and 1 in GCC (119177). As expected, the campaign with only "successful" mutators also reveals crashes faster. Figure 7 shows the progress of crashes triggered over time. For example, at the 2-hour mark, fuzzing with only "successful" mutators triggered an average of 50.8 distinct crashes.

RQ3 (Fuzzing with only "successful" mutators)
Running fuzzing campaigns with only "successful" mutators is beneficial: the focused campaign (i) triggers **20** crashes *not* triggered by campaigns that use all mutators (**mu.s**, **mu.u**, and **bhis**) in an 24 hours time budget; and (ii) triggers crashes faster.

4.6 Answering RQ4

We report on the ability of IssueMut's mutators to reveal bugs (as opposed to crashes). The bug reporting process spanned several months as we iteratively improved IssueMut by adding mutators. Throughout this process, we conducted multiple fuzzing campaigns to evaluate IssueMut's bug-finding ability. We reported bugs immediately upon discovery and ensured that newer versions consistently found a superset of crashes from prior campaigns. To contextualize our findings, compiler testing researchers report various numbers of bugs in their studies. For example, GrayC [10] evaluated their approach on GCC, LLVM, MSVC, and Frama-C. Considering GCC and LLVM alone (i.e., the two compilers that we use), they found, respectively, 11 and 10 bugs that were confirmed and fixed by compiler developers.

For bug reporting, we use *all* 78 crashes from the set bhis, represented in the venn diagram from Figure 6b (merged with union). Our artifacts show a list of all bugs we reported to developers [4]. Of these, int(27 + 25) bugs were only found by bhis mutators. Table 3a lists bugs that LLVM and GCC developers *fixed*. Column "Id" shows bug identifiers in the corresponding bug-tracking system, "Status" is the status of the bug report at the time of writing, and "Mutators" shows the number of mutators that contributed to finding the bug. For the "Mutators" column, once we detect a crash, we identify the length of the chain of mutations that produced the crash-triggering input and manually eliminate mutations that are unnecessary for crash reproduction. The notation "$M_1 \oplus M_2$" indicates that we can reproduce a bugx by applying M_1 or M_2 to a seed file. The artifacts show examples of bugs that require a sequence of mutations. Highlighted rows in Table 3a indicate bugs that *only* bhis mutators find, i.e., a subset of the 64 unique crashes from Figure 6b.

Table 3: Reported bugs.

(a) Fixed bugs reported with IssueMut. Highlighted rows indicate crashes that only IssueMut finds. The full list of bugs reported can be found elsewhere [4].

LLVM bugs		
Id	Status	Mutators
118892	**Fixed**	M5 ⊕ M6
123410	**Fixed**	M3
144771	**Fixed**	M4
149023	**Fixed**	M6
GCC bugs		
Id	Status	Mutators
118061	**Fixed**	M11 ⊕ M18 ⊕ M14 ⊕ M23
118674	**Fixed**	M2
118868	**Fixed**	M1
118948	**Fixed**	M54
119001	**Fixed**	M7
119204	**Fixed**	M30
121127	**Fixed**	M37
121130	**Fixed**	M6
121131	**Fixed**	M45;M20

(b) Summary of reported bugs.

C Compiler	Reported	Confirmed	Duplicate	Fixed
LLVM	37	27	6	4
GCC	28	11	3	9
Σ	65	38	9	13

Table 3b shows a summary of the reported bugs, indicating the current category of the report. Note that we report fewer bugs compared to the number of crashes we observe: 78 (=64+8+4) crashes observed versus int(37 + 28) bugs reported. The reason for the difference is that we only report bugs that we can reproduce in the most recent compiler release. It is also worth noting that we find duplicate bugs. As in prior work [26, 44], we mention these bugs here as we could get credit if reported earlier. Overall, results show that (i) int(100*(31 / 587))% of IssueMut's mutators revealed at least one bug; and (ii) developers confirmed or fixed majority of our bug reports.

RQ4 (Usefulness of crashes for bug finding)
Developers acknowledged **60** (confirmed, fixed, or duplicate) out of **int(37 + 28)** bugs that we report in LLVM (37) and GCC (28). Of these, **int(27 + 25)** are uniquely manifested with IssueMut, i.e., the two MetaMut's mutator sets missed them.

It is worth noting that MetaMut's authors report bugs in a longitudinal study, i.e., across several compiler versions and over several months. A side-by-side longitudinal comparison with MetaMut considering number of bugs reported would be unfair and unproductive as we would need to match compiler revisions and MetaMut authors reported bugs in those revisions earlier, so bugs could be already fixed as MetaMut's authors accessed corresponding revisions earlier.

4.7 Threats to Validity

The main threat to *internal validity* consists of the bias that we might have inadvertently introduced in our implementation. To mitigate that threat, we built IssueMut by retrofitting their mutators onto the MetaMut framework and Kitten, our experiment baselines. In addition, our approach relies on large language models

to generate negative tests and mutators, which introduces the risk of LLM hallucinations and model-specific behavior. We mitigate this risk through a combination of manual and automated validation: all LLM-generated negative tests are manually inspected against their issue descriptions and discarded if deemed invalid, and all generated mutators are validated using an automatic mutator validator that checks whether they preserve expected input-output behavior on available test cases. To further assess robustness to changes in LLM versions, we reran the pipeline on a small subset of issues using more recent models (GPT-5.1 and Gemini-3) and observed that the resulting negative tests and mutators passed validation and closely matched the originals. The main threat to *construct validity* is the stochastic nature of the result that could have favored our results. To mitigate that threat we repeat our experiments multiple times with random seeds. The main threat to *external validity* is the selection of issues we selected. We follow a well-defined method to assure that we only analyze bugs that have been already fixed.

5 Discussion

This section reports on a comparison against Fuzz4All, an evaluation of mutators effectiveness across compilers, a discussion on a sample of bugs we find, and lessons we learned.

5.1 Comparison with Fuzz4All [65]

Fuzz4All uses LLMs to generate inputs for fuzzing compilers and other systems. Fuzz4All's workflow has two stages. First, it uses auto-prompting [52, 63, 74] to distill user-provided documentation, examples, or specifications into prompts for querying an LLM. Then, it implements a fuzzing loop that continuously generates inputs iteratively, updating prompts with previously generated inputs and using different strategies. Fuzz4All's compute requirements differs from IssueMut's. Fuzz4All runs on a GPU that fits the LLM, but IssueMut runs on CPUs. So, a side-by-side comparison is impractical. Yet, given Fuzz4All's recency and similar goal, we attempt to replicate and compare its results with IssueMut mutators.

Setup. We run Fuzz4All using its author's original setup in a 24-hour fuzzing campaign, compared to IssueMut's 24 hours, on the same compiler versions as IssueMut: GCC (87492fb) and Clang (9bdf683). Note that Fuzz4All's original evaluation used older versions of these compilers, e.g., they use GCC 13, but we use GCC 15, which is in development. We run Fuzz4All on an NCSA Delta A40 GPU node [40] with a 64-core AMD EPYC 7763 CPU, an NVIDIA A40 GPU, and 128GB RAM, running RHEL 8.8. We repeat our 24-hour campaigns 7 times each for GCC and Clang (including auto-prompting) or a cap of one million generated inputs, whichever comes first. In all experiments, we use the configurations in [65], including using standard C library documentation as input, and the Docker containers provided in the replication package.

Results. Fuzz4All produces 244,722 and 203,483 inputs for Clang and GCC, respectively, which are close to the numbers in [65] (e.g., their five 24-hour GCC campaigns produce 260,221 inputs). Our Fuzz4All campaigns collectively triggered 11 crashes: 7 in Clang and 4 in GCC. Our analysis shows that (i) 9 of these 11 crashes are unique; and (ii) 4 of 9 unique crashes were also triggered by IssueMut (§4.4). We leave as future work the investigation of how Fuzz4All performs in other revisions of GCC and LLVM, and the effects of incorporating Fuzz4All's generated inputs into MetaMut's seed corpora.

5.2 Within-compiler vs. cross-compiler effects

We evaluate the effects of mutators originated from a compiler in a different compiler. Among our reported bugs, we find that 15 out of 37 LLVM bugs (≈40%) are exposed by mutators sourced from GCC bug reports, and 9 out of 28 GCC bugs (≈32%) are exposed by mutators sourced from LLVM bug reports, which suggests meaningful cross-compiler generalization. To further isolate within-compiler and cross-compiler effects under a controlled budget, we conduct a small-scale ablation study. We sample 100 mutators derived from GCC bug reports and 100 mutators derived from LLVM bug reports, and apply them in the four settings defined by the cross product of mutator source {GCC, LLVM} and target compiler {GCC, LLVM}. For each setting, we run IssueMut for 8 hours using 10 CPU cores, resulting in four runs in total. Table 4 summarizes the results of our small-scale ablation. Across both within-compiler and cross-compiler settings, mutators sourced from GCC and LLVM consistently trigger compiler crashes under the same testing budget. To sum up, results show that mutators remain effective when used across compilers, although their effectiveness varies.

Table 4: Effectiveness of mutators across compilers.

Source	Target	Avg. crashes
GCC	GCC	2.25
GCC	LLVM	1.25
LLVM	LLVM	6.00
LLVM	GCC	5.25

5.3 IssueMut-triggered bugs sample

LLVM bug #123410. This IssueMut input triggers a crash in Clang. The crash occurs because mutator **M3** [39] (Figure 1) replaces the `extern` keyword with `static`, changing the behavior of functions in `immintrin.h`. Developers fixed the bug, saying, "*Seems we missed to add AMX FP8 intrinsics into* `X86LowerAMXType.cpp`".

```
#include <mm_malloc.h>
- #define extern
+ #define static
#define __inline
#include <immintrin.h>
```

LLVM bug #120086. This input also triggers a crash in Clang. The crash occurs because applying mutator **M5** [39] results in an undeclared `printf` function and an assertion failure during LLVM IR generation for `__builtin_dump_struct()`. (**M5** replaces a type with `char` while transforming negative to positive test cases.) LLVM developers confirm this as a regression: "*... git bisect points to this commit that caused the regression: ef395a4...We really should land a fix ASAP, this is now three regressions linked to the same change*".

```
#include <stdint.h>
struct A { char i1; };
- int printf(const char *fmt, ...);
+ int printf(char fmt, ...);
void test(struct A *a) {
 __builtin_dump_struct(a, printf);}
```

GCC bug #118868. This input triggers a crash in GCC:

```
#include <stdlib.h>
void *wrapped_malloc(size_t size) {
-   return malloc(size); }
+   return __builtin_assoc_barrier(malloc(size)); }
```

The crash occurs because applying mutator **M1** [39] causes a failure to validate the correctness of generated GIMPLE IR. (That mutator wraps return expressions in a call to `__builtin_assoc_barrier()`.)

GCC developers confirmed the bug, saying, "*the ICE (Internal Compiler Error) started happening since r12-1608-g2f1686ff70b25f.*" So, the bug had been in GCC for over three years.

GCC bug #119001. This input also triggers a crash in GCC:

```
union U4 {
-  char a[4];
+  char a[];
  int i; }; const union U4 u4[2] = {{"123"}};
```

The crash occurs because applying mutator **M7** [39] revealed missing initialization. (That mutator removes literal array size in array declarations.) GCC 15 —under development— adds support for flexible array members in unions, but does not correctly handle their initialization in arrays of unions. Developers fixed the bug, saying "*r15-209 allowed flexible array members inside unions, but as the test case shows, not everything has been adjusted.*".

5.4 Lessons Learned

We list actionable lessons that we learn from our IssueMut work: (1) "Bug history" mutators find many bugs missed by mutators obtained via other means, e.g., through program-agnostic transformations (§ 4.3) or LLMs (§4.4). So, we recommend finding more of these mutators and to use them in existing mutational fuzzers; (2) Most successful mutators reveal only one bug. Likewise, triggering most bugs requires only one mutation (§4.4.2). Bounded-exhaustive fuzzing [3] is a promising direction to explore all combinations of files and (single) mutations, so coverage would be unnecessary, thereby accelerating compilation, the main bottleneck; (3) Identifying "hot" features is a promising direction for bug hunting. For example, we find that mutators related to the recent C23 standard were successful (§ 4.4.1); (4) Fuzzing with only successful mutators is effective (§4.5). Developers could prioritize successful mutators over others in evolutionary contexts;

6 Related Work

Generation-Based Fuzzing. Generational fuzzing methods test compilers by generating inputs and checking properties on them (e.g., absence of crashes). CSmith [67] generates random C programs that have successfully exposed hundreds of latent defects across various C compiler implementations. Analogous to CSmith, CCG [36] produces chaotic C89 code designed specifically to induce compiler crashes through rigorous stress-testing. More recently, YARPGen [27, 28] creates semantically diverse programs free from undefined behavior, effectively addressing the saturation issue encountered with earlier generation tools such as CSmith [1]. Several specialized generational-based fuzzers exist [5, 18–20, 51, 56]. Fuzz4All [65] is a generational LLM-based fuzzer for systems in general. To sum up, generational fuzzers can produce seed corpora for mutational fuzzers. Future work can evaluate such integration.

Mutation-Based Fuzzing. Mutational fuzzing methods test compilers by modifying existing inputs. Orion [24, 24] introduced Equivalence Modulo Inputs (EMI), a methodology that creates semantically equivalent program variants with respect to specific input sets. Athena [25] extends Orion's capabilities by employing Markov Chain Monte Carlo optimization to facilitate the deletion and insertion of code into un-executed regions. Hermes [54] preserves program semantics while mutating live code regions. Skeletal Program Enumeration (SPE) [71] systematically enumerates all possible variable usage patterns within syntactic structure-based skeletons. GrayC [10] implements coverage-guided mutation-based fuzzing with semantics-aware mutation operators. Kitten [66] proposes program-agnostic transformations and a caching mechanism to efficiently evaluate them. Differently from these approaches, which rely on manually designed or heuristic-driven mutation operators, IssueMut extracts mutators from real-world bug reports using a semi-automated approach. Recent work has explored leveraging LLMs for mutational compiler fuzzing. ClozeMaster [13] applies LLMs to infill masked regions of historical bug-triggering programs, while LegoFuzz [41] synthesizes reusable code fragments that can be strategically combined to construct test programs. ATLAS [64] explores compilation space through strategic attribute insertion into test programs. Unlike these LLM-based approaches, which primarily apply LLMs to directly mutate or complete test programs using historical bug information, IssueMut mines reusable mutators from bug reports and applies them across a seed corpus. Other work applies LLMs to generate mutators for compiler testing. MetaMut [44] synthesizes mutators using LLMs. Unlike MetaMut, IssueMut mines mutators directly from real-world bug reports.

Mutation Testing and Bug Fixes. Tufano *et al.* [62] proposed to mutate source code from bug fixes. Their idea is similar to ours but the context is different. Their work is on mutation testing, which focuses on evaluating test-suite quality. IssueMut's purpose is fundamentally different: we focus on creating mutations that guide the exploration of the mutational compiler fuzzing search space towards areas that are likely to trigger bugs.

7 Conclusions

Ensuring reliability of compilers is an important problem. Mutational fuzzers were recently shown to outperform several compiler fuzzers, but their effectiveness depends on the availability of high-quality mutators. We report on a comprehensive study showing that bug histories are good starting points for semi-automated creation of effective mutators. Our study is supported by IssueMut, an approach and an enhanced fuzzer framework that we also propose. IssueMut mines "bug history" mutators and retrofits them into SoTA mutational fuzzers, MetaMut [44] and Kitten [66]. IssueMut finds many bugs that SoTA fuzzers miss. We share several actionable lessons and plans for future work in this direction.

Data Availability

The artifacts of IssueMut are publicly available:
https://github.com/ncsu-swat/IssueMut.

Acknowledgments

We thank reviewers for their comments. This work was partially supported by an Intel Rising Star Faculty Award, a Google Cyber NYC Institutional Research Award, and the U.S. National Science Foundation under Grant Nos. CCF-2045596, CCF-2319473, CCF-2403035, CCF-2525243, CCF-2319472, and CCF-2349961.

References

[1] Alex Groce and John Regehr. 2020. The Saturation Effect. https://blog.regehr.org/archives/1796 Online; accessed Aug 21 2024.
[2] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC*

Conference on Computer and Communications Security (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2329–2344. doi:10.1145/3133956.3134020

[3] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: automated testing based on Java predicates. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis* (Roma, Italy) *(ISSTA '02)*. Association for Computing Machinery, New York, NY, USA, 123–133. doi:10.1145/566172.566191

[4] bug-list [n. d.]. IssueMut Reported Bugs. https://github.com/ncsu-swat/IssueMut/blob/msr-2026/data/Evaluation/RQ4/issuemut_bug_table.pdf.

[5] Stefanos Chaliasos, Thodoris Sotiropoulos, Diomidis Spinellis, Arthur Gervais, Benjamin Livshits, and Dimitris Mitropoulos. 2022. Finding typing compiler bugs. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 183–198. doi:10.1145/3519939.3523427

[6] Harrison Chase. 2022. *LangChain*. https://www.langchain.com Software available from https://www.langchain.com.

[7] 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 (Feb. 2020), 36 pages. doi:10.1145/3363562

[8] 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* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 70, 13 pages. doi:10.1145/3597503.3623343

[9] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated testing of graphics shader compilers. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 93 (Oct. 2017), 29 pages. doi:10.1145/3133917

[10] 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* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1219–1231. doi:10.1145/3597926.3598130

[11] Zhenye Fan, Guixin Ye, Tianmin Hu, and Zhanyong Tang. 2024. History-driven Compiler Fuzzing via Assembling and Scheduling Bug-triggering Code Segments. In *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*. 331–342. doi:10.1109/ISSRE62328.2024.00040

[12] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi

[13] Hongyan Gao, Yibiao Yang, Maolin Sun, Jiangchang Wu, Yuming Zhou, and Baowen Xu. 2025. Clozemaster: Fuzzing Rust Compiler by Harnessing Llms for Infilling Masked Real Programs. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 1422–1435. doi:10.1109/ICSE55347.2025.00175

[14] GCC-bug-issue [n. d.]. GCC Bug Report - 108424. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=108424.

[15] GCC-bug-issue [n. d.]. GCC Bug Report - 108449. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=108449.

[16] GCC-bug-issue [n. d.]. GCC Bug Report - 108777. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=108777.

[17] Jina AI GmbH. [n. d.]. PromptPerfect website. https://promptperfect.jina.ai.

[18] William Hatch, Pierce Darragh, Sorawee Porncharoenwase, Guy Watson, and Eric Eide. 2023. Generating Conforming Programs with Xsmith. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Cascais, Portugal) *(GPCE 2023)*. Association for Computing Machinery, New York, NY, USA, 86–99. doi:10.1145/3624007.3624056

[19] Yann Herklotz and John Wickerson. 2020. Finding and Understanding Bugs in FPGA Synthesis Tools. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '20)*. Association for Computing Machinery, New York, NY, USA, 277–287. doi:10.1145/3373087.3375310

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

[21] ISO/IEC. [n. d.]. C23 Specification (ISO/IEC 9899:2024). https://en.cppreference.com/w/c/23.

[22] Yue Jia and Mark Harman. 2009. Higher Order Mutation Testing. *Information and Software Technology* 51, 10 (2009), 1379–1393. doi:10.1016/j.infsof.2009.04.016 Source Code Analysis and Manipulation, SCAM 2008.

[23] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *Trans. Softw. Eng.* 37, 5 (2011).

[24] 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* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 216–226. doi:10.1145/2594291.2594334

[25] 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* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 386–399. doi:10.1145/2814270.2814319

[26] Shaohua Li, Theodoros Theodoridis, and Zhendong Su. 2024. Boosting Compiler Testing by Injecting Real-World Code. *Proc. ACM Program. Lang.* 8, PLDI, Article 156 (jun 2024), 23 pages. doi:10.1145/3656386

[27] 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 (Nov. 2020), 25 pages. doi:10.1145/3428264

[28] 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 (June 2023), 22 pages. doi:10.1145/3591295

[29] llvm-bug-issue [n. d.]. LLVM Bug Report - 113692. https://github.com/llvm/llvm-project/issues/113692.

[30] llvm-bug-issue [n. d.]. LLVM Bug Report - 69352. https://github.com/llvm/llvm-project/issues/69352.

[31] llvm-bug-issue [n. d.]. LLVM Bug Report - 71911. https://github.com/llvm/llvm-project/issues/71911.

[32] llvm-bug-issue [n. d.]. LLVM Bug Report - 72017. https://github.com/llvm/llvm-project/issues/72017.

[33] Haoyang Ma. 2023. A Survey of Modern Compiler Fuzzing. arXiv:2306.06884 [cs.SE] https://arxiv.org/abs/2306.06884

[34] Michaël Marcozzi, Qiyi Tang, Alastair F. Donaldson, and Cristian Cadar. 2019. Compiler fuzzing: how much does it matter? *Proc. ACM Program. Lang.* 3, OOPSLA, Article 155 (Oct. 2019), 29 pages. doi:10.1145/3360581

[35] William M. McKeeman. 1998. Differential Testing for Software. *Digit. Tech. J.* 10, 1 (1998), 100–107. http://dblp.uni-trier.de/db/journals/dtj/dtj10.html#McKeeman98

[36] Mrktn. 2013. CCG. https://github.com/Mrktn/ccg/.

[37] mutator-distribution [n. d.]. Distribution of Actions for Successful Mutators. https://github.com/ncsu-swat/IssueMut/blob/msr-2026/data/Evaluation/RQ2/issuemut_action.pdf.

[38] mutator-distribution [n. d.]. Distribution of Program Elements for Successful Mutators. https://github.com/ncsu-swat/IssueMut/blob/msr-2026/data/Evaluation/RQ2/issuemut_program_element.pdf.

[39] mutator-list [n. d.]. Representative IssueMut Mutator List. https://github.com/ncsu-swat/IssueMut/blob/msr-2026/data/Evaluation/RQ2/issuemut_successful_mutators.md.

[40] National Center for Supercomputing Applications. 2023. Delta: System Architecture. https://docs.ncsa.illinois.edu/systems/delta/en/latest/user_guide/architecture.html Accessed: 2025-03-11.

[41] Yunbo Ni and Shaohua Li. 2025. Interleaving Large Language Models for Compiler Testing. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 301 (Oct. 2025), 27 pages. doi:10.1145/3763079

[42] OpenAI. [n. d.]. GPT 4o-mini. https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/.

[43] OpenAI. [n. d.]. GPT4. https://openai.com/index/gpt-4/.

[44] Xianfei Ou, Cong Li, Yanyan Jiang, and Chang Xu. 2024. The Mutators Reloaded: Fuzzing Compilers with Large Language Model Generated Mutation Operators. ASPLOS.

[45] Wendkûuni C. Ouédraogo, Laura Plein, Kader Kabore, Andrew Habib, Jacques Klein, David Lo, and Tegawendé F. Bissyande. 2024. Extracting Relevant Test Inputs from Bug Reports for Automatic Test Case Generation. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings* (Lisbon, Portugal) *(ICSE-Companion '24)*. Association for Computing Machinery, New York, NY, USA, 406–407. doi:10.1145/3639478.3643537

[46] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 329–340. doi:10.1145/3293882.3330576

[47] Jens Palsberg and C. Barry Jay. 1998. The Essence of the Visitor Pattern. In *Proceedings of the 22nd International Computer Software and Applications Conference (COMPSAC '98)*. IEEE Computer Society, USA, 9–15.

[48] prompt [n. d.]. Mutator Creator Prompt. https://github.com/ncsu-swat/IssueMut/blob/msr-2026/src/mutator_mining/Mutator_Creator_prompt.txt.

[49] prompt [n. d.]. Mutator Refiner Prompt. https://github.com/ncsu-swat/IssueMut/blob/msr-2026/src/mutator_mining/Mutator_Refiner_prompt.txt.

[50] Md Rafiqul Islam Rabin and Mohammad Amin Alipour. 2021. Configuring test generators using bug reports: a case study of GCC compiler and Csmith. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing* (Virtual Event, Republic of Korea) *(SAC '21)*. Association for Computing Machinery, New York, NY, USA, 1750–1758. doi:10.1145/3412841.3442047

[51] Mayank Sharma, Pingshi Yu, and Alastair F. Donaldson. 2023. RustSmith: Random Differential Compiler Testing for Rust. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA,

USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1483–1486. doi:10.1145/3597926.3604919

[52] Taylor Shin, Yasaman Razeghi, Robert L. Logan IV, Eric Wallace, and Sameer Singh. 2020. AutoPrompt: Eliciting Knowledge from Language Models with Automatically Generated Prompts. arXiv:2010.15980 [cs.CL] https://arxiv.org/abs/2010.15980

[53] Nima Shiri Harzevili, Mohammad Mahdi Mohajer, Moshi Wei, Hung Viet Pham, and Song Wang. 2024. History-Driven Fuzzing for Deep Learning Libraries. *ACM Trans. Softw. Eng. Methodol.* 34, 1, Article 19 (Dec. 2024), 29 pages. doi:10.1145/3688838

[54] 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* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 849–863. doi:10.1145/2983990.2984038

[55] Maolin Sun, Yibiao Yang, Ming Wen, Yongcong Wang, Yuming Zhou, and Hai Jin. 2023. Validating SMT Solvers via Skeleton Enumeration Empowered by Historical Bug-Triggering Inputs. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 69–81. doi:10.1109/ICSE48619.2023.00018

[56] Azul Systems, Mohammad R. Haghighat, Dmitry Khukhro, Andrey Yakovlev, Nina Rinskaya, and Ivan Popov. 2013. JavaFuzzer. https://github.com/AzulSystems/JavaFuzzer/. Original authors: Mohammad R. Haghighat, Dmitry Khukhro, Andrey Yakovlev (Intel Corporation); 2017-2018 modifications by Nina Rinskaya, Ivan Popov (Azul Systems).

[57] Calculator.net team. [n. d.]. Sample Size Calculator. https://www.calculator.net/sample-size-calculator.html.

[58] GCC team. [n. d.]. GCC test suite. https://github.com/gcc-mirror/gcc/tree/master/gcc/testsuite.

[59] LLVM team. [n. d.]. Compiling with coverage enabled. https://clang.llvm.org/docs/SourceBasedCodeCoverage.html#id3.

[60] LLVM team. [n. d.]. Introduction to the Clang AST. https://clang.llvm.org/docs/IntroductionToTheClangAST.html.

[61] LLVM team. [n. d.]. LLVM-clang test suite. https://github.com/llvm/llvm-project/tree/main/clang/test.

[62] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. Learning How to Mutate Source Code from Bug-Fixes. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 301–312. doi:10.1109/ICSME.2019.00046

[63] Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R. Lyu. 2022. No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 382–394. doi:10.1145/3540250.3549113

[64] Jiangchang Wu, Yibiao Yang, Maolin Sun, and Yuming Zhou. 2025. Unveiling compiler faults via attribute-guided compilation space exploration. In *Proceedings of the 2025 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '25)*. USENIX Association, USA, Article 65, 17 pages.

[65] 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* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 126, 13 pages. doi:10.1145/3597503.3639121

[66] Yuanmin Xie, Zhenyang Xu, Yongqiang Tian, Min Zhou, Xintong Zhou, and Chengnian Sun. 2025. Kitten: A Simple Yet Effective Baseline for Evaluating LLM-Based Compiler Testing Techniques. In *Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Clarion Hotel Trondheim, Trondheim, Norway) *(ISSTA Companion '25)*. Association for Computing Machinery, New York, NY, USA, 21–25. doi:10.1145/3713081.3731731

[67] 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* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 283–294. doi:10.1145/1993498.1993532

[68] Guixin Ye, Tianmin Hu, Zhanyong Tang, Zhenye Fan, Shin Hwei Tan, Bo Zhang, Wenxiang Qian, and Zheng Wang. 2023. A Generative and Mutational Approach for Synthesizing Bug-Exposing Test Cases to Guide Compiler Fuzzing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1127–1139. doi:10.1145/3611643.3616332

[69] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. 2020. EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2307–2324. https://www.usenix.org/conference/usenixsecurity20/presentation/yue

[70] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2019. *The Fuzzing Book*.

[71] 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* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 347–361. doi:10.1145/3062341.3062379

[72] Yingquan Zhao, Zan Wang, Junjie Chen, Mengdi Liu, Mingyuan Wu, Yuqun Zhang, and Lingming Zhang. 2022. History-driven test program synthesis for JVM testing. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1133–1144. doi:10.1145/3510003.3510059

[73] Hao Zhong. 2023. Enriching Compiler Testing with Real Program from Bug Report. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 40, 12 pages. doi:10.1145/3551349.3556894

[74] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2023. Large Language Models Are Human-Level Prompt Engineers. arXiv:2211.01910 [cs.LG] https://arxiv.org/abs/2211.01910