Detecting Error Diagnostic Defects in C Compiler via Invalid Program Mutation

Yuxiao Peng[1], Bixin Li[2*], Zhihao Xu[2*], Lulu Wang[2], Li Liao[2], Ying Zhou[2]
[1]College of Software Engineering, Southeast University, Nanjing, China
[2]School of Computer Science and Engineering, Southeast University, Nanjing, China
Email: {220246428, bx.li, Hosea.xu, wanglulu, lliao, zy}@seu.edu.cn

***Abstract*—As a crucial part of outputs in C compilers, compiler diagnostics are widely used by developers to locate syntax errors and resolve build failures. However, inconsistent diagnostics across compilers can mislead root cause analysis, prolong debugging, and reduce development efficiency. Existing diagnostic testing techniques primarily focus on warning diagnostics and rely on valid programs, which limits their ability to uncover error diagnostic defects exposed by invalid programs. Although the prior method has explored error diagnostic defects, it still suffers from limited invalid program diversity and redundant exploration, making it inefficient to expose more error diagnostic defects. To address these limitations, we propose EIDETIC, an effective framework for detecting error diagnostic defects in C compilers. Specifically, EIDETIC consists of three components: a Semantic Mutation Component that generates invalid mutants using semantic mutators to extend the invalid program diversity, an Adaptive Mutation Scheduling Component that leverages code embeddings and an MAB strategy to prioritize effective mutation strategies, and a Differential Diagnosis Component that adopts a hierarchical differential strategy to improve the precision of defect detection across compilers. Over a three-month evaluation, the results show that EIDETIC consistently outperforms four baseline methods and discovers 10 error diagnostic defects in recent GCC and Clang releases, 7 of which have been confirmed.**

***Index Terms*—Compiler, Testing, Error Diagnostic, Program Mutation**

I. Introduction

As a fundamental software of the software development process, C compilers play a critical role in code generation and translation [1]. Prior works [2], [3] indicate that both novice and experienced programmers use the compiler output information, which is called diagnostics, to hypothesize the causes of compiler errors. In industrial practice, Google developers also frequently address routine build failures by inspecting compiler-diagnosed code issues [4]. However, inconsistent diagnostics may mislead root-cause analysis, prolong debugging, and reduce development efficiency [5], [6].

Recently, several methods have been proposed to detect diagnostic inconsistency and ensure the accuracy of compiler diagnostics. These methods focus on the warning diagnostic inconsistency. For instance, Epiphron [7] constructs syntactically valid test programs that can be compiled by inserting warning-free code blocks into conditional branches to detect unexpected warnings. Building upon this foundation, DIPROM [8] further employs diversity-guided mutation to systematically explore a broader range of code patterns that trigger heuristic-based diagnostics of compilers. By the diversity-guided mutation, DIPROM identified 8 warning diagnostic defects in two months. However, there are two parts of diagnostics in compilers, which are warnings and errors. A warning diagnostic usually indicates a potentially problematic construct; the compiler usually continues compilation, but the warning alerts developers to issues that may affect correctness or maintainability [9], [10]. On the contrary, an error diagnostic reports code that the compiler cannot legally or reliably translate, so successful compilation of the affected unit is blocked until the error is resolved, even if the compiler continues temporarily for error recovery. Consequently, existing warning diagnostic defect detection methods, which rely on well-formed and compilable programs, are not well-suited to detecting error diagnostic inconsistencies that must be exposed by invalid programs. To address this gap, Tang et al. proposed CERTest [11], which starts from well-formed seed programs and deliberately injects syntactic errors through program mutation, then detects compiler error diagnostic defects by differentially comparing the emitted recovery diagnostics across compilers. CERTest discovered 9 error diagnostic defects in recent GCC and Clang versions, 5 of which were confirmed by developers.

However, existing methods still have limitations that hinder further detection of error diagnostic defects. First, the diversity of the generated invalid programs remains limited. Although CERTest is able to construct invalid programs for testing compiler error diagnostics, its mutation strategy primarily relies on invalid mutations of symbols, operators, and variable attributes. Such mutations are effective at triggering syntax-level diagnostics, the mutators of CERTest, which can be called syntax mutators. But they can only offer limited support for exposing error diagnostic defects rooted in context-sensitive semantic inconsistencies, such as type-relation conflicts, binding errors, and long-range constraint violations. This limitation restricts the defects explored, because most of the error diagnostic defects in the compiler often require coordinated changes across semantic variations like type relations, binding constraints, conversion behaviors, and context-dependent language rules to be exposed [12]–[14]. Therefore,

*Corresponding authors: Bixin Li and Zhihao Xu.

effectively increasing program diversity to reach more error diagnostic defects becomes our first challenge.

On the other hand, the testing efficiency of existing methods is still limited by redundant exploration. Although CERTest leverages historical buggy mutation configurations to guide mutation selection, it does not directly learn which mutation strategies are most effective for exposing new defects. Instead, it prioritizes candidates according to a distance-based diversity heuristic. Such a strategy can improve exploration to some extent, but it may allocate substantial testing budget to mutation configurations that appear diverse yet yield limited defect-discovery benefit. Moreover, its distance measurement only considers mutator usage while ignoring mutation positions and structural context, which further handicaps the exploration of more effective mutation strategies [8], [15]–[17]. Therefore, how to reduce redundant testing and adaptively select more effective mutation strategies becomes our second challenge.

To address this challenge, we propose EIDETIC (d*E*tect*I*ng *D*iagnostic d*E*fec*T*s *I*n *C* compiler): an effective method to generate invalid test programs that detect error diagnostic defects in the C compiler. EIDETIC consists of three components: a Semantic Mutation Component (SMC), an Adaptive Mutation Scheduling Component (AMSC), and a Differential Diagnosis Component (DDC). To generate invalid programs that can expose deeper error diagnostic defects, EIDETIC employs SMC to transform well-formed seed programs into invalid mutants. Specifically, SMC first identifies semantically sensitive program positions in a seed program and then applies 63 semantic mutators that target type relations, binding constraints, conversion behaviors, and context-dependent language rules. To improve testing efficiency and reduce redundant exploration, EIDETIC employs AMSC to adaptively prioritize mutation strategies for generating mutants. AMSC uses a code embedding model to characterize the semantic diversity of candidate mutants and formulates mutation scheduling as a multi-armed bandit (MAB) problem, so that exploration of under-explored strategies and exploitation of high-yield ones can be balanced according to rewards of program diversity. Finally, to accurately identify error-diagnostic defects, EIDETIC employs DDC to compare the diagnostics emitted for the same mutant across multiple C compilers. Because diagnostics for invalid programs often vary across compilers in wording and reported program details, raw text comparison cannot provide an accurate test oracle. DDC therefore adopts a hierarchical differential strategy to improve the precision of inconsistency detection.

To evaluate the performance of EIDETIC, we conduct an extensive study over the two most popular C compilers, GCC [18] and Clang [19]. Evaluation results show that EIDETIC performs better than four baselines, i.e., DIPROM [8], CERTest [11], AFL++ [15], and HiCOND [20]. In total, in the two-week comparative experiments conducted under the GCC and Clang compilers, EIDETIC detected 2, 3, 2, and 2 more error diagnostic defects, respectively, compared to CERTest, DIPROM, AFL++, and HiCOND. In particular, EIDETIC has detected 10 error diagnostic defects in the recent release versions of GCC and Clang, where 7 of them have been confirmed by developers.

To sum up, this work makes the following major contributions:

- We propose EIDETIC—a diagnostic-oriented framework that generates invalid C programs through mutation and reveals inconsistencies in error diagnostics across different compilers.
- We propose a MAB-guided mutation method that treats mutators as arms and uses an embedding-based diversity reward to adapt mutator selection.
- We conduct extensive experiments on GCC and Clang and show that EIDETIC is more effective than the comparison approaches in detecting compiler error diagnostic defects.
- We apply EIDETIC to the recently released versions of GCC and Clang. EIDETIC helps developers detect 10 error diagnostic defects, 7 of which have been confirmed.

II. BACKGROUND AND MOTIVATION

This section introduces the foundations of compiler error diagnostics and clarifies why testing error diagnostic consistency is necessary. It then motivates our problem setting and defines the key requirements that guide the design of our methodology.

A. Background of Compiler Error Diagnostics

In this section, we introduce the background of the compiler error diagnostics, including the importance of compiler error diagnostics, the principles of compiler error diagnostics, the categories of compiler error diagnostics defects, and the compiler error diagnostic testing.

Compiler error diagnostics are crucial for program development [6], [21], [22]. When code contains syntactic or semantic errors, the compiler should terminate compilation and output diagnostics to help developers pinpoint the exact location and category of the erroneous code segment, understand the cause of the compilation failure, and apply the correct measures [23], [24]. Therefore, the quality of error diagnostic messages directly impacts development efficiency: precise error diagnostics shorten debugging time, while inaccurate localization leads to repeated trial-and-error attempts, potentially causing code to become increasingly convoluted until it breaks down entirely [6], [22]. As a result, improving the correctness and consistency of error diagnostics is important not only for compiler quality but also for productivity and toolchain interoperability.

A compiler outputs an error diagnostic when it detects a violation of language constraints during compilation, typically in parsing and semantic analysis [9]. Diagnostics usually include a source location and a textual description; some compilers also attach a diagnostic identifier or rule name [24]–[26]. In general, error diagnostics stem from either syntactic violations or semantic violations, where the latter arise after successful parsing but fail subsequent static checks such as type consistency and declaration compatibility. Diagnostic

```c
#include <stdio.h>
#include <stdint.h>
void test(void)
{
    float *p = (float*)0;
    /* arbitrary pointer expression */
    int64_t x = p;
    /* initialize integer from pointer, no explicit
    cast */
    (void)x;
}
```

Listing 1: A reduced mutant with an illegal integer initialization from a pointer expression (ID #123395)

outcomes can be sensitive to implementation details, making diagnostic behavior a distinct aspect of compiler quality.

According to how the reported errors deviate from expected behavior, error diagnostic defects can be grouped into three common categories: erroneous errors, spurious errors, and missing errors. Erroneous errors include inaccurate locations or confusing explanations that mischaracterize the violation [7], [11], [21]. Spurious errors refer to unjustified rejections of otherwise valid inputs, which manifest as false positive errors. Missing errors arise when the compiler fails to output an expected error for an invalid construct, either by silently accepting the program or by terminating with incomplete or suppressed diagnostics [11], [27].

Compiler error diagnostic testing seeks to expose diagnostic defects by generating or transforming programs that trigger errors and then comparing the resulting diagnostic outputs. Differential testing is widely used: the same input is compiled under multiple compilers, and inconsistencies are treated as candidates for defects [28], [29]. Because diagnostics differ in surface form, effective testing requires structured parsing and normalization into a shared category space, followed by an alignment criterion that determines whether the reported errors are consistent in terms of location and category.

B. Motivation of Error Diagnostics

Two confirmed defect examples are discussed in this subsection to motivate and illustrate EIDETIC. In this study, we only present the reduced versions of the defects because the original mutants are too large for presentation (with over thousands of lines).

Example 1. As shown in Listing 1, this example is produced by applying a semantic mutator in the conversion behaviors category to a valid C program. Concretely, the mutator rewrites the initializer of an int64_t variable by replacing the original integer value with a float* expression. It violates the C constraint on initialization: a pointer value cannot be used to initialize an integer object without a cast.

When compiling the mutant, Clang rejects it and reports an error at the mutated initialization site, consistent with the C constraint that disallows such initialization. In contrast, GCC fails to accurately report the defect and completes the compilation. This discrepancy represents an error diagnostic defect that is particularly detrimental because it may allow invalid code to conceal invalid-memory patterns. This defect is difficult to expose with the warning diagnostic defect-detecting methods because it is based on a mutant with a parsable but semantically invalid construct. Our method finds this defect because it uses a complex mutator that performs semantic mutations on the program, which passes parsing, thereby exposing deeper error diagnostic defects.

```c
#include <stdio.h>
int main()
{
    int k = 1, i = 2, j = 3;
    printf(k, i, j, "index = [%d][%d][%d]\n");
    return 0;
}
```

Listing 2: A reduced mutant with an illegal printf call due to a mismatched argument type (ID #123232)

Example 2. As shown in Listing 2, this example is produced by applying a semantic mutator in the type relations category to a valid C program. Specifically, the mutator rewrites a printf function call statement by replacing the string parameter with an integer variable. This violates the C binding constraint: the first parameter of printf must be a const char* format string, and an int argument cannot be bound to that parameter.

When compiling the mutant, Clang rejects it and reports an error at the mutated call site, consistent with the C constraint that disallows such a call. However, GCC only reports a warning for the same call and still completes compilation with a note. This divergence represents an error diagnostic defect because the same binding-constraint violation caused by the semantic mutator is treated with inconsistent diagnostics across compilers. This defect is difficult to expose with the warning diagnostic defect-detecting methods, which typically focus on compilable programs and warning outputs. Our method finds this defect by applying a semantic mutator that keeps the program parsable but makes the call incorrect, thus exposing deeper error diagnostic defects.

III. FRAMEWORK

In this section, we first present an overview of the framework of EIDETIC. We then explain the Semantic Mutation Component (SMC) and the Adaptive Mutation Scheduling Component (AMSC) to address the two challenges. Finally, the details of the Differential Diagnosis Component (DDC) are presented.

A. Overview

Our method targets error diagnostic defects by systematically generating syntactically valid but semantically invalid C programs, then comparing how compilers report errors. As shown in Fig. 1, the workflow consists of three components: (i) SMC constructs a clean seed pool and transforms well-formed seeds into invalid mutants via abstract syntax tree (AST)-guided semantic mutators, (ii) AMSC adaptively schedules mutators using embedding-based diversity rewards and a MAB policy to reduce redundant exploration, and (iii)

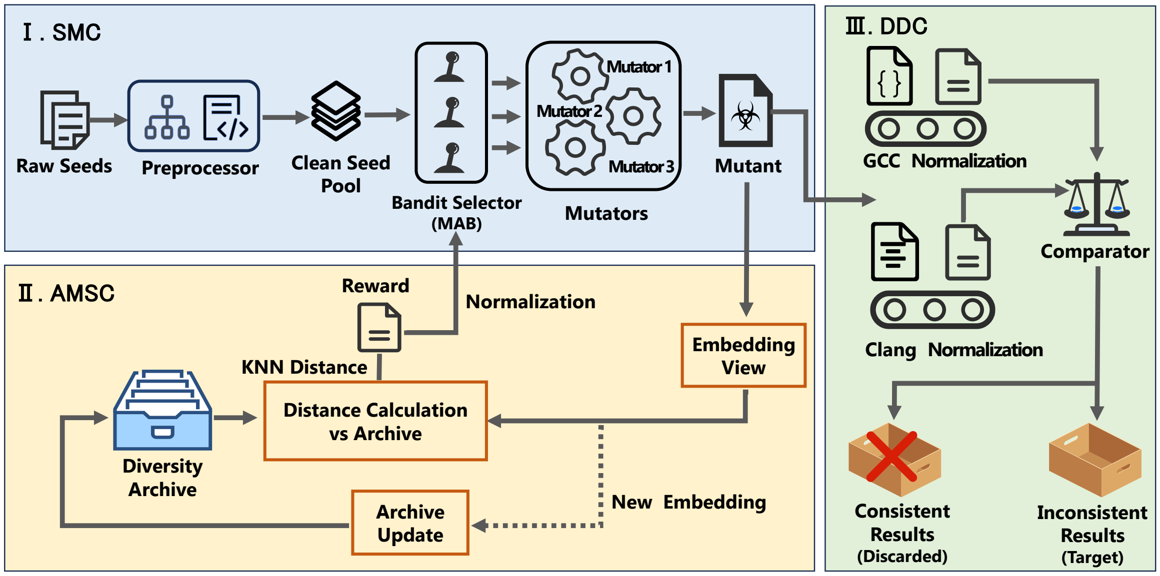


Fig. 1: Framework of our method

DDC performs differential diagnosis by normalizing compiler diagnostics and comparing them hierarchically to identify error diagnostic defects.

B. Semantic Mutation Component

SMC is responsible for producing invalid programs that still reach deep semantic checks, so that we can detect deeper error diagnostic defects.

III-B.1 Preprocessing

Our framework starts from compilable base programs and systematically transforms them into illegal test programs that reach deep diagnostic logic. Preprocessing serves two purposes: (i) building a stable, clean seed pool that compilers can parse and type-check far enough to expose meaningful diagnostics, and (ii) enriching the syntactic surface of seeds so that later mutations have more opportunities to trigger diverse classes of errors.

We generate an initial set of C programs using an automated C program generator as raw seeds. These seeds are intended to be compilable, providing a reliable baseline for subsequent transformations. We then apply a syntax-safe enrichment pass that inserts simple placeholders to increase the availability of structural contexts, thereby facilitating the execution of mutation operations in the next stage.

The enriched seeds are validated using lightweight compilation checks on the compilers. Seeds that compile successfully are retained as the clean seed pool. This pool becomes the sole input to the mutation stage, ensuring that any subsequent compilation failures are attributable to our mutations.

III-B.2 Code Mutation

The core of the framework is a mutation engine that transforms compilable programs into illegal ones, while using diversity-guided feedback to select mutators adaptively.

The mutation stage converts each clean seed into an illegal program through the application of one or more mutators guided by AST. Unlike warning-oriented mutation strategies that restructure code while preserving compilability in order to diversify warning manifestations, the mutators in this work are explicitly constructed to violate the semantics.

Each mutator is specified by three elements. First, it identifies a target site in the AST based on the semantic constraints that the site must satisfy. Second, it performs a localized rewrite that takes the form of insertion, deletion, duplication, or replacement. Third, it is associated with an anticipated class of diagnostic outcomes. The rewrites are deliberately confined to small regions of the program. This locality preserves strong correspondence with the original seed and reuses the surrounding context so that the produced mutants remain close to realistic code.

We group our mutators into four categories, which keeps the design compact while covering different sources of semantic diagnostics. To construct mutators systematically, we start from C constraint violations that are likely to trigger front-end error diagnostics, and organize them according to semantic sources. Within each category, we instantiate concrete mutators by selecting representative constraint violations that can be introduced through localized AST rewrites while keeping the surrounding program parsable. This process results in 63 semantic mutators.

Type-relations mutators. These mutators violate typing relations while preserving the surrounding declarations and use sites. We introduce conflicting declarations, incompatible redeclarations, and inconsistent pointer and array shapes. The type-relations mutators are useful for exercising deeper type checking and for observing the error diagnostic defect of a type conflict.

Binding-constraints mutators. These mutators break the link between a name and its declaration. We delete or move a required declaration, definition, or label, leaving later uses unchanged. The binding-constraints mutators are useful for exercising binding checks and for observing the error diagnostic defect of incorrect name resolution.

Conversion-behaviors mutators. These mutators violate conversion rules by perturbing casts and implicit conversions at precise use sites. We poison implicit conversions and casts at specific sites, such as assignments, argument passing, and return statements. The conversion-behaviors mutators are useful for exercising conversion checks and for observing the error diagnostic defect of misleading conversion diagnostics.

Context-dependent-language-rules mutators. These mutators violate language rules that depend on the surrounding context. We introduce illegal constructs that depend on the surrounding context, such as non-constant initializers, invalid designators, and invalid returns. The context-dependent-language-rules mutators are useful for exercising context-sensitive checks and for observing the error diagnostic defect of incorrect rule enforcement.

As shown in Table I, we list representative mutators from all categories. Overall, our mutator set is constructed to generate compact, realistic mutants that consistently trigger meaningful diagnostics, enabling fine-grained comparison across compilers.

C. Adaptive Mutation Scheduling Component

AMSC improves testing efficiency by prioritizing mutators that yield diverse mutants, while still exploring underused mutators to avoid premature convergence. Algorithm 1 summarizes the MAB-guided mutation workflow.

III-C.1 MAB-guided Mutator Selection

A core challenge is that mutators vary widely in effectiveness. To reduce the reliance on handcrafted schedules, mutator selection is formulated as an MAB problem, where each mutator corresponds to an arm, and each mutation attempt yields feedback reflecting the utility of the generated mutant.

Let $\mathcal{A} = \{a_1, \ldots, a_M\}$ denote the set of mutators. For each arm $a \in \mathcal{A}$, we maintain a probabilistic belief of its expected utility, and we select a small subset of arms to mutate a seed program. After the mutant is evaluated, the resulting reward is used to update the belief of the chosen arms.

In this work, we employ Thompson Sampling (TS) due to its principled exploration-exploitation trade-off and minimal parameterization. Specifically, each arm a is associated with a Beta posterior $\mathrm{Beta}(\alpha_a, \beta_a)$, and TS draws one sample from each posterior to estimate the arm's plausibility of being optimal.

The TS score of arm a is drawn as

$$\theta_a \sim \mathrm{Beta}(\alpha_a, \beta_a). \tag{1}$$

Given the sampled scores $\{\theta_a\}$, the framework selects the top-m arms with the largest sampled values, where m is small and fixed for the mutation budget.

Algorithm 1: MAB-guided mutation with archive-relative diversity reward

Input: $\mathcal{S}$, clean seed pool (each seed is a compilable C program); $\mathcal{A}$, set of mutators (arms), $|\mathcal{A}| = M$; $\mathcal{E}$, diversity archive storing embeddings of previously accepted mutants; T, target number of mutants to generate; k, number of nearest neighbors used in reward computation

Output: $\mathcal{P}$, set of generated invalid test programs ($|\mathcal{P}| = T$)

State: $\mathcal{B}$, bandit state for arms in $\mathcal{A}$;
INITIALIZEBANDIT($\mathcal{B}, \mathcal{A}$);
$\mathcal{P} \leftarrow \emptyset$;
while $|\mathcal{P}| < T$ **do**
  $s \leftarrow$ NEXTSEED($\mathcal{S}$);
  $m \leftarrow$ SAMPLEBUDGET($[m_{\min}, m_{\max}]$);
  $Ops \leftarrow$ SELECTARMSTS($\mathcal{B}, m$);
  $p \leftarrow$ APPLYMUTATIONS(s, Ops);
  $v \leftarrow$ BUILDEMBEDDINGVIEW(p);
  $\mathbf{e} \leftarrow$ EMBED(v);
  $r \leftarrow$ REWARDFROMARCHIVE($\mathbf{e}, \mathcal{E}, k$);
  UPDATEBANDIT($\mathcal{B}, Ops, r$);
  UPDATEARCHIVE($\mathcal{E}, \mathbf{e}$);
  $\mathcal{P} \leftarrow \mathcal{P} \cup \{p\}$;
end
return $\mathcal{P}$

$$S \leftarrow \mathrm{Top}_m(\{\theta_a \mid a \in \mathcal{A}\}). \tag{2}$$

After obtaining a reward $r \in [0, 1]$ for the mutant, the posterior parameters of the selected arms are updated. To support continuous rewards without introducing additional modeling complexity, we adopt a fractional Bernoulli interpretation, updating each chosen arm by the same reward share. S denotes the selected set of arms.

For each $a \in S$,

$$\alpha_a \leftarrow \alpha_a + \frac{r}{|S|}, \qquad \beta_a \leftarrow \beta_a + \left(1 - \frac{r}{|S|}\right). \tag{3}$$

This update preserves the essential behavior of TS: arms with limited observations retain higher posterior uncertainty and thus remain selectable, preventing premature convergence to a small subset of mutators.

III-C.2 Diversity Reward via Code Embeddings and the Archive

A core requirement of MAB-guided mutation is a reward that correlates with search progress. In our framework, progress is measured as diversity in representation space, rather than simple syntactic difference. Concretely, each mutant program is transformed into an embedding view—a normalized textual representation intended to reduce irrelevant variability while preserving salient structural and semantic cues for similarity comparison. The embedding view is then

TABLE I
Typical mutators

Mutation type	**Typical mutator**	**Example (before → after)**	**Intended diagnostic**
Type relations	`insert_conf_func_decl`	`int foo(int);` → `int foo(int); double foo(double);`	conflicting types for foo
	`arraypointer_subscript`	`int a[4][5]; int v=a[1][2];` → `int *a; int v=a[1][2];`	subscripted value is not array/ pointer (invalid indexing chain)
Binding constraints	`vardecl_delete`	`int x=0; return x;` → `/* delete decl */ return x;`	use of undeclared identifier `x`
	`delete_label_stmt`	`goto L; L: x++;` → `goto L; /* label removed */ x++;`	label `L` used but not defined
Conversion behaviors	`insert_illegal_cast`	`int *p;` → `int *p; int x=(int)p;`	illegal cast/pointer to integer conversion
	`add_const_keep_write`	`int x=0; x=1;` → `const int x=0; x=1;`	assignment of read-only variable `x`
Context-dependent language rules	`insert_nonconst_global`	`int g=0;` → `int x; int g=x;`	initializer element is not constant
	`insert_return_in_void`	`void f(){ ... }` → `void f(){ return 1; }`	return-statement with a value, in function returning `void`

mapped to a fixed-dimensional vector by a code embedding model.

The reward is computed by comparing the mutant's embedding against a diversity archive, which stores embeddings of previously observed mutants. The archive is not merely a log; it acts as a reference distribution that defines what has already been explored, and it supports stable reward scaling as the experiment progresses. Given a new embedding vector $\mathbf{e}$ and an archive $\mathcal{E} = \{\mathbf{e}_1, \dots, \mathbf{e}_N\}$, we compute pairwise cosine distances. The cosine distance between $\mathbf{e}$ and an archived vector $\mathbf{e}_i$ is defined as:

$$d(\mathbf{e}, \mathbf{e}_i) = 1 - \frac{\mathbf{e} \cdot \mathbf{e}_i}{\|\mathbf{e}\| \, \|\mathbf{e}_i\|}. \tag{4}$$

To reduce sensitivity to outliers and to focus on local neighborhood novelty, we use the mean distance to the k nearest neighbors in the archive. Let $d_{(1)} \leq \cdots \leq d_{(N)}$ denote the sorted distances. The KNN mean distance is computed as:

$$\bar{d}_k(\mathbf{e}) = \frac{1}{k} \sum_{j=1}^{k} d_{(j)}. \tag{5}$$

Raw distance magnitudes may drift with archive size and embedding distributions. To obtain a stable reward in $[0, 1]$ with minimal manual thresholding, we normalize $\bar{d}_k(\mathbf{e})$ using percentiles of the full distance set. Let P_{10} and P_{90} denote the 10th and 90th percentiles of $\{d(\mathbf{e}, \mathbf{e}_i)\}_{i=1}^{N}$. The normalized reward is then computed as:

$$r(\mathbf{e}) = \mathrm{clip}\left(\frac{\bar{d}_k(\mathbf{e}) - P_{10}}{P_{90} - P_{10}}, \, 0, \, 1 \right). \tag{6}$$

This archive-relative reward has two practical advantages. First, it directly captures whether a mutant expands the explored region in embedding space, which aligns with the objective of generating diverse invalid programs. Second, it yields rewards on a comparable scale over time, enabling the bandit to update arms without requiring manual renormalization.

After reward computation, we update the bandit state for the selected mutators and then insert the new embedding into the archive. The archive update is therefore downstream of reward computation: the archive is read to compute distances and written to incorporate newly generated diversity. In addition, an embedding cache keyed by a stable hash of the embedding view can be used to reuse embeddings for identical content, thereby reducing repeated embedding calls without changing the reward definition.

D. Differential Diagnosis Component

This stage targets error-level diagnostic inconsistencies. For each mutant program, we invoke compiler front-ends in syntax-only mode to collect structured diagnostics. We then normalize diagnostics to a common representation and apply a comparator that focuses on where the error is reported and what category it belongs to.

Because compilers differ in diagnostic formats, wording, and option identifiers, we normalize each diagnostic into a compact tuple containing: severity, source location, and a normalized category derived from rule-based mapping. This normalization enables robust comparisons even when the original diagnostic messages are not directly comparable.

For cross-compiler testing, each mutant is compiled by GCC and Clang. The two resulting normalized diagnostic sets are compared using a location-aware rule: two diagnostics are treated as consistent if they report the same error line and exhibit compatible categories at that location. Mutants that violate this criterion are retained as potential cases for further analysis.

IV. EVALUATION

In this section, four experiments are conducted to evaluate the effectiveness of EIDETIC. Specifically, our evaluation aims at answering the following Research Questions (RQs).

- **RQ1:** How is the defect-detecting capability of EIDETIC compared to state-of-the-art methods?
- **RQ2:** Can EIDETIC detect real error diagnostic defects in the latest GCC and Clang?
- **RQ3:** How effective is the multi-mutation strategy of EIDETIC over single-mutation?
- **RQ4:** How effective is the MAB-guided sampling strategy used by EIDETIC?

In our experiment, RQ1 and RQ2 are used to evaluate the defect-detecting capability of EIDETIC compared to the state-of-the-art methods. RQ3 and RQ4 are employed to evaluate the multi-mutation strategy and the MAB-guided sampling strategy of the code mutation component.

A. Seed Programs

To evaluate the effectiveness of EIDETIC, we construct a seed pool of C programs using CsmithEdge. The seed programs are designed to be well-formed and compilable, so that subsequent mutation can reliably introduce controlled invalidity and exercise deep diagnostic checks. We filter out seeds that fail to compile and retain the remaining programs as the final seed pool.

B. Baselines

We compare EIDETIC against DIPROM [8], CERTest [11], AFL++ [15], and HiCOND [20]. They are four state-of-the-art methods for diagnostic testing. We reproduce them with the source code provided by their works and use their default configurations.

C. Experimental Configuration

Our evaluation was conducted on a machine running Ubuntu 22.04 system, equipped with an Intel Core i7-12700 CPU @4.9 GHz. In the experiment, we evaluate EIDETIC on GCC and Clang, two widely used C compilers studied in prior work [20], [30], and we use their recent releases (i.e., GCC 15.1.0 and Clang 22.0.0) because compiler developers usually enhance new compiler error diagnostics and prefer to fix defects in the recent releases rather than in stable versions. Consequently, error diagnostic defects discovered in the recent compiler releases have greater value, motivating us to investigate whether EIDETIC can detect compiler error diagnostic defects in practice.

For embedding-based diversity measurement, we use Qwen text-embedding-v4 to obtain 1024-dimensional vectors for normalized program views.

TABLE II
Number of error diagnostic defects detected by five methods

Method	**GCC**		**Clang**		**Total**
	New	**Known**	**New**	**Known**	
EIDETIC	1	2	1	0	4
CERTest	0	1	0	1	2
DIPROM	0	1	0	0	1
AFL++	0	2	0	0	2
HiCOND	1	1	0	0	2

D. Answer to RQ1

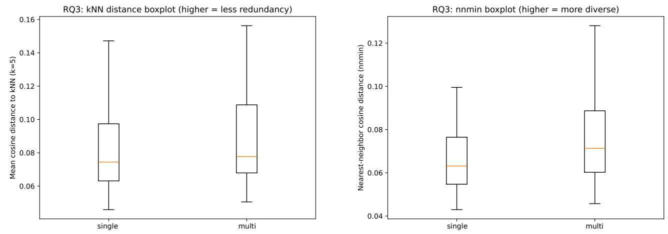
Approach. To evaluate the effectiveness of EIDETIC, we compare the error-detecting capability of EIDETIC with the four state-of-the-art methods(i.e., CERTest, DIPROM, AFL++, HiCOND), since finding more defects within a time period is the main objective of these methods. In the experiment, we set a single testing period of two weeks for each method; that is, every method tests GCC and Clang for two weeks. Since the five methods all conduct the mutation on the existing seed programs, we use the same automated C program generators to generate C programs as the seeds for fair comparisons. Besides, the number of generated program variants for each seed in the five methods is identical. Since a unique compiler defect may be triggered by many similar programs and these programs are merged into one root-cause defect after triage, the defect counts are not independent per-test observations.

Results. As shown in Table II, defects detected in the experiment can be classified into new defects (New) and known defects (Known). Defects labeled as Known mean they are duplicates of the defects in the defect repository. Each defect denotes a unique root-cause diagnostic defect rather than a single triggering program; multiple generated programs that expose the same root cause are counted only once after deduplication. It is obvious from Table II that EIDETIC outperforms the other four baselines in terms of the defect-detecting capability. EIDETIC can detect 4 defects in two weeks on the two compilers, of which 2 defects are new and 2 defects are known, while CERTest, DIPROM, AFL++, and HiCOND can only detect 2, 1, 2, and 2 error diagnostic defects, respectively. This outcome fully demonstrates the effectiveness of our method.

Beyond the raw defect counts, we further inspect the defects detected by each method. The defects found by CERTest and AFL++ are mainly triggered by shallow invalid constructs, such as malformed declarations, simple type mismatches, or repeatedly generated front-end errors. DIPROM and HiCOND can generate diverse programs, but they do not explicitly target error-diagnostic defects, so their detected defects are mostly cases where generated variants accidentally violate front-end semantic constraints. In contrast, EIDETIC detects defects involving more structured semantic contexts, including conflicting declarations, invalid type relations, context-dependent control-flow constructs between GCC and Clang. These defects require both a diagnostic-relevant invalid mutation and sufficient program context to expose compiler-specific

```
#include<stdio.h>
void f(int x)
{
    switch (x)
    {
        default: break; // Line 3: Valid
        case 1:  break;
        default: break; // Line 5: The actual duplicate
     /violation
    }
}
```

Listing 3: A reduced mutant with an illegal duplicate default label in a switch statement (ID #180429)



(a) KNN dist **(b)** NNMin dist

Fig. 2: Distribution comparison for KNN and NNMin distances

diagnostic behavior, which explains why the baselines are less likely to expose them within the same budget. Since each reported defect corresponds to a unique root cause rather than an independent test outcome, we do not apply conventional statistical significance tests directly to the defect counts in Table II.

EIDETIC effectively addresses the shortcomings inherent in the four methods above. EIDETIC drives compilation beyond early parsing into deeper inspection stages, where error diagnostic defects are more likely to be exposed. To systematically reach such cases, EIDETIC uses an MAB to select mutators, and leverages a code-embedding–based diversity score as the reward to update the bandit's state and steer subsequent mutator choices effectively.

Conclusion. EIDETIC outperforms CERTest, DIPROM, AFL++, and HiCOND for detecting error diagnostic defects. These results verify the defect-detecting capability of EIDETIC.

E. Answer to RQ2

Approach. We conduct an experiment over three months from December 2025 to February 2026 to evaluate the defect-detecting capability of EIDETIC in practice. In this experiment, we test the latest version of GCC and Clang (i.e., GCC 15.1.0 and Clang 22.0.0), since compiler developers fix defects primarily in the recently released version rather than in old versions. We submitted all the detected defects to the GCC Bugzilla and LLVM's defect reports website.

Results. In three months, we ultimately report 10 distinct compiler error diagnostic defects after deduplication, including 4 Clang defects and 6 GCC defects; 7 of the 10 defects have already been confirmed. Specifically, we merge reports that correspond to the same root cause. Two reports are treated as duplicates if they involve the same compiler, the same acceptance-or-rejection pattern, the same normalized diagnostic category, and the same reduced root cause construct after manual reduction. Therefore, the reported number reflects the count of distinct defects, not the number of executions that triggered failures. Among the 10 defects we reported, 5 were newly discovered defects—that is, defects that had not been reported by anyone before. These defects include acceptance-versus-rejection mismatches, constraint/semantic diagnostic inconsistencies, and other error-related divergences. Since these defects stem from semantically invalid programs, this forces the compilation process into deeper inspection stages, thereby exposing error diagnostic defects in the compiler.

As shown in Listing 3, we discovered a new Clang diagnostic defect involving multiple default labels in a single switch statement. In the C language, the first default label is valid; the program becomes ill-formed only when a later default label duplicates it. However, Clang reports the error at the first default, blaming a correct construct and misleading developers. EIDETIC can detect this previously unnoticed defect because it can generate syntactically well-formed but semantically invalid programs that drive compilers deep into semantic checking, and it applies location-aware differential analysis to flag inconsistencies. This confirmed defect directly supports EIDETIC can uncover new error diagnostic defects.

Conclusion. EIDETIC is effective in detecting compiler defects. Within three months, we reported 10 defects, of which 7 have been confirmed.

F. Answer to RQ3

Approach. First, we generate compilable seed programs through an automated C program generator. Subsequently, we generate two mutant sets of equal size using single-mutation and multi-mutation based on the seed programs. We quantify embedding-space diversity with complementary indicators that capture global spread (i.e., mean pairwise cosine distance) and local redundancy (i.e., mean KNN distance and nearest-neighbor distance, nnmin), and we report coverage@τ, defined as the fraction of mutants whose nearest-neighbor distance satisfies nnmin $\geq \tau$; this directly captures how often a mutant is not a near-duplicate of any existing sample.

Results. As shown in Table IV, multi-mutation improves every diversity indicator: the mean pairwise distance increases from 0.1550 to 0.1633, mean KNN distance from 0.0830 to 0.0884, and mean nnmin from 0.0691 to 0.0777, while Coverage@0.10 nearly doubles, indicating that multi-mutation produces substantially more mutants that exceed a minimum separation from their closest neighbor. As shown in Fig. 2(a) and Fig. 2(b), it is obvious that the multi-mutation setting shifts upward, indicating mutants are farther from their closest neighbors and thus less locally redundant. This supports that multi-mutation explores a broader neighborhood in the embedding space than single-mutation. What's more, Fig. 3 plots

TABLE III
Defects reported by EIDETIC

Number	Compiler	ID	Summary	Status
1	Clang	171283	Dynamic array OOB access fails to trigger a proper error	Confirmed
2	Clang	175414	Make -Werror=<group>override #pragma clang diagnostic ignored	Pending
3	Clang	179454	Nested pointer qualifier mismatch emits warning only	Confirmed
4	Clang	180429	Wrong diagnostic location for duplicate default in switch	Confirmed
5	GCC	123232	printf format type mismatch should be non-suppressible	Confirmed
6	GCC	123248	ASan misses diagnostics at -O1 due to optimization	Pending
7	GCC	123395	C front end accepts invalid pointer-to-integer initialization	Confirmed
8	GCC	123928	Non-standard main signature only warns under -Wall	Confirmed
9	GCC	124017	Misleading locations for repeated undefined label in goto	Confirmed
10	GCC	124030	Missing prior-definition note for duplicate default in switch	Pending

There are two types of status feedback from compiler developers on our reports (i.e., Confirmed = defect has been patched or acknowledged, Pending = report is still under review).

TABLE IV
Embedding-space diversity for single- vs. multi-mutation

Group	Pairwise	KNN		NNMin		Coverage@0.10
	mean dist	mean dist	dist range	mean dist	dist range	
single	0.1550	0.0830	[0.0796, 0.0866]	0.0691	[0.0661, 0.0722]	0.0850
multi	0.1633	0.0884	[0.0847, 0.0923]	0.0777	[0.0746, 0.0810]	0.1800

TABLE V
Effectiveness of mutation scheduling strategies

Strategy	mean_all	mean_first50	mean_last50
TS	0.6311	0.6173	0.6470
Rand	0.6016	0.6201	0.5974
RF	0.6034	0.6066	0.5988

Coverage@τ as a function of τ. The multi-mutation curve stays consistently above the single-mutation curve across the entire range, indicating that the advantage of multi-mutation is not an artifact of a particular τ value but persists under varying minimum-separation requirements.

This gap is expected because a single-mutation often perturbs a localized construct while preserving most structure unchanged, keeping mutants clustered. By composing multiple edits, multi-mutation is more likely to cross structural boundaries, thereby reducing local redundancy and expanding the explored diagnostic neighborhood.

Conclusion. Multi-mutation broadens embedding-space exploration and mitigates redundancy compared with single-mutation.

G. Answer to RQ4

Approach. We compare TS against two baselines-random selection (RAND) and random forest (RF) under the same budget and the same reward definition. For each generated mutant, we compute a reward based on our diversity objective (i.e., embedding-space gain relative to the current pool under the same measurement used in RQ3), and we feed this reward back to the policy immediately after evaluation. We report the mean reward over all mutants (mean_all). We also compare early performance with late performance (mean_first50 and mean_last50) to test whether a policy improves as feedback accumulates.

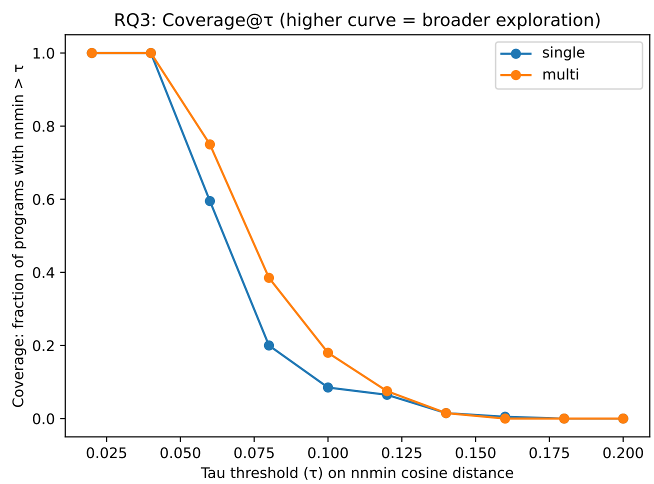


Fig. 3: Coverage@τ under different mutation strategies.

Results. As shown in Table V, TS achieves the highest overall reward (mean_all=0.6311) compared with Rand (0.6016) and RF (0.6034). More importantly, TS exhibits a clear learning trajectory: its average reward increases from 0.6173 (first 50) to 0.6470 (last 50), while both baselines decline over time.

This pattern suggests that TS is effectively exploiting feedback to allocate more trials to higher-yield mutators. However, Rand cannot exploit feedback and thus continues spending the budget on mutators with lower expected reward. Although

RF is also learning-based, it is disadvantaged by sparse and noisy online supervision. With limited labels early in the run and coarse features, RF can struggle to model the interaction between mutator and outcome, and frequent refits may be unstable when rewards depend on relative novelty against the evolving mutant pool. In contrast, TS is better matched to this regime: by explicitly modeling uncertainty and sampling from posteriors, it remains robust under limited feedback, continues exploring under-sampled mutators, and steadily exploits mutators that repeatedly deliver higher reward.

Conclusion. TS is more effective than both Rand and RF, and the improvement of reward demonstrates that the policy learns to prioritize higher-reward mutators.

V. THREATS TO VALIDITY

A. Threats to Internal Validity

The main threat to the internal validity of our method is whether the observed effects are influenced by inconsistencies in program diversity introduced by different automated C program generators. Different program generators may produce seed programs with systematically different syntactic and semantic characteristics, such as control-flow structure, type usage, diagnostic-triggering constructs, and so on. Such differences can change the reachable diagnostic regions and, consequently, affect both the diversity metrics and the downstream defect-detecting outcomes. To mitigate this threat, we (i) fix generator configurations and random seeds whenever possible, (ii) apply the same seed-generation pipeline across compared settings, and (iii) evaluate diversity using multiple complementary embedding-based metrics (i.e., pairwise, KNN distance, nnmin, and Coverage@τ). Nevertheless, we acknowledge that the diversity induced by generator shifts could still partially explain performance differences. Broader validation across multiple generators and configurations would further strengthen causal attribution.

B. Threats to External Validity

A key threat to the external validity of EIDETIC is the lack of open-source implementations for CERTest [11] and DIPROM [8]. Because their code is not publicly available, we implemented CERTest and DIPROM by closely following the algorithms, workflows, and parameter settings described in their papers. However, without reference implementations, we cannot guarantee that our re-implementations match the original systems' engineering details or achieve the same performance as the authors' implementations. Therefore, our empirical comparisons should be interpreted as comparisons against duplication of CERTest and DIPROM, and any performance gaps may partly reflect this inherent reproducibility barrier rather than purely algorithmic differences.

VI. RELATED WORK

A. Test Program Generation

Automated test program generation is a cornerstone of compiler testing [31], [32]. A representative generator is Csmith [29], which produces large numbers of randomly structured C programs while actively avoiding undefined behavior (UB) so that the generated programs have valid semantics. Building on Csmith, CsmithEdge [33] aims to reduce the immunity effect that arises from overly conservative UB-avoidance: it relaxes some of Csmith's generation-time restrictions with controlled probabilities, then uses UB-detection and post-processing to keep UB-free programs for miscompilation testing while still retaining UB-violating programs for crash or hang testing. In parallel, Yarpgen [34] follows a more targeted generation logic and constructs expressive, UB-free programs with patterns that are effective at stressing optimization-heavy compiler paths.

Many approaches obtain tests by mutating existing inputs [35]. AFL++ [15] is a representative coverage-guided fuzzer: it repeatedly mutates seeds, runs the target, and retains inputs that increase coverage or trigger new behaviors as future seeds, so the search gradually concentrates on previously unexplored paths [36]. HiCOND [20] explores diversity from another perspective by searching the configuration space of generators based on historical defect-triggering data, aiming to produce more diverse tests and increase the likelihood of triggering compiler defects.

Recent work also shows growing interest in compiler diagnostics, including warning diagnostics and error diagnostics on invalid code [37], [38]. DIPROM [8] uses diagnostic-oriented mutators and diversity-driven exploration to generate inputs that exercise warning logic more thoroughly than naive mutation. CERTest [11] specifically targets error-recovery defects by exploring mutation neighborhoods around seeds to uncover problems in how compilers recover from errors and how they attribute diagnostics. Our method follows this intuition by applying diagnostic-oriented mutators to generate a large number of variants that trigger diverse error diagnostics.

B. Differential Testing

Differential testing addresses the oracle problem by comparing outputs across multiple comparable executions, flagging inconsistencies as potential defects [1], [39]. In compiler testing, common strategies include cross-compiler testing, cross-optimization testing, and cross-version testing [40]–[42]. These strategies have been repeatedly validated in practice and have uncovered large numbers of long-standing defects [8], [43]. Differential testing also appears in Equivalence Modulo Input (EMI) style workflows, where semantics-preserving variants are generated and then compared to detect deviations [44], [45].

Beyond classical miscompilation or crash detection, recent work increasingly applies differential thinking to front-end behavior, such as warning diagnostics and error diagnostics: instead of comparing only runtime results, systems may compare diagnostic content, locations, or structured diagnostic signatures [46]. This is particularly important for error diagnostic testing, where the objective is not merely to detect failure, but to uncover erroneous errors, spurious errors, and missing errors in compiler diagnostics across compilers, optimization settings, or versions [11], [42].

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we present EIDETIC, an effective framework for detecting error diagnostic defects in C compilers by mutating valid seeds into semantically invalid programs, then performing location-aware and category-aware differential diagnosis across compilers. EIDETIC combines (i) semantic mutators that target deeper constraint violations beyond surface syntax, (ii) an adaptive mutation scheduler that uses code embeddings and a MAB (Thompson Sampling) to reduce redundant exploration, and (iii) a hierarchical differential strategy to normalize and compare diagnostics robustly. In evaluation on GCC and Clang, EIDETIC outperforms four baselines (CERTest, DIPROM, AFL++, HiCOND) and reports 10 distinct error diagnostic defects in recent releases, with 7 confirmed by developers.

In future work, we plan to evaluate EIDETIC on more compiler versions, additional toolchains, and more language modes, so that the conclusions generalize beyond our current GCC and Clang settings. We will also refine our mutators and scheduling strategy to generate a wider range of controlled invalid programs while keeping the search efficient, so that EIDETIC can exercise deeper corner cases without being dominated by redundant mutants.

REFERENCES

[1] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A survey of compiler testing," *Acm Computing Surveys (Csur)*, vol. 53, no. 1, pp. 1–36, 2020.

[2] P. Denny, J. Prather, and B. A. Becker, "Error message readability and novice debugging performance," in *Proceedings of the 2020 ACM conference on innovation and technology in computer science education*, 2020, pp. 480–486.

[3] N. Shrestha, C. Botta, T. Barik, and C. Parnin, "Here we go again: Why is it difficult for developers to learn another programming language?" in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 691–701.

[4] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, "Programmers' build errors: a case study (at google)," in *Proceedings of the 36th international conference on software engineering*, 2014, pp. 724–734.

[5] C. Zhang, B. Chen, L. Chen, X. Peng, and W. Zhao, "A large-scale empirical study of compiler errors in continuous integration," in *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2019, pp. 176–187.

[6] T. Barik, J. Smith, K. Lubick, E. Holmes, J. Feng, E. Murphy-Hill, and C. Parnin, "Do developers read compiler error messages?" in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 575–585.

[7] C. Sun, V. Le, and Z. Su, "Finding and analyzing compiler warning defects," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 203–213.

[8] Y. Tang, H. Jiang, Z. Zhou, X. Li, Z. Ren, and W. Kong, "Detecting compiler warning defects via diversity-guided program mutation," *IEEE Transactions on Software Engineering*, vol. 48, no. 11, pp. 4411–4432, 2022.

[9] LLVM Project, "Clang cfe internals manual," Online documentation, accessed 2026-02-15. [Online]. Available: https://clang.llvm.org/docs/InternalsManual.html

[10] GNU Project, "Porting to gcc 14," GCC release documentation, accessed 2026-02-15. [Online]. Available: https://gcc.gnu.org/gcc-14/porting_to.html

[11] Y. Tang, J. Zhang, X. Li, Z. Huang, and H. Jiang, "Detecting compiler error recovery defects via program mutation exploration," *IEEE Transactions on Software Engineering*, vol. 51, no. 2, pp. 389–412, 2024.

[12] LLVM Project, "Llvm language reference manual," Online documentation, accessed 2026-02-15. [Online]. Available: https://llvm.org/docs/LangRef.html

[13] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," Technical report, 2003, accessed 2026-02-15. [Online]. Available: https://llvm.org/pubs/2003-09-30-LifelongOptimizationTR.pdf

[14] GNU Project, "Tree ssa (gnu compiler collection (gcc) internals)," Online documentation, accessed 2026-02-15. [Online]. Available: https://gcc.gnu.org/onlinedocs/gccint/Tree-SSA.html

[15] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "{AFL++}: Combining incremental steps of fuzzing research," in *14th USENIX workshop on offensive technologies (WOOT 20)*, 2020.

[16] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "{MOPT}: Optimized mutation scheduling for fuzzers," in *28th USENIX security symposium (USENIX security 19)*, 2019, pp. 1949–1966.

[17] P. Jauernig, D. Jakobovic, S. Picek, E. Stapf, and A.-R. Sadeghi, "Darwin: Survival of the fittest fuzzing mutators," *arXiv preprint arXiv:2210.11783*, 2022.

[18] Free Software Foundation, "GCC, the GNU compiler collection," Project website, accessed 2026-02-15. [Online]. Available: https://gcc.gnu.org/

[19] LLVM Project, "Clang: a C language family frontend for LLVM," Project website, accessed 2026-02-15. [Online]. Available: https://clang.llvm.org/

[20] J. Chen, G. Wang, D. Hao, Y. Xiong, H. Zhang, and L. Zhang, "History-guided configuration diversification for compiler test-program generation," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 305–316.

[21] V. J. Traver, "On compiler error messages: what they say and what they mean," *Advances in Human-Computer Interaction*, vol. 2010, no. 1, p. 602570, 2010.

[22] B. A. Becker, P. Denny, R. Pettit, D. Bouchard, D. J. Bouvier, B. Harrington, A. Kamil, A. Karkare, C. McDonald, P.-M. Osera *et al.*, "Compiler error messages considered unhelpful: The landscape of text-based programming error message research," *Proceedings of the working group reports on innovation and technology in computer science education*, pp. 177–210, 2019.

[23] ISO/IEC JTC1/SC22/WG14, "Iso/iec 9899:201x committee draft n1570: Programming languages — c," Working Draft, 2011, accessed 2026-02-15. [Online]. Available: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf

[24] LLVM Project, "Clang compiler user's manual," Online documentation, accessed 2026-02-21. [Online]. Available: https://clang.llvm.org/docs/UsersManual.html

[25] GNU Project, "Options to control diagnostic messages formatting," GCC Online Documentation, accessed 2026-02-21. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Diagnostic-Message-Formatting-Options.html

[26] ——, "Guidelines for diagnostics," GCC Internals Documentation, accessed 2026-02-21. [Online]. Available: https://gcc.gnu.org/onlinedocs/gccint/Guidelines-for-Diagnostics.html

[27] S. Zhang and M. D. Ernst, "Proactive detection of inadequate diagnostic messages for software configuration errors," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 12–23.

[28] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[29] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 283–294.

[30] H. Zhong, "Enriching compiler testing with real program from bug report," in *Proceedings of the 37th IEEE/ACM International conference on automated software engineering*, 2022, pp. 1–12.

[31] M. Marcozzi, Q. Tang, A. F. Donaldson, and C. Cadar, "Compiler fuzzing: How much does it matter?" *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, 2019.

[32] C. Cummins, P. Petoumenos, A. Murray, and H. Leather, "Compiler fuzzing through deep learning," in *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, 2018, pp. 95–105.

[33] K. Even-Mendoza, C. Cadar, and A. F. Donaldson, “Csmithedge: more effective compiler testing by handling undefined behaviour less conservatively,” *Empirical Software Engineering*, vol. 27, no. 6, p. 129, 2022.
[34] V. Livinskii, D. Babokin, and J. Regehr, “Random testing for c and c++ compilers with yarpgen,” *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–25, 2020.
[35] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, “Fault injection techniques and tools,” *Computer*, vol. 30, no. 4, pp. 75–82, 1997.
[36] J. Li, B. Zhao, and C. Zhang, “Fuzzing: a survey,” *Cybersecurity*, vol. 1, no. 1, p. 6, 2018.
[37] M. Kooli, F. Kaddachi, G. Di Natale, A. Bosio, P. Benoit, and L. Torres, “Computing reliability: On the differences between software testing and software fault injection techniques,” *Microprocessors and Microsystems*, vol. 50, pp. 102–112, 2017.
[38] X. Li, X. Liu, L. Chen, R. Prajapati, and D. Wu, “Fuzzboost: Reinforcement compiler fuzzing,” in *International Conference on Information and Communications Security*. Springer, 2022, pp. 359–375.
[39] S. Guo, H. Jiang, Z. Xu, X. Li, Z. Ren, Z. Zhou, and R. Chen, “Detecting simulink compiler bugs via controllable zombie blocks mutation,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1061–1072.
[40] E. Hedlund, “Compiler correctness is super creal: An experimental study on the construction of cross-compiler test oracles for compiler fuzzing,” 2025.
[41] J. Yang, C. Fu, X.-Y. Liu, H. Yin, and P. Zhou, “Codee: A tensor embedding scheme for binary code search,” *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2224–2244, 2021.
[42] S. Liu, Z. Guo, Y. Li, C. Wang, L. Chen, Z. Sun, and Y. Zhou, “An extensive empirical study of inconsistent labels in multi-version-project defect data sets,” *arXiv preprint arXiv:2101.11749*, 2021.
[43] K. Lin, X. Song, Y. Zeng, and S. Guo, “Deepdiffer: Find deep learning compiler bugs via priority-guided differential fuzzing,” in *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*. IEEE, 2023, pp. 616–627.
[44] X. Yu, W. K. Wong, and S. Wang, “Emi testing of large language model (llm) compilers,” in *2024 IEEE 35th International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2024, pp. 187–190.
[45] S. A. Chowdhury, S. L. Shrestha, T. T. Johnson, and C. Csallner, “Slemi: Finding simulink compiler bugs through equivalence modulo input (emi),” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, 2020, pp. 1–4.
[46] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, “An empirical comparison of compiler testing techniques,” in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 180–190.