On the Feasibility of Deduplicating Compiler Bugs with Bisection

XINTONG ZHOU, University of Waterloo, Canada
ZHENYANG XU, University of Waterloo, Canada
YONGQIANG TIAN, Monash University, Australia
CHENGNIAN SUN, University of Waterloo, Canada

Random testing has proven to be an effective technique for compiler validation. However, the debugging of bugs identified through random testing presents a significant challenge due to the frequent occurrence of duplicate test programs that expose identical compiler bugs. The process to identify duplicates is a practical research problem known as bug deduplication. Prior methodologies for compiler bug deduplication primarily rely on program analysis to extract bug-related features for duplicate identification, which can result in substantial computational overhead and limited generalizability.

This paper investigates the feasibility of employing bisection, a standard debugging procedure largely overlooked in prior research on compiler bug deduplication, for this purpose. Our study demonstrates that the utilization of bisection to locate failure-inducing commits provides a valuable criterion for deduplication, albeit one that requires supplementary techniques for more accurate identification. Building on these results, we introduce BugLens, a novel deduplication method that primarily uses bisection, enhanced by the identification of bug-triggering optimizations to minimize false negatives. Empirical evaluations conducted on five real-world datasets demonstrate that BugLens significantly outperforms the state-of-the-art analysis-based methodologies Tamer and D3 by saving an average of 33.56% and 10.68% human effort to identify the same number of distinct bugs. Given the inherent simplicity and generalizability of bisection, it presents a highly practical solution for compiler bug deduplication in real-world applications.

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

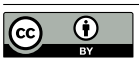
Additional Key Words and Phrases: compiler bug deduplication, bisection, random testing, debugging

ACM Reference Format:
Xintong Zhou, Zhenyang Xu, Yongqiang Tian, and Chengnian Sun. 2026. On the Feasibility of Deduplicating Compiler Bugs with Bisection. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA066 (October 2026), 11 pages. https://doi.org/10.1145/3832157

1 Introduction

Compilers are among the most fundamental and critical software systems, underpinning nearly all modern software. Hence ensuring compiler correctness is crucial to guarantee the reliability of software built upon them. However, the increasing complexity of modern compilers makes them susceptible to various bugs. These bugs can result in incorrect code generation, leading to unexpected program behavior and potential security vulnerabilities. Therefore, detecting and resolving compiler bugs is paramount to maintaining the reliability and correctness of software systems built upon them.

Authors' Contact Information: Xintong Zhou, x27zhou@uwaterloo.ca, University of Waterloo, Waterloo, ON, Canada; Zhenyang Xu, zhenyang.xu@uwaterloo.ca, University of Waterloo, Waterloo, ON, Canada; Yongqiang Tian, Monash University, Melbourne, Australia, yongqiang.tian@monash.edu; Chengnian Sun, cnsun@uwaterloo.ca, University of Waterloo, Waterloo, ON, Canada.



ACM 2994-970X/2026/10-ARTISSTA066
https://doi.org/10.1145/3832157

Random testing has been proven highly effective for improving compiler correctness by discovering numerous bugs [12, 17–20, 29, 38]. For instance, CSmith [38] successfully discovered hundreds of bugs in GCC and LLVM by randomly generating C programs to perform differential testing. By producing numerous test programs, these techniques effectively uncover compiler bugs. However, in practice, a single bug may manifest through various test programs, resulting in redundant programs that trigger the same underlying bug. Given the substantial volume of test programs, classifying these programs based on the specific bugs they expose, referred to as *bug deduplication*, remains challenging yet crucial for prioritizing bug reports and reducing developers' debugging effort. In particular, unlike crash bugs, which can be readily deduplicated using crash information [9, 37], *miscompilation bugs* (also known as wrong-code bugs) lack such distinguishing characteristics, making deduplication significantly more difficult [9, 37].

In the literature, several techniques have been proposed to address the bug deduplication problem in compilers [9, 11, 37]. These techniques are typically *analysis-based*, relying on collecting program features and runtime information to measure similarity among different test cases. For example, Chen et al. [9] proposed Tamer, which distinguishes bug-triggering programs by analyzing compiler code coverage during test execution. Subsequently, Yang et al. introduced D3 [37], integrating more detailed static and runtime information to improve bug deduplication accuracy. While these approaches demonstrate effectiveness in certain contexts, they often involve significant implementation complexity and dependencies on external tools, substantially limiting their practical applicability and generality. In particular, these techniques always depend on test input minimization tools [25, 31, 39] to clear irrelevant code and reduce the analysis complexity, which introduces additional overhead to the deduplication process. Thus, a more *lightweight, generalizable, and independent* approach for bug deduplication is highly desirable in real-world scenarios.

Modern software projects (e.g., GCC and LLVM) extensively employ version control systems like Git [3] to manage development activities. These version histories document all the code changes, thus naturally including the introduction of bugs. In practice, developers frequently perform binary search, i.e., bisection, on the commit history to identify the exact commit that causes the failure, facilitating efficient root-cause analysis and bug resolution. While bisection is widely recognized as a standard debugging technique for its simplicity and effectiveness, leveraging it explicitly for bug deduplication has remained largely overlooked and unexplored in the literature.

To bridge this gap, we conduct the first systematic study on the feasibility of employing bisection for deduplicating compiler bugs, specifically focusing on miscompilation bugs. The fundamental insight guiding our study is that test failures triggered by separate code changes typically indicate different root causes, i.e., distinct bugs. Thus, the commit identified through bisection, i.e., the first commit exhibiting the bug, naturally serves as a criterion to deduplicate various bug-triggering test programs. However, given multiple bugs can be introduced or triggered by a single commit—a common scenario in compiler development—relying solely on this criterion raises concerns about that distinct bugs may be mistakenly identified as identical (referred to as false negatives in § 3.3). Determining whether such false negatives can be effectively minimized without substantial additional effort is essential for establishing the practicality of bisection-based deduplication. Furthermore, the generality and efficiency of this approach are critical factors determining its applicability in real-world scenarios. To systematically evaluate these aspects, we formulate the following research questions (RQs):

- RQ1: How effective is using bisection as the sole criterion to deduplicate compiler bugs?
- RQ2: How can false negatives be mitigated when leveraging bisection for compiler bug deduplication?

- RQ3: Can a bisection-based bug deduplication approach effectively function without test input minimization?
- RQ4: How efficient is the bisection-based bug deduplication in practical settings?

RQ1 and RQ2 focus on effectiveness, RQ3 evaluates generality by analyzing dependence on test input minimization, and RQ4 assesses practical efficiency. By answering these research questions, we aim to present a comprehensive evaluation on the feasibility of leveraging bisection to deduplicate compiler bugs. To assist the practical application, based on our study findings, we propose BugLens, a bisection-based bug deduplication approach. BugLens leverages the failure-inducing commit identified through bisection as the primary criterion, accompanied with examining the bug-triggering optimizations as a subsidiary factor to mitigate false negatives. Our empirical evaluations demonstrate that BugLens significantly outperforms two state-of-the-art analysis-based techniques, Tamer [9] and D3 [37], by saving 33.56% and 10.68% of the human effort on average to identify the same number of unique bugs, respectively. Additionally, results from RQ3 confirm that the approach remains largely effective with unminimized test programs, thus providing a valuable alternative when effective test input minimization is unavailable. RQ4 confirms the practical efficiency of our approach.

Our study demonstrates that the bisection-based approach is a promising alternative to existing analysis-based techniques for compiler bug deduplication, releasing the task from the excessive complexity and limited generality. Our study emphasizes the importance of re-evaluating simple techniques before resorting to more elaborate solutions. Overall, this paper makes the following contributions:

- We conduct the first systematic study on the practical feasibility, including effectiveness, generality, and efficiency, of employing bisection for compiler bug deduplication.
- Based on our study findings, we propose BugLens, a bisection-based bug deduplication approach, which combines the bisection results with the bug-triggering optimizations to provide effective bug deduplication.
- Empirical evaluations conducted on five real-world datasets demonstrate that BugLens significantly outperforms the state-of-the-art analysis-based methodologies in deduplication effectiveness, while exhibiting greater simplicity and generality.

2 Preliminaries

This section provides essential background information required to understand compiler bug deduplication and underscores the limitations of existing approaches.

2.1 Challenges in Random Testing

Random testing enhances compiler correctness by automatically generating large amounts of random test programs to expose compiler bugs [17, 19, 29, 38]. Although random testing has proven effective at identifying new bugs, it faces two major challenges in practice.

First, randomly generated test programs that trigger compiler bugs are often large and complex, typically containing a substantial amount of bug-irrelevant code, which complicates the debugging process. A recent study [30] on bug characteristics in GCC and LLVM reports that minimized test programs average only about 30 lines of code to trigger compiler bugs, whereas the original, non-minimized programs frequently span hundreds or even thousands of lines. Automated test input minimization techniques, such as Delta Debugging [39], have proven effective in alleviating this issue. However, the process remains computationally expensive, often requiring hours to minimize a single program [25, 31].

The second issue is that program generators tend to produce numerous duplicate programs which, despite syntactic differences, trigger the same underlying bug. For instance, in one dataset used in our study, 1,235 C programs generated by CSmith [38], a state-of-the-art C program generator, trigger only 29 unique bugs in the GCC-4.3.0 compiler. Deduplicating these programs and selecting proper ones for investigation are crucial for efficient debugging. The objective is to identify as many distinct bugs as possible while minimizing the number of test programs developers need to analyze. Ideally, with an effective deduplication strategy, only 29 programs would be required to manifest all of these bugs. Effective bug deduplication would significantly reduce the number of cases developers need to investigate, saving considerable time and effort.

2.2 Analysis-Based Compiler Bug Deduplication

In the literature, several techniques have been proposed to tackle the bug deduplication problem in compilers [9, 11, 37]. A common strategy among these techniques is to reformulate bug deduplication as a test case prioritization problem. Specifically, by assessing the similarity among test cases, these approaches prioritize cases likely to reveal distinct bugs. When measuring the similarity between test cases, these approaches are typically *analysis-based*, collecting various program features and runtime information to predict if two test cases trigger the same underlying bug. The state-of-the-art techniques include:

Tamer. Chen et al. [9] are the first to systematically formulate and study the bug deduplication problem in compiler testing. They proposed a technique named Tamer, which measures the distance between pairs of bug-triggering programs using various static program features (e.g., types, operators and loops) and runtime information (e.g., execution output, line and function coverage). Based on the distance, Tamer prioritizes test programs that are more distant from others, as they are more likely to trigger distinct bugs. Their evaluation showed that Tamer achieved optimal performance with using *compiler function coverage* as the distance metric.

Transformer. Donaldson et al. [11] proposed a technique, referred to as Transformer, specifically tailored for transformation-based compiler testing. The fundamental insight of Transformer is that two bug-triggering programs generated by identical transformation operations tend to trigger the same bug. While Transformer incurs minimal additional cost — since transformation operations are inherently part of the testing process — its practical utility is limited due to its narrow focus on transformation-based testing. Another study [37] further demonstrated that Transformer performs poorly when extended to general compiler testing scenarios.

D3. Most recently, Yang et al. [37] introduce D3, which performs a three-dimensional analysis to enhance the effectiveness of bug deduplication. Their approach is grounded on the observation that three distinct dimensions of information are crucial for effectively characterizing compiler bugs: ① the program features of the test program triggering the bug, ② the compiler optimizations necessary to reproduce the bug, and ③ the execution information of the compiler on the program. D3 assesses the similarity between test programs using a three-dimensional distance metric that integrates these dimensions. Although collecting detailed information facilitates more effective deduplication, it also increases the complexity and overhead of the process.

While these techniques are theoretically sound and show acceptable results in certain scenarios, inherent limitations restrict their practical applicability in real-world scenarios.

Complex Implementation and Limited Generality. Analysis-based methods rely heavily on collecting detailed program and execution features to measure the similarity between test cases. Such implementations are typically complex and heavily dependent on external tools. For example, Tamer and D3 both utilize Gcov [1] to collect coverage information, and D3 additionally requires tree-sitter [2] for AST transformations, GumTree [13] for AST difference extraction, and

spectrum-based fault localization (SBFL) tools [27] for identifying buggy code. These dependencies, often language-specific, limit the generality of these methods, requiring substantial effort to adapt them to different domains. Although Transformer is less dependent on detailed program analysis, its scope is narrowly confined to transformation-based testing scenarios. *In this context, a more generalizable bug deduplication approach applicable across diverse domains is highly desirable.*

Strong Reliance on Test Input Minimization. As discussed in § 2.1, compiler fuzzers typically generate large and complex programs filled with bug-irrelevant code, which complicate the debugging process. These extraneous elements also hinder the effectiveness of current analysis-based bug deduplication methods, making them unable to provide prominent improvements over a random selection baseline when handling unminimized programs [9, 37]. In particular, Chen et al. [9] explicitly emphasize the necessity of test input minimization for effective bug deduplication:

> *"Our view is that attempting to tame a fuzzer without the aid of a solid test-case reducer is inadvisable."*
>
> — Chen et al. in Tamer's paper [9]

Moreover, D3 is even infeasible to apply to unminimized test programs. For instance, a critical step in D3 is to mutate a bug-triggering program into a passing one, which is nearly impossible to success on large and complex programs. As a result, both Tamer and D3 require test input minimization as a prerequisite, relying on external tools such as C-Reduce [25] and Perses [31].

However, test input minimization is time-consuming and computationally expensive. For example, it could take hours to minimize a single bug-triggering C program generated by CSmith [38], with state-of-the-art reducers like C-Reduce [25]. To this end, minimizing every test input for the purpose of deduplication becomes undesirable. Ideally, deduplication would occur independently of minimization, with only a small subset of test inputs needed to be minimized for reporting and debugging purposes. Given the high volume of duplicate test inputs, minimization has become a critical bottleneck in current analysis-based deduplication approaches. Additionally, bug slippage [9]—where the original and minimized versions of a test case trigger different bugs—may occur during minimization, further complicating the process. *In this context, a deduplication approach less reliant on test input minimization would be significantly beneficial.*

2.3 Version Control Systems and Bisection

Version Control Systems (VCSs) [28] are essential tools for managing changes to source code over time. They facilitate collaboration, track modifications, and enable reversion to earlier versions when necessary. VCSs play a vital role in modern software development by ensuring that all changes are systematically recorded and maintained. As a result, the documentation of code changes naturally includes the introduction of bugs. Bisect is a powerful feature supported by many VCSs—most notably in Git [14]—that allows developers to efficiently identify the specific commit in which a bug or issue was introduced. By automating a binary search through the commit history, bisection significantly streamlines the debugging process. While bisection is widely adopted and studied for debugging purposes, its potential application in bug deduplication remains largely overlooked and underexplored in the existing literature.

2.4 Motivation

The limitations explained in § 2.2 inevitably restrict the practicality of *analysis-based* deduplication approaches in real-world scenarios. This highlights the practical need for a more generalizable, easy-to-implement, and less-dependent approach. Bisection, given its high simplicity and effectiveness, motivates us to conduct this study to explore its potential for compiler bug deduplication. The

fundamental insight driving this approach is that test failures introduced by different commits should correspond to different root causes, i.e., distinct bugs.

By exploring development history through bisection, our study aims to provide a simple, generalizable, and effective perspective on bug deduplication, reducing the complexity and overhead associated with existing analysis-based techniques.

3 Effectiveness

Effectiveness is of paramount importance in compiler bug deduplication, as it directly impacts the practical utility of the deduplication approach by determining how much human effort can be saved in debugging. To explore this, we pose two effectiveness-oriented research questions:

- **RQ1 (Bisection as the Sole Criterion):** How effective is using bisection as the sole criterion to deduplicate compiler bugs?
- **RQ2 (False Negative Mitigation):** How can false negatives be mitigated when leveraging bisection for compiler bug deduplication?

By answering these questions, we aim to demonstrate the feasibility of deduplicating compiler bugs with bisection and provide practical guidelines for its application in real-world scenarios.

3.1 Datasets

In this study, we use five datasets. Four are drawn from prior work on compiler bug deduplication [9, 37] and compiler testing [7, 8], while the fifth is newly constructed for this study. When constructing these benchmark suites, we strive to reproduce each test failure within our experimental environment, while excluding test programs that meet either of the following criteria: (1) they exhibit undefined behaviors (UBs), as compilers are not designed to produce consistent outputs for such cases and may make arbitrary decisions—such as applying optimizations—around UBs; or (2) the associated bugs could not be reliably reproduced in our experimental environment. The latter exclusion primarily results from differences in system configurations or architectures and affects only a minimal number of cases.

Table 1. The number of test programs and unique bugs in each dataset.

Dataset	Number of Programs	Number of Bugs
GCC-4.3.0	1,235	29
GCC-4.4.0	647	11
GCC-4.5.0	26	7
LLVM-2.8.0	80	5
GCC-13.1.0	42	7

Through considerable effort, we constructed five well-organized datasets. As summarized in Table 1, each dataset comprises a collection of bug-triggering test programs associated with a specific compiler version. The first four datasets, namely GCC-4.3.0, GCC-4.4.0, GCC-4.5.0, and LLVM-2.8.0, cover all publicly available benchmarks that have been used in prior work on compiler bug deduplication, thereby ensuring both comparability and comprehensiveness.

Although comprehensive, these datasets are relatively old, involving compiler versions released more than a decade ago. To further validate the generalizability of our approach in modern compiler development settings, we constructed a new dataset for a recent compiler version, i.e., GCC-13.1.0. We built this dataset as follows. To obtain a set of distinct bug-triggering programs, we first used LegoFuzz [**?**], a recently proposed compiler fuzzing framework, to stress test GCC-13.1.0. Consistent

with the scope of this study, we focused exclusively on silent miscompilation bugs. During a two-week fuzzing campaign, LegoFuzz generated 42 programs that triggered miscompilation bugs in GCC-13.1.0. To determine the ground truth, i.e., the unique bugs triggered by these programs, we adopted the correcting-commit method used in prior work by Tamer [9]. Specifically, for each bug-triggering program, we first identified a good version later than GCC-13.1.0, such as GCC-14.1.0, that executes the program correctly. We then performed bisection between GCC-13.1.0 and the good version to identify the first commit that fixed the bug and caused the program to execute correctly. Programs that shared the same correcting commit were then grouped as triggering the same bug. Note that this method cannot be used to deduplicate new bugs without known patches, thus does not threaten the validity of our study. Using this method, we constructed a dataset of 42 test programs that trigger 7 unique miscompilation bugs in GCC-13.1.0.

Importantly, all bugs and bug-triggering programs in these datasets were originally collected from real-world compiler testing efforts [7, 8, 38**?**], which ensures the practical relevance of our study. The datasets feature two major compilers, GCC and LLVM, both of which have long and complex development histories, allowing the datasets to faithfully reflect the intricacies of real-world compiler development and testing scenarios. The high ratio of test programs to unique bugs in each dataset underlines the practical challenge of compiler bug deduplication in realistic settings. Among the five datasets, the GCC-4.3.0 dataset stands out as the largest and most complex, representing the most challenging case for deduplication.

3.2 Bisection as the Sole Criterion

In the first place, we investigate how effective the results of bisection, i.e., the failure-inducing commits, serve as the sole criterion for compiler bug deduplication. This examination establishes the foundation for our subsequent study.

3.2.1 Methodology. We follow the workflow of existing bug deduplication research [9, 11, 37], which prioritizes test programs based on customized distance metrics. Specifically, our bisection-based deduplication approach involves the following steps:

► *Step 1 (Bisection Range Initialization):* For each test program P in the dataset, we initialize a bisection range with a known-good version ($V_{\checkmark}$) and a known-buggy version ($V_{\times}$). $V_{\times}$ is obviously the compiler version exhibiting the bug, e.g., GCC-4.3.0. To quickly determine $V_{\checkmark}$, we build a series of main releases of the target compiler (e.g., GCC-4.2.0, GCC-4.1.0, etc.). Then we check the behavior of P with each of these versions, and set $V_{\checkmark}$ to the first version before $V_{\times}$ where the bug no longer manifests with P. We maintain a list of pre-built compiler binaries to facilitate quickly narrowing down the bisection range.

► *Step 2 (Failure-inducing Commit Localization):* For each program P in the dataset, we perform bisection, technically a binary search, within the identified range ($V_{\checkmark}$ to $V_{\times}$) on the compiler's commit history to locate the earliest commit c that induces the failure for P.

► *Step 3 (Distance Calculation):* The distance between two test programs reflects their estimated similarity in terms of bug triggering. In our approach, we calculate the distance between two programs based on the time interval between their corresponding failure-inducing commits. Given two test programs P^{α} and P^{β}, with their failure-inducing commits c^{α} and c^{β}, the distance $\mathcal{D}$ between them is defined as:

$$\mathcal{D}(P^{\alpha}, P^{\beta}) = \mathcal{D}_{bisect}(c^{\alpha}, c^{\beta}) = |t(c^{\alpha}) - t(c^{\beta})| \tag{1}$$

where $t(c)$ denotes the timestamp of the commit c in integer seconds. The underlying rationale is that a larger time interval indicates greater likelihood that the test programs correspond to distinct bugs.

▶ *Step 4 (Test Program Prioritization):* Prioritization aims to order test programs such that those triggering distinct bugs appear earlier, thus facilitating faster identification of unique bugs. Using the calculated distances, we prioritize test programs according to the *furthest point first* (FPF) algorithm [15], which always selects the next element that is farthest from all previously chosen ones, thereby ensuring that more diverse programs appear earlier in the ranked list. Specifically, we start with an empty list L, and randomly select the initial test program to append to L. Then, for each remaining program, we compute its distances to all programs already in L, and record the *minimum distance*. The program with the *largest minimum distance* is selected next and appended to L. This process continues until L contains all the test programs. The resulting ranked list then serves as the guidance for the bug reporting and fixing process, allowing developers to efficiently identify and resolve diverse bugs while reviewing fewer programs, significantly reducing their debugging effort.

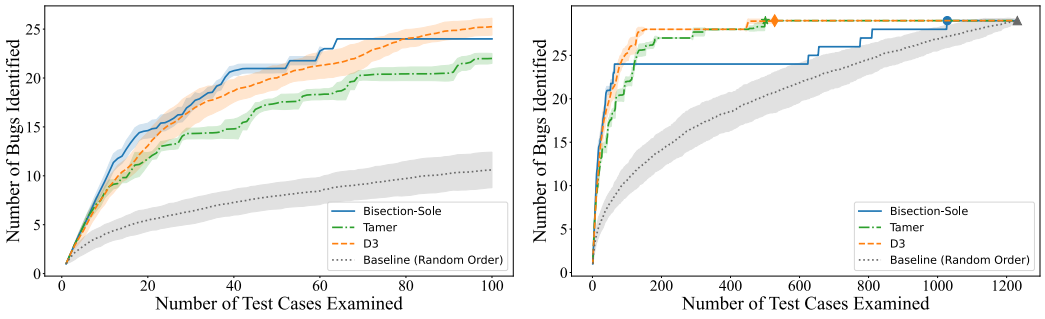
3.2.2 Implementation. We automated the bisection-based bug deduplication approach using `git bisect` [14], a built-in Git functionality, together with a set of helper scripts. Specifically, `git bisect` performs binary search automatically within the compiler repository and, at each step, invokes a predefined test script to determine whether the current compiler version is good or bad for the given test program. This script builds the compiler at the current commit, compiles and executes the test program with the resulting compiler, and returns 0 or 1 to indicate whether the behavior is good or bad. To further accelerate the process, we took several additional measures. First, we maintained a list of prebuilt compiler binaries to quickly narrow the bisection range. Specifically, we built a series of major releases of the target compiler. For example, when deduplicating bugs in GCC-4.3.0, we built several earlier releases, such as GCC-4.2.0 and GCC-4.1.0. For each test case, we then checked the program's behavior on these versions from newest to oldest and selected as the good version the latest one before the bad version in which the bug no longer manifests. This substantially narrowed the bisection range and reduced the cost of bisection. In addition, compiler versions built during bisection were cached and reused in subsequent bisection jobs, further reducing time and resource consumption. As shown in § 5, this caching mechanism significantly reduces the total number of required builds and thereby improves efficiency.

Because some compilers in our datasets are relatively old and may be incompatible with modern system configurations, we adopted a container-based setup to provide a consistent and controlled environment for the bisection process. In particular, we ran the process inside an Ubuntu 12.04 container, whose software environment is better suited to the targeted compiler versions. This container-based setup was required only for our study because of the age of the datasets; it does not limit the applicability of the approach in real-world settings, where the compilers under test are typically much more recent.

3.2.3 Compared Techniques. In this study, we compare the bisection-based deduplication approach with two state-of-the-art analysis-based techniques introduced in § 2.2, namely Tamer [9] and D3 [37]. For Tamer, we adopt the compiler function coverage achieved by the bug-triggering test program, i.e., the number of execution times for each function, as the distance metric, which is shown to be the most effective metric [9] and has been adopted by prior work for evaluation [37]. For D3, we follow the default configurations provided in the original paper. We do not include Transformer [11] in our comparison, as its scope is limited to transformation-based compiler testing. Moreover, prior work [37] has demonstrated that its performance degrades significantly when applied to general compiler testing scenarios.

3.2.4 Metrics. The primary goal of bug deduplication is enabling developers to uncover the maximum number of distinct bugs by examining as few test programs as possible. As this research

question serves as the initial exploration, we adopt the intuitive metric of a *bug discovery curve* for preliminary evaluation of effectiveness. A bug discovery curve visually illustrates how efficiently a ranked list enables the examination of the test cases one by one to encounter at least one representative from each distinct bug category [22, 35]. Thus, a steeper curve is preferable, indicating greater deduplication effectiveness. The bug discovery curve provides a visually intuitive assessment of whether bisection alone effectively functions as a bug deduplication criterion. For a more quantitative evaluation at a large scale, we later adopt the metric of *wasted effort*, which calculates the number of test cases examined before identifying each distinct bug.



(a) First 100 tests (b) All tests

Fig. 1. Bug discovery curves of Bisection-Sole, Tamer, D3 and the baseline on the GCC-4.3.0 dataset.

3.2.5 Results and Analysis. To answer RQ1, we perform the bisection-based deduplication approach described in **??** on the GCC-4.3.0 dataset, selected for its largest size and highest complexity among the five datasets. To alleviate the impact of randomness in the prioritization process, each deduplication approach was repeated one hundred times, and we report the averaged results. The bug discovery curves of the three evaluated techniques, namely Tamer, D3, and Bisection-Sole, are presented in Figure 1, along with a random ordering baseline. The shaded region represents one standard deviation above and below the mean. Figure 1a specifically shows the results for the first 100 test cases examined, reflecting common developer practices where not all bug reports would be thoroughly examined [9, 10, 16, 37]. Figure 1b provides a view of the entire deduplication process. Overall, the results shown in these curves are encouraging, demonstrating the technical feasibility of using bisection as a criterion for deduplicating compiler bugs. Specifically, as depicted in Figure 1a, Bisection-Sole successfully identifies 24 unique bugs within the first 100 test programs examined, which is more than the 22 bugs identified by Tamer and slightly fewer than the 25 bugs identified by D3. Given the inherent simplicity of bisection, these findings offer promising support for the practical potential of using bisection for compiler bug deduplication.

However, the limitation of using such a single-criterion strategy is also apparent. As shown in Figure 1b, after identifying 24 bugs by examining 64 test programs, the discovery curve of Bisection-Sole exhibits several prolonged plateaus, significantly slowing the identification of new bugs. Ultimately, Bisection-Sole requires examining an average of 1027.17 test programs to uncover all 29 bugs in the dataset, which is substantially more than the 489.83 and 453.21 cases required by Tamer and D3, respectively, and only slightly fewer than the random baseline. This low effectiveness is primarily attributed to the misclassifications in the deduplication results. These misclassifications can be categorized into two types:

(1) ***False positives***, where programs triggering the same underlying bug are incorrectly classified as distinct due to different bisection results.

(2) ***False negatives***, where programs triggering distinct bugs are incorrectly grouped together due to the same bisection result.

False positives naturally arise because a single bug may manifest in various ways, leading to different failure-inducing commits (a failure-inducing commit is not necessarily the root cause of the bug, which may simply make the bug triggerable by a program). Given this inherent nature of compiler bugs, false positives are not unique to our bisection-based approach, but prevail in other deduplication methods as well. For instance, for analysis-based approaches, false positives occur when two test programs triggering the same bug are classified as distinct due to different analysis results, i.e., different execution paths or different covered code regions. The direct consequence of false positives is the generation of duplicate bug reports, which require manual effort from developers to filter out.

Conversely, false negatives pose a more serious problem, as they directly result in overlooked bugs unless all test cases are examined—contrary to the fundamental goal of bug deduplication, and often impractical in real-world scenarios [9, 10, 16]. In our approach which uses bisection results as the sole criterion, false negatives typically stem from scenarios such as large single commits introducing multiple different bugs, which are frequent occurrences in compiler development. In other cases, commits modifying configuration flags (e.g., macro definitions) might disclose multiple bugs across different compiler components. Instead of manually investigating each occurrence, developing an effective yet lightweight strategy to mitigate such false negatives is crucial, forming the core focus of RQ2.

RQ1: Employing bisection as the criteria to deduplicate compiler bugs is technically feasible, as demonstrated by its effectiveness in identifying the majority of bugs within the GCC-4.3.0 dataset. However, the overall performance and practical utility could be further improved with extra strategies to mitigate the occurrence of false negatives.

3.3 RQ2: False Negative Mitigation

The presence of false negatives poses a practical limitation to the utility of bisection-based bug deduplication. In this research question, we investigate strategies to mitigate this issue and improve the overall effectiveness of bug deduplication using bisection. To preserve the inherent simplicity and efficiency of bisection, any additional measures introduced to mitigate false negatives must remain lightweight, avoiding complex, analysis-based methods. Within this context, examining various compiler modes and configurations that trigger bugs across different programs provides a promising direction. In particular, given that modern compilers typically support customizable optimization settings—and that complex optimizations are frequently a source of compiler bugs [17, 38]—exploring these optimizations as potential bug-triggering factors aligns well with our goal of developing a simple yet effective solution.

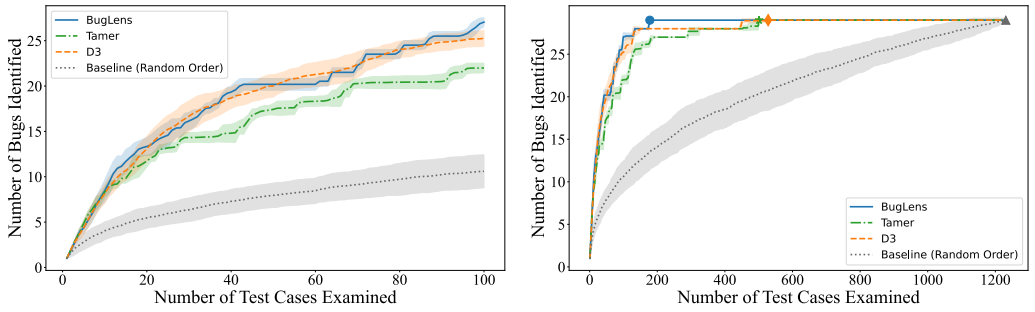
In compiler testing, test programs are typically executed under various optimization levels, and bugs may manifest exclusively when specific optimizations are enabled. Common optimization flags in GCC and LLVM, such as `-O1`, `-O2`, `-O3`, and `-Os`, enable predefined sets of optimizations that can influence bug manifestation. However, not all optimizations enabled by these flags are directly responsible for triggering the bugs. For example, two test programs might both trigger bugs under `-O3`, but one bug could be caused specifically by the optimization `-tree-fre`, while the other might result from `-inline-small-functions`. Thus, pinpointing the exact optimizations responsible for triggering each bug is crucial.

Previous work [37] leverages the idea of Delta Debugging [39] to identify the minimal optimization set that still triggers a given bug. While this approach is effective in principle, it faces a practical limitation: some basic optimizations within predefined optimization levels (e.g., `-O1`, `-O2`) are implicitly enabled and cannot be directly controlled via command line flags. These implicit optimizations are sometimes the precondition for triggering a bug. For example, in our datasets, some test programs trigger bugs under `-O1`, yet the bugs persist even after all explicit optimizations associated with `-O1` are disabled. Conversely, enabling these optimizations individually without using `-O1` fails to reproduce the bug. Identifying such implicit optimizations typically requires in-depth examination of the compiler source code—an effort-intensive process that conflicts with our goal of maintaining simplicity. To resolve this issue, we propose *reversing* the original strategy. Rather than explicitly enabling optimization flags to trigger a bug, we retain the original optimization level (e.g., `-O1`), and selectively add flags to *disable* optimizations outside the minimal triggering set. This reversed approach allows us to isolate critical optimizations without the need for source-level analysis. The resulting optimization configuration is then integrated with the bisection outcomes to form a combined deduplication criterion.

3.3.1 Methodology. The overall deduplication process remains consistent with that described in RQ1, except we introduce an additional step to extract optimization information for false negative mitigation, and modify the distance calculation step accordingly.

► *Extra Step (Optimization Information Extraction):* For each test program P in the dataset, suppose it initially triggers a bug under a certain optimization level, denoted generically as O. In our datasets, O is one of the standard optimization levels defined by the GCC and LLVM compilers, i.e., `O0`, `O1`, `O2`, `O3`, and `Os`. Each optimization level explicitly enables a set of optimization passes $O := \{o_1, o_2, ..., o_n\}$. We then apply the minimizing Delta Debugging (ddmin) algorithm [39] to identify the minimal set of optimization passes that is necessary to trigger the bug with P. Generally, ddmin is a divide-and-conquer approach that iteratively partitions a set into smaller subsets, identifying and removing unnecessary items. Specifically, ddmin first splits the list of passes into $n = 2$ partitions. Then, it traverses each partition and checks whether there is a partition that can solely trigger the bug with P. If so, it removes all other partitions and starts over with the remaining partition. Otherwise, it traverses each partition again to check whether the complement of any partition can trigger the bug with P. If so, it removes the partition and starts over with the remaining partitions. If the list cannot be reduced during the above process, ddmin further splits the list into $2n$ partitions and repeats the above steps. Throughout the process of ddmin, when checking the bug-triggering with a subset of optimizations passes, we always keep the original optimization level O in execution, and selectively disable optimization passes outside the subset by adding disabling flags (formatted as `-fno-{pass_name}` in GCC and LLVM) to the compiler command line. This approach ensures that basic optimizations inherently tied to the optimization level remain active, eliminating the need for manual inspection of the compiler's internal implementation. Eventually, we obtain a minimal set of optimization passes, $O' = \{o_1, o_2, ..., o_m\} \subseteq O$, where $m \leq n$, that are necessary to reproduce the bug for the given test program P.

► *Revised Step 3 (Distance Calculation):* To calculate distances between test programs, we construct vectors that represent their respective bug-triggering configurations. Specifically, each test program is encoded as a vector $\mathbf{v} = [v_0, v_1, ..., v_n]$, where n is the total number of optimization passes supported by the compiler. The first element $v_0 \in \{\texttt{O0}, \texttt{O1}, \texttt{O2}, \texttt{O3}, \texttt{Os}\}$ denotes the optimization level under which the bug manifests. The remaining elements $v_1, v_2, ..., v_n$ are binary values indicating whether the corresponding optimization pass is included in the minimal bug-triggering set O'. Specifically, for $i \in [1, n]$, $v_i = 1$ if the corresponding optimization pass is part of O', and 0 otherwise. Then, the distance between two vectors $\mathbf{v}^\alpha = [v_0^\alpha, v_1^\alpha, ..., v_n^\alpha]$ and $\mathbf{v}^\beta = [v_0^\beta, v_1^\beta, ..., v_n^\beta]$ is computed as



(a) First 100 tests (b) All tests

Fig. 2. Bug discovery curves of BugLens, Tamer, D3 and the baseline on the GCC-4.3.0 dataset.

follows:

$$\mathcal{D}_{opt}(\mathbf{v}^{\alpha}, \mathbf{v}^{\beta}) = \frac{1}{n+1} \sum_{k=0}^{n} \mathbb{I}(v_k^{\alpha} \neq v_k^{\beta}) \tag{2}$$

where $\mathbb{I}(x)$ is the indicator function, returning 1 if its argument is true and 0 otherwise. The distance value is normalized to the range $[0, 1]$ and serves as a complementary criterion to the original distance measure based on the time interval between failure-inducing commits. Thus, the revised distance $\mathcal{D}$ between two test programs P^{α} (with bug-inducing commit c^{α} and optimization vector $\mathbf{v}^{\alpha}$) and P^{β} (with bug-inducing commit c^{β} and optimization vector $\mathbf{v}^{\beta}$) is defined as:

$$\mathcal{D}(P^{\alpha}, P^{\beta}) = \mathcal{D}_{bisect}(c^{\alpha}, c^{\beta}) + \mathcal{D}_{opt}(\mathbf{v}^{\alpha}, \mathbf{v}^{\beta}) \tag{3}$$

where $\mathcal{D}_{bisect}(c^{\alpha}, c^{\beta})$ is the original distance measure defined in **??**, computed as the time interval between the failure-inducing commits c^{α} and c^{β}. This revised distance metric allows us to distinguish between bugs that cannot be differentiated by bisection alone, thereby reducing false negatives.

The remaining steps of the deduplication process follow the same procedure as in RQ1. Test programs are prioritized using the combined distance measure in a furthest-point-first (FPF) order, ensuring that those triggering distinct bugs appear earlier in the list. This ranked list then serves as a strong guide for bug reporting and resolution, potentially saving substantial developer effort. We refer to this refined approach as BugLens.

3.3.2 Results and Analysis. We maintain the same experimental settings from RQ1, and perform bug deduplication with BugLens, Tamer, and D3 on the GCC-4.3.0 dataset. The bug discovery curves are shown in Figure 3. As illustrated in Figure 3a, during the early stage, BugLens continues to perform competitively with D3 and still outperforms Tamer. Specifically, during the first 100 test case examinations, BugLens successfully identifies 27 distinct bugs, while D3 and Tamer identify 25 and 22 bugs, respectively, indicating that BugLens is slightly more effective than the analysis-based approaches in the early stages. The benefit of incorporating optimization information as a complementary criterion becomes evident in the middle-to-late stages, where it successfully distinguishes cases that could not be differentiated solely by bisection results. Specifically, as presented in Figure 3b, with BugLens, only 176.23 test case examinations are required on average to identify all 29 bugs in the dataset, which is a substantial reduction of 64.02% compared to Tamer (489.83) and 61.12% compared to D3 (453.21). Notably, compared to Bisection-Sole, which requires examining

1027.17 test cases on average to identify all bugs, BugLens reduces the number of test case examinations by 82.9%. These results demonstrate that leveraging specific bug-triggering optimizations provides an effective strategy for mitigating false negatives in compiler bug deduplication with bisection.

RQ2: Exploring bug-triggering optimizations offers a promising way to reduce false negatives in bug deduplication while requiring only a modest extra effort. BugLens then significantly outperforms the other evaluated techniques in overall effectiveness, by saving 64.02% and 61.12% of the wasted effort compared to Tamer and D3, respectively. Given the simplicity of techniques involved, BugLens offers a highly practical solution for compiler bug deduplication in real-world development and testing.

3.4 Large Scale Evaluation

The findings from RQ1 and RQ2 give rise to BugLens, a new bisection-based approach to compiler bug deduplication, that follows a fundamentally different technical direction from existing analysis-based techniques. BugLens leverages the failure-inducing commits obtained from bisection as the primary criterion of deduplication, and identifies the specific bug-triggering optimizations as a complementary factor to mitigate false negatives. Given the encouraging finding from RQ1 and RQ2, we stand poised to evaluate BugLens on a larger scale, across all five datasets introduced earlier, to further validate its effectiveness and practical utility. In this comprehensive evaluation, we employ a more fine-grained metric, namely *wasted effort*. Compared to the visually intuitive bug discovery curve, the *wasted effort* metric provides a quantitative measure, indicating precisely how many test cases must be examined before each distinct bug is identified. A lower wasted effort reflects a more effective deduplication process, directly highlighting the reduction in human debugging effort achieved by the deduplication technique.

We compare BugLens with Tamer and D3, and present the detailed results in Table 2. The "ID" column represents each distinct bug identified; however, it is worth noting that an "ID" does not necessarily correspond to the same bug across different techniques, as the order of bug identification may vary. The columns $\Delta_{\%}$(Tamer) and $\Delta_{\%}$(D3) indicate the relative improvement of BugLens over Tamer and D3, respectively. Overall, the results clearly demonstrate that BugLens significantly outperforms both Tamer and D3 by saving the wasted effort in identifying each distinct bug. Specifically, BugLens outperforms Tamer in 48 out of 54 cases by reducing an average of 20.98 (33.56%) test case examinations to identify each bug. Similarly, BugLens achieves better performance than D3 in 40 cases, saving an average of 6.75 (10.68%) test case examinations per bug. While a few cases exist where Tamer and D3 achieve slightly better results, the differences remain marginal. In fact, Tamer and D3 only save an average of 0.26 and 1.53 wasted effort than BugLens in these cases, respectively.

Furthermore, to validate the statistical significance of the results, we conduct pairwise Wilcoxon signed-rank tests [36] on the wasted effort data from the two larger datasets (GCC-4.3.0 and GCC-4.4.0); the other two datasets have too few unique bugs to justify statistical analysis. The p-values are all less than 0.05 (0.0001 and 0.03 for Tamer and D3 on GCC-4.3.0, respectively, and 0.002 for both Tamer and D3 on GCC-4.4.0), indicating that the improvements of BugLens over both Tamer and D3 are statistically significant.

4 Generality

The results presented in § 3 successfully demonstrate the effectiveness of bisection-based bug deduplication. However, effectiveness alone does not ensure the practical utility of a bug deduplication

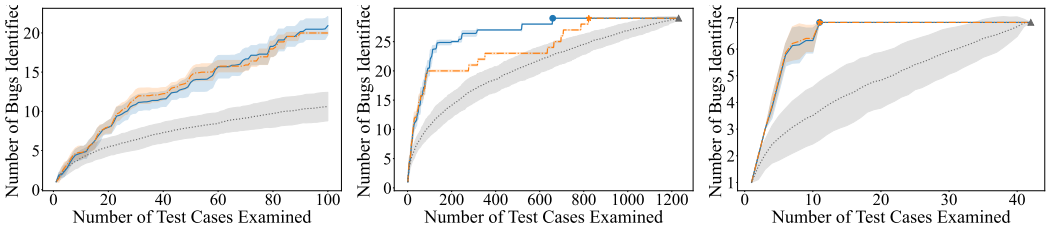
Table 2. Comparison of wasted effort between BugLens and other analysis-based techniques.

ID	BugLens	Tamer	$\Delta_{\%}$(Tamer)	D3	$\Delta_{\%}$(D3)	ID	BugLens	Tamer	$\Delta_{\%}$(Tamer)	D3	$\Delta_{\%}$(D3)
GCC-4.3.0											
2	2.06	2.0	-3.00%	2.0	-3.00%	16	28.05	41.67	32.69%	27.24	-2.97%
3	3.08	3.0	-2.67%	3.16	2.53%	17	30.76	44.93	31.54%	30.11	-2.16%
4	4.08	4.16	1.92%	4.46	8.52%	18	34.69	52.74	34.22%	34.01	-2.00%
5	5.71	5.16	-10.66%	5.75	0.70%	19	38.12	62.13	38.64%	38.99	2.23%
6	6.92	6.24	-10.90%	6.94	0.29%	20	42.57	67.47	36.91%	48.35	11.95%
7	7.96	7.77	-2.45%	8.32	4.33%	21	57.78	83.95	31.17%	54.86	-5.32%
8	9.35	9.34	-0.11%	9.59	2.50%	22	67.09	94.89	29.30%	61.76	-8.63%
9	10.57	10.88	2.85%	11.12	4.95%	23	71.18	112.25	36.59%	68.4	-4.06%
10	11.76	14.52	19.01%	12.94	9.12%	24	77.01	119.29	35.44%	76.5	-0.67%
11	13.24	17.22	23.11%	15.08	12.20%	25	84.15	125.72	33.07%	91.08	7.61%
12	15.18	18.51	17.99%	17.53	13.41%	26	92.8	142.27	34.77%	103.83	10.62%
13	18.0	22.62	20.42%	19.55	7.93%	27	99.26	173.8	42.89%	118.84	16.48%
14	20.16	27.33	26.23%	21.99	8.32%	28	128.41	302.32	57.53%	132.83	3.33%
15	24.21	36.74	34.10%	24.56	1.43%	29	176.23	489.83	64.02%	453.21	61.12%
GCC-4.4.0											
ID	BugLens	Tamer	$\Delta_{\%}$(Tamer)	D3	$\Delta_{\%}$(D3)	ID	BugLens	Tamer	$\Delta_{\%}$(Tamer)	D3	$\Delta_{\%}$(D3)
2	2.00	3.90	48.72%	2.72	26.47%	7	7.00	14.46	51.59%	9.41	25.61%
3	3.00	5.92	49.32%	4.36	31.19%	8	8.00	16.69	52.07%	10.74	25.51%
4	4.00	7.94	49.62%	5.44	26.47%	9	9.00	18.43	51.17%	11.80	23.73%
5	5.00	8.97	44.26%	6.61	24.36%	10	10.00	20.54	51.31%	13.17	24.07%
6	6.00	10.92	45.05%	8.11	26.02%	11	16.00	24.25	34.02%	17.81	10.16%
GCC-4.5.0											
ID	BugLens	Tamer	$\Delta_{\%}$(Tamer)	D3	$\Delta_{\%}$(D3)	ID	BugLens	Tamer	$\Delta_{\%}$(Tamer)	D3	$\Delta_{\%}$(D3)
2	2.00	2.20	9.09%	2.10	4.76%	5	7.14	8.81	18.96%	7.16	0.28%
3	3.12	4.50	30.67%	3.13	0.32%	6	9.66	12.47	22.53%	16.02	39.70%
4	5.24	6.53	19.75%	4.80	-9.17%	7	11.33	19.54	42.02%	21.64	47.64%
LLVM-2.8.0											
ID	BugLens	Tamer	$\Delta_{\%}$(Tamer)	D3	$\Delta_{\%}$(D3)	ID	BugLens	Tamer	$\Delta_{\%}$(Tamer)	D3	$\Delta_{\%}$(D3)
2	2.02	2.13	5.16%	2.05	1.46%	4	7.23	39.29	81.60%	6.09	-18.72%
3	3.54	4.05	12.59%	3.65	3.01%	5	10.14	46.66	78.27%	11.06	8.32%
GCC-13.1.0											
ID	BugLens	Tamer	$\Delta_{\%}$(Tamer)	D3	$\Delta_{\%}$(D3)	ID	BugLens	Tamer	$\Delta_{\%}$(Tamer)	D3	$\Delta_{\%}$(D3)
2	2.00	2.62	23.66%	2.46	18.70%	5	5.00	7.74	35.40%	6.78	26.25%
3	3.00	4.30	30.23%	3.54	15.25%	6	7.08	10.54	32.83%	8.21	13.76%
4	4.00	5.82	31.27%	4.95	19.19%	7	9.26	19.86	53.37%	9.98	7.21%

approach; other properties, particularly generality, also play a critical role. As discussed in § 2.2, analysis-based techniques often face criticism regarding their generality due to two primary issues: (1) domain-specificity, requiring additional human effort to adapt them to different domains, and (2) ineffective or even infeasible on unminimized test programs, relying on test input minimization as a pre-processing step. Bisection-based bug deduplication inherently addresses the first issue because it can be directly adapted to different domains without extra effort, provided the target compiler is maintained in a version control system and supports multiple configuration modes. However, the second issue remains concerning. While bisection theoretically should not be affected by the minimization status of test programs, practical scenarios may differ. In particular, irrelevant fragments

in unminimized programs might lead to variations in compiler behavior across different versions, resulting in unexpected situations during the bisection process. For instance, we encountered a scenario where a program triggering a miscompilation bug caused intermediate compiler versions to crash during bisection. Such crashes prevent us from determining whether the miscompilation bug is present in the current version, as they may result from an earlier-stage failure triggered by code irrelevant to the bug of interest. These unexpected situations could increase false positives, wherein identical bugs are incorrectly identified as distinct due to inconsistent bisection outcomes, ultimately reducing deduplication effectiveness. To evaluate the generality of bisection-based bug deduplication, we investigate the following research question:

- **RQ3:** Can a bisection-based bug deduplication approach effectively function without test input minimization?



(a) First 100 tests on GCC-4.3.0 (b) All tests on GCC-4.3.0 (c) All tests on GCC-13.1.0

Fig. 3. Bug discovery curves of Bisection-Sole, BugLens, and the baseline on the unminimized GCC-4.3.0 and GCC-13.1.0 datasets.

To answer this RQ, we perform bisection-based bug deduplication (both Bisection-Sole and BugLens) on the original unminimized test programs in the GCC-4.3.0 and GCC-13.1.0 datasets, which are the only datasets for which the original unminimized test programs are available.

As discussed in § 2.2, D3 is practically unrealistic to apply to unminimized programs. Additionally, the authors of Tamer also acknowledged that they failed to conduct experiments on unminimized test cases due to the excessively large data volume involved [9]. Therefore, we limit our comparison to a random selection strategy for unminimized test programs. The resulting bug discovery curves are presented in Figure 4, clearly illustrating that bisection-based bug deduplication remains effective even with unminimized test programs. For example, on dataset GCC-4.3.0, BugLens successfully identifies 21 bugs during the first 100 test programs, while random selection can only discover 10 bugs. Furthermore, BugLens requires 658.38 test case examinations to identify all the 29 bugs, saving 41.33% of the effort compared to random selection. Notably, exploiting the bug-triggering optimizations can still effectively improve the overall performance by reducing false negatives. However, as anticipated, irrelevant fragments in unminimized programs negatively impact deduplication performance, primarily due to increased false positives. In response to this challenge, we provide practical suggestions in § 6 to help balance deduplication effectiveness with the overhead of minimization. On the more recent GCC-13.1.0 dataset, the bisection-based approach remains highly effective, identifying all 7 bugs with an average of only 9.75 test case examinations, and reducing effort by 71.17% compared with random selection. Ultimately, bisection-based bug deduplication offers a valuable alternative to analysis-based techniques, particularly in scenarios where effective test input minimization is unavailable.

RQ3: Bisection-based bug deduplication remains effective even without test input minimization, demonstrating superior generality. Nevertheless, irrelevant code fragments in original test programs can complicate the bisection process, thereby reducing deduplication performance. The trade-off between deduplication effectiveness and minimization overhead should be carefully balanced in practice.

5 Efficiency

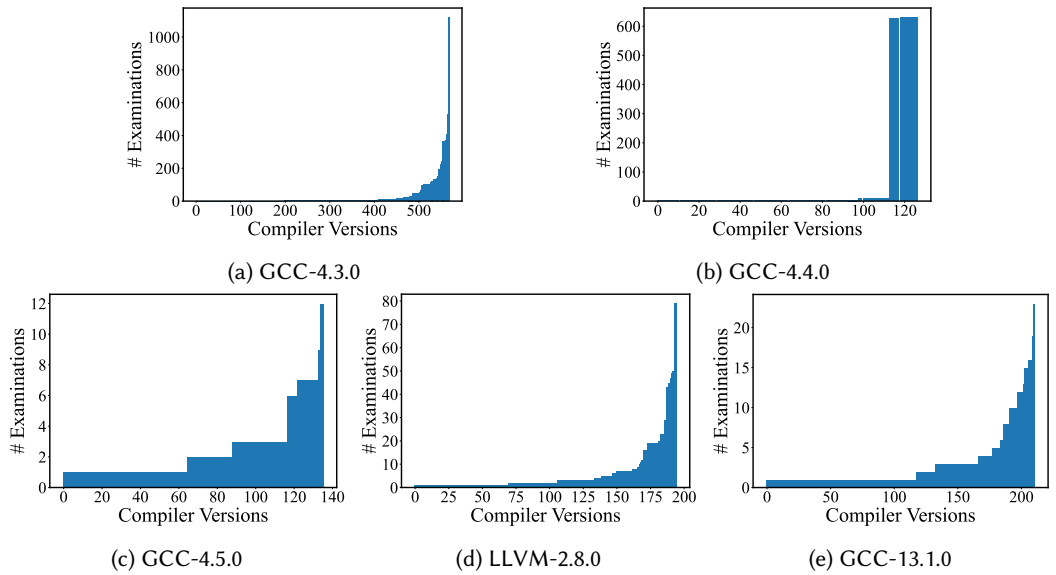


Fig. 4. Number of times each compiler version being examined by different test programs during bisection in each datasets.

Another important consideration when using bisection to deduplicate bugs is its efficiency. Bisection requires building a series of intermediate compiler versions from the commit history, which constitutes the most time-consuming component of the entire deduplication process. For a dataset containing n test programs, and an average bisection range of m commits, the theoretical maximum number of compiler versions built would be $n \cdot \log_2 m$. Given the typically large volume of test cases, efficiency becomes a crucial factor influencing the practical utility of bug deduplication through bisection. Our last research question aims to investigate this issue:

- **RQ4:** How efficient is the bisection-based bug deduplication in practical settings?

Building a compiler from source code can take varying amounts of time on different machines, depending on the hardware configuration and the number of available cores. To formalize the efficiency evaluation, we treat the time required for a single compiler build as a constant, and measure efficiency by counting how many compiler versions must be built during the bisection process. Notably, although the same compiler version may be examined multiple times by different programs, it only needs to be built once and can then be cached for reuse—a strategy adopted in our implementation. Given the large number of test programs in our datasets, this caching mechanism significantly reduces the total number of builds required, thereby enhancing efficiency.

Table 3. Number of compiler versions built during the whole bisection-based bug deduplication process.

	GCC-4.3.0	GCC-4.4.0	GCC-4.5.0	LLVM-2.8.0	GCC-13.1.0
# Versions	569	127	136	195	211
# Versions per Case	0.46	0.19	5.23	2.44	5.02

The number of compiler versions built during the entire deduplication process for each dataset is presented in Table 3. Among the five datasets, the average number of compiler versions built per case ranges from 0.19 to 5.23, typically completing within a relatively short timeframe. Therefore, although our approach theoretically has high worst-case complexity, it remains practically efficient. Figure 5 illustrates how frequently each compiler version is examined by various test programs during bisection, where the x-axis represents the number of compiler versions being examined, and the y-axis represents the number of times a version is examined by different test programs. The results show that most compiler versions are examined multiple times by different test programs. The data confirms that a single compiler build can effectively serve multiple test programs, aligning with our expectations. In practice, the higher the proportion of duplicated bugs, the fewer compiler versions need to be built, as programs triggering the same bug generally produce similar bisection traces. Moreover, the efficiency can be further improved by maintaining a list of pre-built compiler binaries for quickly narrowing down the bisection range. Leveraging incremental build [24] can also bring efficiency benefits by accelerating each build process.

RQ4: The average number of compiler versions needed to be built during the deduplication process with bisection is relatively small, ranging from 0.19 to 5.23 per test program. This indicates that bisection-based bug deduplication maintains high practical efficiency. The most time-consuming aspect—building multiple compiler versions—is well mitigated by reusing the same version across multiple test programs.

6 Discussion

Practical Suggestions. Our study confirms the feasibility of using bisection to deduplicate compiler bugs in terms of effectiveness, generality, and efficiency. Prior research has typically treated bug deduplication and test input minimization as separate tasks, relying heavily on test input minimizers to enable effective deduplication, thus inevitably suffering from the overhead of minimization. In contrast, our approach provides an alternative solution that reduces this dependency. Nevertheless, noise present in unminimized test programs can still interfere with the bisection process, potentially diminishing deduplication effectiveness. To address this, we propose the following suggestions for practitioners.

Rather than minimizing the entire set of bug-triggering test programs upfront, we recommend first applying bisection-based deduplication directly on the original test programs. Once preliminary deduplication results are obtained, practitioners can select a smaller subset of test programs for inclusion in bug reports—only this subset requires minimization. The number of selected programs can be adjusted according to the development team's available resources. After the reported bugs are fixed, remaining duplicate test programs can be efficiently filtered by executing them on the patched compiler version. This workflow can be iteratively applied until all bugs are addressed. Crucially, the deduplication algorithm remains central to this process. Without it, randomly selecting test programs for reporting may result in redundant submissions of the same bug—wasting valuable developer time on diagnosing duplicate reports. An effective deduplication system ensures that the

selected bugs for reporting are as diverse as possible, thereby maximizing developer productivity and minimizing duplicated effort.

Originality & Contributions. This study successfully demonstrates the feasibility and effectiveness of using bisection for compiler bug deduplication. Our findings show that leveraging bisection to identify failure-inducing commits provides a useful signal for bug deduplication, albeit one that requires supplementary techniques for more accurate identification. Admittedly, the underlying technique itself is not new. For example, Theodoridis et al. [**? ?**] used a bisection-based approach to identify bug-inducing commits that lead to compiler performance degradations, and Yang et al. [**?**] revisited automated compiler fault isolation and showed that bisection can effectively serve as a lightweight alternative to more sophisticated techniques. Nevertheless, the originality of our work lies in being the *first systematic study to investigate bisection as a signal for compiler bug deduplication and to directly compare it empirically with prior deduplication techniques*. This constitutes a novel application of the technique. While prior work has proposed sophisticated and often complex algorithms for this task, our study shows that this long-standing practical challenge can be addressed effectively with a lightweight and straightforward approach. In this sense, our study *highlights the value of re-evaluating simple techniques before resorting to more elaborate solutions* . We believe this *simplicity is a strength*, and our findings suggest that such an approach can serve as a strong and practical baseline for future work on compiler bug deduplication.

Generality. A primary threat to the generality of bisection-based bug deduplication is that it requires bugs to be regressions, i.e., there must exist both a *good* version and a *bad* version. As a result, our approach is not applicable to non-regression bugs, such as bugs that persist throughout the entire history of the compiler. Nevertheless, we argue that the proposed approach remains highly applicable in the context of modern compilers, where most correctness bugs are regressions. Taking GCC as an example, developers often indicate regression information in bug report titles [**?**]. Based on this convention, we found that most miscompilation bugs in GCC are regressions, as reflected by titles beginning with "*xx Regression*". Furthermore, because our approach mainly targets the deduplication of compiler testing results, we manually examined the bugs reported in several major compiler testing papers [17, 29**? ?** , 30]. These papers typically report the "ages" of the bugs they uncover. Although some techniques can discover long-lived bugs, i.e., bugs that also manifest in older compiler versions, none report bugs that persist throughout the compiler's entire history, that is, non-regression bugs. Based on our experience with modern compiler testing, finding non-regression bugs in modern compilers is theoretically possible but practically unlikely, given their long and stable development history. Therefore, although our approach is conceptually limited to regression bugs, it remains highly applicable in practice.

Limitations and Future Work. One of the limitations of our proposed approach is that it requires the developing history of the target software available. This issue is practically alleviated by the prevalence of version control systems like Git. Even for close source compilers, e.g., Intel C++ Compiler [4], they normally maintain a private git-like repository for development, which can be used for bisection-based bug deduplication. To this end, the bisection-based bug deduplication approach is practically applicable to a wide range of domains. Another issue is the existence of false positives in the deduplication results, where test programs associated with the same bug are incorrectly identified as distinct. False positives are not unique to our approach, but rather prevalent in other bug deduplication approaches. Future work can focus on further improving the deduplication effectiveness by mitigating the happening of false positives.

7 Threats to Validity

Internal Threats. The primary internal threats stem from the correctness of the implementation of the bisection-based approach and its evaluation process. Our implementation leverages the well-established `git bisect` tool, which is widely adopted in practice, thereby significantly mitigating risks associated with the correctness of the bisection process. Furthermore, we incorporate an additional verification step after executing `git bisect`, explicitly ensuring that the identified commit indeed triggers the bug and that the preceding commit remains unaffected. To further ensure correctness, all components of the implementation have undergone rigorous reviews by the authors, who carefully verified both the code and the associated experimental results.

External Threats. The main external threat concerns the generalizability of the results. While bisection-based bug deduplication is conceptually applicable across various compilers, our evaluation focuses exclusively on the GCC and LLVM compilers. All existing work suffers from the same issue due to the lack of diverse datasets. However, considering the complexity and active development of GCC and LLVM, we argue that our findings possess a reasonable degree of generalizability. Specifically, the effectiveness demonstrated in these challenging scenarios suggests that our approach may perform comparably well in other contexts. Future evaluations extending to additional domains would be valuable to more conclusively affirm the general applicability of our approach.

Construct Threats. The primary construct threat in this study stems from the datasets used. First, the correctness of the ground truth—i.e., the actual bug triggered by each test program—is critical for evaluating deduplication effectiveness. All test programs in our datasets include ground truth labels provided by the original authors. To enhance the reliability of these labels, we have manually verified their correctness to the best of our ability. Another potential threat arises from the age of the datasets. The existing four datasets can be traced back to decades ago, which may raise concerns about their relevance to the current state of compiler infrastructures. To mitigate this threat, we have constructed a new dataset on GCC-13.1.0, a more recent compiler version, and included it in our evaluation. The results from this dataset are consistent with those from the older datasets, suggesting that the findings may still be applicable to modern compilers.

8 Related Work

We discuss two lines of related work.

Bug Deduplication. Our study focuses on deduplicating compiler bugs, making the most relevant related work Tamer [9], D3 [37] and Transformer [11], which are introduced in § 2.2. Additionally, bug deduplication techniques for general-purpose software have been studied in the literature. These approaches typically rely on analyzing the textual descriptions of bugs—such as bug reports and error messages—to identify duplicates. For example, BM25F [23] is a widely used similarity metric for detecting duplicate bug reports. Building on this, Sun et al. [32] proposed REP, which incorporates both textual content and non-textual metadata (e.g., product, component, version) to improve similarity measurement. They further extended this line of work by introducing discriminative models for more accurate duplicate detection [33]. Nguyen et al. [21] proposed DMTM, which combines topic-based and textual features to characterize bug reports for deduplication. Zou et al. [**?**] introduced a multi-factor analysis approach that integrates topic modeling, enhanced N-gram similarity, and contextual metadata for improved accuracy. Lin et al. [**?**] proposed manifold correlation features and an enhanced SVM model that significantly boosts duplicate detection performance. Wang et al. [**?**] presented a hybrid technique that leverages both natural language descriptions and execution traces, while Lerch and Mezini [**?**] developed a proactive approach that predicts potential duplicates using call stack structures and machine

learning. Although these techniques are effective in their respective domains, they are not directly applicable to compiler bug deduplication. Compiler bug reports typically contain minimal textual information—often limited to the bug-triggering test program and the compiler configuration used—rendering text-based techniques ineffective. In this context, our study introduces a novel perspective on compiler bug deduplication by proposing a lightweight, analysis-free approach that does not rely on rich textual content or extensive program analysis.

Failure-inducing Commit Localization. Localizing the exact commit that causes a test failure is a common practice in software debugging and fixing, and has been extensively studied in the literature. One of the most well-known techniques is the SZZ algorithm [26], along with its numerous variants [**? ? ? ?**]. In general, given a bug-fixing commit, the SZZ algorithm traces the version history to identify commits that last modified the lines removed or changed in the fix—these are considered the bug-inducing commits. However, SZZ and its variants are not applicable in our context, as they rely on the existence of a bug-fixing commit, which is not available at the time of bug detection. Other approaches have also been proposed: Wen et al. [**?**] leverage information retrieval techniques to match bug reports, while An et al. [5, 6] incorporate code coverage analysis to aid in localizing bug-inducing commits. More recently, Tang et al. [34] introduced a semantic analysis approach that uses program slicing and semantic reasoning to detect suspicious commits based on data flow and control flow differences. While effective, these techniques either depend heavily on the quality of bug reports or involve complex program analysis, making them unsuitable for our setting. To this end, we adopt bisection to localize failure-inducing commits in this study, aligning with our goal of developing a lightweight and analysis-free bug deduplication approach.

9 Conclusion

Debugging compiler bugs identified through random testing is a painful process, suffering from the bug deduplication problem, i.e., numerous duplicate test cases expose the same underlying bug. Existing deduplication algorithms are primarily analysis-based, leading to high complexity and limited generalizability in practice. In this paper, we conduct the first systematic study on leveraging bisection for deduplicating compiler bugs. Motivated by the insights gained from this study, we propose BugLens, a simple, generalizable, and effective approach for compiler bug deduplication. Our method outperforms the state-of-the-art analysis-based techniques, by significantly reducing the human effort required for debugging. We emphasize the practical value and applicability of bisection-based bug deduplication in real-world scenarios.

Data Availability

We publicly release the artifact to support the reproducibility of our study [**?**].

Acknowledgments

We thank all the anonymous reviewers in ISSTA'26 for their insightful feedback and comments. This research is supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) through the Discovery Grant, and by CFI-JELF Project #40736.

References

[1] 2020. *Gcov*. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html
[2] 2022. *Tree-sitter*. https://github.com/tree-sitter/tree-sitter
[3] 2024. *Git*. https://git-scm.com/
[4] 2024. *Intel C++ Compiler*. https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler.html
[5] Gabin An, Jingun Hong, Naryeong Kim, and Shin Yoo. 2023. Fonte: Finding bug inducing commits from failures. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 589–601.
[6] Gabin An and Shin Yoo. 2021. Reducing the search space of bug inducing commits using failure coverage. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1459–1462.
[7] Junjie Chen and Chenyao Suo. 2022. Boosting compiler testing via compiler optimization exploration. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 4 (2022), 1–33.
[8] Junjie Chen, Guancheng Wang, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Lu Zhang. 2019. History-guided configuration diversification for compiler test-program generation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 305–316.
[9] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming compiler fuzzers. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 197–208.
[10] Zhichao Chen, Junjie Chen, Weijing Wang, Jianyi Zhou, Meng Wang, Xiang Chen, Shan Zhou, and Jianmin Wang. 2023. Exploring better black-box test case prioritization via log analysis. *ACM Transactions on Software Engineering and Methodology* 32, 3 (2023), 1–32.
[11] Alastair F Donaldson, Paul Thomson, Vasyl Teliman, Stefano Milizia, André Perez Maselco, and Antoni Karpiński. 2021. Test-case reduction and deduplication almost for free with transformation-based compiler testing. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1017–1032.
[12] 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*. 1219–1231.
[13] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 313–324.
[14] Git. 2024. *Git Bisect*. https://git-scm.com/docs/git-bisect
[15] Teofilo F Gonzalez. 1985. Clustering to minimize the maximum intercluster distance. *Theoretical computer science* 38 (1985), 293–306.
[16] Josie Holmes and Alex Groce. 2018. Causal distance-metric-based assistance for debugging after compiler fuzzing. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 166–177.
[17] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices* 49, 6 (2014), 216–226.
[18] Shaohua Li, Theodoros Theodoridis, and Zhendong Su. 2024. Boosting Compiler Testing by Injecting Real-World Code. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 223–245.
[19] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2020. Random testing for C and C++ compilers with YARPGen. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–25.
[20] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2023. Fuzzing loop optimizations in compilers for C++ and data-parallel languages. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1826–1847.
[21] Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N Nguyen, David Lo, and Chengnian Sun. 2012. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 70–79.
[22] Dan Pelleg and Andrew Moore. 2004. Active learning for anomaly and rare-category detection. *Advances in neural information processing systems* 17 (2004).
[23] José R Pérez-Agüera, Javier Arroyo, Jane Greenberg, Joaquin Perez Iglesias, and Victor Fresno. 2010. Using BM25F for semantic search. In *Proceedings of the 3rd international semantic search workshop*. 1–8.
[24] Georges Aaron Randrianaina, Xhevahire Tërnava, Djamel Eddine Khelladi, and Mathieu Acher. 2022. On the benefits and limits of incremental build of software configurations: an exploratory study. In *Proceedings of the 44th International Conference on Software Engineering*. 1584–1596.
[25] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. 335–346.

[26] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes? *ACM sigsoft software engineering notes* 30, 4 (2005), 1–5.
[27] Jeongju Sohn and Shin Yoo. 2017. Fluccs: Using code and change metrics to improve fault localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 273–283.
[28] Diomidis Spinellis. 2005. Version control systems. *IEEE software* 22, 5 (2005), 108–109.
[29] 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*. 849–863.
[30] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward understanding compiler bugs in GCC and LLVM. In *Proceedings of the 25th international symposium on software testing and analysis*. 294–305.
[31] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: Syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering*. 361–371.
[32] Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang. 2011. Towards more accurate retrieval of duplicate bug reports. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. IEEE, 253–262.
[33] Chengnian Sun, David Lo, Xiaoyin Wang, Jing Jiang, and Siau-Cheng Khoo. 2010. A discriminative model approach for accurate duplicate bug report retrieval. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. 45–54.
[34] Lingxiao Tang, Chao Ni, Qiao Huang, and Lingfeng Bao. 2024. Enhancing Bug-Inducing Commit Identification: A Fine-Grained Semantic Analysis Approach. *IEEE Transactions on Software Engineering* (2024).
[35] Pavan Vatturi and Weng-Keen Wong. 2009. Category detection using hierarchical mean shift. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. 847–856.
[36] Robert F Woolson. 2005. Wilcoxon signed-rank test. *Encyclopedia of biostatistics* 8 (2005).
[37] Chen Yang, Junjie Chen, Xingyu Fan, Jiajun Jiang, and Jun Sun. 2023. Silent compiler bug de-duplication via three-dimensional analysis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 677–689.
[38] 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*. 283–294.
[39] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on software engineering* 28, 2 (2002), 183–200.