Singapore Management University

Institutional Knowledge at Singapore Management University

Research Collection School Of Computing and Information Systems

School of Computing and Information Systems

7-2023

Silent compiler bug de-duplication via three-dimensional analysis

Chen YANG
Tianjin University

Junjie CHEN
Tianjin University

Xingyu FAN
Tianjin University

Jiajun JIANG
Tianjin University

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Follow this and additional works at: https://ink.library.smu.edu.sg/sis_research

Part of the Software Engineering Commons

Citation

YANG, Chen; CHEN, Junjie; FAN, Xingyu; JIANG, Jiajun; and SUN, Jun. Silent compiler bug de-duplication via three-dimensional analysis. (2023). *ISSTA 2023: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, Seattle, July 17-21*. 677-689.

Available at: https://ink.library.smu.edu.sg/sis_research/8149

This Conference Proceeding Article is brought to you for free and open access by the School of Computing and Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Computing and Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email cherylds@smu.edu.sg.

Silent Compiler Bug De-duplication via Three-Dimensional Analysis

Chen Yang
yangchenyc@tju.edu.cn
College of Intelligence and Computing, Tianjin University
Tianjin, China

Junjie Chen*
junjiechen@tju.edu.cn
College of Intelligence and Computing, Tianjin University
Tianjin, China

Xingyu Fan
fxyqaq@tju.edu.cn
College of Intelligence and Computing, Tianjin University
Tianjin, China

Jiajun Jiang
jiangjiajun@tju.edu.cn
College of Intelligence and Computing, Tianjin University
Tianjin, China

Jun Sun
junsun@smu.edu.sg
Singapore Management University
Singapore

ABSTRACT

Compiler testing is an important task for assuring the quality of compilers, but investigating test failures is very time-consuming. This is because many test failures are caused by the same compiler bug (known as bug duplication problem). In particular, this problem becomes much more challenging on silent compiler bugs (also called wrong code bugs), since these bugs can provide little information (unlike crash bugs that can produce error messages) for bug de-duplication. In this work, we propose a novel technique (called D^3) to solve the duplication problem on silent compiler bugs. Its key insight is to characterize the silent bugs from the testing process and identify three-dimensional information (i.e., *test program*, *optimizations*, and *test execution*) for bug de-duplication. However, there are huge amount of bug-irrelevant details on the three dimensions, D^3 then systematically conducts causal analysis to identify bug-causal features from each of the three dimensions for more accurate bug de-duplication. Finally, D^3 ranks the test failures that are more likely to be caused by different silent bugs higher by measuring the distance among test failures based on the three-dimensional bug-causal features. Our experimental results on four datasets (including duplicate bugs of both GCC and LLVM) demonstrate the significant superiority of D^3 over the two state-of-the-art compiler bug de-duplication techniques, achieving the average improvement of 19.36% and 51.43% in identifying unique silent compiler bugs when analyzing the same number of test failures.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

Compiler Bugs, Bug Deduplication, Fuzzing

ACM Reference Format:
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 (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598087

1 INTRODUCTION

Compilers are one of the most fundamental software systems since almost all the software are built on them. Due to the important role of compilers, compiler bugs could potentially render all programs buggy, leading to unexpected behaviors, even disasters in safety-critical domains [10, 16, 22]. Therefore, evaluating the quality of compilers is critical. In practice, compiler testing is the most widely-used method for compiler quality assurance [8, 14, 47]. In the literature, many compiler testing techniques have been proposed [11, 14, 17, 32, 54, 58], which in general run a large number of test programs to detect as many bugs as possible [14, 16].

Although some compiler testing techniques have been demonstrated to be effective, diagnosing test failures is still a tedious and time-consuming task. One major reason is that many test failures are caused by the same compiler bug, which is known as the *bug duplication problem* [20, 23]. Since debugging compiler bugs are time-consuming [13], investigating duplicate bugs can cause the huge waste of time and resource. Compiler bugs that produce error messages (called *crash bugs*) can be effectively de-duplicated by comparing the error messages produced by different test failures as demonstrated by the existing work [20, 28]. However, there are also a significant number of bugs that do not lead to crashes (called *silent bugs* or *wrong code bugs*), and as a result, there is little information facilitating the task of bug de-duplication. Regarding these silent bugs, the bug duplication problem is thus much more challenging.

In the literature, there are de-duplication techniques proposed for ordinary software, which are mostly based on textual description, test inputs, or error messages [7, 40, 48]. However, compilers have different forms of test inputs with ordinary software (i.e., the inputs of compilers are programs) and meanwhile silent compiler bugs do not have available textual description or error messages. Hence,

[*] Junjie Chen is the corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0221-1/23/07…$15.00
https://doi.org/10.1145/3597926.3598087

these techniques are not applicable to the duplication problem on silent compiler bugs. In recent years, two de-duplication techniques specific to compiler bugs have been proposed [20, 23], but they are ineffective with respect to silent bugs. Specifically, Chen et al. [20] proposed the first technique, which measures the distance among test failures by investigating various combinations of program features for bug de-duplication. The most effective one for silent bugs works by comparing the function coverage achieved by each bug-triggering test program and reports those bugs that have similar function coverage as duplicates. However, the effectiveness is still unsatisfactory (also confirmed by our study in Section 4.5) due to the large gap between function coverage and root causes. Fundamentally, this is because many covered functions are not causal to the bug. Subsequently, Donaldson et al. [23] proposed a technique that is specific to the test failures produced by transformation-based compiler testing. Hence, it is still very necessary to design a *general* and *effective* de-duplication technique for silent compiler bugs.

In this work, we propose a novel technique, called **D**^{3} (**D**e-**D**uplication via three-**D**imensional analysis), to solve the de-duplication problem on silent compiler bugs. As there is no bug-relevant information from the *testing result* for a silent bug (e.g., error messages), D^3 characterizes the silent bug from the *testing process* for de-duplication. The general process of triggering a compiler bug can be depicted as follows: a test program is provided to the compiler, then the compiler is executed by compiling the test program under different optimization levels (in which the buggy compiler code is triggered), finally a bug is detected via differential testing (i.e., comparing the outputs under different optimization levels). That is, a test failure involves three key elements: *test program*, *optimizations*, and *test execution*. We remark that the three aspects are essential for a compiler bug (i.e., each aspect alone may not lead to the test failure) and thus in order to determine whether two test failures are caused by the same bug, we must comprehensively compare all the three aspects.

However, due to the huge amount of irrelevant details on all the three dimensions (such as irrelevant optimizations or irrelevant test program features), directly comparing the three aspects (e.g., comparing the function coverage of the executed compiler code) is unlikely to be effective due to the significant level of noise. Note that this is evident in the results reported by the existing study [20]. D^3 thus solves the problem by reducing the irrelevant details on each of the three dimensions systematically using a simple causality-analysis method.

- For *a bug-triggering test program*, D^3 first reduces it to the minimal one that can still trigger the bug based on the idea of generalized delta debugging [42, 56]. Actually, the minimal program still contains some bug-irrelevant program elements in order to ensure its validity, and thus D^3 further utilizes some mutation rules to transform the minimal bug-triggering test program to a set of passing test programs. Then, D^3 extracts the differences between the minimal bug-triggering test program and the set of passing ones as the bug-causal features in this dimension.
- For *optimizations*, D^3 opens the bug-triggering optimization level as a set of optimization options and then adapted delta debugging [56] to reduce it as a minimal set of optimization options that can still trigger the bug, which is taken as the bug-causal features in this dimension.
- For *test execution*, D^3 collects function coverage achieved by the bug-triggering test program as the original information in this dimension (similar to the existing work [20]). Then, it identifies the bug-causal functions from all the covered functions by estimating their suspicious scores via the idea of spectrum-based bug localization [6], and finally treats highly suspicious functions as the bug-causal features in this dimension.

By combining the three-dimensional bug-causal features, D^3 calculates the distance among test failures, and prioritizes them based on the furthest-point-first strategy [25] following the existing work [20]. In this way, the test failures that are more likely to trigger different bugs can be diagnosed by developers earlier.

To evaluate the effectiveness of D^3, we conducted an empirical study based on four released datasets, which contain 2,024 test failures caused by 62 uniques bugs from four versions of GCC [3] and LLVM [4]. Our results show that D^3 significantly outperforms the two state-of-the-art compiler bug de-duplication techniques (called Tamer [20] and Transformer [23] in our work). The average improvement of D^3 over Tamer and Transformer is 19.36% and 51.43% in identifying unique silent compiler bugs when analyzing the same number of test failures. Besides, our study also confirmed the significant contribution of each dimension of features to the overall effectiveness of D^3.

In summary, we make the following major contributions:

- We propose a novel technique (D^3) for addressing the de-duplication problem on silent compiler bugs, which systematically considers three-dimensional information and extracts bug-causal features from each dimension.
- We conducted an empirical study to investigate the effectiveness of D^3. The results demonstrate that D^3 significantly outperforms existing approaches, which illustrates the effectiveness of noise reduction and extraction of bug-causal features for characterizing test failures.
- We developed a tool for D^3 and released it as well as our experimental data for replication and future research.

2 MOTIVATION

Here, we use an example to motivate our technique. Figure 1 shows two bug-triggering test programs that correspond to the same bug (i.e., duplicate bugs) in GCC-4.4.0. Both of them produced inconsistent outputs under the optimization levels "-O0" and "-O1".

Identifying duplicate bugs is important to save debugging effort. The state-of-the-art technique proposed by Chen et al. [20] measures the similarity of compiler function coverage achieved by the corresponding bug-triggering test programs for silent bug de-duplication. However, the complete coverage tends to be large due to the complexity of compilers, but the coverage causal to a test failure is often very small [13, 15]. Hence, there could be too much noise in the compiler coverage for bug de-duplication, thus negatively affecting the de-duplication effectiveness. For example, although the two test programs (in Figure 1) trigger the same bug, the compiler coverage achieved by them differs largely. Specifically, the first test program covers 3,427 functions while the second one

```
1  int printf(const char *, ···);
2  union {
3     signed a : 29;
4     int b;
5  } const c = {447019919};
6  int main() {
7     printf("%d\n", c.b);
8     return 0;
9  }
```

(a) Failing test program - 1

```
1  int printf(const char *, ···);
2  union {
3     char a;
4     unsigned b : 5;
5  } const c[9] = {34};
6  int main() {
7     int d = 0;
8     for ( ; d < 9; d++) {
9        printf("%d\n", c[d].b);
10    }
11    return 0;
12 }
```

(b) Failing test program - 2

Figure 1: Illustrative example

covers 4,331 functions. This causes that the two test failures are regarded as non-duplicates by the function-coverage-based technique proposed by Chen et al. [20]. Therefore, *identifying the bug-causal coverage information from the complete coverage can be helpful to achieve more accurate bug de-duplication.* Indeed, we adopted the spectrum-based fault localization method (i.e., Ochiai [6]) to identify highly suspicious functions for each test failure, and found that Top-5 suspicious functions for the two test failures are the same, indicating that they are very likely to be duplicate.

In fact, the triggering of a bug is not only relevant to the test execution information (i.e., compiler coverage). Both the bug-triggering test program and the bug-triggering optimizations can also provide bug-causal information, which could help improve the effectiveness of bug de-duplication. In this example, the two bug-triggering test programs look dissimilar, but actually the bug-causal program elements in them are the same. Specifically, when we delete the initialization of the `union` variable from the two test programs (i.e., `c` at Line 5 in Figure 1a and `c[9]` at Line 5 in Figure 1b), both of them become passing, indicating that the initialization of the `union` variable is bug-causal for both of test failures. With the bug-causal test-program features, they can be identified as duplicates accurately. Moreover, after reducing the set of optimizations enabled in the bug-triggering optimization level to the minimal set of optimizations that still trigger the bug, both of test failures obtain the same minimal set (i.e., only the `-ftree-fre` optimization). This is also helpful to identify them as duplicates. Therefore, *both test program and optimization information can facilitate bug de-duplication, but it is also necessary to extract bug-causal features from them.*

3 APPROACH

In this work, we propose an effective technique for solving the duplication problem on silent compiler bugs, called D^3. D^3 identifies duplicate silent bugs from a set of test failures by systematically analyzing them. Specifically, for a silent bug (denoted as **b**), D^3 considers three-dimensional information to depict the test failure, i.e., *test program* (denoted as **p**), *optimizations* (denoted as **o**), and *test execution* (denoted as **c**). Then, D^3 measures the distance among test failures based on the three-dimensional information for silent bug de-duplication. However, only a small portion of information in

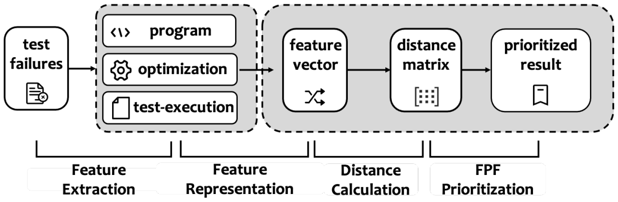


Figure 2: Overview of D^3

p, **o**, and **c** is bug-causal. Hence, D^3 systematically conducts causal analysis to identify those bug-causal information from each of the three dimensions. That is, D^3 extracts bug-causal features from **p**, **o**, and **c**, denoted as $\mathbf{p}^{\checkmark}$, $\mathbf{o}^{\checkmark}$, and $\mathbf{c}^{\checkmark}$, for improving the accuracy of bug de-duplication. Figure 2 presents the workflow of D^3. In the following, we introduce the bug-causal feature extraction and distance calculation process for the three dimensions, respectively (Section 3.1 for *test program* dimension, Section 3.2 for *optimization* dimension, Section 3.3 for *test execution* dimension). Then, we present how to integrate them for test failure prioritization (thus silent bug de-duplication) (Section 3.4).

3.1 Test Program Dimension

3.1.1 Bug-Causal Feature Extraction. As discussed in the existing work [11], only a small portion in a test program is causal to the triggering of a compiler bug, which tends to involve the combination of some program elements. We call them *bug-causal test-program features.* However, a bug-triggering test program can be large, e.g., test programs generated by Csmith [54] (one of the most widely-used C test program generators) contain thousands of lines of code. Hence, identifying bug-causal features from a large bug-triggering test program is challenging. Following the compiler debugging practice [45], D^3 first reduces the large bug-triggering test program to the minimal one that still triggers the bug, which can largely reduce the feature space. In the literature, a number of methods have been proposed for the task of test program reduction [9, 38, 42, 57]. Instead of re-inventing the wheel, D^3 adopts a widely-used test program reduction tool (i.e., C-Reduce [42]) to reduce each bug-triggering test program.

Even after such program reduction, not all the program elements in the reduced program are causal to the triggered bug. This is because some program elements are required to ensure the validity of the minimal test program. However, for bug de-duplication, these validity-required but bug-irrelevant features are also noise. To identify the program elements causal to the triggering of a bug, D^3 implements a set of mutation rules and applies them to the minimal bug-triggering test program in order to construct passing test programs with minimal changes. Then, the minor differences between them can be regarded as the bug-causal features. Since a bug tends to correspond to a combination of several program elements, different passing test programs can be constructed when applying different mutation rules to the bug-triggering test program or applying the same mutation rule to different locations in the bug-triggering one. Hence, D^3 aggregates the differences between a set of passing test programs and the bug-triggering test program

```
int printf(const char *, ···);
union {
  signed a : 29;
  int b;
} const c = {447019919};
int main() {
  printf("%d\n", c.b);
  return 0;
}
```

(a) Failing test program

```
int printf(const char *, ···);
union {
  signed a : 29;
  int b;
} const c;
int main() {
  printf("%d\n", c.b);
  return 0;
}
```

(b) Passing test program

Figure 3: Example of passing test program generation

Algorithm 1: Passing Test Program Generation

```
Input: $\mathcal{R}$: A set of mutation rules
       $\mathcal{FP}$: A failing test program
Output: $P$: A set of passing test programs
1  Function opportunities($r$, $\mathcal{FP}$):
2      $\mathcal{L}(r) \leftarrow \{\}$ ;
3      for $pos$ in $iterator(\mathcal{FP})$ do
4          if $is\_available(r, pos)$ then
5              $\mathcal{L}(r) \leftarrow \mathcal{L}(r) \cup \{pos\}$
6      return $\mathcal{L}(r)$ ;
7  Function ProgramGeneration($\mathcal{T}$, $\mathcal{FP}$):
8      for $r$ in $\mathcal{R}$ do
9          $\mathcal{L}(r) \leftarrow opportunities(r, \mathcal{FP})$ ;
10         for $l$ in $\mathcal{L}(r)$ do
11             $P' \leftarrow mutate(\mathcal{FP}, r, l)$ ;
12             if $P'$ is valid && $P'$ is passing then
13                 $\mathcal{P} \leftarrow \mathcal{P} \cup \{P'\}$ ;
14     return $\mathcal{P}$ ;
```

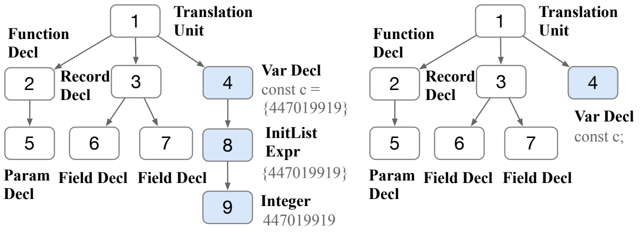


Figure 4: The AST differences of the example in Figure 3

to identify bug-causal features, which is also helpful to avoid incurring bias due to a single passing test program. In the following, we first illustrate the process of passing test program construction via mutation, and then present the process of difference extraction for identifying bug-causal features.

Passing test program construction. Inspired by the existing work [42], we consider four categories of mutation rules in D^3: (1) *identifier-level mutation* (e.g., changing an identifier to a constant, removing the qualifier of an identifier), (2) *operator-level mutation* (e.g., changing an operator to another one), (3) *delimiter-level mutation* (e.g., removing a pair of balanced parentheses), and (4) *statement-level mutation* (e.g., removes one or more statements following Berkeley Delta [56], changing an union to a struct following the source-to-source transformations in C-reduce [42]). The complete list of mutation rules can be found at our project homepage [2]. In particular, D^3 just performs first-order mutation to construct each passing test program, so that the difference between each pair of passing and bug-triggering test programs can be guaranteed to be minimal for affecting the triggering of the bug, indicating that noise can be effectively filtered out.

We formally illustrate the process of passing test program construction with Algorithm 1. For each mutation rule r, D^3 first identifies all the locations in the bug-triggering test program (denoted as $\mathcal{L}(r)$), where this mutation rule is applicable (Line 1-5, 9). For each location l in $\mathcal{L}(r)$, D^3 then applies r to l for constructing a mutated test program (denoted as P') (Line 11). If P' is a passing test program, D^3 saves it (Line 13). If P' is still a bug-triggering test program, D^3 discards it and moves to the next location. After enumerating all the mutation rules on all the corresponding applicable locations, the process terminates and a set of passing test programs are produced. In particular, Figure 3 shows the illustrative example, where Figure 3a is a minimal bug-triggering test program shown in Figure 1 while Figure 3b is a passing test program constructed by the mutation process. In this example, the triggering of this bug is related to the initialization and access of c. When we removed this part by mutation, the mutated program becomes a passing one.

Difference extraction. After obtaining a set of passing test programs from a minimal bug-triggering test program, D^3 extracts their differences at the AST (Abstract Syntax Tree) level as the bug-causal features. Instead of the token level, the AST level can capture both syntactic and structural differences, which can more comprehensively represent bug-causal features. Following the existing work [27, 51], D^3 depicts the difference between the bug-triggering test program and each of the passing ones as the operations on the AST corresponding to the bug-triggering test program. Given a modified node N_s and its parent node N_p, we consider the three kinds of operations in D^3: *insertion* (inserting N_s under N_p), *deletion* (deleting N_s under N_p), and *update* (updating N_s under N_p). Here, the parent node provides the context information in order to more precisely represent bug-causal features, since the same operation under different contexts could lead to different testing results (i.e., passing or failing). Note that if all the nodes of a subtree are modified, D^3 directly uses the root node of this subtree as N_s following the existing work [27, 33]. In this way, D^3 extracts the bug-causal features from the test-program dimension, which are a group of operations based on the set of AST differences between passing test programs and the bug-triggering one.

Figure 4 shows the ASTs for the pair of bug-triggering and passing test programs (shown in Figure 3). From Figure 4, the difference at the AST level between them can be represented as "deleting an *InitList Expression* under a *Var Declaration*". Although there is another modified node in Figure 3 (i.e., *Integer*), we directly use the *InitList Expression* node as N_s. This is because (1) the two modified nodes belong to the same subtree and the *InitList Expression* node is the root of this subtree, (2) and all the child nodes of this subtree (i.e., *Integer*) are modified, which can be completed by a single operation on the root node of the subtree (i.e., deleting the subtree).

3.1.2 Distance Calculation. Before distance calculation, D^3 vectorizes the extracted bug-causal test-program features (i.e., a group of AST operations) for each test failure. Here, D^3 counts the occurrence times of each operation in the group of AST operations and ranks these AST operations as the descending order of their occurrence times. Then, D^3 obtains the test-program feature vector, where each element represents a specific operation and the element value is the rank of the operation. The reason why we adopt such a vectorization method is that a frequently-occurring operation indicates that it is more likely causal to the triggered bug. Accordingly, if similar lists of operations occur more frequently for two test failures, they are more likely to be caused by the same bug.

After obtaining the test-program feature vector for each test failure, D^3 calculates the distance between two test failures based on the Spearman correlation coefficient [55]. This is because our test-program feature vectors embody the ranking information, and this metric is a statistical measure of the strength of a monotonic relationship between two paired variables or ranking lists [55]. In our scenario, it measures how similar two ranking lists of operations for two test failures are. Specifically, given two test failures b_i and b_j and the corresponding test-program feature vectors $v(\mathbf{p}_\mathbf{i}^{\checkmark}) = (x_{i1}, x_{i2}, \ldots, x_{iu})$ and $v(\mathbf{p}_\mathbf{j}^{\checkmark}) = (x_{j1}, x_{j2}, \ldots, x_{ju})$, where u refers to the number of AST operations, Formula 1 shows the calculation of the Spearman correlation coefficient between $v(\mathbf{p}_\mathbf{i}^{\checkmark})$ and $v(\mathbf{p}_\mathbf{j}^{\checkmark})$:

$$spearman(v(\mathbf{p}_\mathbf{i}^{\checkmark}), v(\mathbf{p}_\mathbf{j}^{\checkmark})) = 1 - \frac{6 \times \sum_{k=1}^{u} (x_{ik} - x_{jk})^2}{u(u^2 - 1)} \quad (1)$$

The Spearman correlation coefficient ranges from -1 to 1 and a larger value indicates the higher similarity. Moreover, the distance cannot be negative numbers. Hence, D^3 calculates the distance between b_i and b_j in this dimension as shown in Formula 2.

$$d_p(\mathbf{p}_\mathbf{i}^{\checkmark}, \mathbf{p}_\mathbf{j}^{\checkmark}) = 1 - spearman(v(\mathbf{p}_\mathbf{i}^{\checkmark}), v(\mathbf{p}_\mathbf{j}^{\checkmark})) \quad (2)$$

3.2 Optimization Dimension

3.2.1 Bug-Causal Feature Extraction. In general, a test program is compiled under several optimization levels (e.g., -O1, -O2, -O3, and -Os for GCC and LLVM) [16]. The bugs may be triggered under certain optimization levels rather than any levels. Hence, a test failure has the corresponding bug-triggering optimization level. An optimization level enables a set of optimizations pre-defined by compiler developers, which aims to ease the practical use of compiler optimizations [17]. That is, the set of optimizations in an optimization level are not pre-defined for detecting bugs, indicating that not all the enabled optimizations in the bug-triggering optimization level are causal to the triggering of the bug. Therefore, it is necessary to identify the bug-causal optimizations from the whole set as the bug-causal features in this dimension.

Here, D^3 adapts the idea of delta debugging [56] to identify the minimal bug-triggering optimizations. Specifically, supposed that the whole set of optimizations in the bug-triggering optimization level is denoted as c, if c contains only one optimization, it is the bug-triggering one. Otherwise, D^3 evenly splits c into two subsets, denoted as c_1 and c_2, where c_1 just enables half of optimizations in c and disables the other half while c_2 is the opposite of $c1$. Then, it checks whether each of them still makes the test program trigger the bug or not. There are three possible outcomes:

1. c_1 makes the test program trigger the bug, indicating that c_1 contains the bug-triggering optimizations;
2. c_2 makes the test program trigger the bug, indicating that c_2 contains the bug-triggering optimizations;
3. both of them cannot make the test program trigger the bug, indicating that the combination of some optimizations in c_1 and c_2 composes the bug-triggering optimizations.

For the first two cases, D^3 continues the binary search in the bug-triggering subset. For the last case, D^3 performs the binary search in one subset by still reserving all the optimizations in the other subset, respectively. The reduction process terminates until a minimal set of optimizations that can trigger the bug is found, which is regarded as the bug-causal features extracted from the optimization dimension. We refer the readers to the existing work [56, 57] for a discussion on the soundness of such a method for identifying the minimal causal optimizations.

3.2.2 Distance Calculation. Since a compiler optimization can be enabled or disabled, D^3 represents the extracted bug-causal optimization features for a test failure as a vector, where each element represents a specific optimization and the element value is 0 (indicating the optimization is disabled) or 1 (indicating the optimization is enabled). Then, D^3 calculates the distance between two test failures based on the Euclidean metric. Specifically, assuming that the optimization feature vectors for b_i and b_j are $v(\mathbf{o}_\mathbf{i}^{\checkmark}) = (x_{i1}, x_{i2}, \ldots, x_{iv})$ and $v(\mathbf{o}_\mathbf{j}^{\checkmark}) = (x_{j1}, x_{j2}, \ldots, x_{jv})$ (where v refers to the total number of compiler optimizations provided by the compiler under test) respectively, Formula 3 shows the calculation of the Euclidean distance between $v(\mathbf{o}_\mathbf{i}^{\checkmark})$ and $v(\mathbf{o}_\mathbf{j}^{\checkmark})$:

$$d_o(\mathbf{o}_\mathbf{i}^{\checkmark}, \mathbf{o}_\mathbf{j}^{\checkmark}) = Euclidean(v(\mathbf{o}_\mathbf{i}^{\checkmark}), v(\mathbf{o}_\mathbf{j}^{\checkmark})) = \sqrt{\sum_{k=1}^{v} (x_{ik} - x_{jk})^2} \quad (3)$$

3.3 Test Execution Dimension

3.3.1 Bug-Causal Feature Extraction. For a test failure, the bug-triggering test program must execute the buggy code in the compiler under test. With this intuition, Chen et. al. [20] uses the *function coverage* achieved by the test failure as the features for bug de-duplication. While it outperforms the other features studied at the time [20], indicating the importance of test-execution features for the task of bug de-duplication, its effectiveness is still limited (also confirmed by our study in Section 4.5). The main reason is that, the compiler code executed by the bug-triggering test program is often large, since various functionalities of the compiler (e.g., lexical analysis, syntactic analysis, and semantic analysis) have to be invoked when compiling any test programs. However, the buggy code tends to just occupy a very small portion [13, 15], indicating that there is much noise in the compiler code covered by the test failure for bug de-duplication. Therefore, it is necessary to identify bug-causal test-execution features from this dimension for more accurate de-duplication.

In particular, D^3 considers the test-execution features at the compiler function level following the existing work [20]. Inspired by the research on automatic fault localization [6, 46], it is challenging to accurately identify the buggy function from all the covered functions by the bug-triggering test program, Here, D^3 adopts the idea of the widely-studied spectrum-based fault localization (SBFL) [29] to estimate the suspiciousness of each compiler function covered by the bug-triggering test program. In our scenario, the key insight of SBFL is to utilize a set of passing test programs to reduce the suspiciousness of each covered compiler function. If a compiler function covered by the bug-triggering test program is covered by more passing test programs, its suspicious score should be smaller, and vice versa. D^3 uses the developer-provided test suite as the set of passing test programs rather than the constructed passing test programs through mutation (presented in Section 3.1). The reasons are twofold: (1) The number of the former is significantly larger than the latter, which is more helpful to accurately estimate the suspiciousness of each covered function; (2) The set of passing test programs for all the test failures are the same when using the former, indicating the coverage collection process for the set of passing test programs is conducted only once and then the collected coverage information can be used for all the test failures. Hence, the coverage collection overhead can be largely reduced.

D^3 adopts the state-of-the-art aggregation-based SBFL [46] to estimate the suspicious score of each covered function based on the set of passing test programs. D^3 first estimates the suspicious score of each statement in each covered function, where the widely-used formula [6] is adopted as shown in Formula 4.

$$score(s) = \frac{ef_s}{\sqrt{(ef_s + nf_s)(ef_s + ep_s)}} \tag{4}$$

where ef_s and nf_s represent the number of bug-triggering test programs that execute and do not execute the statement s, and ep_s represents the number of passing test programs that execute the statement s. Since D^3 estimates the suspicious score of each covered function for each test failure, there is only one bug-triggering test program, and thus $ef_s \leq 1$. Then, D^3 aggregates the suspicious scores of the covered statements in a function to the suspicious score of the function following the existing work [11, 46]. The aggregation formula is defined as follows.

$$score(f) = \max(score(s_i)), 1 \leq i \leq n_f \tag{5}$$

where n_f is the total number of the statements in the function f. Overall, for a test failure, D^3 can identify bug-causal test-execution features based on the suspicious scores of all the compiler functions covered by the bug-triggering test program.

3.3.2 Distance Calculation. For a test failure, D^3 represents the extracted test-execution features as a vector, where each element represents a specific compiler function and the element value is the suspicious score of the function. Then, D^3 uses the Euclidean metric to calculate the distance between two test failures following the practice of distance calculation for optimization feature vectors. Specifically, let the test-execution feature vectors for b_i and b_j be $v(\mathbf{c}_\mathbf{i}^{\checkmark}) = (x_{i1}, x_{i2}, \ldots, x_{iw})$ and $v(\mathbf{c}_\mathbf{j}^{\checkmark}) = (x_{j1}, x_{j2}, \ldots, x_{jw})$ (where w refers to the total number of compiler functions) respectively, Formula 6 shows the distance calculation between $v(\mathbf{c}_\mathbf{i}^{\checkmark})$ and $v(\mathbf{c}_\mathbf{j}^{\checkmark})$:

$$d_c(\mathbf{c}_\mathbf{i}^{\checkmark}, \mathbf{c}_\mathbf{j}^{\checkmark}) = Euclidean(v(\mathbf{c}_\mathbf{i}^{\checkmark}), v(\mathbf{c}_\mathbf{j}^{\checkmark})) = \sqrt{\sum_{k=1}^{w}(x_{ik} - x_{jk})^2} \tag{6}$$

3.4 Test Failure Prioritization

Following the existing work [20], we also transform the bug de-duplication problem as the test failure prioritization problem. That is, D^3 ranks the test failures that are more likely to be caused by different silent bugs higher by calculating the distance among test failures based on the three dimensional bug-causal features. Specifically, D^3 calculates the distance between two test failures b_i and b_j according to Formula 7, where ω_1, ω_2 and ω_3 are constant parameters weighting the aforementioned three-dimensional features to avoid the influence of different distributions among the three dimensions. The larger the distance value is, the smaller the possibility that b_i and b_j are caused by the same bug is.

$$\begin{aligned} dist(b_i, b_j) = \omega_1 * \overline{d}_p(\mathbf{p}_\mathbf{i}^{\checkmark}, \mathbf{p}_\mathbf{j}^{\checkmark}) + \omega_2 * \overline{d}_o(\mathbf{o}_\mathbf{i}^{\checkmark}, \mathbf{o}_\mathbf{j}^{\checkmark}) \\ + \omega_3 * \overline{d}_c(\mathbf{c}_\mathbf{j}^{\checkmark}, \mathbf{c}_\mathbf{j}^{\checkmark}) \end{aligned} \tag{7}$$

In particular, since the scales of the results of d_p, d_o, and d_c are different, D^3 normalizes the distance results in each dimension into the interval [0, 1] through the widely-used *min-max* normalization method [31] respectively, in order to adjust them to a common scale. In the formula, we use $\overline{d}_p$, $\overline{d}_o$ and $\overline{d}_c$ to represent the corresponding normalized value. Finally, we set $\omega_1 = 1$, $\omega_2 = \frac{mean(\overline{d}_p(\cdot))}{mean(\overline{d}_o(\cdot))}$, and $\omega_3 = \frac{mean(\omega_1 * \overline{d}_p(\cdot) + \omega_2 * \overline{d}_o(\cdot))}{mean(\overline{d}_c(\cdot))}$, where $\overline{d}_p(\cdot)$, $\overline{d}_o(\cdot)$ and $\overline{d}_c(\cdot)$ denote the distance results for all the test failures computed by the corresponding function, and *mean* refers to the mean function. In this way, the three-dimensional features can be well balanced.

With the final distance between each pair of test failures from the overall perspective of the three-dimensional features, D^3 ranks all the test failures by the furthest-point-first (FPF) algorithm [25] following the existing work [20]. Specifically, D^3 first randomly selects a test failure as the starting point of the prioritization result and labels it as *already-prioritized.* Then, D^3 selects the test failure that has the maximal minimum final distance with the test failures labeled as *already-prioritized*, as the next one in the prioritization result, and also labels it as *already-prioritized.* The process terminates until all the test failures are labeled as *already-prioritized.* According to the prioritization result, developers can diagnose these test failures in turn and then can obtain more unique bugs by investigating fewer test failures, thus largely saving developers' effort.

4 EVALUATION

In our study, we aim to address the following research questions:

- **RQ1:** How does D^3 perform in silent compiler bug de-duplication?
- **RQ2:** Does each dimension of features contribute to D^3?
- **RQ3:** How does D^3 perform under different configurations?

4.1 Datasets

Our datasets used in the study are from two sources, i.e., the dataset released by the existing study on compiler bug de-duplication [20]

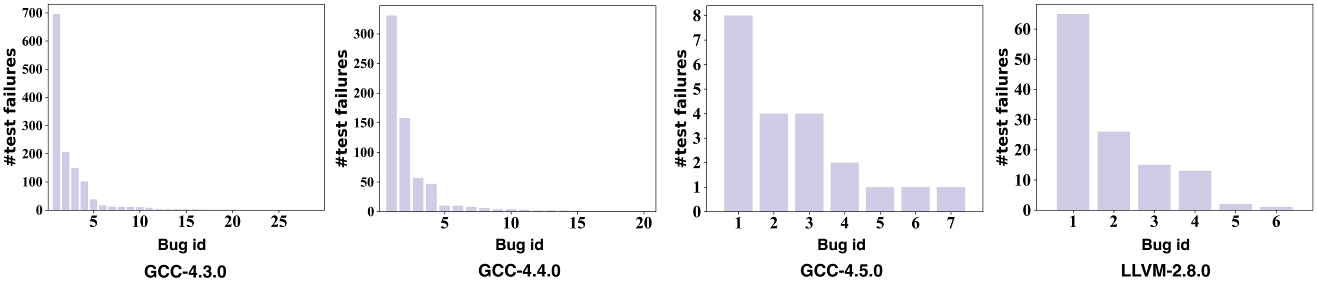


Figure 5: The relationship between test failures and unique bugs

and the datasets released by the existing studies on testing compilers [17, 18]. Regarding the former, it consists of 1,275 test failures caused by 35 unique silent bugs for GCC-4.3.0. With some non-trivial effort, we aim to reproduce each test failure in our experimental environment, which is necessary as D^3 needs to collect the function coverage for each test failure. Finally, 1,235 test failures can be successfully reproduced, which corresponds to 29 unique silent bugs. The remaining test failures are not reproduced mainly due to the environment and architecture differences. Regarding the latter, we collected three datasets in total. All of them are the results of evaluating some compiler testing techniques in the existing studies [17, 18]. They contains 647 test failures caused by 20 unique silent bugs in GCC-4.4.0, 26 test failures caused by 7 unique silent bugs in GCC-4.5.0, and 116 test failures caused by 6 unique silent bugs in LLVM-2.8, respectively. In particular, the original datasets also contain crash bugs but we filtered them out, since our work aims to address the challenging duplication problem on silent compiler bugs. All the test failures from the three datasets can be successfully reproduced in our experimental environment. In total, we used four datasets from four versions of two popular C compilers (i.e., GCC and LLVM), including 2,024 test failures caused by 62 unique silent bugs. All of these bugs are real-world, which is helpful to demonstrate the practicability of D^3 to some degree.

Figure 5 shows the relationship between test failures and unique bugs, where the x-axis represents each unique silent bug while the y-axis represents the number of test failures caused by the corresponding bug. We found that the distribution of test failures is very uneven to the bugs for each dataset. For example, for the dataset of GCC-4.3.0, there is a bug that caused 696 test failures, yet some bugs caused only one test failure. This phenomenon further shows the compiler bug de-duplication task is significantly challenging.

4.2 Implementation and Environment

We implemented D^3 based on several mature tools or third-party libraries. Specifically, it transforms a test program to an AST via tree-sitter [5], extracts AST differences via GumTree [24], collects function coverage via Gcov [1]. All the studied techniques (D^3 and the compared ones to be presented in Section 4.3) involve randomness (e.g., randomly selecting the first test failure during prioritization), and thus we repeated each technique on each dataset 100 times and calculated the average result for comparison.

We released our implementation and all the experimental data at the project homepage **https://github.com/chenyangyc/D3** for replication, future research, and practical use. All of our experiments were conducted on a workstation with 28-core CPU, 120G memory and CentOS 7 operating system.

4.3 Compared Techniques

To answer RQ1, we compared D^3 with two state-of-the-art compiler bug de-duplication techniques.

- *Chen et al.'s technique (named **Tamer** in this paper)* [20]: It investigates various combinations of several program features for compiler bug de-duplication. Here, we adopted the most effective one, which utilizes the compiler function coverage achieved by the bug-triggering test program (i.e., the number of execution times for each function) to calculate the distance between each pair of test failures. Then, Tamer ranks all the test failures with the FPF algorithm. We re-implemented it based on the corresponding paper and reproduced their results to ensure the correctness of our re-implementation.
- *Donaldson et al.'s technique (named **Transformer** in this paper)* [23]: It is designed specific to transformation-based compiler testing. Its insight is that if two bug-triggering test programs are generated based on the same set of transformations, they are regarded to trigger the same bug. However, our datasets are not based on transformation-based compiler testing, and thus we adapted it to fit our scenario. Specifically, we applies our designed mutation rules (presented in Section 3.1) to transform each bug-triggering test program to be passing. If two bug-triggering test programs obtain the similar sets of transformations, they are more likely to trigger the same bug. For sufficient comparison, the last step of Transformer is to rank all the test failures with the FPF algorithm as well.

To answer RQ2, we constructed six variants of D^3 to investigate the contribution of each dimension of features in D^3, including $\mathbf{D}^3_{noP}$ (removing the test-program dimension), $\mathbf{D}^3_{noO}$ (removing the optimization dimension), $\mathbf{D}^3_{noC}$ (removing the test-execution dimension), $\mathbf{D}^3_{P}$ (only using the test-program dimension), $\mathbf{D}^3_{O}$ (only using the optimization dimension) and $\mathbf{D}^3_{C}$ (only using the test-execution dimension).

In RQ3, we considered two important configurations in D^3, i.e., the used SBFL formula for estimating the suspiciousness of each function and the method of integrating the distance results from the three dimensions. Indeed, there are some other common methods to complete the two tasks in the literature. Regarding the former, besides the default Ochiai in D^3, we also studied the other five widely-studied SBFL formulae, including Tarantula [30], Jaccard [19], Ochiai2 [39], Kulczynski2 [37], and D^* [52]. The details about them can be found in the survey of fault localization [29, 46]. Regarding the latter, besides the default weighted summation method in D^3, we also studied the *min-max summation* method, which directly sums the distance from each dimension after min-max normalization. That is, ω_1, ω_2, and ω_3 are all equal to 1 in Formula 7.

4.4 Measurements

Following the existing work [20], we solved the bug de-duplication problem by prioritizing test failures. Hence, we adopted the widely-used metric in prioritization, i.e., **RAUC-n**, to measure the effectiveness of each de-duplication technique. Specifically, it transforms the prioritization result produced by a technique to a plot, where the x-axis represents the number of test failures and the y-axis represents the number of corresponding unique bugs. Then, it calculates the ratio of the area under the curve for a technique to that for the ideal prioritization. Larger RAUC values mean more unique bugs that developers can identify when investigating the same number of test failures, indicating better de-duplication effectiveness. Since test failures tend to be not diagnosed by developers *completely* in practice as demonstrated by the existing work [20, 21, 28], we set *n* to Top 20% and 100% test failures for measuring the effectiveness of each de-duplication technique, respectively.

RAUC-n measures the effectiveness of each technique from the overall view for the prioritization result. We further analyzed the effectiveness of each technique for identifying each unique bug according to the prioritization result. Specifically, we measured the number of investigated test failures before identifying each unique bug. This metric is called **wasted effort** in the study. Fewer test failures are investigated before identifying each unique bug means that less developers' effort is wasted on analyzing duplicate bugs, indicating better bug de-duplication effectiveness.

4.5 Results and Analysis

4.5.1 RQ1: Effectiveness of D^3. Table 1 shows the comparison results among D^3, Tamer, and Transformer in terms of RAUC-20% and RAUC-100%. From this table, D^3 outperforms both Tamer and Transformer on all the datasets regardless of RAUC-20% or RAUC-100%, indicating its stably good effectiveness. For example, on GCC-4.3.0, the RAUC-20% value of D^3 is 0.8768 while those of Tamer and Transformer are 0.8017 and 0.4037, where the improvement of the former over the latter two is 9.37% and 117.19% respectively. In terms of the average RAUC-20% value across all the datasets, D^3 improves Tamer and Transformer by 19.36% and 51.43%, respectively. In terms of the average RAUC-100% value across all the datasets, D^3 improves Tamer and Transformer by 9.54% and 20.85%, respectively. The results demonstrate the overall effectiveness of D^3. In particular, the superiority of D^3 is more obvious in terms of RAUC-20% (than RAUC-100%), also demonstrating the practicability of D^3.

Table 1: The RAUC values of different techniques

RAUC-20%				
APP.	GCC-4.3.0	GCC-4.4.0	GCC-4.5.0	LLVM-2.8.0
Tamer	0.8017	0.6999	0.7805	0.6057
Transformer	0.4037	0.4277	0.7875	0.6576
D^3	0.8768	0.7722	0.9165	0.8716
RAUC-100%				
APP.	GCC-4.3.0	GCC-4.4.0	GCC-4.5.0	LLVM-2.8.0
Tamer	0.9527	0.9088	0.7920	0.8156
Transformer	0.7302	0.7560	0.7923	0.8576
D^3	0.9701	0.9380	0.9041	0.9778

Then, Table 2 presents the comparison results in terms of wasted effort for identifying each unique bug. In this table, Column “ID” presents each unique bug. Columns “D^3”, “Tamer” and “Transformer” present the number of test failures investigated before identifying each unique bug according to the prioritization result produced by each technique. Columns “$\Uparrow_{Tamer}(\%)$” and “$\Uparrow_{Transformer}(\%)$” present the improvements of D^3 over Tamer and Transformer, respectively. For example, before identifying the 16th unique bug in the dataset of GCC-4.3.0, developers need to investigate on average 27.11 test failures with D^3, while they need to investigate 41.53 and 258.96 test failures with Tamer and Transformer, respectively. That is, D^3 saves 34.72% and 89.53% effort compared with them. From this table, compared with Transformer, D^3 is able to save more effort for identifying each unique bug in each dataset. The improvement of D^3 over Transformer ranges from 3.08% to 90.09%. Compared with Tamer, D^3 performs better for 88.7% (55 out of 62) cases across the four datasets, and the improvement ranges from 1.79% to 85.18%. The results further demonstrate the superiority of D^3 in identifying each unique bug.

Although D^3 performs slightly worse than Tamer on some cases, the performance decline is quite small. In particular, on these cases, both D^3 and Tamer can accurately identify the unique bugs. For example, the first 7 unique bugs from GCC-4.3.0 can be identified by investigating 8.02 and 7.66 test failures on average when using D^3 and Tamer, respectively. In other words, almost no effort was wasted, demonstrating their effectiveness. Furthermore, we performed a paired sample Wilcoxon signed-rank test [53] at the significance level of 0.05 to investigate whether D^3 can significantly outperform the compared techniques in statistics. Since the number of unique bugs in the datasets of GCC-4.5.0 and LLVM-2.8.0 is relatively small, which cannot support the statistical test well, we performed the statistical test on the other two datasets. The *p-values* on the two datasets (GCC-4.3.0 and GCC-4.4.0) are both smaller than 0.002, indicating the significant superiority of D^3 over both Tamer and Transformer.

4.5.2 RQ2: Contributions of Each Dimension of Features. To investigate the contribution of each dimension of features in D^3, we compared D^3 with a set of its variants (introduced in Section 4.3). Table 3 presents the comparison results, where the second row present the average RAUC-20% result across all the dataset. We also performed a paired sample Wilcoxon signed-rank test at the significance level of 0.05 to investigate whether D^3 significantly

Table 2: Comparison between D^3, Tamer and Transformer in terms of wasted effort

GCC-4.3.0											
ID	D^3	Tamer	⇑ *Tamer(%)*	Transformer	⇑ *Transformer(%)*	ID	D^3	Tamer	⇑ *Tamer(%)*	Transformer	⇑ *Transformer(%)*
2	2.01	2.00	-0.50%	2.60	22.69%	16	27.11	41.53	34.72%	258.96	89.53%
3	3.15	3.04	-3.62%	5.41	41.77%	17	30.67	51.20	40.10%	302.11	89.85%
4	4.34	4.63	6.26%	9.06	52.10%	18	34.62	59.71	42.02%	349.23	90.09%
5	5.52	5.65	2.30%	14.86	62.85%	19	40.22	67.62	40.52%	393.40	89.78%
6	6.70	6.65	-0.75%	22.86	70.69%	20	48.19	76.75	37.21%	440.83	89.07%
7	8.02	7.66	-4.70%	31.98	74.92%	21	54.54	90.19	39.53%	501.09	89.12%
8	9.39	8.88	-5.74%	45.04	79.15%	22	61.24	102.58	40.30%	558.72	89.04%
9	11.01	11.34	2.91%	63.02	82.53%	23	67.64	109.68	38.33%	621.68	89.12%
10	12.79	13.98	8.51%	83.54	84.69%	24	76.77	119.48	35.75%	692.86	88.92%
11	15.00	18.06	16.94%	107.93	86.10%	25	87.98	138.56	36.50%	781.19	88.74%
12	17.46	21.37	18.30%	131.27	86.70%	26	101.20	153.73	34.17%	853.59	88.14%
13	19.40	24.26	20.03%	156.39	87.60%	27	118.69	165.72	28.38%	936.48	87.33%
14	21.72	29.06	25.26%	183.50	88.16%	28	133.22	202.61	34.25%	1,034.27	87.12%
15	24.36	37.78	35.52%	219.01	88.88%	29	454.07	602.25	24.60%	1,131.07	59.85%
GCC-4.4.0											
ID	D^3	Tamer	⇑ *Tamer(%)*	Transformer	⇑ *Transformer(%)*	ID	D^3	Tamer	⇑ *Tamer(%)*	Transformer	⇑ *Transformer(%)*
2	2.08	3.47	40.06%	2.50	16.80%	12	32.87	27.33	-20.27%	143.30	77.06%
3	3.43	5.48	37.41%	5.37	36.13%	13	37.77	40.47	6.67%	173.08	78.18%
4	4.87	6.50	25.08%	10.03	51.45%	14	44.05	50.50	12.77%	212.47	79.27%
5	5.98	7.75	22.84%	17.45	65.73%	15	54.97	67.15	18.14%	250.62	78.07%
6	7.25	8.76	17.24%	29.49	75.42%	16	69.59	97.99	28.98%	297.23	76.59%
7	10.38	10.57	1.80%	40.41	74.31%	17	81.31	111.64	27.17%	350.31	76.79%
8	14.02	18.02	22.20%	53.79	73.94%	18	88.75	123.01	27.85%	414.23	78.57%
9	17.94	21.39	16.13%	70.66	74.61%	19	103.31	137.66	24.95%	488.89	78.87%
10	21.71	22.61	3.98%	88.95	75.59%	20	403.15	577.71	30.22%	550.67	26.79%
11	26.82	23.75	-12.93%	113.55	76.38%						
GCC-4.5.0											
ID	D^3	Tamer	⇑ *Tamer(%)*	Transformer	⇑ *Transformer(%)*	ID	D^3	Tamer	⇑ *Tamer(%)*	Transformer	⇑ *Transformer(%)*
2	2.20	2.24	1.79%	2.27	3.08%	5	7.19	9.60	25.10%	9.42	23.67%
3	3.31	3.52	5.97%	4.09	19.07%	6	8.89	13.69	35.06%	12.92	31.19%
4	4.78	5.83	18.01%	6.39	25.20%	7	10.97	16.44	33.27%	16.22	32.37%
LLVM-2.8.0											
ID	D^3	Tamer	⇑ *Tamer(%)*	Transformer	⇑ *Transformer(%)*	ID	D^3	Tamer	⇑ *Tamer(%)*	Transformer	⇑ *Transformer(%)*
2	2.04	2.12	3.78%	2.56	20.35%	5	7.21	48.67	85.18%	27.90	74.15%
3	3.52	3.74	6.02%	5.15	31.75%	6	15.96	59.51	73.19%	57.79	72.39%
4	5.17	13.87	62.76%	9.27	44.28%						

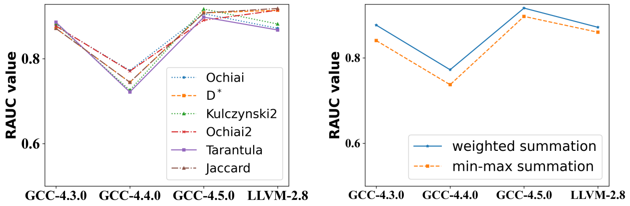
[1] The ID starts from 2, as all the techniques find the first bug with only one test failure investigated.

outperforms each variant by integrating all the datasets together, and present the p-values at the third row in Table 3. We further counted the percentage of the cases where D^3 outperforms each variant across all the datasets, and present the results at the last row in Table 3.

From Table 3, in terms of RAUC-20%, using three-dimensional features (i.e., D^3) outperforms using any combination of two dimensional features (i.e., D^3_{noP}, D^3_{noO}, D^3_{noC}), and the latter also outperforms using only one dimensional features (i.e., D^3_P, D^3_O, D^3_C). The results demonstrate the contribution of each dimension of features

Table 3: Comparison between D^3 and its variants

	D^3_P	D^3_O	D^3_C	D^3_{noP}	D^3_{noO}	D^3_{noC}	D^3
RAUC-20%	0.7540	0.7491	0.7549	0.8406	0.8175	0.8098	0.8593
p-value	1.49e-7	7.27e-7	2.17e-7	2.45e-6	2.46e-5	2.27e-8	-
Percentage	84.48%	81.03%	82.76%	75.86%	70.69%	93.10%	-



(a) Different SBFL formulae **(b) Different summations**

Figure 6: Performance of D^3 with different SBFL formulae and summations

to the overall effectiveness of D^3. Moreover, all the p-values are much smaller than 0.05, demonstrating that the contribution of each dimension of features is statistically significant. While the improvement of D^3 over D^3_{noP} is relatively small in terms of RAUC-20%, there are actually more than 80% bugs for which D^3 outperforms D^3_{noP}. Through investigation, we found the reason for the relatively small improvement is that for some cases, D^3 generates a very small number of passing test programs via mutation, which limits the effectiveness of the test-program features. In the future, we may introduce high-order mutation to increase the number of passing test programs, in order to further improve the effectiveness of D^3.

4.5.3 RQ3: Influence of Different Configurations. In RQ3, we first investigated the effectiveness of D^3 with different SBFL formulae, which are used to estimate the suspiciousness of each compiler function in D^3. Figure 6a shows the comparison results in terms of RAUC-20% on each dataset. From this figure, different SBFL formulae have similar effectiveness on all the datasets, and meanwhile there is no SBFL formula that can perform the best on all the datasets. That demonstrates the stable effectiveness of D^3 and in the future we may integrate several SBFL formulae in D^3 to improve the effectiveness of D^3.

We then compared our weighted summation of the three-dimensional features (which aims to avoid the influence of different distributions of the distance results in different dimensions) with the min-max summation method (without weighting). Figure 6b shows the comparison results in terms of RAUC-20%. From this figure, we found that on each dataset, our weighted summation method in D^3 outperforms the other method. The results demonstrate that inconsistent distributions among distances do have a negative impact. Hence, adopting the weighted summation method in D^3 is useful.

5 THREATS TO VALIDITY

The *internal* threat mainly comes from the implementations of D^3 and the baseline techniques. To mitigate this threat, we have performed a code review and replicated the results of baseline techniques to ensure the correctness. Besides, we have published all the implementations at our project homepage for replication and promoting future research.

```
int printf(const char *, ⋯);
struct {
    signed a : 4;
    signed b : 8;
} c, d = {5}
int main() {
    c.b |= d.b;
    printf(“%d\n”, c.a);
}
```

(a) Failing test program - 1

```
              ...
static short e() {
    b = 2;
    for (; b>=0; b - -) {
        c = 0;
        for (; c <= 7; c++) {
            a[b][c] = - (short)a[b][b];
            a[b][1] = a[b][b];
              ...
```

(b) Failing test program - 2

Figure 7: Deduplication example from LLVM-2.8

The *external* threat mainly lies in the used subjects. In our experiment, we have utilized the widely-used GCC and LLVM compilers like previous work [20, 28]. Furthermore, we have included all the 62 real-world silent compiler bugs from previous studies, which can be representative to some degree. However, the effectiveness of D^3 on a wider range remains to be evaluated in the future.

The *construct* threats mainly lie in the metrics, configurations, and the randomness involved in the results. In our study, we have employed the widely-used metrics by following previous studies [11, 20, 21, 28]. In fact, we also analyzed the overhead of our technique. The average time spent on each test program by D^3 is 45.64s while that by Tamer and Transformer is 1.36s and 25.57s. The overhead is acceptable in view of its best de-duplication effectiveness and much saved manual effort. Since this process is performed offline in practice and it can be further accelerated via parallel executions. To avoid configuration bias, we further conducted a series of comparative experiments to investigate their influence to the effectiveness of D^3 (Section 4.5.3). Finally, to mitigate the influence of randomness, we have repeated our experiments 100 times for each technique and calculated the average results as the representative.

6 DISCUSSION

6.1 Case Study

As presented in section 1, the three dimensions are essential for a compiler bug, i.e., each aspect alone may not lead to test failures. That is, the three dimensions are complementary to each other. We used two reduced test programs triggering different bugs as an example to further demonstrate their complementarity. They are selected from our LLVM-2.8 dataset and shown in Figure 7 (note that for the presentation purpose, we have used the simplified versions of programs).

Through the extraction of bug-causal features from the test-program dimension, we observed that the most frequent operation on AST differences for both test programs was “deleting an assignment expression under a compound statement”. Based solely on this dimension, the test failures were classified as duplicates. However, upon extracting bug-causal features from the other two dimensions, we found that the Top-20 suspicious functions did not overlap for both test failures, and the bug-causal optimizations were different. The first test failure utilized *-instcombine* optimization, whereas the second one employed *-adce -basiccg -constmerge -correlated-propagation -verify*. By integrating these two dimensions with the test-program dimension, we accurately identified the failures as non-duplicates. Here, Tamer incorrectly identifies them as duplicates due to a 98.2% overlap in the covered functions.

6.2 Limitation

D^3 is inapplicable in black-box scenarios, as it relies on compiler-code-coverage information for test execution. Tamer also suffers from this limitation, whereas Transformer does not, making it a complementary technique. In the case of closed-source compilers, compiler developers and maintainers can use D^3 in-house, which aligns with one of our research goals.

In the optimization dimension, the use of fine-grained optimizations is necessary, rendering D^3 impractical when such optimizations are unavailable. However, based on the results presented in Section 4.5.2, D^3 still outperforms Tamer and Transformer without this dimension. For instance, the average RAUC-20% of D^3 without this dimension is 0.8175, while Tamer and Transformer achieve 0.7219 and 0.5691, respectively (as computed from the results in Section 4.5.1). Consequently, it is acceptable to disregard this dimension when fine-grained optimizations are not available.

D^3 currently treats different optimizations equally in a binary manner. However, various optimizations in the minimal set may contribute differently to a test failure. Some of them are inherently buggy, while others are solely meant to activate buggy optimizations. Moreover, parameterized optimizations are currently configured with default settings without any tuning. We plan to conduct further studies to alleviate these limitations.

7 RELATED WORK

Bug De-duplication. Since our work aims at the problem of silent compiler bug de-duplication, the most related work to us are Tamer [20] and Transformer [23], which have been introduced in Section 4.3. We have compared the performance of our technique with them in our experimental study. Correcting Commits is an effective method of de-duplicating historical compiler bugs [14], which regards bug-triggering test programs with the same correcting commit (the first commit making a bug-triggering program pass) as duplicates. It has the advantage of not requiring fully-reduced programs, but cannot be used to de-duplicate new bugs, which can be relieved by D^3.

Besides, there are also many studies that target the problem of bug de-duplication based on the textual descriptions, such as crash messages and bug reports. For example, Alipour et al. [7] proposed to use BM25F [41] to extract the contextual features from bug reports for bug de-duplication. Sun et al. [48] proposed REP, which performs statistical analysis on the textual information in bug reports to measure the similarity between bug reports. After that, Sun et al. [49] further proposed a discriminative model for searching similar bug reports in bug tracking systems using information retrieval method. Similarly, Nguyen et al. [40] proposed a combination of information retrieval and topic modeling methods for characterizing the failures from bug reports. Furthermore, Zou et al. [59] proposed to combine LDA [26] and N-gram models to assist in the measurement of similarity between bug reports, while Lin et al. [36] proposed SVM-SBCTC that employs the SVM discriminative schema for text feature embedding. Besides, Wang et al. [50] proposed to combine the textual information in bug reports and the test execution information for bug de-duplication, while Lerch et al. [34] incorporated the stack traces of failures to promote bug de-duplication.

However, for silent compiler bug de-duplication, it is common that only bug-triggering test programs and bug-triggering optimizations [20, 28] are available, whereas stack traces and descriptive bug reports are usually unavailable. Hence, the aforementioned techniques cannot be applied to our target problem.

Compiler Test Prioritization. Like the existing work [20], we solve the problem of silent compiler bug de-duplication by prioritizing test failures (including bug-triggering test programs). Hence, our work is also related to compiler test prioritization to some degree. The compiler test prioritization techniques aim to accelerate the compiler testing process. For example, Chen et al. [12] proposed a text-vector based prioritization technique, which transforms each test program into a text-vector and then measure the distance of the vectors. Besides, Chen et al. [11] further proposed the idea of learning-to-test, which first learns a capability model from the history data for predicting the bug-revealing possibility of test programs as prioritization guidance. There are also many other test prioritization techniques that are not specific to compilers [35, 43, 44]. However, all these techniques are different from ours since their target is the testing efficiency while our target is bug de-deplication, which is the downstream task of the former.

8 CONCLUSION

Compiler debugging suffers from the bug de-duplication problem, i.e., many test failures are caused by the same compiler bug. Unlike crash bugs, which can be de-duplicated based on the produced error messages, silent compiler bugs produce little information for bug de-duplication. Hence, this problem becomes much more challenging. To solve the challenging problem, we propose a novel technique (called D^3) by characterizing three-dimensional information that is essential for a compiler bug. They are *test program*, *optimizations*, and *test execution*. To improve the de-duplication accuracy, D^3 systematically conducts causal analysis to identify bug-causal features from each dimension. Based on the three-dimensional bug-causal features, D^3 ranks the test failures that are more likely to be caused by different silent bugs higher by measuring their distance. Our experimental results on four datasets from GCC and LLVM demonstrate that D^3 significantly outperforms the two state-of-the-art compiler bug de-duplication techniques.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive suggestions to help improve the quality of this paper. The work has been supported by the National Natural Science Foundation of China under Grant Nos. 62002256, 62232001, 62202324 and Ministry of Education, Singapore under its Academic Research Fund Tier 3 (Award ID: MOET32020-0004).

REFERENCES

[1] 2020. Clang Static Analyzer. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[2] 2022. D3 homepage. https://github.com/chenyangyc/D3

[3] 2022. GCC. https://gcc.gnu.org

[4] 2022. LLVM. https://llvm.org

[5] 2022. tree-sitter. https://github.com/tree-sitter/tree-sitter

[6] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 89–98.

[7] Anahita Alipour, Abram Hindle, and Eleni Stroulia. 2013. A contextual approach towards more accurate duplicate bug report detection. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 183–192.

[8] Mohammad Amin Alipour, Alex Groce, Rahul Gopinath, and Arpit Christi. 2016. Generating focused random tests using directed swarm testing. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 70–81.

[9] Cyrille Artho. 2011. Iterative delta debugging. *International Journal on Software Tools for Technology Transfer* 13, 3 (2011), 223–246.

[10] Scotty Bauer, Cuoq Pascal, and Regehr John. [n. d.]. Deniable Backdoors Using Compiler Bugs. *International Journal of PoC||GTFO* ([n. d.]).

[11] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Bing Xie. 2017. Learning to prioritize test programs for compiler testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 700–711.

[12] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. Test case prioritization for compilers: A text-vector based approach. In *2016 IEEE international conference on software testing, verification and validation (ICST)*. IEEE, 266–277.

[13] Junjie Chen, Jiaqi Han, Peiyi Sun, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. Compiler bug isolation via effective witness test program generation. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 223–234.

[14] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. An empirical comparison of compiler testing techniques. In *Proceedings of the 38th International Conference on Software Engineering*. 180–190.

[15] Junjie Chen, Haoyang Ma, and Lingming Zhang. 2020. Enhanced compiler bug isolation via memoized search. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 78–89.

[16] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A survey of compiler testing. *ACM Computing Surveys (CSUR)* 53, 1 (2020), 1–36.

[17] Junjie Chen and Chenyao Suo. 2022. Boosting Compiler Testing via Compiler Optimization Exploration. *ACM Transactions on Software Engineering and Methodology* (2022).

[18] 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.

[19] M.Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. 2002. Pinpoint: problem determination in large, dynamic Internet services. In *Proceedings International Conference on Dependable Systems and Networks*. 595–604. https://doi.org/10.1109/DSN.2002.1029005

[20] 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.

[21] Zhichao Chen, Junjie Chen, Weijing Wang, Jianyi Zhou, Meng Wang, Xiang Chen, Shan Zhou, and Jianmin Wang. [n. d.]. Exploring Better Black-Box Test Case Prioritization via Log Analysis. *ACM Transactions on Software Engineering and Methodology* ([n. d.]).

[22] Alastair F Donaldson, Hugues Evrard, and Paul Thomson. 2020. Putting randomized compiler testing into production (experience report). In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[23] 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.

[24] 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.

[25] Teofilo F Gonzalez. 1985. Clustering to minimize the maximum intercluster distance. *Theoretical computer science* 38 (1985), 293–306.

[26] Dan Han, Chenlei Zhang, Xiaochao Fan, Abram Hindle, Kenny Wong, and Eleni Stroulia. 2012. Understanding android fragmentation with topic analysis of vendor-specific bugs. In *2012 19th Working Conference on Reverse Engineering*. IEEE, 83–92.

[27] Foyzul Hassan and Xiaoyin Wang. 2018. Hirebuild: An automatic approach to history-driven repair of build scripts. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 1078–1089.

[28] 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.

[29] Jiajun Jiang, Ran Wang, Yingfei Xiong, Xiangping Chen, and Lu Zhang. 2019. Combining spectrum-based fault localization and statistical debugging: An empirical study. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 502–514.

[30] James A Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. 273–282.

[31] Ajmeera Kiran and D Vasumathi. 2020. Data mining: min–max normalization based data perturbation technique for privacy preservation. In *Proceedings of the Third International Conference on Computational Intelligence and Informatics*. Springer, 723–734.

[32] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices* 49, 6 (2014), 216–226.

[33] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. 2020. Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer. In *29th USENIX Security Symposium (USENIX Security 20)*. 2613–2630.

[34] J. Lerch and M. Mezini. 2013. Finding Duplicates of Your Yet Unwritten Bug Report. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*.

[35] Zheng Li, Mark Harman, and Robert M Hierons. 2007. Search algorithms for regression test case prioritization. *IEEE Transactions on software engineering* 33, 4 (2007), 225–237.

[36] Meng-Jie Lin, Cheng-Zen Yang, Chao-Yuan Lee, and Chun-Chang Chen. 2016. Enhancements for duplication detection in bug reports with manifold correlation features. *Journal of Systems and Software* 121 (2016), 223–233.

[37] Fernando Lourenco, Victor Lobo, and Fernando Bacao. 2004. Binary-based similarity measures for categorical data and their application in Self-Organizing Maps. (2004).

[38] Ghassan Misherghi and Zhendong Su. 2006. HDD: hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering*. 142–151.

[39] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A Model for Spectra-Based Software Diagnosis. *ACM Trans. Softw. Eng. Methodol.* 20, 3 (2011), 32 pages. https://doi.org/10.1145/2000791.2000795

[40] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun. 2012. Duplicate bug report detection with a combination of information retrieval and topic modeling. (2012).

[41] 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.

[42] 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.

[43] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. 1999. Test case prioritization: An empirical study. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*. IEEE, 179–188.

[44] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 2001. Prioritizing test cases for regression testing. *IEEE Transactions on software engineering* 27, 10 (2001), 929–948.

[45] Hanan Samet. 1977. Toward automatic debugging of compilers. In *Proceedings of the 5th international joint conference on Artificial intelligence-Volume 1*. 379–379.

[46] 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.

[47] 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.

[48] 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.

[49] C. Sun, D. Lo, X. Wang, J. Jing, and S. C. 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, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*.

[50] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. 2008. An approach to detecting duplicate bug reports using natural language and execution information. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*.

[51] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *2018*

IEEE/ACM 40th International Conference on Software Engineering (ICSE). IEEE, 1–11.

[52] W Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. 2013. The DStar method for effective software fault localization. *IEEE Transactions on Reliability* 63, 1 (2013), 290–308.

[53] Robert F Woolson. 2007. Wilcoxon signed-rank test. *Wiley encyclopedia of clinical trials* (2007), 1–3.

[54] 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.

[55] Jerrold H Zar. 2005. Spearman rank correlation. *Encyclopedia of biostatistics* 7 (2005).

[56] Andreas Zeller. 1999. Yesterday, my program worked. Today, it does not. Why? *ACM SIGSOFT Software engineering notes* 24, 6 (1999), 253–267.

[57] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200.

[58] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal program enumeration for rigorous compiler testing. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 347–361.

[59] Jie Zou, Ling Xu, Mengning Yang, Xiaohong Zhang, Jun Zeng, and Sachio Hirokawa. 2016. Automated duplicate bug report detection using multi-factor analysis. *IEICE TRANSACTIONS on Information and Systems* 99, 7 (2016), 1762–1775.