Enhanced Compiler Bug Isolation via Memoized Search

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

Haoyang Ma*
College of Intelligence and
Computing, Tianjin University
China, Tianjin
haoyang_9804@tju.edu.cn

Lingming Zhang
University of Illinois at
Urbana-Champaign
USA, IL, Urbana
lingming@illinois.edu

ABSTRACT

Compiler bugs can be disastrous since they could affect all the software systems built on the buggy compilers. Meanwhile, diagnosing compiler bugs is extremely challenging since usually limited debugging information is available and a large number of compiler files can be suspicious. More specifically, when compiling a given bug-triggering test program, hundreds of compiler files are usually involved, and can all be treated as suspicious buggy files. To facilitate compiler debugging, in this paper we propose the first reinforcement compiler bug isolation approach via structural mutation, called RecBi. For a given bug-triggering test program, RecBi first augments traditional local mutation operators with structural ones to transform it into a set of passing test programs. Since not all the passing test programs can help isolate compiler bugs effectively, RecBi further leverages reinforcement learning to intelligently guide the process of passing test program generation. Then, RecBi ranks all the suspicious files by analyzing the compiler execution traces of the generated passing test programs and the given failing test program following the practice of compiler bug isolation. The experimental results on 120 real bugs from two most popular C open-source compilers, i.e., GCC and LLVM, show that RecBi is able to isolate about 23%/58%/78% bugs within Top-1/Top-5/Top-10 compiler files, and significantly outperforms the state-of-the-art compiler bug isolation approach by improving 92.86%/55.56%/25.68% isolation effectiveness in terms of Top-1/Top-5/Top-10 results.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Compilers**; • **Theory of computation** → *Reinforcement learning*.

KEYWORDS

Compiler Bug Isolation, Fault Localization, Reinforcement Learning

ACM Reference Format:
Junjie Chen, Haoyang Ma, and Lingming Zhang. 2020. Enhanced Compiler Bug Isolation via Memoized Search. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416570

1 INTRODUCTION

Compilers are one of the most fundamental software systems since almost all software systems (ranging from operating systems, web browsers, to script code written by end-users) are compiled by them. Although dedicated efforts have been devoted to ensuring their quality, compilers are still error-prone due to their extremely large-scale and complicated codebases [17, 18, 21, 58, 70]. In practice, compiler bugs are very harmful, and can potentially affect all the software systems compiled by the buggy compilers. Therefore, it is essential to detect, isolate, and fix all possible compiler bugs.

In the literature, many approaches have been proposed to automatically detect compiler bugs [12–14, 18, 19, 28, 58, 66, 70, 75], but there is limited research efforts dedicated to automated debugging of compiler bugs, such as bug isolation and fixing. That is, compiler bug isolation and fixing are still a rather tedious and time-consuming process for modern compilers. In particular, compiler bug isolation is a more fundamental problem since it also directly helps with effective compiler bug fixing. Although many automated bug localization approaches (such as spectrum-based [7, 27, 73], slicing-based [69], mutation-based [34, 45, 48, 51, 74], and the recent program-repair-based [9, 46] approaches) have been proposed for common software systems, these existing approaches can hardly isolate compiler bugs due to either extremely high costs or poor effectiveness; please refer to the extensive discussion in a recent work [16] for more details.

To facilitate compiler bug isolation, Chen et al. [16] proposed the first approach, named DiWi, which transforms the problem of compiler bug isolation to the problem of passing test program generation. More specifically, given a failing test program, DiWi first generates a set of passing test programs by traditional *local* mutation operators (which change minimal program elements such as modifiers and constants), and then leverages existing bug localization techniques [6, 36] to identify the compiler buggy files by comparing the execution traces between the generated passing programs and the given failing test program. Although the generated passing test programs via DiWi has been demonstrated to perform better than both developer-written test programs and the test programs generated by the widely-used compiler fuzzing technique (i.e., Csmith [70]) for compiler bug isolation [16], DiWi still suffers from the effectiveness issue. For example, as demonstrated by the

*Both authors contributed equally to this paper.
[†]Junjie Chen is the corresponding author.

ASE '20, September 21–25, 2020, Virtual Event, Australia
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416570

existing work [16] and our study (to be presented in Section 4.5), developers using DiWi still need to check about 15 innocent files before finding the really buggy one on average; about 62.5% studied bugs cannot be successfully isolated after checking 5 most suspicious files recommended by DiWi (please note that in practice, most developers tend to abort the automated debugging tools if they cannot localize buggy elements within Top-5 positions [38]).

To further advance state-of-the-art compiler bug isolation, in this paper, we propose an enhanced compiler bug isolation approach via memoized search and structural mutation, called **RecBi** (**Re**inforcement **c**ompiler **B**ug **i**solation). More specifically, since compiler bugs tend to occur in the components of compiler optimizations that tend to depend on test program structure, for a given compiler bug with a failing test program, RecBi first augments the traditional *local* mutation operators used by DiWi with *structural* mutation operators (which change the test program structure by inserting some control-flow-alerting statements such as branch and loop statements) to effectively generate similar test programs that can flip the compiler execution results (i.e., from `failing` to `passing`). This is because traditional local mutation operators usually have small influence on program structure due to its minor modification, while structural mutation operators can augment the ability of changing program structure by effectively altering the control-flow of test programs and in the meanwhile optimizations are often involved in compiler bugs and structural mutation is good at skipping the buggy optimizations. However, not all the generated passing test programs can facilitate isolating compiler bugs [16], and thus casually or simply heuristically performing mutations on the given failing test program may not be effective. Thus, RecBi further incorporates reinforcement learning [37] (a kind of memoized search), which can effectively learn both historical and future knowledge, to intelligently guide how to conduct mutation in order to generate a set of more effective passing test programs during a given period. Finally, similar to the existing work [7, 16], RecBi ranks all the suspicious files according to their suspicious scores by comparing the compiler execution traces between the generated passing test programs and the given failing program. In a word, the novelties of RecBi are twofold: 1) *it opens a new dimension for compiler bug isolation via structural mutation*; 2) *it leverages reinforcement learning for more intelligent compiler bug isolation.*

To evaluate the effectiveness of RecBi, we conducted an extensive study based on 120 real compiler bugs from GCC [2] and LLVM [4], which are the most widely-used C compilers in both industry and academia [16, 41, 58, 70]. Our experimental results show that RecBi is able to isolate 27, 70, 93, 107 bugs (out of 120 compiler bugs) within Top-1, Top-5, Top-10, and Top-20 files, respectively. That is, about 23%, 58%, 78%, and 89% bugs can be isolated successfully within Top-1, Top-5, Top-10, and Top-20 files through RecBi, respectively. In particular, RecBi substantially outperforms the state-of-the-art approach DiWi. For example, the improvements of RecBi over DiWi are up to 92.86%/55.56% in terms of Top-1/Top-5 results, 45.55% in terms of MFR (Mean First Rank, measuring the effectiveness in detecting the first buggy file for each bug), and 44.62% in terms of MAR (Mean Average Rank, measuring the effectiveness in detecting all the buggy files for each bug). Furthermore, we investigated the contributions of both major components in RecBi (i.e., structural mutation and reinforcement learning for passing test program generation), as well as the impacts of different RecBi configurations. To sum up, this paper makes the following main contributions:

```
int printf(const char* ,...);
int a,b=1;
int main(){
  int i;
  for(i=0;i<56;i++)
    for(;a;a   )
      ;
  int *c=&b;
  if(*c)
    *c=1%(unsigned int)*c|5;
  printf("%d\n",b);
  return 0;
}
```

(a) Failing Program

```
int printf(const char* ,...);
int a,b=1;
int main(){
  int i;
  for(i=0;i<56;i++)
    for(;a;a   )
      ;
  int *c=&b;
  while(a--)
    if(*c)
      *c=1%(unsigned int)*c|5;
  printf("%d\n",b);
  return 0;
}
```

(b) Passing Program

Figure 1: GCC Bug 64682

- This work opens a new dimension of compiler bug isolation via structural mutation, i.e., leveraging carefully designed structural mutation operators for generating passing test programs to boost compiler bug isolation.
- This work brings reinforcement learning to the compiler bug isolation area for the first time, i.e., leveraging state-of-the-art reinforcement learning to intelligently guide the structural-mutation-based compiler bug isolation process.
- The proposed technique has been implemented as a practical compiler bug isolation system, named RecBi, based on mature tools and libraries, i.e., Clang Libtooling library [1], Gcov [3], and PyTorch [5].
- This work conducts an extensive study based on 120 real compiler bugs from two most widely-used C compilers, i.e., GCC and LLVM, to evaluate the effectiveness of RecBi. The results reveal the effectiveness of RecBi (significantly outperforming the state-of-the-art DiWi), the contribution of each major component in RecBi, and the impacts of different RecBi configurations.

2 BACKGROUND

2.1 Test Program Mutation for Compiler Bug Isolation

To solve the problem of compiler bug isolation, Chen et al. [16] transforms this problem to the problem of passing test program generation. According to the idea of spectrum-based bug localization (also called SBFL) [7, 67], all the compiler files touched by a given failing test program during compilation are suspects and passing test programs are helpful to reduce the suspicion of innocent files. If a passing test program has similar execution trace (except the buggy files) with the given failing test program, the buggy files are more likely to be isolated by comparing the execution trace between the passing test program and the given failing test program. Therefore, DiWi designs three categories of local mutation operators to produce such similar passing test programs by changing three minimal program elements (i.e., variables, constants, and operators) of the given failing test program.

Although these traditional local mutation operators in DiWi can generate some passing test programs as demonstrated by the existing work [16], their effectiveness actually is restricted. This is because compiler bugs tend to occur in the components of compiler optimizations, which tend to depend on test program structure; on the contrary, traditional local mutation operators usually have small influence on program structure due to their minor modification and thus could omit many effective passing test programs. Therefore, incorporating novel mutation operators that can effectively altering program structure is necessary for compiler bug isolation. In this work, we introduce the notion of *structural mutation*, which augments the program-structure-altering ability by inserting some control-flow-alerting statements (such as branch and loop statements).

We use Figure 1 to illustrate the effectiveness of structural mutation. Figure 1a shows a failing test program, which triggers a GCC bug (ID: 64682) when using `-O2` and above of GCC revision 219832 to compile it. This bug lies in the file `combine.c` and the root cause is that an `isns` (the RTL representation of the code for a function in GCC) sets a wrong note for a pseudo register after the correct note has been distributed. When using traditional local mutation in DiWi to isolate this bug, during the given period (i.e., one hour), 79 passing programs are generated and the buggy file is ranked at the 30^{th} position. However, when introducing structural mutation, there are 20 passing programs generated by structural mutation. Figure 1b shows one of the 20 passing programs. By inserting the statement `while(a--)` in Line 9, the buggy optimization can be effectively avoided, leading to passing execution. With the help of structural mutation, the buggy file is ranked at the 7^{th} position.

2.2 Reinforcement Learning

To more efficiently generate effective passing test programs, we aim to incorporate reinforcement learning to guide the process of passing test program generation, and thus we briefly introduce some background of reinforcement learning.

Reinforcement learning aims to learn how an agent should take actions in an environment in order to maximize cumulative reward in a long run [37, 61]. An agent has many states and actions, and during the learning process, it performs an action at a state, then measures the reward obtained by this action, and moves to the next state. Through such a process, the agent gradually learns to select a better action at the next state in order to obtain more reward. In general, reinforcement learning can be divided into two categories: value-based algorithms (e.g., Deep Q Learning algorithm [47]) and policy-based algorithms (e.g., Policy Gradients algorithm [60]). The former is a deterministic strategy that approximates the optimal value function to select the best action at each state and tends to be efficient and steady, while the latter is a probabilistic strategy that learns the probability distribution (i.e., policy) of actions to obtain the most reward and tends to fit continuous and stochastic environments and have faster convergence.

With the development of reinforcement learning, the algorithms merging both value-based and policy-based strategies, called Actor-Critic algorithms (AC), are proposed [62]. In AC, the actor controls how the agent behaves by learning the best policy via an actor neural network (ANN), while the critic predicts the reward achieved by the action by calculating the value function via a critic neural network (CNN). Subsequently, the improved versions of AC are proposed, i.e., Advantage Actor-Critic (A2C) [63] and Asynchronous Advantage Actor-Critic (A3C) [63]. Instead of the value function, A2C learns the advantage value function by evaluating both how good the action is and how much better it can be (which incorporates future knowledge), which can reduce high variances of neural networks. A3C further incorporates the asynchronous mechanism to improve the learning efficiency. In this work, we utilize the framework of A2C to solve the problem of compiler bug isolation, since it is both effective (compared with AC) and suitable to single-thread and multi-thread systems (compared with A3C).

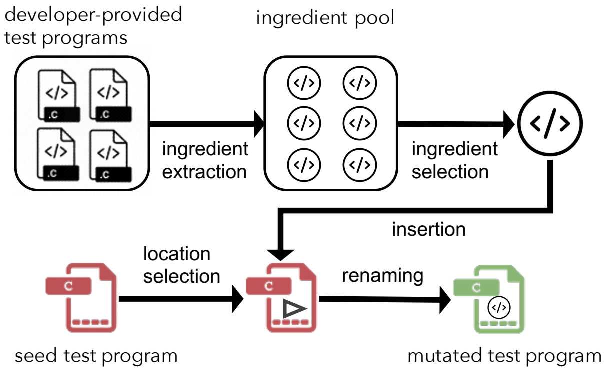


Figure 2: Overview of Structural Mutation

3 APPROACH

In this section, we present our reinforcement compiler bug isolation approach via structural mutation, named **RecBi**. Given a compiler bug with a failing test program, RecBi first generates a set of effective passing test programs by mutating the given failing test program through a reinforcement learning based mutation process. Second, based on the given failing test program and the set of generated passing test programs, RecBi leverages off-the-shelf SBFL to identify the compiler buggy files. The main contribution of RecBi lies in the first step. During the process of passing test program generation, besides changing minimal program elements via traditional local mutation, RecBi further incorporates structural mutation to change the structure of the given failing test program, which could enlarge the mutation space to include more effective passing test programs. However, not all the passing test programs can facilitate to isolate compiler bugs as demonstrated by the existing work [16], and thus casually conducting mutation on the given failing test program may not be effective. Therefore, RecBi further incorporates reinforcement learning (which could effectively learn both historical and future knowledge) to learn how to intelligently conduct mutation on the given failing test program so that a set of more effective passing test programs can be generated.

In the following, we introduce our structural mutation in Section 3.1 and present our reinforcement-learning based test program generation in Section 3.2. Although we do not propose any new SBFL formula to calculate the suspicious score of each compiler file in the second step of RecBi, we still briefly introduce the application of SBFL in RecBi in Section 3.3 to make the paper self-contained.

```c
volatile
int a,b,c=2;
unsigned d;
int main(){
  int e= 32;
  d= 31;
  for(;d>2;d++)
    for(e++;;b    )
      if(c)
        break;
  e&&a;
  return 0;
}
```

(a) Failing

```c
volatile
int a,b,c=2;
unsigned d;
int main(){
  int e= 32;
  ▹d= 31;
  ▹for(;d>2;d++)
    ▹for(e++;;b    )
      ▹if(c)
        break;
  ▹e&&a;
  return 0;
}
```

(b) Locations

```c
volatile
int a,b,c=2;
unsigned d;
int main(){
  int e= 32;
  if(((long)(a-a))<1)
  d= 31;
  for(;d>2;d++)
    for(e++;;b    )
      if(c)
        break;
  e&&a;
  return 0;
}
```

(c) Mutant

Figure 3: Example of Structural Mutation

Table 1: Summary of mutation operators in RecBi

ID	Description
1	Insert a branch (i.e., `if`) statement;
2	Insert a loop (i.e., `while`) statement;
3	Insert a function call;
4	Insert a `goto` statement;
5	Insert/remove a qualifier (i.e., `volatile`, `const`, and `restrict`);
6	Insert/remove a modifier (i.e., `long`, `short`, `signed`, and `unsigned`);
7	Replace a variable with another valid one
8	Replace a constant with another valid one;
9	Replace/remove an unary operator;
10	Replace a binary operator with another valid one.

3.1 Structural Mutation

The goal of mutation is to flip the compiler execution result (from `failing` to `passing`) by transforming a given failing test program. As presented in prior work [16, 18, 19, 58], most of compiler bugs occur in the components of compiler optimizations, while the triggering of compiler optimizations tends to depend on the structure of test programs. Therefore, transforming the structure of a given failing test program is helpful to generate effective passing test programs. However, the existing local mutation operators in DiWi usually have small influence on program structure due to its minor modification, and thus we further explore structural mutation in RecBi. More specifically, we design four structural mutation operators, which insert four different types of statements to a given failing test program respectively, to change the control-flow of the failing test program. The four types of inserted statements are 1) **branch statements**, 2) **loop statements**, 3) **function calls**, and 4) **`goto` statements**, since they are recognized to be effective to change the control-flow of a program [23, 24, 42].

Except `goto` statements, the other three types of statements require additional ingredients (i.e., conditions in a branch or loop statement, as well as the called function and its parameters in a function call) to complete insertion. However, it could be inefficient to casually construct these ingredients. As demonstrated by the existing work [16], although the state-of-the-art compiler bug isolation approach DiWi outperforms the approach using the developer-provided test programs to isolate compiler bugs, the latter is able to perform no worse than the former in some cases. Therefore, it may be promising to adapt the ingredients already within the developer-provided test programs for our structural mutation. In this way, the unique value of the developer-provided test programs embodied in the existing work [16] can be incorporated by RecBi.

Figure 2 shows the overview of our structural mutation, which consists of three steps. First, RecBi extracts all the branch conditions, loop conditions, declared functions and the corresponding function calls, in the developer-provided test programs for the compiler under test, as an ingredient pool. Second, RecBi randomly selects an ingredient from the ingredient pool according to the type of the statement to be inserted, and randomly selects an insertable location in the seed test program. It would produce invalid test programs or fake passing test programs [16] (i.e., the generated passing test programs are not really passing and just remove the test oracles) when inserting a statement to an improper location. There are three types of non-insertable locations in RecBi: 1) the locations outside functions, 2) the locations before declarations for the sake of maintaining the identifier scope, and 3) the locations before the statements used as test oracles (such as `printf`/`__builtin_abort`/`return` statements). Third, RecBi performs insertion, and then conducts refactoring for new variables in the selected ingredient, i.e., renaming the new variable to those within the seed test program with compatible types, to make the mutated test program valid. Figure 3 shows an illustrative example for structural mutation, where Figure 3a is a failing test program, Figure 3b identifies all the insertable locations (denoted as ▹) in the failing test program, and Figure 3c is a generated passing test program via our structural mutation (by inserting a branch statement).

Local Mutation Operators. Besides these structural mutation operators, RecBi also incorporates the traditional local mutation operators targeting the minimal program elements, which have been studied by the existing work for compiler bug isolation [16]. The reason is that 1) the generated test programs via these local mutation operators have been demonstrated to outperform the developer-provided test programs and the test programs generated via the widely-used compiler fuzzing technique (i.e., Csmith [70]) [16], and 2) for the compiler bugs in the front-end component (although the number of this type of compiler bugs is rare), local mutation could be very useful. Therefore, RecBi has 10 mutation operators in total, which are summarized in Table 1.

Test Oracles. After mutation, it is also required to check whether the generated test program is passing or still failing [15, 16]. According to the types of compiler bugs (i.e., crash bugs and wrong-code bugs) [16, 20, 58], RecBi considers two types of test oracles accordingly. Regarding crash bugs (i.e., the compiler crashes when using some compilation options to compile a test program), the used test oracle is whether the compiler still crashes when using the same compilation options to compile a generated test program. Regarding wrong-code bugs (i.e., the compiler mis-compiles a test program without any failure messages, causing the test program to have inconsistent execution result under different compilation options), the used test oracle is whether a generated test program still produces inconsistent execution results under the compilation options producing inconsistencies before.

3.2 Test Program Generation via Reinforcement Learning

Since not all the passing test programs are helpful to isolate compiler bugs as demonstrated by the existing work [16], it could be ineffective to randomly perform mutation on a given failing test program. In particular, different compiler bugs have different characteristics and root causes, and thus the effects of these mutation operators on different compiler bugs can be different. Therefore, it is necessary for each specific compiler bug to learn the effect of each mutation operator in order to generate more effective passing test programs efficiently during the given time period.

As presented in Section 2.2, reinforcement learning is a well-recognized strategy to guide an agent to behave better in an environment so as to obtain the most reward [37], which highly matches our problem, i.e., learning to generate more effective passing test programs. Therefore, to achieve our goal, we leverage the power of reinforcement learning in RecBi, where reward refer to the quality of generated passing test programs (to be explained in Section 3.2.1). More specifically, an agent has many states and actions, and during the iterative process of reinforcement learning, the agent learns more and better by conducting an action at a state and then measuring the reward of this action. In RecBi, a *state* refers to a state of the set of mutation operators, i.e., the number of times that each mutation operator has been selected to generate passing test programs, while an *action* refers to selecting and then applying a mutation operator to a given failing test program.

Here, RecBi adopts the framework of A2C [63] to learn the effects of these mutation operators in order to guide the generation of effective passing test programs for a given compiler bug. This is because it has been demonstrated to be effective and efficient in practice and perform stably with low variance [29, 56, 57]. Moreover, A2C converges faster than the traditional AC algorithm. Figure 4 illustrates the overview of the reinforcement learning based strategy in RecBi. Following the framework of A2C, RecBi constructs two neural networks, i.e., ANN (Actor Neural Network) and CNN (Critic Neural Network). ANN aims to predict the probability distribution of actions based on *historical knowledge* and then choose an action to be performed, while CNN aims to predict the potential reward to be accumulated from the current state to a future state after performing the selected action, which incorporates *future knowledge*. Based on the predicted potential reward and the actual reward obtained by performing the selected action, RecBi adopts an advantage loss function (to be introduced in Section 3.2.2) to update both ANN and CNN in order to make them learn more and better. In particular, following the practice of A2C [62, 63], both ANN and CNN in RecBi contain only one hidden layer in order to be light-weight and converge fast. The process is repeated until the terminating condition is reached, and its output is a set of generated effective passing test programs. In the following, we introduce the actual reward measurement (Section 3.2.1) and the advantage loss function (Section 3.2.2) in detail.

3.2.1 *Actual Reward Measurement.*

An important aspect to the success of the A2C based approach is how to measure the actual reward after applying a mutation operator to the given failing test program. Inspired by the existing work [16], a set of effective passing test programs should satisfy both similarity and diversity criteria. The former refers to that each passing test program should share a similar compiler execution trace with the given failing test program. In this way, according to the idea of SBFL, the suspiciousness of more buggy-free files can be reduced. The latter refers to that different passing test programs should have diverse compiler execution traces between each other in order to reduce the suspiciousness of different buggy-free files. In this way, aggregating a set of passing mutated programs is helpful to effectively isolate the really buggy files by avoiding bias. Given a set of generated passing test programs denoted as $P = \text{f}p_1, p_2, \ldots, p_n\text{g}$ and the failing test program denoted as f, we define the similarity and diversity metrics achieved by the set of passing test programs on average as shown in Formulae 1 and 2 respectively:

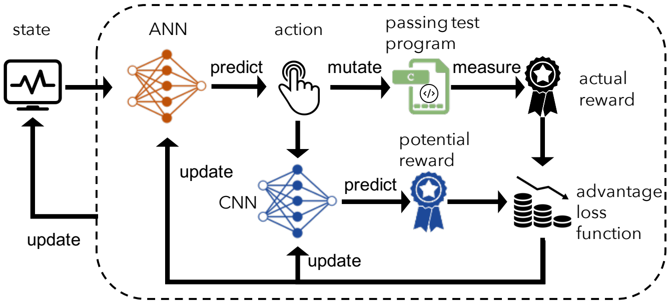


Figure 4: Overview of Our Reinforcement Learning based Test Program Generation Strategy

$$sim = \frac{\text{Í}_{i=1}^{n} {}^{1}1 \quad dist^{1}p_i, f^{\text{ºº}}}{n} \tag{1}$$

$$div = \frac{\text{Í}_{i=1}^{n \quad 1} \text{Í}_{j=i \text{¸} 1}^{n} dist^{1}p_i, p_j{}^{\text{º}}}{\frac{n^{1}n \quad 1^{\text{º}}}{2}} \tag{2}$$

$$dist^{1}a, b^{\text{º}} = 1 \quad \frac{Cov_a \setminus Cov_b}{Cov_a \ [\ Cov_b} \tag{3}$$

where $dist^{1}, {}^{\text{º}}$ is the coverage distance between two test programs and is measured by Jaccard Distance; Cov_a and Cov_b represent the set of statements in the compiler covered by test programs a and b, respectively; n is the number of generated passing test programs.

At a time step denoted as t, after generating a passing test program, RecBi measures the quality of the current set of passing test programs by linearly combining the achieved similarity and diversity as shown in Formula 4. Then, RecBi determines whether or not to accept the generated passing test program according to whether it can improve the quality of the passing program set compared with the last time step denoted as t-1. Formula 5 presents the calculation of the improved quality compared with the last time step.

$$Q_t = n^{1}\alpha \quad div_t \ \text{¸} \ {}^{1}1 \quad \alpha^{\text{º}} \quad sim_t{}^{\text{º}} \tag{4}$$

$$\begin{aligned} \Delta Q_t &= Q_t \quad Q_{t \quad 1} \\ &= {}^{1}n \quad 1^{\text{º}1}\alpha \quad \Delta div \ \text{¸} \ {}^{1}1 \quad \alpha^{\text{º}} \quad \Delta sim^{\text{º}} \\ &\text{¸} \ {}^{1}\alpha \quad div_t \ \text{¸} \ {}^{1}1 \quad \alpha^{\text{º}} \quad sim_t{}^{\text{º}}, \\ \Delta div_t &= div_t \quad div_{t \quad 1}, \Delta sim_t = sim_t \quad sim_{t \quad 1} \end{aligned} \tag{5}$$

where α is the coefficient for the linear combination between diversity and similarity. Formula 4 also has a coefficient n (the size of the current passing test program set) due to the following intuition – when the size of the passing test program set is small, it is preferable for RecBi to accept a new passing test program even though it may decrease the diversity and similarity (because when n is smaller, the actual delta is less important); but with the size of the passing test program set increasing, we have less interests to generate such low-quality passing test programs with RecBi. Therefore, we incorporate n to reflect such intuition in RecBi.

However, at each state only one mutation operator is selected to generate a passing test program, and a mutation operator could perform extremely differently due to various mutated locations, which could lead to slow convergence for A2C. Moreover, it means that the improved quality of the passing program set in the current time step cannot precisely reflect the effect of the selected mutation operator. Therefore, to reduce the influence of various performance of a mutation operator, RecBi combines the improved quality at the current time step and the historically improved quality by the current mutation operator as the actual reward obtained at the current time step, instead of directly using the improved quality:

$$Reward_t = \frac{\sum_{i=1}^{t} \Delta Q_i}{T(m_i)} \tag{6}$$

where, ΔQ_i=0 if the selected mutation operator is not m_i at the i^{th} time step, otherwise ΔQ_i is calculated by Formula 5, and $T(m_i)$ refers to the number of times that m_i has been selected to mutate the given failing test program.

3.2.2 Advantage Loss Function. After obtaining the actual reward at the current time step, RecBi further uses CNN to obtain the predicted potential reward. To better take the future factors into account, A2C designs an advantage loss function in order to reduce the high variance of the two neural networks and avoid falling into the local optimal [63], which is shown in Formula 7:

$$A(t) = \sum_{i=t}^{t+u} (\gamma^{(i-t)} Reward_i) + \gamma^u PR_{t+u} - PR_t \tag{7}$$

where, u represents that CNN considers the future u consecutive states and actions when predicting the potential reward, γ is the weight of the actual future reward, PR_{t+u} and PR_t are the predicted potential rewards at the $(t+u)^{th}$ and t^{th} time steps by CNN respectively. In particular, RecBi repeats the process in a time step for u times and get the approximation of the actual future reward.

Based on the loss calculated by the advantage function in Formula 7, RecBi updates the weights of both ANN and CNN according to Formula 8.

$$\omega = \omega + \beta \frac{\partial(logP_\omega(a_t|s_t)A(t))}{\partial\omega} \tag{8}$$

where, s_t and a_t are the current state and action, $P_\omega(a_t|s_t)$ refers to the probability that a_t is performed at s_t based on the parameters ω in ANN and CNN, β is the learning rate.

3.3 Compiler Buggy File Identification

Based on the set of generated passing test programs and the given failing test program, RecBi leverages the idea of SBFL to identify the buggy compiler files via comparing the coverage of failing and passing tests [16]. More specifically, following prior work on compiler bug isolation [16], RecBi first adopts state-of-the-art SBFL formula, i.e., Ochiai [7] as shown in Formula 9, to calculate the suspicious score of each statement:

$$score(s) = \frac{ef_s}{\sqrt{(ef_s + nf_s)(ef_s + ep_s)}} \tag{9}$$

where ef_s and nf_s represent the number of failing tests that execute and do not execute statement s, and ep_s represents the number of passing tests that execute statement s. Since in RecBi there is only one given failing test program, ef_s is 1. Moreover, RecBi only considers the statements executed by the given failing test program, and thus nf_s is 0. Therefore, in RecBi the Ochiai formula can be simplified as:

$$score(s) = \frac{1}{\sqrt{1 + ep_s}} \tag{10}$$

After obtaining the suspicious score of each statement, RecBi further calculates the suspicious score of each compiler file. Following prior work [16], RecBi aggregates the suspicious scores of the statements executed by the given failing test program in a compiler file as the suspicious score of the compiler file:

$$SCORE(f) = \frac{\sum_{i=1}^{n_f} score(s_i)}{n_f} \tag{11}$$

where n_f is the number of statements executed by the failing test program in the compiler file f. According to the descending order of the suspicious score of each compiler file, RecBi produces a ranking list of compiler files, where the higher a compiler file is ranked, the higher possibility the file has to be buggy.

4 EVALUATION

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

RQ1: How does RecBi perform on compiler bug isolation?
RQ2: How does each main component contribute to RecBi?
RQ3: How does different RecBi configurations impact the effectiveness of RecBi?

4.1 Compilers and Bugs

In the study, we used both GCC and LLVM as subjects to investigate the effectiveness of RecBi, covering almost all popular open-source C compilers used in the existing work [13, 16, 21, 41, 70]. Regarding the subject bugs, we used the released benchmark, including 120 real compiler bugs (60 GCC bugs and 60 LLVM bugs), including all bugs from prior compiler bug isolation work [16]. Each compiler bug contains the following information: the buggy compiler version, the failing test program, the compilation options to reproduce the bug, and the buggy files (served as the ground-truth in our study). On average, a GCC buggy version has 1,758 files with 1,447K source lines of code (SLOC), while a LLVM buggy version has 3,265 files with 1,723K SLOC.

4.2 Implementation and Parameters

We implemented our proposed approach RecBi based on Clang Libtooling library [1], Gcov [3], and PyTorch [5]. They are used to parse a test program to an AST (Abstract Syntax Tree), collect compiler coverage information, and provide the framework of A2C, respectively. Following the default setting in the existing work [8, 63], we also set γ and β in A2C to be 0.9 and 0.01, respectively. In

RecBi, the default settings of α and u are 0.8 and 5, respectively. Note that we investigated the impacts of such main parameters on RecBi in RQ3. Following the existing work [16], we set the terminating condition to be one hour limit. That is, we compared all the studied compiler bug isolation approaches under the same time limit for fair comparison. To reduce the influence of randomness, we repeatedly ran all the approaches for 5 times, and calculated the median results. Our study is conducted on a workstation with 32-core CPU, 120G memory and Ubuntu 14.04 operating system. We have released our tool and experimental data at our project homepage: **https://github.com/haoyang9804/RecBi**.

4.3 Independent Variables

4.3.1 Compared Approaches. We compared RecBi with the state-of-the-art compiler bug isolation approach **DiWi** [16] to answer RQ1. DiWi isolates compiler bugs via local mutation and the traditional MH (Metropolis-Hasting) algorithm [25], which depends on the most recent behavior of each mutation operator to determine the next mutation operator. Moreover, in traditional SBFL, developer-provided tests are always used as the passing tests to reduce the suspicion of innocent program elements. Thus, in RQ1 we also investigated whether the generated passing programs via RecBi outperform the developer-provided passing test programs for the compiler under test. We call the approach using the latter **Dev**, which uses the same strategy to rank all the compiler files as RecBi (presented in Section 3.3) but uses the developer-provided passing programs instead of the generated passing programs via RecBi.

In RQ2, we investigated the contributions of two main components in RecBi, including newly designed structural mutation and the reinforcement learning based test program generation strategy. Therefore, we designed the following variants of RecBi.

RecBi$_{mh}$ replaces the reinforcement learning based test program generation strategy with the traditional MH algorithm used in DiWi. That is, RecBi$_{mh}$ adopts the same strategy to guide the process of test program generation as DiWi.

RecBi$_{rand}$ removes the reinforcement learning based test program generation strategy from RecBi. That is, RecBi$_{rand}$ does not have any guidance to generate test programs by randomly selecting a mutation operator in each time step.

RecBi$_{filter}$ removes the reinforcement learning based test program generation strategy from RecBi, but keeps the part of measuring the quality of a generated passing test program since the measurement method is the base of the reinforcement learning based test program generation strategy. That is, RecBi$_{filter}$ randomly selects a mutation operator in each time step, then measures the quality of a generated passing test program in the same way as RecBi, and finally filters the low-quality passing test program ($\Delta Q_t < 0$). Actually, RecBi$_{filter}$ is an updated version of RecBi$_{rand}$ by adding a measuring component.

We compared RecBi$_{mh}$ and DiWi to investigate the contribution of our designed structural mutation operators. We then compared RecBi, RecBi$_{rand}$, and RecBi$_{mh}$ to investigate the contribution of our proposed reinforcement learning based test program generation strategy. Besides, we compared RecBi$_{rand}$ and RecBi$_{filter}$ to investigate the effectiveness of our designed measurement for the quality of a generated passing test program, which is the base of our reinforcement learning based test program generation strategy.

4.3.2 Different RecBi Configurations. In RQ3, we investigated different configurations of RecBi. Here, we discussed two main parameters in RecBi, including α (used to combine similarity and diversity as shown in Formula 4) and u (the number of future time steps that RecBi takes into account in Formula 7). Regarding α, we studied α = 0, 0.2, 0.4, 0.6, 0.8, and 1, respectively. Here, $\alpha = 0$ means that RecBi only considers similarity, while $\alpha = 1$ means that RecBi only considers diversity. Regarding u, we studied u = 1, 2, 3, 4, 5, 6, and 7, respectively.

4.4 Measurements

Each compiler bug isolation approach produces a ranking list of suspicious compiler files, and thus we measured the position of each buggy file in the ranking list to measure the effectiveness of each approach. Regarding the tie issue (i.e., multiple compiler files have the same suspicious scores), we adopted the worst ranking following the existing work [35, 52]. More specifically, we calculated the following metrics, which are widely-used by the existing work in the area of bug localization [16, 40, 48, 55].

Top-n measures the number of bugs that are isolated successfully within the Top-n position (i.e., $n \in \{1, 5, 10, 20\}$ in our study) in the ranking list. The larger the Top-n value is, the more effective the approach is.

Mean First Ranking (MFR) measures the mean of the rank of the first buggy file in the ranking list for each bug. MFR focuses on isolating the first buggy element fast in order to facilitate debugging. The smaller the MFR value is, the more effective the approach is.

Mean Average Ranking (MAR) measures the mean of the average rank of all buggy files in the ranking list for each bug. MAR focuses on isolating all buggy elements precisely. The smaller the MAR value is, the more effective the approach is.

4.5 Results and Analysis

4.5.1 RQ1: Overall effectiveness of RecBi. We illustrated the comparison results among various approaches in Table 2. Overall, RecBi is able to isolate 27, 70, 93, 107 compiler bugs (out of 120 compiler bugs) within Top-1, Top-5, Top-10, and Top-20 files, respectively. That is, nearly 23%, 58%, 78%, and 89% bugs can be isolated successfully within Top-1, Top-5, Top-10, and Top-20 files through RecBi, respectively. We further analyzed the effectiveness of RecBi on different subject compilers, and surprisingly found that although there are a larger number of compiler files in LLVM compared with GCC, RecBi achieves better results on LLVM than GCC. For example, the MFR and MAR values of RecBi on LLVM are 7.77 and 7.85 respectively while those of RecBi on GCC are 8.75 and 9.35 respectively. Moreover, we found that the other approaches indeed perform worse on LLVM than GCC. The results demonstrate that, the effectiveness of RecBi is not affected when facing larger compiler systems, indicating its scalability.

We then compared RecBi with the state-of-the-art compiler bug isolation approach DiWi. From Table 2, RecBi performs better than DiWi in terms of all the metrics and on both of subject compilers.

Table 2: Compiler bug isolation effectiveness comparison

Sub	Approach	Top-1	*Top 1	Top-5	*Top 5	Top-10	*Top 10	Top-20	*Top 20	MFR	*MFR	MAR	*MAR
LLVM	RecBi	13	—	38	—	48	—	54	—	7.77	—	7.85	—
	DiWi	6	116.67	23	65.22	37	29.73	47	14.89	16.80	53.75	16.92	53.61
	Dev	2	550.00	12	216.67	22	118.18	37	45.95	37.36	79.20	37.49	79.06
	RecBi$_{mh}$	10	30.00	31	22.58	42	14.29	50	8.00	11.17	30.44	11.48	31.62
	RecBi$_{filter}$	7	85.71	27	40.74	42	14.29	49	10.20	13.77	43.57	17.91	56.17
	RecBi$_{rand}$	3	333.33	29	31.03	39	23.08	49	10.20	40.12	80.63	40.16	80.45
GCC	RecBi	14	—	32	—	45	—	53	—	8.75	—	9.35	—
	DiWi	8	75.00	22	45.45	37	21.62	49	8.16	13.53	35.33	14.15	33.92
	Dev	3	366.67	12	166.67	25	80.00	32	65.62	22.44	61.01	23.04	59.42
	RecBi$_{mh}$	13	7.69	30	6.67	41	9.76	49	8.16	10.52	16.83	10.92	14.38
	RecBi$_{filter}$	14	0.00	30	6.67	43	4.65	50	6.00	10.10	13.37	10.30	9.22
	RecBi$_{rand}$	4	250.00	18	77.78	26	73.08	39	35.90	19.40	54.90	19.99	53.23
ALL	RecBi	27	—	70	—	93	—	107	—	8.26	—	8.60	—
	DiWi	14	92.86	45	55.56	74	25.68	96	11.46	15.17	45.55	15.53	44.62
	Dev	5	440.00	24	191.67	47	97.87	69	55.07	29.90	72.38	30.26	71.58
	RecBi$_{mh}$	23	17.39	61	14.75	83	12.05	99	8.08	10.84	23.80	11.20	23.21
	RecBi$_{filter}$	21	28.57	57	22.81	85	9.41	99	8.08	11.93	30.76	14.10	39.01
	RecBi$_{rand}$	7	285.71	47	48.94	65	43.08	88	21.59	29.76	72.24	30.08	71.41

[*] Columns "* " present the improvement rates of **RecBi** over a compared approach in terms of various metrics.

The overall improvements of RecBi over DiWi in terms of Top-1, Top-5, Top-10, Top-20 are 92.86%, 55.56%, 25.68%, and 11.46%, respectively. In particular, as demonstrated by the existing work [38], the Top-5 metric is more important in practice since most developers tend to abort the automated debugging tools if they cannot localize buggy elements within Top-5 positions [38], and thus RecBi is more practical than DiWi by largely improving the effectiveness of compiler bug isolation in terms of Top-5. The MFR and MAR values of RecBi are 8.26 and 8.60 respectively while those of DiWi are 15.17 and 15.53 respectively, demonstrating 45.55% and 44.62% improvements of RecBi over DiWi respectively. That demonstrates that RecBi indeed significantly outperforms the state-of-the-art approach DiWi for compiler bug isolation.

We also compared RecBi with the approach using the developer-provided passing test programs Dev. From Table 2, RecBi significantly outperform Dev in terms of all the metrics and on both GCC and LLVM. The overall improvements of RecBi over Dev are 440.00%, 191.67%, 97.87%, and 55.07% in terms of Top-1, Top-5, Top-10, and Top-20, respectively. Also, the overall improvements of RecBi over Dev are 72.38% and 71.58% in terms of MFR and MAR, respectively. The results demonstrate the apparent superiority of RecBi compared with Dev.

Qualitative Analysis. We further performed qualitative analysis on RecBi with two examples. Figure 5 shows two programs, where the left one is the given failing test program and the right one is a passing test program generated via our designed structural mutation (i.e., inserting a `while` statement). This bug is triggered when compiling the failing test program using GCC revision 228291 at -O2 and above. The root cause lies in the compiler file "`ifcvt.c`", which incorrectly uses 8-bit registers for optimization instead of 32-bit ones. By inserting a `while` statement with a false predicate, a passing test program is generated as shown in Figure 5b, since it invalidates the statement "`c=(b&15)^e;`" that triggers the buggy optimizations. We further calculated the similarity between the two test programs following Formula 1, which is 0.974. That demonstrates the power of our structural mutation that guarantees the generated passing test program to share a similar execution trace with the given failing test program. In particular, RecBi ranks the buggy file at the 2^{nd} position.

```
int printf(const char *,...);
int a;
int b=10;
char c;
int main (){
  char d;
  int e=5;
  for(a=0;a;a    ){e=0;}
  c=(b&15)^e;
  d=c>e?c:c<<e;
  printf("%d\n",d);
  return 0;
}
```

(a) Failing Program

```
int printf(const char *,...);
int a;
int b=10;
char c;
int main (){
  char d;
  int e=5;
  for(a=0;a;a    ){e=0;}
  while(e<a){c=(b&15)^e;}
  d=c>e?c:c<<e;
  printf("%d\n",d);
  return 0;
}
```

(b) Passing Program

Figure 5: GCC Bug 67786

```
int a;
void fn1 (){
  char b=0;
  for(;b!=  2;b    )
    for(a=0;a<1;a++)
      if((unsigned int)b>1)
        return;
}
int main (){
  fn1();
  if (a!=0)
    __builtin_abort();
  return 0;
}
```

(a) Failing Program

```
int a;
void fn1 (){
  char b=0;
  goto Label;
  for(;b!=  2;b    )
    for(a=0;a<1;a++)
      Label:
      if((unsigned int)b>1)
        return;
}
int main (){
  fn1();
  if(a!=0)
    __builtin_abort();
  return 0;
}
```

(b) Passing Program

Figure 6: LLVM Bug 24356

Figure 6 shows another example, which presents a passing test program (shown in Figure 6b) by inserting a `goto` statement to the failing test program (shown in Figure 6a). This bug is triggered when compiling the failing test program at `-O1` and above using LLVM revision 243961. The buggy file is `ScalarEvolution.cpp`, which causes that Line 6 is directly executed after Line 4 by skipping Line 5 due to incorrect optimization. By inserting a `goto` statement, the program structure avoid triggering the buggy optimization, leading to passing execution. The similarity between the two programs is 0.914, further confirming the effectiveness of RecBi. In particular, RecBi ranks the buggy file at the 5^{th} position.

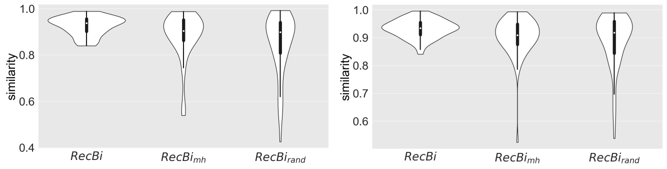
4.5.2 *RQ2: Contributions of Main Components.*

To answer RQ2, we investigated the contributions of two main components in RecBi, i.e., structural mutation and reinforcement learning based test program generation strategy.

Contribution of Structural Mutation in RecBi. We investigated the contribution of structural mutation by comparing DiWi and RecBi$_{mh}$ shown in Table 2. We found that RecBi$_{mh}$ performs better than DiWi in terms of all the metrics on both GCC and LLVM. More specifically, RecBi$_{mh}$ successfully isolates 23, 61, 83, and 99 bugs within Top-1, Top-5, Top-10, and Top-20 files respectively while DiWi only isolates 14, 45, 74, and 96 bugs respectively. The overall improvements of RecBi$_{mh}$ over DiWi are 28.54% and 27.88% in terms of MFR and MAR, respectively. The experimental results demonstrate that incorporating structural mutation indeed improves the effectiveness of compiler bug isolation, confirming the contribution of structural mutation in RecBi.

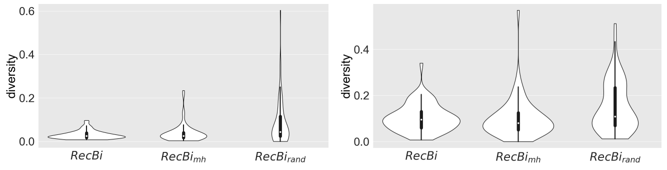
We further analyzed the contribution of each structural mutation operator to isolate compiler bugs. We found that for each studied bug, there exist the passing test programs generated by our designed structural mutation operators. More specifically, the four structural mutation operators (i.e., inserting branch statements, loop statements, function calls, and `goto` statements) generated passing test programs for 61%, 76%, 3%, and 25% bugs, respectively. That is, all the four structural mutation operators are indeed useful to generate passing test programs during compiler bug isolation. In particular, the operator inserting loop statements make the most contributions while that inserting function calls make the least contributions among them. This phenomenon is as expected since the operator inserting function calls tend to more largely change the program structure than the other three operators, leading to much less similarity between the generated passing test program and the given failing test program (i.e., a more low-quality passing test program). Then, due to the quality measurement (Formulae 4 and 5) in RecBi, such low-quality passing test programs are more likely to be filtered.

Contribution of Reinforcement Learning based Test Program Generation Strategy. We then investigated the contribution of our reinforcement learning based test program generation strategy, by comparing RecBi, RecBi$_{rand}$, and RecBi$_{mh}$ shown in Table 2. We found that among the three approaches, RecBi performs the best while RecBi$_{rand}$ performs the worst. More specifically, the improvements of RecBi over RecBi$_{rand}$ in terms of Top-1, Top-5, Top-10, Top-20, MFR, and MAR are 285.71%, 48.94%, 43.08%, 21.59%, 72.24%, and 71.41% respectively, and those of RecBi over RecBi$_{mh}$ are 17.39%, 14.75%,12.05%, 8.08%, 23.80%, and 23.21% respectively. That is, our reinforcement learning based strategy outperforms both the random strategy and the MH-based strategy used by the state-of-the-art approach DiWi, demonstrating the contribution of our reinforcement learning based test program generation.



(a) GCC **(b) LLVM**

Figure 7: Similarity between the given failing test program and generated passing test programs

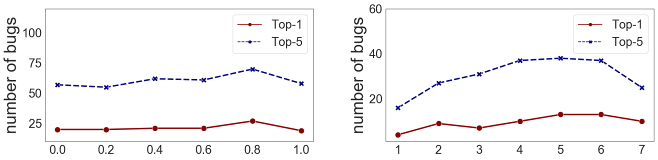


(a) GCC **(b) LLVM**

Figure 8: Diversity among generated passing test programs

We further analyzed the reason why our reinforcement learning based strategy outperforms the other two strategies by calculating the similarity (Formula 1) and diversity (Formula 2) for the generated passing test programs via the three strategies, respectively. Figure 7 presents the average similarity between the generated passing test programs and the given failing test program across all the bugs for the three strategies respectively, while Figure 8 presents the average diversity among all the passing test programs across all the bugs. In the two figures, the violin plots show the density at different values, and the box plots show the median and interquartile ranges. From Figures 7 and 8, both similarity values and diversity values for RecBi are distributed more intensively than those for RecBi$_{mh}$ and RecBi$_{rand}$, indicating that the quality of the generated passing test programs through our reinforcement learning based strategy is more stable. In general, the similarity achieved by RecBi is larger than that achieved by both RecBi$_{mh}$ and RecBi$_{rand}$, and RecBi does not have the low-quality passing test programs with little similarity. Moreover, we found that the diversity achieved by RecBi does not have superiority compared with RecBi$_{mh}$ and RecBi$_{rand}$. This phenomenon is as expected since both similarity and diversity actually contradict each other to some degree. The results indicate that when staying high similarity, enlarging the diversity would facilitate effective compiler bug isolation.

We then compared RecBi$_{filter}$ with RecBi$_{rand}$ to investigate the contribution of our designed measurement for the quality of a generated passing test program, which is the base of our reinforcement learning based strategy. From Table 2, we found that RecBi$_{filter}$ significantly outperforms RecBi$_{rand}$, although performing worse than RecBi. More specifically, the MFR and MAR values of RecBi$_{rand}$ are only 29.76 and 30.08 respectively while those of RecBi$_{filter}$ are 11.93 and 14.10 respectively. That is, incorporating our designed



(a) Impact of α **(b) Impact of u**

Figure 9: The impact of different parameter settings

quality measurement into the random strategy is able to largely improve the effectiveness of the random strategy, demonstrating the necessity of filtering low-quality passing test programs in RecBi.

4.5.3 RQ3: Impact of Different RecBi Configurations. We investigated the impact of two main parameters in RecBi, i.e., α and u. Figure 9a shows the effectiveness of RecBi at different α values in terms of Top-1 and Top-5 metrics. As demonstrated by the existing work [38], Top-1 and Top-5 results are more important in practical, and thus we used the two metrics as the representatives to evaluate the impact of different RecBi configurations. In Figure 9a, the x-axis represents the α values while the y-axis represents the metric values on both GCC and LLVM. From this figure, we found that our default setting (i.e., 0.8) is better than the other settings in terms of both Top-1 and Top-5 metrics, especially better than the settings of 0 and 1, indicating that combining both similarity and diversity outperforms individual similarity or diversity for generating effective passing test programs.

Figure 9b shows the effectiveness of RecBi at different u values in terms of Top-1 and Top-5 metrics. From this figure, we found that the best setting ranges from 4 to 6, including our default setting (i.e., 5). This is as expected since when the value of u is small, there is little future knowledge considered in RecBi, leading to worse effectiveness, while when the value of u is large, the prediction for the future potential reward could become more inaccurate, leading to worse effectiveness of RecBi.

5 DISCUSSION

5.1 Threats to Validity

The *internal* threat to validity mainly lies in the implementations of our approach RecBi and the compared approach DiWi. To reduce this threat, regarding the implementation of DiWi, we adopted the implementation released by the existing work [16]. Regarding the implementation of RecBi, we implemented it based on mature libraries as presented in Section 4.2 and carefully checked the code.

The *external* threats to validity mainly lie in the used compilers and bugs. Regarding the used compilers, following the existing work of compiler bug isolation [16], we also used two most popular C open-source compilers, i.e., GCC and LLVM. Regarding the used bugs, we used 120 real compiler bugs including all bugs from prior compiler bug isolation work [16]. To further reduce these threats, we will collect more real compiler bugs from more compilers to evaluate the effectiveness of RecBi.

The *construct* threats to validity mainly lie in the settings of parameters, randomness, and the used measurements. Regarding the settings of parameters, we have presented our specific settings in Section 4.2 and investigated the influence of main parameters in Section 4.5.3. Regarding the involved randomness in our study, we repeatedly ran all the approaches for 5 times, and calculated the median results. Regarding the used measurements, we have used several widely-used measurements in the area of bug localization. To further reduce this threat, we will try to apply RecBi to the industry and collect feedback from developers to evaluate RecBi.

5.2 Future Work

We discuss the following extensions of RecBi as our future work. First, RecBi does not specially deal with undefined behaviors [30] (i.e., the semantics of certain operations are undefined in the programming languages standards). Similar to the existing work [16], our mutation may also introduce undefined behaviors to a test program, causing that the compiler may produce uncertain results for the test program with undefined behaviors. However, undefined behaviors tend to affect bug detection since different results of a "failing" test program may be caused by real bugs or undefined behaviors. RecBi only keeps the passing test programs with the same results to isolate compiler bugs, and thus undefined behaviors may not affect RecBi. As demonstrated by the existing work [43, 65], it is challenging to identify undefined behaviors in compiler research, in the future we will try to relieve this problem by adopting existing efficient techniques [43]. Second, we will further extend RecBi to isolate compiler bugs at more fine-grained levels (e.g., methods) by calculating the suspicious scores at the corresponding levels.

6 RELATED WORK

RecBi is based on both mutation and reinforcement learning for compiler bug isolation, and thus we introduce three categories of related work, including compiler debugging, mutation-based bug isolation, and learning-based bug isolation.

Compiler Debugging. Besides the most related work DiWi [16] described before, Zeller [71] proposed to produce an entire cause-effect chain from input to result in GCC to facilitate debugging. Actually, RecBi is complementary to the cause-effect chain: 1) the latter produces bug-diagnosis information at the program-state level while the former does this at the source-code level, 2) the latter manipulates in memory and may not handle external states, 3) the former is more lightweight.

Besides, in compiler debugging there are many work focusing on providing debugging messages and visualization [11, 31, 39, 49, 54]. Some work also focused on simplifying the test programs triggering compiler bugs in order to facilitate debugging [10, 32, 53, 59, 72]. In our work, the failing test programs are collected from compiler bug reports, and *all of them have already been the simplified ones* as required by compiler developers [16]. Some work focused on identifying the test programs triggering the same compiler bug for efficient debugging [22, 33]. Different from them, our work focuses on *compiler bug isolation* by generating passing test programs via reinforcement learning and mutation.

Mutation-based Bug Localization. Mutation-based bug localization considers the actual impacts of code elements in the software systems under test on test outcomes to localize bugs through mutation testing [48, 50, 51, 74]. More specifically, it injects mutation bugs to each code element to simulate the actual impact of each code element. For example, Papadakis et al. [50, 51] proposed the first mutation-based bug localization approach, named *Metallaxis*.

Its basic insight is that if mutating a code element can change the outcome of some failing tests, the code element may have potential impact on the failing tests and thus may have been buggy. Meanwhile, Zhang et al. [74] independently proposed *FIFL*, the first mutation-based bug localization approach for evolving systems. The basic insight is that regression bugs can be simulated and localized via mutating corresponding code elements on the old program version. More recently, Moon et al. [48] proposed another mutation-based bug localization approach, named *MUSE*, based on the idea that mutating faulty code elements may cause more failed tests to pass than mutating correct elements. Different from these traditional mutation-based bug localization approaches, which aim to mutate the software systems under test, our approach RecBi aims to mutate the failing test cases (i.e., test programs) to generate passing test programs for compiler bug isolation.

Learning-based Bug Localization. In recent years, a lot of learning-based bug localization approaches have been proposed [40, 44, 55, 68]. For example, Xuan and Monperrus [68] proposed to utilize the learning-to-rank algorithm to localize bugs by combining different suspicious scores calculated by SBFL. Le et al. [40] further considered both the suspicious scores calculated by SBFL and program invariant to localize bugs through the learning-to-rank algorithm. Recently, Li et al. [44] proposed to use deep learning techniques to localize bugs by considering the suspicious scores calculated by SBFL and mutation based bug localization, as well as static features extracted from the defect prediction area [64] and information retrieval area [26]. Different from these learning-based bug localization approaches, which use learning techniques to rank all the suspicious code elements, our approach RecBi utilizes the *reinforcement learning* algorithm (i.e., A2C) to guide the process of passing test program generation for compiler bug isolation.

7 CONCLUSION

In this paper, we propose a reinforcement compiler bug isolation approach via structural mutation, which is called RecBi. RecBi first augments traditional local mutation operators with structural ones in order to generate a set of effective passing test programs for a given compiler bug with a failing test program. In particular, RecBi incorporates reinforcement learning to intelligently guide the process of passing test program generation. Based on the set of generated passing test programs and the given failing test program, RecBi ranks all the suspicious files by comparing the execution trace between them. We conducted an extensive study to evaluate RecBi based on two most popular C open-source compilers (i.e., GCC and LLVM) and 120 real bugs from them. The experimental results demonstrate the effectiveness of RecBi, significantly outperforming the state-of-the-art compiler bug isolation approach.

ACKNOWLEDGEMENTS

This work was partially supported by National Science Foundation under Grant Nos. CCF-1763906 and CCF-1942430, and Alibaba.

REFERENCES

[1] Accessed: 2020. Clang Libtooling library. http://clang.llvm.org/docs/LibTooling.html.

[2] Accessed: 2020. GCC. https://gcc.gnu.org.

[3] Accessed: 2020. Gcov. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[4] Accessed: 2020. LLVM. https://llvm.org.

[5] Accessed: 2020. PyTorch. https://pytorch.org/.

[6] R. Abreu, P. Zoeteweij, and A. J. c. Van Gemund. 2006. An Evaluation of Similarity Coefficients for Software Fault Localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. 39–46.

[7] 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.

[8] Milan Aggarwal, Aarushi Arora, Shagun Sodhani, and Balaji Krishnamurthy. 2018. Improving Search Through A3C Reinforcement Learning Based Conversational Agent. In *18th International Conference on Computational Science*. 273–286.

[9] Samuel Benton, Xia Li, Yiling Lou, and Lingming Zhang. 2020. On the Effectiveness of Unified Debugging: An Extensive Study on 16 Program Repair Systems. In *ASE*. to appear.

[10] Jacqueline M. Caron and Peter A. Darnell. 1990. Bugfind: A Tool for Debugging Optimizing Compilers. *SIGPLAN Notices* 25, 1 (1990), 17–22.

[11] Bor-Yuh Evan Chang, Adam Chlipala, George C. Necula, and Robert R. Schneck. 2005. Type-based verification of assembly language for compiler debugging. In *Proceedings of TLDI'05: 2005 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*. 91–102.

[12] Junjie Chen. 2018. Learning to accelerate compiler testing. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. 472–475.

[13] 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.

[14] 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*. 266–277.

[15] Junjie Chen, Yanwei Bai, Dan Hao, Lingming Zhang, Lu Zhang, Bing Xie, and Hong Mei. 2016. Supporting oracle construction via static analysis. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 178–189.

[16] 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 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. 223–234.

[17] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Lu Zhang. 2019. Static duplicate bug-report identification for compilers. *SCIENTIA SINICA Informationis* 49, 10 (2019), 1283–1298.

[18] 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.

[19] 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 (02 2020), 1–36.

[20] Junjie Chen, Guancheng Wang, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Lu Zhang. 2019. History-Guided Configuration Diversification for Compiler Test-Program Generation. In *34th IEEE/ACM International Conference on Automated Software Engineering*. 305–316.

[21] Junjie Chen, Guancheng Wang, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and XIE Bing. 2018. Coverage prediction for accelerating compiler testing. *IEEE Transactions on Software Engineering* (2018).

[22] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Z. Fern, Eric Eide, and John Regehr. 2013. Taming compiler fuzzers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 197–208.

[23] Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep differential testing of JVM implementations. In *Proceedings of the 41st International Conference on Software Engineering*. 1257–1268.

[24] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 85–99.

[25] Siddhartha Chib and Edward Greenberg. 1995. Understanding the metropolis-hastings algorithm. *The american statistician* 49, 4 (1995), 327–335.

[26] Tung Dao, Lingming Zhang, and Na Meng. 2017. How does execution information help with information-retrieval based bug localization?. In *Proceedings of the 25th International Conference on Program Comprehension*. 241–250.

[27] Nicholas DiGiuseppe and James A. Jones. 2011. On the Influence of Multiple Faults on Coverage-Based Fault Localization. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. 210–220.

[28] Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated testing of graphics shader compilers. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 93:1–93:29.

[29] Ivo Grondman, Lucian Busoniu, Gabriel AD Lopes, and Robert Babuska. 2012. A survey of actor-critic reinforcement learning: Standard and natural policy gradients. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42, 6 (2012), 1291–1307.

[30] Chris Hathhorn, Chucky Ellison, and Grigore Rosu. 2015. Defining the undefinedness of C. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 336–345.

[31] K. Scott Hemmert, Justin L. Tripp, Brad L. Hutchings, and Preston A. Jackson. 2003. Source Level Debugger for the Sea Cucumber Synthesizing Compiler. In *11th IEEE Symposium on Field-Programmable Custom Computing Machines.* 228.

[32] Satia Herfert, Jibesh Patra, and Michael Pradel. 2017. Automatically reducing tree-structured test inputs. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering.* 861–871.

[33] Josie Holmes and Alex Groce. 2018. Causal Distance-Metric-Based Assistance for Debugging after Compiler Fuzzing. In *29th IEEE International Symposium on Software Reliability Engineering.* 166–177.

[34] Shin Hong, Byeongcheol Lee, Taehoon Kwak, Yiru Jeon, Bongsuk Ko, Yunho Kim, and Moonzoo Kim. 2015. Mutation-Based Fault Localization for Real-World Multilingual Programs. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering.* 464–475.

[35] Dennis Jeffrey, Neelam Gupta, and Rajiv Gupta. 2008. Fault localization using value replacement. In *Proceedings of the 2008 international symposium on Software testing and analysis.* 167–178.

[36] 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.

[37] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. 1996. Reinforcement learning: A survey. *Journal of artificial intelligence research* 4 (1996), 237–285.

[38] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis.* 165–176.

[39] Nico Krebs and Lothar Schmitz. 2014. Jaccie: A Java-based compiler-compiler for generating, visualizing and debugging compiler components. *Sci. Comput. Program.* 79 (2014), 101–115.

[40] Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. 2016. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis.* 177–188.

[41] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation.* 216–226.

[42] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications.* 386–399.

[43] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. 2017. Taming undefined behavior in LLVM. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 633–647.

[44] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis.* 169–180.

[45] Xia Li and Lingming Zhang. 2017. Transforming programs and tests in tandem for fault localization. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 92:1–92:30.

[46] Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. 2020. Can Automated Program Repair Refine Fault Localization? A Unified Debugging Approach. In *ISSTA.* to appear.

[47] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. (2013).

[48] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the mutants: Mutating faulty programs for fault localization. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation.* IEEE, 153–162.

[49] Kazunori Ogata, Tamiya Onodera, Kiyokuni Kawachiya, Hideaki Komatsu, and Toshio Nakatani. 2006. Replay compilation: improving debuggability of a just-in-time compiler. In *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications.* 241–252.

[50] Mike Papadakis and Yves Le Traon. 2012. Using Mutants to Locate "Unknown" Faults. In *Fifth IEEE International Conference on Software Testing, Verification and Validation.* 691–700.

[51] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. *Softw. Test. Verification Reliab.* 25, 5-7 (2015), 605–628.

[52] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE).* IEEE, 609–620.

[53] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation.* 335–346.

[54] Anthony M. Sloane. 1999. Debugging Eli-Generated Compilers With Noosa. In *Compiler Construction, 8th International Conference, CC'99, Held as Part of the European Joint Conferences on the Theory and Practice of Software.* 17–31.

[55] 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.

[56] Sriram Srinivasan, Marc Lanctot, Vinicius Zambaldi, Julien Pérolat, Karl Tuyls, Rémi Munos, and Michael Bowling. 2018. Actor-critic policy optimization in partially observable multiagent environments. In *Advances in neural information processing systems.* 3422–3435.

[57] Pei-Hao Su, Pawel Budzianowski, Stefan Ultes, Milica Gasic, and Steve Young. 2017. Sample-efficient actor-critic reinforcement learning with supervised data for dialogue management. *arXiv preprint arXiv:1707.00130* (2017).

[58] 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.

[59] 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.

[60] Richard Sutton, David Mcallester, Satinder Singh, and Yishay Mansour. 2000. Policy Gradient Methods for Reinforcement Learning with Function Approximation. *Adv. Neural Inf. Process. Syst* 12 (02 2000).

[61] R. S. Sutton and A. G. Barto. 1998. Reinforcement Learning: An Introduction. *IEEE Transactions on Neural Networks* 9, 5 (1998), 1054–1054.

[62] Konda Vijay, R. and Tsitsiklis John, N. 2000. Actor-critic Algorithms. *SIAM Journal on Control and Optimization* (April 2000).

[63] Mnih Volodymyr, Badia Adrià, Puigdomènech, Mirza Mehdi, and Graves Alex. 2016. Asynchronous Methods for Deep Reinforcement Learning. In *ICML2016.* 1928–1937.

[64] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering.* 297–308.

[65] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. 2013. Towards optimization-safe systems: analyzing the impact of undefined behavior. In *ACM SIGOPS 24th Symposium on Operating Systems Principles.* 260–275.

[66] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. Deep Learning Library Testing via Effective Model Generation. In *The 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* to appear.

[67] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Trans. Software Eng.* 42, 8 (2016), 707–740.

[68] Jifeng Xuan and Martin Monperrus. 2014. Learning to Combine Multiple Ranking Metrics for Fault Localization. In *30th IEEE International Conference on Software Maintenance and Evolution.* 191–200.

[69] Jifeng Xuan and Martin Monperrus. 2014. Test case purification for improving fault localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering.* 52–63.

[70] 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.

[71] Andreas Zeller. 2002. Isolating cause-effect chains from computer programs. In *Proceedings of the Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering.* 1–10.

[72] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Software Eng.* 28, 2 (2002), 183–200.

[73] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. 2011. Localizing failure-inducing program edits based on spectrum information. In *2011 27th IEEE International Conference on Software Maintenance (ICSM).* 23–32.

[74] Lingming Zhang, Lu Zhang, and Sarfraz Khurshid. 2013. Injecting mechanical faults to localize developer faults for evolving software. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications.* 765–784.

[75] 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.