LocSeq: Automated Localization for Compiler Optimization Sequence Bugs of LLVM

Zhide Zhou, He Jiang, *Member, IEEE*, Zhilei Ren, Yuting Chen, and Lei Qiao

***Abstract*—Compiler bugs may be triggered when programs are optimized with optimization sequences. However, diagnosing compiler optimization sequence bugs is difficult due to limited debugging information. Although some techniques (e.g., DiWi and RecBi) have been proposed to automatically localize compiler bugs, no systematic work has been conducted to automatically localize compiler optimization sequence bugs. In this article, we propose LocSeq, a novel technique to automatically localize compiler optimization sequence bugs of LLVM. The core insight of LocSeq is based on the fact that the behaviors of optimizations may be influenced by each other, and thus, the innocent files may be excluded by constructing bug-free optimization sequences. First, given a buggy optimization sequence that triggers a compiler bug, in LocSeq, we transform the problem of the localization for a compiler optimization sequence bug to the problem of the construction for bug-free optimization sequences, which are helpful to localize buggy compiler files. Then, a constrained genetic algorithm is presented in LocSeq to generate a set of bug-free optimization sequences that share similar compiler execution traces with the buggy optimization sequence. Finally, LocSeq leverages a spectrum-based bug localization technique to localize the compiler optimization sequence bug by comparing the execution traces between bug-free optimization sequences and the buggy optimization sequence. To evaluate the effectiveness of LocSeq, we build a benchmark, including 60 optimization sequence bugs of LLVM, and compare LocSeq with the state-of-the-art techniques DiWi and RecBi. The experimental results show that LocSeq significantly outperforms DiWi and RecBi by up to 366.66%/72.27% and 250.00%/56.00% for localizing optimization sequence bugs within Top-1/5 files, respectively.**

***Index Terms*—Compiler bug, fault localization, genetic algorithm, LLVM, optimization sequences.**

Manuscript received August 1, 2021; revised November 5, 2021 and March 4, 2022; accepted March 24, 2022. Date of publication April 28, 2022; date of current version June 2, 2022. This work was supported in part by the National Natural Science Foundation of China under Grant 62032004, Grant 62132020, Grant 62072068, and Grant 61902181. Associate Editor: Ruizhi Gao.*(Corresponding author: He Jiang.)*

Zhide Zhou and Zhilei Ren are with the Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province and the School of Software, Dalian University of Technology, Dalian 116024, China (e-mail: cszide@gmail.com; zren@dlut.edu.cn).

He Jiang is with the Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province, the School of Software, and DUT Artificial Intelligence, Dalian University of Technology, Dalian 116024, China (e-mail: jianghe@dlut.edu.cn).

Yuting Chen is with the Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China (e-mail: chenyt@sjtu.edu.cn).

Lei Qiao is with the Beijing Institute of Control Engineering, Beijing 100190, China (e-mail: fly2moon@163.com).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TR.2022.3165378.

Digital Object Identifier 10.1109/TR.2022.3165378

I. Introduction

OPTIMIZATION sequences (i.e., a set of compiler optimizations in a certain order) are often employed in compilers (e.g., GCC and Clang) to achieve satisfactory performance for programs, e.g., running time, code size, and throughput. Especially, options O1, O2, and O3 in two widely used compilers GCC and Clang are some optimization sequences designed by compiler developers. However, potential compiler bugs (e.g., crashes and wrong code) may be triggered when optimizing programs with some optimization sequences (hereafter, we call these compiler bugs as *compiler optimization sequence bugs*), which may decrease the usability of compilers and introduce unintended application behavior and disasters. Many studies [1]–[4] for compiler autotuning have shown that optimization sequences may lead compilers to crash or produce incorrect program execution. Besides, Jiang *et al.* [5] conducted a study to detect compiler optimization sequence bugs for LLVM. In their study, they have reported more than 100 LLVM bugs caused by optimization sequences. Nevertheless, locating and fixing compiler optimization sequence bugs may be very time-consuming due to the extremely large-scale and complicated codebases of compilers [6]–[10]. Therefore, it is critical to develop automated techniques to help compiler developers localize and fix optimization sequence bugs.

Although the consequence of failures introduced by software bugs has been analyzed [11] and many techniques [12]–[19] have been proposed for the bug localization of common software systems, these existing approaches can hardly localize compiler bugs due to either extremely high costs or poor effectiveness [8], [9]. For example, Holmes and Groce [19] presented a mutation-based method that could be used to localize compiler bugs. However, this method may take a long time to execute only a limited set of compiler mutants [8]. Hence, to facilitate the localization of compiler bugs, Chen *et al.* proposed two techniques based on mutation, namely, DiWi [8] and RecBi [9]. In DiWi and RecBi, Chen *et al.* transformed the problem of locating compiler bugs to the problem of generating passing test programs that cannot trigger the corresponding compiler bug. Then, the generated passing test programs are compiled by compilers with or without the default options (e.g., O1, O2, and O3). Hence, compiler bugs are localized by comparing the execution traces between the passing test programs and the given failing test program. The main differences between DiWi and RecBi are the mutation operators and the strategy to select these mutation operators. However, no systematic investigation has

been conducted to automatically localize compiler optimization sequence bugs.

For a compiler optimization sequence bug, the inputs of compilers are the failing test program and the buggy optimization sequence. Although both DiWi and RecBi can be adjusted and then applied to localize compiler optimization sequence bugs by treating the buggy optimization sequence as a fixed option of compilers, they still suffer from the effectiveness issue. To localize compiler optimization sequence bugs, we may need many bug-free execution traces of compilers due to the complexity of compiler optimizations and the interaction among them. However, in our experiments, DiWi and RecBi can only generate a few passing test programs in a fixed time. For example, DiWi/RecBi only generates about 7/15 passing test programs on average in 1 h. Particularly, in some worst cases, they fail to generate any passing test programs in our experiments. This may affect the effectiveness to localize compiler optimization sequence bugs.

In this article, we aim to investigate the problem of localizing compiler optimization sequence bugs of LLVM [20]. LLVM is a mature and widely used compiler infrastructure, which includes hundreds of analysis and transformation optimizations [21]. Moreover, many tools (e.g., Klee [22] and Phasar [23]) and compilers (e.g., Clang [24] and NVCC for CUDA [25]) of different programming languages have been implemented based on LLVM. In contrast to DiWi and RecBi that aim to obtain bug-free compiler execution traces by generating a set of passing test programs using mutation operators, we aim to obtain bug-free compiler execution traces by constructing a set of bug-free compiler optimization sequences in our study. The key insight of our idea is that the behavior of a compiler optimization may be influenced by other compiler optimizations. Moreover, compiler optimizations with different orders may lead to different behaviors. Thus, based on this observation, we transform the problem of the localization for a compiler optimization sequence bug to the problem of the construction for bug-free optimization sequences. These bug-free sequences cannot trigger the corresponding compiler bug, but share similar execution traces with the buggy optimization sequence. Hence, we may localize the buggy compiler files by comparing the execution traces between bug-free optimization sequences and the buggy optimization sequence.

Based on the above analysis, in this study, we present LocSeq, a novel technique to automatically localize compiler optimization sequence bugs of LLVM. First, in LocSeq, we treat the problem of constructing a set of bug-free compiler optimization sequences as a search problem. That is, LocSeq utilizes a search algorithm to seek a set of bug-free compiler optimization sequences such that the corresponding compiler execution traces are similar to that of the buggy compiler optimization sequence. To this end, we develop a novel constrained genetic algorithm (CGA). Specifically, in the search process, we constrain that each individual in CGA must contain all the optimizations in the buggy optimization sequence. Under this constraint, the CGA randomly builds an initial population of optimization sequences. Then, the CGA utilizes a signal-point crossover and four mutation operators to mutate each individual in the population. After the search process, we can obtain a set of compiler execution traces that are produced by compilers under the bug-free optimization sequences. Finally, similar to DiWi and RecBi, LocSeq measures the suspicious score for each file covered by the buggy optimization sequence based on the spectrum-based software bug localization techniques [26], [27]. The larger the suspicious score of a file, the buggier the file.

To evaluate the effectiveness of LocSeq, we first construct a benchmark that includes 60 real-world reproducible compiler optimization sequence bugs of LLVM. Then, LocSeq is compared with two start-of-the-art techniques, i.e., DiWi and RecBi. The experimental results demonstrate that LocSeq significantly outperforms DiWi and RecBi by up to 366.66%/72.27% and 250.00%/56.00% in terms of Top-1/5 results, respectively. Specifically, LocSeq can generate about 945 bug-free optimization sequences on average in 1 h and successfully localize 23.33%/65.00% bugs within Top-1/5 files of LLVM. Besides, we also investigate the contribution of CGA by comparing LocSeq with LocSeq$_r$. LocSeq$_r$ is a variant of LocSeq that utilizes a random strategy to generate bug-free optimization sequences that still satisfy the aforementioned constraint. To investigate the contribution of the constraint in CGA, we remove the constraint in LocSeq and LocSeq$_r$, resulting in LocSeq$_{rwc}$ and LocSeq$_{wc}$, respectively. The results show that LocSeq also outperforms LocSeq$_r$, LocSeq$_{rwc}$, and LocSeq$_{wc}$. For instance, LocSeq localizes 75.00%, 133.33%, and 27.27% more bugs than LocSeq$_r$, LocSeq$_{rwc}$, and LocSeq$_{wc}$ at the Top-1 position, respectively.

This article makes the following contributions.

1) To the best of our knowledge, this article is the first work to investigate the problem of localizing compiler optimization sequence bugs.
2) We propose LocSeq, a novel technique to automatically localize compiler optimization sequence bugs of LLVM. In LocSeq, we present a novel CGA to seek a set of bug-free compiler optimization sequences, such that the corresponding compiler execution traces are as similar as possible to that of the buggy compiler optimization sequence.
3) We construct a benchmark including 60 real-world reproducible compiler optimization sequence bugs of LLVM for future research on the localization and fixing of compiler optimization sequence bugs.
4) The experimental results based on the constructed benchmark show that LocSeq successfully localizes 23.33%/65.00% bugs within Top-1/5 files of LLVM, which significantly outperforms DiWi and RecBi by up to 366.66%/72.27 and 250.00%/56.00%, respectively.

The rest of this article is organized as follows. We present the motivation of our study in Section II-B. Then, the proposed technique is shown in Section III. Next, we present the evaluation results in Section IV. The threats to validity and related work are described in Sections V and VI. Finally, Section VII concludes this article.

II. Background and Motivation

A. LLVM and Optimization Sequence

In this article, we study the localization of compiler optimization sequence bugs of LLVM. Originally, LLVM represents the low-level virtual machine that is a statically typed intermediate representation developed by Lattner [28]. However, LLVM now is a mature and widely used compiler infrastructure,[1] which provides a collection of modular and reusable compiler and toolchain technologies for arbitrary programming languages. Currently, LLVM has been used in a wide variety of commercial and open-source projects[2] and as well as being widely used in academic research.[3] With a common infrastructure, a broad variety of statically and runtime compiled languages (e.g., C, C++, Rust, Swift, Ruby, Haskell, and WebAssembly) have been implemented based on LLVM.

To improve program performance, many analysis and transformation optimization techniques have been developed in LLVM, such as the dead code elimination[4] and the loop invariant code motion.[5] Generally, for a given program, we need to use a set of optimizations in a certain order to achieve satisfactory performance [5]. Here, this set of optimizations in a certain order is called an optimization sequence. Especially, the default optimization levels (e.g., O1, O2, and O3) in compilers are predefined optimization sequences, which usually contain more than 100 optimizations. These default optimization levels could guarantee that programs achieve acceptable performance in most cases. However, many studies (see, e.g., [29] and [30]) have shown that selecting good optimization sequences for a given program could further improve the performance of the program.

B. Motivation

To illustrate the motivation of our study, we present two concrete compiler optimization sequence bugs of LLVM.

Fig. 1 shows the programs that trigger LLVM Bug#47557 and LLVM Bug#31199. In Fig. 1(a), when LLVM optimizes the program using the buggy optimization sequence "*-simplifycfg -instcombine -early-cse-memssa -loop-unroll -loop-unswitch -loop-reduce -loop-simplifycfg*," the assertion "`Assertion MA → use_empty() && "Trying to remove memory access that still has uses"`" in the source file "*MemorySSA.cpp*[6]" is violated. The reason for this bug is that the optimization "*-loop-reduce*" claims to preserve the information provided by "*MemorySSA.cpp*," but it is broken when splitting critical edges.[7] The second program in Fig. 1(b) incurs a wrong code bug of LLVM when LLVM optimizes this program using the buggy optimization sequence "*-mem2reg -loop-rotate -licm -loop-unroll*." The result of the program in Fig. 1(b) should be 0, while it is 1 after applying

```
1  int a, b;
2  int c[];
3  long d;
4  void e(){
5  f:
6    d = 0;
7    for(;; d++){
8      int g = 0;
9      for(;g<=1;g++){
10           a = 0;
11           for(; a;)
12             c[d] = 0;
13           if(b)
14             goto f;
15     }
16   }
17 }
```

(a)

```
1  int a, b, c, d[1]={1};
2  int main()
3  {
4    int e;
5    for (;c<1;c++)
6    {
7      for(e=0;e<1;e++)
8        for(;a<1;a++)
9          for(;b<1;b++)
10           d[b] = 0;
11     if(d[0])
12       a--;
13   }
14   printf ("%d\n", a);
15   return 0;
16 }
```

(b)

Fig. 1. Programs in crash bug 47557 and wrong code bug 31199 of LLVM. (a) LLVM Bug#47557 (https://bugs.llvm.org/show_bug.cgi?id=47557). (b) LLVM Bug#31199 (https://bugs.llvm.org/show_bug.cgi?id=31199).

the buggy optimization sequence. This is because of the wrong computing of the alias sets for a subloop implemented in the optimization "*-licm*."[8] Note that the optimization sequences of these two LLVM bugs are minimized, which means that we cannot reproduce the corresponding bugs when we remove any optimization in the optimization sequences or change the order of them. In practice, when LLVM developers debug optimization sequence bugs, they always blame the last optimization in the sequence [5]. However, the root causes of optimization sequence bugs may occur in any optimization in the sequence. For example, the root cause of LLVM Bug#31199 in Fig. 1(b) occurs in "*-licm*" rather than the last optimization "*-loop-unroll*" in the sequence. In addition, although we can know the location of the violated assertion as for LLVM Bug#47557, the root causes of bugs are not always introduced by the component that contains the assertion. For LLVM Bug#47557, the root cause is introduced by "*-loop-reduce*" rather than "*MemorySSA.cpp*." Hence, efficient and automated techniques should be developed to help developers analyze and localize compiler optimization sequence bugs.

For the above two LLVM bugs, even if we only consider the source files (i.e., the file ends with ".cpp") executed by LLVM when we apply the optimization sequences, the compiler execution traces of these two bugs still cover 1008 and 713 suspect source files. In the worst case, the root causes of the corresponding bugs may be localized at any one of these files. This may make developers take a long time to find the root causes and fix bugs. However, in practice, the developers of LLVM only need to modify no more than two files on average to fix an optimization bug [10]. This indicates that most files covered by the execution traces of buggy optimization sequences are innocent files. In our study, we find that the behavior of a compiler optimization may be influenced by other compiler optimizations. Moreover, compiler optimizations with different orders may lead to different behaviors [5]. For example, if the positions of the last

[1] http://llvm.org/.
[2] https://llvm.org/Users.html.
[3] https://llvm.org/pubs/.
[4] http://en.wikipedia.org/wiki/Dead_code_elimination.
[5] http://en.wikipedia.org/wiki/Loop-invariant_code_motion.
[6] https://github.com/llvm/llvm-project/blob/main/llvm/lib/Analysis/MemorySSA.cpp.
[7] https://reviews.llvm.org/rG57ae9bb93235.
[8] https://reviews.llvm.org/rGfd2d7c72fcfd.

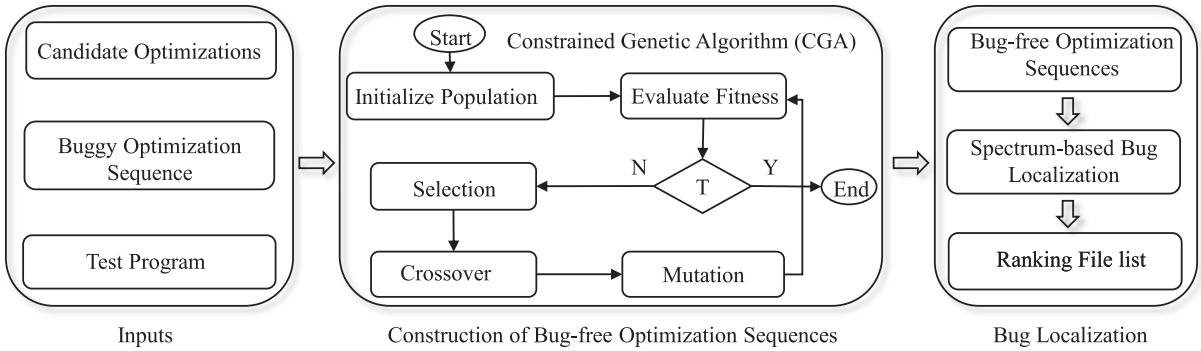

Fig. 2. Framework of LocSeq for localizing optimization sequence bugs of LLVM.

two optimizations in the buggy optimization sequence for LLVM Bug#47557 are swapped, we can obtain a bug-free optimization sequence "*-simplifycfg -instcombine -early-cse-memssa -loop-unroll -loop-unswitch -loop-simplifycfg -loop-reduce*." In this case, the similarity is 0.96 (calculated by formula (2) in Section III-C) for the execution traces between the bug-free optimization sequence and the buggy optimization sequence. Hence, by comparing the execution traces between bug-free optimization sequences and the buggy optimization sequence, we may eliminate innocent files from suspects and localize the buggy files.

Therefore, in our study, we transform the problem of the localization for a compiler optimization sequence bug to the problem of the construction for bug-free optimization sequences. Intuitively, a bug-free optimization sequence that shares more similar compiler execution traces with the buggy optimization sequence tends to be more helpful to eliminate innocent files from suspects. Thus, the problem of the construction for bug-free optimization sequences can be modeled as a search problem. The object of the search problem is to maximize the similarities of the execution traces between the bug-free optimization sequences and the buggy optimization sequence. For instance, by using the proposed CGA in our study, we construct 687 bug-free optimization sequences in 1 h for LLVM Bug#47557, and the buggy file is successfully ranked at the second position. Similarly, the buggy file of LLVM Bug#31199 can be ranked at the first position using the proposed technique.

In the next section, we present LocSeq, a novel technique to automatically localize compiler optimization sequence bugs of LLVM. A CGA is developed to seek a set of bug-free optimization sequences, such that the corresponding compiler execution traces are as similar as possible to that of the buggy optimization sequence.

III. Approach

In this section, we first introduce the framework of LocSeq to automatically localize compiler optimization sequence bugs of LLVM in Section III-A. Then, we present the proposed CGA to construct a set of bug-free optimization sequences, including the solution representation (see Section III-B), the fitness function (see Section III-C), and the details of CGA (see Section III-D). Finally, to make our article self-contained, we also introduce the traditional fault localization technique used in our article in Section III-E.

A. Overview of LocSeq

Fig. 2 shows the framework of LocSeq. The core insight of LocSeq is to construct bug-free optimization sequences that share similar compiler execution traces with the buggy optimization sequence. Thus, we may exclude the innocent files by comparing the execution traces between bug-free optimization sequences and the buggy optimization sequence and then localize the buggy files. Generally, LocSeq consists of three components, namely, "*Inputs*," "*Construction of Bug-free Optimization Sequences*," and "*Bug Localization*." In the left of Fig. 2, the component "*Inputs*" indicates the information of an optimization sequence bug and the optimizations of LLVM. For a given optimization sequence bug of LLVM, we basically need to know the test program and the buggy optimization sequence to localize the buggy files. Besides, to construct a set of bug-free optimization sequences, we also need a set of candidate optimizations. This is because in some cases, only a few optimizations exist in the buggy optimization sequence, which is not sufficient to construct bug-free optimization sequences.

The middle component in Fig. 2 is the proposed CGA to construct a set of bug-free optimization sequences. In our study, we select the genetic algorithm as the search algorithm. The reason is that the genetic algorithm is easy to be implemented and has been widely and successfully used in software engineering problems (see, e.g., [31]–[36]). In general, the process of CGA is identical to the traditional genetic algorithm, including *Initialize Population*, *Evaluate Fitness*, *Selection*, *Crossover*, and *Mutation*. The main difference between CGA and the traditional genetic algorithm is that a constraint is applied in CGA. That is, when we initialize the population and apply the crossover operator or the mutation operators to each individual, all the optimizations into the buggy optimization sequence must be included in each individual. This is because the files covered by the execution trace are directly influenced by the optimizations in the optimization sequence. By applying this constraint, we can improve the possibility to include the files covered by the execution trace of the buggy optimization sequence into

TABLE I
CANDIDATE OPTIMIZATIONS OF LLVM

-adce -argpromotion -bdce -correlated-propagation
-constmerge -dse -early-cse -early-cse-memssa -globaldce
-globalopt -gvn -indvars -instcombine -instsimplify
-ipsccp -lcssa -licm -loop-deletion -loop-distribute
-loop-idiom -loop-load-elim -loop-rotate -loop-simplify
-loop-sink -loop-unroll -loop-unswitch -loop-vectorize
-reassociate -sccp -simplifycfg -slp-vectorizer -sroa

the execution trace of bug-free optimization sequences. This is beneficial to obtain more accurate information to calculate the suspicious values of the files covered by the execution trace of the buggy optimization sequence.

After obtaining a set of bug-free optimization sequences, we adopt the spectrum-based bug localization (also called spectrum-based fault localization—SBFL) technique [26], [27], [37]–[39] to localize the buggy files that include the bug caused by the buggy optimization sequence. SBFL first calculates the suspicious values of each file according to the execution traces between bug-free optimization sequences and the buggy optimization sequence. Then, the files are ranked according to their suspicious values. The larger the suspicious value, the buggier the file. Finally, we obtain a ranking file list, and developers may quickly localize the bug by checking this file list.

From Fig. 2, the second component (i.e., CGA) clearly is the foundation of LocSeq. Thus, to present CGA, we first introduce the solution representation and fitness function in Sections III-B and III-C, respectively. Then, the details of CGA are presented in Section III-D.

B. Solution Representation

In our study, a solution indicates the individual in the population of CGA. An optimization sequence is constituted by some optimizations in a certain order [5]. Besides, an optimization can appear multiple times in an optimization sequence. Hence, we utilize a list to represent a solution in this study. That is, the list contains all the optimizations in an optimization sequence, and the indexes of these optimizations are identical to those of them in the optimization sequence. For example, the buggy optimization sequence of LLVM Bug#47557 in Fig. 1 can be represented as ["*-simplifycfg*," "*-instcombine*," "*-early-cse-memssa*," "*-loop-unroll*," "*-loop-unswitch*," "*-loop-reduce*," "*-loop-simplifycfg*"].

However, as mentioned in Section III-A, we may not construct many bug-free optimization sequences when we only use the optimizations in the buggy optimization sequence. Thus, besides the optimizations in the buggy optimization sequence, we also need to include some candidate optimizations that do not belong to the buggy optimization sequence into a solution. In our study, we collect a set of candidate optimizations of LLVM, as shown in Table I. These candidate optimizations have existed for a long time in LLVM and are also contained in the default optimization levels (e.g., O3) of LLVM. This may help to avoid extra LLVM bugs in the search process of CGA and construct more bug-free optimization sequences. Therefore, assume that O_b and O_c are the sets of the optimizations in the buggy optimization sequence and the candidate optimizations, respectively; a solution *Sol* is defined as

$$Sol = [O_1, O_2, \ldots, O_n] \tag{1}$$

where $O_i \in O_b \cup O_c$ and $O_b \subseteq Sol$. n is the length of Sol and the value of n can be any positive integer that is greater than the length of the buggy optimization sequence. In practice, to save memory, each O_i can be represented as an integer index that points to the corresponding optimization in $O_b \cup O_c$.

C. Fitness Function

In this subsection, we present the fitness function of CGA. The object of CGA is to seek a set of bug-free optimization sequences, such that the corresponding compiler execution traces are as similar as possible to that of the buggy optimization sequence. In our study, to determine whether a generated optimization sequence is bug-free, we use it to optimize the test program of the corresponding bug. If the bug cannot be reproduced or other bugs (e.g., crash) cannot be introduced by the generated optimization sequence, we say that the generated optimization sequence is bug-free. Thus, the fitness function of CGA can be defined as the similarity of execution traces between the bug-free optimization sequence and the buggy optimization sequence. Similar to DiWi and RecBi, we also measure the similarity between two execution traces utilizing the Jaccard similarity coefficient,[9] which is defined as the size of the intersection divided by the size of the union of two sample sets *A* and *B*, i.e., $J(A, B) = |A \cap B|/|A \cup B|$. Assume that Cov_a and Cov_b are the set of statements covered by the execution traces of the bug-free optimization sequence a and the buggy optimization sequence b; then, the similarity between a and b is as follows:

$$Sim(a, b) = \frac{|Cov_a \cap Cov_b|}{|Cov_a \cup Cov_b|}. \tag{2}$$

However, in our study, we only need to measure the similarity between the execution traces of a and b. Hence, if a includes many optimizations that are not in b, Cov_a may be much larger than Cov_b since many files may be covered by these optimizations. This may result in a small similarity between a and b even if Cov_a contains a subset of the execution trace that is similar to Cov_b. Besides, if we only consider the files covered by b and calculate Cov_a according to these files, Cov_a may be not accurate. This is because many common files are dependent on many optimizations in LLVM. Meanwhile, if a covers too many files, we will need more time and resources to process these files. Therefore, we redefined the similarity between the execution traces of a and b as follows:

$$Sim'(a, b) = \frac{Sim_s(a, b)}{1 - Sim_f(a, b)} \tag{3}$$

$$Sim_s(a, b) = \frac{|Cov'_a \cap Cov_b|}{|Cov'_a \cup Cov_b|} \tag{4}$$

$$Sim_f(a, b) = \frac{|Cov_{fa} \cap Cov_{fb}|}{|Cov_{fa} \cup Cov_{fb}|} \tag{5}$$

[9]https://en.wikipedia.org/wiki/Jaccard_index.

TABLE II
DEFINITIONS OF Cov_a, Cov_b, Cov'_a, Cov_{fa}, AND Cov_{fb}

a	The bug-free optimization sequence
b	The buggy optimization sequence
Cov_{fa}	Files covered by the execution traces of a
Cov_{fb}	Files covered by the execution traces of b
Cov_a	Statements covered by the execution traces of a
Cov_b	Statements covered by the execution traces of b
Cov'_a	Statements in the file $f \in (Cov_{fa} \cap Cov_{fb})$ covered by a

where $Sim_s(a, b)$ (or $Sim_f(a, b)$) is the similarity between the execution traces (or the files) of a and b; Cov_{fa} and Cov_{fb} represent the files in compiler covered by the execution traces of a and b, respectively. Cov'_a is the set of statements in the file $f \in (Cov_{fa} \cap Cov_{fb})$ covered by the execution traces of a. By this way, if a covers many files that are not covered by b, i.e., $Sim_f(a, b)$ tends to be small, $Sim'(a, b)$ will be also small. In the best case, if a and b cover the same files (i.e., $Sim_f(a, b) = 1$), $Sim'(a, b) = Sim_s(a, b)$, which is rarely occurred in our experiments. Notably, if the bug is still occurred when applying a, the fitness of a is set to 0. Table II lists the definitions of Cov_a, Cov_b, Cov'_a, Cov_{fa}, and Cov_{fb} to clearly understand the fitness function.

D. Constrained Genetic Algorithm

In Sections III-B and III-C, we have introduced the solution representation and the fitness function in CGA. In this subsection, we will present the details of CGA. Algorithm 1 shows the main steps of CGA, which are the same as those of a standard genetic algorithm. First, CGA starts with the creation of an initial population of random optimization sequences (line 2). Then, the population is evolved using the *crossover* and *mutation* operators (lines 8–15). Finally, CGA selects the fittest individuals according to the fitness function (lines 18 and 19). As indicated by the name of CGA, a constraint must be satisfied in the process of CGA. That is, all the optimizations in the buggy optimization sequence must be contained in each individual of CGA, which is applied in steps of the population initialization, crossover, and mutation. This is the main difference between CGA and the standard genetic algorithm. In Algorithm 1, the fitness of each individual in the population is evaluated by the function *Evaluate*($\cdot$). Specifically, each individual is transformed into the corresponding optimization sequence. Next, the test program is compiled using this optimization sequence, and the fitness is calculated by the fitness function, as described in Section III-C. Since only the initialization, crossover, and mutation are customized in our study, we introduce them as follows.

1) Initialize Population: The routine utilized to generate the initial population is shown in Algorithm 2. Algorithm 2 aims to randomly generate a set of individuals (i.e., optimization sequences) and takes the optimizations O_b in the buggy optimization sequence and the candidate optimizations O_c as inputs. The output of Algorithm 2 is an initial population P_0.

First, to guarantee that each individual in the population contains O_b (i.e., satisfy the constraint mentioned above), in Algorithm 2, we include O_b into the initial sequence s (line 4). Then, a random number is selected to determine how many candidate optimizations can be included in s (line 5). In lines 6 and 7, we select optimizations from O_c one by one and include them into s. After obtaining the initial sequence s, we shuffle it to make the optimizations in O_b have different orders in different optimization sequences (line 8). Finally, we add the generated optimization sequence s into the initial population P_0 (line 9).

2) Crossover: In this study, we adopt the single-point crossover as the crossover operator of CGA. The single-point crossover generates two offsprings by randomly exchanging

Algorithm 1: Constrained Genetic Algorithm.

Input: N: population size.
Input: PB_{cx}: crossover probability.
Input: PB_{mut}: mutation probability.
Input: O_b: optimizations in buggy optimization sequence.
Input: O_c: candidate optimizations.
Output: S_{free}: bug-free optimization sequences.

```
1  g = 0;
2  P_g = Initialize-Population(N, O_b, O_c);
3  Evaluate(P_g);
4  Add bug-free individual in P_g into S_free;
5  while not termination do
6      g = g + 1;
7      S = φ;
8      while |S| < N do
9          p_1, p_2 = two individuals in P_g;
10         if random() < PB_cx then
11             p_1, p_2 = Crossover(p_1, p_2);
12         S = S ∪ {p_1, p_2};
13     for each p ∈ S do
14         if random() < PB_mut then
15             Mutation(p);
16     Evaluate(S);
17     Add bug-free individual in S into S_free;
18     P_g = P_{g-1} ∪ S;
19     P_g = N fittest individuals in P_g;
```

Algorithm 2: Initialize-Population.

Input: N: population size.
Input: O_b: optimizations in buggy optimization sequence.
Input: O_c: candidate optimizations.
Output: P_0: an initial population.

```
1  P_0 = φ;
2  num = |O_b|;
3  while |P_0| < N do
4      s = O_b;
5      size = random(num + 1, MAX_SIZE);
6      while |s| < size do
7          s = s ∪ random(O_c);
8      Shuffle(s);
9      Add s into P_0;
```

Algorithm 3: Crossover.

Input: p_1, p_2: two parent individuals.
Input: O_b: optimizations in buggy optimization sequence.
Output: o_1, o_2: two offsprings.

1 $size_1$ = $|p_1|$;
2 $size_2$ = $|p_2|$;
3 $size$ = $min(size_1, size_2)$;
4 δ = $random(1, size)$;
5 o_1 = first δ optimizations from p_1;
6 o_1 = append last $size_2 - \delta$ optimizations from p_2;
7 **if** *not* $O_b \subseteq o_1$ **then** /*validate constraint*/
8 o_1 = clone of p_1;
9 o_2 = first δ optimizations from p_2;
10 o_2 = append last $size_1 - \delta$ optimizations from p_1;
11 **if** *not* $O_b \subseteq o_2$ **then** /*validate constraint*/
12 o_2 = clone of p_2;

optimizations between two parent individuals (i.e., optimization sequences) p_1 and p_2. Given a random cut-point δ, the first offspring o_1 contains the first δ optimizations from p_1, and the last $|p_2| - \delta$ optimizations from p_2 are appended to o_1. On the other hand, the second offspring o_2 inherits the first δ optimizations from p_2 and the last $|p_1| - \delta$ optimizations from p_1. However, the standard single-point crossover may cause the generated offspring to lose some optimizations of the buggy optimization sequence, which may make the corresponding execution traces not helpful to localize optimization sequence bugs. Hence, in our study, we present a constrained single-point crossover operator for CGA, which can guarantee that the generated offspring contains all the optimizations of the buggy optimization sequence.

Algorithm 3 shows the proposed constrained single-point crossover operator. The main steps of the proposed crossover are identical to the standard single-point crossover. First, we obtain the minimal size of the two parent individuals p_1 and p_2 (lines 1–3). Then, a random cut-point δ is selected from 1 to this minimal size (line 4). Next, we recombine the optimizations from two parent individuals around the cut-point (lines 5, 6, 9, and 10). After this recombination, we verify whether the generated offspring o_1 and o_2 contain all the optimizations of the buggy optimization sequence. If an optimization of the buggy optimization sequence is lost in o_1 (or o_2), we redefine o_1 (or o_2) as a pure copy of its parent p_1 (or p_2) (lines 7, 8, 11, and 12).

3) Mutation: After the crossover, the next step of a genetic algorithm is to mutate the generated offspring. In our study, four mutation operators are applied to mutate the offsprings, namely *Delete*, *Replace*, *Swap*, and *Insert*.

1) *Delete:* This mutation indicates that an optimization in an individual is randomly deleted.
2) *Replace:* In this mutation, we randomly utilize an optimization to replace the one in an individual.
3) *Swap:* By using this mutation, we randomly swap the positions of two optimizations in an individual.
4) *Insert:* In contrast to *Delete*, this mutation aims to randomly insert an optimization into an individual.

Algorithm 4: Mutation.

Input: p: an individual.
Input: O_b: optimizations in buggy optimization sequence.
Input: O_c: candidate optimizations.
Output: mutated p.

1 $size$ = $|p|$;
2 **for** *each optimization* $o \in p$ **do**
3 **if** $random() < 1/size$ **then**
4 m = $random([1, 2, 3, 4])$;
5 **if** m = *1* **then**
6 $Delete(o)$;
7 **else if** m = *2* **then**
8 c = $random(O_c)$;
9 $Replace(o, c)$;
10 **else if** m = *3* **then**
11 o' = another optimization in p;
12 $Swap(o, o')$;
13 **else**
14 idx = index of o in p;
15 c = $random(O_c)$;
16 **if** $random() < 0.5$ **then** /*insert to left*/
17 $Insert(c, idx - 1)$;
18 **else** /*insert to right*/
19 $Insert(c, idx + 1)$;
20 **if** *not* $O_b \subseteq p$ **then** /*validate constraint*/
21 Reverse changes;

Similar to the crossover, the above mutation operators also need to guarantee that all the optimizations of the buggy optimization sequence are included in the mutated individuals. Hence, after each mutation, we verify whether the mutated individuals contain all the optimizations of the buggy optimization sequence. If the constraint is not satisfied, we reverse the changes and abandon the mutated individuals. Notably, in practice, we only need to verify the above constraint in the *Delete* and *Replace* operators since only these two operators may dissatisfy the constraint.

Algorithm 4 presents the main steps of the proposed constrained mutation for CGA. For an individual p, we iteratively mutate each optimization o with the probability $1/size$ (lines 2–21), where $size$ is the number of optimizations in p. In our study, the four mutation operators are randomly selected to mutate the current optimization o (line 4). We utilize 1, 2, 3, and 4 to represent *Delete*, *Replace*, *Swap*, and *Insert* operators, respectively. Hence, if the integer 1 (or 2, 3, 4) is selected, we conduct the *Delete* (or *Replace*, *Swap*, and *Insert*) operator for o. For the *Replace* operator, an optimization c is randomly selected from the candidate optimizations set O_c and we replace o with c (lines 8 and 9). To insert an optimization into the individual, we can place it at the left or right of o. Thus, we randomly generate a number from 0 to 1. If this number is less than 0.5, we place the candidate optimization at the left of o (lines 16 and 17); otherwise, the candidate optimization is placed at the

right of o (line 19). After the mutation of o, we verify whether the current individual contains all the optimizations in the buggy optimization sequence in line 20. If the constraint is not satisfied, we reverse changes for the individual in this iteration (line 21).

E. Buggy File Localization

After obtaining a set of bug-free optimization sequences by CGA, LocSeq localizes optimization sequence bugs by comparing the execution traces between bug-free optimization sequences and the buggy optimization sequence. Similar to DiWi [8] and RecBi [9], we also adopt the idea of SBFL in LocSeq to localize optimization bugs. Specifically, the Ochiai formula [13], [26], [27], a state-of-the-art SBFL formula, is leveraged to calculate the suspicious score of each statement in the execution trace under the buggy optimization sequence. The definition of the Ochiai formula is as follows:

$$score(s) = \frac{ef_s}{\sqrt{(ef_s + nf_s)(ef_s + ep_s)}} \tag{6}$$

where ef_s and nf_s indicate the number of the buggy optimization sequences that make the compiler execute and do not execute statement s, and ep_s represents the number of bug-free optimization sequences that make the compiler execute statement s. Following the previous studies [8], [9], we also only consider the statements executed by the compiler under the buggy optimization sequence; thus, nf_s is 0. Moreover, ef_s is 1 since there is only on buggy optimization sequence. Hence, we can simplify the Ochiai formula as:

$$score(s) = \frac{1}{\sqrt{1 + ep_s}}. \tag{7}$$

As mentioned in Section I, LocSeq aims to localize the files that contain the corresponding bug. Hence, following the prior work [8], [9], we also aggregate the suspicious score of each statement in a file as the suspicious score of this file. For a given file, its suspicious score is defined as follows:

$$SCORE(f) = \frac{\sum_{i=1}^{n_f} score(s_i)}{n_f} \tag{8}$$

where n_f is the number of executed statements in the file f. After obtaining the suspicious score of each file covered by the execution trace of the buggy optimization sequence, LocSeq ranks all these files according to their suspicious scores in descending order. The larger the suspicious score of a file, the buggier the file. Therefore, developers may quickly localize the optimization sequence bug by analyzing the files in the ranking list.

IV. Evaluation

The goal of this study is to automatically localize compiler optimization sequence bugs of LLVM. Hence, we conduct experiments on LLVM to evaluate the effectiveness of LocSeq. Specifically, our evaluation aims at addressing the following three research questions (RQs).

RQ1: How does LocSeq perform on localizing compiler optimization sequence bugs of LLVM?

This RQ investigates the effectiveness of LocSeq to localize compiler optimization sequence bugs of LLVM. In our experiments, we compare LocSeq with two state-of-the-art techniques, namely DiWi and RecBi.

RQ2: How does CGA contribute to LocSeq?

In this RQ, we investigate whether CGA is helpful for localizing compiler optimization sequence bugs of LLVM. We compare LocSeq with LocSeq$_r$ (i.e., LocSeq that uses a random method to generate bug-free optimization sequences). In addition, we also discuss the impact of the constraint for the effectiveness of LocSeq and LocSeq$_r$.

RQ3: How do the parameters of CGA impact the effectiveness of LocSeq?

To investigate the impact of the parameters of CGA for the effectiveness of LocSeq, we discuss the crossover probability and the mutation probability of CGA in this RQ.

A. Implementation

We implemented LocSeq using Python based on DEAP [40]. DEAP is a novel evolutionary computation framework and has been widely used in research work [40]. Similar to DiWi and RecBi, we also leverage Gcov [41] to collect compiler test coverage. Gcov is a widely used test coverage program from the GNU Compiler Collection [42]. For the parameters of CGA, the default values of the crossover probability PB_{cx} and the mutation probability PB_{mut} are set to be 0.5 and 0.3, respectively. Notably, we discussed the impact of PB_{cx} and PB_{mut} for the effectiveness of LocSeq in RQ3. Besides, we set the population size N to 50 as in the studies [32], [34], since the experiments are time-consuming and our computation resources are limited. For example, even though we only consider three candidate values for each of PB_{cx} and PB_{mut}, we take about 25 days to finish all experiments to investigate the impact of PB_{cx} and PB_{mut}. Regarding the parameter MAX_SIZE in Algorithm 2, we set it to be 15 since the maximal length of the buggy optimization sequence is 13 and the average length is 5.6 in our benchmark. In addition, the terminating condition is set to be 1 h limit, and each experiment is repeatedly run five times to reduce the influence of randomness as in prior work [8], [9]. In addition, we take the median results as the final results as in DiWi and RecBi.

B. Benchmark

In our study, we only use LLVM to evaluate the proposed technique LocSeq. This is because, as to our knowledge, only LLVM can allow developers to compile programs with arbitrary optimization sequences. For other mature and widely used compilers in both industry and academia, such as GCC and CompCert, the orders of optimizations are fixed.[10] Although the fixed order of optimizations may improve the development productivity and safety of compilers, it limits the capabilities of compilers to improve program performance for different requirements using different optimization sequences [5]. In contrast, LLVM has been widely used to implement many compilers and tools, since

[10]https://stackoverflow.com/questions/33117294/order-of-gcc-optimization-flags, https://github.com/AbsInt/CompCert/issues/287.

many optimizations have been implemented in LLVM and the orders of them are flexible. Our study may help developers quickly localize compiler bugs caused by optimization sequences, which may be beneficial for improving the reliability of optimizations with arbitrary orders for LLVM. This may also help to improve the correctness of different LLVM-based compilers and tools.

Specifically, we manually collected 60 LLVM bugs (the same scale as in [9]) that are caused by optimization sequences from LLVM bug repository.[11] These bugs are selected according to the following conditions: 1) the test program and the buggy optimization sequence are contained in the bug report; 2) the bug has been fixed and the fix revision has been pointed out by developers in the bug report; and 3) we can reproduce the bug in our experimental environment. These 60 optimization sequence bugs cover 31 versions of LLVM from 2016 to 2020. Although developers have provided the fix revisions for each bug in the bug reports, we also manually checked whether these revisions really fix the corresponding bugs and localized the buggy locations for each bug. In addition, the optimization sequence of each bug is minimized. That is, if we remove any optimization in the sequence, we cannot reproduce the corresponding bug. This is because the optimization sequences of many bugs have been minimized when the reporter reports those bugs [5]. To this end, we manually verified whether the optimization sequence is minimized by removing optimizations in the sequence one by one [5].

Note that, although the default optimization levels (e.g., O3) of compilers are also a kind of optimization sequences, we do not select bugs caused by these default optimization levels in this article. The reason is that many bugs caused by the default optimization levels are hard to be reproduced when we utilize the same optimizations as in these optimization levels, since many parameters of optimizations are specified for these default optimization levels in the source code.[12] To facilitate future research on compiler optimization sequence bugs, we open this benchmark[13] to the public. Specifically, each bug in the benchmark contains the following information: 1) the LLVM version for the optimization sequence bug; 2) the test program; 3) the buggy optimization sequence; and 4) the buggy location. In addition, although LLVM can process intermediate representations (i.e., IR) converted from programs written by many high-level programming languages (e.g., C, C++, and Rust), we only consider bugs that their test programs are written by C programming language in this study. This is because we hope to compare LocSeq with the prior work DiWi [8] and RecBi [9], which mainly focus on compilers for the C programming language.

C. Experiment Setup

1) Hardware: Our evaluation is conducted on an x86_64 computer running Ubuntu 18.04 Linux operating system with an Intel Core i7-6900 K CPU @ 3.20GHZ $\times$ 16 processor and 64 GB of memory.

2) Comparative Approach: In RQ1, we compared LocSeq with two state-of-the-art compiler bug localization approaches *DiWi* [8] and *RecBi* [9] to show the effectiveness of LocSeq. DiWi and RecBi localize compiler bugs by constructing a set of witness test programs. These witness test programs are generated by mutating the test program of the corresponding compiler bug. The differences between DiWi and RecBi are the mutation operators and the strategy to select them [9]. All the codes of DiWi and RecBi are obtained from their websites.[14] We only modified the steps in the code for compiling test programs and collecting code coverages according to our scenario. That is, we need to first compile the test program of an optimization sequence bug to the corresponding intermediate representations using Clang[15] and utilize the optimizer[16] of LLVM to optimize it. Besides, we only collect the coverage information of the optimizer.

To investigate whether CGA is helpful for localizing compiler optimization sequence bugs of LLVM, in RQ2, we designed three variants of LocSeq, namely, $LocSeq_r$, $LocSeq_{rwc}$, and $LocSeq_{wc}$. LocSeq$_r$ is a variant of LocSeq that randomly generates optimization sequences but still satisfies the constraint that all the optimizations in the buggy optimization sequence are included in the generated optimization sequences. $LocSeq_{wc}$ and $LocSeq_{rwc}$ are the same as LocSeq and LocSeq$_r$ respectively, but they do not always satisfy the above constraint. That is, in LocSeq$_{wc}$ and LocSeq$_{rwc}$, the optimizations in the buggy optimization sequence and the candidate optimizations are combined into a set, and they are randomly selected from this set to construct optimization sequences. By comparing LocSeq with LocSeq$_{wc}$ and LocSeq$_{rwc}$, we investigate the contribution of the constraint for LocSeq.

3) Metrics: The output of LocSeq is a ranking list of suspicious files for LLVM. Similar to prior work [8], [9], we also evaluate the effectiveness of each technique in our experiments by measuring the position of each buggy file in the ranking list. For the files that have the same suspicious scores, we use the worst ranking as the final results as in the existing work [8], [9]. Specifically, in our experiments, we utilize the following three metrics as in [8], [9], [43], and [44] to evaluate the effectiveness of each approach.

1) *Top-n:* By using this metric, we calculate the number of bugs that are successfully localized within the Top-*n* positions in the ranking list. In our study, n has four candidate values, namely, 1, 5, 10, and 20. Ideally, we hope all bugs can be located in the first file in the ranking list. Hence, a higher value of Top-*n* indicates a better performance.
2) *Mean first ranking (MFR):* This metric measures the mean of the position that the first buggy file occurs in the ranking list for each bug. A smaller value of MFR indicates that developers may localize the corresponding bug as quickly as possible.

[11] https://bugs.llvm.org/index.cgi.
[12] https://lists.llvm.org/pipermail/llvm-dev/2019-May/132308.html.
[13] https://gitee.com/teazhou/loc-seq.
[14] https://github.com/JunjieChen/DiWi, https://github.com/hao-yang9804/RecBi.
[15] https://clang.llvm.org/.
[16] https://llvm.org/docs/CommandGuide/opt.html.

TABLE III
TOP-*N*, MFR, AND MAR RESULTS OF COMPARISON EXPERIMENTS BETWEEN LOCSEQ AND COMPARATIVE APPROACHES

Approach	Num. Top-1	$\Uparrow_{Top-1(\%)}$	Num. Top-5	$\Uparrow_{Top-5(\%)}$	Num. Top-10	$\Uparrow_{Top-10(\%)}$	Num. Top-20	$\Uparrow_{Top-20(\%)}$	MFR	$\Uparrow_{MFR(\%)}$	MAR	$\Uparrow_{MAR(\%)}$
DiWi	3	366.66	22	77.27	29	68.97	40	25.93	65.37	88.81	100.34	76.01
RecBi	4	250.00	25	56.00	34	44.12	42	28.57	26.48	72.39	58.64	58.95
LocSeq$_r$	8	75.00	27	44.44	37	32.43	42	28.57	24.41	70.05	48.29	50.16
LocSeq$_{rwc}$	6	133.33	25	56.00	33	48.48	39	38.46	30.57	76.08	54.07	55.48
LocSeq$_{wc}$	11	27.27	33	18.18	40	22.50	46	17.39	13.42	45.53	35.41	32.02
LocSeq	14	-	39	-	49	-	54	-	7.31	-	24.07	-

Columns "$\Uparrow *$" indicate the improvement rate (%) of LocSeq over the comparative approaches in terms of "*" metric.

3) *Mean average ranking (MAR):* This metric measures the mean of the positions that all buggy files occur in the ranking list for each bug. Different from MFR, MAR evaluates the effectiveness of each approach to localize all buggy files precisely. Similar to MFR, a smaller value of MAR is better.

D. Answer to RQ1

To evaluate the effectiveness of the proposed technique LocSeq, in this RQ, we compare it with two state-of-the-art approaches, i.e., DiWi and RecBi. Table III shows the comparison results between LocSeq and the comparative approaches. Columns 2–9 are the results of Top-*n* metrics that are calculated based on the median of the results within five iterations for each approach; columns 10–13 are the results of MFR and MAR metrics. From Table III, it is obvious that LocSeq can localize more bugs within Top-1/5/10/20 files than the comparative approaches. Specifically, LocSeq localizes 14, 39, 49, and 54 bugs within Top-1, Top-5, Top-10, and Top-20 files, account for 23.33%, 65.00%, 81.00%, and 90.00% out of the 60 optimization sequence bugs in the benchmark, respectively. However, only 3/22/29/40 and 4/25/34/42 bugs are localized by DiWi and RecBi within Top-1/5/10/20 files, respectively. The reason may be that DiWi and RecBi can only generate a few passing test programs. In our experiments, only about 7/15 passing test programs can be generated by DiWi/RecBi on average in 1 h. However, LocSeq can construct about 945 bug-free optimization sequences in the same period. The larger number of bug-free optimization sequences may provide more evidence about an optimization sequence bug, which, thus, improves the effectiveness of LocSeq. Especially, compared to 14 bugs located by LocSeq within Top-1 files, DiWi and RecBi only locate three and four bugs within Top-1 files; LocSeq significantly outperforms DiWi and RecBi by up to 366.66% and 250.00%, respectively. Notably, in our experiments, RecBi localizes more bugs than DiWi, which also proves that RecBi outperforms DiWi as in [9].

For more clearly observing the results in our experiments, we draw the boxplot of the localized bugs within Top-1/5/10/20 files of LocSeq and the comparative approaches during the five iterations, as shown in Fig. 3. From Fig. 3, we can obviously see that LocSeq significantly outperforms DiWi and RecBi. In each scenario (i.e., Top-1/5/10/20), the number of bugs localized by LocSeq is completely larger than those of DiWi and RecBi. In the best case, LocSeq localizes 17 bugs within Top-1 files, while only four and five for DiWi and RecBi, respectively. Fig. 4 shows

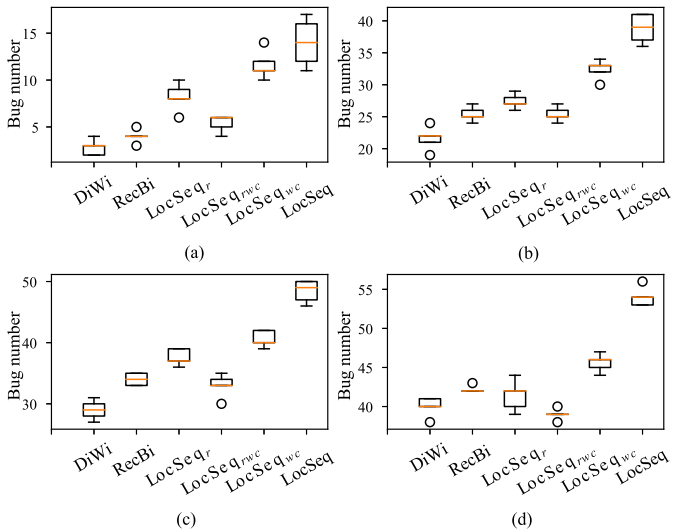

Fig. 3. Comparison of Top-*n* results between LocSeq and comparative approaches within five iterations. (a) Top-1. (b) Top-5. (c) Top-10. (d) Top-20.

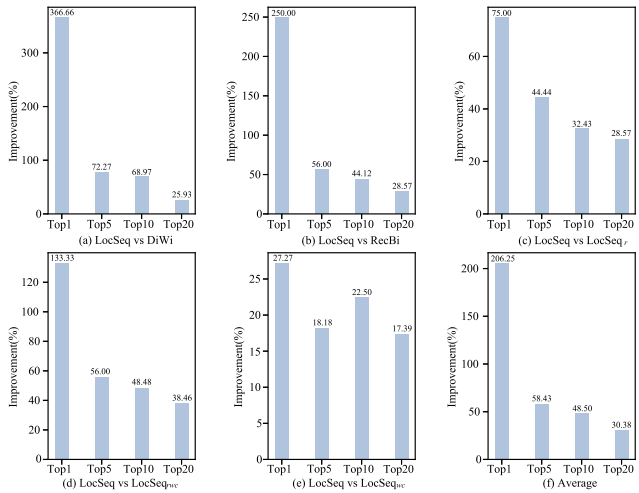

Fig. 4. Improvements of LocSeq over comparative approaches in terms of Top-*n* results.

the improvements of LocSeq over the comparative approaches. From Fig. 4(f), we can see that the improvements of LocSeq over four comparative approaches from Top-1 to Top-20 are decreased on average. For example, the average improvement of LocSeq over four comparative approaches is 206.25% in terms of Top-1, while there are only 58.43%, 48.50%, and 30.38% for

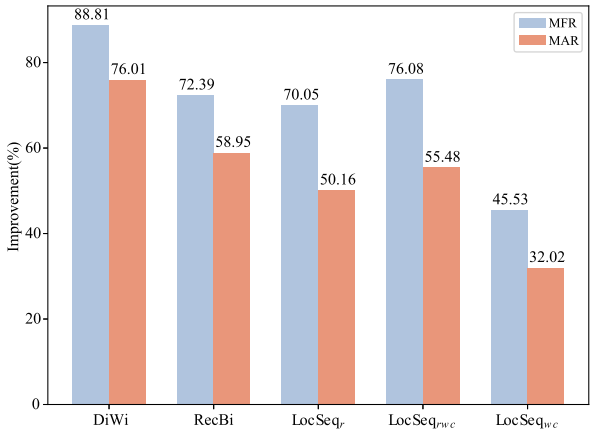

Fig. 5. Improvements of LocSeq over comparative approaches in terms of MFR/MAR.

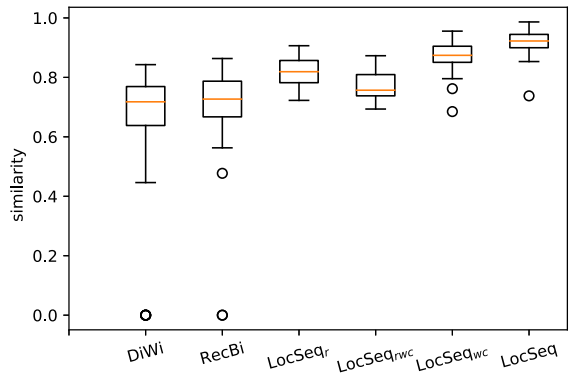

Fig. 6. Similarities between all bug-free execution traces generated by each approach and the given buggy execution trace.

TABLE IV
PEARSON CORRELATION COEFFICIENT BETWEEN THE MEAN SIMILARITIES AND THE NUMBER OF LOCALIZED BUGS IN TERMS OF TOP-1/5/10/20

	Pearson correlation coefficient	P-value
Top-1	0.980	0.0006
Top-5	0.923	0.0087
Top-10	0.919	0.0096
Top-20	0.792	0.0603

Top-5, Top-10, and Top-20, respectively. For the improvements of LocSeq over DiWi and RecBi, we can observe that the overall trend is downward from Fig. 4(a) and (b). Besides, from Fig. 4(a) and (b), we can observe that the overall trend of the improvements of LocSeq over DiWi and RecBi is also downward. For instance, the improvement of LocSeq over DiWi in terms of Top-1 is 366.66%, while it is only 250.00% for RecBi. This is because the effectiveness of RecBi is better than DiWi [9]. Besides, from Top-1 to Top-20, the improvements of LocSeq over DiWi and RecBi are also decreased. For example, the improvement of LocSeq over DiWi in terms of Top-5 is 77.27%, while it is only 25.93% in terms of Top-20. The reason is that over 60% bugs can be localized within Top-20 files by LocSeq, DiWi, and RecBi. However, in practice, most developers only focus on localizing buggy elements within Top-5 positions produced by the automated debugging tools [45]. This further reveals the advantage of LocSeq to localize optimization sequence bugs of LLVM.

For the MFR and MAR metrics in Table III, the smaller the values of MFR/MAR, the better the effectivenesses of LocSeq and the comparative approaches. From columns 10–13 in Table III, we can see that the values of MFR and MAR for LocSeq are much smaller than those of DiWi and RecBi. For example, the value of MFR for LocSeq is 7.31, while it is 65.37 for DiWi, achieving an 88.81% improvement. Fig. 5 further shows the improvements of LocSeq over comparative approaches in terms of MFR and MAR. In general, in terms of MFR and MAR, the effectiveness of LocSeq is at least 76% and 58% higher than those of DiWi and RecBi, respectively. This also proves that LocSeq is more effective than DiWi and RecBi from another perspective.

To investigate the reason why LocSeq is better than DiWi and RecBi, we compute the mean similarities between all the bug-free execution traces generated by each approach and the given buggy execution trace for each bug. Note that, for a fair comparison between LocSeq and the comparative approaches (DiWi and RecBi), we use formula (4) to calculate the similarity since DiWi and RecBi also use the same formula to calculate the similarity. Fig. 6 shows the boxplot of similarities produced by LocSeq and the comparative methods. From Fig. 6, we can see that the similarities of LocSeq are clearly larger than those of DiWi and RecBi. For example, the median of LocSeq is 0.92, while it is only 0.71 for DiWi (or 0.72 for RecBi). Especially, some similarities of DiWi and RecBi are 0 since they fail to generate any passing test programs in our experiments. This is coincident with the bug localization capability of each approach.

Besides, to further show the relationship between similarity and the bug localization capability, we calculate the Pearson correlation coefficient [46] between the mean similarities and the number of localized bugs in terms of Top-1/5/10/20 for LocSeq and the comparative methods using SciPy [47]. Table IV shows the results of the Pearson correlation coefficient. From Table IV, we can see that the Pearson correlation coefficients from Top1 to Top-20 are greater than 0.79. Especially, the Pearson correlation coefficients are 0.980, 0.923, and 0.919 for Top1, Top-5, and Top-10, respectively. The overall trend of the Pearson correlation coefficient is downward. This is because most bugs can be localized by LocSeq and the comparative methods in Top-20 files. Moreover, in Table IV, except for Top-20, the results of P-value are lower than 0.05. The results of Pearson correlation coefficients and p-value in Table IV indicate that the relationship between similarity and the bug localization capability is positive. That is, the higher similarity between bug-free execution traces and the buggy execution trace, the better it is to localize the optimization sequence bug.

Answer to RQ1: The experimental results demonstrate that LocSeq significantly outperforms the state-of-the-art approaches DiWi and RecBi. Specifically, LocSeq could localize 366.66%/77.27%/68.97%25.93% and 250.00%/56.00%/

44.12%/28.57% more bugs within Top-1/5/10/20 files than DiWi and RecBi, respectively.

E. Answer to RQ2

In this section, we conduct an experiment to investigate the contribution of CGA for LocSeq. Specifically, we compare LocSeq with its three variants, namely, LocSeq$_r$, LocSeq$_{rwc}$, and LocSeq$_{wc}$. By comparing LocSeq with LocSeq$_r$, we could confirm the contribution of CGA for LocSeq. From Table III and Figs. 3 and 4, we can see that LocSeq significantly outperforms LocSeq$_r$, LocSeq$_{rwc}$, and LocSeq$_{wc}$. LocSeq$_r$ only localizes 8, 27, 37, and 42 bugs within Top-1, Top-5, Top-10, and Top-20 files, respectively, which are obviously less than those of LocSeq. LocSeq achieves a 75.00%/44.44%/32.43%/28.57% improvement within Top-1/5/10/20 files compared with LocSeq$_r$. In addition, from Table III and Fig. 5, we can observe that LocSeq also outperforms LocSeq$_r$ by up to 70.05%/50.16% in terms of MFR/MAR. These significantly demonstrate the advantage of CGA in LocSeq to generate bug-free optimization sequences for localizing optimization sequence bugs of LLVM.

Besides, we investigate the impact of the constraint (i.e., all the optimizations in the buggy optimization sequence must be included in the generated optimization sequences) for the effectiveness of LocSeq by comparing LocSeq with LocSeq$_{rwc}$ and LocSeq$_{wc}$. Without the constraint, LocSeq$_{rwc}$ and LocSeq$_{wc}$ clearly localize fewer bugs within Top-1/5/10/20 files than LocSeq. For example, although both LocSeq and LocSeq$_{wc}$ generate bug-free optimization sequences using the genetic algorithm, LocSeq$_{wc}$ only localizes 11/33/40/46 bugs within Top-1/5/10/20 files, which is 22.27%/18.18%/22.50%/17.39% less than that of LocSeq. In Fig. 3, although most results of LocSeq are better than those of LocSeq$_r$, LocSeq$_{rwc}$, and LocSeq$_{wc}$, some Top-1 results of LocSeq$_{wc}$ are better than those of LocSeq. However, the median of LocSeq is clearly larger than that of LocSeq$_{wc}$, which illustrates that LocSeq is still more effective in most cases. From Fig. 4(c) and (d), we can see that the improvements of LocSeq over LocSeq$_{rwc}$ are larger than those of LocSeq$_r$. This is because the constraint has a positively impact to localize optimization sequence bugs of LLVM. Moreover, from Table III and Figs. 3–5, it is clear that LocSeq$_{wc}$ is more effective than LocSeq$_r$. This may be because the genetic algorithm without the constraint is more helpful than the random strategy to construct bug-free optimization sequences that share similar execution traces with the buggy optimization sequence.

From Table III and Fig. 5, we can see that LocSeq (or LocSeq$_r$) also outperforms LocSeq$_{wc}$ (or LocSeq$_{rwc}$) in terms of the MFR and MAR metrics. For instance, the MFR value of LocSeq$_{wc}$ is 13.42, which is about two times larger than that of LocSeq. The improvements of LocSeq over LocSeq$_{wc}$ are 45.53% and 32.02% in terms of the MFR and MAR metrics, respectively. This also demonstrates that the constraint in CGA is beneficial to improve the effectiveness of LocSeq.

Similar to RQ1, we also analyze the similarities between all bug-free optimization sequences generated by each approach (i.e., LocSeq$_r$, LocSeq$_{rwc}$, LocSeq$_{wc}$, and LocSeq) and the given buggy optimization sequence. From Fig. 6, we can see that LocSeq significantly outperforms LocSeq$_r$, LocSeq$_{rwc}$, and LocSeq$_{wc}$, which indicates that CGA is beneficial to construct bug-free optimization sequences that are more similar with the buggy optimization sequence. For example, the median of LocSeq$_r$ is 0.81, which is less than 0.92 for LocSeq. In addition, we can observe that without the constraint, the similarities of LocSeq$_{rwc}$ and LocSeq$_{wc}$ are smaller than those of LocSeq$_r$ and LocSeq, respectively. For instance, the median of LocSeq$_{wc}$ is 0.87, while it is 0.92 for LocSeq. This further illustrates that CGA has a positive contribution for the effectiveness of LocSeq.

Answer to RQ2: Compared with LocSeq$_r$, LocSeq$_{rwc}$, and LocSeq$_{wc}$, LocSeq can localize more optimization sequence bugs of within Top-1/5/10/20 files, which significantly reveals the positive contribution of CGA for LocSeq.

TABLE V
BUG NUMBER LOCALIZED IN TOP-*N* FILES UNDER DIFFERENT COMBINATIONS OF PB_{CX} AND PB_{MUT} FOR LOCSEQ

PB_{cx}	PB_{mut}	Num. Top-1	Num. Top-5	Num. Top-10	Num. Top-20
0.2	0.1	10	28	38	51
0.2	0.2	11	32	41	50
0.2	0.3	12	28	44	50
0.5	0.1	11	30	38	53
0.5	0.2	13	35	47	51
0.5	0.3	14	39	49	54
0.8	0.1	11	31	44	54
0.8	0.2	13	33	41	50
0.8	0.3	11	29	40	52

F. Answer to RQ3

In this RQ, we investigate the impact of two main parameters of CGA for the effectiveness of LocSeq, namely, the crossover probability PB_{cx} and the mutation probability PB_{mut} of CGA. Following some studies (see, e.g., [32], [34], and [36]) and the limitation of our computation resources, we study $PB_{cx} = 0.2$, 0.5, and 0.8, respectively; regarding PB_{mut}, we study $PB_{\text{mut}} = 0.1$, 0.2, and 0.3, respectively. Thus, we totally run 45 experiments (nine different combinations of PB_{cx} and PB_{mut}, five runs for each experiment), which take in nearly 25 days to finish all experiments.

Table V shows the number of bugs localized by LocSeq under different combinations of PB_{cx} and PB_{mut} in terms of the Top-*n* metric. From Table V, we can observe that LocSeq achieves better results under the setting $PB_{\text{cx}} = 0.5$ and $PB_{\text{mut}} = 0.3$. For example, 14/39/49/54 bugs are localized within Top-1/5/10/20 files by LocSeq when $PB_{\text{cx}} = 0.5$ and $PB_{\text{mut}} = 0.3$, while 10/28/38/51 for $PB_{\text{cx}} = 0.2$ and $PB_{\text{mut}} = 0.1$. Although the absolute differences between the number of bugs localized by LocSeq with different combinations of PB_{cx} and PB_{mut} are close, the relative improvement is significant. For example, in terms of the Top-1 metric, LocSeq with $PB_{\text{cx}} = 0.5$ and $PB_{\text{mut}} = 0.3$ outperforms LocSeq with $PB_{\text{cx}} = 0.2$ and $PB_{\text{mut}} = 0.1$ by up to 40%. In addition, from Table V, we can see that the number of bugs localized by LocSeq with $PB_{\text{cx}} = 0.2$ or $PB_{\text{cx}} = 0.8$ are smaller than those when $PB_{\text{cx}} = 0.5$, especially for the Top-1 and Top-5 results. The reason may be that a small value (0.2) of PB_{cx} makes it hard to generate new

individual, while a large value (0.8) of PB_{cx} could cause many individuals do not satisfied the constraint after the crossover operation. Besides, the results of Top-1 and Top-5 are affected more obviously by PB_{cx} and PB_{mut}. For example, LocSeq with $PB_{\text{cx}} = 0.5$ and $PB_{\text{mut}} = 0.3$ can localized more 11 bugs than LocSeq with $PB_{\text{cx}} = 0.2$ and $PB_{\text{mut}} = 0.1$ in terms of the Top-5 results, while the difference is only three bugs for the Top-20 results. As shown in the existing work [45], Top-1 and Top-5 results are more important in practice. Hence, in our study, the values of PB_{cx} and PB_{mut} are set to be 0.5 and 0.3, respectively.

Answer to RQ3: The experimental results present that the effectiveness of LocSeq is can be slightly affected by PB_{cx} and PB_{mut} of CGA. When $PB_{\text{cx}} = 0.5$ and $PB_{\text{mut}} = 0.3$, LocSeq obtains better results than other combinations of PB_{cx} and PB_{mut} in our experiments.

V. Threats to Validity

A. Threats to Internal Validity

The threats to internal validity mainly lie in the implementations of LocSeq. To reduce this threat, we implement LocSeq based on DEAP [40], which is a widely used evolutionary computation framework. In addition, we also refer to the code of the existing work (i.e., DiWi and RecBi), such as the code for collecting code coverage of LLVM, to reduce the differences between LocSeq and the existing work for a fair comparison. Besides, the efficiency to collect code coverage of LLVM may influence the effectiveness of LocSeq. In each iteration of CGA, the most time-consuming process is the collection of code coverage of LLVM for calculating the fitness function. This is because we use GCOV to obtain code coverage of each file. However, many files of LLVM need to be processed. To reduce this threat, we utilize multiprocess techniques in Python to accelerate the collection of code coverage of LLVM in our study.

B. Threats to External Validity

The threats to external validity mainly lie in the benchmark and compilers in our experiments to evaluate the effectiveness of LocSeq. On the one hand, the benchmark only includes 60 optimization sequence bugs of LLVM, which may not be sufficient to completely reveal the effectiveness of LocSeq. Thus, in the future, we will continue to collect optimization sequence bugs of LLVM to reduce this threat. On the other hand, we only evaluate LocSeq based on LLVM, since only LLVM currently supports to compile programs using arbitrary optimization sequences. However, as a mainstream compiler infrastructure, LLVM has been widely used to develop compilers and tools in both industry and academia, which demonstrates that LocSeq may be beneficial to improve the reliability of these compilers and tools. We believe that in the future, more compilers will support flexible optimization sequences; thus, LocSeq may be also suitable for localizing optimization sequence bugs for these compilers.

VI. Related Work

A. Compiler Debugging

The most related works for our study are DiWi [8] and RecBi [9]. In DiWi and RecBi, the problem of localizing compiler bugs was transformed into the problem of generating passing test programs that cannot trigger the corresponding compiler bug. Specifically, given a test program (i.e., failing test program) that triggers a compiler bug, DiWi and RecBi first leveraged mutation operators to generate a set of test programs (i.e., passing test program) that are similar to the failing test program but cannot trigger the compiler bug. Then, the spectrum-based software bug localization techniques [26], [27] are utilized to identify the compiler buggy files by comparing the execution traces between the generated passing test programs and the given failing test program. The differences between DiWi and RecBi are the mutation operators and the strategy for selecting mutation operators. DiWi focused on local mutation operators that only change minimal program elements (e.g., modifiers and constants) and leveraged a Markov chain Monte Carlo method to select mutation operators. However, the local mutation operators in DiWi may be inefficient since compiler bugs tend to occur in compiler optimizations that tend to depend on test program structure [9]. Hence, RecBi augmented the mutation operators in DiWi to include structural mutation operators that change the test program structure by inserting some control-flow-alerting statements (e.g., branch and loop statements) and utilized reinforcement learning to select mutation operators. Unlike DiWi and RecBi, our study aims to localize compiler optimization sequences bugs of LLVM by constructing a set of bug-free optimization sequences. Compared with DiWi and RecBi, the proposed technique is more lightweight and may have better generalization to the LLVM-based compilers.

Besides, to facilitate the debugging of GCC, Zeller [48] proposed a method to calculate the cause–effect chain by comparing the program states between a passing run and a failing run. This method could diagnose compiler bugs at the program-state level. Holmes and Groce [19], [49] proposed to localize compiler bugs by comparing a set of compiler mutants. However, these two methods may suffer from scalability or effectiveness problems due to the complexity of compilers. In contrast, the proposed technique in our study localizes compiler optimization sequence bugs at the source-code level and is easy to be conducted.

For other compiler debugging techniques, many studies [50]–[54] focused on providing debugging messages or visualization. For example, Ogata *et al.* [53] proposed an approach to debug the just-in-time compiler in a java virtual machine. This approach used two compilers: one was utilized to record all of the runtime information when compiling a method into a log file; then, another compiler was leveraged to replay the recorded information when debugging the compiler by developers. Clearly, these studies could provide useful information to facilitate compiler debugging, but they are limited to point out the actual location of a bug. In addition, some work [55]–[59] focused on reducing the failure-inducing test programs to help developers analyze compiler bugs. For instance, Regehr *et al.* [58] proposed C-Reduce, a tool that aims to automatically reduce a large C,

C++, or OpenCL files to a much smaller one. Sun *et al.* [59] presented a syntax-guided program reduction framework Perses. In our study, the test programs could be reduced or not since the proposed technique only needs to change the buggy optimization sequences and does not depend on the reduced test programs.

B. Compiler Autotuning

Compiler autotuning aims to select better optimizations and parameters of compilers for improving the performance of target programs. Our study is related to compiler autotuning since both studies try to manipulate compiler optimizations for some tasks. The differences are that our study aims to construct bug-free optimization sequences to localize compiler optimization sequence bugs of LLVM. In general, two kinds of techniques are proposed for compiler autotuning, namely, the search-based approaches and the machine-learning-based approaches. The search-based approaches transform the problem of compiler autotuning as an optimization problem and then resolve it utilizing evolutionary algorithms. For example, a method based on genetic algorithms was proposed by Kulkarni *et al.* [60], [61] for quickly searching effective optimization sequences. Purini and Jain [3] developed a downsampling technique for the reduction of the infinitely large optimization sequence space. Ansel *et al.*[29] developed OpenTuner, which aims to find optimal optimizations for a program using the ensembles of search techniques.

Different from the search-based approaches, the machine-learning-based approaches of compiler autotuning try to use machine learning techniques to predict better optimizations for a target program [1], [2], [4]. Ashouri *et al.* [30] conducted a survey to summarize and classify the recent advances in using machine learning for compiler autotuning. Fursin *et al.* [1] developed a machine-learning-based compiler Milepost that automatically adapts the internal optimization heuristic to improve the performance. A method based on the Markov process was proposed by Kulkarni and Cavazos [2] to select good optimization sequences to improve the performance of a program. Ashouri *et al.* [4] built a predictive model for compiler autotuning using the optimization subsequences and the machine learning technique.

Recently, researchers focused on integrating the machine learning techniques into the search-based approaches for compiler autotuning. Chen *et al.* [62] proposed BOCA, a tool based on Bayesian optimization for efficient compiler autotuning. In BOCA, a tree-based model was used to approximate the objective function for improving the scalability of Bayesian optimization. Similarly, a surrogate-assisted memetic algorithm SMARTEST was developed by Jiang *et al.* [63] to select better compiler optimizations for code size reduction. SMARTEST utilized a machine-learning-based surrogate model to avoid expensive fitness evaluation in the genetic algorithm.

However, potential compiler bugs may be introduced by compiler autotuning due to the complexity of compilers. Our work may help developers quickly localize compiler optimization sequence bugs and then fix them.

VII. Conclusion

In this article, we proposed LocSeq, a novel technique to automatically localize compiler optimization sequence bugs of LLVM. Unlike the state-of-the-art techniques (i.e., DiWi and RecBi) that localize compiler bugs by generating a set of witness test programs via mutation, LocSeq tries to localize compiler optimization sequence bugs by constructing a set of bug-free optimization sequences, which significantly improve the effectiveness for localizing compiler optimization sequence bugs. To this end, a CGA was presented in LocSeq to construct bug-free optimization sequences that share similar compiler execution trace with the buggy optimization sequence. Our evaluation based on 60 real-world optimization sequence bugs of LLVM demonstrates that LocSeq successfully localizes 23.33%/65.00% bugs within Top-1/Top-5 files of LLVM, which significantly outperforms the state-of-the-art techniques DiWi and RecBi by up to 366.66%/72.27% and 250.00%/56.00%, respectively.

For future work, we will investigate fine-grained (e.g., method-level) localization for compiler optimization sequence bugs to further improve the efficiency for compiler debugging.

References

[1] G. Fursin *et al.*, "Milepost GCC: Machine learning enabled self-tuning compiler," *Int. J. Parallel Program.*, vol. 39, no. 3, pp. 296–327, 2011.
[2] S. Kulkarni and J. Cavazos, "Mitigating the compiler optimization phase-ordering problem using machine learning," in *Proc. ACM Int. Conf. Object Oriented Program. Syst. Lang. Appl.*, 2012, pp. 147–162.
[3] S. Purini and L. Jain, "Finding good optimization sequences covering program space," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 1–23, 2013.
[4] A. H. Ashouri, A. Bignoli, G. Palermo, C. Silvano, S. Kulkarni, and J. Cavazos, "MiCOMP: Mitigating the compiler phase-ordering problem using optimization sub-sequences and machine learning," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 3, pp. 1–28, 2017.
[5] H. Jiang, Z. Zhou, Z. Ren, J. Zhang, and X. Li, "CTOS: Compiler testing for optimization sequences of LLVM," *IEEE Trans. Softw. Eng.*, to be published, doi: 10.1109/TSE.2021.3058671.
[6] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 283–294, 2011.
[7] C. Sun, V. Le, Q. Zhang, and Z. Su, "Toward understanding compiler bugs in GCC and LLVM," in *Proc. 25th Int. Symp. Softw. Testing Anal.*, 2016, pp. 294–305.
[8] J. Chen, J. Han, P. Sun, L. Zhang, D. Hao, and L. Zhang, "Compiler bug isolation via effective witness test program generation," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2019, pp. 223–234.
[9] J. Chen, H. Ma, and L. Zhang, "Enhanced compiler bug isolation via memoized search," in *Proc. 35th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2020, pp. 78–89.
[10] Z. Zhou, Z. Ren, G. Gao, and H. Jiang, "An empirical study of optimization bugs in GCC and LLVM," *J. Syst. Softw.*, vol. 174, pp. 1–13, 2021.
[11] W. E. Wong, V. Debroy, A. Surampudi, H. Kim, and M. F. Siok, "Recent catastrophic accidents: Investigating how software was responsible," in *Proc. 4th Int. Conf. Secure Softw. Integr. Rel. Improvement*, 2010, pp. 14–22.
[12] A. P. Mathur and W. E. Wong, "Comparing the fault detection effectiveness of mutation and data flow testing: An empirical study," *Softw. Qual. J.*, vol. 4, pp. 69–83, 1993.
[13] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the accuracy of spectrum-based fault localization," in *Proc. Testing: Acad. Ind. Conf. Pract. Res. Techn.*, 2007, pp. 89–98.
[14] W. E. Wong, Y. Shi, Y. Qi, and R. Golden, "Using an RBF neural network to locate program bugs," in *Proc. 19th Int. Symp. Softw. Rel. Eng.*, 2008, pp. 27–36.

[15] J. Xuan and M. Monperrus, "Test case purification for improving fault localization," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 52–63.

[16] X. Li and L. Zhang, "Transforming programs and tests in tandem for fault localization," *ACM Program. Lang.*, vol. 1, pp. 1–30, 2017.

[17] M. Papadakis and Y. Le Traon, "Metallaxis-FL: Mutation-based fault localization," *Softw. Testing, Verification Rel.*, vol. 25, nos. 5–7, pp. 605–628, 2015.

[18] L. Zhang, L. Zhang, and S. Khurshid, "Injecting mechanical faults to localize developer faults for evolving software," in *Proc. ACM SIGPLAN Int. Conf. Object Oriented Program. Syst. Lang. Appl.*, 2013, pp. 765–784.

[19] J. Holmes and A. Groce, "Using mutants to help developers distinguish and debug (compiler) faults," *Soft. Testing Verification Rel.*, vol. 30, no. 2, pp. 1–33, 2020.

[20] *LLVM Language Reference Manual*, LLVM Compiler Community, 2022. [Online]. Available: https://llvm.org/docs/LangRef.html

[21] *LLVM's Analysis and Transform Passes*, LLVM Compiler Community, 2022. [Online]. Available: https://www.llvm.org/docs/Passes.html

[22] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. 8th USENIX Conf. Oper. Syst. Des. Implementation*, 2008, pp. 209–224.

[23] P. D. Schubert, B. Hermann, and E. Bodden, "PHASAR: An interprocedural static analysis framework for C/C," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2019, pp. 393–410.

[24] Clang, "Clang: A C language family frontend for LLVM," 2022. [Online]. Available: http://clang.llvm.org/

[25] Nvidia's CUDA Compiler, 2022. [Online]. Available: https://developer.nvidia.com/cuda-llvm-compiler

[26] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Trans. Softw. Eng.*, vol. 42, no. 8, pp. 707–740, Aug. 2016.

[27] H. A. D. Souza, M. L. Chaim, and F. Kon, "Spectrum-based software fault localization: A survey of techniques, advances, and challenges," 2016, *arXiv:1607.04347*.

[28] C. Lattner, "LLVM: An infrastructure for multi-stage optimization," M.S. thesis, Dept. Comput. Sci., University of Illinois at Urbana-Champaign, Urbana, IL, USA, 2002.

[29] J. Ansel *et al.*, "OpenTuner: An extensible framework for program autotuning," in *Proc. 23rd Int. Conf. Parallel Archit. Compilation*, 2014, pp. 303–316.

[30] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A survey on compiler autotuning using machine learning," *ACM Comput. Surv.*, vol. 51, no. 5, pp. 1–42, 2018.

[31] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Trans. Softw. Eng.*, vol. 39, no. 2, pp. 276–291, Feb. 2013.

[32] M. Soltani, A. Panichella, and A. van Deursen, "A guided genetic algorithm for automated crash reproduction," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng.*, 2017, pp. 209–220.

[33] J. Xuan, Y. Gu, Z. Ren, X. Jia, and Q. Fan, "Genetic configuration sampling: Learning a sampling strategy for fault detection of configurable systems," in *Proc. Genet. Evol. Comput. Conf. Companion*, 2018, pp. 1624–1631.

[34] J. Castelein, M. Aniche, M. Soltani, A. Panichella, and A. van Deursen, "Search-based test data generation for SQL queries," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 1220–1230.

[35] B. Chen, X. Peng, Y. Liu, S. Song, J. Zheng, and W. Zhao, "Architecture-based behavioral adaptation with generated alternatives and relaxed constraints," *IEEE Trans. Serv. Comput.*, vol. 12, no. 1, pp. 73–87, Jan./Feb. 2019.

[36] Y. Yuan and W. Banzhaf, "ARJA: Automated repair of java programs via multi-objective genetic programming," *IEEE Trans. Softw. Eng.*, vol. 46, no. 10, pp. 1040–1067, Oct. 2020.

[37] S. Wang, D. Lo, L. Jiang, Lucia, and H. C. Lau, "Search-based fault localization," in *Proc. 26th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2011, pp. 556–559.

[38] M. Wen *et al.*, "Historical spectrum based fault localization," *IEEE Trans. Softw. Eng.*, vol. 47, no. 11, pp. 2348–2368, Nov. 2021.

[39] M. Zhang*et al.*, "An empirical study of boosting spectrum-based fault localization via pagerank," *IEEE Trans. Softw. Eng.*, vol. 47, no. 6, pp. 1089–1113, Jun. 2021.

[40] F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné, "DEAP: Evolutionary algorithms made easy," *J. Mach. Learn. Res.*, vol. 13, pp. 2171–2175, 2012.

[41] *GCOV*, GNU Compiler Community, 2022. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[42] *GCC*, GNU Compiler Community, 2022. [Online]. Available: https://gcc.gnu.org/

[43] T.-D. B. Le, D. Lo, C. Le Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," in *Proc. 25th Int. Symp. Softw. Testing Anal.*, 2016, pp. 177–188.

[44] J. Sohn and S. Yoo, "Fluccs: Using code and change metrics to improve fault localization," in *Proc. 26th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2017, pp. 273–283.

[45] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proc. 25th Int. Symp. Softw. Testing Anal.*, 2016, pp. 165–176.

[46] *Pearson Correlation Coefficient*, Wikipedia, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Pearson_correlation_coefficient

[47] *Fundamental Algorithms for Scientific Computing in Python*, SciPy, 2022. [Online]. Available: https://scipy.org/

[48] A. Zeller, "Isolating cause-effect chains from computer programs," in *Proc. 10th ACM SIGSOFT Symp. Found. Softw. Eng.*, 2002, pp. 1–10.

[49] J. Holmes and A. Groce, "Causal distance-metric-based assistance for debugging after compiler fuzzing," in *Proc. 29th Int. Symp. Softw. Rel. Eng.*, 2018, pp. 166–177.

[50] B.-Y. E. Chang, A. Chlipala, G. C. Necula, and R. R. Schneck, "Type-based verification of assembly language for compiler debugging," in *Proc. ACM SIGPLAN Int. Workshop Types Lang. Des. Implementation*, 2005, pp. 91–102.

[51] K. Hemmert, J. Tripp, B. Hutchings, and P. Jackson, "Source level debugger for the sea cucumber synthesizing compiler," in *Proc. 11th Annu. IEEE Symp. Field-Program. Custom Comput. Mach.*, 2003, pp. 228–237.

[52] N. Krebs and L. Schmitz, "JACCIE: A java-based compiler-compiler for generating, visualizing and debugging compiler components," *Sci. Comput. Program.*, vol. 79, pp. 101–115, 2014.

[53] K. Ogata, T. Onodera, K. Kawachiya, H. Komatsu, and T. Nakatani, "Replay compilation: Improving debuggability of a just-in-time compiler," in *Proc. 21st Annu. ACM SIGPLAN Conf. Object-Oriented Program. Syst., Lang., Appl.*, 2006, pp. 241–252.

[54] A. M. Sloane, "Debugging Eli-generated compilers with Noosa," in *Compiler Construction*. Berlin, Germany: Springer, 1999, pp. 17–31.

[55] J. M. Caron and P. A. Darnell, "Bugfind: A tool for debugging optimizing compilers," *ACM SIGPLAN Notices*, vol. 25, no. 1, pp. 17–22, 1990.

[56] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 183–200, Feb. 2002.

[57] S. Herfert, J. Patra, and M. Pradel, "Automatically reducing tree-structured test inputs," in *Proc. 32nd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2017, pp. 861–871.

[58] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for c compiler bugs," in *Proc. 33rd ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2012, pp. 335–346.

[59] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su, "Perses: Syntax-guided program reduction," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 361–371.

[60] P. A. Kulkarni, S. Hines, J. Hiser, D. B. Whalley, J. W. Davidson, and D. L. Jones, "Fast searches for effective optimization phase sequences," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2004, pp. 171–182.

[61] P. A. Kulkarni, S. R. Hines, D. B. Whalley, J. D. Hiser, J. W. Davidson, and D. L. Jones, "Fast and efficient searches for effective optimization-phase sequences," *ACM Trans. Archit. Code Optim.*, vol. 2, no. 2, pp. 165–198, 2005.

[62] J. Chen, N. Xu, C. Peiqi, and H. Zhang, "Efficient compiler autotuning via Bayesian optimization," in *Proc. 43rd Int. Conf. Softw. Eng.*, 2021, pp. 1198–1209.

[63] H. Jiang, G. Gao, Z. Ren, X. Chen, and Z. Zhou, "Smartest: A surrogate-assisted memetic algorithm for code size reduction," *IEEE Trans. Rel.*, vol. 71, no. 1, pp. 190–203, Mar. 2022.