CTOS: Compiler Testing for Optimization Sequences of LLVM

He Jiang, *Member, IEEE,* Zhide Zhou, Zhilei Ren, Jingxuan Zhang, Xiaochen Li

Abstract—Optimization sequences are often employed in compilers to improve the performance of programs, but may trigger critical compiler bugs, e.g., compiler crashes. Although many methods have been developed to automatically test compilers, no systematic work has been conducted to detect compiler bugs when applying arbitrary optimization sequences. To resolve this problem, two main challenges need to be addressed, namely the acquisition of representative optimization sequences and the selection of representative testing programs, due to the enormous number of optimization sequences and testing programs. In this study, we propose CTOS, a novel compiler testing method based on differential testing, for detecting compiler bugs caused by optimization sequences of LLVM. CTOS firstly leverages the technique Doc2Vec to transform optimization sequences into vectors to capture the information of optimizations and their orders simultaneously. Secondly, a method based on the region graph and call relationships is developed in CTOS to construct the vector representations of the testing program, such that the semantics and the structure information of programs can be captured simultaneously. Then, with the vector representations of optimization sequences and testing programs, a "centroid" based selection scheme is proposed to address the above two challenges. Finally, CTOS takes in the representative optimization sequences and testing programs as inputs, and tests each testing program with all the representative optimization sequences. If there is an output that is different from the majority of others of a given testing program, then the corresponding optimization sequence is deemed to trigger a compiler bug. Our evaluation demonstrates that CTOS significantly outperforms the baselines by up to $24.76\% \sim 50.57\%$ in terms of the bug-finding capability on average. Within seven month evaluations on LLVM, we have reported 104 valid bugs within 5 types, of which 21 have been confirmed or fixed. Most of those bugs are crash bugs (57) and wrong code bugs (24). 47 unique optimizations are identified to be faulty and 15 of them are loop related optimizations.

Index Terms—Compiler testing, Optimization sequences, LLVM, Program representation, Software testing

1 Introduction

As an important infrastructure for software development, compilers (e.g., GCC, LLVM) usually provide many optimizations to improve the performance of programs, e.g., running time, code size, and throughput. With these optimizations, many studies [1], [2], [3], [4], [5], [6] have been conducted on the compiler phase-ordering problem [7], [8], namely how to select good optimization sequences (i.e., a set of compiler optimizations in a certain order) to achieve satisfactory performance for programs. However, potential compiler bugs may be triggered when optimizing programs with some optimization sequences, which may lead to unintended application behavior and disasters, especially for safety-critical domains [9]. In the literature, Purini *et al.* [4] state that "*there are optimization sequences which crash the compiler*". Fursin *et al.* [1] and Ansel *et al.* [2] have also reported some bugs that lead compilers to crash or produce incorrect program execution when applying some optimization sequences. We present two examples as well in Section 2.1 that illustrate both a crash bug and a wrong code bug of LLVM caused by certain optimization sequences.

Although some methods [9], [10], [11], [12], [13], [14], [15], [16], [17], [18] have been proposed to automatically test compilers in recent years, no systematic investigation has been conducted on compiler bugs caused by optimization sequences. To date, these existing methods to test compilers can be roughly divided into three types, namely, Randomized Differential Testing (RDT) [10], [11], [12], [13], [14], Different Optimization Levels (DOL, a variant of RDT), and Equivalence Modulo Inputs (EMI) [9], [15], [16], [17], [18], [19], [20]. Given a program, RDT detects compiler bugs by comparing the outputs of distinct compilers implemented based on the same specification. In contrast, DOL and EMI only require one compiler. DOL determines whether a compiler contains bugs by comparing the outputs of a program optimized by predefined compiler optimization sequences (e.g., sequences represented by compiler flags O1, O2 and O3), while EMI compares the outputs of some equivalent variants of a seed program [19]. However, these existing methods only focus on detecting bugs related to predefined optimization sequences in certain orders, rather than arbitrary optimization sequences.

In this paper, we investigate how to detect compiler bugs caused by optimization sequences for LLVM [21]. LLVM is a mature and widely used compiler infrastructure. Hundreds of analysis and transformation optimizations have been implemented in LLVM [22]. Moreover, many compilers (e.g., Clang [23], Rust [24], Swift [25], and WebAssembly [26]) of different programming languages and tools (e.g.,

- *H. Jiang, Z. Zhou, Z. Ren are with the School of Software, Dalian University of Technology, Dalian, China, and Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province. H. Jiang is also with DUT Artificial Intelligence Institute, Dalian, China. E-mail: jianghe@dlut.edu.cn (corresponding email), cszide@gmail.com, zren@dlut.edu.cn*
- *J. Zhang is with the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China. E-mail: jxzhang@nuaa.edu.cn*
- *X. Li is with the SnT Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg. E-mail: xiaochen.li@uni.lu*

Klee [27], Phasar [28]) have been implemented based on LLVM. In contrast to DOL and EMI, two challenges need to be addressed. On the one hand, the number of potential optimization sequences is extremely huge, such that it is intractable to test each optimization sequence. For example, more than 150 optimizations exist in LLVM, and there are 150^{30} optimization sequences[1] to be tested, when the sequence length is set to be 30. Additionally, similar optimization sequences may trigger duplicate bugs for a given testing program. Hence, we need to address the challenge of **the acquisition of representative optimization sequences**, namely, how to acquire a set of representative optimization sequences to accelerate the testing of LLVM. On the other hand, there is an almost infinite number of testing programs which may incur compiler bugs, and it is impossible to test them all. Moreover, distinct testing programs may incur duplicate compiler bugs. Thus, we need to address the second challenge, namely **the selection of representative testing programs**.

In this paper, we propose a novel compiler testing method based on differential testing, called Compiler Testing for Optimization Sequences (CTOS), for detecting bugs caused by optimization sequences for LLVM. CTOS first leverages the representation technique Doc2Vec [29] in natural language processing to transform optimization sequences into vectors on the basis of its optimizations and their orders. Then, a method based on the region graph and call relationships of programs is presented to construct the vector representations of testing programs. Next, with the vector representations of optimization sequences and testing programs, we present a centroid based selection scheme to select representative optimization sequences and testing programs, thus addressing the above two challenges. Finally, CTOS utilizes differential testing to validate all representative optimization sequences and testing programs. Specifically, CTOS takes in representative optimization sequences and testing programs as inputs and tests each testing program with all the representative optimization sequences. If there is an output that is different from the majority of others of a given testing program, then the corresponding optimization sequence is deemed to trigger a compiler bug.

To evaluate the effectiveness of CTOS, we conduct a comparative experiment that compares CTOS with 14 baselines. Our evaluation demonstrates that CTOS significantly outperforms the baselines by detecting $24.76\% \sim 50.57\%$ more bugs on average. In addition, we conduct an experiment for seven months to show the bug-finding capability of CTOS in practice. After running CTOS on LLVM for seven months, we have reported 104 valid bugs within 5 types (see Section 4), of which 21 have been confirmed or fixed. Most of those bugs are crash bugs (57) and wrong code bugs (24). 47 unique optimizations have been identified to be faulty and 15 of them are loop related optimizations.

In summary, the main contributions of this paper are as follows:

1) In this paper, we first investigate how to detect compiler bugs caused by optimization sequences. As to the best of our knowledge, this is the first systematic work for this problem.
2) We present a novel method CTOS to find compiler bugs by optimization sequences. In CTOS, efficient vector representation methods and a selection scheme are designed to address two challenges, namely the acquisition of representative optimization sequences and the selection of representative testing programs.
3) Extensive testing efforts have been conducted on LLVM. We have reported 104 valid bugs within 5 types, of which 21 have been confirmed or fixed. These bugs cover 47 unique optimizations, where 15 of them are loop related optimizations.

```
1 int c;
2 struct m{
3   unsigned:20;
4   signed a;
5   signed b
6 };
7 struct m d() {}
8 e() {
9   for (; c; c++)
10  d();
11 }
```

(a) LLVM Bug#41294

```
1 #include<stdio.h>
2 int a = 0;
3 int d() {
4   int e = 2;
5   for (a = 0; a <= 8; a++);
6   return e;
7 }
8 void main() {
9   int f = 0;
10  d();
11  printf("%d\n",a);
12 }
```

(b) LLVM Bug#41720

Fig. 1. Programs in crash bug 41294 and wrong code bug 41720 of LLVM (trunk 355281). (https://bugs.llvm.org/show_bug.cgi?id=41294, https://bugs.llvm.org/show_bug.cgi?id=41720.)

The remainder of this paper is organized as follows. Section 2 shows the background of our study. Section 3 presents the testing process and the proposed methods. Next, we describe the evaluation results in Section 4. Section 5 presents some discussions about this paper. The threats to validity and related work are described in Sections 6-7. Section 8 concludes our study.

2 Background

2.1 Illustrative Examples

In this subsection, we present two concrete examples to illustrate the compiler bugs in LLVM caused by optimization sequences. Note that these examples in this section only aim to show the phenomenon of the compiler bugs introduced by optimization sequences.

Fig. 1(a) shows a program that triggers a crash bug by the *loop-vectorizer*[2] optimization when the program is optimized by the optimization sequence "*-functionattrs -loop-rotate -licm -sroa -loop-vectorize*." An assertion fails when *loop-vectorizer* works on the Intermediate Representation (IR) file optimized by "*-functionattrs -loop-rotate -licm -sroa*." This bug occurs because the loop invariant operands are scalarized, but they are used outside the loop and should be ignored when computing the scalarization overhead[3]. However, when we delete any optimization from {*functionattrs*, *loop-rotate*, *licm*, *sroa*}, or change the order, or only use *loop-vectorizer* to optimize the program, this bug disappears. In

1. It is valid for an optimization to appear multiple times in an optimization sequence.

2. https://llvm.org/doxygen/LoopVectorize_8cpp_source.html.
3. https://reviews.llvm.org/D59995.

addition, there does not exist any bug when the program is optimized by the default standard optimization levels, e.g., O1, O2, and O3 provided by LLVM.

The second program in Fig. 1(b) incurs a wrong code bug of LLVM when applying the optimization sequence "*-gvn -licm -loop-rotate -loop-vectorize*". Variable *a* is a global variable and is initialized to 0. If the program executes correctly, the output should be 9. However, after optimization, the output is 15, which is caused by incorrectly optimizing the calculation of variable *a* on line 5 with *loop-vectorizer*. Similar to the first example, this bug cannot be reproduced when we remove any optimization in the optimization sequence, or change the order of them, or use the default standard optimization levels.

From the above two examples, it is evident that optimization sequences may heavily affect the behavior of LLVM and potential bugs of LLVM may be exposed by employing certain optimization sequences to optimize programs. This is important for ensuring the correctness of the behavior of a program when developers tune the compiler optimizations for specific programs to achieve better performance (e.g., size or speed), especially for safety-critical programs.

Compiler testing, currently, is the dominant technique for detecting compiler bugs due to its simplicity and easy application. However, there does not exist any work that focuses on finding compiler bugs caused by optimization sequences. Therefore, this motivates us to detect bugs of LLVM by testing it with arbitrary optimization sequences. To achieve this goal, we need to address two challenges, namely the acquisition of representative optimization sequences and the selection of representative testing programs, due to the enormous number of optimization sequences and testing programs.

2.2 Doc2Vec

Doc2Vec is a fundamental component to represent the optimization sequences and testing programs in our study. It was originally designed to transform documents (sentences or paragraphs) into low-dimensional vectors [29]. Doc2Vec is an extension to Word2Vec [30] to extend the learning of embeddings from words to word sequences. Similar to Word2Vec that has two models, i.e., Continuous Bag-Of-Words model (CBOW) and continuous Skip-gram model (Skip-gram), there are also two approaches within Doc2Vec, namely, Distributed Bag-Of-Words version of Paragraph Vector (DBOW) and Distributed Memory model of Paragraph Vectors (DMPV) [29]. DBOW and DMPV work in a similar way as Skip-gram and CBOW, respectively. We take the DMPV model as an example to explain Doc2Vec as it is the default model in the tool that we use to implement our methods.

Fig. 2 shows the framework of DMPV, which is similar to the framework of CBOW. In Fig. 2, the top half of the figure is the framework of CBOW. The only change for the DMPV model is the additional document token [29]. DMPV consists of an input layer, an output layer, and a hidden layer. The hidden layer h is a $1 \times V$ vector to represent words in a low dimensional space. Each column in the matrix $W_{Voc \times V}$ is a unique vector representation of a word; and the

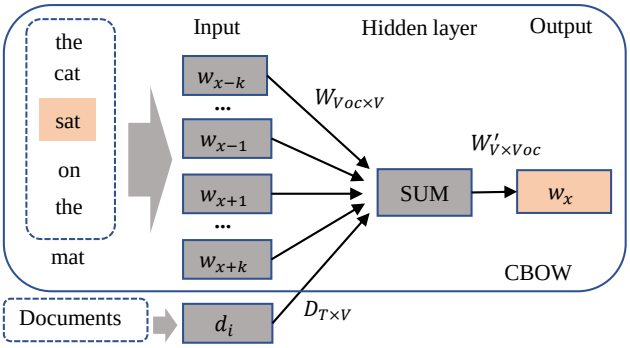

Fig. 2. The framework of DMPV model for Doc2Vec.

matrix $W'_{V \times Voc}$ is the parameter matrix of h; each column in the matrix $D_{T \times V}$ is the unique vector representation of a document. V is pre-defined by users. T is the documents and Voc is the vocabulary of the training set.

Initially, the matrix $W_{Voc \times V}$, $W'_{V \times Voc}$, and $D_{T \times V}$ are randomly initialized. Each word w_x in Voc is represented as a one-hot vector with the size of $|Voc|$, which is a zero vector with the exception of a single 1 to uniquely identify the word. The document d_i in T is represented in a similar way to the word.

Given a document d_i and its word sequences, DMPV tries to predict the center word with its surrounding context in a fixed window size k by using the vector representation of a document as the context information [29]. Specifically, DMPV takes in the vectors of the surrounding words $w_x^k = \{w_{x-k}, \cdots, w_{x-1}, w_{x+1}, \cdots, w_{x+k}\}$ in a $2k$ sized window and the vector of the corresponding document d_i as inputs. Then the vector of the center word w_x is the expected output. For example, if we want to obtain the vector of word '*sat*' and $k = 2$, the surrounding words are '*the*', '*cat*', '*on*', and '*the*'. Based on $W_{Voc \times V}$ and $D_{T \times V}$, the inputs are propagated to the hidden layer

$$h = \frac{1}{2k+1}(sum(w_x^k) \cdot W_{Voc \times V} + d_i \cdot D_{T \times V}).$$

Then, the prediction of w_x is typically done via softmax function:

$$w_x = Softmax(h \cdot W'_{V \times Voc}).$$

Finally, the objective of the DMPV is to maximize the average log probability

$$\frac{1}{|Voc|} \sum_{x=1}^{|Voc|} \log p(w_x | w_x^k).$$

After training, the column vectors in $W_{Voc \times V}$ and $D_{T \times V}$ are the final vector representation of the words and documents, respectively.

Unlike Word2Vec that only learns the vector representations of words, Doc2Vec takes word order into consideration and can learn the vector representations of word sequences. Thus, we adopt Doc2Vec to represent the optimization sequences and the instruction sequences in our study.

In the next section, we present CTOS, a method based on differential testing [31] for catching compiler bugs caused by optimization sequences of LLVM. Based on Doc2Vec, we present efficient methods to resolve the challenges of the

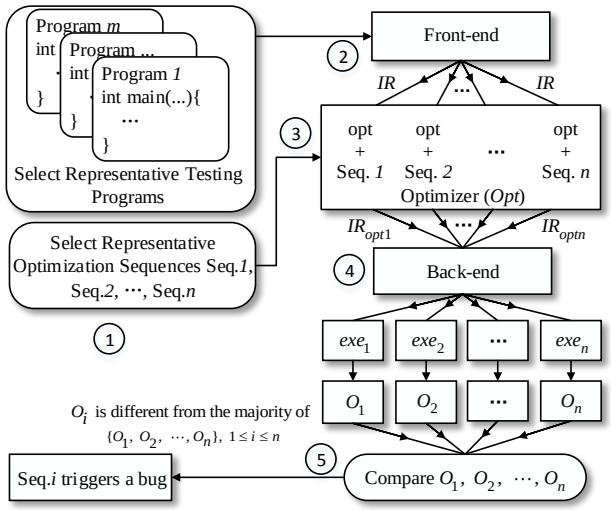

Fig. 3. Framework of CTOS for catching compiler bugs caused by optimization sequences of LLVM.

acquisition of representative optimization sequences and the selection of representative testing programs.

3 Approach

In this section, we first introduce the framework of CTOS. Then, we present algorithms to address two challenges, namely the acquisition of representative optimization sequences and the selection of representative testing programs.

3.1 Overview of CTOS

Fig. 3 shows the framework of CTOS. Generally, CTOS is based on differential testing [31]. However, unlike other studies (e.g., [10], [11], [13], [14]) that compare the outputs of testing programs compiled by different compilers or optimized by the different default optimization levels, CTOS determines whether there are compiler bugs by comparing the outputs of testing programs optimized by distinct optimization sequences. In Fig. 3, the front-end and the back-end indicate the compiler front-end and the compiler back-end that are used to transform the source code of a testing program into its LLVM IR and generate executables, respectively. We adopt Clang [23] as the front-end, a widely used language front-end and compiler driver based on LLVM for C language family (e.g., C and C++), since C programs are used as the testing programs in our study. As the front-end, Clang takes in the source code and outputs the corresponding IR of a program. For the back-end, many tools are utilized to generate the final executables from IRs, such as assembler and linker. For simplicity, we also use Clang as a driver to schedule and execute the tools (e.g., linker) of back-end, and it can take in LLVM IRs to generate executables. The optimizer *Opt* of LLVM is in charge of scheduling and executing optimizations.

CTOS is composed of 5 steps. (1) The first step is to select representative optimization sequences and testing programs. (2) Then, the front-end of a compiler is used to emit the IR file of a given testing program without optimizations. (3) The third step is to optimize the IR produced in the previous step using the optimizer Opt of LLVM with the selected optimization sequences. For the IR of a testing program, if there are n selected optimization sequences, n optimized IRs (i.e., IR_{opt1}, IR_{opt2}, $\cdots$, IR_{optn}) will be produced by the optimizer Opt. (4) In the fourth step, the n optimized IRs are loaded by the back-end of a compiler to generate n executables (i.e., exe_1, exe_2, $\cdots$, exe_n). (5) The final step is to obtain the outputs (i.e., O_1, O_2, $\cdots$, O_n) of n executables and compare them to determine whether there are bugs. The outputs may be different, but the majority of them should be identical. Thus, if there is an output O_i that is different from the majority of $\{O_1, O_2, \cdots, O_n\}$, $1 \leq i \leq n$, then the ith optimization sequence is deemed to trigger a compiler bug for the given testing program.

From Fig. 3, the first step clearly is the foundation of CTOS. Generally, we can randomly generate optimization sequences and testing programs. However, this random strategy may not be efficient due to the huge space of optimization sequences and testing programs. Meanwhile, many duplicate bugs may be triggered by similar optimization sequences and testing programs. Thus, we need to select representative optimization sequences and testing programs.

In Section 3.2 and 3.3, we first introduce the vector representations of optimization sequences and testing programs, respectively. Some optimizations in a certain order constitute an optimization sequence. Thus, if optimization sequences have similar optimizations in a similar order, they may trigger duplicate compiler bugs. In addition, the semantics and structure information of a testing program is the key to distinguish different testing programs. Thus, if testing programs have similar semantics and structure information, they may also incur duplicate compiler bugs. Therefore, a Doc2Vec based method is introduced to transform an optimization sequence into a vector, which captures optimizations and their orders of the corresponding optimization sequence simultaneously; a method based on the region graph and call relationships of a program is proposed to represent testing programs as vectors, such that the semantics and structure information of a program can be captured by vectors. With the vector representations of optimization sequences and testing programs, we assume that similar optimization sequences and testing programs are close to each other in their corresponding vector spaces, respectively. Hence, we present a centroid based selection scheme to select representative optimization sequences and testing programs in Section 3.4, such that the distances among the selected representative optimization sequences and testing programs are maximized, respectively.

3.2 Representation of Optimization Sequences

An optimization sequence is constituted of some optimizations in a certain order. Thus, the representation of an optimization sequence should reflect the specific optimizations and their orders contained in the sequence. Intuitively, an optimization sequence is similar to a sentence in natural language, which consists of some words in a certain order. Hence, in this study, we treat optimization sequences as sentences, such that efficient representation methods of sentences can be adopted to transform optimization sequences

into vectors. However, many state-of-the-art representation methods of sentences, such as the bag-of-words [32], cannot reflect the word order. They fail to distinguish different sentences with the same words. For capturing optimizations and their orders of an optimization sequence simultaneously, we employ Doc2Vec [29], a popular and widely used sentence vector representation technique to represent optimization sequences as vectors. Doc2Vec is an unsupervised method for learning continuous distributed vector representations of sentences or documents. Doc2Vec takes word orders into consideration such that the sequences with different orders of the same words have different vector representations. In addition, Doc2Vec can be applied to variable-length word sequences, so variable-length optimization sequences can be easily transformed into vector representations.

In this study, Doc2Vec is applied in a relatively straightforward way. That is, optimizations and optimization sequences are viewed as words and sentences, respectively. We leverage the DMPV model (see Section 2.2) of Doc2Vec as the representation method of optimization sequences. Then we input optimizations and optimization sequences into the DMPV model of Doc2Vec to obtain the vector representations of optimization sequences.

For example, if we only take five optimizations in LLVM into consideration, i.e., {*-functionattrs*, *-gvn*, *-loop-rotate*, *-loop-vectorize*, *-sroa*}, and set the max length of optimization sequences to 5, we can obtain $5^1+5^2+5^3+5^4+5^5 = 3905$ optimization sequences. Take the following three optimization sequences as an example: (a) *"-functionattrs -loop-rotate -sroa -gvn -loop-vectorize"*; (b) *"-functionattrs -sroa -loop-vectorize -loop-rotate -gvn"*; (c) *"-loop-rotate -sroa -gvn -loop-vectorize"*. If we can only test two sequences among them due to the limitation of resources (e.g., time), testing sequences (a) and (b) may uncover more bugs, since sequence (c) is a subsequence of sequence (a) only without the optimization *"-functionattrs"*. However, in this case, the order of optimizations is hard to be captured by some bag-of-words methods. For instance, we can calculate the similarity between optimization sequences utilizing the Jaccard similarity coefficient[4], 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., $J(A,B) = |A \cap B|/|A \cup B|$. For the sequences (a) and (b), they have the same optimizations, i.e., $J(a,b) = 1$; while $J(a,c) = 4/5$ for the sequences (a) and (c). It indicates that sequences (a) and (b) are identical, and sequences (a) and (c) should be tested. This contrasts with the observation, since the order of optimizations in the sequence (a) is completely different from that in the sequence (b). By using Doc2Vec, we can resolve the difficulty, which captures optimizations and their orders of optimization sequences simultaneously.

3.3 Representation of Testing Programs

Testing programs are another critical factor to trigger compiler bugs caused by optimization sequences. Different testing programs may trigger different bugs. Thus, we need to select representative testing programs to improve the test efficiency for finding more distinct bugs. Our study focuses on finding compiler bugs caused by optimization sequences of LLVM, which makes us decide to construct vector representations of testing programs using LLVM IR. LLVM IR is a light-weight and low-level while expressive, typed, and extensible representation of programs [21]. In this subsection, we present a vector representation method based on the region graph and call relationships generated from the unoptimized IR to transform a testing program into a vector. With this approach, we can capture the semantics and structure information of programs, which are useful for selecting representative testing programs. We divide the vector representation of a testing program into two parts, namely, the representation of a function and the representation of the whole program. Firstly, we transform the instruction sequences of each edge in the region graph of a function into vectors using the Doc2Vec technique; then a deep region-first algorithm is employed to aggregate vectors of each edge under two constraints to construct the vector representation of a function. Secondly, after obtaining the vector representations of all functions, we aggregate them according to their call relationships to form the vector representation of the whole program.

3.3.1 *Representation of a function*

A function consists of basic blocks, branches, and loops. Basic blocks contain the basic semantics of a function, while branches and loops control the structure of a function [33]. Thus, we use the region graph [34], [35] of a function to construct its vector representation, since the region graph could simultaneously capture the semantics and structure information of a function [34], [35]. Definition 1 shows the general definition of a region graph.

Definition 1. *A region graph is a special control flow graph, in which each node (i.e., basic block) exactly belongs to a region. Specifically, a region is a connected subgraph of the control flow graph that has exactly two connections to the residual graph [34], [35], [36].*

In a region graph, each node (i.e., basic block) exactly belongs to a region. Fig. 4 shows an example of the region graph. This graph is derived from a simple bubble sort algorithm using the tool *Opt* in LLVM. We remove the contents (i.e., statement sequences) of some basic blocks for simplicity. Clearly, there are 10 basic blocks, 12 edges, and 4 regions colored by four colors in Fig. 4. The number next to each edge is its index. In these basic blocks, blocks "%11" and "%15" are entry nodes of the outer loop and the inner loop respectively, and block "%19" is the entry node of a branch. From Fig. 4, the structure information (e.g., the outer loop, the inner loop, and the branch) of the program are clearly captured by each region.

To obtain the semantics of a function, we use the instruction sequences of each basic block [21] to represent the behavior of a function. This is because the proposed representation method of a program is based on LLVM IR and each instruction has precise and fine-grained semantics. In addition, the order of instructions significantly impacts the semantics of a function. For example, we can easily know that the behavior of the basic block "%11" is to load a variable and compare it with 0 through the instructions "*load*" and "*cmp*". However, we do not simply translate all the instruction sequences of each basic block to vectors and

4. https://en.wikipedia.org/wiki/Jaccard_index.

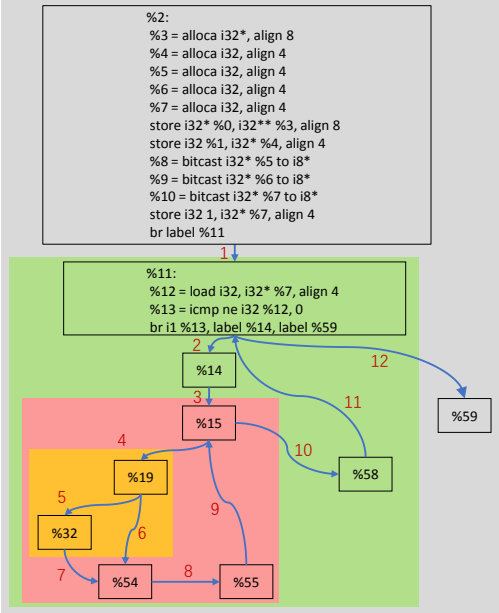

Fig. 4. Example of a region graph.

aggregate them (e.g., sum or average), since this will lose the structure information of a function. For instance, the block "%58" has no influence on the block "%19". Simply aggregating these vectors diminishes the importance of the blocks introduced by the structure of a function. Therefore, in this study, we present a deep region-first algorithm to aggregate the information on each edge of the region graph.

Specifically, the information of an edge is the concatenation of the instruction sequences from its start node (i.e., basic block) to the end node. The reason for this is that if the program executes from the start node to the end node through this edge, the code in these two nodes will be executed. For instance, the information of the first edge in Fig. 4 is "*alloca alloca alloca alloca alloca store store bitcast bitcast bitcast store br load icmp br*". After obtaining the information of each edge, we also leverage Doc2Vec to transform the information into vectors. The reason for this is that the instructions and instruction sequences can also be treated as words and sentences in natural language, respectively. In addition, the instruction sequences are also order sensitive like the optimization sequences.

Then, the deep region-first algorithm is utilized to aggregate the information of each edge to form the final vector representation of a function. Algorithm 1 presents the details of the proposed aggregation algorithm. Given a region graph of a function with vector representations of edges, we first recognize the entry node ($entry\_node$), the exit node ($exit\_node$), and the loop entry nodes ($loop\_entry\_nodes$) of the region graph in lines 2 to 4. Then all outgoing edges of $entry\_node$ are marked as visited in line 7, since the $entry\_node$ does not have incoming edges. The work list ($work\_list$) initialized by $entry\_node$ in line 9 is used to store the candidate nodes. Next, from line 10 to 35, the vectors of incoming edges of a node are aggregated to its outgoing edges until $work\_list$ is empty.

In the aggregation process, two constraints need to be satisfied in the proposed aggregation algorithm. The first

Algorithm 1: Vector representation of a function.

Input: Reg: region graph of a function with vector representation of each edge.

```
1  /*edge.vector: vector representation of an edge*/
2  entry_node = entry node of function in Reg;
3  exit_node = exit node of function in Reg;
4  loop_entry_nodes = entry nodes of loops in Reg;
5  visited_nodes = list to store visited nodes;
6  visited_edgs = list to store visited edges;
7  add all outgoing edges of entry_node into visited_edgs;
8  /*work list initialized with entry_node*/
9  work_list = [entry_node];
10 while work_list ≠ φ do
11     cur_node = node in work_list with the deepest region;
12     add cur_node into visited_nodes;
13     delete cur_node from work_list;
14     succ_nodes = successor nodes of cur_node in Reg;
15     for each node ∈ succ_nodes do
16         out_edges = all outgoing edges of node in Reg;
17         if node ∉ loop_entry_nodes then
18             in_edges = all incoming edges of node in Reg;
19             if in_edges ⊆ visited_edges then
20                 temp_vec = sum of vectors of in_edges;
21                 num = number of edges in in_edges;
22                 for edge ∈ out_edges do
23                     edge.vector = edge.vector + temp_vec;
24                     edge.vector = edge.vector/(num + 1);
25                     add edge into visited_edgs;
26                 if node ∉ (visited_nodes ∪ work_list) then
27                     add node into work_list;
28         else
29             in_edge = edge from cur_node to node;
30             for edge ∈ out_edges do
31                 edge.vector = edge.vector + in_edge.vector;
32                 edge.vector = edge.vector/2;
33                 add edge into visited_edgs;
34             if node ∉ (visited_nodes ∪ work_list) then
35                 add node into work_list;
36 return vector representation of edge ended with exit_node;
```

constraint (line 11) is called a deep region-first constraint. It means the node in the innermost region will be first selected to aggregate the vectors of incoming edges to the outgoing edges. The reason is that the semantics of the outer region is based on the inner region. In Fig. 4, after propagating information to edges 2 and 12, nodes "%14" and "%59" are two candidate nodes. However, the region of node "%14" is contained in the region of node "%59", i.e., the previous region is "deeper" than the later one, thus node "%14" is selected. When the current node (cur_node) within the deepest region in $work\_list$ is selected, we delete it from $work\_list$ and add it into the list $visited\_nodes$ that stores the visited nodes (line 12 and 13).

Next, the second constraint is employed to aggregate the vectors of incoming edges and outgoing edges of each successor node of cur_node in line 15 to 35. The second constraint indicates that, for a node with predecessors and successors in a region graph, such as node "%54" in Fig. 4, if the information on its incoming edges can be propagated to the outgoing edges, the information before its predecessors

must have been propagated to the incoming edges. This constraint is valid by line 19, i.e., all incoming edges of a node must have been visited. For example, if we want to propagate the information on the incoming edges 6 and 7 to the outgoing edge 8, the information on edges 5 and 4 must have been propagated to edges 6 and 7, respectively. In our study, to aggregate the information of incoming edges and outgoing edges, we average their sum and assign it to the outgoing edges (line 22 to 25). Then the successor node of cur_node is added into $work\_list$ if it does not exist in $work\_list$ and $visited\_nodes$ (line 25 and 26).

However, for the loop entry node, the second constraint must be relaxed due to the back edge of a loop. For instance, edge 9 is a back edge and it is also an incoming edge of the loop entry node "%15". We cannot propagate any information to the outgoing edges of node "%15", since the information before node "%55" has not been propagated to edge 9 under the second constraint. Thus, we propagate the information on the incoming edges of the loop entry node separately. That is, if the information has been propagated to edge 3, we will propagate it to edges 4 and 10. Similarly, the information on edge 9 will be propagated to edges 4 and 10. Thus, in line 17, we verify that whether a node belongs to $loop\_entry\_nodes$. Lines 29 to 35 in Algorithm 1 are the corresponding pseudocode for processing the loop entry node.

Therefore, starting from the entry node (e.g., "%2") of a region graph, when all nodes have been processed, the vector representation of the edge related to the $exit\_node$ (e.g., "%59") is treated as the vector representation of a function. In particular, if there is only one basic block in the region graph, the vector representation of the instruction sequence in this basic block is treated as the vector representation of a function.

Algorithm 2: Vector representation of the whole program.

Input: CG: call graph of a program with vector representation of each function.

```
1  /*node.vector: vector representation of a function*/
2  delete edges of recursive calls in CG;
3  start_nodes = functions without predecessors in CG;
4  end_nodes = functions without successors in CG;
5  /*vector representation of node "start" is 0*/
6  add a node with name "start" into CG;
7  add edges from "start" to node ∈ start_nodes;
8  visited_nodes = list to store visited nodes;
9  add node ∈ end_nodes into visited_nodes;
10 /*work list initialized with end_nodes*/
11 work_list = [end_nodes];
12 while work_list ≠ φ do
13     cur_node = pick up a node in work_list;
14     pre_nodes = predecessor nodes of cur_node in CG;
15     for node ∈ pre_nodes do
16         if node ∉ visited_nodes then
17             suc_nodes = successor nodes of node in CG;
18             if suc_nodes ⊆ visited_nodes then
19                 temp_vec = sum of vectors of succ_nodes;
20                 node.vector = node.vector + temp_vec;
21                 num = number of nodes in succ_nodes;
22                 node.vector = node.vector/(num + 1);
23                 add node into visited_nodes;
24                 add node into work_list;
25     delete cur_node from work_list;
26 return vector representation of node "start";
```

3.3.2 Representation of the whole program

Generally, a program may contain many functions with calling relationships. In our study, we construct the vector representation of a program based on its call graph. Algorithm 2 presents the procedure to generate the vector representation of a program. Given a call graph with the vector representation of each function of a program, the key idea of Algorithm 2 is to propagate the vectors of callees to their callers. This is because the callee should have different weights due to the call relationships. For example, in Fig. 5, the function "*func6*" is only called by functions "*func1*" and "*func2*". It has no any direct impact on other functions. Thus, if we only simply aggregate (e.g., sum or average) all the vectors of functions, we lose the impact of the weight of each function. However, one constraint must be satisfied in the propagation process. That is, all vectors of callees of a caller must be the final results, i.e., there is not any vector that needs to be propagated to these callees. Specifically, the vectors of end_nodes are their final results that have been marked in line 9.

In the algorithm, we first remove edges of recursive calls in line 2, such that the call graph is a directed acyclic graph. On the one hand, the recursive calls make it impossible for the algorithm to decide which nodes have the final result. For example, there is an indirect recursive call between "*func1*" and "*func4*" in Fig. 5. According to the constraint, if we want to aggregate the vector of "*func4*" to "*func1*", the vector representation of "*func4*" should be the final results. However, at this moment "*func1*" is the successor of "*func4*", it does not have the final vector representation. On the other hand, the recursive calls have almost no effect on improving the representation of functions. For instance, "*func7*" recursively calls itself. Its vector representation does not change after the aggregation.

Then, the start nodes ($start\_nodes$) and the end nodes (end_nodes) in the call graph are recognized in lines 3 and 4. $start\_nodes$ are the nodes without predecessors, while end_nodes are the nodes without successors in the call graph. We add a node "start" (i.e., a fake function with the name "start") to the call graph with a zero vector as its representation. Next, we add edges from the node "start" to $start\_nodes$. The reason is that there may exist some functions (besides the "*main*" function) which are not called by any other functions. Notably, the function without any caller can also be split into an independent program, but in our study, we treat the functions in a program generated by Csmith in a uniform way for simplifying the processing of the program. The work list ($work\_list$) is initialized with end_nodes. Thus, from line 12 to 25, we propagate the vectors of callees to callers until $work\_list$ is empty. In line 13, the current node (cur_node) is randomly picked up from $work\_list$. In line 14, the predecessor nodes (pre_nodes) of (cur_node) are selected. Thus, for each node in pre_nodes, we propagate the vectors of its successors to it from line 15 to 24. The constraint is verified in line 18. If all nodes in suc_nodes of a node have been visited, we average the sum

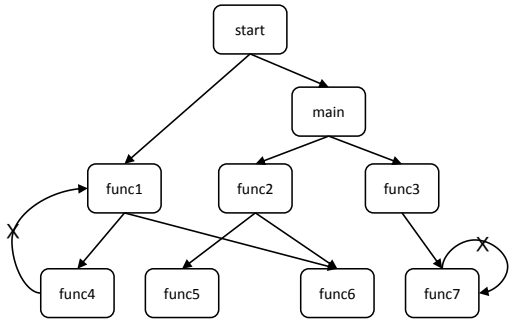

Fig. 5. Example of call graph.

of its vector and those of its successors (line 19 and 20). Then this average value is assigned to the node as its final vector representation in line 22.

In Fig. 5, when the recursive calls are deleted, "*func4*", "*func5*", "*func6*", and "*func7*" are the nodes with final vector representations. Thus, we can propagate these vectors back to "*func1*", "*func2*", and "*func3*" according to their calling relationships. After obtaining the final vector representations of "*func1*", "*func2*", and "*func3*", the vector representations of "*func2*" and "*func3*" can be propagated back to function "*main*". However, we cannot propagate the vector representations of "*func1*" and "*main*" back to "*start*", since the vector representation of "*main*" is not the final result. Lastly, the vector representation of "*start*" is treated as the vector representation of the whole program.

3.4 Selection Scheme

With the vector representations of optimization sequences and testing programs, we present a selection scheme in this subsection to select representative optimization sequences and testing programs.

Given a set of instances (i.e., optimization sequences and testing programs), we aim to select a small set of instances with better diversity, since the space of instances is huge and duplicate bugs may be triggered by the instances with high similarities. Algorithm 3 presents the proposed selection scheme. The central idea is to select instances one by one such that the total distances among the selected instances are maximized. First, M instances will be generated by a random generator (for the generation of initial optimization sequences and testing programs, see Section 4.2). We then cluster these M instances into groups. After that, the central instances of each group are selected as the initialization of the set of selected instances $selected\_insts$, which leads at least one instance in each group to be selected. This is because the distribution of instances may be unbalanced, which causes that some instances with special features that can trigger bugs may be lost. In this paper, we use the X-means algorithm [37] to cluster the target instances, since it can automatically determine the best number of groups. Next, from line 7 to 18, we select the best candidate instance in each iteration until k required instances are selected.

Specifically, the selection procedure is a "centroid" based scheme. Suppose the solar system is a set of instances, the task is to select some instances around the center "Sun". For a candidate instance, we first calculate the distance

Algorithm 3: Selection of representative instances

```
   Input: M: total number of instances.
   Input: k: number of instances should be selected.
 1 total_insts = random_instance_generator(M);
 2 insts_clusters = Clustering(total_insts);
 3 total_center = center of total_insts;
 4 center_of_clusters = centers of each cluster in insts_clusters;
 5 selected_insts = center_of_clusters;
 6 count = number of instances in selected_insts;
 7 while count < k do
 8     best_inst = null;
 9     best_score = 0;
10     for each inst ∈ total_insts and inst ∉ selected_insts do
11         dist2center = dist(inst, total_center);
12         min_dist2sel = min(dist(inst, selected_insts));
13         score = dist2center * min_dist2sel;
14         if score > best_score then
15             best_score = score;
16             best_inst = inst;
17     selected_insts = selected_insts ∪ best_inst;
18     count = count + 1;
19 return selected_insts;
```

$dist2center$ from this instance to the center, such that the instances with different orbits can be distinguished like the "Earth" and "Mars". In addition, the minimum distance $min\_dist2sel$ from the candidate instance to the selected instances is calculated for avoiding similar instances in the same or similar orbits. For example, if the instances "Earth" and "Mars" are selected, the instance "Moon" cannot be selected since it is very close to "Earth". Therefore, to balance the effect of these two distances, we leverage the product of these two distances as the score of a candidate instance. The larger the score a candidate instance has, the better it will be. In this study, we utilize the Euclidean distance function $dist(u, v)$ to calculate the distance between two instances. Suppose $\vec{u} = (u_1, u_2, \cdots, u_n)$ and $\vec{v} = (v_1, v_2, \cdots, v_n)$ are two candidate instances represented by n dimensional vectors, $dist(\vec{u}, \vec{v}) = \sqrt{(u_1 - v_1)^2 + ... + (u_n - v_n)^2}$. In Algorithm 3, for a candidate instance that has not been selected, we firstly calculate the distance from it to the centroid in line 11; then the minimum distance from it to the selected instances is calculated in line 12, the score of the current instance is calculated via the product of these two distances in line 13. In lines 10 to 16, the current best instance with the maximum score is selected. This process repeats until k instances are selected.

4 Evaluation

In this study, we conduct experiments on LLVM to evaluate the effectiveness of CTOS. Specifically, our evaluation aims at answering the following two Research Questions (RQs).

- **RQ1**: Can the proposed selection scheme help to detect bugs?

This RQ investigates whether the proposed selection scheme for selecting representative optimization sequences and testing programs is helpful for detecting compiler bugs of LLVM. Besides the proposed selection scheme, we also consider two alternative methods for acquiring optimization

sequences and four alternative methods for selecting testing programs (see details in Section 4.3). Thus, we conduct an experiment for comparing the results of CTOS under three kinds of optimization sequences and five kinds of testing programs.

- **RQ2**: How is the bug-finding capability of CTOS in practice?

In this RQ, we investigate the capability of CTOS for detecting LLVM bugs caused by optimization sequences in practice. Specifically, we evaluate CTOS from three aspects, namely, the number of reported bugs, the type of bugs, and the number of buggy optimizations.

4.1 Implementations

We implemented CTOS with approximately 2,000 lines of Python code[5]. In our study, the used Doc2Vec [29] model is implemented in Gensim [38], which is a widely used Python library for the representation of natural language. For the vector representation of optimization sequences, we set $vector\_size = 100$, $window = 3$, $alpha = 0.05$, and $epochs = 20$ to train the Doc2Vec model. All the parameters of Doc2Vec are set according to the documents of Gensim and the suggestions by [39]. Algorithms 1 and Algorithm 2 for the vector representation of a given program are implemented based on NetworkX [40]. First, a C program is transformed into its IR using Clang without any optimization. Then the optimizer Opt of LLVM takes in this IR to generate the corresponding region graph and the call graph. Similarly, we employ Doc2Vec to obtain the vector representation of instruction sequences on each edge of a region graph, and the parameters of Doc2Vec are the same as those in the representation of optimization sequences except that $window = 5$.

Note that in this study, we train two Doc2Vec models in each iteration to obtain the vector representations of optimization sequences and testing programs, respectively. A typical Doc2Vec model usually requires a large number of training data to solve the out-of-vocabulary problem which may be time-consuming. Since the space of optimization sequences is extremely huge, the out-of-vocabulary problem may also negatively affect the representation of optimization sequences (or testing programs). To alleviate this problem, in each iteration of testing, we train local Doc2Vec models specialized to learn the knowledge of current optimization sequences and testing programs. That is, we train two new Doc2Vec models using the new randomly generated optimization sequences and testing programs. After training, we obtain the vector representations of the optimization sequences and testing programs (For vector representations of testing programs, we still need the aggregations shown in Algorithms 1 and Algorithm 2). These vectors are then taken as the inputs of the selection algorithm to select representative optimization sequences and testing programs. By this setting, CTOS can efficiently learn the representation of optimization sequences and testing programs and be less influenced by the out-of-vocabulary problem.

For the selection scheme presented in Section 3.4, the used clustering algorithm X-means is implemented in Py-Clustering [41], which is a data mining library that provides a wide range of clustering algorithms. The parameters of X-means are set to the default values. In Algorithm 3, the total numbers of initial optimization sequences and testing programs (i.e., M) are 300,000 and 100,000, respectively, they reach the limitation of memory of our system on processing optimization sequences and testing programs. The numbers of optimization sequences and testing programs (i.e., k) that should be selected by Algorithm 3 are 3,000 and 1,000, respectively. This is because given the testing period (i.e., two weeks for one testing process on LLVM in practice), we can test about 3,000 optimization sequences and 1,000 testing programs. Besides, we set a single testing period to two weeks, because we hope to timely update and test the latest development version of LLVM.

4.2 Testing Setup

Hardware. Our evaluation is conducted on an x86_64 computer running Ubuntu 18.04 Linux with an Intel® Core™ i7-7700 CPU @ 3.60GHZ x 8 processor and 16GB of memory.
Compilers. In our study, we only conduct our experiments on LLVM. The reason is that, as to our knowledge, only LLVM currently can allow developers to adjust the orders of optimizations. For GCC, another mature and widely used compiler in both industry and academia, the orders of optimizations are fixed[6]. Although any order of optimizations can be passed as command-line arguments to GCC, the orders of these optimizations cannot affect the behavior of GCC. The same case also occurs for CompCert that is a verified and high-assurance compiler for the C language [7]. The fixed order of optimizations is beneficial for the rapid implementation and safety of compilers. However, there is no doubt that the fixed order of optimizations limits the capabilities of compilers to optimize programs for different requirements. In contrast, as a compiler providing support for arbitrary orders of optimizations, LLVM has been widely used to implement many compilers and tools. Our study aims to improve the reliability of optimizations with arbitrary orders for LLVM, which helps to guarantee the correctness of different LLVM-based compilers and tools.
Optimizations. In our study, C programs are used as the inputs of LLVM. Thus we mainly focus on testing the machine independent optimizations of LLVM that are useful for the C programming language. We currently do not consider the optimizations for object-oriented programming languages (e.g., C++ and Objective-C), profile guided optimizations, and results visualization of optimizations. Finally, 114 optimizations are selected[8]. In LLVM, each optimization may depend on certain other optimizations as the preconditions. LLVM provides a mechanism to manage the dependencies, i.e., PassManager[9]. Thus, we do not need to manually manage the dependencies of optimizations.

For generating the initial optimization sequences, we assign an index to each optimization. The length of an op-

5. All the source code of CTOS will be publicly available after the publication.

6. https://stackoverflow.com/questions/33117294/order-of-gcc-optimization-flags.

7. https://github.com/AbsInt/CompCert/issues/287.

8. The full list of 114 optimizations can be found on the website, https://github.com/CTOS-results/LLVM-Bugs-by-Optimization-sequences.

9. https://llvm.org/docs/WritingAnLLVMPass.html.

timization sequence is randomly selected in the range from 50 to 200. This is because there are 84 unique transformation optimizations in the -O3 optimization level of the current released LLVM 7.0.1 (the optimizations in -O3 optimization level may be different in different versions of LLVM). The range from 50 to 200 could guarantee that the generated optimization sequences have different lengths. Then we leverage a uniform random number generator to randomly generate the indexes of optimizations until the length of the current optimization sequence is reached. Next, the index sequence is translated into the corresponding optimization sequence. In addition, the parameters of optimizations are set to the default values.

Testing Programs. We use Csmith [13], a widely used program generator that supports most features of the C programming language to generate initial testing programs. To detect compiler bugs caused by optimization sequences, the testing program needs to be valid, free to undefined behaviors, diverse, and executable. However, other program generators (e.g., CCG [42], Yarpgen [43], Orion [9]) can not meet our requirements. For example, CCG can not generate runnable testing programs, and the maintenance of CCG has stopped for a long time. Yarpgen is a generator to produce correct runnable C/C++ programs, but it only supports a few C/C++ features. Orion is a mutation-based tool to generate new programs for seed programs by deleting the dead code in the seed programs. For the mutation-based tools (e.g., Orion), the diversity of the programs generated by these tools is limited by the seed programs. In addition, some grammar-based program generators (e.g., Grammarinator [44]) can also be utilized to generate testing programs, but the generated programs always are invalid and contain undefined behaviors. Thus, we employ Csmith to generate testing programs in this paper. The minimum size of the generated programs is set to 80KB as suggested in [13]. Other parameters of Csmith are set to the default values. In addition, we leverage LLVM warnings and Frama-C[10] to detect undefined behaviors of the generated programs, since the undefined behavior may cause invalid compiler bugs.

Test Case Reduction. Similar to other related studies (e.g., [9], [13], [17]), all test cases that trigger compiler bugs should be reduced before we report them to the developers, such that the developers can quickly locate the real reasons of the bugs and fix them. Test case reduction includes two parts in our study, namely, optimization sequence reduction and testing program reduction. For the reduction of optimization sequences, we remove each optimization in the optimization sequence one by one. If the bug still occurs, which indicates that this optimization has no impact on the bug, we delete it from the optimization sequence; otherwise, the removed optimization will be put back into the original position. This process continues until no optimization can be deleted. Additionally, similar to the related work [9], [13], [17], we also use Creduce [45], a widely used tool for reducing C, C++, or OpenCL programs, to reduce the testing programs that have triggered bugs. LLVM warnings and Frama-C are used to detect undefined behaviors during the reduction process to ensure that the resultant program is valid.

Note that, we first reduce the optimization sequences. This is because the reduction of testing programs always takes more time than the reduction of optimization sequences. The developers of LLVM think that "*the passes are mostly designed to operate independently, so if we see an assert/crash, then we can always blame the last pass in the sequence. And if the test ends with the same assertion and backtrace in the last pass in the sequence, then we can assume that it is a duplicate*." (see LLVM Bug#40927 [46]) Thus, if the last optimizations of the reduced sequences are identical (for the crash bug, the failed assertion or backtrace also should be identical), the corresponding bugs are treated as a duplicate. Therefore, reducing optimization sequences firstly can save the total time to reduce test cases.

Duplicate Bug Identification. In our study, we also adopt the above strategy to filter out duplicate bugs. We treat the last optimization in a reduced optimization sequence as a buggy optimization. Thus, if the last optimizations in the two reduced optimization sequences of two bugs are the same, these two bugs are treated as a duplicate. Besides, for crash bugs, to improve the accuracy of duplicate bugs identification, we further use the failed assertion or backtrace to determine duplicate crash bugs. That is, when two crash bugs have the same failed assertion or backtrace, we treat them as duplicate crash bugs. The reason for adopting this strategy rather than just distinct optimization sequences to identify duplicate bugs is because many duplicate bugs can be triggered even though the reduced optimization sequences are distinct. This is also the strategy applied by the LLVM developers, as can be seen in LLVM Bug#40926 [47], #40927 [46], #40928 [48], #40929 [49], and #40933 [50], which are marked as duplicates of LLVM Bug#40925 [51]. Although the optimization sequences of these bugs are distinct, they are marked as duplicates because the root causes of these bugs are introduced in the same last optimization in the sequences. Through this strategy, we may avoid reporting too many duplicate bugs to developers.

Bug Types. In our study, we mainly find the following five types of compiler bugs caused by optimization sequences of LLVM.

1) **Crash**. The optimizer *Opt* of LLVM crashes when optimizing the IR of a program.
2) **Invalid IR**. In LLVM, each optimization takes in the valid IR of a program as input, and its outputs also should be a valid IR. However, invalid IR may be generated by some optimizations due to the interaction among optimizations. In our evaluation, we tune on the option "*-verify-each*[11]" of the optimizer *Opt* to verify whether the output IR is valid after every optimization. If the IR is invalid after an optimization, the optimizer *Opt* will be stopped and will output some error messages.
3) **Wrong code**. The optimized IR produced by the optimizer *Opt* may contain different semantics to the original program, which makes the corresponding executable produce wrong outputs, or occur segmentation faults or floating point exceptions.
4) **Performance**. When the out-of-memory of the optimizer *Opt* occurs, we treat it as performance bugs. It could slow the compilation of programs. In the worst

10. http://frama-c.com/.

11. http://llvm.org/docs/CommandGuide/opt.html.

case, the computer system may be jammed due to the performance bug. Generally, we set the maximum size of the memory of an optimizer process to 4GB, since it is sufficient to optimize a testing program using 4GB memory in most cases. Thus if the maximum memory of the optimizer Opt is greater than 4GB, the optimizer will be stopped like a crash bug. We also try not to limit the size of the memory, but it has the same results as the 4GB limitation.

5) **Code generator bug**. The code generator in the back-end is used to generate the assembly code of a program from the corresponding IR. However, the IR of a program optimized by some optimizations may trigger some bugs in the code generator. Currently, we only find one bug for this type, it makes the code generator not emit a machine instruction and stops the code generator.

4.3 Answer to RQ1

To evaluate the effectiveness of the proposed selection scheme, we design an experiment for comparing the results of CTOS with three kinds of optimization sequences and five kinds of testing programs. The experiment is conducted on the recently released version of LLVM 7.0.1, such that we can verify the detected bugs using the latest development version of LLVM. For a detected bug, if it does not exist in the latest development version of LLVM, we then think it has been fixed; otherwise, we report it to the developers of LLVM to further identify whether it is a valid bug.

Specifically, there are three kinds of optimization sequences. Besides the optimization sequences selected by the proposed selection scheme (*SS*) in CTOS, we adopt a random strategy and a combinatorial testing technique [52] to generate optimization sequences. For randomly generating optimization sequences (*RS*), an index is assigned to each optimization, and then a uniform random number generator is utilized to randomly generate the indexes of optimizations until the length (that is randomly selected in the range from 50 to 200) of the current optimization sequence is reached. Next, we translate the index sequence into the corresponding optimization sequence. This random strategy is identical to the strategy described in Section 4.2. The minimum and maximum lengths of *RS* are the same as those of *SS*, and the number of *SS* and *RS* is identical (i.e., 3,000). In addition, we utilize the combinatorial testing techniques implemented in ACTS [53] to generate optimization sequences. ACTS is a widely used test generation tool for constructing t-way combinatorial test sets to detect failures triggered by interactions of parameters in the software [52]. However, we can only generate 2-way combinatorial optimization sequences *2W* since there are too many optimizations. In our study, we set the length of the optimization sequences generated by ACTS to 200 for simplicity, and then ACTS generates 34,473 2-way combinatorial optimization sequences.

In addition, besides the testing programs selected by the proposed selection scheme (*SP*) in CTOS, four alternative methods are used to select testing programs. Similar to other studies (e.g., [13]), we also use Csmith with the default configuration to randomly generate testing programs (*RP*). Additionally, the swarm testing technique [54] is utilized to guide Csmith to generate diverse testing programs by randomizing test configurations. As in the study [55], we also consider two versions of swarm testing: an original version of swarm testing [54] that randomly sets the value of each configuration option to be 0 or 100, and a variant of swarm testing that randomly sets the value of each configuration option to be a floating-point number ranging from 0 to 100 [55]. Thus, we obtain two kinds of testing programs by swarm testing, namely the testing programs generated by the original swarm testing (*RPS*) and the testing programs generated by the variant of swarm testing (*RPSv*). Moreover, apart from the proposed vector representation of testing programs, the static features of programs presented in [20] are used to select testing programs with the proposed selection scheme (*SPS*). The static features of programs include language features, operation features, and structure features (see [20] for details). The optimization sequences and testing programs used in RQ1 are summarized in Table 1. Therefore, besides CTOS that takes in *SP* and *SS* as inputs in this study, we obtain 14 variants of CTOS, namely, (1) CTOS(*RP*+*RS*), (2) CTOS(*RP*+*SS*), (3) CTOS(*RP*+*2W*), (4) CTOS(*RPS*+*RS*), (5) CTOS(*RPS*+*SS*), (6) CTOS(*RPS*+*2W*), (7) CTOS(*RPSv*+*RS*), (8) CTOS(*RPSv*+*SS*), (9) CTOS(*RPSv*+*2W*), (10) CTOS(*SP*+*RS*), (11) CTOS(*SP*+*2W*), (12) CTOS(*SPS*+*RS*), (13) CTOS(*SPS*+*SS*), (14) CTOS(*SPS*+*2W*).

TABLE 1
Optimization sequences and testing programs for RQ1.

RS	Optimization sequences generated by the random strategy
SS	Optimization sequences selected by the proposed selection scheme
2W	Optimization sequences generated by the combinatorial testing technique
RP	Testing programs randomly generated by Csmith
RPS	Testing programs randomly generated by the swarm testing
RPSv	Testing programs randomly generated by the variant of swarm testing
SP	Testing programs selected by the proposed selection scheme
SPS	Testing programs selected by the proposed selection scheme with the static features of programs

For CTOS and its 14 variants, we run each 10 times and the timeout is 90 hours in each time as the setting in [19]. Therefore, it takes nearly 50 days to run these experiments. Notably, the 10 runs of CTOS and its 14 variants are independent, i.e., the optimization sequences and testing programs are different in each time except the 2-way combinatorial optimization sequences, which are identical in each variant of CTOS since the optimization sequences generated by ACTS are constant. In addition, the testing programs and optimization sequences for CTOS and its 14 variants are prepared before we carry out the experiments. The initial number of testing programs for each experiment is 1,000, because we cannot know how many testing programs can be tested before the experiments. It is rapid to generate *RS*

TABLE 2
Results of CTOS and its 14 variants.

	Average								Min. total bugs	Max. total bugs	*P*-value	Effect size (A_{12})
	TP.	Crash	WC.	Inv. IR	Perf.	CGB	Total bugs	*imp.*				
CTOS(*RP+RS*)	113.2	6.4	1.0	1.4	1.0	0.0	9.8	33.67%	7	12	$<$ .001	0.955
CTOS(*RP+SS*)	100.7	6.7	1.7	1.0	0.9	0.0	10.3	27.18%	8	12	.001	0.905
CTOS(*RP+2W*)	9.2	7.2	1.3	0.4	1.0	0.0	9.9	32.32%	8	11	$<$ .001	0.960
CTOS(*RPS+RS*)	28.8	5.2	1.0	1.4	1.0	0.1	8.7	50.57%	7	10	$<$ .001	1.000
CTOS(*RPS+SS*)	27.5	5.4	1.4	1.2	0.9	0.0	8.9	47.19%	7	11	$<$ .001	0.980
CTOS(*RPS+2W*)	3.9	5.6	1.3	0.9	1.0	0.0	8.8	48.86%	6	11	$<$ .001	0.990
CTOS(*RPSv+RS*)	61.4	5.7	0.9	1.3	1.0	0.1	9.0	45.56%	7	12	$<$ .001	0.965
CTOS(*RPSv+SS*)	53.0	5.7	1.7	1.4	1.0	0.1	9.9	32.32%	9	11	$<$ .001	0.980
CTOS(*RPSv+2W*)	5.5	6.6	1.3	1.2	1.0	0.1	10.2	28.43%	8	13	$<$ .001	0.915
CTOS(*SP+RS*)	122.4	6.6	1.4	1.2	1.0	0	10.1	29.70%	9	11	$<$ .001	0.960
CTOS(*SP+2W*)	8.6	6.8	1.8	0.6	1.0	0	10.2	28.43%	8	12	$<$ .001	0.935
CTOS(*SPS+RS*)	117.8	5.8	1.1	1.1	1.0	0	9.0	45.56%	7	10	$<$ .001	1.000
CTOS(*SPS+SS*)	102.4	6.9	2.0	0.6	1.0	0	10.5	24.76%	9	12	.001	.0930
CTOS(*SPS+2W*)	8.2	7.0	1.5	0.5	1.0	0	10.0	31.00%	8	12	$<$ .001	0.955
CTOS	109.9	7.4	3.5	1.2	1.0	0	13.1	–	11	15	–	–

TP.: Testing Programs, Inv. IR: Invalid IR, WC.: Wrong Code, CGB.: Code Generator Bug.

and *RP*. While the time for generating *RPS* and *RPSv* needs about 3 hours, since some random values of configuration options may cause Csmith takes more time to generate a testing program[12]. Generating *SS*, *SP*, and *SPS* takes 3 to 6 hours in our system, respectively. Hence, compared to the testing period, the time for preparing optimization sequences and testing programs is relatively short. The most time-consuming part is to generate *2W*, which takes about 10 hours in our system. But we only need to generate *2W* once, all the experiments use the same *2W*.

In RQ1, we do not include the time spent on generating testing programs and optimization sequences into the testing period for two reasons. On the one hand, we adopt this evaluation strategy due to the limitation of computational resources. Our experiments are conducted on a computer with 16GB of memory. Although we have tried our best to optimize our programs to select representative testing programs from the initial set of 100,000 testing programs in our experiments, the selection process can consume 4-8GB of memory. Besides, we need to run 150 experiments (CTOS and its 14 variants, 10 runs for each experiment) in RQ1, which makes us run many experiments simultaneously so that we can finish all experiments in nearly 50 days. For each experiment, the testing process will consume about 100M-4GB of memory. Thus, if we integrate the selection of testing programs into the testing process, the testing efficiency may be dramatically decreased due to the possible memory swapping. However, under our current experiment setting, we may only use a small fraction of the initial 1,000 testing programs to test different optimization sequences. This may be a threat of validity for our experiments, which will be discussed in Section 6. On the other hand, regarding the baselines using the combinatorial testing technique, it generates the same set of optimization sequences (i.e., *2W*) each time (taking about 10 hours). The experiment reuses *2W* for each run of different baselines. Thus, it may bring unfair comparisons between the baselines with *2W* and without *2W*, when the optimization sequence generation time is included.

12. https://github.com/csmith-project/csmith/blob/master/doc/probabilities.txt

Table 2 presents the experiment results of CTOS and its 14 variants. The second column is the average total number of testing programs, and the following 5 columns are the average number of unique bugs for each type. Actually, many duplicate bugs can be found by CTOS and its 14 variants. We filter out these duplicate bugs using the strategy described in Section 4.2. Next, the eighth column is the average total number of unique bugs for CTOS and its 14 variants. From Table 2, the numbers of testing programs for CTOS(*RP + 2W*), CTOS(*SP + 2W*), and CTOS(*SPS + 2W*) are smaller than those of CTOS and other variants, since the number of 2-way combinatorial optimization sequences is about 11 times larger than those of other type optimization sequences. In addition, the six variants with *RPS* and *RPSv* test a small number of testing programs compared to CTOS and other variants. For example, for the variants with *RS*, CTOS(*RPS + RS*) and CTOS(*RPSv + RS*) only test 28.8 and 61.4 testing programs on average, respectively. However, the numbers of testing programs of CTOS(*RP + RS*), CTOS(*SP + RS*), and CTOS(*SPS + RS*) are 113.3, 122.4, and 117.8, respectively. The reason is that testing programs generated by swarm testing may contain some complicated structures, which cause a long time to optimize them and execute the corresponding executables. Generally, the majority of time for the whole testing process is utilized to optimize testing programs and execute the corresponding executables. This time is decided by both of the corresponding program and optimization sequence. Given a testing program, if it is complicated (e.g., including many nest loops), the time for optimizing and executing it could be longer than a simple testing program. Besides, different optimization sequences can also affect the time for optimizing and executing the same testing program. In Table 2, the number of testing programs of CTOS is larger than CTOS(*RP+SS*) since there are two experiments that CTOS tests more than 120 testing programs. Nevertheless, this result cannot suggest that CTOS can always test more testing programs than its variants, because the numbers of testing programs of the variants (e.g., CTOS(*RP+SS*)) in some experiments are larger than CTOS.

It is obvious from Table 2 that CTOS significantly outperforms its 14 variants in terms of the bug-finding capability.

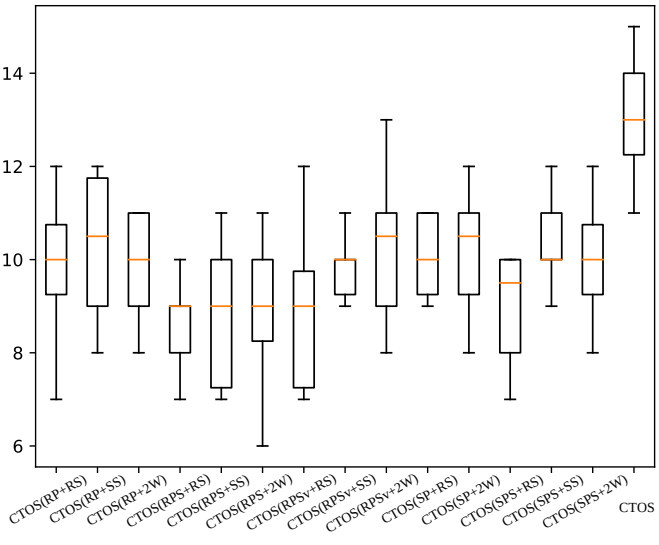

Fig. 6. Comparison of 10 total bugs of CTOS and its 14 variants.

CTOS can find 13.1 unique bugs on average, while the average total number of unique bugs for the best variant CTOS(*SPS+SS*) is only 10.5. The ninth column shows the improvement of CTOS over the baselines (i.e., the 14 variants of CTOS) in terms of the average total number of unique bugs, i.e., $imp = (CTOS - baseline)/baseline * 100\%$. From the ninth column of Table 2, we can see that CTOS can find more unique bugs than the baselines by up to 24.76% to 50.57%. Specifically, CTOS can detect more crash bugs and wrong code bugs. It finds 7.4 crash bugs and 3.5 wrong code bugs on average. In addition, we could observe from Table 2 that CTOS and its variants with the selected optimization sequences (*SS*) and testing programs (*SP*) can find more bugs. On the one hand, CTOS and the variants with *SS* outperform others when the testing programs are *RP*, *RPS*, *SP*, and *SPS*, respectively. For example, CTOS(*RP+SS*) finds 10.3 bugs on average which outperforms CTOS(*RP+RS*) and CTOS(*RP+2W*). The only exception is *RPSv*, which makes CTOS(*RPSv+2W*) detect more bugs than CTOS(*RPSv+SS*). On the other hand, *SP* also leads CTOS, CTOS(*SP+RS*), and CTOS(*SP+2W*) to detect more bugs than others except CTOS(*RPSv+2W*). For instance, CTOS(*SP+RS*) finds 10.1 bugs, while there are only 9.8 and 9.0 bugs for CTOS(*RP+RS*) and CTOS(*SPS+RS*), respectively. The reason is that the optimization sequences and testing programs selected by the proposed selection scheme may have better diversity.

From Table 2, almost all experiments can find the only one unique performance bug[13]. This performance bug is caused by the optimization "*-newgvn*". However, CTOS and the 14 variants with *RP*, *SP*, and *SPS* fail to detect the code generator bug in this experiment. Only some variants with *RPS* and *RPSv* detect one code generator bug[14] that has been fixed. This is because testing programs generated by the swarm testing may cover a larger portion of input space than the other approaches due to the random mechanism of swarm testing for constructing test configurations. Although the proposed selection scheme aims to select divers testing programs, in fact it may be limited by Csmith with the default configuration within the given testing period. Hence, the proposed selection scheme may miss some corner cases, which means that there is still room to further improving the selection scheme.

Additionally, most bugs found by CTOS and its variants are the crash bugs. However, the difference between the number of crash bugs for CTOS and each variant is not very large. The reason is that there are some bugs which are relatively easy to be triggered by some optimization sequences. For example, LLVM Bug#39626[15] can be easily triggered by the optimization sequences that contain the subsequence "*.*-early-cse-memssa.*-early-cse-memssa.**".

Figure 6 is the boxplot of the total bugs found during the 10 runs of CTOS and its 14 variants. In this boxplot, CTOS significantly outperforms the variants. The columns "Min. total bugs" and "Max. total bugs" in Table 2 show the minimal and maximal numbers of the total bugs in the 10 runs of CTOS and each variant, respectively. From Table 2 and Fig. 6, we can see that the minimal and maximal numbers of the total bugs of CTOS are 11 and 15 respectively, which are clearly larger than those of the variants. Although in some cases the number of total bugs found by CTOS is lower than the variants, the median of CTOS illustrates that the bug-finding capability of CTOS is better than the baselines in most cases. In addition, we conduct the Mann-Whitney U-test with a level of significance 0.05 on the total bugs between CTOS and the variants according to the suggestions by Arcuri and Briand [56]. The P-value ($p \leq .001$) in Table 2 shows that CTOS performs significantly better than the variants. Furthermore, we also calculate the effect size of the differences between CTOS and the baselines using the Vargha and Delaneys A_{12} statistics[16] [56]. If CTOS and the baselines are equivalent, then $A_{12} = 0.5$; if the effect of CTOS is small compared to the baselines, then $A_{12} < 0.5$; otherwise, $A_{12} > 0.5$. From Table 2, we can see that all the effect sizes are greater than 0.9, which indicates that CTOS has a higher probability to obtain better results than the baselines. Particularly, the values of the effect size for CTOS(*RPS+RS*) and CTOS(*SPS+RS*) are 1.000. This means that the total bugs found during the 10 runs of CTOS are completely larger than those of CTOS(*RPS+RS*) and CTOS(*SPS+RS*).

Answer to RQ1: The experimental results demonstrate that CTOS significantly outperforms the baselines by detecting $24.76\% \sim 50.57\%$ more bugs on average, which reveals the advantage of the proposed selection scheme for selecting representative optimization sequences and testing programs.

4.4 Answer to RQ2

To evaluate the bug-finding capability of CTOS in practice, we conduct an experiment over seven months from January 2019 to July 2019. In this experiment, we mainly test the latest development version of LLVM, since the developers of LLVM fix bugs primarily in the latest development version

13. https://bugs.llvm.org/show_bug.cgi?id=41290.
14. https://bugs.llvm.org/show_bug.cgi?id=42452.
15. https://bugs.llvm.org/show_bug.cgi?id=39626.
16. We use the open source code shared by Tim Menzies to calculate A_{12}, https://github.com/txt/ase16/blob/master/doc/stats.md.

TABLE 3
Excluded optimization subsequences.

ID	Regular expressions of subsequences	Type
1	".*-early-cse-memssa.*-early-cse-memssa.*"	Crash
2	".*-gvn-hoist.*-early-cse-memssa.*"	Crash
3	".*-loop-unroll.*-licm.*"	Crash
4	".*-loop-reduce.*-loop-reduce.*"	Crash
5	".*-loop-rotate.*-loop-vectorize.*"	Wrong code
6	".*-loop-unroll.*-loop-reroll.*"	Wrong code
7	".*-memcpyopt.*-gvn.*"	Wrong code
8	".*-reg2mem.*-newgvn.*"	Wrong code

TABLE 4
Reported bugs.

Type	num.	duplicate	confirmed/fixed
Crash	57	8	16
Invalid IR	13	0	2
Wrong Code	24	1	1
Performance	9	0	1
Code Generator Bug	1	0	1
Total	104	9	21

rather than in stable versions [17], [57]. The time of one testing process is about two weeks since CTOS can test 1,000 testing programs and 3,000 optimization sequences in this period on our system. Notably, we gradually exclude some optimization subsequences that easily cause duplicate bugs for improving the test efficiency until the corresponding bugs are fixed. These optimization subsequences are the most frequent pairs of optimizations in the reduced optimization sequences that triggered a mass of duplicate bugs. Table 3 shows 8 regular expressions of subsequences that have been excluded. The first four subsequences trigger many duplicate crash bugs, while the latter four are used to avoid duplicate wrong code bugs. All the detected bugs have been fed back to the LLVM bug repository. In the seven months, we have reported in total 104 valid bugs within 5 types, of which 21 have been confirmed or fixed. Table 4 summarizes the testing results[17].

From Table 4, we can observe that the most reported bugs are crash bugs and wrong code bugs. We have detected 57 crash bugs, which are mainly caused by assertion failures and segmentation faults. For these crash bugs, 16 of them have been confirmed or fixed. However, 8 of these crash bugs are duplicate, which are caused by the constant hoisting optimization. Although the optimization sequences that trigger these 8 bugs are different, the backtrace information of these bugs is identical. Thus developers treat them as duplicate bugs [46]. We also adopt this strategy to filter out duplicate crash bugs. In addition, 24 valid wrong code bugs have been reported, one of them is confirmed. Compared to crash bugs, the number of confirmed or fixed wrong code bugs is very small. The reason is that the root causes of wrong code bugs are hard to be isolated [58]. For crash bugs, developers can leverage the backtrace information to analyze the root causes of the bugs, while only limited information (e.g., intermediate results of compilers) could be used to help developers logically understand the root cause of a wrong code bug.

Apart from crash bugs and wrong code bugs, we also reported 13 invalid IR bugs, 9 performance bugs, and 1 code generator bug. For these bugs, 2 invalid IR bugs, 1 performance bugs, and 1 code generator bug have been confirmed or fixed. Especially, the code generator bug that prevents the code generator from emitting *physreg copy instruction*[18] is fixed only after one day since it is critical for LLVM to generate the correct assembly code. The invalid IR bugs are also important for LLVM as they may reveal the design flaws of the optimizations. For example, LLVM bug#41723[19] found by CTOS shows that the "-scalarizer" optimization may produce the invalid IR that cannot be processed by other optimizations, since it cannot correctly process the unreachable blocks.

From Table 4, only 21 of the reported bugs are confirmed or fixed. To investigate this phenomenon, we collect 1,323 unique bugs related to scalar optimizations (most optimizations in LLVM belong to this component) from the LLVM bug repository[20] from October 2003 to Jun 2019. In these bugs, 828 bugs have been confirmed or fixed, and 495 bugs are still kept as "NEW". For the 828 confirmed or fixed bugs, although 428 bugs are confirmed or fixed in one month, the developers take more than 15 months to confirm or fix the most residual bugs. The average number of months for confirming or fixing these bugs is 5.6. In addition, 495 bugs with "NEW" status have already existed for a long time, an average of 14.1 months. This indicates that the overall speed to confirm or fix LLVM bugs is relatively slow, not just for our reported bugs. One possible reason is that it is hard and time-consuming to analyze and find the root causes of compiler bugs [58]. Especially, for a bug caused by an optimization sequence, the root causes of this bug may lie in any optimization of this sequence. In addition, the study by Sun *et al* [19] also finds that the bug-fixing rate of LLVM is lower than GCC. The authors explain that this is due to the limited human resources since some LLVM developers in Apple are pulled into other projects like Swift [19]. Furthermore, we also talk with three developers of an international company that has a team to develop compilers using LLVM. We ask these three developers what are the difficulties to fix an LLVM bug during their development. All these three developers say that it is difficult to analyze and find the root causes of compiler bugs, especially for the wrong code bugs.

Table 5 presents the buggy optimizations of the 104 reported bugs arranged in the alphabetical order. These buggy optimizations are the last ones in the reduced optimization sequences, since the optimizations of LLVM are mostly designed to operate independently and the developers always blame the last optimizations in the reduced sequences [46]. 47 unique optimizations have been reported to be faulty. Specifically, there are 15 buggy loop related optimizations

17. The details of these bugs can be found on the website, https://github.com/CTOS-results/LLVM-Bugs-by-Optimization-sequences.

18. https://bugs.llvm.org/show_bug.cgi?id=42452.
19. https://bugs.llvm.org/show_bug.cgi?id=41723.
20. https://bugs.llvm.org/.

TABLE 5
Buggy optimizations for reported bugs.

Type	Optimizations
Crash	adce bdce consthoist correlated-propagation early-cse-memssa flattencfg gvn gvn-hoist gvn-sink indvars inline instcombine ipconstprop ipsccp jump-threading **licm loop-deletion loop-distribute loop-extract-single loop-instsimplify loop-reduce loop-rotate loop-unswitch loop-versioning-licm** mem2reg memoryssa mergefunc newgvn partial-inliner separate-const-offset-from-gep simplifycfg slp-vectorizer sroa
Invalid IR	hotcoldsplit indvars **loop-extract-single loop-instsimplify loop-interchange loop-reroll loop-rotate loop-unroll-and-jam loop-unswitch loop-versioning-licm** sroa scalarizer
Wrong Code	called-value-propagation constprop functionattrs globalopt gvn gvn-hoist indvars inline instcombine ipsccp jump-threading **loop-reroll loop-simplify loop-unroll loop-vectorize** newgvn structurizecfg
Performance	jump-threading **licm loop-extract** newgvn slp-vectorizer
Bug of Code Gen.	indvars

(in bold fonts), such as *loop-rotate*, *loop-unroll* and *loop-vectorize*. From Table 5, we can see that the loop related optimizations are more bug-prone than other optimizations. This result indicates that the design of loop optimizations may exist some flaws and should be further enhanced by the developers.

Figure 7 shows the statistics of the top 5 most used optimizations in the reduced optimization sequences for the 104 reported bugs. From Fig. 7(f), the optimizations *jump-threading*, *gvn*, *licm*, *loop-rotate*, and *instcombine* are the 5 most used optimizations in all reduced optimization sequences. In particular, *jump-threading* appears 44 times in all reduced optimization sequences for the 104 reported bugs. This optimization is used to turn conditional into unconditional branches that can greatly improve performance for hardware with branch prediction, speculative execution, and prefetching. However, the code structure may be complicated after performing *jump-threading*, since it will add some new paths and duplicate code[21]. This may cause other optimizations to produce wrong results. In Fig. 7 (a), Fig. 7 (b), and Fig. 7 (c), we can see that the optimization *jump-threading* is the top 2 most used optimization for the crash bugs, invalid IR bugs, and wrong code bugs. In addition, from Fig. 7 (a)-(e), optimizations that change the structure of a program (*loop-rotate* changes the structure of a loop and *structurizecfg* transforms the control flow structure of a program) are widely used in the buggy optimization sequences. This indicates that the design flaws of optimizations may be introduced by the edge cases of the structure of a program, which may help developers to pay more attention to the interactions among these optimizations when they design and implement new optimizations.

Answer to RQ2: Our testing efforts over seven months clearly demonstrates that CTOS is effective in detecting LLVM bugs caused by optimization sequences. In the seven months, we reported 104 valid bugs within 5 types, of which 21 have been confirmed or fixed. 47 unique optimizations are identified to be faulty and 15 of them are loop related optimizations.

5 Discussion

Importance of optimization sequence. The optimization sequences are mainly used to improve the performance (e.g., size, speed, and energy) of a program. Especially, the default optimization levels (e.g., -O1, -O2, and -O3) provided by a compiler are specific optimization sequences designed by the compiler experts. Although the default optimization levels can significantly improve the program performance, many studies (e.g., [1], [2]) have shown that the autotuning of optimization sequences helps to further improve the performance of a program. In addition, a program in different scenarios may have different performance requirements. For example, the energy consumption may be more important for a program in an embedded system, while the speed of a scientific program may be sensitive on a high-performance supercomputer. Nevertheless, these techniques may be invalid due to the potential bugs in the selected optimization sequences. On the other hand, compiler developers of a new program language (e.g., Rust, Swift) based on LLVM need to design better optimization sequences as the default optimization levels (e.g., O1, O2, and O3 in LLVM) to meet the features in the new program language. In this case, if there are bugs in the optimization sequences, compiler developers could be frustrated, thus badly slowing down the development. Hence, it is critical to guarantee the correctness of optimization sequences.

Buggy subsequences exclusion. In the testing process, we exclude some optimization subsequences listed in Table 3 that can easily trigger duplicate bugs. However, the excluded subsequences do not always trigger duplicate bugs. For example, LLVM bug#39626[22] can be easily triggered by the optimization sequences with first subsequence in Table 3. Especially, the optimizer *Opt* must be crashed when any program is optimized using "*-early-cse-memssa -early-cse-memssa*", while the subsequence "*-early-cse-memssa -gvn -early-cse-memssa*" cannot trigger this bug. Even so, we think this exclusion strategy contributes to improving the testing efficiency. Firstly, the excluded subsequences are manually summarized from many duplicate bugs, which indicates that the optimization sequences that contain these subsequences may trigger duplicate bugs with high probability. This helps us to test more optimization sequences and reduce the time to analyze duplicate bugs. Secondly, when the corresponding bugs have been fixed, we will remove the limitation of these subsequences, such that the deep bugs caused by these subsequences could be detected. In the future, automation techniques may be developed to make the summarization of subsequences more precise and filter out duplicate bugs to improve the testing efficiency.

Buggy optimization isolation challenge. In our study, we treat the last optimization in a reduced optimization sequence as a buggy optimization. However, this strategy is not absolutely true, since the real reason for a bug may lie

21. http://beza1e1.tuxen.de/articles/jump_threading.html.

22. https://bugs.llvm.org/show_bug.cgi?id=39626.

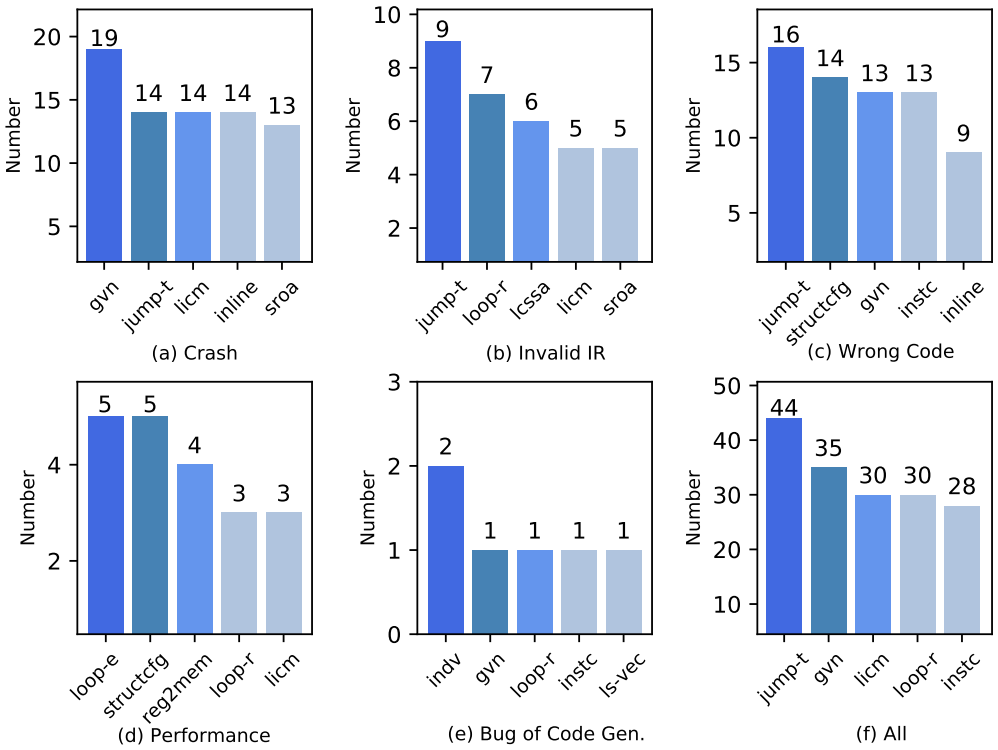

Fig. 7. Top 5 most used optimizations for reported bugs. jump-t: jump-threading, instc: instcombine, structcfg: structurizecfg, loop-e: loop-extract, loop-r: loop-rotate, ls-vec: load-store-vectorizer.

in any optimization of the reduced optimization sequence. This indicates that the results in Table 5 may not be accurate. For example, an assertion fails in LLVM bug#42264[23] when the optimizer *Opt* optimizes a program using "*-early-cse-memssa -die -gvn-hoist*". We then treat "-gvn-hoist" as the buggy optimization. Nevertheless, the developers show that the root cause is introduced by "-die" since it does not correctly preserve the information generated by "-MemorySSA" (an analysis method in LLVM[24]). Hence, we may underestimate the effectiveness of CTOS in the experiments since some bugs may be wrongly labeled as duplicates. However, we must make a tradeoff to avoid reporting too many duplicate bugs to developers. In practice, it may not be acceptable for developers to receive hundreds of reported bugs in a few days, where the majority of the bugs are duplicates. This situation is currently difficult to be alleviated. Firstly, to the best of our knowledge, there does not exist a perfect method to locate the real reasons for a compiler bug. In our work, we manually validate the reduced optimization sequences to guarantee that the bugs cannot be reproduced when omitting the last optimization of the sequences. Secondly, the developers of LLVM also utilize the same strategy to roughly determine whether the bugs caused by optimization sequences are duplicate. The optimizations in LLVM are mostly designed to operate independently, the developers always blame the last optimization in a reduced optimization sequence [46]. Thus, we believe that the results of Table 5 are reasonable. In the future, we plan to introduce advanced fault localization techniques [58], [59], [60], [61] to address this challenge.

Limitation of the selection scheme. The experimental results show that CTOS is effective to detect LLVM bugs caused by optimization sequences. However, the selection scheme in CTOS may be limited. In our study, the selection scheme is based on the hypothesis that the effects of two testing programs (or optimization sequences) for testing LLVM are similar if they are closed to each other. Therefore, for a set of testing programs (or optimization sequences), our goal is to select representative testing programs (or optimization sequences) such that the total distances among them are maximized. Nevertheless, the selection scheme may miss some corner test cases, as it is difficult to know which testing program (or optimization sequence) can trigger a bug before execution. Besides, we currently use Csmith with the default configuration to generate the initial testing programs, which may limit the diversities of the generated testing programs and affect the effectiveness of the proposed selection scheme. In future work, we will consider more advanced techniques (e.g., combine the selection scheme with coverage information) to select representative testing programs and optimization sequences to include more corner cases.

6 Threats to Validity

Threats to Internal Validity. The threats to internal validity mainly lie in the implementations of CTOS. As mentioned in Section 3, the vector representations of the optimization sequences and testing programs rely on the Doc2Vec technique. Hence, the testing efficiency may be impacted by

23. https://bugs.llvm.org/show_bug.cgi?id=42264.
24. https://www.llvm.org/docs/MemorySSA.html.

the implementation of Doc2Vec. To reduce this threat, we adopt the widely used tool Gensim [38] that has an efficient implementation of Doc2Vec. In addition, the parameters of Doc2Vec are currently set according to the documents of Gensim and the suggestions by [39]. For the parameters of Algorithm 3, we set the values of these two parameters according to the hardware limitation of our system. We do not investigate the impact of these parameters for CTOS in this paper, due to the heavy time cost to fine-tune the parameters of CTOS. There are in total 6 parameters, i.e., 4 parameters for Doc2Vec and 2 parameters for Algorithm 3. Assuming that each parameter has 10 candidate values, we will get 10^6 parameter combinations. The large number of parameter combinations can lead to a long time to investigate the impact of the parameters for CTOS, even the testing period is only 90 hours (as set in RQ1) for evaluating one parameter combination. Despite this, the experimental results illustrate that CTOS can achieve good results under the parameter settings in our paper.

Besides, in our experiments for RQ1, the time to obtain testing programs and optimization sequences is not included in the testing period, which may cause unfair comparisons between CTOS and the baselines. For example, compared with a random strategy (i.e., *RS* and *RP*), COTS spends 3-6 more hours in generating testing programs and optimization sequences. However, we do not expect this can dramatically affect the experiment results in Table 2, because the time to obtain testing programs and optimization sequences (3-6 hours) is much smaller than the testing period (i.e., 90 hours). To investigate its potential impact on the results of CTOS, we analyze the number of bugs detected by CTOS in the first 80 hours. That is, we exclude 10 hours from the testing period which are assumed to be used for CTOS to obtain testing programs and optimization sequences. As presented in the supplemental material[25], CTOS can find the majority of bugs (11.7 on average) in the first 80 hours; it outperforms all the baselines which run in 90 hours. For example, CTOS(*RP*+*RS*) detects 9.8 bugs on average in 90 hours, which is 19.39% fewer than the number of bugs detected by CTOS in the first 80 hours. Thus, we believe that CTOS outperforms the baselines even when the time to obtain testing programs and optimization sequences is considered.

Threats to External Validity. The threats to external validity mainly lie in duplicate bugs and testing programs. Firstly, many duplicate bugs are triggered in the testing process though we have leveraged the selection scheme to obtain the representative optimization sequences and testing programs. To alleviate this threat, we summarize the subsequences (listed in Table 3) that could easily trigger duplicate bugs and remove the optimization sequences that contain these subsequences in the next testing process until the corresponding bugs are fixed. In addition, we utilize the duplicate bug identification strategy described in Section 4.2 to identify duplicate bugs. However, this strategy may be not precise, which may influence the effectiveness of CTOS. This is because the root causes for a bug can be introduced by any optimization in the reduced optimization sequence. Since our strategy is also adopted by LLVM developers to identify duplicate bugs caused by optimizations in practice, this strategy can still significantly reduce the negative influence of duplicate bugs on the experiments. In the future, we will consider to apply advanced software fault localization techniques to improve the strategy for identifying duplicate bugs in our study.

Secondly, we utilize Csmith to generate the testing programs in this study. However, Csmith has been widely used to test LLVM for a long time, which makes LLVM, to a certain extent, resistant to it. In the future, the advanced techniques (e.g., the test-program generation approach via history-guided configuration diversification [62]) may be employed to further improve the diversity of testing programs.

25. https://github.com/CTOS-results/LLVM-Bugs-by-Optimization-sequences/blob/master/Appendix.pdf.

7 Related work

7.1 Compiler Testing

Compiler testing is currently the most important technique to guarantee the quality of compilers. In the literature, the techniques of compiler testing fall into three categories, namely, Randomized Differential Testing (RDT), Different Optimization Levels (DOL), and Equivalence Modulo Inputs (EMI) [9], [15], [16], [17], [18]. For a given testing program, RDT detects compiler bugs by comparing the outputs of some compilers with the same specification. DOL is a variant of RDT, and compares the outputs produced by the same compiler with different optimization levels to determine whether a compiler has bugs. Most of the techniques [12], [13], [14] belonging to RDT and DOL use randomly generated testing programs to test a compiler. Zhao *et al.* [12] develop a tool, called JTT, that automatically generates testing programs to validate the EC++ embedded compiler. In particular, Csmith [13] as the most successful random C program generator has been widely used to test C compilers. Lidbury *et al.* [14] develop CLsmith based on Csmith to generate programs for testing the OpenCL compilers.

Different from RDT and DOL, EMI compares the outputs produced by equivalent variants of a seed program to detect compiler bugs. If an output is different from others, the compiler then contains a bug [9]. There are three instantiations of EMI, namely, Orion [9], Athena [15], and Hermes [17]. Orion tries to randomly prune unexecuted statements to generate variant programs [9], while Athena can delete code from or insert code into code regions that are not executed under the inputs [15]. In contrast to Orion and Athena, Hermes [17] can generate variant programs via mutation performed on both live and dead code regions. An empirical study conducted by Chen *et al.* [19] compares the strength of RDT, DOL, and EMI, and reveals that DOL is more effective in detecting compiler bugs related to optimizations.

To accelerate compiler testing, a method [63] based on machine learning is proposed to predict the bug-revealing probabilities of testing programs, such that the testing programs with large bug-revealing probabilities can be executed as early as possible. Recently, Chen [20] *et al.* present a more efficient technique to predict test coverage statically for compilers, and then leverage the predicted coverage information to prioritize testing programs.

Our work is similar to DOL. However, unlike the traditional DOL which only considers the default optimization levels with fixed orders of optimizations, CTOS tests LLVM with arbitrary optimization sequences.

7.2 Compiler Phase-Ordering Problem

The compiler phase-ordering problem aims to improve the performance of target programs by selecting good optimization sequences [7], [8], [64]. Currently, two methodologies have been proposed to resolve the compiler phase-ordering problem. The approaches in the first category treat the compiler phase-ordering problem as an optimization problem and then evolutionary algorithms are used to resolve it. For example, Kulkarni *et al.* [65], [66] develop a method based on genetic algorithms for quickly searching effective optimization sequences. Purini *et al.* [4] propose a downsampling technique to reduce the infinitely large optimization sequence space. OpenTuner [2] uses the ensembles of search techniques to find optimal optimizations for a program.

The approaches in the second category tackle the compiler phase-ordering problem based on machine learning [1], [3], [6]. Most recent methods based on machine learning for compiler auto-tuning have been introduced by the survey [64]. Milepost [1] is a machine-learning based compiler that automatically adapts the internal optimization heuristic to improve the performance. Kulkarni and Cavazos [3] propose a method based on the Markov process to mitigate the compiler phase-ordering problem. Ashouri *et al.* [6] leverage the optimization subsequences and machine learning to build a predictive model. Huang *et al.* [67] present AutoPhase, a deep reinforcement learning method to tackle the compiler phase-ordering problem for multiple high-level synthesis programs.

However, there is no guarantee that the programs optimized by different optimization sequences are correct. No systematic work has been conducted to detect compiler bugs caused by optimization sequences. We present CTOS to mitigate this problem to further improve the reliability of optimization sequences.

8 Conclusion

In this study, we presented CTOS, a method based on differential testing, for catching compiler bugs caused by optimization sequences of LLVM. Rather than only testing compilers with predefined optimization sequences like the state-of-the-art methods, our technique validates compilers with arbitrary optimization sequences, which significantly increases the test efficiency for detecting bugs. Our evaluation demonstrates that CTOS significantly outperforms the baselines by detecting $24.76\% \sim 50.57\%$ more bugs on average. Within only seven months, we have reported 104 valid bugs within 5 types, of which 21 have been confirmed or fixed. 47 unique optimizations are identified to be faulty and 15 of them are loop related optimizations.

For future work, we will keep actively testing LLVM with CTOS, and report the detected bugs. Furthermore, we plan to design more efficient compiler fuzzing techniques with coverage information of LLVM to further improve the reliability of compiler optimizations.

Acknowledgments

We would like to thank the LLVM developers for analyzing and fixing our reported bugs. This work is supported in part by the National Natural Science Foundation of China under grant no. 61772107, 61722202, 61902181, and 62032004.

References

[1] G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois *et al.*, "Milepost gcc: Machine learning enabled self-tuning compiler," *International journal of parallel programming*, vol. 39, no. 3, pp. 296–327, 2011.

[2] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 2014, pp. 303–316.

[3] S. Kulkarni and J. Cavazos, "Mitigating the compiler optimization phase-ordering problem using machine learning," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '12. New York, NY, USA: ACM, 2012, pp. 147–162.

[4] S. Purini and L. Jain, "Finding good optimization sequences covering program space," *TACO*, vol. 9, no. 4, pp. 56:1–56:23, 2013.

[5] L. G. Martins, R. Nobre, J. M. Cardoso, A. C. Delbem, and E. Marques, "Clustering-based selection for the exploration of compiler optimization sequences," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 1, p. 8, 2016.

[6] 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. 29:1–29:28, Sep. 2017.

[7] D. B. Loveman, "Program improvement by source-to-source transformation," *Journal of the ACM (JACM)*, vol. 24, no. 1, pp. 121–145, 1977.

[8] S. R. Vegdahl, "Phase coupling and constant generation in an optimizing microcode compiler," in *ACM SIGMICRO Newsletter*, vol. 13, no. 4. IEEE Press, 1982, pp. 125–133.

[9] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 216–226.

[10] C. Lindig, "Random testing of c calling conventions," in *International Symposium on Automated Analysis-driven Debugging*, 2005.

[11] F. Sheridan, "Practical testing of a c99 compiler using output comparison," *Software: Practice and Experience*, vol. 37, no. 14, pp. 1475–1488, 2007.

[12] Z. Chen, Y. Xue, Q. Tao, G. Liang, and Z. Wang, "Automated test program generation for an industrial optimizing compiler," in *Workshop on Automation of Software Test*, 2009.

[13] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 283–294.

[14] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15. New York, NY, USA: ACM, 2015, pp. 65–76.

[15] L. Vu, S. Chengnian, and S. Zhendong, "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*, ser. OOPSLA 2015. New York, NY, USA: ACM, 2015, pp. 386–399.

[16] L. Vu, S. Chengnian, and S. Zhendong, "Randomized stress-testing of link-time optimizers," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 327–337.

[17] C. Sun, V. Le, and Z. Su, "Finding compiler bugs via live code mutation," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2016. New York, NY, USA: ACM, 2016, pp. 849–863.

[18] Q. Zhang, C. Sun, and Z. Su, "Skeletal program enumeration for rigorous compiler testing," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: ACM, 2017, pp. 347–361.

[19] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "An empirical comparison of compiler testing techniques," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 180–190.

[20] J. Chen, G. Wang, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and X. Bing, "Coverage prediction for accelerating compiler testing," *IEEE Transactions on Software Engineering*, 2018.

[21] LLVM Compiler Community, "LLVM language reference manual." [Online]. Available: https://llvm.org/docs/LangRef.html.

[22] LLVM Compiler Community, "LLVM's analysis and transform passes." [Online]. Available: https://www.llvm.org/docs/Passes.html.

[23] Clang, "Clang: a c language family frontend for llvm." [Online]. Available: http://clang.llvm.org/.

[24] Rust, "Rust program language." [Online]. Available: https://www.rust-lang.org/.

[25] Swift, "Swift program language." [Online]. Available: https://developer.apple.com/swift/.

[26] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: ACM, 2017, pp. 185–200.

[27] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224.

[28] P. D. Schubert, B. Hermann, and E. Bodden, "Phasar: An interprocedural static analysis framework for c/c++," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2019, pp. 393–410.

[29] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ser. ICML'14. JMLR.org, 2014, pp. II–1188–II–1196.

[30] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in Neural Information Processing Systems*, vol. 26, pp. 3111–3119, 2013.

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

[32] Z. Harris, "Distributional structure," *Word*, vol. 10, no. 23, pp. 146–162, 1954.

[33] A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers, principles, techniques," *Addison wesley*, vol. 7, no. 8, p. 9, 1986.

[34] R. Johnson, D. Pearson, and K. Pingali, "The program structure tree: Computing control regions in linear time," in *ACM SigPlan Notices*, vol. 29, no. 6. ACM, 1994, pp. 171–185.

[35] J. Vanhatalo, H. Völzer, and J. Koehler, "The refined process structure tree," *Data & Knowledge Engineering*, vol. 68, no. 9, pp. 793–818, 2009.

[36] Region graph, "https://www.llvm.org/doxygen/regioninfo_8h_source.html," 2019.

[37] D. Pelleg and A. Moore, "X-means: Extending k-means with efficient estimation of the number of clusters," in *In Proceedings of the 17th International Conf. on Machine Learning*. Morgan Kaufmann, 2000, pp. 727–734.

[38] R. Řehůřek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," in *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. Valletta, Malta: ELRA, May 2010, pp. 45–50.

[39] J. H. Lau and T. Baldwin, "An empirical evaluation of doc2vec with practical insights into document embedding generation," in *Proceedings of the 1st Workshop on Representation Learning for NLP*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 78–86.

[40] D. A. Schult, "Exploring network structure, dynamics, and function using networkx," in *In Proceedings of the 7th Python in Science Conference (SciPy*, 2008, pp. 11–15.

[41] A. Novikov, "PyClustering: Data mining library," *Journal of Open Source Software*, vol. 4, no. 36, p. 1230, apr 2019. [Online]. Available: https://doi.org/10.21105/joss.01230

[42] A. Balestrat, "CCG: A random c code generator." [Online]. Available: https://github.com/Merkil/ccg/.

[43] V. L. Dmitry Babokin, John Regehr, "Yarpgen." [Online]. Available: https://github.com/intel/yarpgen.

[44] R. Hodován, A. Kiss, and T. Gyimóthy, "Grammarinator: A grammar-based open source fuzzer," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, ser. A-TEST 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 4548.

[45] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for c compiler bugs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 335–346.

[46] LLVM bug 40927, "https://bugs.llvm.org/show_bug.cgi?id=40927," 2021.

[47] LLVM bug 40926, "https://bugs.llvm.org/show_bug.cgi?id=40926," 2021.

[48] LLVM bug 40928, "https://bugs.llvm.org/show_bug.cgi?id=40928," 2021.

[49] LLVM bug 40929, "https://bugs.llvm.org/show_bug.cgi?id=40929," 2021.

[50] LLVM bug 40933, "https://bugs.llvm.org/show_bug.cgi?id=40933," 2021.

[51] LLVM bug 40925, "https://bugs.llvm.org/show_bug.cgi?id=40925," 2021.

[52] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Comput. Surv.*, vol. 43, no. 2, pp. 11:1–11:29, Feb. 2011.

[53] D. R. Kuhn, R. N. Kacker, and Y. Lei, "Practical combinatorial testing," 2010.

[54] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr, "Swarm testing," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. New York, NY, USA: Association for Computing Machinery, 2012, p. 7888.

[55] J. Chen, G. Wang, D. Hao, Y. Xiong, H. Zhang, and L. Zhang, "History-guided configuration diversification for compiler test-program generation," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '19. IEEE Press, 2019, p. 305316.

[56] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE 11. New York, NY, USA: Association for Computing Machinery, 2011, p. 110.

[57] C. Sun, V. Le, and Z. Su, "Finding and analyzing compiler warning defects," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 203–213.

[58] J. Chen, J. Han, P. Sun, L. Zhang, D. Hao, and L. Zhang, "Compiler bug isolation via effective witness test program generation," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 223–234.

[59] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE 17. IEEE Press, 2017, pp. 609–620.

[60] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 197208.

[61] J. Holmes and A. Groce, "Using mutants to help developers distinguish and debug (compiler) faults," *Softw. Test. Verification Reliab.*, vol. 30, no. 2, 2020.

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

[63] J. Chen, Y. Bai, H. Dan, Y. Xiong, H. Zhang, and X. Bing, "Learning to prioritize test programs for compiler testing," in *IEEE/ACM International Conference on Software Engineering*, 2017.

[64] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A survey on compiler autotuning using machine learning," *ACM Computing Surveys (CSUR)*, vol. 51, no. 5, p. 96, 2018.

[65] 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 *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*, W. Pugh and C. Chambers, Eds. ACM, 2004, pp. 171–182.

[66] 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 Transactions on Architecture and Code Optimization (TACO)*, vol. 2, no. 2, pp. 165–198, 2005.

[67] A. Haj-Ali, Q. Huang, W. Moses, J. Xiang, I. Stoica, K. Asanovic, and J. Wawrzynek, "Autophase: Compiler phase-ordering for high level synthesis with deep reinforcement learning," *arXiv preprint arXiv:1901.04615*, 2019.

APPENDIX

In our experiments for RQ1, we first generate testing programs and optimization sequences for CTOS and all baselines. Thus, the testing period does not include the time to obtain testing programs and optimization sequences for CTOS and the baselines. This may be threats to the validity of CTOS since this may cause unfair comparisons between CTOS and the baselines. For example, CTOS takes 3 to 6 hours to generate testing programs and optimizations, while we can quickly generate random testing programs and optimization sequences. To investigate the potential impact of the time to obtain testing programs and optimization sequences on the results for RQ1, we analyse the detected bugs within the first 80 hours out of the testing period 90 hours. That is, we assume that 10 hours are used to obtain testing programs and optimization sequences for CTOS.

Table 1 shows the results for the 10 runs of CTOS within 80 hours. From Table 1, we can see that the majority of bugs can be detected by CTOS within the first 80 hours. Particularly, CTOS can detect 11.7 bugs on average within 80 hours, while it detects 13.1 bugs within the testing period 90 hours. Although the detected bugs within 80 hours are less than those in 90 hours, the results also outperforms all the baselines (see Table 2 in the paper). For example, compared with CTOS(*RP*+*RS*) that detects 9.8 bugs on average within 90 hours, CTOS also find 19.39% more bugs within 80 hours. Also, the time to obtain testing programs and optimization sequences is much less than the total testing period. Thus, we believe that the results for RQ1 cannot be affected dramatically by the time to obtain testing programs and optimization sequences.

TABLE 1
Results of CTOS within the first 80 hours.

ID.	TP.	Crash	WC.	Inv. IR	Perf.	CGB	Total bugs
1	92	7	3	2	1	0	13
2	100	6	4	1	1	0	12
3	97	10	3	0	1	0	14
4	94	6	3	1	1	0	11
5	102	7	3	0	1	0	11
6	101	7	2	0	1	0	10
7	104	7	2	1	1	0	11
8	107	7	2	1	1	0	11
9	90	6	4	2	1	0	13
10	97	7	3	0	1	0	11
Average	98.4	7.0	2.9	0.8	1.0	0	11.7

TP.: Testing Programs, Inv. IR: Invalid IR, WC.: Wrong Code, CGB.: Code Generator Bug.