History-Guided Configuration Diversification for Compiler Test-Program Generation

Junjie Chen[1], Guancheng Wang[2,3], Dan Hao[2,3†], Yingfei Xiong[2,3†], Hongyu Zhang[4], Lu Zhang[2,3]
[1]College of Intelligence and Computing, Tianjin University, Tianjin, China, junjiechen@tju.edu.cn
[2]Key Laboratory of High Confidence Software Technologies (Peking University), MoE
[3]Department of Computer Science and Technology, EECS, Peking University, Beijing, China
{guancheng.wang,haodan,xiongyf,zhanglucs}@pku.edu.cn
[4]The University of Newcastle, NSW, Australia, Hongyu.Zhang@newcastle.edu.au

***Abstract*—Compilers, like other software systems, contain bugs, and compiler testing is the most widely-used way to assure compiler quality. A critical task of compiler testing is to generate test programs that could effectively and efficiently discover bugs. Though we can configure test generators such as Csmith to control the features of the generated programs, it is not clear what test configuration is effective. In particular, an effective test configuration needs to generate test programs that are bug-revealing, i.e., likely to trigger bugs, and diverse, i.e., able to discover different types of bugs. It is not easy to satisfy both properties. In this paper, we propose a novel test-program generation approach, called HiCOND, which utilizes historical data for configuration diversification to solve this challenge. HiCOND first infers the range for each option in a test configuration where bug-revealing test programs are more likely to be generated based on historical data. Then, it identifies a set of test configurations that can lead to diverse test programs through a search method (particle swarm optimization). Finally, based on the set of test configurations for compiler testing, HiCOND generates test programs, which are likely to be bug-revealing and diverse. We have conducted experiments on two popular compilers GCC and LLVM, and the results confirm the effectiveness of our approach. For example, HiCOND detects 75.00%, 133.33%, and 145.00% more bugs than the three existing approaches, respectively. Moreover, HiCOND has been successfully applied to actual compiler testing in a global IT company and detected 11 bugs during the practical evaluation.**

***Index Terms*—Compiler Testing, Configuration, History, Search**

I. Introduction

Compilers are one of the most important software systems, since almost all software systems are built from them. However, like other software systems, compilers also contain bugs [1]–[7]. Due to their crucial roles, buggy compilers could cause unintended behaviors, even disasters for all software systems built from them. Moreover, compiler bugs also increase the debugging difficulty for software developers. For example, when a software system built from a compiler crashes, it is very hard for the developers to determine whether the crash is caused by the software they are developing or the compiler they are using. Therefore, guaranteeing the quality of compilers is very essential.

Compiler testing is the most widely-used way to assure compiler quality and has attracted extensive attention over the years [1], [2], [8]–[13]. In the area of compiler testing, automated test-program generation is an important aspect, since it is the initial step in the testing process and its performance has large impacts on the following testing process [2], [3], [14]. To support the generation of test programs, several test-program generators such as Csmith [1] or CLsmith [15] have been developed. These tools generate a random set of test programs based on a test configuration (consisting of many options) to control what features these test programs are likely to include. For example, the test configuration of Csmith consists of 71 options (such as the probability of the occurrence of `return` statement and `int` type) to control the generation of test programs, where each option directly reflects the probability of a specific program feature (an element of a language grammar) to be included.

An important goal for automated test-program generation is to discover bugs as many as possible. However, it is not easy to achieve this goal, and there are two major challenges. First, as shown in the existing studies [8], [16], compiler bugs are not evenly distributed across the whole input space, and thus it is more important to generate test programs that are more likely to trigger bugs. However, although we can control the test-program generation through a test configuration, it is not clear what configuration would lead to such test programs. Second, it is important to improve the diversity of the generated test programs to cover a wide range of compiler bugs. As justified by the state-of-the-art random compiler test-program generation approach, swarm testing [17], randomizing the configuration options could lead to more bugs being discovered.

In this paper we propose **HiCOND** (**Hi**story-Guided **CON**figuration **D**iversification), a novel test-program generation approach for compilers. HiCOND consists of two stages, the first stage is to produce a set of test configurations, and the second stage is to generate test programs using this set of test configurations. To address the above-mentioned two

This work is supported by the National Key Research and Development Program of China under Grant No.2017YFB1001803, and the National Natural Science Foundation of China under Grant No. 61672047, 61828201, 61861130363, 61872263, 61802275. This work was mainly done when Junjie Chen was at Peking University. Dan Hao, Yingfei Xiong, and Hongyu Zhang are sorted in the alphabet order of the last names.

†Corresponding authors.

challenges, the test configurations produced by our approach HiCOND should be able to generate both bug-revealing and diverse test programs. To address the first challenge, HiCOND utilizes the historical data of compiler testing. More specifically, HiCOND collects test programs that have triggered or have not triggered compiler bugs, mines the associations between program features and compiler bugs, and infers the range for each configuration option to generate test programs with bug-revealing features. To address the second challenge, HiCOND measures the diversity of test programs generated under different test configurations based on their program features, and then uses the particle swarm optimization (PSO), a heuristic search algorithm, to find a set of test configurations that both are able to generate diverse test programs and are within the bug-revealing range of each option. For the ease of presentation, we call such a set of test configurations *a set of bug-revealing and diverse test configurations* in this paper.

We conducted an empirical study to evaluate the effectiveness of our compiler test-program generation approach HiCOND, using two most popular C compilers (five versions in total), i.e., GCC and LLVM, based on the most widely-used test-program generator Csmith [1]. Our experimental results show that, during the given testing period, HiCOND performs significantly better than three comparison approaches, i.e., test-program generation with the default test configuration and the state-of-the-art random testing techniques—swarm testing and its variant, for each compiler subject in terms of the number of detected bugs, the number of unique bugs, and the time spent on detecting each bug. For example, HiCOND detects 75.00%, 133.33%, and 145.00% more bugs than the three comparison approaches for all the used compiler subjects, and 61.22% (30 out of 49) bugs detected by HiCOND cannot be detected by the other three approaches. Moreover, HiCOND spends shorter time than the other three approaches on detecting almost every bug.

Furthermore, HiCOND has been successfully applied to the practical compiler testing in a global IT company $\mathcal{A}$[1], and detected 11 bugs in a private benchmark containing compilers with real bugs developed in $\mathcal{A}$, whereas Csmith with the default configuration detected a subset of 3 bugs during the same testing period.

Our work makes the following major contributions:

- A novel test-program generation approach for compilers through searching a set of bug-revealing and diverse test configurations for existing test-program generators based on testing history.
- Experimental results on GCC and LLVM demonstrating that HiCOND significantly outperforms all the three existing approaches, e.g., detecting 75,00%, 133.33%, and 145.00% more bugs than them.
- Practical effectiveness evaluation of HiCOND for testing the compilers in a global IT company $\mathcal{A}$, detecting 11 bugs during one-week testing.

[1]Due to the company policy, we hide the name of the company.

Our tool and experimental data are publicly available at: **https://github.com/JunjieChen/HiCOND**.

II. Background

A. Compiler Test Program Generation

Over the years, various compiler testing techniques have been proposed, such as RDT (Randomized Differential Testing) [18], DOL (Different Optimization Levels) [3], and EMI (Equivalence Modulo Inputs) [2]. These techniques tend to be used by first randomly generating test programs via certain compiler test-program generator like Csmith [1] and then using their own test oracles to determine whether compiler bugs are detected by these test programs. For example, RDT first uses a generator to randomly generate a test program, and then compares the outputs of several comparable compilers for the test program. If the produced results are different, a compiler bug is detected.

As the initial step of compiler testing, compiler test-program generation has attracted extensive attention [1], [15], [17], [19]–[21]. Many efforts focus on creating test-program generators. These test-program generators typically provide a set of configuration options for the users to configure the features of the generated programs. For example, Csmith [1], the most widely-used C program generator, randomly generates C programs according to a test configuration with 71 options. It generates a C program by conducting a series of decisions, i.e., determining whether a program feature (e.g., a `return` statement) is produced at a decision point. In particular, Csmith introduces some heuristics and safety checks to avoid undefined behaviors. Afterwards, CLsmith [15], a test-program generator for OpenCL compilers, is developed by adapting Csmith. CLsmith contains six modes and can generate different types of OpenCL kernels under different modes. Some research efforts focus on finding effective test generation methods based on these generators. For example, Groce et al. [17] proposed an interesting idea called swarm testing, to generate diverse test programs by randomizing test configurations of test-program generators.

B. Test Configuration

A typical random test-program generator uses a test configuration to randomly generate valid test programs. A test configuration consists of a set of options, each of which directly reflects the probability of a specific program feature to be included. More specifically, during the generation of a test program, there are many decision points for each feature, and at each point the option is used to decide whether the feature is produced here with the corresponding probability. For example, the test configuration used in Csmith includes the following options (i.e., 71 options in total) [1]:

- the generation probability of various kinds of statements, e.g., `return` statement and `if` statement;
- the generation probability of various kinds of types, e.g., `int`, `char`, and `struct`;
- the generation probability of various kinds of operators, e.g., logical operators and arithmetical operators;

- the generation probability of various kinds of modifiers, e.g., `const` and `volatile`;
- some specific options, e.g., the probability to produce more structs and unions, and the probability to produce an inline function;

Under a test configuration, a test-program generator randomly generates a large number of new test programs for compiler testing. Currently, there are two existing ways to set the test configuration:

1) *Default*: A test-program generator has a default test configuration, each option in which has a default value. The default test configuration is heavily tuned according to the experience and knowledge of developers, which tends to be regarded as an optimal test configuration [17].
2) *Swarm Testing*: Swarm testing aims to generate diverse test programs by randomizing test configurations [17]. More specifically, swarm testing randomly sets the value of each option in a test configuration. In this case, a test-program generator generates test programs by first randomly constructing a test configuration and then using the test configuration to generate test programs.

To sum up, the *Default* way is to use a fixed test configuration for a test-program generator to generate test programs, while the *Swarm Testing* way is to randomize test configurations to generate test programs.

III. Approach

An important goal for automated compiler test-program generation is to generate test programs that could lead to more bugs being discovered. For illustration, we shall call the space of test programs that trigger bugs as the *bug space* and the goal is to generate diverse test programs within the bug space. As demonstrated by the existing swarm testing work [17], it is hard to use one uniform test configuration to generate test programs to thoroughly explore bug space. To sufficiently explore bug space, we need a set of test configurations to generate test programs. Since a test configuration can explore only a portion of bug space and there are still many test configurations that even do not enter into bug space at all, each test configuration in the desired set should be able to generate test programs exploring a (large) portion of bug space, which is the first criterion for the desired set. However, it is impossible to enumerate each test configuration that can generate bug-revealing test programs, to thoroughly explore bug space. A more efficient way is to make different test configurations in the desired set be able to explore different portions of bug space. That is, the set of test configurations should have diversity for bug detection, which is the second criterion for the desired set.

To achieve this goal, we propose HiCOND, a novel compiler test-program generation approach via history-guided configuration diversification, which explores the whole bug space as much as possible by finding such a set of test configurations for compiler test-program generation. To satisfy the first criterion, HiCOND determines the range of each option in a test configuration where the bug-revealing test programs are more likely to be generated. Here, we utilize the statistics on the basis of historical data to infer the range for each option. To satisfy the second criterion, HiCOND considers their diversity when constructing the set of test configurations. Here we first propose a method to measure the diversity of test programs generated under different test configurations based on their program features. Then, we use the PSO algorithm to search such a set of test configurations within the bug-revealing range of each option. Figure 1 shows the overview of HiCOND.

In the following, we present the history-based range inference in Section III-A, the diversity measurement in Section III-B, and the PSO-based searching in Section III-C.

A. History-based Range Inference

In this step, we need to find a range for each option that has a high probability of triggering bugs. To do this, we assume the existence of an optimal setting for each option, and try to deduce a range to include the setting. We first introduce the background of options and data collections, then describe three properties that we assume to hold for the optimal setting, and finally give our formula utilizing the three properties.

Background. As presented in the existing work [17], each option in a test configuration controls a specific program feature during test-program generation. An option is usually a probability (i.e., a floating-point number ranging from 0 to 100), which refers to the possibility that the corresponding program feature is successfully produced at each decision point during test-program generation. That is, there exists a mapping between an option in a test configuration and a program feature. Therefore, we can analyze the features in the generated test programs for testing historical bugs to infer the range of each option.

In testing history, there are a huge number of test programs that trigger compiler bugs and those that do not trigger compiler bugs. Here, we call the former *failing test programs*, denoted as $P_F = \{p_{f_1}, p_{f_2}, \ldots, p_{f_m}\}$, and the latter *passing test programs*, denoted as $P_P = \{p_{p_1}, p_{p_2}, \ldots, p_{p_n}\}$, where m and n are the size of the failing and passing program set, respectively. For each test program, we can record the decisions for all program features during test-program generation. Then, a test program can be represented as a feature vector, denoted as $p = \{e_1, e_2, \ldots, e_r\}$, where e_i $(1 \leq i \leq r)$ is the number of times the i_{th} feature is successfully produced at all its decision points and r refers to the number of features.

Based on these feature vectors, we compute the probability that a feature is produced during the generation of all the failing and passing test programs, respectively. The computation is shown as Formula 1, where N is the size of the test program set, e_{ij} refers to the number of times the i_{th} feature is produced at all its decision points during the generation of the j_{th} test program, and t_{ij} refers to the total number of decision points for the i_{th} feature during the generation of the j_{th} test program. Here we denote *Pr* in the Formula 1 on the failing test programs as *Prf* and denote *Pr* on the passing test programs as *Prp*.

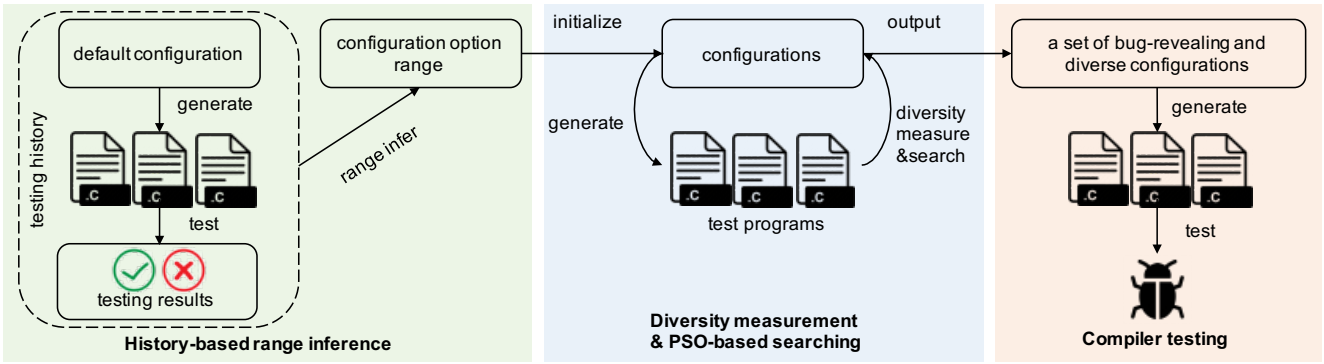


Fig. 1: Overview of HiCOND

$$Pr(i) = \frac{\sum_{j=1}^{N} e_{ij}}{\sum_{j=1}^{N} t_{ij}} \quad (1)$$

Properties of Optimal Setting. Now we assume that for each feature there is an optimal setting that maximizes the probability of producing failing programs, and the further we move away from the optimal setting, the lower the probability of producing failing programs is. Given two features i, j we denote the initial setting of the corresponding options for features i, j as p_i, p_j, and the optimal settings of the options for feature i, j as q_i, q_j. It is easy to see that the following properties hold. In the following properties, we ignore the statistical errors and assume *Prf* and *Prp* to be the precise probabilities.

$\mathcal{A}1$: $q_i \geq p_i$ iff $Prf(i) \geq p_i$; $q_i < p_i$ iff $Prf(i) < p_i$.
When $q_i \geq p_i$, it means that a program where more instances of feature i are produced at decision points has a higher probability to fail, and thus $Prf(i)$ will be within $[p_i, q_i]$, i.e., $Prf(i) \geq p_i$. Same for the case $q_i < p_i$.

$\mathcal{A}2$: Suppose $p_i = p_j$. We have $|q_i - p_i| > |q_j - p_j|$ iff $|Prf(i) - p_i| > |Prf(j) - p_j|$.
Following the above analysis, the further the optimal setting deviates the initial setting, the further the *Prf* value deviates from the initial setting.

Also, we assume the initial setting reflects the confidence of the developer. By default the developer would set the initial setting to 50, the even distribution. The more the developer changes it away from the even distribution, the higher their confidence is. Therefore, we have the following property.

$\mathcal{A}3$: If $|p_i - 50| < |p_j - 50|$, then $|q_i - p_i| > |q_j - p_j|$.
In other words, the closer the initial setting is to 50, the smaller the confidence about the feature is, indicating that the distance between the optimal setting and the initial setting is larger.

Calculating Ranges. Based on the above three properties, we now propose a formula to calculate the ranges for options. Our formula borrows the calculation method of *big support difference* [22] for identifying *emerging patterns* [23], and utilizes the three properties to include the optimal setting as much as possible. More specifically, the range of the i_{th} option $R(i)$ is inferred as follows, where p_i refers to the initial setting of the option for generating these historical data and $diff(i) = Prf(i) - Prp(i)$.

$$R(i) = \begin{cases} [max(0,\ p_i + diff(i)),\ p_i], & diff(i) < 0 \\ [p_i,\ min(100,\ p_i + diff(i))], & diff(i) \geq 0 \end{cases} \quad (2)$$

Now we analyze how the formula utilizes the above three properties. Regarding $\mathcal{A}1$, it is easy to see that $diff(i) < 0$ implies $Prf(i) < p_i$, and thus we set the range to be smaller than p_i. The same for the other direction. Regarding $\mathcal{A}2$, the size of the range depends on $diff(i)$, so when q_i deviates more from p_i, the range also becomes large to include q_i. Regarding $\mathcal{A}3$, *Prf* and *Prp* are calculated based on the generated programs under p_i, so the closer to 0 or 100 p_i is, the more difficult for *Prf* and *Prp* to deviate from p_i, and thus *diff* becomes smaller.

B. Diversity Measurement

Measuring the diversity of test configurations actually refers to measure the diversity of generated test programs under these test configurations. As presented in Section III-A, each generated test program can be represented as a feature vector, and thus HiCOND uses the distance between these feature vectors to measure the diversity of test programs. More specifically, assuming a set of test configurations are denoted as $C = \{c_1, c_2, \ldots, c_g\}$, where g is the number of test configurations and $c_i = \{o_{i1}, o_{i2}, \ldots, o_{ir}\}$, and the generated test programs under c_i are denoted as $P_i = \{p_{i1}, p_{i2}, \ldots, p_{is}\}$, where s is the number of generated test programs under each test configuration, HiCOND computes the diversity among P_i $(1 \leq i \leq g)$ to measure the diversity of test configurations.

Intuitively, the generated test programs under a test configuration tend to concentrate on an area of input space. With this intention, HiCOND first sets a group center for these generated test programs, and then computes the distance between different group centers. For test program set P_i generated under test configuration c_i, HiCOND sets the group center pc_i by computing the mean of each feature on all the generated test programs. Then, for a test configuration HiCOND computes all the distances between the group center

and other group centers, and uses the minimum distance among them as the distance between this test configuration and all other test configurations. The computation is shown in Formula 3, where HiCOND uses the *Manhattan Distance* [24] as the distance formula for $Dist(pc_i, pc_j)$.

$$Diversity(c_i) = min_{j\in[1,g]\&j\neq i}(Dist(pc_i, pc_j)) \quad (3)$$

C. PSO-based Searching

To construct a set of test configurations exploring bug space as much as possible, HiCOND adopts the particle swarm optimization (PSO) algorithm [25] to search for an optimal set of test configurations. The reason why we choose PSO is that PSO is very effective to search in a continuous space [26], [27], which aligns to our problem where we search the floating-point probability of each option also in continuous space.

In our problem, each particle in PSO is a test configuration c_i, and our expected output is a set of diverse test configurations exploring the whole bug space. Therefore, we hope particles to fly to different portions of bug space during the searching process. According to this intention, we define the fitness function of the PSO-based searching as the diversity among test configurations, which is computed based on Section III-B. The larger the fitness value of a test configuration is, the larger diversity the test configuration has with other test configurations.

During the PSO-based searching process, HiCOND first initiates a set of particles (i.e., test configurations). Since HiCOND should search the test configurations under which the generated test programs are more likely to trigger compiler bugs, it sets the search space for each option based on the inferred range in Section III-A. In particular, each particle has its own velocity for each moment $v_i^t = \{v_{i1}^t, v_{i2}^t, \ldots, v_{ir}^t\}$, where t refers to the t_{th} moment. Also, we denote a particle for the t_{th} moment as $c_i^t = \{c_{i1}^t, c_{i2}^t, \ldots, c_{ir}^t\}$. In each moment, HiCOND utilizes the fitness function to evaluate the quality of each particle, and then updates each particle including its velocity. Formula 4 shows the updating of the velocity. In this formula, ω, ς_1, and ς_2 are three weights where ω refers to the inertia weight and the other two weights refer to the acceleration factors; γ_1 and γ_2 are two random numbers between 0 and 1; b_{ij}^t refers to the j_{th} option of the personal optimum test configuration that the i_{th} test configuration has reached before moment t; and b_{gj}^t refers to the j_{th} option of the global optimum test configuration that all test configurations have reached before moment t; Please note that optimum here refers to the largest fitness values. Then the test configuration can be updated as shown in Formula 5.

$$v_{ij}^{t+1} = \omega v_{ij}^t + \varsigma_1\gamma_1(b_{ij}^t - c_{ij}^t) + \varsigma_2\gamma_2(b_{gj}^t - c_{ij}^t) \quad (4)$$

$$c_{ij}^{t+1} = c_{ij}^t + v_{ij}^{t+1} \quad (5)$$

Each particle (i.e., test configuration) keeps changing as above, and finally when reaching the terminating condition (i.e., the pre-defined number of iterations/moments), a set of bug-revealing and diverse test configurations are found.

Finally, HiCOND utilizes the set of test configurations to generate test programs for testing compilers. Since only one test configuration can be used when generating a test program, HiCOND randomly selects a test configuration from the set *every time* to generate a test program.

IV. Experimental Study Design

In the study, we address the following research questions:

- **RQ1**: How does HiCOND perform compared with existing compiler test-program generation approaches?
- **RQ2**: Does HiCOND perform well in different scenarios (including cross-version and cross-compiler scenarios)?
- **RQ3**: Does each component contribute to HiCOND?

A. Subjects and Test Programs

In the study, we used two popular C compilers as subjects, i.e., GCC and LLVM, following the existing compiler testing research [1], [3], [8], [14], [28]–[30]. More specifically, we used three versions of GCC compilers and two versions of LLVM compilers for the x86 64-Linux platform, i.e., GCC-4.4.0, GCC-4.5.0, GCC-4.6.0, LLVM-2.6, and LLVM-6.0.1. On average, the size of GCC is 1,411K SLOC (source lines of code) and the size of LLVM is 1,470K SLOC. In particular, these used compiler versions include both old release versions and a recent release version. The reason is that the older releases usually contain more bugs and give more statistically significant results, while we also used a recent release of LLVM to investigate whether HiCOND still works well for a new release version. Besides, HiCOND relies on historical data. In the study, we collected historical data on GCC-4.3.0, which is released before all the subjects in our study. Here the collected historical data include the test programs triggering bugs and the test programs not triggering bugs on GCC-4.3.0. Please note that all the used historical bugs on GCC-4.3.0 were fixed before all the used subjects in our study are released.

B. Tools and Implementations

In our study, we used Csmith [1], the most widely-used C program generator [3], [8], [17], [20], [31], as the studied random test-program generator. As described in Section II-B, Csmith uses a test configuration with 71 options to control the test-program generation. It takes a test configuration file as input and then generates test programs based on the given test configuration in the file. When collecting the historical data on GCC-4.3.0, we used Csmith with its default test configuration to generate test programs, which are divided into two sets (i.e., failing test program set and passing test program set) according to their testing results. Here we used the DOL technique to test compilers, which is one of the most widely-used compiler testing techniques [3], [8], [28]. That is, if a test program produces different outputs under different optimization levels of a compiler given the same test inputs, a compiler bug is detected and this test program is a bug-revealing test program.

To record all the decisions of program features during test-program generation for HiCOND, we adapted Csmith to output all the decisions when generating a test program. For the PSO algorithm used in HiCOND, we set ς_1 and ς_2 to be 2, ω to be 1, the number of particles to be 10, the times of iterations to be 100, and the number of generated test programs under each test configuration s to be 350. In particular, we investigated the impact of main parameters in PSO on HiCOND in Section VIII. In addition, we set the testing period to be 10 days in our study. We implemented HiCOND and all experimental scripts using C++, Python, and Perl. Our study was conducted on a workstation with four-core CPU, 120G memory, and Ubuntu 14.04 operating system.

C. Comparison Approaches

HiCOND is a compiler test-program generation approach, which searches a set of bug-revealing and diverse test configurations of random test-program generators. Therefore, we chose the comparison approaches that also utilize test configurations of random test-program generators to control compiler test-program generation.

As described in Section II-B, there are two ways to set a test configuration for test-program generation, i.e., *Default* and *Swarm Testing*. For swarm testing, its original version proposed in the paper [17] randomly sets the value of each configuration option to be 0 or 100. In this paper we also consider a natural variant of swarm testing that randomly sets the value of each option to be a floating-point number ranging from 0 to 100. For the ease of presentation, we call the approach utilizing the *Default* way **DefaultTest**, that utilizing the original *Swarm Testing* way **OriSwarm**, and that utilizing the variant *Swarm Testing* way **VarSwarm** in this paper.

There are two key components in HiCOND, i.e., range inference and PSO-based searching. To investigate whether each component contributes to HiCOND, we have two variants of HiCOND. The first variant is HiCOND$_{global}$, which removes range inference from HiCOND and conducts PSO-based searching globally (i.e., ranging from 0 to 100 for each option). The second variant is HiCOND$_{random}$, which removes PSO-based searching from HiCOND and conducts random searching within the inferred range for each option. To answer RQ3, we compared HiCOND with the two variants.

D. Application Scenarios

Here we consider two application scenarios of HiCOND, including cross-version scenario and cross-compiler scenario.

Cross-version scenario: HiCOND searches a set of bug-revealing and diverse test configurations based on the historical data of a compiler version and then uses the set of test configurations to generate test programs for a later version of the compiler. In our study, HiCOND used the data of GCC-4.3.0 to search test configurations and then used the searched test configurations to generate test programs for GCC-4.4.0, GCC-4.5.0, and GCC-4.6.0, respectively. Here we considered a series of later versions to investigate the stability of the test configurations searched by HiCOND.

Cross-compiler scenario: HiCOND searches a set of bug-revealing and diverse test configurations based on the historical data of a compiler and then uses the set of test configurations to generate test programs for another compiler. In our study, HiCOND also used the data of GCC-4.3.0 to search test configurations and then used the searched test configurations to generate test programs for LLVM-2.6 and LLVM-6.0.1, respectively. In this way, we can evaluate the generality of HiCOND cross different compilers.

E. Metrics

In our study, we used two metrics to measure the effectiveness of compiler test-generation approaches. The first one is the number of bugs detected during the given testing period. We adopted *Correcting Commits* [3], a method commonly used in existing studies [8], [32], to identify the number of detected bugs from a set of failing test programs. More specifically, for each failing test program, the *Correcting Commits* method is to find the first commit correcting the bug, i.e., the committed version makes the test program pass. If two failing test programs have the same correcting commit, they are regarded as triggering the same bug. The number of correcting commits is approximately regarded as the number of detected bugs. The accuracy of this method has been demonstrated in the existing study [3]. The second one is the time spent on detecting a compiler bug. This metrics can reflect the performance of HiCOND on detecting each bug.

F. Experimental Process

First, we collected historical data on GCC-4.3.0 by running 20,000 test programs generated by the adapted Csmith with the default test configuration. In this way, we collected a set of failing test programs (around 4000) and a set of passing test programs (around 16,000). Also, we recorded all the decisions of program features during the generation of each test program.

Second, we applied HiCOND, HiCOND$_{global}$, and HiCOND$_{random}$ to search a set of test configurations and fed the set of test configurations to generate test programs to test GCC-4.4.0, GCC-4.5.0, GCC-4.6.0, LLVM-2.6, and LLVM-6.0.1 for 10 days, respectively. We recorded the testing result and execution time for each test program.

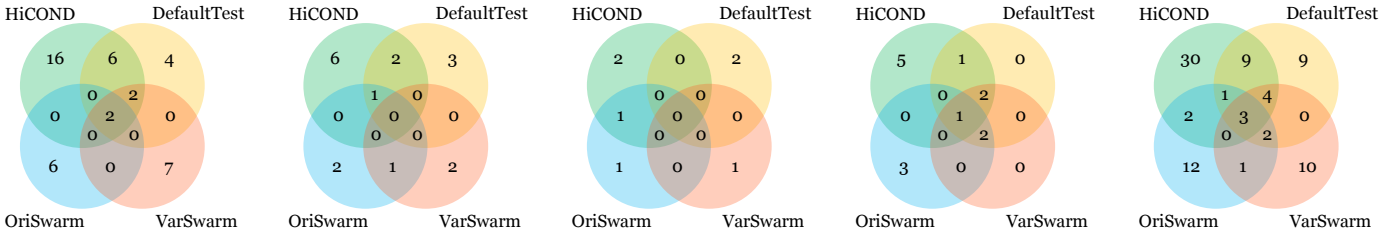
Third, we also applied DefaultTest, OriSwarm, and VarSwarm to test each subject for 10 days, and recorded the testing result and execution time for each test program.

Finally, we measured the number of detected bugs for each test-program generation approach. In particular, it took us over half of a year to run our experiments.

V. Results and Analysis

A. Overall Effectiveness of HiCOND

We analyzed the overall effectiveness of HiCOND compared with the three comparison approaches from three aspects, including the number of detected bugs, the number of unique bugs, and the time spent on detecting each bug.



(a) GCC-4.4.0 (b) GCC-4.5.0 (c) GCC-4.6.0 (d) LLVM-2.6 (e) ALL

Fig. 2: Number of unique bugs

TABLE I: Number of detected bugs within 10 days

Subject	HiCOND	DefaultTest	OriSwarm	VarSwarm
GCC-4.4.0	**26**	14	8	11
GCC-4.5.0	**9**	6	4	3
GCC-4.6.0	**3**	2	2	1
LLVM-2.6	**9**	6	6	5
LLVM-6.0.1	**2**	0	1	0
Total	**49**	28	21	20

1) Number of detected bugs: Table I shows the number of detected bugs during the given 10-day testing period. From this table, for each subject, HiCOND detected the largest number of bugs among the four approaches, demonstrating the effectiveness of HiCOND. In particular, the total number of detected bugs by HiCOND is 49, which is much larger than that by DefaultTest (i.e., 28), OriSwarm (i.e., 21), and VarSwarm (i.e., 20) achieving 75.00%, 133.33%, and 145.00% improvements, respectively. From the results of LLVM-6.0.1, a recent release version of LLVM, HiCOND also detects the largest number of bugs, and DefaultTest and VarSwarm do not detect any bug during the given 10-day testing period, indicating that HiCOND still works on the new release version. Therefore, HiCOND does significantly outperform all the comparison approaches.

Although OriSwarm and VarSwarm also consider the diversity of test configurations, they perform worse than HiCOND. We analyzed the reason behind this phenomenon. For OriSwarm, it uses 0 or 100 to replace a floating-point number for each option, which makes all the decisions of a feature the same during the generation of a test program. Although it increases the diversity of test configurations, it actually limits the diversity of generated test programs to some degree. For VarSwarm, due to its random mechanism of test-configuration construction, it often produces test configurations that outside our inferred ranges. A small experiment on 100 configurations show that in 100% of the time the generated options are outside the range inferred by HiCOND. As our experiment for RQ3 will show later, ignoring the ranges lead to significantly lower performance.

2) Number of unique bugs: These Venn diagrams in Figure 2 show various relations among the bugs detected by the four approaches, e.g., the unique bugs and the common bugs detected by only two approaches. Here we do not show the results of LLVM-6.0.1 since only HiCOND and OriSwarm detected bugs on this subject, and there is one common bug for them and one unique bug for HiCOND. Figure 2e shows the overall results on all the subjects. From this figure, each approach is able to detect some unique bugs, but HiCOND always detects the largest number of unique bugs. In particular, from Figure 2e, the total number of unique bugs detected by HiCOND is 30, which is much larger than that by DefaultTest (i.e., 9), that by OriSwarm (i.e., 12), and that by VarSwarm (i.e., 10). Also, 61.22% (30 out of 49) bugs detected by HiCOND are unique. That demonstrates that the four approaches are able to complement each other to some degree and HiCOND has the largest unique value among them.

Interestingly, although OriSwarm and VarSwarm detected fewer bugs, more than half of these bugs are not detected by the other approaches. The reason is that OriSwarm and VarSwarm have the ability to explore a larger portion of input space than the other approaches due to their random mechanisms of test-configuration construction. Therefore, they can reach a certain bug space that is not explored by the other approaches, respectively. Although HiCOND aims to explore the whole bug space, in fact it is very hard to perfectly achieve this goal within the given testing period. Therefore, it may miss a portion of bug space to explore. That also means that there is still room for further improving HiCOND.

Furthermore, there is also some unique bugs detected by DefaultTest. The reason is that DefaultTest always focuses on certain small portion of bug space, and thus it is more likely to sufficiently explore this portion during the given testing period. However, the other three approaches have larger exploration space, and thus they may not sufficiently explore the portion focused by DefaultTest, causing to miss the detection of some bugs. To sum up, there exists a trade-off between the deep exploration for a small portion of bug space and the wide exploration for the whole bug space, and HiCOND seems to reach a good balance for it.

3) Time spent on detecting each bug: Table II shows the time spent on detecting each bug for each approach. In this table, Column "Bug" refers to each bug, e.g., "1" refers to the first detected bug and "2" refers to the second detected bug; Column "HiCOND" refers to the time spent on detecting each bug by HiCOND; Columns "ΔDefaultTest", "ΔOriSwarm", and "ΔVarSwarm" refer to the difference of

the time spent on detecting each bug between the comparison approach and HiCOND, where a positive value means the comparison approach spent longer time on detecting the bug than HiCOND while a negative value means it spent shorter time on detecting the bug. Note that since different approaches detected different number of bugs, we used "—" to align them in the table. For example, for the 15_{th} bug in GCC-4.4.0, it is only detected by HiCOND, and thus we marked "—" in the cells of ΔDefaultTest, ΔOriSwarm, and ΔVarSwarm. From this table, for almost all of bugs HiCOND spent the shortest time on detecting the bug. There are only four bugs that HiCOND spent longer time than DefaultTest to detect, and only one bug that HiCOND spent longer time than OriSwarm and VarSwarm to detect, respectively. That demonstrates that HiCOND is able to detect almost every bug more efficiently, indicating the stably good effectiveness of HiCOND.

B. Effectiveness on Different Application Scenarios

We analyzed the effectiveness of HiCOND in different application scenarios according to Tables I, II and Figure 2. Since the used historical data by HiCOND is from GCC-4.3.0, the results of the three GCC versions reflect the effectiveness of HiCOND in cross-version scenario, and the results of the two LLVM versions reflect the effectiveness of HiCOND in cross-compiler scenario.

Through analyzing the effectiveness of HiCOND on three GCC versions, we found that HiCOND always performs the best among the four approaches, demonstrating its stably good effectiveness cross different versions. That means that it is not necessary to re-search test configurations using HiCOND for each new version. Also, we found that the improved effectiveness of HiCOND compared with the other approaches is reduced, with the gap between the historical version and testing version increasing. The reason may be either the number of bugs in new versions becomes smaller or the capability of the searched test configurations becomes weaker due to larger gap. Assuming the latter is the reason, we may boost the capability of HiCOND by adding some new data on the versions that are closer to the testing version.

Through analyzing the effectiveness of HiCOND on two LLVM versions, HiCOND also achieves better results than the other approaches. That demonstrates the cross-compiler ability of HiCOND. That is, HiCOND is able to be generalized to different compilers even though it is based on the historical data on only one compiler.

In summary, HiCOND does perform well in both of cross-version and cross-compiler scenarios.

C. Contributions of Each Component in HiCOND

Table III shows the comparison results among HiCOND, HiCOND$_{global}$, and HiCOND$_{random}$ in terms of the number of detected bugs during the 10-day testing period. From this table, HiCOND significantly outperforms HiCOND$_{global}$, demonstrating the contribution of the range inference component in HiCOND. When conducing PSO-based searching globally, HiCOND$_{global}$ can find a set of diverse test configurations for the whole input space. However, it is very difficult to ensure these test configurations to be bug-revealing, since the bug space tends to be only a small part of the whole input space.

In addition, HiCOND performs better than HiCOND$_{random}$, demonstrating the contribution of the PSO-based searching component in HiCOND. When randomly searching a set of test configurations, it is hard to ensure the diversity of these test configurations. That is, some test configurations searched by HiCOND$_{random}$ may explore the similar portion of the bug space, leading to the worse effectiveness. To sum up, both of the components significantly contribute to HiCOND, and the range inference component makes more contributions.

VI. Industrial Evaluation

HiCOND has been successfully applied to the practical compiler testing in a global IT company $\mathcal{A}$. This company has several their own compilers, and a lot of products in company $\mathcal{A}$ are built from them. Therefore, guaranteeing the quality of these compilers is very important and efficient compiler testing is in demand. HiCOND aims to efficiently explore the whole bug space as much as possible, which admirably serves their needs.

In company $\mathcal{A}$, before integrating a new tool into the practical testing infrastructure, an effectiveness evaluation of the tool is necessary. In the practical evaluation of HiCOND, they used six widely-used compilers developed by company $\mathcal{A}$ with several known real bugs as subjects. In particular, these compilers are in different application areas and different platforms, the total size of them is over 9 million SLOC, and these bugs have different characteristics. HiCOND achieved great effectiveness for these diverse industrial compilers. More specifically, HiCOND detected bugs for each of these compilers and detected 11 bugs in total during one-week testing. During the same testing period, Csmith with the default configuration detected 3 bugs, all of which were also detected by HiCOND. The results demonstrate that HiCOND is effective in practice. Also, HiCOND is largely appreciated by the compiler-testing team of company $\mathcal{A}$ according to the practical evaluation.

VII. Discussion

A. Generality of HiCOND

HiCOND is a general approach. First, HiCOND is able to be generalized to compilers of other programming languages. Although in our study we evaluated the effectiveness of HiCOND on only C compilers, there is no part in HiCOND specific to C. The input of HiCOND is a random test-program generator that can be controlled by a test configuration. If a compiler of certain programming language contains such a random test-program generator, it is easy to apply HiCOND to it by running the generator to collect a collection of historical data. Therefore, HiCOND can facilitate testing of any compiler with such a random test-program generator.

Besides, HiCOND is also able to be generalized to other software systems taking structurally complex test inputs such as operating systems and browsers. There are two conditions

TABLE II: Time spent on detecting each compiler bug (*10^3 seconds)

Subject	Bug	HiCOND	ΔDefaultTest	ΔOriSwarm	ΔVarSwarm	Subject	Bug	HiCOND	ΔDefaultTest	ΔOriSwarm	ΔVarSwarm
GCC-4.4.0	1	1.01	-0.23	0.08	-0.53	GCC-4.5.0	1	106.54	48.46	-103.27	159.6
	2	1.11	21.4	10.09	0.27		2	133.77	32.72	8.93	133.33
	3	4.2	20.42	7.5	12.04		3	158.98	76.22	50.5	279.27
	4	8.79	45.73	3.04	92.23		4	194.11	141.02	215.15	—
	5	12.51	161.39	5.03	180.07		5	243.92	392.72	—	—
	6	31.13	149.6	60.83	177.49		6	782.8	-75.3	—	—
	7	80.17	174.9	581.82	150.09		7	802.59	—	—	—
	8	83.3	242.75	724.61	184.9		8	812.15	—	—	—
	9	83.66	292.83	—	305.33		9	821.36	—	—	—
	10	88.58	288.8	—	695.6	GCC-4.6.0	1	270.65	51.11	29.6	239.38
	11	129.65	393.31	—	670.0		2	356.88	137.35	190.8	—
	12	211.81	327.53	—	—		3	383.09	—	—	—
	13	225.54	338.59	—	—	LLVM-2.6	1	1.43	-0.9	2.27	0.81
	14	226.91	532.26	—	—		2	1.5	0.9	2.39	2.29
	15	267.41	—	—	—		3	4.37	9.8	44.84	41.7
	16	307.41	—	—	—		4	18.57	-0.08	43.73	64.29
	17	321.49	—	—	—		5	34.11	93.41	79.39	112.47
	18	407.36	—	—	—		6	70.69	66.22	92.99	—
	19	568.81	—	—	—		7	103.92	—	—	—
	20	575.29	—	—	—		8	370.45	—	—	—
	21	595.19	—	—	—		9	557.51	—	—	—
	22	727.57	—	—	—	LLVM-6.0.1	1	328.35	—	332.81	—
	23	765.37	—	—	—		2	793.60	—	—	—
	24	777.54	—	—	—						
	25	778.78	—	—	—						
	26	815.89	—	—	—						

TABLE III: Comparison among HiCOND, HiCOND$_{global}$, and HiCOND$_{random}$

Subject	HiCOND	HiCOND$_{global}$	HiCOND$_{random}$
GCC-4.4.0	**26**	3	14
GCC-4.5.0	**9**	1	3
GCC-4.6.0	3	1	**4**
LLVM-2.6	**9**	5	8
LLVM-6.0.1	**2**	0	0
Total	**49**	10	29

applying HiCOND to other software systems: 1) the software system has such a random test generator; 2) the test inputs of the software system have features relevant to the options of the test configuration. There are a large number of software systems (e.g., operating systems and browsers) taking structure complex test inputs, which tend to contain many features, and thus HiCOND can be applied to them to improve their testing.

B. Limitation of HiCOND

Our study has demonstrated the effectiveness of HiCOND for compiler testing, but there exists a limitation in HiCOND. HiCOND currently treats each option individually. However, there may exist various constraints among program features, and thus different options may have interactions. Also, as presented in the existing work [8], various combinations of program features may be also related to compiler bug detection. Therefore, neglecting the coupling effect may affect the effectiveness of HiCOND. Even though HiCOND does not explicitly consider interactions of options, HiCOND measures diversity based on the generated test program under constraints, which can be regarded as an implicit way. In fact, the constraints of various program features may be very complex, and thus it is challenging to consider all of them. In the future, we will first consider the constraints formed by two program features.

VIII. Threats to Validity

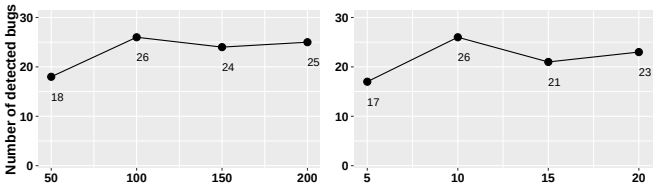
The *internal* threat to validity mainly lies in *our implementations, including HiCOND, swarm testing, and the experimental scripts*. For swarm testing, we re-implemented it based on the description in the paper [17]. To reduce this threat, the first two authors carefully checked all the code.

The *external* threats to validity mainly lie in *the subjects, the studied random test-program generator, and the used technique for compiler testing*. Regarding the subjects, we used five versions of two compilers as subjects, and these subjects may not be representative enough for different compilers. To reduce this threat, we selected the two most popular C compilers following the existing studies [1], [3], [8], [14], [16], [28], [33]–[35]. More specifically, we considered different versions of different compilers and both old and recent release versions, to evaluate the effectiveness of HiCOND from various aspects. Furthermore, our industrial evaluation indicates that our results hold on other compilers. Regarding the studied random test-program generator, we only used Csmith in our study, which may not represent other random test-program generators. However, this threat may not be serious due to the following reasons. First, Csmith is recognized as the most effective C program generator and has been widely used to test C compilers [1], [4]. Second, all recent C compiler testing studies used only the Csmith test-program generator [2]–[5], [8], [14]. Third, many other random test-program generators are adapted from the Csmith generator [2], [9], [15]. Regarding the used technique for compiler testing, we used the DOL technique in our study. However, we believe that HiCOND can be generalized to various compiler testing techniques such as RDT [18] and EMI [2]. This is because all these compiler

testing techniques first utilize a test-program generator like Csmith to generate test programs and then use their test-oracle mechanisms to detect compiler bugs. HiCOND is a novel test-program generation approach that is orthogonal to these compiler testing techniques. Therefore, HiCOND can be combined with any compiler testing technique and improve their performance.

The *construct* threats to validity mainly lie in *the metrics, randomness, the given testing period, the collected historical data, the used comparison approaches, and the setting of parameters in HiCOND*. Regarding the used metrics, we adopted the Correcting Commits method to estimate the number of detected bugs. This method may not be perfectly precise, but it is the only metric which can automatically measure the number of bugs with some precision so far [3]. Regarding randomness, since these test-program generation approaches generate test programs randomly based on a test configuration, the randomness may impact the effectiveness of these approaches. To reduce the threat, we use a long testing period, instead of repeating the testing process several times. Regarding the testing period, we set the period to be 10 days, which is relatively long compared with the existing work [3], [8]. Also, we analyzed the time spent on detecting each bug so that we can learn whether HiCOND performs well during various testing periods within 10 days. Regarding the collected historical data, we ran 20,000 test programs on GCC-4.3.0 to collect historical data, and the used historical compiler and the number of test programs may impact the effectiveness of HiCOND. In the future, we will scale up our historical data to evaluate the effectiveness of HiCOND and investigate the impact of different historical data on HiCOND. Regarding the used comparison approaches, we adopted three existing approaches and two variants of HiCOND. In the future, we will try to use other search-based algorithms to investigate the impact of our used PSO algorithm.

Regarding the parameter setting, we set them following existing work [36]–[38]. However, the setting may impact the effectiveness of HiCOND. To reduce this threat, we evaluated the impacts of two main parameters (i.e., the number of particles and the times of iterations for PSO) on HiCOND by using GCC-4.4.0. Here we changed the value of one parameter each time and remained the values of all other parameters unchanged. Figure 3 shows the impacts of the two parameters, where the x-axis represents the various used parameter values. First of all, HiCOND can always detect more bugs than the three comparison approaches under all these studied parameter values during the 10-day testing period, demonstrating the stable effectiveness of HiCOND. From Figure 3, the default setting in HiCOND indeed performs better than the other settings, indicating the good choice of parameter values in HiCOND. Also, when the value is set to be the smallest for the two parameters, HiCOND performs the worst, since HiCOND cannot sufficiently explore the bug space in these cases.



(a) Times of iterations (b) Number of particles

Fig. 3: Impact of parameters on HiCOND

IX. Related Work

A. Compiler Testing

In the area of compiler testing, test-program generation attracts the most attention all the time [1], [15], [19], [39], [40], and our work also targets at this topic. The most related work to ours is swarm testing, which facilitates test program generation by randomizing test configuration of a test-program generator. Different from it, our work utilizes historical data to search a set of bug-revealing and diverse test configurations for test-program generation. Afterward, Alipour et al. [20] proposed directed swarm testing to generate test programs focusing on a given compiler code element, making the targeted code element be tested more frequently, which also uses historical data. However, our work has a different goal with it. Our work aims to explore the whole bug space while the latter focuses on the testing of the given code element.

Besides test-program generation from scratch, some other test-program generation is to mutate existing test programs. For example, EMI [2] is to generate an equivalent program variant with the original test program given a set of test inputs by mutating the original test program, and then use the program pairs to test compilers. EMI has three instantiations, including Orion [2], Athena [14], and Hermes [29].

Furthermore, some existing work focuses on prioritizing test programs [8], [32]. Chen et al. [32] proposed a text-vector based test prioritization approach for compilers. Their approach transforms each test program to a text vector by extracting a set of characteristics of programs, and then ranks them based on their distances between vectors. Afterwards, Chen et al. [8] proposed a machine-learning based test prioritization approach, called LET. LET first learns two models based on historical data, including a capability model and a time model. Then LET prioritizes test programs based on the two models. Our work is also based on historical data. Different from LET, our work targets at compiler test-program *generation* rather than compiler test-program *prioritization*. More specifically, our work aims to search a set of bug-revealing and diverse test configurations for test-program generation.

B. History-based and Search-based Software Testing

History information has been used to solve software testing problems [8], [41]–[49]. For example, Kim and Porter [41] proposed to prioritize tests in resource constrained environments based on test execution history. Different from them, HiCOND uses program features of *historical* passing and

failing test programs to infer ranges of configuration options, where it is more likely to generate test programs triggering compiler bugs. Search-based software testing refers to transform the problem of software testing to a search problem, and then use a search algorithm to solve it [50]. In the literature, there are many search-based test generation approaches [51]–[57]. For example, Fraser and Arcuri [58] proposed Evosuite, a search-based unit test generation tool for Java projects based on evolutionary search. Different from them, our work targets at *test program generation for compilers*, where we proposed HiCOND to search a set of bug-revealing and diverse test configurations for better test-program generation.

X. Conclusion

In this paper, we propose a novel test-program generation approach via history-guided configuration diversification, called HiCOND. HiCOND first utilizes historical data to infer the range of each configuration option of a test-program generator where bug-revealing test programs are more likely to be generated. It then utilizes the PSO algorithm to search a set of diverse test configurations within the inferred ranges. Finally, HiCOND generates test programs using the identified test configurations for compiler testing. In this way, we can obtain test programs that are more likely to explore the whole bug space. We conducted experiments to evaluate the effectiveness of HiCOND in testing GCC and LLVM. The results show that HiCOND significantly outperforms the existing approaches. Furthermore, HiCOND has been successfully applied to actual compiler testing in company $\mathcal{A}$, and detected 11 bugs in production environment.

References

[1] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *PLDI*, 2011, pp. 283–294.

[2] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *PLDI*, 2014, p. 25.

[3] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "An empirical comparison of compiler testing techniques," in *ICSE*, 2016, pp. 180–190.

[4] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *PLDI*, vol. 48, no. 6, 2013, pp. 197–208.

[5] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for c compiler bugs," in *PLDI*, vol. 47, no. 6, 2012, pp. 335–346.

[6] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su, "Perses: Syntax-guided program reduction," in *ICSE*, 2018, to appear.

[7] J. Chen, J. Han, P. Sun, L. Zhang, D. Hao, and L. Zhang, "Compiler bug isolation via effective witness test program generation," in *FSE*, 2019, pp. 223–234.

[8] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, and B. Xie, "Learning to prioritize test programs for compiler testing," in *ICSE*, 2017, pp. 700–711.

[9] A. F. DONALDSON, H. EVRARD, A. LASCU, and P. THOMSON, "Automated testing of graphics shader compilers," in *OOPSLA*, 2017, pp. 93:1–93:29.

[10] E. Nagai, H. Awazu, N. Ishiura, and N. Takeda, "Random testing of c compilers targeting arithmetic optimization," in *Workshop on Synthesis And System Integration of Mixed Information Technologies*, 2012, pp. 48–53.

[11] I. Hussain, C. Csallner, M. Grechanik, Q. Xie, S. Park, K. Taneja, and B. M. M. Hossain, "RUGRAT: evaluating program analysis and testing tools and compilers with large generated random benchmark applications," *Softw., Pract. Exper.*, vol. 46, no. 3, pp. 405–431, 2016.

[12] S. Souto, M. d'Amorim, and R. Gheyi, "Balancing soundness and efficiency for practical testing of configurable systems," in *ICSE*, 2017, pp. 632–642.

[13] S. Souto and M. d'Amorim, "Time-space efficient regression testing for configurable systems," *JSS*, vol. 137, pp. 733–746, 2018.

[14] V. Le, C. Sun, and Z. Su, "Finding deep compiler bugs via guided stochastic program mutation," in *OOPSLA*, 2015, pp. 386–399.

[15] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," in *PLDI*, 2015, pp. 65–76.

[16] C. Sun, V. Le, Q. Zhang, and Z. Su, "Toward understanding compiler bugs in gcc and llvm," in *ISSTA*, 2016, pp. 294–305.

[17] 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*. ACM, 2012, pp. 78–88.

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

[19] A. S. Boujarwah and K. Saleh, "Compiler test case generation methods: a survey and assessment," *IST*, vol. 39, no. 9, pp. 617–625, 1997.

[20] M. A. Alipour, A. Groce, R. Gopinath, and A. Christi, "Generating focused random tests using directed swarm testing," in *ISSTA*, 2016, pp. 70–81.

[21] E. Nagai, A. Hashimoto, and N. Ishiura, "Scaling up size and number of expressions in random testing of arithmetic optimization of c compilers," in *Workshop on Synthesis And System Integration of Mixed Information Technologies*, 2013, pp. 88–93.

[22] S. D. Bay and M. J. Pazzani, "Detecting change in categorical data: Mining contrast sets," in *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 1999, pp. 302–306.

[23] G. Dong and J. Li, "Efficient mining of emerging patterns: Discovering trends and differences," in *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 1999, pp. 43–52.

[24] M. Kokare, B. Chatterji, and P. Biswas, "Comparison of similarity metrics for texture image retrieval," in *TENCON 2003. Conference on Convergent Technologies for the Asia-Pacific Region*, vol. 2. IEEE, 2003, pp. 571–575.

[25] R. Poli, J. Kennedy, and T. Blackwell, "Particle swarm optimization," *Swarm intelligence*, vol. 1, no. 1, pp. 33–57, 2007.

[26] J. Zhang, J. Chen, D. Hao, Y. Xiong, B. Xie, L. Zhang, and H. Mei, "Search-based inference of polynomial metamorphic relations," in *ASE*, 2014, pp. 701–712.

[27] B. Zhang, H. Zhang, J. Chen, D. Hao, and P. Moscato, "Automatic discovery and cleansing of numerical metamorphic relations," in *ICSME*, 2019, to appear.

[28] C. Sun, V. Le, and Z. Su, "Finding and analyzing compiler warning defects," in *ICSE*, 2016, pp. 203–213.

[29] ——, "Finding compiler bugs via live code mutation," in *OOPSLA*, 2016, pp. 849–863.

[30] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, and L. Zhang, "Static duplicate bug report identification for compilers," *SCIENCE CHINA Information Sciences*, 2019, to appear.

[31] 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, to appear.

[32] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "Test case prioritization for compilers: A text-vector based approach," in *ICST*, 2016, pp. 266–277.

[33] V. Le, C. Sun, and Z. Su, "Randomized stress-testing of link-time optimizers," in *ISSTA*, 2015, pp. 327–337.

[34] A. Groce, M. A. Alipour, C. Zhang, Y. Chen, and J. Regehr, "Cause reduction: delta debugging, even without bugs," *STVR*, vol. 26, no. 1, pp. 40–68, 2016.

[35] ——, "Cause reduction for quick testing," in *ICST*, 2014, pp. 243–252.

[36] R. Eberhart and J. Kennedy, "A new optimizer using particle swarm theory," in *MHS*, 1995, pp. 39–43.

[37] Y. Shi and R. Eberhart, "A modified particle swarm optimizer," in *1998 IEEE International Conference on Evolutionary Computation Proceedings*, 1998, pp. 69–73.

[38] J. Kennedy, "Particle swarm optimization," in *Encyclopedia of machine learning*, 2011, pp. 760–766.

[39] Q. Zhang, C. Sun, and Z. Su, "Skeletal program enumeration for rigorous compiler testing," in *PLDI*, 2017, pp. 347–361.

[40] A. Groce, C. Zhang, M. A. Alipour, E. Eide, Y. Chen, and J. Regehr, "Help, help, i'm being suppressed! the significance of suppressors in software testing," in *ISSRE*, 2013, pp. 390–399.

[41] J.-M. Kim and P. Adam, "A history-based test prioritization technique for regression testing in resource constrained environments," in *ICSE*, 2002, pp. 119–129.

[42] J. Chen, "Learning to accelerate compiler testing," in *ICSE*, 2018, pp. 472–475.

[43] Y.-C. Huang, K.-L. Peng, and C.-Y. Huang, "A history-based cost-cognizant test case prioritization technique in regression testing," *JSS*, vol. 85, no. 3, pp. 626–637, 2012.

[44] E. Engström, P. Runeson, and A. Ljung, "Improving regression testing transparency and efficiency with history-based prioritization–an industrial case study," in *ICST*, 2011, pp. 367–376.

[45] E. D. Ekelund and E. Engstrom, "Efficient regression testing based on test history: An industrial evaluation," in *ICSME*, 2015, pp. 449–457.

[46] T. Noguchi, H. Washizaki, Y. Fukazawa, A. Sato, and K. Ota, "History-based test case prioritization for black box testing using ant colony optimization," in *ICST*, 2015, pp. 1–2.

[47] Y. Lou, J. Chen, L. Zhang, D. Hao, and L. Zhang, "History-driven build failure fixing: how far are we?" in *ISSTA*, 2019, pp. 43–54.

[48] J. Chen, Y. Lou, L. Zhang, J. Zhou, X. Wang, D. Hao, and L. Zhang, "Optimizing test prioritization via test distribution analysis," in *FSE*, 2018, pp. 656–667.

[49] J. Chen, X. He, Q. Lin, H. Zhang, D. Hao, F. Gao, Z. Xu, Y. Dang, and D. Zhang, "Continuous incident triage for large-scale online service systems," in *ASE*, 2019, to appear.

[50] P. McMinn, "Search-based software testing: Past, present and future," in *ICSTW*, 2011, pp. 153–163.

[51] ——, "Search-based software test data generation: a survey," *STVR*, vol. 14, no. 2, pp. 105–156, 2004.

[52] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *CCS*, 2016, pp. 1032–1043.

[53] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *CCS*, 2017, pp. 2329–2344.

[54] J. M. Rojas, G. Fraser, and A. Arcuri, "Seeding strategies in search-based unit test generation," *STVR*, vol. 26, no. 5, pp. 366–401, 2016.

[55] F. Gross, G. Fraser, and A. Zeller, "Search-based system testing: high coverage, no false alarms," in *ISSTA*, 2012, pp. 67–77.

[56] A. Alourani, M. A. N. Bikas, and M. Grechanik, "Search-based stress testing the elastic resource provisioning for cloud-based applications," in *SSBSE*, 2018, pp. 149–165.

[57] D. Romano, M. D. Penta, and G. Antoniol, "An approach for search based testing of null pointer exceptions," in *ICST*, 2011, pp. 160–169.

[58] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *FSE*, 2011, pp. 416–419.