Scope-based Compiler Differential Testing

Rong Qu[1,3], Jiangang Huang[1,2], Long Zhang[1,*], Tianlu Qiao[4], Jian Zhang[1,3,*]
[1]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
[2]Hangzhou Institute for Advanced Study, University of Chinese Academy of Sciences, Hangzhou, China
[3]University of Chinese Academy of Sciences, Beijing, China
[4]School of Computer Science, Beijing University of Posts and Telecommunications, Beijing, China
qurong@ios.ac.cn, huangjiangang22@mails.ucas.ac.cn, zlong@ios.ac.cn, qiaotianlu@bupt.edu.cn, zj@ios.ac.cn
*corresponding authors

Abstract—Compilers are among the most critical components in the software development. Obviously, their correctness is very important, yet they are among the most complex software systems. The traditional grammar-based compiler random testing measures have two shortcomings. Firstly, the technique generating test programs for one programming language is difficult to migrate to another. The second one is that traditional grammar-based technique haven't optimized the relation of identifier definition and use yet, causing the undefined identifiers problems or the low quality of the generated test programs. To address these problems, we propose a scope-based compiler testing method ScopeGen in this paper. To generate runnable and diverse test programs, ScopeGen supports two types of identifier strategies based on the scope information, one is scope distance based and the other is global optimization based. These identifier strategies guide the definition and use of identifiers and balance the distribution of identifiers in different scopes. Benefiting from the public grammar dataset Grammar-v4, ScopeGen can be easily migrated to various programming languages. We implement a program generator and generate grammatically correct and runnable test programs for C, Java and Python. Next, we conduct differential testing to identify various bugs in compilers by comparing the output of different compilers. The experimental evaluation of 9 compilers (gcc, clang, icc, icx, Ark, Javac, CPython, Pypy and Codon) shows that ScopeGen outperforms the two state-of-the-art methods (i.e., Csmith and YARPGen) improving more than 69% in inconsistency finding ability. By running ScopeGen we have reported 114 bugs for 4 compilers, 84 of which were confirmed.

Keywords–compiler testing, differential testing, random testing, code generation, program generation, grammar-based.

1. Introduction

Compilers are used by every programmer almost every day because they are the core components in the software development tool chain. Compiler random testing is one of the commonly used techniques to test the functionality and safety of compilers. The random generator produces grammatically correct test cases to test the deep parts of the compiler. Random testing technology detects software defects by generating a large number of test inputs and monitoring the execution status of the target program. One challenge faced by compiler random testing is to generate grammatically correct test cases. The researches [1][2][3][4] showed that software defects in compilers mainly exist in the middle-end and back-end. To carry out back-end testing (such as code optimization and code generation components), early research work generated test cases based on the grammar specification of the target language. For example, the test program randomly generated by Csmith [5] based on a given C grammar can easily pass the compiler's parsing stage. Livinski [6] developed another random C/C++ program generator YARPGen and it is specifically designed to trigger compiler optimization bugs and is intended for compiler testing. However, these two works pointed out that providing the grammar specification of the target language is a time-consuming and laborious work. In addition, it is difficult to migrate to new target languages and compilers. In recent years, some techniques and methods generate test programs from context-free grammars [7][8][9]. They are easier to migrate to new target languages and compilers. However, the methods of this type always have trouble in handling the undefined identifiers problem.

In this paper, we propose a method ScopeGen to generate grammatically correct and runnable programs based on context-free grammars. In order to solve the problem of undefined identifiers, we propose several identifier selection strategies based on the scope information of the generated program. These strategies can also balance the use of identifiers and improve the diversity of the relations among identifiers and scopes in the generated programs. In program generation, we regard the scope information as tree structure. The following two types of strategies can be used to select available identifiers. The first type is based on the distance between the scope where the identifier is defined and used. There are four strategies in this type: (1) new identifier (NIS); (2) randomly identifier (RIS); (3) closest identifier (CIS); (4) farthest identifier (FIS). The second type is based on the global optimization about the distribution of identifier usage on all the scopes. There are two strategies in this type: (1) scope uniform identifier (SUIS); (2) strength decrease identifier (SDIS).

We use the above strategies to solve the problem of undefined identifiers in generating test programs, and optimize the use of identifiers based on scope information. In order to compare the effects of test programs under different identifier strategies in compiler testing, we conduct differential test experiments on five C compilers. Experiments show that

2693-9177/23/$31.00 ©2023 IEEE
DOI 10.1109/QRS60937.2023.00043

under the same number of test programs, the SDIS and CIS outperform other strategies in finding more inconsistencies of compilers. We develop a program generator for generating test programs based on context-free grammars, and implement the above methods in it. Contributions of this article are as follows:

1) We propose ScopeGen, a testing framework aiming to detect bugs in compilers.
2) We design several identifier selection strategies based on scope tree to solve the undefined identifier problem.
3) We implement a general program generator for ScopeGen and generate runnable programs for C, Java and Python.

The rest of this article is organized as follows. Section 2 presents the background of grammar based program generation and gives the motivation of our study. Section 3 introduces ScopeGen in detail. Experimental evaluation are presented in Section 4. The threats, and related work are described in Sections 5 and 6. Finally, Section 7 concludes this article.

2. BACKGROUND

In this section, we will introduce the grammar format in Grammar-v4 [10] firstly. Secondly, we will illustrate how to solve the deep recursion problem in the grammar.

2.1 Grammar

Grammar is the description of a language. Grammar-v4 [10] contains hundreds of grammars of different programming languages. The definition of a rule(non-terminal) is similar to the EBNF (Extended Backus-Naur Form) in compilation theory, including its name, a separator colon, a rule body and an ending semicolon. For example, the following rules describe the syntax of assignment statements in C language. It means that an assignment statement is composed of an id, an equal sign, an expression and a semicolon.

Example 1: $Assignment\_start$: id '= ' $expr$ ';' ;

The definition of a token(terminal) is similar to that of a rule. It is comprised of a token name, a separator colon, a token body and an ending semicolon. For instance, the following token definition represents a decimal integer. It means that the first digit of a number is between 1 and 9, followed by 0 or more digits between 0 and 9.

Example 2: NUM : ('1'...'9')('0'...'9')* ;

Example 3 : Part of Grammar File JavaParser.g4

```
typeType :
    classOrInterfaceType | primitiveType;
classOrInterfaceType :
    identifier typeArguments?;
typeArguments :
    '<' typeArgument (',' typeArgument)* '>';
typeArgument :
    typeType* '?' ((extends | super) typeType)?;
primitiveType :BOOLEAN | CHAR | BYTE | SHORT | INT | LONG | FLOAT | DOUBLE ;
```

2.2 Deep Recursion Exploration

Some production rules in the grammar may form infinite loops, which causes deeper recursion problem in program generation. For example, the grammar of `type` in Java language is described by the rules in Example 3. We can find that the first four rules form a recursion in generating Java programs from top to bottom. When we try to generate the rule *typeType* randomly, this recursion may cause endless generation problem. In order to avoid this problem, we explore all the nodes(rules) in a recursion and try to find the paths to break out of the recursion. As described in Example 3, the rule *typeType* has two replacements *classOrInterfaceType* and *primitiveType*. If we choose *classOrInterfaceType* to replace *typeType*, the generation will fall into the recursion. If we choose *primitiveType*, it will get out of the recursion. Note that there should be at least one path to break out the recursion of each rule, or the grammar is incorrect or invalid.

2.3 Motivation

In this section, we first present an example to illustrate potential problems in C compiler according to different identifier selection strategies. Then, we conduct a quantitative study of historical compiler bugs to show the prevalence and importance of those problems.

Figure 1 shows two C programs generated using NIS and FIS identifier strategies of our method. These two programs are structurally identical from line 3 to 5. The output of the program in Figure 1(a) is 0 after being compiled and executed on different C compilers. But the C program in Figure 1(b) outputs 1 under gcc and 0 under clang. The two programs are same in grammatical structure but differ in the use of identifiers. NIS always creates a new identifier and uses it for each position that needs to use an identifier, while FIS usually selects the identifier from the farther scopes. We find that the programs with the same grammatical structures may also perform differently in compiler testing. It may be necessary to design strategies to optimize the use of identifiers.

```
//p1.c                                  //p2.c
1:  int main() {                        1:  int main() {
2:      int a=0,b=0,c=0;                2:      int a=0;
3:      if (a < b--) {                  3:      if (a < a--) {
4:          c=1;                        4:          a=1;
5:      }                               5:      }
6:      printf ("%d\n", c ) ;           6:      printf ("%d\n", a ) ;
7:      return 0;                       7:      return 0;
8:  }                                   8:  }
```

(a) NIS (b) FIS

Figure 1. Output differently under gcc and clang.

To further understand the importance of identifier usage and motivate our research, we conduct a quantitative study of compiler history bugs in this section. Specifically, we collect bug reports from the gcc bug repository. Here, we only collect gcc bug reports because gcc has a long history of development and has a clear keyword mechanism to show bug types. Over the 36560 bug reports collected, 2106 (5.8%) of them belong to C-related components. Next, we categorize bugs based on keywords. The top 4 categories of all bugs of C-related components in gcc are ice-on-valid-code, missed-optimization,

diagnostic, and wrong-code. In order to investigate how many of the top 4 categories of bugs are indeed related to the identifier usage, rather than other bugs of the compiler, we perform a small-scale analysis of bugs in these 4 categories. Specifically, we randomly select 100 fixed bugs of each type. Then, we manually check whether the program fragments corresponding to each of the selected 400 bugs contain the definition and use of identifiers. Due to our limited knowledge, we count the bugs that the program segment consisting of definition and use of identifiers as positive samples. Even so, the results in Figure 2 show that 53%, 62%, 43%, and 46% of 100 involved bugs are related to the definition and use of identifiers respectively. Therefore, more advanced identifier selection strategies are needed to help compiler testing.

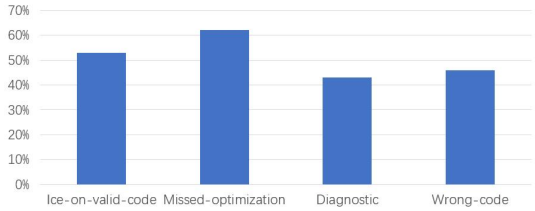

Figure 2. The percentage of bugs related to identifiers in GCC.

Summary: Due to the importance of ensuring compiler reliability and the prevalence of bugs related to the definition and use of identifiers in compilers, we design several scope-based identifier selection strategies to generate test programs for compiler testing. Specifically, we employ a grammar-based random program generation method to generate diverse test programs. Then, we preserve the affiliation between the scope of the program and the identifier, and use different selection strategies of identifiers to control the use of identifiers in the generated programs.

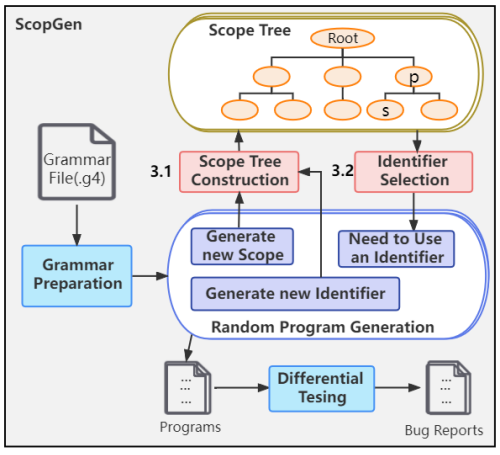

Figure 3. ScopeGen Framework.

3. ScopeGen

Figure 3 shows the framework of ScopeGen. It can be seen that the program generation part takes the grammar file as input and outputs programs that satisfy the grammar. After using these programs to conduct differential testing on compilers, a series of bug reports can be produced. Specifically, we process the grammar file before generating programs automatically. The main task of pre-processing is marking the production rules related to scopes and identifiers.

In the process of randomly generating programs based on grammar, if a new scope needs to be created, we will update the scope tree by adding a scope node to it. If we need to create a new identifier, we will also update the scope tree by associating this identifier with its scope. If we need to use an existing identifier, we will use scope-based or global optimization based identifier strategies to increase the diversity and effectiveness of test programs. We recognize the bugs from the output inconsistencies.

3.1 Scope Tree Construction

The main task of scope tree(ST) construction is to maintain and update the information in the ST. It mainly includes two parts, as shown in Figure 3, one is updating of the ST structure in generating new scopes, and the other is updating the identifier information on each scope node of ST in generating new identifiers. Before introducing the construction of scope trees, we will introduce the scope in programming at first.

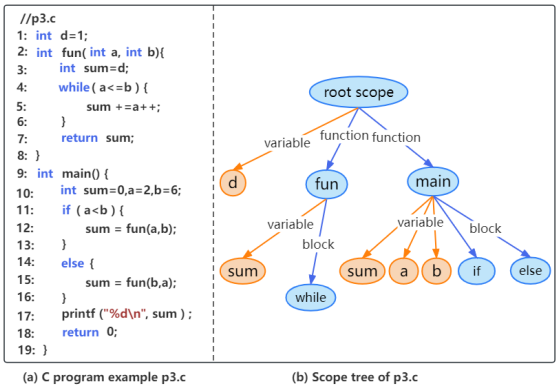

Figure 4. Example of scope tree.

Scope is a concept in programming. Generally speaking, the identifiers used in a program code are not always valid(usable). The code scope that limits the availability of an identifier is the scope of it. The use of scope improves the locality of program logic, enhances program reliability, and reduces identifier name conflicts.

Figure 4 shows an example of a C program p3.c and its scope structure. There are totally 6 scopes in p3.c, root, function 'fun', function 'main', 'while' block, 'if' block and 'else' block. In the scope tree of Figure 4, we also mark the identifiers defined within each scope. For example, the variable `d` is defined in the first line of the program, so we add an identifier record `d` on the root scope node. The variables `sum`, `a`, and `b` are defined in line 10 of the program, so we add three identifier records `sum`, `a`, and `b` on the main function.

In order to solve the problem of undefined identifiers in randomly generating programs, the scope information in the program must be taken into account. The existing grammar-based program generation methods haven't supported a general framework for identifier strategies based on scope yet, which causes it difficult to migrate the method of generating test programs based on one language to another. Therefore, we design a general framework to select identifier based on scope tree to solve the undefined identifier problem. The main content of constructing and updating scope trees will be introduced below.

Each statement belongs to one scope in programs. The correctness of the definition and use in statement can be guaranteed by the scope information. The entire program can be seen as the root scope of ST. The scopes arranged sequentially in the program are sibling nodes in ST, and the nested scopes are parent-child nodes in ST. Assume that the scope of the current production rule R we want to generate is S, which can be represented as $Scope(R) = S$. There are n children scopes of S, $Children(S) = \{S_1, S_2, \ldots S_i, \ldots S_n\}, 1 \leq i \leq n$. Assume that R will create a new scope S' (such as function, block, condition or loop statement, etc.), then establish a new connection of parent-child between S and S'. Update $Children(S) = \{S_1, S_2, ...S_i, ...S_n, S'\}, 1 \leq i \leq n$. Then switch the current scope position from S to S'. Update the current scope position from S' to S when the generation of R is finished.

Assume the current production rule R will create a new identifier, and the content corresponding to the identifier may be the type (int, float, char, bool, array, pointer, etc.), the category (variable, function, etc.) and so on. In order to perform type checking to avoid generating invalid test programs, associate the information of the new identifier with current scope node S. That is, each scope node S contains parent-child relation information and the related information of all identifiers defined in S. We give three examples of associating the identifiers with the scope they belong to in creating new identifiers in Examples 4-6. We first define a variable `a` of int type in scope 'S'. So we add an identifier record {name: "a", type:int, category:var, scope: S} to the identifier table of the generated program. For the identifier of array, the length information is also saved in the identifier record.

Example 4: Define a variable `a' of type int in the scope S
```
int a=0;
```
{name: ``a" , type: int, category: var, **scope: S**}

Example 5: Define an array `arr' of type int in the scope P
```
int arr[3]={0,1,2};
```
{name: ``arr" , type: int[], category: array, **scope: P**, length: 3}

Example 6: Define a pointer `p' of type int in the scope root
```
int *p=&a;
```
{name: ``p" , type: int*, category: pointer, **scope: root** }

Figure 5. Example of new identifier definition.

3.2 Identifier Selection

Assume that the current rule R needs to use an identifier ID_{need} of type $Type_{need}$. To increase the diversity of identifier usage in generated programs, and solve the undefined identifier problems in grammar based measures, we propose two types of identifier selection strategies. One is based on scope distance and the other is based on global optimization. We will introduce them in the next two sections respectively.

3.2.1 Strategies Based on Scope Distance

In most programming languages, an identifier defined in one scope can be used in all the children scopes. To select an identifier defined in ancestor scopes as ID_{need}, and explore the promotion effect of using identifiers in different distance scopes on the diversity of generated programs, we propose four strategies as shown in Figure 6. They are the new identifier strategy (NIS), the closest identifier strategy (CIS), the farthest identifier strategy (FIS) and the random identifier strategy (RIS). To illustrate these strategies, we need to define the following concepts:

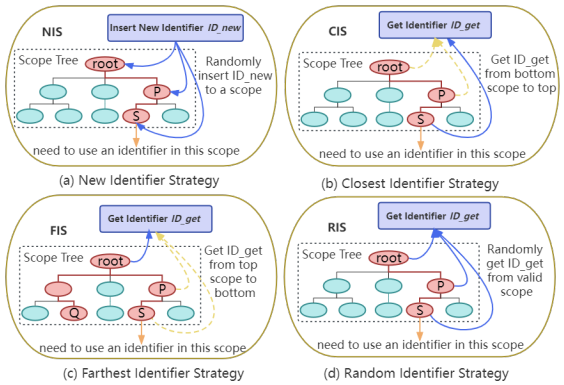

Figure 6. Identifier strategies based on scope distance.

parent. For scopes S_a and S_b, if S_b is directly defined in S_a, we call S_a the parent of S_b, that is $parent(S_b) = S_a$, write as $S_a > S_b$.

path. For scopes S_a and S_b, the path from S_a to S_b is defined as $path(S_a, S_b) = \{S_a, S_1, S_2, \ldots S_j, \ldots S_m, S_b\}$, where $S_a > S_1 > S_2 > \ldots S_j > \ldots S_m > S_b$, $0 \leq j \leq m$.

ancestor. For scopes S_a and S_b, if S_b is indirectly or directly defined in S_a, we call S_a the ancestor of S_b, there is a path $path(S_a, S_b)$ from scope S_a to S_b, the relation of S_a and S_b can be expressed as $S_a >> S_b$.

ancestors. For scope S_a, the set of all its ancestors can be expressed as ancestors(S_a).

distance. The distance between scopes S_a and S_b is defined as $distance(S_a, S_b) = |path(S_a, S_b)|$. The distance between S_a and S_b is the path length from S_a to S_b. It is noted that the distance between a scope and itself is the smallest, i.e. 0. The distance from the parent to the child is 1, that is, if $S_a > S_b$, $distance(S_a, S_b) = 1$. The distance from the child to the parent is infinite, if $S_a > S_b$, $distance(S_b, S_a) = +\infty$.

(1) NIS The relation among identifiers is the weakest by using the new identifier strategy (NIS) among all four scope distance based strategies. It means that when we need to use an identifier of type $Type_{need}$ in scope S, we create a new identifier according to the grammar rules and insert the definition statement of this variable into S or the ancestors of S. Specifically, as shown in Figure 6(a), for the current scope S, firstly construct a scope set $ValidScopes(S) = ancestors(S) \cup S$ in which the new identifier ID_{new} can be defined in. From Figure 6(a), we can see that $ValidScopes(S) = \{root, P, S\}$. After that, we randomly select a scope from $ValidScopes(S)$ and associate ID_{new} with this selected scope.

(2) CIS The closest identifier strategy (CIS) will make the relation among the identifiers that are close to each other stronger than NIS. CIS may improve the internal correlation of each scope in the program relatively. In this case, the definition and use of identifiers are more cohesive. When an identifier of a certain type $Type_{need}$ needs to be used, an identifier defined in $Type_{need}$ type is always randomly taken from the scope closest to the current scope S. That is, the scope with the minimum distance to S. In fact, the identifiers defined in S will be given higher priority in CIS. As shown in Figure 6(b), the closest identifier is selected according to the following steps. Firstly, get the identifier set of type $Type_{need}$ as $Identifiers(S, Type_{need})$ from the current scope node S. If the identifier set is not empty, then an identifier is randomly selected from this set as the result of ID_{need}. If the set of identifiers is empty, we take out the parent in turn, and then take out the set of identifiers available in the parent until we find out a set of identifiers that is not empty. In Figure 6(b), the nodes of scope are traversed sequentially from bottom S to top $root$. If there are still no available identifiers, we will create a new identifier and use it which is described in NIS.

(3) FIS If we use the farthest identifier strategy (FIS), the relation among identifiers will be stronger than CIS, and the scopes in the program may be strongly correlated with each other. In this case, the definition and use of identifiers in different scopes are more coupled. When an identifier of type $Type_{need}$ needs to be used, always get the identifiers of type $Type_{need}$ randomly from the scope farthest from the current scope. At this time, the variables in each unrelated scope can often be more related through the variables defined in ancestor scopes. Two scopes are unrelated to each other means that there is no parent-child or ancestor-descendant relationship between these two scopes, such as the relationship between sibling nodes in the scope tree. In Figure 6(c), the scopes P and Q are unrelated, but the statements in P and Q may use the same variables defined in scope $root$. In this way FIS enhances the relation of variables defined in unrelated scopes. As shown in Figure 6(c), FIS first takes out all the identifiers of type $Type_{need}$ from the $root$ node as a set $Identifiers(root, Type_{need})$. If the set is not empty, it will randomly select an identifier from this set as the identifier selection result. If the set of identifiers is empty, we start at the $root$ node and follow the $path(root, S)$ from top to bottom to take out the identifiers of type $Type_{need}$ in each scope nodes in turn until the set of identifiers $Identifiers$ is not empty, and then randomly select an identifier from this set as the result. In Figure 6(c), if we try to use an identifier defined before in scope S, the nodes are traversed sequentially from top to bottom: $root$, P and S. If there is still no available identifiers, a new identifier will be created by NIS.

(4) RIS Random identifier strategy (RIS) may enrich the diversity of the relation between identifiers in different scopes. When an identifier of type $Type_{need}$ needs to be used, an identifier of $Type_{need}$ is always randomly taken from current scope or the ancestors of current scope. It is possible to get the identifier from both the closer scopes, and the farther scopes. As shown in Figure 6(d), RIS selects the identifier according to the following steps. Firstly, traverse from the current scope S to the $root$ on the scope tree ST to get all the identifiers of type $Type_{need}$ as a set $Identifiers$, In Figure 6(d), the set $Identifiers = Identifiers(S, Type_{need}) \cup Identifiers(P, Type_{need}) \cup Identifiers(root, Type_{need})$. And then RIS randomly takes an identifier from $Identifiers$ as the result of the identifier selection. However, grammar-based program generation is incremental. So that the earliest defined identifiers have more chances to be selected, which causes the imbalance of the use of identifiers. Therefore, the diversity of identifier usage in RIS can be further improved.

3.2.2 Strategies Based on Global Optimization

The strategies based on scope distance such as CIS and FIS can control the diversity of generated test programs than random strategy RIS, but they don't take the usage of various identifiers within different scopes into account. To illustrate this situation, take the C program in Figure 7 as an example. In line 4 of these 2 programs, we need to generate an addition statement that uses three identifiers. If we use an identifier selection strategy based on scope distance, we may generate the program as $p_4.c$ in Figure 7(a), where the addition statement is `a=a+a`. This is because the identifiers `a`, `b` and `c` defined in line 2 are within the same scope, i.e. the 'main function' scope. The distance between the 'main function' scope and the 'while' scope is 1. Then for the strategies based on scope distance, the identifiers `a`, `b` and `c` will be weighted equally. Therefore, it is possible for the three identifiers in the addition operation statement to be selected as the same under CIS, FIS and RIS, that is, in line 4 of Figure 7(a), `a=a+a`.

```
//p4.c                                  //p5.c
1:  int main() {                        1:  int main() {
2:      int a=1,b=2,c=3,i=0;            2:      int a=1,b=2,c=3,i=0;
3:      while(i++< 10) {                3:      while(i++< 10) {
4:          a = a + a;                  4:          a = b + c;
5:      printf ("%d\n", a ) ;           5:      printf ("%d\n", a ) ;
6:      }                               6:      }
7:      return 0;                       7:      return 0;
8:  }                                   8:  }
```

(a) Based on scope distance (b)Based on gloabl optimization

Figure 7. Motivation of global optimization based strategies.

For the program $p_5.c$ in Figure 7(b), we use identifiers `a`, `b` and `c` in line 4, so that each identifier is used in the while scope. We believe that this testing program is more diverse on the aspect of identifier use than the program in Figure 7 (a). In order to maximize the relation strength between identifiers and scopes in a program, we propose two global optimization strategies to select available identifiers. The first is a strategy of uniform scope, and the second is of decreasing strength.

(1) SUIS The scope uniform identifier strategy (SUIS) expects that in the generated program, the use of each identifier can be evenly distributed on different scopes. In the test program generation, we maintain an identifier scope uniformity coefficient (ISUC) for each program defined as follows:

$$ISUC = \sum_{i=1}^{n}(\frac{varUsed_i}{scopeNum_i} - varMinTime_i) \quad (1)$$

where n represents the number of identifiers in the current program, $varUsed_i$ represents the number of times the i-th identifier var_i is used in the generated program, and its definition is as follows:

$$varUsed_i = \sum_{j=1}^{scopeNum_i} varUsedScope_{ij} \quad (2)$$

where $scopeNum_i$ indicates the number of scopes that can use the identifier var_i, that is, the number of scopes that define the identifier var_i and all its descendant scopes. $varUsedScope_{ij}$ indicates the number of times that var_i is used in the j-th scope. $varMinTime_i$ represents the minimum use times of var_i in all $scopeNum_i$ scopes, and its definition is as follows:

$$varMinTime_i = \min_j(varUsedScope_{ij}) \quad (3)$$

In order to make the identifiers in the generated program evenly distributed in the scopes where they can be used, when an identifier needs to be used, we select the identifier by minimizing the value of the uniformity coefficient ISUC of the program, that is, the optimization goal is $\min(ISUC)$.

(2) SDIS The strength decrease identifier strategy (SDIS) expects that we can always select the identifier maximizing the strength of the identifier and scope. We define the identifier scope strength coefficient as ISSC, and the definition of this coefficient is as follows:

$$ISSC = \sum_{i=1}^{n}\sum_{j=1}^{scopeNum_i}(1 + \frac{1}{2} + \frac{1}{4} + ... + \frac{1}{2^{t_j-1}}) \quad (4)$$

where n, i, $scopeNum_i$ and j share the same meanings as described before, t_j represents the number of times the i-th identifier is used in its j-th scope. When an identifier needs to be used, we select the optimal identifier by maximizing the value of the strength coefficient ISSC of the program. If an identifier is used too many times in the same scope, the increment it can bring to the optimization function will decrease exponentially. That is, the optimization goal is $\max(ISSC)$.

4. EVALUATION

In this section, we evaluate the proposed method and we mainly focus on the following three research questions:

RQ1: Can our method find more compiler inconsistencies than state-of-the-art methods?

RQ2: Which of the identifier selection strategies we proposed can find more compiler inconsistencies?

RQ3: How well does our proposed program generation method ScopeGen perform in actual compiler testing?

RQ1 evaluates the bug finding ability of ScopeGen compared to state-of-the-art methods (i.e., Csmith and YARPGen). In particular, we tested compilers using ScopeGen, Csmith and YARPGen to generate the same number of test cases, comparing both the quality of the generated programs and the number of inconsistencies in compiler behavior detected. RQ2 investigates the impact of the six proposed identifier selection strategies on the bug-finding ability of ScopeGen. We used these strategies to generate the same number of C programs for differential testing of the compiler, and analyzed the contribution of different strategies to ScopeGen by comparing the number of compiler inconsistencies detected. RQ3 evaluates the ability of ScopeGen to detect compiler bugs in practice. Specifically, we implemented our method ScopeGen for three languages C, Java, and Python, and used the test programs generated by ScopeGen to conduct differential testing on the compilers. And we evaluated the bug finding ability of ScopeGen from the number of bugs submitted and confirmed.

4.1 Experimental Setup

Our evaluation is performed on the linux operating system Ubuntu 20.04.5 LTS equipped with an AMD Ryzen 9 5950X processor (16 cores and 32 threads). We followed the compiler testing research [5][6], testing 5 C compilers (gcc, clang, icc, icx, Ark C [11]), 2 Java compilers (javac and Ark Java [11]), and 3 Python compilers (CPython, Pypy, Codon [12]).

For the implementation of the program generator ScopeGen, we take C, Java, Python grammar files in Grammar-v4 [10] as the input, which is a collection of various ANTLR [13] grammars. Grammar-v4 is publicly available, contributed by developers around the world. To generate runnable programs, we manually deleted the complex features, and marked the special rules in the grammar, such as function rules, scope-related rules, identifier rules, etc. In order to prevent program generation from falling into the deeper recursion of grammar rules, we pre-processed all the deeper recursions in the grammar automatically by the strategies mentioned in Section 2.2. After that, we generated test programs by replacing the rules with their replacements randomly. The length of programs generated by ScopeGen is configurable. In order to ensure the dynamic correctness of the program, we solved some problems such as division by zero, array index out of bounds, null pointer, infinite loop, and expression type checking.

To illustrate the bug-finding capability of ScopeGen, we compare it with two state-of-the-art methods, namely Csmith [5] and YARPGen [6]. Csmith is a widely used compiler testing tool that randomly generates valid C programs to help

TABLE I
COMPARISON OF GENERATING 1000 C PROGRAMS

	Total Time	Average lines	Invalid Programs
Csmith	6m42s	1363 lines	12%
YARPGen	3m8s	1357 lines	0%
ScopeGen	11s	38 lines	0%

TABLE II
COMPARISON OF COMPILING 1000 C PROGRAMS

	Executable File Size(MB)			**Compilation Time(s)**		
	ScopeGen	Csmith	YARPGen	ScopeGen	Csmith	YARPGen
clang14O2	15.4	29.4	46.3	17.6	35.3	169.3
clang14	15.5	71.3	56.6	18.4	33.1	47.7
clang15O2	**15.3**	**29.2**	45.7	**34.8**	**184.0**	**1545.0**
clang15	15.4	71.2	56.5	27.8	94.7	62.8
gcc9O2	15.4	36.0	36.8	15.4	38.2	110.4
gcc9	15.4	119.0	51.8	14.8	33.5	27.0
gcc11O2	15.4	36.5	**35.9**	16.0	40.1	114.7
gcc11	15.4	**119.7**	51.7	15.0	33.0	33.4
iccO2	**28.6**	83.9	38.4	6.4	18.8	72.7
icc	28.5	72.9	58.5	**6.2**	**8.1**	**18.6**
icxO2	15.4	44.2	**187.1**	22.8	47.2	363.8
icx	15.4	61.1	54.8	18.6	30.6	46.9
Average	**17.6**	**69.6**	**64.6**	**19.4**	**52.3**	**217.7**

find bugs in the compiler. The programs generated by Csmith won't include 52 unspecified behaviors and 191 undefined behaviors in C99. The idea of using YARPGen to verify the compiler is the same as that of Csmith, which is carried out by differential testing. But YARPGen can support C++ language.

We tested compilers by using the optimization option -O2 or not on two newer versions of gcc, gcc 11.3.0 and gcc 9.5.0. For clang, we tested clang 15.0.7 and clang 14.0.0. In addition to these two commonly-used compilers, we also tested the latest version of icc (Intel C/C++ Compiler Classic) and the latest version of icx (Intel oneAPI DPC++/C++ Compiler 2023.0.0). In addition, we also tested javac-17.0.6, Ark compiler 1.0.0 (C and Java), CPython-3.8.10, CPython-3.9.16, Pypy-3.9 and Codon-0.15.5 [12].

4.2 Answer to RQ1

Motivation: This RQ aims to investigate the inconsistency-finding capability of ScopeGen compared with two state-of-the-art approaches, i.e., Csmith and YARPGen.

Approach: To evaluate RQ1, we ran ScopeGen, Csmith and YARPGen to generate the same number of C programs, using the RIS strategy in ScopeGen. Table I is the basic information of generating the same number of 1000 C programs. Among them, the invalid Programs refers to the proportion of programs that will cause timeout problem on all the compilers. Table II lists the program compilation time generated by different methods and the size comparison data of executable files.

Results: From Table I we find that ScopeGen generated the same number of test programs in much less time, 11 seconds, while Csmith needs 6 minutes and 42 seconds, and YARPGen needs 3 minutes and 8 seconds. Among 1000 programs, 120 of Csmith are invalid and all of the programs of ScopeGen and YARPGen are valid. It indicates that ScopeGen can generate

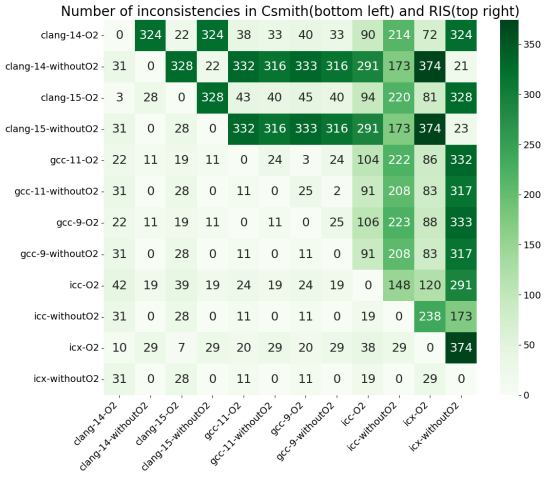

Figure 8. Csmith and ScopeGen (RIS).

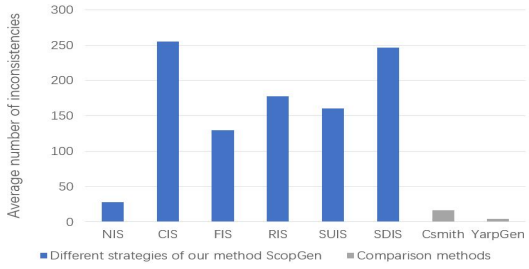

Figure 9. Average inconsistency.

smaller and useful test programs in a shorter time.

In Table II we find that all the "Executable File Size" of ScopeGen under different compilers are smaller than Csmith and YARPGen. Meanwhile, ScopeGen needs shorter time in compilation than Csmith and YARPGen. The first reason may be that the length of test programs of Csmith and YARPGen are much longer (1363 and 1357 lines) than ScopeGen (38 lines) according to Table I. The second reason may be that the programs generated by Csmith must be compiled with other 17 files including header files and C files provided by Csmith. In contrast, the programs generated by ScopeGen can be compiled and executed independently. All the "Compilation Time" of ScopeGen are less than those of Csmith and YARPGen. The experimental results mentioned above illustrate that ScopeGen have more advantages in generating test programs.

In order to compare the performance of different methods in inconsistency-finding, we generate 10,000 C programs by ScopeGen (RIS), Csmith and YARPGen respectively. And then we conduct a differential testing on 12 compilers and the results of each two compilers are compared. We list the differential testing results of ScopeGen (RIS) in the top right corner of Figure 8. The results of Csmith are in the bottom left corner. On 10,000 programs, there are only 1 compilation error and 9 run-time errors of YARPGen under icc compiler with O2 option. So we don't list YARPGen in Figure 8.

By comparing the experimental results in Figure 8, we find

that the number of inconsistencies detected by ScopeGen (RIS) is much more than that of Csmith. Specifically, Figure 9 shows the average number of inconsistencies between pairwise compilers over 10,000 programs is 177.8 under ScopeGen (RIS), and 22.7 under Csmith. On the one hand, this indicates that our method is effective in inconsistency-finding. On the other hand, the reason may be that Csmith and YARPGen prevent unspecified behaviors while ScopeGen (RIS) prevents only part of unspecified behaviors.

> Conclusion: The results demonstrate that ScopeGen (RIS) has a better inconsistency-finding capability compared with Csmith and YARPGen, achieving an improvement of over 69% in finding compiler inconsistencies.

4.3 Answer to RQ2

Motivation: This RQ evaluates the impact of the proposed six identifier selection strategies NIS, CIS, FIS, RIS, SUIS and SDIS on the inconsistency-finding capability of ScopeGen.

Approach: We tested the same compiler on the same number of test cases as RQ1. Then we compared the inconsistency-finding capabilities of different identifier selection strategies in terms of the number of inconsistencies detected.

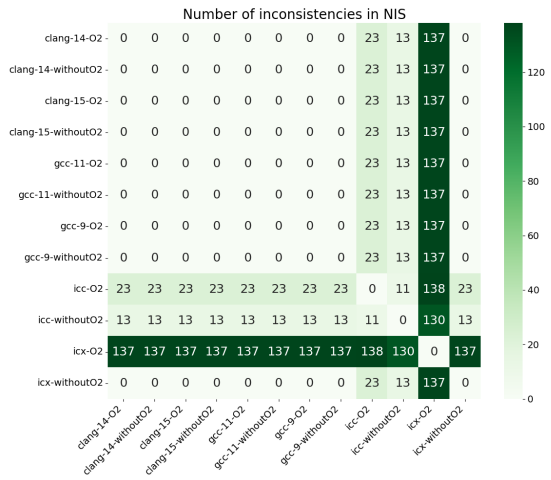

Figure 10. New identifier strategy NIS.

Results: Figures 10-12 show the number of inconsistencies detected by 10,000 C programs using NIS, CIS, FIS, SUIS, and SDIS strategies. The results about RIS have been shown previously. The average number of inconsistencies for each strategy is shown in Figure 9 and we can observe that the results of the NIS strategy is significantly worse than other strategies. This might because in NIS, a new identifier is always generated for each production rule that needs an identifier, the definition and usage relation between identifiers in different scopes are relatively lower. CIS and FIS are based on the scope distance. Figure 11 shows that the number of compiler inconsistencies detected by CIS is relatively more than that of FIS. The average numbers pf inconsistencies in Figure 9 are 255.3 and 129.9 respectively. The reason may be

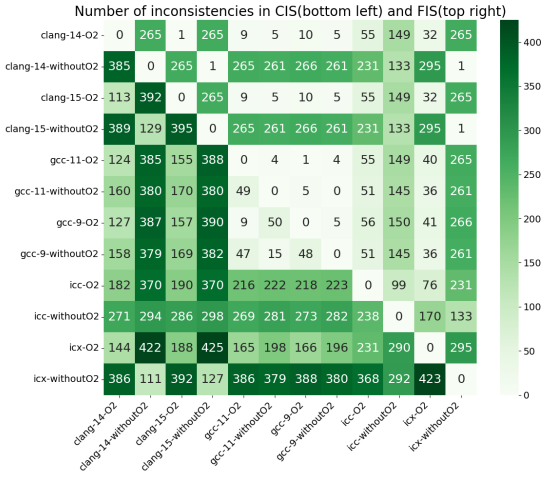

Figure 11. Scope distance based strategies CIS and FIS.

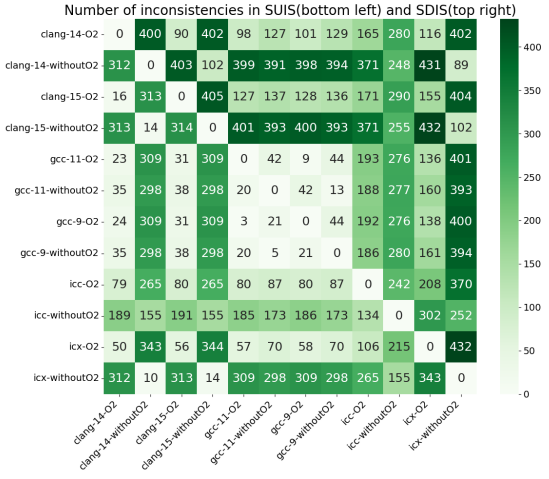

Figure 12. Global optimization strategies SUIS and SDIS.

that CIS prefers to select identifiers in the nearest scope, which means that identifiers defined in each layer of scopes have a chance to be used in the next layer of scopes, while FIS prefers to use the identifiers in the outer layer of scopes, resulting in the identifiers defined in inner scopes have no chances to be used. This will reduce the diversity of the relations of identifier in definition and usage. It is worth noting that the average number of inconsistencies in RIS is 177.8, which is less than CIS but more than FIS. The reason may be that in the process of incremental program generation under RIS, the earliest defined identifiers have more chances to be selected. Therefore, the diversity of RIS in the definition and use of identifiers is between FIS and CIS. Both SUIS and SDIS are based on global optimization. From the results in Figure 12, we find that SDIS is better than SUIS. The average numbers of inconsistencies about SDIS and SUIS in Figure 9 are 246.7 and 160.8 respectively and SDIS is about 54% better than SUIS. This may because that SDIS uses an exponential decline when an identifier is used too many times in the same scope, while

TABLE III
NUMBER OF REPORTED BUGS BY SCOPEGEN

	Compilers	Submitted	Confirmed	Pending	Rejected
C	icx-O2	2	0	2	0
	icx	1	0	1	0
	Ark-C	97	75	22	0
Java	Ark-Java	3	3	0	0
Python	Pypy-3	1	1	0	0
	Codon	10	5	0	5
Total		114	84	25	5

SUIS is a linear decline. SDIS is more effective in balancing the identifiers usage in different scopes than SUIS, resulting in more diverse test programs. From the comparison in Figure 9, it is easy to find that in terms of the inconsistency finding capability, CIS>SDIS>RIS>SUIS>FIS>NIS.

> Conclusion: The identifier selection strategies can effectively help ScopeGen detect more inconsistencies in C compilers. Specifically, the CIS and SDIS strategies in ScopeGen can detect more inconsistencies than the other strategies, especially more than 43% and 38% improvements of the random strategy RIS respectively.

4.4 Answer to RQ3

Motivation: Detecting real bugs in compilers is difficult. This RQ evaluates the actual bug-finding ability of ScopeGen in C, Java and Python compilers.

Approach: We tested 4 types of C compilers (gcc, clang, icc, icx) mentioned above. We also tested the newer version of Javac, the Ark compiler, and 3 Python compilers CPython, Pypy, and Codon.

Results: Specifically, Table III lists the number of bugs we have submitted so far. Among them, 3 bugs were submitted to icx-2023.0.0, which until now have not been addressed by the developers. A total of 97 C bugs and 3 Java bugs were submitted to Ark-1.0.0, of which a total of 75 C bugs were confirmed, 3 Java bugs were all confirmed, and the other 22 C bugs reported have not been processed yet. 1 bug submitted to Pypy-3.9 was confirmed. All 10 bugs submitted to Codon-0.15.5 were addressed, 5 of which were confirmed and the other 5 didn't meet the grammar specification of Codon, which supports a smaller grammar subset of Python and general Python programs cannot be compiled by Codon.

> Conclusion: ScopeGen is effective in detecting compiler bugs in practice. It reported a total of 114 bugs on icx, Ark Compiler, Pypy and Codon. Of these, developers have confirmed 84 bugs.

5. THREATS TO VALIDITY

In this section we discuss the limitations of ScopeGen. Firstly, we find that the test programs containing unspecified and undefined behavior lead to a significantly higher number of inconsistencies under clang-15 with O2 option. Secondly, the length of programs generated by ScopeGen is configurable, but we did not fully test the impact of different lengths of programs on experimental results. Since longer program tends to contain more complex behaviors, thus, having more potential to trigger compiler bugs. Thirdly, ScopeGen randomly generates a test program based on language grammar, and then tests the compiler based on differential testing. We cut down the grammar in order to produce valid and runnable test programs, which prevents us from supporting all grammar features in Grammar-V4 [10].

6. RELATED WORK

Compiler differential testing [14] is currently the main method to ensure compiler quality [15][16]. Existing compiler testing techniques can be classified into three categories, namely Random Difference Testing (RDT), Different Optimization Levels (DOL), and Equivalent Modulo Input (EMI) [16][17]. RDT detects compiler errors by comparing the output of different compilers with the same specification, while DOL compares the results produced by the same compiler with different optimization levels. Most RDT and DOL based techniques [5][18][19][20][21][22][23][24], use randomly generated test programs to test the compiler. Csmith [5] and YARPGen [6] are two widely used C program generators for testing C compilers. However, Csmith and YARPGen only target C and C++.

The idea of using grammars as test generation models has several decades of history. As early as in the 1970s, Purdom [7] experimented with testing parser programs with test cases generated from context-free grammars. Kifetew [25] used a stochastic context-free grammar in BNF format as a test generation model. Unfortunately, their prototyping tools are not available to the public. Dharma [9] allows users to define grammar files and then generate programs according to the given grammar. Grammarinator [8] is a random test program generator that creates test programs based on grammars in Grammar-v4 [10], but it doesn't solve problems such as undefined identifiers and deeper recursion. POLYGLOT [26] is a generic fuzzing framework for exploring processors of different programming languages, but it cannot generate test programs with more diverse identifier relations.

Our research is also based on RDT. However, we focus on generating test programs to test compilers using a scope structure-based identifier selection strategy. Our approach is easy to generate programs for different programming languages based on grammars in the Grammar-v4 [10] ensemble. By marking the grammatical rules related to the scope, symbol table, etc. in the grammars of different languages, it is possible to generate valid and effective programs for different languages for compiler testing.

7. CONCLUSION

In this paper, we propose a framework named ScopeGen to generate test programs for compiler testing. ScopeGen addresses two challenges, generating runnable test programs based on grammars, and obtaining test programs with diverse definitions and usage relations based on scope information. The experimental evaluation shows that ScopeGen detect

over 69% more inconsistencies than the two state-of-the-art methods Csmith and YARPGen on C compilers. We use ScopeGen to generate C, Java, and Python test programs and conduct differential testing. We submitted a total of 114 bugs to icx, Ark, Pypy, and Codon compilers, of which 84 were confirmed. Our method can identify many inconsistencies in compiler behavior, but they may be related to the specification of programming language and require further research.

ACKNOWLEDGMENT

We are grateful to Baoquan Cui, Mengze Hu, Jiwei Yan, Junjie Chen, Jun Yan, Xutong Ma, Yu Zhang, and the anonymous reviewers for their helpful comments and suggestions. This work is supported by the National Natural Science Foundation of China (NSFC) under grant number 62132020.

REFERENCES

[1] C. Cummins, P. Petoumenos, and A. Murray, "Compiler fuzzing through deep learning," in *27th International Symposium on Software Testing and Analysis, ISSTA*. ACM, 2018, pp. 95–105.

[2] M. Sassa and D. Sudosa, "Experience in testing compiler optimizers using comparison checking," in *International Conference on Software Engineering Research and Practice & Conference on Programming Languages and Compilers*, 2006, pp. 837–843.

[3] R. Morisset, P. Pawan, and F. Z. Nardelli, "Compiler testing via a theory of sound optimisations in the C11/C++11 memory model," in *Conference on Programming Language Design and Implementation, PLDI*. ACM, 2013, pp. 187–196.

[4] C. Béra, E. Miranda, and M. Denker, "Practical validation of bytecode to bytecode JIT compiler dynamic deoptimization," *J. Object Technol.*, vol. 15, no. 2, pp. 1:1–26, 2016.

[5] X. Yang, Y. Chen, and E. Eide, "Finding and understanding bugs in C compilers," in *32nd Conference on Programming Language Design and Implementation, PLDI*. ACM, 2011, pp. 283–294.

[6] V. Livinskii, D. Babokin, and J. Regehr, "Random testing for C and C++ compilers with YAPPGen," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 196:1–196:25, 2020.

[7] P. Purdom, "A sentence generator for testing parsers," in *BIT Numerical Mathematics*, 1972, pp. 366–375.

[8] R. Hodován, Á. Kiss, and T. Gyimóthy, "Grammarinator: a grammar-based open source fuzzer," in *9th International Workshop on Automating TEST Case Design, Selection, and Evaluation, FSE*. ACM, 2018, pp. 45–48.

[9] Dharma. (2020). [Online]. Available: https://github.com/MozillaSecurity/dharma.

[10] Grammar-v4. (2014). [Online]. Available: https://github.com/antlr/grammars-v4

[11] OpenArkCompiler. (2020). [Online]. Available: https://gitee.com/openarkcompiler/OpenArkCompiler.

[12] Codon. (2022). [Online]. Available: https://github.com/exaloop/codon.

[13] Antlr. (2020). [Online]. Available: https://www.antlr.org.

[14] W. M. McKeeman, "Differential testing for software," *Digit. Tech. J.*, vol. 10, no. 1, pp. 100–107, 1998.

[15] M. Marcozzi, Q. Tang, and A. F. Donaldson, "Compiler fuzzing: how much does it matter?" *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 155:1–155:29, 2019.

[16] J. Chen, J. Patra, and M. Pradel, "A survey of compiler testing," *ACM Comput. Surv.*, vol. 53, no. 1, pp. 4:1–4:36, 2021.

[17] J. Chen, W. Hu, and D. Hao, "An empirical comparison of compiler testing techniques," in *38th International Conference on Software Engineering,ICSE*. ACM, 2016, pp. 180–190.

[18] C. Lindig, "Random testing of C calling conventions," in *Sixth International Workshop on Automated Debugging, AADEBUG*. ACM, 2005, pp. 3–12.

[19] H. Tu, H. Jiang, and Z. Zhou, "Detecting C++ compiler front-end bugs via grammar mutation and differential testing," *IEEE Trans. Reliab.*, vol. 72, no. 1, pp. 343–357, 2023.

[20] Y. Yang, Y. Zhou, and H. Sun, "Hunting for bugs in code coverage tools via randomized differential testing," in *41st International Conference on Software Engineering, ICSE*. IEEE / ACM, 2019, pp. 488–498.

[21] G. Ofenbeck, T. Rompf, and M. Püschel, "Randir: differential testing for embedded compilers," in *7th Symposium on Scala*. ACM, 2016, pp. 21–30.

[22] A. F. Donaldson, H. Evrard, and A. Lascu, "Automated testing of graphics shader compilers," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, pp. 93:1–93:29, 2017.

[23] V. Le, C. Sun, and Z. Su, "Randomized stress-testing of link-time optimizers," in *International Symposium on Software Testing and Analysis, ISSTA*. ACM, 2015, pp. 327–337.

[24] M. A. Alipour, A. Groce, and R. Gopinath, "Generating focused random tests using directed swarm testing," in *25th International Symposium on Software Testing and Analysis, ISSTA*. ACM, 2016, pp. 70–81.

[25] F. M. Kifetew, R. Tiella, and P. Tonella, "Combining stochastic grammars and genetic programming for coverage testing at the system level," in *Search-Based Software Engineering - 6th International Symposium*, vol. 8636. Springer, 2014, pp. 138–152.

[26] Y. Chen, R. Zhong, and H. Hu, "One engine to fuzz 'em all: Generic language processor testing with semantic validation," in *IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 642–658.