REMGEN: Remanufacturing a Random Program Generator for Compiler Testing

Haoxin Tu[1,2], He Jiang[1], Xiaochen Li[1], Zhilei Ren[1], Zhide Zhou[1], and Lingxiao Jiang[2]
[1]*School of Software, Dalian University of Technology, Dalian, China*
[2]*Singapore Management University, Singapore*
haoxintu@gmail.com, {jianghe, xiaochen.li, zren}@dlut.edu.cn, cszide@gmail.com, lxjiang@smu.edu.sg

***Abstract*—Program generators play a critical role in generating bug-revealing test programs for compiler testing. However, existing program generators have been tamed nowadays (i.e., compilers have been hardened against test programs generated by them), thus calling for new solutions to improve their capability in generating bug-revealing test programs. In this study, we propose a framework named REMGEN, aiming to *Rem*anufacture a random program *Gen*erator for this purpose. REMGEN addresses the challenges of the synthesis of diverse code snippets at a low cost and the selection of the bug-revealing code snippets for constructing new test programs. More specifically, REMGEN first designs a grammar-aided synthesis mechanism to synthesize diverse code snippets. Then, a grammar coverage-guided strategy is used to select the most diverse code snippets that may be bug-revealing. As a case study to demonstrate the effectiveness of the REMGEN framework, we have remanufactured an old C program generator CCG and named it REMCCG. Our evaluation results show that REMCCG can generate significantly more bug-revealing test programs than the original CCG; notably, REMCCG has found 56 new bugs for two mature compilers (i.e., GCC and LLVM), of which 37 have already been fixed by their developers.**

***Index Terms*—Reliability, software testing, compiler testing, automated testing, random program generator, compiler defect**

I. INTRODUCTION

Compiler bugs may have catastrophic consequences on software systems [1]. However, due to the tremendous lines of code and the sophisticated logic in compilers, discovering compiler bugs is non-trivial. To improve the reliability of compilers, a rich collection of studies [2]–[14] is proposed to construct diverse test programs for testing compilers.

As shown in Fig. 1, the prevalent test program construction approaches for compiler testing can be classified into two categories: generation-based approaches (program generators, e.g., CCG [3]) and mutation-based approaches (program mutators, e.g., Hermes [4]). Both the two categories usually start from a program generator. In the former, a generation-based approach aims to design aggressive program generators to directly produce bug-revealing test programs (i.e., route ① in Fig. 1). In the latter, a mutation-based approach regularly takes two steps to obtain test programs. The first step is to collect a seed program, which usually comes from program generators. In the second step, different mutations (i.e., code snippets insertion/deletion) upon the seed program are conducted to generate bug-revealing test programs, which follow the route ② in Fig. 1. Based on the above widespread usage of program generators, it is clear that high-quality generators are the foundation of both generation-based and mutation-based approaches; they can have a significant impact on detecting new bugs in compilers. Typical program generators include CCG [3], Csmith [2], and YARPGen [8]. CCG is designed for generating syntactically valid C programs, while Csmith and YARPGen target generating test programs without undefined and unspecified behaviors. All the above program generators have discovered hundreds of compiler bugs and they are expected to produce bug-revealing test programs to consistently make compilers reliable.

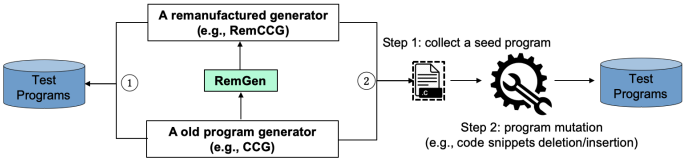

Fig. 1: The general idea of REMGEN and the two prevalent approaches toward constructing test programs for compiler testing

Unfortunately, existing program generators seem to be tamed and can hardly uncover new bugs directly anymore. For example, Csmith [2] is one of the most notable program generators, but prior studies show that current production compilers are already resilient to it [4], [5]. In addition, a plurality of active researchers/developers is profoundly complaining about this problem. For instance, the developer of CCG [3] (an old program generator that has existed for more than 10 years) mentioned that "*Compilers have now caught up with CCG (since it's been pretty hard to spot crashes last time I tried)*"[1]. John Regehr, one of the creators of Csmith, said, "*I hadn't run Csmith for a while and it turns out LLVM is now amazingly resistant to it, ran a million tests overnight without finding a crash or miscompilation*"[2]. Also, Dmitry Babokin, one of the contributors of YARPGen [8], followed the comment with "*Same with YARPGen*"[3]. This is reasonable as maintainers of modern compilers (e.g., GCC and LLVM) are actively and promptly fixing reported bugs.

* He Jiang is the corresponding author. Haoxin Tu's first affiliation is with the School of Software, Dalian University of Technology.

[1]https://github.com/Mrktn/ccg/blob/master/README
[2]https://twitter.com/johnregehr/status/1134866965028196352
[3]https://twitter.com/DmitryBabokin/status/1134907976085516290

A large portion of bugs (directly or indirectly) discovered by program generators have been fixed, making compilers robust to the test program generation mechanism in these program generators [2], [4]–[6], [9]. Under such circumstances, ***is it possible to improve the bug-finding capability of tamed program generators towards compiler testing***?

Challenges. To investigate the above question, two main challenges need to be addressed for making those generators produce bug-revealing test programs again. The first one is **the synthesis of diverse code snippets at a low cost**. Although random program generators can generate numerous code snippets in seconds, it is non-trivial to synthesize diverse code snippets to reveal compiler bugs, as we do not know what are the characteristics of bug-revealing code snippets [15], [16]. Besides, the costs of existing mutation-based approaches (e.g., Hermes [4]) are relatively high as they need to collect the required context information (e.g., names and types of global or local variables) by laboriously profiling the seed program and maintaining the validity of mutated test programs based on the context. The second challenge is **the selection of the bug-revealing (i.e., more likely to trigger bugs) code snippets**. Only a few constructed test programs can trigger bugs [2], [6], [15], [17]. To save computing or human resources, it is necessary to select the code snippets that are more effective for constructing new bug-revealing test programs.

Solutions. In this study, we migrate the idea of *remanufacturing* a product (a process to make an *old* product *new*, see more details in Section II-B) to a program generator and propose a framework named REMGEN for this purpose. In general, two intuitions are behind RemGen: (1) certain capabilities (i.e., the new valuable code snippets generation and the lightweight context reservation) in existing program generators could help construct new code snippets that may exert deeper code in compilers; (2) effectively leveraging those capabilities and introducing a "grammar coverage" metric to guide the construction (rather than a random generation) of test programs, which could detect more bugs in those deeper code regions in compilers. Specifically, in REMGEN, we leverage those capabilities in existing generators and design two new components for effective remanufacturing. The first component is called diverse code snippets synthesis, which uses a grammar-aided code snippets synthesis for producing various kinds of code snippets. In particular, this component first reserves the context (e.g., local and global variables in a function) that is produced during the generation process in a program generator and then leverages the context to synthesize new code snippets by invoking the built-in functions in the generator, thus addressing the first challenge. To address the second challenge, the component of the bug-revealing code snippets selection is proposed. It adopts a grammar coverage metric to record the use frequency of grammar rules of selected code snippets during the synthesis. Such recorded coverage is used to measure the diversity of every synthesized code snippet. Guided by the coverage metric, REMGEN selects the code snippets with the most diverse grammar coverage to construct new bug-revealing test programs.

To assess the effectiveness of REMGEN, we remanufactured a tamed program generator CCG into REMCCG under our proposed framework (see Fig. 1), and extensively evaluated the capability of REMCCG in generating bug-revealing test programs over two mainstream compilers (i.e., GCC and LLVM). First, we evaluate its capability in boosting prevalent program construction approaches over old versions of compilers. Specifically, when compared with the generation-based approach (i.e., CCG [3]), REMCCG can find 16% and 11% more bugs than CCG in GCC and LLVM, respectively. For the comparison with the mutation-based approach (i.e., Hermes [4]), we use CCG and REMCCG to generate seed programs and conduct the same mutation operations to produce test programs for compiler testing. The results show that the seed programs generated by REMCCG can help Hermes to yield 14% and 11% more bugs than CCG in GCC and LLVM, respectively. Then, we run REMCCG over the development versions of GCC and LLVM to evaluate its practical bug-finding capability. REMCCG has reported 56 new bugs (37 out of them have been fixed already) for the compilers. It is worth noting that many of the reported bugs are serious, deep, and long-latent bugs (5 bugs have been marked with the highest severity and 2 bugs are lurking for 1 year and 3 years).

In summary, this paper makes the following contributions:

- To the best of our knowledge, we first propose the idea of remanufacturing a program generator for compiler testing.
- We design REMGEN which leverages the diverse code snippets synthesis and the bug-revealing code snippets selection for effectively remanufacturing a program generator.
- We implemented the tool REMCCG[4], a case study of REMGEN, and conducted extensive experiments to show the effectiveness of REMCCG. In practice, REMCCG has found 56 (37 fixed already) new bugs in two compilers.

The remainder of this paper is organized as follows. Section II introduces the background and an illustrative example. Section III describes our framework REMGEN. Evaluation results are presented in Section IV. The discussion, threats, and related work are described in Sections V-VII, respectively. Section VIII concludes this paper with future work.

II. BACKGROUND AND ILLUSTRATIVE EXAMPLE

In this section, we first introduce the background of program generators and remanufacturing. Then, we present the idea of REMGEN and use an illustrative example to highlight the advantages of the new generator remanufactured by REMGEN.

A. Program Generators

Program generators, CCG [3], Csmith [2], and YARPGen [8], allow users to create brand new test programs for compiler testing. These generators share the same simplified workflow as shown in Fig. 2. Typically, the program generator starts by initializing a seed number (①). Under the given seed number, a test program is deterministically generated (i.e., the generated test program is the same when executing the program

[4]REMCCG is publicly available at https://github.com/haoxintu/RemCCG.

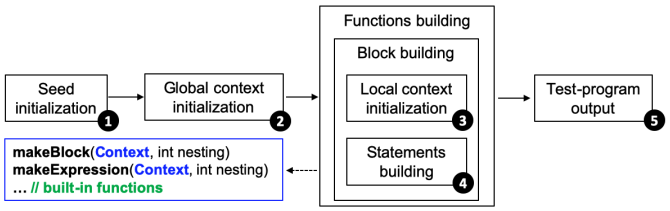

Fig. 2: The basic workflow of a program generator

generator with the same seed number). To generate a test program, the program generator first creates a global context (②), which contains the basic properties in a specific scope (e.g., variable names in the global scope). Next, the program generator builds functions based on the current context. Inside the function building, a block is built with a local context (③), which includes the definitions of local variables; some statements are also produced inside the block (④). When all functions are created, the program generator prints the fresh test program (⑤). Users can redirect the *stdout* to a source file and use this file for compiler testing.

It is worth noting that those program generators hold at least two important and useful capabilities. First, they support various built-in functions (e.g., `makeBlock` and `makeExpression`) to generate different new valuable code snippets. Typically, by invoking those functions multiple times, various code snippets can be generated. Second, the context (i.e., one of the parameters used in the built-in functions) used in generating code snippets can be reserved and then reused in a lightweight manner. For example, we can directly reserve the context and call different built-in functions to generate required code snippets and then use them to synthesize new test programs. Due to the hardness of the creation of such tools (e.g., Csmith [2] was implemented for more than two years), such hidden capabilities should be effectively activated in generators so as to continually detect new compiler bugs. However, the above capabilities are not fully studied yet.

B. Remanufacturing

Remanufacturing is a process of rebuilding a used product into a "like-new" product, aiming to make a used product effective again, which is being regarded as a sustainable mode of manufacturing because it can be profitable and less harmful to the environment than conventional manufacturing [18], [19]. It has been successfully adopted in many areas, e.g., automobile parts, aerospace, and medical devices [19]–[21]. Typically, remanufacturing can be broadly performed in the following three processes:

(1) Preparation for remanufacturing. The task of this process is to investigate and assess the feasibility of the old product for remanufacturing, including inspecting, disassembling, and preprocessing subparts of the old product.

(2) Remanufacturing. This process focuses on remanufacturing by introducing new components into disassembled parts and then reassembling the new parts together.

(3) Testing the remanufactured product. This process tests the effectiveness of the remanufactured product.

C. The idea of REMGEN and an illustrative example

1) The idea of REMGEN*:* In this study, we propose REMGEN to migrate the idea of remanufacturing a product to a program generator for compiler testing. Our key insight is that by leveraging the hidden capabilities in existing program generators, we can potentially make tamed generators effective again. Specifically, by doing so, REMGEN can have at least the following three benefits. First, by directly reserving the context from generators, we do not need to perform heavy profiling to extract context information like existing approaches, e.g., Hermes [4], (see more details in Section III-B1). Second, we do not need to take extra care to maintain validity of code when conducting code synthesis as the old and new code snippets share the same semantic context. Third, the code snippets generated by built-in functions are notoriously valuable for revealing new compiler bugs as existing program generators are proven to be effective for compiler testing. Benefited from the capabilities, in REMGEN, we mainly apply the three corresponding processes mentioned in Section II-B in traditional remanufacturing on a program generator as follows:

(1) Preparation process. We first inspect whether the source code of the generator is available and can be built and run in our machine. If yes, we then disassemble the generator by dividing different built-in functions mentioned in Section II-A into subparts (e.g., `makeBlock`). Lastly, we preprocess those subparts to make them easy to integrate with other parts (e.g., the newly designed components).

(2) Remanufacturing process. We design two new components, i.e., the diverse code snippets synthesis and the bug-revealing code snippets selection, into the input generator, thus enabling the remanufacturing of an old generator into a new generator. This is the main process of remanufacturing.

(3) Testing process. We test the bug-finding capability of the remanufactured generator.

In summary, given an old program generator (e.g., CCG), by remanufacturing it under REMGEN, the remanufactured generator (e.g., REMCCG) can generate bug-revealing test programs again. Next, we use an example to show the limitations of the existing approaches and the advantages of REMCCG.

2) An illustrative example: Fig. 3 shows a *performance* bug in LLVM triggered by a bug-revealing test program generated by REMCCG. For the sake of presentation, we only show its *reduced* version. The bug-revealing code snippet synthesized by REMCCG is in between Lines 4-15. Among these lines, Lines 9-14 (highlighted in gray) are generated by calling the built-in function (i.e., `makeBlock`) in program generators.

The root cause of the bug. The bug-revealing program makes the trunk version of LLVM spend an infinite compile-time under -O3. The root cause of this bug is that LLVM misuses an unbound *llvm.assume* scanning when performing the "loop unrolling" optimization, an important and widely-used loop optimization technique in modern compilers [22]. Specifically, the *llvm.assume* instruction allows the optimizer to assume that the provided condition is true. In this case, LLVM applies an unlimited assume scanning on the *if-branch*

```
1  int a, b, c, d;
2  void e() {
3      ... // code snippets generated by CCG
4      a = 7;
5      for (; a <= 78; a++) {
6          d = 3;
7          for (; d <= 73; d++) {
8              // code produced by makeBlock() highlighted in gray
9              int f = 0;
10             b += c;
11             if (b) {
12                 int g = 0;
13                 for (f = 5; f; g);
14             }
15         }
16     }
17 } /*  Grammar Coverage : G={0,0,0,2,0,0,0,0,0,0,0} */
```

Fig. 3: LLVM 13.0 hangs at compile time (#49171)

at Line 11, thus causing the compiler-time explosion issue. Developers have fixed this bug promptly.

Limitations in existing approaches. This bug is difficult to be triggered by the generation-based approach CCG [3], as the bug disappears after removing our synthesized code snippet (between Lines 4-6). Besides, existing mutation-based approaches (e.g., Orion [6], Athena [5], or Hermes [4]) are also limited to generating such a code snippet for the following reasons. First, Orion and Athena can only modify the dead region of the code, while the bug-revealing code snippet in the example is in a live code region. Second, although Hermes is able to insert code snippets in the live code region, such code snippets synthesized by Hermes are restricted. For example, the local variable `f` in Line 9 can not be synthesized under Hermes's synthesizing strategy. Worse still, the time cost for code snippets synthesis in Hermes is relatively high which may further burden generating bug-revealing test programs.

Advantages of our approach. REMCCG is capable of generating such code snippets to trigger compiler bugs. Here, REMGEN is a framework that remanufactures a program generator by appending two new components (i.e., diverse code snippets synthesis and bug-revealing code snippet selection) to the existing workflow of the program generator. Specifically, during code snippets synthesis, REMCCG first reserves (rather than extracting from source code) the global and local context, including all possible variables such as `a`,`b`,`c`,`d`, and `f` from the program generator. Then, REMCCG invokes the code snippets synthesis component to yield two *for-loop* statements, along with code snippets produced by executing `makeBlock` using the reserved context. Finally, the whole code snippets are integrated on Line 3 onward to generate a new bug-revealing test program. Noted that during the generation process, the number of synthesized code snippets could be large. We leverage the component of the bug-revealing code snippet selection to select the most diverse to construct new programs. To do so, we introduce the grammar coverage to differentiate various code snippets. For example, in Fig. 3, we maintain a recording list (i.e., `{0,0,0,2,0,0,0,0,0,0,0}`), where each dimension represents a specific grammar rule, and the values show the frequency of the grammar rule in the code snippet. Specifically, the value 2 in the list means that two *for-loop* grammar rules are covered in this code snippet (after excluding code snippets synthesized by the built-in functions). We update the list during the synthesis process and use it as a metric for guiding the selection.

Note that existing generators may still generate similar code snippets to the code synthesized by REMCCG in theory. However, it is only possible if those program generators tailor their designs for different purposes. For Csmith [2] and YARPGen [8], they are designed to generate test programs free of undefined behaviors, which is a relatively strict requirement that may decrease the diversity of test programs. For CCG [3], it may generate test programs similar to ours but it is practically hard due to too much randomness of its test program generation. In contrast, our approach targets generating more diverse and complex test programs (i.e., syntactic valid programs which may include undefined behaviors) that complement Csmith [2] and YARPGen [8].

III. APPROACH

In this section, we first present the overview of our proposed REMGEN. Then, we detail the processes (i.e., preparation, remanufacturing, and testing) designed in the framework.

Overview. The high-level architecture of REMGEN is presented in Fig. 4, which generally takes an old program generator as input and outputs certain potential compiler bugs. Specifically, the main workflow in REMGEN can be separated into three major processes: (1) a preparation process to investigate the feasibility of the input program generator for remanufacturing, including inspecting, disassembling, and preprocessing subparts of the input generator, (2) a remanufacturing process to append two new components (i.e., ⑥ code snippets synthesis and ⑦ code snippets selection) into the original workflow of the input generator (as shown in Fig. 2), then reassemble all the components together to a new remanufactured generator, and (3) a testing process to evaluate the effectiveness of the remanufactured generator, either conducting generation-based or mutation-based approaches to test compilers. Next, we describe each process in detail.

A. Preparation process

The task of preparation is to investigate and assess the feasibility of the input program generator in terms of remanufacturing. Generally, we assess the remanufacturing of an old program generator by manually building and running it. We also perform a code review to confirm whether it has the same workflow as presented in Fig. 2. It is worth noting that such an assessment does not need to involve too much laboratory work, as the developers of existing program generators (e.g., CCG [3], Csmith [2], and YARPGen [8]) have spent many years implementing the tool. Therefore, the documents of the building and running tutorial and code styles of them are easy to follow and understand. Specifically, as shown in Fig. 4, we perform the preparation as follows:

Inspect: Checking the functionality from the "appearance" of the program generator, e.g., checking whether the program generator can be built and run normally.

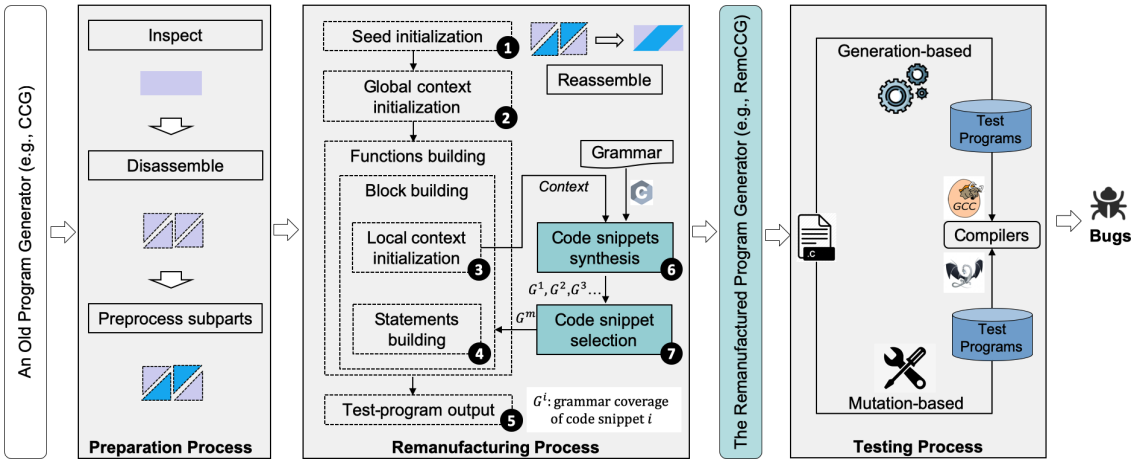

Fig. 4: The high-level architecture of REMGEN

Disassemble: Decomposing program generation components to be modularized, e.g., separating the logic of different expressions/statements/blocks building in the implementation of the program generator (as shown in Fig. 2).

Preprocess subparts: Reconstructing required components (e.g., built-in functions in program generators) to be easily integrated with other components. For example, the context (either global or local) should be smoothly collected as an external variable so as to be used in the new components.

After the preparation process, two built-in functions `makeBlock` and `makeExpression` are prepared for the following remanufacturing process, as they can generate different kinds of supported statements/expressions in the old generators. Therefore, it is sufficient to use them to synthesize diverse code snippets for our purpose.

B. Remanufacturing process

In this subsection, we first describe the fundamental workflow of the remanufacturing process. Then, we detail two main components of REMGEN: the diverse code snippets synthesis component and the bug-revealing code snippet selection component. As shown in Fig. 4, traces ①, ②, ③, ④, and ⑤ are the same in the original generator as presented in Fig. 2. which are the subparts disassembled from a program generator. Apart from the existing components in the program generator, two new components (i.e., the code snippets synthesis component ⑥ and the code snippet selection component ⑦) are designed respectively for synthesizing diverse code snippets and selecting the most diverse one for constructing new bug-revealing test programs. Appending to the two new components, all the disassembled subparts are reassembled to a new remanufactured program generator for compiler testing.

*1) **Diverse code snippets synthesis**:* This component is designed for synthesizing diverse code snippets at a low cost. It takes the required context and a set of grammars as inputs, and outputs a set of diverse code snippets. In this subsection, we first give the metric definition of "*diversity*", which is adapted from related studies, i.e., the grammar coverage of code snippets. Then, we illustrate our strategy for reserving the required context, which is one of the main capabilities used in REMGEN for reducing the synthesizing overhead compared with existing approaches. Finally, we present the detailed synthesis process.

Definition of grammar coverage. The essence of REMGEN is to synthesize new code snippets from the code snippets generated by calling built-in functions in the tamed program generators and integrate the new code snippets into the complete test program. Thus, it is necessary to use a metric to distinguish different code snippets so as to select the most diverse one to construct new test programs. To measure the diversity of those code snippets quantitatively and efficiently, we adopt a metric, i.e., grammar coverage, from related works in this study. We mainly leverage this metric to approximate the complexity/expressiveness of the generated test programs. We follow the underlying assumption in existing studies [2], [8] that complex/expressive test programs are more likely to exercise the deep code regions in compilers thus potentially triggering more compiler bugs. Specifically, we define the metric *grammar coverage* for distinguishing various code snippets as follows:

DEFINITION 3.1 (*Grammar coverage*) *the grammar coverage of a code snippet* `CS` *refers to the number of grammar rules invoked during the synthesis process:*

$$G(\texttt{CS}) = (G_1, G_2, \ldots, G_i, \ldots),$$

where G is a recording list which represents the frequency of the grammar rule used in the code snippet. Specifically, it contains a set of grammar rules G_i, and i represents a specific grammar rule. For example, G in Fig. 3 is "`{0,0,0,2,0,0,0,0,0,0,0}`", which means only two *for-loop* statements are used in the synthesized code snippet. This metric can be used to represent various kinds of code snippets; a code snippet with a larger grammar coverage implies that it may have a more complex structure, which is likely to exercise the compilers more thoroughly.

Context reservation. Before synthesizing new code snippets, we need to collect the required code contexts. Typically, there are two kinds of contexts to be considered based on

the different scopes of program semantics. One is the context in the local area (i.e., inside a function definition); another is the context in the global area (i.e., outside a function definition). Generally, a global context (e.g., global variables) can be used in both global and local areas. However, a local context can only be used in a local area, because assigning a local variable outside the definition scope could introduce an "undefined variable" error. Existing strategies usually collect the context of code snippets by laboriously extracting/profiling source code [4], [5]. However, such strategies are proven to be time-consuming. For example, the previous study shows that it takes 1.7s on average (at most 6s) to perform profiling for synthesizing a new test program, while it usually takes less than 1s in generating the seed program [4].

To reduce the overhead of the context collection, unlike existing approaches, we reserve the required context from the input program generator in a more lightweight manner. Specifically, as shown in Fig. 4, since we have disassembled the generators into different subparts, we can easily obtain and reserve the global or local context from the input program generator. A minor issue we need to consider is to distinguish the different uses of the context based on different scopes (i.e., global or local). For example, it works well if we use a global context to generate code snippets and then integrate them into a function definition. However, a local context can only be used in a local area due to the reason aforementioned. In this study, we reserve both global and local contexts before synthesizing new code snippets. In particular, our reservation strategy has two advantages. First, it can naturally guarantee the validity of the generated code snippets, because those code snippets are synthesized under the same semantic context. Second, the collection is lightweight, as we do not need to profiling the original test program for extracting the required context. Given the reserved contexts of an original code snippet, we start to synthesize diverse code snippets.

Code snippets synthesis. One straightforward solution is to directly synthesize new code snippets by invoking the built-in functions in program generators and integrating code snippets into a test program (i.e., the way works by the tamed generators). However, it is insufficient for constructing bug-revealing test programs (more details in Section V). Therefore, we propose a grammar-aided synthesis strategy to mitigate the limitation. Specifically, our code snippets synthesis is aided by a set of grammars (C grammars in this study). The synthesis is built top-down based on the reserved context. Algorithm 1 presents the detailed process of the code snippets synthesis. The synthesis takes a context `C`, a set of grammars `T`, and a bound `N` as inputs; it outputs a set of code snippets `CS` along with the updated grammar coverage G for each code snippet. The bound `N` is used to limit the number of code snippets. Specifically, the process is driven by the function `synCode`. `synCode` first initializes a grammar coverage G and a depth counter `depth` in Lines 2-3. Then, it synthesizes new statement sequences by calling the function `synStmtSeq` with the arguments of the context, the depth counter, and the grammar (Line 6). When the bound is reached, the synthesis is done

Algorithm 1: Code snippets synthesis in REMGEN

```
   Input: a context C, a set of grammars T, and a bound N
   Output: a set of code snippets CS (with grammar coverage G)
1  Function synCode(C, T, N):
2      G[11] ← ∅;   // initialize a coverage vector
3      depth = 0;   // initialize a variable depth
4      i = 0;   // initialize a counter
5      while i < N do
6          CS_i(G^i) = synStmtSeq(C, depth, T);
7  Function synStmtSeq(Context cc, int depth, T):
8      i = i + 1;
9      if random(0,1) then
10         CS(G^i) = synStmt(cc, depth, T);
11     else
12         CS(G^i) = synStmtSeq(cc, depth, T);
13     return CS(G^i);
14 Function synStmt(Context cc, int depth, T):
15     if depth > max_depth then
16         return CS(G^i);
       // Randomly choose a statement to synthesize
17     switch T.type do
18         case AssignStmt do synAssignStmt(cc); G^i[0]++;
19         case WhileStmt do synWhileStmt(cc, depth);G^i[1]++;
20         ... // handle other statements
21         case CompStmt do synCompoundStmt(cc);G^i[9]++;
22         case JumpStmt do synJumpStmt(cc, depth);G^i[10]++;
23     return CS(G^i);
```

and a set of code snippets `CS` will be the candidates for constructing new bug-revealing test programs.

The function `synStmtSeq` in Lines 7-13 in Algorithm 1 is to synthesize a sequence of statements by calling itself or a single statement by calling the function `synStmt`. We randomly decide the order of the statements in sequences. Such a random strategy is also the suggested strategy in the existing studies [4], [5]. The main logic of the function `synStmt` is divided into two parts (Lines 14-22). First, it checks whether the current `depth` is larger than the defined `max_depth` (Lines 9-10). Second, if the check fails, a random statement is synthesized aided by `T` (Lines 17-22), and the grammar coverage will be updated accordingly. It is worth noting that existing synthesis strategies tend to generate less diverse code snippets as they only synthesize a small fraction of limited statements (e.g., Always False Conditional Block (FCB), Always True Guard (TG), and Always True Conditional Block (TCB) [4]). We then apply all the defined statement-level C grammar rules to aid the synthesis of diverse code snippets. Note that CCG does not support a few statements (i.e., *switch*, *while*, and *do-while*); we evaluated this and noted that the improved bug-finding capability of REMCCG is mainly from our proposed components rather than the newly supported statements (more discussion in Section V). Since the process of synthesizing different statements is similar, for simplicity, we only show the skeleton of synthesizing a *for-loop* statement in detail.

Algorithm 2 shows the synthesis skeleton of a *for-loop* statement. Notably, instead of exactly following the semantics of defined statements in C grammars, we manually construct the structure of a statement based on the following two guidelines: (1) we try to synthesize a larger code block to be executed both as live or dead code (that can or cannot

Algorithm 2: Synthesizing a *for-loop* statement

```
Input: a context C, a synthesis depth depth, and a set of grammars T
Output: a for-loop code snippet
1  Function synForStmt(C, depth):
2      cmp ← {<, >, =, >=, <=}; // comparison operators
3      op ← {++, + =, −−, − =}; // increment/decrement op
4      depth + = 1;
5      var = randomVar (C); // select a variable from C
6      buildPredicate (var, cmp, op, randomNumber)
7      out « "{" // for-loop body starts
8      makeBlock (C, nesting)
9      CS_for = synStmtSeq (C, depth, T)
10     out « "}" // for-loop body ends
```

be covered when executing the code, respectively), as both live/dead code synthesis [4]–[6] have proven to be effective for discovering deep bugs in compilers, and (2) we try to reuse as much information in the collected context as possible (such as variables in for conditions) to increase the diversity of code snippets. As such, the synthesis of a *for-loop* statement in Algorithm 2 includes two major parts, i.e., predicate building (Line 6) and body building (Lines 8-9).

For the predicate building, we separately synthesize the true and false predicates. According to the true/false of the built predicate, we can synthesize a statement as a live code or a dead code. To make the *for-loop* statement executable as a live code, we first initialize two kinds of operators in advance, i.e., the comparison operator and the increment/decrement operator. Then, we randomly select a variable from the existing context `C` via the function `randomVar`. Next, we synthesize a live predicate for *for-loop* condition. Specifically, we build a predictive to be true following two guidelines: (1) choose an appropriate *randomNumber* and cmp that evaluate "*var cmp randomNumber*" to be true, and (2) choose an appropriate op; for example, if we select the compare operator "<" or "<=", the corresponding operator selected later should be "++" or "+=". For synthesizing a false predicate, we directly produce a false condition (e.g. 0) as the predicate. For the possibility of synthesizing a true or false predicate, we set the same possibility (50% from each) for them. When synthesizing the body (Lines 7-10), we first call a built-in function (i.e., `makeBlock`) in the input program generator to build a block and then invoke the `synStmtSeq` function to synthesize statements sequences in the body. Here, we use the default parameter *nesting* in the input program generator to control the program size and the number of recursions.

In summary, the above synthesis process could yield a vase number (controlled by the parameter `N` in Algorithm 1) of diverse code snippets, and any of them can be a candidate to be integrated into the test programs. To improve the effectiveness of bug-revealing test program generation, we need to consider a feasible way to select the bug-revealing code snippet to construct new test programs.

2) ***Bug-revealing code snippets selection****:* We now describe our solution for selecting bug-revealing code snippets by leveraging the introduced grammar coverage in Section III-B1.

Given a vast number of code snippet candidates along with their unique grammar coverage for each candidate, we calculate the sum of the square of the grammar coverage of each candidate, where a larger value represents higher diversity in a code snippet. This choice is justified by the fact that measuring the Euclidean distance between the grammar coverage of a candidate and coordinate origin was shown to be effective in differentiating test programs [23]. Then, we apply the above calculation results in ordering various code snippets.

We opt for the diverse code snippet with the largest value as the bug-revealing one. When the code snippet is selected, we integrate it into suitable positions in a test program. Conceptually, the code snippet can be integrated into any possible positions after local context initialization in ③ in Fig. 4. In our study, we consider the positions where a statement has been built in a function body, i.e., integrating it into the position where after the last statement is built (④) inside the function building. This choice is motivated by the fact that there is a clearer boundary between newly synthesized code snippets by our proposed components and existing code snippets by the existing program generator, which makes it easier for investigating the effectiveness of REMCCG.

C. Testing process

As shown in Fig. 4, a new generator REMCCG is produced after remanufacturing the given program generator CCG, and we expect it to generate bug-revealing test programs again. Thus, we conduct a testing process to assess the effectiveness of the remanufactured program generator in terms of bug-finding capability. Specifically, a generation-based or mutation-based approach can be applied to test compilers. We present this process in the following section.

IV. EVALUATION

In this section, we conduct extensive experiments to evaluate the effectiveness of REMGEN. In particular, we seek to investigate the following two research questions (RQs):

- **RQ1**: Can REMCCG boost both generation-based and mutation-based approaches for compiler testing?
- **RQ2**: Can REMCCG find new compiler bugs in practice?

RQ1 uses old versions of compilers to assess the effectiveness of REMCCG in boosting the state-of-the-art generator-based and mutation-based approaches in terms of bug-finding capability. RQ2 uses the development versions of compilers in practice to assess the bug-finding capability of REMCCG. Besides, we have also evaluated the effectiveness of the two proposed components, but only present abbreviated results in Section V due to the page limit.

A. Implementation and evaluation setup

1) ***Implementation****:* We take the tamed program generator CCG [3] as a case study. We remanufacture CCG and implement REMCCG in our study. The main reason is that CCG is old enough and has not been updated for a long time (which is no longer maintained after 2016 when we used it). This fact would be a shred of convincing evidence to demonstrate the effectiveness of our approach if we can find new compiler bugs through such an old tamed program generator. That means,

TABLE I: Results of Boosting in Generation-based Approach

Subject	Tools	Average Statistics			
		Cra.	*Perf.*	*Sum.*	*Imp.*
GCC	CCG [3]	2.9	0.3	3.2	16%
	REMCCG	3.1	0.6	3.7	-
LLVM	CCG [3]	9.2	2.7	11.9	11%
	REMCCG	9.7	3.5	13.2	-

TABLE II: Results of Boosting in Mutation-based Approach

Subject	Tools	Average Statistics			
		Cra.	*Perf.*	*Sum.*	*Imp.*
GCC	Hermes(CCG)	3.0	0.5	3.5	14%
	Hermes(REMCCG)	3.2	0.8	4.0	-
LLVM	Hermes(CCG)	9.8	3.6	13.4	11%
	Hermes(REMCCG)	10.6	4.3	14.9	-

the remanufactured REMCCG should have a good bug-finding capability in practice and be able to boost both the state-of-the-art generation-based and mutation-based approaches. Due to the limited functionality of CCG (as explained in Section V), we aim to detect *crash* or *performance* bugs in compilers in this study. To implement the synthesis strategy, we combine a subset of C grammars in Grammar-v4 [24], which is a rich collection including various ANTLR v4 [25] grammars, with CCG [3]. We take ProtoBuffer [26], a practical format that can be easily manipulated during transformation, as an intermediate representation to smoothly convert C grammars into real C programs. Besides, the bound of synthesized code snippets in the synthesis phase is user-configurable and we set the bound to 10 (see more discussion in Section V).

*2) **Evaluation setup**:* Our experiments run on a Ubuntu 18.04 server with Intel(R) Core(TM) i7-6900K CPU @ 3.20GHz × 16 processors and 64GB RAM. For testing subjects and running options, we choose two popular compilers (i.e., GCC and LLVM) on their five standard options, "-O0", "-O1", "-Os", "-O2", and "-O3", which follows the existing compiler testing studies [2], [4]–[7].

Comparison approaches for RQ1. To demonstrate the effectiveness of REMCCG in boosting generation-based approaches for compiler testing, we compare REMCCG with CCG [3]. To evaluate the effectiveness of REMCCG in boosting mutation-based approaches for compiler testing, we opt for Hermes [4] as our target since it has proven to be the state-of-the-art approach that outperforms other existing approaches (i.e., Orion [6] and Athena [5]).

Duplicate bug identification. In our evaluation, every bug detected by each tool is counted as a unique bug. Therefore, we need to filter duplicate bugs. For crash bugs, we compare the stack trace or the assertion information emitted by compilers. However, it is not easy to filter performance bugs, because there is no other information that can be referred to rather than analyzing compiler source code deeply, which is not a practical solution. To this end, we adopt a different strategy to filter them. We follow the previous study [27] (i.e., comparing the first buggy commit) to identify unique bugs for RQ1. For RQ2, we detect brand new compiler bugs on the development version of compilers. We directly file new bug reports to developers and seek their help to identify them.

B. Answer to RQ1

To answer RQ1, we run REMCCG against state-of-the-art generation-based approach (i.e., CCG [3]) and mutation-based approach (i.e., Hermes [4]) over two old versions of mainstream compilers (i.e., GCC-4.4.3 and LLVM-2.6) following the existing work [10], [11]. To conduct a fair comparison, we run each approach under the same testing period of 90 hours and 10 times (the same setting as [9], [27]) and then count the average of the total number of detected bugs.

The comparison with the generation-based approach is shown in Table I. The first column represents four tools under comparison. The next four columns are the average statistic of bugs detected by each approach. We count the number of crash bugs (*Cra.*), the performance bugs (*Perf.*), the sum of detected bugs (*Sum.*), and the improvement of detected bugs (*Imp.*). Here, *Imp.* is the relative improvement of REMCCG over the comparative approach. As shown in Table I, we can observe that REMCCG finds 16% and 11% more bugs than CCG under the testing subject of GCC and LLVM.

To evaluate the effectiveness of REMCCG in boosting the state-of-the-art mutation-based approach (i.e., Hermes [4]), we use the baseline CCG and remanufactured REMCCG as a program generator to generate seed programs. Following the mutation strategy in Hermes, we conduct mutations (i.e., inserting FCB, TG, and TCB) upon seed programs to construct test programs for compiler testing. Table II shows the results of the two approaches. It clearly shows that Hermes with REMCCG can detect 14% and 11% more bugs than Hermes with CCG in GCC and LLVM, respectively. For all the results in Table I and Table II, we conducted the Mann-Whitney U-test [28] with a level of significance of 0.05 on the total bugs between REMCCG and the comparative approaches. The calculated p-value is less than 0.05, which indicates our experiments are statistically significant. Note that the number of detected bugs in LLVM is larger than in GCC (e.g., 3.7 vs 13.2 in Table I). This may be because of the age of the two compilers. Although two compilers were released very close in 2010, GCC has a longer history than LLVM (the first version of GCC was released in 1987 while LLVM was in 2003). Therefore, it is reasonable that LLVM is more fragile. It is also worth noting that the capability hidden in REMCCG is to find new bugs, which is demonstrated later in RQ2.

C. Answer to RQ2

To evaluate the practical bug-finding capability of REMGEN, we test the daily updated development trunk version of GCC and LLVM in the non-continuous period from middle February to late September in 2021. We follow the existing studies to detect bugs over trunk versions as compiler developers always fix bugs in the development version more promptly than the released versions [4]–[6], [9]. We evaluate REMCCG from three aspects, i.e., the number of detected bugs, the type of fixed bugs, and the importance of those bugs.

TABLE III: Results of All the Reported Bugs

Bug Status	GCC	LLVM	Total
Fixed	8	29	37
WorksForMe	0	2	2
Duplicate	2	3	5
Pending	0	12	15
Total	10	46	56

TABLE IV: Results of Bug Types of Fixed Bugs

But Types	GCC	LLVM	Total
Crash	6	16	22
Performance	2	13	15
Total	8	29	37

Detected bugs. As shown in Table III, we reported 56 new bugs for two compilers, of which 61% of the bugs (i.e., 37) have been fixed already. Notably, the number of bugs reported to LLVM (29 fixed among a total of 46) is larger than GCC's (8 fixed among a total of 10). The reason could be that the unique and complex features of optimization components in LLVM can cause more bugs. For example, the different sequences of optimal options in LLVM may also lead to severe bugs [9]. Since it takes some time for developers to confirm reported bugs, the trunk version can update very frequently during that time. Therefore, some changes may suppress the reported bugs. Those bugs are marked as "WorksForMe". We have two such kinds of bugs. GCC does not have such a status as developers of GCC have a quick response to the newly filed bugs. Besides, because developers normally spent more than one year on average to fix bugs [29], [30], there are still 12 bugs pending the response of developers. In addition, we have 5 duplicate bugs, of which 3 bugs are *performance* bugs. Two tricky GCC crash bugs (e.g., *bug#100578*) were also marked as "duplicate" because they emit the different assertion information as the existed bug report (*bug#100512*). We misunderstand this bug thus causing the duplicate report. Table V further lists all the 37 fixed bugs, including the subject compiler and its bug ID (Compiler-ID), priority, type, affected optimizations (Affected. Opt.), and affected versions.

Bug types. We classify bugs by the following criteria. If a compiler crashes during compilation (e.g., abnormally terminates with an internal compiler error in GCC or an assertion failure in LLVM), a *crash* bug is detected. If a compiler spends a quite long time (i.e., 30 seconds, following a prior study [2]), to compile a test program, a *performance* bug is detected. Based on the above taxonomy, we classify our 37 fixed bugs into two main categories as shown in Table IV.

Bug importance. Developers treat our reported bugs seriously, and they have fixed most (66%) of them so far. Specifically, GCC developers are generally more responsive and fix all of our reported bugs except for a duplicate one. The explicit way to measure the importance of our bug is via the "Importance" field set by developers in bug reports. From Table V, developers marked 5 out of 8 GCC bugs as "P1" or "P2", i.e., the two highest priorities (the default is "P3"). For LLVM, developers marked all our bugs as the default value "P normal" as they usually do not classify the bug priority. Even so, a plurality of bugs is backporting on the released version (e.g., *bug#49205*, *bug#49218*, *bug#49475*, *bug#49541* to 12.0.0, and *bug#50308* to 12.0.1), which demonstrates the importance of those bugs as developers usually backport the most severe bugs to the released versions.

TABLE V: Details of Fixed Bugs

	Compiler-ID	Priority	Type	Affected. Opt.	Affected Versions
1	GCC-99694	P2	*Perf.*	-O1,2,3	**9.3-11.0 (trunk)**
2	GCC-99880	P2	*Crash*	-O3	10.2-11.0 (trunk)
3	GCC-99947	P1	*Crash*	-O3	11.0 (trunk)
4	GCC-100349	P2	*Crash*	-O2,3,s	11.0-12.0 (trunk)
5	GCC-100512	P3	*Crash*	-O2,3,s	12.0 (trunk)
6	GCC-100626	P2	*Crash*	-O1,2,3,s	11.0-12.0 (trunk)
7	GCC-102057	P3	*Crash*	-O1,2,3,s	12.0 (trunk)
8	GCC-102356	P3	*Perf.*	-O3	11.0-12.0 (trunk)
9	LLVM-49171	P3	*Perf.*	-O3	13.0 (trunk)
10	LLVM-49205	P3	*Perf.*	-O1,2,3,s	11.0-13.0 (trunk)
11	LLVM-49218	P3	*Crash*	-O1	12.0-13.0 (trunk)
12	LLVM-49396	P3	*Crash*	-O2,3,s	12.0-13.0 (trunk)
13	LLVM-49451	P3	*Crash*	-Os	13.0 (trunk)
14	LLVM-49466	P3	*Crash*	-O2	13.0 (trunk)
15	LLVM-49475	P3	*Perf.*	-O1	12.0-13.0 (trunk)
16	LLVM-49541	P3	*Perf.*	-O2,s	**7.0-13.0 (trunk)**
17	LLVM-49697	P3	*Crash*	-O3	**7.0-13.0 (trunk)**
18	LLVM-49785	P3	*Perf.*	-O3	13.0 (trunk)
19	LLVM-49786	P3	*Perf.*	-O2	13.0 (trunk)
20	LLVM-49993	P3	*Crash*	-O3	13.0 (trunk)
21	LLVM-50009	P3	*Crash*	-Os	12.0-13.0 (trunk)
22	LLVM-50050	P3	*Crash*	-O2,3,s	13.0 (trunk)
23	LLVM-50191	P3	*Crash*	-O2	13.0 (trunk)
24	LLVM-50238	P3	*Crash*	-O1,2,3,s	13.0 (trunk)
25	LLVM-50254	P3	*Perf.*	-O2,3	13.0 (trunk)
26	LLVM-50279	P3	*Perf.*	-O3	13.0 (trunk)
27	LLVM-50302	P3	*Perf.*	-O3	13.0 (trunk)
28	LLVM-50307	P3	*Crash*	-Os	13.0 (trunk)
29	LLVM-50308	P3	*Perf.*	-O1,2,3,s	12.0-13.0 (trunk)
30	LLVM-51553	P3	*Crash*	-O3	14.0 (trunk)
31	LLVM-51584	P3	*Perf.*	-O1,2,3,s	14.0 (trunk)
32	LLVM-51612	P3	*Crash*	-O2,3	14.0 (trunk)
33	LLVM-51656	P3	*Crash*	-O2,3	14.0 (trunk)
34	LLVM-51657	P3	*Perf.*	-O2,3,s	12.0-14.0 (trunk)
35	LLVM-51762	P3	*Perf.*	-O1	14.0 (trunk)
36	LLVM-52018	P3	*Crash*	-O3	14.0 (trunk)
37	LLVM-52024	P3	*Crash*	-O2	14.0 (trunk)

Affected optimization levels. Our reported bugs exist in the deep code regions of compilers. Based on previous studies [29], [30], optimization bugs are tricky to be discovered. Our aim in this study follows the existing work, and all of our bugs are related to the optimization phase. Specifically, among the fixed bugs, 26 are caused by the option of "-O3" (which turns on almost all the optimization options by default), and 6 bugs occurred in all "-O1" to "-Os" options. The above result further bears out REMCCG is able to reveal deep bugs in compilers.

Affected compiler versions. Our approach can find many long-latent bugs. Although our focus is to test the development versions of GCC and LLVM, we have found 10 bugs in the relatively old versions of the two compilers (shown in Table V). It is also worth noting that two bugs have existed for many years. One is for GCC (*bug#99694*) and another is for LLVM (*bug#49541*), for which exist more than 1 year and 3 years, respectively. Note that the trunk version of GCC and LLVM was changed during our testing period [31], [32].

V. DISCUSSION

Effectiveness of the two proposed components. To evaluate the effectiveness of two proposed components, we compare REMGEN with its variants, including REMCCG(¬G) (with

random synthesis using the built-in functions in program generators rather than the grammar-aided synthesis), REMCCG($\neg$S) (without the selection strategy), and REMCCG(S_R) (with random selection rather than grammar coverage guided selection). Specifically, we take CCG as the *baseline* and run those variants for 24 hours in GCC-4.4.3. We repeat the experiments 5 times and count the average number of detected bugs. The results show that REMCCG can find 2.6 bugs, while REMCCG($\neg$G), REMCCG($\neg$S), and REMCCG(S_R) can only find 2.4, 2.0, and 2.2 bugs, achieving 8%, 30%, and 18% improvement, respectively. To further understand the impact of newly introduced language features (e.g., *while-loop*) in REMCCG, we check the bug-revealing test program reported in RQ2. The results show only 5 out of 56 contain new language features that do not exist in the original CCG, which indicates that 91% of bug-revealing test programs are generated by our effective remanufacturing process.

Bound selection. We have set the bound (i.e., N) to 10 in Algorithm 1. However, it is unclear which bound could be a better choice. We thus set different bounds (i.e., 3, 5, 10, 20, 40, 60, 80, and 100) to evaluate its bug-finding capability. We run various bounds in GCC-4.4.3 for 24 hours, repeating 5 times. The results show that the value 10 is the preferred setting. This is reasonable as a small bound restricts the diversity of the generated test program. Although a diverse test program can be synthesized by a large bound, it also increases the time cost of code snippet selection and program compilation. For a better trade-off between selection and bug-finding, we opted for a moderate value (i.e., 10) in REMCCG.

Comparison with Csmith [2] and YARPGen [8]. We also conducted extra experiments to compare REMGEN with another two well-known generation-based approaches (i.e., Csmith [2] and YARPGen [8]) under the same setting used in RQ1. The results show REMCCG remarkably outperforms Csmith and YARPGen, i.e., REMGEN can find 164%/363% and 120%/595% more bugs than Csmith and YARPGen, in GCC/LLVM, respectively. The results are reasonable as they have different designs and implementations. Csmith and YARPGen are designed to mainly detect miscompliation bugs, another important category of bugs in compilers that require the test program free of undefined behaviors. Since REMCCG can generate complementary test programs (i.e., diverse syntactic valid but may contain undefined behaviors) that Csmith is hard to generate, it is rational that the bug-finding capability of REMCCG is orders of magnitude better than Csmith and YARPGen. It is worth noting that test programs generated by REMCCG are important to disclose critical bugs, which is also confirmed by developers with their positive feedback[5].

Limitation of REMCCG. REMCCG inherits the limitation from CCG [3]. That is, REMCCG can only find two kinds of bug (i.e., *crash* and *performance* bugs) in compilers. However, our proposed framework could apply to other program generators, such as Csmith [2] and YARPGen [8]. Specifically, to remanufacture program generators that have a high demand for test programs, e.g., those test programs should be free of undefined or unspecified behaviors, a checking module should be considered in the synthesized code snippets (i.e., before ④ in Fig. 4). Such a checking module is used for guaranteeing the validity (i.e., free of undefined behavior) of the synthesized code snippet. If a synthesized code snippet fails in the checking, the snippet should be rejected and re-synthesized again. Note that the design of such checking is non-trivial. For example, Csmith adopts complex heuristic algorithms to make sure the test programs are free of undefined behaviors, and such a process is proven to be heavy [8]. Therefore, a more practical solution (e.g., practical static analysis) which makes the checking process more efficient, should be taken into account during remanufacturing. We leave this extension as our near future work.

[5]https://gcc.gnu.org/bugzilla/show_bug.cgi?id=99694#c15

VI. THREATS TO VALIDITY

Internal threats. The major internal threat comes from the old program generator chosen in this study. REMCCG can only detect two types of bugs (i.e., crash or performance bugs) in this study, as we opt for CCG rather than other existing program generators. However, based on the evaluation results in Section IV, REMCCG is not only able to boost two prominent test program construction approaches but also has a promising practical bug-finding capability. We leave the work of remanufacturing other program generators for future work. Another threat lies in the implementation of REMCCG. The choice of the bound value could affect the effectiveness of REMCCG. In this study, we set the bound value by evaluating the influence of different values of bounds as we discussed in Section V. We do not expect this to be a serious threat because we have conducted intensive experiments to evaluate the effectiveness of different bound values. We spend a long time evaluating the effectiveness of REMCCG. For example, only considering evaluating the bug-finding capability of each approach in RQ1, the entire experiment lasted for more than 30 days (running each approach for 90 hours and 10 times). We believe the long testing period can alleviate such a threat.

External threat. The external threat mainly lies in the testing subjects. We used two versions of two compilers as subjects, and these subjects may not be representative enough for different compilers (e.g., *icc* or *msvc* is not tested). To reduce this threat, we selected the two most popular and widely studied C compilers following the existing studies [2], [4], [5], [7], [8], [10], [33]. More specifically, we considered different compilers with both old (in RQ1 and extra experiments in Section V) and development (in RQ2) versions, to evaluate the effectiveness of REMCCG from various testing subjects.

VII. RELATED WORK

In this section, we survey related works on constructing test programs and test program selection for compiler testing.

Test program generation for compiler testing. There is a plethora of work on constructing diverse test programs via generator-based and mutation-based approaches. Quest [34] generates various interesting function call signatures to

test calling conventions. Eide and Regehr [35] design a tool for producing abundant volatile-qualified variables to exercise compilers. CCG [3] is designed for generating diverse syntactically valid test programs. Csmith [2] and YARPGen [8] excel at generating free of undefined behavior programs to detect miscompilation bugs. The Orange family [36]–[38] targets generating diverse test programs to validate arithmetic optimization in compilers. Ldrgen tool sets [39], [40] are designed for searching missed optimization via generating liveness code. Machine learning-based approaches (e.g., DeepFuzz [41] and DeepSmith [42]) are also applied to randomly generate test programs. Another direction is to obtain diverse test programs by modifying existing programs. Orion [6], Athena [5], and Hermes [4] are three state-of-the-art approaches that target mutating existing programs by deleting/inserting code snippets on *dead* or *live* or *both* code regions. Followed by Orion, CLsmith [43] is proposed to validate the OpenCL compiler which uses a similar strategy to only mutate the dead code area. Later, Donaldson and Lascu [44] target generating diverse yet valid expressions to exercise graphic OpenGL compilers. Recent coverage-guided fuzzing approach PolyGlot [45] is proposed to mutate on intermediate representation (LLVM bitcode) code rather than source code.

Different from the existing approaches, we focus on remanufacturing program generators. That means, our proposed approach could boost both the generation-based and the mutation-based approaches for compiler testing. Specifically, in the former, compared to generation-based approaches, we increase the diversity of test programs by leveraging a grammar-aided diverse code snippets synthesis during program generation. In the latter, REMGEN is able to make a tamed program generator (i.e., CCG) generate new code snippets at a low cost compared with mutation-based approaches. Moreover, the generated code snippets can be further used by mutation-based approaches to detect more compiler bugs. To sum up, our approach could be complementary to existing test program generation approaches.

Test program selection for compiler testing. Our code snippet selection is related to the studies on test program selection as a test program consists of various code snippets. Chen et al. [23] propose a text-vector-based approach to prioritize test programs. Further, Chen et al. [11] learn and build a model to produce bug-revealing programs with certain features. Since different programs may have the same test capabilities (e.g., exercise the same region in the compiler), Chen et al [46], [47] learn to predict coverage statically based on test program features without executing.

Unlike the existing work, we focus on the bug-revealing code snippet selection for effectively remanufacturing a tamed program generator. Specifically, the novelty of our selection strategy is two-fold. First, we adopt grammar coverage, rather than existing text-vector-based or coverage-based metrics, to characterize different code snippets; it is effective for distinguishing various code snippets after synthesizing at a low cost. Furthermore, we leverage the introduced coverage metric to select the bug-revealing code snippets to construct new test programs. Augmented by the grammar coverage guided selection, the remanufactured program generator has shown to be effective for compiler testing during our evaluation.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we present a framework named REMGEN to remanufacture a program generator for compiler testing. Two main challenges have been addressed, i.e., the synthesis of diverse code snippets at a low cost and the bug-revealing code snippet selection for constructing new test programs. To demonstrate the effectiveness of REMGEN, we performed a case study on an old C program generator CCG and implemented REMCCG. Evaluation results show that REMCCG not only boosts both the generation-based and the mutation-based approaches in terms of bug-finding capability but also has a promising bug-finding capability in practice. Notably, REMCCG has found 56 new bugs for GCC and LLVM, of which 37 have already been fixed by developers.

For future work, we are actively pursuing to apply the proposed framework to remanufacture other program generators (e.g., Csmith) for detecting more types of compiler bugs.

ACKNOWLEDGMENT

The authors would like to appreciate all developers who participated in this work, especially for GCC and LLVM developers who promptly confirmed and fixed our reported bugs, and the anonymous reviewers for their insightful comments. This work is supported in part by the National Natural Science Foundation of China under grant no. 61902181, 62032004, and CCF-SANGFOR OF 2022003. This article is also partially supported by the National Research Foundation (NRF) Singapore and National Satellite of Excellence in Trustworthy Software Systems (NSoE-TSS) award number NSOE-TSS2019-04.

REFERENCES

[1] S. Bauer, P. Cuoq, and J. Regehr, "Deniable backdoors using compiler bugs," *International Journal of PoC or GTFO, 0x08*, pp. 7–9, 2015.

[2] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011, pp. 283–294.

[3] A. Balestrat. (2006) A random c code generator. [Online]. Available: https://github.com/Mrktn/ccg

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

[5] V. Le, C. Sun, and Z. Su, "Finding deep compiler bugs via guided stochastic program mutation," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015, pp. 386–399.

[6] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 216–226.

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

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

[9] H. Jiang, Z. Zhou, Z. Ren, J. Zhang, and X. Li, "CTOS: Compiler Testing for Optimization Sequences of LLVM," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2339–2358, 2022.

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

[11] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, and B. Xie, "Learning to prioritize test programs for compiler testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering*, 2017, pp. 700–711.

[12] Y. Tang, H. Jiang, Z. Zhou, X. Li, Z. Ren, and W. Kong, "Detecting compiler warning defects via diversity-guided program mutation," *IEEE Transactions on Software Engineering*, no. 01, pp. 1–1, 2021.

[13] H. Tu, H. Jiang, Z. Zhou, Y. Tang, Z. Ren, L. Qiao, and L. Jiang, "Detecting C++ compiler front-end bugs via grammar mutation and differential testing," *IEEE Transactions on Reliability*, pp. 1–15, 2022.

[14] Y. Zhao, Z. Wang, J. Chen, M. Liu, M. Wu, Y. Zhang, and L. Zhang, "History-driven test program synthesis for JVM testing," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1133–1144.

[15] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A survey of compiler testing," *ACM Comput. Surv.*, vol. 53, no. 1, pp. 1–36, 2020.

[16] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013, pp. 197–208.

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

[18] M. Matsumoto and W. Ijomah, *Remanufacturing*, 2013, pp. 389–408.

[19] M. Matsumoto, S. Yang, K. Martinsen, and Y. Kainuma, "Trends and research challenges in remanufacturing," *International Journal of Precision Engineering and Manufacturing-Green Technology*, vol. 3, pp. 129–142, 2016.

[20] R. Steinhilper, "Remanufacturing-the ultimate form of recycling," *Fraunhofer IRB Verlag*, 1998.

[21] U. Commission, "Remanufactured goods: An overview of the U.S. and global industries, markets, and trade," pp. 1–206, 2013.

[22] Loop unrolling. [Online]. Available: https://en.wikipedia.org/wiki/Loop_unrolling

[23] 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 *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2016, pp. 266–277.

[24] Grammars written for antlr v4. [Online]. Available: https://github.com/antlr/grammars-v4

[25] T. Parr, *The Definitive ANTLR 4 Reference*, 2nd ed., 2013.

[26] Protocol buffers: Google's data interchange format. [Online]. Available: https://github.com/protocolbuffers/protobuf

[27] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "An empirical comparison of compiler testing techniques," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 180–190.

[28] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 1–10.

[33] C. Sun, V. Le, and Z. Su, "Finding and analyzing compiler warning defects," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 203–213.

[29] C. Sun, V. Le, Q. Zhang, and Z. Su, "Toward understanding compiler bugs in GCC and LLVM," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 294–305.

[30] Z. Zhou, Z. Ren, G. Gao, and H. Jiang, "An empirical study of optimization bugs in GCC and LLVM," *Journal of Systems and Software (JSS)*, vol. 174, p. 110884, 2021.

[31] GCC trunk version was changed from 11.0 to 12.0 on April 20, 2021. [Online]. Available: https://github.com/gcc-mirror/gcc/commit/0cc79337ad265aabccab63882a810f9dc509a9d0

[32] LLVM trunk version was changed from 13.0 to 14.0 on July 28, 2021. [Online]. Available: https://github.com/llvm/llvm-project/commit/08c766a7318ab37bf1d77e0c683cd3b00e700877

[34] C. Lindig, "Random testing of C calling conventions," in *Proceedings of the Sixth International Symposium on Automated Analysis-Driven Debugging*, 2005, pp. 3–12.

[35] E. Eide and J. Regehr, "Volatiles are miscompiled, and what to do about it," in *Proceedings of the 8th ACM International Conference on Embedded Software*, 2008, pp. 255–264.

[36] 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 (SASIMI 2012)*, 2012, pp. 48–53.

[37] 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 (SASIMI 2013).*, 2013, pp. 88–93.

[38] E. Nagai, H. Awazu, N. Ishiura, and N. Takeda, "Reinforcing random testing of arithmetic optimization of C compilers by scaling up size and number of expressions," *IPSJ Transactions on System LSI Design Methodology*, vol. 7, pp. 91–100, 2014.

[39] G. Barany, "Finding missed compiler optimizations by differential testing," in *Proceedings of the 27th International Conference on Compiler Construction*, 2018, pp. 82–92.

[40] Barany, "Liveness-driven random program generation," in *Logic-Based Program Synthesis and Transformation*, 2018, pp. 112–127.

[41] X. Liu, X. Li, R. Prajapati, and D. Wu, "Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2019, pp. 1044–1051.

[42] C. Cummins, P. Petoumenos, A. Murray, and H. Leather, "Compiler fuzzing through deep learning," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 95–105.

[43] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson, "Many-core compiler fuzzing," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015, pp. 65–76.

[44] A. F. Donaldson and A. Lascu, "Metamorphic testing for (graphics) compilers," in *Proceedings of the 1st International Workshop on Metamorphic Testing*, 2016, pp. 44–47.

[45] Y. Chen, R. Zhong, H. Hu, H. Zhang, Y. Yang, D. Wu, and W. Lee, "One engine to fuzz'em all: Generic language processor testing with semantic validation," in *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*, 2021, pp. 642–658.

[46] J. Chen, "Learning to accelerate compiler testing," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*, 2018, pp. 472–475.

[47] J. Chen, G. Wang, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "Coverage prediction for accelerating compiler testing," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 261–278, 2021.