Detecting C++ Compiler Front-end Bugs via Grammar Mutation and Differential Testing

Haoxin Tu, He Jiang, Zhide Zhou, Yixuan Tang, Zhilei Ren, Lei Qiao, and Lingxiao Jiang

***Abstract*—C++ is a widely used programming language with complex grammars, and the C++ front-end is a critical part of a C++ compiler. Although many techniques have been proposed to test compilers, few studies are devoted to detecting bugs in C++ compiler front-ends. In this study, we take the first step to detect bugs in C++ compiler front-ends. To effectively detect diverse types of bugs in compiler front-ends, two main challenges need to be addressed, namely the acquisition of test programs that are more likely to trigger bugs in compiler front-ends and the bug identification from complicated compiler outputs. In this paper, we propose a novel framework named CCOFT to detect bugs in C++ compiler front-ends. To address the first challenge, CCOFT implements a practical program generator. The generator first transforms C++ grammars into a flexible structured format and then utilizes an Equal-Chance Selection (ECS) strategy to conduct structure-aware grammar mutation to generate diverse C++ programs. Next, CCOFT employs a set of differential testing strategies to identify various kinds of bugs in C++ compiler front-ends by comparing complex outputs emitted by C++ compilers, thus tackling the second challenge. Empirical evaluation results over two mainstream compilers (i.e., GCC and Clang) show that CCOFT greatly improves two state-of-the-art approaches (i.e., Dharma and Grammarinator) by 135% and 111% in terms of the numbers of detected bugs, respectively. By running CCOFT for three months, we have successfully reported 136 bugs for two C++ compilers, of which 78 (57 confirmed, assigned, or fixed) for GCC and 58 (10 confirmed or fixed) for Clang.**

***Index Terms*—Reliability, software testing, compiler testing, automated testing, compiler defect, front-end**

I. INTRODUCTION

SOFTWARE systems developed by manifold programming languages, such as C/C++, Java, Python, R, PHP, and Kotlin, are everywhere, and all of those languages are devoted to building systems that satisfy the desired requirements of developers. Among the different kinds of programming languages, C++ is a widely used and popular one that exits over 40 years since its origin in 1979[1]. It is not only a language defined by a specification but also includes a set of rich toolsets [1]. The authoritative surveys (e.g., from JetBrain [2]) show that the population of C++ users is at least 4.5 million with a steady growth of about 100,000 developers every year. Among the various phases during the compilation of programming languages, passing the front-end (including lexical, syntactic, and semantic analysis) is usually the initial step [3]–[5]. Therefore, compiler front-ends play an important role in compilers. Specifically, due to the complicated C++ grammars and hand-written C++ compiler front-ends in modern compilers [6], [7], C++-related components are one of the buggiest components in GCC and Clang (two widely used and mature C++ compilers) [8], [9]. Typically, the task of compiler front-ends is to report any error in an intelligible fashion and then output the intermediate representation of the input, which will be used in the following middle-end [10], [11]. Moreover, well-performed compiler front-ends can protect software systems from talent attackers using compiler outputs to exploit potential security vulnerabilities [12]–[15]. Thus, to ensure the correctness and reliability of C++ compilers, it is crucial to detect and fix bugs in C++ compiler front-ends.

Although many studies have been conducted on compiler testing in the literature [16]–[24], few studies focus on testing C++ compiler front-ends. In general, an approach for compiler testing first employs some program generators to generate test programs and feeds them to stress-test compilers. Then, the approach compares either the outputs of distinct compilers or the execution results of compiled programs to detect inconsistencies in the outputs and thus potential compiler bugs. Csmith [19] and Yarpgen [25] are two well-known C++ program generators, however, they mainly generate completely semantic valid C++ test programs with limited C++ language features (e.g., no "template" or "class"), which satisfy all grammar and type-checking rules [26]. Such kind of C++ programs is difficult to incur potential front-end bugs in compilers as they are all assumed to be passed soon in compiler front-ends. Besides the above tools, grammar-based approaches, such as Dharma [27] and Grammarinator [28], can also be tuned to generate C++ test programs with more features with the help of C++ grammars. However, both Dharma and Grammarinator are limited in generating diverse test programs that are more likely to trigger bugs in compiler front-ends. Specifically, Dharma is unable to cover all grammar rules and alternatives, while Grammarinator struggles to operate complex AST (abstract syntax tree) when the test programs become complex (more details in Section III-A). The above limitations may significantly obstruct the effectiveness of discovering bugs in C++ compiler front-ends.

Haoxin Tu is with the School of Software, Dalian University of Technology, Dalian, China, and also with the School of Computing and Information Systems, Singapore Management University, Singapore. E-mail: haoxintu@gmail.com

He Jiang, Zhide Zhou, Yixuan Tang, and Zhilei Ren are with the School of Software, Dalian University of Technology, Dalian, China, and Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province. He Jiang is also with DUT Artificial Intelligence, Dalian, China. E-mail: jianghe@dlut.edu.cn, cszide@gmail.com, tangyixuan@mail.dlut.edu.cn, zren@dlut.edu.cn

Lei Qiao is with Beijing Institute of Control Engineering, Beijing, China. Email: fly2moon@aliyun.com

Lingxiao Jiang is with the School of Computing and Information Systems, Singapore Management University, Singapore. E-mail: lxjiang@smu.edu.sg

[1]https://www.stroustrup.com/TechRepublic-interview-Bjarne-Stroustrup.pdf

* This article has been accepted for publication but has not been fully edited.

Due to the complexity of C++ grammars and the hardness of testing hand-written C++ compiler front-ends, two technical challenges need to be addressed to effectively test C++ compiler front-ends. Firstly, the ability to generate diverse test inputs is key to any software testing-related activities [29]–[31]. In the realm of compiler testing, it is significantly important as the test inputs are high-structured test programs. Thus, we need to address the challenge of **the acquisition of test programs that are more likely to trigger bugs in C++ compiler front-ends**. Secondly, compiler outputs are tricky. For example, as GCC and Clang have different mechanisms in the diagnostic system, their compiler outputs could be different when compiling the same program. Furthermore, existing approaches are unable to dispose of the complex compiler outputs, which will make it difficult to identify potential bugs in C++ compiler front-ends. Thus, we need to address the challenge of **the bug identification (i.e., expose buggy behaviors to identify potential bugs) from complicated compiler outputs**.

In this paper, we propose a novel framework named **C++ CO**mpiler **F**ront-end **T**ester (CCOFT) to detect bugs in C++ compiler front-ends. To address the first challenge, CCOFT implements a practical program generator. More specifically, the generator first transforms C++ grammars into a flexible structured format and then conducts structure-aware grammar mutation with a strategy named **E**qual-**C**hance **S**election (ECS), thus generating diverse C++ test programs. To address the second challenge, CCOFT employs a set of differential testing strategies to identify different kinds of bugs in compiler front-ends by comparing inconsistent compiler outputs.

To assess the effectiveness of CCOFT, we conduct an extensive empirical evaluation over two mainstream compilers, namely GCC and Clang. First, we compare CCOFT against two state-of-the-art approaches, i.e., Dharma [27] and Grammarinator [28] for evaluating the bug-finding capability of CCOFT. The results show that CCOFT can detect 40 bugs, while Dharma and Grammarinator only detect 17 and 19 bugs within the same testing period, achieving 135% and 111% improvement, respectively. Second, the results also validate the impact of ECS in CCOFT. By employing the ECS strategy, CCOFT is able to detect 18 more bugs than its variant without ECS, achieving over 82% improvement. Finally, we show the promising practical bug-finding capability of CCOFT. Within three months, we reported 136 bugs in C++ compiler front-ends, of which 78 bugs (57 confirmed, assigned, or fixed) for GCC and 58 bugs (10 confirmed or fixed) for Clang.

In summary, this paper makes the following contributions:

- We propose CCOFT, a testing framework aiming to detect bugs in C++ compiler front-ends.
- We design a grammar mutation-based (equipped with ECS) C++ program generator and leverage a set of differential testing strategies to identify potential bugs.
- We implement CCOFT and empirically evaluate its effectiveness against two state-of-the-art approaches. Moreover, we reported 136 (67 confirmed, assigned, or fixed) bugs for GCC and Clang, which clearly demonstrates the practical bug-finding capability of CCOFT.

```
//s1.cc
1 typedef int T;
2 using typename :: T;
/* GCC-trunk output:
s1.cc:2:18: error: expected nested-name-specifier before 'T' */
```

(a) ***Reject-valid*** (GCC Bug #95597)

```
//s2.cc
1 template <class> ;
/* GCC-trunk output:
s2.cc:1:18: error: expected unqualified-id before ';' token
Clang-trunk output:
//no error */
```

(b) ***Accept-invalid*** (Clang Bug #46231)

```
//s3.cc
1 decltype(auto) foo () {};
/*GCC-trunk output:
s3.cc:1:10: error: expected primary-expression before 'auto'
Clang-trunk output:
s3.cc:1:1: error: deduced return types are a C++14 extension */
```

(c) ***Diagnostic*** (GCC Bug #96103)

```
//s4.cc
1 struct g_class : decltype (auto) ... { };
/* GCC-trunk output:
s4.cc:1:35: internal compiler error: in
      cxx_incomplete_type_diagnostic, at cp/typeck2.c:584 */
```

(d) ***Crash*** (GCC Bug #95672)

```
//s5.cc
1 void a () { .operator b }
/* GCC-trunk compile-time-hog */
```

(e) ***Time-out*** (GCC Bug #96137)

Fig. 1. Five bugs in C++ compiler front-ends

The remainder of this paper is organized as follows. Section II presents illustrative examples and a quantitative study to motivate the study. Section III describes the framework of CCOFT. Empirical evaluation results are presented in Section IV. The discussion, threats, and related work are described in Sections V-VII. Section VIII concludes this paper.

II. MOTIVATION

In this section, we first present five examples to illustrate potential bugs in C++ compiler front-ends. Then, we conduct a quantitative study of historical compiler bugs to show the prevalence and importance of those bugs.

A. Illustrative examples

Fig. 1 presents five bugs in C++ compiler front-ends found by CCOFT. For the sake of clarity, in the rest of this paper, we use the term "*valid*" to refer to a test program that is semantic valid and use the term "*invalid*" to represent a test program that is syntactic or semantic invalid. Based on the description of the task of a compiler front-end [10], we categorize those bugs into five types as follows.

Reject-valid. A C++ compiler front-end may reject a valid program. Fig. 1(a) describes a GCC bug where the C++ compiler front-end of GCC rejects this valid program and emits an unacceptable error message. The *Reject-valid* bug has a high priority and should be considered equally important to

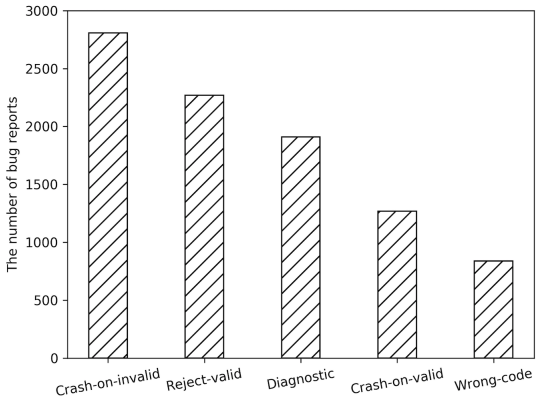


Fig. 2. Top 5 bug types of all the bugs of C++-related components in GCC

fix as those that lead to the wrong-code (the most important bug in compilers) issue[2].

Accept-invalid. In contrast to the *Reject-valid* bug, a C++ compiler front-end may accept an invalid program. Fig. 1(b) shows a Clang bug where the C++ compiler front-end of Clang accepts an empty declaration in a template declaration, which violates the C++ standard and is correctly rejected by the C++ compiler front-end of GCC.

Diagnostic. Diagnostic messages are important to help developers find and fix mistakes in their programs, while a C++ compiler front-end may emit unclear or duplicated error messages and even miss the exact location of an error diagnostic. Fig. 1(c) describes a bug that the C++ compiler front-end of GCC outputs an unclear error message for the error of deduced return type in C++, while the C++ compiler front-end of Clang exactly reports the real reason. If developers compile this program by GCC, it may be hard for them to debug and fix the error based on the confusing compiling outputs, which may delay the schedule of software development.

Crash. Given a valid or an invalid program, a C++ compiler front-end may crash during compilation. *Crash* bugs can be divided into two subtypes, namely *Crash-on-valid* and *Crash-on-invalid*. Fig. 1(d) describes an invalid program which includes an incomplete template pack expansion. This case makes the C++ compiler front-end of GCC crash. This bug exists in almost all versions of GCC before we reported it.

Time-out. A C++ compiler front-end may spend much time analyzing a valid program. Fig. 1(e) shows a *Time-out* bug, which makes the C++ compiler front-end of GCC stuck and conducts an endless analysis.

All the five types of bugs in C++ compiler front-ends may deeply impact the usability of compilers and even cause an obstacle for developers to quickly learn and fix programming errors [32] [33]. Even worse, as aforementioned in Section I, they can also yield the compromise of critical software systems [12]–[15].

B. A quantitative study of historical bugs

To further understand the importance of bugs in C++ compiler front-ends and motivate our study, we conduct a quantitative study of historical compiler bugs in this subsection. The previous study [8] on understanding compiler bugs points out that the components used to implement a C++ compiler in GCC and Clang are more buggy compared to other components. We further investigate the composition of bugs ID with from 1 to 93,000 for the components related to C++. Specifically, we collect bug reports from the GCC bug repository[3]. Here, we only collect GCC bug reports because GCC has long development history and a clear keyword[4] mechanism to show bug types. Among all the collected 86,222 bug reports, 20,441 (23.7%) of them belong to the components related to C++. Next, we categorized bugs of C++-related components according to the keywords in each bug report. Fig. 2 shows the Top 5 types of all bugs of C++-related components in GCC, namely *Crash-on-invalid*, *Reject-valid*, *Diagnostic*, *Crash-on-valid*, and *Wrong-code*, where the *Wrong-code* indicates the compiler produces a wrong compiled program.

According to compiler front-end bug categories mentioned in Section II-A, we were surprised that the Top 4 bug types in Fig. 2 may relate to C++ compiler front-ends and we wanted to know more details about bugs in compiler front-ends. However, to the best of our knowledge, no prior study focuses on analyzing bugs in compiler front-ends. Thereby, to investigate how many bugs in the Top 4 bug types in Fig. 2 are indeed in the compiler front-end rather than other components in the compiler, we conducted a small-scale analysis of the bugs in these Top 4 bug types. Specifically, we randomly selected 100 fixed bugs for each type. Then, we manually checked whether each bug in the selected 400 bugs is a front-end bug. Due to our limited knowledge, we only confirmed bugs that are definitely inside the compiler front-end. Even so, the results showed that there are 69, 63, 52, and 72 bugs that are bugs in the compiler front-end in 100 bugs of each type, respectively. Thus, we can know that at least 64% of bugs on average belong to the C++ compiler front-end of GCC. Therefore, more advanced techniques and tools are needed to help test C++ compiler front-ends and improve their quality.

Summary: Due to the importance of ensuring the reliability of C++ compilers and the prevalence of bugs that are relevant to C++ compiler front-ends, in this study, we design a novel framework named CCOFT. Specifically, as mentioned in Section I, it is non-trivial to generate test programs that are more likely to trigger bugs in C++ compiler front-ends. To address this challenge, we adopt a grammar mutation-based program generation approach to generate diverse C++ test programs, such as the sampled programs presented in Fig. 1. Then, the compiler outputs can be complicated, and directly identifying potential bugs from them is difficult. For example, as shown in Fig. 1, test programs (a) and (b) make GCC and Clang yield inconsistent outputs and on test program (c), two compilers produce different diagnostic messages. We then leverage a set of differential testing-based strategies to address such a challenge.

[2]https://www.gnu.org/software/gcc/bugs/management.html

[3]https://gcc.gnu.org/bugzilla/

[4]https://gcc.gnu.org/bugzilla/describekeywords.cgi

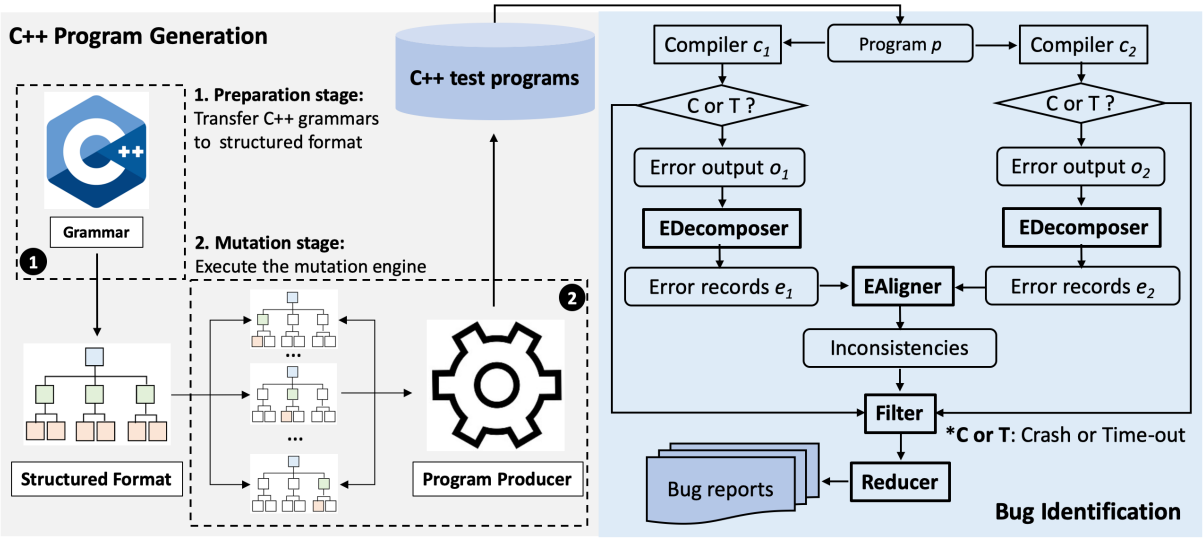


Fig. 3. The CCOFT Framework

III. FRAMEWORK OF CCOFT

In this section, we describe the design of our proposed CCOFT framework. Fig. 3 shows the overall workflow of CCOFT, which includes two parts, namely **C++ program generation** and **bug identification**. The first part aims to generate test programs that are more likely to trigger bugs in C++ compiler front-ends, while the second part conducts differential testing strategies to identify potential bugs.

As aforementioned in Section I, one of the most challenging parts of testing compiler front-ends is the effectiveness of generated test programs. We acknowledge that generating lexical or syntactic invalid test programs are trivial and those programs can also detect some bugs in compiler front-ends to some extent. However, those bugs are too shadowed in compilers. In this study, we target generating test programs that are more likely to pass syntactic analysis and are possible to pass semantic analysis thus detecting various bugs in C++ compiler front-ends. To do so, we opt for mutation-based program generation approach as they are proven to be effective in software testing [21], [31], [34]. Specifically, we design a grammar mutation that mutates grammar directly rather than test programs. The major benefit behind this is that grammatically validness is guaranteed compared with existing mutations on test programs directly. With the assistance of effective C++ test programs, we conduct differential testing strategies to identify bugs thus addressing the second challenge. To our knowledge, this is the first work to leverage the differential testing techniques to detect various kinds of compiler bugs in front-ends.

A. C++ Program Generation

As C++ programs are highly structured, it may be ineffective when using random generation methods or well-known mutation-based fuzzers (i.e., AFL [35] or LibFuzzer [36]) to generate C++ test programs. Specifically, Existing state-of-the-art approaches, such as Dharma [27] and Grammarinator [28], also have major limitations on generating compiler front-end bug-revealing test programs. For example, Dharma has trouble in handling the use of undefined identifiers and can hardly manifest the appearance of production rules. For small-scale grammars such as XML[5] or JSON[6], it is possible to adjust it to generate required test programs, but it is non-trivial to do so when meeting the extremely sophisticated C++ grammars. It's worth noting that the capability of handling undefined identifiers could be key to effectively detecting bugs in compiler front-ends [28]. As the errors of undefined identifiers often happen in the very early stage of a compiler front-end, the following logic, such as syntactic or semantic analysis, will not be reached. This is also the reason why we aim to generate those test programs that are more likely to pass in syntactic analysis but maybe fail in semantic analysis. Secondly, an alternative approach, Grammarinator, provides a configurable distribution of the grammar rules. Unfortunately, it suffers from the following limitations. First, the undefined identifiers problem is not fully solved as it only supports a simple symbol table to maintain trivial variables. However, there are many types of identifiers in complex C++ grammars. Second, it struggles in dealing with deeper recursion as it generates test programs based on the AST construction and operation, which is a time-consuming process (our experience shows that it gets stuck if the recursive depth is large than 50) and thus impeding the generation of complex test programs. We next detail how our proposed approach works and addresses the limitations.

In this paper, we adopt a simple and effective approach to facilitate the process of C++ program generation, which includes two stages, i.e., the preparation stage and the mutation stage. The preparation stage transforms C++ grammars into a structured format file, the mutation stage executes the structure-aware grammar mutation and then produces C++ test programs. To clarify how the C++ program generator works, we take a simplified C++ grammar for template declaration

[5]https://github.com/antlr/grammars-v4/blob/master/xml/XMLcompilerfront-end.g4

[6]https://github.com/antlr/grammars-v4/blob/master/json/JSON.g4

```
Templatedeclaration
    : Template "<" Templateparameter ">" Declaration
    ;
Templateparameter
    : Class Identifier?
    | Typename Identifier?
    ;
```

Fig. 4. A C++ Grammar of a simplified template declaration

(as shown in Fig. 4) as an example to describe the details of the above two stages.

*1) **Preparation stage**:* To successfully generate test programs, we employ a C++ grammar and transform it into a structured format file. Although the C++ grammars are often publicly available (e.g., in ANTLR's community [37]) and the structured formats are usually easy to obtain (e.g., JSON [38], Cpn's Proto [39], and Protobuf [40]), not every of the format can meet our requirement, i.e., it should be flexible enough so that can be combined with existing mutation engines. Therefore, in this paper, the C++ grammars are finally transformed into the Protobuf format [40] (a language-neutral, platform-neutral, extensible mechanism for serializing structured data, but smaller, faster, and simpler). We choose Protobuf for two reasons. First, it has a corresponding field relationship with a normal grammar definition. For example, it can transfer normal text to "*required*", "?" to "*optional*", and "|" to "*oneof*", respectively. Besides the field of "*required*", other fields are alternative options in which we can choose the probability to control whether a field ("*optional*" or "*oneof*") is selected. Second, the Protobuf format can be easily combined with a structure-aware mutator, e.g., *libprotobuf-mutator*. The *libprotobuf-mutator* is able to effectively mutate "Protobuf" inputs, as demonstrated for Compression and PNG [7].

For example, Fig. 4 describes a piece of C++ grammars about template declaration, which includes five elements, i.e., "Template", ,"<","Templateparameter", ">", and "Declaration". According to the corresponding relationship between the grammar and the structured format, we obtain the structured format file in Fig. 5. Each element in Fig. 4 is represented as a standalone "*message*", followed by the body under special fields, such as "*required*", "*optional*", or "*oneof*", numbering sequentially from 1. More specifically, in Fig. 4, the "Template", "<", and ">" are three fixed elements when transforming, while "Templateparameter" and "Declaration" are two variable elements that can be replaced by other elements. To simplify the representation of the grammar of template declaration, we break down the "Templateparameter" into two elements, corresponding to two messages (i.e., TemplateParameter1 and TemplateParameter2) in Fig. 5. For the "Declaration", for the sake of simplicity, we assume it can be replaced by three basic elements, i.e., ";", "*class A {}*;", or "*void foo(){}*".

*2) **Mutation stage**:* In this stage, as shown on the left side in Fig. 3, we first take the structured format file as

[7]https://github.com/google/fuzzing/blob/master/docs/structure-aware-fuzzing.md

```
message TemplateDeclaration {
    required Tempalate template_name = 1;
    required TemplateParameter template_param = 2;
    required Declaration declaration = 3;
}
message TemplateParameter {
    oneof _ {
    TemplateParameter1 template_parameter1 = 1;
    TemplateParameter2 template_parameter2 = 2;
    }
}
message TemplateParameter1 {
    required Class class_name = 1;
    optional Identifier identifier_name = 2;
}
message TemplateParameter2 {
    required Typename typename_name = 1;
    optional Identifier identifier_name = 2;
}
```

Fig. 5. A simplified example of the structured format

input and then mutate it to various mutants. Each mutant will correspond to a test program. Since there are three fields in a structured format file, how to choose suitable alternative fields (i.e., "*oneof*" and "*optional*") is important to ensure the effectiveness of the structure-aware mutation. Here, we apply a strategy, called the **E**qual-**C**hance **S**election (ECS) strategy, which uses the equal probability to select alternative fields in the relationship of "*optional*" and "*oneof*". For "*required*" fields, we choose all of them; For "*optional*" fields, we have a probability of 0.5 to choose; For "*oneof*" fields, we choose them with a probability of 1/n (n is the number of total elements in the "*oneof*" field). Finally, the mutated file is delivered to the program producer where the C++ programs can be generated. The rationale behind this strategy is that we give a relative equal possibility to each element in grammar specification, which will potentially improve the diversity of generated test programs. Note that the probability of the selection can be easily changed to satisfy other requirements, which addresses the main limitation in Dharma [27].

As the program producer, we take the following guidelines to conduct the generation:

- Each element could be converted to one real C++ code snippet, e.g., "Class" is converted to the keyword "class".
- For variable identifiers, we give fixed names in different basic types (e.g., char, int, long) and reuse them in arithmetic expressions. For other types of identifiers, e.g., class name or template name, we maintain different recording lists to catch them and fetch different identifiers in those lists when needed. Since we provide the management of various types of identifiers and guarantee the free of undefined variables while Grammarinator [28] only supports a subset of variable identifiers, our approach could perform better in generating required test programs
- We set an upper bound during the mutation to avoid infinite recursion.

By performing the above guidelines, we can finally generate multiple test programs for our requirements. For example, in Fig. 5, the mutator takes the message "TemplateDeclaration" as input. After employing the ECS strategy, the mutated

file is delivered to the program producer to generate a C++ program. According to the generation guidlines, each part in the "TemplateParameter" grammar can be constructed by the following basic elements: Class — "*class*", Typename — "*typename*", Identifier — "*T*", and Declaration — ";", "*class A {}*;", or "*void foo(){}*".

After the mutation stage, we can produce numerous C++ test programs. For the above example, we can obtain the following unique 12 real C++ code snippets:

```
code 1: template <class> class A {};
code 2: template <class> void foo() { }
code 3: template <class> ;
code 4: template <typename> class A { };
code 5: template <typename> void foo() { }
code 6: template <typename> ;
code 7: template <class T> class A { };
code 8: template <class T> void foo () { }
code 9: template <class T> ;
code 10: template <typename T> class A { };
code 11: template <typename T> void foo() { }
code 12: template <typename T> ;
```

In this way, code snippets 3, 6, and 9 will trigger the "*Accept-invalid*" bug illustrated in Fig. 1(b) in Section II-A.

As a brief summary, through the above two stages, many test programs that are likely to trigger bugs in C++ compiler front-ends are generated by CCOFT. Such test programs follow the grammar rules well, but they could be invalid because of lacking type-checking or including invalid semantics. Note that a compiler can not exactly tell whether a test program is syntactic valid or not, it only successfully compiles a program when it is semantic valid (if no bugs). Therefore, most of our generated test programs can not be successfully compiled. However, after using the strategies in Section III-B, a filtered and reduced small code snippet that follows the grammar rules can be compiled and further used to expose bugs in compilers. Overall, compared with the state-of-the-art approaches, CCOFT is first equipped with a full set of variable records to avoid the undefined identifiers problem. Then, we provide a configurable option to enable the grammar rule controlling with a little effort. Overall, the above capabilities make CCOFT much more effective in generating test programs that are more likely to trigger bugs in compiler front-ends.

B. Bug Identification

Bug identification aims to identify potential bugs through differential testing strategies. Specifically, bug identification includes five components, including (1) differential testing strategies, which are based on two different compilers $c1$ and $c2$ to produce error outputs $o1$ and $o2$ if one of the compilers does not crash or time out, (2) an error decomposer *Decomposer* to decompose $o1$ and $o2$ to error records $e1$ and $e2$, (3) an error aligner *EAligner* to align to find the inconsistent records based on the above two error records, (4) a bug filtering *Filter* to filter potential bugs to real ones, and (5) a bug reducer *Reducer* to reduce the programs to small code snips that can trigger the same symptoms.

Our bug identification assumes that the two compilers $c1$ and $c2$ should emit the same or similar set of compiler output records (i.e., $e1$ and $e2$) for the same input p in the ideal situation. This assumption is critical for the effectiveness of differential testing because any detected inconsistent compiler outputs between two compilers would be considered as a bug in either $c1$ or $c2$ (or both).

Algorithm 1: Decomposing compiler error output

```
Input: text, the textual error output of a compiler
Output: a set of decomposed error records
Function EDecomposer(String text):
    /* lines contain error messages                    */
    error_lines ⟵ ∅
    /* a dictionary record of a error message
       compiler front-end                              */
    dict ⟵ ∅
    foreach line in text do
        if "error" in line then
            error_lines.append(line)
    result ⟵ ∅
    foreach line in error_lines do
        dict = line.split(":")
        result += dict
    return result
```

*1) **Differential testing strategies**:* In this study, we adopt the following strategies to identify potential bugs in C++ compiler front-ends.

Crash or Time-out Detecting (*CTD*). This strategy detects crash or time-out bugs when the compiler $c1$ or $c2$ crashes or times out during the compilation of a program p.

Cross-Version Strategy (*CVS*), and **Cross-Compiler Strategy (*CCS*)**. The above two are widely used differential testing strategies in compiler testing [18] [31] [41]. *CVS* selects different versions of a compiler for differential testing. *CCS* chooses different compilers that have been maintained independently. We do not apply the cross-optimization strategy because we focus on bugs in compiler front-ends.

Cross-Standard Strategy (*CSS*). This strategy compiles a program p by a single compiler under different ISO C++ standards (a program language standard provided by the international organization for standardization). For example, we can compile a program by GCC with the C++11 standard (i.e., -std=c++11) enabled in $c1$ and with the C++14 standard (i.e., -std=c++14) enabled in $c2$. Here, we assume that inconsistent compiler outputs are either caused by the ISO C++ standard upgrade, or C++ compiler front-ends should emit upgrade prompt diagnostic messages. Otherwise, there might be a bug, e.g., the *Diagnostic* bug in Fig. 1(c) is caused by an unclear upgrade indication of compiler outputs.

*2) **EDecomposer**:* The *EDecomposer* is designed to decompose the complex compiler output. It takes the original error messages as input and gives out the records which can be easily applied in *EAligner*.

It is challenging to compute the symmetric differences of $e1$ and $e2$ directly because compiler errors are in natural language and different compiler emits error diagnostic messages in different ways. To resolve this problem, we design a specific *EDecomposer* to decompose error messages for each compiler. Algorithm 1 describes the general workflow

Algorithm 2: Aligning two sets of error records

Input: $e1$ and $e2$, error records decomposed of two compiler front-ends
Output: symmetric difference between $e1$ and $e2$

```
1  Function EAligner(String text):
   |  /* a set of elements to remove from e1      */
2  |  rm1 ⟵ ∅
   |  /* a set of elements to remove from e2      */
3  |  rm2 ⟵ ∅
   |  /* Step 1. Remove equivalent pairs          */
4  |  foreach (a, b) ∈ (e1 × e2) do
5  |  |  if (a, b) is an equivalent pair then
6  |  |  |  rm1 ⟵ rm1 ∪ {a}, rm2 ⟵ rm2 ∪ {b}
   |  /* Step 2. Compute pairs with missing
   |     records                                  */
   |  /* a set of pairs with missing records     */
7  |  missing ⟵ ∅
8  |  foreach a ∈ (e1\rm1) do
9  |  |  missing = missing ∪{(a, ⊥)}
10 |  foreach b ∈ (e2\rm2) do
11 |  |  missing = missing ∪{(⊥, b)}
12 |  return missing
```

Algorithm 3: Filtering crashes and inconsistencies

Input: crashed_source (the source of crashed program), missing (the missing record after aligning)
Output: a set of unquie error records

```
1  Function Filter(File crashed_source, String re_missing):
   |  /* a set of unique crashing records        */
2  |  crash_set ⟵ ∅
   |  /* a set of unique missing records         */
3  |  missing_set ⟵ ∅
   |  /* Step 1. Filter crashes                  */
4  |  foreach s.cc in crashed_source do
5  |  |  if s.cc not in crash_set then
6  |  |  |  crash_set.append(s.cc)
   |  /* Step 2. Filter inconsistencies          */
7  |  foreach miss in missing do
8  |  |  if miss not in missing_set then
9  |  |  |  missing_set.append(miss)
10 |  return [crash_set, missing_set]
```

of decomposing the error outputs of a given compiler. The function *EDecomposer()* obtains a string containing all the error messages (between lines 4 and 6) and then splits it into a list (between lines 7 and 10). Thus, each element, e.g., the line number, the column number, and the error description line, in the text format can be represented as an individual error record. Specifically, the function decomposes the output string into a dictionary-like record by extracting the line number, the column number, and the error description.

For example, the error message under GCC compilation in Fig. 1(b) or cases 3, 6, or 9 from the collection stage in Section III-A2 can be decomposed into one record as follows: {"line" : "1", "column" : "18", "message" : "error: expected unqualified-id before ';' token "}.

3) ***EAligner****:* The objective of *EAligner* is to obtain inconsistent records. In this component, two sets of decomposed error records are taken as inputs. After aligning, the missing records will be the outputs. The missing records may be duplicated, so we filter them in the next step.

By aligning errors in $e1$ and $e2$, we can obtain the inconsistencies among compilers, compiler versions, or ISO C++ standards. The output of the aligner is a list of pairs, of which the first element is either an error in $e1$ or $\perp$ (i.e., nothing) and the other is either an error in $e2$ or $\perp$. The process of the aligner produces the following two categories of pairs (a,b).

- **Equivalence** $a \in e1 \wedge b \in e2$, and both have the same location (i.e., the line number and the column number) and do not consider the description of two error messages. This category does not indicate compiler front-ends bugs and we ignore it.
- **Missing Records** $(a \in e1 \ \wedge b = \perp) \vee (a = \perp \wedge b \in e2)$. This category includes the main body of inconsistencies for users to investigate.

Algorithm 2 presents the workflow of *EAligner*. It first cuts down all equivalent pairs from $e1$ and $e2$ (between lines 5 and 7), then it constructs the inconsistent pairs from the remaining errors in $e1\backslash rm_1$ and $e2\backslash rm_2$ (between lines 9 and 13). From the given example in Section III-B2, as Clang emits nothing under the program, i.e., occurring inconsistent outputs, the pair will save the missing record "$(e1, \perp)$".

4) ***Filter****:* After obtaining the crashes or time-out cases and inconsistent missing records from *EAligner*, we cut out duplicate cases or records in the filter.

Algorithm 3 describes the overall procedure to filter out duplicates. For crashes or time-out programs, we execute each program that makes the compiler crash. If the crash point (i.e., specific place crashed or assertion failed) is not in crash_set, we add it to the set (between lines 4 and 6). For example,"*internal compiler error: in cxx_incomplete_type_diagnostic, at cp/typeck2.c:584*" and "*TextDiagnostic.cpp:1026 Assertion 'StartColNo <= EndColNo "Invalid range!"' failed*" are two different records in crash_set for GCC and Clang. As the number of time-out cases is small (only two cases), we do not filter them.

For inconsistent error diagnostic programs, we first remove the duplicates according to the message part in the $e1$ or $e2$ record (between lines 7 and 9). Specifically, due to the incompatibilities between GCC and Clang, one error in GCC may correspond to two or more errors in Clang and vice versa. We record every inconsistent recording after we manually analyze them, then we use those records to filter the same corresponding error records automatically. With the assistance of such an incremental process, we only need to analyze the new inconsistent records to improve the capability of identifying real bugs. In practice, the number of recordings is less than 100. For example, cases 3, 6, and 9 collected in Section III-A2 will produce three duplicated missing records in *EAligner*. We filter the above three cases into a unique one to make the process of *Reducer* more efficient.

5) ***Reducer****:* As concluded in [8], the bug-revealing test cases are typically small, with 80% having fewer than 45 lines of code. Thus, we adopt it and try to reserve the smaller code snippet. Once a test program triggers a bug in compilers, it is

critical to reduce the program to a smaller size by removing irrelevant programs before submitting the bug into GCC or Clang bug repository. In this process, we reduce test programs into small ones as the reduced programs not only help us to understand the issue and avoid reporting a duplicate but also assist developers in triaging or fixing the bug.

For crashes and time-out programs, we use C-Reduce [42] to reduce them. For the program that triggers inconsistent error diagnostic bugs, we reduce the programs manually since C-Reduce can not deal with these cases well. For example, when we only need to reserve a specific error message for a test program, the reduced test program by C-Reduce always triggers many error messages, which is not helpful to analyze the root causes of a bug for developers. In our study, each test program generated by CCOFT is relatively small, thus this manual reduction does not need a lot of time (one at most ten minutes). In the manual process, we try to keep the unique error message while compiling a code snippet and adjust the code by the author's experience. The goal is to make GCC produce one error message while Clang does not, and vice versa. Thus, one *Reject-valid*, *Accept-invalid*, or *Diagnostic* bug can be detected.

Even though we use the above manual process, our reduction process is effective in practice, as a test program can be finally reduced to a few lines (usually within five lines). We will investigate more about auto-reduction in future work.

IV. Empirical Evaluation

In this section, we evaluate the effectiveness of CCOFT. In particular, we seek to investigate the following research questions (RQs):

- **RQ1:** Can CCOFT find more bugs in compiler front-ends compared with state-of-the-art approaches?
- **RQ2:** Can the proposed ECS strategy help CCOFT detect more bugs in compiler front-ends?
- **RQ3:** How is the bug-finding capability of CCOFT in practice?

RQ1 evaluates the bug-finding capability of CCOFT compared with two state-of-the-art approaches (i.e., Dharma and Grammarinator). In particular, we run CCOFT, Dharma, and Grammarinator in the same testing period and compare them from two aspects, i.e., the number of detected bugs and the number of unique bugs (that can be detected by one approach but can not be detected by other approaches). RQ2 investigates the impact of the proposed ECS strategy on the bug-finding capability of CCOFT. We compare CCOFT with its variant (one with the default selection strategy) to examine how the ECS strategy contributes to CCOFT. RQ3 evaluates the capability of CCOFT for detecting bugs in C++ compiler front-ends in practice. Specifically, we run CCOFT on the newly developed versions of compilers, and evaluate the practical bug-finding capability of CCOFT from three aspects, i.e., the number of detected bugs, the bug type of confirmed bugs, and the bug importance (i.e., severity and priority).

A. Experimental Setup

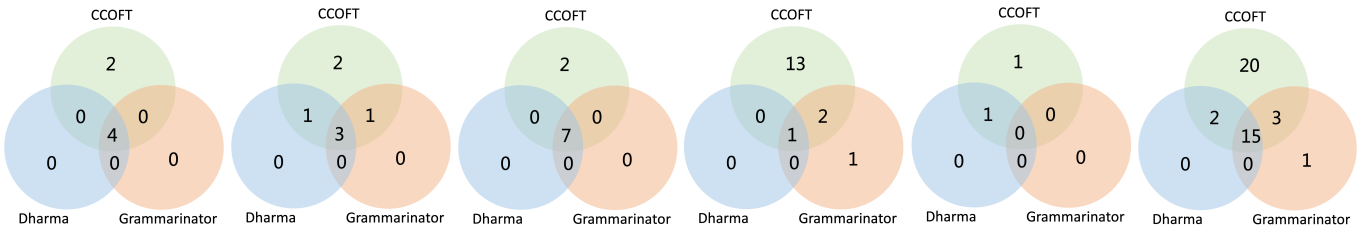
Our evaluation is performed on a Linux PC with Intel(R) CoreTM i7-7700 CPU @3.60GHZ × 8 processor and 16GB RAM running Ubuntu 18.04 operating system. In the study, we use two popular mainstream compilers as subjects, i.e., GCC and Clang, following the existing compiler testing studies [18] [20] [21] [43] [26].

CCOFT implementation. For the implementation of C++ program generator, we take the C++ grammar file in Grammar-v4 [44] as input, which is a collection of various ANTLR [37] grammars. The grammars in Grammar-v4 are publicly available and are contributed by developers around the world. The proposed structure-aware mutation strategy ECS is implemented by Google *libprotobuf-mutator* [45], which is a useful library to randomly mutate structured format (e.g., Protobuf [40]) file and has good scalability in supporting a user-defined mutation strategy. Specifically, we provide a standard structured format file (i.e., protobuf-specification file) that describes the structure of the inputs (i.e., C++ grammars). This structured format file is then compiled into a C++ class *C*. A program input corresponds to an object of class C; the mutator generated via *libprotobuf-mutator* operates on such objects: it modifies a given object into a mutant object. Further, we also provide a producer function that transforms an object of class C into a C++ test program, as aforementioned in Section III-A2. For the implementation of the bug identification, each part, i.e, *EDecomposer*, *EAligner*, *Filter*, and *Reducer*, is written by Python or Shell.

Baseline approaches for RQ1. To illustrate the bug-finding capability of CCOFT, we compare CCOFT with two state-of-the-art approaches, i.e., Dharma [27] and Grammarinator [28]. Dharma is a grammar-based fuzzer provided by Mozilla, which allows a user to define a grammar file and then generate programs under the given grammar. Grammarinator is a random test program generator that creates test programs according to the grammar in Grammar-v4 [44]. We choose Dharma and Grammarinator since (1) they are the most directly related to our study as both of them use a grammar-aided method to generate test programs, and (2) they are relatively state-of-the-art approaches and have been widely used in recent researches (e.g., [46] and [47] both use the above two approaches in their experiments).

CCOFT variant for RQ2. To show the effectiveness of the proposed ECS strategy, we compare CCOFT with CCOFT(¬ECS), a variant of CCOFT without ECS, to examine how the ECS strategy contributes to CCOFT. Here, we use the default mutation strategy provided by *libprotobuf-mutator* to conduct the structure-aware mutation in CCOFT(¬ECS). This default strategy selects the "required" and "oneof" fields with a high probability (99%) while applying the "optional" field in a structured format file with a low probability (1%). This is because the default mutation strategy of *libprotobuf-mutator* is also efficient when users do not customize the mutation strategy for their applications.

Differential testing scenarios. In the bug identification process, we consider four strategies to differentially test C++ compiler front-ends, i.e., *Crash* or *Time-out* Detecting(*CTD*), Cross-Version Strategy(*CVS*), Cross-Compiler Strategy(*CCS*), and Cross-Standard Strategy(*CSS*). For *CVS* and *CCS*, we use two versions of GCC (GCC-10.1 and a developed version



(a) *Reject-valid* (b) *Accept-invalid* (c) *Diagnostic* (d) *Crash* (e) *Time-out* (f) *All*

Fig. 6. The number of unique bugs in GCC and Clang found by CCOFT, Dharma [27], and Grammarinator [28].

of GCC) and Clang (Clang-10.0 and a developed version of Clang). In the *CSS* scenario, we use a few well-known ISO C++ standard versions, i.e., C++11, C++14, and C++17, to detect bugs in C++ compiler front-ends among different C++ standards, on the development (trunk) versions of GCC and Clang at the time of our study are used in our study. For *CTD*, we detect bugs if the above three strategies get stuck, i.e., crash or time out, when compiling a test program. *CCS*, *CVS*, and *CSS* strategies target detecting bugs caused by inconsistent compiler outputs, i.e., *Reject-valid*, *Accept-invalid*, and *Diagnostic* bugs, while *CTD* aims to detect *Crash* or *Time-out* bugs.

B. Answer to RQ1

Motivation. This RQ aims to investigate the bug-finding capability of CCOFT compared with two state-of-the-art approaches, i.e., Dharma and Grammarinator.

Approach. To evaluate RQ1, we run CCOFT, Dharma, and Grammarinator under the same testing period of 10 days (the same as [48]). Both Dharma and Grammarintor can not directly run with C++ grammars defined in the grammar-v4 repository. To set up Drama, we follow the instruction [8] and transfer the C++ grammars into the ".dg" format to generate test programs. The running setup of Grammarinator can be found here [9]. Note that Grammarinator can not handle a deeper depth (e.g., 50) as we aforementioned, therefore we set the recursion depth to 30 for all the three tools.In order to verify whether the detected bugs are real bugs by analyzing them in the bug repositories of GCC and Clang, we opt for two developed compilers that are committed on 2020-05-31 of GCC[10] and Clang[11]. Further, we analyze the bug-finding capability of CCOFT compared with two state-of-the-art approaches from two aspects, i.e., the number of all detected bugs and the number of unique bugs.

Results. Table I shows the number of detected bugs by each approach and Fig. 6 presents the number of detected unique bugs. In Table I, the first column is the type of detected bugs in RQ1, and the following three columns are the number of all detected bugs for Dharma, Grammarinator, and CCOFT. Specifically, "*n(x+y)*" indicates that the corresponding approach totally finds "*n*" bugs, and "*x*" and "*y*" mean the certain number of detected bugs in GCC and Clang, respectively. From Table I, we can see that the total number of detected bugs by CCOFT is 40, which is much larger than those detected by Dharma (i.e., 17) and Grammarinator (i.e., 19), achieving 135% and 111% improvement, respectively.

TABLE I
THE NUMBER OF BUGS (TOTAL BUGS (GCC BUGS + CLANG BUGS)) DETECTED BY CCOFT AND THE COMPARATIVE APPROACHES

Bug Types	Dharma	Grammarinator	CCOFT
Reject-valid	4 (4+0)	4 (4+0)	**6 (6+0)**
Accept-invalid	4 (3+1)	4 (3+1)	**7 (5+2)**
Diagnostic	7 (6+1)	7 (6+1)	**9 (7+2)**
Crash	1 (1+0)	4 (4+0)	**16 (13+3)**
Time-out	1 (1+0)	0 (0+0)	**2 (1+1)**
Total	17	19	**40**

To show the relationships among bugs detected by the three approaches, we draw five Venn diagrams in Fig. 6. In particular, Fig. 6(a) to Fig. 6(e) show the Venn diagrams of detected bugs by each approach categorized by bug types, and Fig. 6(f) is the number of overall bugs detected by each approach. From Fig. 6, we can see that CCOFT always detects the largest number of unique bugs. However, Grammarinator only detects a unique *Crash* bug and Dharma is not able to detect any unique bugs in our testing period. In particular, from Fig. 6(d), the total number of unique bugs detected by CCOFT is 13, which is much larger than those detected by Dharma (i.e., 0) and Grammarinator (i.e., 1). Also, CCOFT can detect 98% (40 out of 41) of bugs during a 10-day testing period.

Conclusion: The results demonstrate that CCOFT has a better bug-finding capability compared with Dharma and Grammarinator, achieving an improvement of 135% and 111% in the number of detected bugs, respectively.

C. Answer to RQ2

Motivation. It is unknown how the proposed ECS strategy affects the effectiveness of CCOFT. This RQ evaluates the impact of the proposed ECS strategy on the bug-finding capability of CCOFT.

Approach. To investigate the impact of the proposed ECS strategy, we conduct an experiment to compare CCOFT and

[8]https://github.com/MozillaSecurity/dharma
[9]https://github.com/renatahodovan/grammarinator/issues/21
[10]GCC commit by 05430b9b6a7c4aeaab595787ac1fbf6f3e0196a0
[11]Clang commit by f4b0ebb89b3086a2bdd8c7dd1f5d142fa09ca728

TABLE II
THE NUMBER OF BUGS (TOTAL BUGS (GCC BUGS + CLANG BUGS)) DETECTED BY CCOFT AND CCOFT(¬ECS)

Bug Types	CCOFT(¬ECS)	CCOFT
Reject-valid	3 (3+0)	**6 (6+0)**
Accept-invalid	6 (4+2)	**7 (5+2)**
Diagnostic	8 (7+1)	**9 (7+2)**
Crash	4 (2+2)	**16 (13+3)**
Time-out	1 (1+0)	**2 (1+1)**
Total	22	**40**

TABLE III
THE NUMBER OF ALL THE REPORTED BUGS FOR GCC AND CLANG

Bug Status	GCC	Clang	Total
Fixed	13	7	20
Confirmed	43	3	46
Assigned	1	0	1
Worksforme	0	3	3
Pending	10	39	49
Duplicate	10	3	13
Invalid	1	3	4
Total	78	58	136

CCOFT(¬ECS), a variant of CCOFT which uses the default selection strategy. Specifically, we run CCOFT and CCOFT(¬ECS) with the same compiler versions during the same testing period as in RQ1. Then, we compare the bug-finding capability of CCOFT and CCOFT(¬ECS) in terms of the number of detected bugs.

Results. Table II shows the comparative results of the number of detected bugs during a 10-day testing period on CCOFT and CCOFT(¬ECS). From Table II, we can observe that CCOFT with ECS strategy can always detect more bugs than the default selection strategy. In particular, CCOFT can detect 16 *Crash* bugs whereas the CCOFT(¬ECS) is only able to detect 4 *Crash* bugs, achieving a 300% improvement in detecting *Crash* bugs. Totally, CCOFT can detect 40 bugs while CCOFT(¬ECS) can only find 22 bugs. In other words, CCOFT achieves an improvement of 82% in terms of the number of detected bugs over CCOFT(¬ECS). The reason is apparent. Without more chances to select different kinds of grammar elements, the generated programs can only consist of the shadow piece of code snippets, which are less likely to trigger bugs in compilers. This is also the reason why our mutation strategy performs well in generating diverse test programs, thus significantly increasing the bug-finding capabilitiy.

Conclusion: The results show that our proposed Equal-Chance Selection strategy is effective to help CCOFT detect more bugs in C++ compiler front-ends. Specifically, under the same testing period, CCOFT can detect 82% more bugs than CCOFT(¬ECS).

D. Answer to RQ3

Motivation. Detecting real bugs in mature compilers is difficult. This RQ assesses the practical bug-finding capability of CCOFT for detecting bugs in C++ compiler front-ends.

TABLE IV
THE NUMBER OF BUG TYPES OF CONFIRMED BUGS

But Types	GCC	Clang	Total
Reject-valid	5	0	5
Accept-invalid	8	2	10
Diagnostic	9	3	12
Crash	34	5	39
Time-out	1	0	1
Total	57	10	67

Approach. To evaluate the bug-finding capability of CCOFT in practice, we choose the daily built trunk version of GCC and Clang on the non-continuous period within three months (from early June to mid-August in 2020) as the developed version. This is because compiler developers always fix bugs in the development trunk more promptly than in stable versions [20] [21] [43]. In detail, we evaluate the practical bug-finding capability of CCOFT from three aspects, i.e., the number of detected bugs, the bug type of confirmed bugs, and the bug importance.

Results. In this subsection, we first describe the quantitative and qualitative results of our reported bugs and then assort some impactful bugs found by CCOFT.

*1) **Quantitative and qualitative results**:* This subsection describes some statistical properties of the discovered bugs, including the number of reported bugs and their quality.

Basic Statistics of Detected Bugs. Table III shows the detail of all the reported bugs so far. In total, we have reported 136 bugs, of which 67 are confirmed/assigned/fixed (the first three rows in the table) by developers. It may take some time before developers consider our reported bugs. During this time, the trunk has changed and these changes may suppress the bug. Developers mark such bug reports as "WorksForMe" in Clang, and we have three Clang bugs of this kind. We do not expose this kind of bug in GCC because GCC developers responded to our bugs much more quickly. Note that for Clang, only 10 out of 58 bugs are confirmed or fixed, which is likely due to limited human resources as active Clang developers went to the Swift project [18]. Besides, we have reported a few duplicate invalid reports, which were rejected by the developers (see more details about the false positive rate below). All the reported bugs can be found in the summarized table here [12].

Table V further lists the details of all confirmed or fixed bugs, including their identities (ID), priorities (Prio.), current status (Status), bug types (Type), identification strategies (Stra.), and affected versions (Affe.Vers.). We do not list the severity status here because only one bug is marked as "minor" and two bugs are marked as "enhancement" in all confirmed bugs. Note that we only list affected versions in our tested compilers. There can be many bugs that affect older versions. For example, the first bug listed in Table V affects all versions from GCC-4.1 to current trunk versions. Those long lurking bugs also confirm the usefulness of our reported bugs.

[12]https://github.com/haoxintu/CCOFT/blob/main/reported-bugs.md

TABLE V
DETAILS OF CONFIRMED/ASSIGNED/FIXED BUGS REPORTED BY CCOFT

	ID	Pri.	Status	Type	Stra.	Affe. Vers.
1	GCC-95597	P3	Conf.	*Rej.-val.*	CCS	10.1-11.0 (trunk)
2	GCC-95610	P3	Conf.	*Rej.-val.*	CCS	10.1-11.0 (trunk)
3	GCC-95641	P3	Conf.	*Diag.*	CCS	10.1-11.0 (trunk)
4	GCC-95657	P3	Conf.	*Diag.*	**CSS**	10.1-11.0 (trunk)
5	GCC-95672	P3	Fixed	*Crash*	CTD	10.1-11.0 (trunk)
6	GCC-95742	P5	Conf.	*Diag.*	CCS	10.1-11.0 (trunk)
7	GCC-95744	P3	Conf.	*Diag.*	CCS	10.1-11.0 (trunk)
8	GCC-95807	P3	Conf.	*Acc.-inv.*	CCS	10.1-11.0 (trunk)
9	GCC-95820	P3	Fixed	*Crash*	CTD	10.1-11.0 (trunk)
10	GCC-95872	P5	Conf.	*Diag.*	CCS	6.1-11.0 (trunk)
11	GCC-95925	P5	Conf.	*Crash*	CTD	10.1-11.0 (trunk)
12	GCC-95927	P5	Conf.	*Crash*	CTD	10.1-11.0 (trunk)
13	GCC-95932	P5	Conf.	*Crash*	CTD	10.1-11.0 (trunk)
14	GCC-95935	P3	Assi.	*Crash*	CTD	10.1-11.0 (trunk)
15	GCC-95937	P3	Conf.	*Crash*	CTD	10.1-11.0 (trunk)
16	GCC-95938	P3	Conf.	*Crash*	CTD	10.1-11.0 (trunk)
17	GCC-95945	P5	Conf.	*Crash*	CTD	10.1-11.0 (trunk)
18	GCC-95954	P5	Conf.	*Crash*	CTD	10.1-11.0 (trunk)
19	GCC-95956	P5	Conf.	*Crash*	CTD	10.1-11.0 (trunk)
20	GCC-95972	P3	Conf.	*Crash*	CTD	10.1-11.0 (trunk)
21	GCC-95999	P3	Conf.	*Crash*	CTD	10.1-11.0 (trunk)
22	GCC-96045	**P1**	Fixed	*Diag.*	**CVS**	11.0 (trunk)
23	GCC-96048	P5	Conf.	*Crash*	CTD	10.1-11.0 (trunk)
24	GCC-96068	P3	Fixed	*Rej.-val.*	CCS	10.1-11.0 (trunk)
25	GCC-96077	P3	Fixed	*Rej.-val.*	CCS	10.1-11.0 (trunk)
26	GCC-96082	P3	Fixed	*Rej.-val.*	CCS	10.1-11.0 (trunk)
27	GCC-96103	P3	Fixed	*Diag.*	CCS	10.1-11.0 (trunk)
28	GCC-96116	P3	Conf.	*Acc.-inv.*	CCS	10.1-11.0 (trunk)
29	GCC-96119	P3	Conf.	*Acc.-inv.*	CCS	10.1-11.0 (trunk)
30	GCC-96137	**P1**	Fixed	*Time-out*	CTD	10.1-11.0 (trunk)
31	GCC-96162	P4	Conf.	*Crash*	CTD	10.1-11.0 (trunk)
32	GCC-96182	P3	Conf.	*Diag.*	**CSS**	10.1-11.0 (trunk)
33	GCC-96183	P3	Conf.	*Acc.-inv.*	**CSS**	10.1-11.0 (trunk)
34	GCC-96184	**P2**	Conf.	*Acc.-inv.*	**CSS**	10.1-11.0 (trunk)
35	GCC-96209	P3	Conf.	*Diag.*	**CSS**	10.1-11.0 (trunk)
36	GCC-96328	P4	Fixed	*Crash*	CTD	10.1-11.0 (trunk)
37	GCC-96329	P4	Conf.	*Crash*	CTD	10.1-11.0 (trunk)
38	GCC-96359	P4	Conf.	*Crash*	CTD	10.1-11.0 (trunk)
39	GCC-96360	P3	Conf.	*Crash*	CTD	10.1-11.0 (trunk)
40	GCC-96364	P3	Conf.	*Crash*	CTD	10.1-11.0 (trunk)
41	GCC-96380	**P2**	Fixed	*Crash*	CTD	10.1-11.0 (trunk)
42	GCC-96437	P4	Conf.	*Crash*	CTD	10.1-11.0 (trunk)
43	GCC-96438	P5	Conf.	*Crash*	CTD	10.1-11.0 (trunk)
44	GCC-96440	P4	Conf.	*Crash*	CTD	10.1-11.0 (trunk)
45	GCC-96441	P3	Conf.	*Crash*	CTD	10.1-11.0 (trunk)
46	GCC-96442	P4	Conf.	*Crash*	CTD	10.1-11.0 (trunk)
47	GCC-96462	**P2**	Fixed	*Crash*	CTD	10.1-11.0 (trunk)
48	GCC-96464	P3	Conf.	*Acc.-inv.*	CCS	10.1-11.0 (trunk)
49	GCC-96465	P4	Conf.	*Crash*	CTD	10.1-11.0 (trunk)
50	GCC-96467	P4	Conf.	*Crash*	CTD	10.1-11.0 (trunk)
51	GCC-96478	P3	Conf.	*Acc.-inv.*	CCS	10.1-11.0 (trunk)
52	GCC-96552	P3	Conf.	*Acc.-inv.*	CCS	10.1-11.0 (trunk)
53	GCC-96553	P3	Conf.	*Crash*	CCS	10.1-11.0 (trunk)
54	GCC-96623	**P1**	Fixed	*Crash*	CTD	10.1-11.0 (trunk)
55	GCC-96636	P3	Conf.	*Crash*	CTD	10.1-11.0 (trunk)
56	GCC-96637	P5	Conf.	*Crash*	CTD	10.1-11.0 (trunk)
57	GCC-96638	P4	Conf.	*Crash*	CTD	10.1-11.0 (trunk)
58	Clang-46231	P3	Fixed	*Acc.-inv.*	CCS	10.0-12.0 (trunk)
59	Clang-46417	P3	Fixed	*Diag.*	CCS	10.0-12.0 (trunk)
60	Clang-46425	P3	Conf.	*Diag.*	CCS	10.0-12.0 (trunk)
61	Clang-46428	P3	Conf.	*Diag.*	CCS	10.0-12.0 (trunk)
62	Clang-46484	P3	Fixed	*Crash*	CTD	10.0-12.0 (trunk)
63	Clang-46487	P3	Fixed	*Crash*	CTD	10.0-12.0 (trunk)
64	Clang-46540	P3	Conf.	*Crash*	CTD	10.0-12.0 (trunk)
65	Clang-46682	P3	Fixed	*Crash*	CTD	10.0-12.0 (trunk)
66	Clang-46729	P3	Fixed	*Acc.-inv.*	CCS	10.0-12.0 (trunk)
67	Clang-46859	P3	Fixed	*Crash*	CTD	10.0-12.0 (trunk)

False Positive Rate. We adopt the following mechanism to calculate the false positive rate (same as [18]):

$$[\frac{rejected}{reported}, \frac{rejected + pending}{reported}]$$

In our evaluation, the range is $[\frac{4}{136}, \frac{4+49}{136}] = [3\%, 39\%]$. Note that 39% is simply an upper bound of the false positive rate, which is mainly due to a relatively large number of pending bugs (especially for Clang). For each potential bug, we have carefully checked its validity before we reported it, so we believe most of the pending bugs will be accepted.

Bug Types. We categorize the bugs reported by our study into five classes as mentioned in Section II-A, namely *Reject-valid, Accept-invalid, Diagnostic, Crash*, and *Time-out*. Table IV shows the number of each type of confirmed bug detected by CCOFT. From Table IV, we can see that the number of crash bugs is larger than those of other types of bugs. There are 39 crash bugs out of the 67 confirmed/assigned/fixed bugs, which indicates that crash bugs currently are the most prominent cause of reducing the quality of C++ compiler front-ends. For the implications of those crashing bugs, please refer to the discussion in Section V for more details.

Bug Importance. In the bug repository of GCC and Clang, the importance of bugs is described as a combination of priority and severity. Priority means the level of priority to fix a bug, and severity measures the impact of bugs, ranging from the most severe, release blocker, to the least severe, enhancement. Both fields are adjusted by developers when they debug bugs. As shown in the online summarized table[12], most of our confirmed bugs have the default priority P3, i.e., 42 (69%) of them are marked as P3 and above. Only one reported bug is labeled as "*Minor*" and two are labeled as "*Enhancement*" by developers, and the rest have the normal severity. Compiler developers are also concerned about bugs in compiler front-ends, 19 of our reported bugs have been fixed in the latest released versions of GCC and Clang. Note that one bug is confirmed as "*ASSIGN*" which means that they are on the way to fixing the bug.

It is worth noting that our reported bugs are important for improving the quality of two mainstream compilers. The evidence is that 6 bugs are marked as "P1" (i.e., GCC#96045, GCC#96137, GCC#96623) or "P2" (i.e., GCC#96184, GCC#96380, and GCC#96462) which are treated as the most severe and urgent level bugs in the GCC community. It is also worth mentioning that these bugs obtained high appraisals, as a GCC developer said, "*This case is useful and it shows that the change in somewhere has a corner case that I didn't consider.*" [13] and a Clang developer also convinced "*These are useful bug reports. Thank you for filing them!*"[14]. Furthermore, we can notice that a vast number of bugs are crashes caused by invalid test programs. Those bugs can have important implications in practice (more details are discussed in Section V). The above positive feedbacks confirm that our reported bugs are indeed important and useful.

*2) **Assorted Confirmed Bug Samples**:* This subsection samples some bugs detected by CCOFT to demonstrate its ability to find a broad range of bugs in C++ compiler front-ends. These bugs have a real impact on developers and some are even marked as the highest severity "P1" or "P2", such as GCC bug #96137, which was discussed in Section II-A.

GCC *Reject-valid* bug #96068. The following program is rejected by the compiler front-end of GCC but accepted by the

[13]https://gcc.gnu.org/bugzilla/show_bug.cgi?id=96137#c2

[14]https://bugs.llvm.org/show_bug.cgi?id=46487#c2

compiler front-end of Clang. The problem in GCC's compiler front-end is that an extra semicolon outside of a function should be allowed after C++11, but GCC rejects this in almost all versions.

```
1 void foo() { };
```

GCC *Reject-valid* bug #95610. In the following code, the compiler front-end of GCC cannot deal with global variables in a class definition. This situation also occurs when replacing the class with other *class* specifiers, i.e., *struct*, *union*.

```
1 class s;
2 class :: s { } ss;
```

GCC *Accept-invalid* bug #96116. The following invalid program is accepted by the compiler front-end of GCC. "*enum struct/class*" can only be used when defining an enumeration or as the part of a standalone forward declaration, but GCC accepts it to be a "using declaration".

```
1 using alias1 = enum struct E1;
2 using alias2 = enum class E2;
```

Clang *Accept-invalid* bug #46729. The following invalid program is another case that the compiler front-end of GCC accepts well. A template, a template explicit specialization, and a class template partial specialization shall not have a C linkage, but Clang treats it as a valid program.

```
1 template <class> void F( ){ }
2 extern "C" {
3  template < > void F<int>();
4 }
```

GCC *Diagnostic* bug #96045. The following program just misses ";" in line 2, but the compiler front-end of GCC leaves out the column number in the error diagnostic message. This bug has been marked as "P1", the most urgent level in the GCC bug repository, and the GCC developer fixed it soon.

```
1 template <class> class A {};
2 struct A <int>
```

Clang *Crash* bug #46682. The following small program that the compiler front-end of Clang compiling an invalid explicit declaration, makes the compiler front-end of Clang crash. It means that Clang could not successfully deal with the error recovery in some cases.

```
1 int b = 0;
2 int foo () { explicit ( && b );}
```

GCC *Crash* bug #95820. In the following code, the compiler front-end of GCC crashes while compiling. According to the developer's experience, although the above program is a *Crash-on-invalid* program, it appears in various reduced test cases from different situations. Thus, it is important to enhance C++ compiler front-ends to output the error messages rather than crashing directly.

```
1 constexpr (*a)()->bool,
```

GCC *Time-out* Bug #96137. The following program makes the C++ compiler front-end of GCC stuck in an endless analysis for the program. *Time-out* bug is important since it may waste developers' time in compiling their programs and is hard to find the root cause in the long compilation time. Notably, such a small test program triggers a corner case that the GCC developer didn't consider [15], and this bug has been marked as "P1" as well. The above fact indicates our strategy for detecting bugs in compiler front-ends is useful.

```
1 void a () { .operator b }
```

Conclusion. The above results clearly demonstrate that CCOFT is effective in detecting bugs in C++ compiler front-ends in practice. In three months, it has reported a total of 136 new bugs in 5 types for GCC and Clang. Among them, 67 bugs have been confirmed/assigned/fixed by developers.

V. DISCUSSION

In this section, we discuss the relationships among the used differential strategies in our evaluation, the implication of crashes caused by invalid code, comparison with existing coverage-guided fuzzing tools, location of reported bugs, and limitations of CCOFT.

The relationships among the used differential strategies. Each strategy has a unique ability in detecting bugs in compiler front-ends. Generally, *CTD*, *CVS*, and *CSS* have lower false-positive rates because it is easy to detect the crash or time-out bugs, as well as the bugs in different versions, different optimization levels, and different standards. *CCS* can hunt more types of bugs in compiler front-ends than others, but it can also report more false positives. This is because of the difference between two different compilers, e.g., GCC and Clang. Although the design of Clang is to be a replacement for GCC, it still has some incompatibilities with GCC. Thus, there would be false positives if the compiler $c1$ and $c2$ support different sets of error messages. For example, Clang allows narrowing conversion of a value from "int" to "bool" by default, while GCC has no such problem. In this paper, the *CTD*, *CVS*, and *CSS* strategies may serve as good complements to *CCS*, because they test compilers from different perspectives and only require a single compiler.

The implication of crashes caused by invalid code. Among all the reported bugs, it is clear to see a large portion of them belongs to crashes, which indicates that the error handling or error recovery capability in compiler front-ends is quite limited. Notably, the flaws of incorrect error handling in compilers (especially for compilers used in web applications [49]) can lead to severe security vulnerabilities and even exploits by attackers [13], such as XML compiler front-end [14]. The apparent behavior of such crashes is that detailed internal error messages such as stack traces, database dumps, and error codes are displayed when the crash is emitted. These messages reveal implementation details that should never be revealed. Furthermore, such details can provide hackers with important clues on potential flaws in the site. In short, messages followed by crashing could lead to information leakage [50], [51]. For example, in CVE-2017-5638[16], a compiler front-end bug caused by invalid test input has been proved to be exploitable.

[15] https://gcc.gnu.org/bugzilla/show_bug.cgi?id=96137#c2
[16] https://nvd.nist.gov/vuln/detail/cve-2017-5638

Although there are no such open exposes caused by crashes in C++ compiler front-ends, by leveraging the detailed stack information emitted from two compilers, e.g., in bug GCC#96359 and Clang#46560, we conjecture that it is possible to induce similar exploits in some elegant manners by expert attackers. We leave it as future work.

Comparison with coverage-guided fuzzing tools. The interested readers may also be concerned about the comparison result between our approach and coverage-guided fuzzing techniques. We now discuss our experience when conducting the comparison. AFL [35] is a traditional coverage-guided fuzzing and has a great impact in both academia and industry. It should be applicable to test compilers to some degree. However, we can not find a single crash or performance issue during a 7-day testing period while running AFL upon GCC compiler. We also run the test programs using our differential testing strategies to identify other types of bugs, however, we can not find any possible bugs. It is no surprise to us because the intrinsic bit/byte level mutation operations make little sense for producing bug-revealing test programs for testing compiler front-ends. We also compared CCOFT with Prog-fuzz [52], which aims to find compiler crashes. We run Prog-fuzz in 10 days. Finally, Prog-fuzz only finds one GCC crash which can also be found by CCOFT.

Location of reported bugs. As stated in Section II-B, we can not exactly tell whether a *Reject-valid, Accept-invalid, Diagnostic, Crash*, or *Time-out* bug is indeed inside the which part of a compiler front-end, i.e., lexical analysis, syntactic analysis (parsing), or semantic analysis. We can only know the precise location of a bug until the bug is fixed. For example, for the bug GCC#96077, the developer fixed it on the source file "parser.c". Furthermore, from the code review conclusion "Fix tentative parsing of enum-specifier", we can know this bug occurred in the parser (syntactic analyzer) of the GCC compiler front-end indeed. Another reason comes from the implementation of GCC and Clang, as different phases in front-ends are always interleaved. For instance, although one crash occurred in semantic analysis, the root cause may come from the former parsing errors. In our study, among 20 fixed bugs, with the assistance of the fixed file name or the description from developers, 19 of them are in the parsers and one in semantic analysis.

Limitations. CCOFT randomly generates C++ test programs and then tests compilers based on differential testing. We did not use any coverage feedback information although not all coverage measurements are equal [53]. Therefore, CCOFT may have trouble in finding deeper semantic bugs in C++ compiler front-ends. However, several automatic coverage-based fuzzing approaches [54] [55] have been proposed, and we plan to integrate such techniques for wider applications. Another limitation in our approach may be the programs generated by CCOFT, which means those programs are not semantically valid thus may be hard to trigger optimization bugs in compilers. Specifically, we can mostly test the front-end in compilers in this study, as the generated programs are most grammatically correct but still could be invalid, for example, some type-checking mechanisms are not satisfied. However, as the results show, these programs indeed are more likely to trigger bugs in C++ compiler front-ends, which is complementary to existing random program generators (e.g., Csmith [19] or Yarpgen [25]).

VI. Threats to validity

In our evaluation, there are two major threats to validity.

The Threat to Internal Validity. The internal threat to validity mainly comes from the implementation of CCOFT. In our study, an efficient implementation of the proposed mutation strategy is key to successfully employing CCOPT to detect C++ bugs in compiler front-ends. Hence, the implementation of the proposed mutation strategy may influence the testing efficiency of CCOFT. To alleviate this threat, we adopt *libprotobuf-mutator* [45], a widely used library developed by Google to randomly mutate protobuffers, to implement the proposed mutation strategy.

The Threat to External Validity. The threat to external validity mainly lies in the reduction of test programs. For the bugs detected through inconsistent compiler outputs, we manually reduced them. This is because C-Reduce cannot work well for these bugs. If we only want to preserve one certain error in the reduced test program, some other errors besides the target error are emitted during the reduction process. Thus, this reduction may depend on the researcher's proficiency in the C++ programming language, such that the reduction process may be time-consuming and the reduced test program may not be minimized. To reduce this threat, the first two authors manually reduce the corresponding programs, and the third author carefully checks the reduced bugs, because all of them have many years of C++ development experience. Another threat comes from the generality of our proposed framework. In general, it could be easily adapted for testing other compilers for other languages. For one thing, all the 100+ language grammars in Grammar-v4 [44] can be tuned to generate other structured test cases as CCOFT does. For another, the bug identification strategy can be tailored for other compiler front-end targets, e.g., Javascript [56] [57]. Javascript is widely used and issues in these systems can cause severe security vulnerabilities.

VII. Related Work

Compiler testing is currently the predominant approach to guarantee the quality of compilers [22] [31]. The existing compiler testing techniques could be divided into three categories, i.e., Random Differential Testing (RDT), Different Optimization Levels (DOL), and Equivalence Modulo Inputs (EMI) [31] [58]. RDT detects compiler bugs by comparing the outputs of different compilers with the same specification, whereas DOL compares the results produced by the same compiler with different optimization levels. Most of techniques [19] [26] [59]–[65] [66] based on RDT and DOL use randomly generated test programs to test a compiler. Csmith [19] and Yarpgen [25] are two widely used C++ program generators to test C++ compilers. However, the test programs generated by Csmith and Yarpgen are completely valid and free of undefined behavior, which makes it hard or impossible to find bugs in C++ compiler front-ends as the front-ends will be

passed quickly. Dharma [27] and Grammarinator [28] are also two widely used test program generators that generate C++ programs by taking the C++ grammar format as inputs. To generate new test programs, some studies focus on mutating the existing programs by employing a set of mutation rules [20] [21] [52] [67] [43] [62], such as Prog-fuzz [52] and Clang-fuzzer [67]. Prog-fuzz can generate a subset of semi-valid C++ test programs, while Clang-fuzzer [67] generates a subset of valid C++ test programs to test Clang API.

Different from RDT and DOL, EMI [21] is derived from metamorphic testing [68], which detects bugs on a single compiler by comparing the outputs of a set of semantically equivalent test programs. The core idea of EMI is that the given equivalent test programs should produce the same results when executing under the given test inputs. Otherwise, there must be a compiler bug [21]. In particular, EMI has three instantiations, i.e., Orion [21], Athena [20], and Hermes [43]. Orion randomly prunes unexecuted statements to generate variant programs [21], while Athena uses the specific operation (e.g., delete or insert) in code regions that are not executed under the inputs [20]. In contrast, Hermes [43] generates variant programs via both live and dead code regions mutation.

Our study is also based on RDT. However, we focus on testing C++ compiler front-ends with test programs generated by a structure-aware grammar mutation strategy. Besides, our program generation approach enables the support of various variable records to avoid the undefined identifiers problem and a configurable option to control the selection of grammar rules with little effort. In addition, we use a new differential testing strategy (i.e., cross-standard strategy) to detect bugs in C++ compiler front-ends based on different ISO C++ standards.

VIII. Conclusion and Future work

In this paper, we present a framework named CCOFT to detect bugs in C++ compiler front-ends. Two challenges have been addressed in CCOFT, namely the acquisition of test programs that are more likely to trigger bugs in C++ compiler front-ends and the bug identification from complex compiler outputs. The empirical evaluation results show that CCOFT can detect 135% and 111% more bugs than two state-of-the-art approaches, i.e., Dharma and Grammarinator, respectively. Within three months, we have reported 136 bugs for two mature C++ compilers, i.e., GCC and Clang, and 67 of them have been confirmed/assigned/fixed by developers.

In the future work, we are actively pursuing to (1) extend the proposed framework to test compiler front-ends for other languages (e.g., Javascript), (2) combine it with coverage feedback in compiler source code to detect deeper semantic bugs, and (3) integrate it with advanced techniques that can help generate semantic valid test programs to disclose more tricky optimization bugs in C++ compilers.

Acknowledgments

The authors would like to thank all developers who participated in this work 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.

References

[1] B. Stroustrup, "Thriving in a crowded and changing world: C++ 2006–2020," *Proc. ACM Program. Lang.*, vol. 4, no. HOPL, 2020. [Online]. Available: https://doi.org/10.1145/3386320

[2] "Infographic: C and c++ facts we learned before going ahead with clion," 2015. [Online]. Available: https://blog.jetbrains.com/clion/2015/07/infographics-cpp-facts-before-clion/

[3] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon, "Semantic fuzzing with zest," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 329–340.

[4] W. Pan, Z. Chen, G. Zhang, Y. Luo, Y. Zhang, and J. Wang, "Grammar-agnostic symbolic execution by token symbolization," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 374–387.

[5] C. Salls, C. Jindal, J. Corina, C. Kruegel, and G. Vigna, "Token-Level fuzzing," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2795–2809. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/salls

[6] "New_c_parser," 2020. [Online]. Available: https://gcc.gnu.org/wiki/New_C_Parser

[7] "A single unified parser for c, objective c, c++, and objective c++," 2020. [Online]. Available: http://clang.llvm.org/features.html#unifiedparser

[8] 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. [Online]. Available: https://doi.org/10.1145/2931037.2931074

[9] Z. Zhou, Z. Ren, G. Gao, and H. Jiang, "An empirical study of optimization bugs in gcc and llvm," *Journal of Systems and Software*, vol. 174, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220302740

[10] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, & tools*. Pearson Education India, 2007.

[11] W. M. Waite and G. Goos, *Compiler construction*. Springer Science & Business Media, 2012.

[12] M. A. Howard, "A process for performing security code reviews," *IEEE Security & privacy*, vol. 4, no. 4, pp. 74–79, 2006.

[13] S. Jana, Y. J. Kang, S. Roth, and B. Ray, "Automatically detecting error handling bugs using error specifications," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 345–362.

[14] C. Späth, C. Mainka, V. Mladenov, and J. Schwenk, "{SoK}:{XML} parser vulnerabilities," in *10th USENIX Workshop on Offensive Technologies (WOOT 16)*, 2016.

[15] "Cve-2017-5638: The apache struts vulnerability explained," 2020. [Online]. Available: https://www.synopsys.com/blogs/software-security/cve-2017-5638-apache-struts-vulnerability-explained/

[16] H. Jiang, Z. Zhou, Z. Ren, J. Zhang, and X. Li, "Ctos: Compiler testing for optimization sequences of llvm," *IEEE Transactions on Software Engineering*, 2021.

[17] 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*, 2021.

[18] C. Sun, V. Le, and Z. Su, "Finding and analyzing compiler warning defects," in *2016 IEEE/ACM 38th International Conference on Software Engineering*, 2016, pp. 203–213.

[19] 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. [Online]. Available: https://doi.org/10.1145/1993498.1993532

[20] 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. [Online]. Available: https://doi.org/10.1145/2814270.2814319

[21] 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. [Online]. Available: https://doi.org/10.1145/2594291.2594334

[22] M. Marcozzi, Q. Tang, A. F. Donaldson, and C. Cadar, "Compiler fuzzing: How much does it matter?" *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, 2019.

[23] 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.

[24] R. Morisset, P. Pawan, and F. Zappa Nardelli, "Compiler testing via a theory of sound optimisations in the c11/c++ 11 memory model," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 187–196, 2013.

[25] V. Livinskii, D. Babokin, and J. Regehr, "Random testing for c and c++ compilers with yarpgen," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–25, 2020.

[26] 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.

[27] "Dharma," 2020. [Online]. Available: https://github.com/MozillaSecurity/dharma

[28] R. Hodován, Á. Kiss, and T. Gyimóthy, "Grammarinator: a grammar-based open source fuzzer," in *Proceedings of the 9th ACM SIGSOFT international workshop on automating TEST case design, selection, and evaluation*, 2018, pp. 45–48.

[29] P. M. Bueno, W. E. Wong, and M. Jino, "Improving random test sets using the diversity oriented test data generation," in *Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering*, 2007, pp. 10–17.

[30] A. M. R. Vincenzi, J. C. Maldonado, M. E. Delamaro, E. S. Spoto, and W. E. Wong, "Component-based software: An overview of testing," *Component-Based Software Quality*, pp. 99–127, 2003.

[31] 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, 2020. [Online]. Available: https://doi.org/10.1145/3363562

[32] B. A. Becker, P. Denny, R. Pettit, D. Bouchard, D. J. Bouvier, B. Harrington, A. Kamil, A. Karkare, C. McDonald, P.-M. Osera, J. L. Pearce, and J. Prather, "Compiler error messages considered unhelpful: The landscape of text-based programming error message research," 2019, pp. 177–210. [Online]. Available: https://doi.org/10.1145/3344429.3372508

[33] D. McCall and M. Kölling, "A new look at novice programmer errors," *ACM Trans. Comput. Educ.*, vol. 19, no. 4, 2019. [Online]. Available: https://doi.org/10.1145/3335814

[34] P. K. Aditya and W. E. Wong, "Comparing the fault detection effectiveness of mutation and data flow testing: An empirical study," 1993.

[35] "Afl," 2020. [Online]. Available: https://lcamtuf.coredump.cx/afl/

[36] "Libfuzzer," 2020. [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[37] "Antlr," 2020. [Online]. Available: https://www.antlr.org/

[38] "Json," 2020. [Online]. Available: https://www.json.org/json-en.html

[39] "Cap'n proto," 2020. [Online]. Available: https://capnproto.org/

[40] "Protocol buffers," 2020. [Online]. Available: https://developers.google.com/protocol-buffers

[41] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of jvm implementations," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 85–99. [Online]. Available: https://doi.org/10.1145/2908080.2908095

[42] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for c compiler bugs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012, pp. 335–346. [Online]. Available: https://doi.org/10.1145/2254064.2254104

[43] 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. [Online]. Available: https://doi.org/10.1145/2983990.2984038

[44] "Grammar-v4," 2020. [Online]. Available: https://github.com/antlr/grammars-v4

[45] "Libprotobuf-mutator," 2020. [Online]. Available: https://github.com/google/libprotobuf-mutator

[46] R. Gopinath and A. Zeller, "Building fast fuzzers," 2019. [Online]. Available: https://arxiv.org/abs/1911.07707

[47] N. Havrikov and A. Zeller, "Systematically covering input structure," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, 2019, pp. 189–199. [Online]. Available: https://doi.org/10.1109/ASE.2019.00027

[48] J. Chen, G. Wang, D. Hao, Y. Xiong, H. Zhang, and L. Zhang, "History-guided configuration diversification for compiler test-program genera-

[49] "Improper error handling," 2020. [Online]. Available: https://owasp.org/www-community/Improper_Error_Handling

tion," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering*, 2019, pp. 305–316.

[50] "Owasp top 10 security risks & vulnerabilities," 2020. [Online]. Available: https://sucuri.net/guides/owasp-top-10-security-vulnerabilities-2021/

[51] "Vulnerabilities by types," 2020. [Online]. Available: https://www.cvedetails.com/vulnerabilities-by-types.php

[52] "Prog-fuzz," 2020. [Online]. Available: https://github.com/vegard/prog-fuzz

[53] Y. Wang, X. Jia, Y. Liu, K. Zeng, T. Bao, D. Wu, and P. Su, "Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization." in *27th Annual Network and Distributed System Security Symposium*, 2020.

[54] D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang, "Fudge: Fuzz driver generation at scale," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 975–985. [Online]. Available: https://doi.org/10.1145/3338906.3340456

[55] K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "FuzzGen: Automatic fuzzer generation," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2271–2287. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/ispoglou

[56] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering*, 2019, pp. 724–735.

[57] B. Mathis, R. Gopinath, M. Mera, A. Kampmann, M. Höschele, and A. Zeller, "Parser-directed fuzzing," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 548–560. [Online]. Available: https://doi.org/10.1145/3314221.3314651

[58] 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.

[59] C. Lindig, "Random testing of c calling conventions," in *Proceedings of the Sixth International Symposium on Automated Analysis-Driven Debugging*, 2005, pp. 3–12. [Online]. Available: https://doi.org/10.1145/1085130.1085132

[60] M. A. Alipour, A. Groce, R. Gopinath, and A. Christi, "Generating focused random tests using directed swarm testing," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 70–81. [Online]. Available: https://doi.org/10.1145/2931037.2931056

[61] 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. [Online]. Available: https://doi.org/10.1145/2737924.2737986

[62] A. F. Donaldson, H. Evrard, A. Lascu, and P. Thomson, "Automated testing of graphics shader compilers," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, 2017. [Online]. Available: https://doi.org/10.1145/3133917

[63] 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. [Online]. Available: https://doi.org/10.1145/2771783.2771785

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

[65] G. Ofenbeck, T. Rompf, and M. Püschel, "Randir: Differential testing for embedded compilers," in *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala*, 2016, pp. 21–30. [Online]. Available: https://doi.org/10.1145/2998392.2998397

[66] Y. Yang, Y. Zhou, H. Sun, Z. Su, Z. Zuo, L. Xu, and B. Xu, "Hunting for bugs in code coverage tools via randomized differential testing," in *2019 IEEE/ACM 41st International Conference on Software Engineering*, 2019, pp. 488–499.

[67] "Clang-fuzzer," 2020. [Online]. Available: https://github.com/llvm/llvm-project/tree/master/clang/tools/clang-fuzzer

[68] T. Y. Chen, "Metamorphic testing: A simple method for alleviating the test oracle problem," in *Proceedings of the 10th International Workshop on Automation of Software Test*, 2015, pp. 53–54.