A Comprehensive Study of OOP-Related Bugs in C++ Compilers

Bo Wang, Chong Chen, Junjie Chen, Bowen Xu, Chen Ye, Youfang Lin, Guoliang Dong and Jun Sun.

***Abstract*—Modern C++, a programming language characterized by its extensive use of object-oriented programming (OOP) features, is popular for system programming. However, C++ compilers often struggle to correctly handle these sophisticated OOP features, resulting in numerous high-profile compiler bugs that can lead to crashes or miscompilation. Despite the significance of OOP-related bugs, OOP features are largely overlooked by existing compiler fuzzers, hindering their ability to discover such bugs. To assist both compiler fuzzer designers and compiler developers, we conduct a comprehensive study of the compiler bugs caused by incorrectly handling C++ OOP-related features. First, we systematically extract 788 OOP-related C++ compiler bugs from GCC and LLVM. Second, derived from the core concepts of OOP and C++, we manually identified a two-level taxonomy of the OOP-related features leading to compiler bugs, which consists of 6 primary categories (e.g., *Abstraction & Encapsulation*, *Inheritance*, and *Runtime Polymorphism*), along with 17 secondary categories (e.g., *Initialization and Destruction* and *Multiple Inheritance*). Third, we systematically analyze the root cause, symptom, options, and C++ standard versions of these bugs. Based on the analysis, we report 10 findings. Inspired by these findings, we developed a proof-of-concept compiler fuzzer OOPFuzz, specifically targeting OOP-related bugs in C++ compilers, and applied it against the newest release versions of GCC and LLVM. In about 3 hours, it detected 9 bugs, of which 3 have been confirmed by the developers, including a bug of LLVM that had persisted for 13 years. The results indicate our taxonomy and analysis provide valuable insights for future research in compiler testing.**

***Index Terms*—C++ OOP features, compiler testing, taxonomy, empirical study**

I. Introduction

COMPILERS are fundamental system software that transforms a program written in a high-level language (e.g., C/C++) into lower-level machine code or assembly language. Modern compilers are inherently large and complex and need to support intricate high-level language features and perform sophisticated optimizations. Similar to other complex software, compilers inevitably contain bugs, which may result in crashes or subtle miscompilation. Compiler crashes lead to failure in generating target code, and miscompilation may lead to bugs that are extremely difficult for application developers to handle.

Substantial research efforts have been devoted to identifying and mitigating compiler bugs, such as (automated) testing [1], [2] and verification [3]. Testing works by designing test case generators (i.e., compiler fuzzers) to generate diverse source code to trigger different compiler components. Verification aims to establish certain equivalence relations between the source and the compiled program. Both of these families require a better understanding of existing bugs of the compilers for the target language. Existing methods have successfully revealed thousands of bugs in a range of compilers [1], [4]–[6]. Yet, an important class of compilers is largely overlooked by existing approaches.

C++, initially known as "C with Classes", emerged in the early 1980s when Bell Labs enhanced the C language with object-oriented programming (OOP) support. It is still one of the most popular programming languages in system programming. To support OOP, C++ introduces many language features allowing for flexible and diverse programming styles. Moreover, C++ keeps evolving, i.e., numerous new concepts, operators, and syntactic sugars, have been introduced over the years, many of which have sophisticated semantics. It is a significant challenge to correctly implement and effectively test C++ compilers.

In this paper, we conduct a systematic study of OOP-related bugs in C++ compilers. Our motivations are threefold. First, OOP-related bugs are prominent in C++ compilers. As shown in Section III, for C++ compilers such as GCC and LLVM, 18% of the bugs are related to OOP features. Second, these bugs are of significant importance, i.e., 40% of them are labeled with the top two priorities, and more than 90% of them result in a compiler crash or miscompilation. Third, existing C++ compiler fuzzers rarely generate complex code that includes OOP features. For example, CSmith [4], the most influential C compiler fuzzer, supports none of the OOP features in C++. Another high-impact C++ fuzzer, YARPGen [5], [7], does not generate classes and their dynamic memory allocation. Fuzz4All [8] and MetaMut [9], the recently developed LLM-based fuzzers, support limited OOP features and hardly generate correct non-trivial programs.

Although several empirical studies on compiler bugs have been conducted, OOP-related bugs have been largely overlooked so far. For example, Sun *et al.* analyzed the distribution of bugs in GCC and LLVM, pointing out that the C++ components have the most bugs in both compilers, and yet

Manuscript received April 19, 2005; revised August 26, 2015.

This work was supported by the National Natural Science Foundation of China under Grant No. 62202040.

Bo Wang, Chong Chen, Chen Ye, and Youfang Lin are with the School of Computer Science and Technology, Beijing Jiaotong University, Beijing 100044, China. E-mail: wangbo_cs@bjtu.edu.cn, 22120350@bjtu.edu.cn, 23125279@bjtu.edu.cn, yflin@bjtu.edu.cn.

Junjie Chen is with the College of Intelligence and Computing, Tianjin University, Tianjin 300350, China. E-mail: junjiechen@tju.edu.cn.

Bowen Xu is with the Department of Computer Science, North Carolina State University, NC 27695, USA. E-mail: bxu22@ncsu.edu.

Guoliang Dong and Jun Sun are with the School of Computing and Information Systems, Singapore Management University, Singapore 178902, Singapore. E-mail: gldong@smu.edu.sg, junsun@smu.edu.sg.

they did not further analyze the bug types, symptoms, and root causes [10]. Zhou *et al.* studied the optimization bugs of GCC and LLVM [11] but did not analyze whether OOP-related features are relevant.

To address this gap, we systematically analyze OOP-related bugs in two major C++ compilers, GCC and LLVM, offering insights into the challenges faced by compiler developers and testers dealing with complex OOP features. In total, we identified 788 bugs and then systematically analyzed them from the following aspects: (1) the *C++ OOP features* leading to compiler bugs, (2) the *symptoms* of these bugs, (3) the characteristics of the *patches*, and (4) the characteristics of the relevant compiler *options*. In general, our study aims to address the following research questions which we believe are essential for developing effective fuzzing techniques for revealing OOP-related bugs.

- **RQ1: What C++ OOP features contribute to compiler bugs?**
- **RQ2: What are the symptoms of these bugs?**
- **RQ3: What are the root causes of these bugs?**
- **RQ4: What is the relationship between the bug types and the symptoms?**
- **RQ5: What compiler options are relevant for triggering OOP-related bugs?**
- **RQ6: Do the bugs from different C++ compilers share certain commonality?**
- **RQ7: How does the evolution of C++ standards affect OOP-related bugs?**

We develop a taxonomy of C++ OOP features that are related to the compiler bugs by manually analyzing the contents of all 788 bugs. Our hierarchical taxonomy includes 6 primary categories with 17 secondary categories. The primary categories are *1) Abstraction & Encapsulation*, *2) Object*, *3) Inheritance*, *4) Runtime Polymorphism*, *5) Compile-time Polymorphism*, and *6) Others*. Based on a thorough analysis that aims to answer the above-mentioned questions, we obtain 10 major findings that we believe are helpful for designing testing methods for revealing OOP-related compiler bugs. Based on the empirical study results and findings, we further propose practical guidelines for designing future C++ compiler fuzzers, to cover more features of modern C++ language and more components of compiler implementations. Furthermore, based on the guidelines we designed and implemented a simple yet effective proof-of-concept C++ compiler fuzzer, called **OOPFuzz**, enhancing generating test cases that cover OOP-related components of C++ compilers. By applying OOPFuzz to the latest releases of GCC and LLVM, we discovered 9 bugs within just about 3 hours. Among these, 3 were confirmed by the compiler developers, and 2 had already been fixed in the trunk branch. This highlights the effectiveness of our approach for testing and debugging C++ compiler bugs. The results demonstrate the usefulness of our findings.

The experimental data and results of our study are publicly available[1], for the sake of reproducibility and open science.

To sum up, we make the following major contributions.

[1]https://github.com/Rush10233/OOP-Related-Bugs-Study-TSE-Artifacts

- We conduct a systematic study on OOP-related bugs in C++ compilers based on 788 bugs from the mainstream compilers, GCC and LLVM.
- For OOP-related bugs, we create a taxonomy for the bug contributing OOP features, and we identify the symptoms, root causes, options, and involved C++ standards.
- We obtain 10 findings and provide guidelines for developing further fuzzers and debugging OOP-related bugs.
- We develop a proof-of-concept fuzzer based on our findings, which successfully finds 9 bugs from the newest releases of GCC and LLVM.

II. Background and Motivation

OOP is a foundational program paradigm of modern programming languages, offering a schema for creating code that is highly reusable, maintainable, and extensible. Although modern C++ supports multiple programming paradigms, OOP has always been its heart and soul. In a C++ compiler, a substantial portion is dedicated to the implementation of OOP-related features. In this section, we first highlight some of the sophisticated OOP features of C++, and then present one OOP-related bug that highlights the challenge and importance of discovering such OOP-related bugs.

A. Key OOP Features

C++ provides various language features for realizing key OOP concepts such as *abstraction*, *encapsulation*, *object*, *inheritance*, and *polymorphism*, and additional OOP-related features for specifying the life cycles of objects, calling constructors and destructors, and so on. In addition, these OOP-related features can be flexibly combined with other (existing or newly introduced) language features, e.g., the newly introduced `consteval` is frequently used to optimize code generation for member functions.

1) Abstraction and Encapsulation: Both of them are fundamental OOP concepts that are difficult to separate. C++ provides numerous features to support these principles, including data abstraction, visibility control, and the binding of methods and data. For example, to encapsulate data, C++ provides not only several kinds of *class-like* data structures (such as `class`, `struct`, `enum`, and `union`), but also a general visibility control `namespace`. Each member of a C++ class has a legal scope, which could be shaped by composition and relationship (i.e., using internal members of other classes) or inheritance relationship (i.e., using members of parent classes). Inside a class, C++ provides access modifiers (e.g., `private` and `public`) to control the members that can be accessed. Even in lambda expressions, which are syntactic sugar of anonymous functions, users can encapsulate operations and data.

2) Object: The definition of a *class* specifies the structure of all *objects* instantiated from it. Conceptually, an object's data members and member functions are stored within a memory layout determined by the structure of the *class*. In C++, the creation, deletion, clone, and ownership transfer of an object are governed by constructors (i.e., *ctor*), destructors (i.e., *dtor*), copy/move constructors, and copy/move

```
/* ---- The bug-triggering program ---- */
  struct base {
    template <typename t> requires false base(t);
    template <typename t> requires true base(t);
  };
  struct derived : base { using base::base; };
  int main() { derived{'G'}; }
/* ---- The corresponding patch ---- */
-   if (flag_new_inheriting_ctors)
+   if (!DECL_TEMPLATE_INFO (t))
```

Listing 1: An example OOP-related bug: GCC-94549

assignment operators, respectively. Additionally, C++ supports operator and function overloading to simplify operations on objects and enhance code expressiveness. Overloading allows C++ objects to associate multiple behaviors with a single operator or function, making interactions with objects more flexible and sometimes more challenging.

3) Inheritance: C++ has extensive support for single and multiple inheritance. To handle the diamond problem introduced by multiple inheritance, it additionally offers virtual inheritance. In addition, C++ provides access control between a derived class and its parent classes. A key issue introduced by inheritance is to ensure the order of calling the constructors and deconstructors of the classes under an inheritance relationship. Similarly, assignments, conversion, and typecasting between a derived class and its base classes are also covered by the language specification, resulting in many implicit rules that compiler developers often overlook.

4) Polymorphism: C++ supports both *runtime polymorphism* and *compile-time polymorphism*, for writing highly reusable code that can interact with objects of different types. Runtime polymorphism occurs when the function to be executed is determined at runtime, achieved through inheritance and virtual functions, and often facilitated by Runtime Type Information (RTTI) to ensure type safety during dynamic type casting. Compile-time polymorphism, in contrast, occurs when the method to be executed is determined at compile time, achieved through templates and overloading. C++ templates are intrinsically powerful and complex, even for experienced C++ veterans. Combining compile-time and runtime polymorphisms leads to additional complexity and numerous corner cases.

B. An OOP-Related Bug

Figure 1 shows an OOP-related bug with the highest priority, GCC-94549[2]. This bug-triggering program shows the challenge of triggering and debugging OOP-related bugs in C++ compilers. The program involves key concepts of OOP (e.g., class, inheritance, template, and object initialization) and advanced features covering multiple C++ standards. For example, the concepts (i.e., the keyword `requires`) is introduced in C++17, the `using` style of constructor inheritance in `derived` is introduced in C++11, the initialization in the `main` function involves template argument deduction is introduced in C++17. Moreover, the option `-std=c++17` is required to trigger this bug. Based on our taxonomy, the OOP feature of this bug-triggering program is *[Inheritance]-[Inherited Ctor & Dtor]*. Referring to the corresponding patch, we find the root cause of this bug is the incorrect branch condition.

[2]https://gcc.gnu.org/bugzilla/show_bug.cgi?id=94549

The complexity of this bug (and similar issues) poses challenges even for experienced developers. This program is valid but was incorrectly rejected by GCC 10.0. To the best of our knowledge, no existing compiler fuzzers could generate such complex code. More importantly, we do not even know what is missing from existing compiler fuzzers to discover such bugs. Therefore, an in-depth understanding of the characteristics of OOP-related bugs is necessary, which is one core contribution of our study.

TABLE I: Bug Selected in This Study

Compiler	Start	End	Reports	Bugs	P1/S1	P2/S2
GCC	2017	2023	3401	586	91	171
LLVM	2014	2023	1005	202	9	40
Total	-	-	4406	788	100	211

III. Methodology

In this section, we introduce how our study is designed and conducted.

A. Bug Collection

Several popular C++ compilers exist, including LLVM, Microsoft Visual C++, and Intel C++ Compiler. Following the existing studies [10]–[12], we select two mainstream compilers, i.e., GCC and LLVM, because they are open-source and have well-maintained bug-tracking systems.

We focus on the compiler bugs that occur when handling OOP-related concepts. Since only the C++ components process OOP-related features, we first collect the Bugzilla bug reports and GitHub issues from these components, and then filter them using the OOP keywords `class` and `struct`, which are the core data structures of C++ OOP programs. Finally, we manually determine whether they are indeed OOP-related.

To analyze the unique characteristics of the OOP-related bugs, we aim to analyze both the bug reports and the corresponding patches. Therefore, we retain only those closed bug reports marked as `FIXED`, along with developers' commit messages that provide the URL of the version control system linking to the corresponding patches. Note that in both bug tracking systems, duplicated bug reports are marked as `DUPLICATE` which are not collected.

Table I summarizes the bugs we collected and analyzed in our study. Since it is unaffordable for us to manually analyze all historical bugs, we selected a range of bugs from the most recent years. Recent bugs cover more features of the newer versions of C++, making them more relevant. Initially, we collected bugs between 2017 and 2023, covering the newest C++ standard C++20. We collected 3401 and 1005 closed bug reports from GCC and LLVM, respectively. Then we filter them by the OOP-related keyword, and afterwards we manually check whether the bug is OOP-related. At last, 586 remained for GCC, while LLVM had only 93. This aligns with existing studies showing that the number of bug reports in the LLVM community is significantly lower than in GCC [10]. To compare bug characteristics across different compilers and mitigate data imbalance, we additionally included bug reports from 2014, 2015, and 2016 for LLVM. For LLVM, we finally collected 566 closed bug reports, of which 218 were kept after filtering. Among these bugs, 91 and 171 of GCC are categorized as the top 2 priority levels (P1 and P2), respectively, while 9 and 40 bugs of LLVM are classified as the top 2 severity levels (S1 and S2), respectively. Therefore, 39.5% of bugs are labeled as the top 2 important levels, confirming the importance of OOP-related bugs.

B. Constructing the Taxonomy of OOP Features

To construct the taxonomy of the OOP features, we follow the recommended procedure adopted in [13]. We refer to the concepts and terms from the popular C++ textbook of *Bjarne Stroustrup* [13], the father of C++, and build the taxonomy by a pilot analysis by the first two authors. In this pilot analysis, we randomly selected 100 bugs and manually analyzed them, extracted five key concepts of OOP as the primary categories, including *Abstraction & Encapsulation*, *Object*, *Inheritance*, *Inheritance*, *Runtime Polymorphic*, and *Compile Time Polymorphic*. For each key concept, we further partitioned them into several secondary categories, as shown later in Section IV. In total, we built 16 secondary categories for the five prime categories. Additionally, the remaining rare cases are categorized as *Others*.

Based on this pilot experiment, the first two authors discussed and summarized the typical characteristics of each category and identified key distinctions from similar categories. They then drafted classification guidelines, which were reviewed and discussed with all co-authors. Our taxonomy is orthogonal, i.e., a bug can only be classified into one category. All the authors agreed on the final design of the taxonomy.

For the remaining 688 bugs, the first two authors each independently performed label classification of the remaining bugs. Following existing studies [14]–[20], we measure the inter-rater reliability via the Cohen's kappa coefficient (κ) [21], which is calculated by the following formulae:

$$\kappa = \frac{p_o - p_e}{1 - p_e} \quad p_o = \frac{\#\text{agreements}}{N} \quad p_e = \frac{1}{N^2}\sum_k n_{k1} n_{k2}$$

p_o is the observed agreement, p_e is the expected agreement, N is the total number of ratings, and n is the number of samples assigned category k by each rater 1 and 2. A larger κ value indicates higher agreement, with $\kappa > 0.75$ representing a strong agreement and $\kappa = 1$ representing perfect agreement.

The confusion matrices and kappa values for the primary and secondary categories are shown in the columns κ_p and κ_s of Table II. From the κ_p column of Table II, we can find that for the primary categories, the lowest κ is 0.888, and the remaining are all larger than 0.900. From the κ_s column of Table II, we can find that the lowest κ of the secondary categories is 0.874. The results indicate a high agreement between the two authors. After labeling, the two authors reviewed and analyzed the conflicted bugs, and finally assigned a unique label to each bug. All the labeling processes are recorded in our repo.

C. Classifying and Labeling Symptoms, Root Causes, Compiler Options, and C++ Standards

These four aspects are consistent with other compiler studies, we thus briefly outline the methods and process for classification and labeling.

1) Symptoms: We directly employ the symptoms given by compiler developers, which are also adopted by existing studies [10], [15], [18]. Bug symptoms can be directly extracted from the issue for labeling, as they are provided by reporters and developers.

2) Root Causes: We adopt the root cause taxonomies of the following studies [15], [16], [18], [22]. Note that certain domain-specific categories are irrelevant to our study, therefore we focus exclusively on the common ones. Because the root causes are consistent with other types of bugs, after determining the classification, we adopt the approach of the existing study [14]. First, the first two authors sampled 259 bugs from all 778 bugs, ensuring a 95% confidence level with a 5% confidence interval, and independently labeled them. The Cohen's kappa coefficient for the root causes labeling is 0.922, indicating a strong agreement. Second, the two authors reviewed and analyzed the conflicts to achieve a deeper understanding. Finally, the second author labeled the remaining bugs based on the established understanding.

3) Compiler Options: Since each bug report includes a reproduction command, we manually extracted the compilation commands from the report to obtain the options.

4) C++ Standards: For the programs that trigger the compiler bugs, we use a parser to analyze the code. When a language feature specific to a particular C++ standard is detected, it is labeled accordingly. Note that this is a multi-labeling process, as a program may include features from multiple C++ standards.

IV. C++ OOP Features Taxonomy

In this section, we present the taxonomy identification results of the C++ OOP-related features leading to compiler bugs, and their distribution.

A. RQ1: Feature Taxonomy Identification Results

We develop a hierarchical taxonomy consisting of 6 primary categories and 17 secondary categories. The primary categories are: *1) Abstraction & Encapsulation*, *2) Object*, *3) Inheritance*, *4) Runtime Polymorphism*, *5) Compile-time Polymorphism*, and *6) Others*. In this subsection, we introduce all the categories and provide examples for each secondary category.

1) Abstraction & Encapsulation: This category of bugs is triggered by data abstraction and encapsulation features of C++. C++ defines access control for derived classes or other classes, and C++ allows the `friend` keyword to access private or protect members of other classes. Moreover, the

TABLE II: Taxonomy of OOP Features

Primary Category	Secondary Category	Rater1 & Rater2				κ_s	κ_p	Distribution			
		NN	NY	YN	YY			GCC	LLVM	Sum	All
Abstraction & Encapsulation	Member Access	654	5	2	33	0.899	0.906	35	13	48	128 (16.2%)
	Class Scope	631	6	5	52	0.896		53	12	**65**	
	Friends	680	2	1	11	0.878		9	6	15	
Object	Ctor & Dtor	534	3	10	147	0.946	0.944	136	38	**174**	**263 (33.4%)**
	Copy & Move	659	1	2	32	0.953		28	14	42	
	Overloading	652	4	1	37	0.933		34	13	47	
Inheritance	Inherited Member Access	670	0	3	21	0.931	0.925	21	9	**30**	79 (10.0%)
	Multiple Inheritance	675	0	4	15	0.879		11	8	19	
	Inherited Ctor & Dtor	665	3	1	25	0.923		25	5	**30**	
Runtime Polymorphism	Override Control	685	2	0	7	0.874	0.888	8	3	11	31 (3.9%)
	Casting	672	3	1	18	0.897		13	7	**20**	
Compile-time Polymorphism	Template Alias	656	1	2	35	0.957	0.927	34	3	37	261 (33.1%)
	Template Parsing	660	1	4	29	0.917		19	15	34	
	Template Instantiation	589	12	6	87	0.891		77	35	**112**	
	Constraint & Concepts	643	0	4	47	0.956		48	7	55	
	Template Specialization	665	4	1	24	0.902		15	8	23	
Others	—	665	2	2	25	0.923	0.923	20	6	26	26 (3.3%)

In the **Sum** column, the most frequent secondary category within each primary category is bolded. In the **All** column, the most frequent primary category is bolded.

```
/* -- The bug triggering program -- */
struct X {
private:
   int i;
};
struct Y {
  Y (int) { }
};
int main (){
  Y ([] { X x; x.i = 3; return 0; } ()); // Trigger
}
/* ---- The corresponding patch ---- */
  cp_parser_end_tentative_firewall (parser, start,
       lambda_expr);
  + pop_deferring_access_checks ()
```

Listing 2: Member Access Bug GCC-70218 (Accept Invalid)

```
1  /* -- The bug triggering program -- */
2  extern int meminfo ();
3  struct meminfo {};
4  void frob () {
5    meminfo (); // Trigger
6  }
```

Listing 3: Class Scope Bug GCC-80913 (Hang)

```
1  /* -- The bug triggering program -- */
2  template <class> struct A {
3    template <class> struct B {
4     template <class> friend class B; // Trigger
5    protected:
6     int protected_member_;
7    public:
8     template <class T> int method(const B<T>& other)
          const {return other.protected_member_;}
9    };
10 };
11 int main() {
12   A<int>::B<int> a;
13   A<int>::B<char> b;
14   a.method(b);
15 }
```

Listing 4: Friend Bug GCC-87571 (Crash)

`namespace` and `using` statements also affect encapsulation. This complex mechanism usually leads to compiler bugs in implementing access control to class members. For this bug type, the most common root causes are missing invocation of access control APIs and incorrect branch conditions in lookup logic.

① *Member Access.* These bug-contributing features are related to subtle visibility constraints and access permissions of class members. Compiler bugs are caused by member access features when the compiler misinterprets visibility rules, resulting in incorrect access permissions. For instance, compilers may erroneously allow external access to private class members or block member functions from accessing their own private members. Listing 2 shows an invalid program of this category that GCC incorrectly accepts. The compiler incorrectly allows access to the private member variable `x.i` inside a lambda expression, due to a missing API invocation.

② *Class Scope.* The class scope features are related to name lookup across different levels of scopes. These compiler bugs are typically associated with misinterpreting the corresponding variable's scope. One typical scenario is looking up global variable declarations within a class, and searching for class declarations within a namespace, as well as other similar scenarios. Listing 3 shows a program with a naming conflict in the class-level scope, where the function declaration and the struct definition share the same name, `meminfo`. The program correctly calls the function at Line-5 but causes GCC to enter an infinite loop.

③ *Friends.* Friend functions and classes could break C++'s encapsulation restrictions, enabling the sharing of private or protected members between different classes. These compiler bugs arise due to incorrectly handling friend classes or friend function declarations. Listing 4 shows a correct program in this category, where the `friend` declaration inside a nested template class at Line-4 leads to the GCC compiler crash.

2) Object: C++ is a class-based programming language in which *classes* define the structure and behavior of all *objects* created from them. Each object contains storage for both data and function members, following a well-defined lifecycle. C++ also strictly specifies the rules for copying objects, moving objects, and overloading function members. Object-related features trigger this category of bugs. The most

```
1  /* -- The bug triggering program -- */
2  struct A { int a; };
3  struct B { int b; };
4  struct C { A a; B b; };
5  C c{.a=0, .b={12}}; // Trigger
```

Listing 5: Ctor & Dtor Bug LLVM-49020 (Accept Invalid)

```
1  /* -- The bug triggering program -- */
2  struct immovable {
3    immovable() = default;
4    immovable(immovable &&) = delete;
5  };
6  struct S { static immovable f() { return {}; } };
7  immovable g() {
8    return S().f(); // Trigger
9  }
10 /* ---- The corresponding patch ---- */
11   if (TREE_SIDE_EFFECTS (a))
12 -  return build2 (COMPOUND_EXPR, TREE_TYPE (result), a,
        result);
13 +  return cp_build_compound_expr (a, result, tf_error);
```

Listing 6: Copy & Move Bug GCC-110441 (Reject Valid)

```
1  /* -- The bug triggering program -- */
2  struct bar {};
3  void wot(bar x, bar y)
4  {
5    bool operator+(bar, bar);
6    x + y; // Trigger
7  }
8  /* ---- The corresponding patch ---- */
9  - R.setFindLocalExtern(R.getIdentifierNamespace()&Decl::
        IDNS_Ordinary);
10 + R.setFindLocalExtern(R.getIdentifierNamespace()&(Decl
        ::IDNS_Ordinary | Decl::IDNS_NonMemberOperator));
```

Listing 7: Overloading Bug LLVM-27027 (Reject Valid)

```
1  /* -- The bug triggering program -- */
2  struct A { static void f(); };
3  struct B : A {
4    using A::f;
5    static void f(int);
6    auto g() -> decltype(f()); // Trigger
7  };
8  /* ---- The corresponding patch ---- */
9  + tree name = DECL_NAME (using_decl);
10 + tree old_value = lookup_member (t, name, /*protect=*/
        0, /*want_type=*/false, tf_warning_or_error);
11 + tree access = declared_access (using_decl);
```

Listing 8: Inherited Member Access Bug GCC-104476 (Reject Valid)

common root causes of the bugs are erroneous API parameter passing for operator overloading and overlooking certain cases for constructor selection in object initialization, respectively.

① *Constructors & Destructors.* Constructors (i.e., *ctor*) and destructors (i.e., *dtor*) specify the start and the end of the lifecycle of an object. C++ supplies various forms of creating and deleting objects, possibly contributing to compiler bugs. For example, a compiler may incorrectly perform non-static data member initialization (NSDMI), or destroy an object too early or too late. Listing 5 shows an invalid program in this category, which is erroneously accepted by LLVM. The list initialization at Line-5 triggers the compiler bug caused by missing this special case.

② *Copy & Move.* The copy and move mechanisms enable transferring the value of an object to another. The copied object is both equivalent to and independent of the original, i.e., any modification to one does not affect the other. Copying can be costly when dealing with large objects, whereas moving simply transfers ownership. C++ involves plenty of features related to copy and move, such as *copy/move constructor*, *copy/move assignment*, *lvalue*, and *rvalue*. The compiler bugs that contributed to these features are classified in this category. Listing 6 shows a program in this category, which is incorrectly rejected by the GCC compiler. The compiler fails to account for the explicitly deleted move constructor of the struct `immovable` when processing the return statement in function `g`. The root cause of this bug lies in the use of an incorrect API.

③ *Overloading.* C++ supports *function overloading* and *operator overloading*, enhancing the expressiveness and usability of custom types by allowing them to behave like built-in types more intuitively. C++ compilers may incorrectly reference overloaded functions or operators for an object. Listing 7 shows a valid program in this category, which is incorrectly rejected by LLVM. The compiler overlooks the operator overloading declaration at Line-5 in the function. The root cause of the bug is using an incorrect parameter when calling the API.

3) Inheritance: Inheritance is one of the key features of OOP. This category of bugs is triggered by complex inheritance relationships. Compilers are particularly prone to bugs with inheritance due to C++'s support for multiple inheritance, which can lead to intricate hierarchies and introduce the concept of virtual base classes. The most common root causes of these bugs are API misuse and mixing up branch conditions when handling complicated derived class hierarchies.

① *Inherited Member Access.* C++ compilers may incorrectly handle inherited member variables and functions under single inheritance. It is often triggered by the use of `using` declarations, which can alter the visibility of base class members in specific derived classes. Listing 8 shows a valid program in this category, incorrectly rejected by GCC. The reference of `f` at Line-6 should correctly resolve to the function from the base class The root cause of this bug is overlooking the special code logic to handle these cases.

② *Multiple Inheritance.* C++ is one of the few OOP languages that support multiple inheritance. Multiple inheritance brings challenges such as more complex member conflicts and access control issues. To support multiple inheritance, C++ introduces concepts like virtual inheritance and pure virtual functions. C++ compilers are prone to bugs when handling these features. Listing 9 shows a valid program in this category, which causes LLVM to generate the wrong assembly code. The root cause of this bug is using incorrect API.

③ *Inherited Ctor & Dtor.* Compared to languages such as Java and Rust, C++ supports more mechanisms for customizing the inherited class, such as determining the construction order of base classes and leveraging inherited constructors. Bugs arise when C++ compilers incorrectly handle these mechanisms. Listing 10 shows a valid program in this category that GCC

```
/* -- The bug triggering program -- */
struct A { virtual void f() {} };
struct B : virtual A {};
struct C : virtual B, A {}; // Trigger
C c;
/* ---- The corresponding patch ---- */
- if (Layout.getVBaseOffsetsMap().count(NextBase)) {
+ if (isDirectVBase(NextBase, RD)) {
```

Listing 9: Multiple Inheritance Bug LLVM-19066 (Wrong Code)

```
/* -- The bug triggering program -- */
struct payload {};
struct base: private payload {
    base(payload) {}
};
struct derived: base {
    using base::base;
};
int main(){
    derived demo(data); // Trigger
}
/* ---- The corresponding patch ---- */
- tree ctype = DECL_INHERITED_CTOR_BASE (fn);
- if (same_type_ignoring_top_level_qualifiers_p (ptype,
      ctype))
+ tree dtype = DECL_CONTEXT (fn);
+ if (reference_related_p (ptype, dtype))
```

Listing 10: Inherited Ctor & Dtor Bug GCC-79503 (Reject Valid)

```
/* -- The bug triggering program -- */
struct Res { };
struct A {
    virtual Res &&foo() &&;
};
struct B : A {
    Res &foo() && override; // Trigger
};
/* ---- The corresponding patch ---- */
- fail = cp_type_quals (base_return) != cp_type_quals (
     over_return);
+ if (cp_type_quals (base_return) != cp_type_quals (
     over_return))
+    fail = 1;
```

Listing 11: Override Control Bug GCC-99664 (Accept Invalid)

```
/* -- The bug triggering program -- */
struct A { virtual int foo () { return 1; } };
struct B : A { virtual int foo () { return 0; } };
consteval int foo (){
  A *a = new B ();
  return a->foo (); // Trigger
}
/* ---- The corresponding patch ---- */
+  if (VAR_P (obj) && DECL_NAME (obj) == heap_identifier
      && TREE_CODE (objtype) == ARRAY_TYPE)
+    objtype = TREE_TYPE (objtype);
+  if (!CLASS_TYPE_P (objtype)) {
+    if (!ctx->quiet) error_at (loc, "expression␣%qE␣is␣
     not␣a␣constant␣expression", t);
+    *non_constant_p = true;
+    return t;
+  }
```

Listing 12: Casting Bug GCC-93633 (Crash)

incorrectly rejects. The compiler mishandles the inherited constructor in the context of private inheritance, with the root cause of using incorrect APIs.

4) Runtime Polymorphism: To support runtime Polymorphism, which determines incorporates types at runtime, C++ involves features about overloading and runtime type inference (RTTI). The most common root causes of these bugs are API misuse and incorrect branch logic.

① *Override Control*. C++ uses override-related grammars and virtual function tables to support the overriding mechanism, which may be incorrectly implemented by C++ compilers. Listing 11 shows an invalid program that GCC incorrectly accepts. The compiler fails to recognize that the return type of the function `foo` in the derived class `B` (an *lvalue* reference) is not covariant with the return type in the base class `A` (an *rvalue* reference), erroneously permitting the overriding despite the mismatch in return types. The root cause of this bug is the erroneous assignment.

② *Casting*. This type of bug arises from explicit type conversion (or casting) operations between different classes. Specifically, safely casting a base class type to a derived class type typically requires an RTTI check, such as `dynamic_cast`. However, C++ compilers may fail to correctly resolve the type information at runtime. Listing 12 shows an invalid program that triggers a crash in GCC. The compiler overlooked the special case of casting a derived class to its base class in a local `consteval` function.

5) Compile-time Polymorphism: C++ implements compile-time polymorphism through templates, which enable the generation of different instantiations for various type parameters. C++ templates introduce intricate syntax and compile-time checks, when combined with OOP features, these features bring more complex scenarios. Compilers are prone to bugs during stages such as parsing, instantiation, and specialization. The most common root causes of these bugs are overlooked cases and API misuse.

① *Template Alias*. Template alias declarations enable the concise representation of complex types (e.g., `using Type1 = Type2<Type3>`), which may involve recursive. Bugs arise when C++ compilers fail to parse or resolve these alias declarations correctly. Listing 13 shows an invalid program that triggers a crash in GCC. The compiler crashes when parsing the template argument deduction statement involving a template alias. The root cause of this bug is overlooking the special cases.

② *Template Parsing*. C++ template classes often involve complex syntax, such as intricate template parameter declarations, which can lead to compiler parsing failures. The code that triggers these bugs is typically concise but contains expressions that appear deceptively complex, which may be either syntactically valid or invalid. Note that, instantiating the corresponding template class is not required to trigger such bugs. Listing 14 shows a valid program that triggers a crash in GCC. GCC incorrectly handles the `nonexcept` expression declaration at Line-4 in templates involving lambda expressions.

③ *Template Instantiation*. The instantiation phase of C++ template classes produces concrete code definitions based on the provided template arguments. This process involves several intricate steps, including name binding (i.e., associating a type

```
/* -- The bug triggering program -- */
template <class> struct A;
template <class, class B> using C = A<B>; // Trigger
void *f { C(); }
/* ---- The corresponding patch ---- */
+ if (complain & tf_error)
+     error ("alias␣template␣deduction␣only␣available␣
      with␣%<-std=c++20%>␣or␣%<-std=gnu++20%>");
```

Listing 13: Template Alias Bug GCC-99008 (Crash)

```
/* -- The bug triggering program -- */
template <typename XK>
struct zq {
  void ky () noexcept ([]{}); // Trigger
};
/* ---- The corresponding patch ---- */
+ else if (t == error_mark_node)
+   dependent_p = false;
+ else
    dependent_p = (type_dependent_expression_p (t)
                   || value_dependent_expression_p (t));
```

Listing 14: Template Parsing Bug GCC-92531 (Crash)

```
1  /* -- The bug triggering program -- */
2  template<typename T> void f() {
3  struct S {
4    void g(int n=T::error) noexcept(T::error);// Trigger
5   };
6  }
7  template void f<int>();
```

Listing 15: Template Instantiation Bug LLVM-21332 (Accept Invalid)

```
1  /* -- The bug triggering program -- */
2  template<typename, typename>
3  concept false_ = false;
4  template<typename> struct s {
5    static bool f(false_<char> auto); // Trigger
6  };
7  auto x = s<char>::f(1);
```

Listing 16: Constraints and Concepts Bug LLVM-44809 (Accept Invalid)

parameter with its corresponding type variable), type inference, and type-dependence validation (ensuring valid member access for classes used as type parameters). These complex type-related features make C++ compilers particularly prone to bugs. Listing 15 shows an invalid program that LLVM incorrectly accepts. The instantiation at Line 7 is expected to instantiate the entire function, including the local class. However, LLVM fails to verify whether the type parameter includes the member `error`.

④ *Constraints and Concepts*. Constraints and concepts in C++ templates are features introduced in C++20, which enable defining requirements for template parameters in a more structured and expressive way. As C++20 is under development, the C++ compilers easily incorrectly implement these features. Listing 16 shows an invalid program that LLVM incorrectly accepts. LLVM fails to evaluate the concept expression at Line 3, causing the constraint at Line 5 to be incorrectly satisfied.

⑤ *Template Specialization*. C++ template specialization in C allows us to customize the behavior of a template for specific types or conditions. C++ compilers are particularly prone to bugs when handling these features, especially the mechanisms for partial specialization. Listing 17 shows a valid program that LLVM incorrectly rejects. The compiler fails to search for the specialization at Line-5 during the instantiation of the template class.

6) Others: The features can not be classified into any other categories.

B. RQ1: OOP Feature Distribution

The right-most section (i.e., the *Distribution* section) of Table II illustrates the number of C++ compiler bugs contributed via OOP features. The columns **GCC** and **LLVM** show the number of bugs of a secondary category, respectively. The column **Sum** shows the sum of the bugs of the two compilers, while the column **All** shows the number of bugs of a primary category.

From the table, we can find that *Object* is the most common primary category, accounting for 263 bugs (i.e., 33.4%) of the total bugs. Within this primary category, the secondary category *Ctor & Dtor* is the most prevalent, accounting for 174 bugs (22.1%), which also ranks as the most frequent secondary category across all categories. Our analysis shows that the reason why object initialization and destruction introduce so many bugs is two-fold. First, C++ offers a variety of initialization methods, such as default initialization, direct initialization, value initialization, list initialization, aggregate initialization, etc. For example, there are many ways for initializing a new object of `T`, such as `T();`, `T{};`, `new T();`, and `new T{};`. Especially, more complex mechanisms are provided for initializing arrays and structs. Second, the initialization process in C++ often interacts with other features, leading to more corner cases. For instance, marking a constructor as `constexpr` requires the compiler to verify whether class member variables are constants at compile-time.

Finding 1: *Object* is the most common primary category, accounting for 33.4% of all bugs. Among its secondary categories, *Ctor & Dtor* is the most common OOP feature that contributes to C++ compiler bugs, accounting for 22.1% of all bugs.

Compile-time Polymorphism is the second most common primary category, accounting for 261 bugs (i.e., 33.1%). In this primary category, *Template Instantiation* is the most common category, accounting for 112 bugs (i.e., 14.2%), achieving the second most among all secondary categories. Our analysis shows that the reason why compile-time polymorphism introduces many bugs is two-fold. First, the semantics of template classes are intrinsically complex, involving type deduction, template-level type/member checking, template instantiation, and template specification. Second, every new C++ standard version introduces new syntax for template classes. As a result, many bugs arise due to causes such as overlooking corner cases and branching conditions errors.

```
/* -- The bug triggering program -- */
enum EnumType { A };
class MyClass {
 template<int I> inline void set(int v);
 template<EnumType P> inline void set(int v, int I); //
      Trigger
};
template<> void MyClass::set<A>(int v, int I) { }
```

Listing 17: Template Specialization Bug LLVM-24921 (Reject Valid)

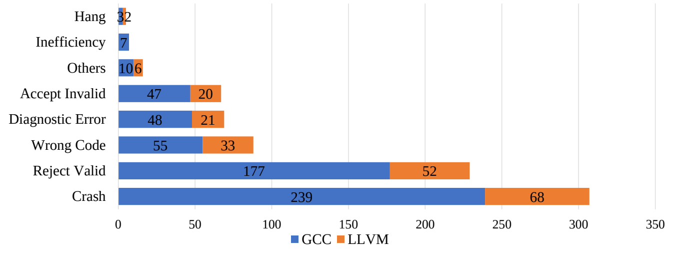


Fig. 1: Bug Distribution by Symptoms

Finding 2: *Compile-time Polymorphism* is the second most common primary category, accounting for 33.1% of all bugs. Within this category, *Template Instantiation* is the most prevalent secondary category, accounting for 14.2% of all bugs.

V. Analysis of Symptoms, Root Causes, Compiler Options, and C++ Standards

In this section, we present the results and analysis of the symptoms, root causes, bug-triggering compiler options, and C++ standards.

A. RQ2: Symptoms

1) Symptoms Identification Results: We identify the symptoms of compilers following existing studies [16]–[18], [22].

① *Accept Invalid*: the compiler accepts an invalid code (i.e., syntactically incorrect code) that should be rejected.

② *Reject Valid*: the compiler rejects a valid code (i.e., syntactically correct code) that should pass the compilation.

③ *Crash*: the compiler unexpectedly terminates during compilation, often with error messages such as stack trace.

④ *Wrong Code*: the compiler successfully finishes the compilation but generates incorrect results or middle results.

⑤ *Diagnostic Error*: the compiler misses or throws incorrect errors or warnings.

⑥ *Inefficiency*: the compiler consumes an excessive amount of time or memory than expected.

⑦ *Hang*: the compiler cannot terminate within a long period.

⑧ *Others*: the symptoms beyond the aforementioned types, such as linkage errors or platform support issues.

2) Symptoms Distribution: Figure 1 presents the bug distribution by the symptoms, where the numbers for each compiler are labeled on their bars. We can find that *Crash* is the most common symptom, accounting for 39.1% of the bugs. In addition, *Reject Valid*, *Wrong Code*, *Diagnostic Error*, and *Accept Valid* account for 29.1%, 11.2%, 8.8%, and 8.5% of the bugs, respectively, which indicates that they are non-negligible. *Crash* is the most common symptom partially because it can be easily observed by compiler users or testers. In contrast, detecting other symptoms is often challenging due to a lack of proper oracles. For example, developing a general oracle for the *Accept Invalid* bug (for instance, the one shown in Listing 15) would be extremely challenging. Different from other types of compilers [18], [22] where crash dominates the symptoms, the majority of OOP-related C++ compiler bugs do not lead to crashes. This suggests that we must address the oracle problem properly to reveal such bugs.

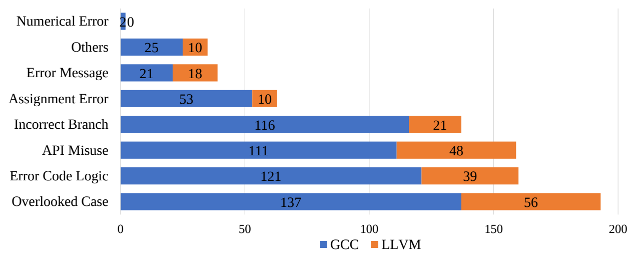


Fig. 2: Bug Distribution by Root Cause

Finding 3: Although *Crash* is the most common symptom, the majority (i.e., 61.0%) of bugs manifest non-crash symptoms.

B. RQ3: Root Causes

1) Root Cause Identification Results: We identify the root causes of compiler bugs following existing studies related to compiler [15], [16], [18], [22].

① *API Misuse*: developers misunderstand APIs, including: 1) calling an incorrect APR; 2) using wrong API parameters; 3) omitting a required API call; and 4) redundantly calling an unnecessary API.

② *Overlooked Case*: developers overlook some special cases, e.g., adding a new block of logic or a new case branch into a large switch statement.

③ *Incorrect Branch*: developers use incorrect conditions in branch statements, e.g., narrowing or widening condition expressions.

④ *Incorrect Assignment*: developers assign wrong values to variables.

⑤ *Error Message*: developers use the wrong diagnostic messages.

⑥ *Numerical Issue*: developers use incorrect numerical computations, e.g., using wrong arithmetic operands or wrong constant values.

⑦ *Error Code Logic*: developers incorrectly implement an algorithm or a process related to parsing, code generation, or optimization. This root cause involves significant changes that combine multiple other root causes.

⑧ *Others*: caused by other reasons.

2) Root Cause Distribution: Figure 2 shows the root cause distribution, from which we can find the top three common root causes are *Overlooked Case*, *Error Code Logic*, and *API Misuse*, accounting for 24.5%, 20.3%, and 20.2% of bugs, respectively. We conjecture that these causes are common mainly due to the complexity of the OOP-related syntax and semantic rules, leading to developers often overlook special cases. The *Numerical Error* bugs are the least common, comprising only 0.3%, suggesting that OOP-related bugs seldom involve arithmetic calculations.

> **Finding 4**: The top three common root causes are *Overlooked Case*, *Error Code Logic*, and *API Misuse*.

C. RQ4: The Correlation between Bug Types and Symptoms

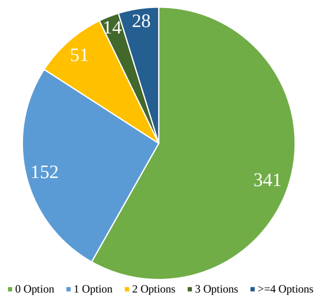
We investigate whether a certain correlation exists between the bug-triggering OOP features and the bug symptoms. Table III presents the distribution of symptoms for each bug type. The rows represent the bug types of secondary categories, with the most frequent symptom highlighted in dark gray and the second most frequent in light gray. Each column represents a symptom where the bug type with the highest frequency is in bold.

From the table, we make the following observations. First, in the majority of secondary categories, *Crash* and *Reject Valid* are the most and second most frequent symptoms, respectively. Specifically, across all 17 secondary categories, *Crash* is the most common symptom in 14 categories. Meanwhile, *Reject Valid* is the most frequent symptom in 4 categories and the second most frequent in 11 categories. Second, among all symptoms, only *Wrong Code* and *Efficiency* have a dominant associated OOP feature. Specifically, for bugs exhibiting *Wrong Code*, the majority are triggered by *Ctor & Dtor*, occurring three times more frequently than the second most common feature, *Overloading*. The results suggest that *Crash* can be directly used as the test oracle, while differential testing may be able to detect *Reject Valid* bugs.

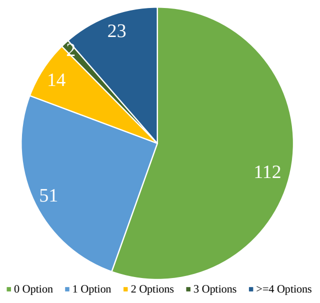
> **Finding 5**: The OOP feature contributing bugs cover a broad spectrum of symptoms, where *Crash* and *Reject Valid* are the most frequent ones among them.

D. RQ5: Bug Triggering Compiler Options

Besides source code, compiler options are another kind of important input, which should be properly set to trigger bugs [23]–[25]. To study the impact of compiler options on discovering bugs related to OOP features in C++ compilers, we analyze the user-submitted options extracted from the discussions among developers. We believe that the options used in the simplified examples provided by issue reporters and developers are already simplified. Therefore, we directly extract the options from bug reports. Figure 3 shows how many options GCC and LLVM use to trigger the bugs. The results show that for GCC, 58.2% of bugs are triggered by no options, and 25.9% only by 1 option. Similarly, for LLVM, 55.4% of bugs do not require any options, and 25.2% are triggered by only 1 option. The results show that 656 (i.e.,

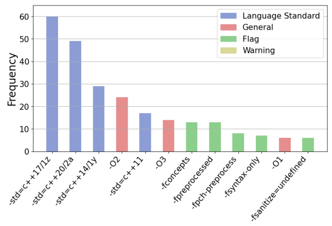


(a) GCC

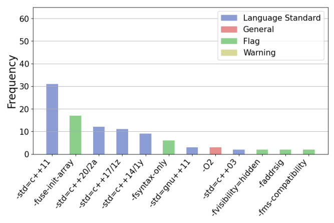


(b) LLVM

Fig. 3: Option Number Distribution



(a) GCC



(b) LLVM

Fig. 4: Most Frequent Options

83.2%) bugs can be triggered by no more than 1 option. For the cases of a single option, we find the majority of cases specify the language standard. As shown later, many bug-triggering programs contain new features from new language standards, and some new features require setting the corresponding standard. The result suggests we could use a simple strategy for generating the compiler options, such as trying one option at a time, to identify many of the bugs. We remark that a non-negligible portion (i.e., 6.5%) of bugs need at least 4 options. This result however should be taken with a grain of salt since not all options may be relevant, i.e., we can minimize the failure-triggering options in these cases.

> **Finding 6**: To trigger the OOP-related bugs in C++ compiler, 83.2% of the bugs need no more than 1 option.

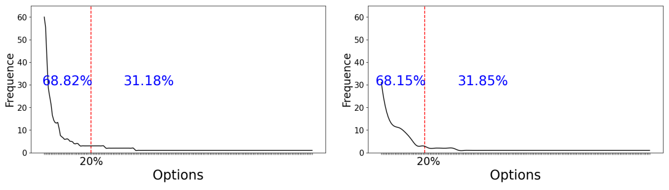
We categorize the involved options into *Language Standard* options to set standard version (e.g., `-std=c++20`), *General* options to set optimization level (e.g., `-O2`), *Flag* options controlling the compile functions that start with `-f` (e.g., `-fsanitize`), and *Warning* options that start with `-W` (e.g., `-Werror`).

Figure 4 shows the 12 most frequently used options, organized by category. Although in both GCC and LLVM, the most frequently used options are *Language Standard*, GCC's most frequently used option is `-std=C++17`, while for LLVM, it is `-std=C++11`. Compared to C++17 and C++11, C++14 has fewer bugs because it introduces few features. In addition, in both compilers, *Language Standard* options are the most frequent in the top 12. These results confirm the previous finding on option numbers, and further imply that we should

TABLE III: The Correlation Between OOP Features and Symptoms

Primary Category	Secondary Category	Acc. Inv.	Rej. Val.	Crash	Wrong Code	Hang	Effic.	Diag.	Others	Sum
Abstr. & Encap.	Member Access	5	13	17	2	0	0	10	1	48
	Class Scope	6	30	15	4	1	0	7	2	65
	Friends	3	1	7	1	0	0	2	1	15
Object	Ctor & Dtor	**13**	**41**	**67**	**31**	1	**5**	**13**	3	174
	Copy & Move	3	12	8	8	0	0	8	3	42
	Overloading	1	14	19	10	1	0	2	0	47
Inheritance	Inherited Member Access	2	8	13	2	0	0	4	1	30
	Multiple Inheritance	2	3	7	5	0	0	2	0	19
	Inherited Ctor & Dtor	0	10	8	9	0	0	2	1	30
Runtime Poly.	Override Control	1	1	4	1	0	0	4	0	11
	Casting	4	6	7	0	0	0	3	0	20
Compile-time Poly.	Template Alias	3	9	23	0	1	1	0	0	37
	Template Parsing	2	9	19	2	0	0	2	0	34
	Template Instantiation	12	38	48	6	1	1	3	3	112
	Constraints & Concepts	6	24	21	2	0	0	2	0	55
	Template Specialization	1	4	14	2	0	0	2	0	23
Others	——	3	6	10	3	0	0	3	1	26
Sum	——	67	229	307	88	5	7	69	16	788

Given a row, the dark gray and light gray cells represent the most and second most frequent symptoms for the corresponding OOP feature, respectively. Given a column, the most frequent feature(s) is bolded.



(a) GCC (b) LLVM

Fig. 5: Long Tail Effect on Compiler Options

properly set the language standard in some cases.

> **Finding 7**: *Language Standard* options are the most frequently used to trigger OOP-related bugs.

We observe from Figure 3 that a non-negligible portion (i.e., 6.4%) of bugs need at least 4 options, so we further calculate the frequency of each option and sort them from high to low, shown in Figure 5. For the top 20% options by frequency, GCC options account for 68.82% of all option usage counts, while LLVM options account for 68.15%. The results exhibit the Long Tail Effect, where the remaining options are still significant. It also shows that the combination of options plays a crucial role in triggering bugs [26]. We remark that the majority of existing bugs requiring few options may be due to inadequate exploration of option combinations, i.e., it is conceivable that there may be subtle bugs that require a combination of many options that are under-explored.

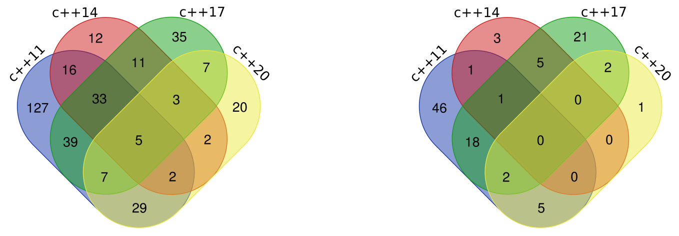
> **Finding 8**: The options in both GCC and LLVM represent the Long Tail Effect.

E. RQ6: The Commonality across C++ Compilers

Following existing studies [16], [18], [27], we adopt the Spearman correlation coefficients to evaluate the commonality, which indicates whether two ranked lists are strongly correlated if the value is in the range of [0.8, 1.0]. We calculated the coefficients of the bug types, symptoms, and the number of

TABLE IV: Spearman Correlation Coefficients of GCC and LLVM

-	Spearman rank	p-value
OOP Feature	0.943	0.005
Symptom	0.976	0.000
Option Number	0.899	0.038



(a) GCC (b) LLVM

Fig. 6: Involving C++ Standards of Bug-Triggering Programs

options to trigger compiler bugs, shown in Table IV. We can find that all the Spearman coefficients are larger than 0.8 and the p-values are less than 0.04, indicating GCC and LLVM have a high correlation with high confidence.

> **Finding 9**: GCC and LLVM have significant commonalities in bug types, symptoms, and the number of options to trigger compiler bugs.

F. RQ7: C++ New Features

We count the features of the modern C++ standards (i.e., C++11, C++14, C++17, and C++20) involved in each bug-triggering source program, shown in Figure 6. In GCC and LLVM, respectively, 59.4% and 52.0% of bugs involve features from the new standards, and the most common one is C++11, which accounts for 44.0% and 36.1% of the bugs. Moreover, 26.3% of bugs in GCC and 15.3% in LLVM cover at least two versions of new standards. These results show that new freshly introduced features may interact with existing OOP features in unexpected ways, resulting in many new cases to be considered and resultant bugs. For example, the bug shown in Listing 1 covers features from multiple standards.

Finding 10: In both compilers, 57.5% of bugs cover C++ new standards, and 23.5% cover at least two versions.

VI. Discussion

In the following, we first discuss the implication of our findings in terms of developing effective fuzzers for revealing OOP-related bugs, then discuss the threats to validity.

A. Implications

Based on our findings, we discuss the implications for improving compiler fuzzers and understanding OOP-related bugs in C++ compilers.

1) New program generation and mutation strategies for fuzzers: According to **Findings 1** and **2**, we find that about 40% of bugs in C++ compilers are related to *compile-time polymorphism*, *object initialization* and *encapsulation*. According to **Finding 10**, many bugs are due to newly introduced OOP features. However, to the best of our knowledge, very few existing fuzzers provide support for these features. We thus suggest that corresponding generation or mutation strategies must be developed and integrated into existing fuzzers to address the bug types in RQ1. For example, we could design mutation operators to rewrite initializers and type parameters of templates. The seed or sketch programs should cover new standard features. According to **Finding 9**, we can use the same fuzzing strategies for GCC and LLVM.

2) Oracle enhancement for automated testing: According to **Findings 3** and **5**, we find that while *Crash* serves as a strong oracle in compiler fuzzing [10], [18] since it is the most common symptom, using it as the sole oracle means that a significant portion (i.e., 61.2%) of bugs will be missed, indicating their limitations in effectively discovering OOP related bugs for C++ compilers. Differential testing could only partially detect bugs of non-crash symptoms, such as *Reject Valid* and *Wrong Code*. Therefore discovering new oracles for compiler testing deserves more attention.

3) Reducing the difficulty of debugging: According to **Finding 4**, when debugging OOP-related bugs, developers should focus on APIs, branch conditions, and corner cases. According to **Finding 5**, except *Diagnostic Error*, none of the symptoms have a strong correlation with the bug types, indicating these bugs are hard to localize, understand, and repair, which calls for specified debug techniques.

4) Considering compiler options: According to **Finding 6**, which reveals that 80.6% of OOP feature handling bugs are triggered by no more than one option, which suggests we employ a minimizing set of options when the test schedule is under a limited time budget. Setting no options or a single predefined option is acceptable in practical automated compiler testing. According to **Findings 7** and **9**, we can employ a proper *Language Standard* option. According to **Finding 8**, if time permits, we should endeavor to explore as many option combinations as possible.

TABLE V: The Bugs Reported via OOPFuzz

ID	Bug ID	Symptom	State
1	LLVM-80963	Accept Invalid	**Confirmed**
2	LLVM-81089	Reject Valid	Still Unconfirmed
3	LLVM-81731	Reject Valid	**Confirmed**
4	LLVM-81733	Reject Valid	Still Unconfirmed
5	LLVM-99491	Accept Invalid	Still Unconfirmed
6	GCC-113830	Accept Invalid	Still Unconfirmed
7	GCC-113916	Accept Invalid	**Confirmed**

B. Threats to Validity

Similar to existing empirical studies, our study is potentially subject to the threats introduced by manual inspections, including identifying keywords, bug types, symptoms, root causes, options and involved C++ standards in bug-triggering inputs. To reduce the threat, we referred to existing empirical studies on bugs [16], [18], [20], [22], [28]–[30], adopted their process for extracting bug types, symptoms and root causes, and adapted their symptoms to C++ compilers. In addition, the two authors labeled the bugs separately, and if different results arose, we discussed them until an agreement was reached. The most experienced author goes through all labeling results to ensure quality.

Another threat mainly lies in the compiler and bugs used in our study. To reduce this threat, we use the most popular C++ compilers which are widely used in existing studies [10]–[12], and we systematically collect the bug reports that are successfully fixed. Moreover, our study is large-scale, covering 788 real-world bugs across 10 years, ensuring the generalizability of our results.

VII. Proof-of Concept Application

To demonstrate the usefulness of our findings, we developed a preliminary proof-of-concept application **OOPFuzz**, which aims to automatically generate programs enhanced with OOP features for testing C++ compilers. Based on the findings and implications, we design a mutation-based strategy for OOPFuzz, equipped with the following mutation operators: (1) Expanding useless type parameters in the definitions of template classes; (2) Changing the access control modifiers of constructors and destructors; (3) Adding an empty base class; (4) Replacing object initialization to another semantic equivalent form. We use crash and differential testing as test oracles, i.e., the behaviors between compilers should be consistent. For the compiler options, we only use the language standard option.

We applied OOPFuzz on the latest version of GCC and LLVM (i.e., GCC 13.2 and Clang 17.0), and employed the bug-triggering programs collected from our study as seed programs. OOPFuzz detected 9 bugs in about 3 hours. We submitted 7 new bug reports because 2 bugs had already been fixed in the trunk branches 3 weeks ago. All the 7 bug reports as shown in Table V, in which 3 have been confirmed by the developers, whereas 2 are from LLVM and 1 is from GCC. In particular, OOPFuzz found a bug in LLVM that has remained hidden for 13 years, where the developer commented: "*This goes all the way back to clang-3.0, I am surprised we have not seen this before*"[3].

[3]https://github.com/llvm/llvm-project/issues/81731

Although OOPFuzz is simple and is given a rather short time budget, it can detect several new bugs that are confirmed by developers. The results demonstrate that our finding is useful, and it is promising for detecting OOP-related bugs in C++ compilers.

VIII. Related Work

A. Empirical Studies of Bugs

The most related work is the empirical studies on the bugs of GCC and LLVM. Sun *et al.* conducted a quantitative analysis of the overall distribution of bugs in GCC and LLVM [10], pointing out that the C++ component is the most buggy-prone. Sun *et al.* also studied the characteristics of warning defects in both compilers [12]. Zhou *et al.* analyzed their characteristics in optimization bugs [11]. These studies do not analyze the OOP-related bugs and the relationship between compiler bugs and language features.

Besides, other types of compilers are extensively empirically studied, such as JVMs [31], WebAssembly compilers [17], deep learning compilers [18], [32], MarkDown [33], Python [15], [34], Solidity [35], and Rust [36]. Moreover, researchers invest much research effort in analyzing bug characteristics in other fields, such as OS [37], quantum computing platforms [22], deep learning frameworks [16], deep learning models [29], SMT solvers [38], and Android apps [39].

B. Compiler Fuzzers

Compiler fuzzers generate programs to test compilers, which have detected thousands of bugs and attract much academic [1], [2], [40] and industrial attention [41]. Existing compiler fuzzers can be classified into generation-based and mutation-based. Generation-based approaches generate programs from scratch by pre-defined grammar rules. For example, the CSmith family randomly generates programs under simplified semantics [4], [25], [42]–[46]. YARPGen [5], [7] is a recent fuzzer for C/C++, armed with more grammar features. The most recent generation-based approaches leverage LLM for generating code [8], [9], [47]. However, these fuzzers exclude OOP-related grammar rules and thus lack OOP support. Moreover, many generation strategies are designed for other types of compilers [48]–[50]. Mutation-based approaches apply mutation operators to seed programs and transform them into new programs. Fuzzers of this family including C approaches [6], [26], [27], [51]–[55], JVM testing [56]–[64], MLIR compilers [44], [65], and deep learning compilers [24].

IX. Conclusion

In this work, we conduct a comprehensive study of the OOP-related bugs in C++ compilers. We analyzed the OOP features, symptoms, root causes, options, and C++ standards of these bugs, and proposed 10 findings. Our findings are useful and provide guidelines for improving compiler fuzzers. Based on these guidelines, we developed a simple fuzzer, which finds 9 real bugs and 3 of them have been confirmed.

References

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

[2] 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.

[3] T. Hoare, "The verifying compiler: A grand challenge for computing research," *Journal of the ACM (JACM)*, vol. 50, no. 1, pp. 63–69, 2003.

[4] 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.

[5] 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.

[6] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," *ACM Sigplan Notices*, vol. 49, no. 6, pp. 216–226, 2014.

[7] V. Livinskii, D. Babokin, and J. Regehr, "Fuzzing loop optimizations in compilers for c++ and data-parallel languages," *Proceedings of the ACM on Programming Languages*, vol. 7, no. PLDI, pp. 1826–1847, 2023.

[8] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," *Proc. IEEE/ACM ICSE*, 2024.

[9] X. Ou, C. Li, Y. Jiang, and C. Xu, "The mutators reloaded: Fuzzing compilers with large language model generated mutation operators." ASPLOS, 2024.

[10] 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.

[11] 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, p. 110884, 2021.

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

[13] B. Stroustrup, "The c++ programming language," 2013.

[14] F. Macklon, M. Viggiato, N. Romanova, C. Buzon, D. Paas, and C.-P. Bezemer, "A taxonomy of testable html5 canvas issues," *IEEE Transactions on Software Engineering*, vol. 49, no. 6, pp. 3647–3659, 2023.

[15] D. Liu, Y. Feng, Y. Yan, and B. Xu, "Towards understanding bugs in python interpreters," *Empirical Software Engineering*, vol. 28, no. 1, p. 19, 2023.

[16] J. Chen, Y. Liang, Q. Shen, J. Jiang, and S. Li, "Toward understanding deep learning framework bugs," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 6, pp. 1–31, 2023.

[17] A. Romano, X. Liu, Y. Kwon, and W. Wang, "An empirical study of bugs in webassembly compilers," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 42–54.

[18] Q. Shen, H. Ma, J. Chen, Y. Tian, S.-C. Cheung, and X. Chen, "A comprehensive study of deep learning compiler bugs," in *Proceedings of the 29th ACM Joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2021, pp. 968–980.

[19] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in *Proceedings of the ACM/IEEE 42nd international conference on software engineering*, 2020, pp. 1110–1121.

[20] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," *Empirical software engineering*, vol. 19, pp. 1665–1705, 2014.

[21] A. J. Viera, J. M. Garrett *et al.*, "Understanding interobserver agreement: the kappa statistic," *Fam med*, vol. 37, no. 5, pp. 360–363, 2005.

[22] M. Paltenghi and M. Pradel, "Bugs in quantum computing platforms: an empirical study," *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA1, pp. 1–27, 2022.

[23] H. Jia, M. Wen, Z. Xie, X. Guo, R. Wu, M. Sun, K. Chen, and H. Jin, "Detecting jvm jit compiler bugs via exploring two-dimensional input spaces," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 43–55.

[24] C. Li, Y. Jiang, C. Xu, and Z. Su, "Validating jit compilers via compilation space exploration," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 66–79.

[25] J. Chen, C. Suo, J. Jiang, P. Chen, and X. Li, "Compiler test-program generation via memoized configuration search," in *Proc. 45th International Conference on Software Engineering*, 2023.

[26] H. Jiang, Z. Zhou, Z. Ren, J. Zhang, and X. Li, "Ctos: Compiler testing for optimization sequences of llvm," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2339–2358, 2021.

[27] 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*, vol. 48, no. 11, pp. 4411–4432, 2021.

[28] J. Garcia, Y. Feng, J. Shen, S. Almanee, Y. Xia, and Q. A. Chen, "A comprehensive study of autonomous vehicle bugs," in *Proceedings of the ACM/IEEE 42nd international conference on software engineering*, 2020, pp. 385–396.

[29] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2019, pp. 510–520.

[30] F. Thung, S. Wang, D. Lo, and L. Jiang, "An empirical study of bugs in machine learning systems," in *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. IEEE, 2012, pp. 271–280.

[31] S. Chaliasos, T. Sotiropoulos, G.-P. Drosos, C. Mitropoulos, D. Mitropoulos, and D. Spinellis, "Well-typed programs can go wrong: A study of typing-related bugs in jvm compilers," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–30, 2021.

[32] X. Du, Z. Zheng, L. Ma, and J. Zhao, "An empirical study on common bugs in deep learning compilers," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2021, pp. 184–195.

[33] P. Li, Y. Liu, and W. Meng, "Understanding and detecting performance bugs in markdown compilers," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 892–904.

[34] X. Xia, Y. Feng, Q. Shi, J. A. Jones, X. Zhang, and B. Xu, "Enumerating valid non-alpha-equivalent programs for interpreter testing," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 5, pp. 1–31, 2024.

[35] H. Ma, W. Zhang, Q. Shen, Y. Tian, J. Chen, and S.-C. Cheung, "Towards understanding the bugs in solidity compiler," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1312–1324.

[36] X. Xia, Y. Feng, and Q. Shi, "Understanding bugs in rust compilers," in *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*. IEEE, 2023, pp. 138–149.

[37] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *Proceedings of the eighteenth ACM symposium on Operating systems principles*, 2001, pp. 73–88.

[38] M. Bringolf, D. Winterer, and Z. Su, "Finding and understanding incompleteness bugs in smt solvers," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–10.

[39] Y. Xiong, M. Xu, T. Su, J. Sun, J. Wang, H. Wen, G. Pu, J. He, and Z. Su, "An empirical study of functional bugs in android apps," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1319–1331.

[40] 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.

[41] Y. Zhao, J. Chen, R. Fu, H. Ye, and Z. Wang, "Testing the compiler for a new-born programming language: An industrial case study (experience paper)," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 551–563.

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

[43] M. Sharma, P. Yu, and A. F. Donaldson, "Rustsmith: Random differential compiler testing for rust," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1483–1486.

[44] H. Wang, J. Chen, C. Xie, S. Liu, Z. Wang, Q. Shen, and Y. Zhao, "Mlirsmith: Random program generation for fuzzing mlir compiler infrastructure," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 1555–1566.

[45] 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.

[46] J. Wu, Y. Yang, and Y. Zhou, "Boosting compiler testing via eliminating test programs with long-execution-time," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2023, pp. 593–603.

[47] C. Yang, Y. Deng, R. Lu, J. Yao, J. Liu, R. Jabbarvand, and L. Zhang, "White-box compiler fuzzing empowered by large language models," *arXiv preprint arXiv:2310.15991*, 2023.

[48] H. Ma, Q. Shen, Y. Tian, J. Chen, and S.-C. Cheung, "Fuzzing deep learning compilers with hirgen," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 248–260.

[49] Z. Wang, P. Nie, X. Miao, Y. Chen, C. Wan, L. Bu, and J. Zhao, "Gencog: A dsl-based approach to generating computation graphs for tvm testing," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 904–916.

[50] J. Liu, J. Lin, F. Ruffy, C. Tan, J. Li, A. Panda, and L. Zhang, "Nnsmith: Generating diverse and valid test cases for deep learning compilers," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 530–543.

[51] K. Even-Mendoza, A. Sharma, A. F. Donaldson, and C. Cadar, "Grayc: Greybox fuzzing of compilers and analysers for c," 2023.

[52] E. Liu, S. Xu, and D. Lie, "Flux: Finding bugs with llvm ir based unit test crossovers," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 1061–1072.

[53] H. Tu, H. Jiang, Z. Zhou, Y. Tang, Z. Ren, L. Qiao, and L. Jiang, "Detecting c++ compiler front-end bugs via grammar mutation and differential testing," *IEEE Transactions on Reliability*, vol. 72, no. 1, pp. 343–357, 2022.

[54] H. Zhong, "Enriching compiler testing with real program from bug report," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[55] K. Even-Mendoza, C. Cadar, and A. F. Donaldson, "Csmithedge: more effective compiler testing by handling undefined behaviour less conservatively," *Empirical Software Engineering*, vol. 27, no. 6, p. 129, 2022.

[56] R. Schumi and J. Sun, "Spectest: Specification-based compiler testing," in *Fundamental Approaches to Software Engineering: 24th International Conference*. Springer International Publishing, 2021, pp. 269–291.

[57] M. Wu, M. Lu, H. Cui, J. Chen, Y. Zhang, and L. Zhang, "Jitfuzz: Coverage-guided fuzzing for jvm just-in-time compilers," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 56–68.

[58] T. Gao, J. Chen, Y. Zhao, Y. Zhang, and L. Zhang, "Vectorizing program ingredients for better jvm testing," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 526–537.

[59] Z. Zang, F.-Y. Yu, N. Wiatrek, M. Gligoric, and A. Shi, "Jattack: Java jit testing using template programs," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2023, pp. 6–10.

[60] Z. Zang, N. Wiatrek, M. Gligoric, and A. Shi, "Compiler testing using template java programs," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

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

[62] 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.

[63] Y. Chen, T. Su, and Z. Su, "Deep differential testing of jvm implementations," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1257–1268.

[64] S. Chaliasos, T. Sotiropoulos, D. Spinellis, A. Gervais, B. Livshits, and D. Mitropoulos, "Finding typing compiler bugs," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2022, pp. 183–198.

[65] C. Suo, J. Chen, S. Liu, J. Jiang, Y. Zhao, and J. Wang, "Fuzzing mlir compiler infrastructure via operation dependency analysis," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1287–1299.