

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

Bo Wang¹, Chong Chen¹, Junjie Chen¹, Bowen Xu², *Member, IEEE*, Chen Ye¹, Youfang Lin¹,
Guoliang Dong¹, and Jun Sun¹

Abstract—Modern C++, a programming language characterized by its extensive use of object-oriented programming (OOP) features, is widely used 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, existing studies largely overlook OOP features, 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., *Constructors & Destructors* and *Multiple Inheritance*). Third, we systematically analyze the root causes, symptoms, fixes, options, and C++ standard versions of these bugs. Our analysis yields 13 key findings, highlighting that features related to the construction and destruction of objects lead to the highest number of bugs, crashes are the most frequent symptom, and while the average time from bug introduction to discovery is 1856 days, fixing the bug once discovered takes only 174 days on average. Additionally, more than half of the bugs can be triggered without any compiler options. These findings offer valuable insights not only for developing new compiler testing approaches but also for improving language design and compiler engineering. Inspired by these findings, we developed a

proof-of-concept compiler fuzzer OOPFuzz, specifically targeting OOP-related bugs in C++ compilers. We 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 transform programs written in high-level languages (e.g., C/C++) into lower-level representations such as assembly or machine code. 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 many 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 the existing bugs of the compilers for the target language. Existing methods have successfully revealed thousands of bugs in many compilers [1], [4], [5], [6]. Yet, an important class of compiler bugs 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 influential programming languages in system-level programming. To support OOP, C++ introduces a large number of language features allowing for flexible and diverse programming styles. Moreover, C++ continues to evolve, with numerous new concepts, operators, and syntactic sugars being introduced over the years. Many new features have complex semantics. It is a significant challenge to implement and test C++ compilers. Compiler bugs that are caused by handling C++

Received 13 December 2024; revised 24 April 2025; accepted 27 April 2025. Date of publication 5 May 2025; date of current version 16 June 2025. This work was supported in part by the National Natural Science Foundation of China under Grant 62202040, Grant 62322208, and Grant 12411530122, and in part by the Ministry of Education, Singapore, under its Academic Research Fund Tier 2 under Grant T2EP20222-0037. Recommended for acceptance by H. Zhong. (Corresponding authors: Junjie Chen; Youfang Lin.)

Bo Wang, Chong Chen, Chen Ye, and Youfang Lin are with the School of Computer Science and Technology, Beijing Jiaotong University, Beijing 100044, China, and also with Beijing Key Laboratory of Traffic Data Mining and Embodied Intelligence, 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, Raleigh, 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 (e-mail: gldong@smu.edu.sg; junsun@smu.edu.sg).

Digital Object Identifier 10.1109/TSE.2025.3566490

OOP features are referred to as *OOP-related* bugs. The widely studied bug types of state-of-the-art research primarily focus on optimization or code generation, which significantly differ from OOP-related bugs. For example, most OOP-related bugs do not involve complex computations or advanced architecture mechanisms but instead arise from the use of OOP-specific mechanisms or data structures.

In this paper, we conduct a systematic study of OOP-related bugs in C++ compilers. Our motivations are threefold. First, OOP-related bugs are both prevalent and influential in C++ compilers. As shown in Section III, 18% of reported bugs in C++ compilers involve OOP features, making them a significant category of compiler issues. Moreover, 40% of these bugs are assigned high priority, and over 90% result in a compiler crash or miscompilation, posing risks to software reliability. Understanding these bugs is crucial not only for compiler correctness but also for ensuring the safety and correctness of compiled applications, as miscompilation can silently introduce unintended behavior. Second, understanding the OOP-related features where compilers struggle, providing insights that can enhance both compiler engineering and language design. As C++ continues to evolve with more complex OOP mechanisms, analyzing the bug-leading features can serve as a cautionary guide for language designers and compiler developers. Third, a deeper understanding of OOP-related bugs can guide new directions to improve compiler testing and debugging. Existing C++ compiler fuzzers rarely generate complex code that includes OOP features. CSmith [4], the pioneering C compiler fuzzer, supports none of the OOP features in C++. Another influential C++ fuzzer, YARPGen [5], [7], does not generate classes and their dynamic memory allocation. CCOFT [8], a C++ front-end fuzzer, supports limited class definitions based on predefined templates. Fuzz4All [9] and MetaMut [10], the recently developed LLM-based fuzzers, support limited OOP features and hardly generate correct non-trivial programs. These highlight the need for more OOP-centered compiler fuzzing approaches, which can be inspired by our study.

Although several empirical studies on compiler bugs have been conducted, OOP-related bugs have been largely overlooked so far. Sun et al. conducted the first empirical study on the characteristics of bugs in GCC and LLVM [11], analyzing buggy-prone components, the length of triggering test cases, the size of patches, and the lifespan of bugs. However, they omitted further analysis of the bug types, symptoms, and root causes. Moreover, they focused on the general compiler bugs of both compilers, overlooking advanced features of C++. To motivate their compiler fuzzer design, Tu et al. studied the bugs of the C++ components in GCC [8]. However, their study was limited to a single compiler and focused solely on the symptom distribution of general C++ bugs. Zhou et al. studied the optimization bugs of GCC and LLVM [12] 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 the bugs, (3) the *root causes* of the bugs, (3) the characteristics of the *fixes*, 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 OOP-related bugs?**
- **RQ3: What are the root causes of OOP-related bugs?**
- **RQ4: What are the characteristics of the fixes of OOP-related bugs?**
- **RQ5: What is the relationship between the bug-triggering OOP features and the symptoms?**
- **RQ6: What compiler options are relevant for triggering OOP-related bugs?**
- **RQ7: Do the bugs from different C++ compilers share certain commonality?**
- **RQ8: How does the evolution of C++ standards affect the OOP-related bugs of compilers?**

We develop a taxonomy of C++ OOP features 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 13 major findings that we believe help design testing methods for revealing OOP-related compiler bugs. These findings reveal that features related to the construction and destruction of objects contribute to the highest number of bugs, with crashes being the most frequent symptom, though not the dominant one. Additionally, while the average time from bug introduction to discovery is 1856 days, once identified, bugs take an average of 174 days to be fixed. More than half of the bugs require only minor modifications, and more than half of the bugs can be triggered without any compiler options.

Based on the empirical study results and findings, we propose practical implications for the design of future C++ compiler fuzzers, as well as for language design and compiler engineering. These implications improve the testing coverage of modern C++ features, provide cautionary guidelines for new feature design and compiler implementation, and finally enhance C++ compiler robustness and reliability. Furthermore, based on the guidelines, we designed and implemented a simple yet effective proof-of-concept C++ compiler fuzzer, called **OOPFuzz**, enhancing the generation of 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 available¹, for the sake of reproducibility and open science.

To sum up, we make the following major contributions.

- 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, fixes, options, and involved C++ standards.
- We obtain 13 findings and provide guidelines for developing further fuzzers, debugging OOP-related bugs, language design, and compiler engineering.
- 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 `constexpr` is frequently used to optimize code generation for member functions.

1) *Abstraction and Encapsulation*: *Abstraction* is the modeling of real-world concepts, while *Encapsulation* is the structured organization and access control of the abstracted members. Both are fundamental and inherently inseparable in OOP. C++ provides numerous features to support these principles, including data abstraction, visibility control, and function and data binding. 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. Many other OOP language, such as Java, supports a more restricted set of class forms and do not provide constructs like `union` or `namespace`. The OOP features of C++ are centered around the notion of *class*, which models the real-world abstracted entities by encapsulating their attributes (i.e., data members) and behaviors (i.e., member functions). 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 by other classes. Even in lambda expressions, which are syntactic sugar for anonymous functions, users can encapsulate operations and data.

2) *Object*: Similar to many other OOP languages, C++ uses a *class* to abstractly define a common structure of entities, while instantiating a class creates *objects* that share this structure. Conceptually, an object's data members and member functions are stored within a memory layout determined by the structure of the *class*. An OOP programming language should specify the features of object management, such as lifecycle and ownership semantics. Many OOP languages, such as Java, rely on garbage collection to automate object lifecycle management and provide relatively simple rules for ownership transfer. In contrast, C++ offers explicit and fine-grained control over both the lifecycle and ownership of objects through a rich set of language features. Object creation and destruction are managed via constructors (*ctor*) and destructors (*dctor*), respectively. Additionally, C++ supports precise control over ownership transfer and duplication through copy constructors, move constructors, copy assignment operators, and move assignment operators. 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*: The inheritance in C++ is used to establish a hierarchical relationship between classes, allowing a derived class to inherit attributes and behaviors from a base class. 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. While Java does not support multiple inheritance of classes, nor does it allow changing inheritance visibility through `private` or `protected` inheritance. All class inheritance in Java is `public`, and developers must rely on interfaces or composition to implement other forms of inheritance. A key issue introduced by inheritance is to ensure the order of calling the constructors and destructors 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*: Polymorphism enables flexibility by allowing code to operate on different *types*, facilitating code reuse and extensibility. C++ supports both *runtime polymorphism* and *compile-time polymorphism*, enabling 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. It enables dynamic dispatch, allowing a base class reference or pointer to invoke derived class implementations. Additionally, *Runtime Type Information* (RTTI) can be used to ensure type safety during dynamic type casting. Compared to C++, Java enforces runtime polymorphism

¹<https://github.com/Rush10233/OOP-Related-Bugs-Study-TSE-Artifacts>

```

1  /* ---- The bug-triggering program ---- */
2  struct base {
3      template <typename t> requires false base(t);
4      template <typename t> requires true base(t);
5  };
6  struct derived : base { using base::base; };
7  int main() { derived{'G'}; }
8  /* ---- The corresponding patch ---- */
9  - if (flag_new_inheriting_ctors)
10 + if (!DECL_TEMPLATE_INFO (t))

```

Listing 1. An Example OOP-Related Bug: GCC-94549

more strictly. For instance, all non-static methods are virtual by default, and dynamic dispatch is applied uniformly, and Java performs automatic dynamic type checks during downcasting.

Compile-time polymorphism, in contrast, occurs when the function to be executed is determined at compile time, which is achieved through templates and overloading in C++. In C++, overloading allows multiple functions or operators with the same name or symbol to be defined, each accepting different parameter types. C++ templates enable generic programming by allowing functions and classes to operate with any data type, defined at compile time. C++ also introduced *concepts* in C++20, which provide a way to specify constraints on template parameters, ensuring that only types meeting certain criteria can be used with a template. The `requires` keyword is used to define these constraints. Compared to C++, Java supports compile-time polymorphism mainly through method overloading and generics. However, Java generics are implemented via type erasure, which removes type information after compilation, limiting their expressiveness compared to C++ templates. Java also lacks features such as template metaprogramming and `constexpr`-based compile-time computation. Combining compile-time and runtime polymorphisms leads to additional complexity and numerous corner cases.

5) *Others*: Besides the aforementioned core features, C++ provides additional keywords, syntax sugars, operators, etc., to facilitate OOP. For example, exception handling inside classes is enhanced with the `noexcept` keyword, which specifies whether a member function is guaranteed not to throw exceptions. Another example is *bitfields*, which enables bit-level memory storage within classes for low-level embedded system programming. In C++, bitfields are integrated with the class system and can even be inherited by derived classes, which is not possible in C. The memory layout of bitfields is specified by the ABI, which may involve bugs at the machine code level. Many other OOP languages, such as Java, do not support features like `noexcept` and *bitfields*.

B. An OOP-Related Bug

Listing 1 shows an OOP-related bug with the highest priority, GCC-94549². This bug-triggering program shows the challenge of triggering and debugging OOP-related bugs in C++ compilers. The program is valid, which defines a struct `base` (Line 2)

with two templated constructors. The first is permanently disabled using the constraint `requires false` (Line 3), while the second is always enabled due to `requires true` (Line 4). The struct `derived` inherits from `base` and explicitly imports its constructors via `using` (Line 6). In the `main` function, `derived` is instantiated with the argument 'G' (Line 7). By Line 4, the compiler should infer the type parameter `t` in `base` as `char` and correctly resolve the constructor constrained by `requires true`. However, GCC fails to accept this program due to incorrectly handling constructor inheritance.

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., declared by the newly introduced keyword `requires`) are 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 (i.e., Lines 9-10), we find the root cause of this bug is the incorrect branch condition, `flag_new_inheriting_ctors` (Line 9).

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. For the general fuzzers, CSmith [4] only supports C style struct (i.e., data records without member functions and inheritance), while YARPGen [5], [7] offers no OOP support. Fuzz4All [9], MetaMut [10] and GrayC [13] lack complex mutation operators for modifying advanced features such as `requires` and `using`. The front-end fuzzer CCOFT [8], partially supports OOP features such as *class*, but cannot modify inheritance relationships. 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.

III. METHODOLOGY

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

A. Bug Collection

Several popular C++ compilers exist, including GCC, LLVM, Microsoft Visual C++ (MSVC), and the Intel C++ Compiler (ICC). This study selects GCC and LLVM for the following reasons. First, MSVC and ICC are closed-source and lack public bug reporting, making them infeasible for analysis. Second, GCC and LLVM are widely used, high-impact, and open-source, providing accessible data for empirical research. Third, prior studies on C/C++ compiler bugs have also focused exclusively on GCC and LLVM [11], [12], [14].

²<https://gcc.gnu.org/bugzilla/showbug.cgi?id=94549>

TABLE I
BUGS SELECTED IN THIS STUDY

Compiler	Start	End	Reports	Remaining	Bugs	P1/S1	P2/S2
GCC	2017	2023	3401	1268	586	91	171
LLVM	2014	2023	1005	353	202	9	40
Total	-	-	4406	1621	788	100	211

Reports: the number of bugs only from the C++ components. **Remaining:** the number of remaining bugs after filtering via the keywords. **Bugs:** The number of finally studied bugs.

We focus on the compiler bugs that occur when handling OOP-related concepts. Since only the C++ components handle OOP-related features, we first collect the Bugzilla bug reports and GitHub issues from these components, then filter them using the OOP keywords `class` or `struct`, which represent the core data structures of C++ OOP programs. Note that `class` and `struct` in C++ have no intrinsic difference except for their default settings on member accessibility. The C++ OOP paradigm is centered around its definitions, making the keywords sufficient for filtering OOP-related bugs. 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 their corresponding patches. Therefore, we retain only those closed bug reports marked as `FIXED`, along with developers' commit messages that provide the commit hash of the version control system linking to the corresponding patches. Note that in both bug tracking systems, duplicated bug reports are marked as `DUPLICATE` and 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 [11]. 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 (i.e., P1 and P2), respectively, while 9 and 40 bugs of LLVM are classified as the top 2 severity levels (i.e., 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 [15]. We extract the

concepts and terminology from the widely recognized C++ textbook by *Bjarne Stroustrup* [15], the creator of C++, to construct our taxonomy. The first two authors conducted a pilot analysis to identify key terms and core concepts from the book, forming the foundation of our classification. 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*, *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 the results of 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 [16], [17], [18], [19], [20], [21], [22], we measure the inter-rater reliability via the Cohen's kappa coefficient (κ) [23], 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.

In Table II, the section of **Rater1 & Rater2** presents the label results on the remaining 688 bugs after analyzing the first 100 bugs. The columns **NN**, **NY**, **YN**, and **YY** present the number of bugs labeled by both raters. Specifically, **NN** indicates that both raters labeled a bug as not belonging to this category, **NY** and **YN** indicate cases where one rater marked it as belonging while the other did not, and **YY** indicates that both raters agreed that the bug belongs to this category. Based on the data from the four columns, we can calculate the κ scores via the formulae. 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, respectively. 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 conflicting bugs, and finally assigned a unique label to each bug. All the labeling processes are recorded in our repo.

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 (6.1%)	128 (16.2%)
	Class Scope	631	6	5	52	0.896		53	12	65 (8.2%)	
	Friends	680	2	1	11	0.878		9	6	15 (1.9%)	
Object	Ctor & Dtor	534	3	10	147	0.946	0.944	136	38	174 (22.1%)	263 (33.4%)
	Copy & Move	659	1	2	32	0.953		28	14	42 (5.3%)	
	Overloading	652	4	1	37	0.933		34	13	47 (6.0%)	
Inheritance	Inherited Member Access	670	0	3	21	0.931	0.925	21	9	30 (3.8%)	79 (10.0%)
	Multiple Inheritance	675	0	4	15	0.879		11	8	19 (2.4%)	
	Inherited Ctor & Dtor	665	3	1	25	0.923		25	5	30 (3.8%)	
Runtime Polymorphism	Override Control	685	2	0	7	0.874	0.888	8	3	11 (1.4%)	31 (3.9%)
	Casting	672	3	1	18	0.897		13	7	20 (2.5%)	
Compile-time Polymorphism	Template Alias	656	1	2	35	0.957	0.927	34	3	37 (4.7%)	261 (33.1%)
	Template Parsing	660	1	4	29	0.917		19	15	34 (4.3%)	
	Template Instantiation	589	12	6	87	0.891		77	35	112 (14.2%)	
	Constraint & Concepts	643	0	4	47	0.956		48	7	55 (7.0%)	
	Template Specialization	665	4	1	24	0.902		15	8	23 (2.9%)	
Others	—	665	2	2	25	0.923	0.923	20	6	26 (3.3%)	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.

C. Classifying and Labeling Symptoms, Root Causes, Fixes, 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 many existing studies [11], [17], [20]. 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 [17], [18], [20], [24]. 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 studies [16], [22], i.e., randomly select a set of statistically representative samples. 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) *Fixes*: Since all the bugs we studied are *fixed*, we extract the corresponding fixes from the version control system. Usually, a fixing commit consists of patches, change logs, and test suites. We only retain the modifications of source code as *fixes* and discard other parts.

4) *Compiler Options*: During issue triage for each bug report, compiler developers usually reduce the options to reproduce the bug in the discussion, we manually extracted the compilation commands from the discussion or the test-cases to obtain the options.

5) *C++ Standards*: For the programs that trigger the compiler bugs, we use a parser to analyze the code. Since we extract

the minimized triggering input programs, either provided by the bug reporter or requested by the compiler developers during issue triage, they typically retain only the essential constructs that expose the bug. Consequently, the relevant C++ standards can be accurately identified. Language features specific to a C++ standard are labeled accordingly upon detection. Note that this process involves multi-labeling, since a program may incorporate 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 comprising 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 protected members of other classes. Moreover, the `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. The bugs are caused by member access features when the compiler misinterprets visibility rules, resulting in incorrect access permissions. For instance, compilers may erroneously

```

1  /* -- The bug triggering program -- */
2  struct X {
3  private:
4      int i;
5  };
6  struct Y {
7      Y (int) { }
8  };
9  int main () {
10     Y ([ { X x; x.i = 3; return 0; } ()]); // Trigger
11 }
12 /* ---- The corresponding patch ---- */
13 cp_parser_end_tentative_firewall (parser, start,
14     lambda_expr);
15 + 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)

allow external access to private class members or block member functions from accessing their private members.

Listing 2 shows an invalid program of this category. The class X (Line 2) has a private member `i` (Line 4), which other classes should not access. The class Y (Line 6) has a constructor accepting a single integer as its parameter (Line 7). However, in the main function, GCC incorrectly allows accessing `x.i` (Line 10) in the anonymous lambda expression, which is used as the parameter of class Y's constructor. Referring to its patch (Lines 13-14), the bug is due to the missing API invocation for access check.

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 presents a valid program that causes GCC to enter an infinite loop. The program first declares an `extern` function `meminfo` (Line 2), then defines a struct with the same name (Line 3), deliberately creating a name conflict at the class-level scope. Inside the function `frob` (Line 4), the intended function call should resolve to the `extern` function rather than the struct (Line 5). However, due to the erroneous code implementation logic, GCC fails to handle this resolution correctly, leading to this bug. The fix for this issue is too long and thus is not presented here.

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 which causes GCC to crash. The template class A (Line 2) contains a nested template class B (Line 3), which declares all its

```

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)
9              const {return other.protected_member_;}
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)

```

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)

specializations as friends (Line 4). The inner class B has a protected data member `protected_member_` (Line 6) and a public function `method` (Line 8). The `method` function accesses the `protected_member_` of its parameter `other` (Line 8), which is permitted by the friend declaration, as it grants access to protected members across different specializations of B. The resolution of the friend causes the crash.

2) *Object*: In C++, an *object* is an instance of a *class* that defines its properties, including type, structure, layout, behavior, etc. All objects of the same class share the same properties. Conceptually, 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 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 presents an invalid program in this category that is incorrectly accepted by LLVM. The program defines three aggregate structs, A (Line 2), B (Line 3), and C (Line 4). In C++, aggregate types represent simple data structures that lack user-defined constructors, private or protected members, and inheritance, allowing them to be initialized using designated initializers (Line 5). However, this code should be rejected when compiled with the option `-pedantic-errors`, which strictly enforces compliance with the C++ standard and disallows all LLVM extensions. Following the C++ standard, the

```

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,
13   result);
13 + return cp_build_compound_expr (a, result, tf_error);

```

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

correct designated initialization should use explicit braces, i.e., `.a={0}`. The fix is substantial, as LLVM overlooked 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 presents an interesting case in this category, where the program became valid in C++17 but was invalid in earlier standards. The program is incorrectly rejected by the GCC compiler. The struct `immovable` (Line 2) has a default constructor (Line 3) and explicitly deletes the move constructor (Line 4), preventing any moving operations. The move constructor normally allows transferring ownership of an object from one reference to another, but deleting it ensures that objects of `immovable` cannot be moved. The struct `S` defines a function `f()` (Line 6), which returns a temporary `immovable` object initialized via its default constructor (i.e., `{}`). The function `g()` (Line 7) calls `S::f()` and directly returns its result (Line 8). Before C++17, this would require invoking the move constructor of `immovable`, which is deleted, causing a compilation error. However, starting from C++17, the standard enforces mandatory copy elision, requiring compilers to eliminate unnecessary copies and moves when a function directly returns another function's result. Consequently, the object returned by `S::f()` is constructed directly in the return slot of `g()`, avoiding any move operations. This makes the code valid in C++17 and later. The bug is caused by calling 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 incorrectly rejected by LLVM. The program defines a struct `bar` (Line 2) with an empty body, and a function `wot` (Line 3) with two parameters `x` and `y` of the type `bar`. Inside the function, it

```

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
10   ::IDNS_Ordinary);
10 + R.setFindLocalExtern(R.getIdentifierNamespace()&{
11   Decl::IDNS_Ordinary | Decl::IDNS_NonMemberOperator
12   });

```

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=
11   */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)

declares an overloading operator of `+` for two `bar` operands (Line 5), enabling the expression `x + y` (Line 6). However, LLVM rejects this due to failing to resolve non-member operator overloads (i.e., using an incorrect parameter when calling the API `setFindLocalExtern()` in Line 10).

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 program defines the base class `A` (Line 2) and its derived class `B` (Line 3). `A` declares a static function `f()` with no parameters (Line 2). Inside `B`, it first explicitly brings `A::f()` into its scope (Line 4), then overloads `f(int)` with an integer parameter (Line 5). The access of `f()` at Line 6 is rejected since the compiler failed to find the function in its parent class. The bug is due to the missing assignments of the variable controlling access.

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

```

1  /* -- The bug triggering program -- */
2  struct A { virtual void f() {} };
3  struct B : virtual A {};
4  struct C : virtual B, A {}; // Trigger
5  C c;
6  /* ---- The corresponding patch ---- */
7  - if (Layout.getVBaseOffsetsMap().count(NextBase)) {
8  + if (isDirectVBase(NextBase, RD)) {

```

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

```

1  /* -- The bug triggering program -- */
2  struct payload {};
3  struct base: private payload {
4      base(payload) {}
5  };
6  struct derived: base {
7      using base::base;
8  };
9  int main(){
10     derived demo(data); // Trigger
11 }
12 /* ---- The corresponding patch ---- */
13 - tree ctype = DECL_INHERITED_CTOR_BASE (fn);
14 - if (same_type_ignoring_top_level_qualifiers_p(ptype,
15     ctype))
16 + tree dtype = DECL_CONTEXT (fn);
17 + if (reference_related_p (ptype, dtype))

```

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

concepts like virtual inheritance and pure virtual functions. C++ compilers are prone to bugs when handling these features.

Listing 9 presents a valid program in this category, which causes LLVM to generate incorrect assembly code. The program defines a struct A with a virtual function `f()` (Line 2). Then, it defines struct B (Line 3), which inherits virtually from A, and struct C (Line 4), which inherits virtually from both B and A, forming a multiple inheritance hierarchy. In C++, virtual inheritance is used to resolve member conflicts in diamond inheritance, ensuring that only a single instance of the base A exists within C. Finally, the program instantiates an object of C (Line 5). LLVM fails to handle this case correctly due to the use of an 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 incorrectly rejects. The program first defines an empty struct `payload` (Line 2), then defines a subclass `base` which privately inherits from `payload` (Line 3). The struct `derived` inherits `base`, explicitly importing the constructor that has one `payload` type parameter of `base` via the `using` declaration (Line 7). However, LLVM rejects accessing this constructor (Line 10), due to the incorrect branch condition inside the `if` statement (Lines 13-14).

4) *Runtime Polymorphism*: To support runtime polymorphism, which incorporates type determination 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  /* -- The bug triggering program -- */
2  struct Res { };
3  struct A {
4      virtual Res &&foo() &&;
5  };
6  struct B : A {
7      Res &foo() && override; // Trigger
8  };
9  /* ---- The corresponding patch ---- */
10 - fail = cp_type_quals (base_return) != cp_type_quals
    (over_return);
11 + if (cp_type_quals (base_return) != cp_type_quals (
    over_return))
12 +     fail = 1;

```

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

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 presents an invalid program that GCC incorrectly accepts. The program first defines an empty struct `Res` (Line 2), then defines a base class A (Line 3) and its derived class B (Line 6). The base A defines a virtual function, which returns an rvalue reference `Res &&` (Line 4), which restricts that it can only be invoked by an rvalue instance of A. The class B overrides `foo()` but changes the return type to an lvalue reference `Res &` (Line 7). The overriding is invalid since the return in the derived class must be a pointer or reference to the same type or a more derived type of the one in the base class if they are not identical, whereas `Res&` and `Res&&` have no such relationship. The bug is due to the incorrect assignment of the flag variable.

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 resolve the type information at runtime correctly.

Listing 12 presents a valid program that triggers a crash in LLVM. The program first defines a base class A with a declaration of its virtual destructor (Line 2). The class B is derived from A and marked as `final`, forbidding any class inheriting from it (Line 3). The destructor of B is also virtual, and marked as `=default`, meaning the compiler automatically generates it. The function `cast` attempts to downcast from `A*` to `B*` using runtime casting `dynamic_cast` (Line 4). However, LLVM failed to correctly handle `dynamic_cast` due to overlooking this special case. Specifically, the code passes the incorrect check logic and triggers the compiler assertion failure. The triggered assert enforces conditions under which virtual table (*vtable*) linkage can be queried (i.e., `assert(IsInNamedModule || def ||.)`). The patch (Lines 6-10) bypasses the assertion by restricting *vtable* access for `final` classes, which cannot be inherited and therefore do not require dynamic linkage queries for their *vtables*.

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

```

1  /* -- The bug triggering program -- */
2  struct A { virtual ~A(); };
3  struct B final : A { virtual ~B() = default; };
4  B *cast(A *p) { return dynamic_cast<B*>(p); } //
    Trigger
5  /* ---- The corresponding patch ---- */
6  + if (DestRecord) {
7  +   auto *DestDecl = DestRecord->getAsCXXRecordDecl();
8  +   if (DestDecl->isEffectivelyFinal())
9  +     Self.MarkVTableUsed(OpRange.getBegin(), DestDecl);
10 + }

```

Listing 12. Casting Bug LLVM-64088 (Crash)

```

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

```

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

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 recursively defined template types, where a template alias refers to itself. Bugs arise when C++ compilers fail to parse or resolve these alias declarations correctly.

Listing 13 presents an invalid program that triggers a crash in GCC. The program defines a class template `A` with an unused type parameter (Line 2) and a template alias `C` (Line 3), which discards the first type parameter and instantiates `A` using only the second type parameter `B`. The instantiation on Line 4 is invalid, leading to a compiler crash when attempting to instantiate `C`. The bug occurs due to the compiler failing to handle this specific incorrect case properly.

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 causes GCC to crash. The program defines a template struct `zq` with one type parameter `XK` (Line 2). The member function declaration of `ky` is marked with `noexcept ([ ] {})` (Line 3), which determines whether the empty lambda expression can throw exceptions. In this case, the lambda is non-throwing, so `noexcept` is `true`. The compiler crashes due to a missing case for handling lambda expressions within the `noexcept` specifier inside templates.

③ *Template Instantiation*: The instantiation phase of C++ template classes produces concrete code definitions based on

```

1  /* -- The bug triggering program -- */
2  template <typename XK> struct zq {
3  +   void ky () noexcept ([ ] {}); // Trigger
4  };
5  /* ---- The corresponding patch ---- */
6  + else if (t == error_mark_node)
7  +   dependent_p = false;
8  + else
9  +   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)

the provided template arguments. This process involves several intricate steps, including name binding (i.e., associating a type 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 presents an invalid program that LLVM incorrectly accepts. The program defines a function template `f<T>()` (Line 2), which contains a nested struct `S` (Line 3) with a member function `g()` that accesses the member `error` of the template parameter (Line 4). The explicit template instantiation (template void `f<int>()`) (Line 7) should result in a compilation error because `T::error` is not a valid member of `int`. However, LLVM fails to detect this error due to incorrect handling of template instantiation. The bug fix is extensive and omitted here.

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, compiler implementations often contain errors in these features.

Listing 16 presents an invalid program that LLVM incorrectly accepts. The program first defines a concept `false_` which is set to be constantly `false` regardless of the template parameters (Line 2). Since concepts are used for constraints, this essentially disables any valid instantiation. Then it defines a template struct `s` (Line 3), which has a static function `f()` with a constrained parameter (Line 4). However, `false_ <char>` expands to `false`, so `f()` is never valid. Finally, the program attempts to illegally invoke `f` (Line 6). The bug is due to the missing handling of constraints during instantiation. The bug fix is extensive and omitted here.

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 prone to bugs when handling these features, especially the mechanisms for partial specialization.

```

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

```

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

```

1  /* -- The bug triggering program -- */
2  enum EnumType { A };
3  class MyClass {
4      template<int I> inline void set(int v);
5      template<EnumType P> inline void set(int v, int I);
6      // Trigger
7  };
8  template<> void MyClass::set<A>(int v, int I) { }

```

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

Listing 17 presents a valid program that LLVM incorrectly rejects. The program first defines an enumeration, EnumType, with a single enumerator, A. It then declares a class, MyClass, which contains two function templates (Line 3). The first function template, set<int I>(int v), takes a single integer template parameter I (Line 4), while the second, set<EnumType P>(int v, int I), takes a template parameter of type EnumType (Line 5). The second function template is specialized with an empty body (Line 7). LLVM incorrectly rejects this code due to overlooking this special case of using an enumeration value as a type parameter during template specialization. The bug fix is overlong and omitted here.

6) *Others*: The features can not be classified into any other categories. As mentioned in Section II, C++ provides abundant features to facilitate OOP, including exception handling, machine-level memory management, etc.

Listing 18 shows a valid program that causes GCC to crash. The program first defines a struct X (Line 2) with a default destructor (Line 3). Then it defines a struct Y (Line 5) with an empty destructor, which is marked as noexcept(false), allowing exceptions to be thrown during destruction (Line 6). The function foo (Line 8) normally returns an X object (Line 10) but enters the catch block if any exception occurs (Line 11), where catch(...) handles any type of exception. Within the catch block (Lines 12-13), a temporary Y object is created and immediately destroyed (Line 12), followed by returning another X object (Line 13). GCC crashes due to the erroneous code logic when processing exception handlers.

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.

```

1  /* -- The bug triggering program -- */
2  struct X {
3      ~X() {}
4  };
5  struct Y {
6      ~Y() noexcept(false) {} // Trigger
7  };
8  X foo() {
9      try {
10         return {};
11     } catch (...) {
12         Y{};
13         return {};
14     }
15     try {} catch (...) {}
16 }
17 /* ---- The corresponding patch ---- */
18 - while (TREE_CODE (tsi_stmt (iter)) == DECL_EXPR)
19 + while (!tsi_end_p (iter)
20 +      && TREE_CODE (tsi_stmt (iter)) == DECL_EXPR)
21     tsi_next (&iter);
22 - gcc_assert (!tsi_end_p (iter));
23 + if (tsi_end_p (iter))
24 +     return;

```

Listing 18. Others Bug GCC-113088 (Crash)

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 twofold. First, C++ offers various initialization methods, such as default initialization, direct initialization, value initialization, list initialization, aggregate initialization, etc. For example, there are numerous 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 features introduce many bugs is twofold. 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 errors in branching conditions.

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, FIXES, 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 [18], [19], [20], [24].

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 traces.

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 compilation 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:* Fig. 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 Invalid* 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 [20], [24] 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.

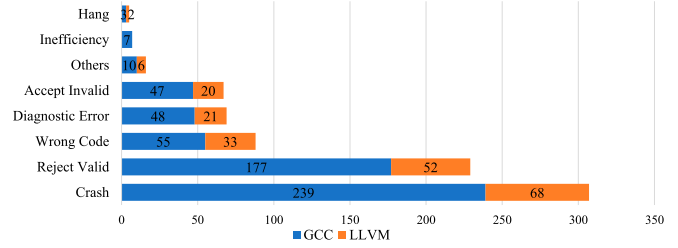


Fig. 1. Bug distribution by symptoms.

Finding 3: Although *Crash* is the most common symptom among all OOP-related bugs, 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 [17], [18], [20], [24].

① *API Misuse:* developers misunderstand APIs, including: 1) calling an incorrect API; 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:* Fig. 2 shows the root cause distribution, from which we can find that 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 overlooking 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*.

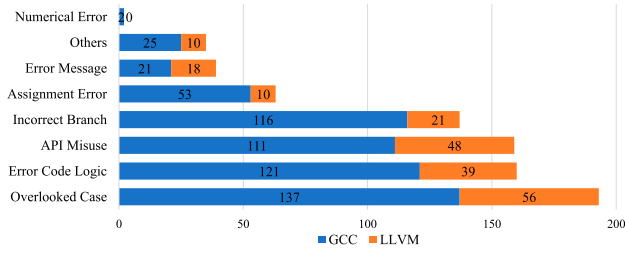


Fig. 2. Bug distribution by root cause.

TABLE III
CUMULATIVE DISTRIBUTION OF BUGS THROUGH TIME

Metric	Compiler	Subm.-to-Disc.	Disc.-to-Fix	Subm.-to-Fix
20%	GCC	303	4	308
	LLVM	231	1	212
40%	GCC	769	20	801
	LLVM	761	7	777
60%	GCC	1701	89	1742
	LLVM	1308	36	1349
80%	GCC	3835	262	3846
	LLVM	1871	165	1995
100%	GCC	10743	1833	10745
	LLVM	4898	1492	4922
Mean	GCC	2082	183	2121
	LLVM	1202	147	1241
Median	GCC	1237	45	1303
	LLVM	980	20	1026
SD	GCC	2220.9	313.0	2238.3
	LLVM	1017.9	290.4	1086.0

C. RQ4: Characteristics of Fixes

To gain insights into the complexity of fixing OOP-related compiler bugs, we analyze each bug's lifespan, patch size, and code locations.

1) *Duration of Bugs*: The duration of a bug can reflect the difficulty of finding and fixing it. For each bug, we record three key timestamps: the *submission time*, when the buggy code is first submitted to the compilers' repo; the *discovery time*, when it is reported in the bug-tracking system; and the *fix time*, when its patch is submitted. We track the following time intervals for each bug: *Submission-to-Discovery* (i.e., the duration the bug remains hidden), *Discovery-to-Fix* (the duration for fixing), and *Submission-to-Fix* (the total lifespan of the bug). We sorted all bugs in ascending order of duration and recorded the values at the 20%, 40%, 60%, 80%, and 100% percentiles. Additionally, we calculated the mean, median, and standard deviation (SD) of the durations, as shown in Table III.

For the *Submission-to-Discovery* interval, at the 20% percentile, GCC has a time of 303 days, while LLVM's time is much lower at 231 days. As the percentile increases, the time for both GCC and LLVM grows significantly, reaching the maximum of 10743 days (more than 29 years) for GCC³ and 4892 days (more than 13 years) for LLVM⁴ at the 100% percentile. For the *Discovery-to-Fix* interval, the values are consistently lower across both compilers compared to the *Submission-to-Discovery* times. At the 20% percentile, GCC and LLVM require 4 and 3 days, respectively, indicating that for 1/5 of the

³The bug is GCC-110809, whose buggy code was committed in 1994 and discovered in 2023.

⁴The bug is LLVM-62174, whose buggy code was committed in 2009 and discovered in 2023.

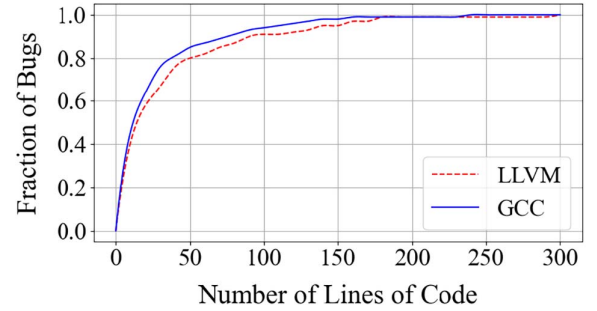


Fig. 3. Cumulative distribution of lines of code in a fix.

TABLE IV
LINES OF CODE MODIFICATION IN BUG FIXES

Compiler	Mean	Median	SD	Min	Max
GCC	27.0	12	37.9	1	277
LLVM	34.4	14	46.5	1	292

bugs, resolution is quick once bugs are discovered. For the *Submission-to-Fix* interval, which combines both the discovery and fix durations, shows an overall upward trend. Notably, the *Submission-to-Discovery* interval is much longer than the *Discovery-to-Fix* interval, with many bugs remaining hidden for over a decade before being reported. This highlights the need for more advanced compiler testing approaches to detect latent OOP-related bugs.

Sun et al. [11] also analyzed the *Discover-to-Fix* duration of general bugs in GCC and LLVM, reporting median durations of 28 and 7 days, respectively. In comparison, the median durations for OOP-related bugs in Table III are 45 and 20 days, respectively. This means OOP-related bugs require 60.7% more time to discover and 185.7% more time to fix than general bugs. These findings indicate that both GCC and LLVM require significantly more effort to resolve OOP-related bugs, suggesting their increased complexity and the challenges involved in diagnosing and fixing them.

Finding 5: On average, an OOP-related bug takes 1856 days to be discovered after submission, but only 174 days to be fixed once discovered. These durations are significantly longer than those of general bugs in both compilers.

2) *Size of Fixes*: The size of fixes (i.e., the lines of modified code or the number of modified functions) partially indicates the difficulty of patching the program. Therefore, we analyze the size of the fixes for OOP-related bugs. We analyze the fixing commit of each bug, extract all modified files, and retain the modifications in the source code. We calculate the modified lines of code and functions.

Shown as Fig. 3, for both compilers, 80% of bug fixes are fewer than 50 lines of code. For GCC, 45.6% of bugs require at most 10 lines of fixes, while for LLVM, the proportion is 41.5%. Table IV shows the statistics of the lines of code in fixes. For the

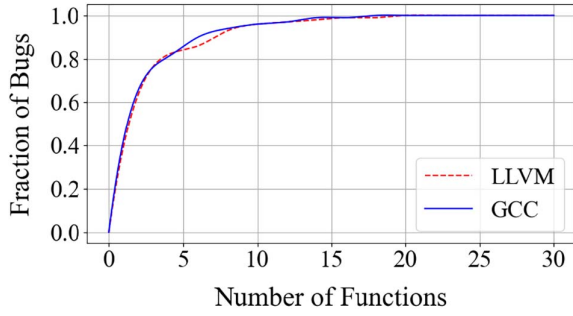


Fig. 4. Cumulative distribution of functions in a fix.

TABLE V
NUMBER OF FUNCTIONS MODIFIED IN BUG FIXES

Compiler	Mean	Median	SD	Min	Max
GCC	2.9	2	3.2	1	29
LLVM	3.1	2	3.4	1	20

lines of code in a fix, the average number is 27.0 for GCC and 34.4 for LLVM, while the median is 12 and 14, respectively.

Fig. 4 presents the number of patched functions in a fix. For GCC, 46.6% of bugs only involve 1 function in fixes, while for LLVM, the proportion is 43.5%. For both compilers, around 85% of fixes require at most 5 functions. Table V presents the statistical data, showing that for both compilers, the mean is around 3, while the median is 2.

Referring to the study by Sun et al. [11], the fix size of OOP-related bugs is consistent with that of general bugs, indicating that most fixes involve only minor modifications.

Finding 6: Most fixes in OOP-related bugs require only minor modifications.

3) *Locations of Fixes:* We counted all the files modified in the patches and identified the top 10 most frequent files in GCC and LLVM, as shown in Table VI, where the left part shows the files of GCC and the right part shows the files of LLVM.

Overall, these files mostly involve semantic analysis, parsing, type checking, templates, overloading, and initialization, aligning with our taxonomy findings. Notably, when compared to the general bug distributions reported in the prior study on the general bugs of both compilers by Sun et al. [11], the top 10 lists of OOP-related bugs exhibit significant differences. For instance, files such as `cp/constexpr.c` and `class.c` from GCC, as well as `SemaInit.cpp` and `SemaLookup.cpp` from LLVM, which are highly relevant to OOP features, are absent from the general buggy file lists. In contrast, files associated with optimization and code generation, such as `fold-const.c` from GCC and `X86ISelLowering.cpp` from LLVM, which appear in the general buggy file lists, are not present in the list of OOP-related bugs. The difference partially emphasizes the necessity of studying OOP-related bugs.

Finding 7: The top 10 most frequently modified files for OOP-related bugs primarily correspond to features categorized in our taxonomy. While there is some overlap with the top 10 buggy files of general bugs in both compilers, OOP-related bugs exhibit distinct characteristics.

D. RQ5: The Correlation between OOP Features and Symptoms

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.

We investigate whether a certain correlation exists between the bug-triggering OOP features and the bug symptoms of compilers. Table VII 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 serve as a straightforward test oracle, while differential testing may be more effective for identifying *Reject Valid* bugs.

Finding 8: Across different OOP features, compiler bugs exhibit a diverse range of symptoms, with *Crash* and *Reject Valid* being the most frequent.

E. RQ6: Bug Triggering Compiler Options

In addition to source code, compiler options are another crucial input that must be appropriately configured to trigger bugs [25], [26], [27]. To investigate the impact of compiler options on detecting OOP-related bugs in C++ compilers, we analyze user-submitted options extracted from developer discussions. We believe that the options used in the simplified examples provided by issue reporters and developers have already been minimized. Consequently, we directly extract these options used by developers from the bug reports. Table VIII illustrates the number of compiler options required by GCC and LLVM to trigger the bugs. The results reveal that for GCC, 58.2% of bugs are triggered without any options, and 25.9% require only

TABLE VI
TOP 10 BUGGY SOURCE FILES OF GCC AND LLVM

(a) GCC		(b) LLVM	
GCC File	# Description	LLVM File	# Description
cp/pt.c	136 parameterized types (templates) handler	SemaDeclCXX.cpp	24 semantic analysis for declarations for C++
cp/call.c	116 function invocations	Sema.h	21 header for semantic analysis and AST building
cp/decl.c	113 declarations and variables	SemaExpr.cpp	18 semantic analysis for general expressions
cp/constexpr.c	110 constexpr evaluation	SemaDecl.cpp	18 semantic analysis for general declarations
cp/parser.c	68 the front-end parser	SemaExprCXX.cpp	17 semantic analysis for C++ expressions
cp/cp-tree.h	67 parsing and type checking	SemaTemplate.cpp	16 semantic analysis for templates
cp/tree.c	48 language-dependent node constructors for parse phase	SemaInit.cpp	13 semantic analysis for initializers
cp/class.c	44 classes and their objects	SemaTemplateInstantiateDecl.cpp	12 C++ template declaration instantiation
cp/typeck.c	43 type checking	SemaOverload.cpp	12 C++ overloading
cp/typeck2.c	43 report error messages and some front-end optimizations	SemaLookup.cpp	12 name lookup

TABLE VII
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.

TABLE VIII
THE DISTRIBUTION OF OPTION NUMBER

# of Opt.	GCC	LLVM	SUM
0	341 (58.2%)	112 (55.4%)	453 (57.5%)
1	152 (25.9%)	51 (25.2%)	203 (25.8%)
2	51 (8.7%)	14 (6.9%)	65 (8.2%)
3	14 (2.4%)	2 (1.0%)	16 (2.0%)
≥ 4	28 (4.8%)	23 (11.4%)	51 (6.5%)

one option. Similarly, for LLVM, 55.4% of bugs do not require options, while 25.2% are triggered by a single option. Overall, 453 bugs (i.e., 57.5%) require no compiler options, while 203 bugs (i.e., 25.8%) require only a single option. Among the cases requiring a single option, the majority specify the language standard. As discussed later, many bug-triggering programs utilize features introduced in newer language standards, which often require setting the corresponding standard. This suggests that a simple strategy for generating compiler options, such as testing one option at a time, could effectively identify a significant portion of the bugs. However, it is worth noting that a non-negligible proportion (i.e., 6.5%) of bugs require at least four options to be triggered.

Finding 9: To trigger OOP-related bugs in C++ compilers, 57.5% of the bugs require no compiler options, while 25.8% require only a single 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`).

Fig. 5 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`. Additionally, in both compilers, *Language Standard* options are the most commonly used among the top 12 options. These findings support the earlier observation regarding the number of options and further highlight the importance of correctly setting the standard in certain cases.

Finding 10: *Language Standard* options are the most frequently used to trigger OOP-related bugs.

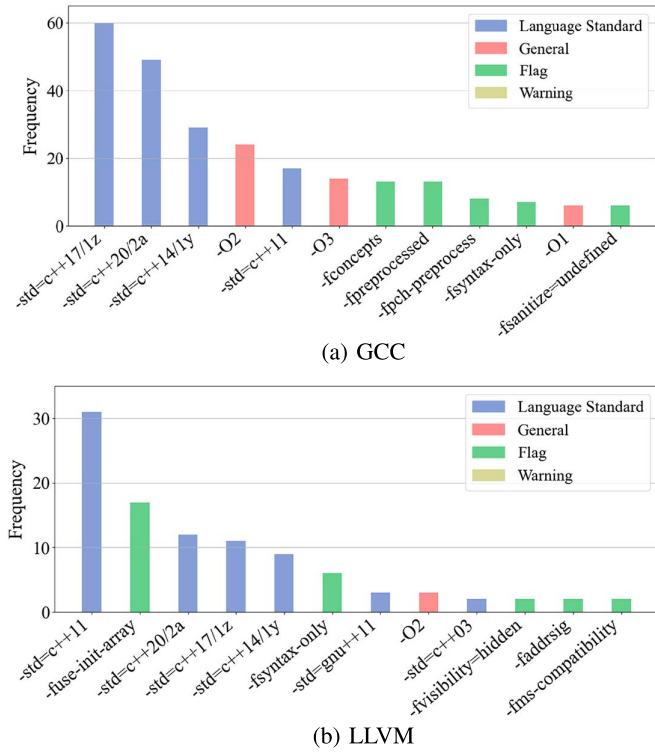


Fig. 5. Most frequent options.

From Table VIII, we observe that a notable portion (i.e., 6.4%) of bugs require at least four options. To gain deeper insights, we calculate the frequency of each option and rank them in descending order, as shown in Fig. 6. We also intend to examine whether the distribution follows the *80/20 rule (Pareto Principle)*, i.e., whether the top 20% of options contribute to 80% of total usage. Among the top 20% most frequently used options, GCC options account for 68.8% of total usage, while LLVM options account for 68.2%. The results reveal a clear *Long Tail Effect*, suggesting that although a few options are frequently used (with the top 20% of options accounting for approximately 70% of total usage), the remaining options still play a significant role in triggering compiler bugs. This is consistent with the existing study [28], which highlights the importance of option combinations in triggering compiler bugs. Additionally, we note that the prevalence of bugs requiring few options may be due to inadequate exploration of option combinations. It is conceivable that there may be subtle bugs that require a combination of many options that are under-explored.

Finding 11: The options in both GCC and LLVM exhibit a clear *Long Tail Effect*. Although the top 20% of options contribute to approximately 70% of total usage, the remaining options are non-negligible.

F. RQ7: The Commonality Across C++ Compilers

Following existing studies [18], [20], [29], we adopt the Spearman correlation coefficients to evaluate the commonality,

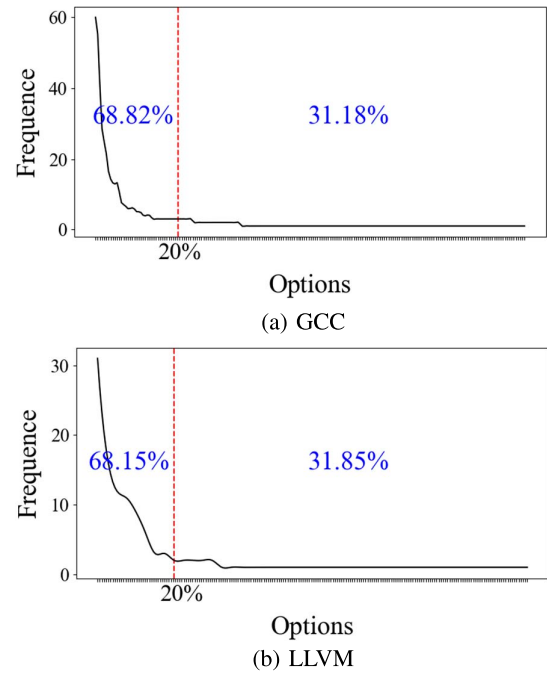


Fig. 6. Long tail effect in compiler option distribution.

TABLE IX
SPEARMAN CORRELATION COEFFICIENTS OF
GCC AND LLVM

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

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 options to trigger compiler bugs, shown in Table IX. We can find that all the Spearman coefficients are larger than 0.8 and the *p*-values are less than 0.05, indicating GCC and LLVM have a high correlation with high confidence.

Finding 12: GCC and LLVM have significant commonalities in bug types, symptoms, and the number of options to trigger compiler bugs.

G. RQ8: 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 Fig. 7. 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 newly introduced features may interact with existing OOP features in unexpected

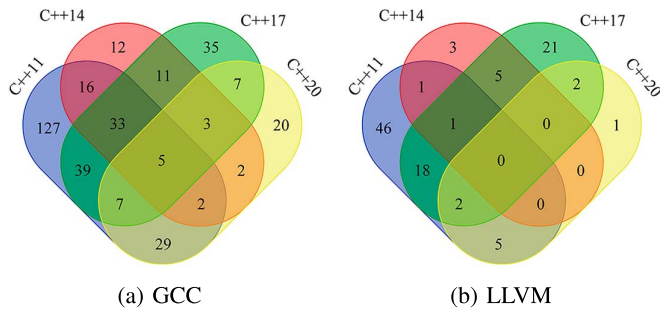


Fig. 7. Involving C++ standards of bug-triggering programs.

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

VI. DISCUSSION

In the following, we first discuss the implications of our findings, then discuss the threats to validity.

A. Implications

Based on our findings, we discuss the implications for enhancing compiler testing, debugging, engineering, and language design.

1) *New Program Generation and Mutation Strategies for Fuzzers:* According to **Findings 1** and **2**, we find that about 66% of bugs in C++ compilers are related to *Object* and *Compile-time Polymorphism*. According to **Finding 7**, the top 10 buggy source files of both compilers also correspond to these features. According to **Finding 13**, many bugs are due to the introduced OOP features of the new standards. However, to the best of our knowledge, very few existing fuzzers provide support for these features. Moreover, according to **Finding 5**, OOP-related bugs tend to remain undetected for long periods, with an average discovery delay of 2082 days in GCC and 1202 days in LLVM. This highlights the need for increasing research efforts to enhance OOP-related bug detection approaches. Thus, we suggest that corresponding generation or mutation strategies be developed and integrated into existing fuzzers to address the OOP-related features 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 **Findings 9** and **10**, compiler fuzzing can be effectively conducted using a single language standard option or even without any compiler options. According to **Finding 12**, we can use the same fuzzing strategies for GCC and LLVM.

2) *Oracle Enhancement for Automated Compiler Testing:* According to **Findings 3** and **8**, we find that while *Crash* serves as a strong oracle in compiler fuzzing [11], [20] since it

```
1 // The _Nonnull keyword is an LLVM-specific extension
2 int fetch(int * _Nonnull ptr)
3 {
4     ...
5 }
```

Listing 19. Limitations of Differential Testing Due to Compiler-Specific Extensions

is the most common symptom, using it as the sole oracle means that a significant portion (i.e., 61%) 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*.

Differential testing may not work in the following cases. First, both compilers accept an invalid program or reject a valid program, leading to missed bugs. Second, compilation differences may arise due to compiler-specific extensions, leading to false positives in differential testing. For example, Listing 19 shows the keyword extension of LLVM, `_Nonnull`, which enforces pointer nullness checks but is not recognized by GCC, resulting in different compilation results, but that is not a true bug. Third, compilation differences may arise in cases of undefined behavior or in areas where the standard lacks strict specifications, leading to false positives. Therefore, discovering new oracles for C++ 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 6**, fixes for OOP-related bugs are relatively small, suggesting that automated program repair could effectively address them. Additionally, **Finding 7** suggests compiler developers should prioritize investigating the top 10 most buggy files. According to **Finding 8**, 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 Options When Testing Compilers:* According to **Finding 9**, 57.5% of bugs require no compiler options, while 25.8% require only a single option. This suggests that using a minimal set of options is an effective strategy when test schedules are constrained by a limited time budget. Setting no options or a single predefined option is acceptable in practical automated compiler testing. According to **Findings 10** and **12**, we can employ a proper *Language Standard* option for both compilers. According to **Finding 11**, if time permits, we should endeavor to explore as many option combinations as possible.

5) *Cautionary Guide for Language Designers and Compiler Developers:* According to the findings derived from the taxonomy, i.e., **Findings 1** and **2**, we provide a cautionary guide for language designers and compiler developers to mitigate the risks associated with OOP-related bugs in C++ compilers. Specifically, when designing new language features, it is crucial to consider the challenges of compiler implementation and potential conflicts with existing features. For instance, the code shown in Listing 6, whose validity varies across different C++ standards, easily leads to compiler bugs. Likewise, compiler

developers should pay special attention to object- and template-related features, ensuring that corner cases are handled as thoroughly as possible.

B. Threats to Validity

Similar to existing empirical studies, our study is potentially subject to the threats introduced by manual inspections, including identifying OOP-related features, symptoms, root causes, fixes, options, and involved C++ standards in bug-triggering inputs. To reduce the threat, we referred to existing empirical studies on bugs [18], [20], [22], [24], [30], [31], [32], 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 [11], [12], [14], 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.

The biases in bug selection and analysis may pose a threat to the validity of our findings. For example, Findings 3 and 8 indicate that *Crash* is the most common symptom both overall and across most features. This is partly because crashes are relatively straightforward to detect and observe, which may skew the frequency of this symptom compared to others that are more difficult to identify. Another potential source of bias arises from the findings related to C++ standards. While C++11 and C++20 are indeed the two most influential versions of modern C++, C++14 and C++17 introduce relatively fewer changes. Since this study was conducted during the ongoing development of C++20, some latent bugs in that version may not have been fully discovered at the time of the study, potentially introducing bias into our conclusions regarding the impact of different C++ standards. Additionally, we directly adopt the bug-triggering input programs and options provided by the developers, which may not be theoretically minimized. This could introduce variability in the analysis. However, the pipeline for processing bugs in both compilers is well-organized, and developers typically minimize input programs and options before initiating debugging. Consequently, while this threat exists, its impact is likely limited.

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 taxonomy, 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

TABLE X
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

of constructors and destructors; (3) Adding an empty base class; (4) Replacing object initialization with another semantic equivalent form. The first operator is derived from Finding 1, which confirms *Template Instantiation* frequently triggers bugs and can be activated via expanding useless type parameters. The second operator is derived from *Member Access* of the taxonomy, changing access control modifiers may confuse compilers. The third operator is derived from *Inheritance*, where the hierarchical structure of classes may trigger bugs. The fourth operator is derived from Finding 1, which reveals *Ctor & Dtor* are the most common secondary category within the taxonomy. 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 X, 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*”⁵.

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. Understanding Compiler Bugs

The most relevant related work consists of empirical studies on the characteristics of bugs in C/C++ compilers, primarily focusing on GCC and LLVM. Sun et al. conducted a quantitative analysis of the overall bug distribution in these compilers [11]. Their study analyzes the general bugs of both compilers, identifying the C++ component as the most bug-prone. Sun et al. also investigated the characteristics of warning defects in both compilers [14]. Zhou et al. further studied optimization-related bugs and their impact [12]. Zhong studied the impact of compiler bugs in real development [33]. However, none of

⁵<https://github.com/llvm/llvm-project/issues/81731>

these studies specifically analyze OOP-related bugs or explore the relationship between compiler bugs and language features.

For compilers, except empirical study of bug characteristics, numerous works have focused on the C/C++ compiler testing methodologies. For example, Chen et al. [34] empirically compared EMI [6] and random differential testing [4]. Marcozzi et al. quantitatively and qualitatively studied the impact of the compiler bugs found by compiler fuzzers [2]. Zhong investigated the role of seed programs in compiler fuzzing [35], revealing that bug reports can serve as a valuable supplement to seed programs. These studies mainly focus on testing approaches, rather than the bug characteristics. Our study can provide insight for these studies.

Beyond C and C++, existing studies have explored a diverse range of programming languages and infrastructures, including JVMs [36], WebAssembly compilers [19], deep learning compilers [20], [37], Markdown [38], Python [17], [39], Solidity [40], Rust [41], etc. Compared to these works, our study focuses specifically on C++ and provides deeper insights into its unique language features and compiler challenges.

B. Understanding Bugs in General Software

The characteristic of bugs varies with the type of their host software. Existing studies have covered a wide range of different types of software. For instance, open-source software [22], [42], [43], operating systems [44], quantum computing platforms [24], deep learning frameworks [18], [45], deep learning models [31], SMT solvers [46], and Android apps [47]. Although all bugs may have some commonalities in root cause or fix patterns, the bugs they studied are significantly different from ours. Our study focuses on OOP-related bugs of C++ compilers.

C. Compiler Fuzzers

Compiler fuzzers generate programs to test compilers, which have detected thousands of bugs and attracted much academic [1] and industrial attention [48].

Existing compiler fuzzers can be classified into generation-based and mutation-based. Generation-based approaches generate programs from scratch by pre-defined grammar rules. CSmith [4] is the pioneering generation-based approach by using a simple subset of C language grammar rules, which is the basis of many existing compiler fuzzers [27], [49], [50], [51], [52], [53], [54]. 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 [9], [10], [55], [56]. Moreover, many generation strategies are designed for other programming languages [57], [58], [59], [60], [61], [62], [63]. Mutation-based approaches apply mutation operators to seed programs and transform them into new programs. EMI [6] is a widely recognized mutation-based approach for generating program variants. It produces runtime-equivalent variants by injecting code into regions that remain unexecuted under given inputs. For C compilers, the mutation-based fuzzers play an important role [8], [13], [28], [29], [35],

[64], [65], [66]. Additionally, mutation-based approaches are widely used in testing JVM [67], [68], [69], [70], [71], [72], [73], [74], [75], deep learning compilers [26], [76], MLIR compilers [51], [77], and Rust [78]. Rather than mutating source code, Bendrissou et al. propose mutating grammar [79]. Moreover, Li et al. proposed a novel approach combining the code from generation-based and mutation-based approaches with real code [80]. These compiler fuzzers lack specified support for OOP features, and our study could provide valuable insights.

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 13 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 et al., "A survey of compiler testing," *ACM Comput. Surv.*, 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?" *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 1–29, 2019.
- [3] T. Hoare, "The verifying compiler: A grand challenge for computing research," *J. ACM*, 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 *Proc. 32nd ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2011, pp. 283–294.
- [5] V. Livinskii, D. Babokin, and J. Regehr, "Random testing for C and C++ compilers with YarpGen," in *Proc. ACM Program. Lang.*, 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," in *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, pp. 1826–1847, 2023.
- [8] H. Tu et al., "Detecting C++ compiler front-end bugs via grammar mutation and differential testing," *IEEE Trans. Rel.*, vol. 72, no. 1, pp. 343–357, Mar. 2023.
- [9] C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang, "Fuzz4all: Universal fuzzing with large language models," in *Proc. IEEE/ACM ICSE*, 2024, pp. 1–13.
- [10] X. Ou, C. Li, Y. Jiang, and C. Xu, "The mutators reloaded: Fuzzing compilers with large language model generated mutation operators," in *Proc. ASPLOS*, 2024.
- [11] C. Sun, V. Le, Q. Zhang, and Z. Su, "Toward understanding compiler bugs in GCC and LLVM," in *Proc. 25th Int. Symp. Softw. Testing Anal.*, 2016, pp. 294–305.
- [12] Z. Zhou, Z. Ren, G. Gao, and H. Jiang, "An empirical study of optimization bugs in gcc and llvm," *J. Syst. Softw.*, vol. 174, 2021, Art. no. 110884.
- [13] K. Even-Mendoza, A. Sharma, A. F. Donaldson, and C. Cadar, "Gray: Greybox fuzzing of compilers and analysers for C," pp. 1219–1231, Jul. 2023.
- [14] C. Sun, V. Le, and Z. Su, "Finding and analyzing compiler warning defects," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 203–213.
- [15] B. Stroustrup, "The C++ programming language," 4th ed. Addison-Wesley Professional, 2013.
- [16] F. Macklon, M. Viggiano, N. Romanova, C. Buzon, D. Paas, and C.-P. Bezemer, "A taxonomy of testable HTML5 canvas issues," *IEEE Trans. Softw. Eng.*, vol. 49, no. 6, pp. 3647–3659, Jun. 2023.
- [17] D. Liu, Y. Feng, Y. Yan, and B. Xu, "Towards understanding bugs in python interpreters," *Empirical Softw. Eng.*, vol. 28, no. 1, p. 19, 2023.
- [18] J. Chen, Y. Liang, Q. Shen, J. Jiang, and S. Li, "Toward understanding deep learning framework bugs," *ACM Trans. Softw. Eng. Method.*, vol. 32, no. 6, pp. 1–31, 2023.

- [19] A. Romano, X. Liu, Y. Kwon, and W. Wang, "An empirical study of bugs in webassembly compilers," in *Proc. 36th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Piscataway, NJ, USA: IEEE Press, 2021, pp. 42–54.
- [20] Q. Shen, H. Ma, J. Chen, Y. Tian, S.-C. Cheung, and X. Chen, "A comprehensive study of deep learning compiler bugs," in *Proc. 29th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2021, pp. 968–980.
- [21] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella, "Taxonomy of real faults in deep learning systems," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.*, 2020, pp. 1110–1121.
- [22] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," *Empirical Softw. Eng.*, vol. 19, pp. 1665–1705, Dec. 2014.
- [23] A. J. Viera et al., "Understanding interobserver agreement: The kappa statistic," *Fam Med.*, vol. 37, no. 5, pp. 360–363, 2005.
- [24] M. Paltenghi and M. Pradel, "Bugs in quantum computing platforms: An empirical study," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA1, pp. 1–27, 2022.
- [25] H. Jia et al., "Detecting JVM JIT compiler bugs via exploring two-dimensional input spaces," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2023, pp. 43–55.
- [26] C. Li, Y. Jiang, C. Xu, and Z. Su, "Validating JIT compilers via compilation space exploration," in *Proc. 29th Symp. Operating Syst. Princ.*, 2023, pp. 66–79.
- [27] J. Chen, C. Suo, J. Jiang, P. Chen, and X. Li, "Compiler test-program generation via memoized configuration search," in *Proc. 45th Int. Conf. Softw. Eng.*, 2023.
- [28] H. Jiang, Z. Zhou, Z. Ren, J. Zhang, and X. Li, "CTOS: Compiler testing for optimization sequences of LLVM," *IEEE Trans. Softw. Eng.*, vol. 48, no. 7, pp. 2339–2358, Jul. 2022.
- [29] Y. Tang, H. Jiang, Z. Zhou, X. Li, Z. Ren, and W. Kong, "Detecting compiler warning defects via diversity-guided program mutation," *IEEE Trans. Softw. Eng.*, vol. 48, no. 11, pp. 4411–4432, Nov. 2022.
- [30] J. Garcia, Y. Feng, J. Shen, S. Almanee, Y. Xia, and Q. A. Chen, "A comprehensive study of autonomous vehicle bugs," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.*, 2020, pp. 385–396.
- [31] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2019, pp. 510–520.
- [32] F. Thung, S. Wang, D. Lo, and L. Jiang, "An empirical study of bugs in machine learning systems," in *Proc. IEEE 23rd Int. Symp. Softw. Rel. Eng.*, Piscataway, NJ, USA: IEEE Press, 2012, pp. 271–280.
- [33] H. Zhong, "Understanding compiler bugs in real development," in *Proc. IEEE/ACM 47th Int. Conf. Softw. Eng. (ICSE)*, Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 2025, pp. 605–605.
- [34] J. Chen et al., "An empirical comparison of compiler testing techniques," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 180–190.
- [35] H. Zhong, "Enriching compiler testing with real program from bug report," in *Proc. 37th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2022, pp. 1–12.
- [36] 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," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, pp. 1–30, 2021.
- [37] X. Du, Z. Zheng, L. Ma, and J. Zhao, "An empirical study on common bugs in deep learning compilers," in *Proc. IEEE 32nd Int. Symp. Softw. Rel. Eng. (ISSRE)*, Piscataway, NJ, USA: IEEE Press, 2021, pp. 184–195.
- [38] P. Li, Y. Liu, and W. Meng, "Understanding and detecting performance bugs in markdown compilers," in *Proc. 36th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Piscataway, NJ, USA: IEEE Press, 2021, pp. 892–904.
- [39] X. Xia, Y. Feng, Q. Shi, J. A. Jones, X. Zhang, and B. Xu, "Enumerating valid non-alpha-equivalent programs for interpreter testing," *ACM Trans. Softw. Eng. Method.*, vol. 33, no. 5, pp. 1–31, 2024.
- [40] H. Ma, W. Zhang, Q. Shen, Y. Tian, J. Chen, and S.-C. Cheung, "Towards understanding the bugs in solidity compiler," in *Proc. 33rd ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2024, pp. 1312–1324.
- [41] X. Xia, Y. Feng, and Q. Shi, "Understanding bugs in rust compilers," in *Proc. IEEE 23rd Int. Conf. Softw. Qual. Rel. Secur. (QRS)*, Piscataway, NJ, USA: IEEE Press, 2023, pp. 138–149.
- [42] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, "A large scale study of programming languages and code quality in GitHub," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 155–165.
- [43] A. Rahman, E. Farhana, C. Parnin, and L. Williams, "Gang of eight: A defect taxonomy for infrastructure as code scripts," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.*, 2020, pp. 752–764.
- [44] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *Proc. 18th ACM Symp. Operating Syst. Princ.*, 2001, pp. 73–88.
- [45] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on tensorflow program bugs," in *Proc. 27th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2018, pp. 129–140.
- [46] M. Bringolf, D. Winterer, and Z. Su, "Finding and understanding incompleteness bugs in smt solvers," in *Proc. 37th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2022, pp. 1–10.
- [47] Y. Xiong et al., "An empirical study of functional bugs in android apps," in *Proc. 32nd ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2023, pp. 1319–1331.
- [48] 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 *Proc. 32nd ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2023, pp. 551–563.
- [49] J. Chen, G. Wang, D. Hao, Y. Xiong, H. Zhang, and L. Zhang, "History-guided configuration diversification for compiler test-program generation," in *Proc. 34th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 305–316.
- [50] M. Sharma, P. Yu, and A. F. Donaldson, "Rustsmith: Random differential compiler testing for rust," in *Proc. 32nd ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2023, pp. 1483–1486.
- [51] H. Wang et al., "MLIRSmith: Random program generation for fuzzing mlir compiler infrastructure," in *Proc. 38th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Piscataway, NJ, USA: IEEE Press, 2023, pp. 1555–1566.
- [52] C. Cummins, P. Petoumenos, A. Murray, and H. Leather, "Compiler fuzzing through deep learning," in *Proc. 27th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2018, pp. 95–105.
- [53] J. Wu, Y. Yang, and Y. Zhou, "Boosting compiler testing via eliminating test programs with long-execution-time," in *Proc. IEEE Int. Conf. Softw. Anal., Evolution ReEng. (SANER)*, Piscataway, NJ, USA: IEEE Press, 2023, pp. 593–603.
- [54] K. Even-Mendoza, C. Cadar, and A. F. Donaldson, "Csmithedge: More effective compiler testing by handling undefined behaviour less conservatively," *Empirical Softw. Eng.*, vol. 27, no. 6, 2022, Art. no. 129.
- [55] C. Yang et al., "Whitefox: White-box compiler fuzzing empowered by large language models," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA2, pp. 709–735, 2024.
- [56] H. Gao, Y. Yang, M. Sun, J. Wu, Y. Zhou, and B. Xu, "Clozemas: Fuzzing rust compiler by harnessing llms for infilling masked real programs," in *Proc. IEEE/ACM 47th Int. Conf. Softw. Eng. (ICSE)*, Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 2025, pp. 712–712.
- [57] H. Ma, Q. Shen, Y. Tian, J. Chen, and S.-C. Cheung, "Fuzzing deep learning compilers with hirgen," in *Proc. 32nd ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2023, pp. 248–260.
- [58] Z. Wang et al., "GENCOG: A DSL-based approach to generating computation graphs for TVM testing," in *Proc. 32nd ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2023, pp. 904–916.
- [59] J. Liu et al., "Nnsmith: Generating diverse and valid test cases for deep learning compilers," in *Proc. 28th ACM Int. Conf. Archit. Support Program. Lang. Operating Syst.*, vol. 2, 2023, pp. 530–543.
- [60] C. Georgescu, M. Olsthoorn, P. Derakhshanfar, M. Akhin, and A. Panichella, "Evolutionary generative fuzzing for differential testing of the Kotlin compiler," in *Proc. Companion Proc. 32nd ACM Int. Conf. Found. Softw. Eng.*, 2024, pp. 197–207.
- [61] S. Kwon, J. Kwon, W. Kang, J. Lee, and K. Heo, "Translation validation for JIT compiler in the v8 JavaScript engine," in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng.*, 2024, pp. 1–12.
- [62] Q. Wang and R. Jung, "Rustlantis: Randomized differential testing of the rust compiler," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA2, pp. 1955–1981, 2024.
- [63] T. Sotiropoulos, S. Chaliasos, and Z. Su, "Api-driven program synthesis for testing static typing implementations," *Proc. ACM Program. Lang.*, vol. 8, no. POPL, pp. 1850–1881, 2024.
- [64] E. Liu, S. Xu, and D. Lie, "Flux: Finding bugs with LLVM IR based unit test crossovers," in *Proc. 38th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, 2023, pp. 1061–1072.
- [65] T. Theodoridis, M. Rigger, and Z. Su, "Finding missed optimizations through the lens of dead code elimination," in *Proc. 27th ACM*

- Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2022, pp. 697–709.
- [66] Y. Rong, Z. Yu, Z. Weng, S. Neuendorffer, and H. Chen, “IRFuzzer: Specialized fuzzing for LLVM backend code generation,” in *Proc. IEEE/ACM 47th Int. Conf. Softw. Eng. (ICSE)*, Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 2025.
- [67] R. Schumi and J. Sun, “SpecTes: Specification-based compiler testing,” in *Proc. Fundam. Approaches Softw. Eng.: 24th Int. Conf.*, New York, NY, USA: Springer Int. Publishing, 2021, pp. 269–291.
- [68] M. Wu, M. Lu, H. Cui, J. Chen, Y. Zhang, and L. Zhang, “JITfuzz: Coverage-guided fuzzing for JVM just-in-time compilers,” in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2023, pp. 56–68.
- [69] T. Gao, J. Chen, Y. Zhao, Y. Zhang, and L. Zhang, “Vectorizing program ingredients for better JVM testing,” in *Proc. 32nd ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2023, pp. 526–537.
- [70] Z. Zang, F.-Y. Yu, N. Wiatrek, M. Gligoric, and A. Shi, “Jattack: Java JIT testing using template programs,” in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng.: Companion Proc. (ICSE-Companion)*, Piscataway, NJ, USA: IEEE Press, 2023, pp. 6–10.
- [71] Z. Zang, N. Wiatrek, M. Gligoric, and A. Shi, “Compiler testing using template Java programs,” in *Proc. 37th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2022, pp. 1–13.
- [72] Y. Zhao et al., “History-driven test program synthesis for JVM testing,” in *Proc. 44th Int. Conf. Softw. Eng.*, 2022, pp. 1133–1144.
- [73] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, “Coverage-directed differential testing of JVM implementations,” in *Proc. 37th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2016, pp. 85–99.
- [74] Y. Chen, T. Su, and Z. Su, “Deep differential testing of JVM implementations,” in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2019, pp. 1257–1268.
- [75] S. Chaliasos, T. Sotiropoulos, D. Spinellis, A. Gervais, B. Livshits, and D. Mitropoulos, “Finding typing compiler bugs,” in *Proc. 43rd ACM SIGPLAN Int. Conf. Program. Lang. Des. Implementation*, 2022, pp. 183–198.
- [76] Y. Tian, Z. Xu, Y. Dong, C. Sun, and S.-C. Cheung, “Revisiting the evaluation of deep learning-based compiler testing,” in *Proc. IJCAI*, 2023, pp. 4873–4882.
- [77] C. Suo, J. Chen, S. Liu, J. Jiang, Y. Zhao, and J. Wang, “Fuzzing MLIR compiler infrastructure via operation dependency analysis,” in *Proc. 33rd ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, 2024, pp. 1287–1299.
- [78] W. Yang, C. Gao, X. Liu, Y. Li, and Y. Xue, “Rust-twins: Automatic rust compiler testing through program mutation and dual macros generation,” in *Proc. 39th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2024, pp. 631–642.
- [79] B. Bendrissou, C. Cadar, and A. F. Donaldson, “Grammar mutation for testing input parsers,” *ACM Trans. Softw. Eng. Method.*, vol. 34, no. 4, pp. 1–21, 2024, Art no. 116.
- [80] S. Li, T. Theodoridis, and Z. Su, “Boosting compiler testing by injecting real-world code,” in *Proc. ACM Program. Lang.*, vol. 8, no. PLDI, pp. 223–245, 2024.