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

***Abstract*—Modern C++, a programming language characterized by its extensive use of object-oriented programming (OOP) features, is popular for system programming. However, C++ compilers often struggle to handle these sophisticated OOP features correctly, resulting in numerous high-profile compiler bugs that can lead to crashes or miscompilation. Despite the significance of OOP-related bugs, OOP features are largely ignored by existing compiler fuzzers, hindering their ability to discover such bugs. In this work, we conduct the first empirical study focused on identifying and analyzing OOP-related bugs in the mainstream C++ compilers, GCC and LLVM. Based on the analysis of 776 bugs, we identify 11 findings that shed light on their types, symptoms, root causes, and so on. These findings provide valuable insights for enhancing compiler fuzzers. Based on our findings, we developed a proof-of-concept compiler fuzzer OOPFuzz, specifically targeting OOP-related bugs in C++ compilers, and applied it against the newest release versions of GCC and LLVM. Within a 12-hour run, it detected 9 bugs, of which 3 have been confirmed by the developers, including a bug of LLVM that had persisted for 13 years.**

I. Introduction

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

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

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

In this work, we conduct a systematic study of OOP-related bugs in C++ compilers. Our motivations are threefold. First, OOP-related bugs are prominent in C++ compilers. As shown later, for C++ compilers such as GCC and LLVM, 22.2% of the bugs are related to incorrect handling of OOP features. Second, these bugs are of significant importance, i.e., 37.9% of them are labeled with the top two priorities, and more than 89% of them result in a compiler crash or miscompilation. Third, existing C++ compiler fuzzers rarely generate complex code that includes OOP features. For example, CSmith [4], the most influential C compiler fuzzer, supports none of the OOP features in C++. Another high-impact C++ fuzzer, YARPGen [5], [6], does not generate classes and their dynamic memory allocation. Fuzz4All [7], a recent LLM-based fuzzer, supports limited OOP features and hardly generates correct non-trivial programs.

Although several empirical studies on compiler bugs have been conducted, OOP-related bugs are largely overlooked so far. For example, Sun *et al.* analyzed the distribution of bugs in GCC and LLVM, pointing out that the C++ components have the most bugs in both compilers, and yet they did not further analyze the bug types, symptoms, and root causes [8]. Zhou *et al.* studied the optimization bugs of GCC and LLVM [9] but did not analyze whether OOP-related features are relevant.

We conduct the first empirical study on OOP-related bugs of C++ compilers. Our study is based on two mainstream C/C++ compilers, GCC and LLVM. We systematically examined the bugs in the C++ components of the two compilers, and identified the bugs related to the OOP features. In total, we identified 776 bugs.We then systematically analyze these bugs from the following aspects. (1) the characteristics of the *bug-triggering programs*, (2) the *symptoms* of these bugs, (3) the characteristics of the *patches*, and (4) the characteristics of the relevant compiler *options*, and (5) shortcomings of existing compiler fuzzers for discovering these bugs. In general, our study aims to address the following research questions which we believe are essential for developing effective techniques for revealing OOP-related bugs.

- RQ1: What are the types of these bugs?
- RQ2: What are the symptoms and root causes of these bugs?
- RQ3: What is the relationship between the bug types and the symptoms?

- RQ4: What compiler options are relevant for triggering OOP-related bugs?
- RQ5: Do the bugs from different C++ compilers share certain commonality?
- RQ6: How does the evolution of C++ standards affects OOP-related bugs?
- RQ7: What do existing C++ compiler fuzzers need for revealing OOP-related bugs?

We propose a hierarchical classification of *bug types* that includes 10 primary categories and a total of 36 secondary categories that are different from the general bug types. Based on a thorough analysis that aims to answer the above-mentioned questions, we obtain 11 major findings that we believe are helpful for designing testing methods for revealing OOP-related compiler bugs. Based on the empirical study results and findings, we further propose practical guidelines for designing future C++ compiler fuzzers, to cover more features of modern C++ language and more components of compiler implementations. Furthermore, based on the guidelines we designed and implemented a simple yet effective proof-of-concept C++ compiler fuzzer, called **OOPFuzz**, enhancing generating test cases that cover OOP handling components of C++ compilers. By applying OOPFuzz on the latest releases of GCC and LLVM, under a time budget of 12 hours, we have found 9 bugs, of which 3 have been confirmed by the compiler developers and 2 have been fixed before we report in the trunk branch, indicating the effectiveness of our findings for testing and debugging C++ compiler bugs. The results demonstrate the usefulness of our findings.

To sum up, we make the following major contributions.

- We conduct a systematic study on OOP-related bugs in C++ compilers based on 776 bugs from the mainstream compilers, GCC and LLVM.
- For OOP-related bugs, we create a taxonomy for the bug types, and we identify the symptoms, root causes, options, and involved C++ standards.
- We obtain 11 findings and provide guidelines for developing further fuzzers and debugging OOP-related bugs.
- We develop a proof-of-concept fuzzer based on our findings, which successfully finds 9 bugs from the newest releases of GCC and LLVM.

II. Background: OOP Features and C++ Compilers

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 C++, 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 highlight the challenge and importance of discovering such OOP-related bugs.

A. Selected OOP Features

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

1) Encapsulation: C++ has many features supporting encapsulation, i.e., visibility control and binding methods and data. For example, to encapsulate data, C++ provides not only several kinds of *class-like* data structures (such as `class`, `struct`, `enum`, and `union`), but also a general visibility control `namesapce`. Each member of a C++ class has a legal scope, which could be shaped by composition and relationship (i.e., using internal members of other classes) or inheritance relationship (i.e., using members of parent classes). Inside a class, C++ provides access modifiers (e.g., `private` and `public`) to control the members that can be accessed. Even in lambda expressions, which are syntax sugar of anonymous functions, users can encapsulate operations and data.

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

3) Polymorphism: C++ supports both *runtime polymorphism* and *compile-time polymorphism*, for writing highly reusable code that can interact with objects of different types. Runtime polymorphism occurs when the method to be executed is determined at runtime, achieved through inheritance and virtual functions. Compile-time polymorphism, in contrast, occurs when the method to be executed is determined at compile time, achieved through templates and overloading. C++ templates are intrinsically powerful and complex, even for experienced C++ veterans. Combining compile-time polymorphism and runtime polymorphism leads additional complexity and numerous corner cases. As shown in Section IV-A, the complexity leads to many compiler bugs.

B. An OOP-Related Bugs Sample

Figure 1 shows an OOP-related bug with the highest priority, GCC-94549 [10]. This bug-triggering program shows the challenge of triggering and debugging OOP-related bugs in C++ compilers. The program involves not only key concepts of OOP (e.g., class, inheritance, template, and object initialization) but also advanced features covering multiple C++ standards. For example, the concepts (i.e., the keyword

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

Fig. 1: An example OOP-related bug: GCC-94549

`requires`) is introduced in C++17, the `using` style of constructor inheritance in `derived` is introduced in C++11, the initialization in `main` function involves template argument deduction is introduced in C++17. Moreover, to trigger this bug, the option `-std=c++17` is required. Referring to the corresponding patch, we find the root cause of this bug is the incorrect branch condition.

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

III. Methodology

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

A. Bug Collection

Several popular C++ compilers exist, including LLVM, Microsoft Visual C++, and Intel C++ Compiler. Following the existing studies [8], [9], [11], we select two mainstream compilers, i.e., GCC and LLVM, because they are open-source and have well-maintained bug-tracing systems.

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

To collect the keywords related to OOP concepts, we randomly selected 400 bugs and then manually analyzed all the bug reports of the C++ components from both GCC and LLVM. If the bug is related to OOP features, we discuss and extract keywords from the title and summary of the bug report. We additionally add a set of OOP keywords of C++ language, and OOP concept-related words and phrases. The result is a set of 91 keywords. Due to space limits, the full list of these keywords is in our anonymous repo [12].

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

TABLE I: Bug selected in this study.

Compiler	Start	End	Reports	Bugs	P1/S1	P2/S2
GCC	2017	2022	2959	558	84	158
LLVM	2014	2022	566	218	10	42
Total	-	-	3525	776	94	200

Table I shows 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 from the five years between 2017 and 2022, covering the newest C++ standard C++20. We collected 2959 and 566 closed bug reports from GCC and LLVM, respectively. Then we filter them by the OOP-related keyword, then we manually check whether the bug is a real OOP-related bug. At last, 558 remained for GCC, while LLVM had only 99. This aligns with existing studies showing that the number of bug reports in the LLVM community is significantly lower than in GCC [8]. 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, 84 and 158 of GCC are categorized as the top 2 priority levels (P1 and P2), respectively, while 10 and 42 bugs of LLVM are classified as the top 2 severity levels (S1 and S2), respectively. Therefore, 37.9% of bugs are labeled as the top 2 important levels, confirming the importance of OOP-related bugs.

B. Bug Labeling

For each bug, we collect its *bug report* (including title, description, and discussions among developers) and *patch* (including the diff files and the source files). We also manually extract *bug-triggering programs* and *options*. Using such information, we categorize each bug into a specific *bug type*, *symptom*, *root cause*, and the *involved C++ standard version*.

To do so, following prior studies, we use the detective coding approach to categorize a bug into a *bug type* [13]–[18]. More specifically, we use the taxonomy of OOP core concepts and C++ OOP features as the starting point of bug types, to understand the bug is related to which OOP core concepts and C++ OOP features. Similarly, we use the taxonomy of root causes and symptoms proposed in prior studies [15], [17], [19] as a starting point to identify the root cause and the symptom of these bugs. During this process, we observe there can be bugs that cannot be categorized in any existing taxonomy. Therefore, we revise and refine the taxonomy so that we can better understand the characteristics of the bug. For the *options*, we analyze the bug reporters' comments and developers' discussions, then extract the shortest options that obversed. To identify the involved C++ standard version of the bug, we first obtain a list of distinct features in each C++ standard version. Then, we manually read each bug to see if there is such a feature in it. If a bug is caused by OOP-related features, we consider the features to be related to the involved C++ standard versions. Extracting options directly from bug reports is reasonable, because as shown later in Section IV-D, the majority of bugs are triggered without any options, and bug reporters intend to provide the reduced options.

We follow the labeling process of the existing empirical studies [15], [18], [20]. One author first traversed all bug reports to determine the initial sets of bug types, symptoms, and root causes based on the general taxonomies [21], then adjusted them to compiler OOP-related categories. Then, the first two authors independently labeled the bugs. We measured the inter-rater agreement reliability among labelers by Cohen's Kappa coefficient [22]. For the first 5% labeling results, the Cohen's Kappa coefficient was near 22%, thus we conducted a training session for the raters. After that, the first two authors labeled 10% of bugs, and Cohen's Kappa coefficient was 80%. We then discussed and analyzed the reasons for the inconsistent labels. After that, the Cohen's Kappa coefficient was always more than 90% in the remaining labeling results. During the labeling process, any disagreements raised by the first two authors were discussed with the other authors. The most senior author reviews the final labeling results to ensure quality. At last, all the bugs are labeled consistently.

IV. Results and Analysis

A. RQ1: Bug Types

1) Bug Type Identification Results: We first investigate the bug reports to categorize their bug types. We identified 10 main categories of bug types, each further divided into several subcategories. All the categories are listed in Table II. However, due to space limits, we only elaborate on several representative bug types, and the details of the remaining categories are illustrated in our repo [12].

① **Encapsulation Related Bugs.** This category of bugs is triggered by encapsulation-related language features of C++. C++ defines access control for derived classes or other classes, and C++ allows the `friend` keyword to access private or protect members of other classes. Moreover, the namespace and `using` statement 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. It has 5 subcategories and we only introduce the following one.

Class member lookup bugs. These bugs occur due to the compiler failing to look up class members that are visible from the outer scope. Listing 1 shows one such bug, GCC-103081 [23], which cannot lookup the member `OINK` due to the wrong branch condition.

```
/* ---- The bug-triggering program ---- */
  enum class Pig { OINK };
  struct Hog {
    using enum Pig;
    Hog(Pig) { }
  };
  template <unsigned> void pen() {
     (void) Hog(Hog::OINK);
  }
  void pen() {   pen<0>(); }
/* ---- The corresponding patch ---- */
-   if (DECL_NAMESPACE_SCOPE_P (t))
+   if (!uses_template_parms (DECL_CONTEXT (t)))
```

Listing 1: Class Member Lookup Bug GCC-103081

```
/* ---- The bug-triggering program ---- */
  class a {
    virtual long b() const;
  };
  class c : a {
  public:
    long b() const;
  };
  class d : c {
    long e();
  };
  long d::e() { b(); }
/* ---- The corresponding patch ---- */
-   if (!fn || DECL_HAS_IN_CHARGE_PARM_P (fn))
+   if (!fn || DECL_HAS_VTT_PARM_P (fn))
```

Listing 2: Override-related Bug GCC-98744

② **Inheritance Related Bugs.** This category of bugs is triggered by complicated inheritance relationships or special corner inheritance cases. The compilers are error-prone regarding inheritance because C++ supports multiple inheritance, which could not only involve complex inheritance hierarchy but also introduce the concept of virtual base class. The most common root causes of this bug type are incorrect assignment of the flag values and overlooking corner cases. It has 4 subcategories and we introduce the following one.

Inherited class member related bugs. The bugs occur when the compiler fails to check for name conflicts between members of a subclass. Listing 1 shows one such example.

③ **Runtime Polymorphism Related Bugs.** This category of bugs is triggered by runtime polymorphism. Runtime polymorphism is one core feature of OOP in C++, which enables the program to decide the function to be executed during execution rather than at compile time. To achieve this, C++ compilers maintain virtual function tables for the base class and insert function pointers when the derived class overrides the base. The most common root cause is API misuse. It has 2 subcategories and we introduce the following one.

Override-related bugs. These bugs occur when the compiler implements polymorphism via override, especially wrongly building the virtual function table. Listing 2 shows one such bug [24], due to using the incorrect API.

④ **Compile-Time Polymorphism Related Bugs.** C++ achieves compile-time polymorphism via templates, which can be stamped out different instantiations under different type parameters. Templates introduce numerous syntactic rules and compile-time checks, often leading the compiler to overlook the handling of some cases. This category of bugs is triggered

```
/* ---- The bug-triggering program ---- */
  template<typename> struct s {
     template<typename... Args> requires true
     static void f(Args...) { }
  };
  auto foo = s<int>::f();
```

Listing 3: Template Parameter List Parsing Bug LLVM-44658

```
/* ---- The bug-triggering program ---- */
  template <typename> struct S {
    struct A;
    struct f A();
  };
  template class S <int>;
/* ---- The corresponding patch ---- */
-   if (tree ti = (TREE_CODE (fn) == TYPE_DECL
-                    && !TYPE_DECL_ALIAS_P (fn)
-                 ? TYPE_TEMPLATE_INFO (TREE_TYPE (fn))
-                 : DECL_TEMPLATE_INFO (fn)))
+   if (tree ti = get_template_info (fn))
```

Listing 4: Template Parameter Instantiation Bug GCC-90916

by parsing, instantiating, or specializing template classes. The most common root causes are missing corner cases and incorrect branch conditions. It has 5 subcategories and we introduce 2 of them.

Template parameter list parsing bugs. To trigger the bugs, the program should use the variadic template class, i.e., `<typename...>`. These bugs are caused by incorrect unpacking type parameter lists of template classes. Listing 3 shows one such example [25] causing the compiler crash, and its root cause is the misuse of APIs.

Template parameter instantiation bugs. This bug type is triggered by the type parameter instantiation of templates. Listing 4 shows one such example [26], which triggers the compiler crash on the valid code during instantiation, and the bug is caused by the incorrect branch condition. Note that the bug-triggering program is confusing, the code `struct f A()` is a function declaration with a returning type `struct f`, and the compiler should distinguish the function name `A` the data member `struct A`.

⑤ **Object Initialization Bugs.** This category of bugs is triggered by object initialization, i.e., the first stage during the lifetime of an object. The life cycle plays an important role in the OOP features of C++, where C++ objects contain many implicit or explicit convention rules during the initialization process, such as the choice of constructors and the initialization order of parent and child classes, which easily lead to compiler bugs. The most common root causes are API misuse and incorrect branch conditions. It has 5 subcategories and we introduce 2 of them.

List/direct initialization handling bugs. List initialization and direct initialization are special initialization approaches. This category of bugs is caused by the compiler incorrectly handling list/direct initialization. Listing 5 shows an invalid code that is accepted by LLVM [27]. The key to triggering this bug is the list initialization at the last line (i.e., `C c{...}`), and the root cause is the compiler missing this case.

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

Listing 5: List Initialization Bug LLVM-49020

```
/* ---- The bug-triggering program ---- */
  struct A {
    constexpr A() { c[0] = 0; }
    char c[2];
  };
  constexpr A a;
```

Listing 6: Non-static Data Member Initializer Bug GCC-99700

Non-static data member initializer bugs. These bugs are caused by incorrectly handling complex expressions assigned to class members during object initialization. Listing 6 shows one such example [28], which is invalid but accepted by GCC, triggered by the initializer of `c[0]`.

⑥ **Overloading Related Bugs.** This category of bug is triggered by overloading, and has 3 subcategories.

⑦ **AST Visiting Bugs.** This category of bug is triggered by parsing complex expressions related to OOP-related operations, which has 3 subcategories.

⑧ **Error Handling Bugs.** This category of bugs is caused by incorrect handling compilation errors, which has 3 subcategories.

⑨ **C++ New Feature Related Bugs.** This category of bug is triggered by new features of new C++ standards, such as C++17 and C++20, which has 5 subcategories.

⑩ **Others.** This category of bugs collects unusual and unique bugs that can not be classified into any other categories.

2) Bug Type Distribution: Table II illustrates the number of compiler OOP bugs with respect to their bug types. The first two columns are the hierarchical categories. The columns **GCC** and **LLVM** respectively show the number of bugs of a secondary category, and the column **Sum** indicates the sum of the bug numbers of both compilers. The column **All** shows the sum of the number of bugs in the primary categories.

From the table, we can find *Compile-Time Polymorphism Related Bugs* is the most common type, accounting for 156 (i.e., 20.1%) of the total bugs, including 104 bugs in GCC and 52 in LLVM. Inside this primary category, *Template parameter instantiation bug* is the most common secondary category, which accounts for 38 bugs, which is the third most common secondary category. Our analysis shows that the reason why compile-time polymorphism introduces many bugs is two-fold. First, the semantics of template classes are intrinsically complex, involving type deduction, template-level type/member checking, template instantiation, and template specification. Secondly, every new C++ standard version introduces new syntax for template classes. As a result, many bugs arise due to causes such as overlooking corner cases and branching conditions errors, as illustrated in Listing 1 and Listing 3.

TABLE II: Bug Distribution by Bug Types

Primary Category	Secondary Category	GCC	LLVM	Sum	All
Encapsulation Related Bugs	Class member lookup bugs	36	13	**49**	129
	Member pointer accessing bugs	13	2	15	
	Class members access control bugs	16	6	23	
	Friend function handling bugs	17	7	24	
	Lambda capturing bugs	14	4	18	
Inheritance Related Bugs	Inherited class member related bugs	14	4	**18**	58
	Empty base classes handling bugs	14	2	16	
	Virtual inheritances handling bugs	4	5	9	
	Incorrect invocation of constructors/destructors	12	3	15	
Runtime Polymorphism Related Bugs	Override implement bugs	7	5	12	25
	Incorrect conversions from the base class to its derived classes	10	3	**13**	
Compile-Time Polymorphism Related Bugs	Template parameter list parsing bugs	34	3	37	**156**
	Static class member parsing bugs	5	6	11	
	Dependent name handling bugs	21	13	34	
	Type deduction bugs	22	14	36	
	Template parameter instantiation bugs	22	16	**38**	
Object Initialization Bugs	List/direct initialization handling bugs	37	9	46	153
	Default constructor/destructor generation bugs	18	15	33	
	Non-static data member initializers bugs	42	8	**50**	
	Object array initialization bugs	5	3	8	
	Interface class initialization bugs	7	9	16	
Overloading Related Bugs	Function overloading bugs	12	7	19	51
	Operator overloading bugs	22	5	**27**	
	Function name mangling bugs	3	2	5	
AST Visiting Bugs	Type conversion management bugs	18	8	**26**	48
	R-value processing bugs	11	4	15	
	Constant propagation/folding bugs during IR generation	7	0	7	
Error Handling Bugs	Missing or wrong compilation error messages	14	12	**26**	54
	Useless compilation error messages	13	9	22	
	Error mark assignment bugs	6	0	6	
C++ New Feature Related Bugs	Coroutines related bugs	9	0	9	83
	Structured binding bugs	11	3	14	
	Constraints and concepts related bugs	22	4	26	
	Incorrect evaluation of `constexpr`/`consteval`	20	7	**27**	
	Incorrect implementation of `noexcept`	7	0	7	
Others	——	12	7	19	19

Finding 1: *Compile-Time Polymorphism Related Bugs* is the most common bug type, accounting for 20.1% of the bugs.

Object Initialization Bugs is the second most common primary category, which accounts for 153 (i.e., 19.7%) bugs, including 109 in GCC and 44 in LLVM. Within this primary category, *Non-static data member initializers bug* is the most common secondary category accounting for 50 bugs, which is also the most common bug among all secondary categories. One reason that object initialization triggers many bugs is that it is rather flexible in C++, and thus some cases are evitably overlooked. For example, there are many ways for initializing a new object of `T`, such as `T();`, `T{};`, `new T();`, and `new T{};`. Another reason is that C++ has many implicit rules to restrict initialization, which may be easily overlooked by developers. For example, the derived class should implicitly invoke its base class's initializer.

Finding 2: *Object Initialization Bugs* is the second most common bug type, accounting for 19.7% of the bugs. *Non-static data member initializers bugs* is the most common one among all secondary categories.

B. RQ2: Symptoms and Root Causes

1) Symptoms Classification Results: We identify the symptoms of compilers following existing studies [14], [15], [19].
① **Accept Invalid:** the compiler accepts an invalid code (i.e., syntactically incorrect code) that should be rejected.
② **Reject Valid:** the compiler rejects a valid code (i.e., syntactically correct code) that should pass the compilation.
③ **Crash:** the compiler unexpectedly terminates during compilation, often with error messages such as stack trace.
④ **Wrong Code:** the compiler successfully finishes the compilation but generates incorrect results or middle results.
⑤ **Diagnostic Error:** the compiler misses or throws incorrect errors or warnings.
⑥ **Inefficiency:** the compiler consumes an excessive amount of time or memory than expected.
⑦ **Hang:** the compiler cannot terminate within a long period.
⑧ **Others:** the symptoms beyond the aforementioned types, such as linkage errors or platform support issues.

2) Symptoms Distribution: Figure 2 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 38.8% of the bugs. In addition, *Reject Valid*, *Wrong Code*, *Diagnostic Error*, and *Accept Valid* account for 29.1%, 12.5%, 8.8%, and 7.7% of the bugs, respectively, which indicates that they are non-negligible. *Crash* is the most common symptom partially because it can be easily observed. In contrast, detecting

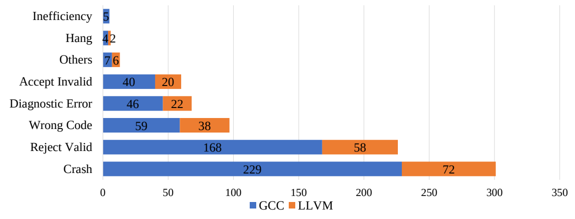


Fig. 2: Bug Distribution by Symptoms

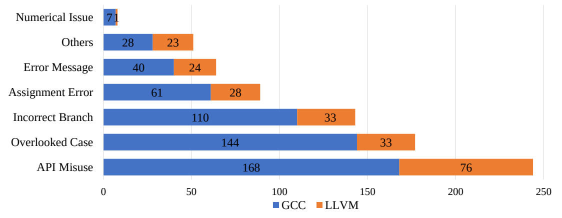


Fig. 3: Bug Distribution by Root Cause

other symptoms is often challenging due to lack of proper oracles. For example, developing a general oracle for the *Accept Invalid* bug (for instance, the one shown in Listing 6) would be extremely challenging. Different from other types of compilers [15], [19] where crash dominates the symptoms, the majority of OOP-related C++ compiler bugs do not lead to crash. This suggests that we must address the oracle problems properly for revealing such bugs.

> **Finding 3**: The majority (i.e., 61.2%) of bugs manifest *non-crash* symptoms.

3) Root Causes Classification Results: We adopt the root cause taxonomies following existing studies [15], [17]–[19]. Note that some domain-specific categories do not exist in our study, so we only adopt the common ones.

① **API Misuse:** developers misunderstand APIs, e.g., using incorrect APIs or calling APIs with wrong parameters.

② **Overlooked Case:** developers overlook some special cases.

③ **Incorrect Branch:** developers use incorrect conditions in branch statements.

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

⑥ **Error Message:** developers use wrong warning messages.

⑤ **Numerical Issue:** developers use incorrect numerical computations.

⑦ **Others:** caused by other reasons.

4) Root Cause Distribution: Figure 3 shows the root cause distribution, from which we can find the top 3 common root causes are *API Misuse*, *Overlooked Case*, and *Incorrect Branch*, accounting for 31.4%, 22.8%, and 18.4% of bugs, respectively. We conjecture that these causes are common mainly due to the complexity of the OOP-related syntax and semantic rules.

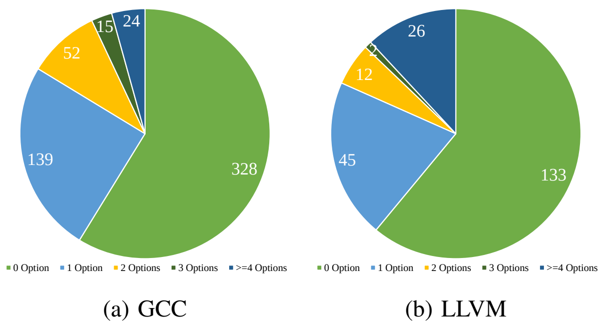


Fig. 4: Option Number Distribution

> **Finding 4**: The top 3 common root causes are *API Misuse*, *Overlooked Case*, and *Incorrect Branch*.

C. RQ3: The Correlation between Bug Types and Symptoms

Next, we investigate whether there is certain correlatiom between the bug types and the bug symptoms. Table III presents the distribution of symptoms for each bug type. The rows represent the bug types of primary categories, with the most frequent symptom highlighted in dark gray and the second most frequent in light gray. Each column represents a symptom, and within each symptom, the bug type with the highest frequency is in bold.

For the most common bug type *Compile-Time Polymorphism Related Bugs*, except *Diagnostic Error* and *Inefficiency*, all kinds of the symptoms occur, although *Crash* dominates, i.e., it accounts for 50% of bugs. Similarly, the second most common bug type *Object Initialization Bugs* shows all kinds of symptoms. The results indicate that the common bug types exhibit a various range of symptoms.

The overall most common symptom *Crash* is the most common one in 8 bug types, the second most common in the remaining 2, confirming that it is the easiest symptom to manifest. Similarly, the second most common *Reject Valid*, ranks in the top 2 most common symptoms for 6 bug types. While in *Wrong Error Handling*, 85.2% of the bugs predominantly manifest the symptom *Diagnostic Error*. The results suggest that *Crash* can be directly used as the test oracle, while differential testing may be able to detect *Reject Valid* bugs.

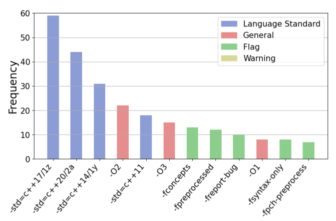
> **Finding 5**: The common bug types cover a broad spectrum of symptoms, where *Crash* and *Reject Valid* are the most common symptoms among them.

D. RQ4: Bug Triggering Compiler Options

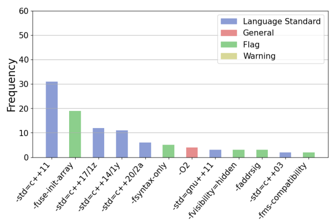
Besides source code, compiler options are another kind of important input, which should be properly set to trigger bugs [29]–[31]. To study the impact of compiler options on discovering bugs related to OOP features in C++ compilers, we analyze the user-submitted options extracted from the discussions among developers. To trigger the bugs, we count how

TABLE III: The Correlation between Bug Types and Symptoms

	Accept Invalid	Reject Valid	Crash	Wrong Code	Diagnostic Err.	Inefficiency	Hang	Others
Encapsulation Related Bugs	10	**55**	45	6	8	1	8	**3**
Inheritance Related Bugs	4	12	24	14	3	0	0	1
Runtime Poly. Related Bugs	3	2	12	7	0	0	0	1
Compile-Time Poly. Related Bugs	13	52	**78**	9	0	0	2	2
Object Initialization Bugs	**15**	44	59	**23**	5	1	**3**	**3**
Overloading Related Bugs	4	15	21	10	1	0	0	0
AST Visiting Bugs	5	15	16	9	2	0	0	1
Error Handling Bugs	0	0	5	1	**46**	**2**	0	0
C++ New Feature Related Bugs	6	29	32	14	2	0	0	0
Others	0	2	9	4	1	0	0	2



(a) GCC



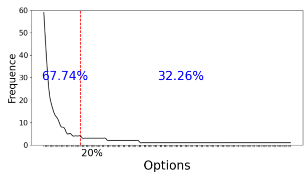
(b) LLVM

Fig. 5: Most Frequent Options

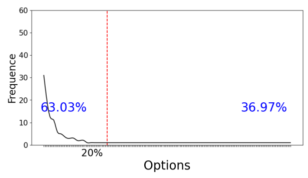
many options are used by GCC and LLVM, shown as Figure 4. The results show that for GCC, 58.8% of bugs are triggered by no options, and 24.9% only by 1 option. Similarly, for LLVM, 61.0% of bugs do not require any options, and 20.6% are triggered by only 1 option. For the cases of a single option, we find the majority of cases specify the language standard. As shown later, many bug-triggering programs contain new features from new language standards, and some new features require setting the corresponding standard. The result suggests we could use a simple strategy for generating the compiler options, such as trying one option at a time, to identify many of the bugs. We remark that a non-negligible portion (i.e., 6.4%) of bugs need at least 4 options. This result however should be taken with a grain of salt since not all options may be relevant, i.e., we can possibly minimize the failure-triggering options in these cases.

Finding 6: To trigger the OOP-related bugs in C++ compiler, 83.1% of the bugs need more than 1 option.

We categorize the involved options into *Language Standard* options to set standard version (e.g., `-std=c++20`), *General* options to set optimization level (e.g., `-O2`), *Flag* options controlling the compile functions that start with `-f` (e.g., `-fsanitize`), and *Warning* options that start with `-W` (e.g., `-Werror`). Figure 5 shows the most frequently 12 used options by categories. Although in both GCC and LLVM, the most frequently used options are *Language Standard*, GCC's most frequently used option is `-std=C++17`, while for LLVM, it is `-std=C++11`. Compared with C++17 and C++11, C++ 14 has fewer bugs because it does not introduce many new semantics. In addition, in both compilers, *Language Standard* options are the most frequent in the top 12. These



(a) GCC



(b) LLVM

Fig. 6: Long Tail Effect on Compiler Options

TABLE IV: Spearman Correlation Coefficients of GCC and LLVM

-	Spearman rank	p-value
Bug Type	0.853	0.004
Symptom	0.976	0.000
Option Number	0.812	0.049

results confirm the previous finding on option numbers, and further imply that we should properly set the language standard in some cases.

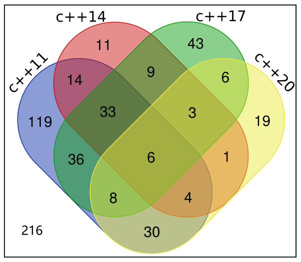
Finding 7: *Language Standard* options are the most frequently used to trigger OOP-related bugs.

We observe from Figure 4 that a non-negligible portion (i.e., 6.4%) of bugs need at least 4 options, so we further calculate the frequency of each option and sort them from high to low, shown as Figure 6. For the top 20% options by frequency, GCC options account for 67.74% of all option usage counts, while LLVM options account for 63.03%. The results exhibit the Long Tail Effect, where the remaining options are still significant. It also shows that the combination of options plays a crucial role in triggering bugs [32]. We remark that the majority of existing bugs requiring few options may be due to inadequate exploration of option combinations, i.e., it is conceivable that there may be subtle bugs that require combination of many options that are under-explored.

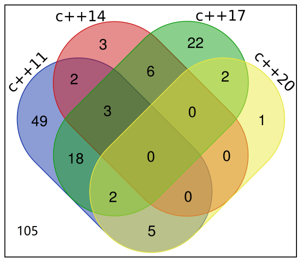
Finding 8: The options in both GCC and LLVM represent the Long Tail Effect.

E. RQ5: The Commonality across C++ Compilers

Following existing studies [15], [18], [33], we adopt the Spearman correlation coefficients to evaluate the commonality,



(a) GCC



(b) LLVM

Fig. 7: Involving C++ Standards of Bug-Triggering Programs

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 as Table IV. 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 9: GCC and LLVM have significant commonalities in bug types, symptoms, and the number of options to trigger compiler bugs.

F. RQ6: 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 as Figure 7. In GCC and LLVM, respectively, 61.3% and 51.8% of bugs involve features from the new C++ standards, and the most common feature is C++11, which involves 44.8% and 36.2% of bugs. Moreover, 26.9% of bugs in GCC and 17.4% in LLVM cover at least two versions of new standards. These results show that new freshly introduced features may interact with existing OOP features in unexpected ways, resulting in many new cases to be considered and resultant bugs. For example, the bug shown in Listing 3 is introduced by the new feature variadic template.

Finding 10: In both compilers, 58.6% of bugs cover C++ new standards, and 24.2% cover at least two versions.

G. The OOP Feature Support of Existing Fuzzers

Existing compilers can be categorized into *generation-based* and *mutation-based* [1]. In our study, we evaluate the capabilities of generation-based fuzzers in generating programs that incorporate OOP features. Specifically, we examine the support provided by fuzzers such as the CSmith family [4], [31], [34], YARPGen [5], [6], and Fuzz4All [7]. Note that Fuzz4All is a recent fuzzer driven by LLMs. Conversely, for mutation-based fuzzers, our analysis focuses on the ability of their mutation operators to alter code snippets containing OOP features. In this category, we include fuzzers like GrayC [35] and FLUX [36].

Based on the bug types, our analysis covers the following OOP features: 1) *definition of C++ classes*, 2) *inheritance relationships*, 3) *composition relationships*, 4) *template classes*, 5) *access to class member variables*, 6) *object initialization*, 7), *object destruction*, 8) *member function overloading*, 9) *arrays of objects*, and 10) *object pointers*. Unfortunately, except for Fuzz4All, none of the other fuzzers support these features. Moreover, Fuzz4All relies on language standards and cannot generate programs with complex feature combinations.

Finding 11: Existing C++ compiler fuzzers lack of support of OOP features.

V. Discussion

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

A. Implications

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

1) New program generation and mutation strategies for C++ compilers: According to **Findings 1** and **2**, we find that about 40% of bugs in C++ compilers are related to *compile-time polymorphism* and *object initialization*. However, according to **Findings 10** and **11**, none of the existing fuzzers support these features. We thus suggest that corresponding generation or mutation strategies must be developed to address the bug types in RQ1. For example, we could design mutation operators to rewrite initializers and type parameters of templates. The seed or sketch programs should cover new standard features. According to **Finding 9**, we can use the same fuzzing strategies for GCC and LLVM.

2) Oracle enhancement for automated testing: According to **Findings 5**, we find that while *Crash* serves as a strong oracle in compiler fuzzing [8], [15] since it is the most common symptom, using it as the sole oracle means that a significant portion (i.e., 61.2%) of bugs, indicating their limitations in effectively discovering OOP related bugs for C++ compilers. Differential testing could only partially detect bugs of non-crash symptoms, such as *Reject Valid* and *Wrong Code*. Therefore discovering new oracles for compiler testing deserves more attention.

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

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

B. Threats to Validity

Similar to existing empirical studies, our study is potentially subject to the threats introduced by manual inspections, including identifying keywords, bug types, symptoms, root causes, options and involved C++ standards in bug-triggering inputs. To reduce the threat, we referred to existing empirical studies on bugs [15], [16], [18]–[20], [37], [38], 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 [8], [9], [11], and we systematically collect the bug reports that are successfully fixed. Moreover, our study is large-scale, involving 776 real-world bugs across 8 years, guaranteeing the generalizability of our results.

VI. Proof-of Concept Application

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

We applied OOPFuzz on the popular C++ compilers GCC and LLVM, on their newest released versions (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 within a running less than 12 hours. We submitted 7 new bug reports because 2 bugs had been already fixed in the trunk branches 3 weeks ago. Among the 7 bugs, 3 of them have been confirmed by the developers, where 2 from LLVM and 1 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*".

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 to detect OOP handling bugs in C++ compilers.

VII. Related Work

Empirical Studies of Bugs. The most related work is the empirical studies on the bugs of GCC and LLVM. Sun *et al.* conducted a quantitative analysis of the overall distribution of bugs in GCC and LLVM [8], pointing out that the C++ component is the most buggy-prone. Sun *et al.* also studied the characteristics of warning defects in both compiler [11]. Zhou *et al.* analyzed their characteristics in optimization bugs [9]. These studies do not analyze the OOP-related bugs and the relationship between compiler bugs and language features.

Besides, other types of compilers are extensively empirically studied, such as WebAssembly compilers [14], deep learning compilers [15], [39], MarkDown compilers [40], Python interpreters [17], and the Rust compiler [41].

Moreover, in other types of software, researchers invest much research effort in analyzing bug characteristics. For example, the bugs of operating systems [42], quantum computing platforms [19], deep learning frameworks [18], deep learning models [20], SMT solvers [43], and Android apps [44].

Compiler Fuzzers. Compiler fuzzers automatically generate programs to test compilers, which have detected thousands of real-world bugs and attract much academic [1], [2], [45] and industrial attention [46]. Existing compiler fuzzers can be classified into generation-based and mutation-based. Generation-based approaches automatically generate programs from scratch by pre-defined grammar rules. For example, the CSmith family randomly generates programs under simplified semantics [4], [31], [34], [47]–[49]. YARPGen [5], [6] is a recent fuzzer for C/C++, armed with more grammar features. The most recent generation-based approaches leverage LLM for generating code [7], [50]. However, these fuzzers exclude OOP-related grammar rules and thus lack OOP support. Moreover, many generation strategies are designed for other types of compilers [51]–[53]. Mutation-based approaches apply mutation operators to seed programs and transform them into new programs. Fuzzers of this family including C appraoches [32], [33], [35], [36], [54]–[56], JVM testing [57]–[64] and deep learning compilers [30].

VIII. Conclusion

In this work, we conduct a comprehensive study of the OOP-related bugs in C++ compilers. We analyzed the bug types, symptoms, root causes, options, and C++ standards of these bugs, and proposed 11 findings. Our findings are useful and provide insights for improving compiler fuzzers.

The original data and results of our study are publicly available at our repo [12], for the sake of open science.

References

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

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

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

[4] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 283–294.

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

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

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

[8] C. Sun, V. Le, Q. Zhang, and Z. Su, "Toward understanding compiler bugs in gcc and llvm," in *Proceedings of the 25th international symposium on software testing and analysis*, 2016, pp. 294–305.

[9] Z. Zhou, Z. Ren, G. Gao, and H. Jiang, "An empirical study of optimization bugs in gcc and llvm," *Journal of Systems and Software*, vol. 174, p. 110884, 2021.

[10] GCC-94549. Accessed: March 15, 2024. [Online]. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=94549

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

[12] Anonymous Repo of Our Study. Accessed: March 15, 2024. [Online]. Available: https://github.com/OOPBugsICSE25Artifacts/OOP-Related-Bugs-Study-ICSE25-Artifacts

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

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

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

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

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

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

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

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

[21] B. Beizer, *Software system testing and quality assurance*. Van Nostrand Reinhold Co., 1984.

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

[23] GCC-103081. Accessed: March 15, 2024. [Online]. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=103081

[24] GCC-98744. Accessed: March 15, 2024. [Online]. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=98744

[25] LLVM-44658. Accessed: March 15, 2024. [Online]. Available: https://bugs.llvm.org/show_bug.cgi?id=44658

[26] GCC-90916. Accessed: March 15, 2024. [Online]. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=90916

[27] LLVM-49020. Accessed: March 15, 2024. [Online]. Available: https://bugs.llvm.org/show_bug.cgi?id=49020

[28] GCC-99700. Accessed: March 15, 2024. [Online]. Available: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=99700

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

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

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

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

[33] Y. Tang, H. Jiang, Z. Zhou, X. Li, Z. Ren, and W. Kong, "Detecting compiler warning defects via diversity-guided program mutation," *IEEE Transactions on Software Engineering*, vol. 48, no. 11, pp. 4411–4432, 2021.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[49] C. Cummins, P. Petoumenos, A. Murray, and H. Leather, "Compiler fuzzing through deep learning," in *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, 2018, pp. 95–105.

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

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

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

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

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

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

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

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

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

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

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

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

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

[63] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of jvm implementations," in *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 85–99.

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