From Bug Reports to Workarounds: The Real-World Impact of Compiler Bugs

Zhixing He
Shanghai Jiao Tong University, China
674815634@qq.com

Hao Zhong
Shanghai Jiao Tong University, China
zhonghao@sjtu.edu.cn

***Abstract*—Compilers are essential in daily development, but their bugs can introduce hidden bugs to many software projects. As compiler bugs are important, researchers have conducted various empirical studies to understand their characteristics. Most of the prior studies analyze only the bug reports and patches of compilers. From such sources, they analyze the root causes, locations, and repairs of compiler bugs. Their findings are useful for understanding compiler bugs themselves, but to the best of our knowledge, the impact of compiler bugs is rarely explored. For example, before compiler bugs are fixed, programmers would bypass such bugs, but no prior study analyzes the workarounds for compiler bugs.**

To complement the prior studies, in this paper, we analyze how the bugs of two mainstream compilers, `gcc` and `llvm`, affect real development. Besides the bug reports and patches of compilers, in this study, from 603 GitHub projects, we collect 806 commits whose messages mention compiler bugs. From the dataset, we analyze how compiler bugs affect real projects. Furthermore, we manually analyze 219 commits whose messages explicitly mention compiler bugs. Our results lead to eight useful findings for programmers, compiler developers, and researchers. For instance, based on our findings, we suggest that it is critical to analyze the long-term impact of compiler bugs. Our story about a `gcc` bug confirms that compiler bugs can affect the compilations of projects even after these compiler bugs have already been fixed.

I. Introduction

Each time when programmers modify source files, they need to compile their files. The compilation process can introduce serious hidden bugs, since compilers can have bugs like other software artifacts [35], [40]. For example, compilers can compile valid programs to wrong machine code. Meanwhile, compilers can fail to identify invalid programs and generate machine code whose behaviors are undefined. As these bugs are difficult to detect, compiler bugs can introduce significant risks to the maintenance of software projects. As compiler bugs are critical, researchers [31], [33], [37], [43], [47] have conducted various empirical studies to understand compiler bugs. These studies have derived interesting findings about the buggy locations [37], repair patterns [37], [43], [47], and causes [33], [37] of compiler bugs. In the above studies, researchers analyze the bug reports and their patches to understand compiler bugs.

When programmers maintain a software project, if they encounter compiler bugs, they can implement workarounds to bypass compiler bugs. These workarounds provide a new angle for understanding compiler bugs. First, as compiler bug reports do not report the projects that encounter compiler bugs, it is difficult to know which projects (*e.g.*, Linux) can encounter compiler bugs. Second, compiler bug reports can be polluted by the reports submitted by researchers. Researchers either use random programs [24], [42] or mutated programs to test compilers, which may not appear in real development. In contrast, the compiler bugs in workarounds are all real bugs in the wild. Finally, compiler bug reports rarely introduce how to bypass them. To resolve the above limitations, Zhong [45] conduct an pioneer study and explore how compiler bugs affect real development. As the first work on this topic, he explored affected (end) users, modified lines of workarounds, and buggy compiler components when compiler bugs are encountered in real development. Although his findings are insightful, his study does not cover all perspectives about the influence of compiler bugs. Many questions are still open. For example, what workarounds would programmers use to bypass compiler bugs with specific symptoms?

To meet the timely need, we conducted an empirical study on two mainstream compilers such as `gcc` and `llvm`. Besides bug reports and their patches, in this study, we collect 2,989 commits whose messages mention the bugs or workarounds of two mainstream compilers. We analyze the characteristics of projects that encounter compiler bugs in real development. Among them, we manually inspected 219 commits whose messages explicitly mention the URLs of compiler bug reports. As our commits are collected from real development, we can analyze the impact of compiler bugs in real development. Compared with the prior studies, our study has multiple advancements. First, the prior studies do not analyze the outreaching impact of bugs, but we explored their impact, *e.g.*, the workarounds of compiler bugs. Second, the distribution of compiler bugs can be distorted since researchers and compiler developers themselves have filed many compiler bugs, but our study reveals the distribution from real development. In this study, we explore the following research questions:

- **RQ1. Which types of projects are affected?**
  Motivation. The distribution of compiler bugs in real-world projects may illustrate their potential impact.
  Protocol. To answer this question, we extract the code lines, issue reports, stars, and contributor numbers of the projects to learn their characteristic.
  Answers. Most compiler bugs occur in projects whose

code lines are above 10^6 (Finding 1). Although most affected projects have only ten stars, most compiler bugs can affect around 100 programmers (Finding 3). As most projects have no issue reports, most compiler bugs are identified by programmers themselves (Finding 2).

- **RQ2. What are the symptoms in the wild?**
  Motivation. Compiler bugs have various behaviors and can result in different degrees of impact on programs.
  Protocol. To answer this question, we build a taxonomy based on the taxonomies of `gcc` and `llvm`.
  Answers. In real development, Finding 4 shows that the symptoms of most compiler bugs are displaying wrong diagnostic messages (29.22%), rejecting valid programs (28.77%), and generating wrong code (23.29%). Compiler bugs that are triggered by valid programs are more frequently mentioned than those that are triggered by invalid programs (Finding 5).
- **RQ3. How do programmers bypass compiler bugs?**
  Motivation. The results are useful for understanding how programmers live with unfixed compiler bugs.
  Protocol. To answer this question, we construct a taxonomy for workarounds.
  Answers. Finding 6 shows that workarounds for compiler bugs either modify programs that trigger bugs (60.73%) or build files to suppress warning messages (33.33%).
- **RQ4. What are the associations between symptoms and workarounds of compiler bugs?**
  Motivation. If programmers can understand the relation between symptoms and workarounds, they can choose the most feasible workarounds.
  Protocol. To answer this question, we count the types of workarounds for each symptom.
  Answers. Finding 7 shows that modifying programs is used to bypass all types of symptoms while modifying build files is often used to bypass wrong warnings and optimization bugs.

II. Compiler Bug in Real Development

Most prior studies [31] analyze the bug reports and patches of only compilers. From such sources, it is feasible to learn the symptoms and causes of compiler bugs, but these sources do not provide the context of compiler bugs. As a result, the prior empirical studies can derive only partial findings. For example, a `gcc` bug [1] complains that `gcc` does not implement some intrinsics. From its description, it is straightforward for researchers like Shen *et al.* [33] to collect its symptom and root cause. As these intrinsics are unimplemented, its symptom could be rejecting valid programs if they call these intrinsics. However, from only the bug report and its patch, it is infeasible to understand its impact on real-world development. For example, which missing intrinsics are more frequently called by real projects? Which one should be implemented first? Can it be bypassed, and if so, how? Due to the limitation, the prior studies do not answer the above questions.

To complement the prior studies, we analyze commits that mention compiler bugs. If programmers encounter compiler bugs in real development, they can leave traces in such commits. For example, `Oblas` is a blas-like routine to solve systems in finite fields. Its commit [877a4b] mentions the same bug and presents a workaround:

```
1 #if defined(__GNUC__) && !defined(__clang__) \
2     && !defined(__ICC)
3 static inline __m256i __attribute__((__always_inline__))
4 _mm256_loadu2_m128i(const __m128i *const hiaddr, const
      __m128i *const loaddr) {
5     ...
6 }
7 #endif
```

Among the missing intrinsics, the above commit shows that `_mm256_loadu2_m128i()` is truly called by a real project. To bypass this bug, `Oblas` programmers implement their own version of this intrinsic. The developers of `gcc` can put more effort into implementing this intrinsic, and they can learn the existing implementation from `Oblas` programmers. Other programmers can also learn how to bypass compiler bugs from this commit. As our data source provides more information, we can derive findings that do not appear in prior studies.

Despite the benefits, it is more difficult to analyze commits than compiler bug reports. In this example, this workaround involves modifying two source files, resulting in 63 additions and 46 deletions. Among them, 8 lines of code are critical to understand the workaround. Unlike reading bug reports, commits often have only short messages. We have to manually read code changes to fully understand how workarounds work. Although it is more difficult to analyze commits than to read bug reports, we have analyzed 219 commits, and the size of the subjects is comparable with the prior studies. For example, Romano *et al.* [31] and Wu *et al.* [38] analyze 146 and 347 compiler bug reports, respectively.

III. Methodology

In this section, we introduce our analysis methodology.

A. Dataset

In this study, we analyze the bugs of two popular compilers, `gcc` [15] and `llvm` [14]. These compiler bugs are mentioned in commit messages. Based on GitHub API [5], we implement a tool to search for such commits. In 2022, with our tool, we queried GitHub for the workarounds or bugs of `gcc` and `clang`. When building the keywords, we concatenate "bug" with compiler names, since it is used to describe abnormal behaviors, and we concatenate "workaround", since it can be used when bypassing compiler bugs. Table I shows the results. Column "Compiler" lists compilers. Column "commit" lists all retrieved unique commits. GitHub's search function can only retrieve up to one thousand search results. From them, we remove duplicated commits from forked projects.

Some retrieved commits are irrelevant to compiler bugs. For example, the message of a retrieved commit [2] is "Fixed *bugs* in flash boiling model, ran *clang* formatting". In this commit, a programmer fixed an irrelevant bug and used `clang` to format source files. Therefore, we consider this commit to be unrelated to compiler bugs and remove it from our dataset. The filtered data is used as the input of our automatic analysis in

TABLE I: Our dataset

Keyword	commit	RQ1		Other RQs	
		filtered	project	commit	project
gcc	1,471	417	329	96+106	182
clang	1,518	389	303	17	17
total	2,989	806	603	219	188

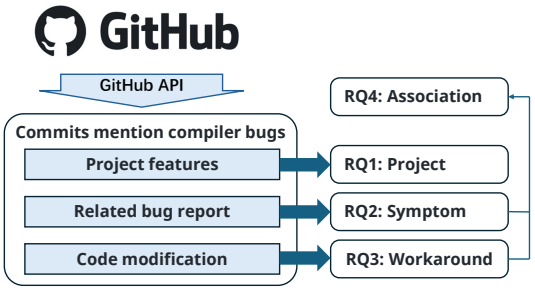


Fig. 1: Workflow of our research

RQ1. Subcolumn "filtered" of column "RQ1" lists the commits after we remove the irrelevant commits. Subcolumn "project" lists the number of projects. As some projects overlap, the totals are less than the sums.

With his support tool, Zhong [45] analyze more compiler bugs, but a tool cannot implement complicated and accurate analysis, *e.g.*, classifying the workarounds of compiler bugs. To complement his study, we introduce manual analysis in RQs 2, 3, and 4. In particular, we manually select the commits whose messages explicitly mention the URLs of compiler bug reports from the commits in RQ1. In total, we select 113 such commits. To collect more commits, we search Gitbhub with "gcc.gnu.org/bugzilla". The urls of all `gcc` bug reports have the above keyword. In this way, we retrieved 106 additional commits. As reported by Zhong [45], about half of mentioned compiler bugs are fixed.

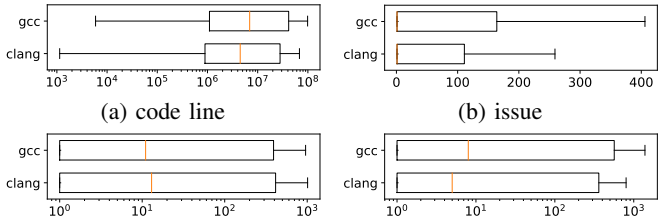
In total, we analyzed 219 commits in RQs 2, 3, and 4. The number is comparable with some prior studies [31], [38].

B. General Protocol

Figure 1 shows the overall workflow of our study.

In RQ1, from the projects these commits belong to, we extract their features like the number of code lines, issue reports, stars, and programmers. We explore which types of projects can encounter compiler bugs in real development. We implement a tool based on GitHub API [5]. The tool can calculate the metrics of projects (*e.g.*, lines of code).

The other RQs analyze the symptoms and workarounds of compiler bugs. We cannot implement a tool for these RQs since it is even difficult to manually determine and classify the symptoms and workarounds of compiler bugs. For example, the commit [3] has only a simple message, "Workaround for old GCC 4/8 bug". From only this message, it is infeasible to determine its symptom. Although it is feasible to check out the commit and compile it locally, it is challenging to identify the symptoms of this bug. First, many commits do not compile after they are checked out [36]. For example, if some API libraries are missing, compilers will report the errors but



(a) code line (b) issue

(c) star (d) programmer

Fig. 2: The characteristics of projects.

the symptoms of compilers are hidden. Second, the symptoms of some compiler bugs are difficult to identify. For example, compilers can compile source files into wrong code. Thus, it is hard to distinguish whether a symptom is caused by source files or compilers. As introduced in Section II, some commit messages explicitly mention the URLs of compiler bug reports. From such bug reports, it is feasible to learn the symptoms of compiler bugs. Both `gcc` and `llvm` have their own categories of symptoms. During our manual inspection, we merge the commits of each compiler from two keywords, and we analyze only those commits whose symptoms are explicitly described. We build our category based on their categories. For the workarounds, we inspect whether they modify source files or build files. If they modify source files, we further analyze which types of source files are modified. If they modify build files, we further analyze which options are disabled.

IV. Empirical Result

This section presents our analysis results. More details are listed on our project website:
https://github.com/Chandlerooo/CompilerWorkaround

We also provide a table on the website to easily access the links to commits and bug reports mentioned in this section.

A. RQ1. Characteristic of Project

1) Protocol: In this research question, we explore which types of projects can encounter compiler bugs. We select all the projects in Table I as the input of this study. For each project, our tool extracts its lines of code, the number of issues, the number of programmers, and the number of stars. We group each type of data and draw a box plot to analyze its distribution.

2) Result: Figure 2a shows the total code lines per project. We present the results on a logarithmic scale as the code lines vary significantly among keywords. The keywords' medians are around 10^7 lines of code and 72.0% of projects have more than one billion lines. Several projects are quite large. For example, when the keywords are "bug + gcc" and "bug + clang", the largest retrieved project is `sourceruckus/linux-mdl` [6]. As a forked Linux kernel, this project has over a hundred billion lines of code. A commit of this project [6] mentions a `gcc` bug [7]. This bug report complains that `gcc` wrongly calculates the length of a `union struct`, and produces false warnings. To bypass the compiler bug, programmers modify a `struct` in this commit. `gcc` stops producing false warnings after the modification. This compiler

TABLE II: The taxonomy of our observed bug symptoms

our taxonomy	gcc taxonomy	llvm taxonmy
wrong-code	wrong-code	miscompilation
rejects-valid	rejects-valid	compile-fail
diagnostic	diagnostic	
optimization	missed-optimization	performance
	ra	quality-of-implementation
	lto	slow-compile
crash	ice-on-valid-code	
c++ feature	c++-lambda	
	C++-coroutines	
link-failure	link-failure	
environment	build	build-problem

bug affects `Linux`, a project with billions of lines of code. The above observations lead to a finding:

> **Finding 1.** Most projects whose commit messages mention compiler bugs have around 10^6 lines of code.

The prior studies [39], [43], [47] report no similar findings. This finding can warn programmers about the importance of compiler bugs if the sizes of their projects reach a bar.

In GitHub, issues serve as an important method for users to report compiler bugs they encounter. For example, `golang/go` is an emerging programming language popular in implementing web servers and storage clusters. Until now, its issue tracker has more than 7,980 issue reports. Among them, we find a bug reported by a `go` programmer [53528]. This programmer implements several projects in `go` but is not a core member of the `go` project. This programmer complains that `go` builds cannot be reproducible due to the `gcc` bug. This `gcc` bug report complains that a `gcc` source file passes the source directory directly to the compiled code. This compiler bug can lead to unreproducible builds of `go` projects. Although the `go` and the `gcc` bug reports are still open, this example shows that users can encounter and even bypass compiler bugs.

Figure 2b presents the number of issues per project. Nearly half of the projects have no issue report. As a result, many projects will not receive bug reports involving compilers, and many programmers must identify compiler bugs by themselves. In particular, 45.6% of projects that count `gcc` bugs and 46.9% of projects that count `llvm` bugs have no issue report. An issue tracker is the official channel for users to report bugs. In the absence of issue reports, programmers must identify compiler bugs on their own. Based on the above observations, we conclude a finding as follows:

> **Finding 2.** Although compiler bugs can affect users, most compiler bugs are identified by programmers themselves since many projects have few or even no issue reports.

The prior studies on compiler bugs [39], [43], [47] do not report similar findings, but this finding is consistent with other studies on open-source projects. For instance, Zhong *et al.* [46] analyze 11,684 open-source projects and find that 80% of projects have fewer than 10 bug reports. Most such

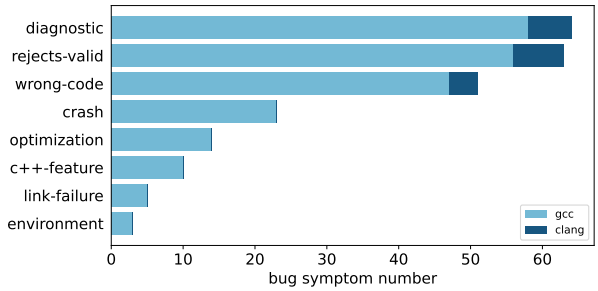


Fig. 3: The overview of bug symptoms

projects are not toy programs, but the distribution highlights the challenges of implementing attractive open source projects. As most projects have no issue reports, programmers must identify the impact of compiler bugs in most projects, but it is challenging even for experienced programmers to identify some compiler bugs. For instance, compilers can accept invalid programs without any warnings and thus silently introduce bugs. According to this finding, there is a strong need for a tool that can notify programmers whether their source code will be affected by compiler bugs.

GitHub users can star projects and keep track of their interested projects. The stars of a project construct an indicator of popularity. Figure 2c shows the distribution of stars. Around 30% of projects have no stars, and the medians for both compilers are under 30. The contributor of a project contributes to at least a commit of the project. Figure 2d shows the distribution of contributors. Around one-quarter of the projects have more than 270 contributors. The above observations lead to a finding:

> **Finding 3.** Most affected projects have only ten stars but can have more than 100 programmers.

The prior studies on compiler bugs [39], [47] do not report similar findings. This finding implies that compiler bugs can affect many small projects and programmers. In small projects, many programmers are not as experienced as those in large projects. They can have more difficulties in identifying whether their code triggers compiler bugs. It is desirable if an approach can actively notify compiler bugs.

B. RQ2. Symptom

1) Protocol: In this research question, we analyze the symptoms of compiler bugs. Here, a symptom of a bug refers to the observable behaviors or outcomes caused by the bug. The messages of commits are often concise. In addition, such messages often mention the symptoms of their own bugs but seldom mention the symptoms of compiler bugs. As a result, it is quite difficult to understand the symptoms of compiler bugs relying only on commit messages. It is also difficult to reproduce such bugs. First, many commits are not compilable after they are checked out [36]. For example, a checked-out project can lose some libraries and a compiler bug may not be triggered before the proper versions of such libraries are

located. Second, a compiler bug can be triggered by specific versions of compilers, but such versions are not recorded.

Instead of identifying symptoms by ourselves, we refer to bug reports to learn their symptoms. As shown in Section II, commits can explicitly mention the URLs of compiler bugs. These bug reports explicitly describe the symptoms of compiler bugs. For example, `gcc` developers explicitly classify their bug reports with keywords [8], and some keywords describe the symptoms of bugs. In this study, we analyze all commits that explicitly mention compiler bug reports, which are mostly the URLs of compiler bug tracer websites.

Table II presents our taxonomy. Both `gcc` [8] and `llvm` [9] and keywords to describe the symptoms of bugs. Columns "gcc taxonomy" and "llvm taxonomy" respectively list the keywords defined by `gcc` and `llvm` . We merge their keywords to build our taxonomy. Column "our taxonomy" shows the result. For instance, we merge "missed-optimization" and "ra" to our optimization. This table does not show all keywords of `gcc` and `llvm` since they do not appear in our dataset.

Among the keywords of a bug report, we consider only the symptom keywords. For example, a bug report [10] has three keywords: `diagnostic`, `patch` and `wrong-code`. The `patch` keyword denotes that a patch is implemented to fix this bug. We ignore this keyword but consider the `diagnostic` and `wrong-code` keywords since they are listed in Table II. This bug report has two symptoms, and we count it for both `diagnostic` and `wrong-code`. As a result, the sum can be more than the total of commits with bug reports.

2) Result: Figure 3 shows the overview of symptoms. The specific types are as follows:

S1 Diagnostic (64/219, 29.22%). This category includes compiler bugs that produce wrong messages. For example, in a commit [bace34] of the project `DXX-Rebirth`, programmers meet the situation that `gcc` wrongly emits warnings. For example, the bug-trigger code is as follows:

```
1 void prepare_error_string(..., const void *t)
```

When an uninitialized array is passed by `const T *` to a function, `gcc 11` assumes the array is an input to the function and issues a warning accordingly. However, in this particular code, the pointer is passed with a write-only purpose – recording the memory address of the affected array of the eventual exception. The called function does not dereference the pointer and thus cannot be affected by any uninitialized values in the underlying array. The developers of `gcc` recognized that this warning message was incorrect and classified it as a `diagnostic` bug. As a result, we put this bug into the diagnostic category.

S2 Wrong code (51/219, 23.29%). If a program is compiled to wrong code, we determine it as a `wrong-code` bug. `QUICK SILVER` is a project of flight controller firmware. A commit of `QUICK SILVER` [cd821b] complains that `gcc` can fail to read unaligned `structs`. For instance, consider the following `struct`:

```
1 struct foo {
2         char c;
3         int x;
4 } __attribute__((packed));
5 struct foo arr[2] = { { 'a', 10 }, {'b', 20 } };
6 int *p0 = &arr[0].x;
7 int *p1 = &arr[1].x;
```

A compiler will lay out the members of a `struct` in their declared order, and insert padding bytes between members to ensure each member is aligned. The `packed` attribute asks `gcc` not to add padding bytes. The compiled code works well on `Ubuntu` but fails to retrieve the values by referencing the pointers (`*p0` and `*p1`) on `Solaris`. The `gcc` developers determine that this is a wrong-code bug and agree that `gcc` should warn the unaligned members. As a result, we count it in both wrong-code and diagnostic categories.

S3 Rejects-valid (63/219, 28.77%). If a program conforms to the language specifications of C/C++, it is considered valid. If a compiler rejects a valid program, we classify the bug in this category. For example, `INET` is an open-source communication networks simulation package designed for the `OMNEST/OMNeT++` simulation system. An `INET` programmer submitted a commit [96b05e] to bypass a `gcc` bug that rejects valid programs. An example is as follows:

```
1 struct A {
2   template<class T> struct B;
3   template <> struct B<int*> { };
4 };
```

In the above program, `Structure A` contains a template `structure B` and a full specialization for the template where `T` is explicitly defined as `int*`. In other words, it provides a specific implementation for the case when `T` is `*int`. According to CWG 727 [11], a full specialization can be declared inside a class definition. Nevertheless, during the compilation process of GCC, an error occurred with the message "error: explicit specialization in non-namespace scope" on the third line of the above program. The `gcc` developers confirm that `gcc` wrongly rejects the valid program. The above observations lead to a finding:

> **Finding 4.** In real development, the symptoms of most compiler bugs are displaying wrong diagnostic messages (29.22%), rejecting valid programs (28.77%), and generating wrong code (23.29%).

When analyzing tool-chain compiler bugs, Xie *et al.* [39] report that the most frequent symptoms are compilation failures, *e.g.*, crashes. Their finding is consistent with the `gcc` issue tracker [8], where the top three symptoms are crashes (33.6%), generating wrong code (16.5%), and optimization issues (16.5%). Research tools mainly detect crashes (see Section VII). Still, we find that programmers tend to discuss more interesting compiler bugs, which are less studied.

Although invalid programs violate language specifications, compilers can fail to identify them due to `accepts-invalid` bugs. When this happens, invalid programs are silently compiled to machine code, but their behaviors are unpredictable. According to the `gcc` issue tracker [8], `rejects-valid` bugs and `accepts-invalid` bugs account for 9.8% and

4.4% of all bugs, respectively. However, in this study, despite the presence of many `rejects-valid` bugs, we did not find any `accepts-invalid` bugs. Programmers typically do not intentionally write invalid programs during real development. As they believe that all their written programs are valid, if compilers do not identify invalid programs, programmers may not notice `accepts-invalid` bugs. Similarly, we find many `iceonvalidcode` bugs, but we do not find `ice-on-invalid-code` bugs, although it accounts for 7.6% of all bugs in the `gcc` issue tracker. The observations lead to the following finding:

> **Finding 5.** Compiler bugs triggered by valid programs leave more traces in commits than those triggered by invalid programs.

The prior studies on compiler bugs [39], [43], [47] do not report similar findings. This finding indicates that compiler bugs stemming from valid programs tend to be more noticeable. As a result, these bugs leave more traces of such bugs in commits. In contrast, compiler bugs involving invalid programs are often overlooked, as programmers may not realize that their code is invalid. As a result, these bugs leave fewer traces and are more challenging to detect.

S4 Crash (23/219, 10.50%). If compilers crash during the compilation process, we put the corresponding bugs into this category. Compiler developers often call them internal compiler errors (ICEs). For example, a commit [482ac5] complains about a crash caused by a compiler bug. The `gcc` compiler crashes at the following code line:

```
1     util::Variant<decltype(NULLPTR), std::shared_ptr<
          Scalar>, ...> value;
```

When compiling with an optimization level that is greater than 2, `gcc` encounters an internal compiler error and crashes with an error message "internal compiler error: Segmentation fault". Therefore, we classified this bug into the crash category.

S5 Optimization (14/219, 6.39%). This category includes compiler bugs that are related to optimizations. A commit from *Xenia* [10ff77], an experimental emulator for the Xbox 360, mentions a compiler bug. The reduced program is as follows:

```
1 static void foo(int *x, long n){
2   long i = 0;
3   for (; i + 4 <= n; i += 4) {} // main loop
4   for (; i < n; i++) { // residual loop
5     x[i]++;
6   }
7 }
8 void bar(int *x){
9   foo(x, 128);
10 }
```

In the above program, when the `n` parameter of `foo` is set to 128, after executing the "main loop", `i` would be equal to `n`, and the line in the "residual loop" would not be executed. Therefore, the loop optimization of `gcc` skips the residual loop in the compilation stage. When the compiler flags are set to `-O1 -ftree-vrp`, `gcc` still evaluates the loop control expression of the residual loop during execution, even if it is always false. In addition, it reports a false warning on Line 5: "warning: iteration 4611686018427387903 invokes undefined behavior [-Waggressive-loop-optimizations]", although the loop does not iterate so many times. The `gcc` developers confirm that the loop optimization is missing and this warning message is wrong. As a result, we count this bug in both our `optimization` and `diagnostic` categories.

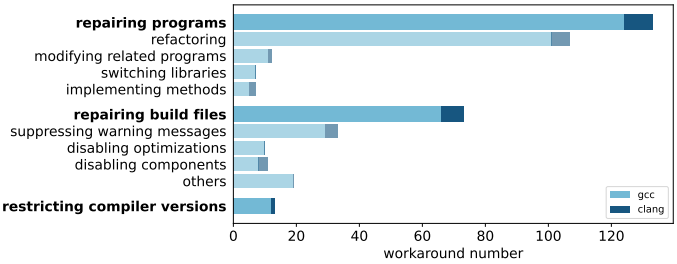


Fig. 4: The distribution of workarounds

S6 C++ feature (10/219, 4.57%). If the bug is related to a C++ feature, we determine its symptom as `C++ feature`. For example, in a commit of `freeciv21` [0df9a4] the programmer mentions a `gcc` bug. This `gcc` bug report complains of a crash when utilizing `sizeof()` in a C++'s lambda expression. An example program is as follows:

```
1 unsigned count = 5;
2 bool array[count];
3 [&array] () {
4   array[0] = sizeof(array) > 5;
5 }();
```

`gcc` fails to resolve the lambda expression of the above program and crashes with an error message "internal compiler error: in expand_expr_real_1". As lambda expressions are a C++ feature, we put it into the `C++ feature` category, and we put it into the `crash` category since it crashes.

S7 Link-failure (5/219, 2.28%). The compiler bugs in this category fail to link due to various reasons, *e.g.*, missing symbols, or undefined references. For example, *libjxl* is a library for manipulating `JPEG` images. In a commit [87fe7c], a programmer complains that `gcc` fails to automatically link the atomic library and requires explicit linking in the `CMake` module. Thus, we classify this bug as a `link-failure` bug.

S8 Environment (3/219, 1.37%). A compiler bug in this category is not triggered by programs but by the environment, such as platforms. `Spack` [12] is a multi-platform package manager project. A commit of this project [6e36c7] mentions a compiler bug. This bug occurs in the bootstrap comparison. The bootstrap builds the target version of `gcc` with the existing C compiler and builds the target version of `gcc` with the newly built `gcc`. If the two target versions are identical, it determines that the compiler is correct. According to this commit, the bootstrap fails when `gcc` 8 is checked on macOS bigsur apple with an error message "Bootstrap comparison failure! gcc/tree-ssa-operands.o differs ...". The developer confirms and later fixes this bug. As this bug occurs in compiler-building environments, we classify it into this category.

In summary, compared with compiler bugs triggered by invalid programs, those triggered by valid programs leave more traces in commits. From our collected commits, in total, we identify 8 symptoms of compiler bugs.

C. RQ3. Workaround

1) Protocol: In this research question, we analyze the category of workarounds for compiler bugs. Here, workarounds refers to the modifications of bypassing bugs. From Table I, we select the commits that implement workarounds for compiler bugs. As the first step, we classify workarounds by the types of their modified targets. We find that workarounds can modify source files, build files, and restrict compiler versions. Among them, if a workaround modifies source files or build files, we further refine it into subcategories. In particular, if a workaround modifies source files, we analyze whether modified code lines trigger compiler bugs. We then compare the code before and after the modifications to learn why a modification can bypass compiler bugs. If a workaround modifies build files, we analyze the functionalities of modified compilation flags.

2) Result: Figure 4 shows the distribution of workarounds.

T1 Repairing programs (133/219, 60.73%). In this category, programmers modify source files.

T1.1 Refactoring (107/219, 48.86%). These compiler bugs are triggered by specific programs, and programmers refactor code to avoid the symptoms. For example, the bug mentioned in S3 shows a `reject-valid` bug from `INET` [96b05e]. The following program triggers this compiler bug:

```
1 class INET_API{ ...
2     template <typename T>
3     const Ptr<T> peek(...) const {
4         ... // general template code
5     }
6     template <>
7     const Ptr<Chunk> peek(...) const {
8         ... // full specification code
9 }}
```

Due to the compiler bug, `gcc` produces an error on the full specification of a template at Line 8. To bypass this bug, programmers modify the above program as follows:

```
1 class INET_API{ ...
2     template <typename T>
3     const Ptr<T> peek(...) const {
4         if (std::is_same<T, Chunk>::value){
5             ... // full specification code
6         }
7         else{
8             ... // general template code
9 }}}
```

The workaround abandons the full specialization method that triggers the compiler bug, opting instead to accomplish the desired functionality by implementing a conditional branch. This new program avoids triggering the compiler bug. Thus, we put this workaround into the `refactoring` category.

T1.2 Modifying related programs (12/219, 5.48%). In this category, programmers do not directly modify source code lines that trigger compiler bugs, but modify related source code lines to handle or avoid bugs. The commit [10ff77] mentions a `gcc` optimization bug. The program is as follows:

```
1 for (i = 0; i + 8 <= count; i += 8) {
2     // main loop
3 }
4 for (; i < count; ++i) {
5     // residual loop
6 }
```

When `count` is set as a constant that is divisible by 8, the *residual loop* will never be executed. Although the *residual loop* could be optimized, `gcc` does not optimize it but produces a wrong warning. To solve this problem, programmers modify the above program as follows:

```
1 for (i = 0; i + 8 <= count; i += 8) {
2     // main loop
3 }
4 for (; i < count; ++i) {
5 +   if ((count % 8) == 0) __builtin_unreachable();
6     // residual loop
7 }
```

The `__builtin_unreachable()` method is a built-in method of `gcc` [13]. It tells the compiler that the following program is unreachable. After programmers add `__builtin_unreachable()` in the *residual loop*, `gcc` does not produce any warnings for this loop. In this case, the structure of the program is not modified. The bug is bypassed by calling the built-in methods of `gcc`. As a result, we put this workaround into this category.

T1.3 Switching libraries (7/219, 3.20%). In this category, compiler bugs are triggered by libraries, and programmers replace the buggy libraries with alternative libraries. For example, `Wesnoth` is an open-source, turn-based tactical strategy game project. The programmer encounters an `optimization` bug [6b52e1]. The program is as follows:

```
1 #include <regex>
2 static const std::regex valid_id("[a-zA-Z0-9_]+");
3 if(std::regex_match(newid, valid_id)) {...}
```

When the above program is compiled by `gcc` 10.2.0 on `mingw64`, the compilation becomes quite slow due to an issue in `regex`. To address this issue, programmers switch from the built-in `regex` library to the alternative library implemented by `boost`:

```
1 #include <boost/regex.hpp>
2 static const boost::regex valid_id("[a-zA-Z0-9_]+");
3 if(boost::regex_match(newid, valid_id)) {...}
```

The alternative library does not suffer from this issue, and the slow compilation is resolved.

T1.4 Implementing methods (7/219, 3.20%). In this category, compiler bugs are triggered by the intrinsic and other low-level methods of compilers. Programmers override the problematic methods. For example, a commit [642dc5] complains that the intrinsic method, `__sync_fetch_and_nand()`, is implemented by `gcc` but not by `clang`. To bypass this issue, a programmer implements this method and calls this method when compiling with `clang`:

```
1 #define CAS_NAND(x, val){                    \
2   __typeof__ (*(x)) tmp = *(x);              \
3   while (!__sync_bool_compare_and_swap( \
4     x, tmp, ~(tmp & (val)))) {               \
5     tmp = *(x);                              \
```

```
6   }                                          \
7   return tmp;                                \
8 }
9 ...
10 #ifdef __clang__
11   CAS_NAND(x, (StgWord8) val)
12 #else
13   return __sync_fetch_and_nand(x, (StgWord8) val);
14 #endif
```

T2 Repairing build files (73/219, 33.33%). This category includes workarounds that modify the build files of compilers.

T2.1 Suppressing warning messages (33/219, 15.07%). These bugs produce false warning messages. To bypass such messages, programmers explicitly disable corresponding compiler flags [4]. For example, `RIOT` is a real-time multi-threading operating system for internet devices. In a commit [6588db] of `RIOT` programmers encounter a false warning message. To bypass it, they modify the `Makefile` as follows:

```
1 CFLAGS += -Wno-maybe-uninitialized
```

In the above flag, `-Wno` asks a compiler to ignore the following warning option. Here, disabling a flag can miss true warning messages.

T2.2 Disabling optimizations (10/219, 4.57%). To bypass optimization bugs, programmers can disable corresponding flags. For example, `MuJS` is an embeddable Javascript interpreter in C. In this project, a commit [90a634] complains a `gcc` bug. When compiling with `-O1`, `gcc` can produce wrong code when a program has pure looping and calls a nonreturn method. This commit adds the following code:

```
1 #ifdef __GNUC__
2 #if (__GNUC__ >= 6)
3     #pragma GCC optimize ("no-ipa-pure-const")
4 #endif
5 #endif
```

If the version of `gcc` is above 6, the above code adds the `no-ipa-pure-const` flag. This flag asks `gcc` not to discover which methods are pure or constant. After that, the wrong code disappears. Although the symptom is removed, disabling optimizations can lead to a loss in efficiency.

T2.3 Disabling components (11/219, 5.02%). Programmers can disable some compiler components to bypass compiler bugs. For example, a commit [6e36c7] complains of a bootstrapping failure. To bypass this bug, programmers add a new conflict:

```
1 #Bootstrap comparison failure: see: ...
2 #gcc.gnu.org/bugzilla/show_bug.cgi?id=100340
3 on XCode 12.5 conflicts('+bootstrap', when='@:11.1 %apple-
      clang@12.0.5')
```

In Line 3, `conflicts()` defines that a component has a conflict and should be ignored in the compilation.

T2.4 Others (19/219, 8.68%). This category includes the other unidentified types of modifications. For example, to bypass the previous `libjxl` bugs in S7, programmers add the following lines to the bug file [87fe7c]:

```
1   check_cxx_source_compiles("${atomic_code}"
        ATOMICS_IN_LIBRARY)
2   set(CMAKE_REQUIRED_LIBRARIES)
3   if(ATOMICS_IN_LIBRARY)
4     set(ATOMICS_LIBRARY atomic)
5     include(FindPackageHandleStandardArgs)
6     find_package_handle_standard_args(Atomics DEFAULT_MSG
          ATOMICS_LIBRARY)
7     set(ATOMICS_LIBRARIES ${ATOMICS_LIBRARY})...
```

The compiler bug fails to locate the atomic library, and the above lines find replacements for this library.

> **Finding 6.** A significant portion of the workarounds are implemented by modifying build files (33.33%).

The prior studies on compiler bugs [39], [43], [47] do not report similar findings. Instead of compiler bugs, Yan *et al.* [41] analyze workarounds for general bugs and report that 11% of the workarounds modify configuration files like build files. Compared with their finding, our finding shows that modifying configurations bypasses more compiler bugs since they have many flags to disable functionalities.

T3 Restricting compiler versions (13/219, 5.94%). If compiler bugs appear in specific compiler versions, a programmer can take `version limitation` workaround that explicitly restricts the use of these compiler versions. For example, a commit [844b4d] mentions a `gcc` bug. Due to this bug, when `__builtin_add_overflow()` is called with `uint32_t` values, `gcc` can incorrectly report overflows. To resolve this issue, the commit report errors for specific compilers:

```
1 #if !defined __GNUC__ || __GNUC__ < 7 || (__GNUC__ == 7 &&
      __GNUC_MINOR__ < 1)
2 #error insufficient compiler for building on s390x
3 #endif
```

The above code lines print an error message for specific compiler versions.

The above observations lead to a finding:

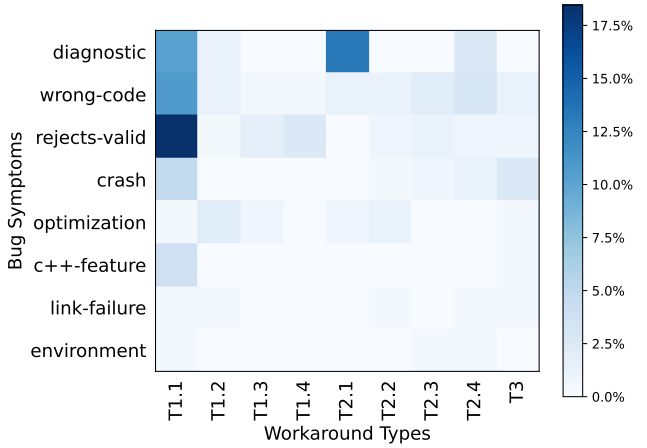
> **Finding 7.** Refactoring programs (48.86%) and suppressing warning messages (15.07%) are the two most frequent workarounds for compiler bugs.

The prior studies on compiler bugs [39], [43], [47] do not report similar findings. As for the workarounds of general bugs, Yan *et al.* [41] report that the most such workarounds are the modifications of API calls. When programmers compile code, they seldom call the APIs of compilers. As a result, when bypassing compiler bugs, programmers often modify the configuration files of compilers. As most modifications only disable compilation options, workarounds can introduce technical debt. For instance, suppressing warning messages may disable true warning messages. As a result, programmers can ignore bugs even if the compiler can detect them.

In summary, unlike bypassing other bugs, bypassing compiler bugs requires modifying the configuration files of compilers and the source files that trigger compiler bugs.

D. RQ4. Association

1) Protocol: In this RQ, we analyze the relationship between symptoms and workarounds of compiler bugs. For the compiler bugs of each symptom, we analyze their workarounds



T1.1: refactor, T1.2: modifying related programs, T1.3: switching libraries, T1.4: implementing methods, T2.1: suppressing warning, T2.2: disabling optimizations, T2.3: disabling components, T2.4: others, T3: restricting versions.

Fig. 5: The associations between symptoms and workarounds

and classify them by the types of workarounds. In this way, we build the matrix between symptoms and their corresponding workarounds.

2) Result: Figure 5 presents the relationship. A cell denotes the proportion of compiler bugs that are bypassed by the corresponding types of workarounds. For instance, 24 compiler bugs have `diagnostic` symptom and are bypassed with `refactor` workarounds (T1.1). As shown in the top left cell, they account for 10.96% of total bugs (24/219), and its color is light blue. Based on Figure 5, we find that when compiler bugs produce wrong warnings (`diagnostic`), more than half of the commits directly disable corresponding compiler flags, and the other half of commits refactor programs. When the symptoms are `wrong-code`, `rejects-valid`, and `crash`, most commits modify programs to bypass compiler bugs. For example, to bypass `wrong-code` bugs, some commits disable buggy components. For `optimization` bugs, only one bug directly modifies programs that trigger compiler bugs. Instead, to bypass optimization bugs, most commits modify related programs, switch libraries, and disable optimization flags. An interesting observation is that while only 5 commits restrict compiler versions, they are all applied to bypass serious symptoms (4 crashes and 1 `rejects-valid` bug).

The above observations lead to a finding:

> **Finding 8.** Modifying programs bypasses most symptoms, and modifying build files often bypasses wrong warnings and optimization bugs.

The prior studies on compiler bugs [39], [43], [47] do not report similar findings. As for the workarounds of general bugs, Yan *et al.* [41] report that many associations between causes and repairs are straightforward. However, as shown in Figure 5, these associations are more complicated in compiler bugs. As a result, it may be worth exploring strategies to bypass compiler bugs.

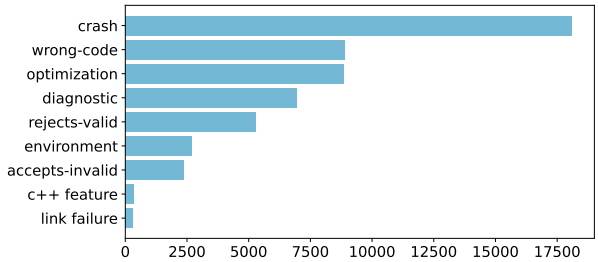


Fig. 6: The distribution of symptoms calculated by `gcc`

E. Threat to Validity

An internal threat to validity is the underlying GitHub API and our keywords. The retrieved results can be incomplete, irrelevant, and duplicated. To reduce this threat, we manually checked whether retrieved commits truly mentioned compiler bugs. Still, we can wrongly identify commits. To eliminate this threat, we released our results on our website. Other researchers can recheck our results. The external threat of our study includes the fact that it is a bit of the time. All empirical studies share this threat. Our study needs to be replicated as time goes by. For example, when we start to write the paper, we rerun our tool, and it retrieves new commits that do not appear in our dataset. The new commits can illustrate new patterns for bypassing compiler bugs. As another example, future compilers can have better channels to collect bugs and large teams to repair bugs. They can leave fewer unfixed bugs and fix compiler bugs faster. The external threat also includes our limited inspected instances. This thread could be reduced if more experienced programmers are invited to inspect commits that do not explicitly mention the URLs of compiler bugs.

V. Interpretation of Our Findings

In this section, we interpret our findings:

Researching compiler bugs in real development. As introduced in Section IV-B1, `gcc` uses a set of keywords to classify the symptoms of compiler bugs. Based on the keywords and the criteria in Table II, we build the distribution of symptoms. Figure 6 shows the results. Finding 2 shows that programmers need to identify and bypass many compiler bugs, but Figure 6 shows that most compiler bugs may not cause noticeable symptoms. For instance, compilers can silently accept invalid programs. It is worth exploring how to actively warn programmers of compiler bugs. In addition, our study reports how to bypass compiler bugs. Researchers can work on approaches to generate such workarounds. Finding 4 shows wrong code, rejects-valid, and diagnostic are the top frequent symptoms in real development. The distribution is different from the distribution calculated from the total bug reports of compilers. For example, Figure 6 shows the distribution of symptoms calculated from all bug reports of `gcc`. Compared with our distribution in Figure 3, the ranks differ. A possible explanation is that crashes can be fixed quickly, and programmers do not need to bypass them.

Understanding the long-term impact of compiler bugs and their workarounds. Although workarounds bypass com-

piler bugs, they can introduce technical debts in software [40], [41]. In addition, programmers often have to live with compilers with bugs. Even if their bugs are fixed, compilers with bugs can affect many projects, and their workarounds are useful in the long run. Finding 6 shows that a significant portion of the workaround may introduce technical debts. This raises the cost of software maintenance. For example, in T2.2, optimization flags are disabled, and in T2.1, warning messages are suppressed. Although these workarounds hide symptoms, they produce less-optimized code and can introduce vulnerabilities to the code. After compiler bugs are fixed, removing workarounds is necessary to resolve such issues. Although programmers write the URLs of bug reports in their commit messages, they will not be notified if compiler bugs are fixed. For the example in Section II, after `gcc` developers implement the missing intrinsic, `Oblas` programmers did not notice this fix, and this workaround had not been removed for a whole year. This year, `Oblas` called their own implemented intrinsic, and the behavioral difference can cause hidden bugs and performance issues. Both `gcc` and `llvm` allow subscribing to the fixing process of compiler bugs. Based on this interface, it is feasible to implement a tool to track and notify the status of compiler bug reports.

Learning how to bypass compiler bugs. As shown in Section IV-D, there are associations between symptoms and workarounds, and the associations can offer guidance on how to bypass compiler bugs. For example, Finding 8 shows that modifying build files is often used to bypass wrong warnings and optimization bugs. Programmers can use the guidance to bypass compiler bugs. Finding 4 shows that reject-valid and wrong-code bugs are among the top three compiler bugs in real development environments. As compilers do not produce errors for these bugs, it is difficult for programmers to identify them, especially when they are unfamiliar with compiler bugs. It can be useful if a tool can actively identify whether source files can trigger such bugs.

VI. Long-term Impact of Compiler Bug

Our interpretations in Section V are actionable. For instance, according to our findings, we advocate that researchers and programmers should understand the long-term impact of compiler bugs and their workarounds. In this section, we explain the long-term impact with an example. In particular, to fulfill our vision, we notify the programmers of a workaround that the corresponding compiler bug is already fixed. This bug is a `gcc` bug [90538] bug, and it rejects the following code:

```
1 template <class... T>
2 void a(const T&...) {}
3 template <class... T>
4 void b(const T&... t) {
5   [&]() { a(t...); a(t...);};
6 }
7 void c() {
8   b(1);
9 }
```

The above code defines two template functions, `a` and `b`, and `b` invokes `a` inside a lambda that captures variables by reference. Although this code snippet is valid, `gcc` rejects this code snippet since it has bugs in handling re-declaration of parameters and uninitialized references. As programmers can use `gcc` to compile `llvm` source code, this `gcc` bug can affect the compilation of `llvm` . To bypass the `gcc` bug, programmers of `llvm` implement the following workaround [6cd232].

```
1 - auto printComplexValue = [&](auto complexValues, auto
      printFn, raw_ostream &os, auto &&... params) {...
2 - printComplexValue(attr.getComplexFloatValues(),
      printFloatValue, os);...
3 + // This lambda was hitting a bug in gcc 9.1,9.2
4 + // and hence was replaced.
5 + if (complexElementType.isa<IntegerType>()) {...
6 +   printDenseElementsAttrImpl(attr.isSplat(), type, os,
      [&](unsigned index) {...
```

To bypass the `gcc` bug, programmers replace the lambda expression with a method invocation in the above workaround. This `gcc` bug appears in `gcc` 9.1, and its patch is applied to 9.3. As the bug is fixed and the latest `llvm` uses a more recent `gcc` to compile source code, researchers can believe that this `gcc` bug no longer affects `llvm` and the workaround should be removed. We thus submit a bug report [68407] to notify `llvm` programmers that the bug is fixed and ask them whether the workaround could be removed. However, `llvm` developers reject our suggestion since they still support the buggy versions of `gcc`. After the workaround is removed, programmers will not compile `llvm` if their `gcc` is a buggy version. Although the `gcc` bug was fixed five years ago, it still affects the latest `llvm` and other projects. The knowledge from its workaround can still apply to the compilation of many projects.

In summary, compiler bugs can significantly affect software development. Even if they are fixed, compiler bugs can affect many projects, and their workarounds can still be useful.

VII. Related Work

Empirical studies on compiler bugs. Romano *et al.* [31] investigate the challenges and characteristics of WebAssembly compiler bugs. Shen *et al.* [33] analyze the characteristics of deep learning compiler bugs and provide suggestions for detection and debugging such bugs. Wang *et al.* [37] analyze the distribution and root causes of Python interpreter bugs. Zhang *et al.* [43] analyze common types, fixes, and patterns of compiler errors in CI builds. Zhou *et al.* [47] analyze the prevalence and characteristics of optimization bugs in GCC and LLVM compilers. The above studies analyze bugs that are reported to compilers, but we analyze compiler bugs mentioned in commit messages. Zhong [45] conducted the first study about the workarounds of compiler bugs. Our study has no overlapped research questions with his study, but we refine some of his findings. For instance, Zhong [45] report the modified lines of workarounds, and we further manually classify such workarounds into categories.

Empirical studies on workarounds. Researchers have analyzed workarounds to promote the maintenance of software. Lamothe *et al.* [27] analyze API workarounds, but the workarounds for compiler bugs are unrelated to APIs. Ding *et al.* [22] analyze workarounds across projects. Yan *et al.* [41] analyze workarounds in general, and they mention compiler

bugs. Our study includes the analysis of bug reports, but we focus on the workarounds of compiler bugs.

Compiler testing. Generating test cases is an important topic in compiler testing [18], [24]. One approach is generating from scratch, which has been continuously studied since the 1970s by Hanford [24] and Seaman [32]. CSmith proposed by Yang *et al.* [42] can generate random C testing programs from scratch. Following CSmith there are various "smiths" expanding this method to other compiler languages, such as CLSmith [30] for OpenCL, Verismith [25] for Verilog. Recently, Chen *et al.* [20], [21] improve CSmith's performance with configuration tuning. Another approach is mutating existing programs to generate equivalent testing programs and then comparing the compiler's results on equivalent programs to detect bugs. There have been various static methods. Sun *et al.* [35] generate equivalent programs by mutating variable and function names. Holler *et al.* [26] mutate programs by traversing their syntax trees. And there are also dynamic approaches like the EMI, proposed by Le *et al.* [28]. EMI, at first, compiles and runs the program and then removes the unexecuted lines from existing programs to generate equivalent programs. This idea is further instantiated and improved by several researches [23], [29], [34]. Besides generating test programs, there are several empirical studies analyzing ranks of test programs [16] and bug-trigger code location of test programs [17]–[19]. The generation method is not the only way to obtain test programmers. Zhong *et al.* [44] retrieve real-world test programs from bug reports and perform differential experiments on compilers to detect bugs. Our study derives findings that can be useful for compiler testing.

VIII. Conclusion and Future Work

Although researchers have conducted various studies on compiler bugs, these studies analyze only bug reports and their patches. From such sources, it is feasible to analyze the causes, locations, and repair of compiler bugs, but it is infeasible to analyze other important angles (*e.g.*, workarounds). To enhance the understanding of compiler bugs, in this paper, we conduct an empirical study from compiler bugs mentioned in real development. From the commits from GitHub, we collected more than one thousand commits whose messages mention compiler bugs. From these commits, we analyze the characteristics of projects that encounter compiler bugs in real development. Furthermore, we manually inspected symptoms, workarounds, and their associations. We summarize our results into eight findings and interpret their significance.

There is sufficient space for follow-up research. First, it is worth exploring which types of projects are more likely to encounter compiler bugs. Second, some components of a project can be particularly susceptible to compiler bugs. Third, it is worth exploring the correlations between compiler bugs and specific code features. Finally, it is worth exploring how compiler bugs cause symptoms in their compiled code. We open a new line of studies, and such studies would offer valuable guidance to software developers and mitigate the risks associated with compiler bugs.

Acknowledgement

We appreciate reviewers for their insightful comments. This study is sponsored by National Nature Science Foundation of China No. 62232003 and No. 62272295. Hao Zhong is the corresponding author.

References

[1] https://gcc.gnu.org/bugzilla/show_bug.cgi?id=91341, 2019.
[2] https://github.com/AMReX-Combustion/PeleMP/commit/aae693cfe44949906a4ee943f9349340d65e2074, 2021.
[3] https://github.com/tsduck/tsduck/commit/b2530ca67815154038d6b93f23c19fbf69747603, 2021.
[4] https://gcc.gnu.org/onlinedocs/gcc-4.6.3/gcc/Warning-Options.html#Warning-Options, 2021.
[5] GitHub REST API documentation. https://docs.github.com/en/rest, 2022.
[6] https://github.com/sourceruckus/linux-mdl/commit/d5f6545934c47e97c0b48a645418e877b452a992, 2023.
[7] https://gcc.gnu.org/bugzilla/show_bug.cgi?id=99578, 2023.
[8] https://gcc.gnu.org/bugzilla/describekeywords.cgi, 2023.
[9] https://bugs.llvm.org/describekeywords.cgi, 2023.
[10] https://gcc.gnu.org/bugzilla/show_bug.cgi?id=51628, 2023.
[11] https://cplusplus.github.io/CWG/issues/727.html, 2023.
[12] https://github.com/spack/spack, 2023.
[13] https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html, 2023.
[14] Clang: a C language family frontend for LLVM. https://clang.llvm.org/, 2023.
[15] GCC, the GNU Compiler Collection. https://gcc.gnu.org/, 2023.
[16] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie. Test case prioritization for compilers: A text-vector based approach. In *Proc. ICST*, pages 266–277, 2016.
[17] J. Chen, J. Han, P. Sun, L. Zhang, D. Hao, and L. Zhang. Compiler bug isolation via effective witness test program generation. In *Proc. ESEC/FSE*, pages 223–234, 2022.
[18] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie. An empirical comparison of compiler testing techniques. In *Proc. ICSE*, pages 180–190, 2016.
[19] J. Chen, H. Ma, and L. Zhang. Enhanced compiler bug isolation via memoized search. In *Proc. ASE*, pages 78–89, 2020.
[20] J. Chen, C. Suo, J. Jiang, P. Chen, and X. Li. Compiler test-program generation via memoized configuration search. In *Proc. ICSE*, pages 2035–2047, 2023.
[21] J. Chen, G. Wang, D. Hao, Y. Xiong, H. Zhang, and L. Zhang. History-guided configuration diversification for compiler test-program generation. In *Proc. ASE*, pages 305–316, 2022.
[22] H. Ding, W. Ma, L. Chen, Y. Zhou, and B. Xu. An empirical study on downstream workarounds for cross-project bugs. In *Proc. APSEC*, pages 318–327, 2017.
[23] A. F. Donaldson and A. Lascu. Metamorphic testing for (graphics) compilers. pages 44–47, 2016.
[24] K. V. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 9(4):242–257, 1970.
[25] Y. Herklotz and J. Wickerson. Finding and understanding bugs in fpga synthesis tools. In *Proc. FPGA*, pages 277–287, 2020.
[26] C. Holler, K. Herzig, and A. Zeller. Fuzzing with code fragments. In *Proc. USENIX Security*, pages 445–458, 2012.
[27] M. Lamothe and W. Shang. When APIs are intentionally bypassed: An exploratory study of API workarounds. In *Proc. ICSE*, pages 912–924, 2020.
[28] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *Proc. PLDI*, pages 216–226, 2014.
[29] V. Le, C. Sun, and Z. Su. Finding deep compiler bugs via guided stochastic program mutation. In *Proc. OOPSLA*, pages 386–399, 2015.
[30] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson. Many-core compiler fuzzing. In *Proc. PLDI*, pages 65–76, 2015.
[31] A. Romano, X. Liu, Y. Kwon, and W. Wang. An empirical study of bugs in webassembly compilers. In *Proc. ASE*, pages 42–54, 2021.
[32] R. Seaman. Testing compilers of high level programming languages. *IEEE Compututer System and Technology*, pages 366–375, 1974.
[33] Q. Shen, H. Ma, J. Chen, Y. Tian, S.-C. Cheung, and X. Chen. A comprehensive study of deep learning compiler bugs. In *Proc. ESEC/FSE*, pages 968–980, 2021.

[34] C. Sun, V. Le, and Z. Su. Finding compiler bugs via live code mutation. In *Proc. OOPSLA*, pages 849–863, 2016.

[35] C. Sun, V. Le, Q. Zhang, and Z. Su. Toward understanding compiler bugs in gcc and llvm. In *Proc. ISSTA*, pages 294–305, 2016.

[36] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process*, 29(4), 2017.

[37] Z. Wang, D. Bu, A. Sun, S. Gou, Y. Wang, and L. Chen. An empirical study on bugs in python interpreters. *IEEE Transactions on Reliability*, 71(2):716–734, 2022.

[38] X. Wu, J. Yang, L. Ma, Y. Xue, and J. Zhao. On the usage and development of deep learning compilers: an empirical study on tvm. *Empirical Software Engineering*, 27(7):172, 2022.

[39] X. Xie, H. Yang, Q. He, and L. Chen. Towards understanding tool-chain bugs in the llvm compiler infrastructure. In *Proc. SANER*, pages 1–11, 2021.

[40] J. Xu, K. Lu, Z. Du, Z. Ding, L. Li, Q. Wu, M. Payer, and B. Mao. Silent bugs matter: a study of compiler-introduced security bugs. In *Proc. USENIX Security*, pages 3655 – 3672, 2023.

[41] A. Yan, H. Zhong, D. Song, and L. Jia. How do programmers fix bugs as workarounds? an empirical study on apache projects. *Empirical Software Engineering*, 28(4):96, 2023.

[42] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proc. PLDI*, pages 283–294, 2011.

[43] C. Zhang, B. Chen, L. Chen, X. Peng, and W. Zhao. A large-scale empirical study of compiler errors in continuous integration. In *Proc. ESEC/FSE*, pages 176–187, 2019.

[44] H. Zhong. Enriching compiler testing with real program from bug report. In *Proc. ASE*, pages 1–12, 2022.

[45] H. Zhong. Understanding compiler bugs in real development. In *Proc. ICSE*, page to appear, 2025.

[46] H. Zhong, Y. Yang, and J. Keung. Assessing the representativeness of open source projects in empirical software engineering studies. In *Proc. APSEC*, pages 808–817, 2012.

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