Understanding Compiler Bugs in Real Development

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

***Abstract*—Compilers are critical in development, but compiler bugs can cause hidden and serious bugs in their compiled code. To deepen the understanding of compiler bugs, in the prior empirical studies, researchers read the bug reports and patches of compilers, and analyze their causes, locations, and patterns. Although they derive many interesting findings, their studies are limited. First, as bug reports seldom explain which projects encounter compiler bugs, it is infeasible to understand the outreaching impact. Second, before compiler bugs are fixed, programmers can bypass such bugs. The bug reports of compilers do not introduce such workarounds. Finally, the distribution of compiler bugs can be distorted, since researchers and compiler developers also file bug reports.**

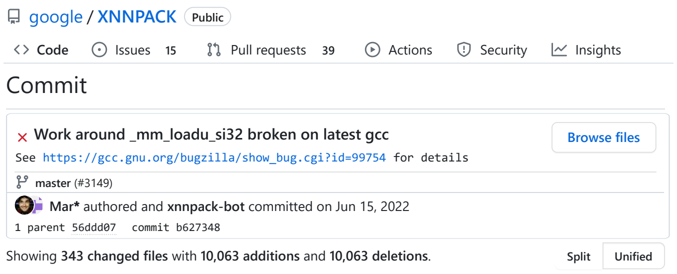
In this paper, we propose a novel angle to analyze compiler bugs. Instead of compiler bug reports, we collect compiler bugs that are mentioned in real development. When programmers encounter compiler bugs in real development, they can leave traces in their commit messages. By searching such messages, we collected 644 unique commits whose messages explicitly mention the urls of compiler bugs. From this angle, in this paper, we conduct the first empirical study to analyze compiler bugs in the wild. We summarize our results into seven useful findings for users, compiler developers, and researchers. For example, for researchers, we find that some large workarounds of compiler bugs involve repetitive and systematic changes, which indicates a new research opportunity for code migration tools. Furthermore, we attempt to apply our findings in real development, and we obtain positive feedback.

I. Introduction

As the medium between programmers and machines, compilers are critical and daily used in programming, but their bugs can lead to hidden and serious consequences [55], [59]. For example, compilers can fail to identify invalid programs, and compile valid programs to the wrong code. Both types of compiler bugs can introduce silent bugs that are difficult to detect [74], [92]. To deepen the knowledge of compiler bugs, researchers have conducted various empirical studies that analyze various compilers (*e.g.*, C compilers [86], [105], Python compilers [93], and deep learning compilers [85], [95]), and their studies focus on different perspectives (*e.g.*, buggy locations [93], repair patterns [93], [105], and causes [85], [93]).

All the prior studies analyze bug reports and patches of compilers, from which it is feasible to understand the symptoms and repairs of compiler bugs. For example, the title of a `gcc` report [17] is "`_mm_loadu_si16` and `_mm_loadu_si32` implemented incorrectly". This title introduces the symptom. A follow-up comment explains that this bug is caused by parameter orders, and another comment points to the patch:

```
- return _mm_set_epi32(*(int *)__P, (int)0,(int)0,(int)0);
+ return _mm_set_epi32(0, 0, 0, (*(__m32_u *)__P)[0]);
```



Work around _mm_loadu_si32 broken on latest gcc

See https://gcc.gnu.org/bugzilla/show_bug.cgi?id=99754 for details

master (#3149)

Mar* authored and xnnpack-bot committed on Jun 15, 2022

1 parent 56ddd07 commit b627348

Showing 343 changed files with 10,063 additions and 10,063 deletions.

(a) a commit of XNNPACK

```
src/amalgam/avx2.c
@@ -3231,19 +3231,19 ...
-...= _mm_loadu_si32(w);
+...= _mm_cvtsi32_si128(((const int*) w)[0]);
src/amalgam/avx512skx.c
@@ -3511,7 +3511,7 ...
-...= _mm_loadu_si32(params->avx512.shift);
+...= _mm_load_si128((const __m128i*) params->avx512.shift);
```

(b) the patch of XNNPACK

Fig. 1: A workaround in XNNPACK

Compilers are employed to compile source files from real projects in daily programming tasks. In this paper, we refer to such scenarios as real development. It is interesting to explore how compiler bugs affect real development, but compiler bug reports and their patches provide no such information. For example, which method will be called by real projects, and how to bypass the wrongly implemented methods? This is an inherent limitation of the prior studies. We notice that the impact can be recorded in the revision history of open-source projects. For example, we find that XNNPACK [49] encountered and bypassed this `gcc` bug. Figure 1a shows that the commit [48], and Figure 1b shows the workaround. In this commit, `_mm_loadu_si32` is intensively called. To bypass this bug, 343 files are modified. Based on this observation, compiler developers can put their effort in fixing the more frequently called method. To bypass this compiler bug, the calls of `_mm_loadu_si32` are replaced with `_mm_cvtsi32_si128` and `_mm_load_si128`. Programmers can learn the knowledge and bypass the calls if they encounter the same compiler bug.

As illustrated in the above example, revision histories provide a new angle for analyzing compiler bugs. Motivated by our example, we retrieve 644 unique commits whose messages mention compiler bugs. With the commits, we can analyze compiler bugs from the viewpoints of end users, programmers, and compiler developers. In particular, RQ1 analyzes to what degree compiler bugs affect end users and programmers. RQ2 analyzes compiler bugs from the viewpoint of programmers. For example, we analyze how many code lines are modified to implement workarounds for compiler bugs. RQ3 analyzes compiler bugs from the viewpoints of compiler developers. For example, we analyze bugs in which components are mentioned in commits. Our explored research questions are as follows:

- **RQ1. Can compiler bugs affect many (end) users?**
  Motivation. The result is useful to understand the out-reaching influence of compiler bugs.
  Protocol. In this research question, we analyze compiler bugs from the viewpoint of users and end users. Here, the users of compilers are programmers, and end users are users of project whose source files trigger compiler bugs. In GitHub, the watchers and forks of a project can measure affected programmers and end users.
  Result. Finding 1 shows that both well-known and less-known projects can encounter compiler bugs. Most compiler bugs affect about ten programmers or end users.
- **RQ2. How are compiler bugs bypassed?**
  Motivation. Programmers can bypass unfixed compiler bugs, *i.e.*, reducing their impact. The result is useful to understand how programmers live with compiler bugs.
  Protocol. In this research question, we analyze from the perspective of compiler users, *i.e.*, programmers. First, to understand why programmers bypass compiler bugs, we analyze how many mentioned compiler bugs are fixed. For each fixed compiler bug, we further count the days to fix the compiler bug. Second, we count the modified lines to implement such a workaround.
  Result. Finding 2 shows that fewer than half of the mentioned compiler bugs are fixed. Fixing most mentioned compiler bugs takes months or even years. Although programmers often have to implement workarounds, Finding 3 shows that most workarounds of compiler bugs modify only ten to twenty lines of code.
- **RQ3. Which types of compiler bugs are bypassed?**
  Motivation. For the developers of compilers, the result is useful to improve their compilers. For researchers, there can be some research opportunities.
  Protocol. In this research question, we analyze compiler bugs from the perspective of compiler developers and researchers. First, we count modifications to bypass compiler bugs by component. Second, for each component, we analyze two sample workarounds, including a large-scale modification and a small-scale modification. Finally, we derive our findings after we analyze the samples.
  Result. Finding 4 shows that the popularity of language features and the symptoms of compiler bugs determine the effort of implementing workarounds. Although most workarounds modify only ten to twenty code lines (Finding 3), some workarounds involve many repetitive and systematical changes (Finding 6). In addition, if a compiler bug hinders the main purpose of a project, programmers will take much effort to implement its workarounds (Finding 5). Meanwhile, Finding 7 shows that programmers and compiler developers can have different opinions on bugs, especially for undefined behaviors.

Compiler	Project	Commit	Bug	Fork
gcc	367	491	390	155,460
both	6	8	6	3
old llvm	91	109	75	21,940
llvm	31	36	37	8,609
total	495	644	508	186,012

TABLE I: Our dataset

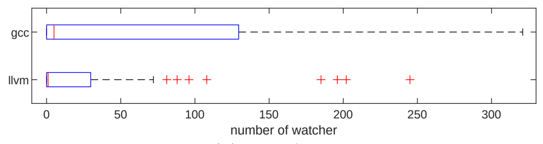
Section II introduce our support tool and dataset. Section III presents our empirical result. Section IV interprets our findings, and a vision is to connect more roles. Section V provides an example of this vision. Section VII concludes this paper.

II. Methodology

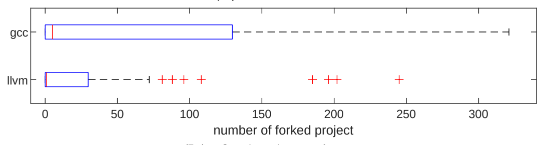
This section introduces our research methodology.

A. Tool Support

GitHub API [65] allows programmers to query the contents of Github projects. By calling this API, our tool retrieves commits whose messages mention `gcc` and `llvm` bugs. According to the formats of their bug reports, our `gcc` keyword is https://gcc.gnu.org/bugzilla/show_bug.cgi?id=, and our `llvm` keywords are https://github.com/llvm/llvm-project/issues/ and https://bugs.llvm.org/show_bug.cgi?id=. GitHub API can retrieve irrelevant results. First, some retrieved commits do not mention compiler bugs. Our tool uses the above three keywords to match commit messages. It collects a commit, only if its message contains at least a keyword. Second, GitHub API retrieves duplicated commits from forked projects. The duplicated commits are harmful to our study, since they contain no new information. Each commit is associated with a hash value. If commits have identical hash values, our tool collects only the first retrieved commit. Even if their hash values are different, commits can be duplicated. For example, from two forked Linux kernels, we find two commits such as `7157` [3] and `ef1c` [4]. Although their hash values are different, the author and the modifications of the two commits are identical. The messages of the two commits explain that the commits intend to fix an optimization issue of `llvm`. The only difference is that the message of `7157` [3] has a hash value. We search with the mentioned hash value, and GitHub API retrieves more than three thousand duplicated commits. If we do not remove the duplicated commits, our results will be seriously biased. To handle this issue, if two commits have an identical author and modifications, our tool compares their messages. If the Levenshtein similarity between them is more than 0.8, our tool considers that they are duplicated, and collects only the first commit. `gcc` and `llvm` have projects on GitHub. Their programmers can write the urls of compiler bugs in commit messages to build the links from bug reports to their fixes. As these commits are irrelevant to our study, our tool ignores commits from the two compilers and their forked projects. Some projects can add the source files of compilers to their repositories. For example, PyOMP [35] adds the source files of `llvm` to its code repository, and many of its commits are inherited from `llvm`. We manually remove the projects if they add the source files of compilers to their repositories.



(a) watchers



(b) forked projects

Fig. 2: The distributions of watchers and forked projects

B. Dataset

Table I shows our dataset. Column "Compiler" lists the compilers. Column "Project" lists the projects whose commit messages mention compiler bugs. Column "Commit" lists our collected commits. From each keyword, GitHub retrieves 100 pages of commits. After applying our filters, our tool collects 644 unique commits in total. Column "Bug" shows the number of compiler bugs mentioned in commit messages. Column "Fork" lists the forked projects of our subjects. On overage, each project has about 62 forked projects. If we do not remove forked projects, our collected dataset can be seriously polluted.

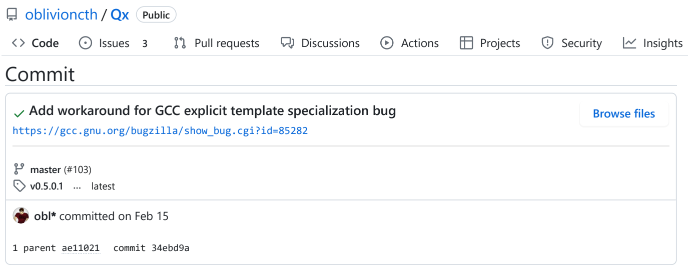
In Table I, Rows "gcc" and "llvm" list the results from the active issue trackers of `gcc` and `llvm`, respectively. Row "old llvm" lists the results from the archived `llvm` issue tracker. Here, `llvm` can compile various programming languages, *e.g.*, c++ and fortran. Row "both" lists the results when commit messages mention bugs of both compilers.

III. Empirical Result

This section presents our results. More details are listed on our project website: https://anonymous.4open.science/r/compilerbug

A. RQ1. Impact on (End) User

1) Protocol: Compiler bugs can silently affect end users and programmers. When a project is updated, the users on the watch list will be notified (*e.g.*, security alerts). A member on the watch list of a project can be an end user who is interested in this project or a programmer who works on this project. Programmers can fork projects for various considerations. For example, a programmer plans to fix a bug in a project, but this programmer has no right to modify the project. The programmer can fork and become the owner of the forked project. After the forked project is modified, it is feasible to generate a pull request for the original project. If a programmer forks a project, this user should have an interest in the development of this project. End users may not watch projects, and programmers can reuse the source files of a project without forking the project. Empirical studies are based on observations and naturally ignore instances that could not be observed. Our observations are the lower bounds. For each project in Table I, we calculate its watchers and forked projects and draw box plots to show their distributions.



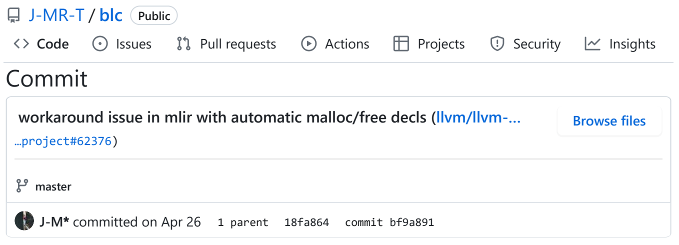
Add workaround for GCC explicit template specialization bug
https://gcc.gnu.org/bugzilla/show_bug.cgi?id=85282

master (#103)
v0.5.0.1 ... latest

obl* committed on Feb 15

1 parent ae11021 commit 34ebd9a

Fig. 3: A compiler bug encountered by Qx



workaround issue in mlir with automatic malloc/free decls (llvm/llvm-...
...project#62376)

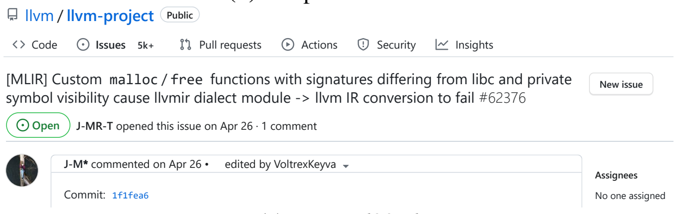
master

J-M* committed on Apr 26 1 parent 18fa864 commit bf9a891

(a) a commit of `blc`

```
// MLIR automatically inserts malloc/free declarations in
    the llvmir dialect module -> LLVM IR module conversion.
// This is annoying, because it doesn't regard custom
    declarations for malloc/free
+void workaroundAutomaticFreeMallocdecls(mlir::ModuleOp mod
    ) noexcept{...
+}
```

(b) the patch of `blc`.



[MLIR] Custom `malloc` / `free` functions with signatures differing from libc and private symbol visibility cause llvmir dialect module -> llvm IR conversion to fail #62376

Open J-MR-T opened this issue on Apr 26 · 1 comment

J-M* commented on Apr 26 • edited by VoltrexKeyva

Commit: 1f1fea6

No one assigned

(c) `llvm` 62376

Fig. 4: A compiler bug reported by `blc`

2) Result: Compiler bugs affect many (end) users, since some projects have many watchers and forks. For `gcc`, the `Linux` kernel [23] has the most watchers and forked projects. For example, a bug report complains that the `Linux` kernel fails to build due to a bug of `gcc11` [16], and in a commit [24], they disable some compilation flags to bypass this `gcc` bug. For `llvm`, `.NET Runtime` [31] has the most watchers and forked projects. In 2023, the programmers of `.NET Runtime` submitted a commit [7]. According to its commit message, the programmers bypassed a crash that was caused by a `llvm` bug [27]. This `llvm` bug was reported in 2021, but was not fixed. As the `.NET Runtime` programmers encountered the bug two years later, they are unlikely to report this `llvm` bug. Both projects are well-known and their bugs affect many end users and programmers.

Figure 2 shows the distributions of the watchers and forks. The vertical axes list the commits that mention `gcc` and `llvm` bugs, and the horizontal axes list the number of watchers and forked projects, respectively. Although famous projects like the `Linux` kernel and `.NET Runtime` have many watchers and forked projects, the results show that most of our subject projects have few watchers and forked projects. For example, `Qx` [36] is C++ UI library. It has only three watchers and two programmers. Its issue tracker has only three bug reports. Although this project is less popular, we find that compiler bugs indirectly influence its users. As shown in Figure 3, this project encounters a compiler bug, and its programmer implements a workaround. This compiler bug was reported in 2018, but is still open. From the viewpoint of users, compiler bugs can influence them even if they use less-known software. The above observations lead to a finding:

> **Finding 1.** Besides famous projects, compiler bugs can affect many regular projects.

Although many programmers never notice compiler bugs, our results show that compiler bugs can affect many projects.

B. RQ2. Repair and Workaround for Compiler bug

1) Protocol: In this research question, we analyze compiler bugs from the viewpoint of programmers. When programmers encounter compiler bugs, they can be curious about whether these compiler bugs will be fixed. We first present the distributions of fixed and open compiler bugs. If a compiler bug is fixed, we further count the days to fix compiler bugs. Compiler bugs can hinder the compilation and even cause crashes. Before a compiler bug is fixed, programmers can implement its workarounds. After a compiler bug is fixed, programmers can remove its workarounds. In both cases, the efforts are recorded in the modifications of handling commits. We count the modification of a commit as the sum of its added and deleted code lines.

2) Result: Programmers can report their encountered compiler bugs. For example, `blc` [8] is a compiler for a B-like language. As shown in Figure 4a, a programmer of `blc` encounters a `llvm` bug, and this bug automatically inserts unnecessary `malloc`/`free` declarations. Although as shown in Figure 4b, a workaround is implemented, on the same day, the `blc` programmer reports this bug to `llvm` [28]. This bug is still open, 60 days after it was reported. From the viewpoint of programmers, it is interesting to know whether their reported bugs will be fixed and how long it will take to fix their bugs.

Our tool extracts whether the mentioned compiler bugs are fixed. Figure 5a shows the results of the top components. In this figure, we group bug reports by components and merge the bug reports of similar components. For example, we merge `tree-optimization` and `rtl-optimization` of `gcc` into the `optimization` category. When they cannot determine the components, `gcc` developers put them in the `other` category, and `llvm` developers put them in the `-newbug` category. We ignore the two categories. In the new issue tracker of `llvm`, developers do not mark whether a bug is fixed. We ignore the bug reports from this tracker. As shown in Table I, only 75 bugs come from the old issue tracker, and the bugs from each component are even fewer. The result of `gcc` is more meaningful. It shows that in total, less than half of compiler bugs are fixed. In particular, compiler developers fix more bugs in `c++`, `c`, and `target` than other components.

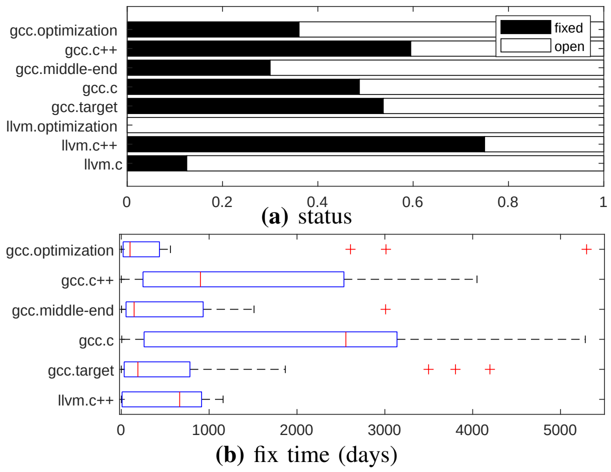


Fig. 5: The distributions of bugs

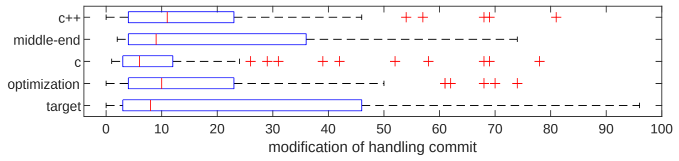
For the fixed compiler bugs, our tool extracts its reported date and fixed date. Figure 5b shows the results. The horizontal axes list the days of fixing compiler bugs. We ignore two components of `llvm`, since they have too few fixed bugs. The result shows that it takes more days to fix bugs in `c` and `c++`. It takes more than 2,500 days to fix most `c` bugs. Still, compiler developers fix more `c` and `c++` bugs. The results highlight the importance of the two components.

For both `gcc` and `llvm`, several outliers take much more time to be fixed. For `gcc`, `LibYUi` is a Widget library, and a commit [5] disabled over-sensitive warnings of `gcc` on February 17, 2023. This `gcc` bug [11] was reported in 2005 and was fixed in 2022. For `llvm`, on January 25, 2023, a commit of the `Linux` kernel [6] enables a warning check, since `llvm` fixed a bug [25]. This `llvm` bug was reported in 2013, and was fixed in 2021. Fixing the above outliers takes much more than 4,000 days. The above observations lead to a finding:

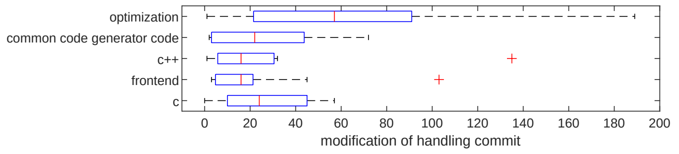
> **Finding 2.** Less than half of the mentioned compiler bugs are fixed. If they are fixed, fixing `c` and `c++` bugs takes more days than other components.

As introduced by Marcozzi *et al.* [76], some programmers (*e.g.*, [1]) criticize that it is not worth fixing compiler bugs reported by researchers. In our study, we find that even if they appear in real development, many compiler bugs are unfixed. If affected projects are actively recommended, compiler developers may repair bugs more effectively.

As many compiler bugs are unfixed, programmers have to bypass them. For each workaround, we calculate its modifications. Figure 6 shows the result of the top components. The vertical axes list components, and the horizontal axes list the number of modified code lines. The `optimization` component of `llvm` has the largest median, but all the medians are below 60. We find that several outliers modify many lines of code. For example, `XNNPACK` [49] is a library that implements optimized floating-point neural network operators. It is a popular project, and it has 1.4k stars on Github. As shown in



(a) gcc



(b) llvm

Fig. 6: The distributions of modifications

Figure 1a, a commit [48] modifies 343 files to bypass a gcc bug [17]. Despite the outliers, the medians of gcc and llvm are around 10 and 20, respectively. Programmers often apply two strategies to bypass compiler bugs. First, programmers can disable some compilation flags that cause compiler bugs. The modifications on build configuration files are typically minor. Second, as shown in Figure 4, programmers can modify their source code to bypass compiler bugs. The modifications depend on how frequently the buggy behaviors are used.

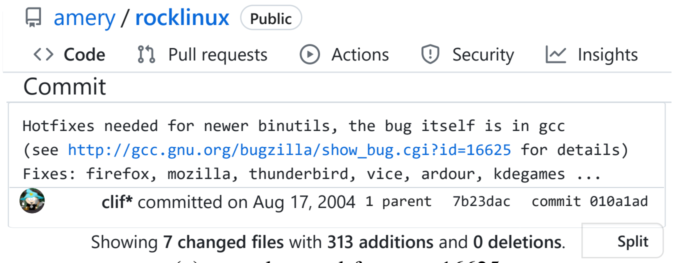
We notice that the workarounds on source files can involve systematic and repetitive changes, and such changes can be reduced. For example, bypassing the gcc bug in Figure 1 modifies many code lines of XNNPACK, but the modification of only two code lines in another project called SIMDe [37]. The programmers of SIMDe may already know the gcc bug. They implement two methods to handle the buggy gcc methods, and the method for _mm_loadu_si32 is as follows:

```
1 simde_mm_loadu_si32 (void const* mem_addr) {
2   #if defined(SIMDE_X86_SSE2_NATIVE) && ( \
3       SIMDE_DETECT_CLANG_VERSION_CHECK(8,0,0) || \
4 -     HEDLEY_GCC_VERSION_CHECK(11,0,0) || \
5       HEDLEY_INTEL_VERSION_CHECK(20,21,1))
6     return _mm_loadu_si32(mem_addr);
7   #else
8     return ...;
9   #endif
```

Line 4 of the above code checks whether the version of gcc, and it ensures that _mm_loadu_si32 is called only when gcc is not the buggy version. The other code locations call simde_mm_loadu_si32, when they need to call _mm_loadu_si32. After the gcc bug is fixed, in a SIMDe commit [38], Line 4 of the above code is deleted. The above observations lead to a finding:

> **Finding 3.** Most workarounds modify 10 to 20 code lines, but in a few extreme cases, programmers modify many lines of code to bypass compiler bugs.

In summary, less than half of the compiler bugs are fixed. Even if they are fixed, repairing compiler bugs takes months or even years. In most cases, it needs to modify only ten to twenty lines of code to bypass a compiler bug. Still, the impacts of compiler bugs shall not be underestimated. It can need much expertise to implement such workarounds.

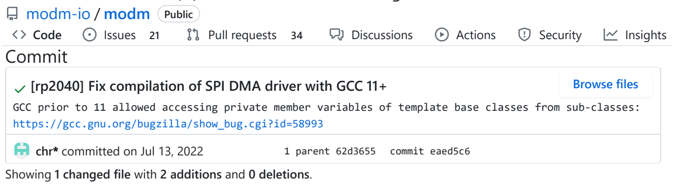


```
Hotfixes needed for newer binutils, the bug itself is in gcc
(see http://gcc.gnu.org/bugzilla/show_bug.cgi?id=16625 for details)
Fixes: firefox, mozilla, thunderbird, vice, ardour, kdegames ...
```

clif* committed on Aug 17, 2004 1 parent 7b23dac commit 010a1ad

Showing **7 changed files** with **313 additions** and **0 deletions**.

(a) a workaround for gcc 16625



✓ [rp2040] Fix compilation of SPI DMA driver with GCC 11+

```
GCC prior to 11 allowed accessing private member variables of template base classes from sub-classes:
https://gcc.gnu.org/bugzilla/show_bug.cgi?id=58993
```

chr* committed on Jul 13, 2022 1 parent 62d3655 commit eaed5c6

Showing **1 changed file** with **2 additions** and **0 deletions**.

(b) a workaround for gcc 58993

Fig. 7: The C++ bugs

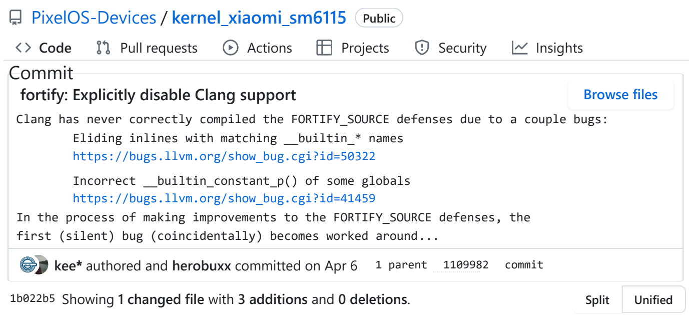
C. RQ3. Compiler Bug Category in Real Development

1) Protocol: Finding 2 shows that compiler developers fix only less than half of the mentioned compiler bugs. Due to the heavy workload, developers may not have time to fix all reported bugs. In this research question, we analyze the impact of different compiler bugs. Although researchers [81] proposed various taxonomies for compiler bugs, to assist compiler developers, we use their practical taxonomies. As the component of a bug report is the major factor in assigning the bug report [73], we classify our results by the components. The old llvm issue tracker records components of bug reports, and the current llvm issue tracker does not record components. As a result, for llvm, we analyze only the bug reports from its old issue tracker.

The impact of a compiler bug can be estimated by the number of its mentioned commits and the modifications of its workarounds. We find that in our dataset, 79.9% of gcc bugs and 76.1% of llvm bugs are mentioned only once in commits. As a result, it is infeasible for most compiler bugs to learn their importance from the first measure. Instead, we calculate the modifications of their workarounds as our measure. To present a full perspective, we introduce examples of both heavy modifications and light modifications.

2) Result: For each compiler, we present the distributions of its top five components. Although we select only five components for each compiler, these components are mentioned in 82.5% of gcc bugs and 73.1% of llvm bugs in our dataset. The percentage of a category is calculated as the commits in this category over all commits. If a category appears in both compilers, we merge their commits before our calculation. If a commit is mentioned in two compiler bugs, we count the commit for each bug once.

1. c++ (24.5%). This category includes the bugs with the C++ compiler front end and its libraries. Both gcc and llvm have this category of bugs. For gcc, we merge the bugs from c++ (the front end) and libstdc++ (the library). For example, a gcc bug of libstdc++ [15] complains that std::fabsf and std::fabsl are missing from <cmath>. For llvm, we



Commit

fortify: Explicitly disable Clang support

```
Clang has never correctly compiled the FORTIFY_SOURCE defenses due to a couple bugs:
        Eliding inlines with matching __builtin_* names
        https://bugs.llvm.org/show_bug.cgi?id=50322

        Incorrect __builtin_constant_p() of some globals
        https://bugs.llvm.org/show_bug.cgi?id=41459
In the process of making improvements to the FORTIFY_SOURCE defenses, the
first (silent) bug (coincidentally) becomes worked around...
```

kee* authored and herobuxx committed on Apr 6   1 parent 1109982   commit 1b022b5

Showing 1 changed file with 3 additions and 0 deletions.

Fig. 8: A target bug (`llvm` 41459)

merge the bugs from `c++`, `c++11`, and `c++17`.

As an example of `C++` bugs, the title of a bug report [10] is "Discarded Linkonce sections in .rodata", and a comment of this bug report provides a sample code. As shown in this sample, when compiling a `switch` statement, `gcc` crashes:

/path/to/i686-unknown-linux-gnu/bin/ld: '.gnu.linkonce.t._ZN3Foo1fEi' referenced in section '.rodata' of lib2.o: defined in discarded section '.gnu.linkonce.t._ZN3Foo1fEi' of lib2.o ... and so on

Figure 7a shows a workaround for this bug. In this commit [42], programmers systematically replace `switch` with `if` statements, if they cause crashes:

```
package/sirkull/ardour/hotfix-binutils-gcc.patch
@@ -0,0 +1,49 @@
-    switch (c) {
-    case '0': return 0;...
-    default: return -1000;
-    }
+    if ( c >= '0' && c <= '9' ) return c - '0';
+    return -1000;
```

In total, as shown in Figure 7a, this commit modifies the `switch` statements in 7 source files.

As another example of `C++` bugs, a bug report [13] complains that even if the base member is declared as `protected`, the primary template can still access it. A commit provides the following sample program:

```
1 class base {
2 private: int foo() { }
3 };
4 template <typename T>
5   struct bar : public base {
6     void test() {
7         &base::foo;   // should be rejected
8     }
9 };
```

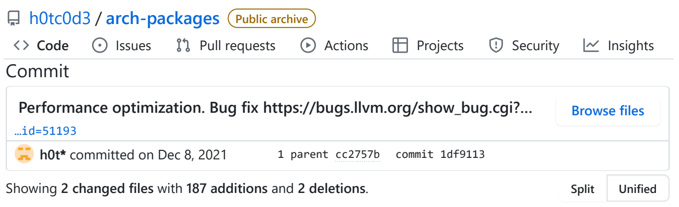
Due to this bug, `gcc` does not identify the illegal access of Line 7. To bypass this bug, a commit [41] changes a modifier from `protected` to `private`. As the situation is rare, this commit modifies only a line of code.

2. Target (16.5%). This category includes compiler bugs that occur in specific platforms. Both `gcc` and `llvm` have this category of bugs. `gcc` puts these bugs into `target`.

As an example of target bugs, a bug report [26] is specific to x-86 platforms, since `_mm_loadu_si32` is defined in Intel Intrinsics Guide [22]. A comment complains that this bug hinders `llvm` from working with the Linux kernel:

This misbehavior of __builtin_constant_p() is blocking getting the kernel's FORTIFY_SOURCE working with Clang...

In the Linux kernel, the `FORTIFY_SOURCE` macro [9] provides support for detecting buffer overflows. As a result, this



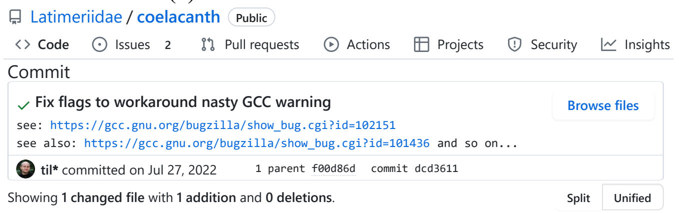
Commit

Performance optimization. Bug fix https://bugs.llvm.org/show_bug.cgi?...
…id=51193

h0t* committed on Dec 8, 2021   1 parent cc2757b   commit 1df9113

Showing 2 changed files with 187 additions and 2 deletions.

(a) a workaround for `llvm` 51193



Commit

✓ Fix flags to workaround nasty GCC warning

```
see: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=102151
see also: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=101436 and so on...
```

til* committed on Jul 27, 2022   1 parent f00d86d   commit dcd3611

Showing 1 changed file with 1 addition and 0 deletions.

(b) a workaround for `gcc` 102151

Fig. 9: The optimization bugs

bug is specific to Linux kernels. Figure 8 shows a workaround. In this commit [46], programmers also complain that `clang` does compile `FORTIFY_SOURCE` defenses. Here, `clang` is the `c` and `c++` frontends of `llvm`. To bypass the problem, this commit makes the following modification:

```
1 security/Kconfig
2 @@ -191,6 +191,9 @@ config HARDENED_USERCOPY_PAGESPAN
3 + # https://bugs.llvm.org/show_bug.cgi?id=50322
4 + # https://bugs.llvm.org/show_bug.cgi?id=41459
5 + depends on !CC_IS_CLANG
```

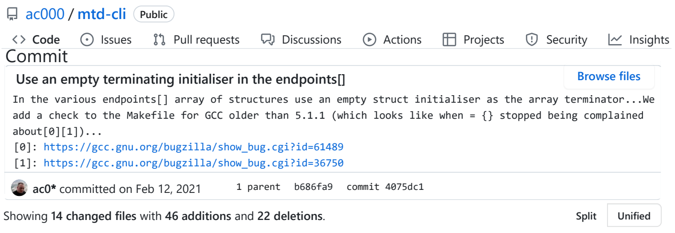
Line 5 of the above code checks whether the compiler is set as `clang` before it compiles the source files that are related to `FORTIFY_SOURCE`. The workarounds have many modifications, when compiler bugs involve frequent code structures (*e.g.*, Figure 7a) or frequent methods (*e.g.*, Figure 1). The above observations lead to a finding:

> **Finding 4.** If a compiler bug involves popular language features and causes serious symptoms, its workarounds can have many modifications.

3. Optimization (14.7%). Both `gcc` and `llvm` have bugs in this category. In `gcc`, a `tree-optimization` bug is related to the tree-ssa optimizers, and a `rtl-optimization` bug is a problem in a low-level intermediate representation called the register transfer language. We merge them to this category. The optimization bugs of `llvm` all come from `scalar optimizations`, and these bugs are related to the libraries and scalar transforms. Our collected commits do not mention other types of optimization bugs.

A `llvm` bug report [30] complains that register promotions are not optimized. Although this bug is confirmed and assigned, it is not fixed. Figure 9a shows a workaround for `arch-packages`. The purpose of `arch-packages` [2] is to optimize the packages of `Arch Linux`. Although the `llvm` bug is not fixed, the programmers of `arch-packages` find a way to bypass the problem. In particular, the code message of a commit [47] explains the details:

```
Please consider this small example:
loop {
```


ac000 / mtd-cli Public
Code Issues Pull requests Discussions Actions Projects Security Insights
Commit
Browse files
Split Unified

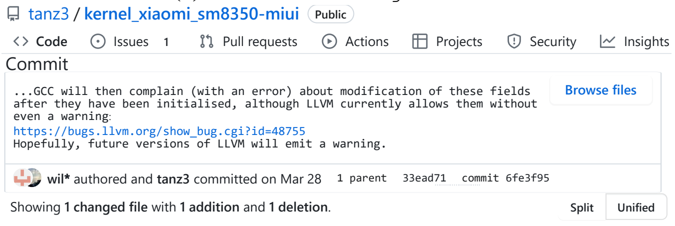
Use an empty terminating initialiser in the endpoints[]

```
In the various endpoints[] array of structures use an empty struct initialiser as the array terminator...We
add a check to the Makefile for GCC older than 5.1.1 (which looks like when = {} stopped being complained
about[0][1])...
[0]: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=61489
[1]: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=36750
```

ac0* committed on Feb 12, 2021 1 parent b686fa9 commit 4075dc1

Showing 14 changed files with 46 additions and 22 deletions.

(a) a workaround for `gcc` 36750


tanz3 / kernel_xiaomi_sm8350-miui Public
Code Issues 1 Pull requests Actions Projects Security Insights
Commit
Browse files
Split Unified

```
...GCC will then complain (with an error) about modification of these fields
after they have been initialised, although LLVM currently allows them without
even a warning:
https://bugs.llvm.org/show_bug.cgi?id=48755
Hopefully, future versions of LLVM will emit a warning.
```

wil* authored and tanz3 committed on Mar 28 1 parent 33ead71 commit 6fe3f95

Showing 1 changed file with 1 addition and 1 deletion.

(b) a workaround for `gcc` 48755

Fig. 10: The C bugs

```
  var = *ptr;
  if (var) break;
  *ptr= var + 1;
}
After this patch, it will be:
var0 = *ptr;
loop {
  var1 = phi (var0, var2);
  if (var1) break;
  var2 = var1 + 1;
  *ptr = var2;
}
This addresses some problems from [0].
[0] https://bugs.llvm.org/show_bug.cgi?id=51193
```

As shown in Figure 9a, the commit modifies two files, involving 187 additions and 2 deletions. As another example, a `gcc` bug report [19] complains spurious warnings when the `-Warray-bounds` flag is set, but `gcc` developers determine that this bug report is invalid. Still, programmers can encounter this problem. For example, as shown in Figure 9b, a commit of `coelacanth` [43] implements a workaround:

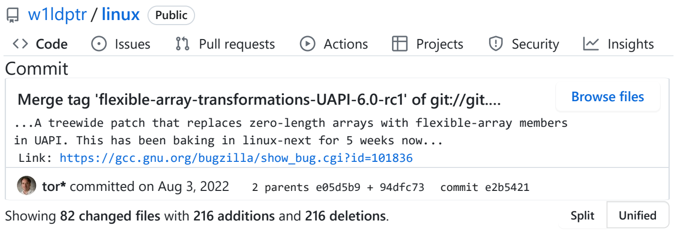
```
cmake/build-flags.cmake
@@ -18,6 +18,7 @@ if (DEFINED UNIX OR DEFINED MINGW)
+ add_compile_options(-Wno-array-bounds)# workaround
for GCC madness
```

The above workaround adds the `-Wno-array-bounds`, and it disables all warnings of `-Warray-bounds`. As the `llvm` bug [30] touches the core function of `arch-packages`, its workaround causes heavy modifications. Meanwhile, the `gcc` bug [19] produces spurious warnings that can be ignored with corresponding flags. The observations lead to another finding:

> **Finding 5.** If compiler bugs affect the core functions, their workarounds can cause heavy modifications.

4. C (12.7%). This category includes the problems with the C frontend. Both `gcc` and `llvm` have this category of bugs, and both compilers mark the components of these bugs as `c`.

As an example of C bugs, a `gcc` bug report [20] complains about spurious warnings with a flag. Figure 10a shows a workaround for this bug. As mentioned in Figure 10a, this commit modifies the compiler-configuration file:


w1ldptr / linux Public
Code Issues Pull requests Actions Projects Security Insights
Commit
Browse files
Split Unified

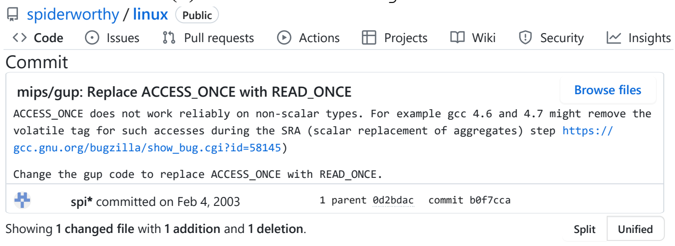
Merge tag 'flexible-array-transformations-UAPI-6.0-rc1' of git://git....

```
...A treewide patch that replaces zero-length arrays with flexible-array members
in UAPI. This has been baking in linux-next for 5 weeks now...
 Link: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=101836
```

tor* committed on Aug 3, 2022 2 parents e05d5b9 + 94dfc73 commit e2b5421

Showing 82 changed files with 216 additions and 216 deletions.

(a) a workaround for `gcc` 101836


spiderworthy / linux Public
Code Issues Pull requests Actions Projects Wiki Security Insights
Commit
Browse files
Split Unified

mips/gup: Replace ACCESS_ONCE with READ_ONCE

```
ACCESS_ONCE does not work reliably on non-scalar types. For example gcc 4.6 and 4.7 might remove the
volatile tag for such accesses during the SRA (scalar replacement of aggregates) step https://
gcc.gnu.org/bugzilla/show_bug.cgi?id=58145)

Change the gup code to replace ACCESS_ONCE with READ_ONCE.
```

spi* committed on Feb 4, 2003 1 parent 0d2bdac commit b0f7cca

Showing 1 changed file with 1 addition and 1 deletion.

(b) a workaround for `gcc` 58145

Fig. 11: The middle end bugs

```
1 src/Makefile
2 @@ -15,6 +15,17 @@...
3 +ifneq "$(GCC_VER_OK)" "1"
4 +  # For GCC < 5.1.1
5 +  CFLAGS += -Wno-missing-field-initializers
6 +endif
```

Line 5 of the above code disables the warnings. As described by the bug report, `gcc` produces these warnings when it checks the initializers of `struct`. Besides the configuration file, this commit modifies many affected initializers, and an example is as follows:

```
src/mtd-cli-biss.c
@@ -45,7 +45,8 @@...
static const struct endpoint endpoints[] = {...
-   { NULL, { NULL }, 0, 0, NULL}
+   { }
```

The initializers appear in multiple files, and in total, this commit modifies 14 files.

As another example of C bugs, a `llvm` bug [44] complains that the constant members of a `struct` can still be modified. Figure 10b shows a workaround for this bug. As shown in Figure 10b, a programmer mentions that `gcc` produces warnings for such accesses. After `llvm` programmers list several items of the C standards, they argue that `llvm` is correct, and the bug report is not fixed. Still, this behavior is strange for some programmers. In this workaround, instead of specific members, programmers add `const` to `struct`. This compiler bug affects only a `struct` datum, and the commit modifies only a code line:

```
include/linux/mm.h
@@ -493,7 +493,7 @@ struct vm_fault {
-    struct {
+ const struct {
```

The modification patterns of `C` are similar to those of `C++`, since their bugs are often shared. Indeed, Figure 10a mentions two `gcc` bugs. The two `gcc` bugs have identical symptoms, but the other bug report [45] is triggered by C++ programs.

Besides C++ and C, our commits involve other programming languages, *e.g.*, Fortran. We do not present their results, since they are not among the top five categories.

5. Middle end (9.4%). This category includes bugs in the middle end. Only `gcc` defines this component, and `llvm` puts this type of bugs in other components. The middle end of a compiler typically handles both the analysis (*e.g.*, data-flow analysis) and the optimization, but `gcc` put the problems of optimization into a separate component. A `gcc` report [18] complains that `__builtin_object_size(P->M, 1)` fails to calculate the size of a `struct` if this `struct` declares a fixed-length array at its end. A sample program is as follows:

```
struct trailing_array {
    int a;
    int b;
    unsigned char c[16];//wrong size
};
```

If `broken` is a `struct` whose type is `trailing_array`, the calls of `__builtin_object_size(broken, 1)` will return -1, but the correct result is 16. Figure 11a shows a workaround, and its strategy is to replace fixed-length arrays with flexible arrays. For example, a modification is as follows:

```
arch/m68k/include/uapi/asm/bootinfo.h
@@ -34,7 +34,7 @@
struct bi_record {...
-   __be32 data[0];
+   __be32 data[];
}
```

Similar modifications appear in other code locations:

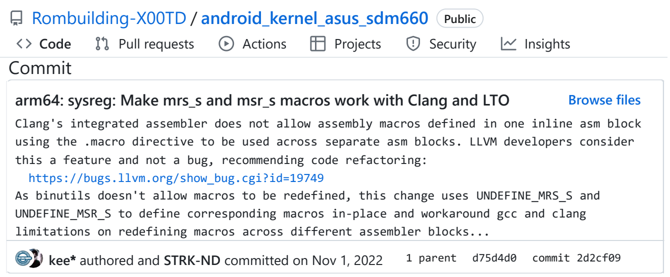
```
arch/s390/include/uapi/asm/hwctrset.h
@@ -30,18 +30,18 @@ ...
struct s390_ctrset_setdata {/* Counter set data */...
- __u64 cv[0]; /* Counter values (variable length) */
+ __u64 cv[]; /* Counter values (variable length) */
}
```

In total, as shown in Figure 11a, this workaround modifies 82 files. Besides this compiler bug, we find repetitive edits in many workarounds. For example, as shown in Figure 1, when bypassing the `target` bug, the workaround includes repetitive replacements of `_mm_loadu_si32`. The observations lead to another finding:

Finding 6. Bypassing some compiler bugs needs repetitive and systematic changes.

As another example of middle-end bugs, a bug report [12] complains that `ACCESS_ONCE` is not reliable to ensure, and a variable protected by it can be accessed more than once. Figure 11b shows a workaround. It replaces `ACCESS_ONCE` with `READ_ONCE`.

6. Frontend (2.0%). This category includes bugs in the frontend (*e.g.*, the lexer and the parser). Only `llvm` defines this component, and `gcc` puts this type of bugs in specific frontends (*e.g.*, `C` and `C++`). For example, a `gcc` bug [15] complains that the macros defined in an asm block cannot be used in other asm blocks. This bug report is marked as "won't fix", since `llvm` developers consider that this is a feature of `llvm`. Figure 12a shows a workaround for this bug. This workaround comes from a customized Linux kernel. Its programmers mention the feature of `llvm`, but point out that `binutils` does not allow macros to be redefined. As a result, they have to define two macros:


Rombuilding-X00TD / android_kernel_asus_sdm660 Public

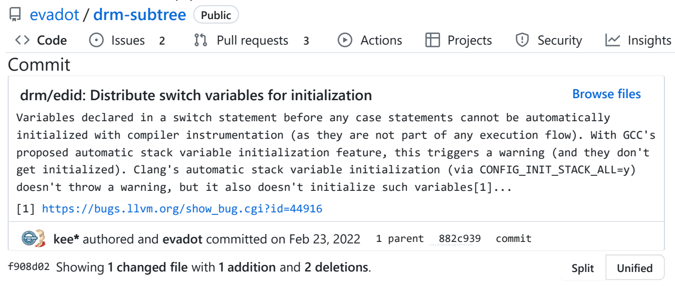
Commit

arm64: sysreg: Make mrs_s and msr_s macros work with Clang and LTO

```
Clang's integrated assembler does not allow assembly macros defined in one inline asm block
using the .macro directive to be used across separate asm blocks. LLVM developers consider
this a feature and not a bug, recommending code refactoring:
  https://bugs.llvm.org/show_bug.cgi?id=19749
As binutils doesn't allow macros to be redefined, this change uses UNDEFINE_MRS_S and
UNDEFINE_MSR_S to define corresponding macros in-place and workaround gcc and clang
limitations on redefining macros across different assembler blocks...
```

kee* authored and STRK-ND committed on Nov 1, 2022 1 parent d75d4d0 commit 2d2cf09

(a) a workaround for `llvm` 19749


evadot / drm-subtree Public

Commit

drm/edid: Distribute switch variables for initialization

```
Variables declared in a switch statement before any case statements cannot be automatically
initialized with compiler instrumentation (as they are not part of any execution flow). With GCC's
proposed automatic stack variable initialization feature, this triggers a warning (and they don't
get initialized). Clang's automatic stack variable initialization (via CONFIG_INIT_STACK_ALL=y)
doesn't throw a warning, but it also doesn't initialize such variables[1]...
[1] https://bugs.llvm.org/show_bug.cgi?id=44916
```

kee* authored and evadot committed on Feb 23, 2022 1 parent 882c939 commit f908d02

Showing 1 changed file with 1 addition and 2 deletions.

(b) a workaround for `llvm` 44916

Fig. 12: The frontend bugs

```
arch/arm64/include/asm/sysreg.h
@@ -232,29 +232,44 @@
+ #define DEFINE_MRS_S ...
+ #define UNDEFINE_MRS_S ...
```

As `llvm` does not allow to use macros in other asm blocks, programmers have to define `DEFINE_MRS_S` inside asm blocks. Meanwhile, as `binutils` does not allow redefined macros, they have to undefine with `UNDEFINE_MRS_S`. An example is as follows:

```
arch/arm64/include/asm/cputype.h
@@ -119,7 +119,10 @@...
- asm("mrs_s  %0, " __stringify(reg) : "=r" (__val));
+ asm(DEFINE_MRS_S
    "mrs_s  %0, " __stringify(reg) "\n"
    UNDEFINE_MRS_S...)
```

We find similar modifications in other code locations. This observation further confirms Finding 6, *i.e.*, bypassing compiler bugs involves repetitive edits.

Another `llvm` bug [29] complains that `llvm` fails to initialize a variable if it is defined in `switch` statements. Figure 12b shows a workaround for this bug. Its programmer mentions that this bug is shared by `gcc`, and proposes a workaround:

```
core/drm_edid.c
@@ -4438,6 +4438,7 @@ ...
+ int sad_count;
switch (cea_db_tag(db)) {
- int sad_count;...
```

In the above code, the `sad_count` variable is moved outside of the `switch` statement. Many researchers are interested in only fixed bugs. In this section, we introduce 11 compiler bugs, but only 5 of them are resolved as fixed. The ratio is consistent with that of Figure 5a. Meanwhile, the bug report mentioned in Figure 9b is resolved as invalid, and the bug report mentioned in Figure 12a is resolved as won't fix. Many researchers believe that only fixed bug reports are useful, our observations lead to another finding:

Finding 7. Invalid and unfixed bug reports can still be useful to understand compiler bugs.

In summary, although the components have a minor impact on the modifications of workarounds, their instances provide insights into compiler bugs. We summarize them into several findings, and will interpret our findings in the next section.

D. Threat to Validity

The internal threat to validity includes our underlying Github API. It retrieves only a subset of all matches. In addition, its retrieved results can be irrelevant and duplicated. To reduce the threat, our tool checks whether compiler bugs are truly mentioned in retrieved commits and removes duplicated commits. The internal threat to validity also includes the wrong status of bug reports, since some bug reports can be still open after their bugs are already fixed. The prior studies [86], [93], [105] calculate the time of fixing compiler bugs as we did, and they also suffer from this threat. As compiler bug reports are strictly managed, such cases should be rare.

The external threat of our study is shared by all empirical studies. Our study is a bite of the time, and 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 handling 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 in a shorter time.

IV. Interpretation

This section interprets our findings.

Connecting more roles. The prior studies analyze compiler bugs from only the viewpoint of compiler developers. We find that programmers and compiler developers can have different opinions on bugs, but their communication is not smooth. For example, the bug report mentioned in Figure 9b is marked as invalid. Although they are not compiler developers, some programmers have rich knowledge of languages and compilers. For example, the commit in Figure 9a implements a workaround to enable the optimization of compilers. If more roles are actively connected in developing compiler bugs, it could be easier to understand the significance of compiler bugs. In addition, Zhong [102] mentions that the confusion about compiler bugs can be caused by undefined behaviors. If more roles are invited to draft language standards, it could resolve such undefined behaviors.

Learning from workarounds for other programmers. As shown in Figure 1, some workarounds modify thousands of lines of code. The huge effort can be reduced, since Finding 6 shows that workarounds of compiler bugs often contain repetitive and systematic changes. Researchers [50], [77], [83] have proposed various approaches that learn edit scripts from given change examples. Still, their approaches are insufficient. They are designed for handling API breaking changes in Java. In contrast, the workarounds for compiler bugs are written in C or C++ and contain more code structure changes than API changes. In addition, even if the workarounds in Figures 11a and 12a involve heavy modifications, the corresponding compiler bugs are not fixed. The symptoms of some compiler bugs are difficult to notice. For example, the `llvm` bug mentioned in Figure 10b can cause illegal access to sensitive data, but such vulnerability issues can be hard to detect. As a result, a project can suffer from compiler bugs, but its programmers are not aware of such bugs. A tool can be useful, if it can determine whether a project can be affected by compiler bugs.

Learning workarounds for compiler developers. Finding 2 shows that less than half of mentioned compiler bugs are fixed. If critical bugs are ignored, programmers can switch compilers as the programmers of OpenMandriva did [14]. Compiler developers may not have sufficient resources to fix all reported bugs, and they have to be focused. To identify critical bugs, compiler developers should check both popular and regular projects, since Finding 1 shows that many compiler bugs are mentioned in regular projects. Finding 4 shows that their popularity of language features is useful in determining the impact of compiler bugs. Dyer *et al.* [63] conduct a large-scale study to understand the popularity of language features. With the rapid development of languages, languages present many new features, and the popularity can change over time. It can be a timely request to replicate their study. In Section III-C, we report how compiler bugs hamper real development. Even if compiler bugs are bypassed, their workarounds may still introduce hidden bugs. For example, in Figure 1, the workaround involves many `cast` statements, which can cause precision-related bugs. After compiler developers fix bugs, a tool can actively notify programmers to avoid such hidden bugs. Finding 7 highlights the importance of controversial compiler bugs. Compiler developers can rethink their decisions and resolve the issues raised by programmers in their real development.

V. The Story of Webpp

The interpretations in Section IV are actionable. For example, we advocate connecting more roles in testing compilers. In this section, we introduce our story of fulfilling this vision. `webpp` [39] is a C++ web framework. This project has more than four thousand commits. Among them, we notice a commit [40], whose message is "fixed a clang bug". All the modifications of this commit are related to templates. For example, a modification is as follows:

```
- template <template <typename> typename Concept,
+ template <template <typename...> typename Concept,
```

We suspect that this compiler bug is related to templates, but the message does not describe the symptom or the steps to trigger the bug. In addition, many commits do not compile when they are checked out [91]. To fix the problem, we must modify build files and even source files. The modifications make it more difficult to reproduce compiler bugs, since they can accidentally ignore compiler bugs that reject valid code. As outsiders, we failed to reproduce the compiler bug.

As researchers, we failed determine this compiler bug, so we reported the problem to both `webpp` [34] and `llvm` [33]. The `webpp` programmer submitted this commit in 2021, but we submitted the `webpp` bug report in 2022. As the commit was submitted a year ago, it took some time for the `webpp` programmer to recall the details. Still, this programmer was able to produce the compilation output:

```
FAILED: tests/CMakeFiles/test-cookies.dir/cookies_test.cpp.o
template <typename... T>
                      ^
```

It looks like a reject-valid bug, since `Clang 14` failed to build a legal template. The programmer reduced the program:

```
template <template <typename> typename T>
struct one {
  using type = T<int>;
};
template <typename...T>
struct two {
};
auto main() -> int {
  using type = one<two>;
  return 0;
}
```

The programmer even built a `godbolt` link [21]. According to this link, `clang` rejects the above program but `gcc` accepts it. Although he identified the bug, this programmer is confused about whether it is a `gcc` or `clang` bug, and left a message:

It seems like it's a GCC bug that it's a bit more permissive than mavc and clang.

To resolve the confusion, in our `llvm` bug report, we added a link to our `webpp` bug report. After `llvm` developers checked the reduced program written by the `webpp` programmer and other details of this bug, they confirmed that this is a tough bug. In particular, a `llvm` programmer explained that he tried to repair this bug, but his initial repair was reverted. He is still working on this problem.

Our `webpp` story illustrates the benefits of connecting programmers, compiler developers, and researchers. Although the `webpp` programmer identified the bug, he was unsure whether this was a `clang` bug or a `gcc` bug. As researchers, although we found this commit, we could not reproduce the mentioned bug since this commit provided insufficient details. Although this bug was known, only after we reported this issue, `llvm` developers could learn its impact on real development, since the `webpp` project provided a concrete example to trigger this bug. Indeed, as our report highlighted this bug, a `llvm` developer added our bug report to `C++ 20 in Clang` [32], and its repairing process could be checked.

VI. Related Work

Compiler testing. Random-based approaches generate random programs for compiler testing. Early approaches can be traced back to 1970s [67], [84]. Yang *et al.* [99] propose CSmith that generates random C programs. Nagai *et al.* [78] generate random C arithmetic expressions to detect arithmetic optimization bugs. Zhang *et al.* [100] propose an approach to identify those equivalent compiler test programs. Chen *et al.* [60], [61] propose approaches to tune CSmith automatically. Even-Mendoza *et al.* [64] loosen the restrictions of CSmith to detect more bugs. Mutation-based approaches modify given programs to generate compiler more test programs. Sun *et al.* [86] mutate variable and function names to generate programs. Holler *et al.* [68] mutate programs by traversing their syntax trees. Given a program, Le *et al.* [71], [72] compile the program, and execute the compiled code to collect its executed source lines. After that, their approach removes unexecuted source lines, and recompiles the remaining code. As test programs can be large, researchers have proposed approaches to narrow down code snippets that trigger compiler bugs by delta debugging [82], mutation testing [58], and coverage [56]. Researchers combine the reduction and the rank of test programs [62] and conduct empirical studies to compare existing approaches [57]. Zhong [102] proposes to extract test programs from bug reports of similar compilers. Marcozzi *et al.* [76] compare existing compiler fuzzing tools. Our study shows that commits can be another source to extract real test programs for compiler testing.

Analyzing bug reports and related software artifacts. Xu *et al.* [96] analyze which bugs will be introduced if compiler bugs are not fixed. Bettenburg *et al.* [54] analyze factors that contribute to a good bug report. A hot research line is to identify duplicate bug reports [79], [87], [89], and the other hot research line is to assign bug reports [51], [52], [97]. Tian *et al.* [90] build the priority list of bug reports. Bachmann *et al.* [53] and Wu *et al.* [94] build the links between bug reports and bug fixes. Zhou *et al.* [104] locate buggy files of a given bug report. ComBug extracts test inputs from bug reports, complementing these approaches. Researchers [51], [66] conducted various empirical studies to understand the characteristics of bug reports. Besides bug reports themselves, by linking bug reports with their fixes, researchers analyze the characteristics of bugs [69], [101], flaky tests [75], [88], and bug fixes [80], [103]. Our study analyzes compiler bugs, complementing the above studies. Lamothe *et al.* [70] analyze API workarounds, but the workarounds for compiler bugs are unrelated to APIs. Yan *et al.* [98] analyze workarounds in general, and they mention compiler bugs. We conduct a more comprehensive study on compiler bugs.

VII. Conclusion

Although researchers have conducted various studies to analyze compiler bugs, their studies do not touch on the impact or the handling strategies of compiler bugs. Previous studies analyze compiler bug reports and their modifications, but these sources are insufficient to answer the new questions. In this study, we analyze compiler bugs from an unvisited source. We use the url of compiler bugs to search Githbub, and retrieve the commits whose messages mention compiler bugs. In this way, we collect compiler bugs that programmers encountered in their real development, and conduct the first empirical study on compiler bugs in the wild. Our study deepens the knowledge about compiler bugs from a new angle, and we identify several promising research opportunities. For example, we find that the workarounds of compiler bugs can contain repetitive and systematical changes. Existing tools can learn how to edit scripts from these workarounds. Leaned scripts can be systematically applied to all feasible code locations.

Acknowledgement

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

References

[1] So you found a bug in my compiler: Whoopee do. http://shape-of-code.coding-guidelines.com/2015/12/07/so-you-found-a-bug-in-my-compiler-whoopee-do/, 2015.
[2] arch-packages. https://github.com/h0tc0d3/arch-packages, 2023.
[3] commit 7157. https://github.com/fcuzzocrea/android_kernel_samsung_universal990/commit/7157ec402d8d7ebee4c4a5749461991f2c6c6bc8, 2023.
[4] commit ef1c. https://github.com/kastentop2005/kernel_xiaomi_camellia/commit/ef1cbbd3b27bce07382175a7bf68597b574354c4, 2023.
[5] A compiler bug from libyui. https://github.com/libyui/libyui/commit/d681ff007d48456ce73aa6e9856fb61cc4b7d294, 2023.
[6] A compiler bug from linux. https://github.com/w-simon/linux-stable-rt_dev/commit/cb2dcd4844d342e3dd9be3b564ceb74f08eb323a, 2023.
[7] A compiler bug from .net. https://github.com/dotnet/runtime/commit/e92289a03ef47cd3ea2ada2cfe1c0c3abd04f196, 2023.
[8] A compiler for a b-like language. https://github.com/J-MR-T/blc, 2023.
[9] Fortify_source. https://www.redhat.com/en/blog/enhance-application-security-fortifysource, 2023.
[10] Gcc 16625. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=16625, 2023.
[11] Gcc 20423. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=20423, 2023.
[12] Gcc 58145. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=58145, 2023.
[13] Gcc 58993. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=58993, 2023.
[14] GCC 59480. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=59480, 2023.
[15] Gcc 79700. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=79700, 2023.
[16] GCC 99652. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=99652, 2023.
[17] GCC 99754. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=99754, 2023.
[18] Gcc101836. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=101836, 2023.
[19] Gcc102151. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=102151, 2023.
[20] Gcc36750. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=36750, 2023.
[21] The goldbolt example. https://godbolt.org/z/qPqbPjEa7, 2023.
[22] Intel intrinsics guide. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#text=_mm_loadu_si32&expand=5236, 2023.
[23] Linux kernel. https://github.com/torvalds/linux, 2023.
[24] A linux kernel commit that mentinos a gcc bug. https://github.com//torvalds/linux/commit/aabf6155dfb83262ef9a10af4bef945e7aba9b8e, 2023.
[25] Llvm 16428. https://bugs.llvm.org/show_bug.cgi?id=16428, 2023.
[26] llvm 41459. https://bugs.llvm.org/show_bug.cgi?id=41459, 2023.
[27] Llvm 50611. https://bugs.llvm.org/show_bug.cgi?id=50611, 2023.
[28] Llvm 62376. https://github.com/llvm/llvm-project/issues/62376, 2023.
[29] Llvm44916. https://bugs.llvm.org/show_bug.cgi?id=44916, 2023.
[30] Llvm51193. https://bugs.llvm.org/show_bug.cgi?id=51193, 2023.
[31] .NET Runtime. https://github.com/hub4j/github-api, 2023.
[32] Our bug report in c++20 in clang. https://github.com/orgs/llvm/projects/14/views/1?filterQuery=55894, 2023.
[33] Our llvm bug report. https://github.com/llvm/llvm-project/issues/55894, 2023.
[34] Our webpp bug report. https://github.com/the-moisrex/webpp/issues/132, 2023.
[35] Pyomp. https://github.com/Python-for-HPC/PyOMP, 2023.
[36] The qx library. https://github.com/oblivioncth/Qx, 2023.
[37] Simde. https://github.com/simd-everywhere/simde, 2023.
[38] A simde commit. https://github.com/simd-everywhere/simde/commit/4b7394fc2609c70540772769092014bf970a0b43, 2023.
[39] webpp. https://github.com/the-moisrex/webpp, 2023.
[40] A webpp commit. https://github.com/the-moisrex/webpp/commit/97a59475d9ff73ce142010215d59da5803e464be, 2023.
[41] A workaround of gcc 58893. https://github.com/modm-io/modm/blob/eaed5c60d00d16974e1410d4660c6d1fcd8dca82/src/modm/platform/spi/rp/spi_master.hpp.in, 2023.
[42] A workaround of gcc 79700. https://github.com/amery/rocklinux/commit/010a1adf8d1d5cfcb08326ab2ad4cb1590fceb5b, 2023.
[43] A workaround of gcc102151. https://github.com/Latimeriidae/coelacanth/commit/dcd3611322fccf348101aa538ffffe145e7cb7c7, 2023.
[44] A workaround of gcc48755. https://bugs.llvm.org/show_bug.cgi?id=48755, 2023.
[45] A workaround of gcc61489. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=61489, 2023.
[46] A workaround of llvm41459. https://github.com/PixelOS-Devices/kernel_xiaomi_sm6115/commit/1b022b5335edaa68b7d1a7432db45d89729f13fa, 2023.
[47] A workaround of llvm51193. https://github.com/h0tc0d3/arch-packages/commit/1df91131601ad276beaaae16339ac557571833aa, 2023.
[48] A workaround of xnnpack. https://github.com/google/XNNPACK/commit/b627348de07cf64e3cd7035105369c4a8a602dbb, 2023.
[49] XNNPACK. https://github.com/google/XNNPACK, 2023.
[50] J. Andersen and J. L. Lawall. Generic patch inference. *Automated software engineering*, 17(2):119–148, 2010.
[51] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proc. ICSE*, pages 361–370, 2006.
[52] J. Anvik and G. C. Murphy. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. *ACM Transactions on Software Engineering and Methodology*, 20(3):10, 2011.
[53] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein. The missing links: bugs and bug-fix commits. In *Proc. ESEC/FSE*, pages 97–106, 2010.
[54] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *Proc. FSE*, pages 308–318, 2008.
[55] A. S. Boujarwah and K. Saleh. Compiler test case generation methods: a survey and assessment. *Information and software technology*, 39(9):617–625, 1997.
[56] 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.
[57] 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.
[58] J. Chen, H. Ma, and L. Zhang. Enhanced compiler bug isolation via memoized search. In *Proc. ASE*, page to appear, 2020.
[59] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang. A survey of compiler testing. *ACM Computing Surveys*, 53(1):1–36, 2020.
[60] J. Chen, C. Suo, J. Jiang, P. Chen, and X. Li. Compiler test-program generation via memoized configuration search. In *Proc. ICSE*, page to appear, 2023.
[61] 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.
[62] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr. Taming compiler fuzzers. In *Proc. PLDI*, pages 197–208, 2013.
[63] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen. Mining billions of ast nodes to study actual and potential usage of java language features. In *Proc. ICSE*, pages 779–790, 2014.
[64] K. Even-Mendoza, C. Cadar, and A. F. Donaldson. Csmithedge: more effective compiler testing by handling undefined behaviour less conservatively. *Empirical Software Engineering*, 27(6):129, 2022.
[65] GithubAPI. Github api for java. https://github-api.kohsuke.org, 2023.
[66] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy. Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In *Proc. ICSE*, pages 495–504, 2010.
[67] K. V. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 9(4):242–257, 1970.
[68] C. Holler, K. Herzig, and A. Zeller. Fuzzing with code fragments. In *Proc. $USENIX$*, pages 445–458, 2012.

[69] L. Jia, H. Zhong, X. Wang, L. Huang, and X. Lu. The symptoms, causes, and repairs of bugs inside a deep learning library. *Journal of Systems and Software*, 177:110935, 2021.

[70] M. Lamothe and W. Shang. When APIs are intentionally bypassed: An exploratory study of API workarounds. In *Proc. ICSE*, pages 912–924, 2020.

[71] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *Proc. PLDI*, pages 216–226, 2014.

[72] V. Le, C. Sun, and Z. Su. Finding deep compiler bugs via guided stochastic program mutation. In *Proc. OOPSLA*, pages 386–399, 2015.

[73] Z. Lin, F. Shu, Y. Yang, C. Hu, and Q. Wang. An empirical study on bug assignment automation using chinese bug data. In *Proc. ESEM*, pages 451–455, 2009.

[74] K. Lu, C. Song, T. Kim, and W. Lee. Unisan: Proactive kernel memory initialization to eliminate data leakages. In *Proc. CCS*, pages 920–932, 2016.

[75] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In *Proc. ESEC/FSE*, pages 643–653, 2014.

[76] M. Marcozzi, Q. Tang, A. F. Donaldson, and C. Cadar. Compiler fuzzing: How much does it matter? In *Proc. OOPSLA*, pages 1–29, 2019.

[77] N. Meng, M. Kim, and K. S. McKinley. Systematic editing: generating program transformations from an example. In *Proc. PLDI*, pages 329–342, 2011.

[78] E. Nagai, H. Awazu, N. Ishiura, and N. Takeda. Random testing of C compilers targeting arithmetic optimization. In *Proc. SASIMI 2012*, pages 48–53, 2012.

[79] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *Proc. ICSE*, pages 70–79, 2012.

[80] J. Park, M. Kim, B. Ray, and D.-H. Bae. An empirical study of supplementary bug fixes. In *Proc. MSR*, pages 40–49, 2012.

[81] A. Rahman, D. B. Bose, F. L. Barsha, and R. Pandita. Defect categorization in compilers: A multi-vocal literature review. *ACM Computing Surveys*, 56(4):1–42, 2023.

[82] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for c compiler bugs. In *Proc. PLDI*, pages 335–346, 2012.

[83] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann. Learning syntactic program transformations from examples. In *Proc. ICSE*, pages 404–415, 2017.

[84] R. Seaman. Testing compilers of high level programming languages. *IEEE Compututer System and Technology*, pages 366–375, 1974.

[85] 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.

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

[87] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo. A discriminative model approach for accurate duplicate bug report retrieval. In *Proc. ICSE*, pages 45–54, 2010.

[88] S. Thorve, C. Sreshtha, and N. Meng. An empirical study of flaky tests in android apps. In *Proc. ICSME*, pages 534–538, 2018.

[89] F. Thung, P. S. Kochhar, and D. Lo. Dupfinder: integrated tool support for duplicate bug report detection. In *Proc. ASE*, pages 871–874, 2014.

[90] Y. Tian, D. Lo, X. Xia, and C. Sun. Automated prediction of bug report priority using multi-factor analysis. *Empirical Software Engineering*, 20(5):1354–1383, 2015.

[91] 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.

[92] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proc. SOSP*, pages 260–275, 2013.

[93] 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.

[94] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung. Relink: recovering links between bugs and changes. In *Proc. ESEC/FSE*, pages 15–25, 2011.

[95] 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.

[96] 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.

[97] J. Xuan, H. Jiang, Y. Hu, Z. Ren, W. Zou, Z. Luo, and X. Wu. Towards effective bug triage with software data reduction techniques. *IEEE Transactions on Knowledge and Data Engineering*, 27(1):264–280, 2015.

[98] 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*, page to appear, 2023.

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

[100] Q. Zhang, C. Sun, and Z. Su. Skeletal program enumeration for rigorous compiler testing. In *Proc. PLDI*, pages 347–361, 2017.

[101] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang. An empirical study on tensorflow program bugs. In *Proc. ISSTA*, pages 129–140, 2018.

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

[103] H. Zhong and N. Meng. Towards reusing hints from past fixes -an exploratory study on thousands of real samples. *Empirical Software Engineering*, 23(5):2521–2549.

[104] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *Proc. ICSE*, pages 14–24, 2012.

[105] 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.