How do Compiler Bugs Affect OS Kernels

Anonymous Author(s)

Abstract

As its name indicates, an operating system (OS) kernel is the core software of operating systems. Like other software, compilers are critical in the development of OS kernels. A recent study shows that compiler bugs can silently introduce bugs into their compiled code. Although compiler bugs are critical in the development of OS kernels, most studies on compiler bugs analyze the characteristics of only compiler bugs. Their findings are less interesting to outsiders, *e.g.*, the OS community. Although two recent papers have started to analyze how compiler bugs affect the development of other projects, no study has been conducted to explore how such bugs affect OS kernels. As a result, many questions are still open. For instance, in the development of OS kernels, are most compiler bugs identified by OS programmers themselves? When bypassing compiler bugs, what is the relationship between compiler-bug symptoms and modified OS components? The answers to the above questions are useful for both compiler and OS programmers.

To answer the above questions, we conducted the first empirical study on how compiler bugs affect OS kernels. We collected 494 workarounds of compiler bugs from the revision histories of 7 popular OS kernels. We analyze these workarounds and explore the answers to four research questions. We summarized our answers into 9 interesting findings. For instance, we found that in most workarounds, compiler bugs are not reported by OS programmers. As another example, we find that the libraries and make files of OS kernels are the most affected per code line. Furthermore, we provide our interpretations of our findings for programmers and researchers.

CCS Concepts: • **Software and its engineering → Software defect analysis**; • **General and reference → Empirical studies**; • **Security and privacy → Operating systems security**; **Vulnerability management**.

Keywords: Impact of bugs, compiler bugs, and operating system kernel bugs

1 Introduction

Operating system (OS) kernels are the core software for managing hardware and software resources like CPU, memory, and devices [25]. Besides managing resources, OS kernels provide many application programming interfaces (APIs) for applications. Like other software, compilers are critical in the development of OS kernels. For instance, `gcc` and `clang` are widely used to compile Linux kernels. Although compilers are essential, their bugs can hinder development and introduce hidden bugs that are difficult to detect [24, 28]. For instance, compilers can compile source code into wrong machine code, *i.e.*, wrong-code bugs. It is difficult for programmers to detect bugs that are caused by compiler bugs, especially when OS kernels have millions of lines of code. In this way, as reported by Xu *et al.* [44], compiler bugs silently introduce bugs into their compiled code. In particular, they analyze the Linux kernel code and report examples where compiler bugs introduce bugs into OS kernels.

Due to the importance of compiler bugs, researchers [37, 38, 42] conducted various empirical studies to understand compiler bugs. In particular, by analyzing compiler bug reports and their patches, researchers summarize findings about the buggy locations [42], repair patterns [42, 47], and causes [38, 42] for compiler bugs. Although these findings are useful, their audience is narrow, and only the developers of compilers and researchers can use such findings in detecting and repairing compiler bugs.

Recently, researchers [31, 46] have started to analyze how compiler bugs affect the development of other projects. These studies are beneficial for both compilers and other projects. For instance, Zhong [46] finds that about half of compiler bugs in real development are unfixed, but He and Zhong [31] report that the most frequently fixed types of compiler bugs are not the most frequent compiler bugs that programmers encounter in real development. Compiler developers can reschedule their resources according to our findings. As another example, He and Zhong [31] report the patterns of bypassing compiler bugs. Programmers can learn how to bypass compiler bugs. Although their findings are beneficial, they do not analyze how compiler bugs affect the development of OS kernels. As a result, many questions are still open. For instance, are most compiler bugs identified and reported by OS kernel programmers themselves, since they are experts? When compiler bugs are bypassed, what are the relationships between compiler-bug symptoms, modified OS components, and the required modifications?

The answers to the above open questions are useful for both OS and compiler programmers. In this paper, we conduct the first study to explore how compiler bugs affect the development of OS kernels. In total, we collected 494 compiler-bug workarounds from the revision histories of seven popular OS kernels. Our study explores the following research questions:

- **RQ1: Who reports the compiler bugs?**
  Motivation. This research question clarifies the knowledge flow directions of bypassing compiler bugs. The answers are useful for understanding the roles of compiler and OS kernel programmers.
  Protocol. We extract compiler-bug reporters and the

programmers who implement OS kernel workarounds for compiler bugs and compare them.
Result. Finding 1 shows that most compiler bugs are not reported by OS programmers, and Finding 2 shows that OS programmers care more about wrong-code and optimization compiler bugs.

- **RQ2: How long does it take from reporting bug reports to implementing their workarounds?**
  Motivation. This research question measures the time interval between reporting compiler bugs and bypassing such bugs in OS kernels. The answers are useful for understanding how long compiler bugs cause bad effects on OS kernels.
  Protocol. From each pair of a compiler bug and an OS kernel workaround, we extract the reporting time of the compiler bug report and the commit time of the workaround. After that, we calculate the time interval to measure the impact.
  Result. Finding 3 shows that it takes 10 times as many days to identify and bypass compiler bugs when they are not reported by OS programmers. Finding 4 shows that crashes, optimization, and wrong-code compiler bugs have shorter time intervals.
- **RQ3: How similar are the code samples from bug reports and the real code from OS workarounds?**
  Motivation. This research question explores the similarity between the code samples in a compiler bug report and the modified source files in its corresponding OS kernel workaround. The answers are useful for reducing the triggering code of compiler bugs and recommending relevant compiler bugs.
  Protocol. From each pair of a compiler bug and an OS kernel workaround, we compare the code sample of the compiler bug report with the modified source files of the OS kernel workaround. We calculate the similarity value and present the distribution.
  Result. Finding 5 shows that most of the code samples in bug reports are dissimilar to OS code, but a few code samples of bug reports are quite similar. Finding 6 shows that the code samples of wrong-code bugs are more different from the triggering code from OS kernels than other symptoms.
- **RQ4: What is the relationship between compiler-bug symptoms, modified OS components, and modified lines of workarounds?**
  Motivation. This research question focuses on how different symptoms affect OS kernel components and the modified code lines of corresponding workarounds. The answers are useful for allocating the resources to handle the impact of compiler bugs.
  Protocol. We group the workarounds by their modified OS kernel components and calculate the modified code lines of each group.
  Result. Finding 7 shows compiler bugs tend to affect larger components and the build files. Finding 8 shows that workarounds in the development, the library, and the kernel core have more modifications, and some modifications are repetitive. Finding 9 shows that the library and the development are affected by compiler bugs with more symptoms, and bypassing crashes and wrong code modifies more lines.

None of our RQs is explored by the prior studies [31, 46], but they have overlapped analysis. Our RQ4 and Zhong [46] calculate modified code lines of compiler-bug workarounds. Zhong groups workarounds by the components of their mentioned compiler bugs, but we group workarounds by the components of their affected OS kernels. In both settings, Zhong [46] and we find that most workarounds modify only 10 to 20 code lines. Like He and Zhong [31], we group compiler bugs by their symptoms. Our top three symptoms are the same as reported by He and Zhong [31], but the ranks are different. The difference can reflect the preference of OS programmers. In addition, we are not limited to symptom frequencies but use symptoms to explore more questions.

The tools, data, and results are presented on the anonymous website: https://os-compiler-bug.netlify.app.

2 Compiler Bug Bypassed by Os Kernel

In this section, we use an example to illustrate how our research angle is different from prior studies. OS programmers report a `llvm` bug [3]. When programmers compile a kernel with `clang-10`, it produces a warning message:

> warning: stack frame size of 1368 bytes in function...

Although this is only an unnecessary warning, in the follow-up comment, a compiler programmer confirms that `clang-10` produces broken optimized code due to this bug:

> clang-10 appears to have a broken optimization stage that doesn't enable the compiler to prove at compile time that certain memcpys are within bounds, and thus the outline memcpy is always called, resulting in horrific performance, and in some cases, excessive stack frame growth.

As an OS programmer reports the above compiler bug, this compiler bug is unlikely to introduce bugs to OS kernels. As Xu *et al.* [44] analyze how compiler bugs introduce bugs to OS kernels, they will not analyze compiler bugs reported by OS programmers. As introduced in Section 7, most empirical studies analyze the characteristics of compiler bugs. Only two recent studies [31, 46] have started to analyze how compiler bugs affect other projects. Compared with their studies, we focus on OS kernels and explore new research questions. For instance, we are the first to analyze the knowledge flow between compilers and other projects. In this example, OS programmers identified and reported compiler bugs to the compilers. Alternatively, after other programmers report compiler bugs, OS programmers can learn from known bug reports and bypass them before compiler bugs are fixed. In our study, we analyze which flow direction is more popular. In addition, this compiler bug [3]

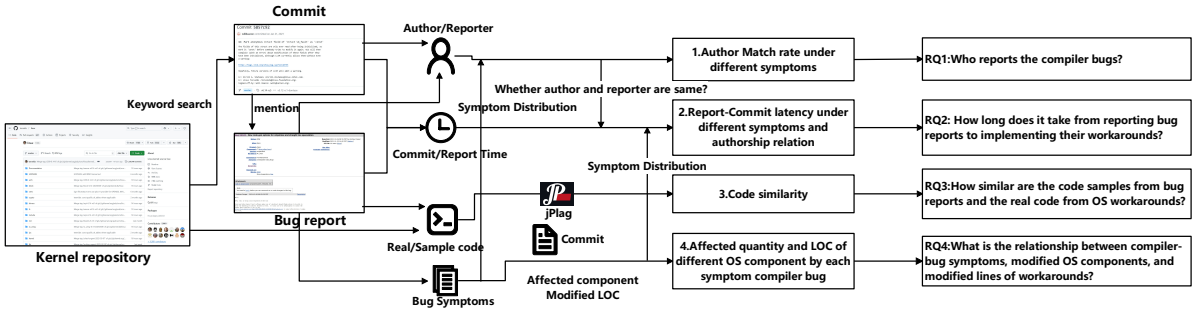


Figure 1. Overview

was reported on May/5/2020. After 24 days, OS programmers implement a workaround [5]:

```
net/bridge/br_multicast.c
@@ -2413,7 +2413,8
- static void mcast_stats_add_dir(u64 *dst, u64 *src)
+ /* noinline for https://bugs.llvm.org/show_bug.cgi?
    id=45802#c9 */
+ static noinline_for_stack void mcast_stats_add_dir(
    u64 *dst, u64 *src)
```

OS programmers bypassed this compiler bug by marking one of the internal functions as `noinline_for_stack`. We calculate the time interval and analyze how the knowledge flow direction affects the time interval. Our study includes other interesting and unexplored analyses.

In our study, we collected 494 workarounds from 7 popular OS kernels. He and Zhong [31] and Zhong [46] analyzed 806 and 644 commits, respectively. As we focus on a specific type of software, we collected fewer workarounds, but the sample size is comparable to the prior studies.

3 Methodology

This section introduces the dataset (Section 3.1) and the overview of our analysis (Section 3.2).

3.1 Dataset

Table 1 shows our subjects. The column "Kernel" lists the names of our selected OS kernels. The Linux [21] kernel is a widely used open-source operating system kernel known for its robustness, performance, and extensive community support. The FreeBSD [13] kernel is part of a Unix-like operating system, renowned for its advanced networking and security features. The Serenity [19] kernel powers SerenityOS, focusing on simplicity and elegance in its design. The ReactOS [18] kernel aims to be binary-compatible with Microsoft Windows NT, providing a free alternative. The OpenBSD [17] kernel emphasizes security, reliability, and portability, making it suitable for secure environments. The Zephyr [20] kernel is a real-time operating system designed for embedded devices, offering modularity and scalability. AROS [12] is an open-source operating system compatible with AmigaOS, known for its flexibility and modern design. Almost all collected kernels contain commits related to `gcc` and `clang`, except for

Table 1. Dataset

Kernel	Workaround	Report	Link	MLOC	Date
Linux	340	231	367	28.29	Sep/4/2011
freebsd	109	53	115	21.44	Sep/5/2011
Serenity	13	14	14	1.15	Dec/2/2018
ReactOS	11	7	13	8.77	Oct/3/2017
openbsd	10	9	10	22.56	Aug/30/2016
zephyr	9	4	9	2.67	May/26/2016
AROS	2	2	2	2.67	May/17/2019
Total	494	320	530	87.55	–

ReactOS, which collected three commit records related to the `msvc` compiler on this project. The column "Workaround" lists the total number of commits whose messages mention compiler bugs. Following the pioneer studies [31, 46], we used the following keywords to query commits from the revision histories of OS kernels:

gcc.gnu.org/bugzilla/show_bug.cgi?id=

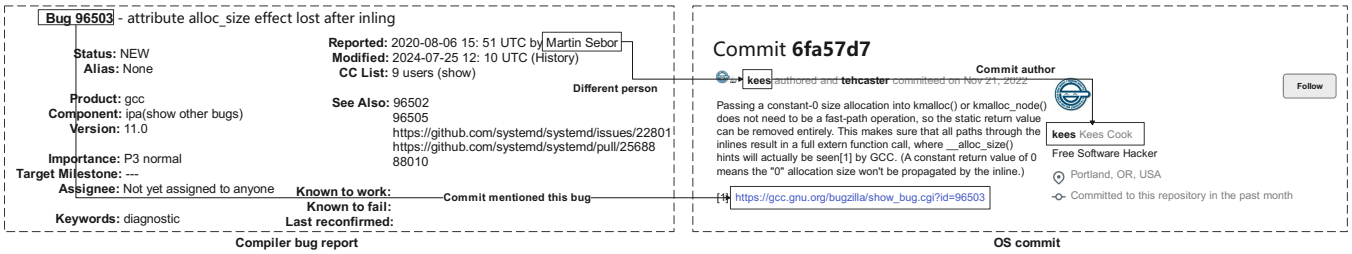
llvm.org/bugs/show_bug.cgi?id=

github.com/llvm/llvm-project/issues

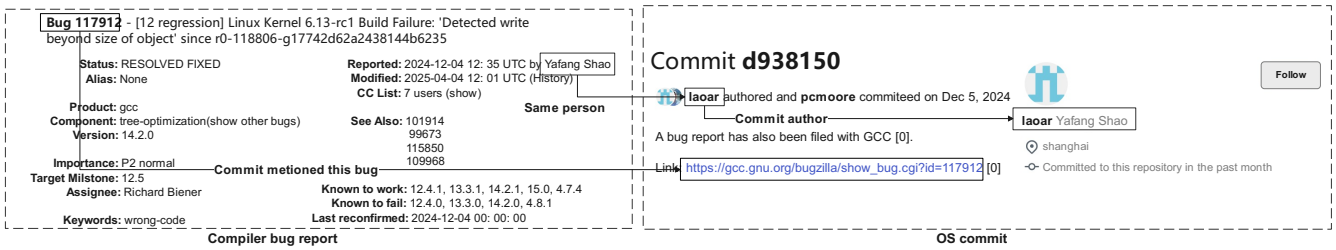
Our tool retrieved 496 commits. After manual inspection, we remove two irrelevant commits. In particular, a Linux commit 2947a45 [15] has a commit message: "update LLVM Bugzilla links". This commit does not bypass compiler bugs. In addition, a Serenity commit 73fab93 [8] has a message: "Toolchain: Update to the latest gcc release 10.3.0". This commit does not bypass compiler bugs either. The column "Report" lists the number of compiler bug reports that are mentioned in the commits of OS kernels. As a compiler bug can be mentioned in more than one commit, we collected more commits than bug reports. The column "Link" lists the identified links between OS workarounds and compiler bug reports. The links are more than the commits since a commit can mention more than one compiler bug. The column "MLOC" lists lines of code in millions for the corresponding OS kernels. The column "Date" denotes the start dates that are recorded by their GitHub repositories.

3.2 General Protocol

Figure 1 shows the layout of our study.



(a) Bug reporter is not workaround programmer



(b) Bug reporter is workaround programmer

Figure 2. Compiler bug reporter and OS-kernel workaround programmer

For RQ1, we analyze whether the authors of OS-kernel workarounds are compiler bug reporters. Based on the answer, we identify two knowledge flow directions. In one direction, OS-kernel developers detect and report compiler bugs before they implement workarounds. In the other direction, other programmers report compiler bugs. OS kernels encounter the known compiler bugs and bypass them. We calculate the instances of both directions. In this process, we follow the symptom taxonomy of He and Zhong [31] and explore the impact of symptoms.

For RQ2, we calculate the time interval between the commit time of an OS-kernel workaround and the reporting time of the corresponding compiler bug that the commit mentions. Meanwhile, we calculate the difference between the two knowledge flow directions.

For RQ3, we compare code samples of compiler bug reports with the modified source files of OS-kernel workarounds. We use a tool called JPlag [36] to calculate code similarity.

For RQ4, we group OS-kernel workarounds by OS components and analyze compiler bugs with different symptoms affecting OS components. We collect modified code lines of workarounds in each category.

4 Empirical Results

4.1 RQ1: Knowledge Flow

4.1.1 Protocol. In this research question, we investigate knowledge flow directions between compilers and OS kernels. In one direction, OS kernel developers detect compiler bugs during their development and report them to the issue trackers of compilers. The knowledge flow is from OS kernels to compilers. In the other direction, OS kernel developers read compiler bug reports identified by other programmers and modify OS code. The knowledge flow is from compilers to OS kernels. The popularity of the two directions can reveal the typical workflow between compilers and OS kernels.

We implement a tool to extract the authors of commits and the reporters of compiler bugs. In particular, from the commits of the OS kernel, it extracts the names and email addresses of the committers, and from compiler bug reports, it extracts details including reporters. We determine that the author of a commit is the reporter of a compiler bug if their names are identical.

The symptoms of compiler bugs can affect the direction of the knowledge flow. When repairing bugs, compiler developers classify bugs by their symptoms. He and Zhong [31] merged the categories of `gcc` and `clang` and built a new taxonomy. Following their taxonomy, we grouped the knowledge flow directions by the symptoms of compiler bugs.

4.1.2 Result. Figure 3 presents the results. The black bars denote the cases where reporters of compiler bugs are not the programmers of OS kernel workarounds. For instance, as shown in Figure 2a, the compiler bug [4] is reported by Mar* Seb*, but the author of the commit [10] is Kee* Coo*. The gray bars denote that they are the same. For instance, as shown in Figure 2b, the compiler bug [14] is reported by Yaf* Sha*, and the author of this commit [16] is also Yaf* Sha*. In this case, the kernel developer may encounter an unknown compiler bug during development and report it to the compiler. The results show that the knowledge from compilers to OS kernels is much more common than the reverse direction. The above observation led to a finding:

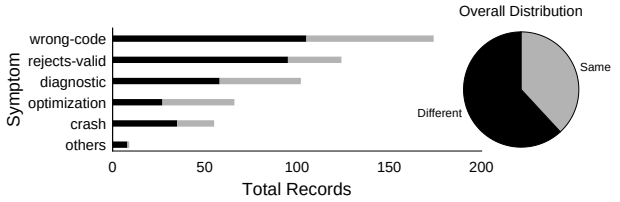


Figure 3. Distribution of knowledge flow

> **Finding 1.** In 61.9% of the links, compiler bugs are not reported by programmers of OS workarounds.

In Figure 3, the vertical axis represents the symptoms of compiler bugs. In particular, wrong-code bugs occur when compilers generate executable code that is different from the source code; rejects-valid bugs occur when compilers erroneously reject syntactically valid code; diagnostic bugs occur when compiler diagnostic messages are wrong; optimization bugs occur when compiler optimizations introduce program misbehavior; and crashes occur when compilers fail with internal crashes. We merge all other compiler bugs into the other category. The horizontal axis represents the number of links between OS commits and compiler bug reports. Among the symptoms, Figure 3 shows that wrong-code, rejects-valid, and diagnostic bugs are the top three symptoms, accounting for 32.8%, 23.4%, and 19.2% of the total bugs. He and Zhong [31] analyze the compiler bugs mentioned in general projects. Their top three symptoms are the same, but they report that the diagnostic, the rejects-valid, and the wrong-code bugs account for 29.22%, 28.77%, and 23.29%, respectively. Our results show that OS kernel programmers bypass more wrong-code bugs. In addition, only in optimization, the knowledge flow from OS kernels to compilers is more common than in the reverse direction. The above observations lead to a finding:

> **Finding 2.** Compared with other programmers, OS kernel programmers care more about wrong-code and optimization compiler bugs.

In summary, the knowledge flow from compilers to OS kernels is more common than the reverse direction. In the OS kernel, the top three symptoms of encountered compiler bugs are wrong-code, reject-valid, and diagnostic as other programmers, but OS programmers care more about wrong-code and optimization bugs.

4.2 RQ2: Time Interval

4.2.1 Protocol. In this research question, we investigate the temporal relationship between the submitted dates of OS commits and the report dates of the mentioned compiler bug reports. Our tool extracts commit dates from GitHub and extracts reporting dates from issue trackers. The reporting dates of compiler bug reports should be earlier than the commit dates of OS kernel workarounds. We then calculate the time interval from the reporting dates to the commit dates. We use the cumulative distribution function (CDF) to calculate the differences. The CDF formula is as follows:

$$F_T(t) = \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}_{\{t_i \le t\}} \tag{1}$$

In this equation, $F_T(t)$ indicates probability of time interval within t days; n indicates the total number of records; $\mathbf{1}_{\{t_i \le t\}}$ is indicator function equaling 1 if $t_i \le t$, 0 otherwise, where t is the time interval we select to know its distribute location in all records and t_i is the i-th time interval data in all records. For instance, as shown in Figure 2b, the compiler bug was reported on Dec/4/2024, and the workaround was submitted on Dec/5/2024. The time interval is 1 day. In 100 records, if there are 10 records with a time interval within 1 day, the CDF will be 0.1.

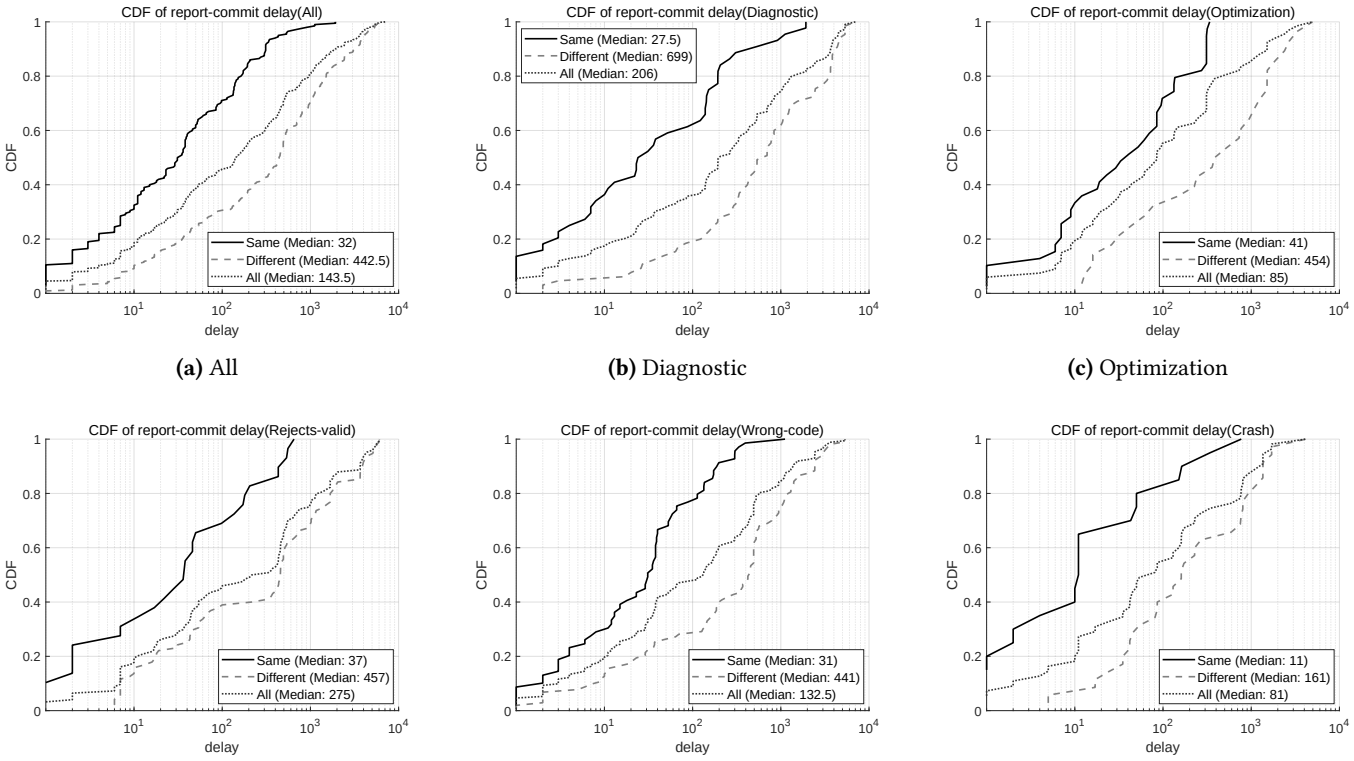
As we introduced in Section 4.1.1, the knowledge flows between compiler bug reports and OS workarounds have two directions, and the compiler bugs have different symptoms. When analyzing the time intervals, we take the directions and the symptoms into consideration and report their impact.

4.2.2 Result. Figure 4 shows the results. The horizontal axes denote the time intervals in days. The vertical axes denote the CDFs. The dotted lines show the results of all data; the solid lines show the results when bug reporters and workaround authors are the same; and the dashed lines show the results when bug reporters are not workaround authors.

In Figure 4a, when the horizontal coordinate is 100, the solid line corresponds to the vertical coordinate of 0.7, indicating that when the bug reporter and workaround author are the same, 70% of the data has less than 100 days of delay. For the dashed line, the corresponding ordinate is about 0.3, indicating that when the bug reporter is different from the workaround author, only 30% of the data has less than 100 days of delay. The median of the dotted line is 143.5 days, the median of the solid line is 32 days, and the median of the dashed line is 442.5 days. The observation leads to a finding:

> **Finding 3.** When compiler bugs are not reported by OS programmers, it takes 10 times as many days to identify and bypass them.

Figure 4 presents the results categorized by symptoms. In compiler testing, most approaches detect crashes since such bugs are more visible. Finding 3 reports that OS programmers pay more attention to optimization and wrong-code bugs. As a result, crashes, optimization, and wrong-code bugs have shorter gaps, and their medians are 81, 85, and 132.5 days,



(a) All **(b)** Diagnostic **(c)** Optimization

(d) Rejects-valid **(e)** Wrong-code **(f)** Crash

Figure 4. CDF from reporting to bypassing compiler bugs

repetitively. As for the time intervals between two knowledge directions, the two symptoms also have fewer gaps. The observations lead to a finding:

> **Finding 4.** The time intervals between reporting compiler bugs and bypassing them in OS kernels are shorter when the symptoms are crashes, optimizations, and wrong-code bugs.

In summary, the time interval between reporting compiler bugs and bypassing them in OS workarounds is 10 times slower if compiler bugs are not reported by OS programmers. If compiler bugs are crashes, optimization, and wrong-code bugs, the time intervals are shorter than other symptoms.

4.3 RQ3: Code Similarity

4.3.1 Protocol. In this research question, we investigate the similarity between the code sample in compiler bug reports and the code in OS kernels. As shown in Table 1, our collected OS commits mention 320 compiler bug reports. Among them, 161 bug reports provide code samples in test cases and comments, and provide 162 code samples. The bug reports with code samples are mentioned in 292 OS commits. Among them, 241 commits modify a reasonable number (1-10) of source files. The rest of the commits either do not modify the code files in the OS or modify too many files. Because some of the commits change the same files, after deduplicating, 239 files are modified by commits that mention bug reports containing the code samples. As a result, we calculate the similarity of 239 links between bug code samples and OS-modified code in this RQ.

For each link, we use the JPlag [36] to calculate the similarity between the sample code of a bug report and the modified files of an OS commit that mentions this compiler bug. JPlag calculates the code similarity with the *Greedy String Tiling* algorithm [43]. The code samples from bug reports are typically much smaller than the modified source files of OS commits. To make a fair comparison, we select the maximal similarity defined by JPlag:

$$S(A,B) = \frac{\sum_{\tau \in \mathcal{T}(A,B)} |\tau|}{\min(T_A, T_B)} \tag{2}$$

In the above equation, A and B denote the token strings of two code samples. $|A|$ and $|B|$ indicate the counts of the total tokens in A and B, respectively. $\mathcal{T}(A,B)$ is a set of maximally overlapped titles. Each tile, τ, is a sequence of identical tokens, and $|\tau|$ is the length of a tile, τ.

For example, if A with 150 tokens and B with 200 tokens have a sequence of 100 identical tokens, $S(A,B)$ is $\frac{100}{150} = 66.7\%$.

We draw box plots for all the links and group the links by symptoms of compiler bugs.

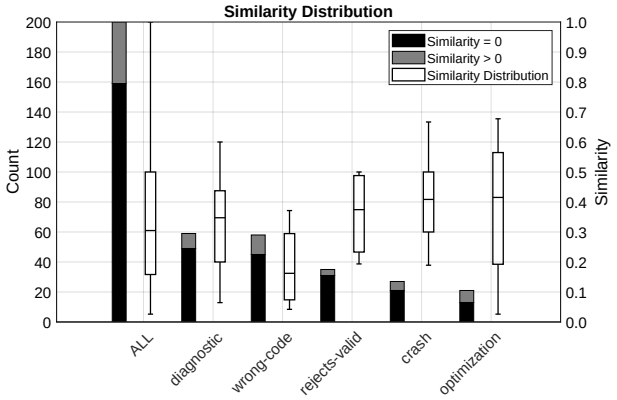


Figure 5. The similarity between code sample and real code

4.3.2 Result. Figure 5 shows the results. The horizontal axis denotes the symptoms of compiler bugs, the right axis denotes the similarity, and the left axis denotes the count of one commit corresponding to one modified file. In the left bar, the black part represents the records with 0 maximal similarity, and the gray part represents the records with maximal similarity greater than 0. The right box plot represents the similarity distribution of records that have maximal similarity not equal to 0. As shown in Figure 5, the similarity values between the code sample of most bug reports and the modified code of OS kernels are zero. The code samples of bug reports are typically reduced and small, but the source files of OS kernels contain many irrelevant fragments. As a result, even if they trigger the same compiler bug, a simple calculation of overlapped tokens is insufficient to identify their similarity in most cases. We draw box plots for links whose similarity values are more than zero. The medians are around 0.3. Figure 6 shows such examples. In particular, Figure 6b shows an example in which the bug reporter is different from the author of the corresponding commit, and Figure 6a shows an example in which the bug reporter is the commit author. In both cases, code samples from bug reports look similar to the modified files of OS kernels, but they have minor differences. For instance, in Figure 6a, the called methods have similar but different names, *i.e.*, `_builtin_bswap64()` and `swab64`. The above observations lead to a finding:

> **Finding 5.** Except for a few cases, the code samples of bug reports are dissimilar to OS kernel code.

When symptoms are considered, we find that only the median of the wrong-code bugs is lower than that of all the data. We manually analyze the results of all wrong-code compiler bug code samples to investigate the reason. For example, we study the gcc bug 101419 [6]. Compiler programmers mark this bug report as a wrong-code bug. The bug code sample in the report is:

```
union diffsize {
    int lg;
    char sm;
};
static void do_wipe(union diffsize *info){
    if (__builtin_object_size(&info->sm, 1) < sizeof(
        info->sm))
        __detected_overflow(__builtin_object_size(&
            info->sm, 1), sizeof(info->sm));
    __builtin_memset(&info->sm, 0, sizeof(info->sm));
    if (__builtin_object_size(&info->lg, 1) < sizeof(
        info->lg))
        __detected_overflow(__builtin_object_size(&
            info->lg, 1), sizeof(info->lg));
    __builtin_memset(&info->lg, 0, sizeof(info->lg));
}
```

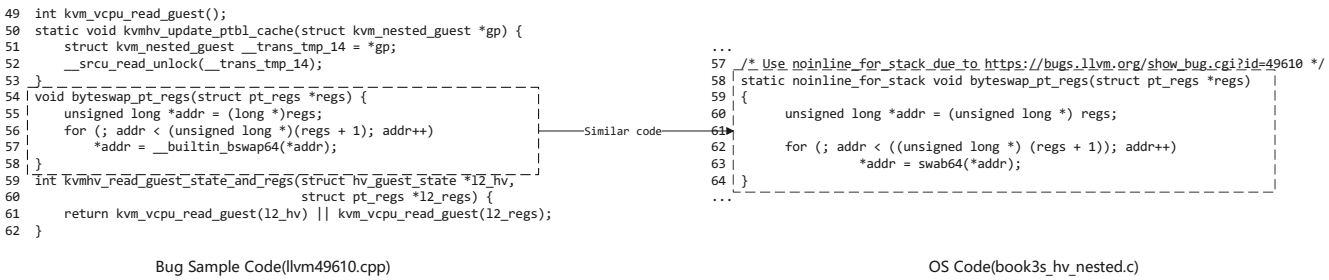
GCC implements two built-in functions, such as `__builtin_memset()` and `__builtin_object_size()`, for the standard functions `memset()` and `sizeof()`, respectively. According to the bug report, this compiler bug is triggered when multiple `__builtin_memset()` methods are called to set the members (`int lg` and `char sm`) of the same union (`diffsize`) with a given value. Due to this bug, `__builtin_object_size()` and `sizeof()` return different values. In a Linux workaround [7], the following code triggers this bug:

```
union ia_css_dvs_grid_u {
  struct ia_css_dvs_stat_grid_info dvs_stat_grid_info;
  struct ia_css_dvs_grid_info dvs_grid_info;
};
sh_css_pipe_get_grid_info(struct ia_css_pipe *pipe,
        struct ia_css_grid_info *info){
memset(&info->dvs_grid.dvs_grid_info, 0,
          sizeof(info->dvs_grid.dvs_grid_info));
memset(&info->dvs_grid.dvs_stat_grid_info, 0,
          sizeof(info->dvs_grid.dvs_stat_grid_info));
}
```

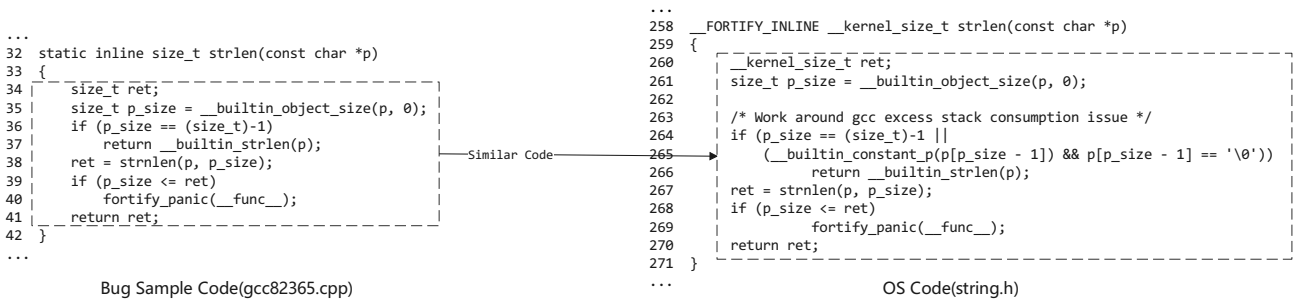
As the built-in functions are optimized, compilers often replace standard functions with their built-in functions during compilation. As a result, like the above code, calling standard functions can trigger compiler bugs involving built-in functions. The names of the union and its members can be different between code samples and real code. Unlike other types of compiler bugs, as wrong code bugs lead to wrong results, their code samples provide code lines for checking whether results are correct, but real code is unlikely to provide code lines. Due to the above observations, the code samples of wrong code bugs are more different than other types of compiler bugs. Consequently, we derive a finding:

> **Finding 6.** The code samples of wrong-code bugs are more different from the modified files of OS kernels than compiler bugs with other symptoms.

In Summary, due to the difference between the bug reproduction environment and the real OS development environment, most of the bug code samples will not show obvious similarity with the real OS code. In a small number of similar cases, the bug code samples are similar but still different compared with the modified files in OS kernels. In terms of symptom distribution, due to the complexity and characteristics of the wrong-code symptom itself, the similarity



(a) Same author



(b) Different authors

Figure 6. Code samples whose similarity value is around 0.3

Table 2. OS kernel component

Component	Linux	FreeBSD	OpenBSD	Serenity	ReactOS	AROS	Zephyr
Driver	drivers/	sys/dev/	sys/dev/	Kernel/Devices/	base/	workbench/devs/	drivers/
Architecture	arch/	sys/{arch}	sys/arch/	Kernel/Arch/	hal/	arch/	arch/
Development	tools/	contrib/	gnu/	ToolChain/, AK/	sdk/tools/	tools/	boards/
Library	lib/, include/	lib/	lib/, include/	Kernel/Library	sdk/lib/	workbench/libs/	lib/, include/
File System	fs/	sys/fs/	sys/nfs/	Kernel/FileSystem/	drivers/filesystems/	workbench/fs/	subsys/fs/
Kernel Core	kernel/, mm/	sys/kern/	sys/kern/	Kernel/Memory/	ntoskrnl/	rom/kernel/	kernel/
Makefile	Makefile	Makefile	Makefile	CMakeLists	CMakeLists	Makefile	cmake/
Network	net/	sys/net/	sys/net/	Kernel/Net/	drivers/network/	workbench/network/	subsys/net/
Security	security/	sys/security/	sys/crypto/	Kernel/Security/	drivers/crypto/	rom/security/	subsys/secure_storage/

Note: Paths are relative to each OS source root directory. `{arch}` denotes architecture-specific subdirectories.

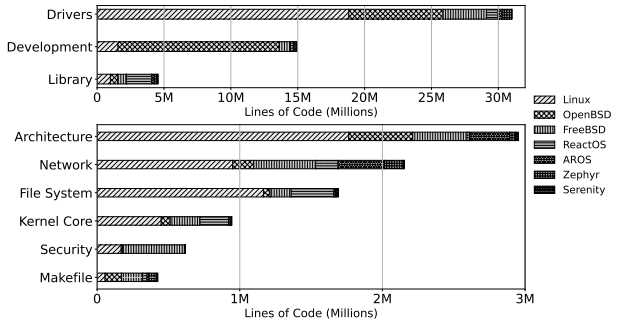
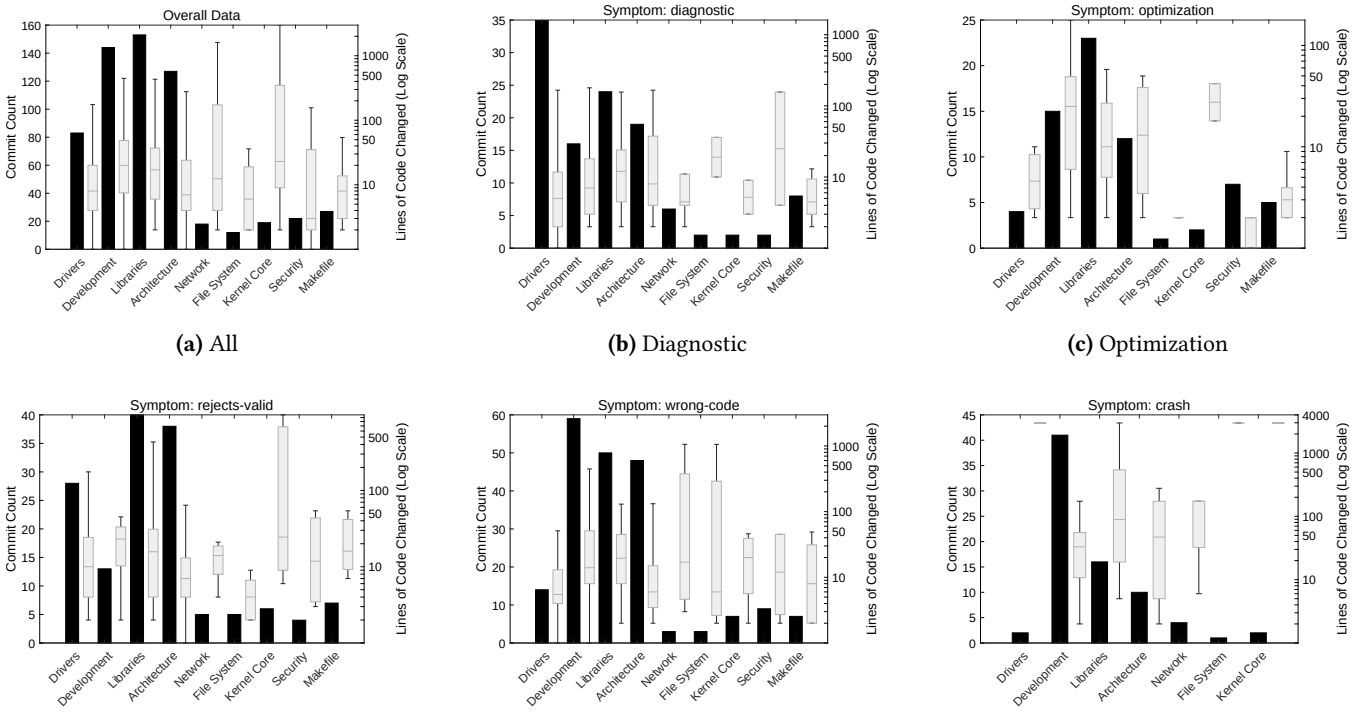


Figure 7. LOCs grouped by OS kernel component

between the bug code samples under this symptom and the real OS code related to this bug will be lower than that of other symptoms and the overall data.

4.4 RQ4: Association

4.4.1 Protocol. In this research question, we investigate the associations among the symptoms of compiler bugs, the affected OS components, and the modified lines of code of OS workarounds. We analyze all the links in Table 1. To determine affected OS components, we merge the directories of OS kernels. Table 2 shows our built taxonomy. The driver contains programs that enable communication between OS kernels and hardware devices. The architecture contains processor-specific implementations. The development includes tools for building and testing OS kernels. The library contains predefined functions that allow applications to access kernel features. The file system manages files and directories on storage devices. The kernel core contains the core implementations. The Makefile defines build configurations. The network handles the communication with other computers. The security contains the implementations for protecting data. If a commit modifies the file from a component, we



(a) All **(b)** Diagnostic **(c)** Optimization

(d) Rejects-valid **(e)** Wrong-code **(f)** Crash

Figure 8. Modified code lines of workaround grouped by OS kernel component

consider that the component is affected. For instance, the gcc bug 117912 [9] is mentioned in a Linux commit d938150 [11], and this commit modified the file, `drivers/ata/ahci.h`. We determined that this compiler bug affects the driver component. If a commit affects more than one component, we count it in each component. Figure 7 shows the lines of code for each component. The driver, the development, and the library have many more code lines than other components. We select a different scale for the other components to ensure the readability of their results.

We count the commits on all the components in Table 2. For a commit on each component, our tool uses the GitHub API to retrieve its added and deleted lines. The number of modified lines is the sum of deleted and added lines. To understand how compiler errors affect the development of an OS component, we draw box plots of modified lines for each component. To understand the impact of symptoms, we group the links by the symptoms of compiler bugs and draw box plots for each symptom.

4.4.2 Result. Figure 8 shows the results. The horizontal axes represent OS components. The left axes list the number of commits. The right axes list the modified lines of commits. For each component, the left black bar represents the number of commits that modify this component, and the right gray box plot shows the modified lines of the corresponding commits. Figure 8a shows the overall results. Although the driver has the most lines of code, compiler bugs affect fewer commits than the development, the architecture, and the library. It has about 2.5 commits per million lines, but the library has 50 commits per million lines. The code from the driver can have simpler structures and is unlikely to trigger compiler bugs. As it defines build configurations, the Makefile component has 54 commits per million lines. The above observations lead to a finding:

> **Finding 7.** Although compiler bugs tend to affect larger components, the library and the Makefile are the most affected per code line.

In Figure 8a, the box plots show the modifications of workarounds. The workarounds on the development and library components have more modifications. The modifications of all components typically involve around 10 lines of code. The result is consistent with Zhong [46]. Although compiler bugs are less likely to affect the kernel core, once it is affected, the workarounds typically involve more modifications. Zhong [46] reports that large modifications contain many repetitive changes. We found that OS kernel commits also have repetitive changes. For instance, a commit [2] of ReactOS bypasses a wrong-code compiler bug [1]. From this commit, we find many replacements from `_SEH_TRY` to `_SEH_TRY2` and `_SEH_HANDLE` to `_SEH2_EXCEPT`:

```
1 reactos/ntoskrnl/config/ntapi.c
2 @@ -38,7 +38,7
```

```
- _SEH_TRY
+ _SEH2_TRY
- _SEH_HANDLE
+ _SEH2_EXCEPT(EXCEPTION_EXECUTE_HANDLER)
reactos/ntoskrnl/config/dbgkobj.c
@@ -1424,16 +1424,16
- _SEH_TRY
+ _SEH2_TRY
- _SEH_HANDLE
+ _SEH2_EXCEPT(EXCEPTION_EXECUTE_HANDLER)
```

This result is consistent with Zhong [46]. The above observation leads to a finding:

> **Finding 8.** The workarounds on the development, the library, and the kernel core have more modifications, and some modifications are repetitive.

Figure 8 shows the results grouped by symptoms. For the majority, the diagnostic compiler bugs mainly affect the driver; the optimization bugs affect the library, the development, and the architecture, the reject-valid bugs affect the library, the architecture, and the driver; the wrong-code bugs affect the development, the library, and the architecture; and crashes affect the development. The library and the development are affected by more than half of the symptoms. As for the modifications, the differences are minor, but bypassing crashes and wrong code bugs requires more modifications. The above observations lead to a finding:

> **Finding 9.** The library and the development are affected by more symptoms, and bypassing crashes and wrong code requires more modifications.

In summary, compiler bugs tend to affect larger components, and the Library and Makefile are the most affected if we consider the density. The workarounds to the development, the library, and the kernel core involve more modifications. As for symptoms, more symptoms affect the library and the development components, and bypassing crashes and wrong code requires more modifications.

5 Threats to Validity

The internal threats to validity include our technique to identify whether a bug report is an author of a workaround. In rare cases, a programmer can use different avatars, and we can underestimate the cases when they are the same. The internal threats to validity also include the taxonomies of compiler bug symptoms and OS components. We mainly adopt the taxonomies of programmers, but programmers may not strictly classify source files by components. Another internal threat is the similarity calculated by JPlag [36]. It detects code clones but does not underestimate semantic similarity. We release our dataset, and other researchers can check and mitigate the threats.

The external threat to validity includes our limited subjects. This threat is shared by all studies and could be reduced with more OS kernels and compilers.

6 Interpretation

This section interprets our findings:

Recommending compiler bugs to OS kernel programmers. Finding 1 shows that in most workarounds, compiler bugs are not reported by OS kernel programmers. Finding 3 shows that it takes 10 times more time if compiler bugs are not identified by OS kernel programmers. There is a strong need to reduce the time gap. Finding 5 shows that the code samples of bug reports can be similar to the source code of OS kernels, but overlapped tokens are insufficient to calculate their similarity in most cases. Researchers need more advanced techniques if they plan to recommend compiler bugs to OS kernel programmers. In addition, Finding 2 reports that OS programmers pay more attention to optimizations and wrong-code bugs. OS programmers can be more interested if tools can recommend the three types of compiler bugs. Meanwhile, it can more visibly reduce the time intervals between reporting compiler bugs and bypassing them if tools can recommend other compiler bugs.

Bypassing compiler bugs based on our findings. Finding 7 shows that larger OS components like the library and build files are more affected by compiler bugs. OS programmers can pay more attention to these components. Finding 8 indicates that the workarounds of the development, the library, and the kernel core have more modifications. Systematic change tools [33, 35] can reduce the effort of implementing workarounds if they contain repetitive changes. Finding 9 shows that compiler bugs with different symptoms can affect specific components. If OS programmers are interested in a compiler bug, they can check the corresponding components more carefully. Researchers have proposed approaches to generate workarounds for CVEs [32], hardware errors [41], and web failures [26]. Our study reveals the relationship between OS components, symptoms, and workarounds. Based on our findings, researchers can propose approaches that generate workarounds for compiler bugs. Talebi *et al.* [40] proposed an approach to undo workarounds for kernel bugs. He and Zhong [31] contact kernel programmers to remove workarounds for compiler bugs after such bugs are fixed, but kernel programmers refuse to remove such workarounds since other programmers can use the buggy versions of compilers in their development. It should be more cautious to remove such workarounds.

Repairing compiler bugs based on our findings. It takes much effort to repair compiler bugs. Zhong [46] reports that about half of compiler bugs are unfixed even if they are encountered in real development. Our findings are useful for compiler programmers to repair bugs more effectively. For instance, Finding 2 shows that wrong-code and optimization

compiler bugs are more likely to attract the attention of OS programmers. Finding 4 shows that crashes, optimizations, and wrong-code compiler bugs have shorter time intervals from the reporting time of bug reports to the committing time of workarounds. Compiler programmers can prioritize bugs with these symptoms. Findings 5 and 6 show that the code samples of most bug reports are different from the modified source files of the workarounds. If programmers check the differences, they can deepen their knowledge of compiler bugs and check whether existing patches truly fix compiler bugs. For compiler researchers, our findings can motivate them to build a detailed compiler bug taxonomy based on bug reports and commit information, laying the foundation for automated diagnosis and intelligent recommended fixes.

7 Related Work

Empirical studies on Compiler bugs. Compiler bugs have been studied on a variety of compilers, such as DL compilers [30, 38], WebAssembly compilers [37], and common C compilers [39, 47]. The above studies analyze the characteristics of compiler bugs themselves, but our study analyzes the outside influence of compiler bugs. Zhong [46] proposed a new angle to analyze compiler bugs by collecting compiler bugs in real development. He and Zhong [31] studied the real-world impact of compiler bugs. Following their ideas, we analyze the impact of compiler bugs, but we are the first to analyze how compiler bugs affect OS kernels.

Empirical studies on OS kernel bugs. Researchers have conducted various empirical studies on bugs in the OS kernel. These studies cover various perspectives such as semantics [27], conditional configuration options [22], and dependencies [23]. These studies analyze the characteristics of OS kernel bugs, but we analyze how OS kernels bypass compiler bugs. Xu *et al.* [44] report that compiler bugs can introduce bugs to OS kernels if kernel programmers fail to identify them. In contrast, we analyze how kernel programmers bypass compiler bugs if kernel programmers notice such compiler bugs.

Empirical studies on workarounds. Programmers implement workarounds to bypass software bugs. Lamothe *et al.* [34] analyze workarounds of API-related bugs. Ding *et al.* [29] analyze the workarounds for Python libraries. Their studies focus on API workarounds, but we focus on workarounds for compiler bugs. Yan *et al.* [45] analyze general workarounds. Most of their analyzed workarounds are irrelevant to compiler bugs. Our study complements the above studies with a detailed analysis of how OS kernels bypass compiler bugs.

8 Conclusion

Although there have been various studies on compiler bugs themselves, only two recent works [31, 46] have started to analyze the impact of compiler bugs, and none have examined how such bugs affect OS kernels. To address this gap in a timely manner, we investigate the workarounds from OS kernels and examine how they bypass compiler bugs. Our study provides answers to four related research questions. For example, we find that in most cases, the code samples in bug reports are quite dissimilar to the modified source files in OS kernels. Thus, it is challenging for OS programmers to determine whether their code will be impacted by a given compiler bug. This issue motivates further research in this area and encourages compiler developers to better highlight the bugs they encounter. Our study also presents other insights that can deepen the understanding of compiler bugs and their influence on OS development.

References

[1] gcc bug report 17982. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=17982, 2004.

[2] Reactos commit c086cc2. https://github.com/reactos/reactos/commit/c086cc2302fce4fb0ec115a003a870d5dd1d5470, 2008.

[3] Bug 45802 - excessive stack usage for fortify_source building the kernel. https://bugs.llvm.org/show_bug.cgi?id=45802, 2020.

[4] gcc bug report 96503. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=96503, 2020.

[5] Linux commit b3b6a84. https://github.com/torvalds/linux/commit/b3b6a84c6a920c60fd1393c43818b3955441424b, 2020.

[6] Gcc bug report 101419. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=101419, 2021.

[7] Linux commit 69aa1de. https://github.com/torvalds/linux/commit/69aa1deeab47a47f1e7876fabb76e1e11496c418, 2021.

[8] serenity 73fab93 commit. https://github.com/SerenityOS/serenity/commit/73fab93ef582328f29a1b475e6da52f2fd68ab3d, 2021.

[9] gcc bug report 107917. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=107917, 2022.

[10] Linux commit 6fa57d7. https://github.com/torvalds/linux/commit/6fa57d78aa7f212fd7c0de70f5756e18513dcdcf, 2022.

[11] Linux commit f077880. https://github.com/torvalds/linux/commit/f07788079f515ca4a681c5f595bdad19cfbd7b1d, 2022.

[12] Aros. https://github.com/aros-development-team/AROS, 2024.

[13] freebsd. https://github.com/freebsd/freebsd-src, 2024.

[14] gcc bug report 117912. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=117912, 2024.

[15] linux 2947a45 commit. https://github.com/torvalds/linux/commit/2947a4567f3a79127d2d540384e7f042106c1a24, 2024.

[16] Linux commit d938150. https://github.com/torvalds/linux/commit/d9381508ea2b590aff46d28d432d20bfef1ba64c, 2024.

[17] openbsd. https://github.com/openbsd/src, 2024.

[18] Reactos. https://github.com/reactos/reactos, 2024.

[19] Serenityos. https://github.com/SerenityOS/serenity, 2024.

[20] zephyr. https://github.com/zephyrproject-rtos/zephyr, 2024.

[21] Linux. https://github.com/torvalds/linux, 2024.

[22] I. Abal, C. Brabrand, and A. Wasowski. 42 variability bugs in the linux kernel: a qualitative analysis. In *Proc. ASE*, pages 421–432, 2014.

[23] M. F. Ahmed and S. S. Gokhale. Linux bugs: Life cycle, resolution and architectural analysis. *Information and Software Technology*, 51(11):1618–1627, 2009.

[24] A. S. Boujarwah and K. Saleh. Compiler test case generation methods: a survey and assessment. *Information and software technology*, 39(9):617–625, 1997.

[25] R. E. Bryant and D. R. O'Hallaron. *Computer systems: a programmer's perspective*. Prentice Hall, 2011.

[26] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè. Automatic workarounds: Exploiting the intrinsic redundancy of web applications.

Transaction on Software Engineering and Methodology, 24(3):1–42, 2015.

[27] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proc. APSys*, pages 1–5, 2011.

[28] 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.

[29] 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.

[30] X. Du, Z. Zheng, L. Ma, and J. Zhao. An empirical study on common bugs in deep learning compilers. In *Proc. ISSRE*, pages 184–195. IEEE, 2021.

[31] Z. He and H. Zhong. From bug report to workrounds:the real-world impact of compiler bugs. In *Proc. SANER*, page to appear, 2025.

[32] Z. Huang, M. DAngelo, D. Miyani, and D. Lie. Talos: Neutralizing vulnerabilities with security workarounds for rapid response. In *Proc. S&P*, pages 618–635, 2016.

[33] M. Lamothe, Y.-G. Guéhéneuc, and W. Shang. A systematic review of api evolution literature. *ACM Computing Surveys*, 54(8):1–36, 2021.

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

[35] N. Meng, M. Kim, and K. S. McKinley. Lase: Locating and applying systematic edits by learning from examples. In *Proc. ICSE*, pages 502–511, 2013.

[36] P. M. Prechelt, Malpohl G. Jplag: Finding plagiarisms among a set of programs. *Journal of Universal Computer Science*, 8(11):1016–1038, 2002.

[37] A. Romano, X. Liu, Y. Kwon, and W. Wang. An empirical study of bugs in webassembly compilers. In *Porc. ASE*, pages 42–54, 2021.

[38] Q. Shen, H. Ma, J. Chen, Y. Tian, S.-C. Cheung, and X. Chen. A comprehensive study of deep learning compiler bugs. In *Proc. FSE*, pages 968–980, 2021.

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

[40] S. M. S. Talebi, Z. Yao, A. A. Sani, Z. Qian, and D. Austin. Undo workarounds for kernel bugs. In *Proc. USENIX Security*, pages 2381–2398, 2021.

[41] Tsung-Po Liu, Shuo-Ren Lin, and Jie-Hong R. Jiang. Software workarounds for hardware errors: Instruction patch synthesis. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(12):1992–2003, 2013.

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

[43] M. J. Wise. String similarity via greedy string tiling and running karp-rabin matching. *Online Preprint, Dec*, 119(1):1–17, 1993.

[44] 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.

[45] 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.

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

[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.