How Well Industry-Level Cause Bisection Works in Real-World: A Study on Linux Kernel

Kangzheng Gu
Fudan University
Shanghai, China
kzgu21@m.fudan.edu.cn

Yuan Zhang
Fudan University
Shanghai, China
yuanxzhang@fudan.edu.cn

Jiajun Cao
Fudan University
Shanghai, China
20210240046@fudan.edu.cn

Xin Tan
Fudan University
Shanghai, China
18212010028@fudan.edu.cn

Min Yang
Fudan University
Shanghai, China
m_yang@fudan.edu.cn

ABSTRACT

Bug fixing is a laborious task. In bug-fixing, debugging needs much manual effort. Various automatic analyses have been proposed to address the challenges of debugging like locating bug-inducing changes. One of the representative approaches to automatically locate bug-inducing changes is cause bisection. It bisects a range of code change history and locates the change introducing the bug. Although cause bisection has been applied in industrial testing systems for years, it still lacks a systematic understanding of it, which limits the further improvements of the current approaches.

In this paper, we take the popular industrial cause bisection system on Syzbot to perform an empirical study of real-world cause bisection practice. First, we construct a dataset consisting of 1,070 publicly disclosed bugs by Syzbot. Then, we investigate the overall performance of cause bisection. Only one-third of the bisection results are correct. Moreover, we analyze the causes why cause bisection fails. More than 80% of failures are caused by unstable bug reproduction and unreliable bug triage. Furthermore, we discover that correct bisection results indeed facilitate bug-fixing, specifically, recommending the bug-fixing developer, indicating the bug-fixing location, and decreasing the bug-fixing time. Finally, to improve the performance of real-world cause bisection practice, we discuss possible improvements and future research directions.

CCS CONCEPTS

• **Security and privacy → Operating systems security**.

KEYWORDS

System Security, Operating System, Linux Kernel, Cause Bisection

ACM Reference Format:
Kangzheng Gu, Yuan Zhang, Jiajun Cao, Xin Tan, and Min Yang. 2024. How Well Industry-Level Cause Bisection Works in Real-World: A Study on Linux Kernel. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE Companion '24), July 15–19, 2024, Porto de Galinhas, Brazil.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3663529.3663828

1 INTRODUCTION

In recent years, fuzz testing (fuzzing) [13, 26, 41, 44, 49, 52, 55, 57] has become one of the most popular techniques of bug discovery, especially in industry [6, 9, 10]. While fuzzing has significantly improved the efficiency of bug discovery, it also generates thousands of bugs for developers to analyze and fix [32, 45], which brings heavy pressure on developers. To mitigate debugging efforts, a technology called cause bisection[4, 5] have been used to assist bug analysis. The cause bisection aims to automatically locate the code changes that introduce the bugs to facilitate developers comprehending and fixing these bugs. In the industrial scenario, software programs are managed with *repositories*, and the code changes are organized as *commits*. Thus, industrial cause bisection is designed to find the bug-inducing commits, which helps developers rapidly pinpoint the buggy code.

Cause bisection has been deployed in industrial testing systems for years, like cause bisection in syzbot [10] and OSSFuzz [9]. Unfortunately, it is unclear whether the cause bisection is effective enough in the real world. First, the correctness is questionable. Some developers complain they are misled by current techniques [7, 8]. Second, the impact of cause bisection on bug fixing has not been studied. Nobody knows whether it can significantly benefit bug fixing under the blooming of bug reports. These two problems make it unclear whether we should put more effort into improving the current cause bisection and how to improve it.

To our knowledge, the only measurement of the industrial cause bisection [1], i.e., the cause bisection of Syzbot [5] for Linux kernel, involves 118 bugs discovered before March 2019. However, this study is still limited. First, the study only shows coarse-grained observations that are not systematic, which hardly helps to further improve the techniques, since the measurement relies on manual analyses on a few of the sampled cases. Second, this study does not discuss the influence of cause bisection on the bug-fixing practice, which we think is very important to comprehend the real-world impact of cause bisection.

Although the study mentioned above is limited, it inspires us to take Syzbot's cause bisection as an evaluation target. First, the Linux kernel is complex enough, whose bugs contain various kinds, on which the cause bisection can be comprehensively evaluated. Second, a large number of kernel bugs have been analyzed by Syzbot's cause bisection, to be specific, 2,305, of which 1,359 have been marked at least one bug-inducing commit by developers until March 29, 2023. This provides a large-scale dataset. Thus, Syzbot's cause bisection is an ideal research target for evaluating the performance and impact of industrial cause bisection.

In this paper, we perform a large-scale empirical study of Syzbot's cause bisection system. The purpose is to understand the performance, reveal the limitations, and comprehend the impact when cause bisection is applied in the industrial scenario. We intend to answer three research questions: (1) how effective and efficient the cause bisection is; (2) what the major causes of the failures are in cause bisection; (3) how cause bisection facilitates bug fixing. Based on the findings, we further discuss possible improvements and research directions in the future.

To be specific, we first construct a dataset of real-world bug reports on the Linux kernel. For each bug report, we collect the bisection results provided by Syzbot and find the "ground-truth" commit it should be. Then, we analyze the failures of cause bisection by reviewing the bisection log. Combining the ground truth, we categorize and count the causes of bisection failures. Next, we perform a statistical analysis to study the relationship between cause bisection results and bug fixing, showing its significant influence on bug-fixing progress, which indicates that it is valuable to further improve the cause bisection. Based on the findings, we discuss the potential improvements of the current version of cause bisection. Our artifacts and dataset are publicly available at https://github.com/seclab-fudan/SyzbotCauseBisectionStudy.

Contributions. The major contributions and findings are summarized below:

- *Large-scale Dataset.* We construct a dataset containing 1,070 bug reports of the real-world Linux kernel with ground truth. To our knowledge, it is the largest dataset to study a real-world industrial cause bisection practice to date.
- *Empirical Evaluation.* We find that only one-third of cause bisection results are correct and reveal the most significant causes of failures are unstable bug reproduction and unreliable bug triage that leads to more than 80% of the failures, and should be the most important issues to be addressed in the future.
- *Impact Comprehension.* We confirm the significant impact of cause bisection in the real-world bug-fixing practice, showing the prospects of cause bisection. Specifically, we discover that a correct cause bisection will help report the bug to a proper fixer, indicating the bug-fixing location, and finally decrease the bug-fixing time.
- *Promising Directions.* Based on our findings, we hope to shed light on the future research of real-world cause bisection. We discuss the potential improvements both for the bisection algorithm and the software testing techniques.

2 BACKGROUND

We first introduce the common methods of locating bug-inducing commits, and then give a brief introduction to Syzbot, an industrial kernel fuzzing system, especially explaining how Syzbot works to locate bug-inducing commits and interacts with developers.

2.1 Methods of Locating Bug-inducing Commits

Locating bug-inducing commits can facilitate the bug-fixing progress. Here are some popular methods that locate bug-inducing commits.

Patch-based Methods [19, 51, 54]. Such methods find the most recent commits that modify the patch-related code. These commits are thought to be bug-inducing commits. However, the availability of a patch is the prerequisite. So patch-based methods cannot be applied to zero-day bugs.

Information-retrieving-based Methods [23, 27, 46, 59, 61]. These methods extract useful information from bug reports. They use such information as quires to retrieve the most related code commits through lexical similarity or language models. Thus, the performance of information-retrieving-based methods is affected by the quality of bug reports. In the real world, the quality of bug reports cannot be always guaranteed, which limits the large-scale application of information-retrieving-based methods.

Fault-localization-based Methods [15, 16]. These methods leverage the power of fault localization [60] to pinpoint a small code slice that is highly related to the root cause of the bug. Then they use the code slice to find which commit most recently introduces the code slice. Fault-localization-based methods usually need a set of failure inputs to precisely locate the root-cause code slice. However, the set of failure inputs is not always available in the real world.

Compared to these methods, **bisection-based methods** is much more preferred in real-world industrial scenarios since the industrial system prefers simpler techniques (meaning more robustness) and fewer restrictions (meaning more scalability). Although plenty of methods have been proposed by the research community, we think it is still necessary to evaluate and comprehend the currently running method, i.e. the cause bisection.

2.2 Cause Bisection in Syzbot

Syzbot is an industrial continuous fuzzing system for the Linux kernel. And cause bisection is a mechanism provided by Syzbot to facilitate bug fixing. The goal of cause bisection is to find the commit that introduces the fuzzing-discovered bug, called ***bug-inducing commit***. The insight behind the cause bisection is that a bug is usually established by certain bug-inducing commits, and if the kernel contains the bug-inducing commits, the PoC found by fuzzing should trigger the bug, otherwise it should not. Thus, cause bisection uses the fuzzer-found PoC to test the kernel to find out the bug-inducing commit. The phenomenon of whether the bug has been triggered is usually a kernel crash caused by a kernel panic, KASAN report, or some other predefined assertions.

Therefore, for a certain version of the kernel, cause bisection could speculate the existence of the bug-inducing commits by executing the fuzzer-found PoC and monitoring whether a crash happens. Using an iterative search on major release versions, cause bisection first determines the version range where the bug-inducing commit is located. Within this range, cause bisection leverages a binary search algorithm to find out the exact bug-inducing commits. The whole procedure is depicted in Figure 1.

Cause bisection consists of three stages: reproduction confirmation, version-level iteration, and commit-level bisection:

2.2.1 Stage A: Reproduction Confirmation. When a bug is found by fuzzing, the bug report usually contains a ***crash commit*** and

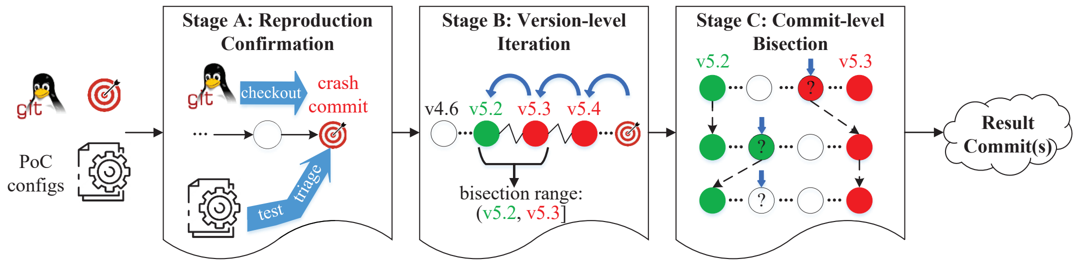
Stage A: Reproduction Confirmation
git
checkout
crash commit
PoC configs
test
triage
Stage B: Version-level Iteration
v4.6
v5.2
v5.3
v5.4
bisection range: (v5.2, v5.3]
Stage C: Commit-level Bisection
v5.2
v5.3
?
Result Commit(s)

Figure 1: Overview of Cause Bisection in Syzbot. A: runs the fuzzer-found PoC to make sure the bug can be reproduced. B: reversely tests the release versions. C: performs the bisection in a certain range until the bug-inducing commit is found.

a PoC. Cause bisection will first skip the crashes that are not able to reproduce on the crash commit with the PoC since the Syzbot cannot determine the existence of the non-reproducible bugs in the next steps.

2.2.2 Stage B: Version-level Iteration. Once a bug is successfully reproduced on the crash commit, cause bisection takes this commit as the beginning of the subsequent tests. First, Syzbot locates the latest release version before the crash commit. From the latest release version before the crash commit, Syzbot reversely iterates all the release versions, i.e., from the latest to the oldest, and executes the PoC on them. If the kernel crashes, it means that the bug exists in the corresponding version. When a non-crash version next to a crash version is found, the bug-inducing commit should lay between these two versions.

2.2.3 Stage C: Commit-level Bisection. After the version range of the bug-inducing commit is confirmed, cause bisection performs a binary search between the first commit of the non-crash version and the last commit of the crash version. To be specific, the cause bisection will test the commit in the middle of the range. If the bug can be reproduced, the range will shrink to the left side, i.e., the bug-inducing commit should be in the predecessors of the middle commit, and vice versa. Cause bisection repeats this procedure until the PoC crashes on a certain commit but does not crash on the previous one. This certain commit is thought to be a bug-inducing commit. Sometimes the cause bisection may fail to determine the bug existence for all the commits within a certain range. In this case, a set containing all the commits in this range will be outputted rather than a single commit. For convenience, in the rest of this paper, we uniformly use the term ***result commit(s)*** representing the commit or the set of commits outputted by cause bisection.

2.3 Interaction between Syzbot and Developers

When a bug is found during fuzzing, Syzbot will report the bug to kernel developers. First, it extracts the source file involved in the crash report, finds the corresponding maintainers, and sends the bug reports to these maintainers. If the bug has a PoC that can reproduce it, the cause bisection will start. The result of cause bisection usually contains one or a set of possible bug-inducing commit(s) and a cause bisection log. The above information will be sent to the developer whose mail address is recorded in the bug-inducing commit(s).

The information in the result commits would facilitate the bug fixing in several ways. First, Syzbot sends the result commits and bisection log to the mailing addresses recorded in these commits. It helps to find a proper developer who is familiar with the relevant program logic to analyze and fix the bug, since the developers recorded in the result commits might be more familiar with the bug-related functionalities than other developers. Second, the bug-inducing commit is a clue to comprehend the root cause of the bug. The developers could rapidly understand the mechanism of the bug by reviewing the code changes in the bug-inducing commit. As a result, the bug-fixing progress would be largely facilitated, i.e., the bug-fixing time is shortened.

3 RESEARCH QUESTIONS

We aim to perform a comprehensive evaluation of the cause bisection system of Syzbot to understand its performance, limitations, and value in practice. Specifically, we study the following research questions.

RQ1: How effective and efficient the cause bisection is. Bisection has proven to be effective under ideal circumstances. However, since some key assumptions like accurate bug existence testing do not hold in the real world, cause bisection is not as effective as expected. In addition, there is lots of negative feedback from developers like [7, 8]. In a word, the performance of cause bisection has not been clearly understood since its birth, motivating us to investigate whether cause bisection works well in practice.

RQ2: What the major causes of the failures in cause bisection are. We are curious under which circumstances the theoretically effective bisection algorithm may fail and why it fails under real-world noises. Therefore, we aim to perform a deep analysis of the failure cases to reveal the technical limitation of current cause bisection in the real world. Such analysis also guides us on how to make improvements to cause bisection practices in the future.

RQ3: How the cause bisection facilitates the bug fixing. Although the cause bisection has been running for years, it is unclear whether it helps the real-world bug-fixing practice significantly. Answering this question would benefit future research on automatic debugging. If cause bisection promotes the bug-fixing practice significantly, it would be worthwhile to make further improvements. Otherwise, it is considerable to turn to other techniques.

RQ4: How we can improve the real-world cause bisection practice. The final goal of our study is to indicate the possible

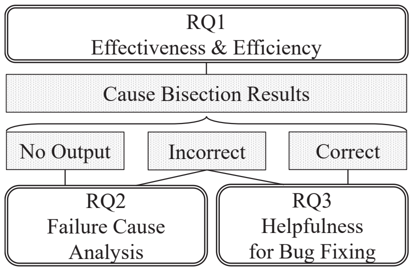

Figure 2: Relationships among RQs.

improvements of real-world cause bisection practice. So, we discuss the possible future directions according to our findings from multiple perspectives of cause bisection.

The overall workflow of our study is shown in Figure 2. To this end, we first construct a large-scale dataset of cause bisection results, which will be introduced in §4. Then, in §5, we perform analysis on the large-scale dataset to answer RQ1. In §6, we answer RQ2 by mining the failure cases (i.e., No output & Incorrect). Further, we answer RQ3 in terms of incorrect results and correct results in §7. Finally, in §8, we discuss possible improvements in real-world cause bisection practice based on our findings.

4 DATASET CONSTRUCTION

To answer the three research questions proposed in §3, we need to gather detailed information about cause bisection, including its output and the analysis process. In addition to cause bisection information, to determine whether the output ***result commit*** is correct (effectiveness in RQ1), we need to identify the real bug-inducing commit, i.e., ***ground-truth commit*** for each bug. Fortunately, the output of cause bisection, as well as the analysis log, is publicly available from Syzbot dashboard [11]. In the following, we will introduce the data collection and ground truth construction.

4.1 Data Collection

Until March 29, 2023, Syzbot has disclosed 5,452 valid kernel bugs on the "Linux" track [12]. However, not all of them are processed with cause bisection, since some bugs do not have a reproducible PoC. So, we filter out the bugs that are not processed by cause bisection and get 2,305 valid bugs for our study. For each bug, we extract useful information from cause bisection logs.

A typical bisection log is shown in Figure 3, which consists of the input crash commit, the run logs for three stages (stage A, B, and C introduced in §2), and the summary of cause bisection result. For each stage, the log records the tested commit or version and the test result. Once the cause bisection locates a result commit, the result commit information (e.g., commit ID and commit message) and the total elapsed time is recorded as the result summary. If any step goes wrong during cause bisection, the result summary records the error message instead of the result commit information. Given a bisection log, we first fetch the input crash commit from the head of the log and extract the result commit or error message from the result summary. Then we locate the log for each stage by matching the keywords such as "testing release" and "git bisect start", and extract the tested commits and test results for each stage.

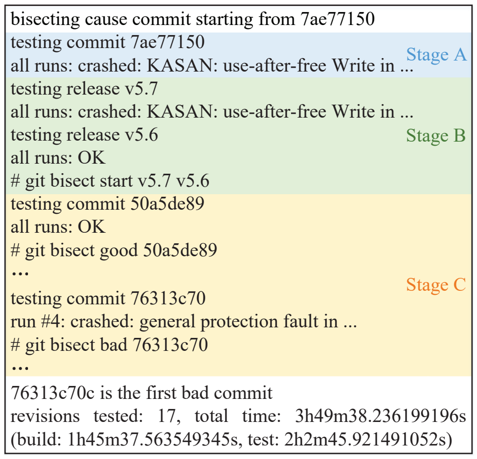
```
bisecting cause commit starting from 7ae77150
testing commit 7ae77150                                   Stage A
all runs: crashed: KASAN: use-after-free Write in ...
testing release v5.7
all runs: crashed: KASAN: use-after-free Write in ...
testing release v5.6                                      Stage B
all runs: OK
# git bisect start v5.7 v5.6
testing commit 50a5de89
all runs: OK
# git bisect good 50a5de89
...
                                                          Stage C
testing commit 76313c70
run #4: crashed: general protection fault in ...
# git bisect bad 76313c70
...
76313c70c is the first bad commit
revisions tested: 17, total time: 3h49m38.236199196s
(build: 1h45m37.563549345s, test: 2h2m45.921491052s)
```

Figure 3: Example of Cause Bisection Log.

4.2 Ground Truth Construction

To construct the ground truth for cause bisection, we need to locate the bug-inducing commits for the 2,305 collected bugs. An intuitive method is to manually analyze each bug and determine the real bug-inducing commit. However, kernel bug analysis requires a high level of domain expertise and is very time-consuming. Therefore, it is infeasible to analyze all the 2,305 bugs manually.

Here we take another approach that is much more practical. We notice that once a bug is fixed, the developer usually adds a "Fixes" tag in the patch's commit message pointing to the commit that introduces the bug[3], as shown in Figure 4. Based on this convention, for a given bug, we first locate its patch from the Syzbot dashboard and extract the commit that the "Fixes" tag points to as human-labeled bug-inducing commits. If there are multiple commits tagged with "Fixes" in a patch, we take the latest commit. The reason is that as the latest bug-inducing commit is introduced, the buggy logic becomes complete and can be triggered with the fuzzer-found PoC. Thus, the latest bug-inducing commit is the expected output of the cause bisection because cause bisection finds the bug-inducing commit according to the triggerability. Since not all bugs collected are fixed and not all patches contain the "Fixes" tag, we locate the human-labeled bug-inducing commits for 1,136 of them in the end.

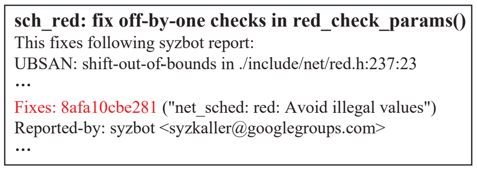
```
sch_red: fix off-by-one checks in red_check_params()
This fixes following syzbot report:
UBSAN: shift-out-of-bounds in ./include/net/red.h:237:23
...
Fixes: 8afa10cbe281 ("net_sched: red: Avoid illegal values")
Reported-by: syzbot <syzkaller@googlegroups.com>
...
```

Figure 4: Example of the "Fixes" Tag in a Patch's Commit Message.

To ensure the credibility of the ground-truth construction, we further validate the human-labeled bug-inducing commits to filter incorrect samples introduced by developer mislabeling. In particular, our validation consists of two heuristic rules, which exploit the order among the real bug-inducing commit, crash commit, and patch commit. (1) The real bug-inducing commit should be in the predecessors of the crash commit reported by the fuzzer on the git tree, i.e., the bug-inducing commit should occur earlier than the commit where the crash is triggered. (2) The real bug-inducing commit should not be in the successors of the patch commit, i.e., the bug-inducing commit should not be later than the patch, because the bug would no longer exist after patching. Applying these two heuristics, we obtain highly reliable ground truth for 1,070 bugs out of all the 1,136 bugs with human-labeled bug-inducing commits.

Finally, our dataset contains 1,070 samples (bugs), in which each data sample has the result commit(s), the ground-truth commit, and other information about the bisection process. The sampled bugs are distributed from kernel version 2.6 to 6.3, reflecting the abundance of our dataset.

5 EFFECTIVENESS AND EFFICIENCY (RQ1)

For effectiveness, we evaluate whether the cause bisection provides a valid result commit and whether the provided commit is correct. For efficiency, we evaluate the time of the cause bisection and the number of tested commits.

5.1 Effectiveness

To evaluate the effectiveness, we sort the bisection results into three categories.

- *Correct.* The process of cause bisection finishes without any error and outputs at least one result commit. If the result commit is a single commit, it should be the same as the ground-truth commit. Otherwise, the set of result commits should include the ground-truth commit.
- *Incorrect.* The process of cause bisection finishes without any error. But the result commit is different from the ground-truth commit or the set of result commits does not contain the ground-truth commit.
- *No Output.* The process of cause bisection exits in the middle for some reason, resulting in the cause bisection outputting no result commit.

The ratio of each kind of result is shown in the third to the fifth column in Table 1, which reflects the overall effectiveness of cause bisection. Cause bisection fails to provide any result commit for 25% bugs. For the remaining 75% bugs, cause bisection outputs at least one result commit, in which 71.8% of the results contain exactly one commit, while 3.2% contain multiple commits (median: 5). In these cases, cause bisection only finds the correct bug-inducing commit for 34% of the bugs, while the others are incorrect. In short, cause bisection is only able to find the correct bug-inducing commit for one-third of the bugs in our dataset. It seems that the cause bisection is not effective enough in the real world, which may confuse the developer who is going to fix the bug.

Finding 1: Cause bisection only finds the correct bug-inducing commit for one-third of the bugs in our dataset, showing its limited effectiveness in the real world.

5.2 Efficiency

We directly use the time recorded in the bisection log to evaluate the efficiency of cause bisection. From the bisection log, we extract the number of testing commits and total time elapsed according to the keywords "revisions tested" and "total time". Furthermore, we extract the time of kernel building and testing respectively according to the keywords "build:" and "test:". We calculate the average versions iterated in stage B, the average commits bisected in stage C, and the average time the cause bisection takes. These results are shown in the last five columns in Table 1. The average time of cause bisection is 4.95h, while 70.09% cases are finished within the average time. About 46.06% time is spent on kernel building, while the remaining is spent on testing. To test a single commit, it takes on average 13.20 minutes.

Table 1: Overall Performance of Cause Bisection. #Bugs=Number of bugs. #NO=no-output. #CR=correct. #IN=incorrect. #Avg.V=average number of tested versions. #Avg.C=average number of tested commits. #Avg.T=average time for single bug: Build=average kernel building time. Test=average testing time. Total=average total time.

Version	#Bugs	Effectiveness			Efficiency				
		#NO	#CR	#IN	#Avg.V	#Avg.C	#Avg.T		
							Build	Test	Total
v6.0-v6.3	102	56 (55%)	24 (24%)	22 (22)	6.40	13.55	3.34h	2.57h	5.91h
v5.0-v5.19	932	204 (22%)	331 (36%)	397 (43%)	5.28	16.45	2.20h	2.68h	4.88h
v4.16-v4.20	36	8 (22%)	9 (25%)	19 (53%)	8.25	11.64	1.48h	2.49h	3.97h
Total	1,070	268 (25%)	364 (34%)	438 (41%)	5.49	16.01	2.28h	2.66h	4.95h

We further investigate why some cases spend much more time. The data in Table 1 reflect that the time cost is higher related to the number of tested commits rather than the number of tested versions, indicating that the most of time is spent in stage C. Theoretically, the expected number of tests in a bisection algorithm is $log_2 n$ in which the n represents the number of commits in the bisection range. However, for the cases whose time is beyond the average, we compare the actual number of their tested commits against the expected and observe that they perform 30.38 tests on average, 2.19 times of the expected number (13.90). Through analyzing their bisection logs, we find that these cases often meet compilation errors and boot failures in stage C, which compels the cause bisection to try more commits in stage C to isolate the bug-inducing commit.

Finding 2: The compilation errors and boot failures waste a lot of time during cause bisection.

6 FAILURE CAUSE ANALYSIS (RQ2)

The failure of cause bisection includes two different kinds: incorrect and no-output. We try to dig out the causes behind these failures.

6.1 Methodology

6.1.1 Conjecture of Failure Causes. To investigate the possible reasons why cause bisection fails, we review the design of cause bisection. The whole cause bisection can be divided into three different layers: algorithm, technique, and configuration layer. The algorithm layer is the top layer, i.e., the binary search algorithm. The technique layer is the middle layer containing the specific techniques relied on by the algorithm to determine the bug's existence on a certain version, like kernel compilation and bug reproduction. The configuration layer is the bottom layer, including the hyperparameters controlling the algorithmic behavior.

Following this framework, we sort the possible failure reasons into categories. Since the algorithm layer contains only a provable binary search, it is always reliable when the bug's existence is correctly evaluated. So, the failure comes from the other two layers. We named the failure causes from the technique layer as unreliable techniques (T), and from the configuration layer as restrictive configuration (C). The possible causes are organized in the Table 2 and we will introduce them in the following text.

Unreliable Techniques. In stage A, the cause bisection tries to confirm the reproducibility of the bug. So the only issue that may lead to a failure in this stage is that the bug is not reproduced. There are two possible reasons. First, the kernel itself is not successfully built or booted, which we call *unreliable kernel setup (T1).* This may due to the environment or dependencies are not properly configured. Second, the kernel is ready but the PoC cannot trigger the bug, which we call *unstable bug reproduction (T2).* Once stage A fails due to T1 or T2, the cause bisection thinks the bug can not be reproduced. As reproducibility is the prerequisite, cause bisection would exit with no output.

In stage B and stage C, the cause bisection tests the versions and commits of the kernel to locate the first version or commit that introduces the bug. The criteria for the bug's existence is whether the bug could be reproduced. However, reproducibility does not always reflect the bug's existence. First, the bug may exist but it is not triggered during testing, which is *unstable bug reproduction (T2).* The reason is complex and we provide a detailed analysis in §6.2.1. Second, the bug does not exist but a crash still happens. The reasons might be the triggering of irrelevant bugs but cause bisection does not recognize them, which we call *inaccurate bug triage (T3).* Under these situations, the cause bisection would misjudge the bug's existence and finally locate an incorrect result commit. *Unreliable kernel setup (T1)* will not directly lead to a failure in stages B and C, because cause bisection will skip these kernels and choose another one.

Restrictive Configuration. We notice that the developers also introduce restrictive mechanisms to stop the cause bisection in the middle. First, the total testing time is unpredictable because the number of tests may fluctuate greatly. So, the developers limit the running time of cause bisection to eight hours. If the *running time limit* is reached, the cause bisection will exit with no output. Second, the setup of the outdated kernel would easily fail, since the outdated kernel usually needs a different compiling and running environment from today. Thus, the developers make the cause bisection stop at version 4.6 even if the bug-inducing version is still undetermined after testing version 4.6. Once the *release version limit* is reached, the cause bisection will also exit with no output.

6.1.2 Statistical Analysis of Failure Causes. To quantitatively evaluate the significance of failure reasons, we count the percentage of different causes by analyzing the cause bisection log for all the incorrect and no-output cases in our dataset, with the help of the ground-truth commits. We check whether the test on each commit in the bisection log derives a correct result. We only consider the first error that the cause bisection's judgment of the bug's existence is inconsistent with what it should be because studying the first error is enough to comprehend why the cause bisection fails since the nature of the bisection algorithm: the subsequent results after the first error are no longer reliable.

Table 2: Possible Failure Causes

Reason	Stage	Consequence
C1 - Running Time Limit	B, C	No Output
C2 - Release Version Limit	B	No Output
T1 - Unreliable Kernel Setup	A	No Output
T2 - Unstable Bug Reproduction	A	No Output
T2 - Unstable Bug Reproduction	B, C	Incorrect
T3 - Inaccurate Bug Triage	B, C	Incorrect

We design the Algorithm 1 to find the first error in the bisection log. In lines 1 to 4, we extract the crash commit from the bisection log (GetCrashCommit) and check whether there is any record indicating the kernel is not successfully built and booted (SetupKernelFailed). If so, the failure cause should be T1. In lines 5 to 19, we orderly examine whether an intermediate test gives an incorrect result. We extract all the tested commits from the log (GetTestedCommits). For a certain test, we determine the bug's existence using the ground-truth commit in our dataset. If the current commit to test is one of the post dominators of the ground-truth commit on the git tree, it means that the tested kernel includes the ground-truth commit and of course the bug, otherwise, the bug does not exist (lines 7-11). Then we infer which kind of error occurs. If the bug exists but the crash does not happen according to the log (GetReproduceResult), the cause should be T2 (lines 13-15). If the bug does not exist but it crashes actually, the cause should be T3 (lines 16-18). At last, if a timeout occurs or the version limit is reached, the cause should be C1 or C2. At last, we get the stage of the first erroneous test (GetStage) since the stage can be inferred from the log (Figure 3).

6.2 Results

The statistical results are demonstrated in Table 3. The most significant reason is T2, i.e., unstable bug reproduction, which accounts for 50% of the failures. The second significant reason is T3, inaccurate bug triage, accounting for 30.2%. These two reasons lead to more than 80% failures in total. The hyper-parameter limit, i.e., C1, C2, contributes to about 17% of failures. It is worth noting that, although we think T2 and T3 would not cause no-output in stages B and C, the data in Table 3 imply that they actually include no-output cases since the number of cases failing because of T2 and T3 in stages B and C (508) exceeds the number of incorrect cases (438). We will discuss this inconsistency in §9.

Algorithm 1: Failure Cause Analysis.

```
Input: The ground-truth commit $commit_{gt}$ and the cause
       bisection log $log$.
Output: The stage $s$ where cause bisection makes the first
        mistake and the cause $c$ of this mistake.
$commit_{crash}$ ← GetCrashCommit($log$);
if $SetupKernelFailed(commit_{crash})$ then
    return $s$ ← GetStage($commit_{crash}$, $log$), $c$ ← T1;
end
$commits_{tested}$ ← GetTestedCommits($log$);
for $commit$ in $commits_{tested}$ do
    $gt$ ← whether $commit \in$ PostDominators($commit_{gt}$);
    status ← GetReproduceResult($commit$, $log$);
    if $gt == True$ and $status == notCrash$ then
        return $s$ ← GetStage($commit$, $log$), $c$ ← T2;
    end
    if $gt == False$ and $status == crash$ then
        return $s$ ← GetStage($commit$, $log$), $c$ ← T3;
    end
end
$commit_{last}$ ← GetLastCommit($log$);
if $Timeout(log)$ then
    return $s$ ← GetStage($commit_{last}$, $log$), $c$ ← C1;
end
if $ReachVersionLimit(log)$ then
    return $s$ ← GetStage($commit_{last}$, $log$), $c$ ← C2;
end
```

Table 3: Breakdown of the First Failure Causes. The number in the table is the number of cases belonging to each category.

Stage \ Cause	C1	C2	T1	T2	T3
Stage A	-	-	20 (2.8%)	58 (8.2%)	-
Stage B	13 (1.8%)	86 (12.2%)	-	241 (34.1%)	94 (13.3%)
Stage C	21 (3.0%)	-	-	54 (7.6%)	119 (16.9%)
All Stages	34 (4.8%)	86 (12.2%)	20 (2.8%)	353 (50.0%)	213 (30.2%)

6.2.1 *Analysis of T2 - Unstable Bug Reproduction.*

According to the result in Table 3, problem T2 influences all three stages. In stage A/B/C, the percentages of T2 are respectively 8.2%, 34.1%, and 7.6%.

In stage A, the cause bisection fails to reproduce the bug so it takes the bug as invalid and exits with no output. To increasingly understand the reason, we randomly sample 10 cases out of the 58 cases and investigate the root cause of the failure manually. We find that five of them are due to the nature of the bug, in which four involve race conditions, and one is probabilistically triggered because it relies on uncontrollable stack content. The rest of them are due to the difference in the compiler used in fuzzing and bisecting, in which for four cases the compiler used in bisecting does not support KASAN so the bug is not observed, and for the remaining one case the compiler optimizes out the buggy path.

In stages B and C, the failure of reproduction means that cause bisection does not observe a bug behavior on a bug-existing kernel. To understand the underlying root causes, we randomly sample 10 cases [1] out of all the 241 cases in stage B and 10 cases out of all the 54 cases in stage C (20 cases in total). Different from stage A, each bug has at least one reproducible commit, i.e., the starting commit of bisection. Thus, we compare the kernel behavior when the bug is reproduced or not to study why the PoC cannot trigger a bug behavior on a bug-existing kernel.

We find that twelve cases involve race conditions. The rest are due to the kernel changes. Among them, four cases are due to the lack of necessary configuration options on the commit being tested, because the kernel has changed a lot from the tested commit to the crash commit and the configuration for the crash commit does not enable the PoC's bug-triggering logic on the tested commit. The remaining four cases are due to the PoC for the crash commit cannot be directly applied on the tested commit because the logic or path constraints to trigger the bug have changed.

Finding 3: Unstable bug reproduction is one of the major causes of the failure of cause bisection. We observe the reasons include race conditions, compiler differences, and kernel changes.

6.2.2 *Analysis of T3 - Inaccurate Bug Triage.*

In our dataset, 30.2% of failures are due to the inaccurate bug triage, i.e., the cause bisection fails to determine whether the reproduced bug is the same as the original bug. We review the implementation of the cause bisection and find that cause bisection simply regards any reproduced bug as the same bug found by fuzzing without any checking.

To confirm that the T3 is indeed caused by irrelevant bugs, we try to reproduce the phenomenon that a crash occurs on the bug-free commit. Among all the 213 cases, the phenomenon is reproduced in 35 cases of them. We use a cross-patching strategy to validate that the crash triggered on the bug-free commit is irrelevant to the original bug. For the reproduced crash, we search the crash title on the Syzbot to determine whether it is a known bug. If so, we apply the corresponding patch on the bug-free commit and observe whether it still crashes. There are 17 cases that no longer crash after we apply the patch, indicating they really trigger the irrelevant bugs. Oppositely, we can also apply the patch of the original bug on the bug-free commit. If it still crashes, we can tell they also trigger the irrelevant bug. Five more cases are confirmed in this way. For the remaining 13 cases that conflict with the patches or the reproduced crash cannot be confirmed as a known bug, we perform manual analysis. Seven are confirmed as irrelevant bugs and others are unsure. In short, we believe that at least 29 out of 35 cases (83%) fail because of triggering an irrelevant bug, which is the major reason for T3.

Finding 4: Inaccurate bug triage is one of the major causes of the failure of cause bisection. The major reason is that irrelevant bugs disturb the cause bisection's judgment of the bug's existence.

6.2.3 *Analysis of C1 and C2 - Hyperparameter Limit.*

The time limit brings about 4.8% of failures and the version limit brings about 12.2% of failures. Firstly, We investigate C1. We sample 10 cases out of

[1] Manual analysis in this section is performed by two kernel developers with more than three-year experience. Every case is examined by both two developers. No divergence appears to the analyzing results. Due to the expensive cost of manual analysis, we only sample a few of bugs for analysis. Possible threats are discussed in §9.3.

the 34 cases, prolong the time limit to 72 hours and observe the cause bisection result. After 72 hours, all 10 cases finish. The average running time is 19.0 hours, while the longest is 41.4 hours. Among them, six cases give a correct result. We further investigate the reason why these cases need longer time. Reviewing the cause bisection log, we find that for these cases, the tests on middle commits for bisection in stage C often fails due to T1, making the cause bisection frequently re-selects the middle commit and performs plenty of extra tests. For the remaining cases, one fails because of unstable reproduction (T2). Three fail because of inaccurate bug triage (T3). Although prolonging the time limit could increase the success rate, failures still occur due to technical issues, implying that the time limit is not the major factor leading to the failure.

As for C2, we random sample 20 out of 86 cases failing with version limit, we relax the restriction on the version to unlimited and retry the cause bisection. However, all the cases still fail. Among them, 4 cases fail because they meet the T2 or T3. The remaining 16 cases fail due to the unsuccessful build or boot on the old Linux kernel before 3.16. Upon this observation, we speculate that the main reason cause bisection avoiding analyzing the outdated kernel is that it is nontrivial to boot these outdated kernels in the modern environment. Although it is a reasonable design, the cause bisection may overlook such long-standing bugs which may be occasionally found but have serious impact [2].

7 HELPFULNESS FOR BUG FIXING (RQ3)

To study how the cause bisection may affect the bug-fixing practice from various perspectives, we remove the cases in our dataset whose result of cause bisection has no impact on the bug fixing. The cases that need to be removed share a common character that the bug fixing is earlier than when the bug is found by Syzbot, implying that the bug has been fixed without the help of cause bisection results. This is because developers also keep auditing kernel code, making some bugs first found by developers instead of Syzbot and quickly fixed. After removing such cases, the number of cases we used in this section is 952 out of 1,070 cases in the whole dataset. Besides, we show the p-value of the Wilcoxon rank-sum test [53] that represents the probability of rejecting our conclusion to validate the significance of statistics.

7.1 Recommendation of Bug-fixing Developer

In practice, it is better to assign the same author of the bug-inducing commit to fix the bug. A developer familiar with bug-related functionality generally pays less effort to bug fixing than a non-familiar developer. We study whether the cause bisection result would help to notify a proper developer, i.e., the result commits include the author of the patch, which is a kind of fixer recommendation. The baseline is the notification policy with only crash reports, which was introduced in §2. The results are shown in Table 4.

First, we find that a correct result of cause bisection would notify a proper bug-fixing developer more significantly than incorrect results, with a p-value less than 10^{-24}. If the result of cause bisection is correct, about 70.5% bugs are notified to the author of the real-world patch, which is 2.78× chances compared to the incorrect results. Second, for correct results, we compare the notification policy of cause bisection and crash report. If only the crash report is available, 39.93% of the bugs can be notified properly, 0.43× less than the cause bisection result, implying that the cause bisection result is much more effective to recommend a proper developer than the crash report, with a p-value less than 10^{-10}.

Table 4: Effectiveness of Notification under Different Situations.

Bisection Status	#Bugs	#Bugs notified to the patch's author
Correct Bisection	308	217 (70.45%)
Incorrect Bisection	395	100 (25.32%)
No Bisection (Crash Report)	308	123 (39.93%)

Finding 5: The correct cause bisection result can help to recommend a proper developer to fix the bug.

7.2 Indication of Bug-fixing Location

Intuitively, the patch of a bug might be close to the code introducing the bug. So we measure the relationship between the modifications in the result commit and the patch commit. If the code modification in the bisection's result commits lies in a close location to the patch commit, the cause bisection result would help the developers fast focus on the possible fix location. We evaluate in three granularity, including the line level, function level, and file level.

Table 5: Relationship between Result Commit and Bug-fixing Location.

Bisection result	#Bugs	#Bugs modify same line	#Bugs modify same function	#Bugs modify same file
Correct	308	176 (57.14%)	250 (81.17%)	280 (90.91%)
Incorrect	395	8 (2.03%)	24 (6.08%)	67 (16.96%)

The results are shown in Table 5. At the line level, about 57% of the correct result commits modify the same line where the patch modifies, but incorrect results hardly cover the patch lines. At the function level, about 81% of the correct result commits modify the same function in the patch, but the ratio is only 6.08% for the incorrect results And at the file level, more than 90% of the correct result commits intersect the patch commit at the same file, which is the 5.4× probability of the incorrect results. In all three granularity, the chance of the code modification of correct result commits colliding with the patch exceeds the chance of incorrect results by a large margin. Thus, we conclude that a correct cause bisection result significantly shares a strong relationship with the patch. The p-value to reject this conclusion for line, function, and file level are respectively less than 10^{-35}, 10^{-64}, 10^{-62}.

Finding 6: The correct cause bisection result can help to indicate the location for bug fixing.

7.3 Impact on Bug-fixing Time

At last, we explore the potential impact of cause bisection results on bug-fixing time. The bug-fixing time is calculated by the difference between the patch application time and the bug-reporting time. We count the average bug-fixing time by the quantile respectively for the correct, incorrect, and no-output cause bisection results. The statistics are shown in Table 6.

Table 6: Impact on Bug-fixing Time (day).

Bisection result	#Bugs	25%	50%	75%	Avg.
No output	249 (26.16%)	5.0	22.0	107.0	114.22
Correct	308 (32.35%)	1.0	5.0	15.0	22.73
Incorrect	395 (41.49%)	3.0	13.0	57.5	88.34
Total	952	3.0	9.0	45.0	73.88

The bug-fixing time with a no-output cause bisection is the longest because the developer cannot get any information and has to learn the bug from scratch. The average bug-fixing time with a correct bisection result is 22.73 days, saving about 80% of time when a cause bisection result is inaccessible and 74% of time when a cause bisection result is incorrect. Half of the bugs can even be fixed within five days if they have a correct cause bisection result.

Finding 7: The correct cause bisection result can help to speed up the bug-fixing practice.

Another interesting observation is that even an incorrect bisection result would facilitate bug fixing compared to no bisection results. It implies that improving the stability of the cause bisection to give a valid result, i.e., providing at least one result commit, is somewhat meaningful.

The p-value is less than 10^{-9} to reject the finding that the correct result brings a shorter fixing time than incorrect results, while less than 0.01 to reject the finding that the incorrect result brings a shorter fixing time than no output.

8 OPPORTUNITIES FOR FUTURE WORK (RQ4)

After answering the research questions, we conclude that cause bisection is quite valuable in real-world debugging practice since it facilitates the bug fixing progress. Based on the failure reasons we observed, we discuss the possible improvements.

8.1 Error-tolerant Bisection Algorithm

First, we observe that the intermediate testing results are usually unreliable. However, the correctness of bisection is based on the assumption that every testing result should be correct. Thus, a possible direction is to enhance the cause bisection even if the assumption is broken. In other words, the bisection algorithm should be tolerant of the inaccuracy of the intermediate testing results.

A topic called Noisy Binary Search has been studied for a long time [20, 30, 40]. Waeber et al. [56] discuss a probabilistic bisection algorithm (PBA) proposed by Horstein et al. [34]. The insight is that although intermediate testing results are not always reliable, we could still estimate the confidence of the results and use the confidence to update a probability distribution model to locate the bug-inducing object, which means even if an error occurs on the intermediate testing result, it still has a chance to give a correct result in the end. Since the noises in the real world are inevitable, combining error-tolerant searching algorithms would be a promising research direction like [33].

8.2 Better Software Testing

Since the current cause bisection supposes the accuracy of intermediate tests, improving the reliability of testing is also a possible way. According to the evaluation results, we find that the reliability of bug reproduction affects the bisection progress most significantly. To be specific, an existing bug may not be stably reproduced, and an irrelevant bug may be mistaken as the target bug. Besides cause bisection, bug reproduction is also critical to other security-related applications, like root cause analysis[25].

Reproduction Stabilization. Even though a bug exists in a certain version of the software, it may not be stably triggered on this version. For the kernel bugs in our dataset, the causes mainly include race conditions, compiling options, and code changes.

(1) For racing (concurrency) bugs, many existing works [18, 24, 35, 37, 47, 58, 62, 63, 65] have explored how to reproduce them. For example, Musuvathi et al. [47] propose CHESS to deterministically schedule the threads and search for a thread-interleaving order that may trigger a kernel bug. Aviram et al. [18] design a special OS kernel to provide a deterministic execution environment. Jeong et al. [37] leverage virtualization to control and search for a fine-grained data-racing order to trigger the bug. Huang et al. [36] analyze the possible memory dependencies according to the execution log and find a scheduling order to reproduce the bug via constraint solving. Leesatapornwongsa et al. [42] propose FlakeRepro to reproduce concurrency failures for .NET applications. During cause bisection, when the PoC looks to be concurrent, these techniques may help stabilize the bug reproduction. **(2)** For compiling options, the reproducer should carefully align the settings between fuzzing and reproducing, to keep the environment as similar as possible. **(3)** For the code changes, the technique named PoC migration may address this problem. Some existing work like [29, 39] can help adapt the original PoC on a certain version, e.g. the version for fuzzing, to another version, e.g. the version to confirm the bug's existence. If the PoC can be successfully migrated, the existence of the bug can be confirmed.

Combining Multiple Bug Detection Techniques. Besides making the testing robust, different techniques rather than testing can be involved to justify the bug's existence. One possible approach is static vulnerability detection[43]. The cause bisection could combine the results of static analysis and dynamic testing to determine whether the bug exists in a certain software version.

Irrelevant Bug Recognition. The current version of cause bisection does not check whether the reproduced bug is the same as the original bug, resulting in false judgment on the bug's existence. The identification of the same bugs is non-trivial[31, 38]. To mitigate the influence of irrelevant bugs with relatively low overhead, we suggest utilizing some easily available information to make a simple verification. When a crash is triggered during cause bisection, the bug title, crashed object, functions in call trace, etc., can be extracted to retrieve whether a bug is a known bug. Once a bug is suspected to be a known bug, cause bisection could automatically apply the corresponding patch and test again. If it no longer crashes, the reproduced crash might be a false positive.

9 THREATS TO VALIDITY

9.1 Reliability of Ground Truth

We take the "fixes" tag provided by the developer in a patch as the ground truth for cause bisection. However, developers may make mistakes sometimes. Although we introduce heuristics to filter out some incorrect ground truths, it is impossible to remove all of them.

This may harm the validity of our conclusion on cause bisection's effectiveness. To address this concern, we sample 90 cases from our dataset and perform manual confirmation. Only four (4.4%) of them are marked with incorrect ground truth. Although they are incorrect, we observe that none of them truly mislead our judgment on the correctness of the bisection result. In the future, we intend to improve the reliability of the ground truth of our dataset and study the possible reasons why the developer makes mistakes.

9.2 Consistency of Theoretical Analysis and Statistics

In our theoretical analysis, the reason T2 and T3 would not lead to no output in stage B and stage C. However, the number of cases that fail because of T2 and T3 in stage B and stage C (Table 3) is beyond the number of incorrect cases, meaning the cases failing because of T2 and T3 also include some no-output cases. It is due to we only consider the first error of the bisection process. Although T2 and T3 would not directly lead to no output, they may combine with other no-output reasons and finally result in no output. For such cases, we still think the failing reasons belong to T2 or T3 rather than a typical no-output reason like timeout because even if they provide a result commit, it must be wrong. So, the statistical results are consistent with our theoretical category of failure reasons.

9.3 Significance of Discovered Causes behind T2

In §6, we only sample 30 cases to study the underlying causes of unstable reproduction because it costs expensive manual effort. Indeed, this approach may not cover all the possible causes. But we think we have covered the most significant causes that occupy more than 20% among all causes. If such a significant cause occurs, random sampling 30 cases out of 353 cases provide a 99.9% probability of discovering this cause. Thus, we believe that sampling 30 cases is reasonable without harming the validity of our findings.

9.4 Generalization of Conclusions

Although our study is performed on the Linux kernel, we believe our conclusions are generalizable to a certain degree. First, the major issues we found like bug reproducibility [48] and same-bug identification [31] are the common challenges for various software testing and bug analysis. Second, some of our constructive proposals, e.g. involving probabilistic bisection, are not limited to Syzbot's cause bisection. Other industrial cause bisection can also introduce such methods since they share a similar bisection algorithm.

10 RELATED WORKS

10.1 Evaluation of Bug-inducing Commit Localization

In general, there are two kinds of evaluation: (1) academic research that proposes a new method would evaluate its method with previous works, and (2) empirical study for a series of algorithms like VSS [28, 50]. Both kinds of evaluation are just for clean algorithms rather than industrial systems, which would not consider system preparation like the kernel setup and system setting like version/time limitation. To the best of our knowledge, the only evaluation for an industrial bug-inducing commit localization system is performed manually with only 118 bugs before March 2019 [1]. We perform a larger scale and more comprehensive study on a more recent dataset that aims to draw a systematic image of the current industrial cause bisection system.

10.2 Research on Factors that Impact Bug-fixing

To improve the bug-fixing practice and better schedule software releases, it is essential to understand the factors that impact bug-fixing. Othmane et al. [22] conduct a qualitative study at the SAP company and identified 65 factors that impact the fixing time of vulnerabilities Furthermore, Othmane et al. [21] quantitatively investigate the major factors that impact the time of fixing a given security issue and propose machine learning models to predict the fixing time. Zhang et al. [64] examine factors affecting bug-fixing time from bug reports, source code involved in the fix, and code changes. Arora et al. [17] study how the vulnerability disclosure affects patch release time. Different from the above research, we study the impact of an auxiliary technique for bug fixing, i.e., cause bisection, in a real-world industrial scenario. A recent study [14] discusses the factors that shorten the fixing time of fuzzer-exposed bugs including cause bisection. Our work increasingly studies how the cause bisection affects bug fixing from various aspects like bug-fixing indication and developer recommendation.

11 CONCLUSION

In this paper, we comprehensively study the performance, limitations, and impacts of an industrial cause bisection system, i.e., the cause bisection of Syzbot. First, we build a dataset containing 1,070 bugs from the database of Syzbot, in which we collect the cause bisection result and ground-truth bug-inducing commit for each case. We find that only one-third of the bugs find a correct bug-inducing commit through case bisection, showing the limited effectiveness of cause bisection in the real world. The most significant problems leading to the failure of cause bisection are unstable bug reproduction and inaccurate bug triage, resulting in more than 80% of errors. Moreover, we figure out the cause bisection significantly facilitates bug-fixing practice, including recommending bug-fixing developers, indicating bug-fixing locations, and accelerating the bug-fixing time. At last, we propose possible suggestions that may improve the performance of the current version of cause bisection and discuss the promising research directions in the future.

ACKNOWLEDGMENTS

We appreciate the anonymous reviewers for their valuable comments. This work was supported in part by the National Natural Science Foundation of China (62172105,62172104,62102091, 62102093) and the Funding of Ministry of Industry and Information Technology of the People's Republic of China under Grant TC220H079. Yuan Zhang was supported in part by the Shanghai Rising-Star Program 210A1400700 and the Shanghai Pilot Program for Basic Research-Fudan University 21TQ1400100 (21TQ012). Min Yang is the corresponding author, and a faculty of Shanghai Institute of Intelligent Electronics & Systems and Engineering Research Center of Cyber Security Auditing and Monitoring.

REFERENCES

[1] 2019. Syzbot bisection analysis. https://groups.google.com/g/syzkaller/c/sR8aAXaWEF4/m/tTWYRgvmAwAJ.

[2] 2021. Serious Security: The Linux kernel bugs that surfaced after 15 years. https://nakedsecurity.sophos.com/2021/03/17/serious-security-the-linux-kernel-bugs-that-surfaced-after-15-years.

[3] 2023. A guide to the Kernel Development Process. https://www.kernel.org/doc/html/latest/process/submitting-patches.html.

[4] 2023. Bisection in OSV. https://github.com/google/osv.dev.

[5] 2023. Bisection in Syzbot. https://github.com/google/syzkaller/blob/master/docs/syzbot.md#bisection.

[6] 2023. ClusterFuzz. https://google.github.io/clusterfuzz.

[7] 2023. Mailing list about Syzbot-bug-39b72114. https://groups.google.com/g/syzkaller-bugs/c/RYROqSKy2qY/m/risLySvtEQAJ.

[8] 2023. Mailing list about Syzbot-bug-c54b440c. https://groups.google.com/g/syzkaller-bugs/c/DCP4rJ6UInM/m/YH25wC4fCgAJ.

[9] 2023. OSS-Fuzz. https://github.com/google/oss-fuzz.

[10] 2023. Syzbot. https://github.com/google/syzkaller/blob/master/docs/syzbot.md.

[11] 2023. Syzbot dashboard. https://syzkaller.appspot.com.

[12] 2023. Syzbot dashboard of Linux upstream. https://syzkaller.appspot.com/upstream.

[13] 2023. Syzkaller. https://github.com/google/syzkaller.

[14] Rui Abreu, Franjo Ivančić, Filip Nikšić, Hadi Ravanbakhsh, and Ramesh Viswanathan. 2021. Reducing time-to-fix for fuzzer bugs. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1126–1130.

[15] Gabin An, Jingun Hong, Naryeong Kim, and Shin Yoo. 2023. Fonte: Finding Bug Inducing Commits from Failures. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 589–601.

[16] Gabin An and Shin Yoo. 2021. Reducing the search space of bug inducing commits using failure coverage. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1459–1462.

[17] Ashish Arora, Ramayya Krishnan, Rahul Telang, and Yubao Yang. 2010. An empirical analysis of software vendors' patch release behavior: impact of vulnerability disclosure. *Information Systems Research* 21, 1 (2010), 115–132.

[18] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. 2012. Efficient system-enforced deterministic parallelism. *Commun. ACM* 55, 5 (2012), 111–119.

[19] Lingfeng Bao, Xin Xia, Ahmed E Hassan, and Xiaohu Yang. 2022. V-SZZ: automatic identification of version ranges affected by CVE vulnerabilities. In *Proceedings of the 44th International Conference on Software Engineering*. 2352–2364.

[20] Michael Ben-Or and Avinatan Hassidim. 2008. The bayesian learner is optimal for noisy binary search (and pretty good for quantum as well). In *Proceedings of the 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. 221–230.

[21] Lotfi Ben Othmane, Golriz Chehrazi, Eric Bodden, Petar Tsalovski, and Achim D Brucker. 2017. Time for addressing software security issues: Prediction models and impacting factors. *Data Science and Engineering* 2 (2017), 107–124.

[22] Lotfi ben Othmane, Golriz Chehrazi, Eric Bodden, Petar Tsalovski, Achim D Brucker, and Philip Miseldine. 2015. Factors impacting the effort required to fix security vulnerabilities: An industrial case study. In *Information Security: 18th International Conference (ISC)*. Springer, 102–119.

[23] Ranjita Bhagwan, Rahul Kumar, Chandra Sekhar Maddila, and Adithya Abraham Philip. 2018. Orca: Differential Bug Localization in {Large-Scale} Services. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 493–509.

[24] Francesco A Bianchi, Mauro Pezzè, and Valerio Terragni. 2017. Reproducing concurrency failures from crash stacks. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 705–716.

[25] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. 2020. AURORA: Statistical crash analysis for automated root cause explanation. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*. 235–252.

[26] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP)*. 711–725.

[27] Agnieszka Ciborowska and Kostadin Damevski. 2022. Fast changeset-based bug localization with BERT. In *Proceedings of the 44th International Conference on Software Engineering*. 946–957.

[28] Daniel Alencar Da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E Hassan. 2016. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering* 43, 7 (2016), 641–657.

[29] Jiarun Dai, Yuan Zhang, Hailong Xu, Haiming Lyu, Zicheng Wu, Xinyu Xing, and Min Yang. 2021. Facilitating vulnerability assessment through PoC migration. In *Proceedings of the 28th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 3300–3317.

[30] Dariusz Dereniowski, Aleksander Lukasiewicz, and Przemyslaw Uznanski. 2021. An Efficient Noisy Binary Search in Graphs via Median Approximation. In *Combinatorial Algorithms - 32nd International Workshop (Lecture Notes in Computer Science, Vol. 12757)*. Springer, 265–281.

[31] Jayati Deshmukh, KM Annervaz, Sanjay Podder, Shubhashis Sengupta, and Neville Dubash. 2017. Towards accurate duplicate bug retrieval using deep learning techniques. In *2017 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 115–124.

[32] Zhen Yu Ding and Claire Le Goues. 2021. An Empirical Study of OSS-Fuzz Bugs. In *Proceedings of the 18th IEEE/ACM International Conference on Mining Software Repositories (MSR)*. 131–142.

[33] Tim AD Henderson, Bobby Dorward, Eric Nickell, Collin Johnston, and Avi Kondareddy. 2023. Flake Aware Culprit Finding. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 362–373.

[34] Michael Horstein. 1963. Sequential transmission using noiseless feedback. *IEEE Transactions on Information Theory* 9, 3 (1963), 136–143.

[35] Jeff Huang and Charles Zhang. 2012. Lean: Simplifying concurrency bug reproduction via replay-supported execution reduction. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 451–466.

[36] Jeff Huang, Charles Zhang, and Julian Dolby. 2013. Clap: Recording local executions to reproduce concurrency failures. *Acm Sigplan Notices* 48, 6 (2013), 141–152.

[37] Dae R Jeong, Minkyu Jung, Yoochan Lee, Byoungyoung Lee, Insik Shin, and Youngjin Kwon. 2023. Diagnosing kernel concurrency failures with AITIA. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 94–110.

[38] Zhiyuan Jiang, Xiyue Jiang, Ahmad Hazimeh, Chaojing Tang, Chao Zhang, and Mathias Payer. 2021. Igor: Crash Deduplication Through Root-Cause Clustering. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 3318–3336.

[39] Zheyue Jiang, Yuan Zhang, Jun Xu, Xinqian Sun, Zhuang Liu, and Min Yang. 2023. AEM: Facilitating Cross-Version Exploitability Assessment of Linux Kernel Vulnerabilities. In *Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP)*. 588–603.

[40] Richard M Karp and Robert Kleinberg. 2007. Noisy binary search and its applications. In *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 881–890.

[41] Kyungtae Kim, Dae R. Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid Fuzzing on the Linux Kernel. In *Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS)*.

[42] Tanakorn Leesatapornwongsa, Xiang Ren, and Suman Nath. 2022. FlakeRepro: automated and efficient reproduction of concurrency-related flaky tests. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1509–1520.

[43] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. 2022. An empirical study on the effectiveness of static C code analyzers for vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 544–555.

[44] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*. 1949–1966.

[45] Dongliang Mu, Yuhang Wu, Yueqi Chen, Zhenpeng Lin, Chensheng Yu, Xinyu Xing, and Gang Wang. 2022. An In-depth Analysis of Duplicated Linux Kernel Bug Reports. In *Proceedings of the 29th Annual Network and Distributed System Security Symposium (NDSS)*.

[46] Vijayaraghavan Murali, Lee Gross, Rebecca Qian, and Satish Chandra. 2021. Industry-scale ir-based bug localization: A perspective from facebook. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 188–197.

[47] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs.. In *OSDI*, Vol. 8.

[48] Mohammad Masudur Rahman, Foutse Khomh, and Marco Castelluccio. 2020. Why are Some Bugs Non-Reproducible?:–An Empirical Investigation using Data Fusion–. In *2020 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 605–616.

[49] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS)*.

[50] Giovanni Rosa, Luca Pascarella, Simone Scalabrino, Rosalia Tufano, Gabriele Bavota, Michele Lanza, and Rocco Oliveto. 2021. Evaluating szz implementations through a developer-informed oracle. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 436–447.

[51] Emre Sahal and Ayse Tosun. 2018. Identifying bug-inducing changes for code additions. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–2.

[52] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*.

167–182.

[53] David J Sheskin. 2020. *Handbook of parametric and nonparametric statistical procedures.* crc Press.

[54] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes? *ACM sigsoft software engineering notes* 30, 4 (2005), 1–5.

[55] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. 2021. HEALER: Relation Learning Guided Kernel Fuzzing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP).* 344–358.

[56] Rolf Waeber, Peter I Frazier, and Shane G Henderson. 2013. Bisection search with noisy responses. *SIAM Journal on Control and Optimization* 51, 3 (2013), 2261–2279.

[57] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V. Krishnamurthy, and Nael Abu-Ghazaleh. 2021. SyzVegas: Beating Kernel Fuzzing Odds with Reinforcement Learning. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security).* 2741–2758.

[58] Dasarath Weeratunge, Xiangyu Zhang, and Suresh Jagannathan. 2010. Analyzing multicore dumps to facilitate concurrency bug reproduction. In *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems.* 155–166.

[59] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: Locating bugs from software changes. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE).* 262–273.

[60] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.

[61] Rongxin Wu, Ming Wen, Shing-Chi Cheung, and Hongyu Zhang. 2018. Changelocator: locate crash-inducing changes based on crash reports. *Empirical Software Engineering* 23 (2018), 2866–2900.

[62] Tingting Yu, Tarannum S Zaman, and Chao Wang. 2017. DESCRY: reproducing system-level concurrency failures. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering.* 694–704.

[63] Xiang Yuan, Chenggang Wu, Zhenjiang Wang, Jianjun Li, Pen-Chung Yew, Jeff Huang, Xiaobing Feng, Yanyan Lan, Yunji Chen, and Yong Guan. 2015. ReCBuLC: reproducing concurrency bugs using local clocks. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 824–834.

[64] Feng Zhang, Foutse Khomh, Ying Zou, and Ahmed E Hassan. 2012. An empirical study on factors impacting bug fixing time. In *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE).* 225–234.

[65] Gefei Zuo, Jiacheng Ma, Andrew Quinn, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. 2021. Execution reconstruction: Harnessing failure reoccurrences for failure reproduction. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation.* 1155–1170.

Received 2024-02-08; accepted 2024-04-18