# An In-depth Analysis of Duplicated Linux Kernel Bug Reports

*Dongliang Mu, ‡Yuhang Wu, ‡Yueqi Chen, ‡Zhenpeng Lin, §Chensheng Yu
¶‡Xinyu Xing, †Gang Wang

*School of Cyber Science and Engineering, Huazhong University of Science and Technology
†University of Illinois at Urbana-Champaign Urbana §George Washington University
‡The Pennsylvania State University ¶Northwestern University
dzm91@hust.edu.cn, {yuhang, ycx431, zplin}@psu.edu, i@shiki7.me,
xinyu.xing@northwestern.edu, gangw@illinois.edu

***Abstract*—In the past three years, the continuous fuzzing projects Syzkaller and Syzbot have achieved great success in detecting kernel vulnerabilities, finding more kernel bugs than those found in the past 20 years. However, a side effect of continuous fuzzing is that it generates an excessive number of crash reports, many of which are "duplicated" reports caused by the same bug. While Syzbot uses a simple heuristic to group (deduplicate) reports, we find that it is often inaccurate. In this paper, we empirically analyze the duplicated kernel bug reports to understand: (1) the prevalence of duplication; (2) the potential costs introduced by duplication; and (3) the key causes behind the duplication problem. We collected all of the *fixed kernel bugs* from September 2017 to November 2020, including 3.24 million crash reports grouped by Syzbot under 2,526 bug reports (identified by unique bug titles). We found the bug reports indeed had duplication: 47.1% of the 2,526 bug reports are duplicated with one or more other reports. By analyzing the metadata of these reports, we found undetected duplication introduced extra costs in terms of time and developer efforts. Then we organized Linux kernel experts to analyze a sample of duplicated bugs (375 bug reports, unique 120 bugs) and identified 6 key contributing factors to the duplication. Based on these empirical findings, we proposed and prototyped actionable strategies for bug deduplication. After confirming their effectiveness using a ground-truth dataset, we further applied our methods and identified previously unknown duplication cases among open bugs.**

## I. Introduction

Kernel is the most important software component of an operating system (OS), the security of which determines the security of the entire OS and user applications. *Vulnerabilities* in kernel programs are often considered to be more severe than those in user programs [26]. Due to the high privilege of kernel programs, they have been an attractive target of major attacks, with well-known examples such as WannaCry [11], DirtyCow [8], and BleedingTooth [13].

To proactively detect and patch kernel vulnerabilities, the security community has investigated significant efforts. These efforts include both developing more advanced fuzzing tools to detect new vulnerabilities [4], [67], [51] and organizing security analysts and kernel developers to analyze the reported bugs and develop patches.

The progress of kernel bug detection has been slow in the past 20 years until recently when Google initialized Syzkaller [73] and Syzbot [74] projects in 2016. These are open-source projects that automatically and continuously fuzz main Linux kernel branches to find bugs. In addition to the advanced fuzzing techniques (Syzkaller), another key advantage is that the system (Syzbot) produces standard crash reports and structured information fields (e.g., vulnerable kernel versions, kennel configurations), which makes it easier for security analysts to reproduce bugs and analyze root causes. These efforts have been vastly successful. As of November 2020, the system has found 3,736 kernel bugs in just three years, which is more than the total number of kernel bugs identified in *the past 20 years before Syzkaller* [31].

**Bug Report Duplication Problem.** While continual fuzzing of Syzbot has significantly improved the efficiency of kernel bug discovery, it also produces an excessive amount of crash reports. In the past three years, Syzbot has generated over 10 million crash reports, the vast majority of which are "duplicated", meaning that the crashes are triggered by the same bugs. Considering that security analysts need to *manually* analyze these reports to assess the severity of the bug and pinpoint the root cause, it is highly desirable to group the crash reports caused by the same bug together. Currently, Syzbot relies on a simple heuristic to perform deduplication: if the crashes share the same *crash function* and *crash type*, then they will be grouped under the same bug report, sharing the same bug title.

Unfortunately, this heuristic-based deduplication method is not accurate. Anecdotally on Syzbot dashboard, we have observed that certain crash reports caused by the same bug were not successfully grouped together (because they have different crash functions or crash types). As a result, the bug reports about the same bug were treated as new (different) bugs, and then were assigned to different analysts. On one hand, multiple groups of analysts working on the same bug in parallel without communicating with each other leads to inefficiency (i.e., redundant manual effort). On the other hand,

Network and Distributed Systems Security (NDSS) Symposium 2022
24-28 April 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.24159
www.ndss-symposium.org

a limited view of the diverse bug behaviors of the same bug may lead to incomplete patches [21].

**Goals and Approaches.** In this paper, we empirically analyze the duplicated kernel bug reports from Syzbot. We define duplicated bug reports as those that *share the same root cause*. Our goal is to understand: (1) the prevalence of duplication; (2) potential costs introduced by duplication; and (3) the causing factors to the report duplication. Based on the insights obtained from the empirical analysis, we further explore low-cost solutions for bug deduplication.

A key challenge in our study is to obtain the "ground-truth" for bug report duplication. While this challenge is difficult to resolve for open bugs that are currently being analyzed by developers, we could group bugs that are already fixed. The idea is to link bug reports based on their "patches" — if multiple bug reports are fixed/closed by the same patch, these bugs are highly likely to be the same bug. After this initial grouping, we then manually analyze the bugs to confirm duplication based on root causes. In this work, we have collected *all* of the fixed kernel bugs reported on Syzbot from September 2017 to November 2020. This includes 2,526 bug reports and over 3.24 million crash reports. Based on their patches, we group these bug reports into 1,686 unique kernel bug groups. Out of the 2,526 bug reports, 1,191 (47.1%) are duplicated with one or more other reports.

We answer the first two questions by analyzing the ground-truth bug groups. We observe that kernel bugs already take a long time to fix, even *without duplication* (the median open time is 70 days). Bug groups *with duplication* take an even longer time to close. Even after the first bug in the group is fixed, the other duplicated bugs will remain open for extra time, consuming valuable resources (e.g., developers' time). Also, not too surprisingly, bug groups with duplication involve more developers in the bug fixing process.

To answer the third question, the most effective approach is to manually analyze these kernel bugs. To ensure the reliability of results, we have organized security experts and developed rigorous analysis procedures (including peer-reviews). We select 120 bug groups that Syzbot failed to detect the duplication (120/351 = 34.2%). Under the guidance of an approved IRB protocol, 5 Linux kernel experts are organized to set up the environments, reproduce the reported bugs, and analyze the code changes and the developer's notes to figure out the causes of bug duplication. In total, we collectively identify 6 main contributing factors include different inputs, thread interleaving, memory dynamics, kernel versions and branches, different sanitizers, and inline function.

Based on the empirical findings, we propose and prototype five actionable strategies for bug deduplication during reporting time. For instance, we enhance the existing memory quarantine mechanism and replace the slab allocator with the slub allocator in the tested kernel to eliminate the influence of memory layout dynamics. Then we swap and run the PoC programs from potentially duplicated bug reports to observe their behaviors (i.e., mitigating the influence of kernel implementation and function inline). We inject delay to the kernel source code and run the PoC programs multiple times to fully expose the possible thread interleaving. To rule out the influence of different sanitizers, we run the PoC programs with the same sanitizer configuration. Finally, we compute the similarity of PoC programs using a customized Levenshtein distance to handle the different behaviors caused by input differences.

We evaluate these strategies with our ground-truth dataset and show that they can effectively identify duplicated bug report pairs, with a true positive rate of 80% and a false positive rate of 0.01%. Among the proposed techniques, the technique that handles "different inputs" is the only source of false positives, and the other four techniques are false-positive-free by design. We further apply our techniques to the *real-world open bugs* and confirm that they can identify *previously unknown* bug duplication cases. From both fixed and open bugs, we have found cases where the developers were misled to produce incomplete patches due to a limited view of the diverse bug behaviors.

**Contributions.** In summary, we have three key contributions:

- First, we empirically analyze duplicated reports of Linux kernel bugs, and identify the limitations of existing deduplication heuristics. We show that the undetected duplication introduces extra costs (time and developer efforts) and even produces incorrect patches.
- Second, we organize Linux kernel experts to perform an in-depth analysis of duplicated bug reports. We identify six key contributing factors to duplication.
- Third, we propose and prototype a series of strategies to alleviate the bug duplication problem. We evaluate them against both ground-truth data and current open bugs to demonstrate their effectiveness. To facilitate future works, we will release our code and dataset with this paper.

Our work provides new insights into the causing factors of kernel bug deduplication, and introduces an initial solution. We have shared our results and findings with the Syzkaller and Syzbot teams (and some kernel developers), and have received positive feedback. In the end of the paper, we discuss the open challenges to kernel bug deduplication. We believe further research is needed in order to fully address the problem.

## II. Background and Motivation

In this section, we describe the background of kernel fuzzing and bug reports, and introduce Syzkaller and Syzbot. Then we describe our problem setup and research goals.

**Syzkaller: Kernel Fuzzing to Detect Bugs.** To harden the security of kernel programs, the security community has developed fuzzing tools to discover kernel bugs [73], [4], [67], [51]. Among existing fuzzers, Syzkaller [73] is by far the most successful efforts *in practice*. Syzkaller is an open-source project initiated by Google in 2016 (popularized in 2017). As of November 2020, Syzkaller has found 3,736 kernel bugs (2,526 of them are now patched) in just three years, which is more than the total number of kernel bugs identified in *the past 20 years before Syzkaller [31].*

Syzkaller has several advanced designs. First, it leverages a declarative description of syscall interface to manipulate programs (sequences of syscalls), and uses code coverage feedback as guidance to explore all the kernel code space. Second, syzkaller leverages the fault injection mechanism [3]

in Linux kernel to inject failures (*e.g.,* allocation failures) into the runtime execution of system calls. After a kernel bug is found, Syzkaller will try to generate (and minimize) syz and C reproducers. Finally, Syzkaller coordinates with many different sanitizers (*e.g.,* KASAN [68], [6], KMSAN [70], [7], KCSAN [12], KUBSAN [17]), kernel detection mechanisms (*e.g.,* KMEMLEAK [5], ODEBUG [10]) and other pre-defined assertions (*e.g.,* BUG_ON, WARN_ON) to detect kernel vulnerabilities at runtime. These detection tools allow syzkaller to expose all mainstream security bugs such as memory error bugs (*e.g.,* Memory Leak, Null Pointer Dereference, Use-After-Free (UAF), pre-defined assertion (*e.g.,* WARN, BUG), deadlocks and concurrency bugs. Therefore, Syzkaller can cover a highly diverse set of bugs and bug types.

**Syzbot: Continuous Kernel Fuzzing and Reporting.** For a long time (before Syzkaller), running kernel fuzzers and reporting bugs have been almost exclusively manual efforts. The lack of automation and bug reporting standards has created significant difficulty in bug reproduction and patching [61]. To automate bug discovery and reporting, the Syzkaller team further developed a *continuous* fuzzing system for kernel programs called *Syzbot* [74]. Syzbot system continuously and automatically updates and fuzzes main Linux kernel branches (*e.g.,* upstream, linux-stable) with different Syzkaller instances. Once bugs are found, they will be *automatically* reported to corresponding kernel developers with standardized information (e.g., crash reports). Analysts from the kernel community will first analyze the bug (manually) to confirm its validity. Then kernel developers will analyze the root cause of the bug, and develop a patch to fix the bug.

To facilitate bug analysis and information sharing, Syzbot provides an open forum (called "Syzbot dashboard") to list and keep track of the reported kernel bugs. Each bug has its own web page that contains key information about the bug, such as the vulnerable kernel versions, the kernel configuration file, the Syzkaller repositories, reproducer (syz repro or C repro). Both syz repro and C repro are PoC (Proof-of-Concept) files to reproduce the crash. The configuration file shows which sanitizers are enabled in the corresponding kernel crash. Finally, it is worth noticing that fixed bugs all have a "Fix commit" which is the kernel commit (patch) that has fixed the underlying bug.

A key side effect of continuous fuzzing is it generates a large number of inputs to trigger bugs, which produces an excessive number of crash reports. In recent three years, Syzbot has produced over 10 million crash reports, many of which were actually triggered by the same bug. Such a duplication level could negatively impact the efficiency of kernel developers who need to analyze the reported bugs *manually*. Currently, Syzbot follows a simple heuristic to group (or deduplicate) crash reports. If the crashes appear at the same *function* and share the same *crash type*, then these crash reports will be grouped together, under the same *bug title*. For example, under the bug title "`KASAN: use-after-free Read in map_lookup_elem`", all the crash reports share the same crash function (i.e., `map_lookup_elem`) and the crash type (i.e., "`KASAN: use-after-free Read`").

**Limitations of the Current Deduplication Method.** The current deduplication method is coarse-grained and error-prone. Anecdotally on Syzbot dashboard, we observed certain crash reports caused by the same bug were not successfully grouped under the same *bug titles*. Instead, they are treated as distinct bugs and are assigned to different analysts. Such "duplicated" open bugs lead to concerning problems. First, multiple groups of analysts working on the same bug in parallel without communicating is an inefficient way of using the analysts' time. Second, once one of the duplicated bugs is fixed, there will be extra delays to close other bugs that share the same root causes (i.e., wasting analysts' time if they keep working on them). Third, without grouping these bugs, analysts do not have the complete view of the bug behaviors, which can lead to incomplete and incorrect patches. We discovered real cases which will be presented later.

For these reasons, we want to empirically understand the bug report duplication problem on Syzbot, and answer the following questions. First, how prevalent is bug report duplication on Syzbot? Does duplication indeed introduce extra costs to analyzing and patching the bug (Section III–IV)? Second, what are the main causes to the report duplcation (Section V)? Third, how can we effectively deduplicate kernel bug reports (Section VI)?

## III. Methodology and Dataset

In this section, we first define our problem scope and then describe the collected dataset to measure the prevalence of bug report duplication. Finally, we describe our workflow to identify the causes of the duplication.

### A. Method Overview

**Definition of the Root Cause of a Bug.** The root cause of a bug is defined as the faulty code leading the Linux kernel into an abnormal state. Take the kernel bug `#bbeb6e43` [24] as an example. The root cause of this bug is in the function `array_map_alloc`. If the `attr->max_entries` field goes beyond a threshold (i.e., `0xfffffffd`), an integer overflow will occur in the variable `array_size`, causing `array_map_alloc` to allocate an oversized buffer. The oversized buffer eventually leads to a general page fault (GPF) or an Out-of-Bound (OOB) memory access.

**Definition of Unique Bug.** A bug is uniquely defined by its root cause. In other words, if multiple crash reports share the same root cause, then we define them as duplicated reports. At the high level, our idea is to collect historical crash reports of Linux kernels and link reports that share the same root cause (*i.e.,* reports of the same bug). Based on the linked reports, we develop an analysis procedure to systematically examine the reasons for duplication.

We first define the key terms used in this paper. In Figure 10 in the Appendix, we use an example to show the hierarchical relationships between bug groups, bug titles, and crash reports. When Syzbot reports a crash, it automatically generates a *bug title* formed by a crash function and crash type. For example, the title "`UBSAN: shift-out-of-bounds in mceusb_dev_recv`" means the crash happens on function "`mceusb_dev_recv`" and the crash type is "`UBSAN: shift-out-of-bounds`". Under continuous fuzzing, it is common for the same bug to be triggered multiple times since

a bug would remain unfixed a certain amount of time. Syzbot currently groups these crash reports under the same *bug title*. In the example of Figure 10, Bug Title B has $N$ crash reports under the same title.

As mentioned before, grouping crash reports using crash function and crash type is often inaccurate, because the same bug may exhibit different crash behaviors (i.e., with different crash functions or crash types). In the example of Figure 10, Bug Titles A, B, C are in fact triggered by the same bug and thus should have been grouped under the same *bug group*. Here, the bug group represents the "ground-truth" unique bug.

### B. Problem Scope

The current crash deduplication method is coarse-grained and error-prone, manifesting two undesired outcomes: 1) false positives (FP)—bug reports with different root causes, grouped into the same title; and 2) false negatives (FN)—bug reports with the same root cause are not grouped under the same title.

By design, Syzbot can handle the false positive problem. Given a false positive case (i.e., bug reports with different root causes, grouped into the same title), kernel developers may only fix one of them during their first attempt, leaving the others unfixed. However, as a continuous fuzzing system, Syzbot will continue to fuzz the patched version and keep filing crash reports for the unfixed bugs. According to Syzbot developers [16], the fact that the new crash reports have the same title as the fixed one is an indication that there are other bugs unfixed, and thus developers will continue to work on it. To this end, the falsely grouped bugs will not be missed[1]. For this reason, our paper will focus on "false negative" cases (*i.e.,* bug report duplication), which can lead to duplicated effort by kernel developers and reduced efficiency.

Furthermore, we also do not consider the case where one crash reported by Syzkaller is triggered by multiple bugs. In practice, this situation is extremely rare. To have a crash tied to multiple bugs, two conditions must be satisfied: 1) the input to trigger multiple bugs need to be carefully crafted; 2) no sanitizers nor internal detection mechanisms are enabled during the fuzzing process, which will allow an error state to propagate sufficiently far away from the bug-triggering site. However, Syzkaller generates random inputs based on the code coverage feedback during kernel fuzzing, and it enables all kinds of detection mechanisms mentioned in Section II to find the bug at its first appearance. As such, it is safe to rule out such cases.

**Challenges.** Our first challenge is to link and verify duplicated reports for the same bug to establish "ground-truth". Second, to understand the reasons behind the report duplication, we need to extensively analyze the crash behaviors (for different kernel subsystems). This process, unfortunately, is difficult to fully automate and is time-consuming. Third, even for manual analysis, kernel bug analysis requires a high level of domain expertise. As such, it is difficult to simply crowdsource the analytic tasks (e.g., via Amazon Mechanical Turk).

[1]As a concrete example, "memory leak in hub_event" [14] contains multiple memory leak bugs in different kernel drivers, grouped under the same bug title. After Syzbot assigns the patch for one of the bugs, we observe that Syzbot still continues to generate this bug report [15] during fuzzing since not all bugs are fixed.

| Category | Crash Reports | Bug Titles | GT Bug Groups |
|---|---|---|---|
| Fixed Bugs | 3,243,946 | 2,526 | 1,686 |
| Duplicated | 803,206 | 1,191 | 351 |
| Sampled | 90,519 | 375 | 120 |

TABLE I: Dataset overview. "GT Bug Group" refers to the number of ground-truth bug groups.

**Approaches.** With these challenges in mind, we consider the following strategies. First, instead of analyzing the "open" bugs, we focus on the historical kernel bugs that have been patched by developers. By analyzing these patches, we can potentially, in turn, link the bug reports caused by the same bug (that were not successfully grouped during reporting). We will further analyze the grouped bug reports manually to confirm their root causes and establish the "ground-truth". Second, considering the time-consuming nature of bug analysis, we prioritize the *depth of the analysis* while maintaining a reasonable coverage for generalizable results. We randomly sample a set of bugs that have different crash behaviors (i.e., with duplicated reports) and form a focused group of domain experts to work on deduplication experiments. Based on our results, we will propose light-weighted solutions to link bug reports automatically.

### C. Kernel Bug Report Dataset

To support our analysis, we collect kernel bug reports from Syzbot dashboard. Syzbot dashboard provides more complete and update-to-date bug reports (compared with CVE [1]). More importantly, Syzbot dashboard has included the necessary information (*e.g.,* kernel version, kernel configuration, Proof-of-Concept) needed for bug reproduction (which are often missing on the CVE site [61]).

On the Syzbot dashboard, there are three main queues for kernel bugs, namely, "Open bugs", "Fixed bugs", and "Invalid bugs". As of November 2020, there are 8,071 bugs with more than 10 million crash reports in total in the three queues. As we discussed above, only the bugs in "Fixed bug" queue can be linked to establish the ground-truth for our analysis. The reason is the fixed bugs contain one additional field called "Fix commit" which shows the patch that fixes the underlying kernel bug. Reported crashes that are eventually fixed by the same patch are highly likely to share the same root cause[2] (i.e., the same bug). In the following, we leverage the patch information as a proxy to group duplicated bugs, and will further validate the ground-truth manually in Section V.

**Our Dataset.** Focusing on "Fixed bugs", we crawl the entire queue from Syzbot dashboard. Our dataset summarized in Table I covers all the corresponding crash reports from September 24, 2017, to November 11, 2020. In total, there are 3,243,946 crash reports grouped under 2,526 bug titles. As mentioned, the bug title only groups crash reports based on the crash function and crash type, which could be highly inaccurate.

[2]Using the patch to group crash reports is a reliable method except for rare cases, e.g., the incorrect patch was assigned due to human errors. We will discuss such rare cases in detail in Appendix-A.

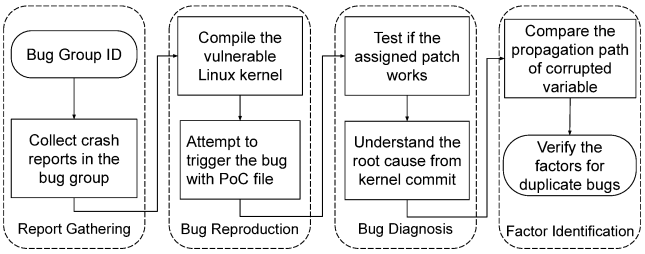


Fig. 1: Workflow to analyze the contributing factors to different crash behaviors

Then we analyze the patches of the bugs and identify 1,686 "ground-truth" bug groups. If a bug group contains more than one unique bug title, we regard the bug group as having *duplication*.

Note that, in theory, this grouping method could incorrectly group bug reports if *one patch is used to fix multiple bugs*. However, our manual analysis on these bug groups (see Section V) has confirmed that there was no such case, i.e., one patch is always used to fix one bug. This is expected since "one patch per bug" is a policy that the Linux kernel community has been enforcing before pushing a patch to the kernel [50]. As such, our ground-truth is valid.

As shown in Table I, we find in total 351 bug groups that have duplication (20.8%), involving 1,191 unique bug titles (47.1%) and 803,206 crash reports (24.8%). In other words, in these groups, multiple bug titles and their crash reports were failed to be linked together during the time of bug reporting and patch development. As a result, developers might have wasted time on analyzing the duplicated crash reports (i.e., incorrectly treating them as new bugs).

**Sampling Set.** To understand the causes of duplication, we organize domain experts to analyze the reported bugs manually. As mentioned, we prioritize the depth of analysis, and sample a subset of bug groups that has duplication. Our sampling is not completely random, but has two preferences. *First*, we prioritize analyzing bugs that cover more *diverse* crash behaviors (*i.e.,* based on the crash type and crash function in the bug title). *Second*, we prioritize analyzing bugs that have more *severe* crash behaviors. For example, Out-Of-Bound, Use-After-Free, Invalid-Free are considered to be more severe bugs [30], [45].

From the 351 bug groups that contain duplication, we sampled 120 bug groups (34.2%) which covers 375 bug titles and 90,519 crash reports. As shown in Figure 11 (in Appendix), this sampled set covers diverse crash types.

**Justifications on the Dataset Size.** We believe our dataset of 120 unique kernel bugs is reasonably large for our purposes. On one hand, the 120 bugs already cover 34.2% of the bug groups that have duplication. On the other hand, this dataset is already several times larger than existing datasets used by previous works for kernel bug analysis. For example, after a literature review, we find that most of the used kernel bug datasets contain fewer than 20 bugs [63], [58], [43], [69], [49], [78], [77], [27]. Several works studied 20-60 bugs [18], [47], [29], [28], but they are not focusing on bug reproduction and root cause analysis. Instead, most of them focus on analyzing the code changes in the patches that can be easily automated. A related work [79] collected 373 CVEs to verify the generated hot patches (for Android kernels), and only used 3 working exploits to verify the correctness of generated patches.

### D. Experiment Design

We design an experiment to examine the reasons that manifested different crash behaviors for the same bug (i.e., the cause of the duplication in bug reports). Our study was reviewed and approved by our IRB(#STUDY00008566).

**Crash Deduplication Workflow.** The workflow is shown in Figure 1. (1) Given a bug group, a security analyst collects all the crash reports and related files (including PoCs). (2) The analyst downloads and compiles the vulnerable Linux kernel based on the kernel commit and configuration file, and then runs the vulnerable Linux kernel in QEMU [20]. The analyst reproduces the bug with the provided PoC. (3) The analyst tests the patch and examines the root cause by reading the commit message and code changes. (4) The analyst identifies the corrupted variables based on the root cause and then compares the propagation path of corrupted variables from the root cause to the crashing site. The analyst then verifies the factors that contribute to the different crash behaviors.

In the above process, manual analysis is needed for (2)–(4). This is because, with the fuzzing log alone, it is difficult to identify the root causes and infer the contributing factors to the diverse crash behaviors.

**The Analyst Team.** We have formed a strong team of 5 security analysts to carry out this experiment. Each analyst has not only in-depth knowledge of the different subsystems in the Linux kernel but also has first-hand experience analyzing Linux kernel bugs, writing exploits, and developing patches. The analysts regularly publish at top security venues and have rich CTF (Capture-the-Flag) experience. When one security analyst finishes the analysis of one bug group, this analyst will present the details of this bug and explain the identified factors to the other four analysts for a peer review. After all the analysts confirm the correctness of the results, we then close the case for the given bug group.

## IV. Result: Impact of Duplication

In this section, we analyze the crash report dataset and illustrate the general impact of duplication on bug fixing time and developer overhead. Later in Section V, we will focus on the reasons behind the duplication in our expert-driven analysis.

We first analyze the crash reports and bug groups in our dataset (Table I). Recall that hundreds or thousands of crash reports are automatically grouped under a bug title (defined by crash function and crash type) by Syzbot. However, the same bug may lead to different crash functions or crash types, and thus have multiple bug titles (i.e., duplication). In the following, we characterize the duplicated bug titles and analyze the costs introduced by duplication.

**Duplication Level.** Figure 2 shows the number of distinct bug titles per bug group in our duplication set (351 bug groups in

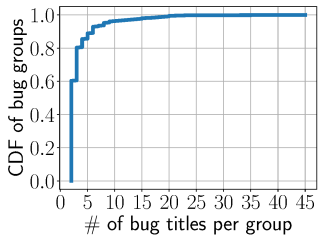


Fig. 2: Number of bug titles for each bug group with duplication.

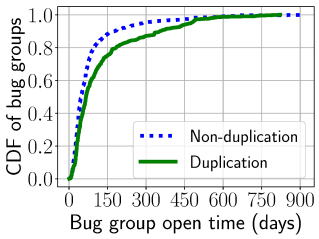


Fig. 3: Open time of each bug group: duplicated vs. non-duplicated.

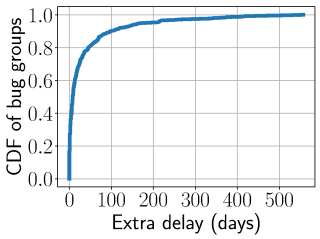


Fig. 4: Extra delay introduced by duplication.

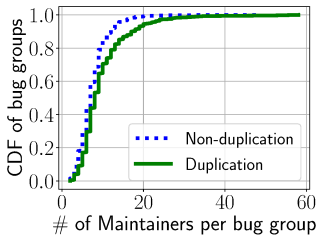


Fig. 5: Number of maintainers per bug group.

total). We find that about 60% of bug groups have only two bug titles and the duplication level is not very high. However, about 14% of the bug groups have more than five bug titles. The largest bug group has 45 distinct bug titles. This indicates the same bug can cause highly diverse crash behaviors (i.e., different crash functions or crash types).

**Extra Delay due to Duplication.** We suspect that bug groups with duplication would take a longer time to close (i.e., having a longer open time) compared to those without duplication.

Figure 3 confirms our hypothesis. More specifically, given a bug group, we define its *open time* based on two timestamps: $t_1$ (the date when the first crash was reported); and $t_2$ (the date when the last bug title in this bug group was closed). The open time of this bug group is simply $t_2 - t_1$. Figure 3 shows two main results. First, kernel bugs, even without duplication, have a longer open time. The median open time for kernel bugs without duplication is around 70 days (more than two months). Among them, about 10% take more than 150 days (5 months) to close. Some bugs take as long as 900 days (2.4 years). Second, bug groups with duplication indeed take a longer time to be closed. For example, for bug groups with duplication, about 20% are open for more than 150 days.

In Figure 4, we explicitly plot the extra delay introduced by the duplication. For bug groups with duplication, if all their bug titles were immediately linked/grouped during the time of reporting, then all of these bug titles should have been closed immediately when *the first bug title* was successfully patched and closed. However, in practice, these bug titles were not grouped automatically, and thus some bug titles remain open even though the bug is already fixed. We calculate the *extra delay* as the time gap between the first bug title closing time and the last bug title closing time in the same bug group. We find that for 80% of the bug groups, the extra delay is within 50 days. However, a small portion (5%) of the bug groups have an extra delay of more than 200 days. The extra delays could be harmful, i.e., wasting the maintainers' time if they still treated the bugs as open bugs and kept working on them.

Ideally, once a bug is fixed, the remaining bug titles in the same group can be recognized (automatically) since these bugs will no longer be triggered after the patch. In practice, there are possible reasons for their delayed closing. First, some bug titles do not have any reproducer, and thus they cannot be tested automatically to recognize that they are already fixed. Second, some bug titles need to be closed by maintainers manually with a `#syz-fix` tag, and thus experience delays.

| Category | Bug Titles | Bug Groups |
|---|---|---|
| Sampled Set | 375 | 120 |
| Reproduced | 339 | 109 |
| Final Ground-truth | 327 | 104 |

TABLE II: Summary of manually analyzed dataset.

**Less-Optimized Manual Efforts.** Another potential cost of duplication is the inefficient coordination of maintainers. For example, different groups of maintainers may independently spend time analyzing the same bug (under different bug titles) without knowing each others' work. This could lead to redundant efforts of maintainers or even incomplete patches (see cases in Section V-C). For each of the fixed bugs in Table I, we attempted to obtain the list of maintainers. While this information is not directly available at the Syzbot dashboard, most bug reports have a google group where maintainers discuss the bug. We obtain the complete maintainer lists for 82.3% of the bug groups with duplication and 80.8% of the groups without duplication. Using this data, we plot Figure 5, which confirms that bug groups with duplication often have more maintainers than non-duplicated bug groups.

For bug groups with duplication, we further analyze how much the maintainer lists under different bug titles overlap. Given a bug group, each bug has its own set of maintainers. We compute an *overlap ratio* as the size of the intersection of these maintainer sets over the size of their union. We find that 16.6% of the bug groups have an overlap ratio of 0, which means the duplicated bug reports are handled by completely different groups of maintainers. The average and median ratio is 0.56 and 0.62 respectively. If these bugs were deduplicated upon reporting, it would be possible to assign them to the same set of maintainers to avoid redundant efforts.

## V. Results: Contributing Factors

Next, to understand the reasons for the duplication (i.e., different crash behaviors for the same bug), we have performed expert-driven analysis following workflow described in Section III-D.

### A. Manual Crash Analysis

This analysis is focused on the sampled set that includes 120 bug groups and 90,519 crash reports under 375 bug titles (see Table I). Among these crash reports, 2,873 of them

| Factors | Bug Groups | Percentage |
|---|---|---|
| Different Inputs | 55 | 50.5% |
| Thread Interleaving | 18 | 16.5% |
| Memory Dynamics | 18 | 16.5% |
| Kernel Versions and Branches | 14 | 12.8% |
| Different Sanitizers | 13 | 11.9% |
| Inline Function | 8 | 7.3% |

TABLE III: The number of bug groups that are affected by each factor. One bug group could be affected by multiple factors.

```
1  int copy_verifier_state(...) {
2    struct bpf_func_state *dst;
3    dst = kzalloc(sizeof(*dst), GFP_KERNEL);
4    if (!dst)
5      return -ENOMEM;
6    return 0;
7  }
8
9  int is_state_visited(...) {
10   struct bpf_verifier_state_list *new_sl;
11   copy_verifier_state(&new_sl->state, cur);
12   // no error handler if allocation fails
13   free_verifier_state(&new_sl->state, false);
14 }
15
16 int push_stack(struct bpf_verifier_env *env,
       ↪ ...) {
17   struct bpf_verifier_stack_elem *elem;
18   err = copy_verifier_state(&elem->st, cur);
19   // no error handler if allocation fails
20   pop_stack(env, NULL, NULL);
21 }
```

TABLE IV: The code snippet showing that injecting allocation fault at different contexts could cause different bug reporting results.

have included the PoC files needed for bug reproduction. The analysis tasks took 5 security analysts about 2,400 man-hours to finish. On average, each kernel bug took 8 hours to complete all the proposed steps. Based on our experience, the most time-consuming part is to understand the root cause and identify the propagation path of corrupted variables from the root cause to the crashing site.

Out of those 120 bugs, we successfully reproduced and identified the root causes for 109 bug groups (see Table II). The other bugs were not reproducible, and thus we could not proceed with the rest of the analysis. Also, among the 109 bug groups, we find 5 bug groups for which the Syzkaller team has incorrectly assigned the patch (see more details in Appendix-A). Therefore, we use the remaining 104 bug groups as the final set to report our findings on contributing factors to duplication.

### B. Reasons for Duplication

We identify 6 factors leading to bug report duplication. We summarize these factors in Table III.

**Factor 1: Different Inputs.** The first reason for duplicated bug titles is the *input difference*. Given a kernel bug and the corresponding buggy code snippet, there could be *various execution contexts* and *many different paths* towards the buggy code. Following these paths under different contexts, the bug

```
1  // Thread A
2  void put_pi_state(... pi_state) {
3    if (atomic_dec_and_test(&pi_state->refcount
       ↪ )) {
4      kfree(pi_state);
5    }
6  }
7
8  // Thread B
9  void exit_pi_state_list(... curr) {
10   struct list_head *h = &curr->pi_state_list;
11   struct futex_pi_state pi_state =
       ↪ list_first_entry(h);
12   lock(&pi_state->pi_mutex.wait_lock);
13   get_pi_state(pi_state);
14 }
15
16 void get_pi_state(... pi_state) {
17   WARN_ON_ONCE(!atomic_inc_not_zero(&pi_state
       ↪ ->refcount));
18 }
```

TABLE V: The code snippet illustrating the difference in thread interleaving could cause duplicated bug reporting.

might demonstrate different errors and thus lead to different types of crash reports.

Take the bug `#bbeb6e43` [24] and its two reports [33], [34] as an example. The input programs extracted from the reports have the exact same sequence of system calls but one difference in the argument. Both programs could interact with kernel code and trigger the bug. However, they reach the buggy code through slightly different execution paths, which stop the kernel execution at two different kernel functions and demonstrate different types of kernel errors. As a result, the crash reports cannot be grouped together by the current deduplication method.

In addition to the *different paths*, differences in *execution contexts* could also lead to duplicated crash reports. Take, for example, the kernel code shown in Table IV. The function `copy_verifier_state` will throw an error code `ENOMEM` ↪ if the allocation function `kzalloc` fails. Since the buggy kernel does not handle this error correctly, when the function `copy_verifier_state` returns, the kernel will experience a GPF. Table IV shows that there are two sites calling the function `copy_verifier_state` (i.e., line 11 and 18). As such, when a kernel fuzzer injects the allocation error at different calling contexts (at the line 11 and 18 respectively), the kernel reports GPF in different kernel functions (i.e., `free_verifier_state` and `pop_stack`), resulting in two different crash reports.

Out of the 109 bug groups, we find 55 bug groups (50.5%) are affected by the input difference (involving 185 distinct bug titles and 88,456 crash reports). This is the most prevalent cause of the duplication problem (see Table III).

**Factor 2: Thread Interleaving.** Linux kernel is an asynchronous system supporting multi-task mechanisms. Incorrect synchronization (or missing synchronization) between kernel threads not only introduces concurrency bugs [48], but may also produce different types of errors (i.e., leading to report duplication).

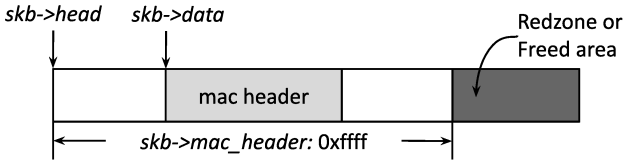


Fig. 6: An example of demonstrating an out-of-bound bug as what is or an use-after-free error.

To illustrate this problem, Table V shows an example where `pi_state` is a variable shared between thread-A and thread-B. Using the same input program but through different thread interleaving, the kernel could experience different errors. For example, thread-A invokes the function `put_pi_state` ↪ , decreasing the reference count for `pi_state` to zero (Line 3). Following the reference count being set to zero, thread-B calls the function `get_pi_state` at the Line 16. This function examines whether the reference count for `pi_state` is zero. If the condition holds, it reports a warning at Line 17. In addition to this thread-interleaving, another synchronization between both threads is to call the function `put_pi_state` in thread-A prior to thread-B executing Line 12. In this thread scheduling, thread-A de-allocates the object `struct futex_pi_state` ↪ , leaving behind a dangling pointer `pi_state`. At Line 12, thread-B dereferences the dangling pointer and a use-after-free error occurs.

Among all the kernel bugs we inspected, we discover that thread-interleaving has affected 16.5% of the bug groups that contain duplicated bug titles. It is the second most prevalent factor that contributes to bug report duplication (see Table III).

**Factor 3: Memory Dynamics.** In Linux kernel, SLAB/SLUB allocator is shared among all kernel components, responsible for dynamic memory management. Therefore, when PoC programs trigger bugs, the memory status is non-deterministic, causing the same bug to manifest different behaviors and produce duplicated reports.

Take the bug `#416dacb8` [71] as an example. Before operating an HID device, the kernel could falsely remove the device or, in other words, mistakenly deallocate the corresponding device object. When this occurs, KASAN should have reported a use-after-free error. However, due to the memory dynamics, the kernel might recycle the `slot` previously holding the device object. In this situation, rather than reporting use-after-free, the KASAN would report a slab-out-of-bound error because the accessed recycled memory region has already been set as a red zone.

Unlike the example above, which confuses KASAN to treat a freed memory region as a recycle one, another example is the bug `#96cc4b69` [44], which indicates another situation of confusing KASAN. As is illustrated in Figure 6, when transmitting a macvlan packet, the kernel fails to reset the `mac_header` ↪ , leaving `skb->mac_header` with the value of 0xffff. Later, when accessing the packet's data, the kernel uses the leftover value as an offset, referencing an out-of-bound memory region far away from the spot that the `skb->data` references. Due to memory dynamics, the out-of-bound access could touch either a red zone of an object or a freed spot. This causes the bug to demonstrate either an out-of-bound or a use-after-free error and eventually contribute to duplicated bug reports.

```
1  unsigned int tipc_poll(... sock) {
2    switch (sk->sk_state) {
3    case TIPC_OPEN:
4    // upstream a8750ddc
5    if (!grp || tipc_group_size(grp))
6    // net-next 594831a8
7    if (!grp || tipc_group_is_open(grp))
8    }
9  }
```

TABLE VI: The example indicating different kernel versions and branches affect the error manifestation.

As we show in Table III, among all the kernel bugs we inspected, we find that 16.5% of the bug groups are affected by this memory dynamics factor.

**Factor 4: Kernel Versions and Branches.** One bug can reside in several kernel versions and repository branches. When the fuzzing tools trigger the same bug in different kernel versions and repository branches, both the fuzzing programs and the crash behaviors can be different, leading to different bug reports. As shown in Table VI, bug `#60c25306` is a use-after-free bug existing in many kernel versions. In the commit `a8750ddc` of upstream repository, the error site is reported in `tipc_group_size`; However, in the commit `594831a8` of net-next repository, the error site is in `tipc_group_is_open`. Both are in the code block that gets executed when the condition `sk` ↪ `->sk_state == TIPC_OPEN` holds. The only difference is the kernel version and branch. As is depicted in Table III, this factor has caused duplicated reports for 12.8% of the bug groups we analyzed.

**Factor 5: Sanitizers.** When performing fuzz testing against the Linux kernel, security analysts may enable or disable a sanitizer or use different sanitizers. These sanitizers are designed to capture errors in different types. Therefore, the sanitizer setup variation also contributes to the difference in crashing behaviors and bug report duplication.

Take, for example, the bug `#0cbb4b4f` shown in Table VII. When forking a new process, the kernel will start an `UFFD_EVENT_FORK` event to duplicate the userfault file descriptor. If this event fails, the kernel frees the `userfaultfd_ctx` ↪ object referenced by the pointer `new` but forgets to nullify the alias pointer `vma->vm_userfaultfd_ctx.ctx` (Line 10). Later on, the function `handle_userfault` is called to handle page fault in the new process. The dangling pointer `ctx` is dereferenced to access the freed `userfaultfd_ctx` object. If `KASAN` is enabled, in Line 11, a use-after-free behavior is reported. Otherwise,the kernel continues execution until Line 13 where a BUG is reported because the reference counter of freed `userfaultfd_ctx` object is already zero.

Different from the example above, which generates duplicated reports because of switching on and off a sanitizer, another example is the bug `#250f2da4` [59], indicating another situation that causes different error behavior reporting. As is shown in Table VIII, the root cause of this bug is that if the first argument `fqname` contains only space or the second argument `n` is zero for the function `aa_splitn_fqname`, kernel will return early in Line 9 without initializing `ns_name` and `ns_len` (Line 10 and 11). The caller function `aa_fqlookupn_profile` then uses the uninitialized `ns_name` and `ns_len` for memory access,

```
1  void userfaultfd_event_wait_completion(...)
       ↪ {
2    struct userfaultfd_ctx *new;
3    new = (struct userfaultfd_ctx *)
4      ewq->msg.arg.reserved.reserved1;
5    userfaultfd_ctx_put(new);
6  }
7
8  int handle_userfault(...) {
9    struct userfaultfd_ctx *ctx =
10     vmf->vma->vm_userfaultfd_ctx.ctx;
11   BUG_ON(ctx->mm != mm);
12   if (!atomic_inc_not_zero(&ctx->refcount))
13     BUG(); // BUG if KASAN is disabled
14 }
```

TABLE VII: An example showing that switching on and off KASAN could result in different error reporting results.

causing errors caught by different sanitizers. If KMSAN [7] is enabled, using the shadow memory designed specifically for uninitalization bugs, it catches an uninit-value error immediately in Line 3 and reports the root cause of the bug. However, if KASAN [6] is enabled, the kernel will not be aware of this error but propagate the uninitialized value to its consecutive execution until the error is amplified as an out-of-bounds read.

Similar to the impact of the aforementioned two factors – memory dynamics and kernel version issues, we find that 11.9% of the bug groups (with duplication) are affected by this factor (see Table III).

**Factor 6: Inline Function.** The ways to compile the kernel code can also affect the behaviors of bugs. In particular, the compiler can make an opposite decision regarding whether to *inline a function*, which depends on the kernel configurations, the compiler (GCC or Clang) selection, and the compiler version. When this happens, we observe that one bug is triggered in different functions while in fact, they are the same program site. Our analysis shows that this factor has affected 7.3% of the bug groups we analyzed.

### C. Case Study

Through our analysis, we observed interesting cases where developers were misled to develop *incorrect* patches due to their incomplete view of the diverse bug behaviors (as the bug reports were not correctly linked).

For example, bug `#416dacb8` manifests two different behaviors: `KASAN: slab-out-of-bounds Read in ↪ hidraw_ioctl` [39] and `KMSAN: use-after-free in ↪ hidraw_ioctl` [40]. The first out-of-bound read behavior was disclosed earlier. The kernel maintainers mistakenly thought that the root cause of this bug was the incorrect output range. Therefore, they developed a patch which limited the output size of `copy_to_user`. However, this bug is actually a use-after-free bug, and it has manifested out-of-bound read due to Factor-3. More specifically, the freed `slot` is recycled to hold another object. The new object is smaller than the freed object and thus the accessed region becomes a red zone, misleading KASAN to report an out-of-bound read behavior. This incorrect patch has been committed (deployed) without being noticed. At a later time, the developers realized that the two behaviors were actually associated with the same

```
1  struct aa_profile *aa_fqlookupn_profile(
2     ..., char *fqname, size_t n) {
3     name = aa_splitn_fqname(fqname, n, ...);
4  }
5
6  char *aa_splitn_fqname(char *fqname, size_t
       ↪ n, ...) {
7     char *name = skipn_spaces(fqname, n);
8     if (!name)
9        return NULL;
10    *ns_name = NULL;
11    *ns_len = 0;
12 }
```

TABLE VIII: An example demonstrating the influence of different sanitizers upon bug reporting results.

bug [46], and the initial root cause was invalid. This example illustrates that the limited visibility to diverse bug behaviors can mislead bug patching.

## VI. Bug Report Deduplication

In this section, we provide a set of new strategies to deduplicate bug reports and prototype these strategies as a tool. Then, we evaluate this prototype tool and examine its applicability to real-world open bugs. It should be noted that our prototype is a straightforward implementation of the proposed strategies. We do not claim our prototype could pinpoint all duplicated reports. Instead, we use these straightforward implementations to answer the following three questions. ❶ What kinds of duplicated reports could be effectively and accurately pinpointed by merely following our strategies and corresponding implementations? ❷ Compared with commonly adopted stack similarity algorithms, does our prototype provide a more accurate detection? ❸ For what kinds of bug reports do we still need technical improvements for the deduplication? We hope the answers to these questions could unveil the directions for future research.

### A. Deduplication Strategies

Based on the observations from our manual inspection, we recommend five additional strategies (in addition to those that are already integrated into kernel fuzzing) that kernel developers (or Syzbot) may follow to deduplicate kernel bug reports.

First, given a bug report that looks different from previously seen bug reports in terms of the enclosed PoC and/or crashing stack trace, a kernel developer wants to determine whether the report is a duplicated copy. This developer can swap the PoC in their report and rerun it on the version of kernel specified in other bug reports. In this way, they can have different PoCs (extracted from different reports) run on the same kernel versions and thus eliminate their influence upon bug report deviation.

Second, a kernel developer can stabilize the memory layout and run the PoC programs under a relatively stable memory layout. With this, they can expect the violated memory access always lies in the same memory regions (e.g., allocated and freed memory spots) and thus minimize the influence of memory layout dynamics on bug reporting results.

Third, a kernel developer should mutate the thread interleaving and run the corresponding PoC multiple times under different thread interleaving. If the thread interleaving matters for bug report difference, this could allow a kernel developer to vary a kernel bug's panic behaviors, expose all its possible reports and thus treat these reports as duplicated bug reports accordingly.

Fourth, recall that a bug report also includes the kernel setup and configuration for the kernel fuzzing (e.g., the sanitizers they enabled). A kernel developer, therefore, should also replace the kernel fuzzing setup with the setup or configuration specified in other reports (e.g., disabling KASAN and enabling KMSAN specified in another report). By doing so, kernel developers could have the PoC run on the same setup and thus eliminate the impact of sanitizers upon bug report difference.

Last but not least, a kernel developer can also compare the PoC program and/or the stack trace with those extracted from other reports. If the bug report duplication does not result from other factors but the difference in PoC or the sites where the kernel faults are injected, the difference between PoCs is generally less significant when the bug reports reference the same kernel bug.

### *B. Deduplication Tool*

Following the strategies mentioned above, we propose a unified tool to facilitate bug report deduplication. The tool takes a pair of kernel bug reports as input, and passes them to five distinct technical components. For each component, it determines whether the pair of reports are duplicated because of the corresponding factor. We present each of the technical components below.

**Different Sanitizers.** Given a pair of bug reports, the first component examines whether the reports are duplicated because of the utilization of different sanitizers (e.g., KASAN and KMSAN). To do so, we first examine the sanitizers involved in each report. If there is only one report that indicates the utilization of a sanitizer (e.g., KASAN or KMSAN) during the kernel fuzzing, our configuration simply disables the sanitizer on the corresponding kernel specified in the report. Then, we re-run the corresponding PoC attached in that report and observe whether the sanitizer-disabled kernel still experiences unexpected termination while it takes as input the PoC program. If an unexpected kernel panic still occurs and the kernel panic is as same as the one observed on the other kernel report (i.e., the same crashing trace), we argue the report difference is contributed by the enabled sanitizer. Thus, we conclude the two reports reference the same kernel bug.

If both reports indicate the usage of a sanitizer but the sanitizers-in-use are different (*e.g.,* one uses KASAN and the other uses KMSAN), we perform the following operation. First, we take one report as our reference and configure the kernel based on the information in this reference report. Meanwhile, we disable the corresponding sanitizer of the reference report and enable the sanitizer specified in the other report. After this configuration and setup, we re-run the PoC program attached to the reference report on the reference kernel and inspect whether the kernel panic still manifests. If unexpected kernel panics still exist and the observed panic is as same as the one demonstrated in the report, we conclude both reports link to the same kernel bug. It is simply because, after having both kernels enable the same sanitizer, we eliminate the influence of "sanitizer difference". If the unexpected panic becomes identical, the two reports are just two different exhibitions of the same bug.

Note that when we switch the reference kernel's sanitizer to the one specified in the other report, it is possible that the new sanitizer is not compatible with the version of the kernel specified in the reference report. For example, KMSAN was introduced only after kernel version 4.4, and it does not support an earlier version of the reference kernel. Moreover, KMSAN is maintained in another GitHub repository. To address this issue, we take an alternative approach that disables sanitizers on both kernels and re-runs the corresponding PoC on both sanitizer-disabled kernels. If the unexpected kernel panic still occurs and the crashing reports indicate the same termination behaviors, we conclude both reports reference the same kernel bug.

**Memory Dynamics.** To offset the influence of memory layout dynamics upon bug report duplication, we not only enhance an existing memory quarantine mechanism but also replace the slab allocator with the slub allocator. Under these two changes, we can minimize the memory layout dynamics. Further, by re-running the PoC programs from different reports, if the new reporting results are the same, we can safely conclude the bug reports are duplicated.

Linux community has implemented a memory quarantine mechanism [9] in both SLAB and SLUB allocators. Its basic idea is to put freed objects in a separate queue and delay their reallocation. Using this mechanism, when a dangling pointer touches a memory region, the kernel can ensure that memory will have a lower chance of being recycled too quickly. As such, the quarantine mechanism could prevent a use-after-free bug exhibiting an out-of-bound activity and thus minimize the impact of memory layout dynamics. However, our manual analysis discovers, even if many Linux kernel distros have adopted this mechanism, memory dynamics still greatly contribute to the report duplication issue.

In this work, we take a closer look at the reason behind the quarantine's incompetency, and discover that the problems roots in the insufficient space of the separate queue. When the extracted PoC program performs race conditions, it usually allocates and deallocates many kernel objects. These freed objects could quickly push the quarantined memory space out of the queue and put them back into recycling. As such, the same bug could be reported differently, resulting in report duplication (e.g., sometimes reported as use-after-free and sometimes as out-of-bound access). To address this problem, we double the size of the quarantine queue and insert sleep operations after each race iteration. In this way, we not only leave sufficient space for freed objects but also prevent the race from exhausting the queue too quickly.

In addition to improving the memory quarantine mechanism, we also replace the slab allocator with the slub allocator. Based on our observations in Section V, we note that both the size of the redzone and that of the kernel cache contribute to the difference in reporting results. Given slab allocators, the KASAN introduces only a relatively small memory space

for the redzone. Therefore, some out-of-bound kernel bugs could jump over the redzone region, touch non-deterministic memory regions, and thus report the post-triggering activities differently. In addition, when the slab allocator is in use, out-of-bounds memory access could more easily cross the `cache` boundary, touching a `cache` totally irrelevant to the bug, and thus come up with different reporting results.

By replacing the slab allocator with the slub allocator, we can minimize the impact of redzone and the cache upon reporting difference. This is because the size of redzone for each `slot` in SLUB allocator is larger than that in SLAB allocator. Following this setup, out-of-bounds memory access is more likely to fall in the redzone. Another reason is SLAB allocator starts allocation from the end of the pages whereas SLUB allocator prioritizes the allocation of `slots` at the beginning of the pages. With this setup, the cross-boundary situation to some extent could be mitigated. Note that in order to further mitigate the cross-boundary issue, we also increase the number of pages assigned to each `cache`. In this setup, the possibility of cross-cache-boundary access could be further reduced.

**Thread Interleaving.** Our idea is to mutate the thread interleaving to determine whether a pair of bug reports is duplicated. To be specific, we first analyze the bug report and extract the crashing stack traces. For the bug report tied to OOB/UAF caught by sanitizers, *i.e.,* KASAN, we also extract stack traces that allocates and frees the object that leads to kernel panic.

With these stack traces, we could know all the kernel functions that have been invoked but not returned at the time of kernel panic. Also, we can know the kernel objects that have been allocated or freed. According to a recent study [80], the functions most close to the crashing site are more likely to be the buggy function. In addition, prior research [55] also finds the crashing stack trace sometimes indicates the execution path of one of concurrent threads. As a result, we follow the work proposed in [60], focusing our consecutive analysis on the last five functions in the stack trace.

Our method could extract three different stack traces (the crashing stack trace, the stack trace related to the object free operation, and the stack trace related to the object allocation operation). For each stack trace, we select their functions most close to the crashing site. Then, we leverage SystemTap[2] to insert time delay to the starting and returning sites of these functions. Following this setup, we further extract the PoC programs from the reports under our examination, and re-run these programs multiple times (10 times by default). If we observe a crash behavior that has already been presented in the other report, we conclude that the corresponding reports point to the same bug. It should be noted that we choose the delay time by following the heuristics applied in KCSAN (i.e., randomly selecting a value between [1,80] µs for tasks, [1,20] µs for interrupts).

**Inline Function and Kernel Versions/Branches.** We propose two straightforward approaches to determine whether report duplication is caused by the function inline mechanism or different kernel versions/branches.

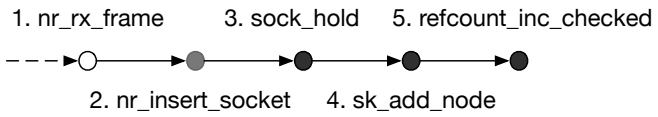


Fig. 7: The last five functions in the crashing trace from the bug group #4638faac. ○ represents function not inlined and ● are for those inlined in both bug reports. ● stands for the function inline in one report but not inline in another.

To address the function inline issue, we simply extract and compare the dead functions from the crashing stack traces. If both crashing stacks share the same dead functions in the same sequence, then the difference between crash titles is due to an inline function, and we can conclude the two reports are about the same bug. The reason behind this design is again based on results from Section II. Syzkaller uses the last crashing function to name the title of the bug report. In the process of the last crashing function extraction, Syzkaller not only skips some generic functions, but also ignores the inline functions. Take the crashing trace shown in Figure 7 as an example. The crashing stack traces show different function inline, and the reports deem crashing function as `nr_insert_socket` and `nr_rx_frame` respectively. In this work, by using the entire crashing stack trace, we can restore the footprint of the inline function, enable a more accurate crashing function comparison, and thus eliminate the influence of function inline for bug report deduplication.

To eliminate the influence of different kernel versions/branches, we first examine both reports and ensure the kernel information specified is truly different. With the confirmation, we then configure the kernels based on the specified information and rerun the corresponding PoC across kernels (i.e., running the PoC extracted from one report on the version of kernel specified in the other report). If we observe the swapped PoC demonstrates the kernel panic as same as that indicated in its original report, we safely conclude the pair of reports are about the same bug. The rationale behind this approach is that, if both reports trigger the same bug but exhibit different behaviors only because of the different versions of the underlying kernel, swapping PoC could remove the influence of this factor and thus have the corresponding kernel demonstrate the same panic behavior under two different PoCs.

**Input Difference.** Given a pair of kernel bug reports, to determine whether they are duplicated by input difference, we compare the similarity of PoC programs extracted from both reports. We deem the reports are duplicated if their PoC programs are highly similar because when the PoC programs trigger the same bug, they generally use similar types of system calls and arguments.

To compare a pair of PoC programs, we first categorize system calls by using the specification provided by Syzkaller. For example, as described in the system call templates of Syzkaller, the system call `lsetxattr` and `fsetxattr` belong to the same category, taking the responsibility of setting an extended attribute for a file. With all the system calls categorized, we extract the system call sequences from each of the PoC programs and compare the sequence as follows.

| Method | TP Rate | FP Rate |
|---|---|---|
| Baseline | 364/717 = 50.8% | 829/79,083 = 1.05% |
| Our method | 572/717 = 79.8% | 10 /79,083 = 0.01% |

TABLE IX: Detecting duplicated bug report pairs.

First, we assign a unique ID for each category of system calls. Second, for both system call sequences extracted from the PoC programs, we map the name of the system call to the corresponding ID (the system calls in the same group share the same ID) and convert the system call sequence into a list of IDs. The arguments of each system call are the elements associated with the ID. Finally, we compute the similarity of two PoC programs by measuring the similarity of the two corresponding ID lists. In this work, we perform the list similarity measurement by using a customized version of Levenshtein distance. Due to the space limit, we present the customized distance measure in Appendix-B. We deem the reports with a similarity score greater than 0.65 as the duplicated ones. Details of the threshold selection method are also presented in Appendix-B.

**Summary of Proposed Techniques.** We propose five technical components to address the 6 factors of bug duplication ("Inline Function" and "Kernel Versions/Branches" are addressed together in one component). It should be noted that four of the five components are false-positive-free (the only exception is the one for "input difference"). This is because these four components determine bug duplication by explicitly resolving the differences in the original pair of reports. The technique for "input difference" is based on PoC similarity, which could have false positives. Further discussion and evaluation are in the next Section VI-C.

### C. Ground-truth Evaluation

**Dataset and Metrics.** We construct a dataset to evaluate our method. First, for the duplicated reports, we directly use the final ground-truth set in Table II which contains 104 bug groups and 327 bug reports (i.e., bug titles). Then we introduce another 73 bug reports randomly selected from the non-duplicated bug titles (we manually confirmed the non-duplication). In total, the dataset contains 400 bug reports covering 177 unique kernel bugs (i.e., 177 bug groups).

Considering our method takes *pairs of bug reports* as inputs, we format the dataset by exhaustively pairing the reports. This generates a ground-truth dataset of 717 duplicated pairs (positive) and 79,083 non-duplicated pairs (negative).

We consider two common evaluation metrics. *TP (True Positive) rate* is the ratio of the real duplicated pairs that are successfully detected (marked out) by our method. *FP (False Positive) rate* is the ratio of the non-duplicated pairs that are incorrectly detected as duplicated pairs. Considering our dataset is skewed, we also report the raw numbers (true positives and false positives) along with these rates. In this section, our analysis is focused on *bug report pairs*. Further discussion of the *bug-group* level performance is in Appendix-C.

**Baseline Method.** It is difficult to find a direct baseline since there is little work on bug deduplication for Linux kernel.

| Factor | GT Pairs | Detected Pairs | Contributed FP |
|---|---|---|---|
| Different Sanitizers | 33 | 33 | 0 |
| Memory Dynamics | 178 | 178 | 0 |
| Inline Function | 43 | 43 | 0 |
| Thread Interleaving | 122 | 62 | 0 |
| Input Difference | 341 | 256 | 10 |
| Total | 717 | 572 | 10 |

TABLE X: Performance breakdown for each factor and its sub-method. "GT Pairs" means the number of ground-truth duplicated pairs associated with each factor. "Detected Pairs" means the number of duplicated pairs detected by the corresponding method designed for each factor.

The most relevant deduplication heuristics for *kernel space* are already used by Syzkaller. Our method is built on top of Syzkaller's deduplication results (i.e., their bug titles) to make further improvements. There are indeed deduplication methods used in *user space* for fuzzing tasks, which could be used as a baseline. Here, we choose a popular stack similarity algorithm from ClusterFuzz [19] which can work well with the stack traces of Syzkaller's kernel crash reports. We follow their deduplication policy and re-implement it to compare the stack traces of kernel crash reports. We present the detail of our re-engineering effort in Appendix-D.

**Detecting Duplicated Bug Pairs.** The detection results are summarized in Table IX. We show that our method achieves good performance and outperforms the baseline. More specifically, our method detects 572 of the 717 duplicated pairs with a true positive rate near 80% (50.8% for the baseline). At the time, we have introduced 10 false positives (0.01%) which is much lower than the baseline (829 false positives, 1.05%). The result confirms that simply comparing the stack traces is insufficient to detect duplicated bugs (baseline method). To further understand the sources of errors (especially false positives), we break down duplicated reports based on the contributing factors, as shown in Table X. For example, there are 33 duplicated pairs caused by "Different Sanitizers". We find that all 33 pairs are successfully marked by the proposed method with 0 (zero) false positive. Table X confirms that the only source of false positives (FP) is the method for "Input Differences". All other methods proposed for other factors are FP-free (by design). At the same time, we observe that the proposed methods for "Different Sanitizers", "Memory Dynamics", and "Inline Function" have a perfect true positive rate (100%). While the method for "Thread Interleaving" has missed some duplicated pairs, it does not introduce any false positives.

**Detailed Error Analysis - FN.** Our methods for "Input Differences" and "Thread Interleaving" have missed some truly duplicated pairs. We manually examine these cases and find that, under "input differences", the errors are mostly caused by the PoC programs which have a low similarity in their system call sequences and arguments. Take the duplicated reports [37] and [36] for example, one PoC invokes 16 system calls while the other involves only one system call. Further work is needed to address such cases with auxiliary information. We argue that this does not necessarily dismiss the value of the

proposed technique since it still recovers 256/341 (75%) of the duplicated pairs caused by input differences.

For "thread interleaving", we find that the missed pairs are mostly caused by three reasons. First, not all crashing stacks are directly involved in the thread synchronization. As such, inserting time delays to the functions extracted from crashing stacks does not always work. Second, for certain cases, the time delay needs to have a specific value to trigger the expected results. As such, a random time delay could be insufficient. Third, the inserted time delay could potentially influence the thread interleaving but does not guarantee the thread synchronization to occur as expected. Overall, we believe further work is needed to improve the time delay insertion (based on static and dynamic program analysis) and perform more fine-grained thread scheduling control for deduplication.

**Detailed Error Analysis - FP.** As shown in Table X, the "input difference" method is the only source of the 10 false positives. In practice, conservative developers (or Syzbot) could use all the other FP-free techniques to automatically group bug reports. For the input difference method, Syzbot can use it to make recommendations to developers on "potentially duplicated" reports or optimize the maintainer assignment process (i.e., assigning the same set of maintainers to the detected bug pairs). Because it has FP (even though small), we do not recommend using it to automatically merge/remove duplicates. We have manually examined these false positives, and determined that their PoC programs have highly similar system calls. For example, bug reports [38] and [35] are about different bugs, but the two PoCs are using the same pseudo system call `syz_usb_connect` to trigger the bug. The only difference between the PoCs is the arguments passed to the system call. Such a minor difference results in a high similarity measure and thus false positives.

As discussed in Section III-B, false positive cases have a lower impact because Syzbot can effectively handle them with continuous fuzzing. If a few corner cases are grouped incorrectly by the *input difference method*, Syzbot can ensure that these bugs will not be ignored by kernel developers. More specifically, after a patch is developed, Syzbot will continue to test the patched kernel version. If not all bugs are fixed, Syzbot will continue to file crash reports, and kernel developers will work on these reports to patch the remaining bugs.

**Applicability.** While this paper is focused on Linux kernel, the proposed techniques are applicable to other open-source kernels such as FreeBSD, NetBSD, and OpenBSD. Note that *Syzbot* already supports fuzzing these kernels, and it has already generated crash reports for them.

**Scalability.** We want to briefly discuss given the fact that deduplication requires testing a large number of bug pairs. During our analysis, we find that the runtime for analyzing each bug pair varies significantly. There are two reasons. First, if one strategy can successfully confirm duplication, we will stop testing the remaining strategies. Second, we only apply a specific deduplication strategy when their corresponding conditions are met. The applicable strategies for each pair may differ, which affects the runtime. For example, if a pair of bug reports have different sanitizers, we will apply the strategy for *different sanitizers*. In this case, the most time-consuming part is kernel recompilation. It only takes about 1 minute to run PoC testing in the QEMU VM. However, it takes an 8-core desktop 15–20 minutes (depending on the kernel configuration) to recompile the Linux kernel.

When applying these deduplication techniques in practice, we do not need to compare a new bug report (title) with all the historical reports. Instead, we only need to compare it with "Open Bugs". The open bug queue usually contains $< 1,000$ open bugs. Given the rate of bug discovery (e.g., about ten bugs per week), this overhead is manageable. Furthermore, we can run deduplication strategies in parallel (*e.g.,* compiling Linux kernel with different configurations or run the same PoC in different VMs) to improve the runtime efficiency. As a reference point, it took about two weeks to run our ground-truth evaluation, which involves testing about 80,000 bug pairs using two commodity servers. With improved parallelization and additional computation resources, we argue such overhead is acceptable to companies such as Google.

### *D. Applying Our Methods to Open Bugs*

The above ground-truth experiments confirmed the effectiveness of our methods. We next apply our methods to real-world *open bugs* to catch previously unknown duplication.

**Analyzing Open Bugs.** In January 2021, we collected all the bug titles from the "Open bugs" queue on the Syzbot dashboard. These bugs are have been reported but are not yet fixed. We obtained 652 bug reports that contained reproducers at the time of the experiment. While we can exhaustively apply our automated techniques to all the bug report pairs (212,226 pairs), to validate the correctness of results, we need to select a small subset of pairs for the time-consuming manual testing and validation.

We first extracted the crash function, crash type, the PoC program, and the stack trace from each bug report. Then, for each bug report pair, we automatically computed the similarity of their crash description, the PoC program using the method discussed in Section VI-B. We kept the bug report pair for further examination if it satisfies one of the following criteria: ❶ The two crashes occur in the same function; ❷ The two bug reports manifest the same crash type associated with the same subsystem (i.e., the crash functions are defined in the same directory); ❸ The similarity of the PoC programs in the two bug reports is greater than 0.65; ❹ The five recently called functions in the two stack traces are the same. After the filtering, we have 455 pairs left for further investigation.

**Findings.** We confirmed that our techniques were practically effective in identifying *previously unknown* bug duplication. In total, we identified 27 groups of duplicate bugs. These bug groups involved 66 bug titles and 66,594 crash reports. By examining the causes of duplication, we find that the most commonly observed factors are 'inline function" (affecting 13 bug groups) and "different inputs" (affecting 8 bug groups). For the rest of the factors, each affects $\leq 7$ bug groups.

Since we do not have the "ground truth" for open bugs, we randomly select 20 groups of duplicate bugs (flagged by our techniques) and 20 non-duplicate groups to validate the effectiveness of our methods. Through manual analysis, we do not observe any false positives. We only find 1 false negative

from the non-duplicate groups. The underlying reason is that the PoCs between the two groups are highly different, with different syscall sequences and different syscall arguments. Our tool cannot effectively group them together.

When analyzing the flagged duplicate bug groups, we have several key observations. First, for cases that are affected by "inline function", our tools can effectively identify the functions with different inline status in different reports. For example, in the bug group of `WARNING: refcount bug in qrtr_node_lookup` and `WARNING: refcount bug in qrtr_recvmsg`, our tool can identify the function `qrtr_node_lookup`, which is inline in one report and non-inline in the other report. Second, there is a case identified by "kernel versions and branches". We find two similar-looking but actually different key functions between two different kernel versions: `sctp_ulpevent_notify_peer_addr_change` and `sctp_ulpevent_nofity_peer_addr_change`. Note that one function name contains the keyword `notify`, the other function name contains the keyword `nofity` (notify vs. nofity). Third, there is a case detected by "different sanitizers", where we switch from KMSAN to KASAN in one crash report and recompile the source code. After that, we observe the same crash behaviors in another crash report. Fourth, for a case affected by "memory dynamics", our method helps to stably manifest the crash behavior in another report. Finally, our PoC similarity analysis has helped to verify cases affected by "input difference". For example, the PoCs for `general protection fault in l2cap_sock_getsockopt` and `general protection fault in sco_sock_getsockopt` have a high similarity except for the socket type in `syz_init_net_socket`.

When analyzing open bugs, we again find a similar case to that described in Section V-C. This bug group contains one bug report that shows a `UBSAN: shift-out-of-bounds in mceusb_dev_recv` behavior [42] and another report shows a `UBSAN: shift-out-of-bounds in mceusb_dev_printdata` behavior [41]. In fact, there has already been a patch developed and discussed in the community for the first behavior. This patch adds a sanity check only in the function `mceusb_dev_recv`. However, after analyzing the root cause, we concluded that the two bug reports are indeed duplicated and more importantly, the patch only mitigates the first behavior. This again confirms that a limited view of the diverse bug behaviors indeed leads to incomplete patches.

**Result Sharing and Communication.** A recent work [21] shows that knowing the multiple behaviors of the same bug can help to improve crash analysis. To this end, we have further reported the duplicated bugs we found to the Syzbot dashboard and notified the corresponding developers. We hope our findings can help with ongoing bug analysis.

At the same time, we have communicated our results and findings with the Syzkaller and Syzbot team. The initial response from the team was positive, echoing the importance of finding the duplicated bugs given the backlog of open bugs they are currently accumulating. We are in the process of introducing our tool, and exploring the opportunity to integrate it into Syzbot to improve the current deduplication system.

## VII. Related Work

In this work, we perform a large-scale measurement to study the bug report duplication phenomenon on Syzbot, with the purpose of improving root cause diagnosis and facilitating bug patching. As such, we consider the works in the following directions as related.

**Bug Report Deduplication.** Prior works have explored different methods to deduplicate bugs discovered by automatic tools. One method is to leverage stack trace in the bug report to determine whether two reports are referring to the same bug. Fuzzing tools, including SmartFuzz [60], VUzzer [64], FuzzSim [76], CERT BFF [25], and Syzkaller [73], generate a stack hash using the function name, line number, and file name on the call stack for deduplication. Other works (*e.g.,* [65], [75]) calculate stack edit distance and TF-IDF for deduplication. ReBucket [32] measures the similarities of call stacking using an Dependent Model (PDM) algorithm. Another method considers the basic block transition in the execution path, which is widely employed by AFL-based fuzzers (*e.g.,* [23], [22], [67], [21], [57], [62], [53], [56]) Unfortunately, a study by Klees *et al.* [52] shows that this approach could yield excessive false positives and false negatives. Finally, Tonder *et al.* [72] propose a method that groups the bugs with similar fixes. Despite these efforts, none of the previous works are focused on reasoning the factors that have caused different crash behaviors and designing solutions to handle these factors (the main focuses of our work).

**Empirical Studies of Kernel Bugs.** Researchers have performed empirical studies of kernel bugs, with different purposes compared to our paper. For example, Abal *et al.* [18] have studied 42 bugs in the Linux kernel. They observed that variability bugs do not exclusively belong to any particular bug types, error-prone features, or source code locations, while the variability property has greatly increased the complexity of bugs in the Linux kernel. PDiff [47] performed a comprehensive study to understand the patch presence testing problem. They identified two essential challenges in the testing: third-party code customization and diversities in the building configuration. Xu *et al.* [79] empirically studied real-world Android kernel vulnerability patches. They found that the code changes of security patches are generally small compared to non-security patches and large security patches usually contain several small individual patches. Li and Paxson [54] conducted an empirical study of security patches to understand their development life cycles. They show that security patches are more localized (than other non-security patches) but usually suffer from a long delay. Mu *et al.* [61] analyzed crowd-reported vulnerability reports to assess their reproducibility. Our work is the first to study the factors causing the report duplication problems for kernel bugs. In addition to the empirical study, we also provide suggestions to deduplicate bug reports to facilitate root cause diagnosis and bug patching.

## VIII. Conclusion and Future Work

We discover the duplicated kernel bug reports are prevalent and could potentially cause the delay of kernel bug remedy. Through intensive manual efforts, we analyze the root cause behind the duplicated reports and summarize six key factors directly contributing to report duplication. Under the guidance of

our discovery, we prototype a series of deduplication strategies. We conclude the newly proposed deduplication strategies could group a majority of duplicated kernel bug reports correctly and even facilitate correct kernel patch development. As is described in Section VI-C, our proposed deduplication method is merely the initial exploration. Our future research will focus on exploring more advanced technical approaches to better cluster duplicated reports or in other words further reduce the errors introducing in the report grouping. Besides, we will study how to use the grouped reports to better diagnose the root cause of corresponding kernel bugs and thus further benefit bug remedy. Finally, we will also study the bug duplication problem for other OSes (e.g., Windows and XNU).

## ACKNOWLEDGMENTS

We would like to thank our shepherd Antonio Bianchi and the anonymous reviewers for their helpful feedback. This project was supported in part by IBM Ph.D. Fellowship (2020-2022), NSF grants 1955719, 1954466, and 2055233, and by National Natural Science Foundation of China under Grant 62102154. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of any funding agencies.

## REFERENCES

[1] "Common Vulnerabilities and Exposures (CVE)," 1999, https://cve.mitre.org/.

[2] "Systemtap," 2005, https://sourceware.org/systemtap/.

[3] "Fault injection capabilities infrastructure," 2006, https://www.kernel.org/doc/html/latest/fault-injection/fault-injection.html.

[4] "Trinity: Linux system call fuzzer." 2006, https://github.com/kernelslacker/trinity.

[5] "Kernel memory leak detector," 2011, https://www.kernel.org/doc/html/latest/dev-tools/kmemleak.html.

[6] "Kernel address sanitizer," 2014, https://www.kernel.org/doc/html/latest/dev-tools/kasan.html.

[7] "Kernel memory sanitizer," 2015, https://github.com/google/kmsan.

[8] "Dirty COW (CVE-2016-5195)," 2016, https://dirtycow.ninja/.

[9] "mm: kasan: Initial memory quarantine implementation," 2016, https://lore.kernel.org/patchwork/patch/658546/.

[10] "The object-lifetime debugging infrastructure," 2016, https://www.kernel.org/doc/html/latest/core-api/debug-objects.html.

[11] "WannaCry Ransomware Attack," 2017, https://en.wikipedia.org/wiki/WannaCry_ransomware_attack.

[12] "The kernel concurrency sanitizer (kcsan)," 2019, https://www.kernel.org/doc/html/latest/dev-tools/kcsan.html.

[13] "Google researcher found bleedingtooth flaws in linux bluetooth," 2020, https://securityaffairs.co/wordpress/109500/hacking/bluetooth-bleedingtooth-vulnerabilities.html.

[14] "memory leak in hub_event," 2021, https://syzkaller.appspot.com/bug?id=66fe8eb71f455a245547576eb8d36fec957d2424.

[15] "memory leak in hub_event (2)," 2021, https://syzkaller.appspot.com/bug?id=91adc3631f0fd2b51356949a8cc20b994856d6ee.

[16] "New bug left in the closed and duplicated bug reports," 2021, https://groups.google.com/g/syzkaller/c/_roBlPUWp04/m/Na_lAPP7AwAJ.

[17] "The undefined behavior sanitizer - ubsan," 2021, https://www.kernel.org/doc/html/latest/dev-tools/ubsan.html.

[18] I. Abal, C. Brabrand, and A. Wasowski, "42 variability bugs in the linux kernel: A qualitative analysis," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14, 2014.

[19] A. Arya and C. Neckar, "Fuzzing for security," 2014, https://blog.chromium.org/2012/04/fuzzing-for-security.html.

[20] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. USENIX ATC '05, 2005.

[21] T. Blazytko, M. Schlögel, C. Aschermann, A. Abbasi, J. Frank, S. Wörner, and T. Holz, "AURORA: Statistical crash analysis for automated root cause explanation," in *Proceedings of the 29th USENIX Security Symposium*, ser. USENIX SEC '20, 2020.

[22] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, 2017.

[23] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, 2016.

[24] D. Borkmann, "bpf, array: fix overflow in max_entries and undefined behavior in index_mask," 2018, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=bbeb6e4323dad9b5e0ee9f60c223dd532e2403b1.

[25] CERT, "BFF - basic fuzzing framework," 2016, https://vuls.cert.org/confluence/display/tools/CERT+BFF+-+Basic+Fuzzing+Framework.

[26] 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 *Proceedings of the Second Asia-Pacific Workshop on Systems*, ser. APSys '11, 2011.

[27] W. Chen, X. Zou, G. Li, and Z. Qian, "KOOBE: Towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities," in *Proceedings of the 29th USENIX Security Symposium*, ser. USENIX SEC '20, 2020.

[28] Y. Chen, Z. Lin, and X. Xing, "A systematic study of elastic objects in kernel exploitation," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20, 2020.

[29] Y. Chen and X. Xing, "SLAKE: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19, 2019.

[30] ClusterFuzz, "Crash type in clusterfuzz," 2019, https://google.github.io/clusterfuzz/reference/glossary/#crash-type.

[31] CVE Details, "Linux Kernel," 2021, https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33.

[32] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel, "ReBucket: A method for clustering duplicate crash reports based on call stack similarity," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12, 2012.

[33] S. dashboard, "KASAN: use-after-free read in map_lookup_elem," 2018, https://syzkaller.appspot.com/bug?id=fcd138b2ad0188e5eed65d3351ab983f4bc1c3b6.

[34] ——, "KASAN: use-after-free write in array_map_update_elem," 2018, https://syzkaller.appspot.com/bug?id=19cf067af88c8f151825cefa7b199e7a8b7dc861.

[35] ——, "general protection fault in flexcop_usb_probe," 2019, https://syzkaller.appspot.com/bug?id=c0203bd72037d07493f4b7562411e4f5f4553a8f.

[36] ——, "general protection fault in ip6_sublist_rcv," 2019, https://syzkaller.appspot.com/bug?id=cdfd70388a396eb80fe860a5251c2e1232b1e407.

[37] ——, "general protection fault in ip_sublist_rcv," 2019, https://syzkaller.appspot.com/bug?id=f07ddeedf116ac76456528915123a4d8d4d709bd.

[38] ——, "KASAN: invalid-free in disconnect_rio (2)," 2019, https://syzkaller.appspot.com/bug?id=35acc31cfe715b32b649129f286c960dba2d49c5.

[39] ——, "KASAN: slab-out-of-bounds read in hidraw_ioctl," 2019, https://syzkaller.appspot.com/bug?id=0141bd6b37153edec9c4ffa0f0e990c7228897f9.

[40] ——, "KMSAN: use-after-free in hidraw_ioctl," 2019, https://syzkaller.appspot.com/bug?id=572cfde68d72a02f41e60c6a6c060157c6e6a750.

[41] ——, "UBSAN: shift-out-of-bounds in mceusb_dev_printdata," 2020, https://syzkaller.appspot.com/bug?id=df1efbbf75149f5853ecff1938ffd3134f269119.

[42] ——, "UBSAN: shift-out-of-bounds in mceusb_dev_recv," 2020, https://syzkaller.appspot.com/bug?id=50d4123e6132c9563297ecad0479eaad7480c172.

[43] L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "PT-Rand: Practical mitigation of data-only attacks against page tables." in *Proceedings of The Network and Distributed System Security Symposium*, ser. NDSS '17, 2017.

[44] E. Dumazet, "macvlan: do not assume mac_header is set in macvlan_broadcast()," 2018, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=96cc4b69581db68efc9749ef32e9cf8e0160c509.

[45] J. Foote, "The exploitable gdb plugin," 2015, https://github.com/jfoote/exploitable.

[46] googlegroup, "KASAN: slab-out-of-bounds read in hidraw_ioctl," 2019, https://groups.google.com/g/syzkaller-bugs/c/O90aBzvp_uE/m/-Q0j14aUBwAJ.

[47] Z. Jiang, Y. Zhang, J. Xu, Q. Wen, Z. Wang, X. Zhang, X. Xing, M. Yang, and Z. Yang, "PDiff: Semantic-based patch presence testing for downstream kernels," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20, 2020.

[48] G. Jin, W. Zhang, and D. Deng, "Automated concurrency-bug fixing," in *10th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '12, 2012.

[49] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "Ret2dir: Rethinking kernel isolation," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. USENIX SEC '14, 2014.

[50] L. Kernel, "Submitting patches: the essential guide to getting your code into the kernel," 2021, https://www.kernel.org/doc/html/v4.10/process/submitting-patches.html.

[51] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "HFL: Hybrid fuzzing on the linux kernel," in *Proceedings of The Network and Distributed System Security Symposium*, ser. NDSS '20, 2020.

[52] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, 2018.

[53] C. Lemieux and K. Sen, "FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '18, 2018.

[54] F. Li and V. Paxson, "A large-scale empirical study of security patches," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS'17, 2017.

[55] G. Li, S. Lu, M. Musuvathi, S. Nath, and R. Padhye, "Efficient scalable thread-safety-violation detection: Finding thousands of concurrency bugs during testing," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19, 2019.

[56] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: program-state based binary fuzzing," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. FSE '17, 2017.

[57] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "MOPT: Optimized mutation scheduling for fuzzers," in *Proceedings of the 28th USENIX Conference on Security Symposium*, ser. USENIX SEC '19, 2019.

[58] A. Milburn, H. Bos, and C. Giuffrida, "SafeInit: Comprehensive and Practical Mitigation of Uninitialized Read Vulnerabilities," in *Proceedings of The Network and Distributed System Security Symposium*, ser. NDSS '17, 2017.

[59] Z. Mithra, "apparmor: Fix uninitialized value in aa_split_fqname," 2018, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=250f2da49cb8e582215a65c03f50e8ddf5cd119c.

[60] D. Molnar, X. C. Li, and D. A. Wagner, "Dynamic test generation to find integer bugs in x86 binary linux programs," in *Proceedings of the 18th USENIX Conference on Security Symposium*, ser. USENIX SEC '09, 2009.

[61] D. Mu, A. Cuevas, L. Yang, H. Hu, X. Xing, B. Mao, and G. Wang, "Understanding the reproducibility of crowd-reported security vulnerabilities," in *Proceedings of the 27th USENIX Security Symposium*, ser. USENIX SEC '18, 2018.

[62] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-Fuzz: fuzzing by program transformation," in *Proceedings of the 2018 IEEE Symposium on Security and Privacy*, ser. SP '18, 2018.

[63] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis, "xMP: Selective memory protection for kernel and user space," in *Proceedings of the 2020 IEEE Symposium on Security and Privacy*, ser. SP '20, 2020.

[64] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware evolutionary fuzzing." in *Proceedings of The Network and Distributed System Security Symposium*, ser. NDSS '17, 2017.

[65] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07, 2007.

[66] S. Santoyo, "A brief overview of outlier detection techniques," 2017, https://towardsdatascience.com/a-brief-overview-of-outlier-detection-techniques-1e0b2c19e561.

[67] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-assisted feedback fuzzing for OS kernels," in *Proceedings of the 26rd USENIX Conference on Security Symposium*, ser. USENIX SEC '17, 2017.

[68] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC '12, 2012.

[69] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee, "Enforcing kernel security invariants with data flow integrity." in *Proceedings of The Network and Distributed System Security Symposium*, ser. NDSS '16, 2016.

[70] E. Stepanov and K. Serebryany, "MemorySanitizer: Fast detector of uninitialized memory use in c++," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '15, 2015.

[71] A. Stern, "HID: hidraw: Fix invalid read in hidraw_ioctl," 2019, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=416dacb819f59180e4d86a5550052033ebb6d72c.

[72] R. van Tonder, J. Kotheimer, and C. Le Goues, "Semantic crash bucketing," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '18, 2018.

[73] D. Vyukov, "Syzkaller," 2016, https://github.com/google/syzkaller.

[74] ——, "Syzbot Dashboard," 2017, https://github.com/google/syzkaller/blob/master/docs/syzbot.md.

[75] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE'08, 2008.

[76] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, "Scheduling black-box mutational fuzzing," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '13, 2013.

[77] W. Wu, Y. Chen, X. Xing, and W. Zou, "KEPLER: Facilitating control-flow hijacking primitive evaluation for linux kernel vulnerabilities," in *Proceedings of the 28th USENIX Conference on Security Symposium*, ser. USENIX SEC '19, 2019.

[78] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou, "FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities," in *Proceedings of the 27th USENIX Conference on Security Symposium*, ser. USENIX SEC '18, 2018.

[79] Z. Xu, Y. Zhang, L. Zheng, L. Xia, C. Bao, Z. Wang, and Y. Liu, "Automatic hot patch generation for android kernels," in *Proceedings of the 29th USENIX Security Symposium*, ser. USENIX SEC '20, 2020.

[80] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps, "Conseq: Detecting concurrency bugs through sequential errors," *SIGPLAN Not.*, vol. 46, no. 3, p. 251–264, Mar. 2011.

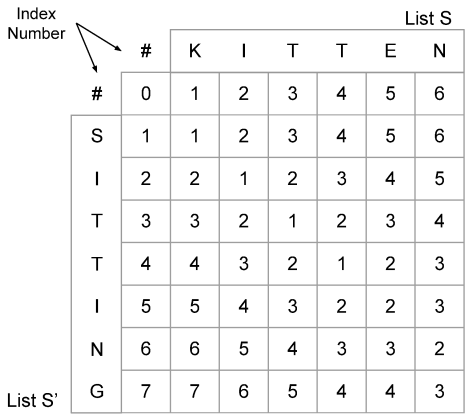


Fig. 8: A sample matrix obtained from Levenshtein distance computation.

## APPENDIX

### APPENDIX-A: CORNER CASES OF BUG GROUPS

Our "ground-truth" bug groups are determined based on the patches of those fixed bugs. This methodology is reliable, except for a handful of corner cases. We discover such corner cases during our manual analysis, which turns out to be caused by human errors of syzbot maintainers. More specifically, thorough our analysis (described in Section III-D), we find that 5 bug groups (4.59%) contain the wrong patches for 10 of their bugs. Note that this is not a contributing factor to bug duplication (but may have some relationship with duplication). More specifically, after a bugfix is developed, certain syzbot maintainers may manually assign the bugfix to one or more bug titles since they expect there are duplications. We find that sometimes the patch assignment is wrong. The mistake is mostly caused by human errors during their manual bug analysis (e.g., recognizing the wrong bug in the same parts of the code). To mitigate this issue, we recommend simply re-running the PoC to verify if the patch is indeed fixing for the assigned bug. We have reported this problem to the corresponding syzbot maintainers.

### APPENDIX-B: SIMILARITY MEASURE

**Levenshtein distance.** The Levenshtein distance equation takes as input two lists, outputting a matrix by using the equation below

$$Lev_{S,S'}(i,j) = \begin{cases} max(i,j), & \text{if } min(i,j) = 0. \\ min \begin{cases} Lev_{S,S'}(i-1,j)+1, \\ Lev_{S,S'}(i,j-1)+1, \\ Lev_{S,S'}(i-1,j-1)+cost(S_i,S'_j), \end{cases} & \text{Otherwise.} \end{cases} \quad (1)$$

As is depicted in Figure 8, the Levenshtein distance computation can identify the element aligned across the lists. In our application, they indicate the syscalls aligned across the pair of PoC. In addition, the value in the lower-right corner of the matrix also indicates the minimal edit needed for flipping one list into the other. It is in a range from 0 to $\infty$. In our work, we normalize this value into the range between 0 and 1 by using the equation below.

$$Similarity(S,S') = 1 - \frac{Lev(S,S')}{max(len(S),len(S'))} \quad (2)$$

We use this normalized value as the similarity between the two lists extracted from the PoCs.

**Customized Levenshtein distance.** Different from the original Levenshtein distance equation, our customized version replaces the cost function of the original version from the form $cost(a_i,b_j) = 1_{a_i!=b_j}; 0_{a_i==b_j}$ to the form below

$$cost(S_i,S_j) = \begin{cases} 1 & ID(S_i) \neq ID(S_j), \\ N/M & ID(S_i) = ID(S_j) \end{cases} \quad (3)$$

Here, the new cost function is designed to augment the Levenshtein distance equation with the ability to capture the argument deviation of system calls in the same family. As we can observe from Figure 9, if the syscall aligned in both PoC have the exact same arguments, we assign 0 to the cost function. Otherwise, we deem $\frac{N}{M}$ as the value of the cost function. Here, $N$ represents the total number of arguments sharing no value whereas $M$ indicates the total number of arguments in the syscall's argument list.

**Threshold selection.** To determine what value of the similarity measure should be taken as a signal for duplicated report identification, we use the customized Levenshtein distance to compute the pairwise similarity for non-duplicated kernel reports gathered from Syzbot. We note the average pairwise similarity measure for these non-duplicated reports is 0.16, and their standard deviation is 0.14. Based on the univariant outlier detection technique [66], we use the mean value plus the 3.5x standard deviation as our cutoff threshold value. We deem a pair of reports have a highly similar PoC program if their similarity measure is above this threshold.

### APPENDIX-C: DETECTION PERFORMANCE OF BUG GROUPS

The evaluation of the proposed techniques (Section VI-C) has been focused on the detection of duplicated *bug pairs*. Here, we briefly discuss the detection performance at the *bug group* level. More specifically, we grouped the detected duplicated pairs into bug groups and compared them with the ground truth. For our proposed method, we find that the detection has 0 (zero) error for 70 bug groups (out of 104 ground-truth groups). This means our techniques have perfectly discovered these bug groups. In comparison, the baseline method only discovered 44 bug groups without any errors. At the same time, our method only produces 5 bug groups that contain false positives (10 FP pairs in total). In comparison, the baseline method has produced 70 bug groups with false positives in the group. The rest of the bug groups do not have false positives, but may have missed duplicated bug titles. Overall, the results confirm that our method significantly outperforms the existing baseline method. Further discussion on how false positives should be handled is in Section VI-C.

### APPENDIX-D: IMPLEMENTATION OF THE BASELINE METHOD

As is mentioned in Section VI, when grouping duplicated bug reports gathered from Syzbot, we re-implemented the

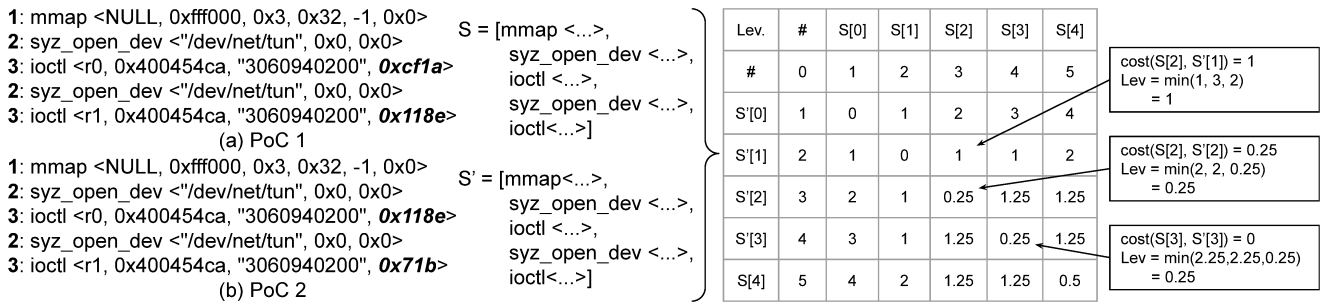


Fig. 9: The demonstration of our customized Levenshtein distance computation.

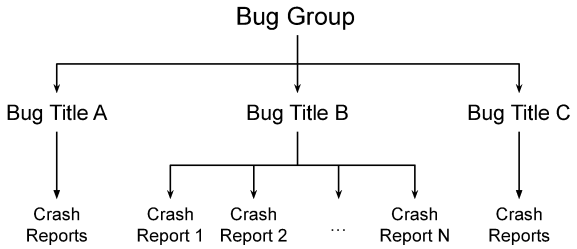


Fig. 10: Relationships between bug groups, bug titles, and crash reports.

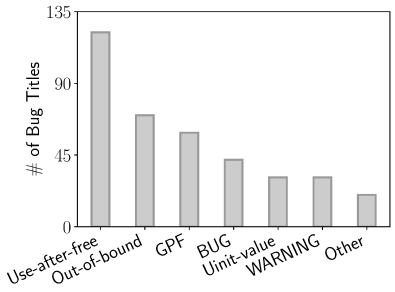


Fig. 11: Number of bug titles under each distinct crash type in our *sampled* dataset.

method used by ClusterFuzz. In this work, we used this method as our comparison baseline to demonstrate the effectiveness of our proposed deduplication method. Here, we detail our re-implementation effort. Given a pair of Linux kernel bug reports, we first extracted the crashing stack trace from each report. In this step, we customized ClusterFuzz's deduplication method by reconsidering the format of Linux kernel stack trace and kernel sanitizers. Meanwhile, we excluded certain functions at the kernel stack trace (*e.g.,* `dump_stack`) to prevent its adverse effect upon the similarity comparison of the stack traces. Second, we then compared the stack traces of the two bug reports one by one, starting from the first function invoked. When performing a comparison against two functions, we utilized the Levenshtein distance to calculate the similarity of their function names. In our implementation, we added up all the similarity scores, and divided the sum by the total number of functions under comparison. This produced a normalized similarity score of the stack traces from the two bug reports. If the similarity of the stack traces is more than a threshold (*i.e.*, 0.8 used in ClusterFuzz), we treated them as duplicate bug reports. Otherwise, we deemed them as non-duplicated bug reports.