Towards Unveiling Exploitation Potential with Multiple Error Behaviors for Kernel Bugs

Ziqin Liu, *Member, IEEE,* Zhenpeng Lin, *Member, IEEE,* Yueqi Chen, *Member, IEEE,* Yuhang Wu, *Member, IEEE,* Yalong Zou, *Member, IEEE,* Dongliang Mu, *Member, IEEE** and Xinyu Xing, *Member, IEEE*

Abstract—Nowadays, fuzz testing has significantly expedited the vulnerability discovery of Linux kernel. Security analysts use the manifested error behaviors to infer the exploitability of one bug and thus prioritize the patch development. However, only using an error behavior in the report, security analysts might underestimate the exploitability of the kernel bug because it could manifest various error behaviors indicating different exploitation potentials. In this work, we conduct an empirical study on multiple error behaviors of kernel bugs to understand 1) the prevalence of multiple error behaviors and the possible impact towards the exploitation potential; 2) the factors that manifest multiple error behaviors with different exploitation potential. We collected *all the fixed kernel bugs* on Syzbot from 2017 to 2022. We observed that multiple error behaviors manifested by kernel bugs are prevalent in the real world. Then we analyze a sample dataset (162 unique bugs) and identified 6 key contributing factors to multiple error behaviors. Finally, based on the empirical findings, we propose an object-driven fuzzing technique to explore all possible error behaviors of kernel bugs. To evaluate the utility of our proposed technique, we implement GREBE and apply it to 60 real-world kernel bugs. On average, GREBE could manifest 2+ additional error behaviors for all kernel bugs. For 26 kernel bugs, GREBE discovers higher exploitation potential. We report to kernel vendors some bugs – the exploitability of which was wrongly assessed and the corresponding patch has not yet been carefully applied – resulting in their rapid patch adoption.

Index Terms—Linux Kernel, Error Behaviors, Empirical Study, Object-driven Fuzzing, Exploitability

1 Introduction

Today, Linux powers a wide variety of computing systems. To improve its security, researchers and analysts introduced automated kernel fuzzing techniques and various debugging features/sanitizers. With their facilitation, it becomes easier for security researchers and kernel developers to pinpoint a bug in the Linux kernel. However, it is still challenging to determine whether bug conditions are sufficient to represent a security vulnerability. For example, a bug that demonstrates out-of-bound error behaviors usually implies a higher chance to exploit than those that exhibit null pointer dereference error behaviors. As such, previous researches [2], [3], [4] indicate that the manifested error behaviors of bugs play a critical role in prioritizing exploit and patch development efforts.

In practice, when existing fuzzing tools identify a kernel bug, the error behavior in the report may be one of its possible manifested error behaviors. For example, as we will elaborate in Section 2, by following different paths, a kernel bug can exhibit not only a less-likely-to-exploit GPF (General Protection Fault) error behavior but also a highly-likely-to-exploit UAF (Use-After-Free) error behavior. As such, with the only manifested error behaviors in the bug report, security analysts may underestimate the exploitability of the underlying bug.

In order to address the above problem (*i. e.,* **Exploitability Underestimation Problem**), one instinctive reaction is to take a bug report as input, analyze the root cause of that kernel bug, and infer all possible consequences that the bug could potentially bring about (*e.g.,* Out-Of-Bound Access, Null Pointer Dereference, Memory Leak, and etc.). However, root cause diagnosis is typically considered a time-consuming and labor-intensive task. As a result, a more realistic strategy for tackling this problem is to explore more possible error behaviors of a given kernel bug without performing root cause analysis. Then, from the newly unveiled error behaviors, security analysts could better estimate its possible exploitability in a more accurate fashion.

Goal and Approach. In this paper, we conduct an empirical study on multiple error behaviors manifested by the same kernel bug. Our goal is to understand 1) the prevalence of multiple error behaviors manifested by kernel bugs and the possible impact of multiple error behaviors towards the exploitation potential; 2) the factors that manifest multiple error behaviors implying different exploitation potential. Based on the insights obtained from our empirical study, we further design a new kernel fuzzing mechanism to explore a new manifestation of error behaviors from the same kernel bug, which helps security analysts assess the kernel bugs more accurately. Note that, we define *a new manifestation of*

**Corresponding author*

Z. Liu, Y. Zou and D. Mu are with Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, 430074, China (email: {ziqinl, yalongz, dzm91}@hust.edu.cn).

Z. Lin, Y. Chen, Y. Wu are with College of Information Sciences and Technology, the Pennsylvania State University, Pennsylvania 16802, USA (e-mail: {zplin, ycx431, yuhang}@psu.edu).

X. Xing, are with Department of Computer Sciences, Northwestern University, Evanston, IL 60208, USA (e-mail: xinyu.xing@northwestern.edu).

An earlier version appeared at the 43rd IEEE Symposium on Security and Privacy (SP '22) [1].

the same bug as a new error behavior manifested by the same kernel bug, different from the error behavior in the original bug report.

The first key challenge in our approach is to obtain the "ground-truth" for multiple error behaviors manifested by kernel bugs. While it 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 by the same patch, these bug reports are highly likely to be manifested by the same bug. After this initial grouping, we then manually analyze these bug reports and the underlying bugs to identify contributing factors behind the manifestation of multiple error behaviors. In this work, we have collected *all* the fixed kernel bugs reported on Syzbot from September 2017 to January 2022. Based on their patches, we separate these 3,352 bug reports into 2,296 unique groups. Out of the 3,352 error reports, 1,496 (44.6%) belong to multiple error behaviors manifested by the same kernel bugs.

To achieve our goal, we take action in two folds. First, we analyze the "ground-truth" dataset and get some observations. 19.2% (*i. e.,* 440) kernel bugs have multiple error behaviors. A phenomenon has been discovered that the more error behaviors one kernel bug has, the higher possibility of likely-to-exploit error behaviors the bugs can manifest. Second, we manually analyze these kernel bugs to identify the causes behind multiple error behaviors. We choose 162 kernel bugs with multiple error behaviors (162/440 = 36.8%), which is reasonably large for our study. Under the guidance of an approved IRB, 5 security experts follow a rigorous analysis procedure to reproduce the kernel bugs, and analyze the code changes and the developer's notes to figure out the causes behind multiple error behaviors. This highly time-consuming experiment took *nearly 3000 man-hours* to complete. Through intensive manual efforts, we collectively identify 6 key contributing factors to multiple error behaviors. And in our sample dataset, *input difference*, *memory dynamics*, *thread interleaving* can manifest multiple error behaviors leading to different exploitation potential.

Another key challenge is in the new kernel fuzzing mechanism. Existing kernel fuzzing methods are mainly designed to maximize the code coverage (*e.g.,* Syzkaller [5], KAFL [6]). They suffer from inefficiency and ineffectiveness issues since their design is unsuitable to find various execution paths and contexts relevant to the same buggy code fragment. To this end, we propose a customized kernel fuzzing mechanism that concentrates its fuzzing energy on the buggy code areas, and meanwhile, diversifies the kernel execution paths and contexts towards the target buggy code fragment.

Technically speaking, our proposed kernel fuzzing mechanism could be viewed as a directed fuzzing approach. It first takes an error report as input and extracts the kernel structures/objects relevant to the reported kernel error. Then, the fuzzing method performs fuzzing testing and utilizes the hits to the identified kernel objects as feedback to the fuzzer. Since the identified kernel structures/objects are essential to the success of triggering the reported bug, using them to guide fuzzing could narrow the scope of the kernel fuzzer, making the fuzzer focus mostly on the paths and contexts pertaining to the reported bug. In this work, we implement this approach as an object-driven kernel fuzzing tool and name it after `GREBE`, standing for "fuzzin**G** fo**R** multipl**E** **B**ehavior **E**xploration".

Using our tool to explore new error behaviors for 60 kernel error reports, we show that `GREBE` could demonstrate 2+ different error behaviors on average for each bug report. For many kernel bugs (26 out of 60), we also observe that their newly identified error behaviors usually demonstrate a higher exploitation potential than those in the original bug reports. More surprisingly, through the paths and contexts that we newly identified, we also discover 6 kernel bugs with seemly unexploitable memory corruption ability (*e.g.,* GPF, WARN) could be turned into ones with powerful memory corruption ability that can be utilized to perform an arbitrary execution. All these bugs have not demonstrated any exploitability before. We report this finding to some kernel vendors – that have not yet applied the ready-to-use patches in their products – resulting in their immediate patch adoption.

To the best of our knowledge, this is the first work that studies and exposes a bug's multiple error behaviors for exploitability exploration. The exhibition of multiple error behaviors could potentially expedite the remedy and elimination of highly exploitable bugs from the kernel. Besides, it could also augment security analysts to turn an unexploitable primitive into an exploitable one. Last but not least, demonstrating a bug with multiple error behaviors could also potentially benefit the bug's root cause diagnosis [7].

In summary, this paper makes the following contributions:

- We empirically study the phenomenon (*i. e.,* multiple error behaviors manifested by the same bug) to explore the prevalence of this phenomenon and its impact on bugs' exploitation potential.
- We organize security experts to perform an in-depth analysis of kernel bug dataset. We identify several factors that manifest multiple error behaviors with different exploitation potentials.
- Based on our empirical findings, we design a novel object-oriented kernel fuzzing approach to explore a bug's multiple error behaviors.
- We implement `GREBE`, and demonstrate its utility in finding multiple error behaviors for real-world kernel bugs. Given a kernel bug demonstrating only a low possibility of exploitation, our proposed method could find its other error behaviors indicating much stronger exploitability.

2 Data Collection Methodology

In this section, we first present our methodology and then describe how to collect our dataset to measure the prevalence of manifested error behaviors. Finally, we depict the experiment workflow to analyze the contributing factors causing multiple error behaviors.

2.1 Methodology

Definition of Unique Bug and Manifested Error Behaviors. A bug is uniquely defined by its root cause, *i. e.,* if different error reports share the same root cause, then we define them as *multiple error behaviors manifested by the same bug*.

Challenges. Our *first* challenge is to link and verify error reports manifested by the same bug to establish "ground-truth". The *second* challenge is that we need to extensively analyze kernel error behaviors to understand the root cause of kernel bugs and underlying reasons for manifested error behaviors. Unfortunately, this analysis is hard to be automated and time-consuming. The *third* challenge is that bug analysis requires a high level of domain knowledge, even with manual analysis.

Approaches. With the above three challenges in mind, we apply the following strategies in our study. First, instead of manually analyzing "open" kernel bugs, we focus on the historical kernel bugs that are already patched by kernel developers. By analyzing these patches, we can potentially group multiple error behaviors from the same kernel bug. Furthermore, we will manually confirm the root causes and create our "ground-truth" dataset; Second, considering the time-consuming nature and high-level domain expertise, we prioritize the *depth* of our analysis while maintaining a reasonable scale of generalizable results. We filter kernel bugs with reproducers that manifest multiple error behaviors and select some of them as our sample dataset. Then we form a focused group of security analysts to identify the factors to the manifestation of error behaviors. Based on the results, we will propose solutions to explore *new* error behaviors.

2.2 Kernel Bug Dataset

To conduct our study, we choose Syzbot platform [8] to collect kernel bugs reported by Syzkaller. The reasons are 1) Syzkaller found 4335 kernel bugs (3352 of them are now patched) in recent five years. Quantitatively speaking, these bugs are more than the kernel bugs identified in the past 20 years before Syzkaller was invented; 2) Syzkaller files and sends error reports to Syzbot platform. Each bug has its own webpage that contains key information (*e.g.,* vulnerable kernel version, kernel configuration file, reproducer). We regard the bug title in the webpage as the error behavior, for example, the bug title `KASAN: use-after-free Read in ↪ map_lookup_elem` indicates that the kernel bug is detected by KASAN [9] and the error `use-after-free` occurs in the function `map_lookup_elem`. The same kernel bug can manifest different error behaviors (*i. e.,* bug titles) due to different execution contexts.

Syzbot marks a kernel bug as "Fixed" when it is fixed, and fills one additional field - "Fix commit". As discussed above, we take advantage of "Fixed" bugs to establish the ground-truth for our study. The rationale behind is that "Fix commit" shows the patch that fixes the underlying kernel bug. And the error behaviors that are eventually fixed by the same patch are highly likely caused by the same root cause [1] (*i. e.,* the same kernel bug).

Our Dataset. We crawl all the "Fixed" bugs from Syzbot dashboard. Table 1 covers all the corresponding error reports from September 6, 2017 to March 1, 2022. In total, there are 3352 error reports with 2296 unique kernel bugs. Then we analyze the patches of these bugs and identify 440 groups containing two or more error reports. In each group, all the error reports share the same patch and each error report represents one error behavior of the same bug.

1. Group manifested error behaviors with the patch is a reliable method except for rare cases, *e.g.,* the incorrect patch was assigned due to human errors.

Category	GT Kernel Bugs	# of Error Behaviors
Fixed Bugs	2296	3352
Manifested	440	1496
Sampled	162	484

TABLE 1: Dataset overview. "GT Kernel Bugs" refers to the number of ground-truth kernel bugs after linking with fix commits.

Note that, our manual analysis of these groups has confirmed that one patch is always used to fix one bug which is consistent with the policy - "one patch per bug" that the Linux kernel community has been enforcing before pushing a patch to the kernel [10]. As such, our ground-truth dataset is valid.

Sample Dataset. To understand the causes of manifested error behaviors, we organize domain experts to analyze the groups of error behaviors manually. As mentioned, we prioritize the *depth* of analysis and sample a subset of groups that has multiple error behaviors. Our sampling strategy is not completely random. Instead, we first prioritize analyzing bugs that cover more diverse error behaviors (*i. e.,* bug titles); second, we prioritize analyzing bugs that have more severe error behaviors. For example, Out-Of-Bound, Use-After-Free and Invalid-Free are considered to be severer bugs [11].

Among 440 groups with multiple error behaviors, we sampled 162 groups with 484 error reports. Meanwhile, Figure 1 shows that our sample dataset covers diverse types of error behaviors. In addition, the size of our sample dataset is larger than most existing datasets in the previous works [12].

Justifications on the Dataset Size. We believe our dataset of 162 unique kernel bugs is reasonably large for our purposes. On one hand, 162 bugs already cover 36.8% of bug groups that have multiple error behaviors. On the other hand, our 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 [13], [14], [15], [16], [17], [18], [19], [20]. Several works studied 20-60 bugs [12], [21], [22], [23], 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 [24] 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.

2.3 Experiment Overview

We design an experiment to examine the reasons that manifest different error behaviors for the same bug, with the approval of IRB (STUDY00008566).

Experiment Workflow. The analyst 1) gathers all the error reports and corresponding information (*e.g.,* vulnerable

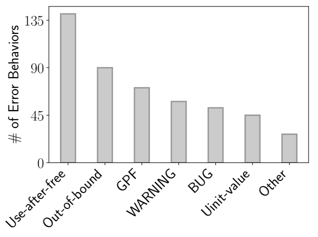


Fig. 1: Number of error behaviors under each crash type in our *sample* dataset.

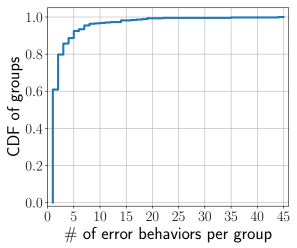


Fig. 2: Number of error behaviors for each bug in the *Manifested* category.

Exploitation Potential	Kernel Bug Errors
Likely to exploit	KASAN (e.g., use-after-free, out-of-bound access, double-free)
Less likely to exploit	BUG, GPF, NULL ptr dereference, panic, WARN, wrappers (e.g., pr_err)

TABLE 2: The summary of the types of error behaviors in bug reports and their corresponding exploitation potential.

source code, kernel configuration, reproducers); 2) compiles the vulnerable Linux kernel and reproduces the kernel bug with the provided reproducer; 3) verifies the effectiveness of patch and examines the reasons by reading the commit message and corresponding code changes; 4) identifies the corrupted data objects based on the root cause and then compares the propagation path of corrupted data objects from the root cause to the crashing site; 5) finally identifies the factors that contribute the manifestation of multiple error behaviors.

The Analyst Team. We organize a strong team with 5 security analysts to conduct our experiment. Among them, two have first-hand experience in regularly analyzing Linux kernel bugs, writing kernel exploits and developing kernel patches. And the rest have in-depth knowledge of different systems in the Linux kernel and actively contribute to the kernel community. When one security analyst finishes the analysis of one group, this analyst presents the details of the underlying bug and explains the identified factors to other analysts for a peer review. After all the analysts confirm the correctness of the results, we close the case for the given group.

3 Study Result

In this section, we analyze kernel bug dataset and illustrate the general impact of multiple error behaviors. Next, we will focus on the reasons behind the manifestation of multiple error behaviors. Finally, we filter out reasons that help manifest multiple error behaviors implying different exploitation potentials.

3.1 Implication of Multiple Error Behaviors.

We analyze kernel bugs and corresponding error reports in our dataset listed in Table 1. In the following, we characterize the manifested error behaviors for kernel bugs.

Prevalence of Multiple Error Behaviors. As shown in Table 1, there are 440 groups of error reports that contain multiple error behaviors. In other words, 19.2% of fixed kernel bugs on Syzbot have two or more different error reports. Figure 2 depicts the number of error reports per group in our *Manifested* category (440 groups in total). We find that about 60.9% of groups have only two error behaviors. However, about 14.3% of the bug groups have more than five error behaviors. The largest group has 45 distinct error reports. This indicates the same bug can cause highly diverse error behaviors.

Category	Likely to exp.	Less likely to exp.
Bugs with OEB	429/1856 (23.1%)	1427/1856 (76.9%)
Bugs with MEB	198/440 (45.0%)	242/440 (55.0%)

TABLE 3: Exploitation Potential of kernel bugs. *Bugs with OEB* means kernel bugs with only one error behavior; while *Bugs with MEB* means kernel bugs with multiple error behaviors.

Exploitation Potential of Multiple Error Behaviors. Table 3 confirms the hypothesis – bugs with multiple error behaviors have higher exploitation potential compared to bugs with only one error behavior. More specifically, given a bug, if any of error behavior(s) is likely to exploit, we define this bug has a higher exploitation potential. To obtain the relationship between an error behavior and its exploitability, we conducted a user study under the approved IRB (STUDY00008566). As is depicted in Table 2, each error behavior is categorized into either "likely to exploit" or "less likely to exploit". Compared with error behaviors (*e.g.,* BUG/GPF/WARN/NULL ptr deref), kernel error behaviors such as double-free, use-after-free, and out-of-bound imply higher exploitability. As a result, in Table 3, bugs with multiple error behaviors has more likely-to-exploit bugs (45%) than bugs with one error behavior (23%). In addition to the above hypothesis, another finding is that, the more error behaviors one bug has, the higher possibility of likely-to-exploit error behaviors the bugs can manifest. More specifically, we divide bugs in the *manifested* category into groups with the same amount of error behaviors. For each group, we calculate the percentage of likely-to-exploit error behaviors. For groups with above 7 error behaviors, this rate is mostly higher than 40%, while groups with less than 7 error behaviors only have less than 40% percentage.

Of all fixed bugs on Syzbot, 19.2% kernel bugs have multiple error behaviors. 14.3% of groups have more than 5 error behaviors, even one bug can have 45 distinct error behaviors. Moreover, the more error behaviors one bug has, the higher possibility of likely-to-exploit error behaviors the bugs can manifest.

3.2 Contributing Factors

Manual Crash Analysis. This manual analysis is focused on the sample dataset that includes 162 kernel bugs and 484 error behaviors (see Table 1). The analysis took 5 security analysts about 3,000 man-hours to finish. On average, each kernel bug took almost 4 hours to complete all the steps proposed in Experiment Workflow. Based on our experience, the most time-consuming part is to understand the

Factors	Groups	Percentage
Different Inputs	70	48.3%
Memory Dynamics	25	17.2%
Thread Interleaving	24	16.6%
Different Sanitizers	21	14.5%
Inline Function	17	11.7%
Kernel Versions and Branches	13	9.0%

TABLE 4: The number of groups that are affected by each factor. One group could be affected by multiple factors.

```
1  static void tun_attach(struct tun_struct *tun, ...)
2  {
3     if (tun->flags & IFF_NAPI) {
4        // initialize a timer
5        hrtimer_init(&napi->timer, CLOCK_MONOTONIC,
6           HRTIMER_MODE_REL_PINNED);
7        // link current napi to the device's napi list
8        list_add(&napi->dev_list, &dev->napi_list);
9     }
10 }
11
12 static void tun_detach(struct tun_file *tfile, ...)
13 {
14    struct tun_struct *tun = rtnl_dereference(tfile->tun);
15    if (tun->flags & IFF_NAPI) {
16       // GPF happens if timer is uninitialized
17       hrtimer_cancel(&tfile->napi->timer);
18       // remove the current napi from the list
19       netif_napi_del(&tfile->napi);
20    }
21    destroy(tfile); // free napi
22 }
23
24 void free_netdev(struct net_device *dev) {
25    list_for_each_entry_safe(p, n,
26          &dev->napi_list, dev_list)
27       netif_napi_del(p); // use-after-free
28 }
```

Listing 1: The code snippet of one bug in Linux kernel. When triggered with different system call sequences and arguments, the bug demonstrates two different error behaviors – a general protection fault error and a use-after-free error.

root cause and identify the propagation path of corrupted variables from the root cause to the crashing site.

Out of the 162 groups in our sample dataset, we successfully reproduced and examined the root causes for 151 groups. Other non-reproducible bugs cannot be analyzed in our experiment. And we find 6 groups in which Syzbot incorrectly assigned the fix commit. Therefore, we use the rest 145 groups as our final set to illustrate our findings on the contributing factors to multiple error behaviors of one bug.

We identify 6 factors leading the manifestation of multiple error behaviors, summarized in Table 4.

Factor 1: Different inputs. The first reason for multiple error behaviors is the input difference. Given a kernel bug, different inputs can lead to *various execution contexts* and *execution paths*. Following these paths under various execution contexts, the bug can demonstrate multiple error behaviors and form different error reports.

Take the group #aec72f3392b1 [2] in Listing 1 as an example, there are two manifested error behaviors - general protection fault [25] and use-after-free [26]. In Linux kernel, a tun device is shared by all opened tun files, and each

2. We take the patch id shown on the Syzbot dashboard as the group id

```
void netlink_ack(..., struct netlink_ext_ack *extack) {
   if (nlk_has_extack && extack && extack->_msg)
      // GPF occurs if _msg points to invalid memory
      tlvlen += nla_total_size(strlen(extack->_msg) + 1);
   if (nlk_has_extack && extack) {
      if (extack->cookie_len)
         // OOB occurs if cookie and length do not match
         WARN_ON(nla_put(skb, NLMSGERR_ATTR_COOKIE,
               extack->cookie_len,
               extack->cookie));
   }
}
```

Listing 2: The code fragment illustrating the difference of memory dynamics can mainfest two different error behaviors – GPF and Out-of-Bound(OOB)

tun file can update `tun->flags` in the function `tun_attach`. The underlying bug results from the potential inconsistent state of the flag `tun->flags` between `tun_attach` at Line 1 and `tun_detach` at Line 12.

In the former report, the PoC program invokes the 1st `ioctl` that calls `tun_attach` with `IFF_NAPI` in `tun->flags` unset. In this way, the kernel neither initializes the hrtimer nor adds the napi into the list of `dev->napi_list`. Next, the PoC program invokes the 2nd `ioctl` that updates `IFF_NAPI` ↪ before `tun_detach`, which causes the inconsistent state between `tun_attach` and `tun_detach`. Finally, the kernel invokes `hrtimer_cancel` on the uninitialized timer object at Line 17, leading to the error behavior - GPF. Compared with the former report, the PoC program in the latter report adjusts the order of two `ioctl` syscalls. In other words, the 1st `ioctl` invokes `tun_attach` with `IFF_NAPI` set, the kernel initializes a hrtimer and add `napi` into the list of `dev->napi_list` from Line 5 to Line 8. Then the 2nd `ioctl` clears `IFF_NAPI` in `tun->flags` ↪ before the `tun_detach`, the kernel skips the cancellation of hrtimer and the deletion of `napi`, but destroys the tun file. As a result, this inconsistency leaves a dangling pointer in the list of `dev->napi_list`. In the last, `free_netdev` at Line 24 dereferences the freed object - `napi`, causing the error behavior - UAF.

Table 4 shows input difference affects 70 of groups (48.3%) we have analyzed. This is the most prevalent reason for multiple error behaviors.

Factor 2: Memory Dynamics. In Linux kernel, the slab allocator is shared among all the kernel subsystems, responsible for dynamic memory management. The memory layout of Linux kernel at runtime is non-deterministic. The same for the memory value of any uninitialized kernel variables. As a result, when the underlying bug is triggered by a PoC program, the memory status (including memory layout and value) can be uncertain, causing the same bug to manifest different error behaviors and produce different error reports.

As depicted in Listing 2, the underlying kernel bug skips the initialization of `extack` (*i. e.*, the 4th argument of `netlink_ack`), leaving this variable uninitialized. Therefore, the content of extack is non-determinstic, *i. e.*, the fields `_msg`, `cookie_len` and `cookie` are unknown at runtime. If `extack->` ↪ `_msg` points to an invalid memory area, the kernel triggers a general protection fault - GPF. However, if `extack->_msg` is correct, the kernel goes down and uses `cookie` and `cookie_len` ↪ as the base address and length to access the cookie. If

```
 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(
18        &pi_state->refcount));
19  }
```

Listing 3: The code fragment illustrating the difference of thread interleaving can lead to two different error behaviors – Warning and Use After Free.

`cookie_len` is greater than the length of `cookie`, it can lead to another error behavior - Out-of-Bound (OOB).

As shown in Table 4, 25 of groups are affected by this factor – memory dynamics.

Factor 3: Thread interleaving. Linux kernel is a fully multi-threaded system that supports many tasks running at the same time. Incorrect synchronization (or missing synchronization) between kernel tasks not only introduces concurrency bugs, but may also produce different errors behaviors.

Listing 3 shows an example where `pi_state` is a variable shared between thread-*A* and thread-*B*. With the same PoC program, the kernel could experience different error behaviors in different thread interleaving. For one possible situation, thread-*A* invokes the function `put_pi_state`, decreasing the reference count for `pi_state` to zero (Line 3). Following thread-*B* calls the function `get_pi_state` and raise a warning at Line 17. However, another synchronization between both threads is to call the function `put_pi_state` in thread-*A* prior to thread-*B* executing Line 12, which then results in `pi_state` being deallocated while `pi_state` of thread-*B* still points to it. 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 affects 24 bug groups (*i. e.,* 16.6%).

Factor 4: Sanitizers. The state-of-art kernel fuzzers usually make use of all kinds of sanitizers to discover kernel bugs. Different sanitizers are designed to detect different kinds of error behaviors. Thus how to setup kernel sanitizers is also a factor contributing to the manifested error behaviors. Similar to previous two factors - memory dynamics and thread interleaving, we find that 21 of groups (*i. e.,* 14.5%) are affected by this factor (see Table 4).

Factor 5: Inline Function. The error behaviors of one bug can also be affected by the compilation of kernel code. To be specific, the compiler can make an opposite decision regarding if a function is *inline*, depending on the kernel configurations, the compiler and its version. In such case, we find that, although error behaviors of one bug have different functions, in fact they share the same crashing trace. Our analysis shows that this factor affects 17 of groups in the sample dataset.

```
unsigned int tipc_poll(... sock) {
  switch (sk->sk_state) {
  case TIPC_OPEN:
     // upstream a8750ddca918
     if (!grp || tipc_group_size(grp))
     // net-next 594831a8aba3
     if (!grp || tipc_group_is_open(grp))
  }
}
```

Listing 4: The code example indicating different kernel versions and branches affect the manifestation of error behaviors

Factor 6: Kernel Versions and Branches. One kernel bug can exist in multiple kernel versions and branches. When the same bug is triggered in the different kernel versions and branches, it can manifest different error behaviors as the code in the execution path changes across time.

As shown in Listing 4, the group `#60c253069632` is a use-after-free bug existing in many kernel versions. In the commit `a8750ddca918` of upstream repository, the error site is reported in `tipc_group_size`; However, in the commit `594831a8aba3` 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.

Table 4 shows there are 9.0% of groups affected by this factor.

Input difference is the most prevalent reason that contribute to multiple error behaviors. The rest factors in sequence are *memory dynamics*, *thread interleaving*, *sanitizers*, *inline function* and *kernel versions and branches*.

3.3 Factors leading to different exploitation potential

In this part, we carry out a further analysis to filter out contributing factor that can lead to different exploitation potential. In the following, we will discuss all the factors one by one.

Input difference. Out of 70 kernel bugs, we found that 23 groups of error behaviors indicating different exploitation potential. Take Listing 1 as an example, this bug manifests two error behaviors – less-likely-to-exploit GPF and likely-to-exploit UAF with different syscall sequences. Therefore, if only based on GPF, security analysts might infer this bug is probably unexploitable. However, with GPF and UAF multiple behaviors, security analysts might treat this bug as probably exploitable.

Memory Dynamics. There are 2 of 25 groups in which error behaviors show different exploitation potential. The bug in Listing 2 generates two different error behaviors – less-likely-to-exploit GPF and likely-to-exploit UAF with different memory values. Compared with only GPF error behavior, security analysts tend to regard this bug as likely-to-exploit with the view of both two error behaviors.

Thread interleaving. Among 24 groups affected by this factor in Table 4, 11 groups exhibit different exploitation potential. As shown in Listing 3, the manifested error behaviors caused by different thread interleaving are less-likely-exploit WARN and likely-to-exploit UAF.

Sanitizers and inline function. Our sampled kernel bugs affected by sanitizers and inline function do not incur any exploitability change. For the former, no matter sanitizers are enabled or not, the underlying execution path is exactly the same. For the later, the underlying execution path is also the same. The difference of error behaviors is due to the naming convention of error reports in Syzkaller.

Kernel versions and branches. As to the exploitability of the same bug in different versions and branches, it is an open question depending on whether the code changes introduce new powerful exploit primitives. From our manual analysis, most bugs only change one function name or replace a function with a highly similar function (See details in Listing 4). They do not exhibit different exploitablity potential with multiple error behaviors.

Input difference, *memory dynamics* and *thread interleaving* can manifest multiple error behaviors with different exploitation potential, but *Sanitizers* and *inline function* cannot. *Kernel versions and branches* may generate error behaviors with different exploitability theoretically, but our sample dataset does not have.

4 Multiple Error Behaviors Exploration

In previous sections, our empirical study not only analyzes the prevalence and implication of multiple error behaviors, but also identifies the key factors that bring out these phenomenon. In this section, we take advantages of these identified contributing factors to explore multiple error behaviors of kernel bugs and prototype the solution as a tool.

4.1 Design Overview

As shown in Section 3, *input difference* and *thread interleaving* are two factors that have the highest possibility to manifest error behaviors with different exploitation potentials. Given a kernel bug report, one instinctive reaction is to manipulate kernel input and thread interleaving, and meanwhile, trigger the same bug to explore other error behaviors. Existing research works (*e.g.*, HFL [27], Ruzzer [28], SnowBoard [29], MUZZ [30]) design testing frameworks to explore only thread interleaving, only test input or intelligently explore both jointly. However, such approaches are not likely to be effective since they avoid executing the same code paths or repeating the same thread interleaving(i.e., the root cause of the underlying concurrency bug) that has already been explored. This does not match our ultimate goal since we need to execute the same buggy code snippets repeatedly with different contexts. Therefore, it is hard to consider *input difference*, *thread interleaving* and *triggering the same bug* at the same time. Since *input difference* is the most prevalent contributing factors, we will only consider *input difference* and *triggering the same bug* in this paper, leaving *thread interleaving* as our future work.

Therefore, we turn our attention to directed fuzzing, which mutates test inputs and exercises the buggy code segment. However, directed fuzzing has its own limitation. First, it is challenging to pinpoint the root causes of kernel bugs required in the directed fuzzing correctly and automatically. Second, even if we can make directed fuzzing repeatedly reach out to the buggy code, it does not mean the kernel bug can manifest multiple error behaviors.

Our Approach. In this work, we address this problem by extending an existing kernel fuzzing approach (*i. e.*, Syzkaller) with kernel-object guidance. From the experiment workflow, we learn that the root cause of a kernel bug usually results from the inappropriate usage of a kernel object contributing to an error behavior and the incorrect value involved in computation with a kernel object, which is further propagated to a critical kernel operation, forcing a kernel to demonstrate an error behavior. As such, guided by the objects relevant to the error behavior, we can have the fuzzer away from those paths and contexts irrelevant to the bug and thus improve its efficiency significantly.

To realize the idea mentioned above, our design combines static analysis and kernel fuzzing techniques. As depicted in Figure 3, we first take as input a kernel bug report, run the enclosed PoC program, and track down those kernel structures involved in the kernel errors (*e.g.*, `struct tun_file` ↪ in the Listing 1). The objects in these types indicate the possible objects under inappropriate usage or involving computation with an incorrect value. Therefore, we further examine the kernel source code and identify the statements that operate the objects in these types. Then we treat these statements as the sites critical to the success of kernel bug triggering. As a result, we instrument these statements so that we can collect the feedback of object coverage when performing kernel fuzzing and then use this new coverage to adjust the corresponding PoC program. In this work, our kernel fuzzing mechanism takes as input the original PoC program attached in the bug report. Using a new mutation and seed generation method, varies the PoC program to improve the efficiency and effectiveness in the exploration of other error behaviors. In the following subsections, we will discuss these techniques in detail.

4.2 Critical Structure Identification

In this work, we utilize backward taint analysis to identify essential kernel structures (*i. e.*, those involved in the error specified in the given bug report). Here, we detail how we identify the source and the sink and thus perform backward taint analysis accordingly.

4.2.1 Report Analysis & Taint Source Identification

The Linux kernel enforces checks during execution and examines whether pre-defined conditions are satisfied. According to the ways of checking, we categorize them into explicit checking and implicit checking. If the conditions do not hold, then the kernel runs into an error state and logs critical information for debugging purposes.

Explicit Checking. The kernel developers explicitly formulate the checking as an expression and pass it to the standard debugging features, such as `WARN_ON` and `BUG_ON`. Take the case in Listing 5 as an example. The error is be reported if and only if the predefined condition "`!list_empty(&dev->work_list` ↪ `)`" at Line 4 is true at runtime. After the condition is

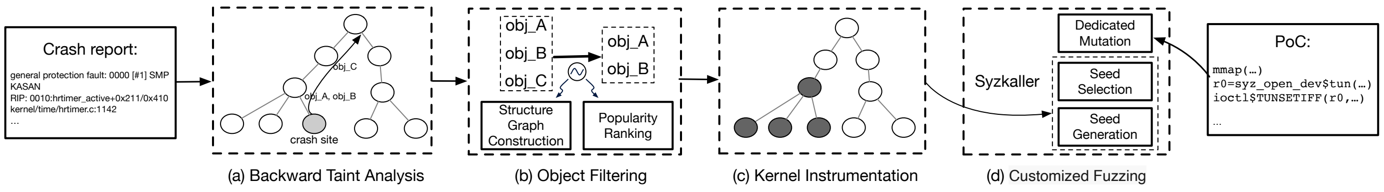
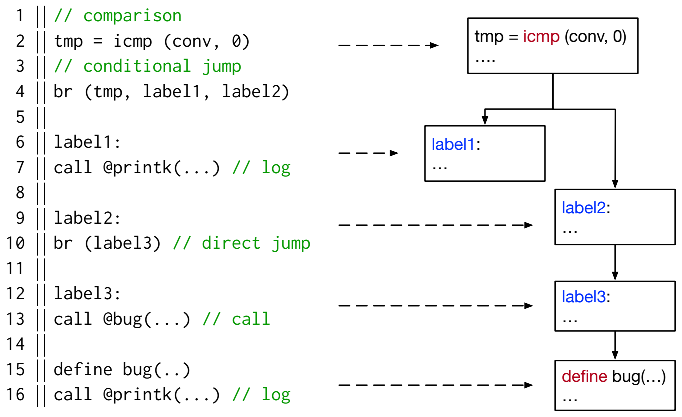


Fig. 3: The workflow of GREBE. (a) Following a kernel error trace obtained from a crash report, GREBE performs backward taint analysis and identifies all the kernel objects involved in the crash. (b) Based on the objects' rareness, GREBE narrows down the objects critical to the kernel error. (c) Guided by the objects filtered out in the last step, GREBE instruments kernel and treats the critical objects' (de)allocation and dereference sites as the anchor sites. (d) GREBE customizes Syzkaller so that it could leverage the anchor sites' reachability feedback to select seeds. Besides, GREBE introduces a customized mechanism to mutate seeds so that GREBE could diversify the ways to trigger the same kernel bug.

```
// in drivers/vhost/vhost.c
void vhost_dev_cleanup(struct vhost_dev *dev)
{
   WARN_ON(!list_empty(&dev->work_list));
   if (dev->worker) {
      kthread_stop(dev->worker);
      dev->worker = NULL;
      dev->kcov_handle = 0;
   }
}
// in include/asm-genric/bug.h
#define WARN_ON(condition) ({       \
  int __ret_warn_on = !!(condition);     \
  if (unlikely(__ret_warn_on))      \
    __WARN();       \
  unlikely(__ret_warn_on);      \
})
```

Listing 5: The code snippet that performs explicit checking.



```
1  || // comparison
2  || tmp = icmp (conv, 0)
3  || // conditional jump
4  || br (tmp, label1, label2)
5  ||
6  || label1:
7  || call @printk(...) // log
8  ||
9  || label2:
10 || br (label3) // direct jump
11 ||
12 || label3:
13 || call @bug(...) // call
14 ||
15 || define bug(..)
16 || call @printk(...) // log
```

Fig. 4: An illustrating example and its dominator tree, which demonstrate two different methods of logging kernel errors. Line 7 is a logging statement responsible for kernel error recording. Line 15 is the wrapper of the logging statement at line 16. The variable `conv` in line 1 is the taint source that our proposed approach identifies. Note that for simplicity we place the two error logging functions at two different branches sharing the same conditional jump block. In the real world, error logging cannot occur in this way.

satisfied, it is a patterned code block inside the macro (`WARN_ON`) that includes a condition statement and a logging statement will be executed. Therefore, the variable `&dev->` ↪ `work_list` indicates a cause of the bug and should become the starting point of our analysis (*i. e.*, the taint source of our backward analysis). Apart from this standardized way to log kernel errors, developers can also build their own

```
// source code
walk->offset = sg->offset;

// pseudo binary code after instrumentation
kasan_check_read(&sg->offset, sizeof(var));
tmp = LOAD(&sg->offset, sizeof(var)); // first access
kasan_check_write(&walk->offset, sizeof(var));
STORE(tmp, &walk->offset); // second access
```

Listing 6: The code snippet that performs implicit checking.

macro that wraps a logging statement in a helper function (*e.g.*, the code in Line 15 & 16 of Figure 4).

To identify the condition that triggers the execution of the logging statement and thus pinpoint the taint source, we first trace back along the dominator tree until we find a dominator basic block, the last statement of which is a conditional jump (*e.g.*, given the wrapped logging statement in Line 16 of Figure 4, Line 4 is the statement linking to the dominator basic block). Second, we treat the corresponding comparison as the condition that triggers the execution of the error logging (*e.g.*, Line 2 in Figure 4). Finally, we extract the corresponding variable in the condition as our taint source (*e.g.*, `conv` in Figure 4).

Implicit Checking. Implicit checking refers to the situation where the checking is instrumented by a compiler or completed by hardware. Kernel Address Sanitizer (KASAN) is such an example of a implicit checking which relies on shadow memory to record the memory status. If the instrumented kernel touches a freed memory region, it will generate a bug report indicating the instruction that triggers a use-after-free error. Regarding the implicit checking done by interrupts (e.g., general protection fault detected by MMU), the interrupt handling routine is responsible for logging the corresponding instruction.

From bug reports generated by these debugging mechanisms, we can easily identify the instruction that performs the invalid memory access. With this information in hand, our next step is to identify the variable associated with that invalid memory access. However, the binary instruction enclosed in the report contains no type information. To deal with this problem, we map binary instructions with their corresponding statements in the source code. In the case the mapped source code is a simple statement with only one load or store, we directly conclude that this statement is the one causing the illegal memory access and treat the operand variable as a taint source. However, if the identified instruction links to a compound statement involving multiple memory loads and stores (e.g., `walk->offset = sg->offset` depicted in Listing 6), we perform further analysis. To be specific, we first examine the bug report and pinpoint the specific instruction that captures the kernel error. Then, we treat the memory access associated with the error-catching instruction as our taint source. Take the case shown in Listing 6 for example, the bug report indicates the error is captured by the statement `kasan_check_read(&sg->offset, ↪ sizeof(var))` which associates with `sg->offset`. We deem `sg->offset` in line 2 as the taint source.

4.2.2 Taint Propagation & Sink Identification

Recall that backward taint analysis aims to find critical structures, i.e., those structures involved in the error specified in the given bug report. To do it, we again extract the call trace from the bug report. Based on the trace, we then construct its control flow graph and propagate the taint source backward on the graph.

Along with the backward propagation, we use the following strategy to perform variable tainting. If the tainted variable is a field of a nested structure or a union variable, we further taint its parent structure variable and treat the parent structure as a critical structure. The reason is that the nested structure or the union variable is part of the parent structure variable in the memory. If a field of the nested structure or the union variable carries an invalid value, it likely results from the inappropriate use of its parent structure variable. When backward taint propagation encounters a loop, we also propagate the taint to the loop counter if the taint source was updated inside the loop. By extending the taint to the loop variable, we can include the corrupted variable, which could further help us identify other structure variables relevant to the corruption.

In this work, we terminate our backward taint process until one of the following conditions holds. First, we terminate our taint analysis if the backward propagation reaches out to the definition of a tainted variable. Second, we terminate our taint propagation if it reaches out to a system call's entry, an interrupt handler, or the entry of the function that starts the scheduler of work queue. It is simply because they indicate the sites where the kernel debugging features start to trace kernel execution for later stage debugging. It should be noted that, while performing backward taint propagation, we also extend propagation to the aliases of the tainted variable. In this work, we treat structural types of all the taint variables as the critical structure candidates for our kernel fuzzing guidance.

4.3 Kernel Structure Ranking

As we will discuss below, using the structures identified by backward taint analysis to guide kernel fuzzing and explore the bug's other error behaviors, we could still confront low efficiency and even poor effectiveness. As a result, before applying these structures and their corresponding objects to guide our kernel fuzzing, we need to further narrow down the kernel structures for kernel fuzzing guidance.

Kernel structure selection. Linux kernel maintains its code quality with plenty of design patterns [31]. These patterns provide a suggested practice and framework to manage data

```
static inline void *__skb_push(struct sk_buff *skb, ...)
{
   return skb->data;
}

int ip6_fraglist_init(...)
{
   struct frag_hdr *fh;
   // type casting from void* to struct frag_hdr*
   fh = __skb_push(skb, sizeof(struct frag_hdr));
}
```

Listing 7: The code snippet indicating type casting.

in a commonly recognized fashion. There are many structures (e.g., `struct list_head`, `struct sk_buff`) used pervasively in Linux kernel codebase. Including such popular structures for kernel fuzzing guidance, the kernel fuzzer would inevitably explore a large code space, driving the fuzzer away from the buggy code attributing to the error specified in the report. Apart from this kind of popular structures, there are other kinds of popular structures pertaining to abstract interface. Therefore, to preserve the kernel fuzzer's efficiency in exploring a bug's multiple behaviors, we need to exclude these popular structures from our kernel fuzzing guidance.

To pinpoint and exclude popular structures for multiple error behavior exploration, we design a systematic approach to ranking the kernel structures based on their popularity. At a high level, this method constructs a graph describing the reference relationship between kernel structures. Each node in the graph represents a kernel structure, and the directed edges between nodes indicate the reference relationships. On the graph, we apply PageRank [32] which assigns each structure a weight. In this work, we deem the structure with a higher weight a more popular structure than others and exclude them while performing kernel fuzzing for other error behavior exploration.

Structure graph construction. To construct the structure graph mentioned above, we first analyze all the structures defined in the kernel source code. Given one structure, we go through all its field members. If the field is a pointer to another structure, we link the given structure to the referenced structure. Suppose the field is a nested structure or union, in that case, we expand them repeatedly until we identify a self-referenced structure, or there is no more nested structure/union in the definition. We link the given structure directly to the structure in the last layer of expansion, ignoring the union in the middle to shrink graph size.

In addition to analyzing the structure definition in kernel source code, we also construct the structure graph with the consideration of typecasting. Since the kernel supports polymorphism that uses a single interface to describe different devices and features, one abstract data type can be cast to a more concrete type. Take the function `ip6_fraglist_init ↪` in Listing 7 as an example. In this function, `skb->data` is cast from `void*` to `struct frag_hdr*` which is further used in the IPV6 networking stack. The `void*` is an abstract data type, whereas the destination structural type `struct frag_hdr ↪ *` is more concretized. As such, we add one more edge to our structure graph, which links `struct skb_buff` to `struct ↪ frag_hdr`.

Intuition suggests that the structures with more references are more popular ones. Besides, the structures referenced by popular structures can also be popular because they can also be used in many program sites in the kernel. To identify these kernel structures, we utilize the PageRank algorithm on the graph to rank their popularity. In this work, we use only those kernel structures and objects with lower ranks to guide our fuzzing process. In Section 5, we discuss how we choose the page-rank score threshold to distinguish popular structures from less popular ones.

4.4 Object-driven Kernel Fuzzing

With the critical structures identified, we now discuss how we utilize these structures to facilitate kernel fuzzing and thus explore multiple error behaviors for one kernel bug.

Instrumentation. Conventional kernel fuzzing methods instrument tracing functions to keep track of basic blocks that have been executed. In this work, our fuzzing mechanism introduces one additional instrumentation component which is designed as a compiler plugin to examines each statement in basic blocks and identifies those basic blocks that take the responsibility for the allocation, de-allocation, and usage of critical objects (i.e., the objects in the type of critical structures). Specifically, the instrumentation component introduces a new tracing function that replaces the most significant 16 bits of the recorded basic block address with a magic number to differentiate these basic blocks from others. By observing the most significant 16 bits of addresses in the code coverage feedback, we can easily pinpoint which basic block pertaining to the critical objects is under the operation of the corresponding fuzzing program.

Seed selection. With the facilitation of the above instrumentation, when running a fuzzing program, we can easily determine whether it reaches a critical object. Once we identify a new critical object coverage, we can add the corresponding fuzzing program to our seed corpus. In this work, we include the mutated seed program or the newly generated seed program into the seed corpus only if one of the following two conditions holds. First, the program reaches out to an unseen basic block involving critical object operations. Second, at least one system call in the program covers more code, and the same system call has demonstrated critical object operation in the previous fuzzing.

Seed generation & mutation. In this work, we initialize the seed corpus with the PoC program enclosed in the bug report. Every time, when generating a new seed fuzzing program, we assemble the new fuzzing program by only using system calls that have already been included in the seed corpus. This is very different from the seed generation method used in the state-of-the-art fuzzing technique (*e.g.,* Syzkaller), which not only adopt the system calls enclosed in the corpus but also bring in the new system calls. The reason behind our design change is that exploring multiple error behaviors of a kernel bug requires triggering a critical object accessing under different contexts or through different execution paths. Randomly introducing new system calls into the new seed fuzzing program could enlarge the code coverage the fuzzing program can explore. However, it inevitably detours the fuzzing program away from the critical objects.

Intuition suggests that using the aforementioned seed generation approach alone is not likely to explore a sufficient number of contexts and paths pertaining to the critical objects. As such, we further introduce the mutation mechanism used in the existing kernel fuzzing technique (*i. e.,* Syzkaller). This mutation mechanism introduces into the seed fuzzing program new system calls that are relevant to the system calls already enclosed in the seed corpus. In this way, we expect the fuzzing program could still stick with the critical object and, at the same time, diversify the execution contexts or the paths towards the object.

Mutation optimization. When performing the mutation for a fuzzing program, the mutation mechanism of Syzkaller utilizes pre-defined templates to guide the synthesis of new fuzzing programs. A template specifies the dependency between system calls and the argument format of corresponding system calls. For example, Syzkaller's template specifies that the system call `read` requires a resource (i.e., a file descriptor) as one of its arguments, and the syscall `openat`, will generate the corresponding resource. Under the guidance of this template, Syzkaller could perform mutation against a fuzzing program by appending the system call `read` ↪ to the system call `openat`. The mutation ensures the seed program is legitimate and thus avoids the kernel's early rejection against the fuzzing program.

As mentioned above, our mutation mechanism borrows the method used in Syzkaller. As we will show in Section 6, while this approach is useful in avoiding generating invalid kernel fuzzing programs, it is still inefficient and sometimes ineffective in guiding our kernel fuzzer to exhibit multiple behaviors for one kernel bug. As we elaborate below, the reasons behind this are two folds.

First of all, Syzkaller attempts to introduce various system calls relevant to the seed program and randomly manipulate system calls' arguments. Mutation without the consideration of them would inevitably incur low effectiveness in exploring multiple error behaviors.

To resolve the problems above, we improve our fuzzing approach by optimizing its mutation mechanism. More specifically, we group the system call specification templates based on the resource type the corresponding system calls reply upon (*e.g.,* categorizing system calls pertaining to the network socket and device file separately). Within each group, we then divide the enclosed system calls into two subgroups. One is responsible for resource creation, and the other is for their usage. With this grouping result, our fuzzing component either replaces system calls with the ones in the same group or inserts system calls that associate with the resource shown in the seed program.

From empirical study, we learn that specific resources and arguments are necessary for successfully triggering a target kernel bug. So in addition to grouping templates based on resource, our mutation mechanism also preserves the values for the arguments seen in the original PoC program if the types of these arguments do not fall into the following four categories – constant, pointer referencing a memory region, checksum, and resource (e.g., a file descriptor for an opened file or an established socket). For arguments in constant, they usually indicate the protocol under fuzzing testing. In the fuzzing test, we need to alter these arguments to switch protocols and thus vary the contexts under which the bug could be triggered again with different error behaviors. For arguments in pointer and resources types, when the kernel fuzzing changes the context or path toward the buggy kernel code, the original PoC program's addresses could be illegal and thus incur early termination of the fuzzing program. Regarding the checksum, if the calculation source's value varies in the mutation process, the checksum should be updated accordingly. Preserving the same checksum value could also result in the fuzzing program's early termination at the data validation phase.

Here we summarize the connection between our empirical study and the fuzzing mechanism. First, the study results show that input difference is the most prevalent contributing factor manifesting multiple error behaviors with different exploitation potentials, manipulated by our fuzzing mechanism; Second, our empirical study analyzes corrupted data objects to understand the root cause and figure out the underlying reasons for multiple error behaviors, forming the idea of our feedback based on corrupted data objects; Third, the seed selection, generation & mutation, even optimization in the fuzzing mechanism get inspired from the experience in our manual analysis.

5 Implementation

Based on LLVM infrastructure and the kernel fuzzing tool Syzkaller, we implement our idea as a tool and name it after `GREBE`. Below, we describe some critical details of our implementation.

Critical structure identification. The input of our tool is LLVM IR and a single bug report. In our implementation, we employ the approach in previous works [33], [34] to generate the bitcode files. Briefly, we patch the LLVM compiler to dump bitcodes before invoking any compiler optimization passes. In this way, we can prevent compiler optimization from influencing the accuracy of our analysis. Recall that we extract the call trace from the bug report. The call trace indicates the functions that have been called but not yet returned when the kernel is experiencing errors. In this extracted call trace, the function that has been called last could be the one instrumented by the compiler. It does not indicate the buggy function contributing to the error. As such, we neglect these functions in the call trace and start our analysis from the statement that activates the debugging feature.

When using the backward taint analysis to identify critical structures, `GREBE` uses three instructions to extract the structures' type information. The first instruction is `BitCast` in which the types before and after casting are specified. `GREBE` records the types extracted from this instruction as critical structures. The second instruction is `Getelementor` ↪ which contains a pointer referencing a kernel object and the object's corresponding type information. Through the analysis of this instruction, we can quickly obtain critical structures. The third instruction is `CallInst`. We infer the type information from the callee's prototype and record the structural type as critical structures. As mentioned earlier, we treat system calls' entries, interrupt handlers, and workqueue processings as our taint sink. In our work, we manually annotate all these sinks based on their naming patterns.

Critical structure ranking. As is described in Section 4.3, when constructing the structure graph for critical structure ranking, we consider typecasting. In our implementation, if the cast variable is the return value of a callee function, we investigate the callee from the return statement and then associate the destination type with the structure field. Again take the case shown in Listing 7 as an example. The cast variable `skb->data` is the return value of the callee function `__skb_push`. By analyzing the callee function, we associate `struct frag_hdr` with `struct sk_buff`.

Recall that we also rank structures based on their page-rank scores and then use a page-rank score threshold to filter out those popular ones. In this work, we choose this threshold by using a standard univariate outlier detection method [35]. This approach computes the mean and standard deviation of the page rank scores and then calculates the Z-score for each structure further. Following the outlier detection method, we use 3.5 more standard deviations as the threshold. Since most kernel structures are less popular, having a significantly low z-score, this threshold could well distinguish popular kernel structures from the others.

Kernel fuzzing. As is described in Section 4.4, we instrument the kernel to collect the usage of critical objects at runtime. Since the support of Clang has been introduced recently, which may not support all Linux kernel versions, we perform instrumentation by using a GCC plugin instead of a Clang pass. While performing a fuzzing program mutation, we follow the design of MoonShine-enhanced [36] Syzkaller, randomly mutating 33% system calls and replacing them with others we have manually grouped.

When implementing the optimization mechanism that reuses the arguments from the original PoC, for each system call in the PoC program, we first find its specification in Syzkaller and analyze the definition of its structural arguments (i.e., `StructType` and `UnionType`). Then, we recursively examine the structural arguments until no more new definitions can be found. Inside each structural definition, we ignore `ConstType`, `VmaType`, `ResourceType`, and `CsumType` because they represent constant, pointer, resource description, and checksum respectively. As we discussed in Section 4.4, they are not likely to help explore new paths to the buggy code.

6 Evaluation

In this section, we first quantify `GREBE`'s effectiveness and efficiency and compare it with a code-coverage-based fuzzing method. Then, we demonstrate and discuss how well `GREBE` could unveil exploitation potential for real-world Linux kernel bugs.

6.1 Experiment Setup & Design

To evaluate our tool – `GREBE`, we select both *open* kernel bugs and *fixed* kernel bugs from Syzbot as our test cases. While selecting these bugs, we follow two key strategies.

Our first strategy is a purely random selection process that follows two criteria. First, the bug report has to attach a PoC program so that we can reproduce the error behavior specified in the report. Second, the reported kernel error cannot associate with Kernel Memory Sanitizer (KMSAN) because KMSAN is still under development and has not yet been merged into the Linux kernel mainline. By following these two criteria, we construct a test corpus containing 50 Linux kernel bugs.

Our second strategy is a process dedicated to different kernel versions (5.6 - 5.10)[3]. For each kernel version, we choose two recently-reported reproducible kernel bugs as our test cases. In this way, we construct another test corpus with 10 Linux kernel bugs. Combining with the first corpus, our dataset contains 60 unique kernel bugs which is the largest dataset used in exploitability research.

It should be noted that we pick up partially kernel bugs with multiple error behaviors and partially kernel bugs with only one error behavior. For the bug that has multiple error behaviors, we pick up one of the manifested error behaviors as our starting point.

For each bug in our dataset, we built the corresponding kernel in four QEMU virtual machines (VMs) for the purpose of evaluating GREBE's effectiveness and efficiency. For the first two VMs, we ran our tool – GREBE and Syzkaller. For the remaining two VMs, we ran GREBE without enabling its mutation optimization and Syzkaller with our mutation optimization (i.e., Syzkaller's variant). Besides, we can compare it with the code-coverage-based kernel fuzzing method and its variant (i.e., Syzkaller with our mutation optimization). It should be noted that we use Syzkaller as our baseline approach for evaluation because it is one of open-sourced, code-coverage-based kernel fuzzing tools but mostly because it can test nearly all kinds of kernel components.

Given a kernel bug of our selection, its report, and a kernel fuzzing tool under our evaluation, we include the PoC program enclosed in the report into the initial seed set and deploy our VMs on bare-metal AWS servers. Each of the servers has two-socket Intel(R) Xeon(R) Platinum 8275CL CPU @ 3.00GHz (48 cores in total) and 192 GB RAM, running Ubuntu 18.04 LTS. For each VM, we configured it with two virtual CPU cores and 2GB RAM. While performing kernel fuzzing, we set each of the fuzzers to run for 7 days. To utilize the computation resource of the AWS server efficiently, we assign only 30 VMs for each server. In total, it takes us two months to gather the experiment results shown in this paper.

After 7 days of fuzz testing against various versions of the Linux kernels by using four different fuzzers, we asked the professional analyst team mentioned in Section 2.3 to collect the fuzzing results (i.e., reports) from all VMs, group the reports based on their title uniqueness, and eventually preserve only the kernel reports truly tied to the 60 bugs of our selection. Note that a kernel fuzzer might trigger other kernel bugs and thus demonstrate errors. To ensure the newly identified error behaviors are truly tied to the bug of our interest, error triaging is needed.

As is illustrated in Figure 5, We combine automatic and manual methods together to triage the newly generated bug reports. If the selected bug has a patch, we will then patch the kernel and run the new PoC. If this new PoC cannot

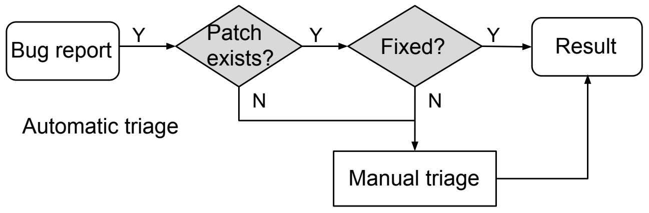


Fig. 5: The procedure of dealing with new error report.

trigger the same error after patching, we assume the new bug report or error behavior is also caused by the selected bug. Otherwise, this new PoC might trigger another bug. The rationale behind is that the patch can correctly fix the underlying kernel bug, preventing the errors triggered by the new PoC from occurring again. It is noteworthy that the above process is fully automatic. However, if any issues (*e.g.,* reproduction failure, compilation issue) occur in the above automation or the selected bug does not have a patch, we will fall back to the manual triage procedure. The team manually examines the bug patch and extracts the triggering condition. With this triggering condition in hand, the team further examines the execution of the PoC program. If the execution aligns with the triggering condition, the team safely concludes the newly discovered error is tied to the bug of our interest. To minimize the possible human mistake, we ask the whole team to form a unanimous agreement before we associate that new error behavior with the bug of our interest.

Furthermore, we also asked our kernel professional analyst team to thoroughly and manually inspect whether there are any other missing paths or contexts that could trigger the kernel bugs and thus exhibit different error behaviors. In this way, we can evaluate GREBE's false negatives, in other words, understand how complete GREBE could expose a bug's multiple error behaviors. Since the Linux kernel's codebase is huge and sophisticated given a kernel bug, it usually requires extensive manual efforts and significant expertise, spending hundreds of hours to perform through manual analysis for exploring all the possible errors. As a result, we evaluate the false negatives of GREBE by sampling 30% of the selected kernel bugs (18 out of 60 selected bugs).

6.2 Experiment Results

Effectiveness. Table 5 shows the sampled experiment results[4]. First, we can observe that our tool – GREBE could demonstrate a significant advantage in finding a bug's multiple error behaviors. In comparison with Syzkaller and Syzkaller variant, which discover a total of 9 additional error behaviors for only 6 and 7 test cases within 7 days, GREBE identifies 132 new error behaviors for 38 out of 60 test cases. Second, we can observe the mutation optimization greatly improves GREBE's utility. In 7 days of our experiment, GREBE without mutation optimization pinpoints 58 new error behaviors for 27 cases and significantly outperforms that of Syzkaller. However, GREBE without mutation optimization still experiences more than 50% of a downgrade in terms of the newly identified error behaviors (132 vs. 58)

3. At the time of our experiment, 5.10 is the latest long-term support Linux kernel version.

4. It should be noted that, due to the space limit, we place the complete experiment results at [40].

SYZ ID	Critical Structures Identified	Initial Error Behavior	Discovered New Error Behaviors	Time (in hours)			
				T1	T2	T3	T4
bdeea91 [37]	aead_instance, crypto_aead, crypto_spawn, pcrypt_instance_ctx crypto_aead_spawn, crypto_type	WARNING: refcount bug in crypto_mod_get	WARNING: refcount bug in crypto_destroy_tfm	6.69	2.62	0.06	1.25
			KASAN: use-after-free Read in crypto_alg_extsize	-	-	-	83.69
5d3cce3 [25]	napi_struct, tun_file	general protection fault in hrtimer_active	KASAN: use-after-free Read in free_netdev	-	-	155.76	30.30
			KASAN: use-after-free Read in netif_napi_add	-	-	77.41	9.08
521a764 [38]	ax25_address, nr_sock	WARNING: refcount bug in nr_insert_socket	KASAN: use-after-free Read in release_sock	-	-	0.03	4.39
			KASAN: use-after-free Read in nr_release	-	-	-	20.00
			KASAN: use-after-free Read in nr_insert_socket	-	-	-	0.06
			KASAN: use-after-free Write in nr_insert_socket	-	-	-	126.82
			KASAN: use-after-free Read in lock_sock_nested	-	-	-	18.20
229e0b7 [39]	delayed_uprobe	general protection fault in delayed_uprobe_remove	KASAN: use-after-free Read in delayed_uprobe_remove	-	-	3.83	6.66
			KASAN: use-after-free Read in uprobe_mmap	-	-	12.69	4.10
			general protection fault in uprobe_mmap	-	-	-	89.49
			KASAN: use-after-free Read in update_ref_ctr	-	-	-	157.46

TABLE 5: The performance of Syzkaller, Syzkaller variant, GREBE and GREBE without mutation optimization under some sampled kernel bugs. The "SYZ ID" column is the case ID. The "Critical Structures Identified" means the structures that are identified by the static analysis tools then are utilized by GREBE. The "Initial Error Behavior" column indicates the error behavior manifested in the corresponding bug report. The "Discovered New Error Behaviors" column is the error behaviors newly discovered. Note that, for each case, we sample only some of its newly identified error behaviors for illustration purposes. In the "Time" column, T1 represents the number of hours Syzkaller took, T2 is for Syzkaller's variant, T3 is for GREBE without optimization, and T4 stands for GREBE. The dash "-" means the corresponding error behavior is not discovered by the corresponding tool.

and about 30% of decrease in terms of the cases it could handle (38 vs. 27).

False Negatives. As is mentioned above, we also randomly selected 30% of test cases, performed manual analysis, and examined how complete GREBE could identify the error behaviors of a given kernel bug. Our manual inspection shows that GREBE misses one error behavior for the cases #d1baeb1, #85fd017 and #695527b, and two error behaviors for the case #d5222b3. To understand the reasons behind these missing error behaviors, we explore the conditions of triggering the missing error behaviors and found that, in addition to finding different paths and contexts by using GREBE, the exhibition of the missing behaviors also requires the manipulation of memory layout. For case like #85fd017, the manifestation of error behaviors depends on the layout of memory. The undiscovered error behavior occurs only if the memory in the overflowed region is unmapped. We do not attribute this to the incompetency of GREBE. Rather, we will leave the manipulation of thread interleaving and memory layout as part of our future research.

Impact of Popular Kernel Structure Removal. Recall that in Section 4.3, we rank the identified critical structures based on their popularity and avoid using popular structures to guide our kernel fuzzing. Intuition suggests this might influence the effectiveness of our kernel fuzzing on finding a bug's multiple error behaviors. However, from the 60 kernel bugs of our selection, we observe there are only 3 out of 60 test cases (5%) the root cause of which ties to popular structures (sk_buff for #d1baeb1, nlattr for #b36d7e4 and #27ae1ae). Even for these cases, GREBE still demonstrates its utility in finding the bugs' multiple error behaviors. These observations well align with our aforementioned arguments – ❶ the kernel bug generally roots in the inappropriate usage of less popular kernel structures, and ❷ focusing on less popular structures can still allow our fuzzer to reach out to popular structures because of the strong dependence between them. In Table 5, we list some kernel object types that GREBE uses for fuzzing guidance. For more complete kernel object types identified for each kernel bug, readers could find them at [40].

Efficiency. Table 5 and the table at [40] show the time that each fuzzer spent on finding a new kernel error behavior. First, we observe that both Syzkaller and its variant have comparable efficiency (21546 hours vs 21528 hours). However, GREBE without mutation optimization spends less time than Syzkaller on identifying the new error behavior (15011 vs. 21546 hours)[5]. After applying the mutation optimization, GREBE further reduces the time spent on new error behavior identification (5445 vs. 15011 hours). This discovery indicates mutation optimization alone provides minimum benefits to the improvement of fuzzing efficiency whereas object-driven component alone or the combination of both brings significant improvement in fuzzing efficiency.

Second, we observe that GREBE succeeds in disclosing 79 new error behaviors for 32 test cases within 24 hours. Take the case #5d3cce3 in Table 5 as an example. GREBE found the use-after-free read error in netif_napi_add in 9 hours. On the contrary, GREBE without mutation optimization spent more than 3 days. The original Syzkaller and its variant performed even worse, failing to find this error behavior within the 7-day time window. This result empirically shows that the design of object-driven fuzzing and mutation optimization in GREBE, to a large extent, can save the time and resources for the discovery of new error behaviors.

Contributing Factor Confirmation We manually analyze the newly identified error behaviors and figure out the effect of contributing factors. By manually inspecting the reasons for new error reports, we can confirm 72.1% of error behaviors are caused by input difference. And the rest new error reports are caused by thread interleaving,

5. Since the new error behaviors discovered by Syzkaller and its variant is too few compared with the other fuzzers, we conservatively use 7 days (7×24=168 hours) to represent the non-discovered error behaviors when computing the time.

SYZ ID	Exploitability Change	SYZ ID	Exploitability Change
d1baeb1	LL $\rightarrow$ L (2) $\star$	de28cb0	LL $\rightarrow$ L (5)
8eceaff	LL $\rightarrow$ L (2) $\star$	f56bbe6	LL $\rightarrow$ L (1)
bb7fa48	LL $\rightarrow$ L (1)	f0ec9a3	LL $\rightarrow$ L (1)
d767177	LL $\rightarrow$ L (2)	5d3cce3	LL $\rightarrow$ L (2) $\star$
460cc94	LL $\rightarrow$ L (1)	692a8c2	LL $\rightarrow$ L (12) $\star$
0df4c1a	LL $\rightarrow$ L (3)	4cf5ee7	LL $\rightarrow$ L (2)
229e0b7	LL $\rightarrow$ L (3)	502c872	LL $\rightarrow$ L (1)
163388d	LL $\rightarrow$ L (1)	b36d7e4	LL $\rightarrow$ L (1)
bdeea91	LL $\rightarrow$ L (1)	1fd1d44	LL $\rightarrow$ L (1)
b9b37a7	LL $\rightarrow$ L (4)	695527b	LL $\rightarrow$ L (1)
0d93140	LL $\rightarrow$ L (1)	85fd017	LL $\rightarrow$ L (4) $\star$
b0e30ab	LL $\rightarrow$ L (1)	6a03985	LL $\rightarrow$ L (3) $\star$
d5222b3	LL $\rightarrow$ L (1)	575a090	LL $\rightarrow$ L (1)
3a6c997	L $\rightarrow$ L (10)	27ae1ae	L $\rightarrow$ L (1)
cbb2898	L $\rightarrow$ L (1)	4bf11aa	L $\rightarrow$ L (1)
e4be308	L $\rightarrow$ L (11)	7022420	L $\rightarrow$ L (1)
3b7409f	L $\rightarrow$ L (1)	ddaf58b	L $\rightarrow$ L (2)

TABLE 6: The summary of exploitation potential improvement. In the column of "Exploitability Change", LL means the original error behavior is less likely to be exploitable. The letter L means the newly discovered error behaviors are likely to be exploitable. The number in the parenthesis represents the amount of newly identified error behaviors tied to probably exploitable. The star $\star$ denotes the bugs for which we have developed exploits based on the newly discovered error behaviors and their provided primitives.

memory dynamics, different sanitizers, inline function etc. The confirmation demonstrates GREBE can effectively leverage input difference to explore multiple error behaviors of kernel bugs.

6.3 Security Implication

Exploitation Potential Exploration. Recall that we design `GREBE` to explore a kernel bug's multiple error behaviors. With the multiple manifested behaviors in hand, we expect some newly exposed error behaviors to indicate a higher exploitation potential for a kernel bug (e.g., finding an out-of-bound write error behavior for a kernel bug that originally manifests null pointer dereference).

In our dataset, we have 60 kernel bugs. The report of 44 bugs demonstrate less-likely-exploit behaviors, while other 16 bugs' reports expose errors tied to likely-to-exploit. As we can observe from Table 6, for 26 bugs (about 60% of 44 less-likely-to-exploit bugs), `GREBE` could find at least one likely-to-exploit error behavior which implies a higher exploitation potential. This observation indicates that `GREBE` can help security researchers better infer kernel bugs' exploitation potential.

By using `GREBE`, there are 8 bugs (50%) among the rest 16 kernel bugs originally tied to likely-to-exploit that manifest other likely-to-exploit error behaviors indicating further exploitation potential. Taking a closer look at the three cases `#` ↪ `e4be308`, `#3b7409f`, and `#ddaf58b`. Their original reports all indicate that the bug provides the ability to perform a write to an unauthorized memory region. However, the newly discovered error behaviors enable the adversaries to perform unauthorized read/write at different memory regions. Take the case `#3619dec5` for example. Its new error behavior can write data to the `kmalloc-64` from 56th to 60th bytes, whereas its error behavior shown in the report corrupts the first eight bytes of `kmalloc-64`. This enlarged memory access potentially diversifies the way to perform exploitation and bypass mitigation.

For the kernel bugs of our selection that do not show exploitation potential improvement (*i. e.*, 26 bugs = 60-26-8), we argue that this does not dilute the contribution of `GREBE`. First, based on the aforementioned small-scale evaluation on the false negatives of `GREBE`, it is very likely that all the possible error behaviors of these bugs are exposed. Second, although the exploitation potential remains unchanged, `GREBE` manages to find many other error behaviors (e.g., `#1fd1d44` in the table at [40]). These additional error behaviors and the corresponding fuzzing programs can potentially facilitate the root cause diagnosis, as is demonstrated in [7].

On average, for the selected bugs with only one error report, GREBE could find other error behaviors for 51.7% (10/29) of them. For the selected bugs with multiple error reports, GREBE found more likely-to-exploit error behaviors for 48.8% (20 out of 41) of them. This result shows GREBE could help us find more error behaviors and thus explore higher exploitation potential for kernel bugs.

Real-world Impact. For all the 44 kernel bugs (the original reports of which implies less-likely-to-exploit), by using `GREBE`, we can turn 26 of them from less-likely-to-exploit bugs to likely-to-exploit ones. For the 26 kernel bugs, we further explore their exploitability manually. We surprisingly discovered that 6 out of the 26 bugs (illustrated by a star sign in Table 6) could be turned into fully exploitable kernel vulnerabilities. Take case `#6a03985` as an example, its original error behavior is a `WARNING` implying less-likely-to-exploit. Using `GREBE`, we identified a use-after-free error behavior for this bug. Starting from this newly discovered error behavior and the primitive the error behavior provides, we successfully demonstrated the bug's exploitability, including leaking sensitive data (e.g., encryption key and hashed password), bypassing KASLR, and redirecting the kernel execution for privilege escalation. Recently, we have shared our working exploits with the corresponding vendors. Because the bug's original report implies less-likely-to-exploit, many vendors defer or completely ignore the adoption of the patch. Upon receiving our exploitation demonstration, they confirmed our findings, took immediate action to apply patches, and assigned us with CVE IDs.

7 Related Work

This section summarizes the works most relevant to ours.

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.* [12] 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 [21] 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.* [24]

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 [41] 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.* [42] analyzed crowd-reported vulnerability reports to assess their reproducibility. Our work is the first to study the factors causing multiple error behaviors for kernel bugs. In addition to the empirical study, we also provide a fuzzing technique to explore all possible error behaviors manifested by one bug.

Kernel fuzzing. Syzkaller [5] is a popular code-coverage-based kernel fuzzer. While doing fuzzing, it leverages templates to specify the dependency between system calls and the expected value range of system calls' arguments. However, with only explicit dependencies between system calls, it is not enough to produce a high-quality fuzzing program because the OS kernel is a massive system with a complicated internal state transition. IMF [43] optimizes kernel fuzzing by tracking the system calls and analyzing them coordinately with type information to infer the kernel system's internal states. This approach, unfortunately, has the limitation of extracting internal dependencies inside the kernel. As such, taking a step ahead, Moonshine [36] leverages lightweight static analysis to detect internal dependencies across different system calls from system call traces of real-world programs. Recently, HFL [27] introduces hybrid fuzzing to the kernel, performing point-to analysis, and symbolic checking to figure out precise constraints between system state variables. To support closed-source kernel, instead of relying on the kernel interface to collect code coverage, kAFL [6] proposes a fuzzing framework that employs a hardware-assisted code coverage measurement. Although the kernel fuzzers above demonstrate effectiveness in finding kernel bugs, like Syzkaller, their design inevitably fails multiple error behavior exploration simply because they rely on code coverage to guide kernel fuzzing tasks, making our task inefficient. In this work, GREBE introduces a new design that utilizes critical kernel objects to improve effectiveness and efficiency for multiple error behavior exploration.

Apart from kernel fuzzers aiming to find various types of bugs in the entire system, there are works focusing on specific kernel modules or bug types. DIFUZE [44] uses static analysis to effectively fuzz device drivers in the Android kernel. Periscope [45] fuzzes a device driver not via system call interfaces but by mutating input space over I/O bus. Razzer [28] combines static and dynamic testing to reach program sites where race condition bugs may exist. KRACE [46] further customizes to find race condition bugs in the file system. While they demonstrate their utility in hunting bugs in specific kernel modules, it is difficult to generalize these techniques to explore kernel bugs' error behaviors. SemFuzz [47] is the only work that aims to trigger a known kernel bug through kernel fuzzing to the best of our knowledge. However, this technique is not designed to diversify the paths and contexts for triggering the bug but simply to enable bug reproduction. Therefore, it is not suitable for the problem we address.

Exploitability assessment. Automating exploit development can also facilitate exploitability assessment. Existing exploitability assessment works are mainly in three directions. The first direction is to obtain exploitable primitives. Xu et al. [48] exploit use-after-free vulnerabilities using two memory collision mechanisms to perform heap spray in the kernel. SLAKE [49] facilitates the exploitation of slab-based vulnerabilities by first building a database of kernel objects and then systematically manipulating slab layout using the kernel objects in the database. Lu et al. [50] exploits use-before-initialization vulnerabilities using deterministic stack spraying and reliable exhaustive memory spraying. As a follow-up work, Cho et al. [51] further propose to use BPF functionality in the kernel for stack spraying. The second direction is to bypass mitigations in the kernel. For example, ret2dir [52] takes advantage of physical memory which is mapped to kernel space for payload injection. KEPLER [53] leverages communication channels between kernel space and user space (e.g., `copy_from/to_user`) to leak stack canary and inject ROP payload to kernel stack. ELOISE [34] bypasses KASLR and heap cookie protectors using a special but pervasive type of structure. The third direction is to explore the capability of vulnerabilities, which is most related to our work. In this direction, FUZE [54] explores new use sites for use-after-free vulnerabilities using under-context fuzzing and identifies exploitable primitives implied by the new use sites using symbolic execution. KOOBE [55] extracts capabilities of a slab-out-of-bound access vulnerability manifested in the PoC program and uncovers hidden capabilities using capability-guided fuzzing. The techniques developed in both works are customized to the characteristics of a specific vulnerability type and are difficult to generalize to others. Besides, they require to manually diagnose root cause of the bug while GREBE does not. Moreover, they cannot explore possible error behaviors for a single bug, which is the main contribution of GREBE.

8 Conclusion and Future Work

With only the error behavior in a bug report, security analysts might underestimate the severity of the underlying kernel bug, since the bug could manifest multiple error behaviors indicating different exploitability. Therefore, we explore multiple error behaviors of the same bug to correctly assess the exploitability. Through an empirical study, we discover the prevalence of multiple error behaviors and more error behaviors help unveil the real exploitability. Through intensive manual analysis, we identify the factors contributing to multiple error behaviors with different exploitation potentials. Under the guidance of our empirical study, we design an object-driven kernel fuzzing mechanism. With our proposed technique, security analysts could explore more error behaviors of kernel bugs. The newly identified error behaviors might have higher exploitation potential than the one shown in the original report. It indicates the bug's exploitability escalation. As such, we safely conclude, given a kernel bug, the object-driven kernel fuzzing method could leverage the contributing factors and help security analysts better understand and infer exploitability for a given kernel bug.

Our future work will focus on other contributing factors to expose more error behaviors. For thread interleaving, we will extract some representative information of underlying kernel bugs (*e.g.*, shared resources, synchronization variables). Motivated by the utility of the object-driven fuzzing approach, we would design a directed fuzzing mechanism that helps expose multiple error behaviors caused by the same thread interleaving. At the same time, we will mainly explore the proposed method against the bugs on other kernels (*e.g.*, XNU, FreeBSD).

Acknowledgements

We thank the anonymous reviewers for their valuable suggestions and feedback. This project was supported in part by the 2020 IBM Ph.D. Fellowship, 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] Z. Lin, Y. Chen, Y. Wu, D. Mu, C. Yu, X. Xing, and K. Li, "GREBE: Unveiling exploitation potential for linux kernel bugs," in *Proceedings of the 43th IEEE Symposium on Security and Privacy (S&P)*, 2022.
[2] S. Team, "!exploitable crash analyzer version 1.6," 2013, https://www.microsoft.com/en-us/security/blog/2013/06/13/exploitable-crash-analyzer-version-1-6/.
[3] B. J. Wever, "Bugid - automated bug analysis," 2017, https://prezi.com/caic9eqayy-o/bugid-automated-bug-analysis/.
[4] J. Vanegue, "In memory safety, the soundness of attacks is what matters," 2020, https://openwall.info/wiki/_media/people/jvanegue/files/soundness_of_attacks.pdf.
[5] D. Vyukov, "Syzkaller," 2020, https://github.com/google/syzkaller.
[6] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-assisted feedback fuzzing for os kernels," in *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*, 2019.
[7] 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 *Proceeding of the 28th USENIX Security Symposium (USENIX Security)*, 2020.
[8] Google, "syzbot," 2020, https://syzkaller.appspot.com.
[9] ——, "KernelAddressSanitizer, a fast memory error detector for the linux kernel," 2020, https://github.com/google/kasan.
[10] 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.
[11] ClusterFuzz, "Crash type in clusterfuzz," 2019, https://google.github.io/clusterfuzz/reference/glossary/#crash-type.
[12] 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 (ASE '14)*, 2014.
[13] 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, pp. 575–589.
[14] 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, pp. 734–748.
[15] 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.
[16] 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.
[17] 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.
[18] 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.
[19] 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.
[20] 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, pp. 545–559.
[21] 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.
[22] 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.
[23] 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.
[24] 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.
[25] syzbot, "general protection fault in hrtimer_active," 2017, https://syzkaller.appspot.com/bug?id=5d3cce34cc09f722e859ae2037801f5b0d67c5c9.
[26] ——, "Kasan: use-after-free read in free_netdev," 2017, https://syzkaller.appspot.com/bug?id=e99ffcb23d080ae2c4790dfc229d32ce283f826f.
[27] 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 2020 Network and Distributed System Security Symposium (NDSS)*, 2020, pp. 96–112.
[28] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P)*, 2019.
[29] S. Gong, D. Altinbüken, P. Fonseca, and P. Maniatis, "Snowboard: Finding kernel concurrency bugs through systematic inter-thread communication analysis," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21, 2021.
[30] H. Chen, S. Guo, Y. Xue, Y. Sui, C. Zhang, Y. Li, H. Wang, and Y. Liu, "Muzz: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs," in *Proceedings of the 29th USENIX Conference on Security Symposium*, ser. SEC'20, 2020.
[31] "Linux kernel design patterns - part 2," https://lwn.net/Articles/336255/, 2009.
[32] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," in *Proceedings of the Seventh International Conference on World Wide Web 7*, ser. WWW7, 1998.
[33] K. Lu and H. Hu, "Where Does It Go? Refining indirect-call targets with multi-layer type analysis," in *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.
[34] Y. Chen, Z. Lin, and X. Xing, "A systematic study of elastic objects in kernel exploitation," in *Proceedings of the 27th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020.
[35] I. Ben-Gal, "Outlier detection," in *Data mining and knowledge discovery handbook*. Springer, 2005, pp. 131–146.
[36] S. Pailoor, A. Aday, and S. Jana, "MoonShine: Optimizing os fuzzer seed selection with trace distillation," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[37] syzbot, "WARNING: refcount bug in crypto_mod_get," 2020, https://syzkaller.appspot.com/bug?id=bdeea91ae259b3a42aa8ed8d8c91afd871eb5d80.

[38] ——, "WARNING: refcount bug in nr_insert_socket," 2019, https://syzkaller.appspot.com/bug?id=521a764b3fc8145496efa50600dfe2a67e49b90b.

[39] ——, "general protection fault in delayed_uprobe_remove," 2019, https://syzkaller.appspot.com/bug?id=229e0b718232b004dfddaeac61d8d66990ed247a.

[40] "Full performance results of syzkaller, syzkaller variant, grebe without mutation optimization and grebe," 2021, https://tinyurl.com/x9ky26ms.

[41] 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.

[42] 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.

[43] H. Han and S. K. Cha, "IMF: Inferred model-based fuzzer," in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[44] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna, "DIFUZE: Interface aware fuzzing for kernel drivers," in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[45] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz, "PeriScope: An effective probing and fuzzing framework for the hardware-os boundary," in *Proceedings of the 2019 Network and Distributed System Security Symposium (NDSS)*, 2019, pp. 481–495.

[46] M. Xu, S. Kashyap, H. Zhao, and T. Kim, "KRace: Data race fuzzing for kernel file systems," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P)*, 2020.

[47] W. You, P. Zong, K. Chen, X. Wang, X. Liao, P. Bian, and B. Liang, "SemFuzz: Semantics-based automatic generation of proof-of-concept exploits," in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[48] W. Xu, J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. Gu, "From Collision To Exploitation: Unleashing use-after-free vulnerabilities in linux kernel," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[49] Y. Chen and X. Xing, "SLAKE: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel," in *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[50] K. Lu, M.-T. Walter, D. Pfaff, and S. Nürnberger and Wenke Lee and Michael Backes, "Unleashing use-before-initialization vulnerabilities in the linux kernel using targeted stack spraying," in *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS)*, 2017, pp. 890–904.

[51] H. Cho, J. Park, J. Kang, T. Bao, R. Wang, Y. Shoshitaishvili, A. Doupé, and G.-J. Ahn, "Exploiting uses of uninitialized stack variables in linux kernels to leak kernel pointers," in *14th USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[52] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "ret2dir: Rethinking kernel isolation," in *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*, 2014.

[53] 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 Security Symposium (USENIX Security)*, 2019.

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

[55] 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 (USENIX Security)*, 2020.

Ziqin Liu Zinqin Liu received the B.E. degree in Hubei University of computer science and information engineering in 2018. She is currently pursuing the M.S. degree in the Department of Cyber Science and Engineering, Huazhong University of Science and Technology. Her current research interests include software and system security.

Zhenpeng Lin received his B.E. degree in cyber security from Xidian University in 2018. He is currently pursing the Ph.D. from Computer Science at the Northwestern University. His current research interest covers vulnerability exploitation and exploit mitigation.

Yueqi Chen receive his B.Sc degree from the Department of Computer Science and Technology at Nanjing University in 2017. He is currently pursing Ph.D. at the Pennsylvania State University. His research area is system and software security in general. He is especially interested in weird machine, exploitability escalation and formalization, and universal protection design for infrastructure cyber-systems.

Yuhang Wu received his B.E. degree in Computer Science school from Wuhan University in 2018. He is currently pursing the Ph.D. from Computer Science at the Northwestern University. His current research areas includes Linux kernel security and network security.

Yalong Zou received the B.E. degree in mechanical science and engineering from Huazhong University of Science and Technology in 2021. He is currently pursuing the M.S. degree in the School of Cyber Science and Engineering, Huazhong University of Science and Technology. His current research interests include software and system security.

Dongliang Mu is an Associate professor in the School of Cyber Science and Engineering, Huazhong University of Science and Technology. He received the Ph.D. degree in the Department of Computer Science and Technology, Nanjing University. His current research interests span the area of fuzzing, vulnerability reproduction, postmortem program analysis, vulnerability diagnosis, and binary analysis.

Xinyu Xing is an Associate Professor in the Computer Science at the Northwestern University. He earned his Ph.D. in Computer Science from Georgia Tech. His research area covers system security, binary analysis and deep learning.