GREBE: Unveiling Exploitation Potential for Linux Kernel Bugs

Zhenpeng Lin[††*], Yueqi Chen[††], Yuhang Wu[††], Dongliang Mu[†‖], Chensheng Yu[‡], Xinyu Xing[§‖], Kang li[¶]
[††]The Pennsylvania State University [†]School of Cyber Science and Engineering, HUST
[‡]George Washington University [§]Northwestern University [¶]Baidu USA
{zplin, ycx431, yuhang}@psu.edu, dzm91@hust.edu.cn,
i@shiki7.me, xinyu.xing@northwestern.edu, kangli01@baidu.com

***Abstract*—Nowadays, dynamic testing tools have significantly expedited the discovery of bugs in the Linux kernel. When unveiling kernel bugs, they automatically generate reports, specifying the errors the Linux encounters. The error in the report implies the possible exploitability of the corresponding kernel bug. As a result, many security analysts use the manifested error to infer a bug's exploitability and thus prioritize their exploit development effort. However, using the error in the report, security researchers might underestimate a bug's exploitability. The error exhibited in the report may depend upon how the bug is triggered. Through different paths or under different contexts, a bug may manifest various error behaviors implying very different exploitation potentials. This work proposes a new kernel fuzzing technique to explore all the possible error behaviors that a kernel bug might bring about. Unlike conventional kernel fuzzing techniques concentrating on kernel code coverage, our fuzzing technique is more directed towards the buggy code fragment. It introduces an object-driven kernel fuzzing technique to explore various contexts and paths to trigger the reported bug, making the bug manifest various error behaviors. With the newly demonstrated errors, security researchers could better infer a bug's possible exploitability. To evaluate our proposed technique's effectiveness, efficiency, and impact, we implement our fuzzing technique as a tool `GREBE` and apply it to 60 real-world Linux kernel bugs. On average, `GREBE` could manifest 2+ additional error behaviors for each of the kernel bugs. For 26 kernel bugs, `GREBE` discovers higher exploitation potential. We report to kernel vendors some of the bugs – the exploitability of which was wrongly assessed and the corresponding patch has not yet been carefully applied – resulting in their rapid patch adoption.**

I. 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/sanitization features. 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, both our survey result (shown in A) and previous research [1], [2], [3] indicate that the manifested error behaviors of bugs play a critical role in prioritizing exploit development efforts.

In practice, when existing fuzzing tools identify a kernel bug, the manifested error behavior of the bug may be one of its many possible error behaviors. Its other possible error behaviors could be far away from the one that has been already exposed. For example, as we will elaborate in Section II, by following different paths or execution contexts to trigger a kernel bug, we can make a kernel bug 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, it could be misleading if security analysts only use singly manifested error behavior to infer the bug's possible exploitability.

In order to address this problem, one instinctive reaction is to take as input a kernel bug report, analyze the root cause of that kernel bug, and infer all possible consequences that the root cause of the bug could potentially bring about (e.g., out-of-bound memory violation, null pointer dereference, and memory leak, etc.). However, root cause diagnosis is typically considered a time-consuming and labor-intensive task. As a result, we argue that a more realistic strategy for tackling this problem is to expose many possible post-triggered error behaviors of a given kernel bug without performing root cause analysis. Then, from the error behaviors unveiled, security analysts could better infer its possible exploitability in a more accurate fashion.

To realize the idea above, we can borrow the concept of kernel fuzzing. However, existing kernel fuzzing methods are mainly designed to maximize the code coverage (e.g., Syzkaller [4], KAFL [5] and Trinity [6], etc.). Using them in our task inevitably suffers from inefficiency and ineffectiveness issues, it is simply because code-coverage-driven kernel fuzzing is not customized nor optimized for finding various paths or 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 one kernel bug 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 structures/objects as feedback

*Part of the work was done during the summer internship at Baidu USA
‖Corresponding author

to the fuzzer. Since the identified kernel structures/objects are essential to the success in 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 a kernel-object-driven fuzzing tool and name it after GREBE, standing for "fuzzin**G** fo**R** multipl**E** **B**ehavior **E**xploration".

Using our tool to explore error behaviors for 60 kernel bug reports, we show that GREBE could demonstrate more than 2 different error behaviors on average for each given bug report. For many kernel bugs in our experiment (26 out of 60), we also observe that their newly identified error behaviors usually demonstrate a higher exploitation potential than those listed in the original bug report. 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, kernel warning, etc.) 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 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 design a new technical approach that utilizes carefully selected kernel objects to guide kernel fuzzing and thus explores a bug's multiple error behaviors.
- Following our design, we extend Syzkaller, implement GREBE, and demonstrate its utility in finding multiple error behaviors for 60 distinct real-world Linux kernel bugs.
- We show that given a kernel bug demonstrating only a low possibility to exploit, our proposed method could find its other error behaviors indicating much stronger exploitability.

II. Motivating Example

A reported kernel error implies the potential exploitability of the corresponding bug. As we mentioned above, the manifested error depends upon how the bug is triggered. As such, using a single bug report (exhibiting one error behavior) could possibly bring about the underestimation of that bug's potential exploitability. In Listing 1, we show a concrete example to illustrate this issue.

As is depicted in the list, the function `tun_attach` is responsible for configuring the network interface. Its argument `tun` ↪ refers to a global variable shared by all the tun files in the opened state. As is shown in line 3, if `IFF_NAPI` is set

```
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 the Linux kernel with a bug. 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.

in `tun->flags`, the kernel will initialize a timer and link the corresponding `napi` to the list of the network device `napi_list` ↪ . In line 12, another function `tun_detach` is responsible for cleaning up the data enclosed in `tun_file` as well as closing the file. If `IFF_NAPI` is set, the kernel will cancel the timer and remove the `napi` from `napi_list` of the device. In line 24, the function `free_netdev` will go through the `napi_list` to delete `napi` in the list.

The kernel bug results from the potential inconsistent state of the flag `tun->flags` in `tun_attach` and `tun_detach`. Take for example the kernel bug report [8] generated by Syzkaller. The PoC program attached to the report shows that a system call invokes `tun_attach` with `IFF_NAPI` unset. In this way, the kernel neither initializes the timer nor adds the corresponding `napi` to the list. Following this setup, the PoC program further invokes the system call `ioctl` to set `IFF_NAPI` in `tun->flags` before calling to `tun_detach`, which causes inconsistent flags between `tun_attach` and `tun_detach`. Then, in line 17, the kernel attempts to stop the timer, which dereferences a pointer enclosed in the timer object in `tun_detach`. However, as is mentioned above, the timer is not initialized in `tun_attach`, which results in a general protection fault. The general protection fault implies accessing storage that is not designated for use. Therefore, based on this single observation, many security analysts may infer the bug is probably unexploitable.

However, after closely looking at this bug, we realize that, by varying the PoC program and thus modifying the way to assign inconsistent value for the shared variable, we can have the kernel demonstrate a use-after-free error. To be specific, we can set `tun->flags` with `IFF_NAPI` before invoking the function `tun_attach`. In this way, after `tun_attach` is called, it could

add the corresponding `tun_file` to the device list `napi_list` ↪ . Following this setup, we can further invoke `ioctl` to clear `tun_flags` and then call `tun_detach`. As is shown in Listing 1, the function `tun_detach` does not remove the corresponding `napi` from the list in line 18 ∼ 19, but frees it in line 21. Therefore, when traversing the device list, the KASAN-instrumented kernel will throw the use-after-free error. In comparison with the error shown in the report [8], instead of accessing an invalid kernel memory address that generates a general protection fault, this non-permitted access ties to a valid kernel memory address and eventually corrupts the kernel memory. Therefore, based on this use-after-free error, many analysts may consider the bug is probably exploitable.

III. Design Rationale & Overview

Given a kernel bug report demonstrating one particular error behavior, one instinctive reaction for exploring its other possible error behaviors is to utilize the concept of directed fuzzing, which explores paths to a program site of our interest. We can expect that, through some of the newly identified routes to the buggy code fragment, one could trigger the bug specified in the report again and observe new error behaviors. However, this approach is not likely to be effective.

First, to use directed fuzzing to expose multiple error behaviors, we need to identify the buggy code fragment (i.e., the root cause of the error), treat it as the point of interest, and feed it to the directed fuzzer. However, it is challenging to pinpoint the root cause of the kernel bug correctly and automatically. Incorrectly deeming a non-root-cause site as the site of the fuzzer's interest could even fail the fuzzer to trigger the bug, let alone finding multiple error behaviors of the bug.

Second, even if we can point out the root cause of the kernel bug and have a directed fuzzer repeatedly reaches out to the buggy code, it does not mean the kernel could manifest multiple error behaviors. In addition to following different paths to the buggy code, the exhibition of error behaviors also relies upon the context after the bug triggering. For example, in addition to following a specific path to the buggy code snippet, we also need a separate kernel thread to vary a global variable, diversifying the contexts needed for triggering the bug and demonstrating different errors. By design, directed fuzzing cannot vary the context after reaching out to the target code of its interest.

In response to the limitation of directed fuzzing, existing kernel fuzzing techniques could handle the aforementioned two problems in a better fashion. Kernel fuzzers like Syzkaller do not require the input of the root cause of a given bug. They simply vary system calls' sequences and their arguments and thus thoroughly test kernel code through different paths. Besides, it also introduces new system calls to vary the execution contexts. These characteristics complement the shortage of directed fuzzing. Unfortunately, as we will show in Section VI, this approach confronts extremely low efficiency and demonstrates poor effectiveness.

The design principle of existing kernel fuzzing techniques is to maximize the kernel code coverage, which avoids executing the code paths that have already been explored. However, to trigger the same bug and explore its other possible error behaviors, the fuzzer needs to execute the same buggy code snippets repeatedly and expects the kernel to run into the same buggy site in a different context. Therefore, as we will show in Section VI, the code-coverage-based kernel fuzzing method (like Syzkaller) has only little benefit for identifying multiple error behaviors of a single kernel bug.

In this work, we address this problem by extending an existing kernel fuzzing approach with kernel-object guidance. Based on our observation from many kernel bugs, we discover the root cause of a kernel bug usually results from two practices. One is the inappropriate usage of a kernel object, which further contributes to a kernel error (e.g., the aforementioned case assigning inconsistent flag value for a kernel object in `tun_struct` type). The other is an 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 (e.g., an unsanitized integer used as the offset of a kernel object, causing an out-of-bounds memory access). As such, guided by the objects relevant to the error specified in the bug report, we can have the kernel fuzzer away from those paths and contexts irrelevant to the bug and thus improve its efficiency significantly.

To realize the idea mentioned above, we design our technical approach as a multi-step procedure that combines static analysis and kernel fuzzing techniques. As is depicted in Figure 1, 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 motivating example 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. In this work, 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 the 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, it varies the PoC, improving the efficiency and effectiveness in a bug's multiple error behavior explorations. In the following section, we will discuss these techniques in detail.

IV. Technical Details

In this section, we elaborate on the technical details of our object-driven kernel fuzzing approach. First, we describe how to analyze a kernel bug report and identify critical structures (i.e., the structures involved in the corresponding kernel error). Second, we discuss how to filter out kernel structures to further improve kernel fuzzing efficiency in error behavior exploration. Finally, we discuss how to use the identified structures to design our object-driven kernel fuzzing mechanism.

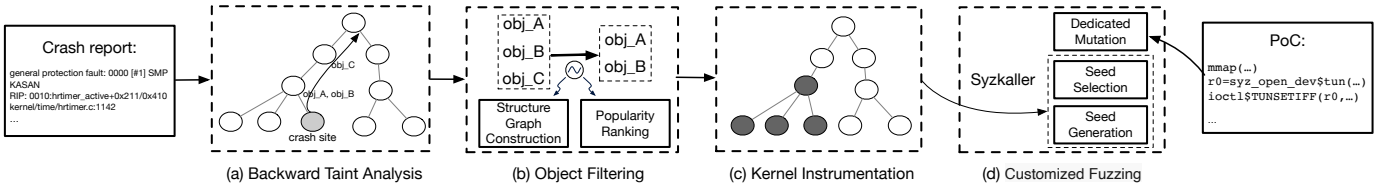

Fig. 1: The workflow of GREBE. (a) Following a kernel error trace obtained from a crash report, GREBE performs backward taint analysis and identifies all the kernel objects involved in the crash/panic. (b) Based on the objects' rareness, GREBE narrows down the objects critical to the kernel error. (c) 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.

```
1  // in drivers/vhost/vhost.c
2  void vhost_dev_cleanup(struct vhost_dev *dev)
3  {
4     WARN_ON(!list_empty(&dev->work_list));
5     if (dev->worker) {
6        kthread_stop(dev->worker);
7        dev->worker = NULL;
8        dev->kcov_handle = 0;
9     }
10 }
11 // in include/asm-genric/bug.h
12 #define WARN_ON(condition) ({      \
13   int __ret_warn_on = !!(condition);    \
14   if (unlikely(__ret_warn_on))     \
15     __WARN();      \
16   unlikely(__ret_warn_on);     \
17 })
```

Listing 2: 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
```

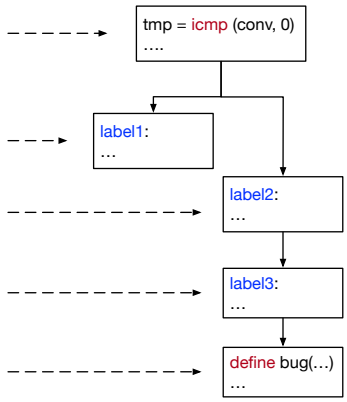

Fig. 2: An illustrating example and its dominator tree, which demonstrate two different methods of logging kernel errors. The line 7 is a logging statement responsible for kernel error recording. The line 15 is the wrapper of the logging statement at line 16. The variable `conv` in line 1 is the taint 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, the error logging cannot occur in this way.

A. 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.

1) Report Analysis & Taint Source Identification: The Linux kernel has a variety of debugging features implemented in different ways (e.g., `BUG`, `WARN`, and `KASAN`). However, most of them follow the same pattern. That is enforcing checks during the kernel execution and examining whether pre-defined conditions are satisfied. If the conditions do not hold, then the kernel runs into an error state and logs critical information for debugging purposes. Following the critical information logging, the kernel may take further action to panic itself or kill the current process.

Take, for example, the case shown in Listing 2. The function `vhost_dev_cleanup()` cleans the worker attached to the `vhost_dev` device. In line 4, the kernel examines the `work_list` ↪ . If the `WARN_ON` macro deems the list is empty, the kernel continues its execution at line 5, which performs the cleanup task. Otherwise, the kernel will execute the code in `WARN_ON` macro and log the error. In this example, the error is reported if and only if the pre-defined condition "`!list_empty(&dev->` ↪ `work_list)`" is true at runtime. Therefore, the variable `dev->` ↪ `work_list` in the condition indicates a cause of the bug and should become the starting point of our analysis (i.e., the taint source of our backward analysis). In this example, the kernel developers explicitly formulate the pre-defined condition as an expression and pass it to the macro `WARN_ON` for error handling. However, for some other debugging features, the checking is instrumented by a compiler or completed by hardware instead of a piece of source code written by kernel developers. For these features, the condition is implicitly formulated and cannot be identified from the kernel source code. In the following, we describe how we deal with various debugging features and their error logging mechanisms and thus identify the taint source.

Explicit Checking. Similar to the aforementioned example, the kernel developers explicitly formulate the checking as an expression and pass it to the standard debugging features such

```
1 // source code
2 walk->offset = sg->offset;
3
4 // pseudo binary code after instrumentation
5 kasan_check_read(&sg->offset, sizeof(var));
6 tmp = LOAD(&sg->offset, sizeof(var)); // first access
7 kasan_check_write(&walk->offset, sizeof(var));
8 STORE(tmp, &walk->offset); // second access
```

Listing 3: The code snippet that performs implicit checking.

as `WARN_ON` and `BUG_ON`. Inside these macros, it is a patterned code block that includes a condition statement and a logging statement that will be executed if the condition is satisfied. Apart from this standardized way to log kernel errors, the developers can also build their own macro that wraps a logging statement in a helper function (e.g., the code in the line 15 & 16 shown in Figure 2).

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 in Figure 2, the 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., the line 2 in Figure 2). Finally, we extract the corresponding variable in the condition as our taint source (e.g., `conv` in Figure 2).

Implicit Checking. Implicit checking refers to the situation where the checking is not part of the kernel source code but instrumented by a compiler or completed by hardware. For implicit checking done by compiler instrumentation, Kernel Address Sanitizer (KASAN) is such an example in which KASAN-enabled compiler instruments every memory access so that the kernel could examine whether the access to a memory address is legal. KASAN relies on shadow memory to record the memory status. If the instrumented kernel, for example, 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, from the debugging information, we map binary instructions with their corresponding statements in the source code. Suppose the mapped source code is a simple statement with only one load or store. In that case, 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 3), 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. To illustrate this, we again take, for example, the case shown in Listing 3. 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.

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 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. An example of this practice is some of the out-of-bound access errors in which the loop counter is corrupted, unexpectedly enlarged, and eventually used as an offset to reach out to an invalid memory region. 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.

B. Kernel Structure Ranking

By analyzing a kernel bug report and performing the backward taint analysis above, we can identify all the kernel structures pertaining to the error in the report. However, as we will discuss below, using the structures identified to guide kernel fuzzing and explore the bug's other error behaviors, we could still confront low efficiency and even poor effectiveness. As a

```
// definition of struct sk_buff
struct sk_buff {
    union {
        struct rb_node rbnode;
    };
    ...
    struct skb_ext *extensions;
};
```

Listing 4: The code snippet indicating structure definition.

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. To maintain the code quality, the Linux kernel developers employ plenty of design patterns [9]. These patterns provide a suggested practice and framework to manage data in a commonly recognized fashion. Take the double-linked list as an example. The `struct list_head` structure can be embedded anywhere in a data structure, and the `list_head` from many instances of that structure can be linked together. As a result, the kernel objects can be managed by standard interfaces, such as `container_of` which gets access to the parent for a given child structure, and `list_add/del` which performs list operations. The `struct list_head` is used pervasively in the entire kernel codebase. If including such popular structures and the corresponding objects for kernel fuzzing guidance, the kernel fuzzer would inevitably explore a large code space, driving the fuzzer away from its attention to the buggy code attributing to the kernel error specified in the report. Therefore, to preserve the kernel fuzzer's efficiency in exploring a bug's multiple behaviors, we need to exclude these structures from our kernel fuzzing guidance.

In addition to the structures mentioned above, Linux kernel developers also implement many other structures pertaining to abstract interfaces. These interfaces are coupled with implementation layers in support of a large number of devices and features. For example, the kernel creates a `struct socket` for all networking services requested from userspace no matter what protocol is specified. Such structures are also popular, appearing in many kernel code sites across various kernel modules. As a result, similar to `struct list_head`, they should also be eliminated from the later-stage kernel fuzzing.

The structures mentioned above are just examples of popular structures. To pinpoint and exclude them 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 [10] 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. For example, in Listing 4, `extensions` is a pointer referencing `struct skb_ext`. We link `struct sk_buff` to `struct skb_ext` in our graph. However, it should be noted that `struct rb_node` is a self-referenced structure in an anonymous union. Following the method above, we skip the anonymous union and link only `struct sk_buff` directly to `struct rb_node` without further expansion.

```
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 5: The code snippet indicating type casting.

In addition to analyzing the structure definition in kernel source code, we also construct the structure graph with the consideration of type casting. 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 5 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. They are more likely to be abstract data types. Besides, the structures referenced by popular structures can also be popular because they can also be used in many program sites in the kernel. For these structures, they are too prevalent to improve our kernel fuzzer's efficiency in better exploring the error behavior of a kernel bug. 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 V, we discuss how we choose the page-rank score threshold to distinguish popular structures from less popular ones.

Technical discussion. While the elimination of popular kernel structures narrows the focus of our kernel fuzzer, intuition suggests that it could also potentially restrict our kernel fuzzer from exploring other error behaviors for a given kernel bug. On the one hand, if the removed popular structures are the root

cause of the kernel bug, our fuzzer may no longer reach out to them and thus miss the opportunity to trigger the bug of our interest. On the other hand, if most of the kernel bugs' root causes are popular structures, our proposed technique might have only limited utility in helping a bug find its multiple error behaviors.

In this work, we argue the concerns above are not likely to be raised in the real world. First, based on our observation of hundreds of real-world kernel bugs, the root causes of most kernel bugs tie to less popular structures. As such, the removal of popular structures does not negatively influence the fuzzer in triggering the bug of our interest. Second, even if the eliminated popular structures are related to the root cause of our interest's kernel bug, having the fuzzer focused on those less popular structures can still allow us to reach out to some objects in the popular structural types. The reason is that less popular structures are usually composed of popular ones (e.g., the rare structure `struct napi_struct` contains the popular structure `struct hrtimer` in Listing 1). Paying attention to those less popular structures still provides opportunities to touch popular structures through fewer instances of these structures. In Section VI, we show several cases in which the corresponding bugs' root causes are relevant to those removed popular kernel structures. We demonstrate that our fuzzing approach can still trigger our interest's bugs and explore their other error behaviors even for those cases.

C. 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 a single 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 preserves this instrumentation ability and, further, introduces one additional instrumentation component. Our instrumentation component is designed as a compiler plugin. The plugin examines each statement in basic blocks and identifies those basic blocks that take the responsibility of the allocation, deallocation, and the usage of critical objects (i.e., the objects in the type of critical structures). More 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. With this instrumentation along with the inherited one, 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 instrumentation's facilitation above, when running a fuzzing program, we can easily determine whether it reaches out to a critical object. Once we identify a new critical object coverage, we can add the corresponding fuzzing program into the corpus of our seed fuzzing programs. In this work, we include the mutated seed program or the newly generated seed program into the seed corpus only if

```
1 r0 = openat(...,
2 '/dev/dsp1\x00');
3 ioctl(r0, ...);
4 write(r0, ...);
5 read(r0, ...);
```

(a) 7022420

```
1 // initial PoC: max = -1
2 bpf$MAP_CREATE(...,
3     @max=0xffffffffffffffff);
4 // exit triggers GFP
5 exit(0);
```

(b) 692a8c2

TABLE I: The example code snippets extracted from the PoC programs in two different kernel bug reports – `7022420` [11] and `692a8c2` [12].

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 previous fuzzing. It should be noted that we include the second condition because this allows kernel fuzzing to accumulate kernel states and thus increase the possibility for future mutations to reach out to unseen basic blocks involving critical objects.

Seed generation & mutation. In this work, we initialize the seed corpus with the PoC program enclosed in the bug report under our examination. Every time, when generating a new seed fuzzing program, we assemble the new fuzzing program by using only the system calls that have already been included in the seed corpus. It should be noted that this is very different from the seed fuzzing program generation method used in the state-of-the-art fuzzing technique (e.g., Syzkaller), which generates a seed fuzzing program by not only adopting the system calls enclosed in the corpus but also bringing 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 that 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`, as well as the system call `socket`, 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` or the system call `socket` to the system call `openat`. The mutation under template guidance ensures the seed program is legitimate and thus avoids the kernel's early rejection against the fuzzing program.

As is mentioned above, our mutation mechanism borrows the method used in Syzkaller. As we will show in Section VI, 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 expose multiple behaviors for a single kernel bug. As we elaborate below, the reasons behind this are two folds.

First, while performing a fuzzing program mutation, the Syzkaller attempts to introduce various system calls relevant to the seed program and randomly manipulate system calls' arguments. However, we note that both the resource and arguments that system calls operate are necessary for successfully triggering a target kernel bug. Mutation without the consideration of these two factors would inevitably incur low effectiveness in exploring multiple error behaviors.

Take the case depicted in Table Ia as the first example. The table shows a code snippet indicating a PoC program that triggers a kernel bug [11]. Taking this PoC as a seed program and performing mutation, Syzkaller inserts the system call `socket` which is irrelevant to the bug. This change would inevitably involve the resource that cannot lead to the bug's triggering and guide the fuzzer to enter a large code space.

Take the case shown in Table Ib as the second example. Similar to the example above, the table shows part of a PoC program triggering a different kernel bug [12]. In the mutation stage, Syzkaller varies the variable `@max=0xffffffffffffffff` because the template indicates that the legit value range for this variable is `[INT_MIN, INT_MAX]`. However, for this specific kernel bug, which triggers an integer overflow in the kernel, the condition of triggering this bug is `@max=-1` or in other words `@max=0xffffffffffffffff`. As a result, this argument's random mutation is futile, significantly influencing the triggering of the bug in different contexts.

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, when mutating a seed program, 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. We will release our newly grouped templates and the source code of this project after the submission is accepted.

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 (e.g., `AF_INET` and `AF_INET6` ↪ in the system call `socket()` indicating the establishment of the IPv4 and IPv6 socket, respectively). 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 types and belonging to resources, when the kernel fuzzing changes the context or path toward the buggy kernel code, the original PoC program's addresses could be illegal. Retaining these addresses could 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.

V. 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. The source code of `GREBE` is available at [13].

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 [14], [15] 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` that 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 is 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 IV-B, 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 5 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 [16]. 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 IV-C, 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 versions of the Linux kernel, 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 [17] 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 IV-C, they are not likely to help explore new paths to the buggy code.

VI. 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.

A. Experiment Setup & Design

Syzbot is a bug reporting platform that well archives the kernel bugs identified by Syzkaller. To evaluate our tool – GREBE, we select kernel bugs and their reports from the platform as our test cases. While selecting these bugs, we follow two different 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 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 bug selection strategy is a process dedicated to different kernel versions. To be specific, we select bugs from 5 different Linux kernel versions (5.6 5.10)[1]. From 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 kernel bugs in the first corpus, we obtain a dataset with 60 unique kernel bugs. To the best of our knowledge, our dataset is the largest used in the exploitability research.

For each bug in our dataset, we built the corresponding kernel in four QEMU virtual machines (VMs) by following the description of their bug reports. 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). With this setup, we can evaluate GREBE's effectiveness and efficiency in different settings. 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[2]. It should also be noted that we extend Syzkaller with our proposed mutation optimization for the following reason. GREBE is an extension of Syzkaller. It combines both the object-driven component and mutation optimization. With our mutation optimization integrated into Syzkaller alone, we could examine whether mutation optimization could become a sole driving force to enable multiple error behavior exploration.

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

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

[2]The State-of-the-art kernel fuzzing tools – HFL [18], SemFuzz [19] have not yet been publicly released. DIFUZE [20], KRACE [21], and Razzer [22] etc. are designed for fuzzing specific bug types or kernel modules. Previous research [18] shows Syzkaller has better performance than KAFL [5] and Trinity [6] in terms of code coverage. Therefore, we choose MoonShine-enhanced Syzkaller as our baseline.

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 formed a 6-member team under the guidance of an IRB approval (STUDY00008566). Among the 6 members, 2 are experienced security analysts regularly developing kernel exploits in the security industry. The other 4 members are academic researchers actively contributing to the Linux community and frequently invited to give talks at the Linux Security Submit or other Linux-related conferences. In our evaluation, we asked this professional team 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. Since there have not yet been highly accurate crash triaging tools, the professional team inspects each of the kernel errors manually and preserves only the errors associated with our selected bugs. The procedure of manually triaging the kernel errors is described in Appendix B.

In addition to the manual effort above, we also asked our kernel professional 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 or, in other words, understand how complete GREBE could expose a bug's multiple error behaviors. It should be noted that 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).

B. Experiment Results

Effectiveness. Table II shows the sampled experiment results[3]. 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. These kernel error behaviors have not been seen in the bug reports that we gathered from Syzbot. 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. This result significantly outperforms that of Syzkaller. However, without mutation optimization, GREBE experiences more than 50% of a downgrade in terms of the newly identified error behaviors (132 vs. 58) and about 30% of decrease in terms of the cases it could handle (38 vs. 27). Third, we discover that, while generally performing worse than GREBE, GREBE without enabling mutation optimization sometimes demonstrates better performance. For the test cases – #8eceaff, #3b7409f, and #d5222b3, GREBE without mutation optimization tracks down 4 additional error behaviors. We argue this does not imply the ineffectiveness of our mutation optimization method. Our manual inspection indicates the missing error behaviors primarily result from the nature of these bugs. Even if our mutation mechanism successfully constructs correct inputs to trigger the bug, making the bug manifest a different error behavior also relies upon a specific thread interleaving that mutation-optimization-disabled solution luckily discovers.

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. In our experiment, the test cases used for our false negative study are listed in Table V in Appendix. 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 first measure the number of basic blocks between the root cause of a kernel bug and its error panic site. We hypothesize that the false negative might relate to the distance between the root cause and the error site. However, as is shown in Table V, we did not find clear correlation between the distance and the effectiveness of GREBE. For more detail about the measurement and hypothesis validation, readers could refer to Appendix C.

With the rejected hypothesis in hand, we further took a look at false-negative cases closely, exploring the conditions of triggering the missing error behaviors. We found that, in addition to finding different paths and contexts by using GREBE, the exhibition of the missing behaviors also requires the manipulation of thread interleaving. 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 IV-B, 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

[3]It should be noted that, due to the space limit, we place the complete experiment results at [26].

SYZ ID	Critical Structures Identified	Initial Error Behavior	Discovered New Error Behaviors	Time (in hours)			
				T1	T2	T3	T4
bdeea91[23]	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[8]	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[24]	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[25]	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 II: 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. For more complete performance information across all 60 selected kernel bugs, the readers could find at [26]. 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.

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 II, 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 [26].

Efficiency. Table II and the table at [26] 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)[4]. 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 II 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.

[4]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 [27]	LL → L (2) ⋆	de28cb0 [28]	LL → L (5)
8eceaff [29]	LL → L (2) ⋆	f56bbe6 [30]	LL → L (1)
bb7fa48 [31]	LL → L (1)	f0ec9a3 [32]	LL → L (1)
d767177 [33]	LL → L (2)	5d3cce3 [8]	LL → L (2) ⋆
460cc94 [34]	LL → L (1)	692a8c2 [12]	LL → L (12) ⋆
0df4c1a [35]	LL → L (3)	4cf5ee7 [36]	LL → L (2)
229e0b7 [25]	LL → L (3)	502c872 [37]	LL → L (1)
163388d [38]	LL → L (1)	b36d7e4 [39]	LL → L (1)
bdeea91 [23]	LL → L (1)	1fd1d44 [40]	LL → L (1)
b9b37a7 [41]	LL → L (4)	695527b [42]	LL → L (1)
0d93140 [43]	LL → L (1)	85fd017 [44]	LL → L (4) ⋆
b0e30ab [45]	LL → L (1)	6a03985 [46]	LL → L (3) ⋆
d5222b3 [47]	LL → L (1)	575a090 [48]	LL → L (1)
3a6c997 [49]	L → L (10)	27ae1ae [50]	L → L (1)
cbb2898 [51]	L → L (1)	4bf11aa [52]	L → L (1)
e4be308 [53]	L → L (11)	7022420 [11]	L → L (1)
3b7409f [54]	L → L (1)	ddaf58b [55]	L → L (2)

TABLE III: 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 ⋆ denotes the bugs for which we have developed exploits based on the newly discovered error behaviors and their provided primitives.

C. Security Implication

Exploitation Potential Exploration. Recall that we design GREBE to explore a kernel bug's multiple error behaviors.

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

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 less-likely-to-exploit error behavior – null pointer dereference). As a result, we further evaluate GREBE's capability in exploitation potential exploration. To do this, we first recruited 20+ security researchers and conducted a user study (detailed in Appendix A) under the approved IRB (STUDY00008566). From the user-study results, we obtain the relationship between a manifested error behavior and the exploitation potential. As is depicted in Table IV, each error behavior is categorized into either "likely to exploit" or "less likely to exploit". Using this error-behavior-to-exploitability mapping obtained from security researchers, we then compare our newly identified error behaviors with those specified in their original bug reports.

In our dataset, we have 60 Linux kernel bugs. For 44 bugs, their reports gathered from Syzbot demonstrate error behaviors associated with less-likely-to-exploit. For the other 16 kernel bugs, their reports expose errors tied to likely-to-exploit. As we can observe from Table III, for 26 bugs (about 60% of 44 less-likely-to-exploit bugs), GREBE could find at least one likely-to-exploit error behavior. From that newly identified error behavior, one could imply a higher exploitation potential. This observation indicates that GREBE can help security researchers better infer kernel bugs' exploitation potential.

Among the rest 16 kernel bugs originally tied to likely-to-exploit, there are 8 bugs (50%). By using GREBE, one can identify their other likely-to-exploit error behaviors. We argue, this does not mean that GREBE has no utility for these kernel bugs. Taking a closer look at the three cases #e4be308 ↪ , #3b7409f, and #ddaf58b. Their original reports all indicate that the bug provides an 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. In this situation, there are fewer chances for a security researcher to find unknown error behaviors indicating a higher exploitation potential. Second, although the exploitation potential remains unchanged, GREBE manages to find many other error behaviors (e.g., #1fd1d44 in the table at [26]). These additional error behaviors and the corresponding fuzzing programs can potentially facilitate the root cause diagnosis, as is demonstrated in [7].

Real-world impact. For all the 44 kernel bugs (the original reports of which implies less-likely-to-exploit), we performed an exhaustive search and found no work demonstrating their exploitability in the past. As described above, using GREBE, we can turn 26 of them from less-likely-to-exploit to likely-to-exploit. For these 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 III) could be turned into fully exploitable kernel vulnerabilities. Take the case #6a03985 as an example. Its error behavior initially reported by Syzkaller 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. In [56], we release one of our exploits temporarily only for the paper review. We will publicly release all 6 exploits (tied to the cases #d1baeb1, #6a03985, #5d3cce3, #8eceaf, #85fd017 and #692a8c2) after the paper is accepted.

VII. Related Work

This section summarizes the works most relevant to ours.

Kernel fuzzing. Syzkaller [4] and Trinity [6] are two popular code-coverage-based kernel fuzzers. While doing fuzzing, they use 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 [57] 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 [17] leverages light-weight static analysis to detect internal dependencies across different system calls from system call traces of real-world programs. Recently, HFL [18] 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 [5] 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 [20] uses static analysis to effectively fuzz device drivers in the Android kernel. Periscope [58] fuzzes a device driver not via system call interfaces but mutating input space over I/O bus. Razzer [22] combines static and dynamic testing to reach program sites where race condition bugs may exist. KRACE [21] 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 [19] 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.

For user space programs, Brumley et al. [59], [60] used preconditioned symbolic execution to generate exploits for stack overflow and format string vulnerabilities. Bao et al. [61] recently proposed shellcode layout remediation and path kneading approaches to transplant existing shellcode. The Shellphish team developed PovFuzzer and Rex to turn a crash to an exploit [62], [63], [64]. Heelan et al. focus on heap buffer overflow vulnerabilities in user space programs. In [65], they use regression test to learn how to automate heap layout manipulation so that one could corrupt the sensitive pointers. In [66], they further improve their proposed approach by using a genetic algorithm to replace the random search algorithm for exploiting heap overflow vulnerabilities in language interpreters. Sharing the similar goals with the works [65], [66], Revery [67] also explores exploitable memory layouts for vulnerabilities in userspace programs. It utilizes fuzz testing along with a program synthesis method to guide the construction of a working exploit. Insu et al. [68] discovers new exploitation primitives in the heap allocator. They provide heap operations and attack capabilities as actions, driving the heap allocator to execute until primitives such as arbitrary write or overlapped chunks are identified. Unlike the works summarized above, GREBE focuses on a bug's exploitability assessment in the kernel space which is naturally more sophisticated than userland programs. Besides, our work is not designed for constructing exploitable memory layout or synthesizing working exploits. Rather, it focuses on exploring all the possible error behaviors for a single kernel bug.

Regarding the kernel space, existing exploitability assessment works are mainly in three directions. The first direction is to obtain exploitable primitives. Xu et al. [69] exploit use-after-free vulnerabilities using two memory collision mechanisms to perform heap spray in the kernel. SLAKE [70] 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. [71] exploits use-before-initialization vulnerabilities using deterministic stack spraying and reliable exhaustive memory spraying. As a follow-up work, Cho et al. [72] 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 [73] takes advantage of physical memory which is mapped to kernel space for payload injection. KEPLER [74] 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 [15] bypasses KASLR and heap cookie protector using a special but pervasive type of structure. The third direction is to explore the capability of vulnerabilties, which is most related to our work. In this direction, FUZE [75] 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 [76] extracts capabilities of a slab-out-of-bound access vulnerability manifested in the PoC program and uncover 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 generalized 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.

VIII. Conclusion and Future Work

We design and develop an object-driven kernel fuzzing method. Using our proposed technique, security analysts could explore various contexts and paths toward a target kernel bug and exhibit the bug's many error behaviors. 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 help security analysts better understand and infer exploitability for a given kernel bug.

In this work, we focus on developing technical methods to expose multiple error behaviors only for Linux kernel bugs. Thus, one of our future directions is to explore the proposed method against the bugs on other kernels (e.g., XNU, FreeBSD). While our design is general and not specific for Linux, we argue intensive engineering efforts are still

necessary. First, the debugging features such as KASAN are not always available on other OSes (e.g., no KASAN support for FreeBSD). Even if GREBE could trigger the bug through different paths or contexts, the lack of debugging features would lead to the failure of reporting the severe error behavior exposed through these new paths or contexts. Second, existing fuzzing tools and templates are mainly designed for Linux kernel and, to enable GREBE on other OSes, we will have to port fuzzing tools to other OSes and enrich the fuzzing templates accordingly (e.g., no open-sourced Syzkaller support for XNU). As part of our future work, we will devote our energy to these engineering efforts.

Acknowledgements

We thank the anonymous reviewers for their valuable suggestions and feedback. This work was partially supported by the 2020 IBM PhD Fellowship Program, NSF 1954466, NSF 1718459, and ONR N00014-20-1-2008. This paper gives the views of the author, and not necessarily the position of the funding agency.

References

[1] S. Team, "!exploitable crash analyzer version 1.6," 2013.
[2] B. J. Wever, "Bugid - automated bug analysis," 2017, https://prezi.com/caic9eqayy-o/bugid-automated-bug-analysis/.
[3] J. Vanegue, "In memory safety, the soundness of attacks is what matters," 2020.
[4] D. Vyukov, "Syzkaller," 2020, https://github.com/google/syzkaller.
[5] 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.
[6] D. Jones, "Trinity," 2020, https://github.com/kernelslacker/trinity.
[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] syzbot, "general protection fault in hrtimer_active," 2017, https://syzkaller.appspot.com/bug?id=5d3cce34cc09f722e859ae2037801f5b0d67c5c9.
[9] "Linux kernel design patterns - part 2," https://lwn.net/Articles/336255/, 2009.
[10] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," 1998.
[11] syzbot, "KASAN: slab-out-of-bounds Read in default_write_copy_kernel," 2019, https://syzkaller.appspot.com/bug?id=7022420cc54310220ebad2da89e499bdb1f0f5e8.
[12] syzbot, "BUG: unable to handle kernel paging request in check_memory_region," 2018, https://syzkaller.appspot.com/bug?id=692a8c2104416b219c0036b0a566eb88f73b1dd5.
[13] Z. Lin, "Grebe's source code," 2021, https://github.com/Markakd/GREBE.
[14] 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.
[15] 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.
[16] I. Ben-Gal, "Outlier detection," in *Data mining and knowledge discovery handbook*. Springer, 2005, pp. 131–146.
[17] 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.
[18] 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.
[19] 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.
[20] 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.
[21] 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.
[22] 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.
[23] syzbot, "WARNING: refcount bug in crypto_mod_get," 2020, https://syzkaller.appspot.com/bug?id=bdeea91ae259b3a42aa8ed8d8c91afd871eb5d80.
[24] syzbot, "WARNING: refcount bug in nr_insert_socket," 2019, https://syzkaller.appspot.com/bug?id=521a764b3fc8145496efa50600dfe2a67e49b90b.
[25] syzbot, "general protection fault in delayed_uprobe_remove," 2019, https://syzkaller.appspot.com/bug?id=229e0b718232b004dfddaeac61d8d66990ed247a.
[26] "Full performance results of syzkaller, syzkaller variant, grebe without mutation optimization and grebe," 2021, https://tinyurl.com/x9ky26ms.
[27] syzbot, "BUG: unable to handle kernel paging request in skb_release_data," 2017, https://syzkaller.appspot.com/bug?id=d1baeb189d38d5ba53517876c89b20d4e6857bc6.

[28] syzbot, "BUG: corrupted list in ___neigh_create," 2019, https://syzkaller.appspot.com/bug?id=de28cb0e686acfa1c9dbad1e11cbb0ac9b05caf2.

[29] syzbot, "Warning: refcount bug," 2020, https://syzkaller.appspot.com/bug?id=8eceaff64a35a9f02c1315bbf12b7f262a0b4f08.

[30] syzbot, "general protection fault in qrtr_endpoint_post," 2020, https://syzkaller.appspot.com/bug?id=f56bbe6668873ee245986bbd23312b895fa5a50a.

[31] syzbot, "WARNING in get_pi_state," 2017, https://syzkaller.appspot.com/bug?id=bb7fa48ebde0db8e3fc683a47bb69ab1dca895bc.

[32] syzbot, "BUG: corrupted list in kobject_add_internal," 2020, https://syzkaller.appspot.com/bug?id=f0ec9a394925aafbdf13d0a7e6af4cff860f0ed6.

[33] syzbot, "KASAN: general protection fault in crypto_chacha20_crypt," 2018, https://syzkaller.appspot.com/bug?id=d767177245c54af614d5241159cce56995eef0db.

[34] syzbot, "WARNING: ODEBUG bug in io_sqe_files_unregister," 2020, https://syzkaller.appspot.com/bug?id=460cc948740aa1e715156c0edf5d5d397401d557.

[35] syzbot, "WARNING in vhost_dev_cleanup," 2018, https://syzkaller.appspot.com/bug?id=0df4c1a9c14776f5fd163180e3580ad88b32649a.

[36] syzbot, "general protection fault in vb2_mmap," 2019, https://syzkaller.appspot.com/bug?id=4cf5ee79b52a4797c5bd40a58bd6ab243d40de48.

[37] syzbot, "general protection fault in strlen," 2018, https://syzkaller.appspot.com/bug?id=502c872feb9bbb5ad6494c349c7faa87a9f1777b.

[38] syzbot, "WARNING in dma_buf_vunmap," 2019, https://syzkaller.appspot.com/bug?id=163388d1fb80146cd3ba22a11a5a1995c3eaaafe.

[39] syzbot, "BUG: unable to handle kernel paging request in ethnl_update_bitset32," 2020, https://syzkaller.appspot.com/bug?id=b36d7e444fe532685b683ae7980f4e3a184f0ad8.

[40] syzbot, "general protection fault in scatterwalk_copychunks," 2018, https://syzkaller.appspot.com/bug?id=1fd1d44caf96ca464e1c1f19299d1f3e7558f6e5.

[41] syzbot, "BUG: corrupted list in mousedev_release," 2020, https://syzkaller.appspot.com/bug?id=b9b37a7aaeb4a4e2357b2dfdd1f689e3ffa66282.

[42] syzbot, "general protection fault in bpf_tcp_close," 2018, https://syzkaller.appspot.com/bug?id=695527bd03b09f741819baddcd231c16fe014a48.

[43] syzbot, "general protection fault in hci_event_packet," 2020, https://syzkaller.appspot.com/bug?id=0d93140da5a82305a66a136af99b088b75177b99.

[44] syzbot, "BUG: unable to handle kernel paging request in pcpu_freelist_populate," 2020, https://syzkaller.appspot.com/bug?id=85fd017144b9b1d6761870ff71852d15e4cdd44e.

[45] syzbot, "general protection fault in kernel_accept," 2019, https://syzkaller.appspot.com/bug?id=b0e30ab5186d097b8e3e23e8ca971fbf1cf54659.

[46] syzbot, "WARNING: Odebug bug in tcf_queue_work," 2020, https://syzkaller.appspot.com/bug?id=6a039858238a38cbc7f372607fc5d49f4469cf2c.

[47] syzbot, "WARNING: bad unlock balance in ucma_event_handler," 2020, https://syzkaller.appspot.com/bug?id=d5222b3e1659e0aea19df562c79f216515740daa.

[48] syzbot, "general protection fault in syscall_return_slowpath," 2020, https://syzkaller.appspot.com/bug?id=575a090948f98f28593563c9d9d9b343eb39bbb4.

[49] syzbot, "KASAN: slab-out-of-bounds read in bitmap_ip_add," 2020, https://syzkaller.appspot.com/bug?id=3a6c9972ff471c4dbc3f45e83dd5fa2f18cb82a4.

[50] syzbot, "KASAN: use-after-free read in ip6_dst_destroy," 2020, https://syzkaller.appspot.com/bug?id=27ae1ae5c54e09f8c86dd9428df048e7886be6dc.

[51] syzbot, "KASAN: use-after-free read in sctp_auth_free," 2020, https://syzkaller.appspot.com/bug?id=cbb289816e728f56a4e2c1b854a3163402fe2f88.

[52] syzbot, "KASAN: slab-out-of-bounds read in hci_extended_inquiry_result_evt," 2020, https://syzkaller.appspot.com/bug?id=4bf11aa05c4ca51ce0df86e500fce486552dc8d2.

[53] syzbot, "KASAN: slab-out-of-bounds write in sha512_final," 2018, https://syzkaller.appspot.com/bug?id=e4be30826c1b7777d69a9e3e20bc7b708ee8f82c.

[54] syzbot, "KASAN: use-after-free read in cma_bind_port," 2018, https://syzkaller.appspot.com/bug?id=3b7409f639067d927b8ad1b11a5e08bae27061af.

[55] syzbot, "KASAN: use-after-free read in tipc_nl_node_dump_monitor_peer," 2019, https://syzkaller.appspot.com/bug?id=ddaf58be21bc0aacece5a53ab1ae5db7e89f5bb0.

[56] "Exploit for #6a03986," 2021, https://www.dropbox.com/sh/pk12ajnrw16hdc8/AACuIC2c0my7-i9RZkk87nCNa.

[57] 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.

[58] 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.

[59] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "AEG: Automatic exploit generation," in *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS)*, 2011.

[60] D. Brumley, P. Poosankam, D. X. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," in *Proceedings of the 29th IEEE Symposium on Security and Privacy (S&P)*, 2008.

[61] T. Bao, R. Wang, Y. Shoshitaishvili, and D. Brumley, "Your exploit is mine: Automatic shellcode transplant for remote exploits," in *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P)*, 2017.

[62] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware," in *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS)*, 2015.

[63] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK:(state of) the art of war: Offensive techniques in binary analysis," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P)*, 2016.

[64] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[65] S. Heelan, T. Melham, and D. Kroening, "Automatic heap layout manipulation for exploitation," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018.

[66] S. Heelan, T. Melham, and D. Kroening, "Gollum: Modular and greybox exploit generation for heap overflows in interpreters," in *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[67] Y. Wang, C. Zhang, X. Xiang, Z. Zhao, W. Li, X. Gong, B. Liu, K. Chen, and W. Zou, "Revery: From proof-of-concept to exploitable," in *Proceedings of the 25nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[68] I. Yun, D. Kapil, and T. Kim, "Automatic techniques to systematically discover new heap exploitation primitives," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, 2020.

[69] 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.

[70] 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.

[71] 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.

[72] 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.

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

[74] 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.

[75] 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.
[76] 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.
[77] ooo, "DC29 final scoreboard," 2021, https://scoreboard.ooo/scores.html.
[78] A. Authors, "Behavior representing exploitability," 2021, https://forms.gle/vdPiASeYycqEzEi29.
[79] syzbot, "KASAN: global-out-of-bounds read in fbcon_resize," 2020, https://syzkaller.appspot.com/bug?id=ebcbbb6576958a496500fee9cf7aa83ea00b5920.
[80] syzbot, "kernel BUG at security/keys/keyring.c:line!" 2019, https://syzkaller.appspot.com/bug?id=f7649aa07ffca82dc93dc5cebc00c665849f5138.
[81] syzbot, "WARNING in snd_info_get_line," 2020, https://syzkaller.appspot.com/bug?id=27ea7ae6337aef698924e3eac5aa2b925374ca4c.
[82] syzbot, "KASAN: use-after-free read in devlink_health_reporter_destroy," 2020, https://syzkaller.appspot.com/bug?id=b7f48618d1139d02d0faba8e5932c51eec329b65.
[83] syzbot, "WARNING: refcount bug in qdisc_put (2)," 2020, https://syzkaller.appspot.com/bug?id=badc9136121e634336bcdb31592a4b70b064e421.
[84] syzbot, "KASAN: use-after-free read in do_madvise," 2020, https://syzkaller.appspot.com/bug?id=33913c931f51814eeb2ecdbe03af91d1d6d73520.
[85] syzbot, "Warning in snd_usbmidi_submit_urb/usb_submit_urb," 2020, https://syzkaller.appspot.com/bug?id=28741ff1906f93db2a398bc40e082da51828ec5b.

APPENDIX

A. Detail of User Study

To find the relationship between a kernel bug's error behavior and its exploitation potential, we designed a survey (i.e. Figure 3) and conducted a user study with IRB approval. In our survey, we first asked the subjects' backgrounds, including their occupations and expertise levels. Following the background inquiry, we investigated whether the participants agree that, in most scenarios, the kernel error behaviors like double free, use-after-free, and out-of-bound access imply higher exploitation potential than the kernel error behaviors such as BUG, GPF, WARN, and NULL pointer dereference. We drew comparison between different error behaviors in the survey and provided examples to help the participants understand the context. For each comparison pair, the participant is required to briefly explain the reason if he/she disagrees with our classification.

We started our recruitment from CTF players in top-tier teams [77]. The invited players were encouraged to distribute our survey to knowledgeable experts further. We offer a $10 gift card for each participant to motivate the completion of our survey. In total, we managed to recruit 21 security experts participating in our survey. Among these human subjects, 12 of them claim themselves as CTF players or exploit practitioners. 14 are researchers in academia. 2 are members of blue team in an enterprise. Note that one participant can have multiple roles. Besides, 10 participants have experience in crafting Linux kernel exploits, 4 reading write-ups or exploits, and 7 debugging kernel and developing patches. As the subjects participating in the survey have diverse backgrounds, we deem that the survey results reflect the viewpoint of most security experts regarding how to assess the exploitability of bugs according to their error behaviors.

1) Which of the following roles do you identify yourself as (multiple choices)?
   a. CTF player/exploit practitioner
   b. Academia researcher
   c. Security analyst in enterprise blue team
   d. Official in government agency
2) How's your experience in Linux kernel exploitation
   a. I've debugged kernel or developed kernel patches but done nothing about exploitation
   b. I've read some writeups
   c. I've written some exploits for CTF challenges or real-world vulnerabilities
3) What's the easiest way to get in touch with you? We ask this question for gift card sending and potential follow-up question. We promise to keep privacy and won't identify you via the contact.
4) Do you agree that, without going into details, double free behavior implies higher exploitability than BUG in most cases?
   a. Yes
   b. No
   c. I don't know

Fig. 3: Sampled questions from the exploitability survey form [78]

Our survey showed that 18 out of 21 participants agree that, for all comparison pairs, error behaviors like double-free, use-after-free, and out-of-bound access imply higher exploitation potential. For the comparison between double-free and BUG/GPF, use-after-free and BUG/GPF, out-of-bound access, and BUG/GPF/WARN/NULL ptr deref, there are 1/2, 2/3, and 3/2/1/1 participants who disagree with our classification, respectively. They explained that, in the situation where the attacker can control the corruption range, errors like GPF/BUG/WARN could imply higher exploitation potential. In our user study, we further contacted those participants for further clarification. In the follow-up interview, they conceded that though they have encountered some particular cases, they agree that our classification works in most situations. As such, we carefully conclude that there is a shared sense among security analysts. That is, compared with error behaviors like GPF/WARN/BUG/NULL ptr deref, kernel error behaviors such as double free, use-after-free, and out-of-bound imply higher exploitation potential.

B. Procedure of Error Triaging

When exploring multiple error behaviors for a target bug, `GREBE` may hit other bugs, demonstrating error behaviors that do not result from the target bug. To ensure the newly identified error behaviors are truly tied to the bug of our interest, error triaging is needed. As we mentioned earlier,

there has not yet been accurate error triaging methods. We, therefore, seek the help of kernel professionals.

In this work, our professional team performed error triaging by following the procedure below. Given a bug of our interest, the team first finds the bug's patch and applies it to the corresponding kernel image. Then, for each newly identified error behavior, the team executes the PoC program tied to that newly discovered error behavior. If the patch fails to block the demonstration of the error (i.e., the patched kernel still crashes), the team excludes that error behavior with the conclusion that it is not associated with the target bug. Otherwise, the team will put their effort into inspecting the execution of the PoC program. In the inspection phase, the team will manually examine the bug patch and extract the condition of the bug triggering. 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 asked the team to form a unanimous agreement before we associate that new error behavior to the bug of our interest.

It should be noted that, like Syzkaller, when GREBE triggers a bug and demonstrates an error behavior, it may not generate a PoC program allowing the team to follow the procedure above. In this situation, the team will take a close look at the call stack of the kernel panic. Following the call stack, the team will manually track the kernel execution reversely and infer if the panic results from the same root cause. In this manual analysis phase, the team utilizes several heuristics to align an error with the bug. First, the team will confirm the functions in the call stack are relevant to the functions where the patch is applied. Second, the team will ensure the panic site is related to the variables that the patch influences.

As we can see, the rationale of the triaging procedure above is as follows. We assume that the patch could successfully block the triggering of the bug and thus prevent it from exhibiting the corresponding errors. If the patched kernel image still demonstrates errors, the manifested error is not likely to associate with the bug of our interest. However, it should be noted that the procedure above might mistakenly exclude some error behaviors tied to the bug of our interest simply because the patch might not be correct, and we falsely rule out the corresponding error behaviors. As a result, we emphasize that the error behaviors we identified could mean only the lower bound of the total number of all possible error behaviors. However, as we showed in Section VI, the lower bound still provides good utility, helping a security researcher explore multiple error behaviors for a given kernel bug.

C. Detail of Distance Measurement & Hypothesis Validation

Section VI-B hypothesizes that the distance (number of basic blocks) between a bug's root cause and the corresponding error site may correlate with the false negatives of GREBE. Here, we detail how we measure the distance and present our hypothesis testing result.

It is challenging to measure the distance between the root cause and the error site for a given bug. The Linux kernel is a multi-process system. The system call that triggers the root cause could be different from the one that brings about the error. As a result, we address this issue as follows. First, we identify all the lines of the kernel code that the patch changes. Second, we examine which of these lines is executed first when replaying the PoC program. In this work, we treat that line as our root cause site. If the patch site and the error site share the same system call, we simply count the basic blocks in between. For the kernel bug, the root cause of which and the error site reside in different system calls, we combine the basic blocks of both system calls. More specifically, we take the total number of basic blocks that the error-site-related system call has executed. Then, we add this number to the number of basic blocks that the root-cause-related system calls have executed (right after the root cause is triggered and before the error occurs).

In Table V ("N of BB" column), we show the distance measure for the error manifested in the bug's original report. We mark the distance measure with a star sign if that bug's root cause and error site do not share the same system call. As we can observe from the table, there is no clear relation between the distance and the false positive. GREBE demonstrates false positives regardless of whether the distance is long or short enough. In addition, the false-negative occurrence does not depend upon whether root cause and error site share (or not share) the same system call. With these observations, we safely reject our hypothesis.

SYZ ID	N of BB	New Behaviors Discovered Manually	New Behaviors Discovered by GREBE
1fd1d44[40]	5128 ⋆	general protection fault in skcipher_walk_done	general protection fault in skcipher_walk_done
695527b[42]	2313 ⋆	KASAN: use-after-free Write in bpf_tcp_close	KASAN: use-after-free Write in bpf_tcp_close
		BUG: unable to handle kernel paging request in qlist_free_all	BUG: unable to handle kernel paging request in qlist_free_all
		WARNING: ODEBUG bug in sock_hash_free	-
ebcbbb6[79]	1	-	-
f7649aa[80]	38	-	-
6a03985[46]	2	general protection fault in hfsc_unbind_tcf	general protection fault in hfsc_unbind_tcf
		WARNING: refcount bug in __tcf_action_put	WARNING: refcount bug in __tcf_action_put
		KASAN: use-after-free Read in route4_get	KASAN: use-after-free Read in route4_get
		WARNING: refcount bug in route4_destroy	WARNING: refcount bug in route4_destroy
		KASAN: null-ptr-deref Read in route4_destroy	KASAN: null-ptr-deref Read in route4_destroy
		KASAN: use-after-free Read in route4_destroy	KASAN: use-after-free Read in route4_destroy
27ea7ae[81]	1	-	-
d5222b3[47]	652 ⋆	WARNING: bad unlock balance in ucma_destroy_id	WARNING: bad unlock balance in ucma_destroy_id
		general protection fault in rdma_listen	general protection fault in rdma_listen
		KASAN: use-after-free Read in addr_handler	-
		KASAN: use-after-free Read in cma_cancel_operation	-
		KASAN: use-after-free Read in rdma_listen	KASAN: use-after-free Read in rdma_listen
		BUG: corrupted list in rdma_listen	BUG: corrupted list in rdma_listen
de28cb0[28]	2	BUG: corrupted list in neigh_mark_dead	BUG: corrupted list in neigh_mark_dead
		KASAN: use-after-free Read in neigh_mark_dead	KASAN: use-after-free Read in neigh_mark_dead
		KASAN: slab-out-of-bounds Read in neigh_mark_dead	KASAN: slab-out-of-bounds Read in neigh_mark_dead
		KASAN: use-after-free Read in ___neigh_create	KASAN: use-after-free Read in ___neigh_create
		KASAN: slab-out-of-bounds Read in ___neigh_create	KASAN: slab-out-of-bounds Read in ___neigh_create
		KASAN: use-after-free Read in neigh_change_state	KASAN: use-after-free Read in neigh_change_state
f56bbe6[30]	2	KASAN: slab-out-of-bounds Read in qrtr_endpoint_post	KASAN: slab-out-of-bounds Read in qrtr_endpoint_post
b7f4861[82]	1	-	-
e4be308[53]	857 ⋆	KASAN: slab-out-of-bounds Write in tgr192_final	KASAN: slab-out-of-bounds Write in tgr192_final
		KASAN: slab-out-of-bounds Write in tgr160_final	KASAN: slab-out-of-bounds Write in tgr160_final
		KASAN: slab-out-of-bounds Write in crypto_sha3_final	KASAN: slab-out-of-bounds Write in crypto_sha3_final
		KASAN: slab-out-of-bounds Write in rmd320_final	KASAN: slab-out-of-bounds Write in rmd320_final
		KASAN: slab-out-of-bounds Write in wp384_final	KASAN: slab-out-of-bounds Write in wp384_final
		KASAN: slab-out-of-bounds Write in sha512_finup	KASAN: slab-out-of-bounds Write in sha512_finup
		KASAN: slab-out-of-bounds Write in sha1_finup	KASAN: slab-out-of-bounds Write in sha1_finup
		KASAN: slab-out-of-bounds Write in sha1_final	KASAN: slab-out-of-bounds Write in sha1_final
		KASAN: slab-out-of-bounds Write in sha256_final	KASAN: slab-out-of-bounds Write in sha256_final
		KASAN: slab-out-of-bounds Write in rmd160_final	KASAN: slab-out-of-bounds Write in rmd160_final
		KASAN: slab-out-of-bounds Write in sha256_finup	KASAN: slab-out-of-bounds Write in sha256_finup
d1baeb1[27]	1043 ⋆	general protection fault in skb_release_data	general protection fault in skb_release_data
		general protection fault in skb_clone	-
		KASAN: wild-memory-access Read in skb_copy_ubufs	KASAN: wild-memory-access Read in skb_copy_ubufs
		KASAN: slab-out-of-bounds Write in pskb_expand_head	KASAN: slab-out-of-bounds Write in pskb_expand_head
7022420[11]	64	KASAN: slab-out-of-bounds Write in default_read_copy_kernel	KASAN: slab-out-of-bounds Write in default_read_copy_kernel
0df4c1a[35]	553	KASAN: use-after-free Read in remove_wait_queue	KASAN: use-after-free Read in remove_wait_queue
		KASAN: use-after-free Read in corrupted	KASAN: use-after-free Read in corrupted
		KASAN: use-after-free Read in eventfd_release	KASAN: use-after-free Read in eventfd_release
badc913[83]	1510 ⋆	-	-
33913c9[84]	1	KASAN: use-after-free Read in do_madvise	KASAN: use-after-free Read in do_madvise
28741ff[85]	2	WARNING in snd_usbmidi_submit_urb/usb_submit_urb	WARNING in snd_usbmidi_submit_urb/usb_submit_urb
0df4c1a[35]	6	BUG: unable to handle kernel paging request in pcpu_freelist_populate	BUG: unable to handle kernel paging request in pcpu_freelist_populate
		BUG: unable to handle kernel paging request in htab_map_alloc	BUG: unable to handle kernel paging request in htab_map_alloc
		BUG: unable to handle kernel paging request in bpf_lru_populate	BUG: unable to handle kernel paging request in bpf_lru_populate
		KASAN: vmalloc-out-of-bounds Write in pcpu_freelist_populate	KASAN: vmalloc-out-of-bounds Write in pcpu_freelist_populate
		KASAN: vmalloc-out-of-bounds Write in bpf_lru_populate	KASAN: vmalloc-out-of-bounds Write in bpf_lru_populate
		KASAN: vmalloc-out-of-bounds Write in htab_map_alloc	KASAN: vmalloc-out-of-bounds Write in htab_map_alloc
		KASAN: vmalloc-out-of-bounds Read in htab_free_elems	KASAN: vmalloc-out-of-bounds Read in htab_free_elems
		BUG: unable to handle kernel paging request in htab_free_elems	-

TABLE V: The results of false negative analysis. The "SYZ ID" column is the case ID. The second column is the number of basic block between the root cause and the crash site. The star ⋆ right after the number indicates that the site of root cause and crash are from different syscalls. Otherwise, they are from the same syscall. The third and fourth column represent the new behaviors discovered manually and by GREBE, respectively. The dash "-" means no such behavior is discovered in the corresponding way. It should be noted that the cases like #ebcbbb6 have two dashes in a row. It is because no new error behavior was discovered either manually or by GREBE.