RPG: Linux Kernel Fuzzing Guided by Distribution-Specific Runtime Parameter Interfaces

Yuhan Chen[1]*, Yuheng Shen[2]*, Guoyu Yin[1], Fan Ding[1], Runzhe Wang[3]
Tao Ma[3], Xiaohai Shi[3], Qiang Fu[1], Ying Fu[2], Heyuan Shi[1]†
[1]Central South University, Changsha, China
[2]Tsinghua University, Beijing, China
[3]Alibaba Co., Ltd, Beijing, China

***Abstract*—The Linux distribution kernel differs significantly from the mainline kernel, incorporating additional features and vendor-specific extensions. Among these additions, many runtime parameter interfaces are unique to distribution kernels, which expands the attack surface and increases the risk of potential vulnerabilities. Fuzzing has been used to assess Linux distributions, but existing tools cannot systematically test these distribution-specific interfaces due to two main challenges: (1) generating test cases for these runtime parameter interfaces, and (2) concentrating test resources on the distribution-specific interface code. To address these challenges, we propose RPG, a distribution-specific runtime parameter-guided kernel fuzzer. RPG operates in three phases: First, RPG extracts distribution-specific runtime parameter interfaces. Then, RPG uses LLM and tuning software databases to model each parameter range to generate meaningful interface test cases. Third, RPG utilizes the distribution kernel's function control flow graph to guide the fuzzer to generate generic test cases that are more closely related to the distribution-specific interface code. We evaluated RPG on four Linux distribution kernels: Ubuntu 22.04, Fedora 42, OpenAnolis 8.8, and OpenAnolis 23.1. RPG detected 22 previously unknown bugs (13 distribution-specific), of which 15 were confirmed and 10 fixed by kernel maintainers. RPG also achieved 20.4% and 21.2% higher branch coverage than Syzkaller and Healer, respectively.**

***Index Terms*—Kernel Fuzzing, Linux Distribution Kernels, Tuning Software.**

I. Introduction

Most Linux distribution kernels are customized versions of the mainline kernel, containing a large number of specific runtime parameter interfaces that are not enabled by default in the mainline kernel. For example, the Ubuntu 22.04 kernel provides 22,592 numeric writable parameter interfaces, which is 35.2% more than the 16,706 in the corresponding mainline Linux 6.8.12 kernel. These runtime parameter interfaces typically exist as files in the `/sys` or `/proc/sys` directories, and their parameter values can affect the kernel execution path [1]. Distribution kernel developers usually use these interfaces in conjunction with private code or tuning software [2] to support various business scenarios and requirements.

Runtime parameter interfaces introduce notable security risks, as evidenced by historical vulnerabilities such as CVE-2025-22053 [3], which demonstrate that these interfaces can be exploited in real-world attacks. Unlike mainline kernel code, the distribution-specific runtime parameter interfaces are often less tested, increasing the likelihood of potential vulnerabilities. Furthermore, their tight integration with the private distribution code can introduce unique, distribution-specific vulnerabilities. Since distribution kernels are deployed directly in production environments and serve end-users, any such error can lead to severe consequences, including data leaks or system compromise [4]. Therefore, it is critical to test these distribution-specific runtime parameter interfaces.

Kernel fuzzing serves as an effective automated method for identifying kernel bugs [5]–[13]. It has been applied to check the Linux distribution's safety. Previous research has explored the challenges of coverage-guided kernel fuzzing in enterprise kernels [14], [15]. While some work [16] has examined the technical difficulties of applying directed kernel fuzzing to distribution kernels. However, existing fuzzing research has focused on applying existing fuzzing tools to distribution environments, without specifically testing the differences between the distribution kernel and the mainline kernel. This phenomenon exists because the differences are difficult to pinpoint. Developers across various distribution communities modify the code in different ways and follow diverse practices. However, distribution-specific runtime parameter interfaces can serve as a good starting point. These specific interfaces clearly represent the differences between this distribution and the mainline kernel. If we plan to use distribution-specific runtime parameters in fuzzing, we need to solve:

(1) Generate distribution-specific runtime parameter interface test cases. Runtime parameter interfaces reside in the kernel's `/sys` and `/proc/sys` directories, allowing developers to modify kernel behavior by writing values to these files [1]. To effectively integrate these interfaces into fuzzing, we must generate meaningful interface test cases. However, kernel fuzzing relies on syscall sequences as test inputs [17]. Accessing a parameter interface involves a specific sequence of syscalls (e.g., open, write, close) that must appear in the correct order. If this syscall sequence is disrupted, the test may fail to correctly access the interface or trigger the expected kernel behavior. Moreover, each interface accepts a limited and specific set of valid values that influence kernel execution paths. For example, `/proc/sys/vm/panic_on_oom` accepts values 0 or 1, while `/proc/sys/vm/swappiness`

*Both authors contributed equally to this research.
†Heyuan Shi is the corresponding author.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00299

accepts values from 0 to 100. Without awareness of such constraints, fuzzers tend to generate invalid or ineffective inputs, wasting test resources and missing deeper kernel vulnerabilities. Although manual analysis can reveal valid parameter ranges, doing so for thousands of interfaces is impractical. Therefore, an automated approach is needed to infer valid value ranges and systematically generate test cases for distribution-specific parameter interfaces.

(2) Concentrate test resources on the distribution-specific interface code. Once interface test cases related to runtime parameters are generated, the next challenge is to concentrate test resources on the distribution-specific interface code. Existing kernel fuzzers rely on randomly generated syscall sequences. This method is inefficient when testing Linux distribution-specific interfaces. For example, the private function `update_cpuectlr_sysctl_handler` in OpenAnolis 8.8 is affected by the value of `sysctl_update_cpuectlr`, which controls register update behavior [18]. The kernel fuzzers' strategies for randomly generating syscalls don't actively generate generic test cases targeting these functions like `update_cpuectlr_sysctl_handler`. However, these areas are poorly tested and contain a large amount of distribution-specific kernel code. As a result, fuzzers may miss deep bugs in the distribution kernel. To overcome this limitation, we need to guide fuzzer to concentrate test resources on distribution-specific interface code.

To address these challenges, we propose RPG, a fuzzing tool guided by distribution-specific runtime parameter interfaces. RPG operates in three phases: (1) RPG extracts Linux distribution-specific runtime parameter interfaces. (2) RPG uses large language model(LLM) and tuning software databases to model parameter ranges, combining them with pseudo-syscalls templates to generate interface test cases. (3) RPG uses the distribution kernel function control flow graph (CFG) to guide the fuzzer to generate more generic test cases closely related to the distribution-specific interface code.

We evaluated RPG on four Linux distribution kernels: Ubuntu 22.04, Fedora 42, OpenAnolis 8.8, and OpenAnolis 23.1. RPG has detected 22 previously unknown bugs, including 13 distribution-specific bugs. Among the 22 bugs, 15 bugs have been confirmed, and 10 bugs have been fixed by kernel maintainers. RPG also achieved 20.4% and 21.2% higher branch coverage than Syzkaller [19] and Healer [20], respectively. These results demonstrate that RPG significantly improves bug detection in the Linux distribution kernel.

In summary, the contributions of this paper are as follows:

- We have solved the problem of generating test cases related to distribution-specific runtime parameter interfaces and the issue of concentrating test resources on the distribution-specific interface code.
- We propose RPG, a kernel fuzzer designed to test Linux distribution-specific runtime parameter interfaces.
- RPG has detected 22 previously unknown bugs (15 confirmed, 10 fixed), and 13 out of 22 were distribution-specific bugs.

II. BACKGROUND

A. Linux Runtime Parameter Interface

The runtime parameter interface is a critical control mechanism within the Linux kernel. Modifying the values of these parameters directly influences the kernel's code execution path. For example, as illustrated in Figure 1, the parameter `/proc/sys/net/ipv4/ip_forward` controls the IP forwarding functionality of the system [21]. When set to 1, the kernel allows the system to forward network packets between network interfaces (lines 3-5). When set to 0, the kernel disables IP forwarding, and the system will not forward packets (lines 6-10).

```
1  // net/ipv4/ip_forward.c
2  void handle_ip_forward(struct net_device *dev,
       struct sk_buff *skb) {
3      if (sysctl_ip_forward == 1) {
4          forward_packet(skb);
5          // Forward the packet
6      } else if (sysctl_ip_forward == 0) {
7          drop_packet(skb);
8          // Drop the packet
9      }
10 }
```

Fig. 1: Runtime Parameter Affect Kernel Execution Flow.

These runtime parameter interfaces have many application scenarios, such as tuning software. Tools like KeenTune [22] dynamically adjust the range of memory management parameters in real-time, adapting to the specific CPU architecture and workload to enhance memory usage efficiency. Similarly, Tuned [2] utilizes predefined templates to statically associate hundreds of parameters, delivering performance-optimized configurations for diverse scenarios. Such tools provide the flexibility to tailor kernel parameters to meet the specific performance demands of various environments. Underpinning these tools are databases that store extensive sets of commonly used runtime parameter values as key-value pairs, enabling users to achieve rapid and precise system performance optimization.

However, runtime parameter interfaces also introduce significant security risks, as demonstrated by the CVE-2025-22053 vulnerability shown in Figure 2 [3]. This vulnerability arises from concurrent write operations to the `/sys/devirtual/net/pool/active` parameter by multiple threads (lines 15-17). When multiple threads simultaneously modify this parameter: Thread A writing 0 invokes `ibmveth_close()` (lines 3-4), which calls `napi_disable()` (lines 9-11). Thread B writing 1 invokes `ibmveth_open()` (lines 5-6), which calls `napi_enable()` (lines 12-14). But the kernel requires strict ordering of these NAPI operations: `napi_disable()` must complete before `napi_enable()` can be safely executed. Without locking in `veth_pool_store()`, concurrent execution may violate this sequence, causing inconsistent NAPI states. This inconsistency triggers recursive locking attempts,

```
void veth_pool_store(Device *dev, int new_state) {
   // Vulnerable: no locking for concurrent writes
   if (new_state == 0) {
      ibmveth_close(dev); // Calls napi_disable()
   } else if (new_state == 1) {
      ibmveth_open(dev); // Calls napi_enable()
   }
}
void ibmveth_close(Device *dev) {
   napi_disable(&dev->napi);
}
void ibmveth_open(Device *dev) {
   napi_enable(&dev->napi);
}
// Exploit scenario:
Thread A: echo 0 > /sys/.../pool1/active // Enters ibmveth_close()
Thread B: echo 1 > /sys/.../pool1/active // Enters ibmveth_open()
```

Fig. 2: CVE-2025-22053 Caused by Runtime Parameter.

resulting in soft lockups that hang the system for over 120 seconds.

B. Distribution-Specific Runtime Parameter Interface

Linux distribution kernels, such as Ubuntu [23] and OpenAnolis [24], represent customized versions of the mainline kernel that introduce substantial modifications, including distribution-specific runtime parameter interfaces. These interfaces exhibit significant quantitative expansion compared to their mainline counterparts. For example, the OpenAnolis 23.1 kernel offers 21,512 numeric writable runtime parameter interfaces, 27.5% more than the 16,872 in the mainline Linux 6.6.25 kernel.

These distribution-specific interfaces mainly serve scenario-specific functionality, including custom functional requirements, and performance tuning for target workloads. More critically, many parameters are tightly integrated with distribution-specific functional modules that form part of the distribution's feature. For example, OpenAnolis 8.8 implements the `update_cpuectlr` runtime parameter interface in the `/alibaba/prefetch_tuning.c` module to control register updates [18]. This file does not exist in the mainline Linux kernel. It represents a functionality deeply customized by the OpenAnolis 8.8 kernel for specific platforms.

The number of specific runtime parameter interfaces in the distribution kernel and their close integration with private code have led to an expansion of the vulnerability surface. Therefore, to ensure the security of the distribution kernel, distribution kernel developers need an effective method to test these specific interfaces.

C. Linux Kernel Fuzzing

Fuzzing [16], [25]–[32] is an automated software vulnerability detection technology. It mainly injects abnormal test cases into the target program and monitors its runtime behavior to identify potential security bugs. It plays a crucial role in the vulnerability discovery of complex systems such as the operating system kernel and the network protocol stack. Syzkaller [19] is one of the most advanced kernel fuzzing tools, which is widely used in Linux kernel security testing. Its core technology is based on the syscall description language (Syzlang) [33]. Syzkaller identifies potential bugs by defining test case generation rules and simulating the kernel execution path. Meanwhile, Syzkaller also supports pseudo-syscalls [34], which are user-defined functions. It can better simulate complex syscalls to detect potential kernel bugs.

In addition to Syzkaller, there are many innovative tools [17], [20], [35]–[38] in the field of kernel fuzzing that have proposed new solutions for addressing kernel complexity. For example, HFL [37] combines symbolic execution and fuzzing, and uses constraint-solving techniques to generate test cases that cover those less executed code paths; Healer [20] adopts a data extraction algorithm to extract high-frequency syscall patterns from real system logs, thereby optimizing the quality of the initial seed set. Healer introduced a relational learning model to analyze the logical dependencies among syscalls and generate a sequence of syscalls that can deeply explore the states of kernel subsystems. Based on eBPF technology, KSG [17] dynamically tracks the parameter types and constraints of syscalls and automatically generates syscall description files for specific kernel modules. These tools collectively enhance test coverage and vulnerability discovery efficiency through differentiated strategies.

III. DESIGN

The overall workflow of RPG is shown in Figure 3. Specifically, RPG includes three phases. In the interface extraction, RPG runs both the Linux distribution kernel and its corresponding mainline kernel on a virtual machine to collect their runtime parameter interfaces and extract the distribution-specific interfaces. In the test case generation, RPG constructs an interface test case template, models the parameter ranges for each interface using LLM [39] and tuning databases, and combines the template with the range model to generate interface test cases that can pass fuzzer syntax checks. In the interface-guided distribution fuzzing loop, RPG calculates the distance between generic test cases and interface code within the distribution's function CFG. This calculation is based on the coverage data collected after each test case execution. Then RPG uses this distance metric to guide the fuzzer in generating more generic test cases closer to the distribution-specific interface code.

A. Interfaces Extraction

The interface extraction stage aims to extract distribution-specific runtime parameter interfaces. RPG first executes the target Linux distribution kernel in a virtualized environment to isolate the impact of hardware. Each kernel is compiled using its default configuration: mainline kernels use `defconfig`, while distribution kernels use their vendor-specific configurations like `anolis_defconfig`.

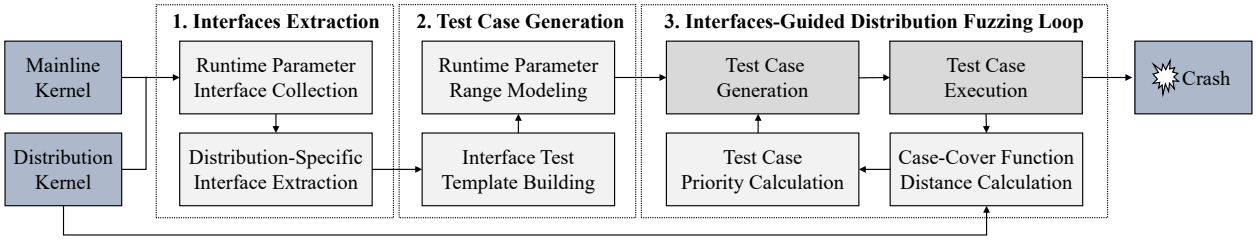


Fig. 3: Workflow of RPG. First, RPG extracts Linux distribution-specific runtime parameter interface. Then, RPG uses LLM and tuning software databases to model parameter ranges, combining them with pseudo-syscalls templates to generate interface test cases. Lastly, RPG utilizes test case coverage function information to guide the fuzzer in generating more generic test cases related to the interface code.

Then, RPG scans the file system collecting runtime parameters, primarily `/sys` and `/proc/sys` directories. These scan results undergo rigorous filtering: non-writable interfaces are discarded due to their limited utility in fuzzing workflows, while string-type parameters are excluded because they often lack official documentation, making it challenging to generate valid test cases automatically. Only writable int-type parameters are collected because of their operational predictability and fuzzing suitability.

Finally, to extract truly distribution-specific interfaces, RPG extracts runtime parameters from the corresponding mainline kernel version using identical methods. A differential comparison is performed between the distribution kernel's interface set and the mainline kernel's interface set. Only interfaces present in the distribution kernel but absent in the mainline kernel are classified as distribution-specific runtime parameters.

B. Test Case Generation

Traditional interface testing methods usually write different parameters directly into interfaces. However, this approach faces several problems: Firstly, kernel fuzzing relies on sequences of syscalls as test inputs. Correctly accessing the runtime parameter interface requires a specific, ordered sequence of syscalls (e.g., open, write, close). Disrupting this sequence can prevent proper interface access or fail to trigger the intended kernel behavior. Meanwhile, each parameter typically has valid value ranges. For parameters lacking documented ranges, testing often resorts to random value writing, leading to a high proportion of invalid test cases. Finally, the distribution kernel developers are concerned about the stability of common parameters because they are directly used in the business, and thus need to identify the range of these common parameters.

To solve the above challenges, RPG first employs templates based on pseudo-syscalls. These templates encapsulate the necessary syscall sequences (such as 'open-write-close') within custom functions, which ensures the correct order of operations for accessing each runtime parameter interface. Meanwhile, custom functions enable diversified testing for individual interfaces, enhancing vulnerability detection capabilities. Second, RPG employs LLM to analyze the valid value range for each parameter. To ensure the reliability of the LLM-generated ranges, RPG categorizes interfaces primarily into

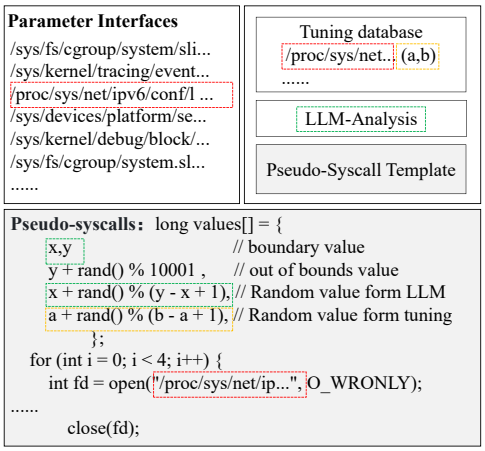


Fig. 4: Distribution-specific runtime parameter interface test case generation. RPG takes parameters interface path as input, combining LLM, tuning databases, and Pseudo-Syscalls templates to generate interface test cases.

three types. (1) For interfaces with clearly defined parameter ranges, it directly generates the range; (2) for interfaces with vast ranges, it caps the maximum value; and (3) for interfaces with no defined range, it generates fixed values. Through this approach, LLM can automatically generate the correct parameter range constraints for most interfaces. Finally, RPG integrates a tuning software database. The tuning database catalogs detailed range information for commonly used parameters across various kernel distributions. It contains data from actual kernel configuration and performance tuning practices, representing commonly used parameters in the real world.

As shown in Figure 4, RPG first gives a distribution-specific runtime parameter interface path to the LLM, and gets the initial range information (x, y). Next, RPG checks its tuning database using this interface path and looks for matching interface records. If found, RPG extracts range information (a, b) from the database. If not found, RPG assigns the LLM's range (x, y) to (a, b). Finally, RPG combines the two range sets with the pseudo-syscalls templates to complete the generation of the interface test cases. This process can ensure

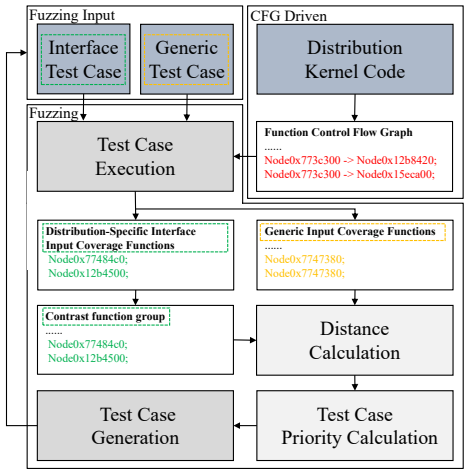


Fig. 5: Interfaces-guided distribution fuzzing loop process. RPG uses the test case coverage function after each execution to guide Fuzzer to actively generate general test cases close to the distribution-specific interface code functions.

the automation and adjustability of the generation. RPG has designed four behavioral strategies for each interface:

- Boundary value testing: Tests minimum and maximum parameter values to check system performance under extreme conditions.
- Out-of-range tests: Deliberately writes values beyond valid ranges to check system error handling.
- Random writing: Writes random values within valid ranges to trigger different code paths.
- commonly used value writing: Writes commonly used parameter values from the tuning database to test the stability of the common parameters.

C. Interfaces-Guided Distribution Fuzzing Loop

After generating interface test cases, RPG adds them to the existing distribution fuzzing loop. The current fuzzing system lacks a mechanism for actively generating generic test cases closed to distribution-specific interface code functions. This limitation reduces the distribution fuzzing efficiency. To maximize the value of interface test cases and improve fuzzing efficiency, RPG proposes the interfaces-guided distribution fuzzing loop mechanism.

As shown in Figure 5, RPG calculates the distance between generic test cases and distribution-specific interface code within the distribution's function CFG, where nodes represent kernel functions and edges encode caller-callee relationships. Shorter paths indicate stronger functional coupling (e.g., shared subsystems), while longer paths suggest weaker interactions (e.g., cross-module calls). By computing the shortest distance between the generic test case coverage function and the distribution-specific interface test case coverage function group, RPG identifies high-value test case candidates for the distribution fuzzing loop.

Algorithm 1: Interface-Guided Distribution Fuzzing

Input: Test case t, Test case cover node group NG, Contrast node group CNG, configurable value p_b

Output: Updated CNG, Priority value p

1 **if** t *is distribution-specific interface test case* **then**
2 **for** *each node* $fn \in NG$ **do**
3 **if** $fn \notin CNG$ **then**
4 $CNG \leftarrow CNG \cup \{fn\}$;
5 // Add new function to contrast group
6 $p \leftarrow p_b$; // Set priority value to p_b
7 **else**
8 $distances \leftarrow \emptyset$; // Initialize empty list for distances
9 **for** *each node* $fn \in NG$ **do**
10 $d \leftarrow$ Algorithm2(fn, CNG);
11 // Shortest distance algorithm
12 $distances$.append(d); // Append the distance
13 $d_e \leftarrow \frac{\text{sum(distances)}}{\text{len(CNG)}}$; // Calculate average distance
14 $p \leftarrow$ Equation 1(d_e); // Calculate priority value
15 **return** (CNG, p);
16 // Return updated contrast node group and priority value

Specifically, RPG first takes both interface test cases and generic test cases as fuzzing input. Then, RPG collects covered function nodes for each test case by executing them. RPG separates these nodes into two groups: The nodes covered by the distribution-specific interface test cases are stored as the contrast node group. This group is constantly updated through the continuous execution of distribution-specific interface test cases. For nodes covered by generic test cases, RPG calculates the shortest distance between the covered node group and the contrast node group. RPG uses this distance with a priority formula to determine the generation priority for each test case.

Algorithm 1 details the core concept of interface-guided distribution fuzzing loop. The input of the algorithm 1 includes the test case t, the test case cover node group NG, the contrast node group CNG and a configurable value p_b. The output is the updated contrast node group CNG and the priority value p. First, the algorithm 1 checks whether the test case is the distribution-specific interface test case (line 1). If it is, the algorithm 1 loops through each node $fn \in NG$ covered by the test case and checks if the node is already in the contrast node group CNG. If the node is not in CNG, it is added to the contrast node group (lines 2–5). The priority value of the interface test case is set to a fixed value $p = pb$(line 6), which is a configurable value. If the test case is not the distribution-specific interface test case, the algorithm 1 first initializes an empty distance list *distances* (line 8). Then, it loops through each node $fn \in NG$ and calculates the shortest distance d from each node to the nodes in the contrast node group using Algorithm 2 (line 10). After calculation, all distance values are

Algorithm 2: Shortest Distance Calculation

Input: Node fn, Contrast node group CNG
Output: Shortest distance d

```
1  queue ← [(fn, 0)]; visited ← {fn};
2  while queue is not empty do
3      (current, cur_dist) ← queue.pop(0);
4      if cur_dist > d_max then
5          d ← d_max; // Distance exceeds limit return d;
6      else
7          if current ∈ CNG then
8              d ← cur_dist; // Found CNG node
9              return d;
10     for each neighbor neb of current do
11         if neb ∉ visited and cur_dist + 1 ≤ d_max
             then
12             visited ← visited ∪ {neb};
13             queue ← queue ∪ [(neb, cur_dist + 1)];
14 return d;
```

added to the distance list *distances* (line 12), and the average distance d_e is calculated based on the distance list (line 13). Finally, the priority value p is calculated using Equation 1 (line 14). The algorithm 1 returns the updated contrast node group (CNG) and the computed priority value p (line 15-16).

Algorithm 2 calculates the shortest functional distance from a given kernel function node fn to the closest node in the contrast node group CNG. Algorithm 2 begins by initializing a queue with the starting node fn and distance 0, while marking fn as visited (lines 1). It then enters a loop where it dequeues the front node $current$ and its associated distance cur_dist. If $current$ is found in the contrast node group CNG, it immediately returns cur_dist as the shortest distance (lines 7-9). If not, the algorithm processes all unvisited neighbors of $current$, adding each neighbor to the visited set and enqueuing them with an incremented distance value (cur_dist + 1) (lines 10-13). Algorithm 2 search continues until either a node in CNG is found or the cur_dist exceeds the maximum value d_max (lines 4-5). The value of d_max can be determined according to Equation 1.

$$w = 1 - \frac{2\arctan(\mathrm{d}_e)}{\pi} \tag{1}$$

As shown in Equation 1, a priority value is assigned based on the shortest distance. To ensure that the priority value remains within a reasonable range, RPG utilizes the $\arctan$ function. Compared to exponential or harmonic functions, the $\arctan$ function offers distinct advantages for priority modeling. Since the output of $\arctan(x)$ always lies within the range $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$, Equation 1 guarantees that the priority w stays within the range [0, 1], gradually decreasing as the distance d_e increases, without dropping too sharply. In contrast, the exponential function e^{-d_e} decreases too rapidly, which may lead to a significant underestimation of distant test points. On the other hand, the harmonic function $\frac{1}{1+d_e}$ decreases too slowly, reducing the distinction between priorities. The $\arctan$ function provides a smoother and more controlled decline, enabling a better balance between prioritizing nearby test points while still exploring distant ones. This ultimately improves the practicality and balance of test case generation.

By continuously executing the process, RPG creates an ever-optimizing distribution fuzzing loop, helping to uncover deeper vulnerabilities in distribution-specific interface code.

IV. IMPLEMENTATION

We implemented RPG on top of Syzkaller. In the interface extraction, RPG performs a dry run of the target distribution kernel and its corresponding mainline Linux kernel on a virtual machine, using Python scripts to analyze and detect runtime parameter interfaces from the /sys and /proc/sys directories, extracting distribution-specific parameter interfaces. In the test case generation, RPG utilizes a Python script to analyze the interfaces, calling the DeepSeek R1 API to generate ranges for each runtime parameter. These ranges are refined based on the KeenTune expert knowledge base, and the range information is combined with the interface pseudo-syscall template to automatically generate interface test cases.

In the interface-guided fuzzing stage, RPG uses Golang and llvm-link to construct the Linux distribution kernel CFG. This graph illustrates the functional relationships and control flows within the distribution kernel. RPG uses Golang and Python to modify Syzkaller's execution components. After each test case execution, RPG will use Python to determine its category, collect the function nodes covered by the test case, calculate the distance between these nodes and the control group nodes, and generate a priority value for the test case based on the distance using Equation 1. RPG utilizes Golang and Python to modify Syzkaller's generation components, enabling the fuzzer to prioritize generating test cases with higher priority values.

V. EVALUATION

To fully evaluate the effectiveness of RPG in enhancing Linux distribution kernel fuzzing, we conducted a series of experiments on four Linux distribution kernels. First, we exhibit RPG's vulnerability detection capabilities by listing previously unknown bugs and presenting two typical bug cases found through RPG. Second, we demonstrate RPG capabilities to explore more execution paths and kernel state space by comparing the coverage between RPG and other existing tools. Finally, we evaluate the contributions of each component to its ability to explore the distribution kernel state space and discover previously unknown vulnerabilities. We designed experiments to answer the following questions.

- **RQ1**: How does RPG perform in vulnerability detection?
- **RQ2**: How does RPG perform in exploring the distribution kernel state space compared to existing tools?
- **RQ3**: What is the contribution of each component to RPG?

TABLE II: Previously Unknown bugs Detected by RPG.

Source File	bug Function	bug Type	Status	Distro-Specific?
1. fs/ext4/mballoc.c	ext4_mb_release_inode_pa()	kernel bug	confirmed	✓
2. kernel/workqueue.c	pwq_dec_nr_in_flight()	general protection fault	confirmed	
3. fs/ext4/balloc.c	ext4_validate_block_bitmap()	deadlock	fixed	
4. mm/gup.c	gup_vma_lookup()	data race	fixed	
5. mm/memory.c	sys_io_submit()	deadlock	fixed	
6. mm/process_vm_access.c	sys_process_vm_writev()	deadlock	fixed	
7. fs/ext4/mballoc.c	mb_avg_fragment_size_order()	memory leak	fixed	✓
8. sound/core/seq/seq_memory.c	snd_seq_info_pool()	slab-use-after-free	reported	✓
9. io_uring/io_uring.c	io_init_req()	out-of-bounds	reported	✓
10. fs/fs_struct.c	chroot_fs_refs()	null-ptr-deref	confirmed	✓
11. kernel/sched/core.c	try_to_wake_up()	null-ptr-deref	fixed	✓
12. fs/ext4/extents.c	ext4_ext_map_blocks()	kernel bug	fixed	✓
13. mm/page_alloc.c	__alloc_pages_nodemask()	memory leak	fixed	✓
14. mm/memory.c	handle_mm_fault()	deadlock	reported	
15. mm/page-writeback.c	bdi_ratio_from_pages()	divide error	reported	
16. fs/ext4/mballoc.c	mb_mark_used()	kernel bug	reported	
17. block/blk-mq-tag.c	__blk_mq_get_tag()	data race	reported	✓
18. mm/vmscan.c	throttle_direct_reclaim()	kernel bug	reported	✓
19. drivers/input/input.c	input_unregister_device()	general protection fault	confirmed	✓
20. fs/buffer.c	bdev_getblk()	data race	confirmed	✓
21. lib/stackdepot.c	stack_depot_save_flags()	kernel bug	confirmed	
22. net/core/datagram.c	__skb_try_recv_from_queue()	general protection fault	confirmed	✓

A. Experiment Setup

We evaluated RPG on four Linux distribution kernels: Ubuntu 22.04, Fedora 42, OpenAnolis 8.8 and OpenAnolis 23.1. By extracting runtime parameters from these four different Linux distribution kernels and their corresponding mainline Linux kernels, we produce the results shown in Table I. Furthermore, we generated interface test cases for each distribution kernel and constructed their respective function CFGs.

TABLE I: The runtime parameters interface differences between the four Linux distribution kernels and their corresponding mainline Linux kernel.

Distribution Version	Mainline Version	Distribution Interface	Mainline Interface	Specific Interface
Ubuntu 22.04	6.8.12	22592	16706	**12168**
Fedora 42	6.14.4	25933	17500	**13750**
OpenAnolis 23.1	6.6.25	21512	16872	**8473**
OpenAnolis 8.8	5.10.134	21627	16867	**8073**

For RQ1, all kernels were compiled with default configurations, enabling `CONFIG_KCOV` to collect code coverage information and `CONFIG_KASAN` to detect memory corruption errors [40], [41]. For RQ2, we compared RPG with Syzkaller and Healer, focusing on branch coverage achieved in the same amount of time. For RQ3, we disabled the parameter range generation component and the test case guidance component of RPG, respectively, to implement two variants of RPG. We evaluate the contribution of each component to the RPG by comparing the branch coverage achieved and the number of unknown vulnerabilities triggered across the different variants at the same time.

Our experiments were conducted on a high-performance server equipped with a 256-core AMD EPYC 7742 CPU and 256 GiB of memory. For bug detection, RPG was run continuously for one week on each distribution kernel version. For other comparisons, we run each fuzzer with the same QEMU configurations (2 VM, with 2 CPU for each fuzzing instance) for 24 hours, each repeated five times, and following the best practices for fuzzing evaluation [42].

B. Bug Finding

To answer RQ1 and evaluate RPG's bug detection capabilities in Linux distribution kernels, we collected and analyzed previously unknown bugs discovered by RPG. As shown in Table II, RPG identified 22 previously unknown bugs in four distribution kernels. Of these, 15 have been confirmed by kernel maintainers (bugs #1-7, #10-13, #19-22), and 10 have already been fixed (bugs #3-7, #11-13, #20, #22). Notably, 13 are distribution-specific bugs (bugs #1, #7-13, #17-20, #22).

Bug Analysis: Among all the bugs, the file system module and the memory management module show the highest proportion of bugs, with each module having seven bugs. The file system module exposed a series of critical errors,including kernel bugs (bugs #1, #12, #16), a deadlock (bug #3), a memory leak (bug #7), a null pointer dereference (bug #10), and a data race (bug #20). Similarly, the bugs in the memory management module include a data race (bug #4), deadlocks (bugs #5, #6, #14), a memory leak (bug #13), a divide error (bug #15), and a kernel bug (bug #18). RPG successfully triggered these vulnerabilities through its runtime parameter interface test cases. These test cases directly interact with the file system and memory management modules and can alter kernel execution flows during fuzzing, thereby helping to detect deeper kernel bugs.

```
//1.Create globally shared pa structure
static long syz_proconfig_set__sys_fs_ext4_sda_
mb_group_prealloc()
{
   set("mb_group_prealloc", "0");
}
//2.Relax metadata write ordering constraints
   mount_options = "data=writeback";
//3.Race condition scenario
   CPU_A: ext4_mb_release_inode_pa(pa);
   // Releasing pa structure
   CPU_B: ext4_mb_use_inode_pa(pa);
   // using pa structure
//4.Vulnerability trigger function
static noinline_for_stack void
ext4_mb_release_inode_pa(struct ext4_buddy *e4b,
   struct buffer_head *bitmap_bh,
   struct ext4_prealloc_space *pa)
{...
  // When pa structure is in release process but
      not marked deleted
  bug_ON(pa->pa_deleted == 0);
  % // pa_deleted==0 is detected causing assertion
      failure,kernel panic
...}
```

Fig. 6: Runtime Parameter Bug Case.

Meanwhile, it should be noted that among the 22 bugs found by RPG, 13 (Bugs #1, #7, #8-13, #17-20, #22) are distribution-specific bugs. These bugs can only be triggered on the distribution kernel and cannot be triggered on the corresponding mainline Linux kernel. RPG successfully triggers these distribution-specific bugs by bootstrap fuzzers to generate more generic test cases close to distribution-specific interface code, which often contains a large number of distribution-private code and code not enabled by the mainline by default. Next, we will analyze the following two typical bug cases to highlight the unique attributes of RPG in detecting distribution kernel bugs.

Runtime Parameter Bug Case: We present bug #1 to illustrate the effectiveness of RPG generating runtime parameter interface test cases in discovering deep logical bugs, as depicted in Figure 6. RPG first sets `mb_group_prealloc=0` through the interface test case `syz_proconfig_set_sys_fs_ext4_sda_mb_group_preallloc()`, which disables per-CPU preallocation pools, forcing all CPU cores to share the same global preallocation structure `pa` (lines 1-6). When the filesystem is mounted with the `data=writeback` option, metadata write ordering constraints are relaxed, allowing asynchronous write operations that significantly extend the concurrency window (lines 7-8). During file system stress testing, a race condition occurs (lines9-13): CPU A begins executing the release operation `ext4_mb_release_inode_pa(pa)` while CPU B is simultaneously using the same `pa` structure via `ext4_mb_use_inode_pa(pa)`. Due to insufficient synchronization mechanisms, when CPU A enters the release function (lines 19-23), it detects `pa->pa_deleted == 0`(line 21), but in reality the structure is being released. This state inconsistency triggers the `bug_ON()` assertion failure, causing kernel panic.

```
//1.Craft SQE with extended opcode
   sqe->opcode = CUSTOM_OPCODE_BEYOND_LAST;
//2.Specific interface entry
static int io_init_req(struct io_ring_ctx *ctx,
                  struct io_kiocb *req,
                  const struct io_uring_sqe *sqe)
{
//3.Read user-supplied opcode(Specific)
   req->opcode = READ_ONCE(sqe->opcode);
//4.Vulnerable bounds check
   if (unlikely(io_op_defs[req->opcode].
   not_supported ||
   req->opcode >= IORING_OP_LAST))
      return -EINVAL;
//5.Specific feature validation
   if ((sqe_flags & IOSQE_BUFFER_SELECT) &&
      !io_op_defs[req->opcode].buffer_select)
      return -EOPNOTSUPP;
   ...
}
```

Fig. 7: Distribution-Specific Bug Case.

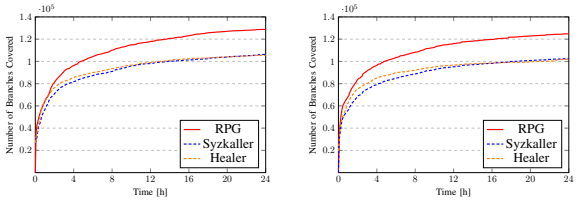
This vulnerability demonstrates how RPG uses the runtime parameter interface test case to discover state management errors deep in the kernel.

Distribution-Specific Bug Case: We use bug #9 as an example to demonstrate the distribution-specific bug. As illustrated in Figure 7. This bug is triggered by the customized implementation of the `io_uring` module in the Anolis OS kernel, resulting in a global out-of-bounds read. RPG triggers this bug by creating an SQE with an extended opcode value that exceeds the mainline Linux kernel's `IORING_OP_LAST` limit (lines1-2). When processing this SQE, the kernel reads the user-supplied `opcode` (line 8) and uses it to index into the `io_op_defs` array. In the Anolis OS kernel, this interface was extended to support custom operations like `IORING_OP_CUSTOM_DMA_READ`, but without updating the validation checks (lines 11-13). The opcode passes the initial bounds check (`req->opcode >= IORING_OP_LAST`) because it is within the extended range supported by the distribution kernel, but exceeds the original array dimensions. When performing buffer selection validation (lines 16-17), the kernel accesses `io_op_defs[req->opcode]` at an out-of-bounds index (line 18), resulting in a global out-of-bounds read that could leak sensitive kernel data or cause memory corruption. RPG triggers this vulnerability by generating more generic test cases that are close to the distribution-specific interface code, which is often associated with distribution-private code.

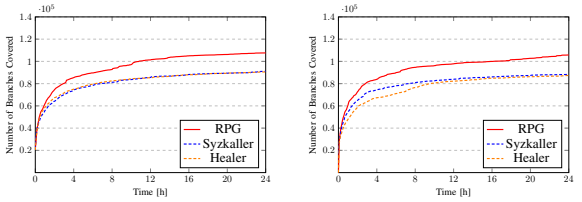
C. Coverage Improvement

To answer RQ2 and demonstrate the effectiveness of RPG in exploring wider execution paths in distributed kernels, we compare the branch code coverage achieved by RPG, Syzkaller, and Healer under the same conditions.

Coverage analysis: Detailed branch coverage statistics are shown in Table III. In fact, RPG achieves the highest branch coverage in all evaluated distribution kernel versions. Specif-

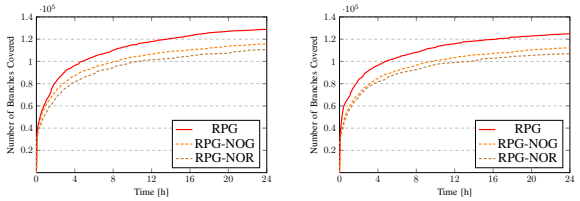


(a) Coverage on Ubuntu 22.04 (b) Coverage on Fedora 42

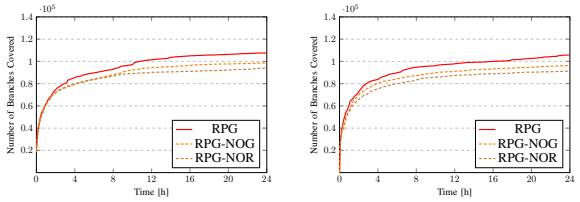


(c) Coverage on OpenAnolis 8.8 (d) Coverage on OpenAnolis 23.1

Fig. 8: Coverage of RPG, Syzkaller, and Healer.



(a) Coverage on Ubuntu 22.04 (b) Coverage on Fedora 42



(c) Coverage on OpenAnolis 8.8 (d) Coverage on OpenAnolis 23.1

Fig. 9: Coverage of RPG and Its Variants.

ically, RPG achieved branch coverage of 128815, 124833, 107644, and 116760, respectively, on the four distribution kernels of Ubuntu 22.04, Fedora 42, OpenAnolis 23.1, and OpenAnolis 8.8, with an average coverage of 116760. Compared with Syzkaller and Healer, the average coverage of RPG increased by 20.4% and 21.2%, respectively. We observed that Syzkaller and Healer have roughly the same branch coverage performance in the distribution kernel. This is because Healer is implemented based on the old version of Syzkaller, which means that it doesn't have a significant advantage when facing constantly updated Syzkaller.

TABLE III: Branch Coverage Comparison of RPG, Healer, and Syzkaller on Four Distribution Kernels

Distribution	RPG	Syzkaller	Healer
Ubuntu22.04	128815	106305(+21.2%)	105494(+22.1%)
Fedora42	124833	102447(+21.8%)	101945(+22.4%)
Anolis8.8	107644	91027(+18.3%)	90529(+18.9%)
Anolis23.1	105748	88289(+19.8%)	87452(+20.9%)
Average	116760	97017(+20.4%)	96355(+21.2%)

Furthermore, we plot the coverage growth curves of RPG in comparison to Syzkaller and Healer, depicted in Figure 8. A common trend observed is an initial rapid increase in branch coverage for all fuzzers within the first few hours, after which the growth rate begins to stabilize. It is worth noting that RPG can grow faster and consistently achieves significantly higher overall coverage throughout the fuzzing process compared to Syzkaller and Healer. This superior performance stems from two of RPG's core mechanisms. First, RPG extends runtime parameter interface test cases, which can help RPG alter kernel execution paths and states. Second, during the fuzzing process, RPG constantly guides the fuzzer to concentrate test resources on code near the distribution-specific interface. By combining these targeted resources with its generated interface test cases, RPG can explore deeper distribution kernel code space. Traditional fuzzers are difficult to cover these areas because they cannot generate runtime parameter interface test input, resulting in lower code coverage.

D. Component Contribution Analysis

The enhancements of RPG to the distribution kernel fuzzing come from two key components: the parameter range generation component and the test case guidance component. Therefore, to evaluate the contribution of each component to the RPG and answer RQ3, we designed and implemented two variants of the RPG:

- RPG-NOR: Disable the parameter range generation component.
- RPG-NOG: Disable the test case guidance component.

We conducted a 24-hour coverage experiment using two RPG variants on four distribution kernels. Then, we collect the code coverage achieved and unknown vulnerabilities triggered by RPG and its two variants. We will analyze the contribution of each component to the RPG based on these information.

Component Coverage analysis: The coverage growth curves presented in Figure 9, which illustrates the code coverage achieved by RPG and its variants in four distribution kernels. It can be seen that the performance of RPG has consistently been superior to its variants, achieving the highest growth rate in coverage and overall coverage. The margin between RPG and RPG-NOG and the margin between RPG-NOG and RPG-NOR indicate the impact of different components on the fuzzing performance.

The detailed coverage statistics in Table IV provide a comparison of the code coverage achieved by RPG and its variants. Specifically, RPG-NOR achieved branch coverage of 115688, 112305, 98575, and 96326, respectively, on the four distribution kernels of Ubuntu 22.04, Fedora 42, OpenAnolis 8.8, and OpenAnolis 23.1, with an average coverage of

TABLE IV: Branch Coverage Comparison of RPG and Its Variants on Four Distribution Kernels

Distribution	RPG	RPG-NOG	RPG-NOR
Ubuntu22.04	128815	115688(+11.4%)	110628(+16.4%)
Fedora42	124833	112305(+11.1%)	106982(+16.7%)
Anolis8.8	107644	98575(+9.2%)	93935(+14.6%)
Anolis23.1	105748	96326(+9.8%)	91272(+15.8%)
Average	116760	105761(+10.4%)	100829(+15.8%)

105761. Similarly, RPG-NOG obtains branch coverage of 110628, 106982, 93935, and 91272 for the same kernel versions, with an average coverage of 100829.

Compared to RPG-NOR, RPG and RPG-NOG achieve higher coverage, with an average improvement of 15.8% and 4.9%, respectively. This gain is primarily attributed to RPG's parameter range generation component, which can generate appropriate ranges for each runtime parameter. This can reduce the generation of invalid test inputs and improve the efficiency of fuzzing. The RPG-NOR don't have this component, resulting in it taking longer to get the same overall branch coverage.

TABLE V: Previously Unknown Bug Number Triggered by RPG and Its Variants.

Distribution	RPG	RPG-NOR	RPG-NOG
Ubuntu22.04	6.9	5.3(1.30×)	4.7(1.47×)
Fedora42	7.4	5.8(1.28×)	5.1(1.45×)
Anolis8.8	7.3	5.5(1.33×)	4.6(1.59×)
Anolis23.1	6.8	4.9(1.39×)	4.2(1.62×)
Average	7.1	5.4(1.32×)	4.7(1.53×)

Component Bug Trigger Statistics: We further counted the number of previously unknown bugs triggered by RPG and its two variants within one day (repeated five times). Detailed data are shown in Table V. Specifically, RPG, RPG-NOR, and RPG-NOG trigger 7.1, 5.4, and 4.1 unknown errors on average, respectively. RPG and RPG-NOR triggered significantly more bugs than RPG-NOG, achieving average improvements of 53% and 21%. This performance advantage stems from RPG's interface test case guidance component, which concentrates test resources on the distribution-specific interface code areas. And these areas often lack systematic testing and thus are more susceptible to revealing unknown bugs. RPG-NOG lack of this component results in a lower number of unknown bugs triggered.

Overall, the parameter range generation component enhances the RPG's ability to explore the distribution kernel code space. And the test case generation guidance component expands the capabilities of RPG in vulnerability detection.

VI. Lesson Learned

1. Distribution Kernels Require Distribution-Aware Fuzzing A key lesson we learned is that applying fuzzers designed for mainline kernels directly to Linux distributions is insufficient [16]. Distribution kernels often include vendor-specific code extensions and configuration divergences. These changes are often undocumented and deeply integrated, making them difficult to capture using existing tools. Our experience shows that tools like Syzkaller and Healer, when directly ported, fail to focus fuzzing effort on these critical differences. As a result, a large number of distribution kernel security vulnerabilities have been missed. Future fuzzing efforts must incorporate mechanisms to automatically detect and prioritize these variant code paths—for example, through static diffing, symbol set comparison, or runtime coverage contrast with mainline kernels.

2. Runtime Parameter Fuzzing Is Promising but Requires Deeper Semantics We also learned that runtime parameters significantly shape kernel behavior and are a powerful axis for fuzzing exploration. Parameters control a wide range of kernel subsystems, such as memory management and networking, I/O scheduling that are difficult to reach [43]. RPG demonstrates the potential of combining LLM and tuning databases to automatically generate interface test cases. However, its current focus on primitive types, such as int, overlooks more complex parameter types, including strings, structs, and enums, which are common in real-world applications. Furthermore, RPG cannot currently identify when two identically named parameters exhibit divergent behaviors between mainline and distribution kernels, due to differing kernel constraints. This limits its ability to capture subtle but critical bugs.

VII. Conclusion

In this paper, we propose RPG, a fuzzing tool guided by distribution-specific runtime parameter interfaces. RPG leverages the differences in runtime parameter interfaces between distribution kernels and the mainline Linux kernel to enhance the performance of fuzzing tools on distribution kernels. We evaluated RPG on four Linux distribution kernels: Ubuntu 22.04, Fedora 42, OpenAnolis 8.8, and OpenAnolis 23.1. RPG has detected 22 previously unknown bugs (15 confirmed, 10 fixed), and 13 out of 22 were distribution-specific bugs. RPG also achieved 20.4% and 21.2% higher branch coverage than Syzkaller and Healer, respectively. The results show that RPG can effectively detect bugs in the distribution kernels and significantly enhances distribution kernel security.

Acknowledgments

This research is supported in part by NSFC Program (No.62472448, 62202500), Alibaba Group through Alibaba Innovative Research Program, National Key R&D Program of China (No.2022YFB3104003), Hunan Provincial Natural Science Foundation (No.2023JJ40772), Changsha Science and Technology Key Project (No. kh2401027), Hunan Provincial 14th Five-Year Plan Educational Science Research Project (No.XJK23AJD022), Ministry of Education Industry-University Cooperation Collaborative Education Project (No.220500643274437), and High Performance Computing Center of Central South University.

REFERENCES

[1] T. L. K. Organization, "The kernel's command-line parameters," 2023. [Online]. Available: https://www.kernel.org/doc/html/latest/admin-guide/kernel-parameters.html

[2] S. Hasanov, S. Nagy, and P. Gazzillo, " A Little Goes a Long Way: Tuning Configuration Selection for Continuous Kernel Fuzzing ," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 521–533. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSE55347.2025.00042

[3] Linux, "Cve-2025-22053," 2025, https://www.cve.org/CVERecord?id=CVE-2025-22053.

[4] M. Bhurtel and D. B. Rawat, "Unveiling the landscape of operating system vulnerabilities," *Future Internet*, vol. 15, no. 7, p. 248, 2023.

[5] J. Choi, K. Kim, D. Lee, and S. K. Cha, "Ntfuzz: Enabling type-aware kernel fuzzing on windows with static binary analysis," in *42th IEEE Symposium on Security and Privacy, SP 2021*, 2021, pp. 677–693.

[6] T. Yin, Z. Gao, Z. Xiao, Z. Ma, M. Zheng, and C. Zhang, "KextFuzz: Fuzzing macOS kernel EXTensions on apple silicon via exploiting mitigations," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 5039–5054. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/yin

[7] H. Han and S. K. Cha, "Imf: Inferred model-based fuzzer," in *24th ACM SIGSAC Conference on Computer and Communications Security, CCS 2017*, ser. CCS '17. New York, NY, USA: ACM, 2017, p. 2345–2358. [Online]. Available: https://doi.org/10.1145/3133956.3134103

[8] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding Kernel Race Bugs through Fuzzing," in *40th IEEE Symposium on Security and Privacy, SP 2019*. IEEE, 2019, pp. 754–768. [Online]. Available: http://dblp.uni-trier.de/db/conf/sp/sp2019.html#JeongKSLS19

[9] D. Maier, B. Radtke, and B. Harren, "Unicorefuzz: On the viability of emulation for kernelspace fuzzing," in *13th USENIX Workshop on Offensive Technologies (WOOT 19)*. Santa Clara, CA: USENIX Association, Aug. 2019. [Online]. Available: https://www.usenix.org/conference/woot19/presentation/maier

[10] M. Fleischer, D. Das, P. Bose, W. Bai, K. Lu, M. Payer, C. Kruegel, and G. Vigna, "ACTOR: Action-Guided kernel fuzzing," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 5003–5020. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/fleischer

[11] kernelslacker, "Trinity: Linux system call fuzzer," 2012. [Online]. Available: https://github.com/kernelslacker/trinity

[12] Q. Zhang, Y. Shen, J. Liu, Y. Xu, H. Shi, Y. Jiang, and W. Chang, "Ecg: Augmenting embedded operating system fuzzing via llm-based corpus generation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 11, pp. 4238–4249, 2024.

[13] B. Ruan, J. Liu, C. Zhang, and Z. Liang, "Kernjc: Automated vulnerable environment generation for linux kernel vulnerabilities," in *Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses*, ser. RAID '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 384–402. [Online]. Available: https://doi.org/10.1145/3678890.3678891

[14] H. Shi, R. Wang, Y. Fu, M. Wang, X. Shi, X. Jiao, H. Song, Y. Jiang, and J. Sun, "Industry practice of coverage-guided enterprise linux kernel fuzzing," in *27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019*. ACM, 2019, pp. 986–995. [Online]. Available: https://doi.org/10.1145/3338906.3340460

[15] J. Liu, Y. Shen, Y. Xu, and Y. Jiang, "Leveraging binary coverage for effective generation guidance in kernel fuzzing," in *31th ACM SIGSAC Conference on Computer and Communications Security, CCS 2024*, ser. CCS '24. New York, NY, USA: ACM, 2024, p. 3763–3777. [Online]. Available: https://doi.org/10.1145/3658644.3690232

[16] H. Shi, S. Chen, R. Wang, Y. Chen, W. Zhang, Q. Zhang, Y. Shen, X. Shi, C. Hu, and Y. Jiang, "Industry practice of directed kernel fuzzing for open-source linux distribution," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 2159–2169.

[17] H. Sun, Y. Shen, J. Liu, Y. Xu, and Y. Jiang, "KSG: Augmenting kernel fuzzing with system call specification generation," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 351–366. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/sun

[18] A. C. K. Developers, "[5.10] [feature]anolis parameter example," 2023. [Online]. Available: https://gitee.com/anolis/cloud-kernel/pulls/1803/files

[19] D. Vyukov and A. Konovalov, "Syzkaller: an unsupervised coverage-guided kernel fuzzer," 2015. [Online]. Available: https://github.com/google/syzkaller

[20] H. Sun, Y. Shen, C. Wang, J. Liu, Y. Jiang, T. Chen, and A. Cui, *HEALER: Relation Learning Guided Kernel Fuzzing*. New York, NY, USA: ACM, 2021, p. 344–358. [Online]. Available: https://doi.org/10.1145/3477132.3483547

[21] R. van Riel, "Documentation for /proc/sys," 1999. [Online]. Available: https://docs.kernel.org/admin-guide/sysctl/index.html

[22] Q. Wang, R. Wang, Y. Hu, X. Shi, Z. Liu, T. Ma, H. Song, and H. Shi, "Keentune: Automated tuning tool for cloud application performance testing and optimization." Association for Computing Machinery, 2023, p. 1487–1490.

[23] C. Ltd., "Enterprise open source and linux — ubuntu," 2025. [Online]. Available: https://ubuntu.com/

[24] O. Community, "Openanolis," 2025. [Online]. Available: https://openanolis.cn/

[25] V. J. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "Fuzzing: Art, science, and engineering," *arXiv preprint arXiv:1812.00140*, 2018.

[26] L. McDonald, M. I. U. Haq, and A. Barkworth, "Survey of software fuzzing techniques."

[27] Y. Shen, H. Sun, Y. Jiang, H. Shi, Y. Yang, and W. Chang, "Rtkaller: State-Aware Task Generation for RTOS Fuzzing," *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 5s, sep 2021. [Online]. Available: https://doi.org/10.1145/3477014

[28] Y. Shen, Y. Xu, H. Sun, J. Liu, Z. Xu, A. Cui, H. Shi, and Y. Jiang, "Tardis: Coverage-guided embedded operating system fuzzing," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 41, no. 11, p. 4563–4574, nov 2022. [Online]. Available: https://doi.org/10.1109/TCAD.2022.3198910

[29] C. Zhou, M. Wang, J. Liang, Z. Liu, and Y. Jiang, "Zeror: Speed up fuzzing with coverage-sensitive tracing and scheduling," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 858–870.

[30] C. Zhou, Q. Zhang, M. Wang, L. Guo, J. Liang, Z. Liu, M. Payer, and Y. Jiang, "Minerva: browser api fuzzing with dynamic mod-ref analysis," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1135–1147.

[31] C. Zhou, Q. Zhang, L. Guo, M. Wang, Y. Jiang, Q. Liao, Z. Wu, S. Li, and B. Gu, "Towards better semantics exploration for browser fuzzing," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA2, pp. 604–631, 2023.

[32] C. Zhou, B. Qian, G. Go, Q. Zhang, S. Li, and Y. Jiang, "Polyjuice: Detecting mis-compilation bugs in tensor compilers with equality saturation based rewriting," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, pp. 1309–1335, 2024.

[33] D. Vyukov and A. Konovalov, "Syzlang: System call description language," 2015. [Online]. Available: https://github.com/google/syzkaller/blob/master/docs/syscall_descriptions_syntax.md

[34] ——, "Syzkaller: Pseudo syscalls documentation," 2023. [Online]. Available: https://github.com/google/syzkaller/blob/master/docs/pseudo_syscalls.md

[35] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *39th IEEE Symposium on Security and Privacy, SP 2018*. IEEE Computer Society, 2018, pp. 711–725.

[36] J. Gao, Y. Xu, Y. Jiang, Z. Liu, W. Chang, X. Jiao, and J. Sun, "Em-fuzz: Augmented firmware fuzzing via memory checking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3420–3432, 2020.

[37] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "HFL: Hybrid Fuzzing on the Linux Kernel," in *NDSS*, 2020.

[38] M. Wang, J. Liang, Y. Chen, Y. Jiang, X. Jiao, H. Liu, X. Zhao, and J. Sun, "SAFL: Increasing and Accelerating Testing Coverage with Symbolic Execution and Guided Fuzzing," in *40th IEEE/ACM International Conference on Software Engineering, ICSE 2018*, ser. ICSE '18. New York, NY, USA: ACM, 2018, p. 61–64. [Online]. Available: https://doi.org/10.1145/3183440.3183494

[39] H. Naveed, A. U. Khan, S. Qiu, M. Saqib, S. Anwar, M. Usman, N. Akhtar, N. Barnes, and A. Mian, "A comprehensive overview of large language models," *arXiv preprint arXiv:2307.06435*, 2023.
[40] Google, "Kernel address sanitizer," 2014, https://www.kernel.org/doc/html/latest/dev-tools/kasan.html.
[41] SimonKagstrom, "Kcov," 2015, https://github.com/SimonKagstrom/kcov.
[42] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 2123–2138. [Online]. Available: https://doi.org/10.1145/3243734.3243804
[43] S. Bai, Z. Zhang, and H. Hu, "Countdown: Refcount-guided fuzzing for exposing temporal memory errors in linux kernel," in *31th ACM SIGSAC Conference on Computer and Communications Security, CCS 2024*, ser. CCS '24. New York, NY, USA: ACM, 2024, p. 1315–1329. [Online]. Available: https://doi.org/10.1145/3658644.3690320