Directed Real-time Linux Fuzzing with Configuration Awareness

YUHENG SHEN[1], JIANZHONG LIU[1], YUHAN CHEN[2], QIANG ZHANG[3], RUNZHE WANG[4], HEYUAN SHI[2], and YU JIANG[1]*, [1] KLISS, BNRist, School of Software, Tsinghua University, China, [2] Central South University, China, [3] Hunan University, China, and [4] Alibaba Group, China

Rt-Linux is a branch of the Linux kernel with modifications to meet real-time requirements, whose components are often less tested due to their deep integration with kernel logic and their dependence on certain configurations, which complicate traditional testing approaches. These challenges make it difficult to achieve comprehensive coverage of *Rt-Linux*'s unique code sections, leaving systems vulnerable to bugs. We present *DRLF-C*, a configuration-aware directed fuzzer tailored for *Rt-Linux*, enabling efficient testing of its unique code sections. *DRLF-C* constructs a kernel-level weighted callgraph to prioritize input sequences closer to the target code and integrates static and dynamic configurations to test in realistic scenarios. Evaluations show that *DRLF-C* achieves a 24.70% increase in target coverage compared to *Syzkaller* and discovers 13 previously unknown bugs, now fixed by the *Rt-Linux* team. *DRLF-C* is currently integrated into Alibaba's CI/CD pipeline of operating systems for continuous testing.

CCS Concepts: • **Security and privacy → Operating systems security**.

Additional Key Words and Phrases: Fuzz Testing, Real-time Linux, Bug Detection

ACM Reference Format:
Yuheng Shen[1], Jianzhong Liu[1], Yuhan Chen[2], Qiang Zhang[3], Runzhe Wang[4], Heyuan Shi[2], and Yu Jiang[1]. 2025. Directed Real-time Linux Fuzzing with Configuration Awareness. 1, 1 (December 2025), 21 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Real-Time Linux (*Rt-Linux*) [21] is a derivative of the Linux kernel designed to provide deterministic response times for various timing-critical tasks, making it ideal for applications requiring strict timing constraints. *Rt-Linux* achieves this by integrating a real-time module within the kernel, enabling it to prioritize tasks effectively and ensure predictable execution under diverse workloads. This unique capability has made *Rt-Linux* an essential component in many industry scenarios, including automotive, aerospace, industrial automation, and medical devices. These industries often customize real-time features to fit their specific needs, ensuring they meet high standards for performance and reliability. As the demand for more diverse real-time applications continues to grow, the *Rt-Linux* codebase has expanded in complexity to accommodate enhanced functionalities and improved performance. While this evolution has enabled *Rt-Linux* to meet the challenges of modern applications, it has also introduced significant challenges in maintaining the robustness, reliability, and security of the codebase. The real-time nature of *Rt-Linux* amplifies the consequences

*Heyuan Shi and Yu Jiang are corresponding authors.

Authors' address: Yuheng Shen[1], shenyh20@mails.tsinghua.edu.cn; Jianzhong Liu[1], liujz21@mails.tsinghua.edu.cn; Yuhan Chen[2], Chenyuhan@csu.edu.cn; Qiang Zhang[3], zhangqiang9413@126.com; Runzhe Wang[4], runzhe.wrz@alibaba-inc.com; Heyuan Shi[2], hey.shi@foxmail.com; Yu Jiang[1], jiangyu198964@126.com[1], KLISS, BNRist, School of Software, Tsinghua University, Beijing, China [2] and Central South University, China [3] and Hunan University, China [4] and Alibaba Group, Hangzhou, China.

Manuscript submitted to ACM

of potential bugs, as even minor oversights can lead to catastrophic results. In critical domains, such as aviation or healthcare, any failure in *Rt-Linux* could result in severe consequences, including financial losses, compromised safety, and, in extreme cases, the loss of human life.

Given the critical importance of *Rt-Linux*'s reliability and security, the industry has adopted numerous testing methods to ensure its robustness. Traditional approaches, such as unit testing, have been widely employed to validate individual system components and interactions. However, these methods often fail to uncover deeper, system-wide vulnerabilities that could emerge under unexpected or extreme conditions. Fuzz testing, also known as fuzzing, has emerged as a powerful and complementary testing technique. The main idea of fuzzing is to generate a large number of malformed test cases for the system under test. Its ability to uncover vulnerabilities has drawn interest from both academics and industry. Fuzzing, by far, has been widely deployed in kernel testing domains. These kernel fuzzers typically generate a set of system call sequences as test input, use network stack or shared memory to send them into the target kernel, and use diverse monitors to detect any abnormal behaviors. A typical example of a kernel fuzzer is *Syzkaller* [18]. It generates test payloads based on detailed system call specifications written by domain experts. The fuzzer then generates system call sequences based on these specifications. Once the payloads are executed, *Syzkaller* monitors the kernel for signs of unexpected behavior, such as crashes, hangs, or incorrect responses. By far, *Syzkaller* has been integrated into the Linux kernel's continuous integration and continuous delivery (CI/CD) pipeline and uncovered numerous critical bugs within it.

While upstream kernels are rigorously tested through CI/CD pipelines, *Rt-Linux*'s real-time specific code often lacks the same level of scrutiny, making it error-prone due to its customization for diverse real-time requirements. Testing the entire kernel code is impractical in industrial scenarios where resource constraints and time limitations exist. As a result, focusing testing efforts on *Rt-Linux*-specific features is critical for efficiently identifying potential defects. This customized code usually needs certain static configurations to enable or disable and is related to certain dynamic configurations, which can be used to fine-tune system behavior or trigger specific features. These configurations play a pivotal role in determining the behavior of *Rt-Linux* features and can significantly impact the system's performance and reliability. Therefore, to conduct efficient testing on *Rt-Linux*'s unique code sections, we first need to consider *Rt-Linux*' related configuration during fuzzing and then direct the testing procedure towards the target code sections. In this way, we can explore *Rt-Linux*'s state space more effectively, ensuring that the fuzzer tests the system under diverse scenarios and ensuring thorough evaluation of these high-priority areas.

However, conducting directed fuzzing on *Rt-Linux* with configuration awareness, we encounter several challenges. *First, leveraging real-time-related configurations is challenging.* These configurations, both static and dynamic, are scattered across different system layers, including kernel structures and runtime settings, and can change dynamically during execution. Traditional fuzzing approaches fail to incorporate these configurations effectively, as they lack mechanisms for systematically collecting and using them to generate meaningful test cases. Without leveraging these configurations, fuzzers struggle to target real-time-specific scenarios and accurately simulate runtime behavior. *Second, directing fuzzing towards real-time-related code is complex.* Directed fuzzing requires precise measurements of the differences and directions between execution traces and the target code within *Rt-Linux*, which cannot be applied directly using conventional methods, as the current coverage analyzer cannot handle such information. This is further complex since we need to utilize the distance and direction information to generate input payloads that test the target code, which requires certain strategies that are tailored for *Rt-Linux*.

To address these challenges, we propose *DRLF-C*, a configuration-aware kernel fuzzer designed to efficiently test specific code sections in *Rt-Linux*. In detail, *DRLF-C* collects static configurations for the target code. Then, based on

these static configurations, *DRLF-C* extracts and selects those dynamic configurations that are related to target code sections. Furthermore, *DRLF-C* constructs a weighted callgraph using kernel-level code analysis and dynamic execution traces. This callgraph maps the relationships between kernel code blocks and identifies the distance to the target patch or module. The weights are calculated based on execution relevance and coverage data, providing a precise framework for directing fuzzing efforts. Finally, *DRLF-C* conducts configuration-aware and directed fuzzing. Leveraging the weighted callgraph, *DRLF-C* chooses those inputs that are closer to the target code to generate. Also, each test case will combine a related dynamic configuration to ensure the test is conducted under a specific configuration. This ensures that the fuzzer generates and prioritizes test cases that are more likely to trigger the target code under different configuration settings, thus improving the efficiency and effectiveness of the fuzzing process.

We evaluated *DRLF-C* against *Syzkaller*, and the results demonstrate that *DRLF-C* significantly improves the efficiency of directed fuzzing. Specifically, for the targeted code sections, *DRLF-C* achieves faster coverage convergence compared to *Syzkaller*, effectively reducing the time required to reach critical execution paths. Additionally, *DRLF-C* attains a 24.70% increase in overall coverage statistics. Furthermore, *DRLF-C* discovered 13 previously unknown bugs in *Rt-Linux*. Each of these vulnerabilities has been confirmed or fixed by the corresponding maintainers.

Contributions: In summary, this paper makes the following contributions:

- We propose *DRLF-C*, a directed kernel fuzzer tailored for *Rt-Linux*, that efficiently tests any specific code region within *Rt-Linux*.
- We introduce configuration and directed fuzzing strategies. Using extracted kernel configurations and a weighted callgraph, we can accurately measure the distance between code blocks in the kernel and guide the fuzzing process towards the target code.
- The evaluation results show that *DRLF-C* achieves a 24.70% coverage improvement compared to *Syzkaller*, and found 13 previously unknown bugs within *Rt-Linux*.

2 BACKGROUND

2.1 *Rt-Linux*

Real-Time Linux (*Rt-Linux*) [21] is a Linux kernel derivative tailored for real-time systems, focusing on deterministic execution to meet strict timing constraints. By integrating the RT_PREEMPT patch, *Rt-Linux* minimizes latency for high-priority tasks and prevents interruptions from non-deterministic operations. Its APIs for priority scheduling, timer management, and interrupt handling enable precise task execution, making *Rt-Linux* critical in industries like automotive, aerospace, and industrial automation. While *Rt-Linux*'s flexibility allows for customization to meet specific real-time requirements, this also increases its complexity, introducing potential vulnerabilities. As real-time applications expand, errors in *Rt-Linux* could result in severe consequences, such as system failures or safety risks, highlighting the need for advanced testing approaches like directed fuzzing to uncover and address bugs in its complex codebase.

Its overall architecture can be referred to in Figure 1. In detail, *Rt-Linux* consists of several modules, including the scheduler, task management, timer and clock management, interrupt handling, memory management, inter-process communication, and networking. These modules enable real-time functionalities by managing key operations like task scheduling, priority handling, IRQ balancing, and CPU affinity. The scheduler ensures that high-priority tasks execute with minimal latency, while the timer and interrupt management modules provide precise timing and efficient handling

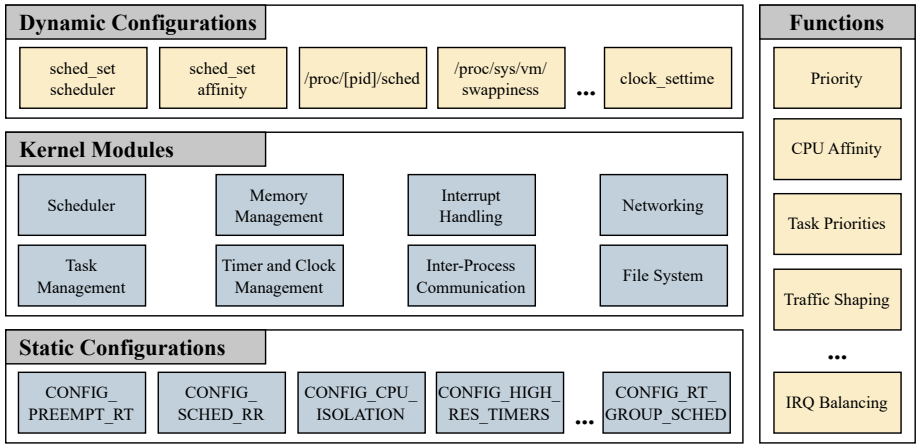


Fig. 1. Architecture of *Rt-Linux*. To enable the Linux kernel's real-time functionalities, we need to enable static configuration during compile time. This ensures that kernel modules such as scheduler, task management, timer, and interrupt handler support real-time features. This can be further fine-tuned using dynamic configurations during runtime.

of hardware interrupts. To enable *Rt-Linux*'s real-time capabilities, certain static configurations must be set during compilation, such as `CONFIG_PREEMPT_RT`, `CONFIG_SCHED_RR`, `CONFIG_CPU_ISOLATION`, and `CONFIG_HIGH_RES_TIMERS`. These configurations ensure low-latency operation and robust task prioritization. At runtime, dynamic configurations such as sched_setscheduler and sched_setaffinity allow fine-tuning of task priorities and CPU assignments. Additional adjustments, like modifying `/proc/[pid]/sched` or kernel parameters (e.g., `/proc/sys/vm/swappiness`), further enhance real-time performance. Together, these configurations and modules provide the flexibility and precision required for reliable real-time operation while emphasizing the need for advanced testing techniques to ensure robustness in such complex systems.

2.2 Fuzz Testing

Fuzz testing, also known as *fuzzing*, is a dynamic software testing technique. Its primary objective is to feed the System-Under-Test (SUT) with large amounts of randomly generated inputs, thus attempting to trigger bugs within the SUT. The inputs used in fuzzing can probe potential weak points of the system, ranging from minor service interruptions to significant security violations. Due to its effectiveness in finding bugs, fuzzing is extensively used in various areas, such as databases and browsers, to enhance the safety and reliability of software applications [5, 6, 9, 19, 20].

Kernel Fuzzing is an application of fuzzing that feeds random or semi-random inputs into the kernel's interfaces and system call surface to identify vulnerabilities [16]. This technique has been increasingly adopted to test operating system kernels [8, 10, 11, 13–16], which are critical components of modern computing systems. By probing kernel functionalities with diverse and unexpected inputs, fuzzing helps uncover security flaws, stability issues, and logic errors in areas that are otherwise difficult to test comprehensively. For *Rt-Linux*, which is designed to operate within specific time constraints, it is particularly susceptible to disruptions originating from unexpected behaviors. As such, fuzzing these kernels is extremely beneficial to uncovering any anomalous behavior caused by bugs. It ensures not only the identification of vulnerabilities but also the consistent performance of systems, where timely responses are essential. For example, RtKaller [12] uses state-aware fuzzing strategies to fix potentially damaged test input, therefore

achieving effective *Rt-Linux* fuzzing. However, it targets the entire system, cannot conduct directed fuzzing on target code, and can hardly leverage configuration during fuzzing.

Directed Fuzzing is a derivative method of fuzzing that targets specific code sections within the SUT. As this is beneficial for testing newly added or recently modified code in an already-extensively-tested system, there have been many attempts to integrate directed fuzzing into CI/CD pipelines in the industry to better save resources and ensure product safety. In detail, a directed fuzzer typically generates inputs targeting specific code locations in target programs. It uses both the control flow graph and the callgraph of the program under test to determine the distance from any known basic block to the target basic block. Subsequently, it uses this distance metric to guide further input generation. In the realm of kernel fuzzing, *Syzkaller* offers the syz-filter module. This module utilizes a coverage bitmap, showing the coverage status between the current fuzzing input and the target code location. By excluding system calls that don't pertain to the target code section, significantly amplifies directed fuzzing efficiency.

3 MOTIVATION AND CHALLENGES

Most of the *Rt-Linux*'s codebase is derived from the upstream Linux kernel, which has undergone rigorous security testing. However, the real-time-related code in *Rt-Linux* is unique and often customized by various developers and vendors to meet diverse real-time requirements. These customizations frequently involve static (to enable) and dynamic (to fine-tune) configurations, which are critical to the system's behaviors. Unfortunately, this customized code does not receive the same level of scrutiny and testing as the upstream kernel. This challenge is further amplified in industrial environments, where testing is constrained by stringent time and computational resources. To address this, testing *Rt-Linux*, particularly its newly added features and configuration-dependent real-time-relevant code, requires leveraging these configurations during testing. By adapting directed fuzzing to the kernel fuzzing domain, we can focus testing efforts on the code influenced by specific configurations. By providing the fuzzer with the position of the target code and configuration details, we can direct its efforts more effectively, thereby increasing the likelihood of uncovering vulnerabilities in these critical sections.

Taking CVE-2021-47553 as a motivating example [7]. As shown in Figure 2, this bug is located in the kernel's scheduler module. It emulates real-world CPU hotplugging scenarios, where idle tasks undergo multiple hotplug cycles. Such cycles expose stale KASAN shadow poisoning and stale Shadow Call Stack pointers, which can lead to KASAN warnings and memory leaks over time.

To trigger this error, we need to compile the kernel with static configuration PREEMPT_RT, KASAN_STACK, and KASAN_STACK opened. During testing, we need to configure the system's dynamic configuration, as shown in lines 2-7. As we can see, many bugs are hidden within the kernel's code logic. Simply covering its code will not trigger such an error; only with proper configuration can trigger such a bug. Furthermore, such code is often hidden deep within the kernel's code path, and random fuzzing can hardly trigger such bugs; we need specific input generation strategies to steer the fuzzing toward target code sections. However, to conduct such fuzzing, we need to address the following two challenges:

Extract and Leverage the Configuration Information. Fuzzing real-time systems, such as *Rt-Linux*, requires effectively extracting and utilizing configurations that influence system behavior. These configurations include static settings, such as compile-time options (CONFIG_KASAN, CONFIG_FIFO), and dynamic configurations (e.g., writing values to sysfs). Spread across kernel structures, runtime environments, and user-space components, these configurations often change during execution, further complicating their collection and integration. As illustrated in the

```
// Bug reproducer
while true; do
  for C in /sys/devices/system/cpu/cpu*/online; do
    echo 0 > $C;
    echo 1 > $C;
  done
done

// Corresponding fix patch
diff --git a/kernel/cpu.c b/kernel/cpu.c
--- a/kernel/cpu.c
+++ b/kernel/cpu.c
@@ -588,6 +589,12 @@ static int bringup_cpu(unsigned int cpu)
  int ret;
+ scs_task_reset(idle);
+ kasan_unpoison_task_stack(idle);

diff --git a/kernel/sched/core.c b/kernel/sched/core.c
--- a/kernel/sched/core.c
+++ b/kernel/sched/core.c
@@ -8641,9 +8641,6 @@ void __init init_idle(struct task_struct *idle, int cpu)
  idle->flags |= PF_IDLE | PF_KTHREAD | PF_NO_SETAFFINITY;
  kthread_set_per_cpu(idle, cpu);
- scs_task_reset(idle);
- kasan_unpoison_task_stack(idle);
```

Fig. 2. Bug Reproducer and Related Patch of Bug CVE-2021-47553.

motivating example, the bug requires specific static configurations (PREEMPT_RT, KASAN_STACK) to enable relevant code paths and runtime dynamic configurations (/sys/devices/system/cpu/cpu*/online) to trigger the error state. Current fuzzing techniques fail to systematically handle such configuration data, limiting their ability to explore diverse system states. Without leveraging these settings, fuzzers struggle to uncover bugs tied to specific configurations, resulting in reduced coverage and missed opportunities for effective testing. Addressing this challenge requires integrating both static and dynamic configuration data into the fuzzing process. By targeting configurations critical to real-time scenarios, fuzzers can explore a broader range of system behaviors, enhancing their ability to identify bugs in complex environments.

Conduct directed Fuzzing on Target Code Section. To perform effective directed fuzzing, it is critical to accurately measure the difference and direction between the current execution trace and the target code within the *Rt-Linux*. As we can see from the motivating example, the buggy code is located deep within *Rt-Linux*'s code logic. To cover the example's error state, the fuzzer needs to randomly generate a large number of test cases to explore the kernel's code space. To speed up such process, we can use directed fuzzing, where fuzzer leverages the distance between target code and current execution path To speed up such process, we can use directed fuzzing, where the fuzzer leverages the distance between the target code and the current execution path to guide the input generation. However, given the vast code space of *Rt-Linux*, discerning the relationship between the current input and the target code during execution is hard. Moreover, in a system as intricate as the *Rt-Linux*, any misjudgment in this measurement can direct the fuzzing process incorrectly, resulting in degraded fuzzing effectiveness. Thus, a precise approach is required to measure the difference and direction before initiating the fuzzing, ensuring a more informed and effective fuzzing process. Furthermore, based on the distant information, the fuzzer needs to generate inputs that can reach and cover this region. This requires an effective method to compute the distance between the target code region and the execution trace for any

given input. Using this information during input generation allows the fuzzer to generate and mutate seeds that trigger code blocks closer to the target code regions. Eventually, this will direct the fuzzing process to test the intended code region, thus achieving our design goals.

4 DESIGN

We propose *DRLF-C*, a configuration-aware directed kernel fuzzer designed to test specific code regions within *Rt-Linux*. The overall workflow of *DRLF-C* is illustrated in Figure 3. For a given kernel source code, *DRLF-C* begins by analyzing the patch source files to identify relevant static and dynamic configurations. Based on the extracted static configurations, *DRLF-C* filters and selects target-specific dynamic configurations, which are incorporated into the payload during input generation. Next, *DRLF-C* performs callgraph extraction during the kernel's compilation process. It constructs the kernel's control graph, identifies the address of each basic block, and calculates a weighted callgraph by correlating the control graph with the target code's address. During the directed fuzzing process, *DRLF-C* generates system call sequences based on the identified configurations and executes them on the system under test (SUT). It collects execution coverage traces, calculates weights for each input sequence using the weighted callgraph, and prioritizes seeds in the corpus based on their proximity to the target code. In subsequent iterations, *DRLF-C* selects the highest-priority inputs for mutation, continuously steering the fuzzing process toward the target code region. By combining configuration awareness with directed fuzzing, *DRLF-C* effectively targets specific regions of the kernel, ensuring efficient bug detection and enhanced testing coverage.

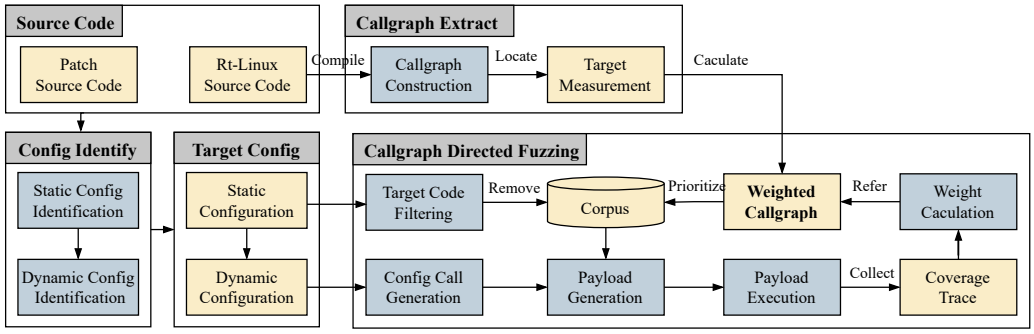


Fig. 3. The overview workflow of *DRLF-C*. Based on the patch code, *DRLF-C* extracts *Rt-Linux*'s dynamic and static configuration. Based on the kernel code, *DRLF-C* generates the callgraph at compiling time and based on the target code position to calculate the weighted callgraph. Then, during the fuzzing phase, *DRLF-C* first select target code-related dynamic configurations and incorporate them during payload generation for *Rt-Linux* kernel. Within each execution iteration, *DRLF-C* collects the corresponding coverage trace. Based on the weighted callgraph and the coverage trace, *DRLF-C* calculates the weight for each input and consequently chooses the highest-priority input for mutation.

4.1 Configuration-Awared Fuzzing

To enable configuration-aware fuzzing, *DRLF-C* first extracts both static and dynamic configurations from the *Rt-Linux* kernel. This required us to first analyze the kernel source code to extract both dynamic and static configurations, then identify the dynamic configurations related to the static configurations, and last, generate a test payload that contains the extracted configurations.

4.1.1 Static Configuration Extraction. Static configurations are compile-time options that determine the inclusion and behavior of various kernel features. However, such configuration information is scattered across different parts of the kernel source code, including header files, configuration files, and inline code segments. To extract these configurations, *DRLF-C* scans the given patch's corresponding file and identifies relevant configuration options. For instance, *DRLF-C* scans for preprocessor directives (e.g., `#ifdef`, `#if defined`) and configuration macros (e.g., `CONFIG_PREEMPT_RT`, `CONFIG_IRQ_FORCED_THREADING`) that influence the inclusion and behavior of specific code sections. By analyzing these directives, *DRLF-C* can determine the static configurations required to enable certain features in the kernel.

```
// in kernel/irq/manager.c
#if defined(CONFIG_IRQ_FORCED_THREADING) && !defined(CONFIG_PREEMPT_RT)
DEFINE_STATIC_KEY_FALSE(force_irqthreads_key);

static int __init setup_forced_irqthreads(char *arg)
{
  static_branch_enable(&force_irqthreads_key);
  return 0;
}
early_param("threadirqs", setup_forced_irqthreads);
#endif
```

Fig. 4. Static Configurations Extraction.

Take Figure 4 for example. In this example, *DRLF-C* identifies the relevant configuration options. In detail, *DRLF-C* scans the related real-time patch and identifies that the `irq/manager.c` contains real-time related modifications. Then, it scans the `manager.c` for the configuration pattern, where it identifies that the kernel enables forced IRQ threading based on the configuration options `CONFIG_IRQ_FORCED_THREADING` and `CONFIG_PREEMPT_RT`. Such configuration will then be saved and loaded during compile time to ensure the related code will be compiled into the kernel.

4.1.2 Dynamic Configuration Extraction. Dynamic configurations are runtime settings that can be modified during the execution of the kernel. These include parameters set through sysfs, procfs, or other kernel interfaces. This configuration is typically concentrated in a certain directory but has different values (i.e., integer, string) and permission (i.e., writable, readable). To extract dynamic configurations from the kernel, *DRLF-C* scans the relevant directories and identifies configuration files and parameters. These directories contain various runtime settings that can be modified to influence the kernel's behavior.

In concrete, *DRLF-C* uses the following steps to extract dynamic configurations. First, *DRLF-C* scans the directories for files and parameters that are relevant to the target code section. This includes files in `/sys/kernel/`, `/proc/sys/`. Then, for each identified configuration file, *DRLF-C* checks the permission of the file; for that file that does not have write permission, it passes. Last, *DRLF-C* reads the current values and possible settings. This involves parsing the file contents to understand the range of acceptable values and their impact on the kernel's behavior. In this way, *DRLF-C* ensures that the fuzzing process explores a diverse set of system states, increasing the likelihood of uncovering configuration-specific bugs.

4.1.3 Configuration Filtering. Once the configurations are extracted, *DRLF-C* filters them to retain only those that are directly related to the target code section. This ensures that the fuzzing process focuses on configurations that meaningfully impact the behavior of the target code. The filtered configurations are then used to guide the input generation process, ensuring that the fuzzer explores a diverse set of system states.

Algorithm 1: Dynamic Configurations Filtering Algorithm

Input: StaticConfigs: Map(ConfigName, ConfigPath);
DynamicConfigs: Map(ConfigPath, ConfigValue);
Output: FilteredDynamicConfigs: Map(ConfigPath, ConfigValue);

```
StaticPaths ← ∅;
for each (ConfigName, ConfigPath) ∈ StaticConfigs do
    StaticPaths.push(ConfigPath);
FilteredDynamicConfigs ← ∅;
CommonSubstrCount ← 0;
for each (ConfigPath, ConfigValue) ∈ DynamicConfigs do
    for each StaticPath ∈ StaticPaths do
        CommonSubstrCount ← CountCommonSubstr(ConfigPath, StaticPath);
        if CommonSubstrCount ≥ 3 then
            FilteredDynamicConfigs.push(ConfigPath, ConfigValue);
return FilteredDynamicConfigs;
```

The detailed algorithm is shown in Algorithm 1. *DRLF-C* first collects all static configuration paths into a set for efficient lookup. Then, using the `CountCommonSubstr` function, the number of common consecutive substrings shared with the static configuration paths is calculated for each dynamic configuration. If the count meets or exceeds 3, the dynamic configuration is added to the filtered list. Specifically, this threshold ensures that a dynamic configuration path shares at least three consecutive components with a related static configuration path (e.g., /sys/module/rcupdate/-parameters/rcu_task_ipi_delay shares module, rcupdate, parameters with the RCU configuration family), which filters out trivial matches on generic prefixes such as /sys or /proc/sys while preserving legitimate correlations. By focusing on configurations that influence the behavior of the target code, *DRLF-C* ensures that the fuzzing process explores a diverse yet relevant set of system states. These filtered configurations are subsequently used to guide the generation of targeted inputs, enhancing the fuzzing effectiveness.

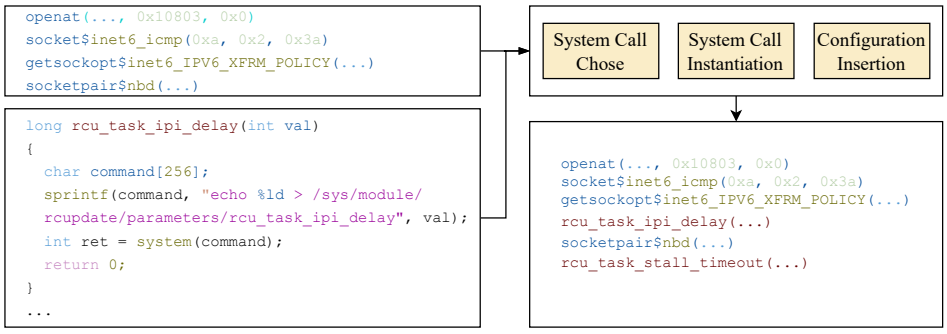


Fig. 5. Configuration-Awared Payload Generation.

4.1.4 Configuration-awared Input Generation. During the fuzzing process, *DRLF-C* incorporates the filtered configurations into the input generation phase to create more targeted and effective inputs. Specifically, the original payload, which consists of a system call sequence, is extended by integrating dynamic configuration adjustments. Using Syzkaller's pseudo call framework, *DRLF-C* randomly inserts configuration adjustment operations into the system call sequence. These adjustments involve modifying the dynamic configurations with random values and simulating various runtime states.

An example of the generation process can be referred to at Figure 5. In detail, on top of the original system call sequence, based on the extracted dynamic configuration, we write a set of configuration modification functions. These functions expose the related configuration value for *DRLF-C* to generate. we assign random values drawn from each interface's admissible domain, as such boolean, and interger ranges. During the payload generation, *DRLF-C* first generates the corresponding system call sequence and randomly chooses configuration modification functions from the filtered configuration set. Then, based on the original system call sequence, *DRLF-C* random inserts those configurations modification function, and therefore, modifies kernel's runtime config during fuzzing.

Algorithm 2: Callgraph Construction and Target Measurement for *Rt-Linux*

Input: $KernelSource$: *Rt-Linux* kernel source directory
Input: $TargetAddress$: Address of target code
Output: $WeightedCallgraph$: Callgraph with distances to target

Function *ConstructCallgraph(KernelSource)*:
    BitcodeFiles ← ∅;
    for each $File \in KernelSource$ **do**
        $AdjustCompileConfigurations(File)$;
        $GenerateLLVMBitcode(File)$;
        $BitcodeFiles.push(Bitcode(File))$;
    $Callgraph \leftarrow \emptyset$;
    for each $Bitcode \in BitcodeFiles$ **do**
        $Callgraph \leftarrow llvm\_link(Callgraph, Bitcode)$;
    return $Callgraph$;
Function *MeasureTarget(Callgraph, TargetAddress)*:
    $WeightedCallgraph \leftarrow Callgraph$;
    $Queue \leftarrow \{TargetAddress\}$;
    while $Queue \neq \emptyset$ **do**
        $CurrentNode \leftarrow Queue.pop()$;
        for each $Neighbor \in Neighbors(CurrentNode)$ **do**
            $Distance \leftarrow Distance(CurrentNode) + 1$;
            if $Distance < WeightedCallgraph[Neighbor]$ **then**
                $WeightedCallgraph[Neighbor] \leftarrow Distance$;
                $Queue.push(Neighbor)$;
    return $WeightedCallgraph$;

4.2 Callgraph Extraction

To effectively conduct directed fuzzing, the first step is to construct a weighted callgraph for the target *Rt-Linux*. This involves extracting the intricate control flow that spans the entire kernel, capturing the relationships and dependencies between various functions and basic blocks. The weighted callgraph serves as a foundation for guiding the fuzzing process, enabling precise navigation toward the target code. A key component of this construction is the computation of the distance between each basic block in the kernel and the target code location. This distance metric not only quantifies the proximity of different code regions to the target but also informs the prioritization of input sequences, ensuring that the fuzzing effort remains focused and efficient. By analyzing the control flow and incorporating distance information, the weighted callgraph can steer the fuzzer toward the target code region.

4.2.1 Callgraph Construction. For an *Rt-Linux* kernel under test, we first need to construct its callgraph. However, constructing the complete callgraph for the *Rt-Linux* kernel is complicated, especially considering the modular nature of *Rt-Linux* and its vast array of files. Therefore, to construct the system-wise callgraph, *DRLF-C* initiates the process by compiling the kernel with Clang, as referred at Algorithm 2, the function `ConstructCallgraph`. During this phase, *DRLF-C* adjusts the compile configurations across every tier of the *Rt-Linux* directory. This is to extract specific compilation commands and then modify them to dump the corresponding LLVM bitcode. Consequently, a bitcode file for each kernel file is generated. These bitcode files encapsulate the control flow of each file, represented as a set of interlinked basic blocks. Once all individual bitcodes are generated, we use *llvm-link* to link the bitcode files progressively in a bottom-up manner. The linking process aggregates the separate bitcode files, creating a unified and comprehensive representation of the kernel's control flow and structure. This detailed callgraph is essential for extracting the intricate control flow from the entire kernel.

4.2.2 Target Measurement. After capturing the control flow of the target kernel, we can then compute the distance from the target position to each basic block. This distance measurement is based on the number of linked edges within the callgraph. To achieve this, we start by disassembling the kernel binary, as shown in Algorithm 2, function `MeasureTarget`. This disassembly process reveals the address of our target code region, which could either be a specific function or a distinct basic block. With this address, we traverse the callgraph using Dijkstra's algorithm to iterate over each edge. We calculate the number of steps required to reach the target code from the current position for every iteration. This step count is then assigned as the weight of each edge.

4.3 Callgraph Guided Fuzzing

Once *DRLF-C* constructs the weighted callgraph, the guided fuzzing process begins, focusing on generating high-quality inputs that effectively target the desired code region within the *Rt-Linux* kernel. By leveraging the weighted callgraph, *DRLF-C* evaluates the weight of each input based on its proximity to the target region. This weight serves as a measure of the input's relevance for driving execution closer to the target code. Using the calculated weights, *DRLF-C* prioritizes inputs in the corpus and steers subsequent mutations toward those with the highest potential. This ensures that each iteration incrementally refines inputs to reduce the distance to the target, enabling the fuzzer to concentrate its efforts on the most critical paths. By dynamically guiding the mutation process, *DRLF-C* achieves efficient and precise exploration of the kernel code, enhancing the likelihood of uncovering bugs in the target region.

4.3.1 PC Weight Calculation. Within each fuzzing iteration, *DRLF-C* retrieves the current coverage bitmap from `KCOV`, which records the addresses of all basic blocks encountered during the test execution. This coverage bitmap serves

as a key input for assessing the progress of the fuzzing process. By leveraging this information, *DRLF-C* consults the weighted callgraph to determine the distance associated with each encountered basic block's address relative to the target code region. These distances are then used to evaluate the relevance and effectiveness of the executed input sequence. To quantify this relationship, *DRLF-C* employs a transformation defined in Equation 1 to convert the computed distance into a normalized weight value, referred to as the program priority:

$$\text{Weight Value} = \frac{\text{atan}(distance) + \frac{\pi}{2}}{\pi} \tag{1}$$

This formula ensures that the resulting weight values are bounded between 0 and 1, regardless of the actual distance. The bounded range provides consistency and stability in prioritizing inputs. Additionally, the use of the arctangent function emphasizes shorter distances, as its growth rate slows significantly for larger distances. This property naturally prioritizes input sequences that are closer to the target code, ensuring that the fuzzing process focuses on areas with higher relevance. By dynamically assigning higher priorities to inputs with shorter distances, *DRLF-C* can effectively direct the fuzzing process toward the target region while avoiding unnecessary exploration of unrelated code paths.

Algorithm 3: Distance-Guided Program Evaluation and Generation

Input: *Corpus*: Set of input programs;
CoverageBitmap: Coverage data from KCOV;
WeightedCallgraph: Map of basic blocks to distances;
Output: *SelectedProgram*: Input with the highest priority for mutation;

```
PrioritizedCorpus ← ∅;
for each Program ∈ Corpus do
    Coverage ← RetrieveCoverage(Program, CoverageBitmap);
    Priority ← 0;
    for each BasicBlock ∈ Coverage do
        Distance ← WeightedCallgraph[BasicBlock];
        Weight ← (atan(Distance) + π/2) / π;
        Priority ← Priority + Weight;
    Program.Priority ← Priority;
    PrioritizedCorpus.push(Program);
PrioritizedCorpus ← SortByPriority(PrioritizedCorpus);
SelectedProgram ← PrioritizedCorpus[0];
return SelectedProgram;
```

4.3.2 Distance-guided Program Evaluation. With the priorities determined, the next phase of distance-guided fuzzing takes over. Each program within our corpus is assigned a priority. During mutation, the fuzzer prefers programs with the highest priority. This strategic selection ensures that our fuzzing endeavors are consistently oriented toward the target region, maximizing the efficacy of our testing process.

The detailed algorithm is shown in Algorithm 3. Initially, each program in the corpus is evaluated by retrieving its coverage data from the KCOV bitmap, which enumerates the basic blocks executed during its run. For each basic block, the algorithm computes a weight using the arctangent-based formula, which normalizes the distance between the basic block and the target region into a value between 0 and 1. This prioritizes closer blocks while gradually reducing the

impact of distant ones. The total weight for a program is calculated by summing the weights of all its covered basic blocks, resulting in a priority score for the program. The corpus is then sorted in descending order of priority, and the program with the highest priority is selected for mutation. By dynamically choosing the most promising inputs for the next iteration, the algorithm ensures that fuzzing remains focused on the target region, maximizing the efficiency of the testing process. Also, the above process is separated from the main fuzzing loop to minimize performance overhead during each iteration.

4.4 Implementation

We implemented *DRLF-C* using Golang and Python, with some components borrowed from *Syzkaller*. To extract kernel's configuration, *DRLF-C* using python to analyze kernel's patch file and source code. To extract the kernel's callgraph, *DRLF-C* compiles the target kernel using Clang and emits its corresponding intermediate representation. Then, *DRLF-C* automatically constructs the weighted callgraph using the IR information, where the weights are based on the address of the target code. During fuzzing, *DRLF-C*'s generation, mutation, and coverage collection modules calculate the runtime weight for each input and prioritize all interested corpus.

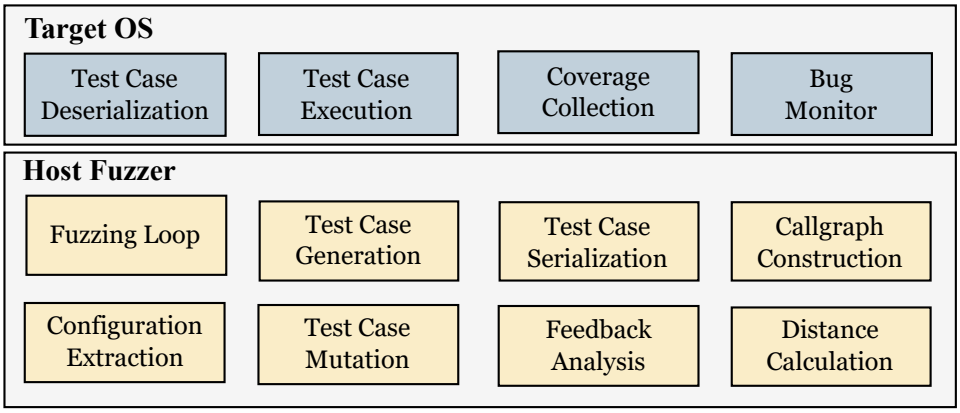


Fig. 6. Implementation Details of *DRLF-C*. The host fuzzer is in charge of fuzzing loop control, test case generation/mutation, corpus management, and feedback analysis and needs to extract configuration, construct, and maintain the callgraph. In the target OS, we run an execution agent, which is used to conduct test case deserialization, execution, coverage collection, and bug detection.

The overall implementation details can be referred to at Figure 6. Concretely, *DRLF-C* works on the host machine and the target OS, which interact within the fuzzing loop. The host fuzzer handles tasks such as configuration extraction, callgraph construction, test case generation, mutation, and feedback analysis. The fuzzing process starts with extracting configurations and constructing a weighted callgraph to guide the testing. In each iteration, *DRLF-C* generates test cases, executes them on the Target OS, collects coverage data, and analyzes feedback to prioritize inputs. This ensures that mutations are focused on driving execution closer to the target code region. Additionally, the Bug monitor tracks crashes and anomalies, enabling the discovery of vulnerabilities efficiently. To scale to the kernel, we extract dynamic configuration interfaces and target addresses, build a pruned call graph of reachable objects, and cache per-object IR for incremental relinking. Distance computation then runs on this reduced graph, yielding weighted guidance signals

while greatly reducing analysis overhead without sacrificing precision. By far, *DRLF-C* is integrated into Alibaba's continuous fuzz testing pipeline called *ABACI* Robot. Changes to a specific part of the kernel trigger the CI/CD process, which invokes the automatic script that generates the weighted graph. Subsequently, the result is sent to the fuzzer, which uses this information to focus its testing on the part of the kernel that needs to be tested.

5 EVALUATION

We list the following research questions to help us understand *DRLF-C*'s performance and effectiveness.

- Whether *DRLF-C* is capable of uncovering new bugs.
- Can *DRLF-C* achieves a better performance than existing methods?

Experiment Setup. We tested *DRLF-C* on eight versions of the *Rt-Linux* kernel, namely 5.10, 5.11, 5.14, 5.19, 6.10, and 6.13. We compare the overall coverage statistic between *DRLF-C* and *Syzkaller* to demonstrate the effect introduced by configuration-aware fuzzing. We choose the functions within `io_uring` and network module as the directed fuzzing target due to its significance in real-time operations. Specifically, the `io_uring` uses two lock-free ring buffers: one for managing submission entries, allowing concurrent request handling, and another for completion events. The primary system call used is `io_uring_submit()`, designed for efficient multi-operation handling, making it highly relevant for real-time operations. For networking, it is a de facto real-time path, since packets must be processed within tight softirq/NAPI budgets to meet latency and QoS guarantees. To showcase the effectiveness of configuration and directed fuzzing, we compare *DRLF-C* with *Syzkaller* and vanilla **DRLF**. We instrument the target kernel with KCOV for coverage collection, with Kernel Address SANitizer (KASAN) and Kernel Concurrency SANitizer (KCSAN) enabled for bug detection. We perform our evaluation on an AMD EPYC 7742 CPU at 2.25GHz with 64 cores running Ubuntu 20.04. All experiments are conducted on the same hardware for 24 hours, with each experiment repeated three times to establish statistical significance.

Table 1. Previously Unknown Bugs Detected by *DRLF-C*

Idx	Versions	Modules	Operations	Risk	Status
1	5.11	io_uring	__io_clean_op	logic error	fixed
2	5.11	io_uring	__io_req_task_submit	deadlock	fixed
3	5.11	io_uring	__io_uring_sq	null ptr deref	fixed
4	5.11	io_uring	io_cqring_overflow_flush	logic error	fixed
5	5.11	io_uring	io_uring_poll	deadlock	fixed
6	5.10	io_uring	io_sq_thread_stop	deadlock	fixed
7	5.10	io_uring	__io_clean_op	null ptr deref	fixed
8	5.10	io_uring	io_wq_submit_work	deadlock	fixed
9	5.10	io_uring	io_uring_create	out of bound	fixed
10	5.10	io_uring	io_commit_cqring	double free	fixed
11	5.10	io_uring	io_rw_reissue	logic error	fixed
12	6.13	memory	bdi_set_min_bytes	logic error	confirmed
13	6.13	kernel	max_vclocks_store	logic error	confirmed

5.1 Bug Detection Capabilities

Found New Bugs. *DRLF-C* found a total of 13 previously unknown bugs, with 11 in the target module `io_uring`, as listed in Table 1. Within the 13 bugs, 4 of which are memory-related, 4 of which are concurrency-related, whereas the rest are logic bugs. All listed bugs have been confirmed or fixed by kernel maintainers.

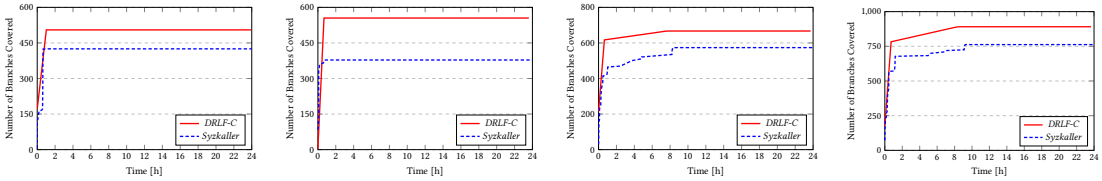
***Rt-Linux* Related Bugs.** *DRLF-C* identified 13 bugs in *Rt-Linux*, with 11 directly located within the targeted `io_uring` module, and 2 in the kernel module, that triggered by `io_uring` operations. This precision is attributed to our directed fuzzing approach, which directs the fuzzing process toward the target area and ensures comprehensive and in-depth testing of the target code. Furthermore, the bugs we found are critical, as many are concurrency-related, causing potential slowdowns or hangs, which is especially detrimental to a system like *Rt-Linux*, where the timing and the system performance are of great importance. In detail, many of the bugs we found are concurrency-related bugs, whereas in industrial scenarios, any slowdown or system hang can have severe consequences.

```

static int io_cqring_wait(struct io_ring_ctx *ctx, ...) {
  ...
  trace_io_uring_cqring_wait(ctx, min_events);
  do {
    io_cqring_overflow_flush(ctx, false, NULL, NULL);
    prepare_to_wait_exclusive(&ctx->wait, &iowq.wq,
           TASK_INTERRUPTIBLE);
  ...
}

static void io_cqring_overflow_flush(struct io_ring_ctx *ctx, ...)
{
    if (test_bit(0, &ctx->cq_check_overflow)) {
        /* iopoll syncs against uring_lock, not completion_lock */
        if (ctx->flags & IORING_SETUP_IOPOLL)
            mutex_lock(&ctx->uring_lock);
        __io_cqring_overflow_flush(ctx, force, tsk, files);
        if (ctx->flags & IORING_SETUP_IOPOLL)
            mutex_unlock(&ctx->uring_lock);
    }
}
```

Fig. 7. Static Configurations Extraction.

Case Study. Here, take bug #4 in Table 1 to briefly describe a previously unknown kernel panic bug found by *DRLF-C* in the io_uring module as the case study to demonstrate the bug detection capability. As shown in Figure 7, this bug in `io_cqring_wait` arises from a violation of the `__might_sleep` condition, where a blocking operation (`mutex_lock`) is called while the task is in a non-running state (`TASK_INTERRUPTIBLE`). The issue occurs when `prepare_to_wait_exclusive` sets the task state to `TASK_INTERRUPTIBLE`, and `io_cqring_overflow_flush` subsequently acquires the uring_lock mutex under the `IORING_SETUP_IOPOLL` flag, leading to a kernel warning. Traditional fuzzers are unlikely to uncover this bug due to their inability to focus on specific configurations and code paths. In contrast, our tool uses configuration-aware input generation, systematically enabling relevant static and dynamic configurations like `IORING_SETUP_IOPOLL`, and employs a weighted callgraph to prioritize test cases that exercise paths closer to the target code. By dynamically adjusting configurations and leveraging feedback-guided prioritization, our

tool efficiently triggers this subtle bug, demonstrating its superiority in uncovering critical vulnerabilities in real-time kernel environments.



(a) Coverage on 5.10 (b) Coverage on 5.11 (c) Coverage on 5.14 (d) Coverage on 5.19

Fig. 8. Coverage Statistics for Syzkaller and *DRLF-C* on *Rt-Linux* Kernel Versions 5,10, 5.11, 5.14, and 5.19, respectively.

5.2 Comparison With Other Tools

Coverage Comparison on Overall Coverage. To further evaluate the effectiveness of *DRLF-C*, we compare the branch coverage within the target code section. The detailed statistics are listed in Table 2. As shown in the table, Syzkaller achieves an average of 476.6 branch coverage on the respective kernel versions, while *DRLF-C* achieves statistics of 582.1 branch coverage on the respective four *Rt-Linux* versions. In comparison, *DRLF-C* gains an average of 24.70% coverage improvement in the `io_uring` module. Also, we expanded the comparison to include a more recent kernel (versions 6.10 and 6.13) and an additional target module, the network subsystem. The results, as shown in Table 3, demonstrate *DRLF-C*is capable of out-performaning both Syzkaller and DRLF. *DRLF-C* achieves the highest branch coverage in all test scenarios.

Table 2. Coverage Comparison On 5.10, 5.11, 5.14, and 5.19 for Io_uring

Version	5.10	5.11	5.14	5.19	Average
Syzkaller	425.0	378.0	573.6	762.3	**534.7**
DRLF-C	**505.0**	**555.0**	**667.0**	**891.0**	**654.5**
Improvement	18.82%	46.83%	16.27%	16.88%	**24.70%**

Furthermore, we plot the coverage growth curve for kernels 5.10, 5.11, 5.14, and 5.19, as shown in Figure 8. As we can see, *DRLF-C* can achieve a higher code coverage at a faster speed compared to *Syzkaller*. However, for version 5.10, *DRLF-C* grew slower than the *Syzkaller* initially but gradually overtook Syzkaller at around 4 hours. This is because, after 6.1, `io_uring` has become a single file. Therefore, the search difficulties are grown for *DRLF-C*. The graph also shows that most of the coverage growth saturates at around 4-8 hours, which is because we only target the `io_uring` modules, resulting in a short time to coverage saturation. The above improvement is attributed to the Target-related code coverage.

Comparison on Covered Files. To conduct a more fine-grained coverage comparison, we analyzed the coverage on each file. Specifically, we choose the *Rt-Linux* v5.11, and we compare the *DRLF-C* with *Syzkaller*. We collect the coverage percentage for all the files within the `io_uring` module and compare the file coverage percentage between the *DRLF-C* and *Syzkaller*.

Table 3. Coverage Comparison on 6.10 and 6.13 for Io_uring and Network Modules

Target	network-6.10	network-6.13	io_uring-6.10	io_uring-6.13
DRLF-C	**5027**	**5017.6**	**736**	**808.6**
DRLF	4652.3 (+8.05%)	4638.3 (+8.18%)	665 (+10.68%)	678 (+19.26%)
Syzkaller	4371 (+15.01%)	4401.3 (+14.00%)	584.6 (+25.90%)	633.6 (+27.62%)

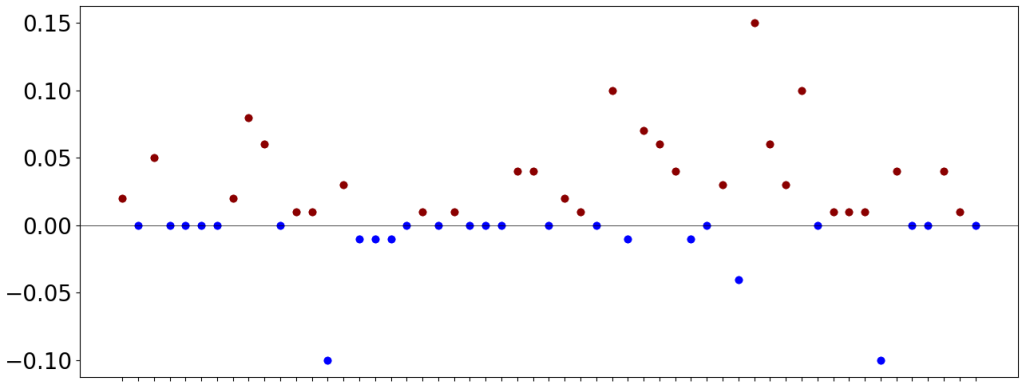


Fig. 9. Coverage Comparison for To-be-tested Files. The y-axis represents the percentage difference between *DRLF-C* and *Syzkaller*.

The results are shown in Figure 9, where positive values represent *DRLF-C* covers more code, and vice versa. We subtract the file coverage percentage between the *DRLF-C* and *Syzkaller* for each file, as we can find that, in most of the files, *DRLF-C* can cover more code compared to *Syzkaller*. We perceive that positive values are the majority, indicating that *DRLF-C* generally covers more code than *Syzkaller*, clearly demonstrating *DRLF-C*'s effectiveness in directed fuzzing. This contributes to the directed fuzzing technique that allows for a deeper test to the target code. For the instances where *Syzkaller* covers more than *DRLF-C*, our analysis shows that the source files do not contain the target code section and, therefor,e are not concerning to us. Also, we noticed that there are some files that *Syzkaller* outperform the *DRLF-C*; this is because these files do not contain the designated code section; therefore, *Syzkaller* have a better fuzzing performance in these files.

Comparison on Bug Detection. To validate the applicability of the *DRLF-C*, we collected all the vulnerabilities uncovered by both our testing method and Syzkaller within 24 hours. As shown in Table 4, Syzkaller discovers an average of 32 kernel vulnerabilities, but only 4.67 of these are related to real-time characteristics on average, with a real-time relevance rate of only 14%. In contrast, the *DRLF-C*, as shown in Table 5, discovers an average of 20.0 kernel vulnerabilities, of which 15.5 are related to the real-time characteristics of io_uring.

Table 4. Number of Vulnerabilities Discovered by Syzkaller

Rt-Linux	**5.10**	**5.11**	**5.14**	**5.19**	**6.10**	**6.13**	**Average**
Target-Related Bugs	5	6	3	5	3	5	**4.67**
Global Kernel Bugs	39	36	30	31	26	30	**32.00**

Table 5. Number of Vulnerabilities Discovered by *DRLF-C*

Rt-Linux	**5.10**	**5.11**	**5.14**	**5.19**	**6.10**	**6.13**	**Average**
Target-Related Bugs	11	19	16	13	15	19	**15.5**
Global Kernel Bugs	18	22	21	17	20	22	**20.00**

While Syzkaller excels at broadly uncovering kernel vulnerabilities, it struggles to identify those specifically related to real-time characteristics. This is because Syzkaller primarily focuses on coverage-guided exploration of the entire kernel rather than targeting real-time-specific code. Conversely, our testing method prioritizes the discovery of real-time-specific defects and differs from traditional coverage-guided testing. *DRLF-C* starts by analyzing kernel configuration, enabling real-time-related configurations and guiding the systematic construction of real-time-specific targets. It then builds a kernel function callgraph, assigning higher weights to basic blocks related to real-time characteristics, which helps focus the testing effort on relevant targets. As a result, the real-time relevance of the bug discovered by *DRLF-C* reaches 77.5%.

Comparison on Coverage. We further compare the overall coverage statistics with and without the extracted configuration. The detailed coverage statistic can be referred to at Table 6. As detailed in Table 6, *DRLF-C* outperforms Syzkaller, achieving an average coverage improvement of 5.59%. This improvement is particularly evident in kernel versions such as 6.10 and 6.13, where the improvement reaches 12.23% and 7.92%, respectively.

Table 6. Coverage Comparison on Configuration

Metric	**5.10**	**5.11**	**5.14**	**5.19**	**6.10**	**6.13**	**Average**
Syzkaller	14273.3	14811.0	15740.0	16092.3	17428.3	17811.6	**16026.8**
DRLF-C	**15138.6**	**15237.0**	**16335.3**	**16437.0**	**19559.3**	**19225.6**	**16922.5**
Improvement	6.07%	2.87%	3.78%	2.14%	12.23%	7.92%	**5.59%**

The reason for this improvement is because *DRLF-C* extracts and incorporates configuration information during the fuzzing process. By leveraging dynamic configurations, *DRLF-C* ensures that the target kernel is tested under realistic and diverse scenarios that are directly relevant to the code being analyzed. Unlike Syzkaller, which failed to consider the configuration, *DRLF-C* integrates configuration awareness into its input generation. This allows it to explore additional code paths activated by specific configurations, leading to broader and deeper coverage.

6 RELATED WORKS

6.1 Kernel Fuzzing

Fuzzing has been widely adopted in the kernel domain to enhance system security and ensure the integrity of kernel components. One of the most prominent kernel fuzzers is Syzkaller, which leverages KCOV for coverage-guided fuzzing on Linux and has been integrated into Linux's CI/CD pipeline for continuous testing. Several works have been proposed to improve the performance and effectiveness of kernel fuzzing. Moonshine [10] introduces a distillation algorithm that enhances the initial corpus by leveraging execution traces from real-world programs, refining test cases for better coverage. ECG [22] utilizes large language models (LLMs) to generate an optimized initial corpus, improving

the diversity of test inputs. For more advanced feedback mechanisms, Healer [16] employs relation learning to automatically extract dependencies between adjacent system calls, enabling the generation of higher-quality test cases that better reflect real-world execution patterns. Countdown [1] enhances input generation by analyzing kernel memory operations, effectively guiding the fuzzer to detect more Use-After-Free (UAF) vulnerabilities. However, these works target the entire kernel space and can hardly concentrate testing on very specific code.

6.2 Directed Fuzzing

Directed fuzzing is a technique for testing specific code regions within a target program. It involves guiding input generation toward predefined locations rather than randomly exploring the execution space. Various approaches have been developed to enhance its effectiveness. One common strategy is distance-based guidance, where fuzzers prioritize inputs that reduce the execution distance to the target code. AFLGo [2] pioneered this with distance metrics and simulated annealing for seed selection, while Hawkeye [3] refined it by integrating function similarity to explore alternative paths. WindRanger [4] improves precision by recognizing that not all execution paths contribute equally to reaching the target code. Existing fuzzers have also been incorporated with directed fuzzing. Syzkaller-covfilter enhances Syzkaller by introducing a selective coverage mechanism that concentrates fuzzing on specific sections. SyzDirect [17] improves kernel driver fuzzing by analyzing system call relationships and guiding test generation with relevant argument preparation. However, they mostly generate system call sequences as inputs, failing to consider the system's configuration and leaving a large kernel space untested.

7 DISCUSSION

7.1 System Adaptability

Beyond *Rt-Linux*, various embedded operating systems, such as FreeRTOS and Zephyr, support real-time functionalities. However, these OSs differ significantly from Linux-based systems, lacking unified interfaces and essential kernel infrastructure support (e.g., KCOV, KASAN). Currently, *DRLF-C* is tailored for *Rt-Linux* and struggles to extend to such embedded OS due to the absence of efficient fuzzing communication mechanisms. In the future, we aim to overcome these limitations by utilizing standardized interaction mechanisms, such as debug interfaces or network-based communication. These approaches would enable stable fuzzing data exchange between the fuzzer and the OS, facilitating effective fuzzing of a broader range of embedded real-time systems.

7.2 Configuration Relation

Rt-Linux contains approximately 3000 dynamic configurations, each of which plays a role in managing kernel runtime settings and system behavior. Efficiently exploring this vast configuration space is time-consuming, yet it is essential for effective fuzzing, as different configurations can significantly impact execution paths and introduce unique system behaviors. Currently, *DRLF-C* optimizes the search space by identifying and correlating static and dynamic configurations, ensuring that only those configurations directly influencing real-time functionalities are considered for fuzzing. In the future, we could further enhance the performance of *DRLF-C* by incorporating automated relation analysis to refine configuration selection, enabling more precise testing of real-time behaviors and uncovering subtle configuration-related bugs.

8 CONCLUSION

In this paper, we present *DRLF-C*, a configuration-aware directed kernel fuzzer tailored for *Rt-Linux*. Traditional kernel fuzzers can hardly conduct directed fuzzing *Rt-Linux*'s specific code section and often ignore the kernel's configuration, therefore leaving a large number of state spaces unexplored. However, *DRLF-C* first identifies relevant static configurations and dynamically adjusts kernel parameters during execution to ensure that fuzzing accurately reflects real-world runtime scenarios. Then, *DRLF-C* extracts kernel control flow and analyzes function dependencies, computing the shortest paths to the target regions, therefore effectively guiding the fuzzing process toward critical execution paths. Our evaluation shows that *DRLF-C* achieves a 24.70% increase compared to *Syzkaller*. Furthermore, *DRLF-C* detected 13 previously unknown vulnerabilities in *Rt-Linux*, all have been confirmed or fixed by maintainers.

9 ACKNOWLEDGMENTS

We thank the shepherd and reviewers for their valuable comments. This research is partly sponsored by NSFC Program (No. 62525207,62472448) and the National Key Research and Development Project (No. 2022YFB3104000).

REFERENCES

[1] Shuangpeng Bai, Zhechang Zhang, and Hong Hu. 2024. CountDown: Refcount-guided Fuzzing for Exposing Temporal Memory Errors in Linux Kernel. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 1315–1329. https://doi.org/10.1145/3658644.3690320

[2] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2329–2344. https://doi.org/10.1145/3133956.3134020

[3] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a Desired Directed Grey-box Fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 2095–2108. https://doi.org/10.1145/3243734.3243849

[4] Zhengjie Du, Yuekang Li, Yang Liu, and Bing Mao. 2022. Windranger: A Directed Greybox Fuzzer driven by Deviation Basic Blocks. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2440–2451. https://doi.org/10.1145/3510003.3510197

[5] lcamtuf. 2013. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/.

[6] J. Liang, M. Wang, C. Zhou, Z. Wu, Y. Jiang, J. Liu, Z. Liu, and J. Sun. 2022. PATA: Fuzzing with Path Aware Taint Analysis. In *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 154–170. https://doi.org/10.1109/SP46214.2022.00010

[7] Linux. 2021. CVE-2021-47553. https://www.cve.org/CVERecord?id=CVE-2021-47553.

[8] Jianzhong Liu, Yuheng Shen, Yiru Xu, Hao Sun, and Yu Jiang. 2023. Horus: Accelerating Kernel Fuzzing through Efficient Host-VM Memory Access Procedures. *ACM Trans. Softw. Eng. Methodol.* 33, 1, Article 11 (Nov. 2023), 25 pages. https://doi.org/10.1145/3611665

[9] Valentin JM Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2018. Fuzzing: Art, science, and engineering. *arXiv preprint arXiv:1812.00140* (2018).

[10] Shankara Pailoor, Andrew Aday, and Suman Jana. 2018. MoonShine: Optimizing OS Fuzzer Seed Selection with Trace Distillation. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 729–743. https://www.usenix.org/conference/usenixsecurity18/presentation/pailoor

[11] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 167–182. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo

[12] Yuheng Shen, Hao Sun, Yu Jiang, Heyuan Shi, Yixiao Yang, and Wanli Chang. 2021. Rtkaller: State-Aware Task Generation for RTOS Fuzzing. *ACM Trans. Embed. Comput. Syst.* 20, 5s, Article 83 (sep 2021), 22 pages. https://doi.org/10.1145/3477014

[13] Yuheng Shen, Yiru Xu, Hao Sun, Jianzhong Liu, Zichen Xu, Aiguo Cui, Heyuan Shi, and Yu Jiang. 2022. Tardis: Coverage-Guided Embedded Operating System Fuzzing. *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 41, 11 (nov 2022), 4563–4574. https://doi.org/10.1109/TCAD.2022.3198910

[14] Heyuan Shi, Runzhe Wang, Ying Fu, Mingzhe Wang, Xiaohai Shi, Xun Jiao, Houbing Song, Yu Jiang, and Jiaguang Sun. 2019. Industry practice of coverage-guided enterprise Linux kernel fuzzing. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 986–995. https://doi.org/10.1145/3338906.3340460

[15] Hao Sun, Yuheng Shen, Jianzhong Liu, Yiru Xu, and Yu Jiang. 2022. KSG: Augmenting Kernel Fuzzing with System Call Specification Generation. In *USENIX Annual Technical Conference*. https://api.semanticscholar.org/CorpusID:252494340

[16] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. 2021. *HEALER: Relation Learning Guided Kernel Fuzzing*. Association for Computing Machinery, New York, NY, USA, 344–358. https://doi.org/10.1145/3477132.3483547

[17] Xin Tan, Yuan Zhang, Jiadong Lu, Xin Xiong, Zhuang Liu, and Min Yang. 2023. SyzDirect: Directed Greybox Fuzzing for Linux Kernel. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda (Eds.). ACM, 1630–1644. https://doi.org/10.1145/3576915.3623146

[18] Dmitry Vyukov and Andrey Konovalov. 2015. Syzkaller: an unsupervised coverage-guided kernel fuzzer. https://github.com/google/syzkaller.

[19] Mingzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Han Liu, Xibin Zhao, and Jiaguang Sun. 2018. SAFL: Increasing and Accelerating Testing Coverage with Symbolic Execution and Guided Fuzzing. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 61–64. https://doi.org/10.1145/3183440.3183494

[20] Mingzhe Wang, Jie Liang, Chijin Zhou, Yu Jiang, Rui Wang, Chengnian Sun, and Jiaguang Sun. 2021. RIFF: Reduced Instruction Footprint for Coverage-Guided Fuzzing. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 147–159. https://www.usenix.org/conference/atc21/presentation/wang-mingzhe

[21] Victor Yodaiken and Michael Barabanov. 2001. RT-Linux. *White paper, Departament of Computer Science, New Mexico Institute of Technology, avalaibe at http://www. rtlinux. org/documents* (2001).

[22] Qiang Zhang, Yuheng Shen, Jianzhong Liu, Yiru Xu, Heyuan Shi, Yu Jiang, and Wanli Chang. 2024. ECG: Augmenting Embedded Operating System Fuzzing via LLM-Based Corpus Generation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43, 11 (2024), 4238–4249. https://doi.org/10.1109/TCAD.2024.3447220