Rtkaller: State-aware Task Generation for RTOS Fuzzing

YUHENG SHEN, HAO SUN, YU JIANG, HEYUAN SHI *, and YIXIAO YANG *, KLISS, BNRist, School of Software, Tsinghua University, China
WANLI CHANG, Hunan University, China

A real-time operating system (RTOS) is an operating system designed to meet certain real-time requirements. It is widely used in embedded applications, and its correctness is safety-critical. However, the validation of RTOS is challenging due to its complex real-time features and large code base.

In this paper, we propose `Rtkaller`, a state-aware kernel fuzzer for the vulnerability detection in RTOS. First, `Rtkaller` implements an automatic task initialization to transform the syscall sequences into initial tasks with more real-time information. Then, a coverage-guided task mutation is designed to generate those tasks that explore more in-depth real-time related code for parallel execution. Moreover, `Rtkaller` realizes a task modification to correct those tasks that may hang during fuzzing. We evaluated it on recent versions of rt-Linux, which is one of the most widely used RTOS. Compared to the state-of-the-art kernel fuzzers Syzkaller and Moonshine, `Rtkaller` achieves the same code coverage at the speed of 1.7X and 1.6X , gains an increase of 26.1% and 22.0% branch coverage within 24 hours respectively. More importantly, `Rtkaller` has confirmed 28 previously unknown vulnerabilities that are missed by other fuzzers.

CCS Concepts: • **Security and privacy → Virtualization and security**; **Domain-specific security and privacy architectures**; **Vulnerability scanners**; • **Computer systems organization** → *Real-time operating systems.*

Additional Key Words and Phrases: Fuzz Testing, RTOS, Vulnerability Detection, Task Generation

ACM Reference Format:
Yuheng Shen, Hao Sun, Yu Jiang, Heyuan Shi, Yixiao Yang, and Wanli Chang. 2021. Rtkaller: State-aware Task Generation for RTOS Fuzzing. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 17 (July 2021), 22 pages. https://doi.org/10.1145/3477014

1 INTRODUCTION

A Real-time Operating System (RTOS) is an operating system designed to serve those applications with certain real-time requirements and has demonstrated its growing importance in various industrial scenarios. As the RTOS is responsible for system-wide resource allocation and program scheduling, any vulnerabilities within the RTOS can jeopardize the system's security, leading the entire system to crash. However, the growing demand for real-time performance results in increasingly complex code logic, making new bugs frequently introduced and reported. These bugs can take many forms, ranging from simple memory corruptions to complex real-time errors.

*Heyuan Shi and Yixiao Yang are correspondence authors.

This article appears as part of the ESWEEK-TECS special issue and was presented in the International Conference on Embedded Software (EMSOFT), 2021.
Authors' addresses: Yuheng Shen, shenyh20@mails.tsinghua.edu.cn; Hao Sun, sun-h20@mails.tsinghua.edu.cn; Yu Jiang, jiangyu198964@126.com; Heyuan Shi, hey.shi@foxmail.com; Yixiao Yang, yangyixiaofirst@163.com, KLISS, BNRist, School of Software, Tsinghua University, Beijing, China; Wanli Chang, Hunan University, China, wanli.chang.rts@gmail.com.

1539-9087/2021/7-ART17 $15.00
https://doi.org/10.1145/3477014

Many widely used RTOS, such as FreeRTOS [31], and rt-Linux [2], have witnessed hundreds of bug reports in recent years, incurring severe consequences. Therefore, it is pivotal to ensure the code qualities of RTOS through vulnerability detection.

However, manually identifying bugs in the massive code base of an RTOS is challenging, as such an approach cannot adequately simulate the actual runtime situation and thoroughly explore the system states of an RTOS. Fuzzing [3, 11, 25] is a prominent approach to ensure the correctness of code. It provides random inputs to explore the program's abnormal behaviors. Some cutting-edge fuzzers, such as AFL [7] and Peach [32], have located millions of vulnerabilities on a wide range of real-world applications. Due to the tremendous performance of fuzzing, many researchers are attempting to apply fuzzing on operating systems testing.

The generic kernel fuzzing method tends to test the operating system interfaces based on the predefined system call descriptions (syscall SPEC) with randomly generated parameters. This methodology significantly increases kernel vulnerability detection efficiency and has exploited many critical kernel bugs [34]. Syzkaller [8] as a well-known kernel fuzzing tool developed by Google, represents such implementation. However, it is difficult to sufficiently expose the bugs in RTOS through the generic kernel fuzzing method due to the real-time related features. Specifically, for effectively fuzzing RTOS, there are mainly two challenges.

For the first challenge, the syscall SPEC used in the generic kernel fuzzing approach cannot convey detailed real-time information. In particular, some abnormal behaviors that are not incurred from the code errors may be mistaken as a crash due to the real-time features of RTOS. For instance, a process may be mistaken as a crash by unexpected abortion when other processes occupy some resources it requires for a long time. Since syscall SPEC only defines the system interface descriptions, it is incapable of conveying real-time related information such as priority and execution states. The lack of such information can lead to many false crashes during the fuzzing process, which seriously undermined the fuzzing speed.

For the second challenge, it is difficult for existing fuzzing methods to test real-time related code effectively. Many vulnerabilities are buried deep in the code logic and often require complex syscall combinations even under concurrent conditions to be triggered. However, the generic kernel fuzzing methods use overall code coverage to guide the test cases mutation and execute test cases in single-thread mode. Such approaches often result in failing to generate those test cases that can effectively test the RTOS, affecting the fuzzing performance.

To bridge the gap between fuzzing with RTOS vulnerability detection, we promote `Rtkaller`, a state-aware kernel fuzzer with coverage guidance that aims to trigger memory-related bugs (e.g. stack-overflow and use-after-free) extensively found in RTOS. Specifically, to express more real-time information, `Rtkaller` initializes tasks as test cases, each containing a concurrency intensity and multiple programs with priority compiled from the generic syscall SPEC. Besides, `Rtkaller` checks the syscalls of each task and further modifies those tasks that may not be invoked for long periods, reducing system hang caused by real-time requirements. To trigger more real-time related operations and test deep into RTOS, we use shared memory to synchronous the runtime information, such as coverage and exit code, to guide task mutations for further execution. `Rtkaller` implements parallel execution, which spawns multiple executors from the fuzzer and executes programs within the task simultaneously to improve the throughput.

For evaluation, we compared `Rtkaller` with two state-of-the-art kernel fuzzers: Syzkaller [8] and Moonshine [17] on several versions of real-time Linux kernels, which are one of the most widely used RTOS. The results demonstrate that `Rtkaller` achieves a higher branch coverage averagely by 26.1% and 22.0% , speedup about 1.7X and 1.6X respectively, when reaching the same coverage. Along with the coverage improvement, we confirmed 28 previously unknown vulnerabilities. In summary, this paper makes the following contributions:

- To fuzz deep into RTOS real-time related code and discover its unknown vulnerabilities, we propose a state-aware fuzzing method capable of generating higher quality tasks as input.
- We implement `Rtkaller`[1], a state-aware RTOS fuzzer. Using the generic syscall SPEC, it performs a task-based test case initialization, mutation, modification and parallel execution. Meanwhile, it output the coverage and vulnerability information for coverage guidance.
- We apply `Rtkaller` on typical RTOS for real practice, and it achieves a higher branch coverage with a faster speed and detects more vulnerabilities than the state-of-the-art kernel fuzzers. In total, it has confirmed 28 previously unknown vulnerabilities.

This paper is organized as follows. We provide the necessary backgrounds and the challenges of RTOS fuzzing in Section **2**. Then we demonstrate the detailed design and implementation of `Rtkaller` in Section **3**. The evaluation of `Rtkaller` can be seen in Section **4**, and the related work is introduced in Section **5**. Last we summarize this paper in Section **6**.

2 BACKGROUND

2.1 Real-time Operating System

RTOS [30] is an operating system designed to serve those applications with real-time requirements and have a wide range of applications in industrial control, aerospace, and power systems. In contrast to a general OS that focuses on the overall throughput, the main goal of an RTOS is to ensure a process will complete by a specific deadline. Hence, an RTOS is desired when there are multiple processes and devices, especially when processes' timing is more important than the system performance. Besides, an RTOS application usually consists of multiple independent tasks; each task has an assigned priority, which indicates its corresponding urgency. In general, an RTOS implements a preemptive multi-tasking algorithm; by using a periodic interrupt routine, it will choose the task with the highest priority to execute at each round. To satisfy the real-time requirements, the RTOS needs to be predictable and deterministic.

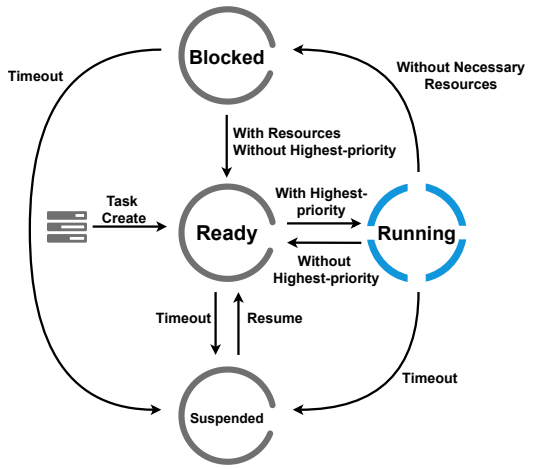


Fig. 1. A Finite State Machine of the Task State Transition

As shown in Fig.1, each task only stays within a small state set, including running, ready, blocked, and suspended; it will transfer from one state to another during execution. In concrete, the default state of a task is ready; it will enter the running state if it has the highest priority. If a running

[1]Implementation details are available through https://github.com/Rtkaller/Rtkaller

task no longer has the highest priority, it will back to the ready state; if a task lacks the necessary resources, it will transfer to the blocked state; furthermore, for those tasks that have run out of time slices, it will switch to the suspended state and wait for further resume.

2.2 Coverage-guided Kernel Fuzzing

Fuzzing is an automatic software testing technique, its core idea is generating random test cases as input to monitor the abnormal behavior of the System Under Test (SUT). As an effective vulnerability detection technique, fuzzing has successfully located millions of critical vulnerabilities among various programs. To achieve a higher coverage and test the SUT more thoroughly, many contemporary fuzzers implement the coverage-guided fuzzing method. Its primary intention is to give those seeds that trigger new coverage from previous execution a higher chance of mutation, so that the fuzzer can increase the probability of finding new paths.

The OS Kernel represents a critical part of the modern software, often too large to be bug-free. To ensure the kernel's code quality, it is necessary to deploy fuzzing on the kernel testing. Some advanced coverage-guided kernel fuzzers, such as MoonShine [17], Syzkaller [8], use syscall descriptions as fuzzing input and exploit the coverage information as guidance.

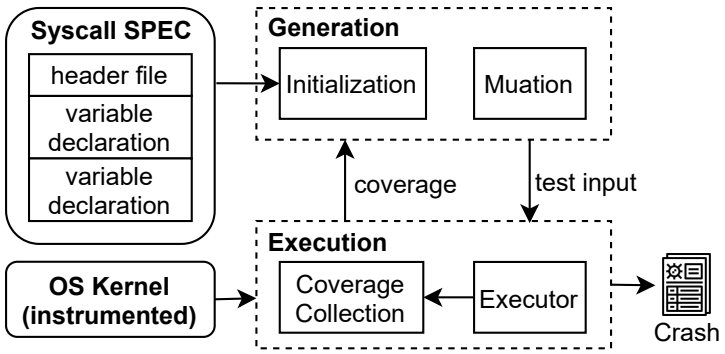


Fig. 2. Workflow of Syzkaller. It uses the syscall SPEC as input to model the kernel's interfaces. Then based on the SPEC, Syzkaller initializes test cases and executes them on the kernel under test. Furthermore, Syzkaller will collection the execution information such as coverage to guide further mutation. If any crash happens, it will generate a crash report.

Taking Syzkaller as an example, its overall workflow is shown in Fig. 2. Syzkaller uses syscall SPEC to generate executable programs as test cases. Then it uses kcov[35] to collect coverage information from the kernel. Suppose that the test case in the last round triggers new coverage, Syzkaller will give this test case a higher mutation probability. Also, if any crash happens, Syzkaller will record the crash message and try to reproduce it. By now, Syzkaller has successfully located significant numbers of critical vulnerabilities within various kernels.

2.3 Challenges of RTOS Fuzzing

Difficulties of Real-time Information Encoding. It is challenging for generic kernel fuzzers to perceive task states and express real-time information during fuzzing. However, since the state is an essential part of the RTOS, it is necessary to be state aware when testing RTOS. For instance, some tasks may not be invoked for a long time due to their low priorities, resulting in an early abort, further leading to a false crash or even some catastrophic consequences.

Listing 1 shows an example of a task hang caused by two tasks. As illustrated in Listing 1, Task A and Task B both attempt to access the same file in a mutex manner through the syscall *open*() and *write*(). However, it is worth noting that Task A lacks a syscall *close*(). In a concurrency scenario, to ensure the correctness of the file's content, the kernel will enable a mutex mechanism to limit the number of tasks to access the resource. Under this circumstance, if Task A has a higher priority than Task B, Task A will execute before Task B. However, at this point, if Task A continues to occupy this file and does not release it, Task B will remain in the blocked state, eventually get hang.

Listing 1. Task Description with Different States

```
1  //Task A
2  int fd = open(FILENAME, O_CREAT | O_RDWR);
3  if(flock(fd, LOCK_NB | LOCK_EX) == 1) {
4    write(fd, CONTENT);
5  }
6  //Task B
7  int fd = open(FILENAME, O_CREAT | O_RDWR);
8  if(flock(fd, LOCK_NB | LOCK_EX) == 1) {
9    write(fd, CONTENT);
10 }
11 close(fd);
```

Such example are cumbersome to handle during RTOS fuzzing, as the fuzzer may mistake such hang as a crash and spend much time reproducing it. It significantly increases the fuzzing overhead and slows down the fuzzing speed. Thereby, `Rtkaller` first proposes to use execution priority to manipulate the task execution order, then monitors the task generation process and modifies the task that could get hang.

Difficulties of Real-time Related Operations Triggering. The current kernel fuzzers may found it challenging in triggering some of the complex vulnerabilities within the RTOS. Most kernel fuzzers adopt a snapshot recovery mechanism and execute test cases in a single-thread manner. In detail, the fuzzers will reset the entire system back to its initial state when a test case is completed. However, as the RTOS is designed for multi-tasking, this single-threaded fuzzing method is incapable of adequately triggering real-time related operations within the RTOS, making it hard to trigger some vulnerabilities can only be triggered in a complicated situation.

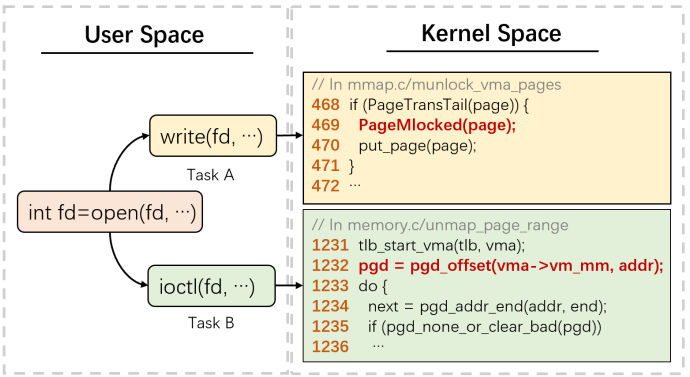


Fig. 3. An exploitation example of CVE-2018-1000200. Two tasks both access the same memory area due to an unchecked memory operations, causing an Out-Of-Memory to occur.

Here we use CVE-2018-1000200 [33] as a motivating example, Fig. 3 shows an out-of-memory (OOM) bug in Linux's memory module. This vulnerability is caused by uncontrolled memory operations. In detail, Task A and Task B both access the same virtual memory area in the system. When Task A executes, it calls the function *PageMlocked*() in the yellow section, line 469, to lock the memory. However, when Task B executes synchronously, it may access this memory area before the lock flag is set by *PageMlocked*(), as shown in the green section, line 1232. At this point, if both tasks write to the same memory area, a large amount of data may write into this memory space, in which the OOM vulnerability will then triggered. Such vulnerabilities are usually hard to be detected in RTOS. To better mitigate these problems, `Rtkaller` implements a task-based parallel execution that performs coverage-guided fuzzing in a multi-threaded manner to trigger more real-time related operations and exploit the potential vulnerabilities more efficiently.

3 SYSTEM DESIGN

The design of `Rtkaller` is presented in Fig. 4. It utilizes the traditional syscall SPEC as input, which is also adapted by existing kernel fuzzers such as Syzkaller[8], KAFL[12] and Moonshine[17]. Instead of generating executable programs, `Rtkaller` generates tasks for the parallel execution and monitors the crashes. Besides, for an RTOS kernel under test, we perform targeting instrumentation with sanitizers to drive the fuzzing process towards those real-time related code and support more types of vulnerabilities detection. The fuzzing engine contains three major modules: task initialization, task generation, and parallel execution.

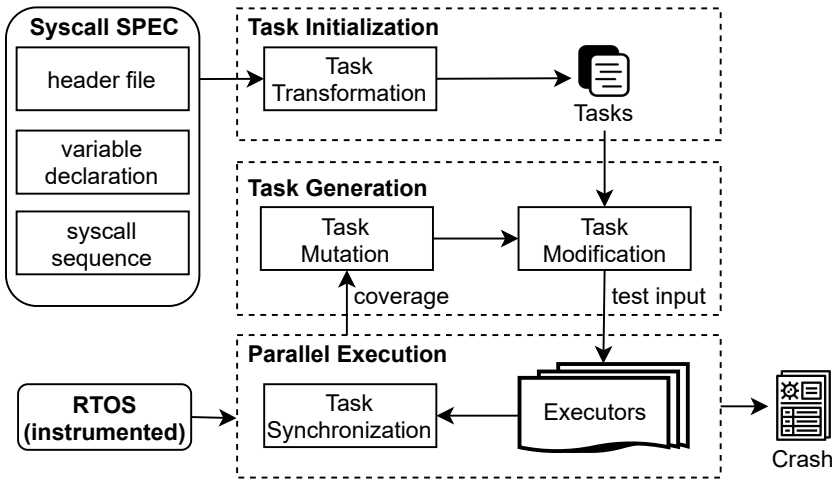


Fig. 4. Overview of `Rtkaller`. In task initialization, we use syscall SPEC to generate tasks, which can be directly taken from the existing specification file of Syzkaller and Moonshine without modification. Then, we perform a task modification to amend those tasks that may hang during fuzzing. For parallel execution, it contains several executors, which will send test cases into instrumented RTOS and return the crash report if any crash happens. It will also collect the execution information such as coverage in task synchronization; later, the coverage information will send back to the task generation module to guide further task mutation.

For task initialization module, `Rtkaller` transforms the syscall SPEC to tasks automatically, accompanied with runtime priorities and a concurrency intensity. By explicitly combining the execution priority with the task, it allows us to express more real-time information during the fuzzing process. For task generation module, based on the coverage information, `Rtkaller` proposes

a task-based mutation to generate more refined tasks. Besides, by analyzing each syscall's impact on the overall system execution, the task modification mechanism will modify those tasks that may hang during fuzzing, ensuring that every program within a task executes correctly. For the parallel execution module, it implements a fork server-like execution mechanism. Each time a fuzzer assigns the programs within a task to several executors for multi-thread execution.

3.1 Task Initialization

As demonstrated in the Section 2.3, the vulnerabilities in RTOS are hard to manifest due to insufficient expression of real-time information and inadequate triggering for real-time related operations. `Rtkaller` is designed to provide the test cases with more real-time related features like scheduling and priority. Concretely, it leverages the task description to introduce tasks into RTOS fuzzing, then it performs the task transformation to generate initial seeds for the fuzzing.

Task Definition. In the traditional kernel fuzzing process, the fuzzer uses a single program as input, compiled from a single syscall SPEC. However, due to the lack of real-time information such as priority, the syscall SPEC often fails to efficiently activate the real-time related behaviors.

To better exploit the real-time features in RTOS, we propose using tasks for execution. A task is an internal data structure within `Rtkaller`, which contains a syscall SPEC sequence, runtime priorities, and a current intensity. The syscall SPEC is used to describe the kernel's interfaces, comprising the definition of a syscall sequence with corresponding parameters. The runtime priorities indicate each program's execution urgency within the task, which is used to manipulate the programs' execution order. The intensity indicate the number of the syscall sequences. They can be initialized randomly and customized automatically.

Listing 2. An Example of Task Definition

```
1     Task:
2         int intensity  = 5
3         int[] priority = [2, 4, 56, 23, 6]
4         Prog[] programs
5     Prog:
6         resource io_ctx[intptr]
7         io_setup(n int32, ctx ptr[io_ctx])
8         io_destroy(ctx io_ctx)
```

A detailed example of automatically transformed task is presented in Listing 2. The intensity shows how many programs a task will encapsulate, as shown in line 2, there will be five different programs within the task. Typically, the higher the intensity of a task, the more programs it will execute concurrently and the more likely it is to induce complex real-time operations. Also, to limit the memory consumption, increasing the fuzzer's throughput, the concurrent intensity generally does not exceed 10. There will be a corresponding priority for each program within the task that indicates its execution order, as shown in line 3. Since RTOS follows the preemptive algorithm, the higher a program's priority is, the sooner it will be executed. Also, different combination of priority, incurring different execution order, which may lead to diverse real-time operations. Moreover, the *Prog* array is used to store the to-be-executed programs compiled from the predefined syscall sequence, the number of programs is the same as the *intensity* reveals.

Task Transformation. Based on the existing syscall SPEC, we will generate serials of programs, and integrate these programs with a concurrency intensity and runtime priorities automatically. As shown in Algorithm 1, `Rtkaller` utilizes the syscall SPEC as initial input, and it can be viewed as a syscall SPEC array, each element contains the definition of a syscall sequence. Specifically, it will

first randomly generate the concurrent intensity to indicate the number of the syscall sequences as shown in line 2. Based on this intensity and the SPEC, `Rtkaller` will then uses the function *program_generation*() to generate executable programs, which is consist of a syscall choose and an arguments generation process, as shown in lines 5-6 and lines 7-12. Upon obtaining the programs, `Rtkaller` will bestow the program a priority, as shown in line 6.

Algorithm 1: The Task Transformation Mechanism

```
   Input: S: syscall SPEC
   Output: T: Task
1  Algorithm
2      T.intensity = int_rand()
3      for i ∈ range(0, intensity) do
4          prog = program_generation(S[i])
5          T.prog = T.prog ∪ prog
6          T.priority[i] = int_rand()
7      Procedure program_generation(S)
8          for i ∈ range(0,S.size()) do
9              call = choose_syscall()
10             args_generation(call)
11             prog = prog ∪ call
12         return prog
```

3.2 Task Generation

During fuzzing, higher-quality test cases are always desired, as low-quality test cases tend to increase the fuzzing overhead and cause many false crashes, leading to a low fuzzing efficiency. To ensure the quality of test cases and further improve the fuzzing efficiency, `Rtkaller` proposes a coverage guided task generation mechanism, including task mutation and task modification.

Task Mutation. Generic kernel fuzzing usually adopts a syscall mutation strategy, in which they tend to mutate the syscalls' parameters as well as randomly remove some non-vital syscalls. However, when fuzzing a kernel with more emphasis on concurrency like the RTOS, such a strategy may not be sufficient. The generic methods focus more on the diversity of syscalls' parameter, which fails to concern the impact on fuzzing efficiency with the different execution order. To handle such issues, `Rtkaller` introduces the task mutation, which mutates the syscall SPEC and the runtime priorities. In this way, we can greatly increase the diversity of real-time related operations, improving the fuzzing efficiency. Algorithm 2 shows an overview of the task mutation process.

To improve the quality of the mutated test cases, we increase the mutation probability of those tasks that are able to trigger new branches. We define λ, which is a constant that makes the probability of S_i converge gradually to 1. For a test case S_i, we record whether S_i finds new coverage in round i as X_i, assign X_i as 1 if it does, 0 otherwise. Initially, the mutation probability P is randomly generated and will be updated based on subsequent execution, as shown in line 3. The mutation probability for seed S_i is:

$$P_{(S_i)} = \lambda \cdot (1 - \frac{1}{\sum_{i=1}^{n} X_i}) \tag{1}$$

Algorithm 2: The Task-based Mutation Methodology

```
   Input: W: WorkQueue
   Input: T: Task
   Output: T: Task
1  Algorithm
2      if W.size() == 0 then
3          p = float_rand(0, 1)
4          if p ≥ 0.5 then
5              task_mutation(T)
6              state_amend(T)
7          else
8              state_amend(T)
9      Procedure task_mutation(T)
10         for i ∈ range(0,T.intensity) do
11             T.prog = syscall_mutation()
12             if T.priority[i] ≥ 50 then
13                 T.priority[i] = int_rand(0, 50)
14             else
15                 T.priority[i] = int_rand(50, 100)
```

For a test case S_i, if P is bigger than 0.5, `Rtkaller` deems it as worth mutating and then performs the mutation. Rather than only mutate the program that triggers new paths, we use the task as a basic mutation unit and mutate all programs within it. In particular, after the execution, if a program *Prog* triggers new coverage, `Rtkaller` will mark its corresponding task *Task* as interesting. Then `Rtkaller` iterates and mutates every program in *Task*, as shown in lines 4-8. Since all mutated programs triggered new coverage multiple times in previous executions, `Rtkaller` would have a greater chance of finding new coverage. Also, to introduce more various real-time operations, `Rtkaller` will mutate each program's execution priority, as shown in lines 11-15. More specifically, it will raise the priority of lower-priority programs and lower the priority of higher-priority programs. So, `Rtkaller` can disrupt the previous execution order and further explore RTOS behavior under different scheduling scenarios.

Task Modification. As to RTOS fuzzing, it is challenging to ensure that the fuzzers can execute each program successfully without any hang, which could severely undermine the efficiency. The mutation strategy may randomly remove some syscalls, resulting in some programs terminated with exceptions. Hence, `Rtkaller` applies a task modification algorithm, by analyzing each syscall's effect on the task's states, to amend those test cases that may cause the task hang during fuzzing.

Algorithm 3 provides an overview of the task modification process. It takes a predefined syscall pair *CallList* and a task *Task* as input. The task is initialized in the previous step, while the *CallList* specifies some syscall pairs that may cause the task to hang. We then start to check every syscall within the task, as shown in lines 1-3. By using a bitset *index*, `Rtkaller` records the syscall's search results. In detail, `Rtkaller` pairs the syscalls in the *CallList* to determine whether a syscall may contain potentially dangerous operations. `Rtkaller` will then mark the syscall in *index*, as shown in lines 4-9. After checking every syscall, if we find that the *index* of the current task is not equal to 0, we consider the task as hazardous, in which a syscall within the task might occupy some

Algorithm 3: The Task Modification Mechanism

```
   Input: T: Task
   Input: L: CallList
   Output: T: Task
1  for prog ∈ T do
2      index = ∅
3      for call ∈ prog do
4          if isPostiveMatch(call, CallList) then
5              pos = get_syscall_position(call)
6              index[pos] += 1
7          if isNegtiveMatch(call, CallList) then
8              pos = get_syscall_position(call)
9              index[pos] -= 1
10     for idx ∈ index do
11         if idx != 0 then
12             task.add_counter_syscall(prog, idx)
```

resources in the critical section and not release them, or it might be waiting to access resources for a long time in future execution. At this stage, `Rtkaller` relies on the predefined *CallList*, identifies the corresponding syscall, and adds this corresponding syscall to the task as shown in lines 10-12.

To better illustrate our approach, we take Listing 3 and Listing 1 as an example. As Listing 3 shows, we define a bitset *index* to record the pairing status of syscalls. We define a mapping from a syscall name to an integer, with its pairing syscall assigning an opposite value. For instance, `Rtkaller` will set *open*() as 1, then *close*() will set as −1. When `Rtkaller` conducts a task amend algorithm, we will attempt to find those syscalls defined in the *CallList*. Take task B in Listing 1 for example, `Rtkaller` first locates *open()*, then takes its mapping value as location index and sets the corresponding location in *Idx* as 1. When iterating to *write()*, `Rtkaller` will skip it because we do not define it at *CallList*. When `Rtkaller` meets *close()*, it will also get a match and acquire *close()*'s corresponding value. Once the iteration is complete, if any bit in the *index* is not equal to 0, `Rtkaller` will find the corresponding syscall and append it to the task. In our case, the first bit in task A is not 0, so `Rtkaller` will find *open()*'s corresponding syscall *close()*, and append it to task A.

Listing 3. A Snippet of Predefined CallList

```
1  map<string,int> CallList
2  CallList["open"]    =  1
3  CallList["close"]   = -1
4  CallList["mount"]   =  2
5  CallList["unmount"] = -2
```

3.3 Parallel Execution

Task Based Execution. Rather than executing one test case at a time, `Rtkaller` adopts the concurrent fuzzing method, which enables us to better simulate the real runtime scenarios of RTOS and test the RTOS more thoroughly by introducing more real-time related operations.

Algorithm 4: Parallel Execution of the Fuzzer

```
   Input: T: Task
   Output: C: CrashList
   Output: W: WorkQueue
1  PidList = ∅, InfoList = ∅
2  for prog ∈ T do
3      pid = generate_pid()
4      while pid ∈ PidList do
5          pid = pid++
6      PidList = PidList ∪ pid
7      spawn
8      exec = create_executor(pid, T)
9      set_priority(exec, T.prio)
10     info = exec.execute_program(prog)
11     InfoList = InfoList ∪ info
12     sync
13     sort(InfoList)
14     for info ∈ InfoList do
15         if info.crash != ∅ & Lock(C) then
16             C = C ∪ info.crash
17             unLock(C)
18         if (have_new_coverage(info)) then
19             W = W ∪ T
```

`Rtkaller` adopts a fork server-like architecture. In detail, the fuzzer acts as the fork server, which is responsible for executors' creation as well as the test case distribution. Meanwhile, each executor is in charge of the program execution and the feedback information collection. In the fuzzing process, to run multiple test cases simultaneously, `Rtkaller` first creates multiple executors from the fuzzer and assigns the pre-generated execution priority to each executor. After it executed, to keep the global execution information consistent with the local execution information, each executor merges and synchronizes the generated runtime reports to the fuzzer. It allows `Rtkaller` to have higher throughput and faster execution speed by executing each task asynchronously. Meanwhile, the synchronization mechanism enables `Rtkaller` to update the global coverage information, identify the test cases that trigger new coverage, and save them for further mutations.

Algorithm 4 demonstrates the basic idea of how `Rtkaller` implements the parallel execution. `Rtkaller` uses the $InfoList$ to store the execution information and uses the $PidList$ for a thread safety check. In detail, for a to-be-executed task, `Rtkaller` will iterates every program within it, then it assigns a pid to each program $prog$ and performs a pid conflict check, as `Rtkaller` needs to protect each executor from crashes caused by pid conflicts by maintaining the uniqueness of each pid, as shown in lines 3-6; Subsequently, it will start to create the executors in a thread-safety manner and allocate each executor a program with a corresponding execution priority, as shown in lines 7-9. After each executor finishes execution, it returns execution information $info$, as shown in lines 10-13, which will then be added to the $InfoList$. When all executions are complete, we refer

to each $info$ in the $InfoList$. Rtkaller identifies interesting tasks by determining if the current task triggers any new crash or coverage, as shown in lines 14-19.

Feedback Collection and Synchronization. To ensure the proper implementation of the fork server-like architecture, Rtkaller needs to ensure a correct and reliable data transfer in multi-threaded scenarios. In detail, for each executor spawned from the fuzzer, it runs the to-be-executed program separately. After the execution, Rtkaller identifies whether the program triggers a new crash or coverage. Then it synchronizes the local feedback information, including the coverage and crashes information, into the global server in a thread-safe manner to guides further fuzzing.

Algorithm 5: Feedback Collection of the Executor

```
   Input: P: Program
   Input: B: Global Crash
   Output: info: Execution Information
1  Procedure ProgExec(P, task)
2      info = run(P)
3      if info.crash != ∅ then
4          if Lock(B) & info.crash ⊄ B then
5              B = B ∪ info.crash
6              unLock(B)
7      else
8          crashInfo = ∅
9      localCover = info.cover
10     if Lock(C) & info.cover ⊄ C then
11         C = C ∪ info.cover
12         unLock(C)
13     return info
```

Algorithm 5 demonstrates the feedback collection mechanism of the local executor. Each local executor first executes the program, as shown in line 2. Then the executor determines if the program triggers any crash. If Rtkaller triggers a new crash, it will return the crash information and save the task that triggers the crash as interesting, as shown in lines 3-8. Finally, we check whether the current program has found new paths. If new paths exist, we synchronize them to the global coverage recorder *Cover* and mark the current task as interesting, as shown in lines 9-13.

3.4 Implementation

We implement the targeting instrumentation in Python and implement Rtkaller in Golang, on top of Syzkaller. For kernel's instrumentation, Rtkaller first uses a real-time code aware script to analyze the RTOS's codebase. In detail, apart from the library and memory management function that a general operating system has, the RTOS has some code to handle real-time requirements like task switching. For instance, the PREEMPT_RT patch modified many files in the Linux kernel, allowing it introduces real-time capabilities to a general operating system. Our script will iterate the entire kernel codebase and identifies files that are related to real-time functionality. Based on the results, Rtkaller applies Breadth-First Search (BFS) algorithm to modify the kernel's build configurations, specify to be instrumented files. It is noteworthy that the kernel forbids the instrumentation of certain files, as it may introduce non-deterministic behaviors or generate

uninteresting code coverage. Hence, we avoid instrumentation on these files during compile time. For task initialization, we modified the program generation algorithm on Syzkaller, which enabled `Rtkaller` to generate tasks and convey more real-time information. For task generation, we extend the program mutation methodology so that `Rtkaller` can simultaneously handle the mutation of multiple programs within a task. We add a task modification strategy for task modification to maintain each program's runtime state during fuzzing automatically. We implement a fork server-like architecture for parallel execution, which allows the fuzzer to spawn multiple executors in multi-thread. We also use a bitset to prevent the fuzzer from crash due to pid conflict. The fuzzer dispatches test cases to each executor, which executes these test cases separately. Then `Rtkaller` synchronous the execution results, such as the crash reports and coverage information, back to the fuzzer to guide further task generation.

4 EVALUATION

We evaluated the effectiveness of `Rtkaller` on the latest versions of the rt-Linux kernel, which is a widely used RTOS. Two widely used state-of-the-art kernel fuzzers are used for comparison: Syzkaller and Moonshine, where Syzkaller is developed by Google and Moonshine is an incremental version of Syzkaller with a seed distillation strategy. We answer the following questions:

- **RQ1:** Can `Rtkaller` improve the code coverage during fuzzing?
- **RQ2:** Can `Rtkaller` improve the vulnerability detection ability?
- **RQ3:** Is the modification mechanism effective in reducing task hang?

4.1 Experiment setup

Data Set. We select several versions (ranging from the latest 5.9-release to the old 3.1-release) of the rt-Linux kernel as the data set, as shown in Table 1. First, we make a fuzzing performance comparison, mainly focus on the code coverage improvement. Second, we measure the effectiveness of the task modification mechanism of `Rtkaller` by recording the total hanged task cases during fuzzing. Last, we demonstrate `Rtkaller`'s ability of identifying vulnerabilities.

Table 1. Data Set for Performance Comparison

RTOS Versions	Release Date	Rt-patch Versions	Line of Code
3.10-release	2013-03-06	patch-3.10.108-rt123	11402.8k
4.14-release	2017-12-12	patch-4.14.103-rt55	17049.8k
5.0-release	2019-03-03	patch-5.0.7-rt4-rt5	18030.8k
5.2-release	2019-07-07	patch-5.2-rt1	18335.6k
5.4-release	2019-11-24	patch-5.4.3-rt1	19297.4k
5.6-ktsan	2020-09-17	patch-5.6.4-rt2-rt3	18866.1k
5.6-release	2019-03-29	patch-5.6-rt1	19752.0k
5.9-release	2020-10-11	patch-5.9-rc7-rt10	20746.1k

Environment Setting. We conduct the experiments on an Intel(R) Xeon(R) CPU with 128G RAM that runs Ubuntu 20.04 as the host kernel. We compile the data set with the same configuration under the x86_64 architecture. All fuzzers are augmented with Kernel AddressSANitizer (KASAN) [13] , which is a dynamic memory error monitor, to detect memory corruption (e.g. stack-overflow, buffer-overflow and use-after-free). Each experiment was conducted for 24 hours and repeated

with 10 times. Rtkaller uses the identical initial syscall SPEC as Syzkaller [2], and for Moonshine, we perform the distillation algorithm on these syscall SPEC to obtain a refined fuzzing input.

4.2 Effectiveness in Coverage Exploration

We answer RQ1 in terms of the branch coverage. To minimize the randomness of the results, we repeated each 24-hour experiment 10 times. We collected execution information, including timestamps and the amount of branch per minute. Then, we computed the average coverage of Syzkaller, Moonshine, and Rtkaller over the 10 rounds of experiments.

Table 2. Average Coverage and Speed Improvement of 10 repeated experiments of the 8 versions

RTOS Versions	Syzkaller	Moonshine	Rtkaller	average-impr	speedup
3.10-release	70597.8	68811.7	76566.4	8.5%/11.3%	+2.1x/+1.8x
4.14-release	55713.4	71789.2	107599.8	93.1%/49.9%	+2.0x/+1.6x
5.0-release	99694.2	109655.0	150821.4	51.3%/37.5%	+1.5x/+1.7x
5.2-release	216589.7	212443.6	245302.6	13.3%/15.5%	+1.5x/+1.4x
5.4-release	117011.6	119548.8	123333.4	5.4%/3.2%	+1.6x/+1.7x
5.6-ktsan	107028.4	109672.2	192071.8	79.5%/75.1%	+2.2x/+2.0x
5.6-release	123369.4	124085.4	126616.1	2.63%/2.04%	+1.2x/+1.3x
5.9-release	121085.2	125784.4	127003.6	4.89%/0.97%	+1.5x/+1.1x
average	113886.2	117723.8	143664.4	26.1%/22.0%	+1.7x/+1.6x

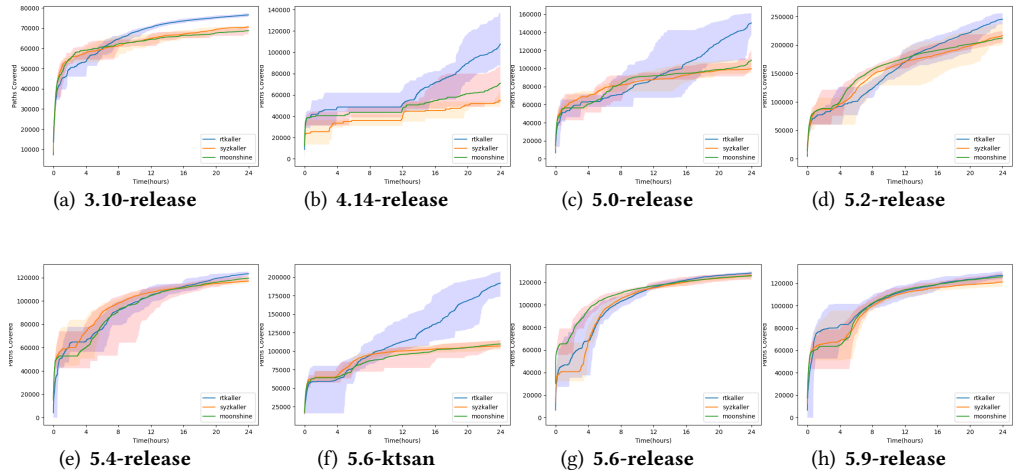
Table 2 demonstrates the statistic improvements of code coverage and fuzzing speed. For each rt-Linux version, we list the average branch coverage of Syzkaller, Moonshine, and Rtkaller. From the table, we can find that compared with Syzkaller and Moonshine, Rtkaller covers 26.1% and 22.0% more branches, respectively. The main reason for this improvement is that the task initialization and the task generation approaches enable Rtkaller to spawn more real-time related operations within RTOS. In this way, we can find more execution paths caused by real-time related operations, which are usually hard to trigger in Syzkaller and Moonshine. Besides, from the table, we notice that compared with Syzkaller and Moonshine, Rtkaller can gain about 1.7X and 1.6X speedup in terms of fuzzing speed; this statistic is calculated by comparing the time consumption of Rtkaller to reach the final coverage of Syzkaller and Moonshine. Benefiting from the parallel execution, Rtkaller can execute multiple programs simultaneously, which allows it to have a higher throughput; also, the task modification algorithm prevent Rtkaller to generate inferior test cases, thus significantly increase the fuzzing efficiency.

Fig. 5 shows the detailed coverage change over time. Rtkaller is able to achieve a higher branch coverage than Syzkaller and Moonshine at a faster speed. All of them grow rapidly at the beginning.However, during some periods, the experiments' growth curve will stop, especially for the results of 4.14-release and 5.6-ktsan. This is due to Syzkaller's reproduction mechanism. In detail, when a crash is detected, Syzkaller will try to reproduce it to identify if it is a real crash; nevertheless, if such a crash is a false crash and is not reproducible, the fuzzer might waste a long time in the reproduction phase.

The task modification mechanism enables Rtkaller to minimize such impact. As we can see from the results of 5.2-release, Rtkaller has a fewer reproduction period than Syzkaller and Moonshine.

[2]The detail of the syscall SPEC can be found at https://github.com/google/syzkaller/tree/master/sys/linux

From the above results, it is reasonable to conclude that `Rtkaller` outperforms Syzkaller and Moonshine in coverage exploration and speed when fuzzing RTOS.



(a) **3.10-release** (b) **4.14-release** (c) **5.0-release** (d) **5.2-release**

(e) **5.4-release** (f) **5.6-ktsan** (g) **5.6-release** (h) **5.9-release**

Fig. 5. Covered Branch - `Rtkaller` is able to cover more branches with a faster speed.

4.3 Effectiveness in Vulnerability Exposure

We answer RQ2 about the potential of `Rtkaller` on vulnerability detection. Specifically, `Rtkaller` found a total of 142 vulnerabilities, while Syzkaller detected 103 vulnerabilities and Moonshine detected 108 vulnerabilities. In total, with the overlap among the vulnerabilities found by the 3 tools, `Rtkaller` found 28 more vulnerabilities than the union set of Moonshine and Syzkaller (114). Fig.6 demonstrates the vulnerabilities detected by each tool. The results show that 3.10-release contains most of the vulnerabilities, due to the fact that it appeared relatively early and did not undergo extensive security checks.

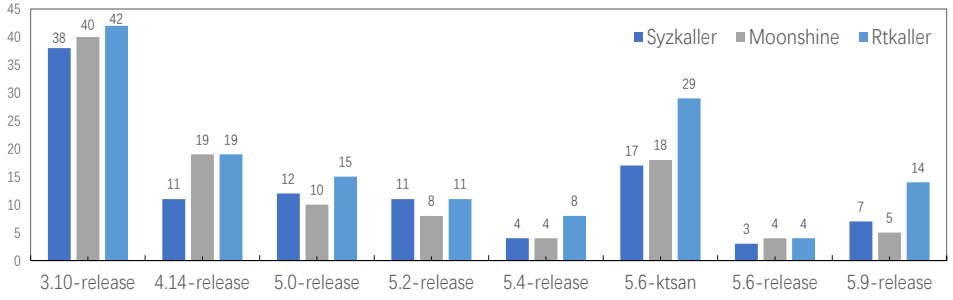


Fig. 6. Vulnerabilities Statistics Compared with Syzkaller, and Moonshine

Table 3 shows those extra vulnerabilities that are only located by `Rtkaller`, including the corresponding module, operation, and impact of each unique vulnerability. Although Syzkaller has been endlessly testing the Linux kernel, these 28 vulnerabilities have not been reported before.

Table 3. Previous Unknown Vulnerabilities Located by Rtkaller Uniquely

RTOS Versions	Module	Operation	Vulnerability
3.10-release	net/core/dev.c	dev_remove_pack	NULL ptr deref
	arch/x86/kernel/cpu/perf_event_intel.c	intel_shared_reg_put_constraints	NULL prt deref
5.0-release	include/linux/rbtree.h	rb_insert_color_cached()	general protection fault
	fs/dcache.c	list_lru_add()	NULL prt deref
	fs/debugfs/file.c	debugfs_remove()	use after free
5.4-release	kernels	swapgs_restore_regs()	stack overflow
	kernel/relay.c	relay_alloc_buf()	deadlock
	drivers/vt_ioctl.c	vt_ioctl()	general protection fault
	fs/proc/proc_sysctl.c	count_subheaders()	general protection fault
5.9-release	driver/tty/vt/vt.c	vc_con_write_normal()	use after free
	include/linux/mm.h	vma_interval_tree_iter_next()	general protection fault
	drivers/video/console/vgacon.c	clear_buffer_attributes()	use after free
	lib/vsprintf.c	vsnprintf()	Bad page map
	mm/interval_tree.c	anon_vma_interval_tree_insert()	NULL prt deref
	kernel/sched/swait.c	swake_up_all_locked()	possible system lock
	drivers/tty/vt/vt.c	complement_pos()	use after free
5.6-ktsan	mm/slub.c	freelist_dereference()	stack segment fault
	lib/find_bit.c	find_next_and_bit()	general protection fault
	lib/idr.c	idr_get_free()	general protection fault
	net/ipv4/route.c	ipv4_dst_check()	general protection fault
	net/ipv6/addrlabel.c	ip6addrlbl_net_exit()	general protection fault
	fs/kernfs/file.c	kernfs_notify_workfn()	general protection fault
	mm/slub.c	__kmalloc()	general protection fault
	mm/slub.c	kmem_cache_alloc()	general protection fault
	mm/shmem.c	shmem_file_read_iter()	rcu detected stall
	fs/proc/proc_sysctl.c	unregister_sysctl_table()	stack segment fault
	security/selinux/avc.c	avc_node_delete()	stack segment fault

Besides, we analyzed the cause of each vulnerability. With the support of parallel execution, Rtkaller successfully located a large amount of kernel hangs issues; also, under the help of the KASAN, the majority of detected vulnerabilities were memory corruption like use-after-free and stack segmentation fault. In summary, 30.6% were memory errors detected with the aid of KASAN. 52.2% were bugs in the execution of system calls, and 17.2% were deadlock issues.

Though Rtkaller incorporates scheduler-specific concepts and techniques, Rtkaller aims to trigger memory-related bugs extensively found in RTOS, with the help of kernel address sanitizer. Hence, most of the detected vulnerabilities spread across the entire RTOS, rather than the scheduler component alone. Also, Rtkaller's testing inputs and oracles are significantly different from those of scheduler testing. Specifically, scheduler testing requires generating periodic, sporadic, and

aperiodic tasks with representative properties to verify its correctness (i.e., finding timing bounds violations) and performance (i.e., statistical timing behavior and resilience). Testing an entire RTOS, as presented in Rtkaller, requires generating more generic tasks to detect vulnerabilities in the implementation (i.e., memory access violations). Rtkaller collects comprehensive feedback from the entire real-time-relevant code (i.e., rt-patch), not just the scheduler itself.

4.4 Effectiveness in Task Hang Reduction

Here we answer the RQ3 about the effectiveness of the task modification mechanism in reducing task hang. We further implemented Rtkaller- , a trimmed version of Rtkaller without the task modification mechanism. During experiments, we record the total number of test cases that get hang from each experiment. Table 4 shows a comparison between Rtkaller and Rtkaller-.

Table 4. Task Hang Statistic Compared with Rtkaller-

RTOS Versions	Rtkaller-	Rtkaller	Improvement
3.10-release	13.4	6.9	48.5%
4.14-release	13	0	100.0%
5.0-release	6.6	1.4	78.8%
5.2-release	10.8	7.2	33.3%
5.4-release	32.2	23.9	25.8%
5.6-ktsan	7.8	1.9	74.7%
5.6-release	8.5	6.1	28.2%
5.9-release	15.5	10.4	32.9%
average	13.4	7.8	41.8%

The result shows that Rtkaller reduces about 41.8% test cases hang on average. Especially for 4.14-release, Rtkaller successfully ensures that no test cases hang. This is due to Rtkaller successfully locating all syscalls that could trigger the process to hang and automatically modifies them. For others, it reduces about 25%-79%, which means that Rtkaller manages to locate and modify some of the damaged test cases. Since the modification relations are manually pre-defined, we cannot cover all potentially dangerous system call pairs. This is why Rtkaller still has hang due to the real-time operations. Our experiment results show that the task modification mechanism does play an essential part in helping fuzzer generate higher-quality test cases.

4.5 Real Vulnerability Case Study

Kernel RSS Count Error. Resident Set Size (RSS) is used to indicate the amount of memory allocation to the processes. When a user process is terminated, the kernel will clean up the process's memory. Before cleanup the memory space, the kernel will check the process's page table; when it finds a corrupted page table, such a bug may occur. Listing 4 shows an example of the RSS count state error triggered by Rtkaller in kernel's locking module when fuzzing the rt-Linux version v5.2-release; this module is responsible for resources allocation. After analyzing the kernel bug report, we locate the vulnerability's source, a failed memory-free operation.

As Listing 4 shows, the shmem_lock function tries to gain access to a critical resource via user_shm_lock in a multi-thread scenario. It will eventually call the rt_spin_lock function within the kernel's locking module. In general, if a process fails to use spin_lock to gain access to critical resources, it will remain in a busy loop. However, when the delay exceeds the time limit for soft interrupts, the process may exit before all resources are freed, resulting in page corruption. When the RTOS kernel performs the page checking, such an error is exposed.

Listing 4. Vulnerable code of shmem_lock.

```
// in function shmem_lock
if (lock && !(info->flags & VM_LOCKED)) {
  if (!user_shm_lock(inode->i_size, user))
   goto out_nomem;
 ...
 }
 if (!lock && (info->flags & VM_LOCKED) && user) {
  user_shm_unlock(inode->i_size, user);
 ...
 }

void __lockfunc rt_spin_lock(spinlock_t *lock)
{
 sleeping_lock_inc();
 rcu_read_lock();
 migrate_disable();
 //Bug Occurs Here
 spin_acquire(&lock->dep_map, 0, 0, _RET_IP_);
 rt_spin_lock_fastlock(&lock->lock, rt_spin_lock_slowlock);
}
```

Kernel UAF Error. Use-after-free is usually a common yet troublesome issue in testing RTOS, often due to the incorrect dynamic memory allocation; it usually refers to the attempt to access memory after it has been freed. It potentially results in a program crash or the execution of arbitrary code. In the kernel scenario, some attackers may use this kind of vulnerability to take over the entire system. During fuzzing the rt-Linux version v5.2-release, `Rtkaller` successfully detects a use-after-free vulnerability in the memory module with the help of KASAN. After analyzing the bug report provided by `Rtkaller`, we located the vulnerability root cause, it is caused by a pointer passing in memory allocation.

As Listing 5 shows, the slab_post_alloc_hook function tries to tag each allocated memory in lines 4 - 10. The kernel uses kasan_slab_alloc, as shown in line 6, to allocate memory space. However, suppose it tries to allocate a large space. In that case, the memory allocation may fail, and these allocated memories will get freed. Nevertheless, in concurrent execution, the kernel calls kmemleak_alloc_recursive as shown in line 8; due to the lack of checking for memory lock, these previously freed memories are passed without freed resulting in a use-after-free situation. Malicious attackers may utilize such vulnerability. By inserting a well-designed program jump instruction address at the specific location of a data buffer, the attacker can achieve an arbitrary code execution, then acquire a higher system privilege or bypass some security checks in the kernel.

Listing 5. Vulnerable code of slab_post_alloc_hook.

```
// in function slab_post_alloc_hook
 size_t i;
 flags &= gfp_allowed_mask;
 for (i = 0; i < size; i++)
 {
      p[i] = kasan_slab_alloc(s, p[i], flags);
      /*The Kernel Bug Occurs Here. */
      kmemleak_alloc_recursive(p[i],
      s->object_size, 1, s->flags, flags);
 }
```

4.6 Threats to validity

The first potential threat is the scalability of Rtkaller. Although the data set in this paper is the most recent version of rt-Linux, the methodology is feasible for most RTOS. For one thing, as the algorithms are independent of the implementation of target RTOS, the task generation and instrumentation modules that run in the host operating system are implemented in GoLang and Python, while the test case executor running within the guest kernel is written in C++, which is easily portable. In addition, Rtkaller uses tasks, which are the basic execution units of RTOS, as fuzzing interface. Therefore, to support other RTOS, the extra work is to prepare corresponding task descriptions for target RTOS. By adding additional task descriptions and modifying the Rtkaller's execution module to support task simulation, we have successfully adapted Rtkaller on Erika fuzzing, and found a previously unknown vulnerability, demonstrated in the Github website.

The second is the performance bottleneck caused by the incompleteness of fuzzing. The current fuzzing performance is limited by less refined seeds and the mono vulnerability detection approach. In concrete, despite that Rtkaller adopts the coverage-guided task mutation strategy to generate higher quality test cases, the current seed generation strategy still has trouble generating more refined test cases. Specifically, different syscalls may have potential dependencies, where some syscalls might depend on the result of previous syscalls. However, Rtkaller cannot accurately perceive this syscall relations. Such implicit relations can result in fuzzer producing many invalid seeds, failing to reach better coverage. Hence, we can further improve Rtkaller by augmenting the task modification module to automatically learn such syscall relations, or leveraging more runtime feedback as guidance (i.e., thread information, memory operations), thus achieving better coverage. Moreover, as for the insufficient bug detection ability, Rtkaller mainly uses KASAN for vulnerability detection, which may limit the detected vulnerabilities mostly on memory corruption. To further improving the bug detection ability, we can adapt more customized detection strategies [10, 14, 19] or adapt more bug monitors [22, 27] to support detecting more bug types.

5 RELATED WORK

5.1 Traditional Fuzz Testing

With the growing promise of fuzzing techniques for vulnerability detection, more and more researchers have focused on probing the potential of fuzzing techniques to ensure software security.

For the generation-based fuzzers [3, 5, 26, 29, 32], their core purposes are that fuzz applications have strict format requirements. They use the input format specifications to generate high-quality test cases and guide further fuzzing according to the execution results. Peach [32] is a generation-based fuzzers widely known for outstanding performance on protocol fuzzing. It defines a format specification, which contains two main models. The data model describes the protocol data structure, while the state model describes how the fuzzer should communicate with the program under test. By far, Peach has located many critical vulnerabilities in various protocols, and many related works try to extend Peach's fuzzing capabilities. Peach* [26] augments Peach with coverage feedback ability and modifies Peach's mutation strategy by introducing seed fragment replacement mechanism, significantly improves Peach's efficiency.

For mutation-based fuzzers, they mainly unitize their diverse mutation strategies to test programs more efficiently. One of the most successful mutation-based fuzzers is American Fuzzy Lop (AFL). AFL has detected thousands of bugs in a wide range of applications, known for its ease-of-use and exceptional performance. Other AFL family tools [9, 18, 20, 21, 24] apply a variety of strategies to boost the fuzzing process. AFL-fast [9] uses search strategies that enable fuzzer to focus on the paths that are less executed. AFL-smart [23] combines the structured input components of Peach with the grep-box fuzzing of AFL.

For domain-specific fuzzers, such as for multi-threaded programs, due to their non-deterministic behavior during runtime, many works attempt to introduce thread intervention techniques to better understand the program's internal state, thus improve the fuzzing efficiency and locates more concurrent bugs such as data race or thread-leak. MUZZ [28] uses additional instrumentation to provide more accurate thread information to guide the seed generation. ConAFL [16] combines static analysis with fuzzing; it can manipulate the thread scheduling, and locate more concurrent vulnerabilities in multi-threaded programs. Compared to multi-threaded programs that attempt to use thread intervention techniques to guide fuzzing, `Rtkaller` is designed to locate various vulnerabilities throughout the RTOS. It achieves the parallel execution, enabling a better compatibility with task-based input and triggers more real-time related operations in the target RTOS, thereby improving the fuzzing efficiency and simulating a more realistic execution scenario.

5.2 Kernel Fuzz Testing

Due to the tremendous amount of code in the kernel and the importance of kernel security, more researchers are applying fuzzing tools to kernel testing. However, there are many difficulties in porting traditional fuzzers to kernel fuzzing, such as the complexity of syscalls and the gap in coverage collection mechanisms between applications and operating systems.

Trinity [6] is one of the first tools to use fuzzing techniques in kernel testing. The Trinity can expose those deeply buried kernel bugs by providing semi-intelligent arguments to a syscall being called. In detail implements a file descriptors pool, when a syscall requires a file descriptor, it will arbitrarily select one from the pool. Also, it shares the descriptor pool between threads, with the attempts to trigger more system crashes.

Syzkaller is a kernel fuzzing tool developed and maintained by Google. As one of the most state-of-the-art kernel fuzzing tools, it has explored many severe kernel bugs and can fuzz a wide range of kernels. It mainly uses the predefined system call descriptions to modeling the operating systems' interfaces. Then, Syzkaller compiles the descriptions into internal representations and performs the fuzzing process. Syzkaller is highly extendable; many academic researchers optimize fuzzing effectiveness by integrating other kernel validation techniques. For instance, Moonshine is proposed to distill the initial seeds of kernel fuzzer by tracing the sequences of system call when executing Linux test suites and performing the light-weight static analysis on the kernel source code. By far, Moonshine has successfully discovered many new Linux kernel vulnerabilities.

Despite that `Rtkaller` was extended from general kernel fuzz, the main difference between them reflects in the test case generation and the execution stage. In detail, to be compatible with RTOS fuzzing, `Rtkaller` proposes using tasks as fuzzing inputs. To better simulate real-world situations, `Rtkaller` emphasizes real-time operation triggering and adopts parallel execution to improve vulnerability detection efficiency.

5.3 RTOS Validation

RTOS has shown its growing importance in a wide range of industrial scenarios. Except for the traditional unit test approach [1], Ling Fang [4] proposes a formal model-based test method. It first constructs an abstract model based on the AUTOSAR standard. With a complete test suite generator, it then automatically starts the test process and verifies real-time operating systems that conform to the AUTOSAR standard. Instead of using AUTOSAR standard, Jean-Luc [15] constructs a complete model for OSEK standard and performs the conformance check based on the complete model. However, these methods cannot avoid the common pitfalls of formal verification, such as state explosion, excessive modeling time and lack of oracles and properties to be verified. These limitations are particularly prominent in operating systems, which makes it hard for real industry

practice. Different from them, Rtkaller implements a task based kernel fuzzing framework, for the automatic vulnerability detection of RTOS.

6 CONCLUSION

In this paper, we present Rtkaller, a state aware fuzzer for the vulnerability detection of the real-time operating system. Rtkaller performs a task-based fuzzing, which gives real-time code section in RTOS a higher chance of executing and thus expose more vulnerabilities. We evaluate Rtkaller on 8 different versions of rt-Linux. Compared with Syzkaller and Moonshine, Rtkaller is able to achieve higher code coverage at a faster speed and expose more bugs. Within the 142 confirmed bugs, 28 are previously unknown. In future studies, we plan to extend Rtkaller to support more RTOS and further improve the test case quality and diversity with relation learning.

ACKNOWLEDGEMENTS

We would also like to thank the anonymous reviewers for their valuable comments and input to improve our paper. This research is sponsored in part by the NSFC Program (No. 62022046) and National Key Research and Development Project (Grant No. 2019YFB1706203) and Key Research and Development Plan in Jiangxi Province Department of Science and Technology under Grant No.20171ACE50025.

REFERENCES

[1] Manthos A Tsoukarellas, Vasilis C Gerogiannis, and Kostis D Economides. 1995. Systematically testing a real-time operating system. *IEEE Micro* 15, 5 (1995), 50–60.

[2] Michael Barabanov. 1997. *A linux-based real-time operating system.* New Mexico Institute of Mining and Technology Socorro, New Mexico, USA.

[3] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-Based Whitebox Fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 206–215. https://doi.org/10.1145/1375581.1375607

[4] Ling Fang. 2012. Formal Model-Based Test for AUTOSAR multicore RTOS. http://icst2012.soccerlab.polymtl.ca/Presentation_Slides/LingFang.pdf.

[5] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Bellevue, WA) *(Security'12)*. USENIX Association, USA, 38.

[6] Dave Jones. 2012. Trinity: Linux system call fuzzer. https://github.com/kernelslacker/trinity.

[7] Michal Zalewski. 2014. American Fuzzy Lop (2.52b). https://lcamtuf.coredump.cx/afl.

[8] Dmitry Vyukov. 2015. Syzkaller: an unsupervised, coverage-guided kernel fuzzer. https://github.com/google/syzkaller. Accessed April 26, 2019.

[9] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-Based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 1032–1043. https://doi.org/10.1145/2976749.2978428

[10] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. 2016. UniSan: Proactive Kernel Memory Initialization to Eliminate Data Leakages. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 920–932. https://doi.org/10.1145/2976749.2978366

[11] 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

[12] 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

[13] Linux Kernel Developers. 2017. The kernel address sanitizer (KASAN)—the Linux kernel documentation. https://www.kernel.org/doc/html/latest/dev-tools/kasan.html

[14] Wenwen Wang, Kangjie Lu, and Pen-Chung Yew. 2018. Check It Again: Detecting Lacking-Recheck Bugs in OS Kernels. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS*

'18). Association for Computing Machinery, New York, NY, USA, 1899–1913. https://doi.org/10.1145/3243734.3243844

[15] Jean-Luc Béchennec, Olivier Henri Roux, and Toussaint Tigori. 2018. Formal model-based conformance verification of an OSEK/VDX compliant RTOS. In *2018 5th International Conference on Control, Decision and Information Technologies (CoDIT)*. IEEE, Thessaloniki, Greece, 628–634. https://doi.org/10.1109/CoDIT.2018.8394813

[16] Changming Liu, Deqing Zou, Peng Luo, Bin B. Zhu, and Hai Jin. 2018. A Heuristic Framework to Detect Concurrency Vulnerabilities. In *Proceedings of the 34th Annual Computer Security Applications Conference* (San Juan, PR, USA) *(ACSAC '18)*. Association for Computing Machinery, New York, NY, USA, 529–541. https://doi.org/10.1145/3274694.3274718

[17] 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

[18] Jie Liang, Yu Jiang, Yuanliang Chen, Mingzhe Wang, Chijin Zhou, and Jiaguang Sun. 2018. PAFL: Extend Fuzzing Optimizations of Single Mode to Industrial Parallel Mode. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. Association for Computing Machinery, Lake Buena Vista, FL, USA, 809–814.

[19] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim. 2018. Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA. https://doi.org/10.1109/SP.2018.00017

[20] 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

[21] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. 2019. FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1099–1114. https://www.usenix.org/conference/usenixsecurity19/presentation/zheng

[22] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzer: Finding Kernel Race Bugs through Fuzzing.. In *IEEE Symposium on Security and Privacy*. IEEE, San Francisco, CA, USA, 754–768. http://dblp.uni-trier.de/db/conf/sp/sp2019.html#JeongKSLS19

[23] Van-Thuan Pham, Marcel Boehme, Andrew Edward Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. 2019. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering* 1, 1 (2019), 1–1. https://doi.org/10.1109/TSE.2019.2941681

[24] J. Gao, Yiwen Xu, Yu Jiang, Zhe Liu, Wanli Chang, Xun Jiao, and Jiaguang Sun. 2020. EM-Fuzz: Augmented Firmware Fuzzing via Memory Checking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39 (2020), 3420–3432.

[25] Patrice Godefroid. 2020. Fuzzing: Hack, Art, and Science. *Commun. ACM* 63, 2 (Jan. 2020), 70–76. https://doi.org/10.1145/3363824

[26] Zhengxiong Luo, Feilong Zuo, Yuheng Shen, Xun Jiao, Wanli Chang, and Yu Jiang. 2020. ICS Protocol Fuzzing: Coverage Guided Packet Crack and Generation. In *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference (DAC '20)*. IEEE Press, Virtual Event, USA, Article 223, 6 pages.

[27] M. Xu, S. Kashyap, H. Zhao, and T. Kim. 2020. Krace: Data Race Fuzzing for Kernel File Systems. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 1643–1660. https://doi.org/10.1109/SP40000.2020.00078

[28] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. 2020. MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Virtual Event, USA, 2325–2342. https://www.usenix.org/conference/usenixsecurity20/presentation/chen-hongxu

[29] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. 2021. Industry Practice of Coverage-Guided Enterprise-Level DBMS Fuzzing. *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)* 1, 1 (2021), 328–337.

[30] Wikipedia contributors. 2021. RTLinux. https://en.wikipedia.org/wiki/RTLinux

[31] Richard Barry. Accessed 2003. FreeRTOS. Website. https://www.freertos.org/.

[32] Tool. Accessed April 5th, 2019. Peach Fuzzing Platform. Website. https://www.peach.tech.

[33] NVD. Accessed Jun 5th, 2018. CVE-2018-1000200. Website. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1000200.

[34] google. Accessed May 14th, 2018. syzbot. Website. https://syzkaller.appspot.com/upstream.

[35] SimonKagstrom. Accessed Nov 1st 2010. kcov. Website. https://github.com/SimonKagstrom/kcov.