Enhancing ROS System Fuzzing through Callback Tracing

Yuheng Shen
Tsinghua University
Beijing, China
shenyh20@mails.tsinghua.edu.cn

Jianzhong Liu
Tsinghua University
Beijing, China
liujz21@mails.tsinghua.edu.cn

Yiru Xu
Tsinghua University
Beijing, China
xuyr21@mails.tsinghua.edu.cn

Hao Sun
ETH Zurich
Zurich, Switzerland
hao.sun@inf.ethz.ch

Mingzhe Wang
Tsinghua University
Beijing, China
wmzhere@gmail.com

Nan Guan
City University of Hong Kong
HongKong, China
nanguan@cityu.edu.hk

Heyuan Shi*
Central South University
Changsha, China
hey.shi@foxmail.com

Yu Jiang*
Tsinghua University
Beijing, China
jiangyu198964@126.com

ABSTRACT

The Robot Operating System 2 (ROS) is the de-facto standard for robotic software development, with a wide application in diverse safety-critical domains. There are many efforts in testing that seek to deliver a more secure ROS codebase. However, existing testing methods are often inadequate to capture the complex and stateful behaviors inherent to ROS deployments, resulting in limited testing effectiveness. In this paper, we propose *R2D2*, a ROS system fuzzer that leverages ROS's runtime states as guidance to increase fuzzing effectiveness and efficiency. Unlike traditional fuzzers, *R2D2* employs a systematic instrumentation strategy that captures the system's runtime behaviors and profiles the current system state in real-time. This approach provides a more in-depth understanding of system behaviors, thereby facilitating a more insightful exploration of ROS's extensive state space. For evaluation, we applied it to four well-known ROS applications. Our evaluation shows that *R2D2* achieves an improvement of 3.91× and 2.56× in code coverage compared to state-of-the-art ROS fuzzers, including *Ros2Fuzz* and *RoboFuzz*, while also uncovering *39* previously unknown vulnerabilities, with *6* fixed in both ROS runtime and ROS applications. For its runtime overhead, *R2D2* maintains an average execution and memory usage overhead with 10.4% and 1.0% in respect, making *R2D2* effective in ROS testing.

CCS CONCEPTS

• **Computer systems organization → Robotics**; • **Software and its engineering → Software testing and debugging**.

*Heyuan Shi and Yu Jiang are correspondence authors.

KEYWORDS

Fuzz Testing, ROS, Bug Detection

ACM Reference Format:
Yuheng Shen, Jianzhong Liu, Yiru Xu, Hao Sun, Mingzhe Wang, Nan Guan, Heyuan Shi, and Yu Jiang. 2024. Enhancing ROS System Fuzzing through Callback Tracing. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650212.3652111

1 INTRODUCTION

The increasing integration of robotics in daily life has positioned ROS (Robot Operating System 2) as the primary framework for robotics application development. With ROS-based systems becoming increasingly prevalent, ensuring their security and robustness is of paramount importance. For example, a study uncovered 13 highly critical vulnerabilities within 6 DDS implementations used by ROS [6]. These bugs can lead to Denial of Service (DoS) attacks and make thousands of devices worldwide vulnerable, including robots owned by NASA, Siemens, and Huawei. Therefore, it is essential to proactively identify and address these vulnerabilities to ensure the robustness, security, and overall performance of ROS-based system, reducing the risk of catastrophic crashes.

Several works have been aimed at ensuring the reliability and security of ROS. For instance, *Ros2Trace* [5] uses manually instrumented tracers to conduct performance profiling of the ROS system. Fuzz testing (fuzzing) [1, 4, 22, 23, 32, 35, 36, 40], especially code coverage-guided fuzzing, has emerged as a promising approach for uncovering bugs in ROS systems. This method involves generating random inputs for a target program and focusing on those that lead to new code coverage, a strategy proven highly effective in various contexts. Google's oss-fuzz, for instance, has identified over 36,000 bugs in more than 1,000 open-source projects using this approach. Several studies attempt to apply fuzzing for bug detection in ROS. *Ros2Fuzz* [13], adapts the AFL [20] to testing certain ROS interfaces. However, this may fall short of providing comprehensive fuzzing for the entire ROS system. There are works attempting to fuzz the entire ROS system. Rozz [39], proposes to use the multi-dimensional input generation approach combined with distributed code coverage collection methodology to detect memory-related bugs in ROS applications. RoboFuzz [15] extracts real-world physical properties from the target ROS system to identify correctness bugs; combined with the code coverage, these oracles are further used to guide the fuzzing process.

Motivation: Nevertheless, relying primarily on code coverage proves inadequate for ROS testing. ROS is a distributed system where nodes communicate through message passing, each hosting multiple callbacks responding to diverse messages or events. A significant portion of logic and state transitions in ROS occurs within these callbacks, handling everything from sensor data processing to actuator control. While code coverage measures the extent of executed code, it may overlook the quality or context of execution in ROS. Different sequences of message handling may execute the same set of callbacks but in different orders or under different system states, leading to diverse behaviors and potential vulnerabilities not evident through code coverage alone. Given a scenario in a robotic arm where the same set of movement instructions (thus the same code coverage) can lead to drastically different physical outcomes based on the order and timing of callbacks, which code coverage would not capture.

To address this limitation, we propose utilizing the `callback trace` to guide the ROS system fuzzing. The callback trace, representing the temporal sequence of callback interactions, provides a more detailed insight into aspects such as callback execution durations and message throughput. Unlike code coverage, which may exhibit little variety across different inputs, the callback trace serves as a more accurate indicator of the system's internal state transitions. Leveraging the callback trace during fuzzing provides a deeper understanding of the system's state transitions, enabling the discovery of inputs that trigger more system behaviors and improving fuzzing performance.

Challenge: The introduction of callback trace involves two key challenges: capturing and understanding state transitions in ROS's volatile and asynchronous environment and generating high-quality payloads informed by these state changes. First, accurately profiling the ROS system's state transitions demands a method to capture and analyze its interactions in real-time, which is a task made difficult by ROS's volatile, asynchronous nature and varied application components. Existing methods for automatically acquiring such state information from ROS are insufficient because they are not tailored to handle this variety and either require manual specifications or cannot provide such information in real-time. Also, due to ROS's distributed nature, information like message handling or callback scheduling is processed across different system components, and collecting such information requires an in-depth understanding of the ROS architecture. Thus, an automatic approach is needed to accurately profile the system interactions during testing, thereby reflecting the states of ROS.

Second, improving the exploration of system states and achieving effective fuzzing requires an approach that leverages state changes to generate high-quality payloads. However, ROS has extensive input space, as it is deployed in diverse environments that need to respond to various sensor data, control commands, and user configurations. Previous fuzzing approaches mainly rely on code coverage or manually derived oracles to guide the input generation. Such an approach may fail in capturing the multifaceted behaviors of the entire system, such as variations in execution duration and scheduling order, thereby failing to explore ROS system's state space efficiently. Consequently, it is essential to analyze the collected interaction behaviors as guidance and generate high-quality payloads to better explore complex behaviors within ROS, thereby uncovering issues that might have been overlooked before.

Solution: To address the challenges above, we propose *R2D2*, a callback trace-guided fuzzer for the entire ROS system. *R2D2* performs fuzzing through the following procedures. First, to better understand the system's behavior *R2D2* instruments customized tracer within the different components of ROS runtime to gather execution interactions systematically. Then, *R2D2* profiles the callback trace based on collected interactions, containing the callback execution duration, executor scheduling operations, and message throughput, to indicate the state transitions of ROS. Furthermore, to generate high-quality payloads, *R2D2* analyzes the callback trace during testing to identify inputs that trigger new system states and guide the generation of high-quality inputs. This feedback-based test case generation approach generates higher-quality inputs, boosting the efficiency of fuzzing the ROS system.

We evaluate *R2D2* on four ROS applications. In detail, *R2D2* found a total of *39* bugs in both ROS runtime and applications, with *6* fixed. Also, compared to existing fuzzing approaches, *Ros2Fuzz* and *RoboFuzz*, *R2D2* achieves an average 3.91× and 2.56× on coverage improvement in respect. Furthermore, we implemented *R2D2-*, which is *R2D2* without the callback trace guidance, where *R2D2* demonstrates an average of 0.27× coverage improvement. For the instrumentation overhead, compared to *Ros2Trace* and the vanilla ROS system, *R2D2* has an average of 10.4% and 1.0% overhead in terms of the execution latency and memory consumption.

Contribution: This paper makes the following contributions:

- We propose *R2D2*, a callback trace-guided fuzzer, leveraging ROS's runtime state to guide the fuzzing process. This method aids in generating high-quality inputs and enables an effective exploration of ROS's extensive state space.
- We present a real-time system behavior collection and profile strategy. This strategy facilitates efficient monitoring and analysis of ROS's runtime states, based on which, *R2D2* can generate more in-depth test cases.
- For evaluation, *R2D2* can detect *39* bugs, with *6* fixed, achieving a higher code coverage compared to state-of-the-art ROS fuzzers, while maintaining a low monitoring overhead.

2 BACKGROUND AND RELATED WORKS

2.1 Robot Operating System 2

The Robot Operating System 2 (ROS) is an open-source software stack that has become the de facto standard for developing various robotic systems. Figure 1 demonstrates the overall architecture of the ROS system. In detail, the ROS system mainly comprises two parts: the ROS application and the ROS runtime, and the ROS system can be deployed across a range of operating systems, including Linux, MacOS, and various RTOS.

The **ROS runtime** serves as the backbone, providing fundamental ROS functionalities, such as message passing and callback scheduling. The ROS runtime is a multi-layer structure, with different components at each layer. The top layer of ROS runtime encompasses multiple client library implementations, including RCLCPP, RCLPY, and RCLJAVA. These libraries offer different API implementations in various languages to facilitate higher-level application developments. Below the top layer is the ROS Client Library (RCL), providing standard interfaces for the above API implementations. Furthermore, to ensure compatibility with different Data Distribution Service (DDS) implementations at the bottom layer, RCL relies on the ROS Middleware Interface (RMW) as an intermediary bridge. This compatibility enables efficient communication between distributed components. The various DDS implementations, such as FastRTPS, CycloneDDS, and eProsima, provide ROS with different message handling and QoS (Quality of Service) mechanisms.

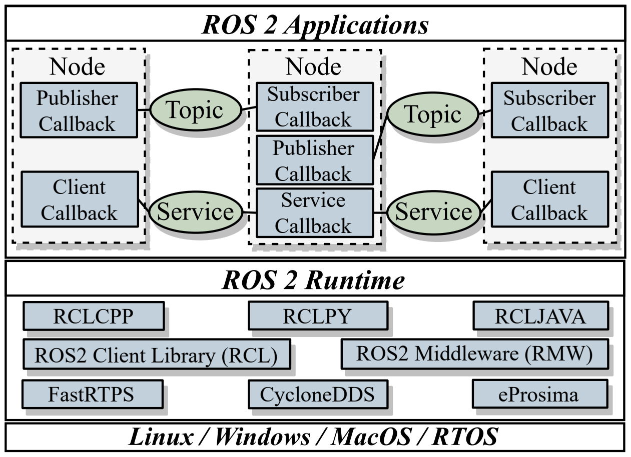


Figure 1: The Architecture of the ROS system.

The **ROS application** is designed to perform specific tasks. Typically, each application consists of multiple nodes, each with different `callbacks` that communicate with one another through topics and services. In detail, the node is responsible for a single, modular purpose. Meanwhile, the callback within it is a function invoked in response to specific events, such as receiving a message or a service request. Managed by the executor within RCLCPP, these callbacks are scheduled for execution to meet real-time requirements of ROS. Also, the executor ensures that these callbacks interact coherently with various messaging events, including topics and services. Concretely, the execution of ROS involving the executor scheduling serails of callbacks to execute on the incoming of different messages, and the temporal order of the above execution can be referred to as the `callback trace`. Understanding this callback trace is of critical importance, as it can reflect the system's runtime behaviors and states.

2.2 ROS Testing

The robustness and security of ROS have gained more and more attention due to its mission-critical use cases. Consequently, various methodologies have been employed to benchmark the performance and identify potential vulnerabilities for ROS.

In detail, performance benchmarking tools [2, 12, 19, 21, 33] aim to collect runtime metrics of ROS, evaluating key performance indicators such as resource consumption, message throughput, and execution latency to manually identify runtime issues, such as performance bottlenecks or timing anomalies. For example, *Ros2Trace* [2] utilizes the Linux Trace Toolkit Next Generation (LTTng) [8] to insert tracers at different points in the ROS runtime, enabling a comprehensive runtime behavior collection and later an in-depth performance profiling. The performance_test [33] measures the performance of a ROS system by setting up different pub/sub configurations, to profile performance metrics, including latency, CPU usage, and resident memory.

Moreover, several testing tools have been developed to ensure ROS's security. Specifically, fuzz testing (fuzzing) [3, 7, 10, 11, 16, 24, 27, 28, 31, 37, 38, 40–42], has been particularly effective in bug detection. It generates random test cases for the System Under Test (SUT) and monitors any erroneous behaviors. To achieve a higher coverage and test the SUT more thoroughly, many fuzzers adopt the coverage-guided fuzzing technique. By giving those seeds that trigger new coverage a higher chance of mutation the fuzzer can increase the probability of finding new paths. Notable coverage-guided fuzzers like AFL [20] and Syzkaller [34] have identified numerous vulnerabilities across various programs. Given the effectiveness of fuzzing, many research efforts in recent years have sought to apply fuzzing techniques to test ROS. For instance, *Ros2Fuzz* [13] is built upon AFL; it extracts specific ROS interfaces from the target component to generate a driver that emulates a publisher or client, sending payloads to the targeted component. However, *Ros2Fuzz* only testing user-specified interface, can hardly effectively test the ROS runtime. There has been research toward testing the entire ROS system. Rozz [39] and *RoboFuzz* [15] have been developed explicitly for testing the ROS system. Rozz collects and merges the coverage from different ROS components to guide the fuzzing process, thereby better exploring ROS's code space. *RoboFuzz* manually summarizes real-world physical laws and documented specifications from the target ROS application as testing oracles, thereby detecting potential behavioral anomalies. However, these works primarily use code coverage or manually summarized specifications from target applications or runtime components, whereas they can hardly understand the comprehensive and detailed behaviors and states within the entire ROS system, resulting in a relatively limited fuzzing performance.

3 MOTIVATION

In this work, we focus on detecting memory-related and concurrency-related bugs throughout the entire ROS system, including the ROS runtime and ROS applications. However, as a system designed to cater to specific tasks, ROS tends to operate along predetermined control pathways, in responding to different income events, i.e., they have little coverage variety. Therefore, the code coverage may be insufficient to reflect the system's state, and the unawareness of the system state during testing can lead to potentially insufficient fuzzing performance.

We use Figure 2, a system hang bug [29] for a detailed illustration. This bug is triggered when a ROS application starts a talker thread and a listener thread, and the talker thread is designed to

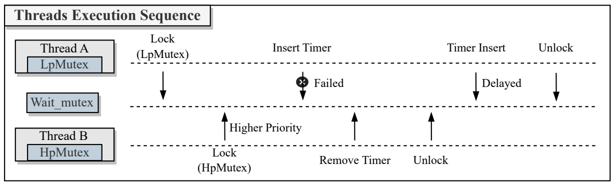


Figure 2: Execution sequence of a system hang bug in ROS.

publish a message every second. During execution, the publish frequency varies irregularly and causes unexpected system delays. The inconsistency is because threads in ROS have mutex classes (low-priority mutex `LpMutex` and high-priority mutex `HpMutex`), which share the same `wait_mutex_` for the timer callback insertion, where the timer callback is used to schedule different callbacks' execution. When the `wait_mutex_` is locked by thread A, a low-priority mutex, it could be taken by a thread with a higher-priority mutex, like thread B. This led to thread A failing to insert the timer on time, thereby causing inconsistent timing behavior.

This bug is located in the executor's scheduling logic, an easy-to-cover and commonly executed code path in ROS. However, previous fuzzing approaches use code coverage or manually derived oracles. They may find it difficult to trigger such bugs, as the guidance strategies they adapted are not able to perceive such a system's internal state changes, such as the irregular callback latency, and thus, it is hard to generate payloads that cover such an erroneous state. Moreover, despite existing benchmarking tools like *Ros2Trace* being capable of collecting the system's internal behaviors, such as the callback duration and message throughput, they are only limited to data collection and lack real-time analysis capabilities, making them insufficient for testing the ROS system. Hence, to boost fuzzing performance and better capture the state changes within ROS, we can use a *callback trace*, which consists of the temporal sequence of the system's interactions, such as callback execution durations, executor scheduling, and message passing, as a reflection of the system states. We can guide the generation of high-quality payloads through real-time profiling and analysis of the callback trace, facilitating a more comprehensive exploration of the ROS's state space. To conduct the callback trace guided fuzzing for ROS system, we need to address the following two challenges.

Collecting artifacts that reflect state information. This requires monitoring and interpreting complex interactions such as callback registration, callback execution, and message passing within the system. This is difficult to perform, since ROS exhibits complex and asynchronous temporal behaviors at runtime. Notably, it needs to accommodate various use cases, each has its unique component composition, such as different callbacks' namespace and connectivity. Additionally, ROS separates its functionalities into different components, so that information like message handling or callback scheduling is processed across different system layers. As we can refer from the motivating example to profile such an erroneous state, we need to capture information, including the execution of different callbacks, the executor's scheduling behavior, and the message throughput. Also, such information may hide in different ROS components, for example, the execution of the callback can be found at the RCL and RCLCPP layer, whereas the message throughput can be found at the DDS layer. Thereby, acquiring such information from different applications is complex, since it requires a comprehensive understanding of the ROS system. To overcome this challenge, a systematic and automatic approach is needed that can not only accurately capture the desired execution interaction in real-time, but also adapt to the intricate and asynchronous within ROS, providing a comprehensive understanding of the system state.

Leverage callback traces to guide input generation. This requires analyzing the current callback trace, to identify if the current input triggers any new state transition. Essentially, ROS is a system characterized by an extensive input space such as various sensor data, control commands, and user configurations. Therefore, we need to generate high-quality input to conduct efficient testing. As we can see from the example, despite the various input vectors and configurations ROS has, this bug can only be triggered when using the service as the input vector, and the publishing frequency should be one message pre-second. Previous approaches mostly use code coverage or target application's specification as guidance, this information may provide less deep insight in understanding the ROS system states, leading to generating less high-quality inputs and rendering a limited fuzzing performance. Therefore, it is essential to harness the collected interaction behaviors and leverage them as guidance to generate high-quality input, which can better explore the complex behaviors within ROS, allowing for a more comprehensive and efficient exploration of the system.

4 DESIGN

To address the above challenges, we propose *R2D2*. It leverages real-time behavior profiling to construct the callback trace as the state identifier, which is later used as guidance for the generation of high-quality inputs to better explore the ROS state space and improve the bug detection capabilities. In contrast with established tools, *R2D2* removes code coverage guidance completely for a more effective callback trace guidance mechanism.

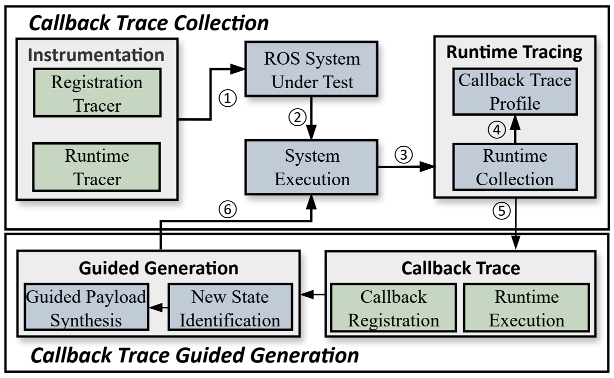


Figure 3: Overall Workflow of *R2D2*.

The overall architecture of *R2D2* is depicted in Figure 3. As demonstrated in the figure, *R2D2* contains two phases: the callback trace collection phase and the callback trace guided generation phase. During the collection phase, *R2D2* uses customized tracers to instrument the ROS runtime, capturing essential system behaviors such as callback execution, executor scheduling, and message passing. Within each fuzzing loop, the instrumented tracers log real-time execution behaviors. *R2D2* subsequently aggregates and profiles the recorded information into the callback trace. Later, during the generation phase, by utilizing this trace as the system state indicator, *R2D2* identifies and prioritizes payloads that induce new system states, thereby refining the future payload synthesis. This iterative approach enables *R2D2* to continually generate high-quality inputs, optimizing the ROS state space exploration and improving its overall fuzzing efficiency.

4.1 Runtime Callback Trace Collection

Perceiving the system states during testing requires accurately acquiring the execution behaviors and effectively processing these behaviors into the callback trace. However, different ROS applications possess unique system configurations, with each having its own set of callbacks and distinct patterns of inter-callback connections These components exhibit complex asynchronous temporal behaviors during execution, making isolating pertinent information for analysis challenging. Therefore, we introduce a multi-layered tracer instrumentation strategy for capturing precise runtime behaviors. Furthermore, we propose to profile the tracer-collected data into the callback trace, facilitating a more comprehensive analysis of the system's runtime state during testing.

4.1.1 ***Runtime Temporal Behaviors Tracing.*** As elaborated in Section 2, ROS incorporates a multi-layer architecture to provide basic functionalities for higher-level applications; this involves components such as the RCL and language-specific libraries like RCLCPP. Therefore, to comprehensively capture ROS's execution behaviors, we instrument different tracers within the RCL and RCLCPP layers. Figure 4 illustrates the instrumentation process. Specifically, we deploy two types of tracers: registration tracers (green section) and runtime tracers (yellow section). Registration tracers focus on capturing callback registration events, while runtime tracers are designed to monitor the initiation and termination points of callback execution. Also, we utilize different buffers to retrieve the collected data in real-time. The implementation detail can refer to Section 4.3.

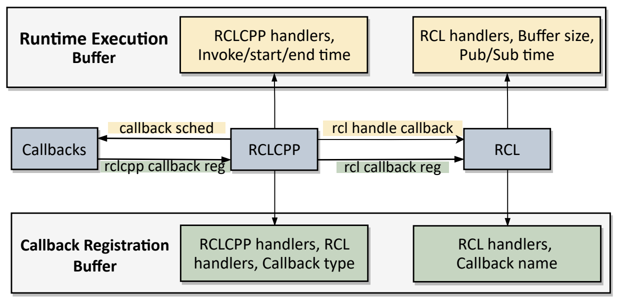

Runtime Execution Buffer
RCLCPP handlers, Invoke/start/end time
RCL handlers, Buffer size, Pub/Sub time
callback sched
rcl handle callback
Callbacks
RCLCPP
RCL
rclcpp callback reg
rcl callback reg
Callback Registration Buffer
RCLCPP handlers, RCL handlers, Callback type
RCL handlers, Callback name

Figure 4: Diagram of the instrumentation process.

Registration Tracer. Registration tracer is invoked by different callback registration events, and it is designed to extract a variety of callback attributes, including the callback's namespace, allocated address in different layers (i.e., different handlers), and its specific types (i.e., subscription, timer, and service). In concrete, the ROS system, especially the RCLCPP and RCL layer, undertakes the responsibility of registering these callbacks by associating them with callback names and allocating corresponding addresses (handler) at multiple layers, namely the RCLCPP handler and RCL handler. To facilitate the capture of the information mentioned above, we capture the callback registration event in both the RCLCPP and RCL layer, and we instrument the RCLCPP and RCL layers with two tracers: `rclcpp_callback_init()` and `rcl_callback_init()`.

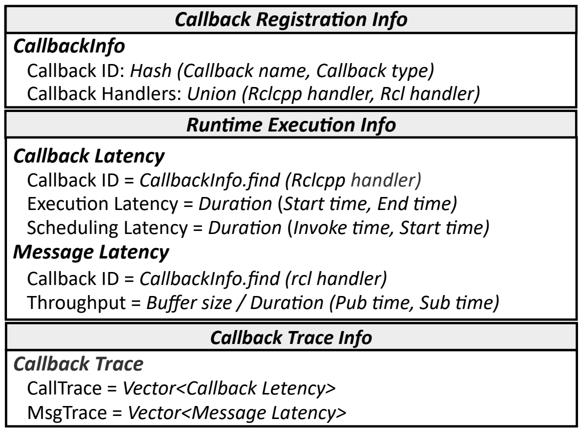
When ROS applications start, it registers different callbacks, this is intercepted by the `rclcp_callback_init()` tracer, which records pertinent details such as the RCLCPP handler, RCL handler, and the types of the callback—whether it is a subscription, timer, or service. This registration activity is further propagated to the RCL layer, where the `rcl_callback_init()` tracer logs additional attributes, including the callback name and the RCL handler. Furthermore, the registration tracer writes the collected data into the callback registration buffer, and *R2D2* reads the buffer during testing, facilitating the construction of the callback trace.

Runtime Tracer. On the other hand, the runtime tracer is designed to monitor the system's execution behaviors. The essence of the execution of the ROS system lies in the executor's role in scheduling the execution of callbacks in response to different events, most commonly the recipient of messages. As a result, runtime tracers primarily focus on the monitor callbacks' scheduling behaviors and messages' publish/subscribe behaviors.

To capture the above behaviors, we employ different tracers, including `executor_execute()`, `callback_start()`, and `callback _end()` that record various metrics, including the RCLCPP handler of the targeting callback, the timestamp of its scheduling time, starting time, and the ending time. Additionally, we utilize the `rcl_take()` tracer to capture the detailed information of message passing. This tracer is designed to log communicating information, including the RCL handlers, the size of the incoming message buffer, and the timestamps associated with the publishing and subscribing activities. Upon the completion of the execution phase, the runtime tracer commits the aggregated data to the runtime execution buffers, and *R2D2* retrieves this data to construct the precise callback trace. The collected information facilitates *R2D2* with a comprehensive perspective on both the ROS applications and ROS runtime. By understanding this crucial information, *R2D2* can effectively depict the target system's structure and behavior, ultimately facilitating the construction of the callback trace.

4.1.2 ***Callback Trace Profile.*** Once we collect the registration information and runtime behaviors, we can then construct the callback trace. The overview description of the callback trace is delineated in Figure 5.

Initially, the raw registration data is processed to form Callback-Info structures. These structures contain the callback identifier (ID) and corresponding handlers (address) at different layers, including the RCLCPP handler and the RCL handler. The callback ID is generated through a hashing function that considers the callback name and type, serving as a unique identifier for differentiating among various callbacks. Subsequently, based on the runtime data


Callback Registration Info

CallbackInfo
Callback ID: *Hash (Callback name, Callback type)*
Callback Handlers: *Union (Rclcpp handler, Rcl handler)*

Runtime Execution Info

Callback Latency
Callback ID = *CallbackInfo.find (Rclcpp handler)*
Execution Latency = *Duration (Start time, End time)*
Scheduling Latency = *Duration (Invoke time, Start time)*

Message Latency
Callback ID = *CallbackInfo.find (rcl handler)*
Throughput = *Buffer size / Duration (Pub time, Sub time)*

Callback Trace Info

Callback Trace
CallTrace = *Vector<Callback Letency>*
MsgTrace = *Vector<Message Latency>*

Figure 5: Description of the callback trace.

within each fuzzing loop, we focus on calculating two key latency metrics: the callback latency and the message latency. The callback latency includes the callback ID, execution latency, and scheduling latency. The latencies for each callback are determined according to the obtained handler, in conjunction with the logged timestamps for invocation (when the executor signals readiness for execution), start, and end of execution. Similarly, message latency is determined by considering the callback ID, message buffer size, and timestamps of message publishing and subscribing, which allows us to calculate the transfer throughput. Finally, we amalgamate the callback latency and message latency metrics to construct the callback trace. This trace is represented as two distinct vectors: one for callback latency and another for message latency. Also, during the execution, the executor may schedule a callback to execute multiple times, hence, the callback latency and message latency can contain repeated elements.

4.2 Callback Trace Guided Generation

Once we profile the callback trace, we can leverage it to guide future input generation. To generate high-quality payloads, we need to accurately acquire the input specifications from the target ROS application and efficiently analyze the profiled callback trace, identify any potential state transitions, and utilize the above information to facilitate the payload synthesis.

4.2.1 *State Identification.*

As mentioned above, the derived callback trace encapsulates various performance metrics, including the execution sequence of callbacks and messages and their associated latencies and throughput. After acquiring the current callback trace, we conduct an initial analysis to compute the aggregate latency for each callback and the mean latency for each message within the trace. Concretely, to perceive the state changes, we maintain two global structures: the callback graph and the global callback latency. The callback graph represents the temporal sequence of callback invocations, while the global callback latency contains the overall callback latency and message throughput, representing the average performance for each encountered callback and message.

Specifically, we identify new system states based on the following indicators: (1) Whether the callback trace introduces a new sequence of execution, offering insights into previously unexplored interactions. (2) Whether there is a significant deviation in the latency of a specific callback from the established average benchmark value. (3) Whether the throughput of a specific message is considerably below the average benchmark. To establish the average benchmark value, we profile the latency for each input for certain times, i.e., we randomly generate input to the system, collect the callback trace, and utilize statistical methods to identify appropriate values. We reason that overhigh or overlow benchmark values will result in either (1) deficiency in state exploring or (2) a large number of trivial state changes being found. The above two scenarios may affect the overall state identification, but it has a limited effect on the overall fuzzing performance, we further discuss this on Section 6.

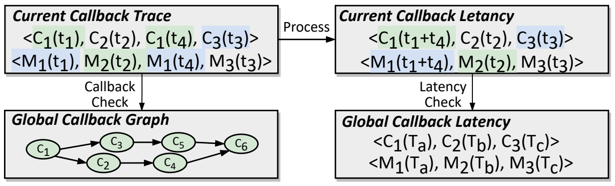

Current Callback Trace
$<C_1(t_1), C_2(t_2), C_1(t_4), C_3(t_3)>$
$<M_1(t_1), M_2(t_2), M_1(t_4), M_3(t_3)>$
Process
Current Callback Letancy
$<C_1(t_1+t_4), C_2(t_2), C_3(t_3)>$
$<M_1(t_1+t_4), M_2(t_2), M_3(t_3)>$
Callback Check
Latency Check
Global Callback Graph
C_1 C_2 C_3 C_4 C_5 C_6
Global Callback Latency
$<C_1(T_a), C_2(T_b), C_3(T_c)>$
$<M_1(T_a), M_2(T_b), M_3(T_c)>$

Figure 6: Diagram of the new state identification process.

To better illustrate our methodology, Figure 6 includes the process of identifying a new system state. In detail, the callback trace analysis is conducted after the execution of each payload. First, we check whether the current callback trace contains any new execution sequence representing a new edge in the global callback graph. If the current callback trace introduces a new execution order, we add corresponding edges to the callback graph. Then, we check whether the latency associated with a particular callback significantly exceeds the established benchmark. Particularly, from the current callback trace, we first calculate the current callback latency, including the message latency and the callback latency. We compare it with the global callback latency, to check if we find a new callback or if the current latencies for the message and the callback have a significant deviation from our benchmark. If we find that, we will update the global callback latency correspondingly. Each new state serves as a guide for generating subsequent inputs, thereby enriching the exploration of the system's state space.

4.2.2 *Guided Payload Synthesis.*

To more efficiently explore the state space of ROS, generating payloads conforming to ROS's interface specifications is crucial. However, ROS has multiple dimensional input vectors with highly structured interfaces, including topic messages and service requests with different data types and formats. To address this, we extract complex interface specifications and subsequently conduct the guided payload generation.

Initially, *R2D2* conducts a dry run, which only boosts the system without sending any input, to extract all interface specifications from the ROS system. This includes a comprehensive list of interfaces—topics and services along with their associated data files, message types, and formats. Then, during the fuzzing phase, *R2D2* examines the current payload pool, if the pool is empty, *R2D2* selects an interface randomly from the extracted specifications and generates payloads accordingly; else, *R2D2* selects a payload that previously triggered new state for mutation. The mutation is conducted recursively based on data files from the interface specification. These prepared payloads are then sent to the ROS system for execution. After execution, *R2D2* checks if the payload induces any system crashes or new system states. If a new state or crash is triggered, the payload is preserved in the pool for future iterations. Through this iterative process, *R2D2* continually produces high-quality payloads, enabling more comprehensive and deep exploration of the ROS system's state space.

4.3 Implementation

We implement *R2D2* using Rust for the core fuzzing framework and C++ for the tracers. *R2D2* can fuzz test the ROS runtime and is adapted to ROS applications, including Turtlesim [9], TurtleBot3 [30], Navigator2 [26], and Autoware [14]. The overview of *R2D2*'s architecture is presented in Figure 7.

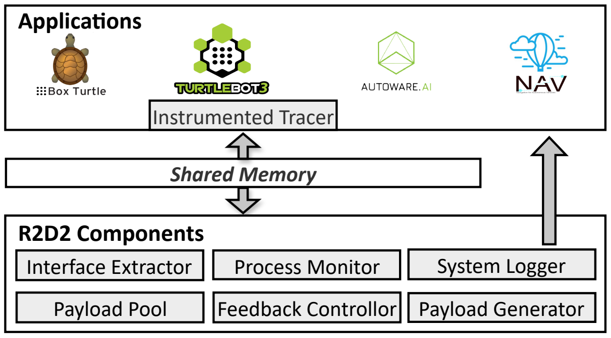


Figure 7: Diagram of the *R2D2* implementation.

First, We extend the capabilities of *Ros2Trace* to profile the callback trace. In detail, we manually add more tracepoints at the RCLCPP and RCL layer to capture a more comprehensive runtime behavior, such as different callback registrations and message transmissions. Also, we extended the capabilities of original tracers to collect more comprehensive runtime data, such as message buffer sizes and publish/subscribe timestamps. Furthermore, we utilize shared memory to write out collected data in real-time. We instrument the shared memory initialization tracer during the initialization of the RCL layer, which will allocate distinct shared memory buffers to record different structure information like callback execution and message passing. Moreover, to accommodate for ROS's asynchronous nature and the abundance of information produced during its runtime, each shared memory object is a circular buffer guarded with a mutex lock for thread- and memory-safety.

To conduct fuzzing, *R2D2* employs an interface extractor to acquire all interface specifications. This allows *R2D2* to be aware of the input vectors that the target ROS system can accept, allowing to generate more comprehensive test payloads. By utilizing this information, the payload generator can craft payloads for the SUT, while the instrumented tracer records runtime data in preallocated shared memory regions. The feedback collector then profiles this data into callback traces, which are used to identify new system states and guide further input generation. If *R2D2* identifies a payload that triggers new system states, it then adds the payload to the payload pool for further mutation. The process monitor checks the runtime status of the entire ROS system; it watches for any unexpected exit codes from the system; in this way, *R2D2* can detect program crashes, memory corruption, and concurrency issues within the target ROS system. Finally, *R2D2* maintains logs of all fuzzing-related activities, including crash logs and system stats.

5 EVALUATION

We list the following research questions to help us understand *R2D2*'s performance and effectiveness.

- **RQ1:** Is *R2D2* able to uncover new bugs in ROS?
- **RQ2:** Is *R2D2*'s callback trace guidance mechanism effective in conducting a more in-depth testing, compared with other fuzzing methods?
- **RQ3:** What is the performance overhead of *R2D2*'s instrumentation strategy?

5.1 Evaluation Setup

We conduct our evaluation of *R2D2* on four widely-used ROS applications: Navigator2, TurtleBot3, Turtlesim, and Autoware. We choose ROS *humble* [18] and *rolling* [25] as the target runtime version, as *humble* is the most stable and widely used ROS version, while *rolling* is the latest release version at the time of writing. To answer **RQ1**, we compile the ROS runtime and ROS applications using Clang with ASAN and TSAN enabled. To answer **RQ2**, and provide a comprehensive coverage comparison, we select *RoboFuzz* and *Ros2Fuzz* as baseline fuzzers, which are the state-of-the-art and open-sourced ROS fuzzers, and choose Navigator2, TurtleBot3, and Turtlesim as the coverage comparison target. To ensure a fair comparison, we instrument both the ROS runtime and applications with SanitizerCoverage, allowing us to collect coverage data from all activated components across the entire ROS system. To determine the benchmark values, we conducted a sampling over an empirical period of 2 hours. To further show the effectiveness of the callback trace guidance, we implement *R2D2-*, which is *R2D2* minus the callback trace guidance, and compare the amount of the bugs detected and the statistics of code coverage between *R2D2* and *R2D2-*. To answer **RQ3**, we compile three different ROS runtimes: one with *R2D2*'s instrumentation, another instrumented with *Ros2Trace*, and a third version with all tracers removed. We utilize the `performance_test` component to measure the statistics for runtime latency and memory usage overhead.

We perform our evaluation on a server with a 64-core AMD EPYC 7742 CPU (2.25GHz) and running Ubuntu 22.04. Since ROS systems often require a graphical user interface, we employ the X virtual frame buffer (Xvfb) for testing in the absence of a connected display, a common practice in graphical application testing. All experiments are conducted on the same hardware for 24 hours and repeated five times, following the fuzzing evaluation best practice [17].

5.2 Bug Detection Capabilities

To answer **RQ1** and evaluate *R2D2*'s bug detection capabilities in the ROS system, we collected and analyzed the crashes reported by *R2D2*. In detail, *R2D2* found *39* previously unknown bugs, as listed in Table 1.

Table 1: Previously Unknown Bugs Found by *R2D2*

#	Scope/Module	Bug Types	Operations
1	Runtime/RCUTILS	OOM	rcutils_reallocf
2	Runtime/Fastrtps	Overflow	copy_from_fastrtps_guid_ to_byte_array
3	Runtime/Rclcpp	UAF	AnySubscriptionCallback
4	Runtime/Cyclone	Data-race	gc_delete_writer / entity_guid_eq_wrapper
5	Runtime/eProsima	Data-race	~Condition / wait
6	Runtime/eProsima	Deadlock	notify
7	Runtime/eProsima	Data-race	do_timer_actions / register_timer_nts
8	Runtime/eProsima	Deadlock	read_or_take
9	Runtime/eProsima	Deadlock	sendSync
10	TurtleBot3/Rviz	SEGV	getParent
11	TurtleBot3/Ompl	SEGV	~RigidBodyEnvironment
12	TurtleBot3/Ompl	Overflow	getRadius
13	TurtleBot3/Ompl	Overflow	setStartAndGoalStates
14	TurtleBot3/Ompl	Overflow	Constraint::project
15	TurtleBot3/Ompl	Overflow	getControl
16	Navigator2/BT.CPP	SEGV	bt3_log_cat
17	Navigator2/Planner	Double-free	test_planer_is_path_valid
18	Autoware/Localization	SEGV	aged_object_queue
19	Autoware/Vehicle	Nullptr-deref	validate_data
20	Autoware/Vehicle	Nullptr-deref	get_row_index
21	Autoware/Perception	Overflow	tlwh_to_xyah
22	Autoware/Perception	SEGV	_ccrrt_dense
23	Autoware/Control	Overflow	filt_vector
24	Autoware/Utils	Overflow	arange
25	Runtime/rmw_fastrtps	Data-race	C̃ondition / wait
26	Runtime/eProsima	Data-race	get_listener_for / delete_datawriter
27	Runtime/eProsima	Data-race	set_status / get_subscription_matched_ status
28	Runtime/Rclcpp	Deadlock	lifecycle_service_client
29	Runtime/Rclcpp	Deadlock	action_client
30	Runtime/eProsima	Data-race	get_publication_ matched_status / set_status
31	Runtime/eProsima	Data-race	set_read_communication_status / set_ status
32	Runtime/Rclcpp	Deadlock	double_unlock
33	Runtime/eProsima	Data-race	deliver_sample_nts / change_received
34	Runtime/geometry2	Data-race	create_new_change / unsent_change_ added_to_history
35	Runtime/eProsima	UAF	write
36	Runtime/Fastrtps	SEGV	new_allocator_impl
37	Runtime/ROSIDL	MemLeaks	get_typesupport_handle_function
38	Runtime/tlsf_cpp	MemLeaks	initialize
39	Runtime/tlsf_cpp	MemLeaks	tlsf_heap_allocator

Among the detected *39* bugs, *8* bugs have been confirmed (bug # 1-5, 17-19) *6* bugs have been fixed by corresponding maintainers (bug # 1, 2, 4, 5, 17, 19), and the rest have been submitted to corresponding maintainers, awaiting further confirmation. As *R2D2* can test the entire ROS system, we can find bugs including the ROS runtime and ROS applications. In detail, *24* bugs were found within the ROS runtime, and *15* bugs were found throughout different ROS applications. Specifically, *R2D2* found 9 in *humble*, 15 in *rolling*, *6* in TurtleBot3, *2* in Navigator2, and *7* in Autoware. Also, with the help of ASAN and TSAN, *R2D2* can detect both memory issues and concurrency issues. Specifically, we find a total number of *23* memory-related bugs and *16* concurrency-related bugs. The identification of these critical issues is significantly facilitated by *R2D2*'s use of callback trace guidance. Also, incorporating callback trace information enables *R2D2* to delve into the deeper state space of the target code. This is noteworthy because many vulnerabilities reside in code segments that were frequently executed and tested but were previously undetected. Therefore, the callback trace information serves as a pivotal asset in uncovering unknown bugs.

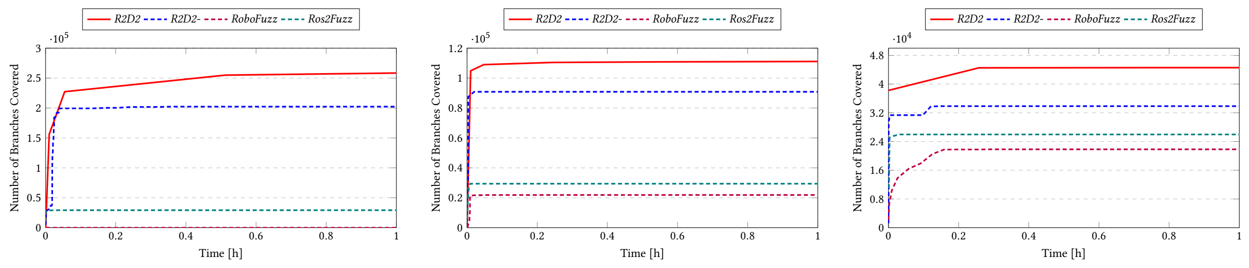
Bug Severity. Bugs in ROS tend to cause severe consequences. Specifically, the bugs detected by *R2D2* can lead to potential data loss, system hang, and system crash, causing ROS to run into erroneous states. Of the new bugs found, 7 bugs can result in data loss, including Bug #1, Bug #3, and Bugs #35 to #39, where OOM, UAF, and memory leak bugs in the ROS runtime can result in critical callback information being lost. Another 17 bugs can cause the system to hang or incur an unreasonable delay, including Bug #2, Bugs #4 to #9, and Bugs #25 to #34, where the overflow in the DDS can crash the service, whereas the data race and deadlock bugs lead to the service hanging, which consequently fails to deliver the message properly. 15 bugs can cause the system to crash, where Bugs #10 to #24 were located in ROS application; these memory-related issues can cause relevant components to fail, thereby affecting overall system functionality and crashing the entire system.

Case Study. We use bug#4 to briefly describe a previously unknown bug found by *R2D2* as the case study to demonstrate the bug discovery capability of our method. Figure 8 shows a data race bug in the function `gc_delete_writer()` and the `entity_guid_eq()` of the ROS runtime, where the developer claims to be a thread-safety code.

```
1  // Thread 1: in q_entity.c
2  static void gc_delete_writer (struct gcreq *gcreq) {
3      struct writer *wr = gcreq->arg;
4      ...
5      ddsi_sertype_unref ((struct ddsi_sertype *) wr->type);
6      endpoint_common_fini (&wr->e, &wr->c);
7      // write operation
8      ddsrt_free (wr);
9  }
10 // Thread 2: in ddsi_entity_index.c
11 static int entity_guid_eq (const struct entity_common *a, const
         struct entity_common *b) {
12     // read operation
13     return a->guid.prefix.u[0] == b->guid.prefix.u[0] &&
14            a->guid.prefix.u[1] == b->guid.prefix.u[1] &&
15            a->guid.prefix.u[2] == b->guid.prefix.u[2] &&
16            a->guid.entityid.u == b->guid.entityid.u;
17 }
```

Figure 8: A previously unknown data-race in ROS runtime.

The figure shows that the function `gc_delete_writer()` in `q_entity.c` is responsible for deallocating the writer object `wr` (line 5). Concurrently, another function `entity_guid_eq()` in `ddsi_entity_index.c` is reading from the same object `wr` (line 13 to line 16). Both functions are executed by different threads, and there is no synchronization mechanism to prevent concurrent access to the shared object `wr`. This leads to a data race condition where the write operation in `gc_delete_writer()` could invalidate the read operation in `entity_guid_eq()`, resulting in unexpected behavior. The unexpected behavior could manifest as incorrect program states or system hang, thereby compromising the reliability and security of the ROS system. To detect this bug, *R2D2* profiles the callback trace for state identification. It identifies the publisher callback has an abnormal latency delay, by giving the input that triggers the delay more execution and mutation priority, and with the help of TSAN, *R2D2* triggers this bug. Despite the maturity of the ROS2 codebase and its extensive set of unit and system-level



(a) Navigator2 Comparison (b) TurtleBot3 Comparison (c) Turtlesim Comparison

Figure 9: Coverage Growth Curve Between *R2D2*, *R2D2-*, *RoboFuzz*, and *Ros2Fuzz*.

tests, traditional testing methods have failed to uncover this subtle yet critical issue. With the proposed callback trace guided fuzzing approach, our tool can generate test cases that explore new state space, including those that lead to data race conditions.

5.3 Effectiveness of Callback Trace Guidance

To address the **RQ2**, and identify the effectiveness of the callback trace guidance, we first compare the code coverage achieved by *R2D2* against the *Ros2Fuzz* and *RoboFuzz*. Also, we implemented *R2D2-*, which is *R2D2* without the callback trace guidance (since *R2D2* has no coverage guidance, *R2D2-* does not have any guidance mechanism) further to validate the effectiveness of the callback trace guidance. The detailed code coverage statistics are presented in Table 2, we observed that due to ROS having limited coverage variety, the coverage tends to reach saturation within the first hour. As ROS is designed to complete certain fixed tasks, thereby following a rather fixed control flow, and most of its code will easily be covered at the initial phase, making the coverage saturate so quickly. This phenomenon is also discussed in previous literature, like Robofuzz. Therefore, we present the coverage growth curve for the first hours, as shown in Figure 9.

Table 2: Coverage Comparison Between the *R2D2*, *Ros2Fuzz*, *RoboFuzz*, and *R2D2-*.

Fuzzers	Navigator2	TurtleBot3	Turtlesim	Average
R2D2	**259111.2**	**111102.8**	**44576.2**	**138263.4**
R2D2-	202274.4(+0.28×)	90843.4(+0.22×)	33846.4(+0.32×)	108988.1(+0.27×)
RoboFuzz	-	21867.6(+4.08×)	21827.0(+1.04×)	21847.3(+2.56×)
Ros2Fuzz	29199.4(+7.87×)	29394.8(+2.78×)	25965.2(+0.72×)	28186.5(+3.91×)

5.3.1 *Coverage Comparison with Ros2Fuzz.*

As can be inferred from the Table 2, *R2D2* achieves 259111.2, 111102.8, and 44576.2 code coverage on Navigator2, TurtleBot3, and Turtlesim in respect, with an average of 138263.4 in total. On the other hand, among the above three target applications, *Ros2Fuzz* achieves 29199.4, 29394.8, and 25965.2 code coverage, with 28186.5 on average. Compared with *Ros2Fuzz*, *R2D2* achieves 7.87×, 2.78×, and 0.72× more code coverage, with an average improvement of 3.91×. The observed improvements in code coverage attest to *R2D2*'s proficiency in generating test cases that accurately adhere to the varied input interface structures and are different from *Ros2Fuzz*, which can only test a specific interface at a time, *R2D2* can comprehensively test all interfaces that the SUT consists of. Moreover, the guidance provided by callback trace information further expands coverage, as it enables the exploration of distinct system states that lead to the discovery of diverse code paths.

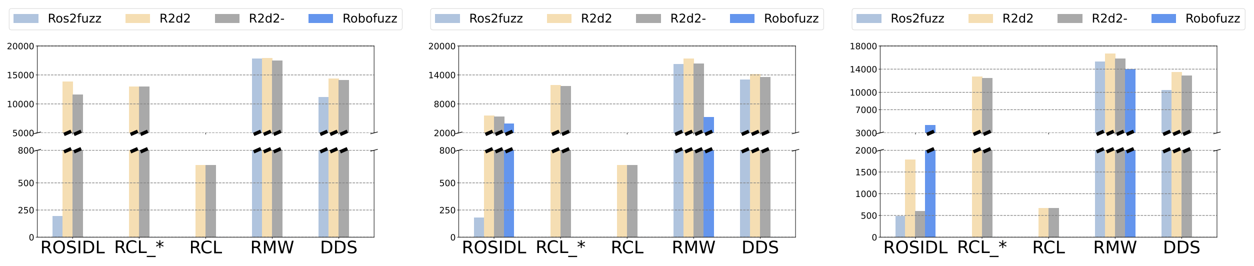
5.3.2 *Coverage Comparison with RoboFuzz.*

We then assessed the effectiveness of *R2D2* in comparison to *RoboFuzz*. As *RoboFuzz* is not adapted to Navigator2, our comparison focused solely on TurtleBot3 and Turtlesim. *RoboFuzz* achieves an average code coverage of 21867.6 and 21827.0 for TurtleBot3 and Turtlesim, respectively. In Comparison, *R2D2* gains a coverage improvement of 4.08× and 1.04× in respect. This enhanced performance can be attributed to the callback trace guidance providing the fuzzer with more detailed system states, compared to the *RoboFuzz* which uses certain application's oracles as system states. This allows *R2D2* to detect state transitions within the target system, thereby contributing to higher code coverage. Also, similar to *Ros2Fuzz*, *RoboFuzz* relies on predefined interface specifications to test ROS, whereas the automatic interface extraction allows *R2D2* to test a broader range of interfaces, thereby covering more code. For the coverage growth curve, as indicated in the figure and similar to that of *Ros2Fuzz*, *RoboFuzz* stopped growing at a very early stage during the testing, reflecting the relatively stable control flow inherent to ROS.

5.3.3 *Comparison with R2D2-.*

To further investigate the effectiveness of the callback trace guidance mechanism, we implemented *R2D2-*, an unguided version of *R2D2* that removes the callback trace guidance. We conducted evaluations on both *R2D2* and *R2D2-* across both Navigator2, TurtleBot3, and Turtlesim, in terms of bug detection abilities and code coverage.

In concrete, for all the bugs listed in Table 1, without the assistance of the callback trace guidance, *R2D2-* detects 5 bugs (# 1, 3, 11, 15, 16). We find that most bugs that *R2D2-* found were located in relatively shallow code paths, thus showing that without the help of callback trace guidance, it is difficult to test code deeper into the ROS code logic, not only in the ROS runtime but also in ROS application. We further compared the code coverage statistics between *R2D2* and *R2D2-*. As demonstrated in Table 2, *R2D2-* covers 202274.4, 90843.4, and 33846.4 branches, an improvement of 0.27× compared to *R2D2* on average. This improvement is purely attributed to the callback trace guidance mechanism, which better perceives the state transitions within the system, thereby exploring



(a) Coverage on Navigator2 (b) Coverage on TurtleBot3 (c) Coverage on Turtlesim

Figure 10: Module Coverage Comparison on Navigator2, TurtleBot3, and Turtlesim

code segments and program states that are usually hard to trigger. Furthermore, we find that, *R2D2-* still outperforms *Ros2Fuzz* and *RoboFuzz*, as this is attributed to *R2D2* acquiring all the interface that the target ROS application possess, and generating inputs that strictly follows the input specifications, further indicating the effectiveness of *R2D2*.

5.3.4 *Component-wise Coverage Comparison*.

To further evaluate the coverage compositions and analyze the effectiveness of the *R2D2*, we conducted a component-wise coverage comparison, focusing on the coverage exclusively of the ROS runtime. Concretely, we focus on the coverage comparison within different layers of the ROS runtime, including the interface definition (ROSIDL), different language implementation of client libraries (RCLPY, RCLCPP), common client libraries (RCL), middleware (RMW), and DDS implementations. We categorize different language implementations as RCL_*. The overall coverage statistic can be found in Figure 10.

As we can see from the figure, *R2D2*, and *R2D2-* successfully covered all essential components within ROS, which is attributed to the fact that *R2D2* can extract all interfaces that the target applications own. Also, we find that *R2D2* can achieve better component-wise coverage compared to *R2D2-*, which is attributed to the callback trace guidance mechanism, as it can guide the fuzzing process deeper into ROS's code logic, thereby increasing *R2D2*'s code exploration ability. For *RoboFuzz*, it covered a limited part of the ROS code, mainly on the ROSIDL and RMW level, as shown in Figure 10(c) and Figure 10(b), because it is designed to uncover logic bugs within the application level, with a focus on certain application specifications. Also, it uses predefined interfaces, so achieving limited code exploration ability in ROS runtime. However, we noticed that for that in Figure 10(c), for ROSIDL, *RoboFuzz* outperforms that of *R2D2*, that is due to the fact that the *RoboFuzz* is specifically tailored towards Turtlesim specific behavioral characteristics, including predefined interaction information, whereas *R2D2* is a generalized tool that does not contain such information. For *Ros2Fuzz*, we find that in most cases, it can cover different components under different applications. This depends on the quality and functionalities of the generated drivers, some drivers are rather simple without too many interactions, like Figure 10(c), while some drivers are complex, with certain communication like Figure 10(a) and Figure 10(b). Furthermore, we can reflect that, for some critical components, like the RMW, and DDS, *R2D2* can outperform the *R2D2-* and *Ros2Fuzz*, which first indicates that ROS follows a rather limited control flow, and second, the callback guidance mechanism can help *R2D2* to dive deep into the ROS's state, and cover more code, even for such code sections with limited control flow variability.

5.4 Instrumentation Overhead

To address **RQ3** and assess the impact of instrumentation on the overall system's performance, we utilized the `performance_test` framework, which is a commonly employed tool for evaluating ROS system performance. Our evaluation specifically targeted the runtime memory usage and the execution latency. To provide a clear comparison, we examined these metrics across three distinct settings: the instrumentation approach of *R2D2*, that of *Ros2Trace*, and the standard vanilla ROS system.

Table 3: Overhead Statistics Between *R2D2*, *Ros2Trace*, and Vanilla ROS2.

	R2D2	***Ros2Trace***	**Vanilla ROS2**	**Overhead**
Memory Usage (MB)	47276.0	47136.0	46480.0	0.3%/1.7%
Latency (MS)	0.16226	0.15512	0.13962	4.6%/16.2%

The detailed statistics are shown in Table 3. First, for the memory overhead, the memory usage for *R2D2* is 47276.0 MB, compared to 47136.0 MB for *Ros2Trace* and 46480.0 MB for the vanilla ROS system. This results in an average overhead of 0.3% relative to *Ros2Trace* and 1.7% relative to the vanilla ROS system, given the complexity of the tasks being performed, this represents a relatively modest memory overhead. Also, it's worth noting that the instrumentation strategy of *R2D2* is adapted from *Ros2Trace*, which explains the relatively closed memory overhead. Second, we turned our attention to execution latency. The latency for *R2D2* is 0.16226 ms, while *Ros2Trace* and the vanilla ROS system registered latencies of 0.15512 ms and 0.13962 ms, respectively. This translates to an overhead of 4.6% compared to *Ros2Trace* and 16.2% compared to the vanilla ROS system. A significant factor to consider here is that *R2D2* utilizes shared memory to write data in real-time, which incurs a latency cost. However, given the context of testing and the benefits of real-time data writing, this increase in latency is justifiable and remains within acceptable bounds for the testing scenario at hand.

In conclusion, the instrumentation methodology employed by *R2D2* results in a relatively moderate and acceptable elevation in both memory consumption and execution latency. The comparable memory usage observed with *Ros2Trace* can be attributed to the shared-memory within the instrumentation techniques. On the other hand, the marginally increased latency is a conscious trade-off made to facilitate real-time data recording capabilities. Given these observations, it's evident that the callback trace guidance mechanism reaches a balance between performance and precision, making it a suitable choice for testing ROS systems without introducing substantial overhead.

6 DISCUSSION

Benchmark Value. Currently, *R2D2* determines benchmark values for new state identification based on runtime statistics. While this method allows us to adapt and change the value in various testing scenarios, it has certain limitations. First, the dynamic nature of ROS systems can lead to variability that might not be captured within a fixed sample size. Additionally, some statistical outliers can skew benchmarks, potentially causing misidentifications. These can neither result in a deficiency in state discovery nor an overwhelming number of states being identified. Although it may slow down the testing performance, it would not affect the bug detection abilities. Future enhancements to *R2D2* can use domain-specific knowledge like documented specifications as benchmark values to provide a more accurate and comprehensive analysis of ROS system states.

Timing Bug Detection. *R2D2* is capable of detecting memory- and thread-related bugs. However, as a system with certain real-time requirements. The timing of its operations is a critical factor that directly impacts ROS's reliability, and timing bugs can lead to a range of issues, from system performance degradation to complete system crashes. Currently, *R2D2* cannot identify bugs related to these real-time constraints. In the future, *R2D2* can incorporate timing analysis. This allows *R2D2* to identify timing discrepancies that could lead to real-time violations.

7 CONCLUSION

In this paper, we introduce *R2D2*, a callback trace-guided fuzzer that is crafted for ROS systems testing. Traditional methods, such as code coverage guided fuzzing, often fall short in capturing the intricate state transitions of ROS systems. In contrast, *R2D2* leverages the callback trace to provide more insight into the system's internal behaviors and states. Through systematic instrumentation in the ROS runtime, *R2D2* dynamically monitors and profiles the system's state transitions, guiding the generation of high-quality test cases. We have adapted *R2D2* to four widely used ROS applications, the evaluations demonstrate *R2D2*'s effectiveness, where *R2D2* found a total number of *39* of bugs, with *6* bugs been fixed. Compared to existing fuzzers, *R2D2* improves code coverage by 2.56× and 3.91×, respectively, while introducing a 10.4% and 1.0% on execution latency and memory consumption.

8 ACKNOWLEDGMENTS

This research is sponsored in part by the National Key Research and Development Project (No. 2022YFB3104000, No. 2022YFB3104003), NSFC Program (No. 92167101, 62021002, 62202500), and Hunan Provincial Natural Science Foundation (No: 2023JJ40772).

REFERENCES

[1] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. 2019. FUDGE: Fuzz Driver Generation at Scale. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 975–985. https://doi.org/10.1145/3338906.3340456

[2] Christophe Bédard, Ingo Lütkebohle, and Michel Dagenais. 2022. ros2_tracing: Multipurpose low-overhead framework for real-time tracing of ROS 2. *IEEE Robotics and Automation Letters* 7, 3 (2022), 6511–6518.

[3] 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

[4] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)*. 711–725. https://doi.org/10.1109/SP.2018.00046

[5] Christophe, Ingo Lütkebohle, and Michel Dagenais. 2022. ros2_tracing: Multipurpose Low-Overhead Framework for Real-Time Tracing of ROS 2. *IEEE Robotics and Automation Letters* 7, 3 (2022), 6511–6518. https://doi.org/10.1109/LRA.2022.3174346

[6] CISA. 2021. Multiple Data Distribution Service (DDS) Implementations. https://www.cisa.gov/news-events/ics-advisories/icsa-21-315-02.

[7] Yinlin Deng, Chenyuan Yang, Anjiang Wei, and Lingming Zhang. 2022. Fuzzing Deep-Learning Libraries via Automated Relational API Inference. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 44–56. https://doi.org/10.1145/3540250.3549085

[8] Mathieu Desnoyers and Michel R Dagenais. 2006. The lttng tracer: A low impact performance and behavior monitor for gnu/linux. In *OLS (Ottawa Linux Symposium)*, Vol. 2006. Citeseer, 209–224.

[9] William Woodall Dirk Thomas. 2020. turtlesim. http://wiki.ros.org/turtlesim

[10] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies*. 10–10.

[11] Emre Güler, Philipp Görz, Elia Geretto, Andrea Jemmett, Sebastian Österlund, Herbert Bos, Cristiano Giuffrida, and Thorsten Holz. 2020. Cupid: Automatic Fuzzer Selection for Collaborative Fuzzing. In *Annual Computer Security Applications Conference* (Austin, USA) *(ACSAC '20)*. Association for Computing Machinery, New York, NY, USA, 360–372. https://doi.org/10.1145/3427228.3427266

[12] Xu Jiang, Dong Ji, Nan Guan, Ruoxiang Li, Yue Tang, and Yi Wang. 2022. Real-time scheduling and analysis of processing chains on multi-threaded executor in ros 2. In *2022 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 27–39.

[13] JnxF and gavanderhoorn. 2021. ros2_fuzz. https://github.com/rosin-project/ros2_fuzz. Commit a01394f on Jul 19, 2021.

[14] Shinpei Kato. 2017. "Autoware. https://github.com/autowarefoundation/autoware

[15] Seulbae Kim and Taesoo Kim. 2022. RoboFuzz: Fuzzing Robotic Systems over Robot Operating System (ROS) for Finding Correctness Bugs. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 447–458. https://doi.org/10.1145/3540250.3549164

[16] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. 2020. Finding Bugs in File Systems with an Extensible Fuzzing Framework. *ACM Trans. Storage* 16, 2, Article 10 (may 2020), 35 pages. https://doi.org/10.1145/3391202

[17] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. 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, 2123–2138. https://doi.org/10.1145/3243734.3243804

[18] Matti Kortelainen. 2023. A short guide to ROS 2 Humble Hawksbill. (2023).

[19] Takahisa Kuboichi, Atsushi Hasegawa, Bo Peng, Keita Miura, Kenji Funaoka, Shinpei Kato, and Takuya Azumi. 2022. CARET: Chain-Aware ROS 2 Evaluation Tool. In *2022 IEEE 20th International Conference on Embedded and Ubiquitous Computing (EUC)*. 1–8. https://doi.org/10.1109/EUC57774.2022.00010

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

[21] Ruoxiang Li, Xu Jiang, Zheng Dong, Jen-Ming Wu, Chun Jason Xue, and Nan Guan. 2023. Worst-Case Latency Analysis of Message Synchronization in ROS. In *2023 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 185–197.

[22] 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

[23] 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

[24] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *Proceedings of the 28th USENIX Conference on Security Symposium* (Santa Clara, CA, USA) *(SEC'19)*. USENIX Association, USA, 1949–1966.

[25] Steven Macenski and Tully Foote. 2023. ros2 rolling. https://docs.ros.org/en/rolling/Installation.html

[26] Steve Macenski, Francisco Martín, Ruffin White, and Jonatan G. Clavero. 2020. The Marathon 2: A Navigation System. *ArXiv* (2020). https://doi.org/10.1109/IROS45743.2020.9341207 Accessed September 25, 2023.

[27] 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

[28] Gaoning Pan, Xingwei Lin, Xuhong Zhang, Yongkang Jia, Shouling Ji, Chunming Wu, Xinlei Ying, Jiashui Wang, and Yanjun Wu. 2021. V-Shuttle: Scalable and Semantics-Aware Hypervisor Virtual Device Fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 2197–2213. https://doi.org/10.1145/3460120.3484811

[29] Ivan Santiago Paunovic. 2020. ROS 2 Issue #1035. https://github.com/ros2/ros2/issues/1035. Accessed: Sep 21, 2020.

[30] ROBOTIS-GIT. 2023. ROS packages for Turtlebot3. https://github.com/ROBOTIS-GIT/turtlebot3 Accessed: 2023-09-25.

[31] Yuheng Shen, Shijun Chen, Jianzhong Liu, Yiru Xu, Qiang Zhang, Runzhe Wang, Heyuan Shi, and Yu Jiang. 2023. Brief Industry Paper: Directed Kernel Fuzz Testing on Real-time Linux. In *2023 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 495–499.

[32] 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. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 11 (2022), 4563–4574. https://doi.org/10.1109/TCAD.2022.3198910

[33] Emmet Snider. 2020. ApexAI / performance_test. GitLab. https://gitlab.com/ApexAI/performance_test

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

[35] 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

[36] 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

[37] Qinglong Wang, Runzhe Wang, Yuxi Hu, Xiaohai Shi, Zheng Liu, Tao Ma, Houbing Song, and Heyuan Shi. 2023. KeenTune: Automated Tuning Tool for Cloud Application Performance Testing and Optimization. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1487–1490.

[38] Trey Woodlief, Sebastian Elbaum, and Kevin Sullivan. 2021. Fuzzing Mobile Robot Environments for Fast Automated Crash Detection. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*. 5417–5423. https://doi.org/10.1109/ICRA48506.2021.9561627

[39] Kai-Tao Xie, Jia-Ju Bai, Yong-Hao Zou, and Yu-Ping Wang. 2022. ROZZ: Property-based Fuzzing for Robotic Programs in ROS. 6786–6792. https://doi.org/10.1109/ICRA46639.2022.9811701

[40] Y. Xu, H. Sun, J. Liu, Y. Shen, and Y. Jiang. 2024. SATURN: Host-Gadget Synergistic USB Driver Fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 51–51. https://doi.org/10.1109/SP54263.2024.00051

[41] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 745–761. https://www.usenix.org/conference/usenixsecurity18/presentation/yun

[42] Chijin Zhou, Quan Zhang, Mingzhe Wang, Lihua Guo, Jie Liang, Zhe Liu, Mathias Payer, and Yu Jiang. 2022. Minerva: Browser API Fuzzing with Dynamic Mod-Ref Analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1135–1147. https://doi.org/10.1145/3540250.3549107

Received 16-DEC-2023; accepted 2024-03-02