RoboFuzz: Fuzzing Robotic Systems over Robot Operating System (ROS) for Finding Correctness Bugs

Seulbae Kim
Georgia Institute of Technology
Atlanta, Georgia, USA
seulbae@gatech.edu

Taesoo Kim
Georgia Institute of Technology
Atlanta, Georgia, USA
taesoo@gatech.edu

ABSTRACT

Robotic systems are becoming an integral part of human lives. Responding to the increased demands for robot productions, Robot Operating System (ROS), an open-source middleware suite for robotic development, is gaining traction by providing practical tools and libraries for quickly developing robots. In this paper, we are concerned with a relatively less-tested class of bugs in ROS and ROS-based robotic systems, called semantic correctness bugs, including the violation of specification, violation of physical laws, and cyber-physical discrepancy. These bugs often stem from the cyber-physical nature of robotic systems, in which noisy hardware components are intertwined with software components, and thus cannot be detected by existing fuzzing approaches that mostly focus on finding memory-safety bugs.

We propose RoboFuzz, a feedback-driven fuzzing framework that integrates with ROS and enables testing of the correctness bugs. RoboFuzz features (1) data type-aware mutation for effectively stressing data-driven ROS systems, (2) hybrid execution for acquiring robotic states from both real-world and a simulator, capturing unforeseen cyber-physical discrepancies, (3) an oracle handler that identifies correctness bugs by checking the execution states against predefined correctness oracles, and (4) a semantic feedback engine for providing augmented guidance to the input mutator, complementing classic code coverage-based feedback, which is less effective for distributed, data-driven robots. By encoding the correctness invariants of ROS and four ROS-compatible robotic systems into specialized oracles, RoboFuzz detected 30 previously unknown bugs, of which 25 are acknowledged and six have been fixed.

CCS CONCEPTS

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

KEYWORDS

Robot Operating System 2 (ROS 2); Correctness bugs; Semantic feedback-driven fuzzing

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9413-0/22/11.
https://doi.org/10.1145/3540250.3549164

ACM Reference Format:
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 (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3540250.3549164

1 INTRODUCTION

Human lives and robots are inextricably linked. Traditionally, robots have been used extensively in agriculture and manufacturing for task automation. Recent developments have brought robots even closer to everyone's daily lives. For example, Amazon and Google are deploying humanless delivery systems using unmanned aerial vehicles [5, 17], the market size of robot vacuum cleaners has grown 23% annually [18], and the use of robots in surgical procedures increased from 2% in 2012 to 15% in 2018 [48], showing how rapidly the robot industry is growing to accommodate human needs. Simultaneously, modern robots, such as autonomous driving vehicles, are becoming more sophisticated, demanding an integration of complex subsystems, *e.g.*, sensing, perception, planning, and actuation.

Responding to such higher demands for developing intricate robotic systems, Robot Operating System (ROS) [27, 33] is gaining popularity. ROS is an open-source middleware suite for robot development that features a message-passing scheme for distributed robot processes, hardware abstraction, a broad collection of development tools (*e.g.*, simulator), and robotic libraries (*e.g.*, path planning algorithm). Using ROS, developers can expedite the development process without having to re-invent the wheel, and instead focus on the core functionality of their robots. With its philosophy of multi-language and multi-platform support, ROS is proliferating as a de facto standard in robotics programming; it has been widely adopted in industry [15, 40], the military [35, 41], research organizations, and by individuals and is estimated to power 55% of the total commercial robots shipped in 2024 [7].

Although having such a universal middleware is generally advantageous, a downside exists; a flaw in ROS itself or in the way developers commonly use ROS in their applications can affect a wide range of robotic systems that depend on ROS and wreak havoc on the safety and security of many users. For example, ROS version 1 had no notion of authentication, allowing any entity on the network to fully access any robotic system, eavesdrop on internal messages, and even hijack the execution if the IP address and port are known. Indeed, over 100 ROS systems deployed in 28 countries were discovered by scanning the default ROS port on the internet for two months [12], being completely vulnerable to such attacks. The latest version of ROS (*i.e.*, ROS 2) is designed and implemented in consideration of such problems and has invited greater public

scrutiny of security. Unfortunately, existing work and solutions focus either on the network security aspects regarding authentication and authorization methods [9, 14, 28, 42], or on regression testing [43], leaving the ROS community deficient in a systematic testing method for finding unknown bugs that affect the robustness and correctness of robotic systems.

Given this situation, fuzzing may seem to be a viable solution, as it is well known for its capability of discovering unknown errors in a variety of systems [1, 2, 8, 10, 45, 51]. However, robotic systems are cyber-physical. They have both physical components (hardware) interacting with the real world, and cyber components (software) conducting computations and making decisions. This unique context introduces a new class of bugs closely related to semantic correctness (§2.3), namely, violation of physical laws, violation of specifications, and cyber-physical discrepancy, which do not exist in traditional software and thus cannot be detected by existing fuzzing approaches.

As a remedy, we propose RoboFuzz, the first feedback-driven fuzzing framework for systematically testing all layers of ROS, including the internal layers comprising the ROS core, as well as the robotic applications built using ROS APIs. Unlike classic fuzzing approaches [1, 51], which target software binaries looking primarily for memory-safety errors, testing ROS requires an understanding of its distributed architecture, underlying components, inter-process communication scheme, and robotic properties. Based on the knowledge we accumlated by studying ROS and ROS-powered systems, we design a flexible fuzzing framework for ROS that features a customizable input mutator, hybrid executor, oracle handler, and feedback engine. According to the target layer in ROS to test, one can select a suitable test strategy by customizing the mutation mode, states to watch, properties to check, and semantic feedback to utilize. Given this strategy, RoboFuzz effectively explores the state space of the system under test (SUT) by mutating and publishing messages to two copies of SUT, one operating in the cyber world (*i.e.*, a simulator) and the other running physically in the real world. States of the SUTs from both worlds are then collected to find errors and generate semantic feedback to assist further input generation.

To demonstrate the effectiveness of RoboFuzz, we tested the ROS core, as well as four ROS-based robotic applications, namely, PX4 quadcopter, TurtleBot3, Move It 2, and Turtlesim, by applying customized correctness oracles. By fuzzing these systems, RoboFuzz revealed 30 bugs, of which 25 are acknowledged and six are fixed. Among the bugs, 13 bugs are high-impact bugs that exist in the internal layers of ROS, *i.e.*, the type system and the client API implementation, affecting a number of ROS-based systems.

In summary, this paper makes the following contributions:

- We introduce and study three types of semantic correctness bugs that are specific to cyber-physical systems.
- We propose and implement RoboFuzz, a fuzzing framework that seamlessly supports testing robotic systems over ROS to detect correctness bugs. The prototype of RoboFuzz is open-sourced at https://github.com/sslab-gatech/robofuzz.
- By designing robot-specific correctness oracles and semantic feedback metrics, we test the latest ROS distribution and four robotic systems, and detect 30 new correctness bugs.

2 BACKGROUND

2.1 Challenges of Testing a Robotic System

A typical robotic system can be characterized by the **physical** (hardware) component, *e.g.*, a GPS sensor or a motor, the **cyber** (software) component, *e.g.*, an implementation of a control algorithm, and the **environment**, *e.g.*, a terrain, that a robot interacts with. This unique configuration poses three challenges in testing robotic systems.

Challenge 1: Heterogeneity. The diversity of robotic systems is enormous; surgical robots require extra precision, factory robots are built to last, and aerial robots fly. Moreover, robots sharing the same software may behave differently if they have different hardware (*e.g.*, sensors from different manufacturers), and the same robots may behave differently under different environments. Such heterogeneity makes testing robotic systems challenging because one testing methodology may not work directly with other robots. As a workaround, we focus on an integral property of a robot that it essentially is a group of distributed processes that exchange data for operation. In other words, *a robotic behavior can be represented by the data flow* (whether it is analog or digital) between the nodes. Considering this property, we design our framework to be capable of capturing and controlling the data flow for testing.

Challenge 2: Huge State Space. Robotic systems operate in many different environments and conditions, with countless variables that exist in the real world. As a result, the state space tends to be bigger than that of traditional software programs. To effectively test such a profound system, we adopt feedback-driven fuzzing, which is proven to be effective in exploring large state spaces, leading to the successful detection of software bugs [1, 2, 8, 23, 51]. Particularly, we propose the use of *semantic feedback*, which is unique to robotic systems, to further accelerate the state space exploration by complementing the coverage-based feedback.

Challenge 3: Noisy Hardware. Unlike conventional software that has neatly defined interfaces to exchange accurate data, the hardware components (*e.g.*, sensors and actuators) of a robot are inherently noisy, as they interact with the real world. For example, the GPS sensor constantly reports slightly changing latitude and longitude values even when a robot is not moving at all. Similarly, it is almost impossible to move a robot to the exact same location with a precision of millimeters, as the actuators are noisy.

To incorporate such noise into testing, we design a hybrid executor, which runs tests simultaneously in a simulator and in the real world, so that the states captured from both worlds, (*e.g.*, the discrepancy between the values reported by the noisy physical sensor and noiseless simulated sensor) can be utilized to assist testing.

2.2 Robot Operating System (ROS)

Robot Operating System (ROS[1]) [33] is an open-source framework designed to help developers build modular, distributed robotic applications. With convenient all-in-one tools and libraries that support multiple operating systems and programming languages, combined with a vibrant user community, ROS established a solid ecosystem and attracted many users worldwide [3, 4] to become a de facto standard in robotics.

[1]Unless otherwise noted, "ROS" refers to ROS 2 herein, which is the latest version.

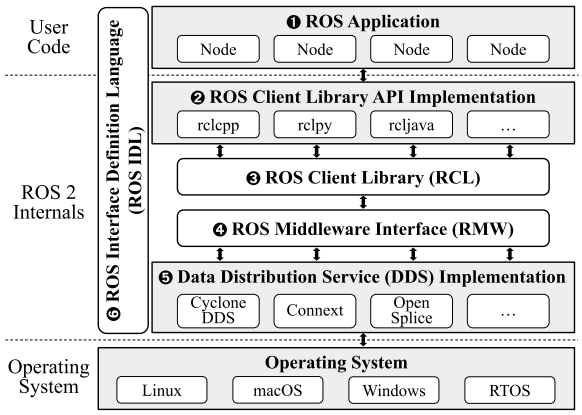

Figure 1: System architecture of ROS 2.

```
uint32 height    # image height
uint32 width     # image width
string encoding  # encoding of pixels
uint8[] data     # actual matrix data of pixels
```

Figure 2: An example message definition for image data.

As shown in Figure 1, ROS separates the application layer (❶) from the internal layers (❷-❻), so users can focus solely on designing the logical composition of the robotic application, while the internal ROS layers manage the data distribution. Specifically, ROS-powered applications are mainly built around three logical concepts: *node*, *topic*, and *message*.

- Node: A node is a minimal self-contained process with a designated function, such as performing computations. Nodes are the foundations of modular robotic systems that assist fault isolation and high reusability.
- Topic: A topic is a message bus connecting publishers (data writer nodes) to subscribers (data reader nodes). Each topic may have multiple publishers and multiple subscribers, enabling multiple nodes to concurrently exchange data anonymously.
- Message: A message is a typed key-value data structure that serves as a unit of data transportation. ROS defines 15 built-in types: bool, byte, char, float{32,64}, int{8,16,32,64}, uint{8,16,32,64}, string and wstring, and array types of the above-mentioned built-in types. As shown in Figure 2, a message may contain multiple fields of different types. ROS topics are strictly typed, meaning that a topic only accepts messages of one type that is pre-registered when initializing the topic.

Application Layer (❶). ROS allows developers to build a ROS-based application by utilizing ROS Client Library APIs (❷) to compose a logical network of nodes connected by topics in desired programming languages (officially, C++ and Python). An illustrative example of a ROS application is shown in Figure 3. Once the developer specifies nodes and connects them by binding publishers and subscribers to topics, ROS orchestrates the data distribution through the publish-subscribe pattern, so that if one node publishes a message to a topic, the topic broadcasts the message to all the nodes subscribing to that topic. Meanwhile, the underlying implementation backing the application by handling the process management and data distribution is completely abstracted away from the user-space, as shown in Figure 1.

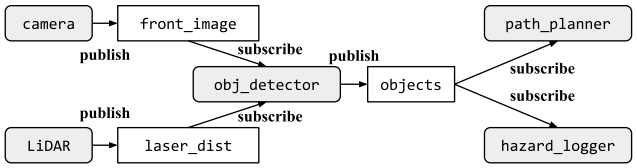

Figure 3: An example use of the publish-subscribe pattern in a ROS application. Nodes (gray rounded rectangles) publish data to other nodes through topics (white rectangles). A node may subscribe to multiple topics (*e.g.*, `obj_detector` gets data from `front_image` and `laser_dist`), and multiple nodes can subscribe to a topic (*e.g.*, both `path_planner` and `hazard_logger` receive data published to `objects`).

ROS Client Library API Implementation (❷). RCL API implementations are the only user-facing APIs that expose ROS functionality to users in specific languages (*e.g.*, `rclcpp` and `rclpy`). Developers can be agnostic of the inner workings of ROS and simply utilize the RCL APIs to compose a ROS network consisting of nodes and topics, which essentially is a modular robotic system.

ROS Client Library (RCL) (❸). RCL is a common unified layer that bridges the user-space APIs implemented in multiple languages (❷) with lower-level functionalities. Core concepts of ROS (*e.g.*, node, topic) are implemented in this layer so that the behaviors of the language-specific APIs can remain constant.

ROS Middleware Interface (RMW) (❹). RMW is an abstraction layer built to support diverse standalone DDS implementations. It wraps the APIs of individual DDS implementations in a uniform interface, simplifying the interaction with the RCL layer.

Data Distribution Service (DDS) (❺). A DDS [31] is a network middleware that implements the publish-subscribe pattern communication for real-time systems. Although DDS is not specific to ROS, it became an essential part of ROS since it includes a number of useful functionalities for delivering arbitrary data, such as marshalling and de-marshalling, Quality of Service (QoS) policy implementation, dynamic node discovery, and message delivery on the transport layer.

Type System Based on Interface Definition Language (ROSIDL) (❻). The type system of ROS utilizes an Interface Definition Language (IDL) to define messages (*e.g.*, Figure 2), to automatically generate corresponding code for the built-in ROS types in each RCL API language (*i.e.*, C++ and Python), and to enforce type checking at the time of value assignment and delivery. It also maps ROS types to the type system used by DDS, making ROS types universally applicable to all ROS layers.

2.3 Correctness Bugs in Robotic Systems

Having the characteristics of cyber-physical systems discussed in §2.1, robotic systems can suffer from not only traditional software-oriented bugs (*e.g.*, memory safety errors), but also a new class of bugs related to the correctness in the operation. Unlike traditional bugs that lead to software crashes, correctness bugs often do not have immediate manifestations but have devastating impacts, such as quickly wearing the hardware of the already deployed robotic system by pushing the robot beyond its physical capability. In particular, we focus on three types of correctness bugs that can critically affect the safety and robustness of robotic systems.

Violation of Physical Laws. Robotic systems have physical parts operating in the real world and are thereby bound by various laws of physics and their derivatives that should always hold. For example, as defined by kinematic equations, if there is a change of displacement, the robot must have a non-zero velocity, and if there is a change of velocity, the acceleration must be non-zero. If the robotic states (either externally observed or broadcasted by the robot itself) violate any of the known physical laws, it implies component failure or logical error in the software.

Violation of Specification. Specifications or datasheets of a robot contain essential information about the robotic system, including the physical capabilities of the components (*e.g.*, sensor fidelity or joint limits), as well as various assumptions about robotic behaviors. Not complying with the specifications, the robot can be physically damaged, and other components that rely on the specification can also be broken, critically affecting the robustness of the system. Regardless of whether the specification is incorrect, or the implementation of the specification is erroneous, those cases should be detected and handled.

Cyber-physical Discrepancy. Simulations are often preferred over real-world environments when developing and testing robotic systems because of the cost and safety benefits. Here, the practical merits of utilizing simulated robots are valid only if the simulated robots are closely modeled to their physical counterparts. In this regard, if the behaviors of two identical robots in the real world and the simulation are sufficiently different, we regard it as a cyber-physical discrepancy bug.

2.4 Motivation and Scope of Work

ROS is undergoing constant development; every year, a stable version with new features, upgrades, and bug fixes is released. At the same time, an increasing number of robotic systems are being built on top of ROS distributions. ROS comes in a complicated package that includes multiple abstraction layers interacting with each other and various DDS implementations from different vendors, which makes eliminating bugs and ensuring correctness challenging. Unfortunately, ROS currently relies on minimal testing approaches (*e.g.*, unit testing [43]), which are not versatile enough to detect a wide range of unknown issues. Thus, in this paper, we are primarily concerned with quickly detecting unknown bugs and correctness issues from such an evolving codebase by building a generic fuzzing framework targeting the internal abstraction layers of ROS (❷-❻), as well as the robotic applications powered by ROS (❶).

3 ROBOFUZZ: A GENERIC FRAMEWORK

3.1 Overview

RoboFuzz is a customizable feedback-driven fuzzing framework tailored for Robot Operating System internals and applications. The key intuition behind the design of RoboFuzz is that a robotic system (ROS-based systems, in particular) is a stateful, noded system in which nodes change their states based on the data they receive. Therefore, *the behavior of robotic systems can be summarized as the data (message) flow among the distributed nodes.* RoboFuzz approaches testing from the data flow standpoint, injecting messages to nodes composing the robotic system under test.

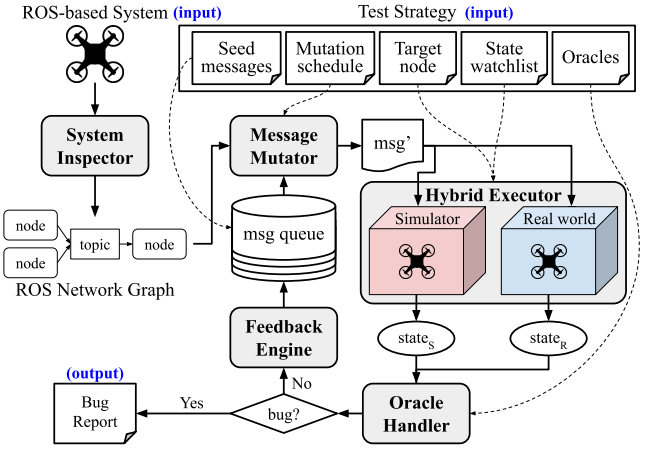

Figure 4: Overview of RoboFuzz's architecture and workflow.

Algorithm 1: Fuzzing algorithm of RoboFuzz

```
Input  : ROBOT - robotic system, STRAT = {S, sched, T, W, O} - test strategy,
         S - set of seed messages, sched - mutation schedule, T - set of target nodes,
         W - set of states to watch, O - set of correctness oracles,
         N_r - Maximum # rounds in a cycle
Output : bug - a detailed bug report, msg - message that triggered the bug
1  procedure robofuzz_main()
2      graph ← inspector.inspect(ROBOT) // §3.2
3      foreach seed ∈ S do
4          fuzz_one(seed, graph)
5  procedure fuzz_one(seed, graph)
6      msg' ← seed
7      last_score ← ∅
8      for rounds ← 1 to N_r do
9          msg' ← mutator.mutate(msg', sched, graph, T) // §3.3
10         state_S ← executor.sim_execute(ROBOT, W, msg') // §3.4
11         state_R ← executor.phy_execute(ROBOT, W, msg') // §3.4
12         if oracle_handler.check_bugs(O, state_S, state_R) /*§3.5*/ == True
             then
13             save_bug_report(msg', state_S, state_R)
14             return
15         else
16             score ← feedback.quantify_score(state_S, state_R) // §3.6
17             if score > last_score then
18                 last_score ← score
19                 S ← S ∪ msg'
```

The architecture and workflow of RoboFuzz are illustrated in Figure 4, and Algorithm 1 presents the formal algorithm. RoboFuzz takes a target system and a test strategy as input and outputs the report of found bugs. The system inspector (§3.2) analyzes the target system and extracts the ROS graph. Based on the extracted graph, the message mutator (§3.3) mutates ROS messages and publishes them to the target node(s) running in the hybrid executor (§3.4). States are collected during the execution of the target under mutated messages and checked against the provided oracles by the oracle handler (§3.5). If the oracle detects any failure, a bug report is generated, and the fuzzing loop terminates. Otherwise, the feedback engine (§3.6) quantifies the interestingness of the target's behavior from the execution states, and provides feedback to the message mutator, guiding subsequent mutations.

3.2 System Inspector

The first step of testing is to scan the robotic system to obtain the ROS network graph, which is a topological view of the robotic

system consisting of all nodes, topics, and associated message types. The system inspector launches the robotic system and examines the ROS network by invoking internal node discovery APIs of ROS. Unless the target node is specified on the test profile provided by the user, all nodes in the identified ROS network that subscribe to at least one topic automatically become the candidates of testing.

3.3 Type-aware Message Mutator

ROS-based systems are driven by ROS messages distributed along the nodes of the ROS network. Thus, a testing framework has to be capable of stressing the system by generating and transferring proper messages to relevant nodes. The message mutator is designed specifically for this task.

Type-aware Mutation. It is important to note that all topics in the ROS network are bound to *strictly typed* messages. Given that the type checker of ROS IDL works correctly, any message is rejected to be published if it does not conform with the topic's type that is determined when the topic is initialized (*i.e.*, when the robotic system is launched). In light of this property, RoboFuzz supports type-aware mutation of messages, associating appropriate mutation operators with each ROS data type to constrain the result of a mutation within the right type.

This is achieved by (1) inheriting the general mutation operators such as bit flip, byte flip, and arithmetic operators from American Fuzzy Lop (AFL) [51], which are proven to be effective in mutating arbitrary data blobs and then (2) associating each operator with ROS data type such that it is guaranteed for the application of an operator to produce the values expressible by the type and also (3) adding augmented operators that produce special values (*e.g.*, `NaN` and `INF` of float) and boundary values (*e.g.*, `uint32_max` for 32-bit unsigned integer) that are often mishandled by the robotic system. In addition, RoboFuzz allows users to register custom mutation operators (*e.g.*, generating float in a certain range) for extensibility.

Specifically, given a message, the mutator selects one of the fields, retrieves the data type associated with the selected field from the ROS network graph, and applies one of the mutation operators defined for that data type.

Message Scheduling. The mutator also manages the schedule of when the mutated messages should be published, considering the context and properties of the target system. For systems that perform a series of actions upon receiving one message, publishing a single message and watching the behavior would be appropriate (*e.g.*, a goal coordinate message to a trajectory following robot). If a system is more realtime, requiring a stream of messages, publishing a sequence of messages would better fit the purpose of the robot (*e.g.*, remote control messages to a drone). RoboFuzz systematically schedules testing campaigns by defining configurable factors such as the number of messages in a sequence, frequency of publication, and the time to watch, and letting users decide the right configuration based on their needs.

3.4 Hybrid Executor

Many developers rely heavily on robotic simulators to benefit from reduced cost and time in developing and testing robotic systems. Even though modern simulators have relatively realistic physics engines with moderately accurate dynamics and kinematics models,

(a) `PX4` drone in the real world (b) `PX4` drone in Gazebo simulator

Figure 5: Hybrid executor running `PX4` drone simultaneously in the real world and the simulator. The actual place is accurately modeled (GPS coordinate, trees, and building) to minimize unhandled discrepancy between two environments.

a certain degree of discrepancy between the simulation and the real-world execution is inevitable, either because of the lack of accuracy in the modeling or simply because the state in concern is not available in the simulator (*e.g.*, the temperature of a motor). In severe cases, the in-simulator position of a robot differs from the position of a physical robot after receiving the identical control commands due to the discrepancy. Such issue makes a Simulator-in-the-Loop (SITL) testing insufficient to test for all potential issues.

To fill the gap between the simulators with insufficient fidelity and profound physical environments, we employ a hybrid execution model in RoboFuzz. As illustrated in Figure 5, the hybrid executor runs the robot in a simulator, and simultaneously in the real world (if the physical robot is available). It then relays the mutated messages to both systems so that both robots can take corresponding actions to the identical messages. In the meantime, the executor monitors both worlds to capture the states and events. RoboFuzz benefits from this design because (1) there are complementary states that are only available in either of the worlds and (2) there are redundant states that show divergence, especially if the relayed messages are related to sensors or actuators, as discussed in §2.1.

3.5 Oracle Handler

Every robotic system has a different definition of *correct* states and, based on these, employs different invariants during the execution. Thus, a practical testing framework must allow developers to conveniently declare and apply correctness oracles that are specific to each robotic system. RoboFuzz provides an oracle template, that allows developers to define oracle states with respect to the states captured by the hybrid executor. The built-in oracle handler checks up on the robotic states with the provided oracles and generates bug reports if any violation is found. In §4, we demonstrate the use of robot-specific developer-defined oracles that we built targeting different systems to reveal correctness bugs.

3.6 Feedback Engine

In a fuzzing process, execution feedback plays an important role in providing guidance to the mutation engine. Typically, the effectiveness of the mutated input in the execution is quantified to determine whether the mutation engine should keep mutating the input or move on to another input. With the intuition that more bugs can be detected by exploring more code paths in the program, modern greybox fuzzers use code coverage as feedback and try to generate inputs that increase the code coverage. Such a strategy works well for general software programs that have diverse code paths along the execution and results in finding a number of bugs.

However, robotic systems, by nature, are *distributed systems whose behaviors are driven by state changes excited by data flow, rather than by the control flow* and thus tend to have a limited number of code paths. This property makes coverage-based feedback less effective in summarizing the execution of robotic systems, as code paths are redundant even when the robot exhibits a wide variety of behaviors, and led us to design an augmented feedback mechanism tailored for robotic systems. To complement the coverage-guided feedback mechanism and enhance the input state exploration for robotic systems, we propose *semantic feedback*, which quantifies the "bugginess" of the execution context by taking advantage of the domain knowledge of the tested robots, as detailed below.

Redundant Sensor Inconsistency Feedback. In robotics, multiple redundant sensors are utilized to improve the accuracy of the acquired data and to provide fault tolerance in the face of sensor failure [19]. The assumption behind utilizing redundant sensors is that even though each sensor provides its own measurement of the state in varying precision, the results should align with each other with high similarity because we simultaneously measure the same property. From an alternative standpoint, a discrepancy in the states measured by redundant sensors indicates potential issues in the system, such as instability. Therefore, RoboFuzz allows developers to register multi-sensor inconsistency as feedback if redundant sensors are present in the system under test. In the case when the robotic SUT has no redundant sensor, one could take advantage of the modular architecture of robotic systems and install a redundant sensor to facilitate the testing.

Control Error-based Feedback. Closed-loop feedback control [25] is a crucial part of robotic controllers; taking a control command input, a robotic system outputs the error between the reference value (*e.g.*, current position) and the setpoint (*e.g.*, desired position), which is fed back to the controller so that it can generate the next control command that can defeat the error. If a controller is erroneous or poorly calibrated, it could fail to reduce the control error, making the robot uncontrollable. Thus, developers can utilize the knowledge of the internal logic of the controller to augment the testing by factoring the control error into the execution feedback.

State Distance-based Feedback. For any numeric state that is bound by physical limits, the difference between the measured value and the limit indicates the *distance* to the erroneous state. Thus, a strategy of favoring the input messages that minimize such distance can potentially contribute to guiding RoboFuzz to trigger errors that specification-based oracles (§3.5) attempt to detect.

Cyber-physical Discrepancy Feedback. When the hybrid executor (§3.4) runs the robotic SUT, it assures that the exact same messages from the message mutator are published to both the physical robot and the simulated clone, which run the same software stack with an identical configuration. Accordingly, the resulting states of both robots are expected to be identical. However, in practice, there are unwanted factors, such as the sensor noise or inaccurate modeling of the simulator, that affect the fidelity of the execution and result in a discrepancy in the states, such as the divergence in the trajectory over time. By providing tools for computing discrepancies and registering them as feedback, RoboFuzz assists developers to take such factors into account in the testing and excite the unhandled aspects of the execution environment.

4 ROBOFUZZ IN ACTION: SPECIALIZATION

With the base framework introduced in §3, one can readily turn robot-specific knowledge and expectations into a specialized fuzzing campaign. To demonstrate how RoboFuzz can be applied to heterogeneous targets and effectively find bugs, we select four robotic applications of distinctive properties, and study them to generate appropriate oracles and semantic feedback metrics:

(1) *PX4 quadcopter* (`PX4`) [29]: an open-source flight control stack, known for its support for a variety of quadcopter models and high configurability.
(2) *TurtleBot3* (`TB3`) [6, 36]: a differential wheeled mobile robot equipped with a LiDAR sensor.
(3) *Move It 2* (`MI2`) [11, 39]: a robotic manipulation library for ROS that implements fundamental concepts of robotic manipulation, such as kinematics, motion planning, and control.
(4) *Turtlesim* (`TSM`) [30]: a lightweight velocity-controlled turtle robot in a 2D simulator, shipped with ROS for tutorials.

For ROS internals, we target (5) *RCL APIs* and the (6) *type system (ROSIDL)* to ensure the building blocks of ROS-based applications are correct. In this section, we discuss notable target-specific factors leading to different mutation strategies, oracles, and feedback.

4.1 Common Oracles

Before we introduce target-specific strategies, there are common oracles that can be universally applied to multiple robotic systems. Unless otherwise noted, these common oracles are utilized by default, along with the specialized oracles.

Physical Constraint Oracle. Robotic systems operate in the real world and are thereby bound by certain physical constraints. The oracle handler adopts a set of checks for *physical sanity*, *i.e.*, states related to the existence of the robot in the physical space can neither disappear nor have infeasible values. In particular, any state indicating the geometry of the robot, *i.e.*, (x, y, z) for the position and $(roll, pitch, yaw)$ for the orientation, should present a feasible value if the robot is in the three-dimensional physical space. For example, it is guaranteed by physics that the z-position of a robot can never be `NULL`, `NaN`, or `INF`. Likewise, kinematics-related states, such as velocity, should always present physically tractable values belonging to the real number range.

Sanitizer Oracle. Although not the primary concern of RoboFuzz, the oracle handler also integrates with compiler sanitizers to support the checking of classic memory-safety issues and miscellaneous software-oriented errors in the robotic SUT. If the target node is compiled with sanitizer flags, the sanitizer oracle detects the error signals emitted by AddressSanitizer [46] or UndefindBehaviorSanitizer [26] when the system terminates.

4.2 Testing `PX4` Quadcopter

Mutation and Scheduling. `PX4` is a complicated system that accepts multiple forms of inputs; one can (1) send **trajectory setpoint** messages that consist of the target position (x, y, z), orientation (yaw), speed $(v_x, v_y, v_z, yawspeed)$, and acceleration (a_x, a_y, a_z) to have the internal motion planner work out movement control plans to reach each setpoint; (2) directly send a stream of **control**

commands that consists of ($throttle, yaw, pitch, roll$) values corresponding to the position of the control sticks of a remote control unit to micro-control the drone; or (3) switch the **parameter** values as suggested by [20] to dynamically configure the drone in runtime. RoboFuzz considers all three input spaces and mutates and schedules messages accordingly; for (1), RoboFuzz mutates one attribute (*e.g.*, v_x) of one message in the sequence of trajectory setpoint messages and publishes them at a given interval (*e.g.*, 20 Hz); for (2), RoboFuzz mutates one of the stick values of one of the control command messages in the sequence and publishes the entire sequence; and for (3), RoboFuzz mutates the value of a parameter and requests parameter change through the MAVLink protocol.

Oracles. Three types of oracles are formulated for `PX4`:

- Parameter specification-based oracle: The behavior of `PX4`-based robots is configured by *parameters*, which are numeric values used in the controller software to configure the robot (*e.g.*, maximum velocity along the z-axis). As the system makes various assumptions based on the parameters, violating the parameters could lead to safety-critical situations, as demonstrated in §6. We parsed the PX4 documentation for parameter values and mapped vehicular states with each parameter so that the oracle checks whether the captured states satisfy the valid range set by the corresponding parameter and reports the violations.
- Safety oracle: A collision is considered one of the most undesirable events during flights, and any collision needs to be investigated regardless of the cause. The safety oracle checks for collisions using the data read by contact sensors.
- Flight mode oracle: `PX4` supports various flight modes, and each mode utilizes different controller logic, leading to different maneuvers. The documentation of `PX4` introduces each mode, specifying the expected behaviors of the mode. For each mode, the oracle models the control algorithm, checking whether the described control is being executed. For example, the "takeoff mode" brings a drone straight to the desired altitude and does not require any external controls. If a drone moves horizontally in this mode, the incident is reported by the oracle as a bug.

Feedback. Two `PX4`-specific feedback metrics are applied:

- Redundant sensor inconsistency: Pixhawk 4 (a flight controller hardware designed to run `PX4` firmware) [32] has two Inertial Measurement Unit (IMU) sensors of different vendors: BMI-055 of Bosch and ICM-20689 of TDK. Both sensors utilize an accelerometer and a gyroscope to measure the change in the position (location) and orientation (rotation) of the drone. Even though the two sensors vary in fidelity, they report consistent acceleration and angular velocity measurements with negligible discrepancy. However, we observed that their responses to anomalous or extreme events, such as collisions or flips that bring about rapid movements, are divergent, resulting in a considerable discrepancy in the readings. Accordingly, we interpret the difference between two IMU readings as an instability metric and use it as an element of the execution feedback, such that inputs that lead to larger discrepancies are favored.
- Position estimation error: `PX4` utilizes the Extended Kalman Filter (EKF) [34] to combine the measurements from the IMU, magnetometer, and GPS sensors into the estimation of the global position (*i.e.*, latitude and longitude). As the controller of PX4 computes the thrust amount to minimize the difference between the setpoint (desired position) and the estimation (current position), the estimation error is critical in the behavior of a drone. One of the reasons jeopardizing the reliability of the EKF estimation is known as excessive vibration of the drone body. In light of this, we compute the position estimation error, *i.e.*, the Euclidean distance between the raw GPS sensor position and the EKF-estimated position, and register it as feedback.

4.3 Testing `TB3` and `TSM`

`TB3` is a differential wheeled robot, whose movement relies on two individually actuated wheels on a common axis. It has a 360-degree laser distance sensor (LDS-01) on the top that enables simultaneous localization and mapping (SLAM) [49] and autonomous driving. `TSM` is essentially a 2D version of `TB3`, which is velocity controlled.

Mutation and Scheduling. `TB3` runs a designated node that receives **twist** messages, which consist of target translational speed (m/s) and rotational speed (rad/s), and runs a differential drive algorithm to control the torque generated by two motors. Accordingly, RoboFuzz mutates the target speed, publishes the message to the node, and then monitors the dynamic states of `TB3`, including its position, velocities, and laser scan data.

Oracles. Along with the common oracles, we designed two `TB3`-specific oracles.

- Specification-based oracle: Similar to the `PX4` oracles, physical invariants of the robot, such as the maximum translational and rotational velocities, are collected from the specification document of `TB3`. In addition, from the datasheet and the code of the laser distance sensor driver, the range of valid scan distance data is obtained, which is also checked by the TB3 oracle.
- Goal-based control oracle: Unless the commanded linear and rotational speeds do not exceed the specified limits, `TB3` is bound to achieve the target speed. This fundamental property is checked through the goal-based control oracle.

Feedback. Odometry is a process of measuring the change in position over time. To a robot that does not have a global position sensor (*e.g.*, GPS), accurate odometry is the key to estimating the global position and guaranteeing correct navigation.

- Odometric inconsistency: `TB3` runs odometry to calculate the change in its position and orientation upon the actuation of control commands. Here, the odometry involves *two redundant methods* to calculate the change of yaw angle, one using the measurement of the IMU sensor and the other using the amount of rotation of the wheels. Under normal execution conditions, both methods yield analogous yaw angles. However, when the instability increases (*e.g.*, if the robot sways and the wheels do not stick on the ground), the wheel-based measurement becomes imprecise, as wheel movement does not always lead to the change in position and results in increased odometric inconsistency.

4.4 Testing `MI2` with PANDA Manipulator

PANDA manipulator by Franka Emika is a popular robot arm equipped with seven revolute joints and torque sensors. Its control interface officially supports ROS and `MI2`, enabling the testing.

```
auto node = std::make_shared<rclcpp::Node>("rclcpp_node");
```

(a) rclcpp API to create a node.

```
node = rclpy.create_node('rclpy_node')
```

(b) rclpy API to create a node.

```
rcl_get_zero_initialized_node
rcl_node_init
└ rcl_node_get_logger_name
└ rcl_node_is_valid
  └ rcl_node_is_valid_except_context
└ rcl_node_get_namespace
  └ rcl_node_is_valid_except_context
└ rcl_node_get_name
⋮
```

(c) Call chain of RCL functions that are internally invoked when `rclcpp` or `rclpy` APIs are called to create a node.

Figure 6: API consistency on the ROS Client Library (RCL).

Mutation and Scheduling. `MI2`'s trajectory planner accepts the goal position, which specifies where the hand of the robot arm should be placed, and determines the position and motion of the joints (*i.e.*, angles of revolute joints over time) to realize the goal position through inverse kinematics.

Oracles. Approaching from the control algorithm's angle, we designed three following oracles:

- Inverse kinematics oracle: Depending on the goal position and constraints, the inverse kinematics solution may or may not exist. This oracle emulates the inverse kinematics using an external module and checks whether `MI2` fails to find an existing solution.
- Specification-based oracle: The motion of the robot is physically constrained by various joint constraints, (*e.g.*, minimum and maximum angle a joint can achieve). Similar to other robotic systems, we parse the datasheet of PANDA robot to extract the joint constraints and turn them into an oracle checking for violations.
- Physical correctness oracle: The controller periodically publishes joint states, which consist of position and velocity setpoints computed by the trajectory planner, the actual position and joint velocities, and errors between these vectors. Here, the positions and velocities should be valid numeric values, and the joint velocities should be non-zero when the manipulator is being actuated.

Feedback. As mentioned in §3.6, the joint controller of PANDA manipulator also performs closed-loop feedback control, utilizing the error between the setpoints and actual position.

- Joint-level controller error: The control error measured for each joint indicates how precisely the torque controller is achieving the planned trajectory. Errors exceeding the capacity of the robot cannot be corrected and lead to control failure. Thus, by favoring mutated messages that result in increased joint-level controller errors, we can excite the failure and reveal bugs.

4.5 Testing ROS - RCL API Consistency

As described in §2.2, RCL API implementations are thin wrappers written in specific languages around the core functionalities of the RCL layer. Regardless of the API language, it is expected that the code paths taken in RCL are identical, such that the same RCL functions are invoked and return the same result (*e.g.*, return values). Figure 6 shows an example of RCL API consistency. ROS applications written in C++ can create a node using the `rclcpp` shown in Figure 6(a). Likewise, the `rclpy` API of Figure 6(b) can be used if the ROS system is built in Python. In both cases, each RCL API invocation ends up invoking the same RCL functions listed in Figure 6(c), ensuring that a ROS node is created and initialized in a way that is uniform and thereby correct. The API consistency oracle is designed to check whether two (or more) API implementations emit consistent behaviors with respect to the RCL code path by dynamically tracing the RCL function invocations and the return values during the execution.

Table 1: Code size of the components of RoboFuzz.

Component	LoC	Language
System inspector	161	Python
Message mutator and Scheduler	2,948	
Hybrid executor and Harnesses	1,415	
Oracles and Feedback engine	1,000	

4.6 Testing ROS - ROSIDL-based Type System

ROS IDL is the centerpiece of ROS type system, which not only generates language-specific code (*e.g.*, C struct) for user-defined message interfaces consisting of ROS native types, but also dynamically checks the types of messages coming into and out of ROS topics. Built upon the assumption that ROS IDL works correctly, ROS nodes often omit redundant type checks of the data it handles, and even boundary (minimum and maximum) checks.

Our IDL correctness oracle is designed to test such foundational assumptions. Given a message containing fields f_1 of built-in data type t_1 and f_2 of fixed array of type t_2, the IDL correctness oracle checks for the following five invariants regarding the type system:

(1) assigning a value of type t' to f_1, where $t \neq t'$ should fail,
(2) assigning a value bigger than the *max* of t_1 to f_1 should fail,
(3) assigning a value smaller than the *min* of t_1 to f_1 should fail,
(4) assigning a value of type t' to one of the elements of f_2, where $t_2 \neq t'$ should fail,
(5) the value assignment succeeds otherwise.

5 EVALUATION

In this section, we evaluate the overall effectiveness of RoboFuzz in finding correctness bugs in heterogeneous robotic systems by answering three questions:

Q1. How effective is RoboFuzz in finding robotic bugs? (§5.1)
Q2. How effective is the semantic feedback mechanism? (§5.2)
Q3. How does RoboFuzz compare to a state-of-the-art robotic fuzzer? (§5.3)

Implementation. We prototyped the RoboFuzz framework and the specialized oracles in 5.5k lines of Python3 code [21], as shown in Table 1. The framework is built based on ROS 2 `foxy`, which is the latest LTS distribution of ROS 2.

Cost of Oracle Creation. Two weeks on average were required for a person with a college-level knowledge of robotics to study one system, review its specifications and code, and implement oracles. For the robot developers who better understand their own systems, less creation cost is anticipated.

Experimental Setup. We performed the evaluation on a laptop machine running Ubuntu 20.04, with Intel i7-8850H 2.6Ghz, 16GB RAM, and Quadro P2000 mobile GPU.

We targeted four robotic systems, `PX4`, `TB3`, `MI2`, and `TSM`, as well as ROS 2 internals (`ROSIDL` and `rclpy/rclcpp`). For `PX4` and `TB3`, we

utilized the actual robots along with their simulated clones; PX4 using Holybro X500 frame with Pixhawk4 flight controller, which runs PX4 v1.12, and TB3 "Burger" build, which runs foxy-devel firmware. All simulations were run in a Gazebo 11 simulator [24, 38], except for TSM, which operates in its own 2D simulator. For internal layer testing, we built ROS 2 foxy from the source.

For each of the six targets, we created separate fuzzing instances per input type the target accepts, *e.g.*, three for PX4 (§4.2) and one for each of the rest. Each instance was run for 12 hours with only target-specific oracles activated to prevent bugs in other layers (*e.g.*, underlying bugs in ROS) from overshadowing the target bugs.

Hybrid Execution. RoboFuzz fully automates the hybrid execution at the software level, but physical execution involves inevitable manual actions, *e.g.*, rebooting a robot through a power switch and putting it back at the initial position. In addition, due to battery life, safety, and the need to visit the test site, we intermittently utilized the hybrid executor for physical robots; we tested each robot for approximately 30 minutes each time, for a total of 12 hours, in addition to other fuzzing instances.

5.1 Effectiveness of RoboFuzz

New correctness bugs RoboFuzz detected from the target robotic systems are listed in Table 2. Utilizing the in-house developed bug oracles, RoboFuzz found 30 previously unknown bugs[2]. We reported all the findings to the developers, and as a result, 25 bugs have been acknowledged and six bugs have been fixed so far. It should be noted that among the bugs RoboFuzz found, 13 bugs (43%) reside in the internal ROS layers, potentially affecting any robotic system that is built on top of ROS.

We validated each bug by replaying the recorded input messages against the robotic system under test and observing the consequences, *i.e.*, checking if the oracle handler reports the same errors. Since we dynamically tested systems using concrete input messages and definitive oracles (*e.g.*, specification violation), none of the detected bugs were identified as a false positive.

5.2 Effectiveness of Semantic Feedback

To show the efficacy of utilizing the semantic feedback as fuzzing guidance, we (1) compare the number of bugs found over time with and without the feedback while fuzzing PX4, which is the largest in code size and state space among the targets, (2) analyze the feedback score, demonstrating the correlation between the metrics proposed in §4.2 with the correctness bug, and (3) show how traditional code coverage feedback performs in the meantime.

Fuzzing with and without Feedback. Figure 7 shows the number of bugs RoboFuzz discovered while fuzzing PX4 for 12 hours, with and without the semantic feedback. With the guidance of the redundant sensor inconsistency and position estimation error feedback (§4.2), RoboFuzz found nine instances of bug #06 in the position mode. In contrast, when we disabled the semantic feedback guidance, RoboFuzz relied solely on the random mutation and found only two cases of the same violation in 12 hours.

[2]Each bug was triggered multiple times by different inputs, but counted once per its root cause identified by manual analysis.

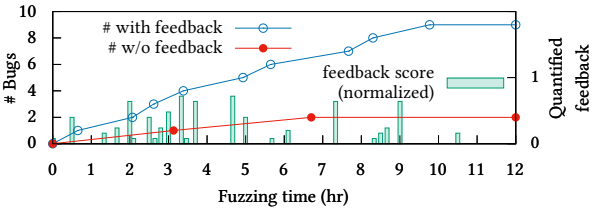

Figure 7: Number of correctness bugs triggered while fuzzing PX4 for 12 hours with and without semantic feedback. Green boxes indicate the quantified semantic feedback generated during the run with feedback, normalized to range [0:1].

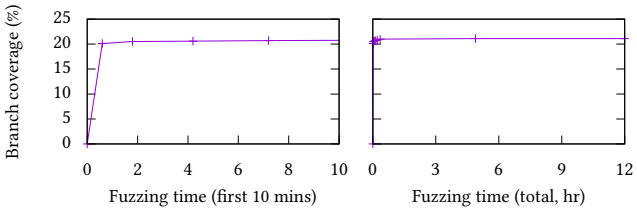

Figure 8: Code coverage measured during the 12-hour fuzzing campaign presented in Figure 7. The first 10 minutes are magnified (left), and the entire result is shown on the right.

Breakdown of the Semantic Feedback Score. Inputs that increase the feedback score are put in the input queue, as they deserve further mutations. In the case of PX4, if the IMU sensor inconsistency or EKF-estimation error increases, the input is marked as interesting. The contribution of feedback scores in finding bugs is illustrated in green boxes in Figure 7. After detecting a bug, the feedback score is reset, as a new fuzzing cycle is initiated. During most cycles, we can observe that the feedback score has a tendency to grow over time, and RoboFuzz ultimately reaches the bugs by focusing on mutating the inputs that generate stronger feedback.

Impact of Traditional Code Coverage Feedback. As discussed in §3.6, robotic systems tend to have limited diversity in the code paths because of their distributed and data-heavy nature. We observed this by instrumenting PX4 and measuring the branch coverage using Gcov [16] during fuzzing, as shown in Figure 8. The code coverage quickly saturated at 21% in the first 10 minutes and then hardly grew during the remaining 12 hours even though the drone exhibited a variety of different behaviors, including nine specification violations. This shows the importance of incorporating supplementary feedback guidance in the robotic context to complement traditional code coverage feedback.

5.3 Comparison with PGFuzz

PGFuzz [20] is a state-of-the-art fuzzer that aims to find policy violations in popular drone control software, including PX4. It identifies policies from the documentation; maps each policy to the input space consisting of configuration parameters, environmental factors, and user commands through static profiling; and mutates the input space trying to minimize the distance to policy violations. By extending the RoboFuzz oracles by implementing 21 PX4 policies PGFuzz identified (including the overlapping policies we already considered), we successfully detected 26 out of 36 policy violations PGFuzz reported. The remaining 10 violations require intentional fault injection (*e.g.*, shutting GPS off via PX4 shell command), which is not the input space considered by RoboFuzz.

Table 2: List of new correctness bugs found and reported by RoboFuzz. R_s: simulated robot, R_p: physical robot.

Layer	#	Target	Bug description	Type	Ack?	Fix?
ROS 2 Applications	01	PX4	(Offboard mode) Horizontal acceleration setpoints are not constrained by MPC_ACC_HOR_MAX parameter	S	✓	
	02	PX4	(Offboard mode) Controller implementation of feed-forward setpoint differs from documentation	S	✓	✓
	03	PX4	(Offboard mode) Mismatch in trajectory setpoint message definition	S	✓	
	04	PX4	(Offboard mode) Incorrect definition of applicable parameters of the flight mode	S	✓	
	05	PX4	(Position mode) Definition of MPC_POS_MODE does not match the implementation, and inconsistent within the documentation	S	✓	✓
	06	PX4	(Position mode) Incorrect usage of MPC_ACC_HOR and MPC_ACC_HOR_MAX definition	S	✓	
	07	PX4	(Position mode) Incorrect description of MPC_ACC_UP_MAX and MPC_ACC_DOWN_MAX parameters	S	✓	
	08	PX4	(Takeoff mode) Switching from Manual/Altitude/Position/Acro mode, drone files towards an arbitrary setpoint	S	✓	
	09	TB3	Motor driver implementation of R_p does not follow the actual motor specification, resulting in a smaller maximum velocity	S	✓	
	10	TB3	Constraining logic on R_p firmware fails to correctly clamp velocity to the valid range	S	✓	
	11	TB3	Maximum torque achievable by R_s is not constrained by the actual maximum torque of the motor specification	S, D		
	12	TB3	Maximum velocity by the differential drive controller of R_s does not match that of R_p	D		
	13	TB3	Mismatch between LiDAR scan data - software side data exceeds the hardware sensor's specified maximum range	D	✓	
	14	MI2	Joint limits are incorrectly defined in the R_s model, exceeding physical limits and allowing invalid postures	S, D	✓	✓
	15	MI2	Joint velocities are broadcasted as zero when the manipulator is moving	P	✓	
	16	TSM	Type confusion while normalizing orientation angles results in physically invalid position values (NaN)	P	✓	✓
	17	TSM	Missing validations of NaN / INF values on the controller inputs leads to physically invalid states	P	✓	
ROS 2 Internals	18	ROSIDL	Code generator for rclpy does not check 32/64 bit float boundaries, treating all float values as double	R	✓	✓
	19	ROSIDL	Runtime message setter does not handle byte types correctly - byte values are internally treated as string literals	R		
	20	ROSIDL	Missing data range checks for the elements of int arrays - *e.g.*, 65535 can be assigned to int8 array	R	✓	
	21	ROSIDL	Missing data range checks for the elements of float arrays - double-sized value can be assigned to float array	R	✓	
	22	ROSIDL	Implicit type casting of array elements alters data without notifying - *e.g.*, assigning -32 to uint8 array, the value becomes 224	R	✓	
	23	ROSIDL	Missing type checks for bool array elements, allowing data of any type to be stored - *e.g.*, string, list, dictionary, ... can become an element	R	✓	
	24	ROSIDL	Missing type checks for byte array elements, allowing data of any type to be stored - *e.g.*, string, list, dictionary, ... can become an element	R	✓	
	25	ROSIDL	Missing type checks for string array elements, allowing data of any type to be stored - *e.g.*, list, dictionary, byte, bool, ... can become an element	R	✓	
	26	rclpy	_on_parameter_event always returns True, regardless of the result	R	✓	✓
	27	rclpy	Missing NULL check of rmw_handle in rclpy_create_publisher leading to null pointer dereference	S	✓	
	28	rclcpp	rclcpp internally throws an exception while validating incorrect topic/service names, which cannot be caught by requester	R	✓	
	29	rclcpp	Node object relies on subscription interface to catch parameter events, violating design principle of parameter callbacks	S		
	30	rclcpp	Unreachable error checking code for parameter event type	R		

P: **P**hysical incorrectness / **S**: **S**pecification violation / **D**: Cyber-physical **D**iscrepancy / **R**: Miscellaneous software error in **R**OS

On the contrary, PGFuzz missed all the new bugs detected by RoboFuzz for the following three reasons. First, the policies PGFuzz manually identified from the documentation do not include all specifications that RoboFuzz oracle checked. Second, some bugs are only triggered by a stream of real-time inputs (*e.g.*, remote control signals) with accurate timing. PGFuzz cannot generate such complicated inputs with its coarse timing model, which only specifies the happens-before relationship between inputs. Finally, due to the lack of soundness of the static profiling engine, PGFuzz missed one policy violation RoboFuzz detected (bug #08 in Table 2) even though the bug violates two of the 21 policies it targeted. Since the static profiler maps inputs to target parameters by tracking dependency, the state transitions that rely on implicit dependencies, such as the case of bug #08, are removed from the input space.

6 CASE STUDY

Correctness bugs in robotic systems are subtle and complicated to reason about, as both hardware and software components are involved. In this section, we introduce and analyze interesting correctness bugs RoboFuzz discovered.

PX4 - Drone Loses Control with Unconstrained Acceleration Setpoint. Offboard mode is one of the autonomous modes that realizes trajectory setpoints received from a companion computer. Bugs #01-02 in Table 2 are related to the combination of specification violation and missing input sanitation, as detailed below. First, even though the documentation of PX4 defines MPC_ACC_HOR_MAX as "maximum horizontal acceleration for auto mode and for manual mode" and assigns a default value of 5 m/s^2, this constraint is not applied to the acceleration setpoints, allowing any value to be forwarded to the controller. Consequently, an abnormally large horizontal acceleration setpoint, *e.g.*, $a_y = 3000\ m/s^2$, immediately contaminates the internal controller states, which results in generating huge horizontal thrust. The controller keeps pushing the drone at the maximum velocity as it tries to realize the setpoint acceleration (3000 m/s^2), while the actual acceleration saturates at the hardware limit. Unfortunately, once the internal states are contaminated and the drone gears up for the anomalous setpoint, it cannot be recovered to a stable state even by updating the acceleration setpoint, *e.g.*, $a_y = 0\ m/s^2$, and safety is no longer guaranteed at this point.

TB3 - Legitimate Linear Velocity Command Being Silently Ignored (Bugs #09-10). The documented maximum linear velocity of TB3 is 0.22 m/s. When we publish a linear velocity control command ($vel_x^{desired} = 0.22$), the ROS node reads the command and writes it to the control table of the firmware. On the firmware side, this value is constrained by the internal logic that computes the maximum linear velocity with respect to the motor power as:

$$\begin{aligned} vel_x^{max} &= wheel\_radius * 2\pi * motor\_rpm/60 \\ &= 0.033 * 2\pi * 61/60 = 0.2108 \end{aligned} \tag{1}$$

Then, the motor driver translates the constrained linear velocity to the wheel velocity and commands the motor if the velocity does not exceed the hardcoded limit, 337. The translation is done by the following equation:

$$\begin{aligned} vel_x^{goal} &= wheel\_radius * RPM * wheel\_vel^{goal} * 0.10472 \\ &= 0.033 * 0.229 * wheel\_vel^{goal} * 0.10472 \end{aligned} \tag{2}$$

Plugging the goal velocity in, we get the desired wheel velocity:

$$\begin{aligned} wheel\_vel^{goal} &= vel_x^{goal} * 1263.6329 \\ &= 0.2108 * 1263.6329 = 266.37 \end{aligned} \tag{3}$$

As the final wheel velocity, 266.37, does not exceed the limit (337), this command is accepted and sent to the actual motor. However, the documented velocity limit of the motor is 265 [37], and the motor silently fails to execute the commanded velocity. As a result, users end up sending a legitimate command that is accepted by the firmware, but cannot make the robot move.

Turtlesim - Type Confusion in normalizeAngle Function (Bug #16). Turtlesim's controller receives desired angular and linear velocities as commands and simulates the motion and position of a turtle every $dt = 16\ ms$. The orientation angle θ is updated by:

$$\theta_{new} = normalizeAngle(\theta_{current} + ang\_vel * dt) \quad (4)$$

Here, normalizeAngle normalizes the orientation angle to $[-\pi : \pi]$, so that the posture can be visualized in the simulation. Unfortunately, the return type is mistakenly assigned as `float`, which makes the orientation `NaN` if the commanded angular velocity is sufficiently large. As a result, the turtle's position becomes physically infeasible.

rclpy - Concurrency Issue Due to Missing NULL Check (Bug #27). When writing a user-space API for creating a publisher, it is recommended by the `RCL` documentation to "get the rmw handle from the publisher using `rcl_publisher_get_rmw_handle` each time it is needed" to avoid potential concurrency issues with the functions that can change the rmw handle pointer. `rclcpp` does this at the end of its `create_publisher` function. However, the same check is missing in `rclpy_create_publisher` of `rclpy` implementation, resulting in a NULL-pointer dereference error when the rmw handle pointer is concurrently invalidated by other functions.

7 DISCUSSION AND FUTURE WORK

Addressing Physical-simulation Discrepancy. A number of uncertain factors in the physical environment could hinder RoboFuzz from reliably obtaining consistent execution data. To this end, we made the following design decisions to minimize the disruption by physical-simulation discrepancies. First, we designed simulation maps as identical as possible to the actual test sites. For example, in Figure 5, the dimensions and locations of the static objects were reflected in the simulation map as accurately as possible. Second, we designed state discrepancy oracles to check for *reasonably obvious* deviations, *e.g.*, the difference of the final velocities rather than their trends, so that trivial noises can be filtered. Last but not least, we ran tests only on clear, windless days to minimize the influence of weather on the robot.

Limitation. With RoboFuzz, we attempt to excite the faults in robotic systems by injecting valid (*i.e.*, correctly formatted) messages on the ROS level. Such positive testing is useful for revealing various issues that can occur during normal operations, including the correctness bugs we addressed in this paper. However, from a holistic point of view, the security and robustness of robotic systems can still be threatened from different vectors. For example, the underlying communication protocol (*e.g.*, DDS) might be flawed, allowing malformed messages to be transmitted and crash the entire system or breaking the integrity of messages, causing unexpected, hard-to-debug errors. Furthermore, various physical attacks can be launched directly to robots. In our future work, we plan to develop testing methodologies for the layers surrounding ROS to cover such attack vectors and render safer ecosystem for robotic systems.

8 RELATED WORK

Testing ROS. Most existing work on ROS focuses on securing the ROS network. Dieber *et al.* [14] and the SROS project [50] analyzed potential network-based attack vectors and proposed a TLS-based secure channel for inter-node communications. ROSPenTo [13] is a penetration testing suite to analyze ROS networks, and McClean *et al.* [28] demonstrated a honeypot. RoboFuzz instead takes an orthogonal approach and focuses on the internal failures that break the assumptions and threaten the robustness of ROS-based systems.

Fuzzing for Cyber-physical Systems. RVFuzzer [22] and PGFuzz [20] target parameter errors and policy violations in specific drone controllers. RoboFuzz complements these approaches by serving as (1) a generic framework that seamlessly integrates with any ROS-based robotic system and (2) a highly extensible framework with which various developer assumptions and specifications can be modeled and tested through feedback-guided mutation. As a result, RoboFuzz detects not only the `PX4` bugs found by PGFuzz, but also more bugs from `PX4` and other robotic systems.

CPFuzz [47] demonstrates the applicability of AFL-based coverage-guided greybox fuzzing to the controller programs of cyber-physical systems (*e.g.*, a heater system) to identify safety violations. Testing various small (< 100 LoC) benchmark controllers, the approach successfully detects specification violations. However, as shown in §5.2, the effectiveness of code-coverage guidance is dampened in larger codebases. RoboFuzz counters this problem by accommodating semantics-based feedback and successfully identifies correctness bugs from large robotic systems.

Feedback-driven Fuzz Testing. A number of feedback-driven fuzzing approaches are proven to be effective in finding bugs in user-space applications [1, 51], and kernels [2, 44]. Unfortunately, existing fuzzing approaches primarily focus on finding memory-safety issues, and thus cannot be applied to finding correctness bugs in robotic systems. RoboFuzz brings the concept of feedback-driven fuzzing into the robotics domain, enabling effective testing.

9 CONCLUSION

This paper presents RoboFuzz, a semantic feedback-driven fuzzer tailored for finding correctness bugs prevalent in ROS and ROS-based robotic systems. By injecting mutated messages to the robots running in both the real world and a simulator and checking the captured robotic states against carefully designed robot-specific correctness oracles, RoboFuzz goes beyond memory-safety bugs and detects subtle correctness bugs, including the violation of physical laws, violation of specifications, and cyber-physical discrepancies. This process is further expedited by incorporating semantic feedback as guidance, which quantifies the bugginess from the execution context. As a result, RoboFuzz has detected 30 new correctness bugs in ROS and ROS-based applications.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their insightful feedback. This research is funded by the Secure Systems Research Center (SSRC) at the Technology Innovation Institute (TII), UAE. We are grateful to Dr. Shreekant (Ticky) Thakkar and his team members at the SSRC for their valuable comments and support.

REFERENCES

[1] 2017. LibFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.
[2] 2018. syzkaller - kernel fuzzer. https://github.com/google/syzkaller.
[3] 2020. ROS Community Metrics Report. http://download.ros.org/downloads/metrics/metrics-report-2020-07.pdf.
[4] 2022. ROS Robots. https://robots.ros.org/.
[5] Amazon. 2022. Prime Air. https://www.amazon.com/primeair.
[6] Robin Amsters and Peter Slaets. 2019. Turtlebot 3 as a Robotics Education Platform. In *International Conference on Robotics in Education (RiE)*. Springer, 170–181.
[7] Bloomberg. 2022. The Rise of ROS: Nearly 55% of total commercial robots shipped in 2024 Will Have at Least One Robot Operating System package. https://www.bloomberg.com/press-releases/2019-05-16/the-rise-of-ros-nearly-55-of-total-commercial-robots-shipped-in-2024-will-have-at-least-one-robot-operating-system-package.
[8] 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*. 2329–2344.
[9] Breiling, Benjamin and Dieber, Bernhard and Schartner, Peter. 2017. Secure communication for the Robot Operating System. In *2017 annual IEEE international systems conference (SysCon)*. IEEE, 1–6.
[10] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 711–725.
[11] Sachin Chitta. 2016. MoveIt!: an introduction. In *Robot Operating System (ROS)*. Springer, 3–27.
[12] DeMarinis, Nicholas and Tellex, Stefanie and Kemerlis, Vasileios P and Konidaris, George and Fonseca, Rodrigo. 2019. Scanning the internet for ros: A view of security in robotics research. In *2019 International Conference on Robotics and Automation (ICRA)*. IEEE, 8514–8521.
[13] Bernhard Dieber, Ruffin White, Sebastian Taurer, Benjamin Breiling, Gianluca Caiazza, Henrik Christensen, and Agostino Cortesi. 2020. Penetration testing ROS. In *Robot operating system (ROS)*. Springer, 183–225.
[14] Dieber, Bernhard and Breiling, Benjamin and Taurer, Sebastian and Kacianka, Severin and Rass, Stefan and Schartner, Peter. 2017. Security for the Robot Operating System. *Robotics and Autonomous Systems* 98 (2017), 192–203.
[15] Edwards, Shaun and Lewis, Chris. 2012. ROS-Industrial: Applying the Robot Operating System (ROS) to Industrial Applications. In *IEEE Int. Conference on Robotics and Automation, ECHORD Workshop*.
[16] GNU. 2022. gcov — a Test Coverage Program. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.
[17] Google. 2022. Wing. https://wing.com/.
[18] iRobot. 2021. Investor Presentation August 2021. https://investor.irobot.com/static-files/a6147f70-f50a-43d3-9161-9af57981ea0f.
[19] Mehdi Jafari. 2015. Optimal redundant sensor configuration for accuracy increasing in space inertial navigation system. *Aerospace Science and Technology* 47 (2015), 467–472.
[20] Hyungsub Kim, Muslum Ozgur Ozmen, Antonio Bianchi, Z Berkay Celik, and Dongyan Xu. 2021. PGFUZZ: Policy-Guided Fuzzing for Robotic Vehicles. In *Network and Distributed System Security Symposium*.
[21] Seulbae Kim and Taesoo Kim. 2022. RoboFuzz Artifact (Version 1). *Zenodo* (2022). https://doi.org/10.5281/zenodo.7036047
[22] Taegyu Kim, Chung Hwan Kim, Junghwan Rhee, Fan Fei, Zhan Tu, Gregory Walkup, Xiangyu Zhang, Xinyan Deng, and Dongyan Xu. 2019. RVFuzzer: Finding Input Validation Bugs in Robotic Vehicles through Control-Guided Testing. In *28th USENIX Security Symposium (USENIX Security 19)*. 425–442.
[23] 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 (CCS 2018)*.
[24] Nathan Koenig and Andrew Howard. 2004. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, Vol. 3. IEEE, 2149–2154.
[25] Jean-Paul Laumond et al. 1998. *Robot motion planning and control*. Vol. 229. Springer.
[26] LLVM. 2021. Clang 13 Documentation - UndefinedBehaviorSanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.
[27] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. 2022. Robot Operating System 2: Design, architecture, and uses in the wild. *Science Robotics* 7, 66 (2022), eabm6074. https://doi.org/10.1126/scirobotics.abm6074
[28] McClean, Jarrod and Stull, Christopher and Farrar, Charles and Mascarenas, David. 2013. A Preliminary Cyber-Physical Security Assessment of the Robot Operating System (ROS). In *Unmanned Systems Technology XV*, Vol. 8741. International Society for Optics and Photonics, 874110.
[29] Lorenz Meier, Dominik Honegger, and Marc Pollefeys. 2015. PX4: A Node-Based Multithreaded Open Source Robotics Framework For Deeply Embedded Platforms. In *2015 IEEE international conference on robotics and automation (ICRA)*. IEEE, 6235–6240.
[30] Jason M O'Kane. 2014. A gentle introduction to ROS. (2014).
[31] Gerardo Pardo-Castellote. 2003. OMG Data-Distribution Service: Architectural Overview. In *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.* IEEE, 200–206.
[32] PX4. 2022. Pixhawk 4. https://docs.px4.io/master/en/flight_controller/pixhawk4.html.
[33] Quigley, Morgan and Conley, Ken and Gerkey, Brian and Faust, Josh and Foote, Tully and Leibs, Jeremy and Wheeler, Rob and Ng, Andrew Y. 2009. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, Vol. 3. Kobe, Japan, 5.
[34] Maria Isabel Ribeiro. 2004. Kalman and Extended Kalman Filters: Concept, Derivation and Properties. *Institute for Systems and Robotics* 43 (2004), 46.
[35] Robotics And Automation News. 2021. US DoD selects Stratom to migrate robotic systems from ROS-1 to ROS-2. https://roboticsandautomationnews.com/2021/08/23/us-selects-stratom-to-migrate-robotic-systems-from-ros-1-to-ros-2/45763/.
[36] Robotis. 2017. TurtleBot3. https://www.robotis.us/turtlebot-3/.
[37] Robotis. 2022. XL430-W250 motor specification. https://emanual.robotis.com/docs/en/dxl/x/xl430-w250.
[38] ROS. 2022. Gazebo. https://gazebosim.org/.
[39] ROS. 2022. MoveIt Motion Planning Framework. https://moveit.ros.org/.
[40] ROS. 2022. ROS-Industrial. https://rosindustrial.org/.
[41] ROS. 2022. ROS-M. https://rosmilitary.org/.
[42] ROS Discourse. 2016. Announcing SROS! Security enhancements for ROS. https://discourse.ros.org/t/announcing-sros-security-enhancements-for-ros/536.
[43] ROS.org. 2016. Automatic Testing with ROS. http://wiki.ros.org/Quality/Tutorials/UnitTesting.
[44] 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)*. 167–182.
[45] Kostya Serebryany. 2017. OSS-Fuzz-Google's continuous fuzzing service for open source software. (2017).
[46] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 309–318.
[47] Fute Shang, Buhong Wang, Tengyao Li, Jiwei Tian, and Kunrui Cao. 2020. CPFuzz: Combining fuzzing and falsification of cyber-physical systems. *IEEE Access* 8 (2020), 166951–166962.
[48] Sheetz, Kyle H and Claflin, Jake and Dimick, Justin B. 2020. Trends in the Adoption of Robotic Surgery for Common Surgical Procedures. *JAMA network open* 3, 1 (2020), e1918911–e1918911.
[49] Sumegh Pramod Thale, Mihir Mangesh Prabhu, Pranjali Vinod Thakur, and Pratik Kadam. 2020. ROS based SLAM implementation for Autonomous navigation using Turtlebot. In *ITM Web of Conferences*, Vol. 32. EDP Sciences, 01011.
[50] Ruffin White, Dr Christensen, I Henrik, Dr Quigley, et al. 2016. SROS: Securing ROS over the wire, in the graph, and through the kernel. *arXiv preprint arXiv:1611.07060* (2016).
[51] Zalewski, Michal. 2014. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl.