2022 IEEE International Conference on Robotics and Automation (ICRA)
May 23-27, 2022. Philadelphia, PA, USA

ROZZ: Property-based Fuzzing for Robotic Programs in ROS

Kai-Tao Xie, Jia-Ju Bai*, Yong-Hao Zou and Yu-Ping Wang

Abstract— **ROS is popular in robotic-software development, and thus detecting bugs in ROS programs is important for modern robots. Fuzzing is a promising technique of runtime testing. But existing fuzzing approaches are limited in testing ROS programs, due to neglecting ROS properties, such as multi-dimensional inputs, temporal features of inputs and the distributed node model. In this paper, we develop a new fuzzing framework named ROZZ, to effectively test ROS programs and detect bugs based on ROS properties. ROZZ has three key techniques: (1) a *multi-dimensional generation method* to generate test cases of ROS programs from multiple dimensions, including user data, configuration parameters and sensor messages; (2) a *distributed branch coverage* to describe the overall code coverage of multiple ROS nodes in the robot task; (3) a *temporal mutation strategy* to generate test cases with temporal information. We evaluate ROZZ on 10 common robotic programs in ROS2, and it finds 43 real bugs. 20 of these bugs have been confirmed and fixed by related ROS developers. We compare ROZZ to existing approaches for testing robotic programs, and ROZZ finds more bugs with higher code coverage.**

I. Introduction

An increasing number of modern robots are built based on ROS [1], which contains many practical software libraries and tools for robot development. However, developing reliable and secure ROS programs is actually difficult, as they often have complex control logics and need to handle various kinds of exceptions (such as invalid user data and abnormal sensor messages). Because ROS programs often control robots to interact with the physical world and humans, even a simple bug (such as null-pointer dereference) can cause dangerous consequences. Moreover, the attackers can exploit security bugs (such as use-after-free bugs) to steal confidential data or even maliciously take over the robot. Thus, testing ROS programs for bug detection is important.

Fuzzing is a popular technique of runtime testing to cover infrequently-executed code and detect bugs. Most fuzzing approaches [2]–[9] automatically mutate and generate program inputs according to the program feedback of code coverage. These approaches have shown good results of bug detection in real-world applications. Thus, applying fuzzing to ROS programs seems promising. But in practice, fuzzing ROS programs is difficult and challenging due to ROS properties:

P1) ROS programs receive multi-dimensional inputs, such as user data, configuration parameters and sensor messages. But existing fuzzing approaches generate test cases from only one dimension, namely user inputs. Thus, these approaches may fail to cover much code in ROS programs.

P2) ROS programs are executed in the distributed node model. Specifically, each ROS program runs as a ROS node, which communicates with other ROS nodes to cooperatively perform a robot task. Thus, testing a separate ROS node is often meaningless, and multiple ROS nodes in the task should be tested together. But existing fuzzing approaches focus on testing a specific program at a time according to its code coverage, and thus they are limited in testing distributed programs, due to lacking an effective coverage metric to describe the overall feedback of these programs.

P3) The inputs of ROS programs have temporal features. Specifically, messages sent from each ROS node are sequential, and such a message sequence can be disordered due to communication instability caused by network interruption or USB disconnection. But existing fuzzing approaches focus on mutating input data without considering temporal information. Thus, these approaches cannot effectively test the code of handling temporal information.

Several recent approaches [10]–[14] explore the application of fuzzing to robotic programs, using some new fuzzing strategies and bug checkers. However, these approaches still fail to solve the above three challenges. For example, these approaches generate test cases from only one dimension, such as user data or sensor messages.

To solve the above three challenges and improve fuzzing on ROS programs, we propose three key techniques:

For *P1*, we design a *multi-dimensional generation method* to generate effective test cases of ROS programs from multiple dimensions, including user data, configuration parameters and sensor messages. Specifically, user data include control information from GUI, command lines and ROS services. Configuration parameters are stored in specific configuration files, and they are used to configure ROS nodes for the robot task. Sensor messages are collected from sensor nodes used in the robot task, and we fuzz them because sensors can be malfunctioning or untrusted [14]. To ensure the syntactical correctness of test cases, we generate the inputs from each dimension according to related grammar rules and formats.

For *P2*, we propose a *distributed branch coverage* to describe the overall code coverage of multiple ROS nodes in the robot task. During the test of robot task, we collect and combine the covered branches of each ROS node as the overall feedback of the involved ROS nodes. If any new branch in any ROS node is covered at runtime, we identify the test case of this test to be an interesting seed, and put it into a seed pool for further test-case mutation and generation.

For *P3*, we design a *temporal mutation strategy* to generate test cases with temporal information. For each sensor node,

Kai-Tao Xie, Jia-Ju Bai, Yong-Hao Zou and Yu-Ping Wang are with the Department of Computer Science and Technology, Tsinghua University, Beijing, China. Jia-Ju Bai is the corresponding author.

2022 IEEE International Conference on Robotics and Automation (ICRA) | 978-1-7281-9681-7/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICRA46639.2022.9811701

978-1-7281-9680-0/22/$31.00 ©2022 IEEE

we collect the message sequence sent from this node to other ROS nodes, and mutate the order of this message sequence to simulate communication instability. Specifically, we provide three available patterns for temporal mutation, including message dropping, resending and reordering.

Based on the three key techniques, we develop a novel and automated fuzzing framework named ROZZ, to effectively test ROS programs and detect bugs. We implement ROZZ with LLVM [15]. Overall, we make three main contributions:

- To improve fuzzing on robotic programs, we propose three key techniques: (1) a *multi-dimensional generation method* to generate test cases of ROS programs from multiple dimensions; (2) a *distributed branch coverage* to describe the overall code coverage of multiple ROS nodes; (3) a *temporal mutation strategy* to generate test cases with temporal information.
- Based on the three key techniques, we develop a novel fuzzing framework named ROZZ to test ROS programs.
- We evaluate ROZZ on 10 common robotic programs in ROS2, and it finds 43 real bugs. 20 of them have been confirmed and fixed by related ROS developers. We also experimentally compare ROZZ to existing approaches of testing robotic programs, and ROZZ finds more bugs with higher code coverage.

The rest of this paper is organized as follows. Section II introduces our key techniques for ROS fuzzing. Section III shows ROZZ. Section IV presents the evaluation. Section V discusses the limitations and future work. Section VI introduces the related work. Section VII concludes this paper.

II. Key Techniques

A. Multi-Dimensional Generation Method

When the robot runs multiple ROS programs to perform a task, it often receives inputs from three dimensions: (1) *user data*, namely the data provided the user to manage the robot, including control information from GUI, command lines and ROS services; (2) *configuration parameters*, which are read from specific configuration files to configure the robot, such as maximum moving speed and minimum rotation angle; (3) *sensor messages*, namely the messages generated from sensors (such as laser lidar) used to communicate with the physical world. In fact, the inputs from these dimensions can be invalid or even malicious (such as messages from untrusted sensors). To test whether ROS nodes can correctly handle such cases, our method generates test cases containing the inputs from these dimensions, as shown in Figure 1.

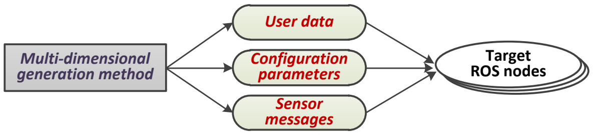


Fig. 1. Test-case generation from multiple dimensions.

Our method stores user data, configuration parameters and sensor messages in some specific input files. ROS programs read these input files to perform the task during fuzzing. In each test, our method mutates and combines the data of these input files to generate a new test case for ROS programs. Note that the inputs from each dimension have specific grammar rules and fixed formats, and the inputs violating these grammar rules and formats are directly dropped by ROS programs without further processing. To generate syntactically-correct inputs, our method automatically parses the input files before mutation.

In practice, sensors used in the robot are provided by hardware vendors, and thus it is difficult for the user to arbitrarily change sensor messages. This is different from user data and configuration parameters, which can be arbitrarily changed by the user. Moreover, we believe that each sensor works normally in most cases, and it only occasionally malfunctions due to hardware problems. Thus, we think that generating totally artificial sensor messages is not realistic enough to find real bugs. With this in mind, before fuzzing, our method first runs the robot normally to performs the task and collects sensor messages during execution; and then our method mutates only several items in the collected sensor messages during fuzzing. In this way, the generated sensor messages can be more realistic to find real bugs.

B. Distributed Branch Coverage

In ROS, robotic programs are executed in the distributed node model. Specifically, each ROS program runs as a ROS node, which communicates with other ROS nodes via message passing to cooperatively perform a task. Thus, multiple ROS nodes in the task should be tested together. To describe the code coverage of these ROS nodes, we first collect the covered code branches of each ROS node in the task, and then combine these code branches with ROS node name to make up the distributed branch coverage of this task.

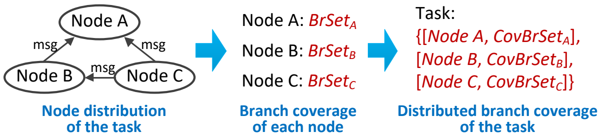


Fig. 2. Example of distributed branch coverage.

Figure 2 shows an example task that involves three ROS nodes. While these nodes perform the task, we collect the sets of their covered code branches as $BrSet_A$, $BrSet_B$ and $BrSet_C$. Then, we combine these branch sets with the names of related ROS nodes, and calculate the distributed branch coverage of this task as $\{[Node\ A,\ BrSet_A],\ [Node\ B,\ BrSet_B],\ [Node\ C,\ BrSet_C]\}$. With this metric, we can clearly identify the code-branch changes of each ROS node in different tests. Thus, we use this metric as program feedback when fuzzing multiple ROS nodes in the given task.

C. Temporal Mutation Strategy

Each ROS node communicates messages with other ROS nodes via socket by default [16], and messages sent from this node make up a message sequence. In practice, such communication can be instable to make a message sequence disordered. This case is relatively common for sensor nodes, as sensors are often connected with the robot via network or USB bus that can be instable. For example, some messages in the message sequence from sensors can be lost, due to

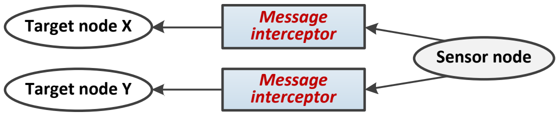


Fig. 3. Message interception.

network interruption or USB disconnection. To test whether ROS nodes can correctly handle such cases, our strategy changes the order of several messages in the sensor-message sequence to generate test cases with temporal information.

As shown in Figure 3, our strategy automatically inserts a *message interceptor* between the sensor node and the target node, to intercept message sequences from the sensor node. To simulate communication instability, this interceptor receives the message sequence generated from the sensor node, then mutates the order of this message sequence to change its temporal information, and finally sends the modified message sequence to the target node. Each message interceptor is implemented as a light-weight ROS node.

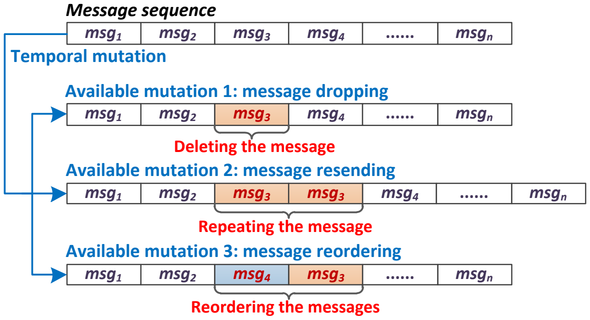

Message sequence
msg1 msg2 msg3 msg4 ...... msgn
Temporal mutation
Available mutation 1: message dropping
msg1 msg2 msg3 msg4 ...... msgn
Deleting the message
Available mutation 2: message resending
msg1 msg2 msg3 msg3 msg4 ...... msgn
Repeating the message
Available mutation 3: message reordering
msg1 msg2 msg4 msg3 ...... msgn
Reordering the messages

Fig. 4. Available patterns of temporal mutation.

Inspired by recent approaches of testing distributed systems [17], [18], our strategy provides three available patterns for temporal mutation of message sequences, including message dropping, resending and reordering. When temporal mutation is performed, one of these patterns is randomly selected. Figure 4 shows the examples of these three patterns. Different from mutating the data of sensor messages in Section II-A, our strategy only changes the order of messages in the sequence without changing message data. When generating test cases, our temporal mutation works together with data mutation of messages, to perform spatio-temporal mutation of sensor messages.

III. ROZZ Design

Based on the three key techniques in Section II, we design a novel fuzzing framework named ROZZ, to effectively test ROS programs and detect bugs based on ROS properties. We have implemented ROZZ using Clang 9.0 [19]. ROZZ automatically performs code instrumentation and dynamic analysis on the LLVM bytecode of the tested ROS programs.

Figure 5 shows the overall architecture of ROZZ, which consists of four parts:

Code analyzer. It first uses Clang to compile the source code of ROS programs into LLVM bytecode, then instruments code branches in the LLVM bytecode, and finally compiles the instrumented LLVM bytecode to generate executable ROS nodes. Moreover, the analyzer automatically modifies the configuration files of ROS nodes in the robot task to enable message interceptors for sensor nodes.

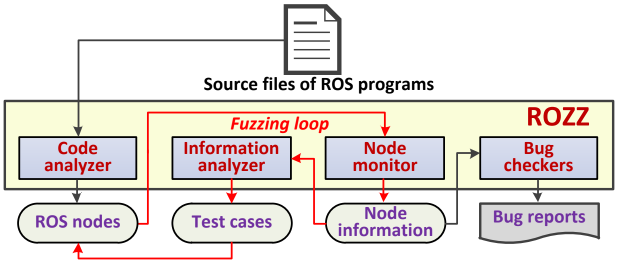


Fig. 5. ROZZ architecture.

Information analyzer. First, it checks the distributed branch coverage for multiple ROS nodes in the test. If this metric is increased, namely any new code branch in any ROS node is covered, the analyzer identifies the test case of this test to be an interesting seed, and puts it into a seed pool for further test-case mutation and generation. Second, the analyzer randomly selects a seed from the seed pool, and uses our multi-dimensional generation method and temporal mutation strategy to mutate this seed and generate new test cases. Each test case contains user data, configuration parameters and sensor messages for multiple ROS nodes, and the analyzer randomly mutates the data of these inputs or the order of sensor messages.

Node monitor. It executes the instrumented code to collect covered code branches in each ROS node, and then calculates the distributed branch coverage of ROS nodes in the test. Besides, the monitor also collects the runtime information about node execution for bug detection.

Bug checkers. They analyze the collected runtime information to detect bugs and generate bug reports. These checkers are independent from ROZZ. The user can implement customized checkers or conveniently run third-party sanitizers (such as ASan [20], MSan [21] and UBSan [22]) in ROZZ.

IV. Evaluation

We test 10 common robotic programs in ROS2, and they are of the latest versions as of our evaluation. Table I shows the information about these ROS programs (the lines of source code are counted by CLOC [23]). These programs are executed on a virtual robot TurtleBot3 Waffle in the robot simulation framework Gazebo 11.5 [24], to perform a robot navigation (for movement and localization programs) and a map building (for movement and SLAM programs) tasks. The used virtual sensors include a laser lidar, an odometry, a 2D camera and an IMU (Inertial Measurement Unit). The experiments run on a regular x86-64 desktop with eight Intel processors and 20GB physical memory. The used operating system is Ubuntu 20.04, and the ROS2 version is ROS2 Foxy.

A. Runtime Testing

We run a common and third-party sanitizer ASan [20] with ROZZ to detect memory bugs at runtime. Each program is tested with the related robot task for 24 hours. Table II shows the testing results.

TABLE I

INFORMATION ABOUT THE TESTED ROS PROGRAMS.

Type	Program	Description	LOC
Movement	nav2_bt_navigator [25]	BT navigator module in ROS2 navigation	6.2K
	nav2_planner [26]	Path planner module in ROS2 navigation	10.4K
	nav2_recoveries [27]	Recovery module in ROS2 navigation	1.2K
	nav2_controller [28]	Controller module in ROS2 navigation	6.3K
Localization	nav2_amcl [29]	Localization module in ROS2 navigation	14.4K
	lama_loc [30]	Alternative localization and mapping method	9.5K
	ekf_loc [31]	Kalman Filter based localization method	0.8K
SLAM	rtab-map [32]	Real-time RGB-D SLAM approach	26.1K
	slam_toolbox [33]	a set of tools and capabilities for 2D SLAM	15.1K
	cartographer [34]	Real-time 2D and 3D SLAM approach	7.8K

TABLE II

RESULTS OF FUZZING ROBOTIC PROGRAMS IN ROS2.

Program	Covered branch		Bug detection	
	Fuzzing	*Test suite*	*Found bug*	*Confirmed bug*
nav2_bt_navigator	24.3K	17.2K	3	2
nav2_planner	27.9K	25.9K	6	5
nav2_recoveries	15.8K	16.4K	2	2
nav2_controller	37.6K	25.1K	3	1
nav2_amcl	12.2K	12.0K	9	7
lama_loc	25.3K	20.7K	3	0
ekf_loc	16.2K	5.2K	1	0
rtab-map	70.9K	59.3K	8	1
slam_toolbox	30.9K	10.7K	3	2
cartographer	59.9K	52.1K	5	0
Total	320.9K	244.5K	43	20

Testing coverage. To understand testing-coverage improvement of ROZZ, we also run official test suites or benchmarks of each ROS program for 24 hours and collect the number of covered code branches. Compared to running test suites, ROZZ covers 31% more code branches in the tested programs, benefiting from our fuzzing approach based on ROS properties. Note that we find no bug when running official test suites or benchmarks.

Found bugs. ROZZ finds 43 real bugs in the test programs with no false positive. These bugs can be actually reproduced using ROZZ and its generated test cases. We have reported these bugs to related ROS developers, and 20 of them have been confirmed and fixed. We are still waiting for the feedback to the remaining bugs. 14 of our patches fixing 18 bugs have been applied. Note that five tested programs whose names contain *"nav2"* are developed and maintained by the ROS community, and these programs are widely used in ROS-based robots. Therefore, the 17 confirmed bugs in these five programs receive serious attention by ROS developers.

Bug types. We also classify the 43 found bugs according to their types, and summarize the results in Table III. In detail, there are 6 null-pointer dereferences, 5 use-after-free bugs, 3 buffer/stack-overflow bugs, 11 invalid-pointer accesses and 18 uncaught exceptions. Once these bugs are triggered with specific inputs, runtime failures and serious security problems can occur at runtime. Specifically, null-pointer dereferences and uncaught exceptions can cause program crashes that abnormally abort the robot task; buffer/stack-overflow bugs, use-after-free bugs and invalid-pointer accesses can cause undefined behaviors of the robot and increase the risks of malicious attacks to the robot.

Bug features. We manually review the 43 found bugs, and find four interesting features of these bugs:

TABLE III

TYPES OF THE FOUND BUGS.

Program	NullPtr	UAF	Overflow	InvalidPtr	Exception	All
nav2_bt_navigator	1	0	0	0	2	3
nav2_planner	0	2	1	2	1	6
nav2_recoveries	0	2	0	0	0	2
nav2_controller	0	0	1	1	1	3
nav2_amcl	2	0	0	6	1	9
lama_loc	2	0	0	0	1	3
ekf_loc	0	0	0	0	1	1
rtab-map	0	0	1	1	6	8
slam_toolbox	1	1	0	1	0	3
cartographer	0	0	0	0	5	5
Total	6	5	3	11	18	43

(1) The 5 use-after-free bugs are caused by data races. Specifically, a memory object is freed in one thread, but this object is still used in the other thread without synchronization. Due to the non-determinism of concurrent execution, these bugs are hard to find in normal execution.

(2) Among the 18 uncaught exceptions, 8 bugs are caused by internal exceptions in the code of the tested programs, and 10 bugs are caused by external exceptions (they are of *"runtime_error"* type in C++) of API calls from third-party software libraries and ROS core components used by the tested ROS programs. This feature indicates that both internal and external exceptions should be carefully caught and handled in ROS programs. For example, in *nav2_bt_navigator*, *nav2_planner*, *nav2_controller* and *nav2_amcl*, four external exceptions about negative time interval from the ROS *rclcpp* component are not caught, which can cause program crashes. These four bugs have been fixed by our patches through catching and handling related external exceptions.

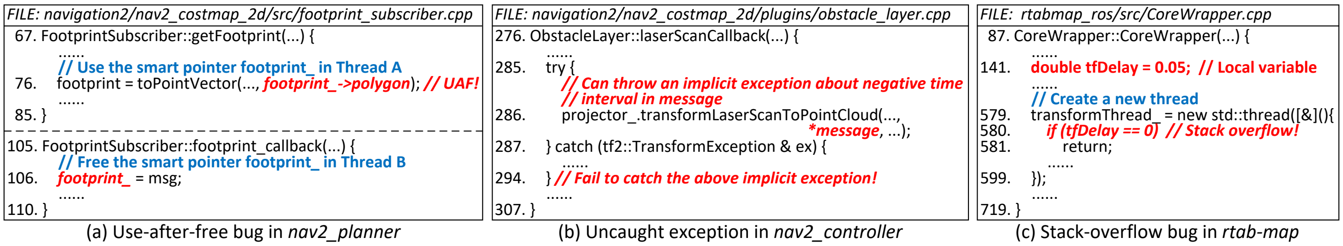
(3) 11 bugs occur in the initialization process of ROS programs. 6 of these bugs are caused by incorrect error handling of invalid user data and bad configuration parameters during initialization; and 5 bugs are caused by missing synchronization between the initialization process and message handling. Thus, the initialization process should receive significant attention when testing ROS programs.

(4) By manually checking the backtraces of the found bugs, we find that 7 bugs are located in third-party libraries and ROS core components used by the tested ROS programs. These libraries and components include *libopencv*, *fastrtps*, *eigen*, *tf2* and *rclcpp*. In the experiments, we use ASan to instrument only the tested ROS programs not these libraries or components, and thus we cannot find the accurate locations of these bugs. To solve these problems, we attempt to use ASan to instrument these libraries and components, but none of them can support ASan in our attempts. Even so, we still locate one bug (stack overflow) in *rclcpp* by manually reviewing its source code without using ASan, and this bug has been confirmed and fixed by the *rclcpp* developers.

B. Root Causes of the Confirmed Bugs

As for the 20 confirmed and fixed bugs, by analyzing the source code and discussing with related ROS developers, we summarize three root causes of these bugs:

(1) 7 bugs are caused by the concurrency of callback functions. In ROS programs, external events (such as message



```
FILE: navigation2/nav2_costmap_2d/src/footprint_subscriber.cpp
 67. FootprintSubscriber::getFootprint(...) {
        ......
        // Use the smart pointer footprint_ in Thread A
 76.    footprint = toPointVector(..., footprint_->polygon); // UAF!
        ......
 85. }
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
105. FootprintSubscriber::footprint_callback(...) {
        // Free the smart pointer footprint_ in Thread B
106.    footprint_ = msg;
        ......
110. }
```

(a) Use-after-free bug in *nav2_planner*

```
FILE: navigation2/nav2_costmap_2d/plugins/obstacle_layer.cpp
276. ObstacleLayer::laserScanCallback(...) {
        ......
285.    try {
          // Can throw an implicit exception about negative time
          // interval in message
286.      projector_.transformLaserScanToPointCloud(...,
                                        *message, ...);
287.    } catch (tf2::TransformException & ex) {
          ......
294.    } // Fail to catch the above implicit exception!
        ......
307. }
```

(b) Uncaught exception in *nav2_controller*

```
FILE:  rtabmap_ros/src/CoreWrapper.cpp
 87. CoreWrapper::CoreWrapper(...) {
        ......
141.    double tfDelay = 0.05;  // Local variable
        ......
        // Create a new thread
579.    transformThread_ = new std::thread([&](){
580.      if (tfDelay == 0)  // Stack overflow!
581.         return;
          ......
599.    });
        ......
719. }
```

(c) Stack-overflow bug in *rtab-map*

Fig. 6. Example bugs found by ROZZ.

arrival) can occur, and each kind of event is handled by a callback function. As an event can occur at any time, its callback function can be concurrently executed with other functions. Thus, concurrency bugs can occur in callback functions due to incorrect synchronization. Figure 6(a) shows an example bug in *nav2_planner*. The callback function `footprint_callback` can be concurrently executed with the function `getFootprint`. In `footprint_callback`, a smart pointer `footprint_` is assigned with a pointer `msg` on line 106, and thus the memory pointed by `footprint_` is freed according to smart-pointer functionality. At the same time, this smart pointer is still used in `getFootprint` to access `footprint_->polygon` on line 76, causing a use-after-free bug [35].

(2) 11 bugs are caused by error handling issues. Specifically, 6 of these bugs are introduced due to missing or incorrect security checks of invalid user data and bad configuration parameters; and the other 5 bugs are introduced due to incorrect error handling after correct security checks. Figure 6(b) shows an example bug in *nav2_controller*. The function `_transformLaserScanToPointCloud` can throw one explicit exception and one implicit exception at runtime. Only the explicit exception is caught on line 294, but the implicit exception of *"runtime_error"* type is not caught [36].

(3) 2 bugs are caused by stack memory issues. One bug is caused by using a local variable in a new thread, and the other bug is caused by excessive recursive calls. Figure 6(c) shows an example bug in *rtab-map*. In the construction function of the class `CoreWrapper`, a local variable `tfDelay` is defined on line 141 in the main thread, but this variable is accessed on line 580 in a new thread created via `new std::thread`. As the new thread cannot access the main thread's stack, a stack-overflow bug occurs [37].

C. Comparison to Existing Approaches

We experimentally compare ROZZ to two state-of-the-art approaches of testing robotic programs, Ros2-fuzz [10] and ASTAA [38]. Ros2-fuzz is an automated fuzzing approach based on AFL [2] to test robotic programs in ROS2. This approach mutates messages for specific topics in a given ROS node. As Ros2-fuzz is open-source, we built it from the source code. ASTAA is a robustness-testing approach for robotic programs. It randomly mutates messages between ROS nodes, and it can also drop several messages at runtime to simulate communication instability. As ASTAA is close-source, we implement an ASTAA-like tool by modifying ROZZ to only enable data mutation of sensor messages and randomly drop messages without using program feedback.

TABLE IV

COMPARISON RESULTS.

Program	Ros2-fuzz		ASTAA-like		ROZZ	
	Branch	*Found bug*	*Branch*	*Found bug*	*Branch*	*Found bug*
nav2_bt_navigator	1.8K	0	23.7K	1	24.3K	3
nav2_planner	1.6K	0	26.4K	2	27.9K	6
nav2_recoveries	4.8K	0	15.1K	1	15.8K	2
nav2_controller	3.1K	0	35.2K	3	37.6K	3
nav2_amcl	0.7K	0	11.9K	2	12.2K	9
Total	12.0K	0	112.3K	9	117.8K	23

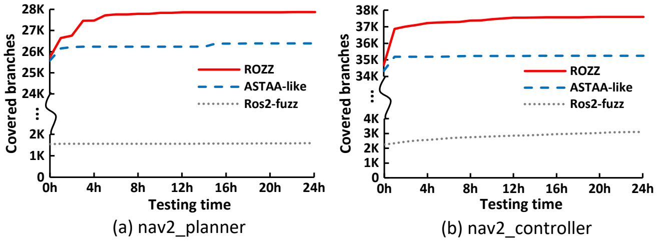
In the experiments, we select five programs whose names contain *"nav2"* in Table I, and run Ros2-fuzz, the ASTAA-like tool and ROZZ to test these programs with a robot navigation task for 24 hours. Table IV shows the comparison results, including covered code branches and found bugs.

ROZZ finds all the 9 bugs found by Ros2-fuzz and the ASTAA-like tool, and it also finds 14 bugs missed by these approaches with higher code coverage. Indeed, Ros2-fuzz and ASTAA generate test cases about only messages between ROS nodes, and thus much code about handling different user data and configuration parameters is not covered during fuzzing. By contrast, ROZZ generates test cases from multiple dimensions (including user data, configuration parameters and sensor messages), and thus ROZZ covers more code missed by Ros2-fuzz and ASTAA. Moreover, ROZZ uses distributed branch coverage to more effectively guide test-case generation for multiple ROS nodes, and performs temporal mutation with three common patterns (ASTAA considers only one of these patterns, namely message dropping) to more effectively cover code about temporal features. For these reasons, ROZZ produces better results than Ros2-fuzz and the ASTAA-like tool in the experiments.

By analyzing the growth of covered branches along with the testing time, we observe that the three tools cover less and less new code branches over time, as many code branches have been covered by the test cases generated from the earlier mutation during fuzzing. Even so, we observe that ROZZ covers more new code branches in the latter tests, thanks to our multi-dimensional generation method, distributed branch coverage and temporal mutation strategy. We randomly select two of the tested ROS programs, namely *nav2_planner* and *nav2_controller*, and show their results in Figure 7.

D. Fuzzing Robotic Programs in ROS1

ROZZ is also applicable to testing robotic programs in ROS1. Thus, we use ROZZ to test three common robotic programs in ROS1 with ASan, including *move_base* [39], *nav1_amcl* [40] and *hector_mapping* [41]. We run *move_base* and *nav1_amcl* for a robot navigation task, and run



(a) nav2_planner

(b) nav2_controller

Fig. 7. Growth of covered code branches with the testing time.

move_base and *hector_mapping* for a map building task. We test each program with its related task for 24 hours. Similar to Section IV-A, we also run official test suites or benchmarks of each program for 24 hours to validate testing-coverage improvement of ROZZ. Table V shows the testing results.

TABLE V

RESULTS OF FUZZING ROBOTIC PROGRAMS IN ROS1.

Program	Covered branch		Found bug		
	Fuzzing	*Test suite*	*InvalidPtr*	*Exception*	*All*
move_base	36.4K	31.9K	0	2	2
nav1_amcl	12.5K	10.9K	2	2	4
hector_mapping	13.1K	11.5K	0	0	0
Total	62.0K	54.3K	2	4	6

Compared to running test suites, ROZZ covers 14% more code branches in the tested programs, benefiting from our fuzzing approach based on ROS properties. Note that we find no bug when running official test suites or benchmarks. Due to the testing-coverage improvement, ROZZ finds 6 real bugs with no false positive, including 2 invalid-pointer accesses and 4 uncaught exceptions. We have reported these bugs to related ROS developers, but we have not received any response. Indeed, the github repositories of the three tested programs have not been updated for a long time.

V. LIMITATION AND FUTURE WORK

ROZZ still has some limitations. First, ROZZ can only test C/C++ programs, as it uses Clang for code instrumention; we plan to extend ROZZ to Python programs. Second, ROZZ is limited in covering special execution situations, such as error handling cases and infrequent thread interleavings; we plan to introduce fault injection [42], [43] and concurrency fuzzing [8], [44] in ROZZ to further improve testing coverage. Finally, ROZZ can only detect memory bugs at present; we plan to apply existing approaches [11], [12] in ROZZ to detect semantic bugs, such as anomalous robotic behaviors.

VI. RELATED WORK

Fuzzing. In practice, fuzzing has been widely used to test infrequently-executed code and detect bugs. Most fuzzing approaches [2]–[9] are coverage-guided, namely they automatically mutate and generate program inputs according to the program feedback of code coverage. Specifically, for a given program input, if it makes the tested program cover new code branches or basic blocks, this input is considered to be an interesting seed, and the fuzzing approach puts it into a seed pool for further test-case mutation and generation. Then, the fuzzing approach selects a seed from the seed pool, and mutates it to generate new inputs as new test cases, making up a basic fuzzing loop. Existing fuzzing approaches focus on improving testing efficiency and effectiveness from seed selection [6], [7], [9], seed mutation [4], [5], [8], etc.

Several recent approaches [10]–[14] explore the application of fuzzing to robotic programs. For example, Delgado et al. [11] propose a fuzzing approach to test robotic behaviors based on SMACH state machines [45]. This approach is grammar-based and generates random values as input keys of state machines, and it monitors executed states to collect logs at runtime. These logs are semi-automatically analyzed to detect the anomalies of robotic behaviors.

However, existing fuzzing approaches are still limited in testing ROS programs, due to neglecting ROS properties, including multi-dimensional inputs, distributed node model and temporal features. To solve this problem, ROZZ considers ROS properties and uses three characteristic techniques to improve fuzzing on ROS programs.

Testing robotic programs. Some approaches exploit unit testing [46], [47], integration testing [48], [49] or mutation testing [38], [50] to check the robustness and reliability of robotic programs. For example, Katz et al. [50] propose a testing approach to detect execution anomalies in robotic programs. This approach generates the inputs based on mutation, monitors program execution and creates clusters of execution profiles to represent nominal execution. If the distances between a given execution profile and these clusters are too large, this execution profile indicates that anomalies can occur. Different from fuzzing, these approaches do not use program feedback to guide test-case generation, and thus they are limited in generating efficient test cases.

Verifying robotic programs. Some approaches [51]–[56] use formal verification to statically check robotic programs. For example, Carvalho et al. [54] propose a model checking approach to verify system-wide safety properties based on message passing for ROS applications, according to given loose specification of the expected behaviors of the individual nodes. Compared to runtime testing, these approaches can achieve higher checking coverage without actually running the program. But they require the user to manually provide detailed specifications, and they spend much time on state exploration when the checked program is large and complex.

VII. CONCLUSION

In this paper, we develop a novel and automated fuzzing framework named ROZZ, to effectively test ROS programs based on ROS properties. ROZZ uses a multi-dimensional generation method to generate test cases of ROS programs from multiple dimensions, a distributed branch coverage to describe the overall code coverage of multiple ROS nodes in the robot task, and a temporal mutation strategy to generate test cases with temporal information. We have used ROZZ to test 10 robotic programs in ROS2, and found 43 real bugs.

ACKNOWLEDGMENT

We thank ROS developers for their feedback to our reports. This work was supported by National Natural Science Foundation of China under Projects 62002195 and 61872210.

REFERENCES

[1] "ROS platform for building robot applications," https://www.ros.org/.

[2] "American Fuzzy Lop," http://lcamtuf.coredump.cx/afl/.

[3] "Syzkaller: a kernel fuzzer," https://github.com/google/syzkaller.

[4] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," *IEEE Transactions on Software Engineering (TSE)*, vol. 45, no. 5, pp. 489–506, 2019.

[5] C. Lemieux and K. Sen, "FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd International Conference on Automated Software Engineering (ASE)*, 2018, pp. 475–485.

[6] S. Pailoor, A. Aday, and S. Jana, "MoonShine: optimizing OS fuzzer seed selection with trace distillation," in *Proceedings of the 27th USENIX Security Symposium*, 2018, pp. 729–743.

[7] A. Herrera, H. Gunadi, S. Magrath, M. Norrish, M. Payer, and A. L. Hosking, "Seed selection for successful fuzzing," in *Proceedings of the 30th International International Symposium on Software Testing and Analysis (ISSTA)*, 2021, pp. 230–243.

[8] H. Chen, S. Guo, Y. Xue, Y. Sui, C. Zhang, Y. Li, H. Wang, and Y. Liu, "MUZZ: thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs," in *Proceedings of the 29th USENIX Security Symposium*, 2020, pp. 2325–2342.

[9] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing," in *Proceedings of the 23rd USENIX Security Symposium*, 2014, pp. 861–875.

[10] "Ros2-fuzz: automatic fuzzing for ROS2," https://github.com/rosin-project/ros2_fuzz.

[11] R. Delgado, M. Campusano, and A. Bergel, "Fuzz testing in behavior-based robotics," in *Proceedings of the 2021 International Conference on Robotics and Automation (ICRA)*, 2021, pp. 9375–9381.

[12] T. Woodlief, S. Elbaum, and K. Sullivan, "Fuzzing mobile robot environments for fast automated crash detection," in *Proceedings of the 2021 International Conference on Robotics and Automation (ICRA)*, 2021, pp. 5417–5423.

[13] C. Wang, Y. C. Tok, R. Poolat, S. Chattopadhyay, and M. R. Elara, "How to secure autonomous mobile robots? an approach with fuzzing, detection and mitigation," *Journal of Systems Architecture (JSA)*, vol. 112, p. 101838, 2021.

[14] S. Rivera, A. K. Iannillo, *et al.*, "DiscoFuzzer: discontinuity-based vulnerability detector for robotic systems," *TechRxiv*, 2020.

[15] "LLVM compiler infrastructure," https://llvm.org/.

[16] "Topics in ROS," http://wiki.ros.org/ROS/Technical%20Overview.

[17] "Fuzzing raft for fun and publication," https://colin-scott.github.io/blog/2015/10/07/fuzzing-raft-for-fun-and-profit/.

[18] C. Scott, V. Brajkovic, G. Necula, A. Krishnamurthy, and S. Shenker, "Minimizing faulty executions of distributed systems," in *Proceedings of the 13th International Symposium on Networked Systems Design and Implementation (NSDI)*, 2016, pp. 291–309.

[19] "Clang compiler," https://clang.llvm.org/.

[20] "ASan: address sanitizer," https://github.com/google/sanitizers/wiki/AddressSanitizer.

[21] "MSan: memory sanitizer," https://github.com/google/sanitizers/wiki/MemorySanitizer.

[22] "UBSan: undefined behavior sanitizer," https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[23] "CLOC: counting code lines," https://github.com/AlDanial/cloc.

[24] "Gazebo: a robot simulation framework," http://gazebosim.org/.

[25] "Bt navigator in ROS2 navigation framework," https://github.com/ros-planning/navigation2/tree/main/nav2_bt_navigator.

[26] "Planner in ROS2 navigation framework," https://github.com/ros-planning/navigation2/tree/main/nav2_planner.

[27] "Recoveries in ROS2 navigation framework," https://github.com/ros-planning/navigation2/tree/main/nav2_recoveries.

[28] "Controller in ROS2 navigation framework," https://github.com/ros-planning/navigation2/tree/main/nav2_controller.

[29] "AMCL localization in ROS2 navigation framework," https://github.com/ros-planning/navigation2/tree/main/nav2_amcl.

[30] "LaMa: alternative localization and mapping for ROS," https://github.com/iris-ua/iris_lama_ros.

[31] "Kalman filter based localization for ROS," https://github.com/rsasaki0109/kalman_filter_localization.

[32] "RTAB-Map: an application of real-time appearance-based mapping," https://github.com/introlab/rtabmap_ros.

[33] "Slam Toolbox: a set of tools and capabilities for 2D SLAM," https://github.com/SteveMacenski/slam_toolbox.

[34] "Cartographer: a system for real-time 2D and 3D SLAM across multiple platforms and sensor configurations." https://github.com/cartographer-project/cartographer_ros.

[35] "Use-after-free bug in nav2_planner," https://github.com/ros-planning/navigation2/issues/2507.

[36] "Uncaught exception bug in nav2_controller," https://github.com/ros-planning/navigation2/issues/2506.

[37] "Stack-overflow bug for new thread in rtab-map," https://github.com/introlab/rtabmap_ros/pull/632.

[38] C. Hutchison, M. Zizyte, P. E. Lanigan, D. Guttendorf, M. Wagner, C. Le Goues, and P. Koopman, "Robustness testing of autonomy software," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 2018, pp. 276–285.

[39] "move_base in ROS1: a package for route planning and movement control for the robot," https://github.com/ros-planning/navigation/tree/noetic-devel/move_base.

[40] "AMCL localization in ROS1 navigation framework," https://github.com/ros-planning/navigation/tree/noetic-devel/amcl.

[41] "hector_mapping in ROS1: a scalable SLAM approach for the robot," https://github.com/tu-darmstadt-ros-pkg/hector_slam.

[42] Z.-M. Jiang, J.-J. Bai, K. Lu, and S.-M. Hu, "Fuzzing error handling code using context-sensitive software fault injection," in *Proceedings of the 29th USENIX Security Symposium*, 2020, pp. 2595–2612.

[43] K. Cong, L. Lei, Z. Yang, and F. Xie, "Automatic fault injection for driver robustness testing," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*, 2015, pp. 361–372.

[44] Z.-M. Jiang, J.-J. Bai, K. Lu, and S.-M. Hu, "Context-sensitive and directional concurrency fuzzing for data-race detection," in *Proceedings of the 29th Network and Distributed System Security Symposium (NDSS)*, 2022.

[45] J. Bohren and S. Cousins, "The SMACH high-level executive]," *IEEE Robotics and Automation Magazine*, vol. 17, no. 4, pp. 18–20, 2010.

[46] G. Breitenhuber, "Towards application level testing of ROS networks," in *Proceedings of the 4th International Conference on Robotic Computing (IRC)*, 2020, pp. 436–442.

[47] A. Bihlmaier and H. Wörn, "Robot unit testing," in *Proceedings of the 2014 International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, 2014, pp. 255–266.

[48] M. A. Brito, S. R. Souza, and P. S. Souza, "Integration testing for robotic systems," *Software Quality Journal (SQJ)*, pp. 1–33, 2020.

[49] J. Ernits, E. Halling, G. Kanter, and J. Vain, "Model-based integration testing of ROS packages: a mobile robot case study," in *Proceedings of the 2015 European Conference on Mobile Robots (ECMR)*, 2015, pp. 1–7.

[50] D. S. Katz, C. Hutchison, M. Zizyte, and C. Le Goues, "Detecting execution anomalies as an oracle for autonomy software robustness," in *Proceedings of the 2020 International Conference on Robotics and Automation (ICRA)*, 2020, pp. 9366–9373.

[51] R. Halder, J. Proença, N. Macedo, and A. Santos, "Formal verification of ROS-based robotic applications using timed-automata," in *Proceedings of the 5th International Workshop on Formal Methods in Software Engineering*, 2017, pp. 44–50.

[52] M. Foughali, B. Berthomieu, S. Dal Zilio, P.-E. Hladik, F. Ingrand, and A. Mallet, "Formal verification of complex robotic systems on resource-constrained platforms," in *Proceedings of the 6th International Workshop on Formal Methods in Software Engineering (FormaliSE)*, 2018, pp. 2–9.

[53] J. Krook, L. Svensson, Y. Li, L. Feng, and M. Fabian, "Design and formal verification of a safe stop supervisor for an automated vehicle," in *Proceedings of the 2019 International Conference on Robotics and Automation (ICRA)*, 2019, pp. 5607–5613.

[54] R. Carvalho, A. Cunha, N. Macedo, and A. Santos, "Verification of system-wide safety properties of ROS applications," in *Proceedings of the 2020 International Conference on Intelligent Robots and Systems (IROS)*, 2020, pp. 7249–7254.

[55] S. Kortik and T. Shastha, "Formal verification of ros based systems using a linear logic theorem prover," in *Proceedings of the 2021 International Conference on Robotics and Automation (ICRA)*, 2021, pp. 9368–9374.

[56] A. Santos, A. Cunha, N. Macedo, and C. Lourenço, "A framework for quality assessment of ROS repositories," in *Proceedings of the 2016 International Conference on Intelligent Robots and Systems (IROS)*, 2016, pp. 4491–4496.