Reinforcement Learning-Based Fuzz Testing for the Gazebo Robotic Simulator

ZHILEI REN*, Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province; School of Software, Dalian University of Technology, China
YITAO LI, School of Software, Dalian University of Technology, China
XIAOCHEN LI, School of Software, Dalian University of Technology, China
GUANXIAO QI, School of Software, Dalian University of Technology, China
JIFENG XUAN, School of Computer Science, Wuhan University, China
HE JIANG*, Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province; School of Software, Dalian University of Technology; DUT Artificial Intelligence Institute, China

Gazebo, being the most widely utilized simulator in robotics, plays a pivotal role in developing and testing robotic systems. Given its impact on the safety and reliability of robotic operations, early bug detection is critical. However, due to the challenges of strict input structures and vast state space, it is not effective to directly use existing fuzz testing approach to Gazebo.

In this paper, we present GzFuzz, the first fuzz testing framework designed for Gazebo. GzFuzz addresses these challenges through a syntax-aware feasible command generation mechanism to handle strict input requirements, and a reinforcement learning-based command generator selection mechanism to efficiently explore the state space. By combining the two mechanisms under a unified framework, GzFuzz is able to detect bugs in Gazebo effectively. In extensive experiments, GzFuzz is able to detect an average of 9.6 unique bugs in 12 hours, and exhibits a substantial increase in code coverage than existing fuzzers AFL++ and Fuzzotron, with a proportionate improvement of approximately 239%–363%. In less than six months, GzFuzz uncovered 25 unique crashes in Gazebo, 24 of which have been fixed or confirmed. Our results highlight the importance of directly fuzzing Gazebo, thereby presenting a novel and potent methodology that serves as an inspiration for enhancing testing across a broader range of simulators.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Robotic Simulator, Software Testing, Fuzz Testing, Gazebo

ACM Reference Format:
Zhilei Ren, Yitao Li, Xiaochen Li, Guanxiao Qi, Jifeng Xuan, and He Jiang. 2025. Reinforcement Learning-Based Fuzz Testing for the Gazebo Robotic Simulator. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA065 (July 2025), 22 pages. https://doi.org/10.1145/3728942

*Corresponding authors

Authors' Contact Information: Zhilei Ren, Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province; School of Software, Dalian University of Technology, Dalian, China, zren@dlut.edu.cn; Yitao Li, School of Software, Dalian University of Technology, Dalian, China, liyitao@mail.dlut.edu.cn; Xiaochen Li, School of Software, Dalian University of Technology, Dalian, China, xiaochen.li@dlut.edu.cn; Guanxiao Qi, School of Software, Dalian University of Technology, Dalian, China, gxqi@mail.dlut.edu.cn; Jifeng Xuan, School of Computer Science, Wuhan University, Wuhan, China, jxuan@whu.edu.cn; He Jiang, Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province; School of Software, Dalian University of Technology; DUT Artificial Intelligence Institute, Dalian, China, jianghe@dlut.edu.cn.

ACM 2994-970X/2025/7-ARTISSTA065
https://doi.org/10.1145/3728942

1 Introduction

As the most widely-used robotic simulator [3, 28, 44, 52], Gazebo provides a powerful and flexible platform for developing, testing, and validating robotic systems in high-fidelity 3D environments [5, 35]. Its role as the default simulator for Robot Operating System 2 (ROS2) [36], the de-facto standard in robotic middleware, further solidifies its importance. By seamlessly integrating with ROS2, Gazebo allows developers to simulate complex robot behaviors, sensor data, and environmental interactions, accelerating the development cycle and reducing the need for physical prototypes [20]. This makes Gazebo an indispensable tool for both research and industry, with applications spanning a wide range of fields, such as NASA space missions [8, 31, 53, 61], DARPA subterranean exploration [21, 25], unmanned maritime vehicle control [14, 37], and autonomous driving [1, 6].

Like other complex software systems, Gazebo is prone to bugs that can have serious consequences, impacting robotic development through simulation inaccuracies, testing failures, and crashes [4]. Inaccuracies in the physics engine or sensor models may cause simulated behavior to deviate from real-world conditions, leading to algorithm failures during deployment and increased development costs [58]. Testing failures, such as incomplete coverage or incorrect results, allow critical bugs to go undetected. Crashes further disrupt simulations, halt workflows, and obscure important bugs, potentially resulting in real-world operational failures and safety risks. For instance, in digital twin systems, simulator crashes can disrupt synchronization between virtual and physical models, causing unexpected downtime or operational issues [2, 22, 32, 46].

In software engineering, fuzz testing is a key method for ensuring software quality by automatically generating and injecting random or malformed inputs to uncover vulnerabilities. Tools like AFL/AFL++ [27], AFLNet [49], and Fuzzotron[1] have proven highly effective in detecting flaws in software and network services. However, directly transferring these existing fuzzing methods to Gazebo may not yield satisfying results. The reason lies in the fact that Gazebo operates with intricate input formats, and a sophisticated plugin-based client-server architecture to realize real-time physics computation as illustrated in Figure 1. These characteristics make it difficult for standard fuzzers to effectively explore the range of potential bugs in Gazebo. In particular, we have identified two major challenges, which are described as follows:

- **Strict Input Challenge**: Gazebo operates with specific and structured command interface, where inputs must follow a strict syntax and semantic rules to interact with the simulation [35]. For instance, a command to spawn a robot must include a valid robot model, positioning data, and other parameters like physics properties. If a fuzzing tool generates random or malformed input that does not follow this structure, e.g., an incomplete command or an invalid robot model, the input will be immediately rejected by Gazebo's parser, leading to meaningless exploration of the system.
- **State Space Challenge**: The simulation process of Gazebo may have highly complex and vast state spaces, involving different models, joints, sensors, and physics interactions using plugins [4]. This challenge refers to the difficulty of systematically exploring all possible states of the simulation. In Gazebo, specific command or sequences of commands may trigger hidden or uncommon states, such as a robot malfunctioning when interacting with a particular object or sensor failing under certain conditions. Random fuzzing is unlikely to efficiently reach these deeper, less obvious states, meaning many potential bugs could remain undiscovered.

To tackle these challenges, in this paper, we propose the first **GaZebo fuzz**ing approach GzFuzz. On the one hand, to tackle the strict input challenge, GzFuzz leverages **Syntax-aware Feasible**

[1] https://github.com/denandz/fuzzotron

Command Generation mechanism to ensure that the fuzzed inputs, specifically Gazebo commands, are syntactically valid within the simulation environment. By incorporating syntax-aware command structures and constraints into the input generation process, GzFuzz is able to create feasible commands that adhere to Gazebo's strict input requirements. This not only ensures that the fuzzed inputs are accepted by the system, but also helps to explore deeper and more relevant parts of the simulation's behavior, effectively increasing the chances of triggering subtle bugs. On the other hand, to address the state space challenge, GzFuzz employs a **Learning-based Command Generator Selection** mechanism that guides the fuzzing process based on multi-dimensional feedback. By using reinforcement learning, GzFuzz learns which types of commands or sequences of commands are more likely to explore untested or under-explored states within Gazebo's simulation environment. This adaptive approach allows GzFuzz to prioritize command sequences that maximize code coverage, systematically pushing the boundaries of the system's state space and improving the likelihood of uncovering bugs. Together, these two components enable GzFuzz to effectively meet Gazebo's complex input requirements, and explore the vast state space, making it a powerful tool for identifying bugs in sophisticated robotic simulations.

To evaluate the effectiveness of GzFuzz, extensive experiments are conducted, to investigate the proposed approach from multiple perspectives. GzFuzz is able to detect an average of 9.6 bugs in 12 hours over the previous release of Gazebo, significantly outperforming existing general-purpose fuzzers AFL++ and Fuzzotron, and exhibits a substantial increase in code coverage, with a proportionate improvement of 239%–363%. By comparing GzFuzz with its variants, the contribution of each proposed mechanism is further confirmed. As a unified framework, GzFuzz uncovered 25 unique crashes in Gazebo in less than six months, 24 of which have been fixed or confirmed.

The contributions of this paper are listed as follows:

- To the best of our knowledge, this is the first study to apply fuzz testing to Gazebo. We have engineered a set of syntax-aware feasible command generator, designed to address the strict input constraints. Additionally, we introduce a reinforcement learning-based command generator selection strategy to efficiently explore the vast state space of Gazebo, tackling the state space exploration challenge.
- Extensive experiments validate the effectiveness of the proposed approach, demonstrating its ability to detect critical issues in Gazebo. Within less than six months, GzFuzz has successfully detected 25 unique crashes in the most widely-used simulator Gazebo, with 24 of these bugs being fixed or confirmed. Our results present a novel and potent methodology that serves as an inspiration for enhancing testing and quality assurance across a broader range of robotic simulators.
- A fully functional prototype of GzFuzz has been implemented, and the replication package is available at https://github.com/liyitao-code/GzFuzz for further validation.

The rest of this paper is organized as follows. Section 2 introduces the background and motivation of this paper. Section 3 presents the details of GzFuzz. In Section 4, we conduct extensive experiments from multiple perspectives to evaluate GzFuzz. Section 5 and 6 describe the threats to validity and related work, respectively. Finally, Section 7 concludes this paper and future work.

2 Background and Motivation

As illustrated in Figure 1, Gazebo operates in a client-server architecture, where the server manages all core simulation tasks, such as physics computations, scene broadcasting, and entity component management, while the client interacts with the server either through direct user inputs or external processes. In a typical simulation task, an input file is first loaded (step 1). Then, the client communicates with the server via well-defined interfaces, which can be triggered through the GUI (steps

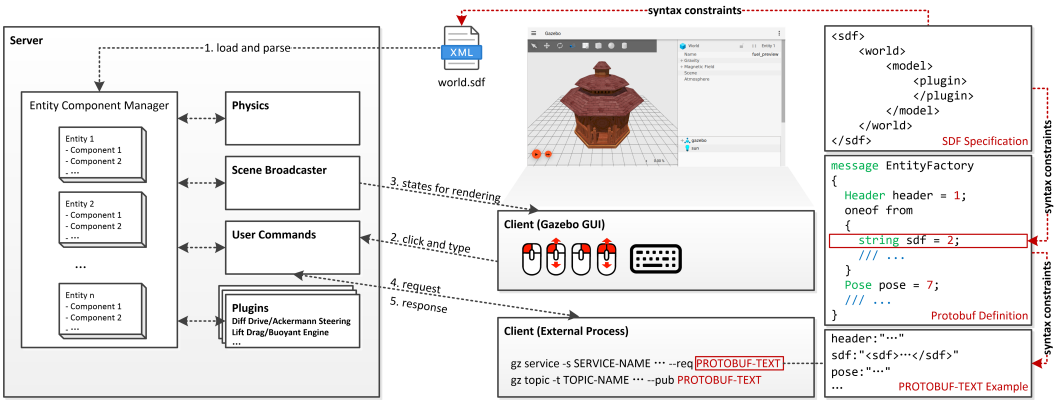


Fig. 1. Architecture of Gazebo

2–3) or through external commands issued by an external process (steps 4–5). This separation allows the server to handle complex simulation workloads while the client focuses on interaction and control. However, testing in this architecture poses several challenges, particularly due to the strict syntax requirements of the inputs and the complex state space of the server.

Using the GUI of Gazebo with mouse and keyboard inputs provides limited control and shallow testing of the simulation, since GUI-driven random inputs often miss deeper system behaviors, making comprehensive testing difficult [42, 56]. To achieve better coverage, it is essential to communicate directly with the server through external commands with services and topics. For the two communication mechanism, services facilitate synchronous tasks such as adding/removing models and plugins, while topics manage asynchronous data exchange for sensor streaming and control commands, enabling real-time interaction and thorough system exploration.

The inputs to the server of Gazebo come in two forms: Simulation Description Format[2] (SDF) files and external commands. As shown in Figure 1, SDF files use XML-based syntax to define objects and environments for robot simulation, visualization, and control, while external commands utilize Protobuf specifications to communicate with the server. Table 1 provides a few examples of Gazebo services and topics. There are over 50 built-in services and more than 10 topics, each requiring messages in different Protobuf formats depending on the service or topic. Additionally, as illustrated in Figure 1, certain Protobuf specifications, such as the sdf field in EntityFactory used for dynamically inserting models, depends on the SDF syntax. This forms the basis of the **Strict Input Challenge**, where any fuzzing technique must generate inputs that strictly adhere to Gazebo's input formats. If the inputs are malformed or invalid, they will be rejected, preventing meaningful interactions with the server and further system exploration.

Furthermore, the server side of Gazebo has a highly complex architecture, involving numerous interacting components, such as the entity component manager, physics engine, and various plugins. In particular, Gazebo includes over 50 built-in plugins that provide various functionalities such as sensor simulation, actuator control, and environmental effects. Examples include plugins for controlling robots, simulating camera and LiDAR sensors, and interacting with the environment. The plugins and other components work together to simulate real-world interactions, but their complexity gives rise to a massive state space. For instance, an entity in the simulation might interact with different physical environments or respond to different external commands, leading

[2] http://sdformat.org/spec

Table 1. Examples of Gazebo services and topics

Service	Type	Topic	Type
/gui/camera/view_control	Built-in	/world/. . . /state	Built-in
/server_control	Built-in	/world/. . . /light_config	Built-in
/world/. . . /create	Built-in	/world/. . . /pose/info	Built-in
/world/. . . /remove	Built-in	/gui/camera/pose	Built-in
/world/. . . /set_pose	Built-in	/world/. . . /scene/info	Built-in
/camera/record_video	Plugin	/model/. . . /buoyancy_engine	Plugin
/optical_tactile_plugin/enable	Plugin	/world/. . . /wind	Plugin

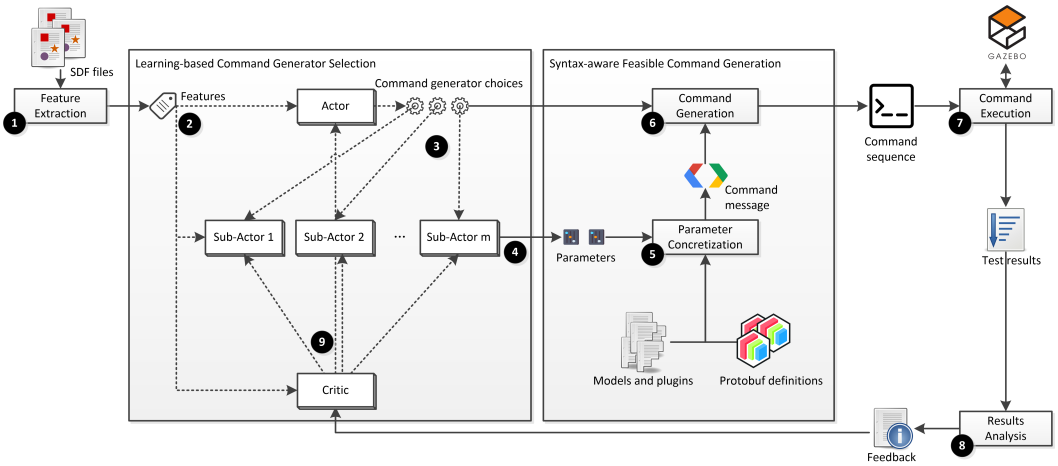


Fig. 2. Workflow of GzFuzz

to a variety of outcomes. The **State Space Challenge** arises from the difficulty in exploring this vast range of possible states through testing. Without proper guidance, randomly generated inputs are unlikely to push the system into deeper, more intricate states, making it difficult to uncover edge-case bugs.

As a brief summary, the client-server architecture of Gazebo, coupled with its strict input format requirements and vast state space, makes comprehensive testing a significant challenge. These challenges underscore the necessity for a specialized fuzzing approach that can handle the strict input validation and efficiently explore the intricate state space of Gazebo. This is the motivation behind developing tools like GzFuzz, which aim to overcome these obstacles, and improve the overall robustness and reliability of Gazebo.

3 Approach

In this section, we describe the details of the proposed GzFuzz framework. The workflow of GzFuzz is presented in Figure 2. Given a SDF file, the fuzz testing process works as follows: ❶ The features of the SDF file is extracted. ❷–❹ A sequence of command generator choices are produced with a reinforcement learning framework. ❺–❻ Based on the choices, Gazebo commands are constructed as client command-line strings, with the help of the mined models, plugins, and Protobuf definitions. ❼ A Gazebo server process is spawned, and the generated commands are

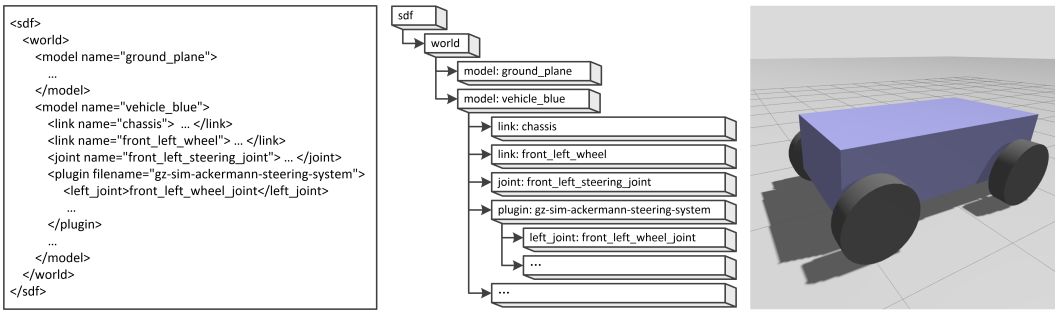


Fig. 3. Example of SDF file, DOM tree, and rendered scenario

executed to communicate with the server. After the execution, we are able to collect the test results. ❽–❾ The test results are analyzed, and the reinforcement learning modules are updated accordingly.

As for the test oracle, we mainly focus on crash-related bugs due to their severity, clarity, and reproducibility [62]. Crashes serve as clear indicators of critical faults, provide unambiguous signals for detecting failures, and are generally easier to reproduce and diagnose. This makes crash-related bugs an effective focus for improving the stability and reliability of Gazebo.

3.1 Preprocessing

To realize the fuzz testing process, the initial step involves collecting a set of seed SDF files. In the preprocessing phase, we focus on extracting relevant SDF files, models, and plugins from the Gazebo source repository for command generation and fuzz testing.

As described in Section 2, Gazebo accepts SDF files as input to define the simulation environment and objects. For this purpose, we first traverse the source repository of Gazebo, and extract a total of 309 SDF files for analysis and fuzzing purposes. Then, we parse all the SDF files into document-object model (DOM) trees using the `lxml`[3] library. For each DOM tree, we traverse the DOM tree, and those nodes with the `model` tag that contain a `plugin` child node are extracted. There are 123 models with plugins in this study. Each model is then labeled with a unique ID. During the following command generation phase, the model ID is used as a parameter to indicate the corresponding model content. The plugin extraction process follows a similar approach to model extraction, using the constructed DOM trees as in the previous step. More specifically, Gazebo distinguishes between two types of plugins: those attached to the `world` node and those attached to the `model` node. We extract both types of plugins from the SDF files, and have collected 117 plugins in total.

In Figure 3, we present an example of SDF file, its parsed DOM tree, as well as the rendered scenario in Gazebo. In the SDF file, the `<world>` node encapsulates the entire environment. Inside the world, `<model>` nodes represent individual objects or robots, which are made up of `<link>` nodes, defining the physical components like the chassis or wheels. These links are interconnected by `<joint>` nodes, allowing for mechanical articulation. Additionally, `<plugin>` nodes provide the ability to inject custom functionality, such as `gz-sim-ackermann-steering-system` for controlling vehicle steering.

As a brief summary, the preprocessing step ensures that we have a diverse set of models and plugins available for generating meaningful and valid commands during the fuzzing process. In this study, we consider fuzzing Gazebo with external commands instead of directly mutating input

[3] https://pypi.org/project/lxml

SDF files, due to the following considerations. First, fuzzing via external commands allows for more dynamic interaction with the server of Gazebo, mimicking real-world usage scenarios in an automated paradigm. Unlike static SDF files, which are loaded at simulation startup, services and topics provide a continuous, real-time means to manipulate the simulation, leading to more immediate and varied feedback. Second, fuzzing the SDF files can also be achieved by invoking the `/world/.../create` service, which allows to pass in diverse SDF content during the simulation. This method combines the benefits of SDF fuzzing with dynamic command-based interaction.

3.2 Syntax-aware Feasible Command Generation

As mentioned in Section 2, it is essential to ensure that the generated commands are syntactically valid within the simulation environment. To achieve this, we introduce a set of syntax-aware command generation approach tailored to the strict input requirements.

More specifically, we categorize the Gazebo commands into three distinct groups based on their characteristics and intended testing objectives, i.e., randomly generated commands, semi-realistic commands, and disruptive commands. Each category serves a specific purpose in uncovering different types of issues within Gazebo.

- Randomly generated commands: These commands are automatically generated to conform to Gazebo's service and topic requirements in Protobuf format. The goal is to explore a wide input space with syntactically valid but unpredictable messages, helping uncover edge cases and potential system crashes. Taking random service as an example, the commands are generated as follows. First, the candidate services are obtained with command `gz service -l`. Then, a service name *SN* is randomly selected, and the required message type is retrieved by `gz service -s` *SN* `-i`. With the message type, the message could be generated with a third-party library `randomproto`[4].
- Semi-realistic commands: This category mimics typical user actions, such as adding or deleting models, attaching plugins, or moving objects within the simulation. For example, to generate a command for inserting a model with a plugin, we first prepare the candidate models during preprocessing. For example, we could invoke the `/world/. . . /create` service and pass in the SDF model, to handle the actual insertion of the model with plugin. Similarly, to generate a command for inserting primitive model without plugin, we leverage the `libsdformat`[5], and implement the construction of links, collisions, and joints.
- Disruptive commands: Building on semi-realistic commands, disruptive commands introduce perturbations, such as modifying plugin parameters or issuing invalid operations (e.g., randomly mutating a leaf node with a potentially invalid value of a plugin DOM tree).

In Table 2, we summarize the command generators engineered in this study. The proposed generators are carefully designed: Generators 1–2 could cover all available services and topics with randomly generated commands. Generators 3–7 simulate realistic use cases to test how Gazebo performs under normal conditions. Generators 8–10 intend to stress-test Gazebo's error handling and ability to deal with extreme or incorrect inputs. Notably, generators 6 through 9 require additional parameters to generate their corresponding commands. For example, generator 6 is used to insert a model with a plugin, where the models are mined from the Gazebo source repository. With over 100 models available, selecting a model is the first step. We have to further determine the appropriate parameters to generate the final, concrete Gazebo commands. Besides, for the other generators, we consider randomly selecting services, topics, and models because the number for these choices may vary across scenarios. Hence, these generators do not further require parameters.

[4] https://pypi.org/project/randomproto
[5] https://github.com/gazebosim/sdformat

Table 2. Summarization of the proposed generators

	Description	Parameter	Category
1	Randomly call a service	None	Random
2	Randomly call a topic	None	Random
3	Insert a randomly generated primitive model	None	Semi-realistic
4	Randomly remove a model	None	Semi-realistic
5	Randomly move the position of a model	None	Semi-realistic
6	Insert a model with plugin	Model ID	Semi-realistic
7	Add a plugin to a random model	Plugin ID	Semi-realistic
8	Insert a model with plugin, with mutated value	Model ID	Disruptive
9	Add a plugin with mutated value to a random model	Plugin ID	Disruptive
10	Insert a primitive model, with mutated value	None	Disruptive

Moreover, we should note that the distinctions among the three categories of generators are not absolute. For instance, in generator 7, certain plugins may require their configurations to be adjusted with respect to the models to function properly. Otherwise, the resulting behavior may resemble that of generator 9. The sole difference between these two generators is whether their parameters are deliberately subjected to random mutation.

3.3 Learning-based Command Generator Selection

As mentioned in the previous subsection, the command generator may require parameters, which are essentially a further decision after the generators are selected. Since fuzz testing is a typical iterative process, at iteration t, the command generator sequence is represented as $c_t = \{\langle c_{t1}, p_{t1}\rangle, \langle c_{t2}, p_{t2}\rangle, \ldots, \langle c_{tn_c}, p_{tn_c}\rangle\}$, where c_{ti} denotes the generator index of the i-th command generator, p_{ti} denotes the parameter index of the i-th generator, and n_c is the length of the sequence. If generator c_{ti} does not require a parameter, then $p_{ti} = None$. Moreover, each sequence c is companioned with a vector V, where each element v_i represents the occurrence count of the i-th command generator in c.

Due to the requirements of selecting both command generators and their parameters, we employ hierarchical reinforcement learning (HRL) [10, 48] for fuzzing `Gazebo` due to its ability to effectively manage complex decision-making processes and explore large state spaces. In the context of fuzzing, `Gazebo` presents a multifaceted environment with parameters and potential interactions, making it challenging to generate effective input commands. HRL allows us to decompose this complexity into a hierarchy of simpler, manageable subtasks.

Inspired by HRL, we can train specialized agents for different levels of abstraction, enabling them to focus on specific aspects of the simulation, such as generating commands and filling in parameters. This structured approach not only enhances the efficiency of `Gazebo` command generation, but also improves the overall exploration of the state space.

In this study, the command generator selection consists of three components, i.e., an actor module, a critic module, and a set of sub-actor modules. More specifically, we employ a two-level decision-making strategy: an actor for selecting command generators and a set of sub-actors for choosing parameters, all guided by a shared critic module. Following existing studies [15, 16], each module is implemented using fully connected neural networks, with the input layer receiving a vector of features, followed by a hidden layer of N neurons. The actor/sub-actors output the

probability distributions over generators or parameters, while the critic predicts a scalar reward value of the current input.

For the actor, the sub-actors, and the critic modules, the input is a vector of features extracted from the SDF file, which primarily characterize the input state by capturing various entity statistics within the Gazebo simulation scene. These numeric features include, for example, the number of models without plugins, models with plugins, plugins within models, and plugins under the world node. Next, we proceed to describe the network structures of the actor, sub-actors, and the critic modules. For each module, given an input vector s, the hidden layer with N neurons uses a weight matrix W_1 and bias vector b_1. The hidden layer output is calculated as:

$$h = ReLU(W_1 \cdot s + b_1). \tag{1}$$

From the hidden layer to the output, the actor generates a one-dimensional vector of length m, where m represents the number of available generator choices. This vector serves as a probability distribution for selecting each generator based on the input features s. The weight matrix W_2 has dimensions $m \times N$, and the bias vector b_2 has dimension m. The output is calculated as:

$$y_{actor} = Softmax(W_2 \cdot h + b_2). \tag{2}$$

The sub-actors work similarly as the actor, except that they are for the parameters of the generators. For the critic module, the output represents the estimated reward of the current iteration, using a weight matrix W_3 and bias scalar b_3. The output is:

$$y_{critic} = W_3 \cdot h + b_3. \tag{3}$$

After the sequence of commands are executed, the actors and the critic modules are to be updated. More detailed, the total reward is the sum of three components:

- Crash reward is designed to encourage finding diverse crashing sequences [16]:

$$R^t_{crash} = \omega \times n_t, \tag{4}$$

  where $\omega > 0$ is a parameter, and n_t represents the number of times the same crash type has been triggered up to iteration t.
- Coverage reward encourages increased code coverage:

$$R^t_{cov} = \begin{cases} \lambda, & \text{coverage increases,} \\ 0, & \text{otherwise,} \end{cases} \tag{5}$$

  where $\lambda > 0$ is a parameter rewarding coverage improvement.
- Diversity reward is used to encourage diversity of the command generator sequences [16], the diversity div_t of an command generator sequence c_t is measured by comparing it with the h most recent sequences:

$$div_t = \frac{1}{|C_h|} \sum_{c_i \in C_h} Dist(c_i, c_t), \tag{6}$$

  where $C_h = \{c_{t-h}, \ldots, c_{t-1}\}$ is a set of command generator sequences closest to the current sequence c_t, with $h = 100$ in this study. $Dist(c_i, c_t) = 1 - cosine(V_i, V_t)$, where V_i and V_t are the vectors of c_i and c_t, respectively, and $cosine(V_i, V_t)$ denotes their cosine similarity. The diversity reward for the current command generator sequence c_t is defined as:

$$R^t_{div} = \frac{1}{h} \sum_{i=1}^{h} (div_t - div_{t-i}). \tag{7}$$

Algorithm 1: GzFuzz

```
Input: SDF files F
Output: Detected Bugs B
begin
    B ← ∅;
    while stopping criteria not met do
        f ← RandomChoice(F) ;
        s ← ExtractFeature(f) ;
        seq ← Actor(s) ;              // Obtain command generator sequence with the actor module
        c ← EmptySequence() ;
        for g ∈ seq do
            if RequiresParameter(g) then
                index ← Sub-ActorIndex(g) ;
                param ← Sub-Actor(index, s) ;          // Obtain parameter with sub-actor modules
                c.append(⟨g, param⟩) ;
            else
                c.append(⟨g, None⟩) ;
            end
        end
        commands ← GenerateCommands(c) ;
        results ← ExecuteCommands(commands) ;
        if ExistCrash(results) then
            B ← B ∪ results ;
        end
        feedback ← AnalyzeResults(results) ;          // Calculate rewards with Equations 4–8
        Update(Critic, Actor, Sub-Actor, feedback) ;   // Update networks with Equations 9–13
    end
    return B ;
end
```

The final reward is:

$$Reward(t) = R^t_{crash} + R^t_{cov} + R^t_{div}. \tag{8}$$

After obtaining the actual reward, GzFuzz further uses the critic module to obtain the predicted potential reward. Inspired by existing studies [15, 16], we consider an advantage loss function in order to reduce the high variance of the neural networks. More specifically, for the critic module, the loss is calculated using the mean squared error:

$$Advantage(t) = Reward(t) - y_{critic}, \tag{9}$$

$$ValueLoss(t) = Advantage(t)^2, \tag{10}$$

where $Reward(t)$ is the actual reward, and y_{critic} is the reward predicted by the critic module at iteration t. For the actor and the sub-actors, the loss is updated using the policy gradient method:

$$PolicyLoss(t) = -\log(P_\theta(s, c_t)) \cdot Advantage(t), \tag{11}$$

where $P_\theta(s, c_t)$ is the transition probability of choosing c_t under the input feature s, which is predicted by the corresponding actor/sub-actor under network parameters θ.

Both the actor/sub-actors and the critic modules use back-propagation to update parameters with *PolicyLoss* and *ValueLoss*, aiming to maximize cumulative rewards. Note that for the sub-actors, only those corresponding to the selected command generators are updated:

$$\theta = UpdateParameter(\theta, PolicyLoss(t), \alpha), \tag{12}$$

```
<model name='vehicle_blue'>
    <pose>0 2 0.325 0 -0 0</pose>
    ...
    <plugin filename="gz-sim-ackermann-steering-system" name="gz::sim::systems::AckermannSteering">
        <left_joint>front_left_wheel_joint</left_joint>
        <left_joint>rear_left_wheel_joint</left_joint>
        <right_joint>front_right_wheel_joint</right_joint>
        ...
    </plugin>
</model>
```

(a) Snippet for the SDF model

```
gz service -s /world/world_0/create \
--reptype gz.msgs.Boolean \
--reqtype gz.msgs.EntityFactory \
--req 'sdf: "<sdf><model name=\"vehicle_blue\">...</sdf>"
pose {
  position {
    x: 1.697057974471674
    y: -5.807148102869744
    z: 0.23030063614806195
  }
}
name: "model"
allow_renaming: true'
```

(b) Command for creating model

```
gz service -s /world/world_0/control \
--reptype gz.msgs.Boolean \
--reqtype gz.msgs.WorldControl \
--req 'pause: true
step: true
multi_step: 3324019261
reset {
  all: true
  time_only: true
  model_only: true
}
seed: 4099918180
run_to_sim_time {
  sec: 1846651340
  nsec: 4481
}'
```

(c) Command for resetting simulation

Fig. 4. Bug #2465: Crash when calling WorldControl service over model with AckermannSteering plugin

$$\phi = UpdateParameter(\phi, ValueLoss(t), \alpha), \tag{13}$$

where θ and ϕ indicate the parameters of the actor/subactors and the critic modules, respectively. In this study, we utilize the Adam optimizer to update the parameters [34], and α indicates the initial learning rates for the neural network modules.

After presenting each component, we present the pseudo code of GzFuzz in Algorithm 1. The proposed approach works in an iterative paradigm. The detected bug set B is firstly initialized as empty, and the fuzzing proceeds to the main loop. For each iteration, a random SDF file is sampled from the input set (line 4). Then, the features are extracted from this file (line 5), and passed to the actor module, which produces a sequence of abstract commands, indicated by a sequence of command generator choices (line 6). If a command requires parameter, its corresponding sub-actor module selects the appropriate parameters, forming a complete sequence of paired command generator ID with parameters (lines 8–16). The commands are then generated and executed in Gazebo (lines 17–18). If crashes are observed, they are logged for further manual reduction (lines 19–21), and feedback from the test results is used to update the actor, the sub-actors, and the critic modules (lines 22–23), ensuring continuous learning and improvement in the fuzzing process. This process iterates, until the stopping criteria are met, e.g., maximum running time or iterations.

Running Example: Figure 4 presents a real-world bug that has been fixed. Specifically, Figure 4(a) shows a model snippet with the AckermannSteering plugin, while Figure 4(b) demonstrates a Gazebo command that creates and inserts this model. Figure 4(c) then illustrates a command that resets the simulation by calling the service `/world/world_0/control/` with the message "`reset {all: true}`". Upon execution, the simulation restarts, and the model `vehicle_blue`

Table 3. Parameter setup for GzFuzz

	Parameter	Value	Description
1	n_c	10	Number of commands in the generated sequence
2	N	128	Number of neurons in the hidden layer
3	ω	4	Reward factor for crash
4	λ	0.2	Reward for coverage increase
5	h	100	Number of command sequences for evaluating diversity
6	α	0.001	Learning rate for neural network modules

is expected to be removed from the scenario. However, due to the underlying implementation, the `AckermannSteering` plugin fails to unload properly, which further leads to the crash of the simulator.

To trigger this bug, the **Learning-based Command Generator Selection** first selects generators 6 and 1 through the actor module. Generator 6 requires its corresponding sub-actor to specify the `vehicle_blue` model as a parameter, while generator 1 selects the `world/world_0/control` service, and constructs a reset message based on the `Protobuf` definition. The **Syntax-aware Feasible Command Generation** mechanism ensures that both commands conform to the syntax requirements of `Gazebo`. Executing these commands triggers the bug, which stems from `Gazebo`'s mishandling of cascading plugin removals.

4 Evaluation

GzFuzz is implemented in Python. The neural network modules are implemented in `PyTorch` [47]. For XML parsing and modification, the `lxml` library (5.3.0) is used. To generate feasible message for given `Protobuf` definitions, the `randomproto` library (0.0.1) is employed. As for the hardware environment, our experiments are conducted on a GNU/Linux PC with Intel Core i9-13900K CPU and 128GB RAM, running Ubuntu 22.04 (x86_64).

To evaluate the effectiveness of GzFuzz, we designed experiments based on the following Research Questions (RQs).

(1) **RQ1**: Is GzFuzz effective in finding bugs for `Gazebo`?
(2) **RQ2**: How effective is GzFuzz compared with existing fuzzing methods?
(3) **RQ3**: What is the impact of the two main components on the effectiveness of GzFuzz?

Specifically, RQ1 aims to assess the ability of GzFuzz to identify bugs in the latest version of `Gazebo`. RQ2 focuses on evaluating the effectiveness of GzFuzz in detecting bugs when compared with existing approaches. RQ3 validates the efficacy of the two primary components at the core of GzFuzz.

Before investigating each RQ, we present the setup of the parameters introduced in GzFuzz in Table 3. The value for ω is set following existing studies [15, 55], and the rest parameter settings of GzFuzz are based on a small-scale experiment. In our preliminary experiments, we observe that GzFuzz is not very sensitive to the parameter setup. For instance, we also implement an variant in which the neural networks are trained with a constant learning rate 0.001 as in Table 3, and observe that its crash-triggering effectiveness is similar as that achieved with the Adam optimizer.

4.1 Investigation to RQ1

To evaluate the ability of GzFuzz to detect real-world bugs in `Gazebo`, we conducted an experiment from June 2024 to October 2024. This RQ mainly concentrates on testing the latest development

Table 4. Summarization of the Detected Bugs

Bug ID	Status	#Steps	Symptom
#614	fixed	2	Crash after calling /gazebo/resource_paths/resolve with empty request
#2458	fixed	2	Crash caused by service "/world/default/enable_collision"
#2459	fixed	3	Crash after adding and removing lift_drag_demo_model
#2464	fixed	2	Crash after removing model with LogicalAudioSensorPlugin
#2465	fixed	3	Crash after calling WorldControl service over model with AckermannSteering plugin
#2466	fixed	3	Crash after removing model with started LinearBatteryPlugin
#2507	fixed	2	Crash after dynamically adding plugin AckermannSteering
#2508	fixed	2	Crash after dynamically adding plugin LiftDrag
#2511	fixed	2	Crash after dynamically adding plugin LinearBatteryPlugin
#2512	fixed	2	Crash after dynamically adding plugin DiffDrive
#2513	fixed	2	Crash after dynamically adding plugin MecanumDrive
#2531	fixed	2	Crash after calling /model/elevator/door_0/lidar topic with empty gz.msgs.LaserScan message
#2541	fixed	2	Crash after calling /depth_camera/points topic
#2543	fixed	2	Crash after calling /optical_tactile_sensor/enable over OpticalTactilePlugin
#2566	fixed	2	Crash after removing model with FollowActor plugin
#2615	fixed	1	Crash after loading an sdf containing an invalid near clip distance parameter
#2616	fixed	1	Crash after loading sdf with negative size parameter
#2624	fixed	3	Crash after adding model with DetachableJoint and resetting
#2646	fixed	2	Crash after adding plugin MulticopterMotorModel
#2471	confirmed	2	Crash after adding model twice to the same location
#2538	confirmed	2	Crash after calling playback/control service on the buoyant_cylinder example
#2542	confirmed	2	Crash after calling WorldControl service over ModelPhotoShoot
#2600	confirmed	2	Crash after calling control/state over OpticalTactilePlugin
#2605	confirmed	2	Crash after calling set_pose service over model with JointTrajectoryController plugin
#661	duplicate	1	Crash caused by bounding box overflow

version of `Gazebo`, and the duration of each testing process is about a week. At the conclusion of each testing process, we choose to update `Gazebo` source code to its latest version if available.

Since the first bug reported on July 1st, 2024, we have submitted 25 bug reports, among which 19 have been fixed, five have been confirmed, and one has been identified as duplicate. Table 4 illustrates the information of these submitted bug reports. In the table, the first column indicates the bug ID. The second column represents the current status of the issue reports (fixed, confirmed, or duplicate). The third column shows the number of steps to reproduce the bug, i.e., the number of commands to invoke. The fourth column lists the symptom of the reported issue.

Accompanied by the table, we discuss the key insights gained from the submitted bugs:

```
gz service -s /gazebo/resource_paths/resolve \
--reptype gz.msgs.StringMsg \
--reqtype gz.msgs.StringMsg \
--req ''
```

(a) Command that triggers the crash

```
std::string common::absPath(const std::string &_path)
{
-  return fs::absolute(_path).string();
+  std::error_code ec;
+  auto path = fs::absolute(_path, ec);
+  if (!fsWarn("absPath", ec))
+  {
+    path = "";
+  }
+
+  return path.string();
}
```

(b) Snippet for the pull request

Fig. 5. Bug #614: Crash calling /gazebo/resource_paths/resolve with empty request

(1) All the bugs we report are crash-related, in line with the test oracle used in this study. This crash-centric approach ensures that the most severe and impactful bugs, which can lead to a complete breakdown of the simulator, are prioritized and detected efficiently during the testing process.
(2) Most submitted bugs are related to Gazebo plugins. Among the 25 submitted bugs, 16 are related with plugins. For example, bugs #2459, #2464, #2466, and #2566 are triggered by removing models with plugins. The issue stems from improper handling of the plugins when the models are removed.
(3) Bug #661 is frequently observed, and is identified as a duplicate. According to the developer's feedback, the crash occurs when objects move so far from the origin that they no longer fit within the bounding box created by the collision detection code[6].
(4) GzFuzz effectively generates compact command sequences, i.e., after manual reduction, all crashes could be reproduced within three steps or fewer. Meanwhile, since we do not employ bit/byte-flipping mutations, the commands constructed by GzFuzz are generally readable and interpretable.

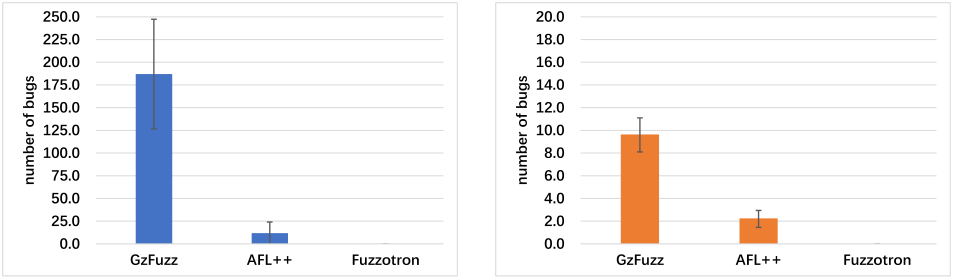
In particular, we highlight Bug #614[7], which we submitted and fixed. This bug-triggering command was generated by the random service command generator. As shown in Figure 5(a), the bug is triggered by invoking the `/gazebo/resource_paths/resolve` service with an empty request. The intended functionality of the service is to read the input request and return its corresponding absolute file path. However, when the request is an empty string, it causes the server of Gazebo to crash. The root cause, according to our discussion with the contributors, lies in the different implementations of the `std::filesystem::absolute(const std::filesystem::path& p)` function across different operating systems[8]. For GNU/Linux, invoking this API with empty `path` leads to a crash. Meanwhile for macOS, no crash occurs for the same parameter. By replacing the API with the non-throwing version, the bug could be fixed. Figure 5(b) presents the patch illustrating the pull request we filed, which has been merged into the codebase of Gazebo.

Answer to RQ1: GzFuzz is able to detect bugs for Gazebo. In less than six months, GzFuzz has successfully detected 25 unique crashes, with 24 of these bugs being fixed or confirmed.

4.2 Investigation to RQ2

Due to the lack of existing studies focusing on robotic simulator fuzzing, to objectively evaluate the effectiveness of GzFuzz, two general-purpose fuzzers are considered as baselines, i.e., AFL++

[6] https://github.com/gazebosim/gz-physics/issues/661
[7] https://github.com/gazebosim/gz-common/issues/614
[8] https://github.com/gazebosim/gz-common/pull/620



(a) Total crashes (b) Unique crashes

Fig. 6. Bugs found by GzFuzz, AFL++, and Fuzzotron

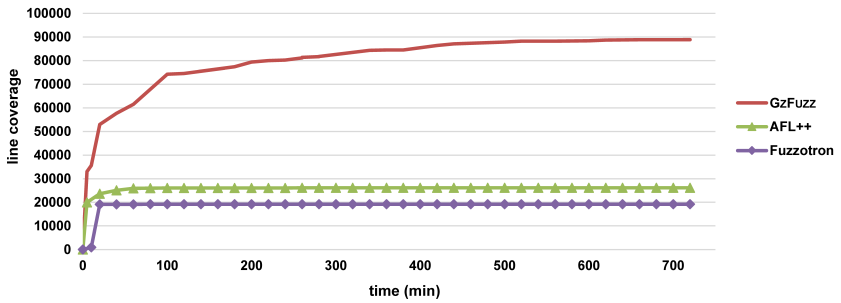


Fig. 7. Line coverage achieved by GzFuzz and baselines

and Fuzzotron. AFL++ is a widely-used fuzzer and has been applied to various domains. However, due to Gazebo's client-server architecture, fuzzing from the client side with AFL++ is challenging. As a workaround, the fuzz testing is realized by mutating the input SDF files. Additionally, while there are several network-based fuzzers [9, 29, 43, 49, 63], most of these fuzzers assume specific formats or protocol stacks. Given Gazebo's unique communication mechanism, Fuzzotron is the only network-based fuzzer we successfully configured for comparison.

The experiment in this RQ is conducted using the previous release of Gazebo, i.e., version 8.0.0, in order to encompass a broader range of bugs for the purpose of comparison. To ensure a fair comparison among the methods, each fuzz testing approach uses the same input files, specifically the SDF files extracted in Section 3.1. We establish a uniform execution time of 12 hours for each testing method, and the experiment is conducted for five independent runs. For newly discovered bugs, if they do not persist in the latest development version, we consider them as actual fixed bugs. Otherwise, we will promptly report them to developers for further confirmation. Besides, we check the uniqueness of bugs based on the stack trace recorded at the time of the crash, and results are further manually verified.

The results obtained by GzFuzz and the baseline approaches are presented in Figure 6. The figure shows the results of each approach with two bars, i.e., one representing the distribution of the total number of bugs detected and the other representing the number of unique bugs found. Each bar illustrates the average number of bugs detected, as well as the error bars. From the figure, GzFuzz is able to detect an average of 9.6 unique bugs in 12 hours, which significantly outperforms the two baseline methods (2.2 for AFL++ and 0 for Fuzzotron). We also conduct Mann-Whitney U

test [39] on the total number of bugs found by GzFuzz and its comparative approaches, with the null hypothesis claiming that the two approaches in comparison perform similarly. The results indicate that GzFuzz is more effective, when compared to AFL++ and Fuzzotron (with p-values < 0.02), respectively. We also calculate the effect size for each comparison scenario [13], using Cureton's rank-biserial correlation (r_{rb}), which ranges from -1 to 1. This statistic quantifies the degree of separation between two independent groups based on their ranks. An r_{rb} value of 1 indicates perfect separation, meaning all observations from one group have higher ranks than all observations from the other group. In our analysis, we observe that $r_{rb} = 1$ when we compare GzFuzz with AFL++ and Fuzzotron, respectively, which implies a significant degree of separation between the results obtained by GzFuzz and its comparative approach.

Furthermore, we investigate how effectively each approach covers different parts of `Gazebo`. In Figure 7, we plot the line coverage obtained by running GzFuzz, AFL++, and Fuzzotron on `Gazebo`. Here, the x-axis represents the time elapsed, while the y-axis indicates the line coverage. The figure shows that GzFuzz achieves substantially higher coverage than the baseline methods. After 12 hours of fuzzing, GzFuzz reaches a coverage of 88,862 lines, which represents an increase of approximately 239% over AFL++ (26,147 lines) and 363% over Fuzzotron (19,185 lines). This result suggests that GzFuzz is capable of thoroughly reaching more parts of `Gazebo`. Notably, AFL++ outperforms Fuzzotron in coverage, consistent with its higher effectiveness in detecting bugs. The relatively low coverage achieved by AFL++ and Fuzzotron is largely due to their lack of input syntax guidance. Consequently, most randomly generated inputs are rejected by `Gazebo`'s parser, limiting their ability to explore deeper execution paths.

Answer to RQ2: GzFuzz significantly outperforms the baseline approaches. Over a 12-hour testing period, GzFuzz detects more bugs on the previous release of `Gazebo`, demonstrating a statistically significant improvement compared to the baseline methods. Additionally, GzFuzz achieves substantially higher code coverage, further highlighting its effectiveness in exploring deeper execution paths and uncovering more bugs.

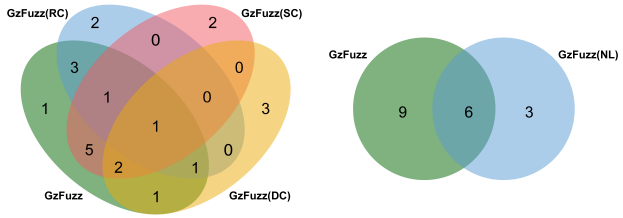
4.3 Investigation to RQ3

In this study, GzFuzz features the combination of syntax-aware feasible command generation and reinforcement learning-based command generator selection, to tackle the strict input challenge and the state-space challenge, respectively. Hence, we are interested in how these two mechanisms contribute to the framework. To evaluate the impact of the two proposed mechanisms, we design a set of variants of GzFuzz. On the one hand, to investigate the effectiveness of the feasible command generation, three variants are developed, each with single category of command generators, i.e., GzFuzz(`RC`), GzFuzz(`SC`), GzFuzz(`DC`) for random, semi-realistic, and disruptive command, respectively. On the other hand, to investigate the usefulness of the learning-based command generator selection, we consider GzFuzz(`NL`), which employs random selection strategy. The experiment setup is the same as RQ2, i.e., each fuzzing process lasts 12 hours, repeated five times.

In Table 5, we present the bug statistics of the approaches in comparison, as well as the p-values and effect sizes when comparing GzFuzz and its variants. From Table 5, the following observations could be obtained. When comparing GzFuzz with its variants, we find that GzFuzz could detect more bugs in average, when compared to GzFuzz(`RC`) and GzFuzz(`DC`) (with p-values < 0.03, $r_{rb} > 0.90$). However, when we compare GzFuzz with GzFuzz(`SC`), a p-value of 0.08 is obtained, which to some extents implies that the semi-realistic command generators contribute more to GzFuzz. To gain more insights into the comparison, in Figure 8, we illustrate the comparison between GzFuzz and its variants, measured by the number of bugs detected. The figure is organized with Venn diagram [12]. From the figure, we observe that compared with each variant, GzFuzz can identify 6–10 distinct bugs, indicating that using single category of command generators alone is insufficient. Since each

Table 5. Bug statistics of GzFuzz and its variants

	Min. Bugs	Max. Bugs	Avg. Bugs	Total Unique	p-value	Effect Size
GzFuzz(RC)	4	7	5.0	8	0.02	0.96
GzFuzz(SC)	**7**	9	7.6	11	0.08	0.68
GzFuzz(DC)	3	8	4.8	8	0.02	0.92
GzFuzz(NL)	3	7	4.6	9	0.02	0.96
GzFuzz	**7**	**11**	**9.6**	**15**	-	-



(a) GzFuzz vs. GzFuzz(RC/SC/DC) (b) GzFuzz vs. GzFuzz(NL)

Fig. 8. Venn diagram illustrating GzFuzz vs. its variants

category of command generators could lead to unique crashes, incorporating all the command generators help improve the overall effectiveness of GzFuzz. Besides, when we compare GzFuzz and GzFuzz(NL), we could observe that GzFuzz achieves better results (p-value=0.02, r_{rb} = 0.96), which demonstrates the usefulness of the learning-based command generator selection mechanism.

Answer to RQ3: The experimental results demonstrate the contributions of the three categories of syntax-aware feasible command generators, as well as the reinforcement learning-based command generator selection. The two mechanisms can collectively enhance the effectiveness in detecting Gazebo bugs.

5 Threats to Validity

5.1 Threats to Internal Validity

One potential threat to internal validity in our study is the accuracy and completeness of the test environment configuration. If Gazebo is not set up correctly or if certain plugins, models, or configurations are missing or misconfigured, the results from GzFuzz may not reflect the true behavior of the system. Additionally, while we designed GzFuzz to effectively target the state space and input challenges, there is a possibility that certain bugs remain undetected due to limitations in the fuzzing strategy or reinforcement learning model, which could skew our results. Another concern is the possibility of non-deterministic behaviors within Gazebo simulations, which might cause inconsistent outcomes between fuzzing runs and affect the reproducibility of the crashes we identified. To mitigate these risks, we carefully verified the setup and configurations, ran multiple tests to account for variability, and cross-validated crash reports to ensure consistency.

Besides, we should note that the scope of the fuzzing process is limited. For instance, GzFuzz currently generates command sequences that add plugins without ensuring the corresponding service or topic commands necessary to invoke those plugins. Consequently, certain plugin functionalities remain unexercised. To address this threat, future work should extend the framework to

explicitly include plugin-activation commands, thereby restoring the validity of causal inferences about plugin behavior.

5.2 Threats to External Validity

A key threat to external validity in our study is the generalizability of the results beyond Gazebo. While GzFuzz is specifically designed for testing Gazebo, it may not perform as effectively on other robotic simulators that have different architectures, input formats, or state-space characteristics, such as Webots [41], CoppeliaSim (formerly V-REP) [50], or MORSE [26]. Additionally, the types of robotic systems and scenarios we tested may not cover the full range of possible use cases in other domains or real-world applications. As a result, the effectiveness of GzFuzz in detecting bugs and crashes could vary when applied to different simulators or robotic configurations.

To address this threat, GzFuzz has been designed with adaptability in mind. Firstly, SDF is a general format supported by multiple simulators, and there exist tools available for relevant format translation, such as Unified Robot Description Format (URDF)[9] and Universal Scene Description (USD)[10]. Hence, the syntax-aware model and plugin command generation components of GzFuzz can be reused with minimal modifications. Secondly, essential commands for model manipulation operations are common across various simulators, and most provide APIs or protocols that facilitate integration with GzFuzz through appropriate adaptations. This inherent flexibility not only supports the extension of GzFuzz to other platforms but also opens avenues for differential testing, enabling advanced test oracles through cross-simulator behavior comparisons.

6 Related Work

6.1 Empirical Software Engineering for Robotics

Robotic software engineering faces growing challenges as robots become more autonomous and complex. Ensuring software quality and reliability is critical, and several studies emphasize the need for empirical methods to address these challenges.

Dos Santos et al. conduct a preliminary systematic mapping on software engineering for robotic systems [24], and analyze studies in the robotic systems area, addressing in a combined way, software engineering approaches and software quality aspects, and observe that the less investigated software quality aspect is security. García et al. provide a large-scale empirical analysis of current practices, challenges, and solutions in robotics software engineering, based on interviews with industry professionals and a survey of global practitioners [28]. The author observe an overall lack of software engineering best practices in robotics development. Albonico et al. conduct a systematic mapping study on ROS-related software engineering research [7]. In this study, 63 primary studies on software engineering research on ROS are reviewed. Interestingly, in both studies, the authors confirm that Gazebo is the most commonly used simulation tool. Farley, Wang, and Marshall conduct a comparative study, on the choice of robotic simulators. CoppeliaSim, Gazebo, MORSE, and Webots are evaluated from multiple perspectives, such as software license, programming language support, physics engine support, etc.

Given the widespread use of Gazebo in robotics development, ensuring its reliability is crucial. Therefore, testing Gazebo is essential to maintain the integrity of robotics development processes.

6.2 Robotic Software Testing

Robotic software testing is essential for ensuring the reliability and safety of autonomous systems, which frequently operate in unpredictable and complex environments. Thorough testing is critical

[9] https://github.com/ros/sdformat_urdf
[10] https://github.com/gazebosim/gz-usd

to identify potential issues and vulnerabilities before deployment, as these systems must handle diverse scenarios and adapt to dynamic conditions in real-world applications.

Currently, research in this area primarily focuses on testing ROS, given its widespread use in the development of robotic applications. Xie et al. develop the fuzzing framework named ROZZ, to effectively test ROS programs and detect bugs based on ROS properties [59]. Bai et al. design a new fuzzing framework named ROFER [11], to effectively test robotic programs in ROS for bug detection. Compared to other ROS fuzzing approaches, ROFER features a dimension-level mutation method that considers the contribution of each input dimension to testing coverage, and a message-guided fuzzing approach that uses a new coverage metric named message feature. Kim and Kim propose a feedback-driven fuzzing framework RoboFuzz [33], to detect semantic correctness bugs, including the violation of specification, violation of physical laws, and cyber-physical discrepancy in ROS and ROS-powered systems. Besides studies focusing on ROS and ROS-powered systems, Xiao, Liu, and Wang propose PhyFu, a novel approach for fuzzing modern physics simulation engines (PSEs) [58]. PSEs generally consist of software stack ranging from simulation algorithms, hardware acceleration modules, to domain specific languages and APIs. PhyFu is designed on the basis of principled physical laws to stress testing logic errors in PSEs. While primarily focusing on logical bugs, PhyFu is also able to identify crashes, two of which have been successfully fixed.

Compared with these studies, GzFuzz introduces the first approach specifically designed for directly fuzzing robotic simulators like Gazebo. By targeting the simulator itself, we aim to uncover bugs that arise from the interaction with virtual environments, which existing approaches do not address.

7 Conclusion

In this paper, we propose GzFuzz, the first fuzzing framework for Gazebo, the most widely-used robotic simulator. GzFuzz features the combination of syntax-aware feasible command generation, as well as a reinforcement learning-based command generator selection. In less than six months, GzFuzz has successfully detected 25 unique crashes, with 24 of these bugs being fixed or confirmed.

For future work, we aim to explore several additional avenues to further improve GzFuzz. One direction is expanding its applicability to other robotic simulation environments, allowing for a broader impact across diverse platforms. For example, incorporating differential testing [19, 30, 51] could help identify inconsistencies between different simulators, improving overall robustness and cross-platform reliability. We also plan to refine the strategies for command generation, focusing on increasing the realism and complexity of simulated scenarios to better capture edge cases. Additionally, metamorphic testing [17, 38, 45, 54] could be incorporated to further enhance the testing process by generating follow-up test cases based on expected relationships between different inputs and outputs. This approach would help in detecting bugs that might not be exposed through standard testing methods. Furthermore, we are interested in exploring the employment of large language models and in-context learning techniques [18, 23, 40, 57, 60] to automate the generation of more sophisticated test commands. This will not only improve the efficiency and precision of test command generation but also enhance the adaptability of our GzFuzz framework.

Acknowledgements

This work is supported in part by the National Natural Science Foundation of China under Grants 62132020, 62032004, 62432010, 62072068, the Foundational Research Funds for the Central Universities under Grant NO. DUT24LAB126, and the National Key Research and Development Program of China under Grant NO. 2018YF-B1003900.

References

[1] Ahmed AbdelHamed, Girma Tewolde, and Jaerock Kwon. 2020. Simulation Framework for Development and Testing of Autonomous Vehicles. In *2020 IEEE International IOT, Electronics and Mechatronics Conference (IEMTRONICS)*. IEEE, 1–6.

[2] Haitham Y. Adarbah, Mehdi Sookhak, and Mohammed Atiquzzaman. 2023. A Digital Twin Environment for 5G Vehicle-to-Everything: Architecture and Open Issues. In *Proceedings of the International ACM Symposium on Performance Evaluation of Wireless Ad Hoc, Sensor, & Ubiquitous Networks, Montreal*, Mónica Aguilar-Igartua, Luis J. de la Cruz Llopis, and Thomas Begin (Eds.). ACM, 115–122.

[3] Afsoon Afzal, Deborah S Katz, Claire Le Goues, and Christopher S Timperley. 2021. Simulation for Robotics Test Automation: Developer Perspectives. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. 263–274.

[4] Afsoon Afzal, Claire Le Goues, Michael Hilton, and Christopher Steven Timperley. 2020. A Study on Challenges of Testing Robotic Systems. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 96–107.

[5] Carlos E Agüero, Nate Koenig, Ian Chen, Hugo Boyer, Steven Peters, John Hsu, Brian Gerkey, Steffi Paepcke, Jose L Rivero, Justin Manzo, et al. 2015. Inside the Virtual Robotics Challenge: Simulating Real-Time Robotic Disaster Response. *IEEE Transactions on Automation Science and Engineering* 12, 2 (2015), 494–506.

[6] Mohamed Fasil Syed Ahamed, Girma Tewolde, and Jaerock Kwon. 2018. Software-in-the-Loop Modeling and Simulation Framework for Autonomous Vehicles. In *2018 IEEE International Conference on Electro/Information Technology (EIT)*. IEEE, 0305–0310.

[7] Michel Albonico, Milica Đorđević, Engel Hamer, and Ivano Malavolta. 2023. Software Engineering Research on the Robot Operating System: A Systematic Mapping Study. *Journal of Systems and Software* 197 (2023), 111574.

[8] Mark Allan, Uland Wong, P Michael Furlong, Arno Rogg, Scott McMichael, Terry Welsh, Ian Chen, Steven Peters, Brian Gerkey, Moraan Quigley, et al. 2019. Planetary Rover Simulation for Lunar Exploration Missions. In *2019 IEEE Aerospace Conference*. IEEE, 1–19.

[9] Anastasios Andronidis and Cristian Cadar. 2022. Snapfuzz: High-Throughput Fuzzing of Network Applications. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 340–351.

[10] Pierre-Luc Bacon, Jean Harb, and Doina Precup. 2017. The Option-Critic Architecture. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI)*, Satinder Singh and Shaul Markovitch (Eds.). AAAI Press, 1726–1734.

[11] Jia-Ju Bai, Hao-Xuan Song, and Shi-Min Hu. 2024. Multi-Dimensional and Message-Guided Fuzzing for Robotic Programs in Robot Operating System. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 763–778.

[12] Philippe Bardou, Jérôme Mariette, Frédéric Escudié, Christophe Djemiel, and Christophe Klopp. 2014. JVenn: an Interactive Venn Diagram Viewer. *BMC bioinformatics* 15 (2014), 1–7.

[13] Janis E Johnston Kenneth J Berry, Paul W Mielke Jr, Berry, and Hiripi. 2018. *Measurement of Association*. Springer.

[14] Brian Bingham, Carlos Agüero, Michael McCarrin, Joseph Klamo, Joshua Malia, Kevin Allen, Tyler Lum, Marshall Rawson, and Rumman Waqar. 2019. Toward Maritime Robotic Simulation in Gazebo. In *OCEANS 2019 MTS/IEEE SEATTLE*. 1–10.

[15] Junjie Chen, Haoyang Ma, and Lingming Zhang. 2020. Enhanced Compiler Bug Isolation via Memoized Search. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 78–89.

[16] Junjie Chen, Chenyao Suo, Jiajun Jiang, Peiqi Chen, and Xingjian Li. 2023. Compiler Test-Program Generation via Memoized Configuration Search. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2035–2047.

[17] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic Testing: A Review of Challenges and Opportunities. *ACM Comput. Surv.* 51, 1 (2018), 4:1–4:27.

[18] Xiangping Chen, Xing Hu, Yuan Huang, He Jiang, Weixing Ji, Yanjie Jiang, Yanyan Jiang, Bo Liu, Hui Liu, Xiaochen Li, et al. 2025. Deep Learning-Based Software Engineering: Progress, Challenges, and Opportunities. *Science China Information Sciences* 68, 1 (2025), 1–88.

[19] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-Directed Differential Testing of JVM Implementations. In *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 85–99.

[20] HeeSun Choi, Cindy Crump, Christian Duriez, Asher Elmquist, Gregory Hager, David Han, Frank Hearl, Jessica Hodgins, Abhinandan Jain, Frederick Leve, et al. 2021. On the Use of Simulation in Robotics: Opportunities, Challenges, and Suggestions for Moving Forward. *Proceedings of the National Academy of Sciences* 118, 1 (2021), e1907856118.

[21] Timothy H Chung, Viktor Orekhov, and Angela Maio. 2023. Into the Robotic Depths: Analysis and Insights from the DARPA Subterranean Challenge. *Annual Review of Control, Robotics, and Autonomous Systems* 6, 1 (2023), 477–502.

[22] Danielle Dauphinais, Michael Zylka, Harris Spahic, Farhan Shaik, Jingda Yang, Isabella Cruz, Jakob Gibson, and Ying Wang. 2023. Automated Vulnerability Testing and Detection Digital Twin Framework for 5G Systems. In *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*. 308–310.

[23] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2024. Large Language Models are Edge-Case Generators: Crafting Unusual Programs for Fuzzing Deep Learning Libraries. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE)*. ACM, 70:1–70:13.

[24] Marcela G Dos Santos, Bianca M Napoleão, Fabio Petrillo, Darine Ameyed, and Fehmi Jaafar. 2020. A Preliminary Systematic Mapping on Software Engineering for Robotic Systems: A Software Quality Perspective. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. 647–654.

[25] Kamak Ebadi, Lukas Bernreiter, Harel Biggie, Gavin Catt, Yun Chang, Arghya Chatterjee, Christopher E Denniston, Simon-Pierre Deschênes, Kyle Harlow, Shehryar Khattak, et al. 2024. Present and Future of SLAM in Extreme Environments: The DARPA SubT Challenge. *IEEE Transactions on Robotics* 40 (2024), 936–959.

[26] Gilberto Echeverria, Nicolas Lassabe, Arnaud Degroote, and Séverin Lemaignan. 2011. Modular Open Robots Simulation Engine: MORSE. In *2011 IEEE International Conference on Robotics and Automation (ICRA)*. 46–51.

[27] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT)*.

[28] Sergio García, Daniel Strüber, Davide Brugali, Thorsten Berger, and Patrizio Pelliccione. 2020. Robotics Software Engineering: A Perspective from the Service Robotics Domain. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 593–604.

[29] Emre Güler, Sergej Schumilo, Moritz Schloegel, Nils Bars, Philipp Görz, Xinyi Xu, Cemal Kaygusuz, and Thorsten Holz. 2024. Atropos: Effective Fuzzing of Web Applications for Server-Side Vulnerabilities. In *USENIX Security Symposium*.

[30] Muhammad Ali Gulzar, Yongkang Zhu, and Xiaofeng Han. 2019. Perception and Practices of Differential Testing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 71–80.

[31] Kimberly A Hambuchen, Monsi C Roman, Amy Sivak, Angela Herblet, Nathan Koenig, Daniel Newmyer, and Robert Ambrose. 2017. NASA's Space Robotics Challenge: Advancing Robotics for Future Exploration Missions. In *AIAA SPACE and Astronautics Forum and Exposition*. 5120.

[32] Yang Hong and Jun Wu. 2024. Fuzzing Digital Twin With Graphical Visualization of Electronic AVs Provable Test for Consumer Safety. *IEEE Transactions on Consumer Electronics* 70, 1 (2024), 4633–4644.

[33] 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)*. 447–458.

[34] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.).

[35] 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)*, Vol. 3. 2149–2154.

[36] 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.

[37] Musa Morena Marcusso Manhães, Sebastian A Scherer, Martin Voss, Luiz Ricardo Douat, and Thomas Rauschenbach. 2016. UUV simulator: A Gazebo-Based Package for Underwater Intervention and Multi-Robot Simulation. In *Oceans 2016 Mts/IEEE Monterey*. 1–8.

[38] Muhammad Numair Mansur, Valentin Wüstholz, and Maria Christakis. 2023. Dependency-Aware Metamorphic Testing of Datalog Engines. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 236–247.

[39] Patrick E McKnight and Julius Najab. 2010. Mann-Whitney U Test. *The Corsini Encyclopedia of Psychology* (2010), 1–1.

[40] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large Language Model Guided Protocol Fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*.

[41] Olivier Michel. 2004. Cyberbotics ltd. Webots™: Professional Mobile Robot Simulation. *International Journal of Advanced Robotic Systems* 1, 1 (2004), 5.

[42] Michel Nass, Emil Alégroth, and Robert Feldt. 2021. Why Many Challenges with GUI Test Automation (Will) Remain. *Information and Software Technology* 138 (2021), 106625.

[43] Roberto Natella. 2022. StateAFL: Greybox Fuzzing for Stateful Network Servers. *Empirical Software Engineering* 27, 7 (2022), 191.

[44] Lucas Nogueira. 2014. Comparative Analysis Between Gazebo and V-REP Robotic Simulators. *Seminario Interno de Cognicao Artificial-SICA* 2014, 5 (2014), 2.

[45] Megan Olsen and Mohammad Raunak. 2019. Increasing Validity of Simulation Models Through Metamorphic Testing. *IEEE Transactions on Reliability* 68, 1 (2019), 91–108.
[46] Ciprian Paduraru., Rares Cristea., and Alin Stefanescu. 2023. Automatic Fuzz Testing and Tuning Tools for Software Blueprints. In *Proceedings of the 18th International Conference on Software Technologies (ICSOFT)*. SciTePress, 151–162.
[47] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in neural information processing systems* 32 (2019).
[48] Shubham Pateria, Budhitama Subagdja, Ah-Hwee Tan, and Chai Quek. 2022. Hierarchical Reinforcement Learning: A Comprehensive Survey. *Comput. Surveys* 54, 5 (2022), 109:1–109:35.
[49] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNet: A Greybox Fuzzer for Network Protocols. In *Proceedings of the 13rd IEEE International Conference on Software Testing, Verification and Validation : Testing Tools Track*.
[50] Eric Rohmer, Surya PN Singh, and Marc Freese. 2013. V-REP: A Versatile and Scalable Robot Simulation Framework. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 1321–1326.
[51] Mayank Sharma, Pingshi Yu, and Alastair F Donaldson. 2023. Rustsmith: Random Differential Compiler Testing for Rust. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 1483–1486.
[52] Alexander Smirnov and Nikolay Teslya. 2020. Modeling of Robot Interaction in Coalition Through Smart Space and Blockchain: Precision Agriculture Scenario. In *International Conference on Enterprise Information Systems (ICEIS)*. 481–497.
[53] L Sopegno, P Livreri, Kp Valavanis, et al. 2022. Using UAVs for Future Mission on Mars. In *Proceedings of the International Astronautical Congress, IAC*, Vol. 2022. International Astronautical Federation, IAF.
[54] Chang-Ai Sun, An Fu, Pak-Lok Poon, Xiaoyuan Xie, Huai Liu, and Tsong Yueh Chen. 2021. METRIC$^+$+: A Metamorphic Relation Identification Technique Based on Input Plus Output Domains. *IEEE Transactions on Software Engineering* 47, 9 (2021), 1764–1785.
[55] Xiaohui Wan, Tiancheng Li, Weibin Lin, Yi Cai, and Zheng Zheng. 2024. Coverage-Guided Fuzzing for Deep Reinforcement Learning Systems. *Journal of Systems and Software* 210 (2024), 111963.
[56] Thomas Wetzlmaier, Rudolf Ramler, and Werner Putschögl. 2016. A Framework for Monkey GUI Testing. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 416–423.
[57] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4all: Universal Fuzzing with Large Language Models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. 1–13.
[58] Dongwei Xiao, Zhibo Liu, and Shuai Wang. 2023. PhyFu: Fuzzing Modern Physics Simulation Engines. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1579–1591.
[59] Kai-Tao Xie, Jia-Ju Bai, Yong-Hao Zou, and Yu-Ping Wang. 2022. ROZZ: Property-Based Fuzzing for Robotic Programs in ROS. In *2022 International Conference on Robotics and Automation (ICRA)*. 6786–6792.
[60] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. 2024. WhiteFox: White-Box Compiler Fuzzing Empowered by Large Language Models. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 296 (Oct. 2024), 27 pages.
[61] Ruyi Zhou, Wenhao Feng, Liang Ding, Huaiguang Yang, Haibo Gao, Guangjun Liu, and Zongquan Deng. 2022. MarsSim: a High-Fidelity Physical and Visual Simulation for Mars Rovers. *IEEE Trans. Aerospace Electron. Systems* 59, 2 (2022), 1879–1892.
[62] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: A Survey for Roadmap. *ACM Comput. Surv.* 54, 11s, Article 230 (Sept. 2022), 36 pages.
[63] Yong-Hao Zou, Jia-Ju Bai, Jielong Zhou, Jianfeng Tan, Chenggang Qin, and Shi-Min Hu. 2021. TCP-Fuzz: Detecting Memory and Semantic Bugs in TCP Stacks with Fuzzing. In *2021 USENIX Annual Technical Conference (USENIX ATC)*. 489–502.

Received 2024-10-31; accepted 2025-03-31