IcFuzz: Fuzzing Isaac Sim with Semantic Stage Guidance and Multi-Level Mutation

Zhixiang Chen[†]
Sun Yat-sen University
Zhuhai, China
chenzhx69@mail2.sysu.edu.cn

Zhuangbin Chen[*†]
Sun Yat-sen University
Zhuhai, China
chenzhb36@mail.sysu.edu.cn

Ruoxi Jia[†]
Sun Yat-sen University
Zhuhai, China
jiarx3@mail2.sysu.edu.cn

Zeqin Liao
Nanyang Technological University
Singapore, Singapore
zeqin.liao@ntu.edu.sg

Wei Li[†]
Sun Yat-sen University
Zhuhai, China
liwei378@mail2.sysu.edu.cn

Jinyang Liu
Chinese University of Hong Kong
Hong Kong, China
jyliu@cse.cuhk.edu.hk

Zibin Zheng[†]
Sun Yat-sen University
Zhuhai, China
zhzibin@mail.sysu.edu.cn

Abstract

Robotics simulators serve as a foundational infrastructure for embodied AI, facilitating safe and scalable robotic system development. NVIDIA Isaac Sim has emerged as one of the most popular simulators, distinguished by its GPU-accelerated physics engine and photorealistic rendering, which enable high-fidelity modeling of complex environments. However, its inherent complexity inevitably introduces software bugs that can compromise simulation reliability. Existing fuzzing approaches struggle to test Isaac Sim effectively due to challenges of context-aware object semantics, hierarchical simulation control, and a vast simulation state space.

In this paper, we propose IcFuzz, the first fuzzing approach for Isaac Sim. IcFuzz first performs an LLM-based semantic stage segmentation, decomposing simulation programs into structured stages that capture context-aware object semantics. Guided by this information, IcFuzz designs multi-level mutation operators to systematically exercise the simulator across hierarchical granularities. To efficiently navigate the vast simulation state space, IcFuzz employs a multi-armed bandit algorithm to adaptively schedule mutation operators. Experimental results show that IcFuzz outperforms the baselines in terms of both code coverage and bug detection. Specifically, IcFuzz achieves approximately 190%–205% of the code coverage of the baselines and detects an average of 3.7 unique crashes over three rounds of 12-hour tests, while no crashes are detected by the baselines. Moreover, IcFuzz has uncovered 11 bugs over approximately four months, 9 of which have been confirmed or fixed by the developers.

[*]Zhuangbin Chen is the corresponding author.
[†]Zhixiang Chen, Zhuangbin Chen, Ruoxi Jia, Wei Li, and Zibin Zheng are also with the Zhuhai Key Laboratory of Trusted Large Language Models, Sun Yat-sen University, Zhuhai, China.

ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3837550

CCS Concepts

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

Keywords

Software Testing, Fuzzing, Robotic Simulator, Isaac Sim

ACM Reference Format:
Zhixiang Chen, Zhuangbin Chen, Ruoxi Jia, Zeqin Liao, Wei Li, Jinyang Liu, and Zibin Zheng. 2026. IcFuzz: Fuzzing Isaac Sim with Semantic Stage Guidance and Multi-Level Mutation. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3832783.3837550

1 Introduction

Robots are increasingly integrated into modern society, expanding their footprint across a variety of sectors [29, 31, 61]. The global robotics market has reached $50 billion and is projected to grow substantially in the coming decade [7]. Embodied AI has recently emerged as a key paradigm that endows robots with increasingly powerful capabilities by extending artificial intelligence from the digital world to physical systems [3, 14, 34]. Given the safety risks and prohibitive costs of direct experimentation in physical environments, robotics simulators have become the indispensable infrastructure for the advancement of embodied AI. They provide safe, efficient, and scalable platforms for training, testing, and validating robotic systems before real-world deployment [3, 14, 34].

Among existing robotics simulators, NVIDIA Isaac Sim [53] stands as one of the most influential and widely used platforms for embodied AI research and development, with adoption by over 100 companies (e.g., Siemens, Boston Dynamics, and Figure AI) [43, 64, 83]. Distinguished by its GPU-accelerated physics engine and extensive library of digital assets, Isaac Sim provides high-fidelity simulation and photorealistic rendering that accurately model real-world environments and complex physical interactions [21, 34, 41]. However, the inherent complexity of Isaac Sim inevitably introduces software bugs that can compromise simulation reliability. For example, simulation crashes [25, 39] can force developers to expend

substantial effort on troubleshooting, thereby undermining overall development efficiency. When such failures occur during model training or within operational digital twin systems, they may also lead to significant data loss and severe safety problems [4, 30, 73]. Therefore, testing Isaac Sim is of paramount importance to ensure simulation reliability and to support the effective development of trustworthy embodied AI systems.

The systematic testing of Isaac Sim necessitates effective automated testing techniques. To the best of our knowledge, there is currently no testing approach dedicated to Isaac Sim. Fuzzing is a widely used method that exposes software defects by automatically generating random or malformed inputs [23, 37, 81]. It has been demonstrated to be a promising approach for uncovering previously unknown bugs in complex software systems [37, 38, 80]. However, the diverse functionality, sophisticated architecture, and complex input space of Isaac Sim pose significant challenges to the application of fuzzing, hindering the generation of effective and diverse test cases. We summarize the main challenges as follows:

- **Context-aware Object Semantics.** The declaration of objects within Isaac Sim is tightly coupled with their execution context, exhibiting complex contextual dependencies and strict logical ordering. For example, *SimulationApp* [50] (which initializes the simulator runtime) must be instantiated prior to any other simulator-specific objects. Similarly, an *Articulation* [48] can only be declared based on pre-existing *prims* [55] (basic scene elements identified by paths in the scene, such as a robot arm segment), since it is a high-level wrapper over one or more *prims*. Inputs that violate these semantics (e.g., declaring an *Articulation* before any *prims* exist) are immediately rejected, thereby restricting the exploration of the simulator's internal logic.
- **Hierarchical Simulation Control.** Simulation control in Isaac Sim is hierarchically structured across multiple granularities. For example, adding a robot requires instantiating a specific object, such as a *WheeledRobot* [52] or a *Manipulator* [51]. Each type of object supports its distinct operations (e.g., setting wheel speeds for a *WheeledRobot* or configuring joint positions for a *Manipulator*). Each operation further requires its own parameters: wheel control requires separate velocities for each wheel, whereas joint control requires precise coordinate values. Control at different granularities exerts varied influences on the simulator. Consequently, systematic testing of Isaac Sim requires test generation across multiple levels of granularities.
- **Vast Simulation State Space.** Isaac Sim possesses a vast simulation state space, including diverse models (e.g., geometries, robots), complex physical interactions (e.g., collisions, lighting), and sophisticated AI workflows (e.g., reinforcement learning). The combinatorial explosion induced by these elements renders exhaustive exploration computationally infeasible. Random fuzzing typically struggles to reach deep or subtle states that may potentially contain bugs. Therefore, effective strategies are required to guide fuzzing and ensure efficient exploration.

Given these challenges, existing fuzzing approaches demonstrate limited efficacy when applied to Isaac Sim. General-purpose fuzzers (e.g., AFL++ [15], Atheris [20]) generate test cases mainly at the level of flat byte streams. They either 1) mutate entire inputs (e.g., via byte insertion/deletion), which are often rejected due to violations of the simulator's format constraints and semantic dependencies, or 2) produce random parameters only for manually specified targets (e.g., functions), thereby enabling only single-granularity testing over limited functionality. GzFuzz [63] is the state-of-the-art (SOTA) approach for fuzzing robotic simulators, specifically designed for Gazebo [28]. It fuzzes Gazebo via a command-line interface and leverages reinforcement learning to generate command sequences. These sequences primarily manipulate (e.g., adding, deleting, or moving) entire models (i.e., objects) and plugins (i.e., components providing custom functionalities) within simulation. However, finer-grained simulation control is not captured (e.g., plugin-provided functionalities and their associated parameters) and therefore remains underexplored. For Isaac Sim, which especially features a more complex architecture and richer functionality [21], such command sequences can explore only a small subset of its full capabilities [57]. Consequently, both general-purpose fuzzers and GzFuzz cannot well handle all three challenges, which limits their ability to effectively explore the vast simulation state space of Isaac Sim.

In this paper, we propose IcFuzz, the first fuzzing approach for Isaac Sim to the best of our knowledge. IcFuzz systematically tests the simulator by generating diverse and semantically valid simulation programs, which encompass the entire execution cycle of automated robotic simulations. Our key insight is that these programs inherently follow a recurring semantic structure, which can be leveraged to constrain and guide mutation. To address the challenge of context-aware object semantics, IcFuzz explicitly identifies these structures as semantic stages, and then leverages an LLM to perform *semantic stage segmentation*, decomposing each seed into code segments aligned with domain-specific execution stages. Such a design is grounded in the official robot development lifecycle [56] and reflects how developers typically construct simulation workflows in practice. Building on this, IcFuzz further utilizes a *context-aware object selection* strategy, which restricts the set of candidate objects according to the corresponding semantic stage, preserving semantic integrity while maintaining object diversity. To systematically exercise the hierarchical simulation control, IcFuzz designs *multi-level mutation* operators that combine program analysis and LLMs to mutate simulation objects, their supported operations, and associated arguments. Furthermore, IcFuzz employs a multi-armed bandit (MAB) algorithm to guide the scheduling of these mutation operators, enabling efficient exploration of the vast simulation state space in Isaac Sim to maximize code coverage and bug detection. In this work, IcFuzz focuses on crash bugs as the test oracle, as they represent a severe and common class of failures in Isaac Sim.

We conduct a comprehensive evaluation of IcFuzz and compare it against Atheris, a widely used general-purpose fuzzer for Python, and GzFuzz, the SOTA fuzzing approach for robotic simulators. Experimental results show that IcFuzz achieves an average code coverage of 20,771 lines, which is approximately 205% and 190% of Atheris and GzFuzz, respectively. IcFuzz detects an average of 7 crashes over three rounds of 12-hour tests (3.7 unique crashes after manual inspection), outperforming both baselines. More importantly, IcFuzz successfully reported 11 bugs, 9 of which have been confirmed or fixed by developers. Ablation studies further validate the necessity and effectiveness of each key component within IcFuzz.

The main contributions of this work are summarized as follows:

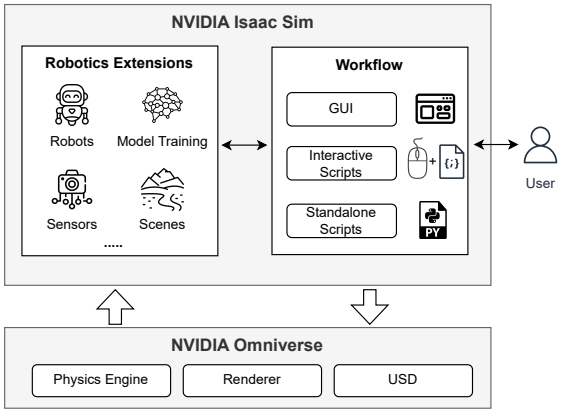

Figure 1: Overview of the Isaac Sim architecture.

- We propose IcFuzz, the first fuzzing approach for Isaac Sim. IcFuzz introduces semantic stage segmentation and context-aware object selection to handle the complex object semantics in simulation. It devises multi-level mutation operators to systematically exercise hierarchical simulation control across multiple granularities. Furthermore, IcFuzz utilizes an MAB algorithm to schedule mutation operators for efficient exploration of the vast simulation state space in Isaac Sim.
- We present semantic stage segmentation as a novel semantic constraint mechanism to support the generation of semantically valid and diverse test cases.
- Experimental results show that IcFuzz outperforms current SOTA baselines in code coverage and bug detection. IcFuzz detected 11 bugs within approximately four months, 9 of which have been confirmed or fixed.
- We implement IcFuzz and release the replication package at [6].

2 Background and Motivation

Overview of Isaac Sim. As shown in Fig. 1, Isaac Sim is a robotics simulation platform built on the closed-source NVIDIA Omniverse [54], which provides fundamental capabilities such as physics simulation, rendering, and scene representation based on Universal Scene Description (USD) [68]. Built upon this foundation, Isaac Sim primarily consists of a set of robotics-specific extensions, such as robots, sensors, scenes, and model training modules. These extensions are open-source and encapsulate the core functionalities required for robotic simulation in Isaac Sim. They can be accessed separately through multiple workflows, namely a graphical user interface (GUI), interactive scripts, and standalone scripts [58]. The GUI provides a visual environment for developers to manipulate virtual scenes, such as assembling robots and attaching sensors. Interactive scripts run asynchronously within the GUI, enabling on-demand script execution with real-time visual feedback. Standalone scripts are complete Python programs that precisely control the simulation process (e.g., rendering steps) and constitute the primary modality for automated simulation tasks (e.g., model training and testing). Therefore, we adopt standalone scripts as the fuzzing input in this work. Fig. 4 presents an example of a standalone Isaac Sim script.

The test subject of IcFuzz is the entire Isaac Sim simulator stack. During fuzzing, each input script exercises both the open-source extensions and the underlying closed-source Omniverse foundation. For coverage measurement, we collect coverage feedback only from the open-source Isaac Sim extensions. As these extensions encapsulate the core functionalities required for robotic simulation, their coverage offers a meaningful signal for guiding the fuzzing process.

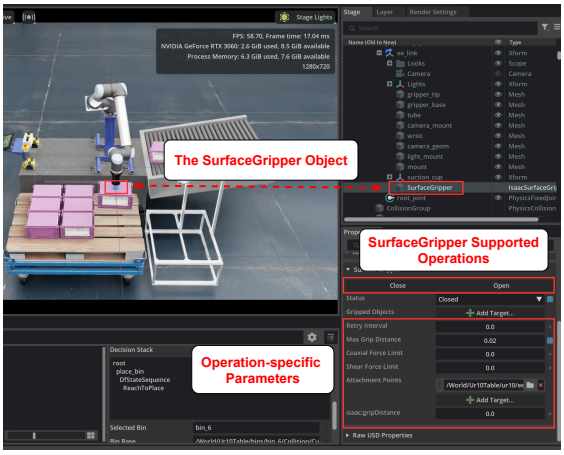

Figure 2: The *SurfaceGripper* in Isaac Sim.

Challenges in Fuzzing Isaac Sim. Fuzzing Isaac Sim is challenging due to the inherent characteristics of the simulator. First, Isaac Sim enforces Context-aware Object Semantics, where object declarations are tightly coupled with the execution context and must follow strict semantic dependencies and logical ordering. Take Fig. 4 as an example, an *Articulation* references prim paths of existing robots (e.g., "*Franka_1*" and "*Franka_2*") for unified control, while an *IMUSensor* attaches to "*Franka_1*", illustrating prim-path dependencies. During the interacting stage, joint positions and sensor data are accessed via the instantiated *Articulation* and *IMUSensor*, reflecting object dependencies. The chain-like logical ordering is enforced throughout the simulation lifecycle, e.g., the *SimulationApp* must be initialized first, followed by the scene setup, before any objects can be meaningfully added or controlled. Inputs that violate these object semantics are rejected, resulting in shallow exploration.

Simulation control in Isaac Sim is hierarchically structured across multiple granularities. Fig. 2 shows a snapshot of a warehouse picking scenario, which involves diverse objects such as the *SurfaceGripper* attached to a robotic arm, the pallet with boxes, etc. Each object supports its distinct operations. For instance, the *SurfaceGripper* enables "*Close*" for engaging grasp and "*Open*" for release. Each operation requires specific parameters, such as the "*max grip distance*" (the maximum allowable distance between the gripper and the target for a successful grasp). Hierarchical Simulation Control at these levels impacts the simulator differently, necessitating fully testing Isaac Sim across different granularities. Moreover, Vast Simulation State Space in Isaac Sim arises from the interplay of numerous objects, their supported operations, and operation-specific parameters. As a result, random fuzzing without appropriate guidance strategies tends to remain at shallow or repetitive states, making it difficult to reach deep or subtle states that may trigger bugs.

Together, these challenges motivate IcFuzz to generate semantically valid inputs, exercise simulation control at multiple granularities, and efficiently explore deep and diverse simulation states.

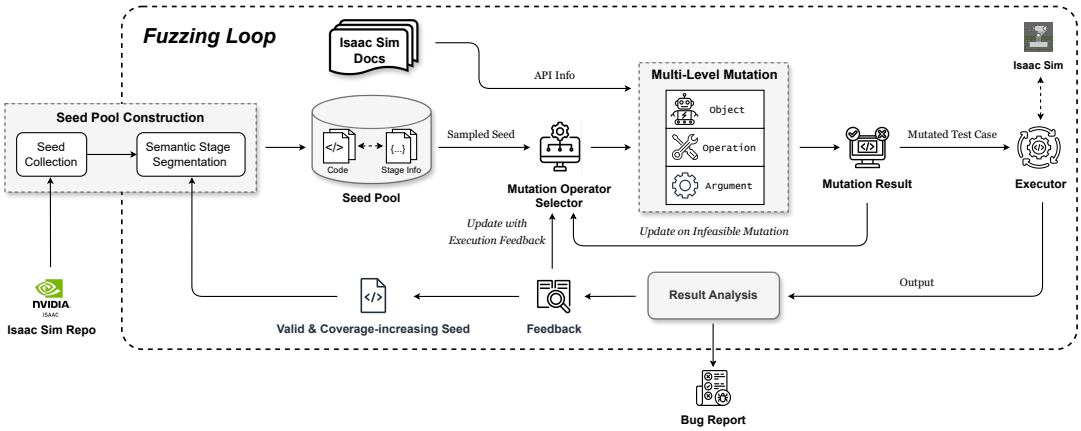

Figure 3: Overall Workflow of IcFuzz.

3 Methodology

Fig. 3 illustrates the overall workflow of IcFuzz. The key insight is that simulation programs inherently follow a structured lifecycle of semantic stages. Based on this observation, IcFuzz introduces *semantic stage segmentation* (Sec. 3.1.2) to decompose each seed into code segments aligned with domain-specific execution stages. The resulting stage annotations serve as explicit semantic constraints that guide context-aware object selection (Sec. 3.2.1) during mutation, preserving semantic validity while maintaining diversity.

Given a sampled seed (Sec. 3.1.3), IcFuzz applies a multi-level mutation operator (Sec. 3.2.2) that systematically exercises the hierarchical simulation control of Isaac Sim. The operators are adaptively scheduled based on an MAB algorithm [2] to efficiently explore the state space. IcFuzz assesses semantic feasibility during mutation generation, discarding infeasible results to reduce costly invalid executions while updating the selector. Finally, IcFuzz executes the feasible mutated seed, analyzes the result, records any detected bugs, and updates the selector (Sec. 3.3). Valid seeds that increase coverage undergo stage segmentation and re-enter the seed pool; this coverage-guided loop repeats until the time budget is reached.

3.1 Seed Pool Construction

We initialize the seed pool with valid Isaac Sim inputs, recording each seed alongside its segmented semantic stage information.

3.1.1 Seed Collection. Isaac Sim can be driven by standalone Python scripts [58], which serve as individual test cases (see Sec. 2). We collect all standalone scripts available in the official Isaac Sim repository [24]. Some scripts require additional dependencies (e.g., robot model descriptions), which makes them difficult to run and mutate in isolation. As a result, we execute each script independently and retain only those that complete successfully without any errors. This process yields 116 seeds, which form a reliable set of initial test cases. These cases are broadly representative of real-world simulator usage and encompass a wide spectrum of functionalities. They cover 20 distinct Isaac Sim packages (e.g., *isaacsim.core*) across 8 major functional categories (e.g., robot manipulation), involving 9 robot platforms (e.g., Franka) and 6 sensor modalities (e.g., IMU), including tutorials and benchmarks maintained in the official repository.

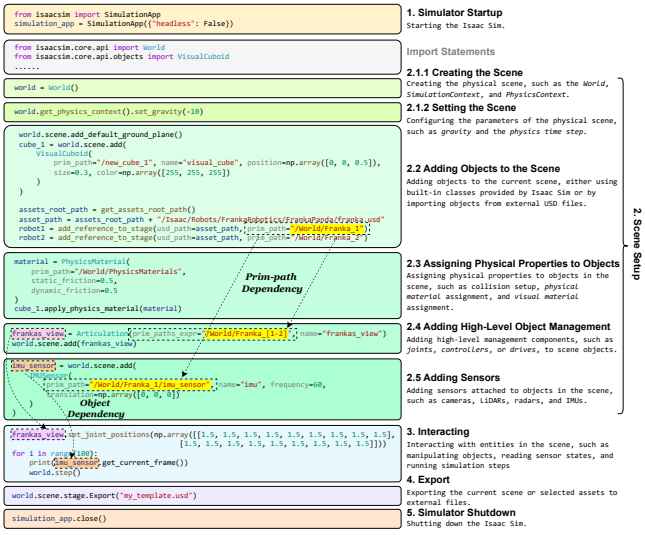

Figure 4: An example of semantic stage segmentation for a standalone Python script in Isaac Sim.

3.1.2 Semantic Stage Segmentation. Object selection is crucial for fuzzing Isaac Sim, as it directly influences the testing scope and the diversity of generated scenarios. However, valid object selection is challenging because object declarations must satisfy strict contextual dependencies and logical ordering constraints (see Sec. 2).

Our key observation is that Isaac Sim programs are not monolithic: they inherently follow a recurring *semantic stage structure* that mirrors the robot development lifecycle. By explicitly identifying this structure, the open-ended problem of ensuring semantic validity can be reduced to a *stage-constrained* problem. Therefore, we propose segmenting each seed into semantic stages based on domain knowledge, which captures context-aware object semantics. Subsequent steps can leverage this information to constrain the set of permissible objects (see Sec. 3.2.1), enabling semantic stage-guided mutations. Code segmentation has demonstrated its effectiveness in other software engineering tasks such as code summarization [67], version identification [19], and improving code

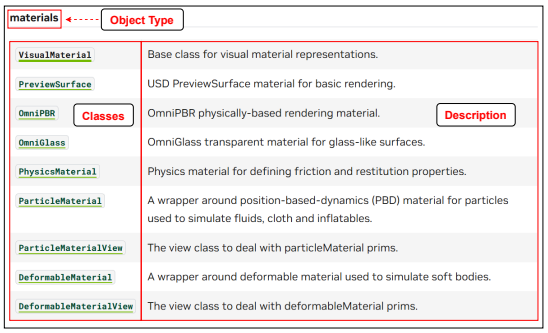

Figure 5: An excerpt from the Isaac Sim documentation [47].

readability [13, 40, 75]. IcFuzz presents a novel use of this technique as a semantic constraint mechanism for fuzzing.

To derive a standardized and unified stage taxonomy, we systematically reviewed standalone scripts in the official Isaac Sim repository [24] and the reference architecture outlining the robot development lifecycle [56], to abstract common execution patterns. Based on this analysis, we consolidate the most critical and representative operations into a set of semantic stages, as shown in Fig. 4. An Isaac Sim test case starts by initializing *SimulationApp* to launch the simulator before any simulator-specific import statements or operations (Stage 1). Execution then proceeds to *Scene Setup* (Stage 2), which includes creating the physical scene (Stage 2.1.1), configuring its parameters (Stage 2.1.2), and adding objects (Stage 2.2). The script then continues to *Assigning Physical Properties* (Stage 2.3), *Adding High-Level Object Management* (Stage 2.4), and *Adding Sensors* (Stage 2.5) to scene objects. Although the three stages (Stages 2.3-2.5) rely on pre-existing objects (e.g., an *Articulation* groups two *Franka robots* together), no strict ordering is imposed among them. This design aligns with procedural flexibility in practice. After completing *Scene Setup* (Stage 2), the script further interacts with entities in the scene (Stage 3). Finally, the current stage is exported to a USD file (Stage 4) and the simulator is shut down (Stage 5).

As LLMs have demonstrated strong capabilities in code understanding [26, 42, 65], we leverage them to automatically segment each seed into its semantic stages. We design a chain-of-thought prompt [77] that contains the complete seed, descriptions of each semantic stage, and an example output, instructing an LLM for accurate segmentation (full prompt available at [6]). Both the seed and its segmented stage information are finally stored in the seed pool.

3.1.3 Seed Sampling. IcFuzz samples a seed at the beginning of each fuzzing iteration. To ensure sufficient exploration of the seed pool and avoid repetitive selection of certain seeds, we follow the seed sampling strategy in [76]. Each seed is assigned a weight inversely proportional to its historical sampling frequency, and these weights are normalized to determine the selection probability. Thus, frequently selected seeds have lower chances of being chosen, while less frequently sampled seeds are prioritized. Newly added seeds during fuzzing can also be sampled for further mutations.

3.2 Multi-Level Mutation

We design multi-level mutation operators that act on seeds at the object, operation, and argument levels to systematically test Isaac Sim.

Table 1: Addable and replacement object types across different semantic stages in Isaac Sim.

Semantic Stages	Addable Object Types	Replacement Object Types
1. Simulator Startup	–	–
2.1.1 Creating the Scene	–	–
2.1.2 Setting the Scene	–	–
2.2 Adding Objects to the Scene	Objects, Robots, Cloners, Lights, Meshes, Shapes, DataLogger	Objects, Robots, Lights, Meshes, Shapes
2.3 Assigning Physical Properties	Materials	Materials
2.4 High-Level Object Management	Robots, Controllers, Grippers, Single Prims, Prims, Manipulators, Wrappers	Robots, Controllers, Grippers, Single Prims, Prims, Manipulators, Wrappers
2.5 Adding Sensors	Sensors	Sensors
3. Interacting	–	–
4. Export	Writers	Writers
5. Simulator Shutdown	–	–

3.2.1 Context-aware Object Selection. Among the three levels of mutation, object-level mutation is the most critical. By altering the composition of objects in the scene, it largely shapes the testing scope and scenario diversity, and thus forms the foundation for the other two levels of mutation. However, as discussed in Sec. 2 and Sec. 3.1.2, object selection is non-trivial: it requires determining which objects can be validly added to a scene and where they can be introduced in the code while preserving semantic correctness. To this end, we identify a stage-specific set of permissible objects for each semantic stage, thereby mitigating the complexity arising from context-aware object semantics.

Isaac Sim groups classes with similar functionalities into distinct categories, which we refer to as *object types*. For example, Fig. 5 illustrates various classes under the *Materials* category, which can be instantiated as concrete objects and applied to prims to endow them with different physical properties. We systematically reviewed all object types within the documentation to identify those eligible for addition to a scene or replacement of existing ones at each stage, where three authors independently annotated the mappings, clarified ambiguous cases by executing minimal scripts, and resolved inconsistencies through discussion. The results are presented in Table 1. For instance, during the *Simulator Startup* stage, no additional objects are expected to be introduced. Therefore, both entries in the table are left empty. In contrast, at the *Assigning Physical Properties* stage, any class in the *Materials* category can be instantiated and then either added to the scene or used to replace an existing material object. During object-level mutation, we select a target object by randomly sampling from the set of permitted objects in Table 1, according to the applied mutation operator and the semantic stage.

This design is motivated by the observation that seed scripts are already semantically valid and typically follow these semantic stages. Therefore, applying functionally similar mutations within the same stage (e.g., replacing one material with another) is likely to preserve semantic validity, as the preceding stages have already established the required context (e.g., objects are added to the scene before physical properties are assigned).

3.2.2 Mutation Operators. We begin by crawling the full documentation of all classes in Isaac Sim, including their methods and associated object-type information, and store them in a database for use during mutation. The mutation operators integrate program analysis with LLMs to generate context-consistent code. For operators that require selecting an existing object from a seed for mutation, we adopt a **program-analysis-based seed object selection** strategy. Specifically, we extract the abstract syntax tree (AST) of the seed and

Object Addition Mutation Prompt

You are an expert Isaac Sim developer and software testing assistant. Your task is to mutate the provided Isaac Sim standalone Python script by referencing the provided API documentation. Specifically, you must **instantiate and insert** the designated class from the API documentation into the specified code stage within the script.

Requirements:
1. **Preserve Original Logic**: Do not remove the original functionality of the script, only extend or mutate it.
2. **Reference the API Documentation**: When adding the class, follow the documented constructor, parameters, and example usage as closely as possible.
3. **Feasibility Check**: Before applying the mutation, check whether the mutation is feasible.
   - If the mutation is **not feasible**, set `"mutation_feasible": false` and return an *empty code* string.
   - If the mutation is **feasible**, set `"mutation_feasible": true` and return the mutated code normally.
4. **Check the Prim Path**: When introducing the class, ensure that its prim_path is valid.
5. **Add to the Specified Code Stage**: Insert the new code snippet within the designated code block for mutation in the script to ensure it integrates naturally with the program flow.
6. **Keep Executable**: Ensure the mutated script remains a valid, executable Isaac Sim standalone script.

Response Format:
Please output the mutated Python code in JSON format without any explanations or additional text. The response should be a complete, runnable Isaac Sim Python standalone program.
{EXAMPLE_OUTPUT}

API Documentation: {API_DOCS}
Source Code: {SOURCE_CODE}
Stage for Mutation: {STAGE_KEY} {STAGE_CODE}

Figure 6: Simplified prompt for object addition mutation.

analyze import statements, constructor invocations, and variable assignments to obtain the objects available either in a specific semantic stage or across the entire seed. Each extracted object is queried against the Isaac Sim documentation, and we randomly sample an object from those that can be successfully found. This ensures that the selected object is relevant to Isaac Sim, rather than a trivial one. For object replacement and deletion, we further protect simulation-essential objects: *SimulationApp*, *World*, and *AppFramework* are not selected as mutation targets. Note that this seed object selection identifies an existing object already present in the seed, whereas *context-aware object selection* in Sec. 3.2.1 determines suitable new objects from the documentation to be introduced into the seed.

Object-Level Mutation Operators. Object-level mutation modifies the composition of objects in the scene. We define three operators to explore diverse objects and their compositions.

- **Addition** inserts a new object into a seed. Based on the seed's semantic stage information, we randomly select a target stage from those both present in the seed and associated with addable object types (i.e., Stages 2.2–2.5 and Stage 4). We then select a target object via *context-aware object selection* (see Sec. 3.2.1) and retrieve the constructor of its class from the documentation. Finally, we compose a prompt (see Fig. 6) and feed it to an LLM to instantiate the target object and insert its code into the seed.
- **Replacement** replaces an existing object with another object. First, we select the target stage for mutation in the same manner as in *addition*, except that we focus on replacement object types when referring to Table 1. We then select the original object from the target stage using the *program-analysis-based seed object selection* strategy, and a replacement object via the *context-aware object selection strategy* (see Sec. 3.2.1). Finally, we compose a prompt and feed it to an LLM. Different from *addition*, the LLM is also responsible for replacing dependencies associated with the original object (e.g., method invocations), or removing them when replacement is infeasible.
- **Deletion** removes an object from a seed. It follows the *program-analysis-based seed object selection* to identify the target object, and instructs an LLM to remove it along with its dependents.

Operation-Level Mutation Operators. Operation-level mutation mutates the supported operations of existing objects, which are concretely manifested as method invocations in seed scripts. We define three operators to explore diverse operations scattered across different objects, as well as their invocation sequences. All three operators begin by selecting a target object via the *program-analysis-based seed object selection* strategy. For replacement and deletion, the *close* methods of *SimulationApp* and *AppFramework* are excluded from target selection to ensure proper simulation termination.

- **Addition** randomly samples a method from the documentation of the selected object's class, then constructs a prompt incorporating the seed and the documentation snippet of the sampled method, and feeds it to an LLM to generate the insertion.
- **Replacement** follows the same method-sampling procedure as *addition*, then selects an existing invocation of the target object and instructs an LLM to replace it with the sampled method.
- **Deletion** randomly selects an existing invocation of the target object and instructs an LLM to remove it from the seed.

Argument-Level Mutation Operators. Argument-level mutation mutates the arguments of object operations, representing the finest mutation granularity. Given a seed, we select a target object via the *program-analysis-based seed object selection* strategy, then randomly choose one of its methods that accepts at least one argument. We collect each parameter's type, default value, and nullability from the documentation. Each argument is mutated with 50% probability using a randomly chosen operator. All argument-level mutations are performed directly on the AST, without using LLMs.

- **Deletion** removes a selected argument from the method call. This operator is applicable only if the argument has a documented default value.
- **Nullification** sets the value of a selected argument to `None`, which can be applied only if the documentation indicates that the argument allows `None` as a valid value.
- **Replacement** modifies the value of a selected argument. This operator is applicable only to arguments of numeric types (e.g., int, float), boolean type (i.e., bool), or lists/arrays containing numeric/boolean elements. For numeric arguments/elements, a new value is randomly generated to replace the original value. For boolean arguments/elements, the value is negated.

Mutation Feasibility Analysis. Executing a test case in Isaac Sim is time-consuming. For example, fully loading a warehouse sample scene can take more than 30 seconds [45]. Therefore, we analyze the feasibility of each candidate mutation and discard infeasible ones when applying mutation operators, thereby reducing the time wasted on executing invalid test cases.

Specifically, we perform two types of feasibility analyses: **1) Rule-based analysis.** This analysis checks whether a seed satisfies the prerequisites of the selected mutation operator. For example, object addition/replacement operators require the seed to contain a semantic stage in which the set of addable/replaceable objects is non-empty (see Table 1), whereas argument-level mutation operators require the seed to include Isaac Sim-specific method invocations with arguments of mutable types. The mutation is deemed infeasible when the prerequisites are not met. **2) LLM-based analysis.** For mutation operators that rely on LLM-based generation (i.e., object- and operation-level mutation operators), we instruct

Algorithm 1: UCB-based Mutation Operator Selection

Input: Mutation operator set $O = \{o_1, \ldots, o_K\}$
1 **Function** *Initialize()*
2 **foreach** *operator* $o_i \in O$ **do**
3 $n_i \leftarrow 0$; // selection count
4 $R_i^{\text{sum}} \leftarrow 0$; // cumulative reward
5 $\text{UCB}_i \leftarrow +\infty$; // force initial exploration
6 $N \leftarrow 0$; // total selections
7 **Function** *SelectOperator()*
8 $i^\star \leftarrow \arg\max_i \text{UCB}_i$;
9 **return** $o_{i^\star}$;
10 **Function** *Update*(o_i, *NewCoverageGain*, *CrashFound*)
11 $R \leftarrow w_{\text{cov}} \times \text{NewCoverageGain} + w_{\text{crash}} \times \text{CrashFound}$;
12 $N \leftarrow N + 1$; $n_i \leftarrow n_i + 1$; $R_i^{\text{sum}} \leftarrow R_i^{\text{sum}} + R$; $\bar{R}_i \leftarrow R_i^{\text{sum}}/n_i$;
13 $\text{UCB}_i \leftarrow \bar{R}_i + \sqrt{\frac{2 \ln N}{n_i}}$;

the LLM to determine whether a semantically valid script (i.e., one that can be executed successfully without errors) can be produced under the given mutation instruction. Leveraging their strong code understanding capabilities, LLMs serve as a practical choice to flexibly reason about the simulator's dynamic semantics and contextual compatibility, avoiding the laboriously crafted rules required by traditional program analysis. An example prompt is shown in Fig. 6. For instance, the LLM may reject a mutation when the object to be added is incompatible with the existing context of the seed. If either type of feasibility analysis determines that a mutation is infeasible, we discard it without generating the mutated seed.

Mutation Operator Selection. Selecting mutation operators is a sequential decision-making problem under uncertainty, where operator effectiveness is only revealed after costly executions. With the goal of maximizing code coverage and bug detection under a limited budget, the selector must balance exploring less-used operators and exploiting those with demonstrated effectiveness. This trade-off naturally motivates modeling the selection as an MAB problem.

Specifically, we formulate the mutation operator selection as a K-armed bandit problem. The set of mutation operators $O = \{o_1, \ldots, o_K\}$ corresponds to the K distinct arms. The fuzzing process evolves over a sequence of discrete time steps $t = 1, 2, \ldots, N$. At each step t, the selector chooses an operator o_i to apply to a seed, an action equivalent to pulling the i-th arm of the bandit. The execution outcome serves as the stochastic reward, with the objective of maximizing the cumulative reward over the total selection horizon.

We solve this problem using the Upper Confidence Bound (UCB) algorithm [2], as detailed in Algorithm 1. In each iteration, we select the operator $o_{i^\star}$ with the highest UCB score (Lines 7–9), which represents the largest upper confidence bound on the expected reward of the operator. The reward function R (Line 11) is computed as a weighted sum of coverage expansion and crash discovery:

$$R = w_{\text{cov}} \times \text{NewCoverageGain} \; + \; w_{\text{crash}} \times \text{CrashFound}, \quad (1)$$

where NewCoverageGain and CrashFound are binary indicators: NewCoverageGain is set to 1 when the mutated seed is valid (i.e., executes successfully without errors) and increases cumulative code coverage, and CrashFound is set to 1 when a crash is triggered. The UCB score (Line 13) of the selected operator is updated as follows:

$$\text{UCB}_i = \bar{R}_i + \sqrt{\frac{2 \ln N}{n_i}}, \quad (2)$$

where $\bar{R}_i$ represents the empirical mean reward of operator o_i (favoring exploitation), while the second term provides an exploration bonus that is positively correlated with the total number of selections N and negatively correlated with the operator's historical selection count n_i, ensuring that less-frequently selected operators still have a chance to be selected.

3.3 Oracle and Feedback

IcFuzz executes mutated test cases in Isaac Sim and analyzes their execution results to identify potential bugs. Regarding the test oracle, we focus on crash bugs occurring during program execution, following prior work [63]. Such bugs cause abrupt termination of the simulation, potentially leading to loss of training data or safety risks, and therefore represent a severe class of failures. In addition, crash bugs constitute a common bug pattern in Isaac Sim and are relatively easy to reproduce and localize, making them a practical entry point for improving the reliability of Isaac Sim. Non-crashing bugs, such as inaccurate sensor readings or violations of physical constraints, are not covered by the current oracle. Extending IcFuzz to detect such bugs would require defining domain-specific semantic oracles (e.g., physics invariant checkers), instrumenting the simulator to collect richer runtime states, and incorporating semantic deviation signals into the feedback mechanism. We leave such extensions to future work.

IcFuzz leverages execution feedback to guide its fuzzing process. For each test case, IcFuzz determines whether it executes successfully and measures the code coverage. A test case that completes without errors is considered valid. If a valid test case further increases coverage, it is subjected to semantic stage segmentation and added to the seed pool. Meanwhile, execution feedback, including validity, coverage gain, and crash occurrence, is fed back to the mutation operator selector. It updates the UCB score of each mutation operator accordingly to support adaptive scheduling.

4 Evaluation

In this section, we evaluate IcFuzz to answer the following research questions (RQs):

- **RQ1 (Bug Detection)**: How does IcFuzz perform in detecting bugs for Isaac Sim?
- **RQ2 (Effectiveness)**: How effective is IcFuzz in its overall fuzzing performance and LLM-dependent intermediate steps?
- **RQ3 (Ablation Study)**: What are the contributions of the key components in IcFuzz?

4.1 Experimental Setup

Baselines. We select GzFuzz [63] and Atheris [20] as baselines. GzFuzz is the SOTA approach for fuzzing robotic simulators and is designed for Gazebo. To apply it to Isaac Sim, we make minimal modifications to its source code, primarily adapting its simulator interaction layer. Atheris is a widely used general-purpose fuzzer for Python. Its fuzz testing is implemented by mutating input USD files, following the general-purpose fuzzer implementation

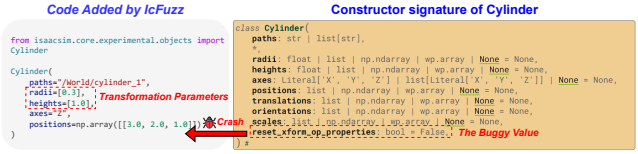

Figure 7: A bug triggered by adding a *Cylinder*.

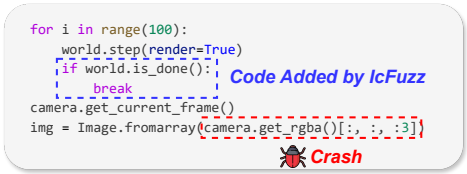

Figure 8: A bug triggered by an unexpected value returned by the camera.

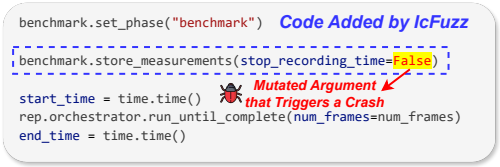

Figure 9: A bug triggered by an incompletely implemented argument.

in [63]. Both fuzzers are run with their default settings. Although grammar-based fuzzers and search-based test generation tools can produce structured test inputs, we do not include them as baselines. Grammar-based fuzzing [1, 74] generates or mutates inputs according to predefined rules, and adapting it to Isaac Sim would require substantial expert effort to define input grammars; moreover, such grammars mainly capture syntactic structure, are hard to extend to simulator-specific semantics, and also limit test diversity. Representative search-based tools (e.g., EvoSuite [17] and Pynguin [35]) mainly generate unit-level tests for classes, functions, or modules, whereas testing the entire Isaac Sim requires interactions across multiple modules, and individual modules in Isaac Sim are difficult to load and execute independently due to complex preconditions.

Implementation of IcFuzz. IcFuzz is implemented in approximately 4,500 lines of Python code [6]. We use SQLite [22] to manage the seed pool and store the Isaac Sim documentation. By default, we use GPT-5-mini [59] for semantic stage segmentation and mutation, and set the UCB reward weights to $w_{\text{cov}} = 1$ and $w_{\text{crash}} = 5$.

Environment. Experiments are conducted on a Windows 10 desktop with an Intel Core i5-13400F CPU at 2.50 GHz, 32 GB of RAM, and an RTX 3060 GPU. RQ1 is evaluated on Isaac Sim 5.0.0 and 5.1.0, while RQ2 and RQ3 use only version 5.1.0, the latest stable release.

Metrics. We detail the metrics used in our study as follows.

- **Number of total/unique crashes.** We report total and unique crash counts. Crashes in Isaac Sim exhibit explicit symptoms in the execution logs, such as messages indicating "crash detected" or the generation of dump files. For each detected crash, we collect its triggering script, exception type and message, and full stack trace. Three authors independently reproduce each crash in a clean Isaac Sim environment and label two crashes as duplicates if they share the same exception type, top stack frames up to the triggering function, and triggering condition; otherwise, they are treated as distinct. We then compare the resulting labels and resolve any inconsistencies through joint discussion until consensus is reached. Each unique crash is reported to the developers for further confirmation.
- **Number of valid/invalid scripts.** A generated script is considered valid if it executes without errors (e.g., uncaught exceptions) as indicated by its logs; otherwise, it is invalid. We also count valid scripts that further increase cumulative code coverage (i.e., the number of valid and coverage-increasing scripts).
- **Number of covered unique classes/replacement pairs.** This metric measures the number of unique classes (for object addition) or unique replacement pairs (for object replacement) that result in valid scripts after mutation. It reflects the diversity of objects effectively explored through object-level mutations.
- **Code coverage.** Code coverage has been widely used in prior software testing studies [60, 63, 70]. We follow GzFuzz [63] and adopt line coverage as our coverage criterion, which is measured using *coverage.py* [8]. We use Welch's t-test with $p < 0.05$ to determine statistical significance, following prior work [82].

4.2 RQ1: Bug Detection

To evaluate the effectiveness of IcFuzz in detecting bugs, we conducted a longitudinal fuzzing campaign over approximately four months, following [63]. We tested the latest stable version of Isaac Sim available at the time. Specifically, we initially evaluated IcFuzz on version 5.0.0 and subsequently switched to 5.1.0 after its official release. During the study, 11 bugs are reported in total, among which 7 are confirmed by the developers (including 5 that are already fixed as of this writing), 2 are still awaiting further responses, and the remaining 2 are considered stale, as they had already been fixed in the latest developer branch at the time of reporting. Next, we present representative bug cases detected by IcFuzz through object-, operation-, and argument-level mutations, respectively.

Fig. 7 illustrates Bug #280 as an example. The gray box shows the mutation code generated by IcFuzz for adding a *Cylinder* object to the scene, while the right side presents the constructor signature of the corresponding class [49]. This bug is caused by an incorrect default value of the parameter *"reset_xform_op_properties"*, which is mistakenly set to False. As a result, when a *Cylinder* object is added to the scene with transformation parameters (e.g., *"radii"* and *"heights"*), Isaac Sim crashes immediately. Notably, this bug is not limited to the *Cylinder* class; instead, it has a broad impact, affecting a total of 15 classes under the *Shapes*, *Lights*, and *Meshes* categories. These experimental classes, although expected to become the main classes in future releases [44], are only recently introduced and currently lack sufficient usage examples, and may also lack thorough testing by the project team. The developers have confirmed this bug and created an internal ticket to address it.

Fig. 8 shows Bug #243. IcFuzz detected this bug by adding a call to the "*is_done*" method of the *World* object and then prematurely terminating the simulation loop according to its return value. This prevents the simulation from completing the necessary warmup

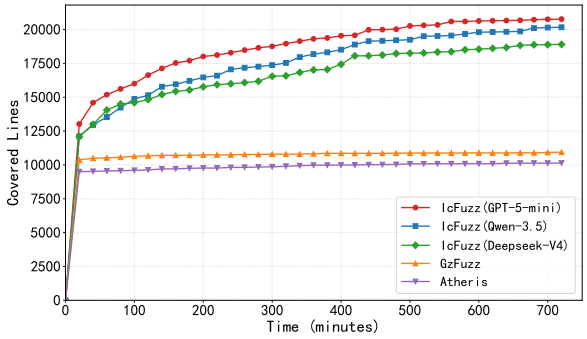

Figure 10: Average code coverage over time.

Table 2: Average crashes and coverage of IcFuzz and baselines.

Method	# Unique Crashes ↑	# Total Crashes ↑	Coverage ↑	p-value
Atheris	0	0	10136.0	4.45E-5
GzFuzz	0	0	10940.7	1.30E-6
IcFuzz (DeepSeek-V4)	**4.3**	5.3	18903.3	–
IcFuzz (Qwen-3.5)	2.3	2.7	20167.7	–
IcFuzz (GPT-5-mini)	3.7	**7.0**	**20771.0**	–

steps before valid camera data is acquired. As a result, "*get_rgba*" returns an unexpected one-dimensional array instead of the expected three-dimensional one, leading to out-of-bounds memory accesses and a crash. This behavior contradicts the original documentation, which claims that "*get_rgba*" always returns an array with the correct dimensions. According to our suggestions, the developers have updated the documentation to alert users to potential output shape issues before the renderer has fully warmed up [46].

Fig. 9 provides an overview of Bug #338. IcFuzz detected this bug through two rounds of mutation. First, it added the method call "*store_measurements*" into the seed. Subsequently, it mutated the value of the argument "*stop_recording_time*" from *True* to *False* via argument replacement, which ultimately triggered the bug. After a month-long investigation, developers confirmed that the root cause was that the functionality for "*stop_recording_time=False*" had not been fully implemented. Notably, directly calling "*store_measurements*" with the default value of "*stop_recording_time*" as *True* works correctly; the bug is only triggered when the argument is negated through mutation.

These case studies demonstrate that the triggering conditions of bugs in Isaac Sim can be intricate and may involve non-trivial interactions across object-, operation-, and argument-level mutations. This observation further justifies the multi-level mutation of IcFuzz, which systematically exercises the hierarchical simulation control of Isaac Sim and effectively detects bugs.

4.3 RQ2: Effectiveness

4.3.1 Overall Effectiveness. This section compares the performance of IcFuzz with prior work and evaluates its overall effectiveness under different LLMs and UCB reward settings. For each IcFuzz configuration and baseline, we conduct three independent 12-hour runs and report the average total/unique crashes and code coverage.

IcFuzz outperforms the baselines in both crash detection and code coverage. The results achieved by IcFuzz with the default LLM (GPT-5-mini) and by the baselines are shown in Table 2. Atheris and GzFuzz fail to detect any crashes across all runs,

Table 3: Average crashes and coverage of IcFuzz under different UCB reward settings.

w_{cov}	w_{crash}	# Unique Crashes ↑	# Total Crashes ↑	Coverage ↑
1	1	2.0	3.0	**20968.7**
1	5	3.7	**7.0**	20771.0
1	10	**4.3**	6.0	19879.0

whereas IcFuzz detects an average of seven total crashes, including 3.7 unique crashes. IcFuzz also achieves the highest code coverage, covering an average of 20,771 lines of code, which is approximately 205% of Atheris (10,136 lines) and 190% of GzFuzz (10,940.7 lines); these differences are statistically significant. Fig. 10 shows the coverage trends over time. The coverage of GzFuzz and Atheris increases rapidly at the beginning but grows slowly after approximately 100 minutes. In contrast, IcFuzz exhibits steady coverage growth from 0–400 minutes and does not plateau until around 600 minutes.

To evaluate the overall effectiveness of IcFuzz under different configurations, we replace the default LLM (GPT-5-mini) with DeepSeek-V4 [11] and Qwen3.5-397B-A17B [62], and vary w_{crash} among 1, 5 (default), and 10. The LLM variant results are included in Table 2 and Fig. 10, and the reward setting results are shown in Table 3. **The performance of IcFuzz varies slightly across different LLMs and UCB reward settings, while all configurations substantially outperform the baselines.** Across the three LLMs, code coverage ranges from 18,903.3 to 20,771.0 lines with 2.3–4.3 unique crashes detected on average. The lowest-coverage LLM configuration still achieves approximately 173% of the code coverage of GzFuzz and 187% of Atheris, and both baselines fail to trigger any crash. As w_{crash} decreases from 10 to 1, code coverage increases from 19,879.0 to 20,968.7 lines. With a lower w_{crash}, the UCB algorithm favors mutation operators that yield coverage gains, thus enabling broader code exploration. Meanwhile, increasing w_{crash} does not proportionally improve crash detection, as crash occurrences are inherently sparse and stochastic, making them difficult to reliably increase by merely biasing operator selection. We set $w_{crash} = 5$ as the default to balance coverage growth and crash detection.

4.3.2 Effectiveness of LLM-dependent Steps. This section evaluates the accuracy of two LLM-dependent intermediate steps in IcFuzz: semantic stage segmentation and LLM-based mutation feasibility analysis. For semantic stage segmentation, we collect all 454 segmentation results produced by the default IcFuzz during the experiments in Sec. 4.3.1 for both the official and newly generated seeds. We randomly sample 20% (91 results) for manual inspection. We carefully inspect each segmented stage and consider it correct only if both its code content and stage category are correct. For ambiguous cases, we consult the official documentation and reach consensus through discussion. The accuracy of each seed is calculated as the proportion of correctly segmented stages. **IcFuzz achieves an average accuracy of 93.5% in semantic stage segmentation across the sampled seeds.** The LLM's strong code-understanding capabilities effectively support the segmentation of complex Isaac Sim scripts. For example, even in a sophisticated robotic grasping scenario with heterogeneous entities and dynamic execution logic, IcFuzz correctly segments the code into six semantic stages. The remaining errors tend to occur in less common code patterns for which the

Table 4: Evaluation of object-level mutation operators with different object selection strategies.

Operator	# Valid & Cov. Inc. Scripts ↑	# Valid Scripts ↑	# Invalid Scripts ↓	# Unique Classes/Pairs↑	Coverage ↑	p-value
LLM-based Addition	**32.0**	**39.7**	**12.8**	17.0	16346.0	3.19E-3
Context-aware Addition	27.7	30.0	20.3	**27.0**	**16891.7**	–
LLM-based Replacement	7.7	8.0	41.7	7.0	13920.7	2.22E-3
Context-aware Replacement	**24.0**	**30.3**	**22.7**	**23.7**	**15188.0**	–

LLM may exhibit limited understanding. For instance, IcFuzz misclassifies camera-addition code involving complex initialization and configuration logic as *Interacting* rather than *Adding Sensors*, narrowing the mutation scope (see Table 1). Future work could inject knowledge relevant to the code context to further enhance accuracy.

LLM-based mutation feasibility analysis assesses whether a requested mutation can produce a valid test case and rejects infeasible mutations to save execution time. A false positive, where an infeasible mutation is deemed feasible, only wastes execution time without otherwise affecting the fuzzing process. In contrast, a false negative may discard a useful or crash-triggering test case. We therefore collect all 139 cases rejected as infeasible by the default IcFuzz during the experiments in Sec. 4.3.1 and randomly sample 20% (28 cases) for manual inspection. For each case, we carefully validate its mutation feasibility by attempting to construct a test case based on the source code and mutation request. We consider a rejected mutation a false negative if this process yields a valid or crash-triggering test case. **IcFuzz's mutation feasibility judgments are highly consistent with the manual validation results.** Only one false negative is identified among the 28 sampled cases. In most cases, IcFuzz can correctly assess mutation feasibility based on the available information. For example, in a Franka robotic arm control scenario, a mutation requests replacing *Articulation* with *XformPrim*. The original program relies on joint-level capabilities provided by *Articulation* (e.g., degree-of-freedom state management), whereas *XformPrim* only supports general transformation operations (e.g., setting poses). By cross-referencing the source code and API documentation, IcFuzz identifies the functional gap and correctly rejects this mutation as infeasible. The sole false negative occurs in a pick-and-place scenario, where a mutation requests removing the *BinFilling* class. Although manually deleting this class along with all its dependent variables yields a valid script, the resulting program retains only an empty *World* shell with no core control logic, providing no useful test outcome. IcFuzz conservatively rejects this mutation due to the extensive dependency chain, which exceeds what the LLM can reliably trace in a single reasoning pass. As a future improvement, incorporating static analysis tools could assist this reasoning process.

4.4 RQ3: Ablation Study

IcFuzz incorporates several key components to enhance its effectiveness, i.e., context-aware object selection, multi-level mutation, and UCB-based mutation operator selection. In this section, we evaluate the contribution of each component via ablation studies. We design four configurations for context-aware object selection, four for mutation operators, and two for operator selection strategy, yielding 10 configurations in total. Each is repeated three times and executed for two hours, resulting in 30 independent runs.

Table 5: Ablation study of mutation operators.

Method	# Valid & Cov. Inc. Scripts ↑	# Valid Scripts ↑	# Invalid Scripts ↓	Coverage ↑	p-value
IcFuzz	21.3	26.7	16.3	**16532.0**	–
IcFuzz w/o Object-level Ops	**22.3**	**27.3**	**12.0**	16043.0	2.57E-2
IcFuzz w/o Operation-level Ops	15.0	20.0	22.0	15646.0	1.82E-2
IcFuzz w/o Argument-level Ops	17.3	25.7	18.7	15991.3	6.09E-2

4.4.1 Context-aware Object Selection. We develop LLM-based addition and replacement baselines, in which the LLM directly selects objects from the seed code and the complete list of Isaac Sim classes (detailed prompts are available at [6]). We compare each baseline with IcFuzz retaining only the corresponding operator. The average results are shown in Table 4. **Context-aware object selection enables IcFuzz to cover more unique classes/pairs and achieve higher code coverage than the LLM-based baseline.** In the addition operator, although it generates 4.3 fewer valid and coverage-increasing scripts than the LLM-based method, it covers 10 more unique classes and achieves an additional 545.7 lines of code coverage. This can be attributed to the LLM's tendency to repeatedly select common classes; for example, *Camera* is selected an average of 5.7 times. For the replacement operator, context-aware object replacement outperforms the baseline across all metrics. It covers 16.7 more unique object pairs and increases code coverage by 1,267.3 lines. The coverage differences are statistically significant for both addition and replacement. In contrast, the LLM frequently generates invalid object pairs for replacement. For instance, it replaces *SimulationApp* with *AppFramework* an average of 27.7 times. Since *AppFramework* is a minimal version that lacks essential functionalities [50], such replacements fail to produce valid scripts.

4.4.2 Mutation Operators. We develop three variants of IcFuzz, each implemented by individually removing the mutation operators of a specific level. The averaged results are shown in Table 5.

Mutation operators at all three levels contribute to improving the coverage achieved by IcFuzz. IcFuzz achieves the highest code coverage (16,532 lines) compared to the variants, and removing any single level of mutation operators leads to a coverage decrease. Specifically, IcFuzz achieves coverage improvements of 489, 886, and 540.7 lines over the variants without object-, operation-, and argument-level operators, respectively. The coverage differences are statistically significant for the variants without object- and operation-level operators, while the variant without argument-level operators yields a p-value of 0.0609. The benefit stems from the systematic exercise of the hierarchical simulation control enabled by mutations at all three levels. Regarding the number of coverage-increasing valid scripts, IcFuzz generates 6.3 and 4 more such scripts than the variants without operation- and argument-level operators, respectively, indicating that the absence of the corresponding levels of mutation operators negatively affects the generation of effective scripts. IcFuzz generates slightly fewer valid scripts with coverage increase (by 1) than the variant without object-level operators, since object-level mutations are inherently difficult and result in

Table 6: Evaluation of mutation operator selection strategies.

Method	# Valid & Cov. Inc. Scripts ↑	# Valid Scripts ↑	# Invalid Scripts ↓	Coverage ↑	p-value
IcFuzz (UCB)	**21.3**	**26.7**	**16.3**	**16532.0**	–
IcFuzz (Random)	16.0	22.3	19.7	15331.3	3.55E-4

more invalid scripts (by 4.3) on average. However, enabling object-level operators facilitates exploration at an additional granularity, ultimately allowing IcFuzz to achieve the highest overall coverage.

4.4.3 Mutation Operator Selection. We develop a variant of IcFuzz that selects mutation operators uniformly at random as a baseline. The average results are reported in Table 6. **UCB-based mutation operator selection allows IcFuzz to generate more valid scripts and achieve higher code coverage than random selection.** Specifically, IcFuzz with UCB generates 4.4 more valid scripts and 5.3 more valid coverage-increasing scripts, while producing 3.4 fewer invalid scripts than random selection. The code coverage increases by 1,200.7 lines, and the difference is statistically significant.

5 Threats to Validity

Threats to internal validity mainly arise from implementation errors and the randomness of fuzzing. To mitigate potential errors, we implemented the baselines using the replication package provided by GzFuzz and the official repository of Atheris. Only necessary modifications were made to GzFuzz for adapting to Isaac Sim. Additionally, we carefully reviewed the code of IcFuzz and inspected intermediate outputs to ensure correctness. Regarding randomness, we mitigate its impact by repeating each experiment in RQ2 and RQ3 under identical settings and reporting the average results.

Threats to external validity mainly concern the generalizability of the results. Not all simulation cases strictly conform to the summarized semantic stages; instead, these stages are intended to capture critical and representative simulation workflows. Although IcFuzz is currently evaluated on Isaac Sim, its methodology is designed around characteristics common to existing robotics simulators to improve generalizability. The potentially transferable aspects of IcFuzz lie in the semantic stage-guided design, i.e., lifecycle-style stage segmentation, stage constraints for semantic validity, multi-level mutation, and feedback-based mutation scheduling, which target common properties of script-driven robotics simulators. For instance, scripts in Genesis [18], MuJoCo [10], and Webots [9] can likewise be divided into stages such as initialization and object addition, and share common object types representing simulation elements (e.g., sensors and materials). Porting IcFuzz to another simulator requires engineering effort to adapt several components, including seed pool construction, stage-to-object mapping, API documentation crawling, simulator-specific prompt rules, and the execution harness. We leave the adaptation and evaluation of IcFuzz on other robotics simulators to future work.

6 Related Work

Bug studies in robotic software. Prior work has studied robotic software bugs from various perspectives. Empirical studies have characterized bugs in the Robot Operating System (ROS) [36], including a curated dataset [72], dependency bugs [16], and interaction bugs [5]. For testing, RoboFuzz [27] leverages domain oracles and feedback to identify semantic correctness bugs in ROS through fuzzing, while R2D2 [66] guides fuzzing in ROS using callback traces. PhyFu [79] mutates physical states to detect physical law violations in physics simulation engines. GzFuzz [63] is the SOTA approach for fuzzing the robotic simulator Gazebo. It does not cover finer-grained simulation control and enables only limited exploration when applied to Isaac Sim, given its more complex architecture. In contrast, IcFuzz targets standalone scripts that enable precise simulation control in Isaac Sim and achieves more systematic testing through the proposed techniques (e.g., multi-level mutation).

Conventional automated test generation. Prior work has explored structured test input generation through grammar-based and search-based methods. Grammar-based fuzzing (e.g., Superion [74] and NAUTILUS [1]) generates or mutates inputs according to predefined rules (e.g., grammars and templates). These extensively handcrafted rules mainly capture syntactic structures and are difficult to extend to simulator-specific semantics (e.g., inter-dependencies among simulation elements). Representative search-based tools synthesize tests by optimizing objectives such as code coverage for individual classes, functions, or modules (e.g., EvoSuite [17] and Pynguin [35]). In contrast, IcFuzz fuzzes the entire simulator by generating syntactically and semantically valid test scripts.

LLMs for Software Testing. Recent studies have applied LLMs to diverse software testing targets with tailored designs. TitanFuzz [12] generates and mutates test programs for deep learning library fuzzing, GPTDroid [33] formulates mobile GUI testing as a multi-turn Q&A task, and Fuzz4All [78] achieves universal fuzzing across languages via iterative LLM-driven generation. While effective, these approaches assume that testing targets are relatively common and that LLMs have acquired sufficient knowledge during pre-training [12, 33, 78]. In contrast, IcFuzz targets Isaac Sim, which involves highly specialized robotics domain knowledge and a rapidly evolving software ecosystem [24], largely absent from pre-training. Thus, directly generating or mutating standalone scripts with LLMs is often ineffective. Instead of the costly fine-tuning to inject domain knowledge as adopted in prior work [32, 69, 71], IcFuzz enhances test effectiveness by strictly guiding, constraining, and validating LLM outputs without updating the model.

7 Conclusion

We propose IcFuzz, the first fuzzing approach dedicated to Isaac Sim. IcFuzz employs LLMs to segment seed scripts into semantic stages and uses the resulting stage information to guide context-aware object selection. It utilizes multi-level mutation operators to systematically exercise the hierarchical simulation control. IcFuzz further schedules mutation operators adaptively based on execution feedback to efficiently explore the simulation state space. Experimental results demonstrate that IcFuzz outperforms existing SOTA fuzzing approaches in both code coverage and bug detection. Our approach also provides insights into testing other robotics simulators.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (No. 62402536).

Data Availability

We release our replication package at Zenodo [6].

References

[1] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/nautilus-fishing-for-deep-bugs-with-grammars/

[2] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. 2002. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning* 47, 2-3 (2002), 235–256. doi:10.1023/A:1013689704352

[3] Roberto Bigazzi. 2025. Autonomous Embodied Agents: When Robotics Meets Deep Learning Reasoning. *arXiv* abs/2505.00935 (2025). doi:10.48550/arXiv.2505.00935

[4] Jennifer Carlson, Robin R. Murphy, and Andrew Nelson. 2004. Follow-Up Analysis of Mobile Robot Failures. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 2004)*. IEEE, 4987–4994. doi:10.1109/ROBOT.2004.1302508

[5] Zhixiang Chen, Zhuangbin Chen, Xingjie Cai, Wei Li, and Zibin Zheng. 2025. An Empirical Study of Interaction Bugs in ROS-based Software. *arXiv* abs/2507.10235 (2025). doi:10.48550/ARXIV.2507.10235

[6] Zhixiang Chen, Zhuangbin Chen, Ruoxi Jia, Zeqin Liao, Wei Li, Jinyang Liu, and Zibin Zheng. 2026. *IcFuzz*. doi:10.5281/zenodo.19244624

[7] George Chowdhury. 2025. *The Global Robotics Market Outlook*. Retrieved January 25, 2026 from https://www.abiresearch.com/blog/global-robotics-market-outlook

[8] coveragepy. 2025. *Coverage.py*. Retrieved January 25, 2026 from https://github.com/coveragepy/coveragepy

[9] Cyberbotics. 2025. *Webots User Guide: Controller Programming*. Retrieved January 25, 2026 from https://cyberbotics.com/doc/guide/controller-programming?tab-language=python

[10] DeepMind. 2025. *MuJoCo Python*. Retrieved January 25, 2026 from https://mujoco.readthedocs.io/en/stable/python.html#standalone-app

[11] DeepSeek-AI. 2026. DeepSeek-V4: Towards Highly Efficient Million-Token Context Intelligence. *arXiv* abs/2606.19348 (2026). doi:10.48550/ARXIV.2606.19348

[12] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023*. ACM, 423–435. doi:10.1145/3597926.3598067

[13] Jacob Dormuth, Ben Gelman, Jessica Moore, and David Slater. 2019. Logical Segmentation of Source Code. In *The 31st International Conference on Software Engineering and Knowledge Engineering, SEKE 2019*. KSI Research Inc. and Knowledge Systems Institute Graduate School, 717–777. doi:10.18293/SEKE2019-026

[14] Jiafei Duan, Shuang Yu, H. L. Tan, Hongyuan Zhu, and C. Tan. 2022. A Survey of Embodied AI: From Simulators to Research Tasks. *IEEE Transactions on Emerging Topics in Computational Intelligence* 6, 2 (2022), 230–244. doi:10.1109/TETCI.2022.3141105

[15] Andrea Fioraldi, Dominik Christian Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies, WOOT 2020*. USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi

[16] Anders Fischer-Nielsen, Zhoulai Fu, Ting Su, and Andrzej Wasowski. 2020. The forgotten case of the dependency bugs: on the example of the robot operating system. In *ICSE-SEIP 2020: 42nd International Conference on Software Engineering, Software Engineering in Practice, 2020*. ACM, 21–30. doi:10.1145/3377813.3381364

[17] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE 2011*. ACM, 416–419. doi:10.1145/2025113.2025179

[18] Genesis. 2025. *Hello, Genesis*. Retrieved January 25, 2026 from https://genesis-world.readthedocs.io/en/latest/user_guide/getting_started/hello_genesis.html

[19] Marcus Gerhold, Lola Solovyeva, and Vadim Zaytsev. 2024. The Limits of the Identifiable: Challenges in Python Version Identification with Deep Learning. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2024*. IEEE, 137–146. doi:10.1109/SANER60148.2024.00022

[20] Google. 2020. *Atheris: A Coverage-Guided, Native Python Fuzzer*. Retrieved January 25, 2026 from https://github.com/google/atheris

[21] Gaurav Gupta. 2024. *NVIDIA Isaac Sim: Photorealistic Rendering for Next-Gen Robot Development*. Retrieved January 25, 2026 from https://www.blackcoffeerobotics.com/blog/isaac-sim-photorealistic-rendering-for-next-gen-robot-development

[22] Dwayne Richard Hipp. 2026. *SQLite*. Retrieved January 25, 2026 from https://sqlite.org/

[23] Linghan Huang, Peizhou Zhao, Lei Ma, and Huaming Chen. 2025. On the Challenges of Fuzzing Techniques via Large Language Models. In *2025 IEEE International Conference on Software Services Engineering (SSE)*. IEEE, 162–171. doi:10.1109/SSE67621.2025.00028

[24] isaac sim. 2025. *Isaac Sim*. Retrieved January 25, 2026 from https://github.com/isaac-sim/IsaacSim

[25] jack zeng. 2024. *Training/simulation crashing, possibly due to NaN values*. Retrieved January 25, 2026 from https://forums.developer.nvidia.com/t/training-simulation-crashing-possibly-due-to-nan-values/310863

[26] Ranim Khojah, Mazen Mohamad, Philipp Leitner, and Francisco Gomes de Oliveira Neto. 2024. Beyond Code Generation: An Observational Study of ChatGPT Usage in Software Engineering Practice. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1819–1840. doi:10.1145/3660788

[27] 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 2022)*. ACM, 447–458. doi:10.1145/3540250.3549164

[28] Nathan P. 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*. IEEE, 2149–2154. doi:10.1109/IROS.2004.1389727

[29] Maria Kyrarini, Fotios Lygerakis, Akilesh Rajavenkatanarayanan, Christos Sevastopoulos, Harish Ram Nambiappan, Kodur Krishna Chaitanya, Ashwin Ramesh Babu, Joanne Mathew, and Fillia Makedon. 2021. A Survey of Robots in Healthcare. *Technologies* 9, 1 (2021), 8. doi:10.3390/technologies9010008

[30] Zeqin Liao, Zibin Zheng, Peifan Reng, Henglong Liang, Zixu Gao, Zhixiang Chen, Wei Li, and Yuhong Nan. 2025. An Empirical Study on Embodied Artificial Intelligence Robot (EAIR) Software Bugs. *arXiv* abs/2507.18267 (2025). doi:10.48550/arXiv.2507.18267

[31] Josip Tomo Licardo, Mihael Domjan, and Tihomir Orehovački. 2024. Intelligent Robotics—A Systematic Review of Emerging Technologies and Trends. *Electronics* 13, 3 (2024), 542. doi:10.3390/electronics13030542

[32] Zhe Liu, Chunyang Chen, Junjie Wang, Xing Che, Yuekai Huang, Jun Hu, and Qing Wang. 2023. Fill in the Blank: Context-aware Automated Text Input Generation for Mobile GUI Testing. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023*. IEEE, 1355–1367. doi:10.1109/ICSE48619.2023.00119

[33] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. 2024. Make LLM a Testing Expert: Bringing Human-like Interaction to Mobile GUI Testing via Functionality-aware Decisions. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024*. ACM, 100:1–100:13. doi:10.1145/3597503.3639180

[34] Liu, Yang and Chen, Weixing and Bai, Yongjie and Liang, Xiaodan and Li, Guanbin and Gao, Wen and Lin, Liang. 2025. Aligning Cyber Space With Physical World: A Comprehensive Survey on Embodied AI. *IEEE/ASME Transactions on Mechatronics* (2025), 1–22. doi:10.1109/TMECH.2025.3574943

[35] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated Unit Test Generation for Python. In *44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2022*. ACM/IEEE, 168–172. doi:10.1145/3510454.3516829

[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. doi:10.1126/scirobotics.abm6074

[37] Sanoop Mallissery and Yu-Sung Wu. 2024. Demystify the Fuzzing Methods: A Comprehensive Survey. *Comput. Surveys* 56, 3 (2024), 71. doi:10.1145/3623375

[38] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* 47, 11 (2021), 2312–2331. doi:10.1109/TSE.2019.2946563

[39] mayank.ukani. 2022. *ISAAC Sim just crashes abruptly*. Retrieved January 25, 2026 from https://forums.developer.nvidia.com/t/isaac-sim-just-crashes-abruptly/213444

[40] Richard Mayer, Michael Moser, Niklas Greif, Florian Schnitzhofer, Verena Geist, and Martin Pinzger. 2024. Evaluating AI-Based Code Segmentation for ABAP Programs in an Industrial Use Case. In *Product-Focused Software Process Improvement. Industry-, Workshop-, and Doctoral Symposium Papers - 25th International Conference, PROFES 2024 (Lecture Notes in Computer Science, Vol. 15453)*. Springer, 131–147. doi:10.1007/978-3-031-78392-0_9

[41] Mayank Mittal, Calvin Yu, Qinxi Yu, Jingzhou Liu, Nikita Rudin, David Hoeller, Jia Lin Yuan, Ritvik Singh, Yunrong Guo, Hammad Mazhar, Ajay Mandlekar, Buck Babich, Gavriel State, Marco Hutter, and Animesh Garg. 2023. Orbit: A Unified Simulation Framework for Interactive Robot Learning Environments. *IEEE Robotics and Automation Letters* 8, 6 (2023), 3740–3747. doi:10.1109/LRA.2023.3270034

[42] Daye Nam, Andrew Macvean, Vincent J. Hellendoorn, Bogdan Vasilescu, and Brad A. Myers. 2024. Using an LLM to Help With Code Understanding. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024*. ACM, 97:1–97:13. doi:10.1145/3597503.3639187

[43] NVIDIA. 2024. *NVIDIA Robotics Adopted by Industry Leaders for Development of Tens of Millions of AI-Powered Autonomous Machines.* Retrieved January 25, 2026 from https://nvidianews.nvidia.com/news/robotics-industry-development-ai-autonomous-machines

[44] NVIDIA. 2025. *Core Experimental API.* Retrieved January 25, 2026 from https://docs.isaacsim.omniverse.nvidia.com/latest/py/docs/overview/experimental.html

[45] NVIDIA. 2025. *Isaac Sim Benchmarks.* Retrieved January 25, 2026 from https://docs.isaacsim.omniverse.nvidia.com/5.1.0/reference_material/benchmarks.html

[46] NVIDIA. 2025. *Isaac Sim Camera Simulation.* Retrieved January 25, 2026 from https://docs.isaacsim.omniverse.nvidia.com/latest/py/source/extensions/isaacsim.sensors.camera/docs/index.html#isaacsim.sensors.camera.Camera.get_rgba

[47] NVIDIA. 2025. *Isaac Sim Core.* Retrieved January 25, 2026 from https://docs.isaacsim.omniverse.nvidia.com/latest/py/source/extensions/isaacsim.core.api/docs/index.html

[48] NVIDIA. 2025. *Isaac Sim Core API (Prims).* Retrieved January 25, 2026 from https://docs.isaacsim.omniverse.nvidia.com/latest/py/source/extensions/isaacsim.core.prims/docs/index.html#isaacsim.core.prims.Articulation

[49] NVIDIA. 2025. *Isaac Sim Core (Objects).* Retrieved January 25, 2026 from https://docs.isaacsim.omniverse.nvidia.com/5.1.0/py/source/extensions/isaacsim.core.experimental.objects/docs/index.html

[50] NVIDIA. 2025. *Isaac Sim Kit Helpers.* Retrieved January 25, 2026 from https://docs.isaacsim.omniverse.nvidia.com/latest/py/source/extensions/isaacsim.simulation_app/docs/index.html#isaacsim.simulation_app.SimulationApp

[51] NVIDIA. 2025. *Isaac Sim Manipulators.* Retrieved January 25, 2026 from https://docs.isaacsim.omniverse.nvidia.com/latest/py/source/extensions/isaacsim.robot.manipulators/docs/index.html

[52] NVIDIA. 2025. *Isaac Sim WheeledRobots.* Retrieved January 25, 2026 from https://docs.isaacsim.omniverse.nvidia.com/latest/py/source/extensions/isaacsim.robot.wheeled_robots/docs/index.html

[53] NVIDIA. 2025. *NVIDIA Isaac Sim.* Retrieved January 25, 2026 from https://developer.nvidia.com/isaac/sim

[54] NVIDIA. 2025. *NVIDIA Omniverse.* Retrieved January 25, 2026 from https://www.nvidia.com/en-us/omniverse/

[55] NVIDIA. 2025. *Prim.* Retrieved January 25, 2026 from https://docs.omniverse.nvidia.com/usd/latest/learn-openusd/terms/prim.html

[56] NVIDIA. 2025. *Reference Architecture and Task Groupings.* Retrieved January 25, 2026 from https://docs.isaacsim.omniverse.nvidia.com/5.1.0/introduction/reference_architecture.html

[57] NVIDIA. 2025. *ROS2 Simulation Control.* Retrieved January 25, 2026 from https://docs.isaacsim.omniverse.nvidia.com/5.1.0/ros2_tutorials/tutorial_ros2_simulation_control.html

[58] NVIDIA. 2025. *Workflows.* Retrieved January 25, 2026 from https://docs.isaacsim.omniverse.nvidia.com/5.1.0/introduction/workflows.html

[59] OpenAI. 2025. *GPT-5 mini Model.* Retrieved January 25, 2026 from https://platform.openai.com/docs/models/gpt-5-mini

[60] Junyoung Park, Yunho Kim, and Insu Yun. 2025. RGFuzz: Rule-Guided Fuzzer for WebAssembly Runtimes. In *IEEE Symposium on Security and Privacy, SP 2025*. IEEE, 920–938. doi:10.1109/SP61157.2025.00003

[61] Luca Pietrantoni, Marco Favilla, Federico Fraboni, Elvis Mazzoni, Sofia Morandini, Martina Benvenuti, and Marco De Angelis. 2024. Integrating Collaborative Robots in Manufacturing, Logistics, and Agriculture: Expert Perspectives on Technical, Safety, and Human Factors. *Frontiers in Robotics and AI* 11 (2024), 1342130. doi:10.3389/frobt.2024.1342130

[62] Qwen Team. 2026. *Qwen3.5: Towards Native Multimodal Agents.* Retrieved July 14, 2026 from https://qwen.ai/blog?id=qwen3.5

[63] Zhilei Ren, Yitao Li, Xiaochen Li, Guanxiao Qi, Jifeng Xuan, and He Jiang. 2025. Reinforcement Learning-Based Fuzz Testing for the Gazebo Robotic Simulator. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 1467–1488. doi:10.1145/3728942

[64] The Robot Report. 2024. *NVIDIA highlights Omniverse, Isaac adoption by robot market leaders.* Retrieved January 25, 2026 from https://www.therobotreport.com/nvidia-highlights-omniverse-isaac-adoption-by-market-leaders

[65] Jonan Richards and Mairieli Wessel. 2024. What You Need is what You Get: Theory of Mind for an LLM-Based Code Understanding Assistant. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2024*. IEEE, 666–671. doi:10.1109/ICSME58944.2024.00070

[66] Yuheng Shen, Jianzhong Liu, Yiru Xu, Hao Sun, Mingzhe Wang, Nan Guan, Heyuan Shi, and Yu Jiang. 2024. Enhancing ROS System Fuzzing through Callback Tracing. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024*. ACM, 76–87. doi:10.1145/3650212.3652111

[67] Aviel J. Stein and Spiros Mancoridis. 2023. Chaos to Clarity with Semantic Inferencing for Python Source Code Snippets. In *17th IEEE International Conference on Semantic Computing, ICSC 2023*. IEEE, 161–166. doi:10.1109/ICSC56153.2023.00034

[68] Pixar Animation Studios. 2025. *Introduction to USD.* Retrieved January 25, 2026 from https://openusd.org/release/intro.html

[69] Maolin Sun, Yibiao Yang, Yang Wang, Ming Wen, Haoxiang Jia, and Yuming Zhou. 2023. SMT Solver Validation Empowered by Large Pre-Trained Language Models. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023*. IEEE, 1288–1300. doi:10.1109/ASE56229.2023.00180

[70] Chenyao Suo, Junjie Chen, Shuang Liu, Jiajun Jiang, Yingquan Zhao, and Jianrong Wang. 2024. Fuzzing MLIR Compiler Infrastructure via Operation Dependency Analysis. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024*. ACM, 1287–1299. doi:10.1145/3650212.3680360

[71] Shams Tarek, Dipayan Saha, Sujan Kumar Saha, and Farimah Farahmandi. 2025. BugWhisperer: Fine-Tuning LLMs for SoC Hardware Vulnerability Detection. In *43rd IEEE VLSI Test Symposium, VTS 2025*. IEEE, 1–5. doi:10.1109/VTS65138.2025.11022958

[72] Christopher Steven Timperley, Gijs van der Hoorn, André Santos, Harshavardhan Deshpande, and Andrzej Wasowski. 2024. ROBUST: 221 bugs in the Robot Operating System. *Empirical Software Engineering* 29, 3 (2024), 57. doi:10.1007/S10664-024-10440-0

[73] Mika Viljanen. 2024. Safety by Simulation: Theorizing the Future of Robot Regulation. *AI & Society* 39 (2024), 139–154. doi:10.1007/s00146-023-01730-0

[74] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019*. IEEE / ACM, 724–735. doi:10.1109/ICSE.2019.00081

[75] Xiaoran Wang, Lori L. Pollock, and K. Vijay-Shanker. 2014. Automatic Segmentation of Method Code into Meaningful Blocks: Design and Evaluation. *Journal of Software: Evolution and Process* 26, 1 (2014), 27–49. doi:10.1002/SMR.1581

[76] Zan Wang, Ming Yan, Junjie Chen, Shuang Liu, and Dongdi Zhang. 2020. Deep learning library testing via effective model generation. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 788–799. doi:10.1145/3368089.3409761

[77] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022*. http://papers.nips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html

[78] 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 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024*. ACM, 126:1–126:13. doi:10.1145/3597503.3639121

[79] Dongwei Xiao, Zhibo Liu, and Shuai Wang. 2023. PhyFu: Fuzzing Modern Physics Simulation Engines. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023*. IEEE, 1579–1591. doi:10.1109/ASE56229.2023.00054

[80] Zhenhua Yu, Zhengqi Liu, Xuya Cong, Xiaobo Li, and Li Yin. 2024. Fuzzing: Progress, Challenges, and Perspectives. *Computers, Materials and Continua* 78, 1 (2024), 1–29. doi:10.32604/cmc.2023.042361

[81] Zhaowei Zhang, Hongzheng Zhang, Jinjing Zhao, and Yanfei Yin. 2023. A Survey on the Development of Network Protocol Fuzzing Techniques. *Electronics* 12, 13 (2023), 2904. doi:10.3390/electronics12132904

[82] Yukai Zhao, Menghan Wu, Xing Hu, and Xin Xia. 2025. HFuzzer: Testing Large Language Models for Package Hallucinations via Phrase-based Fuzzing. In *40th IEEE/ACM International Conference on Automated Software Engineering, ASE 2025*. IEEE, 2746–2758. doi:10.1109/ASE63991.2025.00225

[83] Zhehua Zhou, Jiayang Song, Xuan Xie, Zhan Shu, Lei Ma, Dikai Liu, Jianxiong Yin, and Simon See. 2024. Towards Building AI-CPS with NVIDIA Isaac Sim: An Industrial Benchmark and Case Study for Robotics Manipulation. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP 2024*. ACM, 263–274. doi:10.1145/3639477.3639740

Received 2026-03-26; accepted 2026-06-18