

Review

A Survey of Fuzzing for ROS-Based Robotic Systems

Mucheng Yang ¹, Jinyang Chen ¹, Tong Wang ^{1,*}, Taotao Gu ², Xiao Feng ³, Xiaohui Kuang ¹
and Gang Zhao ^{1,*}

¹ Institute of Systems Engineering, Academy of Military Sciences, Beijing 100101, China; 4349467ymc@gmail.com (M.Y.); cjoy020403@126.com (J.C.); xhkuang@bupt.edu.cn (X.K.)

² School of Software, Tsinghua University, Beijing 100084, China; gutaotao@mail.tsinghua.edu.cn

³ School of Cyber Science and Engineering, Southeast University, Nanjing 210096, China; fengxiao18@mails.ucas.ac.cn

* Correspondence: tongwss@foxmail.com (T.W.); bisezhaog@163.com (G.Z.)

Featured Application

The review supports researchers and practitioners who design fuzzing, runtime testing, and assurance techniques for ROS-based robotic systems.

Abstract

Fuzzing is increasingly used to test ROS-based robotic systems, yet results remain difficult to compare because studies exercise different ROS layers and define failures in different ways. We analyze 92 selected works on ROS/ROS-based fuzzing and closely related robotic, cyber-physical, and autonomous-system testing. The corpus was built through a documented search, screening, and snowballing process. Each work is interpreted by its role in the fuzzing workflow and by the strength of its ROS-specific evidence, which separates direct ROS results from near-domain evidence that can be transferred only under explicit assumptions. The review shows a shift from message-validity testing toward runtime behavior and task-level failures, but the evidence profile is uneven: input modeling, feedback, and oracle construction have advanced more clearly than repeatable execution and diagnosable failure analysis. Treating ROS fuzzing as an evidence-bound workflow clarifies which claims can be compared, where evidence remains weak, and what support is needed for reproducible and repairable ROS fuzzing results.

Keywords: ROS; fuzz testing; robotic systems; embodied intelligence; software testing; security

1. Introduction

ROS-based robotic systems combine graph-structured software with physical tasks. ROS and ROS 2 support reusable interfaces, distributed execution, middleware configuration, and simulation-based development [1–3]; the same features also create failures that may appear at message, runtime, middleware, simulator, or task level. Fuzzing can expose unexpected behavior in such systems, but many fuzzing techniques still assume a single program and a serialized input [4–6]. ROS fuzzing has to drive distributed nodes and judge failures in runtime and task context.

The resulting review problem is comparability. General fuzzing surveys explain workflow mechanisms, protocol and stateful-system surveys clarify message and state exploration, and robotics or ROS surveys describe the system and testing context [5–9]. None of these views alone separates ROS target layers, fuzzing workflow roles, and evidence



Received: 15 June 2026

Revised: 15 July 2026

Accepted: 16 July 2026

Published: 20 July 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

scope. Without that separation, package-level crashes, runtime defects, middleware failures, simulator failures, policy violations, and task outcomes may be treated as comparable evidence when they support different claims.

Figure 1 contrasts the present review with representative existing reviews through qualitative coverage coding. The rows denote review types, not individual papers, and the assessment serves as a coverage-positioning map. The rubric uses three dimensions, namely ROS target layers, fuzzing workflow stages, and cyber-physical evidence and diagnosis, which indicate whether a review separates ROS/ROS 2 interfaces, runtime and middleware mechanisms, simulator or physical-task evidence, fuzzing stages, task-level outcomes, reproducibility, and diagnosis. Coverage levels were assigned using fixed thresholds: “Strong” denotes a main organizing axis with most checks covered through dedicated sections, tables, or systematic comparison; “Partial” denotes background, transferable, or local coverage; and “Weak” denotes absent or brief coverage. The framework is used as a review-specific evidence-organization scheme, and co-authors with ROS, software-testing, and cybersecurity expertise cross-checked the coding to improve consistency and transparency.

Review type	ROS target layers	Workflow stages	Cyber-physical evidence & diagnosis
General fuzzing surveys			
Protocol/stateful fuzzing surveys			
Robotics and ROS 2 surveys			
This survey			

 Strong
  Partial
  Weak

Figure 1. Qualitative coverage positioning of this survey against representative existing reviews. Strong, Partial, and Weak indicate coverage levels under the rubric described in the text; they do not denote review quality.

Under these thresholds, general fuzzing surveys have Strong coverage of workflow stages because they organize input generation, mutation, feedback, coverage, oracle design, and triage; they have Weak coverage of ROS target layers and cyber-physical evidence because they rarely distinguish ROS-specific layers or robotic task outcomes [5,6]. Protocol and stateful-system surveys provide Strong coverage of workflow-related mechanisms by systematizing message formats, sequences, state models, executors, feedback, and oracles. Their coverage of ROS target layers is Partial: these mechanisms can transfer to ROS topics, services, actions, and DDS-based communication, but the surveys do not establish ROS-specific or robotic-task evidence [8,9]. Robotics and ROS 2 surveys have Strong coverage of ROS target layers because they cover architecture, middleware, QoS, security, simulators, and robotic testing. Their cyber-physical coverage is Partial, and their workflow-stage coverage remains Weak because they do not systematically organize fuzzing-specific input generation, mutation, feedback, oracle, and diagnosis stages [3,7]. The present review combines these dimensions by using ROS target layers and fuzzing workflow stages as

organizing axes while treating simulation or physical evidence, oracles, and diagnosis as part of the comparison framework.

The comparison motivates the two-axis organization used below. The target-layer axis asks where the evidence comes from and what claim it can support. The second axis asks which part of the fuzzing workflow a study develops. Interpreting the axes together prevents heterogeneous studies from being compared by tool name or bug count before their applicability and limitations are clear.

The review is guided by four analytical questions. AQ1. Which ROS or ROS-based target layers have been fuzzed or used as testing evidence, and what claims can each layer support? AQ2. How do existing studies instantiate input modeling, mutation or scheduling, execution and reset, feedback, oracle construction, and diagnosis? AQ3. What evidence supports claims about effectiveness and applicability, and where do heterogeneous targets, metrics, oracles, and execution settings prevent direct comparison? AQ4. What cross-stage bottlenecks remain for reproducible, safety-aware, and diagnosable ROS fuzzing?

This review is a structured, evidence-scoped qualitative survey that organizes heterogeneous ROS fuzzing studies by target layer and workflow stage to clarify which claims each can support. On this basis, and guided by the four analytical questions above, the review makes three contributions:

- (1) It develops an evidence-bound target-layer view of ROS fuzzing and relates each layer to the failure evidence that can support ROS-specific or transferable claims.
- (2) It synthesizes existing methods through a workflow-stage view, comparing their applicability, advantages, limitations, and reported empirical evidence.
- (3) It derives bottleneck-driven research routes for reusable models, actionable feedback, portable oracles, repeatable execution, and diagnosis.

2. Background: ROS-Based Robotic Systems

Figure 2 summarizes the ROS-based testing target considered in this survey. ROS 1 provides the computation-graph abstraction, where nodes communicate through typed interfaces and shared configuration contexts. ROS 2 extends this software layer with DDS-based middleware, quality-of-service (QoS) policies, executors, lifecycle nodes, and security mechanisms. Robotic applications then embed these software layers in control loops that involve sensors, actuators, simulators, and physical environments. These are connected layers of one testing target, which is why ROS fuzzing must consider interfaces, runtime execution, and cyber-physical behavior together.

The architecture is used here as an analytical basis, not as a standalone ROS tutorial. Each layer changes four fuzzing decisions: what counts as an input, how executions are driven and reset, what feedback can be observed, and which oracle can judge a failure. Those decisions connect the background to the target-layer and workflow-stage classification used below.

The background sources support different claim types. ROS interaction-bug and misconfiguration studies explain interface, configuration, and contract assumptions; ROS 2 runtime, tracing, and middleware studies identify callback, executor, lifecycle, and serialization observability; robotic and CPS execution studies justify simulator, replay, and task-level oracle requirements [10–15]. They identify failure sources and observation needs, so later fuzzer outcomes are compared by target layer, reported outcome, and cost boundary.

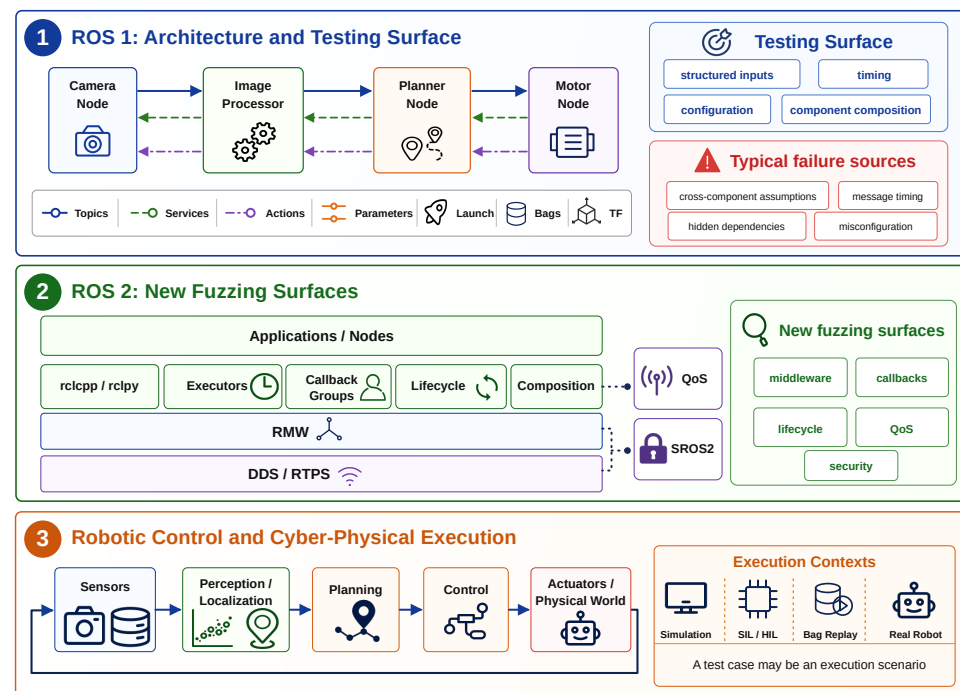


Figure 2. ROS-based robotic systems.

2.1. ROS 1 Architecture and Testing Surface

ROS 1 composes independently developed nodes into a computation graph. These nodes interact through typed communication, configuration, replay, and coordinate-frame mechanisms [1]. From a fuzzing perspective, inputs are not merely bytes or application programming interface (API) arguments. They are constrained by message schemas, runtime configuration, node launch contexts, spatial-frame relations, and timing assumptions.

The same modularity also creates characteristic failure modes. Empirical work on ROS interaction bugs shows that faults often arise from cross-component assumptions, message timing, hidden dependencies, and environmental context [10]. Misconfiguration studies similarly show that configuration contracts are often implicit, distributed across launch files and package documentation, or inconsistent with actual package behavior [11]. For fuzzing, these findings imply that test cases should vary parameters, remappings, message timing, component composition, and message payload fields.

ROS 1-related evidence is strongest for explaining testing surfaces and weaker for ranking fuzzers by effectiveness. Property-based testing and hypothesis-ros show feasible routes for type- or property-valid inputs, while interaction-bug and misconfiguration studies identify message, configuration, transform, context, and timing assumptions. These sources support claim types and failure mechanisms; large-scale effectiveness comparisons require direct fuzzing evidence from later ROS/ROS 2 tools [1,10,11,16,17].

2.2. ROS 2 Architecture and New Fuzzing Surfaces

ROS 2 preserves the computation-graph abstraction but changes the runtime substrate. It removes the centralized ROS master and uses DDS through the ROS Middleware abstraction for discovery and communication. DDS and the Real-Time Publish-Subscribe (RTPS) protocol define data-centric publish-subscribe communication, serialization, and interoperability mechanisms [12]. ROS 2 also exposes QoS policies such as reliability, durability, deadline, lifespan, liveliness, and history depth. These policies are necessary for robotics, but they also create behavioral combinations that affect latency, loss, buffering, deadlines,

and replay [13]. Performance studies further show that middleware choice, message size, QoS settings, and transport behavior affect ROS 2 throughput and latency [18].

From a fuzzing perspective, the ROS 1/ROS 2 difference is a change in target layer. ROS 1 mainly constrains fuzzing through typed topic and service messages, launch-time configuration, remapping, parameters, coordinate-frame assumptions, logging and replay, and cross-node timing [1,10,11,16]. ROS 2 retains these graph-level concerns but adds middleware and runtime surfaces: client-library and executor behavior, callback scheduling, lifecycle transitions, component composition, DDS/RTPS discovery and serialization, QoS combinations, security permissions, and runtime tracing [2,3,12–15,19]. These differences affect input modeling, execution control, feedback collection, and oracle design, which motivates treating ROS target layers as an explicit classification dimension.

For ROS 2, dominant solutions can only be identified within a target surface. Package and interface fuzzing is represented by `ros2_fuzz`, `RPFuzz`, `ROZZ`, and `ROFER`; runtime and callback-state fuzzing by `R2D2`; lifecycle-concurrency fuzzing by `ROCF`; middleware boundary testing by `Broken in Transit`; and tracing or monitoring support by `ros2_tracing` and `ROSMonitoring` [15,20–27]. Their outcomes differ across coverage, crash or memory bugs, lifecycle defects, middleware exceptions, and monitoring overhead, so the comparison is conditional on the target layer and oracle.

2.3. Robotic Control and Cyber-Physical Execution

A ROS application is usually embedded in a closed-loop robotic system: sensor feedback shapes perception and planning, control decisions affect the physical world, and the changed world is observed again by sensors. This makes robotics testing different from ordinary software testing, because correctness depends not only on program behavior but also on environmental assumptions, timing, hardware interaction, and the availability of suitable oracles [7]. ROS fuzzing often requires simulation, replay, controlled reset, and sometimes real-robot execution. A test case is no longer merely an input value; it may be an execution scenario.

The traditional black-box, gray-box, and white-box distinction still describes the level of internal information or instrumentation available to a fuzzer [5,6]. It is not sufficient as the main organizing scheme for ROS fuzzing, because it does not specify the tested target layer, ROS input granularity, runtime or protocol state, execution topology, simulator or physical context, feedback signal, or oracle [8,9]. We treat black/gray/white as a background dimension and organize ROS fuzzing primarily by target layers and workflow stages.

The gap between classic and ROS-specific fuzzing is an assumption mismatch. ROS fuzzers may need typed interfaces, parameters, launch configuration, and sensor or command streams beyond bytes. Their executions may involve distributed nodes, callbacks, lifecycle states, simulators, or hybrid simulator-physical setups. Their feedback may come from message features, callback traces, lifecycle-pair coverage, semantic state, or task-level signals. Their oracles may need to check interface properties, timing constraints, policy violations, physical consistency, or task outcomes in addition to crashes and sanitizers [16,22–25,28]. Classic fuzzing remains relevant, but its input, execution, feedback, and oracle assumptions must be specialized for ROS-based robotic systems.

The useful comparison is at the level of assumptions. Byte- and coverage-oriented fuzzers automate isolated software exploration well; interface-aware ROS fuzzers improve type and contract validity; runtime-state fuzzers expose callback, executor, lifecycle, or middleware behavior; and semantic or task-level fuzzers make correctness and safety failures observable [5,6,22–25,28]. Each step expands the observable failure space while adding engineering cost in schemas, instrumentation, reset, replay, or oracle construction.

The evidence for ROS-specific specialization is strongest within the reported target layer. ROZZ and ROFER support multi-dimensional and message-feature fuzzing on selected ROS programs; R2D2 and ROCF support runtime-state and lifecycle-concurrency feedback; RoboFuzz supports target-specific semantic feedback; and `ros2_fuzz` and RPFuzz mainly support tool or package-level feasibility [20–25,28]. The critical distinction is the failure class made observable, including coverage growth, memory or concurrency defects, middleware crashes, and semantic correctness.

Applicability boundaries follow from these comparisons. Architecture and reliability studies justify target and workflow dimensions, while tool-effectiveness claims require direct ROS/ROS-based fuzzing evidence or explicitly bounded near-domain analogs [1,2,7,10,11,22–25,28]. Benchmarks, bug definitions, execution platforms, and evidence types shape how each claim is compared. In the model used below, interface and package targets constrain schema-aware generation and contract checking; runtime, lifecycle, and middleware targets require tracing and feedback over callbacks, queues, serialization, or scheduling; and simulator or physical-task targets require scenario state, reset control, replay, and task-level or physical oracles.

3. Paper Collection and Classification

3.1. Scope

The scope covers fuzzing techniques for ROS-based robotic systems and closely related testing evidence. A work is included when it directly fuzzes ROS or ROS 2 programs, runtime components, middleware, serialization layers, simulators, or ROS-based robotic systems. We also include robotic, CPS, ADS, protocol, survey, and tool records when they clarify a workflow stage of ROS fuzzing, provide benchmark context, or support taxonomy construction. The corpus combines direct ROS/ROS-based evidence and near-domain evidence, but the two are not interpreted in the same way: direct ROS/ROS-based studies support ROS-specific claims, whereas non-ROS or mixed-domain studies are used as transferable workflow evidence unless their reported evaluation explicitly uses a ROS or ROS-based target. General fuzzing papers are used only as background for workflow concepts or taxonomy.

This broader scope reflects the limited and uneven body of direct ROS fuzzing evidence. Each category has a defined evidentiary role: direct ROS or ROS-based fuzzing studies ground ROS-specific conclusions; ROS testing, system, and security studies define target surfaces and failure mechanisms; near-domain robotic, CPS, and ADS studies provide transferable workflow evidence; and general fuzzing surveys provide terminology and taxonomy background.

3.2. Collection Strategy

Table 1 lists the inclusion and exclusion criteria used to screen the corpus. The criteria keep the review focused on the ROS fuzzing workflow while allowing adjacent evidence to be retained only when its role in the workflow analysis is explicit.

Following established fuzzing-survey practice, we collected records through keyword search, manual filtering, and snowballing across academic databases, publisher pages, major security/software-engineering venues, preprint servers, and open-source repositories. The search window covered January 1990 to June 2026, which includes the earliest foundational fuzzing reference cited in this review while keeping the screening focused on ROS, robotic, CPS, and ADS fuzzing/testing evidence. The 1990 lower bound covers foundational fuzzing and software-testing work used for terminology and workflow concepts, while ROS-specific claims are drawn from direct ROS or ROS-based studies. The June 2026 upper bound marks the final corpus update for this revision. As summarized

in Figure 3, the keyword stage used 12 representative combinations formed from fuzzing-related terms (fuzz, fuzzing, and fuzzer) and domain terms (ROS, ROS 2, robotic, and CPS). During title and abstract screening, near-equivalent expressions such as ROS2, Robot Operating System, robotic systems, cyber-physical systems, autonomous driving systems, and fuzz testing were treated as terminological variants within the same query families. Where Boolean syntax was supported, a query paired a fuzzing-related term with a domain term. Repository, project-page, and tool searches used the same terms as keyword probes because these sources do not expose uniform Boolean interfaces. We report query families and source roles because raw hit counts are not comparable across these sources. Table 2 reports the source groups, representative query forms, and source roles used to apply these query families.

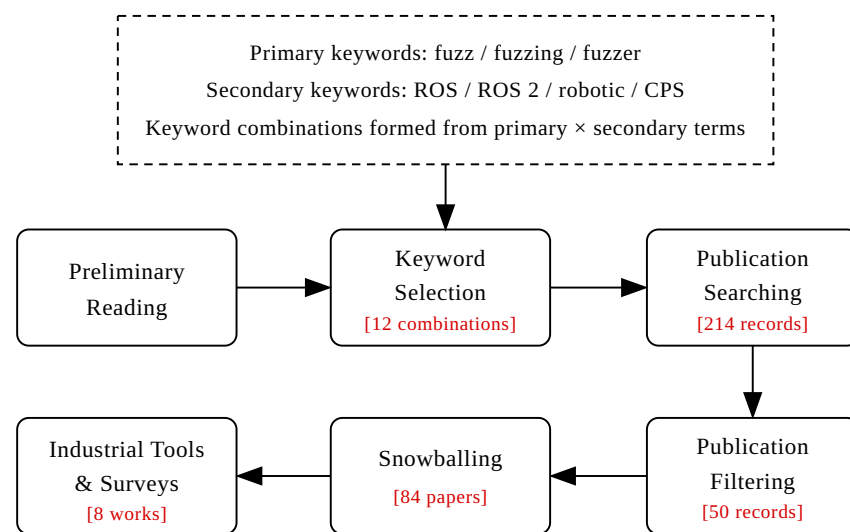


Figure 3. Literature selection process for the 92-work corpus.

Table 1. Inclusion and exclusion criteria.

Criterion	Included	Excluded or Secondary
Target relevance	ROS/ROS 2 programs, runtime components, middleware, serialization layers, simulators, or robotic systems with workflow-level fuzzing relevance.	Generic fuzzing papers without ROS or robotic-system relevance.
Method relevance	Modeling, input generation, mutation, scheduling, execution, monitoring, feedback, oracle, or diagnosis techniques.	Robotics or security papers without testing or fuzzing mechanisms.
Foundation relevance	ROS testing, tracing, runtime verification, specification, and behavior-modeling studies that support fuzzing.	General ROS tutorials or application papers.
Evidence type	Peer-reviewed papers, preprints, technical papers, open-source tools, surveys, and reproducible artifacts.	Items with unclear metadata, insufficient technical details, or no clear connection to the ROS fuzzing workflow.

Table 2. Search sources and representative query forms used for corpus construction.

Source Group	Representative Query Form	Role and Caveat
Google Scholar	("ROS" OR "ROS 2" OR "ROS2" OR "Robot Operating System") ("fuzz" OR "fuzzing" OR "fuzzer" OR "fuzz testing"); ("robotic" OR "robotics" OR "robotic systems" OR "cyber-physical systems" OR CPS OR "autonomous driving systems") ("fuzz" OR "fuzzing" OR "fuzzer" OR "fuzz testing").	Broad discovery and citation links; ranking and raw-hit counts are not stable.
IEEE Xplore	("ROS" OR "ROS 2" OR "ROS2" OR "Robot Operating System") AND (fuzz OR fuzzing OR fuzzer OR "fuzz testing"); ("robotic systems" OR robotics OR "cyber-physical systems" OR CPS OR "autonomous driving") AND (fuzz OR fuzzing OR fuzzer).	Engineering, robotics, and security metadata; field syntax differs from general search engines.
ACM Digital Library	("Robot Operating System" OR ROS OR "ROS 2" OR ROS2) AND (fuzzing OR fuzzer OR "fuzz testing").	Software-engineering and systems venues; publisher metadata was used for verification.
SpringerLink, ScienceDirect, Wiley, and MDPI	("Robot Operating System" OR ROS OR ROS2 OR "ROS 2") AND (fuzz OR fuzzing OR fuzzer); (robotic OR robotics OR "cyber-physical" OR autonomous) AND (fuzz OR fuzzing OR fuzzer).	Publisher and venue checks; Boolean and wildcard behavior differs by platform.
arXiv, TechRxiv, and SSRN	ROS fuzzing; ROS2 fuzzing; "Robot Operating System" fuzzing; robotic fuzzing; cyber-physical fuzzing; autonomous driving fuzzing.	Preprint tracking and early technical records; metadata can change over time.
GitHub, GitLab, and project pages	ROS fuzzer; ROS2 fuzzer; ROS fuzzing; robot fuzzing; tool names, artifact links, and repository names followed from included papers.	Tool and artifact discovery; repository search is not a stable bibliographic interface.

Because these interfaces differ in Boolean syntax, field restrictions, ranking, and repository stability, raw hit counts were not used as comparable evidence. The selection flow uses the merged candidate record as the counting unit; the 214 records are the deduplicated set retained after applying the query families and excluding clearly irrelevant pages.

After merging the keyword-combination results and removing exact duplicate records and clearly non-paper pages, the keyword search returned 214 candidate records. During result merging, duplicates were identified from titles, authors, venues, publication years, DOI or preprint identifiers, publisher metadata, and source or artifact links. We then manually screened titles, abstracts, metadata, and, when necessary, introductions and conclusions. Records were retained when they directly addressed ROS/ROS 2 fuzzing, ROS testing foundations, ROS-based robotic systems, or workflow-relevant robotic, CPS, and ADS fuzzing. Generic fuzzing papers, ROS application papers without testing relevance, incomplete records, and papers outside the workflow scope were excluded at this stage, leaving 50 directly relevant publications. Backward snowballing then inspected the references of these 50 publications, while forward snowballing used citation links, publisher pages, related venue proceedings, tool names, artifact pages, and repository links to identify follow-up or closely related work. Newly found records were screened using the same criteria, expanding the research-paper set to 84 papers. Finally, eight industrial tools, open-source tools, surveys, or technical records were added when they provided necessary workflow evidence, tool background, benchmark context, or taxonomy background. The final corpus contains 92 selected works.

Each work is annotated along seven dimensions: target layer, workflow stage, input or seed model, feedback signal, oracle type, execution setting, and reported empirical evidence. Figure 4 summarizes each work once by corpus category and primary workflow contribution. The corpus contains 21 direct ROS or ROS-based fuzzing works (22.8%), 12 ROS testing and analysis works (13.0%), 9 ROS system or security studies (9.8%), 40 near-domain robotic/CPS/ADS works (43.5%), and 10 survey or methodological records (10.9%). The stacked segments indicate where the reviewed literature contributes most directly to the fuzzing workflow; they are not intended as fuzzer-performance comparisons.

The corpus may over-represent successful and visible work. Confirmed bugs, published tools, English-language indexed papers, and stable repositories are easier to find than null results, failed deployments, closed industrial testing, local venues, or short-lived artifacts. Combining databases, publisher pages, preprints, repositories, project pages, and snowballing reduced but did not remove this bias.

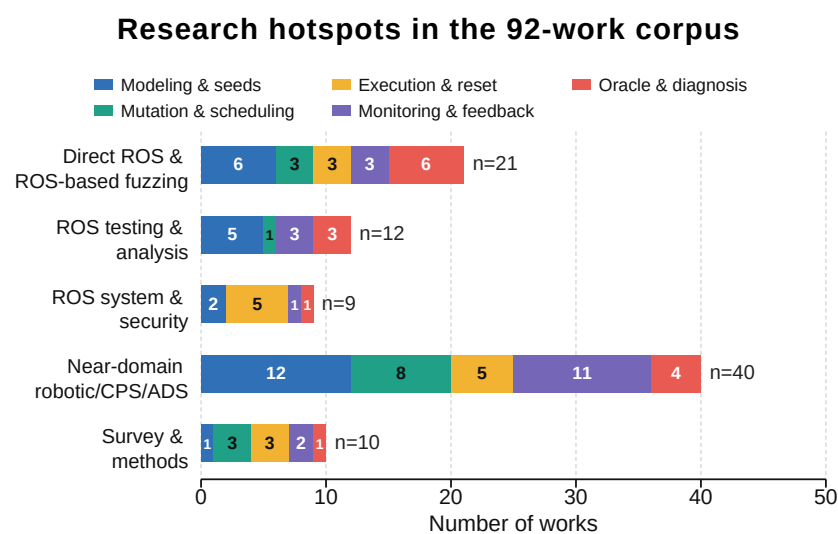


Figure 4. Corpus distribution and research hotspots by workflow contribution.

3.3. Corpus Classification

The corpus classification follows the comparison logic introduced in Figure 1. Target-layer coding records where a study obtains evidence, while workflow-stage coding records which part of the fuzzing process the study substantively advances. The two codes are interpreted jointly. A work can be ROS-specific in its target layer but contribute mainly to one workflow stage, as in runtime-state or lifecycle fuzzing; conversely, a near-domain robotic or simulator study can clarify a transferable workflow mechanism without serving as direct ROS-specific evidence. This joint interpretation is used to decide whether a citation supports a ROS-specific conclusion, a transferable design pattern, a comparability warning, or taxonomy background [22–25,28,29].

We classify the corpus into five evidence categories while organizing the main analysis by ROS fuzzing targets and workflow-stage innovations. Direct ROS and ROS-based fuzzing studies test ROS/ROS 2 programs, message interfaces, runtime mechanisms, lifecycle behavior, middleware, serialization boundaries, simulators, or ROS-based robotic systems, including ROS/Gazebo settings, MoveIt!/RViz packages, RoboFuzz/ROS 2 deployments, and Autoware-based ADS stacks [30–36]. ROS testing and analysis foundations support workflow components such as input generation, feedback collection, oracle design, and diagnosis. ROS system and security studies provide architectural, runtime, middleware, misconfiguration, attack, and fault-injection context. Near-domain or mixed robotic, CPS, and ADS fuzzing studies contribute execution settings, feedback signals, oracles,

scenario-generation methods, and diagnosis ideas, but they are treated according to the target actually evaluated in each cited work. Survey and methodological literature provide taxonomy background and transferable comparison dimensions [37–45].

Evidence-scope coding is used throughout Sections 5–7. Findings from direct ROS or ROS-based studies support ROS-specific conclusions. Findings from near-domain or mixed robotic/CPS/ADS studies serve as transferable workflow analogs, design patterns, or comparability warnings; they are not treated as ROS-specific empirical results unless the cited work’s main or explicitly reported evaluation uses a ROS or ROS-based target.

Structured evidence appraisal complements this coding. For each record, the appraisal identifies source type, evidence scope, evaluation design, reported outcome, artifact support, limitation reporting, and transferability condition (Table 3). These fields determine claim strength: empirical ROS studies support ROS-specific conclusions, tools and artifacts inform reproducibility, surveys support terminology and taxonomy, and near-domain studies supply transferable workflow observations.

Table 3. Evidence-appraisal dimensions used to interpret heterogeneous corpus records.

Dimension	Appraisal Focus	Use in Synthesis
Source type and evidence scope	Whether the record is a direct ROS study, ROS foundation, near-domain study, survey, tool, artifact, or specification.	Separates ROS-specific conclusions from transferable or background evidence.
Evaluation design	Whether the work reports a controlled comparison, ablation, benchmark-style evaluation, case study, demonstration, mapping, or taxonomy.	Controls the strength of effectiveness and comparison statements.
Reported outcome	Bugs, crashes, coverage, overhead, CVEs or issues, false positives, qualitative findings, or unavailable outcomes.	Prevents incompatible outcomes from being mixed as the same evidence.
Artifact support	Availability of source code, container, benchmark, dataset, logs, repository, demo, or artifact link.	Supports reproducibility and artifact-availability statements.
Limitations and transferability	Whether threats, assumptions, target dependence, or ROS adaptation requirements are reported.	Determines when caution language or evidence-limited wording is needed.

For the classification procedure, the coding was treated as consensus-based qualitative coding. After the initial assignment, domain experts who did not perform the initial coding pass independently assessed the target-layer, workflow-stage, and evidence-scope categories and flagged ambiguous cases for discussion. This assessment helps improve the consistency of classification boundaries and the treatment of ambiguous evidence cases.

Technique-level effectiveness is compared through shared dimensions: target or evidence layer, workflow role, reported outcome, comparison support, cost driver, scalability boundary, and applicability boundary. Reported bugs, coverage, CVEs, policy violations, simulator crashes, and safety scenarios are treated as different outcome types. This keeps package-level fuzzers, runtime-state fuzzers, simulator-as-target fuzzers, and task-level robotic fuzzers separated when their evidence is incompatible.

4. Challenges and Evolution of ROS Fuzzing

4.1. Domain-Specific Challenges

Three domain-specific constraints recur across the reviewed work. The first is input dimensionality. A ROS fuzzing campaign may vary messages, service calls, action goals, parameters, remappings, sensor streams, callback-triggering events, lifecycle transitions, maps, obstacles, weather, or bridge messages. ROZZ and ROFER provide direct ROS

evidence that dimension-level modeling and mutation can expose assumptions missed by single-message mutation [22,23]. Related work on configuration constraints, schema-aware protocol messages, static input dependencies, operator-interaction variables, and human-in-the-loop workload extends the candidate input space, but these sources remain transferable evidence until validated on ROS or ROS-based targets [38,40,46,47].

The second constraint is distributed execution. ROS behavior emerges from nodes, callbacks, queues, timers, and middleware interactions, so single-process coverage alone cannot describe graph-level progress; R2D2 and ROCF address this issue through callback tracing and lifecycle-pair coverage [24,25]. The third constraint is failure semantics. A fuzzing outcome may appear as a stale transform, inconsistent control output, task failure, collision, policy violation, discontinuity, or cyber-physical discrepancy even when no process crashes [28,30,48,49]. These constraints are interdependent: richer input abstractions require execution settings that can drive the corresponding state, feedback is actionable only when reset and observation are stable, and higher-level failures require oracle and diagnostic evidence that match the layer being exercised.

The comparative impact of these constraints depends on the outcome they change. Input dimensionality affects interface, configuration, and multi-dimensional mutation because valid but incomplete inputs can miss cross-condition bugs, and selected ROS studies report coverage or bug-discovery gains. Distributed execution affects runtime, callback, lifecycle, and middleware fuzzing because progress depends on cross-node traces or instrumentation and their overhead. Failure semantics affects semantic, policy, and physical-task fuzzing because correctness or safety violations require oracle, simulator-control, and replay support [22–25,28,30,31,48].

4.2. Evolution of ROS Fuzzing Techniques

Figure 5 organizes representative ROS and ROS-adjacent fuzzing techniques by the evidence they make observable. The three layers are interface and configuration validity, runtime and semantic behavior, and scenario and physical interaction. Rather than serving as a tool chronology, the figure maps abstraction levels: interface-level work tests whether valid ROS inputs and contracts can be exercised, runtime-level work tests whether distributed ROS behavior can be observed and guided, and scenario-level work tests whether robot-task failures can be exposed through semantic or physical evidence. Background, survey, benchmark, and attack-only records are synthesized in the text and tables where they support this analysis.

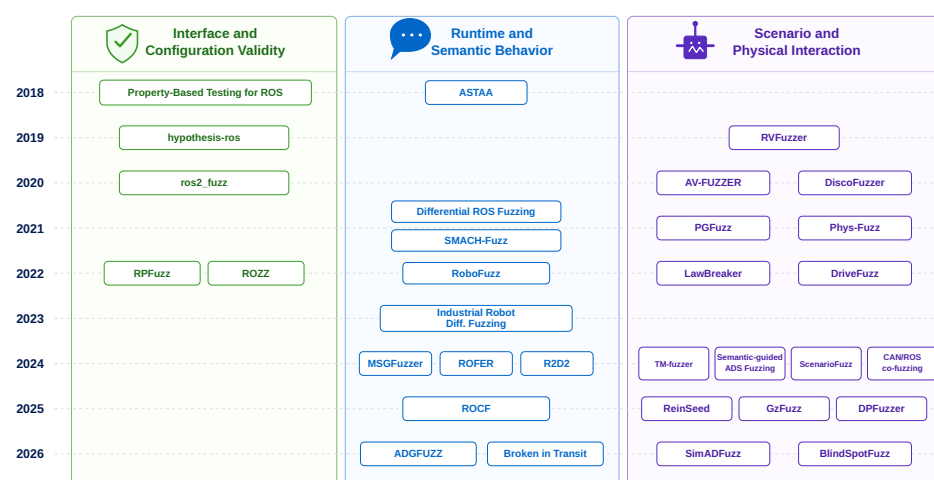


Figure 5. Representative evolution of fuzzing for ROS-based robotic systems from interface validity to runtime state and cyber-physical task behavior.

Two bodies of background evidence support this interpretation. General fuzzing surveys separate input generation, scheduling, feedback, oracle design, and triage as distinct workflow decisions, while protocol/stateful and robotics-testing surveys show that interactive or cyber-physical targets also require message-sequence modeling, reset/replay control, non-crash oracles, and reproducible evaluation [5–9]. The layers in Figure 5 indicate what becomes observable and what modeling, execution, and oracle assumptions are introduced, without inferring performance or publication-rate trends.

The first layer concerns interface and configuration validity. Property-Based Testing for ROS, hypothesis-ros, ros2_fuzz, ROZZ, and RPFuzz show how configuration models, message schemas, property-based generators, American Fuzzy Lop (AFL)-style harnessing, multidimensional ROS inputs, and package-level harnesses support valid and contract-aware ROS test generation [16,17,20–22]. RPFuzz is treated as direct but preprint-level ROS package-fuzzing evidence, so it serves to illustrate package-specific harnessing but should not be overgeneralized as a benchmark result. At this layer, generated inputs become more meaningful for ROS software, but distributed runtime states and task-level semantics remain largely outside the evidence.

The second layer concerns runtime and semantic behavior. ASTAA, Differential ROS Fuzzing, SMACH-Fuzz, RoboFuzz, ROFER, R2D2, ROCF, and Broken in Transit move ROS fuzzing from valid inputs toward live message interception, runtime monitors, differential behavior, behavior-state execution, semantic correctness, message-feature novelty, callback tracing, lifecycle concurrency, and middleware type contracts [23–26,28,50–52]. Near-domain robotic work in the same lane, including differential industrial-robot fuzzing, assignment-dependency-guided robotic-vehicle fuzzing, and message-sequence-guided industrial-protocol fuzzing, supports the view that semantic or stateful robotic failures often require more than valid message mutation [34,42,46]. Runtime guidance improves under this evidence, but physical interaction, simulator fidelity, and scenario-level task failures remain only partly covered.

The third layer expands fuzzing to scenario and physical interaction, where failures may appear as collisions, unsafe trajectories, policy violations, physical discontinuities, or poor task performance. Representative works use control guidance, safety-potential search, discontinuity, physical risk, temporal policies, rule specifications, driving-quality and traffic-management feedback, semantic scenario features, bridge execution, seed optimization, simulator-command generation, drone path risk, simulation-feedback prediction, and blind-spot cues to expose cyber-physical failures [29–33,48,49,53–61]. Other perception and scenario studies provide additional transferable evidence and are synthesized in the text to keep Figure 5 readable [43,62–66]. The price of this broader evidence is higher dependence on simulator reset, replay stability, oracle portability, and cross-node diagnosis.

The three layers show a consistent trade-off. Interface-level methods are portable and schema-aware but shallow with respect to runtime or task state. Runtime-state methods improve callback, lifecycle, and middleware observability while adding tracing, instrumentation, and replay costs. Scenario-level methods expose task and safety evidence at higher simulator, reset, and oracle-modeling cost. Because costs are reported in different units, the comparison focuses on cost drivers and scalability boundaries. This layered view also explains why black-box, gray-box, and white-box access levels are insufficient as the only classification axis: the main shifts across the literature concern input abstraction, runtime-state feedback, and oracle scope beyond instrumentation access. Read together with Figure 4 and Table 4, the evolution is best read as a descriptive evidence trend, not as a claim of monotonic performance improvement. Direct ROS/ROS-based studies anchor ROS-specific evidence, while near-domain studies supply transferable workflow analogs under the evidence-scope rules in Section 3.

4.3. Workflow Organization

Figure 6 presents the workflow used to organize the rest of this survey. Following the component-oriented view commonly used in fuzzing surveys [8,9], we describe ROS fuzzing through three components: the input generator, the executor, and the bug collector. The input generator links target-system knowledge to target modeling, input generation, scheduling, and mutation. The executor combines execution and reset with runtime observation, feedback, and monitoring, because ROS fuzzing depends on repeated launches, simulator or system resets, and observations from traces, monitors, middleware events, or physical signals. The bug collector determines whether an observed behavior indicates a failure and whether the failure can be reproduced, reduced, and diagnosed.

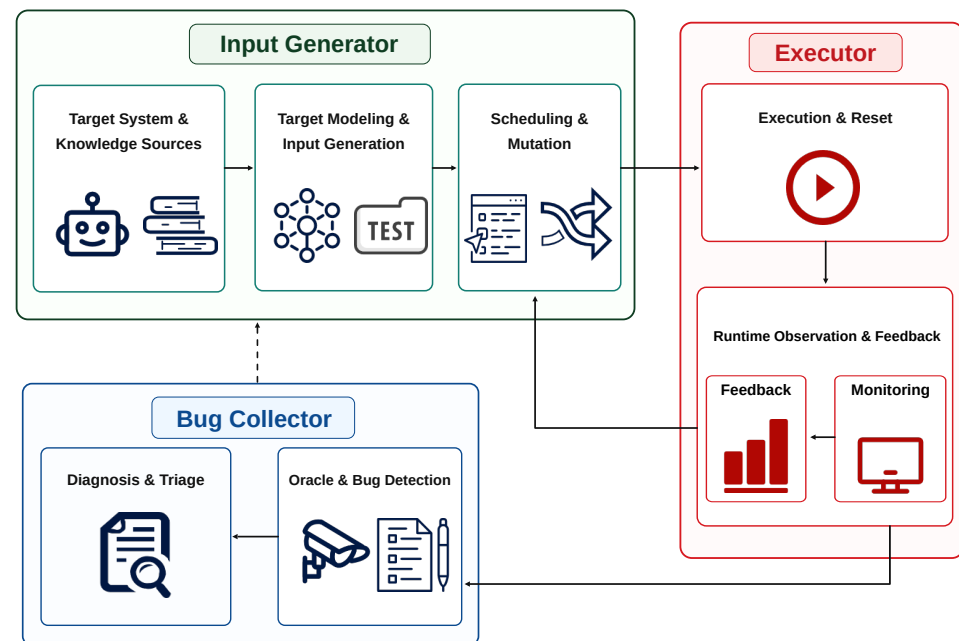


Figure 6. Workflow of fuzzing for ROS-based robotic systems.

The workflow is an organizational scaffold adapted from existing survey traditions. General fuzzing, protocol-fuzzing, stateful-fuzzing, and robotics-testing surveys provide the component concepts for input generation, execution, feedback, oracle design, state modeling, simulation, and reproducibility [5–9]. Direct ROS and ROS-based studies anchor these concepts in target layers such as typed interfaces, runtime and lifecycle state, middleware deserialization, simulator behavior, semantic feedback, and task-level oracles [21–26,28,29]. This scaffold supports cross-study comparison by linking each method to its ROS target layer, workflow role, and evidence scope.

The workflow provides a comparison frame for technique-level contributions across studies with different ROS target layers and evidence types. Table 4 summarizes this difference for the representative mature fuzzing works shown in the evolution map. Survey, benchmark, background, and non-fuzz infrastructure records remain part of the corpus-level synthesis; the matrix focuses on technique-level workflow contributions. The primary and secondary labels indicate only the strength of each work’s contribution to a workflow stage.

Table 4. Workflow-stage contribution matrix of representative mature fuzzing works.

No.	Work	Model /Input	Mutation /Scheduling	Execution /Reset	Feedback	Oracle	Diagnosis /Triage
01	Property-based [16]	•	▲	▲	○	•	○
02	ASTAA [50]	•	•	•	▲	•	•
03	Differential ROS [51]	•	•	▲	▲	•	▲
04	SMACH-Fuzz [52]	•	▲	▲	○	▲	▲
05	DiscoFuzzer [49]	•	•	▲	•	•	▲
06	RoboFuzz [28]	•	▲	▲	▲	•	▲
07	ROZZ [22]	•	▲	▲	•	▲	○
08	ROFER [23]	•	•	▲	•	▲	○
09	R2D2 [24]	▲	▲	•	•	▲	▲
10	ROCF [25]	▲	•	▲	•	•	•
11	Broken in Transit [26]	•	▲	•	○	•	▲
12	PGFuzz [48]	•	▲	▲	•	•	▲
13	Phys-Fuzz [30]	•	•	▲	•	•	▲
14	DriveFuzz [31]	•	▲	▲	•	•	○
15	DPFuzzer [59]	•	▲	▲	•	•	▲
16	hypothesis-ros [17]	•	•	○	○	▲	▲
17	RVFuzzer [53]	▲	•	▲	•	•	▲
18	ros2_fuzz [20]	•	▲	•	▲	▲	○
19	AV-FUZZER [54]	•	•	▲	•	•	▲
20	RPFuzz [21]	•	•	•	▲	▲	○
21	LawBreaker [55]	•	▲	▲	▲	•	▲
22	Ind. Robot Diff. Fuzzing [34]	▲	▲	▲	▲	•	▲
23	MSGFuzzer [42]	•	•	▲	•	▲	▲
24	TM-fuzzer [33]	▲	▲	▲	•	▲	○
25	Semantic ADS Fuzzing [56]	•	•	▲	•	▲	○
26	ScenarioFuzz [32]	•	•	▲	•	▲	▲
27	CAN/ROS co-fuzzing [57]	▲	▲	•	▲	•	▲
28	ReinSeed [58]	▲	•	▲	•	▲	○
29	GzFuzz [29]	•	•	•	•	▲	▲
30	ADGFUZZ [46]	•	•	▲	▲	▲	•
31	SimADFuzz [60]	▲	•	▲	•	▲	○
32	BlindSpotFuzz [61]	•	•	▲	•	•	▲

•: primary contribution; ▲: secondary contribution; ○: absent or limited contribution. Primary and secondary indicate only the strength of each work's contribution to a workflow stage.

The coding protocol has four steps. First, each work was checked for whether it directly tests ROS/ROS 2, a ROS-based robotic system, or a related robotic, CPS, or ADS target. Second, the target layer was recorded, such as message, service, action, node, package, runtime, middleware, lifecycle, simulator, scenario, or physical-task layer. Third, the input granularity was recorded, such as topic/message-level, component-level, runtime-event-level, or scenario/task-level inputs. Fourth, each work was mapped to workflow stages according to the stage addressed by its main technical contribution, implementation, or evaluation. A stage was coded as primary when it was a central design goal, method module, or evaluation focus; secondary when it was required by the method but was not a main claimed contribution or separately evaluated component; and absent when it was not substantively addressed. After the initial coding, the first author rechecked every Table 4 label against this protocol. Two co-authors with doctoral degrees and relevant research backgrounds independently reviewed the full table and flagged ambiguous or potentially inconsistent entries. For entries with disagreement or unclear boundaries, the author team compared the coding with the target scope, target layer, input granularity, and reported evaluation evidence. The final coding was confirmed through discussion under the guidance of two senior domain experts, including the corresponding authors. This author-team-expert cross-check was used to improve classification-boundary consistency.

The protocol structures a heterogeneous survey corpus through consensus-based qualitative coding and workflow-stage contribution coding. The primary, secondary, and absent labels describe contribution focus and are used for synthesis across workflow stages. Table 4

shows uneven primary-contribution coverage among the 32 representative mature fuzzing works: model/input is coded as primary in 24 works, feedback in 19, oracle construction in 17, and mutation/scheduling in 17, whereas execution/reset is primary in 7 works and diagnosis/triage in 3. These counts indicate which stages are most often treated as central technical contributions and which remain more often supporting or underdeveloped. They also guide the remaining sections: Section 5 discusses modeling, seeds, scheduling, and mutation; Section 6 examines execution, reset, monitoring, and feedback; Section 7 analyzes oracle design, bug detection, diagnosis, and triage; and Section 8 summarizes bottleneck-driven research routes. The following sections use the same evidence-scope coding and shared comparison dimensions: target or evidence layer, observable outcome, cost or dependency, applicability boundary, and comparability boundary.

5. Input Generator

5.1. Target Modeling and Test Case Generation

Test-case generation is organized by input abstraction level. Interface and contract seeds describe validity constraints needed to reach ROS logic [11,16,20,67,68]; runtime-state and behavior seeds describe the context that makes a message meaningful [24,25,52,69,70]; scenario and operational-context seeds describe the physical or task conditions under which robot-level failures can be observed [30–32,48,59,71]. These levels are compositional: scenarios may rely on valid message seeds, and behavior-state models may guide parameters, event sequences, or lifecycle transitions [22,23,25,29,47].

Table 5 compares the main modeling levels and seed sources by four criteria: generated-input abstraction, ROS or robotic context, plausible failure classes, and the main cost or evidence limitation reported in the literature. Because studies report outcomes through incompatible proxies such as coverage, bug reports, and scenario-level safety outcomes, the table uses evidence-oriented criteria to indicate where each seed source applies and what kind of claim it can support.

Table 5. Comparison criteria and evidence boundaries for target modeling and seed generation strategies.

Seed Model	Abstraction and Context	Failure Focus	Cost/Evidence Boundary
Interface/contract	ROS interfaces, parameters, launch files, YAML parameter files, and remapping constraints.	Syntactic and configuration validity; hidden interface contracts.	Limited for deep task states without runtime or behavior context; property-based testing, <code>ros2_fuzz</code> , <code>RPFuzz</code> , and configuration fuzzing provide direct or transferable evidence with different levels of evidentiary strength [16,17,20,21,38].
Multi-dimensional ROS input	Commands, configuration, sensor streams, message order, and operator interaction.	Cross-input assumptions in commands, parameters, sensor messages, and ordering.	ROZZ and ROFER report coverage and bug-finding gains in their benchmarks; human-interaction fuzzing is near-domain evidence for a wider command space [22,23,47].
Runtime-state seed	Callbacks, executors or timers, lifecycle transitions, and dependency-relevant inputs.	Asynchronous state, callback behavior, lifecycle behavior, and concurrency defects.	R2D2 and ROCF provide ROS-specific evidence, while dependency-guided robotic fuzzing indicates a static-to-runtime selection route that still needs ROS-specific validation [24,25,46].

Table 5. Cont.

Seed Model	Abstraction and Context	Failure Focus	Cost/Evidence Boundary
Behavior logic	State machines, behavior trees, logs, inferred models, message sequences.	Task phases, state-dependent transitions, and sequence-sensitive behavior.	Model quality, log completeness, and domain knowledge affect portability; sequence-guided protocol fuzzing is transferable evidence, not ROS-specific proof [42,52,69,70].
Physical/scenario	Maps, obstacles, weather, policies, simulator commands, perception perturbations.	Task-level failures, policy violations, unsafe motion, and perception-triggered failures.	Evidence depends on simulator fidelity, perception realism, reset capability, and domain-specific oracle design; many studies are ROS-adjacent, not ROS-specific [30–32,43,48,54,56,59,61,65,66].

The comparison suggests a layered relationship among seed sources. Interface and contract seeds are inexpensive and portable, but they rarely drive a robot into deep task states by themselves [11,16,17,20,21,38,68]. Runtime-state and behavior seeds add execution context, but depend on traces, inferred models, or lifecycle information that may be incomplete or target-specific [24,25,52,69,70]. Scenario seeds can expose cyber-physical failures, but their validity depends on simulator reset, environmental realism, and task oracles [30–32,48,59,71]. In practice, these sources are often complementary: low-level message validity keeps a scenario executable, behavior-state information helps choose meaningful event sequences, and feedback or scheduling can decide which input family should receive the next testing budget [22,23,25,28,29].

The evidence for target modeling is interpreted by level of support: controlled ROS/ROS-based comparisons and ablations support effectiveness claims, direct tool and specification records support feasibility or input-validity claims, and near-domain scenario studies support transferability claims [20–25,28,30,31,68,70]. This evidence ladder keeps modeling conclusions tied to the kind of result each study actually reports.

5.1.1. Interface-Aware and Contract-Aware Seeds

Interface-aware generation starts from typed ROS interfaces, parameters, launch contexts, and remapping rules. Property-based ROS testing, `ros2_fuzz`, and ROS data-modification domain-specific languages (DSLs) show that configuration models, discovered interfaces, harnesses, and topic events can seed tests [16,20,67]. RPFuzz, although available as a preprint record, reinforces the value of package-level harnesses for ROS software packages, while configuration-defect fuzzing in ADS suggests a related route for launch files, YAML parameters, and remapping constraints [21,38]. The design principle is to preserve schema and runtime validity while stressing implicit value, timing, configuration, and dependency assumptions. ROSpec and misconfiguration studies indicate that many such contracts are hidden or incomplete, so seed generators should combine valid interface instances with controlled boundary, ordering, timing, and configuration violations [11,68].

Interface and configuration sources are best interpreted as validity and reachability constraints. They reduce invalid tests and expose implicit ROS contracts; deeper behavior usually requires additional runtime-state, temporal, or scenario context [11,16,20,38,68]. The field-level contribution of this line of work is to make ROS fuzzing executable and contract-aware. Its main boundary is that validity constraints are only a prerequisite; stateful or scenario-level exploration still requires richer execution context.

Support for interface-aware seeds is uneven. The hypothesis-ros and `ros2_fuzz` tools mainly illustrate interface-level generation and harness routes; property-based ROS testing adds property and configuration feasibility; RPFuzz provides preprint-level pack-

age evidence on OMPL; ROZZ and ROFER report stronger bug and coverage evidence in selected ROS/ROS 2 benchmarks; and ROSpec supplies specification and configuration constraints [16,17,20–23,68]. Together, these works show that interface validity improves executability and exposes contracts, but deep state reachability still depends on runtime feedback, reset control, oracle support, and scalable harness or property construction [11,16,17,20,21,68].

5.1.2. Behavior-State Test Case Construction

A ROS test case often needs a setup sequence before the target mutation. SMACH-Fuzz, BTreeFuzz, and ROSInfer use execution records, behavior-tree logs, or static ROS API analysis to derive behavior-state or component models, pointing toward inferred behavior as seed material [52,69,70]. ROS 2 lifecycle nodes make this concrete: managed-state transitions can be combined with delays and callback-triggering events. ROCF shows that lifecycle-pair sequences and order mutation expose lifecycle-related concurrency bugs, so lifecycle traces should be treated as seed material for stateful ROS 2 fuzzing [25].

The cost of this class of seed generation is model availability and maintenance. Execution logs may miss rare states, static models may approximate dynamic ROS behavior, behavior trees or state machines require domain knowledge, and lifecycle traces require instrumentation and repeatable reset [25,52,69,70]. These costs make behavior-state seeds more precise than isolated message seeds, but also less portable across packages, robot platforms, and simulator configurations.

Behavior-state construction scales according to how the state source is obtained. Static recovery methods such as ROSInfer reduce manual model writing for analyzable C++ components while abstracting away message contents; runtime callback tracing in R2D2 observes actual execution but adds tracing and threshold choices; lifecycle-pair tracing in ROCF targets managed-state concurrency with C/C++ instrumentation; and physical or scenario state improves task relevance while depending on simulator state, sensor logs, and mission-specific risk models [24,25,30,59,70].

5.1.3. Scenario and Operational-Context Seeds

Physical and scenario-aware seeds expand the input space beyond ROS messages. We group them by role. Environment seeds vary start poses, maps, obstacles, weather, or simulator state; robotic-vehicle, ADS, and drone studies use them to reach failures that depend on external conditions [30–32,59]. Policy/risk seeds encode temporal policies, hazardousness, driving quality, or perception cues, and guide search toward safety or task-level outcomes [48,54,56,61,64–66]. Execution-substrate seeds target the simulator or supervised execution through Gazebo commands, composable scenarios, or operator interactions [29,47,71]. These scenario studies clarify input dimensions and search constraints, but they provide direct ROS-specific evidence only when the generated environment is connected to ROS interfaces, simulator bridges, lifecycle control, or robot-task oracles.

This grouping supports a hierarchical seed view: interface seeds keep executions valid, behavior seeds drive stateful interactions, scenario seeds expose task context, and execution-substrate or operator-interaction seeds broaden the campaign setting. The seed categories form a literature-derived analytical framework, and their evidence boundaries were checked within the author team against the scope rules introduced in Section 3. The difficult step is integration across seed types. A reusable ROS fuzzing campaign has to compose interface/configuration constraints, lifecycle or callback events, behavior states, and scenario variables while preserving reset capability and oracle interpretability. The main difficulty is controlling modeling cost and keeping failures attributable to specific ROS or robotic-system assumptions [22,23,25,28,29,47,71].

The transfer boundary is evidence-based. DriveFuzz provides ROS-based ADS scenario evidence through Autoware and a ROS bridge, while Phys-Fuzz, DPFuzzer, PGFuzz, and related ADS/UAV studies mainly support task-level feedback and oracle patterns that still need ROS-specific validation [30,31,48,59]. Empirical ADS evaluation further shows that initial scenarios and violation detectors can change apparent fuzzer performance, so scenario-seed claims should report the seed set, detector definition, and execution setting [45].

5.2. Scheduling and Mutation

Where Table 5 compares seed and modeling sources, Table 6 compares mutation and scheduling mechanisms by perturbation unit, required execution context, failure focus, and evidence or cost limitation. Because the studies differ in benchmarks, metrics, failure definitions, and target types, including ROS nodes, ROS 2 lifecycle behavior, simulators, industrial protocols, and autonomous-system scenarios, the table reports context-specific evidence instead of a unified effectiveness score.

Within this evidence scope, the strongest conclusion is conditional effectiveness: message-feature and dimension scheduling are supported for selected ROS 2 programs, callback traces for runtime-state fuzzing, lifecycle-pair feedback for lifecycle-related concurrency bugs, and command-family scheduling for Gazebo crash discovery [23–25,29]. Each mechanism should therefore be reported with its target layer, outcome type, and cost.

Table 6. Comparison criteria and evidence boundaries for mutation and scheduling mechanisms.

Mechanism	Perturbation Unit/Context	Failure Focus	Cost/Evidence Boundary
Type-aware mutation	Typed message fields, ROS data types, structured protocol fields.	Schema and value-boundary assumptions in structured inputs.	Cannot by itself explore timing, lifecycle, or scenario state; structure-aware CAN fuzzing is a related analog [20,28,40].
Multi-dimensional mutation	User data, configuration, sensor messages, order, dependency-relevant variables.	Interactions among commands, parameters, sensor messages, ordering constraints, and dependency-relevant variables.	ROZZ and ROFER report more bugs than their baselines, but semantic faults require extra oracles and dependency-aware guidance [22,23,46].
Temporal mutation	Drop, delay, reorder, or duplicate events.	Ordering, timing, and asynchronous communication assumptions.	Often nondeterministic; replay and minimization become harder; protocol sequence guidance is a transferable analog [22,24,42].
Lifecycle mutation	Managed-state transitions, lifecycle pairs, order mutation, and delay injection.	ROS 2 lifecycle-related concurrency defects.	ROCF reports lifecycle-related C/C++ concurrency bugs, but evidence for other languages and semantic failures remains limited [25].
Scenario mutation	Roads, obstacles, weather, actors, drone paths, maps, perception triggers.	Safety failures, policy violations, unsafe trajectories, and task-level failures beyond ROS process crashes.	The observed failures depend on validity constraints, simulator fidelity, oracle design, and search scheduling [31–33,58,59,62,72–74].
Command-family scheduling	Simulator command generators and parallel scenario segments.	Simulator failures, command-sequence failures, and budget allocation in expensive campaigns.	Relevant for Gazebo-like systems and expensive campaigns, but transfer to application-level bugs requires careful reset design [29,35,74].

5.2.1. Structural Mutation

Mutation in ROS fuzzing should be described by the unit it changes. Interface-level mutators alter typed fields or configuration values, execution-level mutators alter order, timing, or interleaving, and system-level mutators perturb lifecycle transitions, scenarios,

or simulator commands. RoboFuzz, ROZZ, ROFER, and ROS data-modification DSLs show that structure-aware mutation can preserve validity while stressing message/data fields, configuration values, sensor streams, and topic events [22,23,28,67]. Structure-aware CAN fuzzing and ADGFUZZ provide near-domain lessons: message schemas, inferred field structure, or static dependencies can focus search on inputs likely to affect control logic [40,46].

The layer distinction is mainly a classification criterion for comparing studies. Field-level mutation primarily tests schema and value assumptions, temporal mutation tests ordering and asynchronous communication assumptions, and scenario mutation tests task and environment assumptions [22,23,25,28,31,32,40,46,59]. Existing studies rarely evaluate these layers under a common benchmark, so their reported outcomes are best compared by target layer and failure definition, with raw bug counts kept in context [22,23,25,29,31,59].

Structural mutation affects effectiveness through the unit it makes valid and the search space it enlarges. Type- and range-aware field mutation improves executable inputs but remains shallow for task semantics; multi-dimensional ROS mutation can expose interactions among commands, configuration, sensor data, and order, while increasing campaign space; startup-parameter and simulator-command mutation can change reachable paths or simulator state, but the cost shifts to parameter discovery, command-schema engineering, feedback collection, and manual reduction [21–23,28,29]. Coverage or bug gains should therefore be read with the coupled baseline, feedback signal, and execution setting.

5.2.2. Temporal and Concurrency Mutation

Temporal mutation addresses failures that depend on when and in what order ROS events occur. Dropping, delaying, reordering, or replaying messages can expose assumptions that value mutation cannot; ROZZ mutates message order, R2D2 uses callback traces as runtime-state guidance, and ROCF uses order mutation and delay injection over lifecycle transitions [22,24,25]. Compared with protocol fuzzing, ROS scheduling must drive robot task states through message, callback, lifecycle, and physical event sequences, and distributed graybox fuzzing suggests that timeline abstractions can guide such order-sensitive exploration [75]. MSGFuzzer reaches a similar conclusion for industrial robot protocols by guiding message-sequence exploration, which is close to ROS topics, services, and actions when failures depend on multi-message interaction, not on a single malformed payload [42].

Across these studies, temporal mutation can be separated into four mechanisms: perturbing message delivery order or timing, selecting callback-triggering events, mutating lifecycle-transition sequences, and guiding multi-message protocol sequences [22,24,25,42]. The first mechanism is portable but hard to replay deterministically; the second and third are more ROS-specific but require tracing or lifecycle instrumentation; the fourth serves as near-domain evidence for stateful robotic communication, but it should not be treated as quantitative proof that one ROS scheduling policy is dominant [24,25,42,75].

The critical comparison is the failure mechanism made observable. ROZZ supports order-aware message mutation for ROS tasks; R2D2 supports callback-trace feedback for runtime-state exploration; ROCF provides narrower evidence for lifecycle-related concurrency defects and reproducer construction; and MSGFuzzer supports the transferable point that robotic communication may need valid precondition sequences [22,24,25,42]. Temporal gains support bug-detection claims only when replay and oracle design match the targeted failure class; otherwise, coverage gains, lifecycle defects, and protocol crashes remain different outcomes [22,24,25,42,75].

5.2.3. Seed and Event Scheduling

Here, scheduling refers to choosing what to mutate next; the feedback signal itself is discussed later in the monitoring and feedback section. A ROS scheduler may choose among message-level seeds, input dimensions, callback or lifecycle events, scenario seeds, and simulator command families. ROFER schedules input dimensions according to their historical contribution to exploration [23], ROCF schedules lifecycle-transition sequences [25], and GzFuzz schedules structured simulator command generators [29]. Scenario-level evidence suggests that learning guidance, historical seed data, traffic objectives, map-aware mutation, and parallel segmentation can help allocate expensive simulation budgets [33,58,62,73,74].

Schedulers differ in the evidence they trust. Some reuse the past utility of input dimensions; others prioritize new runtime or lifecycle states, semantic or risk goals, diverse scenarios, simulator-command payoff, or parallel scenario segments [23,25,29,33,58,62,73,74]. A practical ROS scheduler has to combine type-aware and state-aware choices with scenario and command-family scheduling under a common execution protocol. Such a protocol should report failures with target layer, oracle type, modeling effort, and reproducibility, so that raw counts remain attached to the experimental setting that produced them [22,23,25,28,29,31,46,48,59].

The most suitable scheduler is therefore conditional: message-feature history fits multi-dimensional ROS input exploration, callback novelty fits runtime-state exploration, lifecycle-pair coverage fits lifecycle-concurrency testing, command-generator payoff fits simulator-as-target fuzzing, and risk or policy feedback fits scenario-level safety search [23–25,29–31,48,59]. Learning-based, map-aware, and parallel scheduling studies help with expensive scenario budgets, and their ROS-specific value remains a validation target before it can support an established ranking [58,60,73,74].

Once a seed or mutated event sequence is selected, it must be executed in a ROS environment that may include multiple nodes, middleware components, simulators, bridges, and physical state.

6. Executor

The executor is part of the experimental method. It determines how realistic the tested behavior is, how controllable reset and replay are, what runtime state can be observed, and how expensive a feedback signal is to collect. Because the reviewed studies pursue different experimental goals, execution and monitoring mechanisms are compared here by observable evidence, reset and replay control, feedback cost, and guidance value. These dimensions keep the comparison tied to evidence scope and experimental control, since targets, simulators, oracles, failure definitions, and execution budgets differ across studies [15,22–25,28,29,31].

6.1. Execution and Reset

6.1.1. Execution Environments

The executor of a ROS fuzzer usually launches a robotic system, not an isolated binary. Its environment may include ROS nodes, simulators, bridges, software- or hardware-in-the-loop (HIL) components, and physical robots. The reviewed evidence spans ROS and ROS-based executions, including hybrid execution, simulator-connected stacks, package-level tests, CAN/ROS boundary testing, and distributed RoboFuzz/ROS 2 deployments; near-domain studies add robotic-vehicle, protocol, CAV, CPS, and UAV analogs [28,30–33,35,36,39,41,42,48,53,57,76–78]. Across these settings, the main design issue is the balance among isolation, cost, and fidelity.

These settings can be compared through the realism-control trade-off. Package-level or software-only execution has low reset cost and can support high-throughput campaigns, but it mainly provides software, runtime, or memory-safety evidence [22–25]. Simulation-based execution adds task context and supports parallel campaigns, but reported failures can reflect simulator fidelity, scenario validity, reset bias, or simulator bugs in addition to ROS application faults [29–32,59]. Hybrid or real-robot execution increases physical realism, as illustrated by RoboFuzz, but introduces physical reset, battery, safety, test-site, and wear constraints [28]. Middleware, bridge, protocol, and simulator-as-target execution expose boundary assumptions, but their failures should not be interpreted as complete application-level robotic failures without additional validation [26,29,39,41,42,57,76,78].

Performance comparison is meaningful mainly within the same execution style. Package-level execution is easier to repeat and instrument; Gazebo-based ROS tasks balance controllability with task context; runtime or middleware execution exposes callback, lifecycle, and deserialization defects; simulator-as-target fuzzing measures simulator robustness; scenario-level simulation makes task failures observable at lower throughput; and hybrid or physical execution adds realism with manual reset, battery, safety, and site constraints [21–26,28–31,59].

6.1.2. Input Injection Channels

Execution also depends on where inputs are injected. A ROS fuzzer may inject data through normal ROS communication interfaces, runtime configuration, recorded traces, bridges, middleware boundaries, or simulator commands. Broken in Transit shows that malformed inputs can be injected at the middleware and deserialization boundary, bypassing ordinary ROS publishers [26]. ASTAA further shows the value of intercepting live messages in autonomy systems [50]. Protocol and bridge-oriented evidence suggests that injection below the ROS application layer can expose assumptions invisible to topic-level mutation alone [39,41,42,57,76,78]. The injection channel affects both the reachable behaviors and the oracle: a topic-level oracle may miss transport failures, while a middleware oracle may miss task-level degradation.

The injection channel also changes the reliability and interpretation of an observed result. Application-level ROS injection through topics, services, actions, parameters, or launch/configuration files usually preserves the semantics expected by nodes, but may miss serialization, bridge, or middleware failures [11,16,20,67,68]. Middleware-level and bridge-level injection can expose transport and deserialization assumptions, but a task-level oracle may not explain whether the observed failure is a transport error, a deserialization error, or an application-level behavior error [26,57]. Simulator-command and physical-environment injection can reach task-level failures, but they increase reset cost and make attribution depend on simulator state, bridge behavior, physical state, and oracle design [28,29,31,42,71].

6.1.3. Reset, Replay, and Nondeterminism

Reset is a central problem in ROS fuzzing because accumulated software, middleware, simulator, or robot state can be confused with a new failure or can hide one through nondeterminism. Practical reset may involve node cleanup, parameter restoration, simulator-world reset, DDS discovery cleanup, rosbag replay stabilization, and physical-state recovery. GzFuzz shows that the simulator itself may fail, while composable scenario construction clarifies which parts of a scenario can be varied and reset independently [29,71]. When simulation feedback, closed-loop scenario adaptation, parallel execution, or human interaction is part of the loop, reset also has to preserve comparable initial states, workload assumptions, and execution budgets; otherwise, differences in violation counts may reflect evaluation bias, not fuzzer quality [35,45,47,60,79].

Nondeterminism in ROS fuzzing comes from three main layers. Runtime nondeterminism arises from callback scheduling, lifecycle transitions, DDS discovery, buffering, and instrumentation. Simulation nondeterminism comes from physics, sensor models, and world state. Physical or human-in-the-loop tests add robot state, operator behavior, and learned components [15,24–26,29,47]. Each layer affects a different part of the workflow: runtime nondeterminism complicates feedback and reproduction, simulation nondeterminism affects violation counts and crash replay, and physical nondeterminism determines whether a failure can be repeated without changing the robot or environment. Existing mitigations combine software cleanup, middleware or simulator reset, trace-based replay, repeated-run confirmation, and test-case reduction [25,28,29,31,48,71].

Published results depend on reset and replay reporting. Without stable reset, differences in coverage, violation counts, runtime states, or bug reports may reflect residual state or nondeterministic scheduling rather than fuzzer behavior. The execution style also changes oracle credibility: physical and hybrid tests add realism but are harder to replay, so reports need the environment, reset policy, nondeterminism controls, replay or minimization support, and validation setting.

6.2. Monitoring and Feedback

Table 7 groups monitoring and feedback signals by what they make observable, how directly they can guide the next fuzzing decision, and what cost or evidence boundary they introduce. Feedback and oracle signals are listed together only when the same measurement can both guide exploration and support failure judgment; otherwise, feedback should be read as a search signal and not as proof that a failure occurred.

Table 7. Comparison criteria and evidence boundaries for monitoring, feedback, and oracle signals.

Signal Class	Observable Evidence	Actionability for Fuzzing	Cost/Evidence Boundary
Code or distributed coverage	Program paths across one or more ROS nodes; hybrid CPS state coverage.	Directly supports seed retention and path exploration in software/runtime campaigns.	Limited for task semantics; source instrumentation and overhead may be unreported or target-specific [20,22,80].
Message feature or protocol novelty	Communication-derived states, protocol sequences, topic patterns, and response behavior.	Can guide seed prioritization when source instrumentation is unavailable or distributed state is approximated through messages.	Feature quality, sequence validity, and replay cost affect reliability; ROFER and MSGFuzzer provide direct and near-domain evidence [23,42].
Callback trace and runtime-state novelty	Callback registration, execution, latency, executor behavior, lifecycle state, and OS scheduling context.	Targets ROS 2 runtime states that may be missed by branch coverage and message-only logging.	Requires tracing or instrumentation; timing perturbation and platform support must be considered [15,24].
Lifecycle and concurrency coverage	Managed-state transitions, call-stack pairs, thread interleavings, and delay-triggered executions.	Actionable for lifecycle-related concurrency exploration and reproducer construction.	Focused on a narrow bug class; evidence is most direct for C/C++ lifecycle-related ROS 2 programs [25].
Semantic or policy distance	Cyber-physical correctness, policy deviation, traffic rules, metamorphic relations, and temporal robustness.	Can guide search toward behavior-level failures and sometimes support oracle judgment.	Specification and domain-model costs are high; portability depends on reusable oracle templates [28,48,55,81,82].
Physical risk or quality	Collision risk, hazardousness, driving quality, path risk, blind spots, perception failures, and interaction risk.	Supports scenario-level search toward task-level or safety-relevant failures.	Domain- and simulator-dependent; execution cost, local optima, fidelity gaps, and diagnosis cost limit comparability [30,31,54,59–61].

Representative fuzzing studies operate at different evidence levels, so their reported findings do not denote a single kind of “bug”. Table 8 reports each representative method by target/evidence level, outcome type, and comparability boundary. The entries distinguish software and package-level bug reports, runtime and lifecycle defects, semantic correctness failures, policy violations, scenario-level safety outcomes, and transferable evidence, so the comparison emphasizes the layer being exercised and the kind of outcome that can be observed there.

Table 8. Evidence scope and outcome types of representative methods.

Work	Target/Evidence Level	Outcome Type	Comparability Boundary
ROZZ [22]	ROS 2 package/runtime evidence; multidimensional inputs and distributed coverage.	Memory and runtime bug reports.	Package-level ROS 2 results; semantic or task failures require different oracles.
ROFER [23]	ROS 2 package/runtime evidence; dimension mutation and message features.	Input-dimension and message-feature bug reports.	Depends on feature quality and target corpus; scenario outcomes require separate criteria.
R2D2 [24]	ROS 2 runtime/executor evidence; callback-trace feedback.	Runtime callback and execution bug reports.	Tracing-based runtime evidence; timing support and evaluation settings differ across studies.
ROCF [25]	ROS 2 lifecycle evidence; lifecycle-pair coverage.	Lifecycle concurrency defects and vulnerability disclosures.	Focused on C/C++ lifecycle behavior, not general ROS correctness.
RoboFuzz [28]	ROS system-level evidence; message mutation and correctness oracles.	Semantic correctness bug reports.	Oracle construction and selected robot systems determine scope.
PGFuzz [48]	Robotic-vehicle system evidence; policy-distance feedback.	Policy-violation bugs under temporal-policy oracles.	Policy/model-dependent outcomes; different from package-level ROS bug reports.
DriveFuzz [31]	ROS-based ADS scenario evidence; driving-quality feedback.	Driving-quality and scenario-level ADS bug reports.	ROS-based ADS evidence; simulator, scenario, and driving-rule oracles differ from ROS package-level tests.
DPFuzzer [59]	Drone path-planning evidence; environmental-risk feedback.	Scenario-level safety-critical cases.	Safety-critical scenarios are not ordinary software bug reports; results depend on planner and scenario models.
ADGFUZZ [46]	Near-domain robotic-vehicle evidence; assignment-dependency-guided input selection.	Transferable static-analysis-guided selection evidence.	Not ROS-specific; transfer requires mapping dependencies to ROS graphs and launch contexts.
SimADFuzz [60]	Near-domain ADS simulation evidence; trajectory and violation prediction.	Simulation-feedback and violation-prediction evidence.	ADS-specific scenarios and oracles; used as scheduling and feedback analog.
Active Sensor Fuzzing [44]	Near-domain CPS evidence; active sensor perturbation.	Model-guided sensor-test selection evidence.	Requires learned sensor-state models before ROS transfer.

Tables 7 and 8 separate two questions that are often conflated: what signal guides exploration, and what evidence supports a failure claim. Because studies differ in targets, benchmarks, baselines, execution budgets, oracle definitions, validation procedures, and outcome types, reported findings are compared by target layer, outcome type, actionability, cost, and evidence scope.

Monitoring quality is assessed here by observability, actionability, overhead, oracle alignment, portability, and diagnostic value. Structural and runtime signals, including coverage, message features, callback traces, lifecycle-pair coverage, tracing, and runtime-

verification monitors, observe software or execution state; semantic, policy, and physical-risk signals observe behavior-level outcomes. The first group adds instrumentation and overhead costs, while the second shifts cost toward domain models, reset stability, and oracle design [15,24,25,27,28,30,31,48,59].

The available quantitative monitoring evidence is signal-specific. R2D2 reports average latency and memory overhead of 10.4% and 1.0%, ROCF reports an 8.24% execution-time increase with ASan, ros2_tracing reports low average message-latency overhead in its ROS 2 tracing benchmark, and ROSMonitoring or ROMoSu quantify monitor overhead or selective monitoring [15,24,25,27,83]. These results support observability and cost claims; bug-discovery claims require a closed loop between feedback and reported failures.

6.2.1. Runtime Monitoring

Monitoring determines what the fuzzer can learn from each execution. Traditional branch coverage remains applicable in ROS settings but is incomplete for distributed runtime behavior. ROZZ uses distributed branch coverage over ROS programs [22]. ros2_tracing reports low-overhead tracing for ROS 2 messages, callbacks, services, executors, and lifecycle states in its evaluated setting [15]. R2D2 collects callback registration, execution, latency, buffer, and throughput information [24]. ROCF records call-stack-pair and lifecycle-pair sequences [25]. These monitoring substrates make ROS 2 runtime behavior observable to the fuzzer.

Runtime verification provides a complementary monitoring path. ROSMonitoring, ROSMonitoring 2.0, ROMoSu, and broader field-based testing show how executable specifications can support ROS assurance across distributions, services, ordered topics, and application monitors [27,83–85]. For fuzzing, such monitors have to remain portable and low-overhead enough for high-throughput exploration. Learning-guided network fuzzing, domain-based fuzzing for anomaly detection, active CPS fuzzing, and active sensor fuzzing suggest another monitoring direction: learning how network or sensor perturbations affect physical state and reusing that model to select informative tests or strengthen runtime monitors [44,86–88]. In ROS settings, this points toward feedback monitors that connect message perturbations to callback behavior, controller state, anomaly signals, and task-level consequences.

The monitoring mechanisms differ in precision and cost. Coverage instrumentation is directly usable by fuzzing schedulers but may saturate or miss semantic failures in long-running ROS tasks [20,22]. Tracepoint-based monitoring exposes callback, executor, lifecycle, and operating-system timing information, but requires platform support, trace analysis, and careful selection of events; reported overhead is setting-specific and should not be treated as a universal constant [15,24]. Runtime-verification monitors can express task or interface properties, but their portability depends on specification availability and maintenance [27,83–85]. Semantic and CPS monitors are closer to robotic outcomes, but they usually require domain knowledge and may reduce fuzzing throughput [28,30,31,48].

6.2.2. Feedback Signals

Feedback is not the same as an oracle. Feedback measures whether an execution is worth exploring further; an oracle judges whether the execution is faulty. ROS feedback has moved beyond code coverage. ROFER uses message features to approximate state transitions [23]; R2D2 uses callback traces [24]; ROCF uses lifecycle-pair coverage [25]; RoboFuzz uses semantic feedback such as sensor inconsistency, control error, state distance, and cyber-physical discrepancy [28]; and ROS-based, ROS-adjacent, or near-domain robotic-system fuzzers use policy distance, hazardousness, driving quality, traffic-management behavior, or risk factors [30,31,33,48,59]. Transferable CPS and ADS work adds feed-

back over hybrid CPS state, simulation trajectories, perception state, and learned sensor effects [44,60,63,65,72,80].

Actionability separates these signals. Structural feedback, such as code or distributed coverage, is cheap and scheduler-friendly but weak for semantic failure diagnosis [20,22]. Communication feedback, such as message features or protocol sequences, can approximate state without full source instrumentation, but its value depends on feature design and sequence validity [23,42]. Runtime-state feedback, such as callback traces and lifecycle-pair coverage, targets asynchronous execution and concurrency behavior but requires tracing or instrumentation [15,24,25]. Semantic, policy, and physical-risk feedback is closer to mission outcomes, but depends on domain models, stable reset, simulator fidelity, and oracle design [28,30,31,48,59]. These classes are interdependent and not interchangeable: runtime-state novelty may need semantic checks to decide whether behavior is faulty, while semantic feedback may need structural or trace evidence for diagnosis. A multi-layer feedback design remains a hypothesis supported by partial evidence from separate studies, not a validated general ROS fuzzing architecture.

6.2.3. Feedback-Guided Scheduling

Feedback- or risk-guided scheduling belongs in the monitoring-feedback loop because the scheduler reacts to observed execution evidence. R2D2 prioritizes payloads that expose new runtime callback states [24]. ROFER prioritizes message-feature and dimension contributions [23]. GzFuzz uses reinforcement learning to select command generators based on exploration payoff [29]. Safety-oriented fuzzers often use risk as the main feedback signal: PGFuzz uses distance to policy violation, Phys-Fuzz uses hazardousness, DriveFuzz uses driving quality, and DPFuzzer uses risk factors [30,31,48,59]. Related corpus evidence suggests that violation prediction, reinforcement learning (RL) seed optimization, map-aware mutation, multi-agent online search, and parallel segmentation can reduce wasted simulation runs [58,60,73,74,89].

Adjacent work suggests how ROS feedback can become more reusable: distributed graybox fuzzing uses timeline abstractions for order-sensitive systems, IJON exposes deep state variables, and CPFuzz combines program coverage with temporal-logic robustness over continuous plant behavior [75,81,90]. For ROS, the corresponding direction is a layered feedback signal that combines callback novelty, message-feature novelty, learned physical-state models, anomaly indicators, and simulator-derived violation probability [44,60,80,86–88]. Existing ROS-specific signals already include callback traces, lifecycle pairs, behavior-tree status graphs, message-feature states, and task-level physical distances, but they are rarely integrated in one reusable feedback channel.

Feedback-guided scheduling helps only when the signal matches the campaign goal. Coverage and callback novelty fit software and runtime exploration. Lifecycle feedback targets concurrency in managed-state execution. Risk, quality, and policy feedback fit scenario-level safety search, while command-family feedback is most relevant when the simulator or command server is part of the target [23–25,29–31,48,59]. Learning-based, map-aware, and parallel scheduling studies provide transferable design evidence, but they do not establish a dominant ROS scheduler [58,60,73,74]. A scheduler can improve exploration only when the signal is stable enough to compare runs, cheap enough to collect repeatedly, and aligned with the failure class targeted by the oracle.

Executor design has become part of the experimental method. A ROS fuzzing campaign must control reset, replay, runtime observability, and feedback cost together. Software-level studies usually provide higher-throughput and more reproducible evidence. Simulation adds task context. Hybrid and physical studies add realism. Semantic feedback improves behavioral guidance but requires domain modeling. Evaluation reports need

the execution setting, reset and replay policy, instrumentation cost, feedback signal, oracle type, and target layer together.

The next layer is failure evidence. Feedback can decide which executions deserve further exploration, but a separate failure-evidence layer is still needed to judge whether the observed behavior is faulty, reproducible, and diagnosable.

7. Bug Collector

A bug collector is meaningful only in relation to the failure level that its oracle can observe. Crash and sanitizer signals provide reproducible evidence for software, middleware, or runtime faults, but they do not by themselves establish whether a robot violates a task rule, a safety constraint, or a physical assumption. Conversely, semantic, physical, and task-level oracles can observe correctness and safety outcomes that do not manifest as crashes, but they require specifications, domain models, simulator state, or replay support. Table 9 compares oracle and diagnosis mechanisms by observable evidence, typical ROS-related use, main contribution, and cost or evidence boundary.

Table 9. Evidence boundaries and trade-offs of bug oracles in ROS fuzzing.

Failure/Evidence Level	Typical Oracle Evidence	Representative Use	Main Contribution	Cost/Evidence Boundary
Crash, memory, runtime exception	Crash logs, sanitizer reports, timeout, assertion failure, uncaught exception.	ROZZ, ROFER, R2D2, ROCF, GzFuzz [22–25,29].	Low-cost and reproducible evidence for software/runtime faults.	Misses many semantic, physical, and task-level failures.
Middleware and interface behavior	Message schema, serialization behavior, response consistency, DDS/Fast-CDR boundary failures.	Broken in Transit, property-based ROS testing, ros2_fuzz, MSGFuzzer [16,20,26,42].	Matches ROS communication and serialization boundaries.	A middleware failure may not explain task-level behavior.
Runtime state and lifecycle behavior	Callback order, lifecycle transition, trace timing, concurrency symptoms.	R2D2, ROCF, ROSInfer, ros2_tracing [15,24,25,70].	Captures distributed execution defects.	Requires tracing, instrumentation, and careful replay.
Semantic and specification correctness	Properties, temporal policies, differential outputs, metamorphic relations, runtime monitors.	Differential ROS Fuzzing, ASTAA, PGFuzz, ROSMonitoring, ROSpec [27,48,50,51,68].	Detects correctness violations beyond crashes.	Specification extraction, ordering, and monitor maintenance are costly.
Physical and task-level safety	Control instability, collision, unsafe distance, trajectory deviation, rule violation, task failure.	RVFuzzer, Phys-Fuzz, DriveFuzz, DPFuzzer, scenario fuzzers [30,31,53,59,60].	Connects tests to safety and mission outcomes.	Simulator fidelity, thresholds, domain knowledge, and execution cost limit portability.
Diagnosis and triage support	Replayable traces, minimized inputs, deduplicated failures, causal clues, dependency evidence.	ASTAA, PGFuzz, RoboFuzz, ROCF, ROSInfer, ros2_tracing [15,25,28,48,50,70].	Turns detected failures into repairable evidence.	Root-cause attribution remains partly manual and cross-layer.

Table 9 compares oracle effectiveness as a trade-off among confidence, semantic reach, replayability, and diagnosis cost. Crash and sanitizer oracles are narrow but reproducible for software, runtime, and middleware faults. Runtime, semantic, policy, differential, and physical oracles observe higher-level failures, but they require specifications, comparable implementations, monitor placement, thresholds, simulator state, and replay evidence [22–25,27,30,31,48,50,51,59,68].

7.1. Crash, Sanitizer, and Middleware Oracles

Crash and sanitizer oracles remain central to detecting low-level ROS failures. ROZZ, ROFER, R2D2, ROCF, Broken in Transit, and GzFuzz use bug checkers, sanitizers, crash/hang reports, lifecycle-aware reproduction, Fast Common Data Representation (Fast-CDR) boundary failures, or simulator crashes as evidence [22–26,29]. CAN, MAVLink, CAV, and CPS protocol fuzzing show that crash, timeout, malformed-response, and service-disruption evidence remains relevant at communication boundaries adjacent to ROS and DDS [39,41,76,78]. These oracles are well matched to memory errors, uncaught exceptions, deserialization failures, lifecycle-related crashes, and simulator-server crashes, but they cannot determine whether a robot completed a task safely or correctly. A ROS system may remain memory-safe while still producing unsafe motion, violating a policy, or behaving inconsistently across nodes. Thus, crash and sanitizer evidence should be interpreted as low-level software or middleware evidence, whereas semantic, policy, physical, and task-level failures require additional domain-specific oracles.

The distinction matters in distributed ROS systems. A node crash, Fast-CDR exception, or simulator crash is a high-confidence failure signal for the component that produced it, but it does not identify whether the root cause is an application input, a middleware type-contract violation, a lifecycle schedule, a simulator command, or a task-level safety condition [25,26,29]. Conversely, a policy or physical violation may occur without any process crash. Classic crash detection and CPS oracles are complementary: the former provides cheap and reproducible low-level evidence, whereas the latter expands observability to semantic and physical outcomes at higher modeling and replay cost.

7.2. Semantic and Specification-Based Oracles

Many ROS bugs concern correctness, not only memory safety. Differential ROS Fuzzing compares outputs from the ROS transform libraries TF and TF2 [51]. Property-based testing checks liveness, interface contracts, and user-defined properties [16]. AS-TAA uses system-specific invariants and runtime monitors [50]. PGFuzz expresses desired behavior as metric temporal logic policies [48]. ROSMonitoring, ROSMonitoring 2.0, ROMoSu, and runtime-verification work point toward executable oracles for ROS behavior [27,83–85]. Related oracle designs further show how metamorphic relations, rule specifications, and differential checks can reduce dependence on a single manually written ground truth [34,55,82].

Four reusable semantic-oracle patterns recur in these studies: interface or property checks, runtime monitors, differential or metamorphic comparisons, and policy or domain rules. Each pattern reduces a different ambiguity. Interface and property-based oracles make contracts and liveness expectations explicit; runtime-verification oracles attach properties to observed topics, services, event traces, or runtime-data streams; differential and metamorphic oracles compare implementations or follow-up executions; and policy or domain-specific oracles connect test outcomes to task expectations [16,27,34,48,50,51,68,82–85]. Their limits also differ: runtime monitors need specifications and stable event ordering, differential or metamorphic checks need comparable implementations or valid relations, and policy oracles need extracted rules and domain vocabulary.

Portability remains the main difficulty: a property written for TurtleBot navigation may not transfer to a manipulator, drone, or autonomous-driving stack. ROSpec can capture configuration and integration intent, but fuzzers still need reusable ways to translate specifications, documentation, launch files, and runtime traces into monitors [68]. A practical route is to keep the oracle template separate from the robot-specific vocabulary. The template may express an interface contract, temporal rule, metamorphic relation, differential comparator, or physical-risk threshold, while deployment binds it to specific

topics, frames, and tasks [34,55,82]. This need motivates intermediate oracle representations, specification mining, and hybrid differential checks. In practice, semantic-oracle evidence also depends on the execution and diagnosis pipeline. Properties need monitored events or logs, violations need stable replay, and traces are needed to distinguish true specification violations from monitor artifacts, simulator artifacts, or invalid environmental assumptions [27,50,84,85].

7.3. Physical and Task-Level Oracles

Physical oracles judge whether robot behavior is safe or correct in context. RVFuzzer detects control instability [53]. DiscoFuzzer searches for discontinuities in outputs that should vary continuously [49]. Phys-Fuzz uses collision and hazardousness-related outcomes [30]. DriveFuzz detects collisions, traffic violations, and stuck behaviors [31]. DP-Fuzzer detects drone path-planning and perception-related safety failures [59]. Additional scenario fuzzers encode physical oracles as safety potential, malicious actor behavior, blind-spot exposure, perception failure, realistic environmental perturbation, or predicted violation probability [54,60,61,65,66,91]. These oracles matter because a robot can remain memory-safe while still behaving dangerously.

The reviewed physical and task-level oracles differ in what they treat as evidence. Some use discrete safety events, such as collision, stuck behavior, or traffic-rule violation [31,59]; some use continuous control or risk metrics, such as control-state deviation, hazardousness, or driving quality [30,31,53]; and some use counterfactual or metamorphic checks to distinguish unavoidable outcomes from software-attributable failures [82]. These oracles make safety and mission outcomes observable, but they are not universally more informative than crash oracles. Their accuracy depends on calibrated thresholds, simulator fidelity, environmental assumptions, and whether replay can reproduce the same physical or scenario state.

The practical trade-off is oracle observability versus implementation cost. Physical or task-level oracles can expose failures that software-level checks miss, but they often require simulator instrumentation, scenario-state access, repeated executions, domain-specific thresholds, and manual review of borderline cases [30,31,53,82]. A physical violation also does not automatically diagnose the faulty component: the cause may lie in a mutated message, a controller parameter, a planner decision, a perception artifact, a simulator bridge, or the scenario itself.

7.4. Diagnosis, Triage, and Root Cause

Bug collection includes evidence collection, replay, minimization, deduplication, and root-cause analysis. ASTAA uses delta debugging to reduce failure-inducing message manipulations [50]. PGFuzz minimizes policy-violating sequences [48]. RoboFuzz validates findings through replay [28]. ROCF constructs minimal proof-of-concept sequences and uses delay injection to reproduce lifecycle bugs [25].

Diagnosis is harder than detection because the observed failure may be far from the injected perturbation. A mutated sensor message can propagate through planning, control, physical state, and oracle judgment before a failure appears. ROSInfer-style behavioral models and `ros2_tracing` traces provide inputs for diagnosis, but the field still lacks a general cross-node causal diagnosis pipeline with event slicing, dependency-aware minimization, timeline summarization, false-positive filtering, and root-cause ranking [15,70]. ADGFUZZ provides near-domain evidence here: by linking static assignment dependencies to matched input sets, it suggests that root-cause ranking in ROS could combine runtime traces with dependency information from source code, launch files, and message schemas [46].

The diagnosis bottlenecks can be prioritized by their role in turning a detected failure into repairable evidence. Replayability and reproducibility are the first requirement, because nondeterministic callbacks, middleware state, simulator state, and physical reset can otherwise make a failure unverifiable [25,28,50]. Minimization and deduplication are the next requirement, because message sequences, policy violations, and simulator scenarios can produce many similar failures that must be reduced before reporting [48,50]. Component-level localization and causal attribution remain harder: a violation may involve several nodes, a timing condition, a configuration choice, and a physical-state transition. Cross-layer explanation is the least mature step, because it requires combining message traces, callback or lifecycle traces, simulator state, oracle verdicts, and static dependency information [15,46,70].

Across representative ROS-specific and ROS-adjacent studies, replay, minimization, and root-cause localization recur as practical bottlenecks, while fully automated cross-layer causal diagnosis remains rare [15,25,28,31,48,50,70]. A ROS bug report that survives beyond its original benchmark has to preserve interface contracts, runtime traces, task policies, and physical observations through replay and minimization while still pointing to the responsible node, interaction, or environmental condition.

8. Challenges and Opportunities

The preceding workflow sections identify five bottlenecks for ROS fuzzing: shallow state reachability, isolated scheduling signals, inconsistent execution comparison, fragmented feedback, and weak diagnosis support. Table 10 turns these bottlenecks into research routes and evaluation criteria. The route selection is descriptive: it follows the workflow-stage coding and evidence boundaries used in this review, while paper counts and reported bug counts are treated as unsuitable for cross-route ranking. Direct ROS/ROS-based studies support near-term needs, while near-domain robotic/CPS/ADS studies indicate design opportunities that still require ROS-specific validation.

The routes are derived from three comparative observations in the preceding synthesis. Representative mature works concentrate primary contributions in model/input, feedback, oracle, and mutation/scheduling, while execution/reset and diagnosis/triage appear less often as primary stages. Direct ROS evidence is strongest for typed interfaces, runtime-state and lifecycle behavior, middleware, and target-specific oracles; scenario-level, learning-assisted, and physical-risk directions are supported mainly by near-domain or mixed-domain studies. Because benchmarks, detectors, execution settings, and failure definitions differ, each route is framed as a bottleneck with an evaluation requirement.

The proposed routes follow the same comparison results. Hierarchical seeds address the gap between interface validity and deeper state reachability; execution protocols address reset, replay, and nondeterminism differences; reusable feedback channels address fragmented observability and overhead; portable oracles address the cost of semantic and physical failure judgment; and diagnosis support addresses the gap between detecting a violation and localizing a repairable cause.

This route selection is also tied to paper-level evidence strength. Direct ROS fuzzers mainly support detection of crashes, memory or concurrency bugs, and target-specific correctness bugs; monitoring and tracing studies strengthen observability; and near-domain ADS/CPS evaluations warn that seeds, detectors, simulators, and metrics can change apparent effectiveness [15,22–25,27,28,45,50]. The remaining weak point is cross-layer causal explanation.

Table 10. Bottleneck-driven research routes for future ROS fuzzing.

Bottleneck	Evidence	Concrete route	Evaluation
Shallow state reachability	Valid interface-level inputs often fail to reach deeper robot states.	Build hierarchical seeds from interfaces, launch/configuration context, traces, behavior models, and scenarios.	Reached states, task phases, and failure-triggering sequences.
Isolated scheduling signals	Schedulers optimize different notions of progress.	Evaluate multi-objective scheduling across input, runtime-state, scenario, and budget dimensions.	Per-signal ablation, replay cost, and bug discovery.
Unreliable execution comparison	Simulation, hybrid execution, middleware injection, and parallel campaigns vary in fidelity, reset, and nondeterminism.	Report execution assumptions, reset/replay protocols, hooks, simulator or middleware issues, and benchmark artifacts.	Reproducibility rate, reset cost, nondeterminism controls, and cross-fuzzer comparability.
Fragmented feedback	Code, message, runtime, semantic, learned-state, and physical-risk signals are rarely evaluated together.	Expose reusable feedback channels and test which combinations remain actionable under realistic cost.	Per-layer contribution, overhead, and oracle alignment.
Weak diagnosis	Failures may be delayed, distributed, and difficult to connect to a responsible node, interaction, or environment.	Add event slicing, dependency-aware minimization, timeline summaries, differential checks, and root-cause ranking.	Replay success, minimized trace length, false positives, and diagnostic value.

Table 10 functions as an evidence map. It highlights where current studies repeatedly lose comparability, portability, or diagnostic value. Reusable seed construction, reset/replay protocols, feedback integration, portable oracles, and diagnosis support are near-term priorities because they appear in direct ROS/ROS-based evidence and affect several workflow stages [15,22–25,28,29,31,48]. Learning-assisted scenario selection, map-aware or multi-agent search, and parallel segmentation are more tentative: they provide design cues for search-budget allocation and long-horizon scenario exploration, but the supporting evidence is mainly near-domain and needs ROS-specific artifacts, ablations, and benchmarks before it can ground ROS-specific conclusions [45,73,74,89,92].

Three limitations qualify these routes. Reported findings and coverage improvements cannot be compared directly across benchmarks, oracles, execution settings, validation procedures, and outcome definitions; empirical ADS-fuzzing evaluation further shows that initial scenarios and detector definitions can bias cross-fuzzer comparison [45]. The 40 near-domain or mixed robotic/CPS/ADS studies provide workflow evidence; they support ROS-specific claims only when they explicitly evaluate a ROS or ROS-based target, including boundary cases that use Autoware or ROS infrastructure. Scenario-level methods depend on simulator fidelity and domain-specific task models, while runtime-level ROS fuzzers depend on instrumentation and target-language support. These limits

motivate reusable benchmarks, explicit execution assumptions, and portable feedback and oracle abstractions.

Corpus construction has limits. The review uses representative query families, manual screening, and snowballing, so source choice and near-domain inclusion can affect representativeness. Near-domain studies are therefore used as transferable workflow evidence unless they explicitly evaluate ROS or ROS-based targets.

Publication and discoverability bias remain possible: successful tools, confirmed bugs, indexed English-language papers, and stable artifacts are easier to find than null results, failed deployments, closed industrial work, local venues, or short-lived repositories. Combining databases, publisher pages, preprints, repositories, project pages, and snowballing reduced but did not remove this bias.

The review uses representative query and appraisal tables, but it does not provide a PRISMA-style exhaustive log, statistical publication-bias test, single quality score, or formal statistical sensitivity analysis. These choices reflect corpus heterogeneity: the records report different evidence types, outcome definitions, and artifact support. Borderline near-domain, tool, and technical records are therefore interpreted conservatively as workflow patterns, benchmark context, or validation needs, while ROS-specific conclusions rely on direct ROS/ROS-based evidence and repeated patterns across studies.

A related methodological limitation is that the classification, workflow, and research-route coding are qualitative. Author-team review and domain-expert assessment were used to improve boundary consistency. Future work could further strengthen the framework through independently coded literature snapshots, open coding protocols, expert-panel assessment of borderline cases, and benchmark-linked validation of the comparison dimensions.

9. Conclusions

Fuzzing for ROS-based robotic systems is best understood as an evidence-bound workflow problem. The analysis connects ROS target layers, fuzzing stages, and cyber-physical evidence to explain why results from ROS fuzzers cannot be compared by tool name or reported bug count alone. The corpus is most interpretable when the target layer, workflow contribution, execution setting, feedback signal, and oracle are reported together.

The corpus composition bounds these conclusions: direct ROS/ROS-based fuzzing studies account for 21 of the 92 selected works, while near-domain robotic/CPS/ADS studies account for 40. The review therefore separates ROS-specific conclusions from transferable patterns and taxonomy background, and near-domain studies are used as design analogs unless direct ROS or ROS-based evidence supports the claim.

Across the corpus, evidence shifts from interface and configuration validity toward runtime, lifecycle, middleware, simulator, and task-level behavior. This shift is uneven. Direct ROS or ROS-based studies provide the basis for ROS-specific claims, while near-domain robotic, CPS, and ADS studies mainly provide transferable design patterns or comparability warnings. More task-relevant feedback and oracles can make failures more meaningful, but they also increase the need for controlled execution, domain models, and replayable diagnosis.

As a qualitative synthesis, the review remains bounded by corpus completeness and coding judgment, even though workflow-stage and evidence-strength coding was cross-checked.

A reusable ROS fuzzing agenda has to make evidence easier to compare and repair. Progress depends on reaching meaningful robot states, controlling reset and replay, evaluating feedback by actionability and cost, stating the failure level of each oracle, and connecting observed failures to repairable causes. Near-domain techniques such as learning-assisted

scheduling, perception-aware seeds, LLM-assisted generation, and benchmark design remain sources of design ideas, but they require ROS-specific validation before they can support established ROS fuzzing conclusions.

The workflow-stage counts make this agenda concrete: among 32 representative mature fuzzing works, model/input, feedback, oracle construction, and mutation/scheduling are primary in 24, 19, 17, and 17 works, whereas execution/reset and diagnosis/triage are primary in only 7 and 3, respectively. Table 10 therefore treats replay and diagnosis as comparison-derived bottlenecks.

Future ROS fuzzing studies should report the target layer, execution setting, reset and replay policy, feedback signal, monitor or instrumentation overhead, oracle construction effort, artifact support, and diagnosis output together. These fields would make effectiveness, cost, scalability, and practical limitations comparable across studies without reducing heterogeneous ROS and robotic failures to a single bug count.

Author Contributions: Conceptualization, M.Y. and T.W.; methodology, M.Y., J.C., and T.W.; formal analysis, M.Y.; investigation, M.Y. and J.C.; resources, T.W. and G.Z.; data curation, M.Y. and J.C.; writing—original draft preparation, M.Y.; writing—review and editing, M.Y., J.C., T.W., T.G., X.F., X.K., and G.Z.; visualization, M.Y.; supervision, T.W.; project administration, T.W. and G.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: No new data were created or analyzed in this study. Data sharing is not applicable to this article.

Acknowledgments: The authors thank the developers and maintainers of the open-source tools and datasets discussed in this survey. During the preparation of this manuscript, the authors used OpenAI GPT-5.5 for language polishing support. The authors reviewed and edited the assisted outputs and take full responsibility for the content of this publication.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

ADS	Autonomous driving system
AFL	American Fuzzy Lop
AI	Artificial intelligence
API	Application programming interface
CAN	Controller Area Network
CAV	Connected autonomous vehicle
CPS	Cyber-physical system
DDS	Data Distribution Service
DSL	Domain-specific language
Fast-CDR	Fast Common Data Representation
HIL	Hardware-in-the-loop
HPC	High-performance computing
LLM	Large language model
MAVLink	Micro Air Vehicle Link
QoS	Quality of service
RL	Reinforcement learning
RMW	ROS Middleware

ROS	Robot Operating System
RTPS	Real-Time Publish-Subscribe
SROS2	Secure ROS 2
TF	ROS transform library
TF2	ROS transform library version 2
UAV	Unmanned aerial vehicle
YAML	YAML Ain't Markup Language

References

1. Quigley, M.; Gerkey, B.P.; Conley, K.; Faust, J.; Foote, T.; Leibs, J.; Berger, E.; Wheeler, R.; Ng, A.Y. ROS: An open-source Robot Operating System. In Proceedings of the ICRA Workshop on Open Source Software, Kobe, Japan, 12–17 May 2009.
2. Macenski, S.; Foote, T.; Gerkey, B.; Lalancette, C.; Woodall, W. Robot Operating System 2: Design, architecture, and uses in the wild. *Sci. Robot.* **2022**, *7*, 66. <https://doi.org/10.1126/scirobotics.abm6074>.
3. Al-Batati, A.; Koubaa, A.; Gabr, K.; Abdelkader, M.; Aloqaily, H. ROS 2 in a Nutshell: A Survey. *ACM Comput. Surv.* **2026**. <https://doi.org/10.1145/3815113>.
4. Miller, B.P.; Fredriksen, L.; So, B. An empirical study of the reliability of UNIX utilities. *Commun. ACM* **1990**, *33*, 32–44. <https://doi.org/10.1145/96267.96279>.
5. Manes, V.J.; Han, H.; Han, C.; Cha, S.K.; Egele, M.; Schwartz, E.J.; Woo, M. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Trans. Softw. Eng.* **2021**, *47*, 2312–2331. <https://doi.org/10.1109/tse.2019.2946563>.
6. Zhu, X.; Wen, S.; Camtepe, S.; Xiang, Y. Fuzzing: A Survey for Roadmap. *ACM Comput. Surv.* **2022**, *54*, 1–36. <https://doi.org/10.1145/3512345>.
7. Chur, N.; Santos de Moura, T.; Ortega, A.; Peldszus, S.; Berger, T.; Hochgeschwender, N.; Noller, Y. Before Autonomy Takes Control: Software Testing in Robotics. *arXiv* **2026**, arXiv:2602.02293. <http://arxiv.org/abs/2602.02293>.
8. Zhang, X.; Zhang, C.; Li, X.; Du, Z.; Mao, B.; Li, Y.; Zheng, Y.; Li, Y.; Pan, L.; Liu, Y.; et al. A Survey of Protocol Fuzzing. *ACM Comput. Surv.* **2024**, *57*, 1–36. <https://doi.org/10.1145/3696788>.
9. Daniele, C.; Andarzian, S.B.; Poll, E. Fuzzers for Stateful Systems: Survey and Research Directions. *ACM Comput. Surv.* **2024**, *56*, 1–23. <https://doi.org/10.1145/3648468>.
10. Chen, Z.; Chen, Z.; Cai, X.; Li, W.; Zheng, Z. An Empirical Study of Interaction Bugs in ROS-based Software. *arXiv* **2025**, arXiv:2507.10235. <http://arxiv.org/abs/2507.10235>.
11. Canelas, P.; Schmerl, B.; Fonseca, A.; Timperley, C.S. Understanding Misconfigurations in ROS: An Empirical Study and Current Approaches. In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA '24; ACM: New York, NY, USA, 2024; pp. 1161–1173. <https://doi.org/10.1145/3650212.3680350>.
12. Object Management Group. *The Real-Time Publish-Subscribe Protocol DDS Interoperability Wire Protocol Specification, Version 2.3*; Object Management Group: Needham, MA, USA, 2019. Available: <https://www.omg.org/spec/DDS-RTPS/2.3> (accessed on 15 July 2026).
13. Parra, S.; Schneider, S.; Hochgeschwender, N. Specifying QoS Requirements and Capabilities for Component-Based Robot Software. In Proceedings of the 2021 IEEE/ACM 3rd International Workshop on Robotics Software Engineering (RoSE); IEEE: New York, NY, USA, 2021; pp. 29–36. <https://doi.org/10.1109/rose52553.2021.00012>.
14. Tang, Y.; Feng, Z.; Guan, N.; Jiang, X.; Lv, M.; Deng, Q.; Yi, W. Response Time Analysis and Priority Assignment of Processing Chains on ROS2 Executors. In Proceedings of the 2020 IEEE Real-Time Systems Symposium (RTSS); IEEE: New York, NY, USA, 2020; pp. 231–243. <https://doi.org/10.1109/rtss49844.2020.00030>.
15. Bedard, C.; Lutkebohle, I.; Dagenais, M. ros2_tracing: Multipurpose Low-Overhead Framework for Real-Time Tracing of ROS 2. *IEEE Robot. Autom. Lett.* **2022**, *7*, 6511–6518. <https://doi.org/10.1109/lra.2022.3174346>.
16. Santos, A.; Cunha, A.; Macedo, N. Property-based Testing for the Robot Operating System. In Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, ESEC/FSE '18; ACM: New York, NY, USA, 2018; pp. 56–62. <https://doi.org/10.1145/3278186.3278195>.
17. Kromer, F. Hypothesis-ros2. GitHub Repository, 2019. Available online: <https://github.com/fkromer/hypothesis-ros2> (accessed on 15 July 2026).
18. Maruyama, Y.; Kato, S.; Azumi, T. Exploring the performance of ROS2. In Proceedings of the 13th International Conference on Embedded Software, ESWEK'16; ACM: New York, NY, USA, 2016; pp. 1–10. <https://doi.org/10.1145/2968478.2968502>.
19. Mayoral-Vilches, V.; White, R.; Caiazza, G.; Arguedas, M. SROS2: Usable Cyber Security Tools for ROS 2. In Proceedings of the 2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS); IEEE: New York, NY, USA, 2022; pp. 11253–11259. <https://doi.org/10.1109/iro47612.2022.9982129>.

20. ROSIN Project. *ros2_fuzz*: Fuzz Testing for ROS 2. GitHub Repository, 2020. Available online: https://github.com/rosin-project/ros2_fuzz (accessed on 15 July 2026).
21. Tian, Y.; Wang, R.; Wang, B.; Guan, Y. Fuzz Testing of ROS Software Packages. *SSRN Electron. J.* 2022, preprint. <https://doi.org/10.2139/ssrn.4238363>.
22. Xie, K.T.; Bai, J.J.; Zou, Y.H.; Wang, Y.P. ROZZ: Property-based Fuzzing for Robotic Programs in ROS. In Proceedings of the 2022 International Conference on Robotics and Automation (ICRA); IEEE: New York, NY, USA, 2022; pp. 6786–6792. <https://doi.org/10.1109/icra46639.2022.9811701>.
23. Bai, J.J.; Song, H.X.; Hu, S.M. 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 '24; ACM: New York, NY, USA, 2024; Volume 2, pp. 763–778. <https://doi.org/10.1145/3620665.3640425>.
24. Shen, Y.; Liu, J.; Xu, Y.; Sun, H.; Wang, M.; Guan, N.; Shi, H.; Jiang, Y. Enhancing ROS System Fuzzing through Callback Tracing. In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA '24; ACM: New York, NY, USA, 2024; pp. 76–87. <https://doi.org/10.1145/3650212.3652111>.
25. Gao, S.M.; Wang, P.; Bai, J.J.; Yu, J.W.; Wang, H. Detecting Lifecycle-Related Concurrency Bugs in ROS Programs via Coverage-Guided Fuzzing. *IEEE Trans. Inf. Forensics Secur.* 2025, 20, 7940–7953. <https://doi.org/10.1109/tifs.2025.3592562>.
26. Nwagwughia, S.; Toribio, J.; Blackstone, J. Broken in Transit: Detecting Type Confusion in ROS 2 Deserialization via Fuzzing. In Proceedings of the 2026 IEEE/SICE International Symposium on System Integration (SII); IEEE: New York, NY, USA, 2026; pp. 578–583. <https://doi.org/10.1109/sii64115.2026.11404710>.
27. Ferrando, A.; Cardoso, R.C.; Fisher, M.; Ancona, D.; Franceschini, L.; Mascardi, V. ROSMonitoring: A Runtime Verification Framework for ROS. In *Towards Autonomous Robotic Systems*; Springer International Publishing: Berlin/Heidelberg, Germany, 2020; pp. 387–399. https://doi.org/10.1007/978-3-030-63486-5_40.
28. Kim, S.; Kim, T. RoboFuzz: Fuzzing Robotic Systems over Robot Operating System (ROS) for Finding Correctness Bugs. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE '22; ACM: New York, NY, USA, 2022; pp. 447–458. <https://doi.org/10.1145/3540250.3549164>.
29. Ren, Z.; Li, Y.; Li, X.; Qi, G.; Xuan, J.; Jiang, H. Reinforcement Learning-Based Fuzz Testing for the Gazebo Robotic Simulator. *Proc. ACM Softw. Eng.* 2025, 2, 1467–1488. <https://doi.org/10.1145/3728942>.
30. Woodlief, T.; Elbaum, S.; Sullivan, K. Fuzzing Mobile Robot Environments for Fast Automated Crash Detection. In Proceedings of the 2021 IEEE International Conference on Robotics and Automation (ICRA); IEEE: New York, NY, USA, 2021; pp. 5417–5423. <https://doi.org/10.1109/icra48506.2021.9561627>.
31. Kim, S.; Liu, M.; Rhee, J.; Jeon, Y.; Kwon, Y.; Kim, C.H. DriveFuzz: Discovering Autonomous Driving Bugs through Driving Quality-Guided Fuzzing. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS '22; ACM: New York, NY, USA, 2022; pp. 1753–1767. <https://doi.org/10.1145/3548606.3560558>.
32. Wang, T.; Gu, T.; Deng, H.; Li, H.; Kuang, X.; Zhao, G. Dance of the ADS: Orchestrating Failures through Historically-Informed Scenario Fuzzing. In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA '24; ACM: New York, NY, USA, 2024; pp. 1086–1098. <https://doi.org/10.1145/3650212.3680344>.
33. Lin, S.; Chen, F.; Xi, L.; Wang, G.; Xi, R.; Sun, Y.; Zhu, H. TM-fuzzer: Fuzzing autonomous driving systems through traffic management. *Autom. Softw. Eng.* 2024, 31, 61. <https://doi.org/10.1007/s10515-024-00461-w>.
34. Wang, B.; Wang, R.; Song, H. Toward the Trustworthiness of Industrial Robotics Using Differential Fuzz Testing. *IEEE Trans. Ind. Informatics* 2023, 19, 2782–2791. <https://doi.org/10.1109/tii.2022.3211888>.
35. Del Río, F.B.G.; Lera, F.J.R.; Llamas, C.F.; Olivera, V.M., Fuzzing Robotic Software Using HPC. In *International Joint Conference 16th International Conference on Computational Intelligence in Security for Information Systems (CISIS 2023) 14th International Conference on European Transnational Education (ICEUTE 2023), Salamanca, Spain, 5–7 September 2023*; Springer Nature: Cham, Switzerland, 2023; pp. 23–33. https://doi.org/10.1007/978-3-031-42519-6_3.
36. Lin, S.; Chen, F.; Xi, L.; Xie, K.; Zheng, Y.; Fei, H.; Sun, Y.; Zhu, H. ScenarioFuzz-LLM: Enhancing Diversity in Autonomous Driving Scenario Fuzzing with LLMs. In *Proceedings of the 2025 28th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*; IEEE: New York, NY, USA, 2025; pp. 1581–1586. <https://doi.org/10.1109/cscwd64889.2025.11033362>.
37. Nguyen, G. A configurable approach to cyber-physical systems fuzzing. In *Proceedings of the 27th ACM International Systems and Software Product Line Conference, SPLC '23*; ACM: New York, NY, USA, 2023; Volume B, pp. 1–5. <https://doi.org/10.1145/3579028.3609005>.
38. Liu, J.; Yu, X.; Zhang, L.; Zhang, Y.; Li, Y.; Tan, K.; Tan, Y., Autonomous Driving System Configuration Defect Detection Method Based on Fuzzing. In *Digital Forensics and Cyber Crime*; Springer Nature: Cham, Switzerland, 2025; pp. 296–312. https://doi.org/10.1007/978-3-031-89360-5_18.
39. Fowler, D.S.; Bryans, J.; Cheah, M.; Wooderson, P.; Shaikh, S.A. A Method for Constructing Automotive Cybersecurity Tests, a CAN Fuzz Testing Example. In *Proceedings of the 2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*; IEEE: New York, NY, USA, 2019; pp. 1–8. <https://doi.org/10.1109/qrs-c.2019.00015>.

40. Kim, H.; Jeong, Y.; Choi, W.; Lee, D.H.; Jo, H.J. Efficient ECU Analysis Technology Through Structure-Aware CAN Fuzzing. *IEEE Access* **2022**, *10*, 23259–23271. <https://doi.org/10.1109/access.2022.3151358>.
41. Ananda, T.K.; Simran, T., G.; Sukumara, T.; Sasikala, D.; Kumar P., R. Robustness Evaluation of Cyber Physical Systems through Network Protocol Fuzzing. In *Proceedings of the 2019 International Conference on Advances in Computing and Communication Engineering (ICACCE)*; IEEE: New York, NY, USA, 2019; pp. 1–6. <https://doi.org/10.1109/icacce46606.2019.9079995>.
42. Zhang, Y.; Fang, D.; Liu, P.; Xi, L.; Lu, X.; Chen, X.; Si, S.; Sun, L. MSGFuzzer: Message Sequence Guided Industrial Robot Protocol Fuzzing. In *Proceedings of the 2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*; IEEE: New York, NY, USA, 2024; pp. 140–150. <https://doi.org/10.1109/icst60714.2024.00021>.
43. Woodlief, T.; Elbaum, S.; Sullivan, K. Semantic image fuzzing of AI perception systems. In *Proceedings of the 44th International Conference on Software Engineering, ICSE '22*; ACM: New York, NY, USA, 2022; pp. 1958–1969. <https://doi.org/10.1145/3510003.3510212>.
44. Zhang, F.; Wu, Q.; Xuan, B.; Chen, Y.; Lin, W.; Poskitt, C.M.; Sun, J.; Chen, B. Constructing Cyber-Physical System Testing Suites Using Active Sensor Fuzzing. *IEEE Trans. Softw. Eng.* **2023**, *49*, 4829–4845. <https://doi.org/10.1109/tse.2023.3309330>.
45. Yang, H.; Zhou, Y.; Chen, T. Empirical evaluation of simulation-based fuzz testing for autonomous driving systems. *Empir. Softw. Eng.* **2025**, *31*, 46. <https://doi.org/10.1007/s10664-025-10787-y>.
46. Wang, Y.; Zheng, Y.; Liu, P.; Fang, D.; Cheng, J.; Shi, D.; Sun, L. ADGFUZZ: Assignment Dependency-Guided Fuzzing for Robotic Vehicles. In *Proceedings of the 2026 Network and Distributed System Security Symposium, NDSS 2026*; Internet Society: Reston, VA, USA, 2026. <https://doi.org/10.14722/ndss.2026.231014>.
47. Chambers, T.; Vierhauser, M.; Agrawal, A.; Murphy, M.; Brauer, J.M.; Purandare, S.; Cohen, M.B.; Cleland-Huang, J. HIFuzz: Human Interaction Fuzzing for Small Unmanned Aerial Vehicles. In *Proceedings of the CHI Conference on Human Factors in Computing Systems, CHI '24*; ACM: New York, NY, USA, 2024; pp. 1–14. <https://doi.org/10.1145/3613904.3642958>.
48. Kim, H.; Ozmen, M.O.; Bianchi, A.; Celik, Z.B.; Xu, D. PGFUZZ: Policy-Guided Fuzzing for Robotic Vehicles. In *Proceedings of the 2021 Network and Distributed System Security Symposium, NDSS 2021*; Internet Society: Reston, VA, USA, 2021. <https://doi.org/10.14722/ndss.2021.24096>.
49. Rivera, S.; Iannillo, A.K.; State, R. DiscoFuzzer: Discontinuity-based Vulnerability Detector for Robotic Systems. *TechRxiv* 2020, e-print. <https://doi.org/10.36227/techrxiv.12613400.v1>.
50. Hutchison, C.; Zizyte, M.; Lanigan, P.E.; Guttendorf, D.; Wagner, M.; Le Goues, C.; Koopman, P. Robustness testing of autonomy software. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE '18*; ACM: New York, NY, USA, 2018; pp. 276–285. <https://doi.org/10.1145/3183519.3183534>.
51. Wang, Y.; Wang, B.Q.; Guan, Y.; Li, X.J.; Wang, R. Differential Fuzz Testing of Robot Operating System. *J. Softw.* **2021**, *32*, 1867–1881. <https://doi.org/10.13328/j.cnki.jos.006254>.
52. Delgado, R.; Campusano, M.; Bergel, A. Fuzz Testing in Behavior-Based Robotics. In *Proceedings of the 2021 IEEE International Conference on Robotics and Automation (ICRA)*; IEEE: New York, NY, USA, 2021; pp. 9375–9381. <https://doi.org/10.1109/icra48506.2021.9561259>.
53. Kim, T.; Kim, C.H.; Rhee, J.; Fei, F.; Tu, Z.; Walkup, G.; Zhang, X.; Deng, X.; Xu, D. RVFuzzer: Finding Input Validation Bugs in Robotic Vehicles through Control-Guided Testing. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA, August 2019; pp. 425–442. Available online: <https://www.usenix.org/conference/usenixsecurity19/presentation/kim> (accessed on 15 July 2026).
54. Li, G.; Li, Y.; Jha, S.; Tsai, T.; Sullivan, M.; Hari, S.K.S.; Kalbarczyk, Z.; Iyer, R. AV-FUZZER: Finding Safety Violations in Autonomous Driving Systems. In *Proceedings of the 2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*; IEEE: New York, NY, USA, 2020; pp. 25–36. <https://doi.org/10.1109/issre5003.2020.00012>.
55. Sun, Y.; Poskitt, C.M.; Sun, J.; Chen, Y.; Yang, Z. LawBreaker: An Approach for Specifying Traffic Laws and Fuzzing Autonomous Vehicles. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, ASE '22*; ACM: New York, NY, USA, 2022; pp. 1–12. <https://doi.org/10.1145/3551349.3556897>.
56. Guo, A.; Feng, Y.; Cheng, Y.; Chen, Z. Semantic-guided fuzzing for virtual testing of autonomous driving systems. *J. Syst. Softw.* **2024**, *212*, 112017. <https://doi.org/10.1016/j.jss.2024.112017>.
57. Aideyan, I.W.; Brooks, R.; Pese, M.D. Fuzzing CAN vs. ROS: An Analysis of Single-Component vs. Dual-Component Fuzzing of Automotive Systems. In *Proceedings of the WCX SAE World Congress Experience, Detroit, MI, USA, 16 April 2024*; SAE Technical Paper Series; ANNUAL; SAE International: Warrendale, PA, USA, 2024; Volume 1. <https://doi.org/10.4271/2024-01-2795>.
58. Jin, Q.; Wu, T.; Dong, Y.; Ding, Z.; Xu, Y. ReinSeed: Reinforcement Fuzz Testing With Multiphase Seed Optimization for Autonomous Driving Systems. *IET Softw.* **2025**, *2025*, 8657455. <https://doi.org/10.1049/sfw2/8657455>.
59. Wang, Y.; Yang, C.; Zhang, X.; Deng, Y.; Ma, J. DPFuzzer: Discovering Safety Critical Vulnerabilities for Drone Path Planners. In *Proceedings of the 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*; IEEE: New York, NY, USA, 2025; pp. 1973–1985. <https://doi.org/10.1109/icse55347.2025.00051>.

60. Yang, H.; Zhou, Y.; Chen, T. SimADFuzz: Simulation-Feedback Fuzz Testing for Autonomous Driving Systems. *ACM Trans. Softw. Eng. Methodol.* **2026**, *35*, 1–32. <https://doi.org/10.1145/3744242>.
61. Moussa, S.; Ma, J.; Abbasi, M.U. BlindSpotFuzz: Testing autonomous driving systems through blind-spot-guided fuzzing. *Autom. Softw. Eng.* **2026**, *33*, 52. <https://doi.org/10.1007/s10515-026-00597-x>.
62. Zhong, Z.; Kaiser, G.; Ray, B. Neural Network Guided Evolutionary Fuzzing for Finding Traffic Violations of Autonomous Vehicles. *IEEE Trans. Softw. Eng.* **2023**, *49*, 1860–1875. <https://doi.org/10.1109/tse.2022.3195640>.
63. Raj, S.; Jha, S.K.; Ramanathan, A.; Pullum, L.L. Testing autonomous cyber-physical systems using fuzzing features from convolutional neural networks: Work-in-progress. In *Proceedings of the Thirteenth ACM International Conference on Embedded Software 2017 Companion, ESWEK'17*; ACM: New York, NY, USA, 2017; pp. 1–2. <https://doi.org/10.1145/3125503.3125568>.
64. Moussa, S. Testing Autonomous Driving Systems Through Blind-Spot Guided Fuzzing. In *Proceedings of the 2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*; IEEE: New York, NY, USA, 2025; pp. 4141–4144. <https://doi.org/10.1109/ase63991.2025.00396>.
65. Pham, T.M.T.; Yang, B.; Yang, J. Perception-Guided Fuzzing for Simulated Scenario-Based Testing of Autonomous Driving Systems. *arXiv* **2024**, <https://doi.org/10.48550/ARXIV.2408.13686>.
66. Sun, Y.; Wang, H.; Poskitt, C.M.; Sun, J. Natural Adversaries: Fuzzing Autonomous Vehicles with Realistic Roadside Object Placements. *arXiv* **2024**, <https://doi.org/10.48550/ARXIV.2409.10562>.
67. Duffy, E. A DSL for Modifying Data in ROS Systems. Master's Thesis, Queen's University, Kingston, ON, Canada, 2024.
68. Canelas, P.; Schmerl, B.; Fonseca, A.; Timperley, C.S. ROSpec: A Domain-Specific Language for ROS-Based Robot Software. *Proc. ACM Program. Lang.* **2025**, *9*, 3313–3341. <https://doi.org/10.1145/3763169>.
69. Kim, H.Y.; Kim, G.; Lee, D.H.; Choi, W. BTreeFuzz: Enhanced Feedback Mechanism for ROS Program Fuzzer Based on Behavior Tree. In *Proceedings of the IEEE/ACM International Conference on Software Engineering, Rio de Janeiro, Brazil, 12–18 April 2026*.
70. Dürschmid, T.; Timperley, C.S.; Garlan, D.; Le Goues, C. ROSInfer: Statically Inferring Behavioral Component Models for ROS-Based Robotics Systems. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ICSE '24*; ACM: New York, NY, USA, 2024; pp. 1–13. <https://doi.org/10.1145/3597503.3639206>.
71. Ortega, A.; Parra, S.; Schneider, S.; Hochgeschwender, N. Composable and executable scenarios for simulation-based testing of mobile robots. *Front. Robot. AI* **2024**, *11*, 1363281. <https://doi.org/10.3389/frobt.2024.1363281>.
72. Hu, Z.; Guo, S.; Zhong, Z.; Li, K. Coverage-based Scene Fuzzing for Virtual Autonomous Driving Testing. *arXiv* **2021**, <https://doi.org/10.48550/ARXIV.2106.00873>.
73. Jin, Q.; Wu, T.; Ding, Z.; Xu, Y.; Dong, Y. Towards Adaptive Multi-Object Fuzzing: A Map-Aware Reinforcement Approach for Autonomous Driving Systems. In *Proceedings of the 2025 IEEE 31st International Conference on Parallel and Distributed Systems (ICPADS)*; IEEE: New York, NY, USA, 2025; pp. 1–10. <https://doi.org/10.1109/icpads67057.2025.11322949>.
74. Chen, F.; Lin, S.; Lin, W.; Xi, L.; Liu, Y.; Li, L.; Zhu, H. AS-Fuzzer: An Optimized ADS Fuzzing Method via Scenario Segmentation and Parallel Evolution. In *Proceedings of the 2024 31st Asia-Pacific Software Engineering Conference (APSEC)*; IEEE: New York, NY, USA, 2024; pp. 51–60. <https://doi.org/10.1109/apsec65559.2024.00016>.
75. Meng, R.; Pîrlea, G.; Roychoudhury, A.; Sergey, I. Greybox Fuzzing of Distributed Systems. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS '23*; ACM: New York, NY, USA, 2023; pp. 1615–1629. <https://doi.org/10.1145/3576915.3623097>.
76. Domin, K.; Symeonidis, I.; Marin, E. Security Analysis of the Drone Communication Protocol: Fuzzing the MAVLink Protocol. ORBilu e-Print, University of Luxembourg, 2016. Available online: <https://orbilu.uni.lu/handle/10993/37613> (accessed on 15 July 2026).
77. Rudo, D.; Zeng, K. Consumer UAV Cybersecurity Vulnerability Assessment Using Fuzzing Tests. *arXiv* **2020**, <https://doi.org/10.48550/ARXIV.2008.03621>.
78. Moukahal, L.J.; Zulkernine, M.; Soukup, M. Boosting Grey-box Fuzzing for Connected Autonomous Vehicle Systems. In *Proceedings of the 2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*; IEEE: New York, NY, USA, 2021; pp. 516–527. <https://doi.org/10.1109/qrs-c55045.2021.00080>.
79. Agarkar, A.S.; R, G.; Panda, M.K.; Srivastava, S. Simulation based Closed Loop Scenario Fuzzing for Safety Critical ADAS Applications. In *Proceedings of the 2023 IEEE 20th India Council International Conference (INDICON)*; IEEE: New York, NY, USA, 2023; pp. 759–764. <https://doi.org/10.1109/indicon59947.2023.10440801>.
80. Sheikhi, S.; Kim, E.; Duggirala, P.S.; Bak, S. Coverage-Guided Fuzz Testing for Cyber-Physical Systems. In *Proceedings of the 2022 ACM/IEEE 13th International Conference on Cyber-Physical Systems (ICCPs)*; IEEE: New York, NY, USA, 2022; pp. 24–33. <https://doi.org/10.1109/iccps54341.2022.00009>.
81. Shang, F.; Wang, B.; Li, T.; Tian, J.; Cao, K. CPFuzz: Combining Fuzzing and Falsification of Cyber-Physical Systems. *IEEE Access* **2020**, *8*, 166951–166962. <https://doi.org/10.1109/access.2020.3023250>.

82. Han, J.C.; Zhou, Z.Q. Metamorphic Fuzz Testing of Autonomous Vehicles. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, ICSE '20*; ACM: New York, NY, USA, 2020; pp. 380–385. <https://doi.org/10.1145/3387940.3392252>.
83. Stadler, M.; Vierhauser, M. ROMoSu: Flexible Runtime Monitoring Support for ROS-Based Applications. In *Proceedings of the 2023 IEEE/ACM 5th International Workshop on Robotics Software Engineering (RoSE)*; IEEE: New York, NY, USA, 2023; pp. 53–60. <https://doi.org/10.1109/rose59155.2023.00013>.
84. Ghaffari Saadat, M.; Ferrando, A.; Dennis, L.A.; Fisher, M. ROSMonitoring 2.0: Extending ROS Runtime Verification to Services and Ordered Topics. *Electron. Proc. Theor. Comput. Sci.* **2024**, *411*, 38–55. <https://doi.org/10.4204/eptcs.411.3>.
85. Caldas, R.; García, J.A.P.; Schiopu, M.; Pelliccione, P.; Rodrigues, G.; Berger, T. Runtime Verification and Field-Based Testing for ROS-Based Robotic Systems. *IEEE Trans. Softw. Eng.* **2024**, *50*, 2544–2567. <https://doi.org/10.1109/tse.2024.3444697>.
86. Chen, Y.; Poskitt, C.M.; Sun, J.; Adepur, S.; Zhang, F. Learning-Guided Network Fuzzing for Testing Cyber-Physical System Defences. In *Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*; IEEE: New York, NY, USA, 2019; pp. 962–973. <https://doi.org/10.1109/ase.2019.00093>.
87. Chen, Y.; Xuan, B.; Poskitt, C.M.; Sun, J.; Zhang, F. Active fuzzing for testing and securing cyber-physical systems. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA '20*; ACM: New York, NY, USA, 2020; pp. 14–26. <https://doi.org/10.1145/3395363.3397376>.
88. Wijaya, H.; Aniche, M.; Mathur, A. Domain-Based Fuzzing for Supervised Learning of Anomaly Detection in Cyber-Physical Systems. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, ICSE '20*; ACM: New York, NY, USA, 2020; pp. 237–244. <https://doi.org/10.1145/3387940.3391486>.
89. Liang, L.; Zheng, X. MARL-OT: Multi-Agent Reinforcement Learning Guided Online Fuzzing to Detect Safety Violation in Autonomous Driving Systems. *arXiv* **2025**. <https://doi.org/10.48550/ARXIV.2501.14451>.
90. Aschermann, C.; Schumilo, S.; Abbasi, A.; Holz, T. IJON: Exploring Deep State Spaces via Fuzzing. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP)*; IEEE: New York, NY, USA, 2020; pp. 1597–1612. <https://doi.org/10.1109/sp40000.2020.00117>.
91. Salgado, I.F.; Quijano, N.; Fremont, D.J.; Cardenas, A.A. Fuzzing Malicious Driving Behavior to find Vulnerabilities in Collision Avoidance Systems. In *Proceedings of the 2022 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*; IEEE: New York, NY, USA, 2022; pp. 368–375. <https://doi.org/10.1109/eurospw55150.2022.00044>.
92. Zhu, T.; Skapars, A.; Mackenzie, F.; Kehoe, D.; Newton, W.; Embury, S.; Sun, Y. SaFliTe: Fuzzing Autonomous Systems via Large Language Models. In *Towards Autonomous Robotic Systems*; Springer Nature: Cham, Switzerland, 2025; pp. 245–258. https://doi.org/10.1007/978-3-032-01486-3_20.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.