Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Empirical Evaluation of Simulation-based Fuzz Testing for Autonomous Driving Systems

Huiwen Yang · Yu Zhou* · Taolue Chen*

Received: date / Accepted: date

Abstract Autonomous driving systems (ADS) are an emerging technology with promising potential in areas such as intelligent cities and transportation. Testing ADS is of paramount importance before their deployment in real-world vehicles, where simulation-based testing provides a cost-effective way to assess the performance of ADS. Several simulation-based fuzz testing tools have been developed, but there lacks a systematic empirical study to evaluate and compare their performance. To address this, we propose the first comprehensive evaluation framework with unified initial driving scenarios and violation oracles to ensure fair comparisons. We conducted extensive experiments of over 500 hours. The results demonstrate that initial driving scenario datasets may impact the performance of fuzzers, and introduce potential evaluation biases. Furthermore, we consider two additional metrics, i.e., map waypoint coverage and code coverage. We find that the map waypoint coverage can be a complementary indicator to the number of unique violations to evaluate the performance of fuzz testing methods, whereas the code coverage fails to distinguish performance differences among fuzzers. These insights may provide guidance for comprehensively evaluating and comparing ADS simulation testing approaches in future studies.

Keywords Autonomous Driving System · Simulation-based Fuzz Testing · Empirical Evaluation

* Corresponding author

Huiwen Yang
Nanjing University of Aeronautics and Astronautics
E-mail: yhw_yagol@nuaa.edu.cn

Yu Zhou
Nanjing University of Aeronautics and Astronautics
E-mail: zhouyu@nuaa.edu.cn

Taolue Chen
Birkbeck, University of London
E-mail: t.chen@bbk.ac.uk

1 Introduction

Rapid advancements in artificial intelligence, particularly in computer vision, have accelerated the transition of autonomous driving technology from theoretical exploration to practical implementation (Grigorescu et al., 2020, Liu et al., 2021). Automotive companies (*e.g.,* Tesla[1]), technology giants (*e.g.,* Baidu[2]) and open-source organizations (*e.g.,* Autoware[3]) are actively developing Autonomous Driving Systems (ADS). These systems are designed to manage the entire driving process, from perception to decision-making and control, with the goal of reducing driver workload and minimizing traffic accidents (Gao et al., 2022). Despite the advancements, significant challenges remain in achieving low-risk autonomous driving. According to recent data from the California Department of Motor Vehicles,[4] 749 autonomous vehicle collisions had been reported as of September 2024. This highlights the need for rigorous validation and testing of ADS prior to their deployment on public roads.

Assessing the reliability of autonomous driving systems necessitates road testing spanning hundreds of millions of miles (Kalra and Paddock, 2016). However, real-world testing is hindered by high costs and, importantly, a limited range of scenarios, making it challenging to comprehensively address the vast diversity of driving conditions.

Simulation-based testing provides a promising complement by enabling the generation of diverse weather conditions, road network structures and traffic flows (Duy Son et al., 2019, Stocco et al., 2023). They can generate high-fidelity sensor data, including images and lidar, providing a cost-effective testing environment with rich scenario complexity. It is especially valuable for high-risk scenarios, such as collision events, where real-world testing raises significant safety concerns. Additionally, its ability to record and replay sensor data and vehicle control commands is highly practical. This reproducibility is a critical feature for regression testing, making simulation an indispensable tool in ADS development. Indeed, a survey found that 87% of participants use simulation platforms for testing autonomous driving systems (Lou et al., 2022).

Driving scenarios serve as the test case for simulation-based testing of ADS, the quality of which directly impacts the effectiveness of testing performance (Dai et al., 2024). Redundant, invalid or other low-quality scenarios may reduce testing efficiency, while diverse scenarios could rapidly trigger violation behaviors and help identify defects and vulnerabilities in subject ADS. Approaches that leverage fuzzing techniques (Li et al., 2022a, 2020) and adversarial-based generation techniques (Ding et al., 2020a, Wang et al., 2022) are widely adopted for scenario generation. In a nutshell, fuzzing-based methods leverage fitness functions to evaluate scenario quality and guide the

[1] Tesla Autopilot, `https://www.tesla.com/autopilot`

[2] Baidu Apollo, `https://apollo.baidu.com/`

[3] Autoware, `https://autoware.org/`

[4] DMV Autonomous Vehicle Collision Reports, `https://www.dmv.ca.gov/portal/vehicle-industry-services/autonomous-vehicles/autonomous-vehicle-collision-reports/`

search process, prioritizing scenarios that are more likely to trigger violations and generating offspring scenarios through various mutation strategies. In contrast, adversarial-based generation utilizes adversarial learning frameworks enhanced with domain-specific knowledge to produce scenarios that can induce system errors (Ding et al., 2023). Although adversarial methods can efficiently generate a large number of scenarios, controlling these scenarios is challenging due to the limited interpretability of deep learning models (Li et al., 2022a). Fuzzing-based methods account for multiple scenario attributes (*i.e.,* fitness objectives) and evaluate scenario quality through carefully designed fitness functions, making them more stable in generating high-quality scenarios.

Several tools have been developed for testing ADS based on simulation. For instance, DriveFuzz (Kim et al., 2022) randomly assigns starting points and destinations to the ego vehicle and selects scenarios for mutation based on the distance between the ego vehicle and other traffic participants during simulation. Similarly, AV-Fuzzer (Li et al., 2020) monitors the distance between the ego vehicle and other participants and employs a safety-oriented fitness function to guide scenario generation, thereby increasing the likelihood of identifying critical safety issues. Despite the success of recent fuzz testing methods, challenges remain in comparing their effectiveness due to inconsistencies in evaluation settings, which manifest to two aspects.

- **Initial Driving Scenarios.** Initial driving scenarios serve as the seed inputs for fuzz testing. Research in structured software fuzzing has shown that initial seeds significantly influence code branch coverage due to the complexity of program branching and stringent input validation (Asmita et al., 2024, Liang et al., 2021). Similarly, in ADS fuzz testing, complex scenarios (such as unprotected left turns at cross-intersection junctions), are more likely to expose safety violations. Consequently, the use of different initial driving scenarios in evaluations can lead to unfair comparisons. Additionally, relying on a single set of initial driving scenarios is insufficient for comprehensively assessing the performance of ADS fuzzers.
- **Violation Oracles.** Fuzz testing aim to uncover violation behaviors in ADS. However, fuzzers often employ different oracles even for detecting similar types of violations, resulting in discrepancies in their assertions. For example, both DriveFuzz and TM-Fuzzer (Lin et al., 2024) use a simple oracle to detect red-light violations by checking if ego vehicle's speed drops below 0.1 upon entering the influence zone of a red light. In contrast, AutoFuzz (Zhong et al., 2022) adopts a more sophisticated oracle inherited from the CARLA Leaderboard[5], which uses geometric calculations to determine whether the rear of the ego vehicle crosses the stop line after the light turns red. Such variations in oracles can introduce biases and hinder a fair comparison.

In this paper, we carry out a systematic empirical evaluation of simulation-based fuzz testing methods for ADS. We define driving scenarios as a data

[5] CARLA Leaderboard, `https://leaderboard.carla.org/`

model consisting of five key elements: ego vehicle, NPC (Non-Player Character) vehicles, pedestrians, puddles, and weather conditions. The datasets are constructed by leveraging multiple scenario generation strategies covering distinct maps provided by the CARLA simulator. Additionally, we design six violation detectors based on existing methods' violation oracles to identify collisions, running red lights, and other violations.

We develop ADSFuzzEval[6], which supports dockerization and parallelized execution of fuzz testing methods. Selecting two end-to-end ADS, *i.e.,* InterFuser (Shao et al., 2023) and LMDrive (Shao et al., 2024), and one of multi-module ADS, *i.e.,* Autoware, as the systems under test, we evaluate five fuzzers: AV-Fuzzer (Li et al., 2020), DriveFuzz (Kim et al., 2022), TM-Fuzzer (Lin et al., 2024), SAMOTA (Haq et al., 2022), and ScenarioFuzz (Wang et al., 2024).

Following a controlled-variable approach, we first investigate how initial driving scenarios affect fuzzer performance. The results demonstrate that initial driving scenarios, whether generated across different maps or within a single map, may influence the number of violations triggered by fuzzers, thus introducing potential evaluation biases.

We also consider two additional metrics, *i.e.,* map waypoint coverage and code coverage, to evaluate the performance of fuzzers. We find that map waypoint coverage could quantify the diversity of scenarios generated by fuzzers, which is supplementary to the number of unique violations for evaluating fuzzer performance. Regarding code coverage, our results confirm the known consensus that this metric is ineffective for testing end-to-end ADS. More importantly, we provide the empirical evidence that code coverage also fails to distinguish performance differences when testing multi-module ADS. Our analysis reveals this is because current fuzzers are not designed with code coverage-guided strategies, thus failing to explore code-logic within the ADS.

Structure. Section 2 provides the background of simulation-based fuzz testing in ADS. Section 3 introduces the study subjects of our evaluation. Section 4 describes the design of our empirical evaluation methodology. Section 5 presents and analyzes the results of our evaluation. Section 6 reviews related work, and Section 7 concludes the study and discusses future directions.

2 Background

Figure 1 illustrates the basic framework of simulation-based fuzz testing for ADS. First, the initial driving scenarios and the ADS under test are inputted into the simulation environment. During the simulation, the states of ego vehicles and other traffic participants are monitored and collected as feedback. The information is used to determine whether the ego vehicle triggers any potential violations based on the predefined violation oracles. Next, the fuzzer generates offspring scenarios based on its designed strategies, including scenario

[6] The implementation is publicly available at `https://github.com/yagol2020/ADSFuzzEval`

evaluation, selection, crossover and mutation, as shown in the green dashed box of Figure 1. The generated offspring scenarios are then fed back into the simulation environment for the next generation of testing. Simultaneously, the termination condition controls the testing budget. Once the termination condition is met, a violation report is produced.

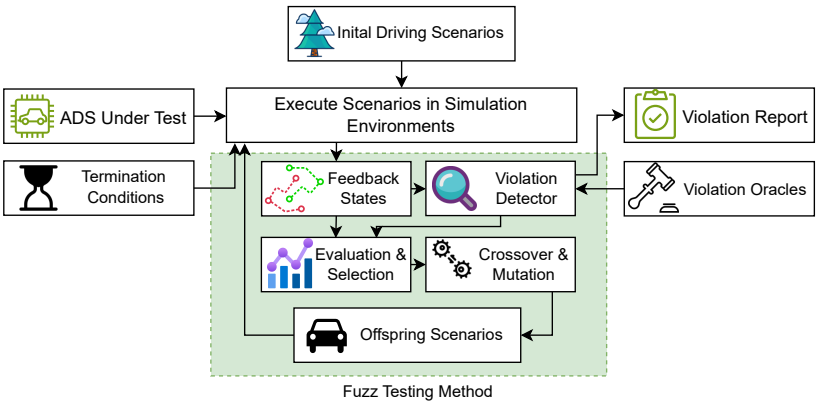


Fig. 1: Framework of Simulation-based Fuzz Testing for ADS

According to the framework, there are four inputs that are independent of the fuzz testing methods: (1) ADS under test, which is the system whose defects and violations the fuzzer aims to discover; (2) initial driving scenarios, which serve as the initial input for the fuzzer when no optimization strategies are enabled; (3) violation oracles, which are the criteria for determining whether the ADS has triggered any violations; and (4) termination conditions, which are used to control the testing budget.

To ensure a fair comparison, it is necessary to unify these inputs. The ADS under test and the termination conditions are consistent across different fuzz testing methods. For example, Apollo and Autoware are commonly used as the ADS under test, and the maximum execution time or maximum generation number are used as termination conditions. However, the initial driving scenarios and the violation oracles are sometimes not consistent due to their multiple configurable parameters. The following subsections will investigate these two components in detail.

2.1 Investigation of Initial Driving Scenarios

Driving scenarios are the test cases to be executed by the simulation environment. They are used to spawn vehicles and pedestrians, set weather conditions, road friction, and configure the traffic flow including the behavior of NPC vehicles and pedestrians.

Several approaches for describing driving scenarios have been proposed, including formal standards such as OpenSCENARIO,[7] domain-specific languages such as Scenic (Fremont et al., 2019), and various scenario specification methods, *e.g.,* Bakikian *et al.* (Babikian et al., 2024) and Bech *et al.* (Bach et al., 2016). However, these solutions are not widely used in existing fuzzers. Instead, most fuzzers tend to rely on the simulator's built-in API to load and configure driving scenarios. From an analysis of these fuzzers, we conclude that a driving scenario is typically composed of the following components.

- **Configuration of ego vehicles.** The ego vehicle's goal is to complete its driving task, which includes starting from an initial point and reaching the destination. For example, DriveFuzz and TM-Fuzzer randomly select the start and end points from all available locations on the map using the CARLA simulator API `map.get_spawn_points()` to define the driving task of the ego vehicle. Additionally, a vehicle model is selected to spawn the ego vehicle with different models having distinct attributes such as size, mass, and operability.
- **Configuration of participants.** NPC vehicles and pedestrians are important traffic participants that simulate realistic traffic flow in the simulation environment. Similar to the ego vehicle, the configuration of NPC vehicles and pedestrians includes their vehicle/pedestrian models, driving tasks defined by start and end points. Additionally, participants require behavior controllers to define their actions. For example, TM-Fuzzer utilizes the CARLA built-in autopilot agent `BehaviorAgent` to control NPC vehicles, while Doppel (Huai et al., 2023b) employs Apollo as the controller for NPC vehicles. DriveFuzz and ScenarioFuzz go beyond autopilot methods by incorporating additional strategies, such as immobile and linear behaviors, to enhance the diversity and realism of NPC behaviors.
- **Configuration of environment.** Weather conditions and road friction are key environmental parameters in the simulation environment. Weather conditions control elements such as clouds, rain, fog and wind, and they also affect lighting conditions on the road, which in turn impacts the sensor and perception modules of the ADS. Road friction determines the surface friction of the road, affecting the control module of the ADS. For example, AutoFuzz leverages the CARLA preset weather conditions[8], such as clear noon and cloudy sunset, to enhance the diversity of driving scenarios. DriveFuzz spawns puddles to simulate road friction, thereby affecting the actuation of the ADS.

Initial driving scenarios are the very first input for simulation-based fuzz testing methods. Table 1 summarizes the initial driving scenarios used by different fuzzers. As shown in the table, most fuzzers adopt straightforward strategies, such as randomly generating driving scenarios as initial input.

[7] OpenSCENARIO, `https://www.asam.net/standards/detail/openscenario`

[8] CARLA Weather Parameters, `https://carla.readthedocs.io/en/latest/python_api/#carlaweatherparameters`

Fuzzers (*e.g.,* AV-Fuzzer, AutoFuzz and ScenarioFuzz) predefine specific scenarios as initial inputs. For example, AutoFuzz manually designs five types of driving scenarios, such as lane changes and left turns at signalized junctions. However, these predefined scenarios inherently limit the diversity of the generated test cases, potentially reducing the effectiveness of the fuzzing process. In contrast, fuzzers like DriveFuzz, TM-Fuzzer and Doppel select the start and destination points for the ego vehicle from all available locations on the map. While this random strategy enhances scenario diversity, it may also introduce inconsistencies due to the complexity and variability of road networks.

Table 1: Initial Driving Scenarios of Different Fuzzers

Method	**Configuration of the Ego Vehicle**	**Configuration of Participants**				**Configuration of Environments**	
		NPC Vehicles		**Pedestrains**		**Puddle**	**Weather**
		Task	**Behavior**	**Task**	**Behavior**		
AV-Fuzzer	Predefined	Predefined		NOT SET	NOT SET	NOT SUPPORT	NOT SUPPORT
DriveFuzz	Random	Random	A,M,L,I	Random	A, L, I	Random	Random
AutoFuzz	Predefined	Predefined		Predefined		Random	Random
TM-Fuzzer	Random	NOT SET	NOT SET	NOT SET	NOT SET	NOT SET	Random
ScenarioFuzz	Predefined	Random	A, M, L, I	Random	A, L, I	Random	Random
Doppel	Random	Random	A	Random	A	NOT SUPPORT	NOT SUPPORT
SAMOTA	Random	Random	A	Random	A	NOT SUPPORT	Random

* A: autopilot, M: maneuver, L: linear, I: immobile, NOT SET: the fuzzer does not set this element, NOT SUPPORT: the simulator fuzzer used does not support this element.

2.2 Investigation of Violation Oracles

Violation oracles are used by detectors to dynamically identify improper behaviors exhibited by the ADS during testing. We categorize the violation oracles into three types: collision, infraction and task failures, and then classify seven state-of-the-art fuzzers into three categories, as shown in Table 2.

Collision. Collision is the most fundamental violation that an ADS must avoid. It occurs when the ego vehicle comes into contact with static or dynamic objects, exerting force on either the ego vehicle or the impacted object. In real-world scenarios, collisions can lead to severe economic losses, vehicle damage, and potentially life-threatening injuries. As a result, detecting and preventing collision events is critical for the reliability and safety of ADS.

CARLA provides convenient API to detect collision events[9], making it straightforward for fuzzers to detect collisions using a consistent oracle.

Infraction. Infraction refers to violations of traffic rules and laws, including behaviors such as speeding, encroaching into opposing lanes and running red lights. Traffic laws provide a legal framework that ensures the safety of all traffic participants. With the increasing integration of autonomous vehicles into

[9] Collision Detector, `https://carla.readthedocs.io/en/latest/ref_sensors/#collision-detector`

Table 2: Violation Oracles Employed by Different Fuzzers

Violation Oracles		AV-Fuzzer	DriveFuzz	TM-Fuzzer	AutoFuzz	Doppel	SAMOTA	ScenarioFuzz
Collision		✔	✔	✔	✔	✔	✔	✔
Infraction	Speeding	✘	✔	✔	✘	✘	✘	✔
	Lane Invasion	✘	✔	✔	✔	✘	✔	✔
	Running Red Lights	✘	✔	✔	✘	✔	✔	✔
Task Fail	Reach the Destination	✘	✔	✔	✘	✘	✔	✘
	Stuck	✘	✔	✔	✘	✘	✘	✔
	Module Respnse Failures	✘	✘	✘	✘	✔	✘	✘

human-driven traffic, it is critical for autonomous vehicles to adhere strictly to these regulations (Li et al., 2024, Ma et al., 2024, Sun et al., 2022). Infractions by autonomous vehicles, even if they do not result in collisions, can significantly elevate safety risks.

For lane invasion, the CARLA simulator provides API similar to the collision detection API, which detects this violation based on OpenDRIVE.[10] However, for other violations, different fuzzers employ diverse oracles, leading to inconsistencies in detection criteria. As mentioned in the introduction section, the detection oracle for running red lights varies between fuzzers. Similarly, for speeding violations, the oracle differ across fuzzers. For instance, DriveFuzz determines speeding by checking if the current speed of the ego vehicle exceeds the road's speed limit,[11] whereas TM-Fuzzer defines speeding as exceeding the speed limit by 2 km/h.[12] These inconsistencies in oracle definitions introduce biases when evaluating the performance of ADS under test.

Task Failure. This category encompasses situations where the ADS fails to complete its assigned driving task, such as failing to reach its destination due to internal issues like perception errors, localization failures or software instability. Manifestations of such failures may include the vehicle remaining stationary for prolonged periods or encountering module-specific errors that prevent task completion. Although this type of failure generally presents a lower safety risk, it highlights the inability of the ADS to autonomously fulfill its driving responsibilities, thereby necessitating manual intervention. If the driver fails to take over promptly and safely, such failures could escalate into safety-critical incidents.

The oracle for determining whether the ADS has reached its destination or is stuck relies on user-specified thresholds. For instance, the default threshold for judging a "stuck" state differs across fuzzers: 20 seconds in DriveFuzz, 60 seconds in TM-Fuzzer, and 30 seconds in ScenarioFuzz. Similarly, the criteria for determining whether the ADS has reached its destination also vary. For DriveFuzz, TM-Fuzzer, and ScenarioFuzz, the threshold is a 2-meter distance between the ego vehicle and the goal. However, TM-Fuzzer introduces an ad-

[10] Lane invasion detector, `https://carla.readthedocs.io/en/latest/ref_sensors/#lane-invasion-detector`

[11] Speeding oracle of DriveFuzz, `https://gitlab.com/s3lab-code/public/drivefuzz/-/blob/master/src/executor.py?ref_type=heads#L1080`

[12] Speeding oracle of TM-Fuzzer, `https://github.com/ldegao/TMfuzz/blob/main/simulate.py#L1308`

ditional condition: the ego vehicle's speed must drop below 0.1 km/h.[13] This stricter requirement differentiates TM-Fuzzer from other fuzzers.

2.3 Summary of findings

Based on the systematic investigation, we summarize the key findings regarding initial driving scenario generation and violation oracle implementation as follows.

Initial Driving Scenario Generation. Current practices predominantly employ random-based strategies for generating initial driving scenarios. However, such approaches may introduce evaluation inconsistencies due to uncontrolled variations in three critical dimensions: (1) driving task parameters, (2) traffic flow characteristics, and (3) environmental conditions. This variability underscores the necessity for a consistent scenario generation framework to enable fair comparison of different fuzz testing methodologies.

Violation Oracle Implementation. While violation oracles serve as essential module for identifying anomalous ADS behaviors, our analysis reveals two critical challenges: (1) definitional ambiguity in violation criteria (*e.g.,* traffic infraction classifications, and mission failure conditions) and (2) implementation discrepancies across testing platforms. These dual challenges potentially introduce evaluation biases, emphasizing the urgent need for consistent oracle specifications to ensure objective cross-fuzzer comparisons.

The performance results, *e.g.,* the number of unique violations, may be influenced by the two components discussed above. Accordingly, in order to conduct a fair and comprehensive empirical evaluation of ADS fuzz testing methods, we need to consistently control these two variables, namely, the initial driving scenarios and the violation oracles. This allows us to eliminate their potential influence on the evaluation results and focus on comparing the violation detection capabilities of different fuzz testing methods.

3 Study Subjects

The subject of our empirical study consists of two components: the fuzz testing tools and the ADS under test. This section will introduce the study subjects selection criteria, followed by a brief introduction of the chosen subjects.

3.1 Fuzz Testing Tools

Table 3 lists 12 ADS fuzz testing tools published in the literature, including the simulator they support, the number of citations according to Google Scholar, the publication details, and their repositories.

[13] Reach the destination oracle of TM-Fuzzer, `https://github.com/ldegao/TMfuzz/blob/main/simulate.py#L754`

Table 3: List of ADS Fuzz Testing Tools

Fuzzer	Simulator	Citation	Publication	Repository Url
AV-Fuzzer (Li et al., 2020)	CARLA* / LGSVL	189	ISSRE 2020	https://github.com/cclinus/AV-Fuzzer
AutoFuzz (Zhong et al., 2022)	CARLA / LGSVL	81	TSE 2023	https://github.com/autofuzz2020/AutoFuzz
SAMOTA (Haq et al., 2022)	CARLA	76	ICSE 2022	https://figshare.com/articles/journal_contribution/16468530
MOSAT (Tian et al., 2022)	LGSVL	42	ESEC/FSE 2022	https://github.com/mogatesing/mosat
LawBreaker (Sun et al., 2022)	LGSVL	40	ASE 2022	https://github.com/lawbreaker2022/LawBreaker-SourceCode
Doppel (Huai et al., 2023b)	SimControl	28	ICSE 2023	https://github.com/Software-Aurora-Lab/DoppelTest
DriveFuzz (Kim et al., 2022)	CARLA	21	CCS 2022	https://gitlab.com/s3lab-code/public/drivefuzz
scenoRITA (Huai et al., 2023a)	LGSVL	16	TSE 2023	https://github.com/Software-Aurora-Lab/scenoRITA
ScenarioFuzz (Wang et al., 2024)	CARLA	4	ISSTA 2024	https://github.com/AtongWang/ScenarioFuzz
FuzzScene (Guo et al., 2024)	CARLA	3	JSS 2024	https://github.com/meng2180/FuzzScene
TM-Fuzzer (Lin et al., 2024)	CARLA	1	ASEJ 2024	https://github.com/ldegao/TMfuzz
VioHawk (Li et al., 2024)	LGSVL	1	ISSTA 2024	https://github.com/emocat/VioHawk

* Re-implementation based on CARLA simulator.

It can be seen that CARLA (Dosovitskiy et al., 2017) and LGSVL (Rong et al., 2020) are two widely used simulators for testing ADS. However, LGSVL ceased maintenance in 2022, which limits access to certain maps and assets. While open-source derivatives (*e.g.,* SORA-SV[14]) can partly alleviate this limitation, ensuring their reliability and robustness still requires significant effort. Moreover, their outdated simulation engines may introduce non-determinism, leading to flaky testing results (Amini et al., 2024, Osikowicz et al., 2025). In contrast, CARLA receives consistent updates, including improvements to its maps, support for new features such as native ROS 2, an upgrade of its game engine from Unreal 4 to Unreal 5, and fixes for bugs and vulnerabilities. These enhancements make CARLA more stable and less prone to non-determinism issues. Furthermore, CARLA hosts official ADS performance competitions via the CARLA Leaderboard, establishing it as a preferred choice for comprehensive ADS testing in both research and industry.

Therefore, our study prioritizes CARLA as the primary simulator for testing ADS. Accordingly, we exclude five fuzzers that rely on either the LGSVL or SimControl simulators (Apollo's built-in simulators). We also remove FuzzScene (Guo et al., 2024), which is specifically designed to test steering-angle models rather than the entire ADS, and AutoFuzz (Zhong et al., 2022), whose execution is highly coupled with the CARLA Leaderboard, making it difficult to adapt to the initial scenario and extend the violation detectors. As a result, our study focuses on the remaining five open-source fuzzers, which are described as follows.

– **AV-Fuzzer (Li et al., 2020).** AV-Fuzzer is a fuzz testing method that employs genetic algorithms to generate safety-violating scenarios by perturbing the driving behavior of traffic participants to increase safety risks. It designs a fitness function that evaluates the ego vehicle's safety potential based on its safe driving distance and the maximum braking distance under comfortable braking conditions. Scenarios with higher risk are selected for further mutation. Although the original version of AV-Fuzzer was imple-

[14] SORA-SVL, https://github.com/YuqiHuai/SORA-SVL

mented in LGSVL, Tong *et al.* re-implemented its strategies in CARLA[15]. Given that AV-Fuzzer is widely used as a baseline in related works, we include the re-implemented version in our study.

- **DriveFuzz (Kim et al., 2022).** DriveFuzz is a feedback-guided fuzz testing method that collects information about the ego vehicle's throttle, brake, and steering states during scenario execution. It designs a fitness function that considers hard acceleration, braking, oversteering, and the minimum distances to other vehicles or pedestrians. Scenarios with the highest fitness scores are selected for mutation. Additionally, DriveFuzz simulates various vehicle behaviors and different weather conditions to generate diverse test scenarios.
- **SAMOTA (Haq et al., 2022).** SAMOTA is a method that combines many-objective search algorithms with surrogate-assisted optimization to address the challenges of generating diverse test data that can trigger safety violations. It leverages surrogate models to mimic computationally expensive simulators, thereby significantly reducing the cost of fitness evaluations.
- **TM-Fuzzer (Lin et al., 2024).** TM-Fuzzer is a fuzz testing method that dynamically manages NPC vehicles during simulation to generate traffic flow. By spawning vehicles in front of the ego vehicle or removing vehicles behind it during the simulation process, TM-Fuzzer increases the ability to trigger more edge behaviors of the ADS. Furthermore, TM-Fuzzer incorporates clustering analysis to identify the similarity between scenarios and improve the quality of test scenarios.
- **ScenarioFuzz (Wang et al., 2024).** ScenarioFuzz leverages map crawling techniques to construct a topological graph from the road semantic structure, which is used to build a scenario seed corpus. It then optimizes scenario selection strategies based on a graph neural network model to predict and filter out high-risk scenarios for further mutation.

3.2 ADS Under Test

Autonomous driving systems rely on sensory inputs, such as data from RGB cameras, radar, GPS and inertial measurement units, combined with high-definition maps that define the road network structure. These inputs enable ADS to make informed decisions based on pre-trained models (Zhao et al., 2023). Typically, ADS can be categorized into two types according to their system architectures (Zhao et al., 2023): *multi-module* and *end-to-end*. Multi-module ADS consists of multiple modules, each responsible for specific tasks such as perception, prediction and route planning. In contrast, end-to-end ADS directly map sensory inputs to control signals, relying on deep learning techniques to learn human driving patterns directly.

To comprehensively evaluate autonomous driving systems, we reviewed both types of ADS and selected representatives that meet the following cri-

[15] Re-implemented AV-Fuzzer: `https://github.com/AtongWang/ScenarioFuzz/blob/master/src/fuzzer_avfuzz.py`

teria.(1) **Availability**: the ADS is open-source, including its source code and model weights;(2) **Compatibility with the CARLA Simulator**: the ADS should be able to receive sensor data from CARLA and output control signals that CARLA supports.

These criteria are designed to ensure that the selected ADS enable consistent and reliable testing, thereby providing valuable insights into their reliability and performance in autonomous driving scenarios. By applying these criteria, we selected the following three ADS.

- **Autoware** is an open-source multi-module ADS that provides a complete software stack for autonomous driving. Autoware is built on ROS and enables commercial deployment of autonomous driving in a broad range of vehicles and applications. It has been widely adopted in the autonomous driving community and is actively maintained.
- **InterFuser (Shao et al., 2023)** is an end-to-end explainable ADS that employs transformer-based multimodal sensor fusion. It integrates information from multiple sensors to achieve global contextual awareness and generates interpretable intermediate outputs. These outputs are then processed by a safety-enhancing controller, which constrains actions within a safe set. InterFuser outperformed other methods including TCP (Wu et al., 2022), LAV (Chen and Krähenbühl, 2022) and TransFuser (Chitta et al., 2022), and secured second place on the CARLA Leaderboard,[16] demonstrating its ability to safely and efficiently handle complex, adversarial urban scenarios.
- **LMDrive (Shao et al., 2024)** is an end-to-end closed-loop ADS leveraging large language models (LLMs). It processes data from cameras and LiDAR sensors, interprets driving commands expressed in natural language, and directly generates vehicle control signals. LMDrive utilizes the pre-trained multimodal LLM, LLaVA, to process sensor data and reason with natural language commands to produce driving actions. This system represents cutting-edge advancements in large model technology.

According to Google Scholar statistics, InterFuser and LMDrive receive 217 and 100 citations, respectively, and Autoware has over 10.3k stars on GitHub, demonstrating their popularity in the autonomous driving system domain. Notably, ADSFuzzEval is not limited to these ADS; any ADS meeting both criteria (1) and (2) can be integrated with minimal modifications.

4 Design of ADSFuzzEval

We develop ADSFuzzEval to fairly evaluate ADS fuzz testing tools. The overall process is illustrated in Figure 2.

First, ADSFuzzEval uses Docker to encapsulate both the ADS under test and the fuzz testing methods, ensuring a consistent and isolated system environment. Specifically, ADSFuzzEval builds Docker images for the ADS and

[16] Accessed in January 2025.

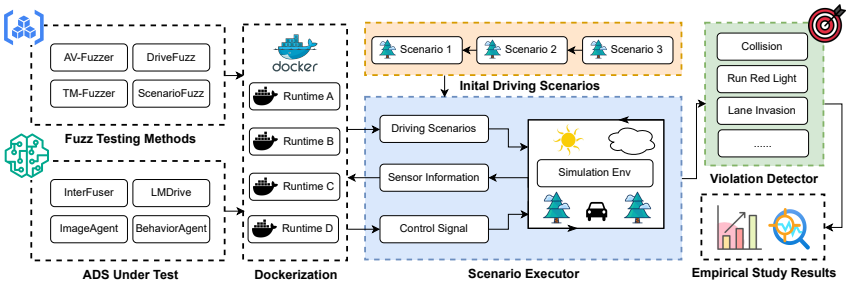


Fig. 2: Overview of ADSFuzzEval

fuzzers and allocates identical computational resources, including CPU, memory and GPU (via Docker's `--cpus`, `--memory` and `--gpus` options), to each runtime container during evaluation. Additionally, Docker enables ADSFuzzEval to support parallelized evaluations, significantly improving the efficiency of the testing process.

Next, ADSFuzzEval defines a structured data model for driving scenarios. Based on this data model, ADSFuzzEval employs multiple strategy to generate initial datasets of driving scenarios. Additionally, we modified the seed input component of fuzzers to ensure they start exploration from the same set of initial driving scenarios. The details of this process are described in Section 4.1.

Subsequently, the initial driving scenarios are loaded by the runtime containers of the fuzzers. The fuzzers generate offspring driving scenarios and send them to the simulation environment to construct specific simulation scenarios. Simultaneously, the ADS within the runtime container receives sensor information from the simulation environment, performs perception and decision-making, and sends control signals back to the simulation environment to control the behavior of the ego vehicle. During this process, violation detectors, such as collisions and running red lights, are applied to the fuzzers to detect violations in real time and generate reports. The details of this process are described in Section 4.2.

Finally, the runtime container terminates once the termination conditions are met. Once the container terminates, ADSFuzzEval analyzes and deduplicates the violation reports to generate the empirical research results.

4.1 Initial Driving Scenarios Generation

Initial driving scenarios are crucial for evaluating the performance of fuzz testing techniques. As discussed in Section 2.1, the initial driving scenarios for autonomous vehicles can differ depending on the fuzz testing tool employed. Furthermore, the composition of the initial driving scenarios significantly affects the effectiveness and efficiency of fuzz testing methods.

Although various scenario specification methods can be used for generating high-quality initial driving scenarios compared to random-based strategies,

they often suffer from compatibility issues with the existing fuzz testing tools. This is because these fuzzers rely on their own ad-hoc data structures or dynamic runtime mechanisms (*e.g.,* the traffic management in TM-Fuzzer) that are conceptually misaligned with the pre-defined nature of formal specifications. This incompatibility makes them challenging to integrate, and forcing adaptation would disrupt the workflow of the fuzzers, invalidating the fairness of the empirical evaluation. Therefore, to ensure a fair and reliable comparison across different tools, it is imperative to establish a consistent set of initial driving scenarios.

We begin by formalizing the driving scenarios as a data model, as illustrated in Figure 3, which covers all components used by existing fuzz testing tools. This model consists of five key components: the configuration of the ego vehicle (ev), NPC vehicles (NV), pedestrians (PE), puddles (PU), and weather conditions (wc). Accordingly, the driving scenario data model S is defined as a tuple

$$S = (ev, NV, PE, PU, wc),$$

where the individual components are defined as follows.

- The ego vehicle (ev) is characterized by its vehicle model, start position, and end position: $ev = (ev_{\text{model}}, ev_{\text{start}}, ev_{\text{end}})$.
- Each NPC vehicle ($nv \in NV$) is specified by its vehicle model, start position, end position, and behavior, i.e., $nv = (nv_{\text{model}}, nv_{\text{start}}, nv_{\text{end}}, nv_{\text{beh}})$.
- Each pedestrian ($pe \in PE$) is defined by its pedestrian model, start position, end position, and behavior, i.e., $pe = (pe_{\text{model}}, pe_{\text{start}}, pe_{\text{end}}, pe_{\text{beh}})$.
- Each puddle ($pu \in PU$) is described by its size, location, and friction coefficient, i.e., $pu = (pu_{\text{size}}, pu_{\text{loc}}, pu_{\text{fc}})$.
- The weather conditions (wc) are represented by various parameters, including cloudiness and sun azimuth angle, i.e., $wc = (wc_{\text{cloud}}, \ldots, wc_{\text{azi}})$.

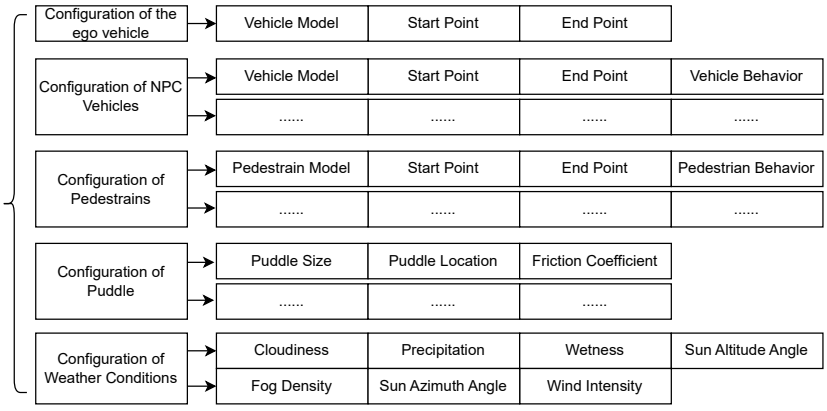


Fig. 3: Driving Scenario Data Model

For each map M provided by the simulator, we generate a set of initial driving scenarios $\mathcal{S}^M = \{S_1, \ldots, S_N\}$, where N is the number of initial driving scenarios to be generated. Each driving scenario $S_i \in \mathcal{S}$ generated following the below five-steps.

(1) Configuration of the Ego Vehicle. As shown in Table 1, random strategies are a straightforward and widely used method for specifying the driving task of the ego vehicle. In our study, we first obtain waypoints at 5-meter intervals along each road of the map M. Two points are then randomly selected as the starting position (ev_{start}) and the ending position (ev_{end}) of the ego vehicle. Furthermore, considering the distance between ev_{start} and ev_{end}, excessively short routes may not trigger significant edge behaviors for the ego vehicle, while excessively long routes may reduce the efficiency and delay the optimization of selection or mutation strategies employed by fuzzers. To address this, we filter the driving task using two thresholds, $task_{\text{min}}$ and $task_{\text{max}}$, which restrict the distance between ev_{start} and ev_{end} to a specific range. This procedure, which is also employed by DriveFuzz and TM-Fuzzer for initial driving scenario generation, uses $task_{\text{min}} = 100$ and $task_{\text{max}} = 200$ in this study. Subsequently, a vehicle model (ev_{model}) is randomly selected from all available models provided by the simulator.

(2) Configuration of NPC Vehicles. The process for generating the configuration of NPC vehicles in the initial driving scenario is outlined in Algorithm 1. This process aims to generate a number of N_{NV} NPC vehicle configurations. As shwon in line 3, we random choice one of three strategies for determining the driving task of the NPC vehicles: *NearBy*, *Intersect*, and *Random*. (1) The *NearBy* strategy, used in DriveFuzz and TM-Fuzzer, constrains the distance between the start points of the ego and NPC vehicles to encourage interaction between them. As shown in lines 6–8 of the algorithm, two distinct waypoints are randomly selected from the map M, and the distance between them is checked to ensure it satisfies the distance constraint. (2) The *Intersect* strategy, employed in Doppel, filters out routes that intersect with the ego vehicle's route, increasing the potential for behavioral interactions. As shown in lines 11–13, we check for intersections by constructing a directed graph to represented the road structure of map M based on the waypoints mentioned above. If there exist intersection, the NPC vehicle's driving task is added to the set NV. (3) The vanilla strategy is *Random*, which generates driving tasks for NPC vehicles without any specific constraints. In this study, we set $N_{\text{NV}} = 1$, $dis_{\text{min}} = 5$, and $dis_{\text{max}} = 40$. Afterwards, a vehicle model (nv_{model}) is randomly selected, and the behavior of NPC vehicle (nv_{beh}) is chosen randomly from the options "Autopilot", "Linear" and "Immobile."

(3) Configuration of Pedestrians. The process for generating pedestrian configurations is similar to that for NPC vehicles; however, the *Cross* strategy is disabled because pedestrians are confined to the sidewalks, while vehicles travel in the center of the road, thus the pedestrian's route does not intersect with the ego vehicle's path. The pedestrian model pe_{model} and behavior pe_{beh} are chosen randomly, similar to the approach used for NPC vehicles.

Algorithm 1: Process of NPC Vehicle Configuration Generation of Initial Driving Scenarios

Input: Road Map M, the number of NPC vehicles N_{NV}, the thresholds $dis_{\min}$ and $dis_{\max}$
Output: The configuration of NPC vehicles NV

$NV \leftarrow \emptyset$
while $|NV| < N_{\text{NV}}$ **do**
  $steg \leftarrow$ `RandomSteg`([“NearBy”, “Intersect”, “Random”])
  if $steg ==$ “NearBy” **then**
    while $True$ **do**
      $nv_{\text{start}}, nv_{\text{end}} \leftarrow$ `RandomWaypoints`$(M, 2)$
      if $dis_{\min} \leq$ `Distance`$(nv_{\text{start}}, ev_{\text{start}}) \leq dis_{\max}$ **then**
        break
  if $steg ==$ “Intersect” **then**
    while $True$ **do**
      $nv_{\text{start}}, nv_{\text{end}} \leftarrow$ `RandomWaypoints`$(M, 2)$
      if `isIntersect`$(nv_{\text{start}}, nv_{\text{end}}, ev_{\text{start}}, ev_{\text{end}})$ **then**
        break
  if $steg ==$ “Random” **then**
    $nv_{\text{start}}, nv_{\text{end}} \leftarrow$ `RandomWaypoints`$(M, 2)$
  $nv_{\text{beh}} \leftarrow$ `RandomBehavior`([“Autopilot”, “Linear”, “Immobile”])
  $nv_{\text{model}} \leftarrow$ `RandomVehicle`()
  $NV \leftarrow NV \cup \{nv\}$
return NV

(4) Configuration of Puddles. The locations of puddles are randomly selected from a set of predefined waypoints. Each puddle is assigned a friction coefficient, which is uniformly distributed within the range of $[0, th_{\text{coef}}]$. Additionally, the size of each puddle is randomly determined within the range of $[0, th_{\text{size}}]$ centimeters. In our study, we adopt the values $th_{\text{coef}} = 10$ and $th_{\text{size}} = 400$, which are inherited from the DriveFuzz configuration.

(5) Configuration of Weather Conditions. For each adjustable weather element, a value is randomly selected from a predefined range. For instance, cloudiness, precipitation, wetness, fog density, and wind intensity are chosen randomly within the range of 0 to 100. The sun's altitude angle is selected randomly from 0 to 360, while its azimuth angle is chosen from -90 to 90.

The generated initial driving scenarios are then passed to the fuzzers as the first-generation driving scenarios for execution. Although fuzzers are capable of directly loading and deploying driving scenarios, there exist minor discrepancies between our data model and the driving scenario structure supported by the fuzzers. For instance, Figure 4 illustrates an example of an initial driving scenario file supported by DriveFuzz, which includes the map and the driving task of the ego vehicle. However, relying solely on this file is insufficient to reproduce a concrete scenario, as it lacks the configuration of traffic participants necessary to simulate a complete traffic flow. To ensure that the fuzzers explore the scenario space from the same starting point, we

adapted the initial driving scenario input component of the fuzzers to align with our data model. The modified versions of the fuzzers are available in our repository, with further details provided in Section 7.

```
"seed":{
    "map":"Town01",
    "sp_x":195.00482177734375,
    "sp_y":195.27003479003906,
    "sp_z":0.29999998211860657,
    "pitch":0.0,
    "yaw":179.999755859375,
    "roll":0.0,
    "wp_x":210.86700439453125,
    "wp_y":199.44183349609375,
    "wp_z":0.29999998211860657,
    "wp_yaw":6.103515261202119e-05
}
```

Fig. 4: Example of DriveFuzz Initial Driving Scenario File

4.2 Violation Detectors

Detecting violations is the core objective of ADS fuzz testing. As shown in Table 2, we summarize the violation categories supported by existing fuzz testing methods and group them into three primary types, including six specific violation detectors. Although fuzzers can detect the same type of violations, the criteria or oracles for judgment differ. To address this issue, we refer to the implementation of existing fuzzers and design six violation detectors. They are implemented as individual components and embedded into each fuzzer without modifying the original code. When the fuzzer terminates, the violation reports R are generated and stored in the output file, each $r \in R$ contains the timestamp, coordinates of the ego vehicle, and additional information.

Collision. As mentioned in Section 2.2, collisions can conveniently be detected by CARLA's built-in sensors. Specifically, ADSFuzzEval deploys the `"sensor.other.collision"` sensor on the ego vehicle. When the vehicle collides with objects, ADSFuzzEval receives a collision event and records its timestamp, the collision coordinates and the collided object.

Speeding. Speeding is defined as the ego vehicle exceeding the speed limit l km/h on a road and failing to adjust its speed p to below l within T seconds, as shown in the following equation:

$$\text{Speeding}(T, \delta) = \mathbb{I}_{\{p(t_0)>l \ \wedge \ p(t_0+T)>l+\delta\}} \tag{1}$$

where $\mathbb{I}(\cdot)$ is an indicator function that returns true if the condition is met, and false otherwise. DriveFuzz and ScenarioFuzz use parameters ($T = 3$, $\delta = 0$), while TM-Fuzzer considers a more lenient criterion ($T = 3$, $\delta = 2$). In our

designed speeding detector, it is configured for strict checking with ($T = 3$, $\delta = 0$). Upon detecting a violation, it records the timestamp, current coordinates, vehicle speed, and the corresponding speed limit.

Lane Invasion. Similar to the collision detection, ADSFuzzEval deploys the CARLA built-in sensor `"sensor.other.lane_invasion"` to the ego vehicle. When the vehicle crosses lane markings that do not allow turning, ADSFuzzEval receives an event and records its timestamp, the coordinates, and the type of the lane.

Running Red Lights. The detection of a red light violation is typically based on two types of oracles. Given that the traffic light is red, a vehicle is considered to have run the red light if either of the following conditions is met: (1) All of the vehicle's speed measurements (p_i) remain greater than a small constant r while it is in the traffic light zone, or (2) The rear end of the vehicle has crossed the stop line. The former oracle is implemented in DriveFuzz, TM-Fuzzer, and ScenarioFuzz, while the latter was designed by the CARLA Leaderboard and is enabled in AutoFuzz. ADSFuzzEval enables both of these oracles, and a violation is flagged if either condition is met, as represented by the equation:

$$\text{RunRedLight}(r) = \mathbb{I}_{\{\forall p_i \in Z, p_i \geq r\}} \vee \mathbb{I}_{\{\text{check}(e_{\text{rear}}, \text{stop\_line})\}} \tag{2}$$

where Z denotes the vehicle speeds in the traffic light zone, e_{rear} and *stop_line* are the coordinates of the rear end of the ego vehicle and the stop line, $check(\cdot)$ is a function that determines whether e_{rear} has crossed the stop line. In our implementation, the parameter is set to $r = 0.1$ km/h. Upon detecting a violation, it records the timestamp, coordinates of the ego vehicle, and the index of the traffic light.

Fail to Reach the Destination. This violation occurs when the ADS route planner has no further routes available, and the ego vehicle is still more than a specified distance D from its destination, as shown in the following equation:

$$\text{FailToReach}(D) = \mathbb{I}_{\{\text{planner}=\emptyset \wedge dist(e_{\text{loc}}, \text{dest}) > D\}} \tag{3}$$

where $dist(\cdot)$ is a function that calculates Euclidean distance. In our detector, the parameter is set to $D = 10$ meters. Upon detecting a violation, ADSFuzzEval records the timestamp and the coordinates of the ego vehicle.

Stuck. This is defined as the speed of the ego vehicle being less than p_{stuck} km/h for a period of T seconds. This condition can be formulated as:

$$\text{Stuck}(p_{\text{stuck}}, T) = \mathbb{I}_{\{\forall t_i \in [t_0 - T, t_0],\ p(t_i) < p_{\text{stuck}}\}} \tag{4}$$

In our implementation, the parameters are set as $p_{\text{stuck}} = 1$, $T = 60$. If this condition is met, ADSFuzzEval records the event as a stuck violation, along with the timestamp and the vehicle's coordinates.

5 Evaluation

We conduct a comprehensive evaluation of five well-know fuzzers (*i.e.,* AV-Fuzzer, DriveFuzz, SAMOTA, TM-Fuzzer and ScenarioFuzz) and three widely-adopted autonomous driving systems (*i.e.,* InterFuser, LMDrive and Autoware). For fair comparisons, we generate $2 \times 2 = 4$ distinct initial driving scenario datasets for two maps provided by CARLA (*i.e.,* Town01 and Town05), each with 100 driving scenarios. The testing time for each fuzzer is set to 10 hours, and the maximum simulation time for individual driving scenarios is set to 10 minutes.

The evaluation was conducted for approximately 500 hours on two servers. The first server, equipped with an Intel i7-12700 CPU, 64GB of RAM, and an NVIDIA RTX 4080 GPU, was used for the InterFuser and Autoware experiments. The second server, featuring an Intel i9-14900K CPU, 64GB of RAM, and an NVIDIA RTX 4090 GPU, ran the LMDrive experiments.

For InterFuser and LMDrive, experiments were performed across all four datasets with all six violation detectors (described in §4.2) enabled. For Autoware, however, the evaluation scale was limited due to two known issues. First, the OpenStreetMap file for Town05 has an improperly configured road network, which can prevent Autoware from changing lanes or finding a valid route (Autoware, 2024). Second, its traffic light module has compatibility issues with CARLA, leading to instability when testing for red light violations (Bridge, 2025). Therefore, Autoware was tested on the single map of Town01, with the red light violation detector disabled.

We aim to address the following research questions:

RQ1 Do different initial driving scenarios result in significant differences in detecting violations?

RQ2 How do different fuzzers perform across various ADS in terms of their violation detection capabilities based on three evaluation metrics?

In RQ1, we investigates whether the initial driving scenarios have an impact on the performance of fuzzers for detecting violations, which is a widely established conclusion in structured software fuzz testing (Cheng et al., 2019, Herrera et al., 2021, Xu et al., 2024). In RQ2, we are interested in comparing the performance of the five fuzzers across three metrics.

5.1 Evaluation Metrics

We select the number of unique violations, map waypoint coverage and code coverage as the evaluation metrics to measure performance.

Number of Unique Violations (UVs). It refers to the number of violations reported by detectors on ADS after deduplication. Specifically, as described in Section 4.2, ADSFuzzEval will output violation reports R after the fuzzer terminates. Then, for each report $r_1 \in R$, if there exists another report $r_2 \in R$ such that the violation type of r_1 is identical to that of r_2, and the Euclidean

distance between the coordinates of the events in r_1 and r_2 is less than a threshold th_d, and the time difference between their timestamps is less than a threshold th_t, then r_1 is considered a duplicate of r_2. Otherwise, r_1 is classified as an unique violation. In our evaluation, we set $th_d = 30$ meters and $th_t = 10$ seconds to account for spatial and temporal proximity that may indicate duplicate violations.

Map Waypoint Coverage. Prior research in software testing has established that improving output diversity can enhance a fuzzer's fault-revealing ability (Akbarova et al., 2025). In the context of ADS testing, the diversity of output can be represented by the variety of road networks and map areas the ego vehicle navigates. Therefore, we introduce map waypoint coverage as a metric to quantify the diversity of scenarios generated by the fuzzer. Specifically, this metric quantifies the extent to which the ego vehicle has traversed the map by measuring the ratio of visited waypoints to the total number of waypoints in the map. We discretize the map by placing a waypoint every 5 meters along the roads, thereby obtaining the set of map waypoints M. For each coordinate of the ego vehicle trajectory, we map it to the nearest waypoint in M, and mark that waypoint as visited. The map waypoint coverage is then calculated as the ratio of the number of visited waypoints to the total number of waypoints in M.

Code Coverage. This metric measures the extent to which the ADS source code is executed during testing. It is calculated as the ratio of executed code lines to the total number of executable lines in the system. We implement this metric using `coveragepy`,[17] a widely-used open-source tool for Python code coverage. For C/C++ files, we use `GCOV`, a tool from the GNU Compiler Collection toolchain designed to instrument source files and collect coverage data. Our analysis targets the Python files in InterFuser and LMDrive, and the C/C++ files in Autoware's implementation. A higher code coverage indicates a more thorough exploration of the ADS's functionality, thereby enhancing confidence in the testing process.

5.2 Statistics of Initial Driving Scenario Datasets

For each selected map in $M = \{\text{Town01}, \text{Town05}\}$, we generate two datasets, *i.e.,* Index1 and Index2. Each dataset contains 100 driving scenarios created using our proposed scenario generation strategies with different random seeds. To validate the diversity and representativeness of these four datasets, we conduct a statistical analysis of the distribution of fundamental properties, including driving routes, weather conditions and initial traffic density.

Scenario and Route Properties. Table 4 presents the core statistics of the generated scenarios, including the driving distances of the ego vehicle, the map waypoint coverage of the ego vehicle's route, and the number of route crossings between the ego vehicle and NPC vehicles. The ego vehicle driving

[17] coveragepy, `https://github.com/nedbat/coveragepy`

distances vary significantly within and across datasets, ranging from 85.00 m to 1,437.25 m, which ensures a mix of both short and long-duration tests. The routes achieve considerable map coverage, up to 68.32% in Town01 and 49.19% in Town05. While the overall road network coverage is not exhaustive, a deeper analysis of critical road networks reveal that all 12 T-junctions in Town01, as well as all 9 T-junctions and 13 cross-junctions in Town05, are included in the route plans of each dataset. This demonstrates that the scenarios are representative as they cover all key road networks (*i.e.,* junctions) within the maps. Furthermore, the number of route intersections between the ego and NPC vehicles, which indicates the potential for interactions, is substantial, with an average of 54 in Town01 and 43.5 in Town05.

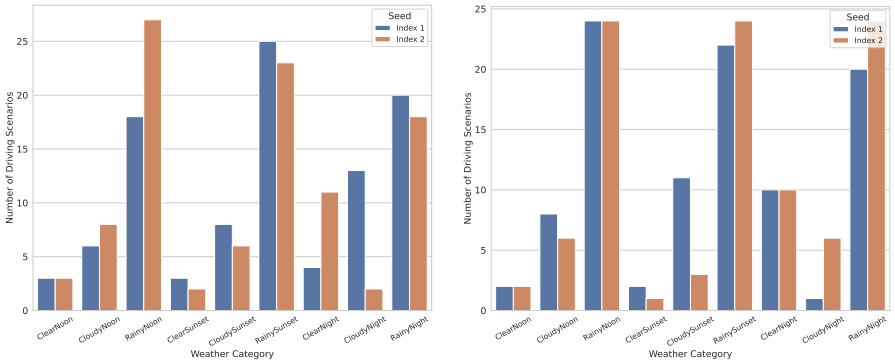
Table 4: Statistical Summary of Initial Driving Scenario Datasets

Initial Driving Scenario Dataset		Driving Distances			Map Waypoint Coverage	Num of Route Intersection
		min	max	avg		
Town01	Index 1	85.00	1404.78	515.45	66.73%	42
	Index 2	96.66	1437.25	539.76	68.32%	66
Town05	Index 1	122.29	1365.50	404.85	46.11%	52
	Index 2	128.88	1249.91	361.00	49.19%	35

Weather Conditions. Figure 5 illustrates the distribution of weather conditions across the scenario datasets. The weather presets from CARLA[18] are grouped into three timezones (Noon, Sunset, Night) and three conditions (Clear, Cloudy, Rainy). The results show a relatively even distribution across the three timezones, ensuring that the system is tested at different times of the day. Notably, rainy conditions occur more frequently than clear or cloudy ones in the generated datasets. Since adverse weather reduces visibility and poses greater challenges to the perception and planning modules, this resulting distribution facilitates a more rigorous evaluation of an ADS' robustness.

Initial Traffic Density. To characterize the traffic environment, we analyze the initial spawning distances between the ego and NPC vehicles. As detailed in Table 5, the distances span a wide range from 5.00m to 497.20m. This range ensures the inclusion of diverse driving contexts, from immediate close-quarters interactions (*e.g.,* car-following) to scenarios involving distant oncoming vehicles. The overall average inter-vehicle distances (121.05m in Town01 and 98.04m in Town05) fall well within the typical less than 200m perception range of on-board sensors (Dai et al., 2022), ensuring that surrounding vehicles are consistently detectable. Furthermore, we quantify the frequency of close-proximity interactions by counting the instances where an NPC spawned within 40m of the ego vehicle. This resulted in a total of 49 to 54 instances in the Town01 datasets and 54 to 57 instances in the Town05 datasets. This high frequency of close-proximity encounters is crucial for establishing a challenging test scenario, enabling a more thorough and realistic evaluation of the ADS.

[18] Weather Presets, `https://carla.readthedocs.io/en/stable/carla_settings/`



(a) Town01 (b) Town05

Fig. 5: Distribution of Weather Conditions in Initial Driving Scenario Datasets

Table 5: Statistical Summary of Initial Spawning Distances Between Ego and NPC Vehicles

Initial Driving Scenario Dataset		**Inter-Vehicle Distance (m)**			**Count of NPCs (< 40m)**
		Min	**Max**	**Avg**	
Town01	Index 1	5.84	410.72	123.73	49
	Index 2	5.00	497.20	118.36	54
Town05	Index 1	5.00	354.75	101.93	54
	Index 2	5.00	365.78	94.15	57

5.3 RQ1

Tables 6, 7, and 8 present the number of unique violations detected by each fuzzer after 10 hours of testing on various initial driving scenarios, using InterFuser, LMDrive, and Autoware as the respective ADS under test. Boldface values indicate the highest number of violations for a given dataset. In this RQ, we aim to investigate whether and how a fuzzer's detection capability is influenced by the initial driving scenario. We analyze this performance variability from two perspectives: first, by comparing a single fuzzer's performance across datasets on different simulation maps, and second, by evaluating the performance on the same map when initiated with different datasets.

(1) Performance Variability Between Different Simulation Maps

Our first finding is that a fuzzer's effectiveness can change dramatically when tested in different simulation maps.

When testing InterFuser (Table 6), this variability is particularly pronounced. For instance, TM-Fuzzer's performance shows a strong dependency on the map: its average detected unique violations in Town05 (116) is 115% higher than in Town01 (54). A similar trend is observed for AV-Fuzzer, whose performance increases from an average of 34 UVs in Town01 to 63 in Town05. Conversely, ScenarioFuzz's effectiveness slightly decreases from Town01 (81)

Table 6: Number of Unique Violations per Fuzzer Across Initial Driving Scenarios with InterFuser as ADS Under Test

Initial Driving Scenario Dataset		AV-Fuzzer	DriveFuzz	SAMOTA	TM-Fuzzer	ScenarioFuzz
Town01	Index 1	36	69	26	54	**84**
	Index 2	32	57	28	54	**78**
	Average UVs	34	63	27	54	**81**
Town05	Index 1	62	62	25	**113**	70
	Index 2	64	64	28	**119**	61
	Average UVs	63	63	26.5	**116**	65.5
Overall Average UVs		48.5	63	26.75	**85**	73.25

Table 7: Number of Unique Violations per Fuzzer Across Initial Driving Scenarios with LMDrive as ADS Under Test

Initial Driving Scenario Dataset		AV-Fuzzer	DriveFuzz	SAMOTA	TM-Fuzzer	ScenarioFuzz
Town01	Index 1	54	48	46	40	**59**
	Index 2	50	57	32	52	**63**
	Average UVs	52	52.5	39	46	**61**
Town05	Index 1	44	56	10	**67**	61
	Index 2	49	59	50	57	**68**
	Average UVs	46.5	57.5	30	62	**64.5**
Overall Average UVs		49.25	55	34.5	54	**62.75**

Table 8: Number of Unique Violations per Fuzzer Across Initial Driving Scenarios with Autoware as ADS Under Test

Initial Driving Scenario Dataset		AV-Fuzzer	DriveFuzz	SAMOTA	TM-Fuzzer	ScenarioFuzz
Town01	Index 1	18	20	20	**36**	35
	Index 2	23	26	15	30	**37**
Average UVs		20.5	23	17.5	33	**36**

to Town05 (65.5). In stark contrast, some fuzzers like DriveFuzz exhibit stability, maintaining an average of 63 UVs across both towns. This demonstrates that different fuzzers possess varying degrees of sensitivity to the initial scenario. Such performance fluctuations may even lead to inconsistent conclusions in the literature. For example, while our results in Town05 show TM-Fuzzer outperforming DriveFuzz, aligning with its reported dominance, the results in Town01 contradict the claim in the TM-Fuzzer paper (Lin et al., 2024). This underscores how conclusions drawn from a single simulation map can be misleading.

This phenomenon is not confined to a single ADS. The results with LMDrive (Table 7) reinforce the trend: TM-Fuzzer's violation detection increases from an average of 46 UVs in Town01 to 62 in Town05, while SAMOTA exhibit the opposite behavior, declining from 39 UVs to 30.

(2) Performance Variability Within the Same Simulation Map

Furthermore, the performance variability is not limited to comparisons between different simulation maps; it also manifests across different initial datasets within the same map.

The most striking example is SAMOTA's performance when testing LM-Drive in Town05 (Table 7). It detected only 10 UVs in the first dataset but surged to 50 UVs in the second, based solely on the initial scenarios. The sensitivity to initial conditions is consistent with SAMOTA's design. The fuzzer utilizes a search algorithm, combining global and local exploration, to identify optimal solutions that violate safety requirements. Such search-based strategies are often highly dependent on their starting point.

This within-map instability impact not only the absolute metric values but also their relative standing, at times even inverting the performance rankings. An inversion can be seen when testing InterFuser in Town05 (Table 6), where the performance ranking between ScenarioFuzz and DriveFuzz flips. In dataset Index1, ScenarioFuzz (70 UVs) outperforms DriveFuzz (62 UVs). However, in dataset Index2, DriveFuzz detected 64 UVs, while the performance of ScenarioFuzz decreased to 61 UVs. A similar inversion occurs in our tests with Autoware in Town01 (Table 8). In this case, the ranking between TM-Fuzzer and ScenarioFuzz reverses. In dataset Index1, TM-Fuzzer was the top performer with 36 UVs, slightly ahead of ScenarioFuzz's 35. However, in dataset Index2 from the same map, their positions switch: ScenarioFuzz leads with 39 UVs, while TM-Fuzzer's count falls to 30. This variability in ScenarioFuzz's performance might stem from its optimization strategy, which leverages graph neural networks and can be sensitive to the initial conditions of scenarios.

In our experimental settings, as all datasets are created using identical generation strategies and the distribution of fundamental properties of these scenarios is consistent (cf. §5.2), we assume that there is no single scenario aspect that universally impacts all fuzzers. Instead, the performance of a fuzzer is influenced by the compatibility of its strategy with the specific driving scenarios it encountered. Based on the observations, we summarize two factors that may introduce this variability:

(1) Map topology and road complexity. The mutation strategies of some fuzzers are more effective on certain map topologies, such as multi-lane roads, which causes their performance to vary across different maps. For example, TM-Fuzzer employs a traffic management system to generate diverse traffic flows. It dynamically searches for valid points near the ego vehicle to spawn NPC vehicles and configure their maneuvers. On a map with multi-lane roads like Town05, TM-Fuzzer can better exploit its strategy compared to a simpler, single-lane map like Town01. This allows TM-Fuzzer to effectively generate complex driving scenarios, such as lane-changing and overtaking maneuvers, which are more likely to trigger violations. Consequently, as shown in Table 6 and 7, TM-Fuzzer detects more unique violations in Town05 than in Town01.
(2) Initial datasets distribution. Fuzzers that rely on search-based strategies or machine learning models are sensitive to the distribution of the initial datasets. Specifically, the effectiveness of SAMOTA's multi-objective search algorithm and ScenarioFuzz's graph neural network is highly dependent on the starting points and features provided by the initial driving scenarios.

This performance variability underscores the necessity of evaluating fuzzers across multiple simulation maps and datasets to obtain an accurate assessment of their capabilities. A fuzzer that excels in one context may not perform as well in another, and vice versa. Therefore, a robust evaluation requires not only multiple simulation maps but also multiple test runs on different datasets within each map to ensure a comprehensive comparison.

Finding:

The initial driving scenarios have a substantial influence on a fuzzer's performance. To mitigate potential evaluation biases, it is recommended to configure initial driving scenarios across different simulation maps and use multiple datasets within each map. A comprehensive assessment should rely on averaged metrics from these tests.

5.4 RQ2

In RQ2, we investigate fuzzer performance through three metrics: (1) number of unique violations, (2) map waypoint coverage and (3) code coverage. Building on RQ1 findings, we sample experimental results at 0.5-hour intervals and average the metrics across different initial driving scenarios.

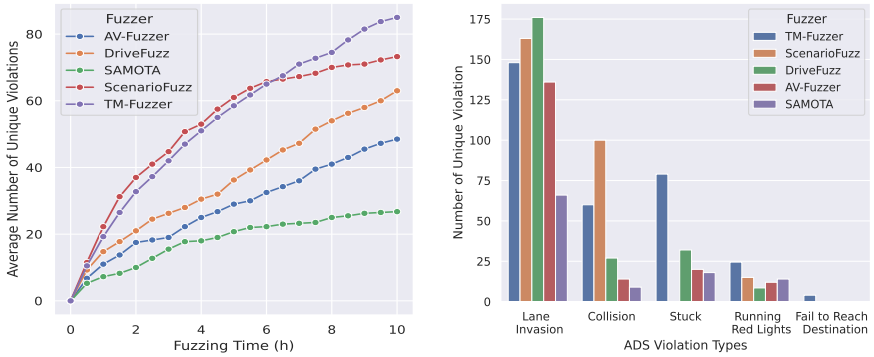
5.4.1 Number of Unique Violations Metric

Figure 6, 7 and 8 illustrates the average number of unique violations detected across fuzzers when testing InterFuser, LMDrive and Autoware, directly reflecting their violation discovery capabilities. Note that none of the fuzzers detect speeding violations, therefore it is omitted in figures.

InterFuser. As shown in Figure 6, TM-Fuzzer achieves superior violation detection throughout the 10-hour fuzz testing against InterFuser. Specifically, with InterFuser as the target ADS at the 5-hour mark, ScenarioFuzz identified 61 UVs, surpassing TM-Fuzzer (58.5), DriveFuzz (36.25), AV-Fuzzer (30) and SAMOTA (20.75). By the 10-hour mark, however, TM-Fuzzer overtook ScenarioFuzz, and this performance ranking stabilized. The final results are documented in the bottom row of Table 6, that is TM-Fuzzer (85), ScenarioFuzz (73.25), DriveFuzz (63), AV-Fuzzer (48.5) and SAMOTA (26.75).

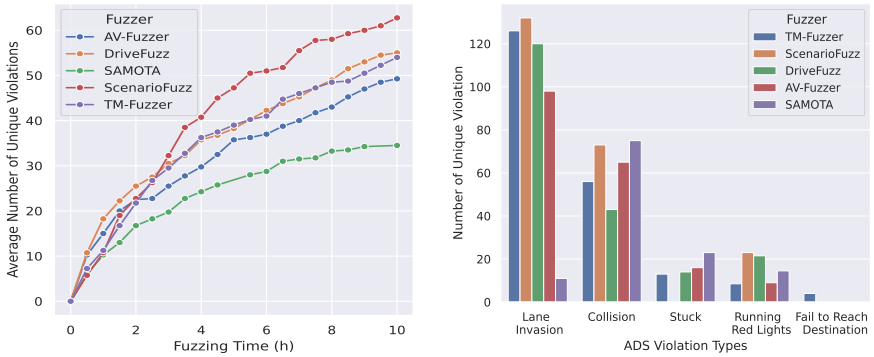
LMDrive. As shown in Figure 7, fuzzers' performance different from InterFuser. Specifically, ScenarioFuzz outperformance TM-Fuzzer and DriveFuzz, detected 47.25 UVs at the first 5-hour interval, followed by TM-Fuzzer (39), DriveFuzz (38.25), AV-Fuzzer (35.75) and SAMOTA (25.75), and the performance ranking was stabled afterwards. The conclusive 10-hour results for this experiment appear in the rightmost column of Table 7, that is ScenarioFuzz (62.75), DriveFuzz (55), TM-Fuzzer (54), AV-Fuzzer (49.25) and SAMOTA (34.5).

Autoware. As shown in Figure 8, the violations discovered by all fuzzers are decreases, because of the performance of Autoware is more stable than



(a) Unique Violations over Time (b) Detected Violation Types

Fig. 6: Fuzzer Performance Comparison based on Unique Violations Metric (InterFuser)

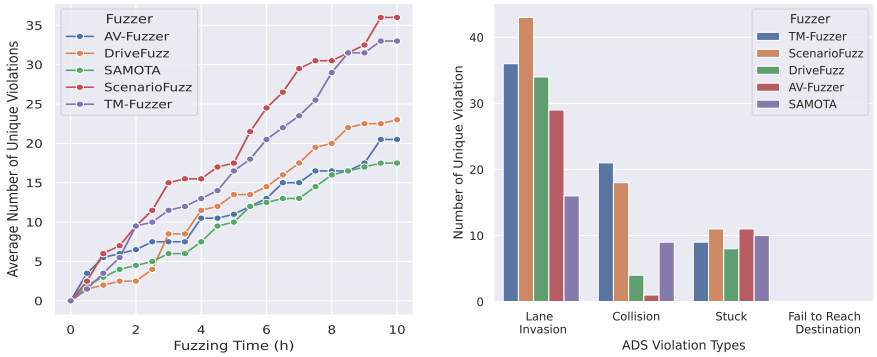


(a) Unique Violations over Time (b) Detected Violation Types

Fig. 7: Fuzzer Performance Comparison based on Unique Violations Metric (LMDrive)

InterFuser and LMDrive. Specifically, ScenarioFuzz detected 36 UVs average two datasets in Town01, and second is TM-Fuzzer (33), following DriveFuzz (23), AV-Fuzzer (20.5) and SAMOTA (17.5).

The distribution of violation types shows that lane invasion is the most prevalent category for all three ADS under test. To understand this further, we performed a detailed location-based analysis of lane invasion events using InterFuser as a representative case. We categorized these events into three road types: straight roads, T-junctions, and cross-junctions. The results, presented in Table 9, indicate that lane invasions occur more frequently at junctions than on straight roads. A typical lane invasion at a junction happens when the ego vehicle turns with an insufficient steering angle, causing it to enter the



(a) Unique Violations over Time

(b) Detected Violation Types

Fig. 8: Fuzzer Performance Comparison based on Unique Violations Metric (Autoware)

oncoming lane or strike the curb. Overall, the more complex cross-junctions, with a greater number of entrances, resulted in more lane invasions than T-junctions. In contrast, lane invasions on straight roads are mainly caused by the ego vehicle's control instability while exiting a turn, as it needs time to stabilize its steering and may consequently cross lane markings.

Table 9: Location Statistics for Lane Invasion Violations

Map	Location	AV-Fuzzer	DriveFuzz	SAMOTA	TM-Fuzzer	ScenarioFuzz	Total
Town01	Stright	22	20	13	15	34	104
	T-junction	23	55	15	2	45	140
	Cross-junction	0	0	0	0	0	0
Town05	Stright	26	25	4	11	37	103
	T-junction	8	30	18	17	14	87
	Cross-junction	57	46	16	103	33	255
Overall		136	176	66	148	163	689

DriveFuzz detected the most lane invasion violations compared to the other fuzzers. We attribute this to its scenario selection strategy. Although DriveFuzz evaluates the potential risk of a scenario based on fitness function, it bypasses this risk assessment to directly select a scenario for mutation if a violation like a lane invasion is detected. While this approach is effective at rapidly triggering violations, it allocates excessive resources to repeatedly finding such similar events, hindering the exploration of more severe violations in other scenarios. Furthermore, because DriveFuzz only mutates the routes and behaviors of NPC vehicles and the weather conditions, without altering the ego vehicle's route, it may struggle to deeply mine other serious violations from the already-identified violation scenarios.

To illustrate this limitation, we quantified the co-occurrence of collisions within the lane invasion scenarios. Although DriveFuzz identified 176 lane invasions, only 10 of these scenarios escalated into a collision. In stark contrast, ScenarioFuzz and TM-Fuzzer triggered collisions in 29 of 163 and 148 lane invasion scenarios, respectively. This highlights the strategies advantages of the other fuzzers. Specifically, ScenarioFuzz achieves a better balance by mutating not only NPC behavior and weather but also the ego vehicle's route, using a graph neural network and sampling to ensure sufficient scenario variation. TM-Fuzzer, on the other hand, employs the NSGA-II genetic algorithm for a multi-objective search, optimizing for factors including the number of violations, minimal distance between vehicles, and the cluster distance to effectively uncover more severe situations.

The above analysis indicates that while DriveFuzz excels at discovering lane invasion violations, it lacks the capability for in-depth exploration of violation scenarios. Future work could focus on optimizing scenario selection and mutation strategies to more effectively uncover severe risks within already-identified violation scenarios.

Moreover, our analysis identified multiple violations that were not caused by the ADS-controlled ego vehicle but by other traffic participants. As illustrated in Figure 9, a collision occurred when the LMDrive-controlled ego vehicle legally traversed an intersection while a red car violated traffic signals by crossing from left to right. Although categorized as a collision violation, culpability analysis reveals that the red sedan's traffic rule violation was the root cause, rendering this event a false-positive.

(a) Awaiting Green Light (b) Entering Intersection (c) Collision Occurrence

Fig. 9: Collision Case Study: Non-Culpable Ego Vehicle in Right-Turn Scenario

Such false-positive events happen when fuzzers create extreme scenarios without considering the dynamic characteristics of the ego vehicle, such as its speed or orientation. These scenarios can leave the ego vehicle with little chance to adjust its control, leading to unmanageable situations and accidents for which it is not responsible (Li et al., 2025). To mitigate this, Huai *et al.* (Huai et al., 2023b) proposed a validated framework that deploys ADS on all vehicles to ensure collisions are exclusively triggered by ADS-controlled agents. However, in our empirical studies, we found it non-trivial to modify each fuzzer's implementation to apply this framework.

To quantify the prevalence of false positives from existing oracles and detectors, we randomly sampled 100 violation scenarios detected by TM-Fuzzer in InterFuser. Then, we manually checked whether the violation was caused by the ego vehicle. The results, shown in Table 10, reveal an overall false positive rate of 13%. Collision violations had the highest false positive rate at 35.29%, particularly in scenarios involving rear-end and lane-change collisions. We also found one false positive among the stuck violations. In contrast, violations of running red lights and failing to reach the destination were all true positives. This was expected, as the two running red light oracles we enabled use environmental context (such as traffic light status, vehicle speed, and location) to assist in detection. Moreover, the fail to reach destination oracle is relatively straightforward. Consequently, their precision is higher than that of the collision and stuck detectors.

Table 10: Manually Annotated Sampling 100 Violation Scenarios

Violation Type	**# Violation Scenario**	**# True Case**	**# False Case**
Lane Invasion	44	44	0
Collision	34	22	12
Stuck	17	16	1
Running Red Lights	4	4	0
Fail to Reach Destination	1	1	0
Total	100	87	13

Figure 10 illustrates a false positive of stuck violation. In this scenario, the ego vehicle is queuing properly, but an NPC vehicle is stopped illegally on its route, causing the ego vehicle to get stuck. This prolonged stop is the fault of neither the ego vehicle nor the ADS under test; rather, it is the result of a traffic rule violation by the NPC vehicle. This external factor is not considered by the current stuck violation detector. Although the ego vehicle is not culpable, the detector still incorrectly flags the incident as a violation, making this report a false positive.

Moreover, although the lane invasions detected by the CARLA built-in sensors are all true actions by the ego vehicle, we find that some of these actions may not be serious enough to trigger traffic accidents. We provide two scenarios in Figure 11. In scenario 1, the white ego vehicle slightly crosses the road line and returns to its lane quickly; the duration of this lane invasion is not long enough to cause an accident or impact other traffic participants. In scenario 2, the white ego vehicle invades the opposite lane and collides with the red NPC vehicle, resulting in a serious traffic accident. However, the current lane invasion detector cannot distinguish between these two scenarios and will report both as violations. This not only increases the manual validation workload but also fails to prioritize genuinely dangerous events like the one in Scenario 2.

Fig. 10: Stuck Case Study: Non-Culpable Ego Vehicle in a Queuing Scenario

(a) Scenario 1: the white ego vehicle slightly cross the road line and back to its road quickly.

(b) Scenario 2: the white ego vehicle invasion opposite road and collided with the red NPC vehicle.

Fig. 11: Comparison of Two Lane Invasion Violations

To address this challenge, we improve the collision, lane invasion and stuck violation detectors by incorporating event and environmental context information, including the area of impact, collision speed, duration of lane invasion and the action of nearby vehicles. This enhancement aims to reduce false positives by attributing responsibility more accurately. Specifically, we design three heuristic filtering strategies to reduce false positives, as follows:

For **Collision**, after a collision event occurs, we check the area of the impact on the ego vehicle. If the impact is on the front, we consider the ego vehicle to be responsible. If the impact is on the rear, the ego vehicle is not considered at fault. If the impact is on the left or right side, we then check the ego vehicle's speed. If its speed was zero, we assume it braked properly to avoid the collision, and thus it is not at fault. Otherwise, the ADS may have

failed to recognize the risk and respond correctly, so it is held responsible for the collision.

For **Lane Invasion**, after a lane invasion event occurs, we start a timer to record its duration. If the ego vehicle returns to its proper lane within five seconds, we consider the invasion minor and do not report it as a violation. Otherwise, the event is classified as a prolonged invasion and is marked as a violation.

For **Stuck**, we check for other traffic participants within a 10-meter radius in the semi-circle in front of the ego vehicle. We consider the ego vehicle to be justifiably stopped (*i.e.,* not a violation) if we find other participants in this area that are also moving at speeds below p_{stuck} (cf. Equation (4)). In all other cases, the event is recorded as a stuck violation.

To evaluate the effectiveness of our improved detectors, we conducted a new 10-hour experiment using TM-Fuzzer with InterFuser in Town01. In this new simulation, the baseline detectors reported a total of 36 potential violations: 17 for Collision, 14 for Lane Invasion, and 5 for Stuck. Our evaluation focused on these 36 specific events to measure the reduction in false positives. After applying our filtering strategies, the number of reported violations was reduced to 16. This final count includes 10 for Collision, 2 for Lane Invasion, and 4 for Stuck. The results are summarized in Table 11.

Table 11: Comparison of False Positives between Baseline and Improved Detectors

Violation Type	**Baseline Detectors**			**Improved Detectors**		
	# Reported	**# False Case**	**FP Rate (%)**	**# Reported**	**# False Case**	**FP Rate (%)**
Collision	17	9	52.94%	10	2	20.00%
Lane Invasion	14	12	85.71%	2	0	0.00%
Stuck	5	1	20.00%	4	0	0.00%
Total	36	22	61.11%	16	2	12.50%

We then manually annotated each of these 36 events to establish a ground truth, identifying which were true violations caused by the ego vehicle versus false positives. This analysis revealed that out of the 36 events reported by the baseline detector, only 14 were true positives, with the remaining 22 being false positives, resulting in a high false-positive rate of 61.11%. In contrast, our improved detector reported 16 violations, of which only 2 were false positives, lowering the false-positive rate to just 12.50%. Overall, our context-aware filters successfully removed 20 non-culpable scenarios, reducing the number of alerts requiring manual inspection by 55.6%.

Specifically, our enhancements completely eliminated false positives for lane invasion and stuck violations. While the collision detector's FP rate was also significantly reduced from 52.94% to 20.00%, two complex false positive scenarios remained. In both cases, a vehicle generated by TM-Fuzzer immediately makes an illegal turn, violating traffic rules. Properly attributing fault in these edge cases requires a deeper analysis of traffic context and participant

behavior, which we leave for future work. (These scenarios are available to review in our supplementary repository.)

Finding:

Number of Unique Violations: The five fuzzers maintain consistent violation detection performance across multiple driving scenarios. However, the scenario selection and mutation strategies of current fuzzers fail to deeply mine already-identified violation scenarios in order to discover potential risks. Additionally, the current violation detectors exhibit false positives, which necessitates more context-aware detectors for a precise evaluation.

5.4.2 Map Waypoint Coverage Metrics

Map waypoint coverage quantifies road network utilization efficiency in driving scenarios generated by fuzzers. Table 12 showcases comparative coverage across fuzzers, where each row represents the average map waypoint coverage that five fuzzers conducted in two initial driving scenario datasets, with InterFuser, LMDrive and Autoware as the ADS under test. Note that map waypoint coverage is calculated as the ratio of the number of waypoints covered by the ego vehicle to the total number of waypoints on the map. This value may be lower than the coverage calculated for the planned route (shown in Table 4) because a fuzzer might terminate scenario execution early. Conversely, it may be higher because a fuzzer can randomly modify the route's start or end points.

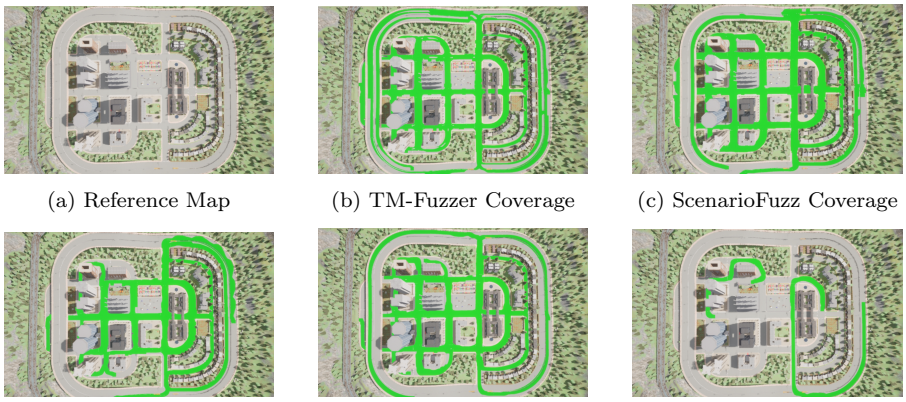
Table 12: Statistical of Map Waypoint Coverage

Map	ADS	AV-Fuzzer	DriveFuzz	SAMOTA	TM-Fuzzer	ScenarioFuzz
Town01	InterFuser	69.05%	73.33%	12.84%	**96.85%**	94.73%
	LMDrive	26.72%	**66.04%**	20.52%	45.23%	62.89%
	Autoware	50.89%	90.94%	69.54%	91.81%	**97.10%**
Town05	InterFuser	21.14%	19.69%	2.94%	**47.39%**	36.35%
	LMDrive	20.28%	14.86%	20.42%	24.08%	**29.03%**

Generally, all fuzzers achieve higher map waypoint coverage in Town01 than in Town05, owing to Town01's simpler road network and smaller map size. Combining map waypoint coverage metric with the number of unique violations, it can be seen that higher map waypoint coverage implies the probability to discover more unique violations, for example, the overall map waypoint coverage for TM-Fuzzer with InterFuser as ADS under test in both Town01 and Town05 is 72.12%, which is the highest coverage compare to others fuzzers, and also, it found the most number of unique violations. This is resonable because higher map waypoint coverage means more road segments are traversed,

which increases the likelihood of encountering edge cases and complex scenarios that can trigger violations. Meanwhile, the unique mechanisms filter out similar violations, and encourage fuzzers to discovery diversity of violations, make the positive feedback loop between map waypoint coverage and unique violations.

To visually demonstrate coverage differences, we collected and visualized ego vehicle trajectories in Town05 during InterFuser testing, as shwon in Figure 12. TM-Fuzzer covered more highway segments than DriveFuzz and AV-Fuzzer, which enabled triggering more lane-changing behaviors and high-speed scenarios. These edge-case scenarios provide more comprehensive testing of ADS capabilities, indicating that higher map coverage not only reflects testing thoroughness but also increases violation exposure likelihood.



(a) Reference Map (b) TM-Fuzzer Coverage (c) ScenarioFuzz Coverage

(d) DriveFuzz Coverage (e) AV-Fuzzer Coverage (f) SAMOTA Coverage

Fig. 12: Trajectory Visualizations of Different Fuzzers in Town05 (InterFuser as the tested ADS)

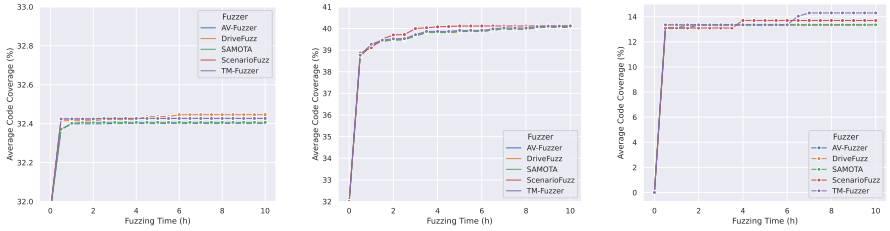
Finding:

Map Waypoint Coverage: The map waypoint coverage metric could reflect the road structure covered by fuzzers during simulation testing, and this metric is a complementary indicator to the number of unique violations to evaluate the performance of fuzz testing methods.

5.4.3 Code Coverage Metrics

Code coverage serves as the third evaluation metric, as shown in Figure 13. The figure plots code coverage percentage over fuzzing time. Each line represents the performance of a single fuzzer, averaged across multiple independent runs. Specifically, the results for InterFuser and LMDrive are averaged across four

runs (2 Towns × 2 seeds), while the results for Autoware are averaged across two runs (1 Town × 2 seeds).



(a) InterFuser Average (b) LMDrive Average (c) Autoware Average

Fig. 13: Average Code Coverage in InterFuser, LMDrive, and Autoware

Based on their architecture, ADS can be categorized into end-to-end systems (including InterFuser and LMDrive) and multi-module systems (Autoware). For end-to-end ADS, the code coverage variance among the fuzzers is negligible. As shown in Figures 13(a) and 13(b), there is minimal inter-fuzzer variance in code coverage across ADS components. For InterFuser, DriveFuzz achieved the highest coverage (32.45%) while AV-Fuzzer obtained the lowest (32.40%), a margin of only 0.05%. For LMDrive, ScenarioFuzz and TM-Fuzzer achieved the highest coverage (40.12%) while others obtained 40.07%, also a margin of only 0.05%. For both end-to-end ADS, the result is a coverage difference of less than 10 lines.

We analyze the code coverage discrepancy between DriveFuzz and AV-Fuzzer in ADS implementations. Figure 14 presents an illustrative case study using the code segment from the `tick()` function in `interfuser_agent.py`, a function handling simulator data processing. Notably, lines 329–333 implement null safety checks for compass data by converting empty values to float numbers, designed to enhance system robustness. Though DriveFuzz successfully triggers this branch (covering line 333), the null compass scenario stems from a known CARLA simulator bug,[19] and executing this branch neither induces ADS failures nor enriches behavioral diversity.

Given that the coverage of end-to-end ADS mainly consists of basic simulation loop logic, for example, in the `run_step()` function, sensor data is gathered and sent to trained deep learning models for decision making, and the code coverage of such fixed execution paths is similar across fuzzers, the code coverage metric has limited discriminative power for evaluating end-to-end ADS testing performance. This finding aligns with previous studies (Ma et al., 2018, Riccio et al., 2020), which also found that code coverage is not a reliable indicator of testing effectiveness in AI-driven systems.

For multi-module ADS, as shown in Figure 13(c), TM-Fuzzer achieves the highest average code coverage (14.30%), followed by ScenarioFuzz with

[19] Issue No.94, `https://github.com/opendilab/LMDrive/issues/94`

```
327  gps = input_data["gps"][1][:2]
328  speed = input_data["speed"][1]["speed"]
329  compass = input_data["imu"][1][-1]
330  if (
331      math.isnan(compass) == True
332  ):  # It can happen that the compass sends nan for a few frames
333      compass = 0.0
```

Fig. 14: Code Coverage in `interfuser_agent.py`

13.70%, while the other three fuzzers obtain the same lowest (13.35%). Note that the number of code lines instrumented in Autoware is approximately 24k, meaning that the coverage difference between TM-Fuzzer and DriveFuzz is over 200 lines. Table 13 illustrates the code coverage of each component in Autoware, where the perception module has the highest coverage (58.53%) across all fuzzers, and the perception and control code coverage is the same across all fuzzers, while the planning and system modules show some variance. Specifically, ScenarioFuzz achieves the highest coverage in the planning module compared to others. It covers more code related to the `StopSpeedExceeded` ROS2 message, which is responsible for handling situations where the vehicle is overspeeding. The variance between TM-Fuzzer and others is mainly focused on system resource management, including `process_monitor` and `gpu_monitor`, which are responsible for monitoring system health and issuing alerts through the diagnostic mechanism when an anomaly occurs.

Table 13: Code Coverage of Autoware Components

Components	AV-Fuzzer	DriveFuzz	SAMOTA	TM-Fuzzer	ScenarioFuzz
Preception	58.53%	58.53%	58.53%	58.53%	58.53%
Planning	11.12%	11.12%	11.12%	11.12%	11.16%
Control	25.00%	25.00%	25.00%	25.00%	25.00%
System Management	13.81%	13.81%	13.81%	16.64%	16.40%

However, although the code coverage for multi-module ADS varies in fuzz testing, it is not a good indicator for evaluating fuzzer performance. First, it does not show a clear relevance to the number of violations. For example, the capability of AV-Fuzzer in generating driving scenarios and discovering violations is lower than that of DriveFuzz, but it has similar code coverage. Second, although the authors of ScenarioFuzz claim that it outperforms DriveFuzz in code coverage because of its capability to generate intersections with ego and NPC vehicles, they do not provide detailed component coverage data, so we cannot verify this claim.

The reason for this is that none of the five fuzzers took code coverage as an objective when designing their fuzzing strategies, such as taking code coverage

into account when selecting driving scenarios for prioritized mutation. Therefore, we encourage further work, especially for testing multi-module ADS, to design coverage-guided fuzzing strategies that can improve code coverage, such as prioritizing scenarios that cover more code lines or branches, or to study the relationship between scenario diversity and code coverage in driving systems.

Finding:

Code Coverage: The code coverage metric fails to distinguish performance differences between fuzzers when testing end-to-end ADS. For multi-module ADS testing, current fuzzers do not take improving code coverage into account and lack consideration of the relationship between code coverage and scenario execution when designing fuzzing strategies. Consequently, the code coverage metric does not accurately reflect the fuzzers' violation detection ability.

5.5 Threats to Validity

The threat to internal validity is related to the implementation of ADSFuzzEval. We adopt methodologies from previous empirical studies (Durieux et al., 2020) and fuzz testing approaches (Du et al., 2022, Luo et al., 2023, Zhang et al., 2024), utilizing Docker for consistent management of computational resources. Specifically, we edited Dockerfiles and built Docker images for the study subjects, ensuring a consistent and reproducible experimental environment. Additionally, we observed that the CARLA simulator may become unstable and crash during long-term experiments. To mitigate this issue, we decoupled the simulation execution into a separate component and designed exception handling mechanisms, thereby enhancing the robustness of the fuzzers and enabling reliable evaluation experiments.

Moreover, another internal validity threat is related to non-determinism in simulators, which has recently attracted the attention of researchers. It refers to situations where vehicles exhibit different behavior given the same driving scenario input over multiple simulation runs (Amini et al., 2024), leading to poor reproducibility and debugging difficulties (Afzal et al., 2021). Although ADSFuzzEval enables CARLA's synchronous mode with fixed timestep to minimize randomness, it cannot completely eliminate flakiness. Consequently, the evaluation results may be affected. Future work could focus on implementing mitigation strategies, such as increased test repetition, using the latest simulator releases (Osikowicz et al., 2025), or employing ML classifiers as proposed by Amini *et al.* (Amini et al., 2024) to assess the degree of flakiness.

The threats to external validity is related to the generalizability of the dataset used for evaluation. To mitigate these threats, we employed multiple strategies when constructing the initial driving scenarios. For instance, we randomly selected one of the "NearBy", "Intersect" and "Random" strategies to generate driving tasks for NPC vehicles, thereby creating diverse initial

driving scenarios. Furthermore, to ensure the reproducibility of our evaluation, we provided the scripts used to construct initial driving scenarios.

The threat to construct validity is related to the metrics utilized for evaluation. We employ three metrics to assess the performance of fuzzers: (1) the number of unique violations, which evaluates the fuzzers' capability to detect violations (Huai et al., 2023b); (2) map waypoint coverage, inspired by Hu *et al.* (Hu et al., 2021), which assesses the diversity and comprehensiveness of the driving scenarios generated by the fuzzers; and (3) code coverage, which measures the extent of code coverage in the ADS during the fuzzing process (Wang et al., 2024). These metrics are widely adopted in ADS testing-related research and provide an effective means of evaluating the performance of testing methodologies.

6 Related Work

ADSFuzzEval is a framework designed to evaluate fuzz testing methods for ADS. In this section, we discuss related work focusing on autonomous driving techniques, testing approaches and safety evaluation techniques.

6.1 Architectures for Autonomous Driving Systems

In recent years, a growing number of autonomous driving systems and techniques have been proposed. Based on their architecture, ADS can be categorized into two types (Zhao et al., 2023), *i.e.,* multi-module ADS and end-to-end ADS.

Multi-module autonomous driving systems, represented by Apollo, Autoware, and Pylot (Gog et al., 2021), utilize a layered architecture that decomposes the complex driving task into several independent modules. These modules, including sensing, perception, planning, and control, are optimized independently yet work in coordination (Schwarting et al., 2018). For instance, the sensing module gathers data from the vehicle's surroundings using various sensors such as cameras, LiDAR, and radar. Subsequently, the perception module leverages machine learning models, like convolutional neural networks, to process this sensor data and extract relevant features for object detection and semantic segmentation. The planning module then uses this processed information for global and local route planning, while the control module outputs the vehicle's control signals, such as steering, throttle, and braking commands, to the vehicle's actuators for execution. This modular approach offers flexibility and scalability, enabling greater interpretability and easier updates to individual components. However, the architectural complexity makes it more challenging and effort-intensive to sufficiently validate and test the entire system.

In contrast, end-to-end ADS, such as InterFuser (Shao et al., 2023), PARA-Drive (Weng et al., 2024), TransFuser (Chitta et al., 2022, Prakash et al.,

2021), and LMDrive (Shao et al., 2024), aim to simulate the learning process of human drivers. They achieve this by learning driving strategies directly from raw sensor data, thereby reducing the need for manual component design and explicit rule-setting (Yoshita et al., 2023). For instance, InterFuser (Shao et al., 2023) employs a transformer-based architecture to fuse multi-modal sensor data, such as LiDAR and camera inputs, for end-to-end driving. This approach allows the model to learn complex driving behaviors and handle diverse driving scenarios. LMDrive (Shao et al., 2024) further enhances this capability by incorporating large language models to improve decision-making in complex environments and achieves superior performance across various driving tasks. However, compared to multi-module ADS, the inherent lack of modularity and interpretability in these models poses significant challenges for debugging, validation, and safety assurance.

Given the complexity of the environment in which autonomous driving vehicles operate, the software quality of ADS is critical for the safety of drivers, passengers, and other traffic participants. Therefore, comprehensive testing of ADS is essential (Koroglu and Wotawa, 2023, Lou et al., 2022, Tang et al., 2023).

6.2 ADS Testing Approaches

In ADS fuzz testing, researchers first focused on the DNN-based components of the system (such as the perception module) and have proposed several methods to detect faults in these components. Adversarial sample attacks are effective methods to detect errors in the ADS perception module, as vehicles rely on sensors to perceive the external environment. Adversarial samples, such as perturbed camera images (Tian et al., 2018, Zhang et al., 2018), lidar data (Cao et al., 2019), road signs (Eykholt et al., 2018, Zhao et al., 2019) and billboards (Zhou et al., 2020), can be used to attack the perception module. For example, Tian *et al.* (Tian et al., 2018) proposed DeepTest, which generates test inputs by perturbing images with transformations such as rotation, scaling, and blurring to maximize the number of activated neurons and detect errors in the steering model. Wu *et al.* (Wu et al., 2021) extended deep learning verification with neuron coverage and relaxation relations to verify the safety of specific steering angles. Zhou *et al.* proposed DeepBillboard (Zhou et al., 2020), which perturbs billboard images to maximize the steering angle error in models like DAVE (Bojarski, 2016). Guo *et al.* proposed FuzzScene (Guo et al., 2024), which represents scenarios in OpenSCENARIO and generates driving scenarios with a grammar-aware strategy in CARLA, detecting faults in the steering model based on metamorphic relations (Chen et al., 2018).

While adversarial samples can effectively reduce the security and reliability of ADS, generating such images in real-world settings may be challenging (Lu et al., 2017, Zhong et al., 2021). Software-in-the-loop is a method of testing and validating code in a simulation environment to quickly and cost-effectively detect bugs and improve code quality (Brambilla et al., 2014, Demers et al.,

2007). Simulation-based ADS testing is one of the best practices, leveraging high-fidelity simulation environments to generate diverse scenarios, monitor the state of the ego vehicle and other traffic participants during simulation, and discover violation behaviors or defects in ADS.

Driving scenarios are test cases in simulation-based testing. The quality of these scenarios significantly impacts the effectiveness and efficiency of simulation-based ADS testing (Dai et al., 2024). To address this challenge, researchers have been exploring various techniques for scenario specification and generation, aiming to create high-quality driving scenarios that accurately reflect real-world conditions and potential edge cases.

For scenario specification, the OpenSCENARIO standard is widely used to describe driving scenarios in a standardized format. It provides a comprehensive framework for defining various aspects of driving scenarios, including the behavior of the ego vehicle, other traffic participants, and environmental conditions. This standardization facilitates scenario sharing and reuse across different simulation platforms, enhancing the interoperability of testing tools. Scenic (Fremont et al., 2019) is a domain-specific language design for scenario modeling, allowing users to specify complex driving scenarios using a high-level language. Scenic can automatically generate diverse scenarios by sampling from the specified distributions, making it a useful tool for simulation-based testing. Bakikian *et al.* (Babikian et al., 2024) present an approach for the concretization of abstract traffic scenario specifications by introducing a domain-specific language, which uses 4-valued partial model semantics to model static and dynamic objects. Bach *et al.* (Bach et al., 2016) present a methodology for the abstract positional and temporal description of driving scenarios, utilizing a movie-related, omniscient view composed of sequential acts to support development and testing of automated driving functions.

As for scenario generation, fuzz testing, combined with evolutionary algorithms such as genetic algorithms, has led to the proposal of several scenario generation methods for ADS. In addition to methods such as DriveFuzz, AV-Fuzzer, TM-Fuzzer, ScenarioFuzz and SAMOTA mentioned in Section 3.1, Hu *et al.* (Hu et al., 2021) introduced a coverage-based ADS fuzz testing method inspired by code coverage metrics from structured software testing. This approach divides the map's driving area into multiple blocks, each with a length and width of 1 meter, tracking the blocks traversed by the ego vehicle to form a coverage trajectory metric. Sun *et al.* (Sun et al., 2022) targeted complex traffic rules, introducing LawBreaker to generate scenarios more likely to trigger traffic regulation violations. By formalizing traffic rules (*e.g.,* obeying traffic signals) into signal temporal logic formulas (Maler and Nickovic, 2004), LawBreaker quantifies the "distance" between the ego vehicle's behavior and rule violations, prioritizing scenarios likely to induce violations for scenario evolution. Similarly, Li *et al.* (Li et al., 2024) proposed VioHawk, a framework that designates hazardous and non-hazardous zones based on traffic conditions. For instance, intersections are marked as hazardous during red lights. VioHawk uses mutation operations to drive the ego vehicle toward these zones, aiming to elicit violations.

The purpose of our study is to evaluate the performance of fuzz testing tools in terms of scenario generation and violation detection. To provide a fair comparison, we need to provide consistent initial driving scenarios and violation oracles used by different fuzzers. Although the aforementioned scenario specification methods achieve outperformance in achieve more diversity or complex driving scenarios, and the existing testing works integrates kinds of violation detector, they are not suitable for our evaluation purpose, as (1) the scenario driving models are not used by or compatible with the five state-of-the-art simulation-based fuzz testing tools we investigated. and (2) the violation detectors have multiple and varying implementations. Therefore, we proposed ADSFuzzEval, which integrates multiple strategies to generated unified initial driving scenarios, and implements six violation detectors with consistent parameters, ensuring a fair evaluation framework.

6.3 ADS Safety Evaluation Techniques

With the rapid development of ADS, the performance of ADS systems has become a major concern. SafeBench (Xu et al., 2022) is a benchmarking platform for safety evaluation of autonomous vehicles, considering eight safety-critical testing scenarios and developing four scenario generation algorithms (*e.g.,* learning-to-collide (Ding et al., 2020b), AdvSim (Wang et al., 2021), *etc.*). SafeBench selects and implements four deep reinforcement learning-based autonomous driving algorithms (*e.g.,* proximal policy optimization (Schulman et al., 2017) and soft actor-critic (Haarnoja et al., 1861)) as study subjects for evaluation. It was shown that the performance of ADS drops when tested on SafeBench. Compared to SafeBench, ADSFuzzEval focuses on fuzzing-based scenario generation approaches and includes more advanced ADS than SafeBench, which focused on systems released before 2018 and no longer represent the latest ADS capabilities.

ScenarioNet (Li et al., 2023) is an open-source platform for large-scale traffic scenario simulation, defining a unified scenario description format to reduce limitations on cross-dataset training. It collects a large-scale repository of real-world traffic scenarios from heterogeneous driving datasets, including nuScenes (Caesar et al., 2020) and Waymo (Ettinger et al., 2021), supporting the replay of driving scenarios based on the MetaDrive simulator (Li et al., 2022b). ScenarioNet configures OpenPilot in reconstructed scenarios, showing that OpenPilot is robust enough to operate in common scenarios such as lane-keeping. Similarly, ADSFuzzEval defines a data model to unify driving scenarios across different fuzz testing approaches and generates initial driving scenarios based on several strategies.

The CARLA Leaderboard is a online challenge supported by CARLA and academic conferences (*e.g.,* CVPR), providing an online ADS performance competition across two tracks: SENSORS and MAP. In the SENSORS track, ADS have access to GNSS, IMU, LiDAR, radar, RGB cameras and speed sensors, while the MAP track additionally provides high-definition maps. ADS

are tasked with completing various complex scenarios, including lane changes, intersection negotiations, and yielding to emergency vehicles, based on the NHTSA typology (Najm et al., 2007). Leaderboard scoring encompasses various factors such as rule violations, timeouts and speed maintenance. In contrast, ADSFuzzEval prioritizes safety performance, evaluating whether scenarios generated by fuzzers can effectively trigger safety violations.

7 Conclusion

In this paper, we present, to the best of our knowledge, the largest empirical evaluation of fuzz testing for ADS. The experiments, conducted for about 500 hours, reveal that different datasets of initial driving scenarios can lead to inconsistent conclusions compared to those presented in the original papers. Additionally, we employ three metrics to assess the performance of fuzzers and find that map waypoint coverage is a good indicator to complement the number of unique violations metric for evaluating the performance of fuzz testing methods.

Based on our empirical study, we recommend that future research should: (1) use multiple environments (*e.g.,* maps) and generate multiple datasets with random seeds as initial driving scenarios for a comprehensive evaluation of fuzz testing methods; (2) optimize fuzzing strategies to thoroughly investigate potential risks in already-identified violation scenarios; and (3) incorporate coverage-guided fuzzing strategies into the design of scenario selection and mutation to both generate diverse driving scenarios and explore the state space and behavioral logic of multi-module ADS in depth.

Declarations

Funding

This work was partially supported by the National Natural Science Foundation of China (NSFC, No. 62372232). T. Chen is partially supported by an oversea grant from the State Key Laboratory of Novel Software Technology, Nanjing University (KFKT2023A04).

Author Contributions

Huiwen Yang contributed to the methodology design, software development, and manuscript writing. Yu Zhou was responsible for funding acquisition, supervision, manuscript writing and review. Taolue Chen provided methodological insights, contributed to manuscript writing and review.

Data Availability

The implementation of ADSFuzzEval framework and our modified version of DriveFuzz, TM-Fuzzer, AV-Fuzzer, SAMOTA and ScenarioFuzz are available at `https://github.com/yagol2020/ADSFuzzEval`.

Conflict of Interest

The authors declare that they have no conflict of interest.

Ethical Approval, Informed Consent and Clinical Trial Number

Not applicable.

References

Afzal A, Katz DS, Le Goues C, Timperley CS (2021) Simulation for robotics test automation: Developer perspectives. In: 2021 14th IEEE conference on software testing, verification and validation (ICST), IEEE, pp 263–274

Akbarova S, Dobslaw F, Neto FGdO, Feldt R (2025) Setbve: Quality-diversity driven exploration of software boundary behaviors. arXiv preprint arXiv:250519736

Amini MH, Naseri S, Nejati S (2024) Evaluating the impact of flaky simulators on testing autonomous driving systems. Empirical Software Engineering 29(2):47

Asmita, Oliinyk Y, Scott M, Tsang R, Fang C, Homayoun H (2024) Fuzzing BusyBox: Leveraging LLM and crash reuse for embedded bug unearthing. In: 33rd USENIX Security Symposium (USENIX Security 24), USENIX Association, Philadelphia, PA, pp 883–900, URL `https://www.usenix.org/conference/usenixsecurity24/presentation/asmita`

Autoware (2024) Carla map and autoware compatibility issues. URL `https://github.com/orgs/autowarefoundation/discussions/4577`, accessed 1 Sep 2025

Babikian AA, Semeráth O, Varró D (2024) Concretization of abstract traffic scene specifications using metaheuristic search. IEEE Transactions on Software Engineering 50(1):48–68, DOI 10.1109/TSE.2023.3331254

Bach J, Otten S, Sax E (2016) Model based scenario specification for development and test of automated driving functions. In: 2016 IEEE Intelligent Vehicles Symposium (IV), pp 1149–1155, DOI 10.1109/IVS.2016.7535534

Bojarski M (2016) End to end learning for self-driving cars. arXiv preprint arXiv:160407316

Brambilla G, Grazioli A, Picone M, Zanichelli F, Amoretti M (2014) A cost-effective approach to software-in-the-loop simulation of pervasive systems and applications. In: 2014 IEEE International Conference on Pervasive Computing and Communication Workshops (PERCOM WORKSHOPS), IEEE, pp 207–210

Bridge (2025) Run carla with autoware with traffic light module enabled. URL `https://autoware-carla-launch.readthedocs.io/en/latest/scenarios/autoware_traffic_light.html`, accessed 1 Sep 2025

Caesar H, Bankiti V, Lang AH, Vora S, Liong VE, Xu Q, Krishnan A, Pan Y, Baldan G, Beijbom O (2020) nuscenes: A multimodal dataset for autonomous driving. In: Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, pp 11621–11631

Cao Y, Xiao C, Cyr B, Zhou Y, Park W, Rampazzi S, Chen QA, Fu K, Mao ZM (2019) Adversarial sensor attack on lidar-based perception in autonomous driving. In: Proceedings of the 2019 ACM SIGSAC conference on computer and communications security, pp 2267–2281

Chen D, Krähenbühl P (2022) Learning from all vehicles. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp 17222–17231

Chen TY, Kuo FC, Liu H, Poon PL, Towey D, Tse T, Zhou ZQ (2018) Metamorphic testing: A review of challenges and opportunities. ACM Computing Surveys (CSUR) 51(1):1–27

Cheng L, Zhang Y, Zhang Y, Wu C, Li Z, Fu Y, Li H (2019) Optimizing seed inputs in fuzzing with machine learning. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pp 244–245, DOI 10.1109/ICSE-Companion.2019.00096

Chitta K, Prakash A, Jaeger B, Yu Z, Renz K, Geiger A (2022) Transfuser: Imitation with transformer-based sensor fusion for autonomous driving. IEEE Transactions on Pattern Analysis and Machine Intelligence 45(11):12878–12895

Dai J, Gao B, Luo M, Huang Z, Li Z, Zhang Y, Yang M (2024) Sctrans: Constructing a large public scenario dataset for simulation testing of autonomous driving systems. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA, ICSE '24, DOI 10.1145/3597503.3623350, URL https://doi.org/10.1145/3597503.3623350

Dai Z, Wolf A, Ley PP, Glück T, Sundermeier MC, Lachmayer R (2022) Requirements for automotive lidar systems. Sensors 22(19):7532

Demers S, Gopalakrishnan P, Kant L (2007) A generic solution to software-in-the-loop. In: MILCOM 2007 - IEEE Military Communications Conference, pp 1–6, DOI 10.1109/MILCOM.2007.4455268

Ding W, Chen B, Xu M, Zhao D (2020a) Learning to collide: An adaptive safety-critical scenarios generating method. In: 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp 2243–2250, DOI 10.1109/IROS45743.2020.9340696

Ding W, Chen B, Xu M, Zhao D (2020b) Learning to collide: An adaptive safety-critical scenarios generating method. In: 2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), IEEE, pp 2243–2250

Ding W, Xu C, Arief M, Lin H, Li B, Zhao D (2023) A survey on safety-critical driving scenario generation—a methodological perspective. IEEE Transactions on Intelligent Transportation Systems 24(7):6971–6988, DOI 10.1109/TITS.2023.3259322

Dosovitskiy A, Ros G, Codevilla F, Lopez A, Koltun V (2017) CARLA: An open urban driving simulator. In: Proceedings of the 1st Annual Conference on Robot Learning, pp 1–16

Du Z, Li Y, Liu Y, Mao B (2022) Windranger: A directed greybox fuzzer driven by deviation basic blocks. In: 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE), pp 2440–2451, DOI 10.1145/3510003.3510197

Durieux T, Ferreira JF, Abreu R, Cruz P (2020) Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In: Proceedings of the ACM/IEEE 42nd International conference on software engineering, pp 530–541

Duy Son T, Bhave A, Van der Auweraer H (2019) Simulation-based testing framework for autonomous driving development. In: 2019 IEEE International Conference on Mechatronics (ICM), vol 1, pp 576–583, DOI 10.1109/ICMECH.2019.8722847

Ettinger S, Cheng S, Caine B, Liu C, Zhao H, Pradhan S, Chai Y, Sapp B, Qi CR, Zhou Y, et al. (2021) Large scale interactive motion forecasting for autonomous driving: The waymo open motion dataset. In: Proceedings of the IEEE/CVF International Conference on Computer Vision, pp 9710–9719

Eykholt K, Evtimov I, Fernandes E, Li B, Rahmati A, Xiao C, Prakash A, Kohno T, Song D (2018) Robust physical-world attacks on deep learning visual classification. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp 1625–1634

Fremont DJ, Dreossi T, Ghosh S, Yue X, Sangiovanni-Vincentelli AL, Seshia SA (2019) Scenic: a language for scenario specification and scene generation. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, Association for Computing Machinery, New York, NY, USA, PLDI 2019, p 63–78, DOI 10.1145/3314221.3314633, URL https://doi.org/10.1145/3314221.3314633

Gao C, Wang G, Shi W, Wang Z, Chen Y (2022) Autonomous driving security: State of the art and challenges. IEEE Internet of Things Journal 9(10):7572–7595, DOI 10.1109/JIOT.2021.3130054

Gog I, Kalra S, Schafhalter P, Wright MA, Gonzalez JE, Stoica I (2021) Pylot: A modular platform for exploring latency-accuracy tradeoffs in autonomous vehicles. In: 2021 IEEE International Conference on Robotics and Automation (ICRA), IEEE, pp 8806–8813

Grigorescu S, Trasnea B, Cocias T, Macesanu G (2020) A survey of deep learning techniques for autonomous driving. Journal of field robotics 37(3):362–386

Guo A, Feng Y, Cheng Y, Chen Z (2024) Semantic-guided fuzzing for virtual testing of autonomous driving systems. Journal of Systems and Software 212:112017

Haarnoja T, Zhou A, Abbeel P, Levine S (1861) Soft actor-critic: Off-policy maximum entropy deep reinforcement. In: Proceedings of the 35th International Conference on Machine Learning. July 10th-15th, Stockholm, Sweden, vol 1870

Haq FU, Shin D, Briand L (2022) Efficient online testing for dnn-enabled systems using surrogate-assisted and many-objective optimization. In: Proceedings of the 44th International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA, ICSE '22, p 811–822, DOI 10.1145/3510003.3510188, URL https://doi.org/10.1145/3510003.3510188

Herrera A, Gunadi H, Magrath S, Norrish M, Payer M, Hosking AL (2021) Seed selection for successful fuzzing. In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA, ISSTA 2021, p 230–243, DOI 10.1145/3460319.3464795, URL https://doi.org/10.1145/3460319.3464795

Hu Z, Guo S, Zhong Z, Li K (2021) Coverage-based scene fuzzing for virtual autonomous driving testing. arXiv preprint arXiv:210600873

Huai Y, Almanee S, Chen Y, Wu X, Chen QA, Garcia J (2023a) sceno rita: Generating diverse, fully-mutable, test scenarios for autonomous vehicle planning. IEEE Transactions on Software Engineering

Huai Y, Chen Y, Almanee S, Ngo T, Liao X, Wan Z, Chen QA, Garcia J (2023b) Doppelgänger test generation for revealing bugs in autonomous driving software. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pp 2591–2603, DOI 10.1109/ICSE48619.2023.00216

Kalra N, Paddock SM (2016) Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability? Transportation Research Part A: Policy and Practice 94:182–193

Kim S, Liu M, Rhee JJ, Jeon Y, Kwon Y, Kim CH (2022) Drivefuzz: Discovering autonomous driving bugs through driving quality-guided fuzzing. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, pp 1753–1767

Koroglu Y, Wotawa F (2023) Towards a review on simulated adas/ad testing. In: 2023 IEEE/ACM International Conference on Automation of Software Test (AST), IEEE, pp 112–122

Li A, Chen S, Sun L, Zheng N, Tomizuka M, Zhan W (2022a) Scegene: Bio-inspired traffic scenario generation for autonomous driving testing. IEEE Transactions on Intelligent Transportation Systems 23(9):14859–14874, DOI 10.1109/TITS.2021.3134661

Li C, Sifakis J, Yan R, Zhang J (2025) Testing autonomous driving systems – what really matters and what doesn't. URL https://arxiv.org/abs/2507.13661, 2507.13661

Li G, Li Y, Jha S, Tsai T, Sullivan M, Hari SKS, Kalbarczyk Z, Iyer R (2020) Av-fuzzer: Finding safety violations in autonomous driving systems. In: 2020 IEEE 31st international symposium on software reliability engineering (ISSRE), IEEE, pp 25–36

Li Q, Peng Z, Feng L, Zhang Q, Xue Z, Zhou B (2022b) Metadrive: Composing diverse driving scenarios for generalizable reinforcement learning. IEEE transactions on pattern analysis and machine intelligence 45(3):3461–3475

Li Q, Peng Z, Feng L, Liu Z, Duan C, Mo W, Zhou B (2023) Scenarionet: Open-source platform for large-scale traffic scenario simulation and modeling. Advances in Neural Information Processing Systems

Li Z, Dai J, Huang Z, You N, Zhang Y, Yang M (2024) Viohawk: Detecting traffic violations of autonomous driving systems through criticality-guided simulation testing. In: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, pp 844–855

Liang J, Jiang Y, Wang M, Jiao X, Chen Y, Song H, Choo KKR (2021) Deepfuzzer: Accelerated deep greybox fuzzing. IEEE Transactions on Dependable and Secure Computing 18(6):2675–2688, DOI 10.1109/TDSC.2019.2961339

Lin S, Chen F, Xi L, Wang G, Xi R, Sun Y, Zhu H (2024) Tm-fuzzer: fuzzing autonomous driving systems through traffic management. Automated Software Engineering 31(2):61

Liu L, Lu S, Zhong R, Wu B, Yao Y, Zhang Q, Shi W (2021) Computing systems for autonomous driving: State of the art and challenges. IEEE Internet of Things Journal 8(8):6469–6486, DOI 10.1109/JIOT.2020.3043716

Lou G, Deng Y, Zheng X, Zhang M, Zhang T (2022) Testing of autonomous driving systems: where are we and where should we go? In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2022, p 31–43, DOI 10.1145/3540250.3549111, URL https://doi.org/10.1145/3540250.3549111

Lu J, Sibai H, Fabry E, Forsyth D (2017) No need to worry about adversarial examples in object detection in autonomous vehicles. arXiv preprint arXiv:170703501

Luo Z, Yu J, Zuo F, Liu J, Jiang Y, Chen T, Roychoudhury A, Sun J (2023) Bleem: Packet sequence oriented fuzzing for protocol implementations. In: 32nd USENIX Security Symposium (USENIX Security 23), pp 4481–4498

Ma L, Juefei-Xu F, Zhang F, Sun J, Xue M, Li B, Chen C, Su T, Li L, Liu Y, et al. (2018) Deepgauge: Multi-granularity testing criteria for deep learning systems. In: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering, pp 120–131

Ma X, Song L, Zhao C, Wu S, Yu W, Wang W, Yang L, Wang H (2024) Law compliance decision making for autonomous vehicles on highways. Accident Analysis & Prevention 204:107620

Maler O, Nickovic D (2004) Monitoring temporal properties of continuous signals. In: International symposium on formal techniques in real-time and fault-tolerant systems, Springer, pp 152–166

Najm WG, Smith JD, Yanagisawa M, et al. (2007) Pre-crash scenario typology for crash avoidance research. Tech. rep., United States. Department of Transportation. National Highway Traffic Safety Administration

Osikowicz O, McMinn P, Shin D (2025) Empirically evaluating flaky tests for autonomous driving systems in simulated environments. In: 2025 IEEE/ACM International Flaky Tests Workshop (FTW), IEEE, pp 13–20

Prakash A, Chitta K, Geiger A (2021) Multi-modal fusion transformer for end-to-end autonomous driving. In: Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, pp 7077–7087

Riccio V, Jahangirova G, Stocco A, Humbatova N, Weiss M, Tonella P (2020) Testing machine learning based systems: a systematic mapping. Empirical Software Engineering 25(6):5193–5254

Rong G, Shin BH, Tabatabaee H, Lu Q, Lemke S, Možeiko M, Boise E, Uhm G, Gerow M, Mehta S, et al. (2020) Lgsvl simulator: A high fidelity simulator for autonomous driving. arXiv preprint arXiv:200503778

Schulman J, Wolski F, Dhariwal P, Radford A, Klimov O (2017) Proximal policy optimization algorithms. arXiv preprint arXiv:170706347

Schwarting W, Alonso-Mora J, Rus D (2018) Planning and decision-making for autonomous vehicles. Annual Review of Control, Robotics, and Autonomous Systems 1(1):187–210

Shao H, Wang L, Chen R, Li H, Liu Y (2023) Safety-enhanced autonomous driving using interpretable sensor fusion transformer. In: Conference on Robot Learning, PMLR, pp 726–737

Shao H, Hu Y, Wang L, Song G, Waslander SL, Liu Y, Li H (2024) Lmdrive: Closed-loop end-to-end driving with large language models. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp 15120–15130

Stocco A, Pulfer B, Tonella P (2023) Model vs system level testing of autonomous driving systems: a replication and extension study. Empirical Software Engineering 28(3):73

Sun Y, Poskitt CM, Sun J, Chen Y, Yang Z (2022) Lawbreaker: An approach for specifying traffic laws and fuzzing autonomous vehicles. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, pp 1–12

Tang S, Zhang Z, Zhang Y, Zhou J, Guo Y, Liu S, Guo S, Li YF, Ma L, Xue Y, et al. (2023) A survey on automated driving system testing: Landscapes and trends. ACM Transactions on Software Engineering and Methodology 32(5):1–62

Tian H, Jiang Y, Wu G, Yan J, Wei J, Chen W, Li S, Ye D (2022) Mosat: finding safety violations of autonomous driving systems using multi-objective genetic algorithm. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp 94–106

Tian Y, Pei K, Jana S, Ray B (2018) Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In: Proceedings of the 40th international conference on software engineering, pp 303–314

Wang J, Pun A, Tu J, Manivasagam S, Sadat A, Casas S, Ren M, Urtasun R (2021) Advsim: Generating safety-critical scenarios for self-driving vehicles. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp 9909–9918

Wang S, Sheng Z, Xu J, Chen T, Zhu J, Zhang S, Yao Y, Ma X (2022) ADEPT: A testing platform for simulated autonomous driving. In: 37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022, ACM, pp 150:1–150:4, DOI 10.1145/3551349.3559528, URL https://doi.org/10.1145/3551349.3559528

Wang T, Gu T, Deng H, Li H, Kuang X, Zhao G (2024) 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, Association for Computing Machinery, New York, NY, USA, ISSTA 2024, p 1086–1098, DOI 10.1145/3650212.3680344, URL https://doi.org/10.1145/3650212.3680344

Weng X, Ivanovic B, Wang Y, Wang Y, Pavone M (2024) Para-drive: Parallelized architecture for real-time autonomous driving. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp 15449–15458

Wu H, Lv D, Cui T, Hou G, Watanabe M, Kong W (2021) Sdlv: Verification of steering angle safety for self-driving cars. Formal Aspects of Computing 33:325–341

Wu P, Jia X, Chen L, Yan J, Li H, Qiao Y (2022) Trajectory-guided control prediction for end-to-end autonomous driving: A simple yet strong baseline. Advances in Neural Information Processing Systems 35:6119–6132

Xu C, Ding W, Lyu W, Liu Z, Wang S, He Y, Hu H, Zhao D, Li B (2022) Safebench: A benchmarking platform for safety evaluation of autonomous vehicles. Advances in Neural Information Processing Systems 35:25667–25682

Xu Y, Jia H, Chen L, Wang X, Zeng Z, Wang Y, Gao Q, Wang J, Ye W, Zhang S, et al. (2024) Isc4dgf: Enhancing directed grey-box fuzzing with llm-driven initial seed corpus generation. arXiv preprint arXiv:240914329

Yoshita, Jatain A, Manju, Kumar S (2023) Perception to control: End-to-end autonomous driving systems. In: International Conference on Communication and Intelligent Systems, Springer, pp 447–454

Zhang M, Zhang Y, Zhang L, Liu C, Khurshid S (2018) Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems. In: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering, pp 132–142

Zhang Y, Liu Y, Xu J, Wang Y (2024) Predecessor-aware directed greybox fuzzing. In: 2024 IEEE Symposium on Security and Privacy (SP), pp 1884–1900, DOI 10.1109/SP54263.2024.00040

Zhao J, Zhao W, Deng B, Wang Z, Zhang F, Zheng W, Cao W, Nan J, Lian Y, Burke AF (2023) Autonomous driving system: A comprehensive survey. Expert Systems with Applications p 122836

Zhao Y, Zhu H, Liang R, Shen Q, Zhang S, Chen K (2019) Seeing isn't believing: Towards more robust adversarial attack against real world object detectors. In: Proceedings of the 2019 ACM SIGSAC conference on computer and communications security, pp 1989–2004

Zhong Z, Tang Y, Zhou Y, Neves VdO, Liu Y, Ray B (2021) A survey on scenario-based testing for automated driving systems in high-fidelity simulation. arXiv preprint arXiv:211200964

Zhong Z, Kaiser G, Ray B (2022) Neural network guided evolutionary fuzzing for finding traffic violations of autonomous vehicles. IEEE Transactions on Software Engineering 49(4):1860–1875

Zhou H, Li W, Kong Z, Guo J, Zhang Y, Yu B, Zhang L, Liu C (2020) Deepbillboard: systematic physical-world testing of autonomous driving systems. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA, ICSE '20, p 347–358, DOI 10.1145/3377811.3380422, URL https://doi.org/10.1145/3377811.3380422