DriveFuzz: Discovering Autonomous Driving Bugs through Driving Quality-Guided Fuzzing

Seulbae Kim
Georgia Institute of Technology
Atlanta, Georgia, USA
seulbae@gatech.edu

Major Liu
University of Texas at Dallas
Richardson, Texas, USA
major.liu@utdallas.edu

Junghwan "John" Rhee
University of Central Oklahoma
Edmond, Oklahoma, USA
jhree2@uco.edu

Yuseok Jeon
UNIST
Ulsan, Republic of Korea
ysjeon@unist.ac.kr

Yonghwi Kwon
University of Virginia
Charlottesville, Virginia, USA
yongkwon@virginia.edu

Chung Hwan Kim
University of Texas at Dallas
Richardson, Texas, USA
chungkim@utdallas.edu

ABSTRACT

Autonomous driving has become real; semi-autonomous driving vehicles in an affordable price range are already on the streets, and major automotive vendors are actively developing full self-driving systems to deploy them in this decade. Before rolling the products out to the end-users, it is critical to test and ensure the safety of the autonomous driving systems, consisting of multiple layers intertwined in a complicated way. However, while safety-critical bugs may exist in any layer and even across layers, relatively little attention has been given to testing the entire driving system across all the layers. Prior work mainly focuses on white-box testing of individual layers and preventing attacks on each layer.

In this paper, we aim at holistic testing of autonomous driving systems that have a whole stack of layers integrated in their entirety. Instead of looking into the individual layers, we focus on the vehicle states that the system continuously changes in the driving environment. This allows us to design DriveFuzz, a new systematic fuzzing framework that can uncover potential vulnerabilities regardless of their locations. DriveFuzz automatically generates and mutates driving scenarios based on diverse factors leveraging a high-fidelity driving simulator. We build novel driving test oracles based on the real-world traffic rules to detect safety-critical misbehaviors, and guide the fuzzer towards such misbehaviors through driving quality metrics referring to the physical states of the vehicle.

DriveFuzz has discovered 30 new bugs in various layers of two autonomous driving systems (Autoware and CARLA Behavior Agent) and three additional bugs in the CARLA simulator. We further analyze the impact of these bugs and how an adversary may exploit them as security vulnerabilities to cause critical accidents in the real world.

CCS '22, November 7–11, 2022, Los Angeles, CA, USA

ACM ISBN 978-1-4503-9450-5/22/11…$15.00
https://doi.org/10.1145/3548606.3560558

CCS CONCEPTS

• **Security and privacy → Software and application security**; • **Computer systems organization → Embedded and cyber-physical systems**.

KEYWORDS

Autonomous driving system; Fuzzing

ACM Reference Format:
Seulbae Kim, Major Liu, Junghwan "John" Rhee, Yuseok Jeon, Yonghwi Kwon, and Chung Hwan Kim. 2022. DriveFuzz: Discovering Autonomous Driving Bugs through Driving Quality-Guided Fuzzing . In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3548606.3560558

1 INTRODUCTION

Autonomous driving technology has recently achieved significant breakthroughs, making self-driving vehicles closer to practical usages [20, 21]. Modern vehicles in an affordable price range are already being shipped with semi-autonomous driving systems on board. Major automotive companies are developing autonomous driving systems (ADSes) to deploy fully autonomous vehicles that can reliably operate on public roads within this decade [30]. However, despite the notable successes in the autopilot technology, reports on fatal accidents caused by erroneous ADSes are continuing [9, 11, 59, 60, 81]. Moreover, recent work has found many unpatched bugs in open-source ADSes [32] and analyzed that comprehensive testing of an ADS still remains challenging [51].

To ensure the safety of autonomous driving, existing work has focused on individual layers of an ADS. Specifically, the security research community has been extensively focusing on finding adversarial examples on the perception layer [13, 17, 24, 39, 57, 75, 77, 79], assuming a threat model in which an attacker attempts to confuse the machine learning model by supplying a deceptive driving scene (*e.g.*, modifying a traffic sign) or spoofing sensor data (*e.g.*, injecting falsified LiDAR points). Some other works test the robustness of the machine learning model using synthesized and transformed images of driving scenes [67, 82, 87]. There are also testing approaches for other layers (e.g., sensing [15, 33, 42] and planning [16, 63, 83]). However, they still focus on individual layers.

Although these works substantially improve the security of the individual layers, they are not designed to cover the attacks exploiting vulnerabilities outside their scopes or specific layers; for example, attacks that target bugs irrelevant to the machine learning model or bugs in the actuation layer. In addition, due to the multi-layer architecture of ADSes where different layers work together in a cascading manner, a bug in one layer may not be detected if tested individually. For instance, a bug in the perception layer may not cause a visible impact when tested alone, but may cause the planning layer to misbehave. Moreover, bugs in multiple layers may jointly contribute to one misbehavior. Such bugs can only be detected if all the integrated layers are tested together as a whole.

In this paper, we introduce a novel approach to enable comprehensive testing of ADSes to uncover critical bugs across all layers. We aim to design a fully automated testing framework that generates realistic test input scenarios on the fly to holistically test ADSes based on the following two key insights:

- With the recent advances in driving simulators for autonomous vehicles [28, 71, 74], it has become feasible to generate an unlimited number of *high-fidelity test input scenarios* with various driving conditions, including the map, vehicles, pedestrians, and weather conditions that closely reflect those of the real environments and offer a highly desirable testing environment to stress *all layers* of the tested system.
- Regardless of which layer they belong to, the impact of bugs ultimately affects *the physical states of the vehicle*[1] (*e.g.*, position and velocity) negatively, for example, causing a collision. Thus, we focus on detecting misbehaviors by monitoring the vehicle states that the ADS continuously alters. These states can also be used as *feedback* to find bugs more efficiently without relying on the information specific to individual layers.

Based on these insights, we propose DriveFuzz, a feedback-guided fuzzing framework for end-to-end testing of ADSes leveraging a driving simulator (CARLA [28]). DriveFuzz plugs a target ADS into the fuzzing framework and tests the self-driving system stack as a whole to facilitate the test coverage to span all layers. It generates and mutates *driving scenarios* in which the ADS has to drive from one point to another, and simulates them in a three-dimensional virtual environment (similar to a racing video game), where it has full control over both *spatial and temporal dimensions of the input spaces as well as multiple actors and entities* including the roads, pedestrians, and vehicles.

During a test, DriveFuzz utilizes our new *driving test oracles* derived from real-world traffic rules and regulations [64], and actively monitors the vehicle states to detect any *misbehavior* that violates the oracles. We define misbehavior of ADS as safety-critical and illegal traffic violations, including collisions, traffic infractions, and immobility, which have apparent symptoms that wreak havoc on the safety of humans. If such illegal misbehaviors are not found, DriveFuzz measures the *driving quality score* of the test input scenario by quantifying the factors that indicate reckless driving, *e.g.*, accelerating too hard. The resulting score is then used as feedback to generate the subsequent test scenario more efficiently (*i.e.*, towards causing more unsafe driving scenarios), such that it will trigger corner case bugs more quickly than completely random fuzzing (as demonstrated by existing software fuzzers [12, 55, 86]).

We evaluate DriveFuzz by testing Autoware [44, 45], which is a full-fledged (Level 4 [25]) ADS extensively used by car manufacturers and academic institutions [5], and Behavior Agent, which is a native ADS integrated into CARLA. To date, DriveFuzz discovered a total of 34 critical bugs; 33 of which are new bugs, including 17 in Autoware, 13 in Behavior Agent, and 3 simulation bugs in CARLA. We have reported all 34 bugs to the developers, and 10 have been confirmed and being patched, and others are under review. Our discovery of the simulation bugs shows that DriveFuzz is capable of finding bugs in both single and multiple layers of the software stack, including various components for self-driving and even the simulator itself. We observe and demonstrate that the bugs we found are *realistic and practical* to exploit; that is, attackers can exploit them by controlling the external inputs in a seemingly legitimate way (*e.g.*, moving nearby objects).

Our design is generic and portable to other ADSes (*e.g.*, Baidu Apollo [4]) and driving simulators [71, 74] as it sits on the interface between the ADS and simulator (*e.g.*, ROS [69]). Specifically, DriveFuzz does not require the source code and instrumentation nor domain knowledge of the target ADS since it only controls the input to the system (input driving scenario) and monitors the physical output (vehicle states).

This paper makes the following contributions:

- We propose a practical automated testing approach capable of fuzzing ADSes end-to-end, and revealing safety-critical misbehaviors based on real-world driving test oracles.
- We design a novel driving quality metric to estimate the effectivness of test driving scenarios in exploring the input space of ADSes based on vehicle states, and leverage the metric to better guide the mutation engine towards test scenarios that trigger safety-critical misbehaviors.
- We implement and evaluate the proposed framework in a prototype called DriveFuzz to demonstrate how feedback-driven fuzzing can be applied to the domain of ADSes. We open-sourced DriveFuzz at https://gitlab.com/s3lab-code/public/drivefuzz.
- In our evaluation, we find 33 new bugs, including 30 critical bugs in real ADSes and three bugs in a full-fledged driving simulator. We show that these bugs can be readily exploited to critically impair the safety of ADSes by causing them to crash, cease to operate, or violate safety-critical traffic laws.

2 BACKGROUND

Figure 1 shows a general ADS, which is the amalgamation of hardware and software layers responsible for four core tasks: *sensing*, *perception*, *planning*, and *actuation* [19, 40, 41], where each layer aims to substitute its counterpart of human drivers. Within each layer, multiple components carry out sub-tasks for autonomous driving. These layers work together in a cascading manner to drive the vehicle, *i.e.*, each layer takes input from the previous layers and the produced output is consumed by the following layers.

Sensing. Autonomous vehicles acquire raw data of the surrounding environment using various sensors, typically including a LiDAR (Light Detection and Ranging), cameras, a radar, a GPS device, and IMU (Inertial Measurement Unit) sensors as components. Any fault

[1] We will use *vehicle states* to refer to the physical states of the vehicle herein.

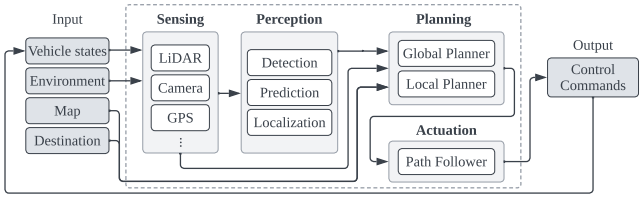


Figure 1: A general autonomous driving system (ADS) consisting of sensing, perception, planning, and actuation layers. By taking the vehicle states and environment perceived by sensors, a 3D map, and a destination as inputs, ADS ultimately outputs control commands, *i.e.*, steering, throttle and brake controls, that in turn update the vehicle states for the next iteration of the loop.

in sensors can feed faulty data to the system, and the error can subsequently propagate to the other layers, resulting in system-level faults in the worst case.

Perception. Perception modules fuse and interpret the captured sensor data to comprehend the current standing and the environment around a vehicle. For example, identifying the traffic signals ahead or predicting the motion of adjacent vehicles by assessing their velocities belongs to the tasks of the perception layer. Many systems leverage various computer vision and machine learning techniques for such tasks. A perception error can mislead the vehicle to make faulty decisions, *e.g.*, estimating the distance to an obstacle to be farther than the actual distance, ending up hitting it.

Planning. With the perceived internal and external states, the planning layer makes a routing plan for the given map and the destination. Generally, it first computes a global trajectory consisting of a sequence of waypoints from the initial position to the destination the user specifies. And then, traffic rules and perceived states (*e.g.*, nearby obstacles) are taken into account by a local planner, which updates the trajectory at runtime to safely drive to the destination. Errors in this layer can cause not only inefficient but also unsafe routing that involves infeasible paths, *e.g.*, crossing a river.

Actuation. Given the generated trajectory to follow, the actuation layer sets up a concrete motion plan consisting of a steering wheel angle, a target speed at waypoints, and the amount of throttling or braking, to seamlessly follow the trajectory. These commands are sent to the steering wheel, throttle, and brake controllers to move an autonomous vehicle as planned. When the commands move the vehicle in the driving environment, the vehicle states are changed and observed by the sensing layer in the following iteration of the loop. Thus, an error in the actuation layer can critically impair the vehicle's ability to properly maneuver in a given situation and may also affect other layers in the loop by changing the vehicle states.

3 THREAT MODEL

Attack surface. We assume an attacker who exploits bugs in *any layer* of an ADS. Specifically, the target attack surface is *not limited* to a single layer (*e.g.*, an adversarial example on the perception layer or injecting falsified data into the sensing layer).

We assume that the attacker does not only exploit bugs that cause an immediate failure of a single layer, but also those that eventually *manifest in other layers*. For example, a bug in the perception layer could make an unnoticeable error in measuring the distance to an object, and cause the planning layer to malfunction when the erroneous distance is used as an input to find the trajectory (bug #15 in §6). Similarly, a bug in the actuation layer could produce an incorrect, but seemingly legitimate command to move the vehicle and cause the perception layer to fail in the next iteration of the control loop through the updated vehicle states (bug #17 in §6).

We do not assume that the attacker takes control over the ADS physically (*e.g.*, attach a device via an OBD-II port) or remotely (*e.g.*, perform remote code execution) to exploit a vulnerability. Instead, the attacker only has control over the *external* inputs such as nearby objects or locations (*e.g.*, moving a nearby vehicle) with a goal to cause critical misbehavior of the autonomous vehicle (*e.g.*, a crash, lane invasion, traffic violations, or becoming immobile). These external inputs are *legitimate* and *authentic* inputs to the ADS, as opposed to maliciously crafted inputs by adversarial attacks (*e.g.*, sensor spoofing) or synthetically generated driving scenes.

Practical feasibility. We argue the attacks in our threat model are realistic and practical. We further discuss the feasibility of exploiting the bugs we identified based on this threat model in §6.3.

Extensibility. The current threat model focuses on the attacks controlling external inputs as they are the most imminent and realistic threats to ADSes. However, since our fuzzer design is generic (§4), the threat model can be extended to other attacks, such as sensor spoofing, *e.g.*, by introducing additional mutable components.

4 DESIGN

4.1 Overview of DriveFuzz

DriveFuzz is a feedback-driven mutational fuzzer that mutates driving scenarios to test an ADS. It aims to generate physically realistic, yet less-tested corner case driving scenarios to discover safety-critical misbehaviors in the ADS. Figure 2 illustrates the workflow of DriveFuzz along with its four main components.

Provided an input driving scenario, the **mutation engine** (§4.2) generates and mutates various aspects associated with the mission (initial and goal positions), weather, actors (vehicles and pedestrians with their trajectories), and puddles (areas with substantially low friction) in the scenario. The **test executor** (§4.3) launches the ADS to be tested, orchestrates the driving simulator to prepare for the mutated driving scenario, and assigns the mission to the ego-vehicle, *i.e.*, the vehicle solely controlled by the ADS [62, 73]. While the ego-vehicle is carrying out the mission, the **misbehavior detector** (§4.4) utilizes our driving test oracles to detect various safety-critical vehicular misbehaviors. If the ego-vehicle completes the mission without any misbehavior, the **driving quality feedback engine** (§4.5) quantifies the overall driving quality by analyzing the vehicle states to guide further mutations towards the generation of the scenarios that decrease the quality.

4.2 Mutation Engine

4.2.1 Test Input Driving Scenario. The input space of an ADS is extremely large and analogous to that of the real world, along both temporal and spatial domains. To efficiently explore the input space, we identify key mutable components in a driving scenario that can affect various components of an ADS when perturbed. As illustrated in Figure 3, our driving scenario consists of (1) a predefined 3D **map** that mimics the real world with fair precision using a standardized

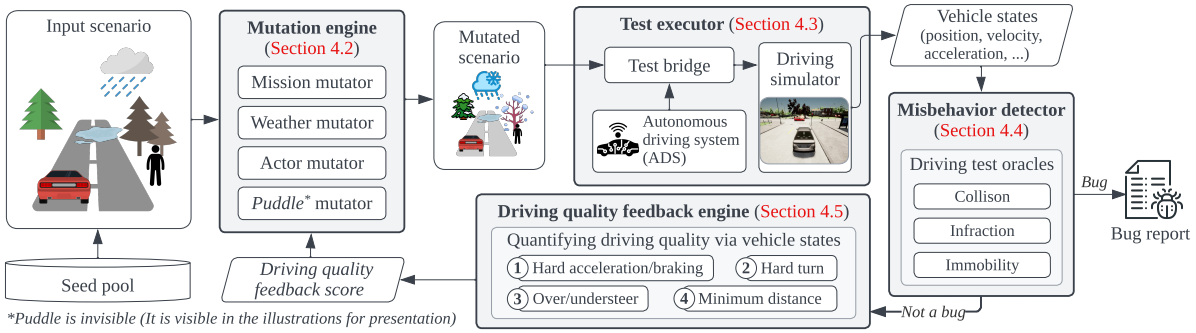


Figure 2: Overview of the architecture and workflow of DriveFuzz.

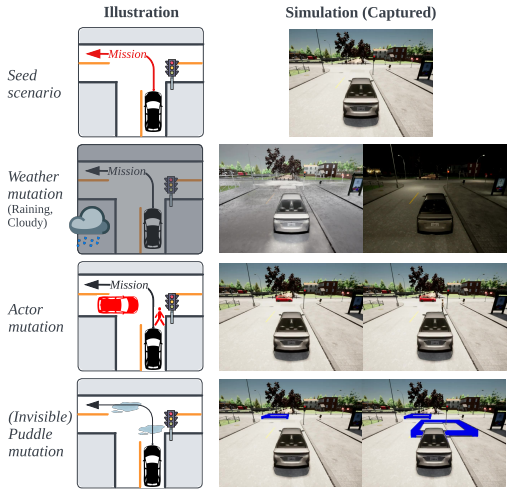


Figure 3: Examples of how mutations are applied to a seed scenario, in which the map and the mission are defined. Blue boxes indicating puddles are added for visualization and invisible during fuzzing.

road network format [10], (2) a **mission** defined by the initial and goal positions, (3) **actors**, *i.e.*, vehicles or pedestrians acting independently to the ego-vehicle, (4) invisible **puddles** that affect the frictional force of the road, and (5) the **weather** conditions.

Similar to traditional mutational fuzzers [2, 86], DriveFuzz best performs when seed scenarios to be evolved are given. It is important to mention that generating input seeds is straightforward and does not require particular expertise in ADS. Specifically, for all the experiments in our evaluation, we construct input seeds from existing maps provided by the simulator, where each map includes a set of valid waypoints (*i.e.*, a mission between two waypoints is guaranteed to be achievable). We obtain the seeds by randomly selecting two of the predefined waypoints for the initial and goal points. We further explain the details with example seeds in Appendix A.

4.2.2 Scenario Mutation. DriveFuzz's scenario mutation aims to gradually increase the mutated scenario's impact on the ego-vehicle under the test. Specifically, by generating and mutating the components of a scenario according to the mutation schedule (§4.2.5), it affects all four layers of the ADS, as summarized in Table 1.

Map and mission selection. A map not only defines static world objects, such as buildings and trees that affect the perception module, but also includes road structures, such as intersections or curbs

Table 1: Layers of an ADS directly affected by taking fuzzing actions to each component of a driving scenario. With all actions combined, the coverage of a mutated scenario effectively spans all layers.

Component	Action	Affected layers
Map and mission	Seed selection	Sensing, Perception, Planning
Actor	Generation & Mutation	Sensing, Perception, Planning
Puddle	Generation & Mutation	Planning, Actuation
Weather	Mutation	Sensing, Perception

that the planner actively interacts with. The purpose of diversifying the mission is to explore different parts of the map and associated objects along with the road structure, enabling DriveFuzz to thoroughly test diverse issues in the perception and planning layers.

Actor generation & mutation. The actors in the scenario affect the sensing, perception, and planning layers, because the behaviors of the actors may force the ADS to deviate from the original routing plan, *e.g.*, by blocking the path.

DriveFuzz *generates* an actor by randomly selecting the type of actor (either a vehicle or a pedestrian), initial position and destination, navigation method, target speed, and trajectory. To have the actor generated within the interactable range of the ego-vehicle (*i.e.*, sensor range spanning the mission), the initial position of an actor is always selected from within a configurable range from the ego-vehicle's initial position. In addition, to diversify the circumstances actors can render, we define four kinds of navigation methods:

1. *Autopilot:* an actor performs a safe and lawful autopilot, using the ground truth traffic data and abiding by all traffic rules while heading to the destination.
2. *Maneuver:* an actor executes a sequence of maneuvers (*i.e.*, drive forward, switch to left/right lane). Each maneuver has a predefined amount of time describing the duration of the action.
3. *Linear:* an actor blindly travels to the destination following a linear trajectory without considering the surrounding traffic or objects, thereby not complying with any rule.
4. *Immobile:* an actor remains stationary at the initial position.

The *mutation* of an actor includes a process of modifying the aspects of a generated actor. Except for the type and the navigation method, all other aspects (*e.g.*, the initial position) can be mutated for an actor to exhibit a variety of behaviors.

Puddle generation & mutation. Invisible puddles (*e.g.*, black ice) reduce the surface friction of the road and thereby affect the actuation of an ADS. For example, based on the surface condition of the road, or the tire condition, an ADS has to adjust the control commands accordingly (*e.g.*, avoid generating large torques on the wheels) to ensure the ego-vehicle does not lose control.

DriveFuzz *generates* a puddle by randomly selecting the location, size, and frictional force. Similarly to the actors, it *mutates* a puddle by modifying the location, size, and friction of a puddle.

Weather mutation. Weather affects the sensing and perception layers, which act as the eyes of an ADS. DriveFuzz mutates the weather concerning the following eight aspects: rain, cloud, wind, fog, wetness, puddle, solar azimuth angle, and solar altitude. With a wide variety of available combinations, a realistic weather condition can be simulated and tested.

4.2.3 Ensuring Physically Valid Mutation. We aim to test an ADS under physically feasible circumstances that can occur in real life. Thus, all mutated driving scenarios need to be semantically practicable; for example, an actor should not suddenly appear in front of the ego-vehicle during a simulation. At the same time, DriveFuzz should not ignore unusual yet possible scenarios such as running into a person on a highway[2]. To this end, we ensure the testing always starts after the simulation is fully loaded with the scenario, including the weather condition and all actors/puddles, preventing the abrupt creation of any objects during testing. In addition, while allowing the random generation of objects, we impose a spatial constraint and a temporal constraint to prevent events defying the physical laws and to forestall false positive scenarios where an ego-vehicle is not at fault of the misbehavior.

Spatial constraint. To prevent unrealistic jams resulting in physically impossible scenarios, such as two distinct vehicles being partially overlapped at an adjacent place, the initial positions of all actors are constrained to be at least a few meters away from each other. The same constraint applies to the static objects (*e.g.*, buildings, traffic lights); the dimensions of the actors at their initial positions cannot offend the bounding boxes of the static objects.

If the spatial constraint is violated, the mutation engine considers it as an infeasible scenario, and attempts a mutation again by randomly selecting the location of the actor that violates the spatial constraint. Note that this random process results in few additional computations (*e.g.*, generating random numbers and checking the spatial constraint), which only cause negligible overhead (see §6.8).

Temporal constraint. To prevent unrealistic movements of actors, DriveFuzz imposes temporal constraints by limiting the maximum speed of actor vehicles and pedestrians to a conservative value, *e.g.*, 20 and 6 *mph*, respectively.

The spatial and temporal constraints, combined with the navigation methods, are designed to preclude most unrealistically reckless scenarios that might lead to false positives. For example, a scenario in which a pedestrian stands still until the ego-vehicle approaches and then suddenly jumps in at the last moment to cause an unavoidable collision cannot be generated because (1) there is an initial distance between the pedestrian and the ego-vehicle (spacial constraint), (2) the pedestrian cannot walk unrealistically fast (temporal constraint) and (3) he/she walks either safely (autopilot), linearly to the destination (linear) or does not move (immobile), adhering to the navigation methods.

4.2.4 Mutation Strategy. Depending on the particular aspect of the target ADS to be stress-tested, different mutation strategies specifying the mutable attributes and constraints can be developed and applied. The strategies we propose include, but are not limited to the following:

- Adversarial maneuver-based: only introduces and alters the maneuver of the adjacent actors, forcing interactions with the target system, *e.g.*, an actor vehicle suddenly cutting the ego-vehicle off by switching lanes.
- Congestion-based: only introduces autopilot actors so that the target ADS drives in increasingly congested, yet lawful scenarios.
- Entropy-based: only introduces a linear or immobile actor, testing the ability of the target system to safely drive around reckless drivers and unlawful pedestrians.
- Instability-based: only inserts a puddle of different size and friction, testing the robustness of the motion controller to deal with sudden instabilities triggered by external forces.

Each strategy can be independently applied to a fuzzing campaign, or orchestrated to be jointly applied under a probabilistic scheduling (*e.g.*, randomly selecting the next strategy to apply after each round).

Algorithm 1: Driving quality feedback-driven fuzzing

```
Input  : S - a set of seed scenarios (seed pool), strategy - mutation strategy,
         N_c - Maximum # cycles, N_p - Size of population
Output : bug - a detailed bug report, s' - the buggy scenario
1  foreach seed ∈ S do
2      fuzz_one(seed)
3  procedure fuzz_one(seed)
4      s ← seed
5      for cycles ← 1 to N_c do
6          s ← mutator.generate(s, strategy) // §4.2.2, §4.2.4
7          last_worst_score ← 0
8          for rounds ← 1 to N_p do
9              s' ← mutator.mutate(s) // §4.2.2, §4.2.3
10             states ← executor.simulate(s') // §4.3
11             if detector.check_misbehavior(states) /*§4.4*/ == True then
12                 save_bug_report(states, s')
13                 return
14             else
15                 score ← feedback.check_driving_score(states) // §4.5
16                 if score ≤ last_worst_score then
17                     last_worst_score ← score
18                     successor ← s'
19         s ← successor
```

4.2.5 Feedback-driven Mutation Scheduling. To efficiently explore the input space, DriveFuzz leverages a feedback mechanism to generate and mutate the components of a scenario as presented in Algorithm 1. At each cycle, DriveFuzz first generates and introduces an actor or puddle to the scenario (line 6). Then, the generated component or the weather is mutated N_p times to create a population of size N_p (line 9). Each mutated scenario is executed, and its quality is evaluated by the feedback engine (§4.5), which measures the driving quality score (line 15). At the end of the cycle, if none of the population triggers a misbehavior (line 11), they are ranked by the driving quality score, and the one that scored the least among the population is selected (line 18) and passed on to the next cycle (line 19). DriveFuzz repeats the process of adding a new component into this chosen scenario and searching for the most "harmful" mutation that disrupts the driving behavior of an autonomous vehicle most significantly.

As the fuzzing cycle repeats, the scenario gradually gains intensity as more actors and puddles are inserted. However, more mutations may not always lead to a critical misbehavior. To prevent

[2]The person could be the driver of a broken car stopped on the side road.

exploring less-promising directions of the mutation, DriveFuzz aborts and starts a new campaign with a new seed (line 2) when it reaches the maximum cycles (N_c) without finding a misbehavior.

4.3 Test Executor

The test executor runs an ADS under the given driving scenario in a driving simulator, collecting various vehicle states for the fuzzing process. For the simulator, we choose to use CARLA [28], a high-fidelity driving simulator implemented using Unreal Engine. CARLA is known for its active development status and usage, professionally designed realistic maps, a wide range of supported sensors, flexibility in controlling various aspects of a driving scenario, and the ability to integrate various ADSes with ease by supporting Robot Operating System (ROS) [69], a universal middleware, which many robotic systems are built on top of.

4.3.1 Test Bridge. The test bridge connects the mutation engine to the ADS and the simulator, testing the mutated driving scenarios.

Loading the input driving scenario. The test bridge first orchestrates the CARLA simulator to set up the input scenario in the simulated world. It connects to the simulation server, opens the map, configures the weather, spawns actors, puddles, and the ego-vehicle as specified by the mutated input scenario. When the loading is finished, the ADS is launched.

Initializing the target ADS for testing. The test bridge launches the ADS stack and waits until it is completely initialized. Then, it attaches the autopilot functionality to the ego-vehicle spawned in the simulated world. Once the system is online and the autopilot agent is loaded, a test is ready to be simulated.

4.3.2 Driving Simulator. The driving simulator plays a key role in synthesizing real-time sensor data as well as computing vehicle states. The simulator in the loop has multiple benefits compared to an alternative option of using a real vehicle [63] equipped with appropriate sensors and a companion computer to bridge the ADS software with the vehicular controllers. We employ the simulator in DriveFuzz to fully leverage the following benefits: (1) test vehicles of different physical specifications and self-driving software stacks without altering the testing scheme, (2) test vehicles with significantly lower cost compared to the testing of physical vehicles, (3) test vehicles under various circumstances including but not limited to unlikely situations without physical constraints, and (4) fully automate a testing sequence.

In a loop, CARLA is responsible for simulating each frame by applying the control commands issued by the ADS to the ego-vehicle, and updating the states of in-simulation actors, *e.g.*, the position of a pedestrian moving at 1.5 m/s towards the North. The ADS combines the updated states of the ego-vehicle with the new sensory data read from the simulator to decide the subsequent control command. The loop terminates when the vehicle reaches the destination, or any issue is found by the misbehavior detector.

4.4 Misbehavior Detector

When the ADS fails to handle the input scenario, it can lead to a wide spectrum of undesirable consequences from software-oriented errors (*e.g.*, memory error in a component) to vehicular misbehaviors (*e.g.*, collision). Our misbehavior detector intends to point out *obvious illegal acts* in the driving behaviors of ADS by applying definitive standards. Inspired by the fact that the ADSes are designed to drive in the real world complying with traffic rules and regulations [64], we build the following three driving test oracles that check for the events that are closely related to human safety: *collisions*, *infractions*, and *immobility* of the ego-vehicle.

- **Collision.** Collision is one of the most destructive events that can cause significant damage to human drivers. By attaching a collision sensor to the ego-vehicle (that would be corresponding to multiple sensors around a real vehicle), a collision to any object is captured and reported.
- **Infraction.** Infractions are traffic violations including (1) speeding, (2) invading lanes, and (3) running on red lights, which are directly involved in approximately 30%, 8.5%, and 4%, respectively, of the annual fatal accidents in the United States in 2018 [58]. As DriveFuzz has full access to the simulated space, it compares the states of the vehicle (*e.g.*, current speed) with the defined traffic rule (*e.g.*, speed limit) to check for any violation.
- **Immobility.** A vehicle that is not moving at a particular location would become a cause of subsequent undesirable events such as collisions (*e.g.*, a car stopped in the middle of an intersection would cause other cars to crash into it). The immobility monitor measures the time duration when the vehicle is not moving, excluding legitimate stops (*e.g.*, at traffic lights). If it exceeds a threshold (60 *sec* in this paper), that is considered a misbehavior.

The misbehavior detector monitors every frame of the simulation and refers to these oracles to check for any violation (line 11 of Algorithm 1). Upon detecting a misbehavior, the incident is reported, and the simulation is terminated immediately after logging all vehicle states for a later inspection.

4.5 Driving Quality Feedback Engine

We propose a new *driving quality metric* that abstracts the performance of ADS under a testing scenario. In particular, the metric is measured by evaluating various events in the driving maneuvers during testing that does not immediately trigger safety-critical misbehaviors, but are likely to lead to those. The metric is later used to guide the input scenario mutation towards buggy conditions.

Note that we develop this new metric because existing metrics such as code coverage are not suitable for our context. Specifically, while the code coverage-guided mutation has proven effective in many modern grey-box fuzzers to approximate the amount of the explored input space for *sequential* programs, it is not effective for distributed and *stateful* systems such as an ADS. In particular, ADS runs smaller nodes changing states driven by data, consisting of loops running state machines. Their code coverage quickly saturates regardless of the testing progress, hence inadequate to approximate the test coverage (see Appendix B).

4.5.1 Driving quality measurement. When no safety-critical misbehavior is detected, DriveFuzz analyzes the driving data to guide the input mutator so that it can effectively mutate the input scenario towards the scenario *likely to* trigger safety-critical misbehaviors. To quantitatively measure how close a vehicle is to the safety-critical misbehaviors, we refer to the official reports [1, 56] from the U.S. Department of Transportation, National Highway Traffic Safety

Administration (NHTSA), which investigates the causes of traffic accidents. According to the reports, 52% of the fatal accidents of known causes are attributed to either *reckless* or *clumsy* driving behaviors, such as hard acceleration or oversteer. Many major car insurance companies (*e.g.*, Allstate, Progressive, and State Farm) also support this idea by having their programs evaluate the driving quality based on the number of hard braking, hard acceleration, and hard turning events to determine the insurance rate [7, 68, 72, 78].

Inspired by the real-world usage, DriveFuzz measures the driving quality based on the number of hard accelerations, hard brakings, hard turns, oversteers and understeers, and the minimum distance to other actors. The following paragraphs ①–④ present how we measure each factor constituting the driving quality by leveraging the vehicle states on the driving data of the simulation.

① **Hard acceleration and hard braking detection.** The ratio of longitudinal acceleration of a vehicle A_x to the gravitational constant g (approximately 9.8 m/s^2) is a generally accepted way of representing the harshness of acceleration or braking events [14, 36]. The hard acceleration/braking indicator K_{ab} is given by:

$$K_{ab} = A_x/g \tag{1}$$

If K_{ab} exceeds a threshold, DriveFuzz counts the frame as either a hard acceleration or hard braking event. For the threshold, NHTSA used $0.4 - 0.6$ to identify hard acceleration or hard braking [26, 49]. Other studies claim that 0.5 is the threshold people typically agree on [14, 36]. Taking the upper bound, we use 0.6 as a decision boundary for K_{ab}, which is the force that a vehicle can reach or stop from 60 *mph* in less than five seconds.

$$\#ha = count(K_{ab} \geq 0.6),\ \#hb = count(K_{ab} \leq -0.6) \tag{2}$$

② **Hard turn detection.** A hard turn occurs when a driver tries to turn the vehicle at an excessive speed. As a hard turn is related to the lateral force applied to the vehicle, we leverage a detection algorithm that uses a hard turn indicator K_t, such that

$$K_t = V_y/SWA \tag{3}$$

where V_y and SWA denote the lateral speed, and the steering wheel angle, respectively. If (1) SWA is greater than a steering threshold, and (2) K_t is above a hard turn threshold, DriveFuzz counts the frame as a hard turn. Both thresholds are configurable, and we empirically determined them as 20 and 0.18, respectively, such that

$$\#ht = count(SWA \geq 20 \wedge K_t \geq 0.18) \tag{4}$$

③ **Oversteer and understeer detection.** Oversteer and understeer represent the reaction of a vehicle to the steering effort. Oversteer occurs when the rear tires lose grip and the vehicle turns more than the amount the driver steers, and understeer occurs when the front tires lose grip, so the vehicle turns less than the steering amount. Both frequently occur in competitive racing sports where aggressive controls are required, and they often lead to accidents as a vehicle loses control and slips while turning. In normal driving conditions, oversteer or understeer can take place as a result of imprecise control, or because of low friction on the road caused by natural events, such as black ice. No matter what the cause is, both are deemed very dangerous [34, 38], being ranked in the 7th in "top 12 causes of fatal car accidents in the USA" by NHTSA.

Broadly, there are two approaches that attempt to detect oversteer and understeer events: model-based detection and fuzzy logic-based detection. Model-based detection tends to be accurate but requires precise models of the vehicles, tires, and friction. On the other hand, fuzzy logic [61, 85] approximates the "truthiness" of a linguistic statement on a continuum as a fuzzy value rather than a boolean value and aggregates multiple values with rules to infer the final level of output. To grasp the overall safety and generate feedback, DriveFuzz does not require the detection to be meticulously accurate. Moreover, model-specific detection is ill-suited to the purpose of DriveFuzz to serve as a generic framework for testing ADS planted on various vehicle models. Thus, we adopted the fuzzy logic-based detection proposed by [8, 66] that works reasonably well across different vehicle models.

In summary, four indicators, SWA (steering angle in deg), V_x (longitudinal velocity in km/h), AV_z (yaw rate in deg/s), and A_y (lateral acceleration in gs), which can be obtained from the driving data are used for fuzzy logic to compute the degree of oversteer K_{os} and understeer K_{us}. With the inferred oversteer and understeer levels, which are floating-point numbers in $\{0, 1\}$, we tuned the threshold to determine the final results as follows:

$$\#os = count(K_{os} \geq 0.4), \#us = count(K_{us} \geq 0.4) \tag{5}$$

④ **Minimum distance** Any failure to maintain a safe distance from other vehicles or pedestrians implies that the system is close to potential misbehaviors. For example, if a minimum distance to a pedestrian is one foot, we can interpret that as the ego-vehicle near-missed hitting the pedestrian, and with a slight mutation, the scenario could cause a collision. To take such events into account, DriveFuzz measures the distances from the ego-vehicle to all other actors per frame and keeps track of the minimum distance, md. The smaller md is, the more deduction is applied to the driving score.

Overall driving quality score. With all the ingredients ready, DriveFuzz computes the overall driving quality score by multiplexing the number of events. The driving quality score starts from zero, and the number of the events captured above is deducted, and then the inverse of the minimum distance (*i.e.*, $1/md$) multiplied by a configurable coefficient is deducted, resulting in the final feedback score. In summary, the driving quality $score$ is given by:

$$score = -(\#ha + \#hb + \#ht + \#os + \#us + c/md) \tag{6}$$

The final $score$ is delivered to the input mutator for the decision of the scenario that is worth further mutating (line 15 of Algorithm 1).

Tuning metrics. Weights for the driving quality factors can be configured to prioritize certain misbehavior, depending on the users' needs and the characteristics of the target system. In our experiments, we treat all factors equally (*i.e.*, Equation 6) to prevent DriveFuzz from being biased toward any particular misbehavior.

4.5.2 Key contribution of driving quality feedback. In the context of testing ADSes, our design of physical vehicular states-based driving quality feedback is highly pertinent for two reasons. First, using the physical states of a vehicle, it allows DriveFuzz to pragmatically quantify the recklessness of the driving without requiring code-level analysis nor examining internal states, of which the availability is not always guaranteed. Second, unlike the feedback suggested by the related work [52] that may lead ADS away from the bugs we detected (see §6.4), our fine-grained feedback mechanism provides proper guidance towards unsafe driving scenarios, resulting in the detection of actual, safety-critical misbehaviors.

Table 2: Implementation complexity of DriveFuzz.

	Component	LoC	Language
DriveFuzz framework	Mutation engine	440	Python
	Misbehavior detector and driving test oracles	119	Python
	Feedback engine and driving quality metrics	636	Python
	Test executor	1125	Python
	Additional bridge for Autoware	48	Shell script

5 IMPLEMENTATION

DriveFuzz is prototyped in approximately 2.3K lines of Python 3 code, as shown in Table 2.

ROS and portability. ROS [69] is a de facto middleware that provides a means of message passing between distributed nodes, hardware abstraction, and a toolset for the easier development of robotic systems. DriveFuzz incorporates ROS in the design of the test executor and makes any ROS-based ADSes [44, 45] and simulators [28, 71, 74] pluggable into the system.

Bug reproduction. ROS leverages a publisher-subscriber message passing scheme; nodes publish messages to a topic, and other nodes subscribe to the topic to receive the messages. Thus, all flows including sensory inputs and control commands are summarized in the messages. DriveFuzz records all underlying ROS messages, essentially capturing all data flows that happened during fuzzing, and later replays them to reproduce and debug the buggy scenarios.

Clock synchronization. Depending on the hardware, the simulation could run slower than a wall clock and stall ADSes from obtaining real-time sensor data. DriveFuzz synchronizes ADSes with the simulator's time, not the wall clock, so that if the simulation runs behind the wall clock while computing and rendering each frame, ADSes can wait for the data and react upon correctly.

6 EVALUATION

We evaluate the effectiveness of DriveFuzz as a fuzzer for ADSes by assessing the number of bugs detected by DriveFuzz (§6.1) with their analyses (§6.2), the feasibility of exploiting the discovered bugs in the real world (§6.3), how DriveFuzz fares against a state-of-the-art approach (§6.4), the correctness of the driving test oracles (§6.5), the correctness of driving quality measurement (§6.6), the effectiveness of feedbacks (§6.7), and the fuzzing performance (§6.8).

Experimental setup. We ran DriveFuzz on a server machine running Ubuntu 18.04, powered by 16-core Intel Xeon Gold 5218 CPU, 192-GB main memory, and 8 GeForce RTX 2080 Ti graphics cards. To allow parallel execution of testing workloads and increase the testing performance, we used Docker containers. We simultaneously ran four pairs of CARLA and ADS containers connected via a ROS bridge, and assigned a dedicated GPU to each container. For the effectiveness and performance evaluations (§6.7 and §6.8) where randomness can skew the results, we report the average of repeated runs, following the suggestions in [50].

Test targets. We tested the following ADSes:

- Autoware: A full-fledged ADS with active development status. Started in 2015, it has been internationally adopted by many well-known automobile manufacturers, *e.g.*, BMW [6], and qualified to run driverless vehicles on public roads in Japan since 2017 [80].
- Behavior Agent: An ADS developed by CARLA, implementing path planning, feedback-based PID control, compliance with traffic laws, and collision avoidance.

Seed scenarios. For the experiments, we used 40 valid seed scenarios to test target systems in various environments and conditions, obtained by the procedure described in Appendix A.

6.1 Detected Misbehaviors

DriveFuzz found multiple scenarios that trigger various safety-critical misbehaviors, stemming from a total of 34 bugs in Autoware, Behavior Agent, and CARLA; 17 previously unknown bugs and one known issue in Autoware, of which 8 bugs are already confirmed, and 13 bugs in Behavior Agent, which are awaiting confirmation. In CARLA, three critical simulation bugs are detected, and two of them have been acknowledged so far. All bugs have been responsibly reported to and discussed with the developers.

Table 3 summarizes all the new bugs we found, the component they are located in, the impact and the root cause of each bug. By comprehensively testing the entire ADS end-to-end with high-fidelity driving scenarios, DriveFuzz identifies misbehaviors from all components of the system, including sensing, perception, planning, and actuation. The videos of the bugs are available at **https://youtube.com/channel/UCpCrUiGanDKX-qxj8jcUVGQ**.

Root cause identification. To identify the bug and the root cause of observed misbehaviors, we replay the recorded simulation data and analyze critical events. This procedure involves a component-wise data- and control-flow analysis of answering the following diagnostic questions:

- Did sensors accurately read the environment?
- Did the perception layer correctly interpret the sensor data?
- Did the planning layer find a feasible path?
- Did the actuation layer emit appropriate control commands?

Contribution of test oracles. The "Impact" column in Table 3 shows the accumulation of all misbehaviors triggered in multiple scenarios, which stem from the same root cause. For example, bug #28 (impact C, L) caused a collision in some scenarios, and a lane invasion in other scenarios, depending on the location on the map and the nearby objects at the moment the bug was triggered. Overall, the collision oracle contributed to the detection of the most (76%) bugs, because ADS bugs usually make the vehicle lose control and susceptible to a collision. Traffic infractions (V, L, and S) were triggered by 60% of the bugs. We can also observe that all bugs that caused a lane invasion caused a collision as well. This does not imply that the collision oracle can replace the lane invasion oracle; these oracles were individually triggered in different scenarios. For versatility under any circumstances, both oracles should be utilized.

6.2 Case Study

We present an in-depth analysis of the selected bugs DriveFuzz found. The bugs are categorized into four different types as follows.

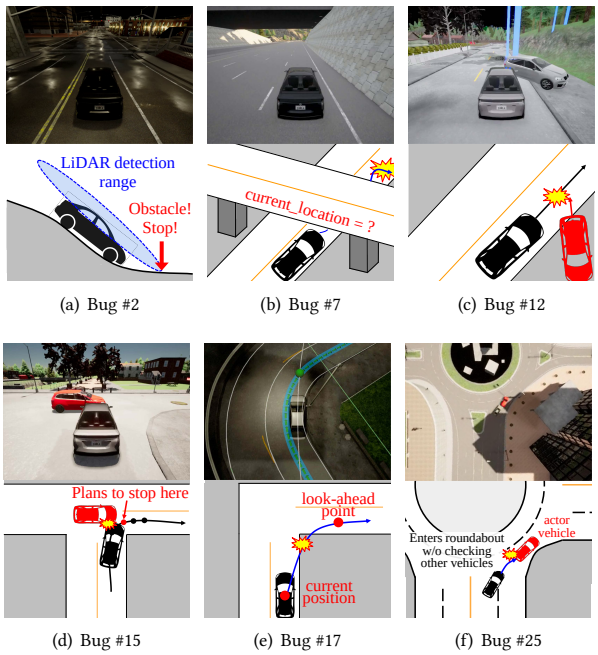
Cross-layer bugs. DriveFuzz identifies bugs that are caused by multiple layers, requiring comprehensive testing of the ADS with all layers. Cross-layer bugs are difficult to detect by testing the individual layers because their symptoms often become visible in a different layer from the buggy layer.

• Bug #15 (see Figure 4(d)) stems from two subtle problems in the perception and planning layers. First, the perception layer measures the distance to the obstacle from the center of the ego-vehicle,

Table 3: New bugs DriveFuzz revealed in multiple layers of Autoware, Behavior Agent, and CARLA simulator. Impact indicates which system-level misbehaviors were captured by the driving test oracles during testing, strategy shows the mutation strategy used, and the root cause is determined by our manual analysis afterward. ACK indicates whether bugs are confirmed by the developers.

	Bug #	Layer	Component	Description	Impact	Strategy	Root cause	ACK
Autoware	01	Sensing	Fusion	LiDAR & camera fusion misses small objects on road	C	all	Logic err	
	02	Perception	Detection	Perceives the road ahead as an obstacle at a steep downhill	I	all	Logic err	✓
	03	Perception	Detection	Fails to semantically tag detected traffic lights and cannot take corresponding actions	C, V	all	Logic err	
	04	Perception	Detection	Fails to semantically tag detected stop signs and cannot take corresponding actions	C, V	all	Logic err	
	05	Perception	Detection	Fails to semantically tag detected speed signs and cannot take corresponding actions	V	all	Logic err	
	06	Perception	Localization	Faulty localization of the base frame while turning	C, L	all	Logic err	✓
	07	Perception	Localization	Localization error when moving underneath bridges and intersections	C, L	all	Logic err	✓
	08	Planning	Global planner	Generates infeasible path if the given goal is unreachable	C, L	all	Logic err	✓
	09	Planning	Global planner	Generates infeasible path if the goal's orientation is not aligned with lane direction	C, I, L	all	Logic err	✓
	10	Planning	Global planner	Global path starts too far from the vehicle's current location	C, I, L	all	Logic err	✓
	11	Planning	Local planner	Target speed keeps increasing at certain roads, overriding the speed configuration	S, C	all	Logic err	✓
	12	Planning	Local planner	Fails to avoid forward collision with a moving object	C	all	Logic err	
	13	Planning	Local planner	Fails to avoid lateral collision (ADS perceives the approaching actor before collision)	C	ent	Not impl	
	14	Planning	Local planner	Fails to avoid rear-end collision (ADS perceives the approaching actor before collision)	C	ent	Not impl	
	15	Planning	Local planner	While turning, ego-vehicle hits an immobile actor partially blocking the intersection	C	ent	Logic err	
	16	Actuation	Pure pursuit	Ego-vehicle keeps moving after reaching the destination	C, L	all	Logic err	✓
	17	Actuation	Pure pursuit	Fails to handle sharp right turns, driving over curbs	C, L	all	Faulty conf	
Behavior Agent	18	Perception	Detection	Indefinitely stops if an actor vehicle is stopped on a sidewalk	I	ent	Logic err	
	19	Perception	Detection	Flawed obstacle detection logic; lateral movement of an object is ignored	C	con	Logic err	
	20	Planning	Global planner	Generates inappropriate trajectory when initial position is given within an intersection	C, L, V	all	Logic err	
	21	Planning	Local planner	Improper lane changing, cutting off and hitting an actor vehicle	C	man	Logic err	
	22	Planning	Local planner	Vehicle indefinitely stops at stop signs as planner treats stop signs as red lights and waits for green	I	all	Logic err	
	23	Planning	Local planner	Vehicle does not preemptively slow down when the speed limit is reduced	S	all	Logic err	
	24	Planning	Local planner	Always stops too far (> 10 m) from the goal due to improper checking of waypoint queue	F	all	Logic err	
	25	Planning	Local planner	Collision prevention does not work at intersections (only checks if actors are on the same lane)	C	all	Logic err	
	26	Planning	Local planner	Fails to avoid lateral collision (ADS perceives the approaching actor before collision)	C	man	Not impl	
	27	Planning	Local planner	Fails to avoid rear-end collision (ADS perceives the approaching actor before collision)	C	man	Not impl	
	28	Planning	Local planner	No dynamic replanning; the vehicle does infeasible maneuvers to go back to missed waypoints	C, L	ins	Not impl	
	29	Actuation	Controller	Keeps over-accelerating to achieve the target speed while slipping, creating jolt back on dry surface	C, L	ins	Not impl	
	30	Actuation	Controller	Motion controller parameters (PID) are poorly tuned, making the vehicle overshoot at turns	C, L	all	Faulty conf	
CARLA	31		Simulator	Simulation does not properly apply control commands	C, L, V	all	Logic err	✓
	32		Simulator	Vector map contains a dead end blocked by objects as a valid lane	I, V	all	Data err	
	33		Simulator	Occasionally inconsistent simulation result	I, V	all	Logic err	✓

[Impact] **C**: **C**ollision / **F**: **F**ails to complete a mission / **I**: Vehicle becomes **I**mmobile / **L**: **L**ane invasion / **S**: **S**peeding / **V**: Miscellaneous traffic **V**iolation
[Strategy] **all**: all strategies / **man**: Adversarial **man**euver-based / **con**: **con**gestion-based / **ent**: **ent**ropy-based / **ins**: **ins**tability-based



(a) Bug #2 (b) Bug #7 (c) Bug #12

(d) Bug #15 (e) Bug #17 (f) Bug #25

Figure 4: Notable cases of detected bugs. The images (top) show the snapshots of the camera feed at the moments when the bugs were detected. The diagrams (bottom) briefly describe the situation.

treating the vehicle as a point. At this layer, this error does not necessarily cause a visible impact. However, when the planner checks if the obstacle is blocking the path based on this invalid distance, it does not consider the dimension of the ego-vehicle, as well. When the trajectory is not linear (*e.g.*, facing an obstacle while turning), the dimension of the ego-vehicle is considered as zero by both layers, making the edge of the bumper hit the obstacle.

• Bug #17 (see Figure 4(e)) is due to a faulty configuration in the actuation layer, which is aggravated by faulty perception. Autoware's controller smooths a trajectory by constantly following the virtual curve from the vehicle's position to the look-ahead point, rather than strictly shooting for every point of a path. The default configuration of the controller sets the minimum look-ahead distance parameter too large, which causes the ego-vehicle to cut through a curb when it enters a sharp right curve at a low speed. When this fault manifests, the perception layer fails to identify the objects on the unexpected trajectory, and the vehicle ends up colliding with the static objects on the curb, *e.g.*, fences or street lights.

Logic errors. A majority of the bugs DriveFuzz discovered turned out to be logic errors, where the logic behind the implementation of a component is the cause of misbehavior.

• Bug #2 (see Figure 4(a)) is caused by both sensor and perception layers; at the end of a steep downhill where the ego-vehicle faces down while the road ahead flattens, the LiDAR of Autoware senses the road ahead as an obstacle. Without any verification of the point cloud data published by the LiDAR, the perception layer concludes

that there is a massive object blocking the way, and the local planner subsequently decides to stop immediately. As the entire path is seemingly blocked, the ego-vehicle becomes immobile thereafter.
• Bug #7 (see Figure 4(b)) is a critical bug that causes a localization error. Autoware utilizes the Normal Distributions Transform (NDT) matching algorithm, which estimates the current position of the ego-vehicle on the map by combining the data from the LiDAR, Inertial Measurement Unit, Global Navigation Satellite System sensors, and vehicle odometer data. The localization plays a pivotal role in the correctness of an ADS, as all driving decisions are made based on the estimated current position. Unfortunately, the NDT matching fails to correctly estimate the position when the ego-vehicle is under a bridge, presumably because its estimation relies solely on the latitude and longitude, but not the altitude.
• Bug #12 (see Figure 4(c)) is notable as it is directly related to the safety of passengers. When a vehicle cuts in from either side to the front of the ego-vehicle, the LiDAR sensor detects the vehicle, and the perception layer perceives it as a vehicle. However, the local planner ignores the perceived vehicle and fails to command a stop.
• Bug #25 (see Figure 4(f)) presents a devastating logic error in the planner of Behavior Agent. The planner should slow down and stop if an obstacle is ahead. However, as a part of optimization, it only checks if anything is on the *same lane* as the ego-vehicle. As a result, when the ego-vehicle is switching lanes or turning at an intersection/roundabout to enter another lane, the planner fails to notice the obvious objects, causing collisions.

Missing features. DriveFuzz found that some of the misbehaviors stem from not implementing essential features.
• Bugs #13, 14, 26, and 27 demonstrate that none of the components of Autoware and Behavior Agent handles lateral and rear-end collision avoidance. Even though the LiDAR sensor covers all 360 degrees and *perceives* approaching vehicles from all directions, the local planner only considers the objects lying in front of the vehicle when revising the path plan, *e.g.*, taking a detour. Thus, when reckless vehicles approached the ego-vehicle from behind or side in some scenarios, the ego-vehicle did not try to avoid them, (*e.g.*, by accelerating or steering), being subject to collisions.
• Bug #29: Electronic Stability Control (ESC) [53] is one of the essential and common in-vehicle safety features that prevents and helps recover from oversteer and understeer by automatically braking individual wheels and limiting engine powers. Unfortunately, this essential feature is missing in Behavior Agent, being vulnerable to bug #29. When the vehicle starts to slip due to a puddle, the rotation of the wheels is not converted to vehicular speed. Not considering the slipping state, the controller keeps generating greater torques on the wheels to achieve the target velocity, which creates an excessive burst of acceleration when the vehicle finally gets out of the puddle and makes the vehicle lose control.

Simulation errors. DriveFuzz also identifies errors within the simulator, showing its end-to-end testing strategy's effectiveness. Bugs #31–33 manifested themselves as one of the misbehaviors the detector examined and later turned out to be the faults of the CARLA simulator while debugging them.
• Bug #31: the ego-vehicle deviated from the planned path while turning left at an intersection. It did not turn as much as it was required to follow the curved path, but still throttled, and crashed into a building. By analyzing the control commands Autoware issued, we found that the ego-vehicle tried to steer more and more towards the left as it deviated from the path. The culprit was CARLA, which did not properly simulate the vehicle states by applying the control commands it received from Autoware. In real vehicles, mechanical errors can cause similar problems if it does not apply physical controls, (*e.g.*, steering), as requested by the software stack.
• Bug #32 is a data-related error. In one of the CARLA maps, the vector map mistakenly listed a dead end blocked by gas tanks as a valid lane. The lane was included in the path found by the global path planner in some scenarios, and the ego-vehicle ended up getting stuck behind the gas tanks blocking the path. Similarly in the real world, an autonomous vehicle could make inadequate path plans if the ground truth data, such as a map, is not up to date.

6.3 Feasibility of Bug Exploitation

It is feasible to reliably exploit all 17 Autoware bugs (except for two) and all 13 Behavior Agent bugs, adhering to the threat model presented in §3; controlling external inputs. Specifically, we evaluate the viability of launching object-based attacks or location-based attacks targeting the discovered bugs.

Object-based attacks. There are 11 bugs (#1, 12–15, 18, 19, 21, 25–27) that enable object-based attacks. To understand how easy it is to exploit the bugs in the real world, we run experiments on how sensitive each bug is to multiple variables (*i.e.*, potential requirements of the exploitation), including the color and shape (model) of the vehicle, location/trajectory of the controlled object, and the weather. If a bug requires many attributes for its exploitation, it essentially means that its exploitability is low. As shown in Table 4, only one attribute needs to be controlled for all bugs, except for bug #1, which requires two attributes to be controlled. Specifically, to exploit bug #1, an attacker can use an object of any color at any location near the path of the ego-vehicle as long as its height is lower than approximately 50 cm, which is commonplace. Bugs #13, 14, 26, and 27 are even easier to exploit; having any vehicle of any model and color, or pedestrian approach towards the ego-vehicle from behind or side is sufficient. The same set of attributes does not affect bug #12. However, after moving into the ego-vehicle's path, the object (vehicle or pedestrian) has to be located within one meter of the ego-vehicle's front bumper. For bug #15, the only relevant attribute is the location of the object. As illustrated in Figure 5, the adversarial object has to be placed within a certain range of distance from the ego-vehicle, which allows a window of 66 cm. While it seems tight, it is obviously viable, considering that the window is approximately a third of the width of a mid-sized sedan.

Location-based attacks. Bugs #2–11, 16, 17, 20, 22–24, 28–30 can be exploited by taking advantage of location-based attacks. Bugs #3–5 can be triggered at any location with traffic lights, stop signs, or speed limit signs, regardless of the weather condition, as they stem from software errors not being able to find matching tags for detected objects. Bugs #8 and 9 are triggered immediately when the planning layer receives the mission, if the goal position is unreachable or not aligned with the lane. An attacker may provide such adversarial destinations to the system through social engineering, *e.g.*, sharing a Google Maps link that sets the destination through navigation API. Bugs #2, 7, 16, 17, 28, and 29 require an attacker to

Table 4: Enumeration of the object types, the attributes that are irrelevant to the successful attack (✱, meaning that they can have arbitrary values), and the attributes that should be controlled.

Bug #	Object type			Irrelevant					Need to control
	O	P	V	C	L	S	T	W	
01	📦			✱			✱	✱	L: be close to the path, S: height < 50cm
12, 19		🚶	🚗	✱	✱	✱		✱	T: cut in from side to dist < 1m
13, 26		🚶	🚗	✱	✱	✱		✱	T: approach from behind
14, 27		🚶	🚗	✱	✱	✱		✱	T: approach from side
15	📦	🚶	🚗	✱		✱	✱	✱	L: located within 66cm range
18			🚗	✱		✱	✱	✱	L: located on sidewalk
21			🚗	✱	✱	✱		✱	T: drive at a similar speed alongside
25	📦	🚶	🚗	✱		✱	✱	✱	L: located on the cross lane of intersection

[Object types] O: object / P: pedestrian / V: vehicle
[Attributes] C: color / L: location / S: shape / T: trajectory / W: weather

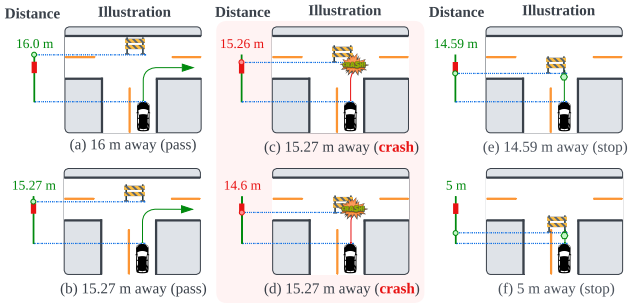


Figure 5: Testing the variants of bug #15 by changing the distance of the obstacle from the initial position of Autoware ego-vehicle. When the obstacle is at a moderate distance (14.6–15.26 m), *i.e.*, (c) and (d), the ego-vehicle initiates a turn and hits the object, even though it senses and perceives the existence of the object correctly.

lure the ego-vehicle to any location that has a certain property; any downward slope of an angle greater than 30 degrees that abruptly flattens at the end (bug #2), any location under a bridge that the vehicle has already passed over, *e.g.*, an underpass of an interchange (bug #7), any destination at a location the ego-vehicle can sufficiently accelerate before reaching it, *e.g.*, the end of a long straight road (bug #16), any 90-degree curve connecting the rightmost lanes, which can be observed at most three-way or four-way intersections, (bug #17), and puddles covering an area vehicle turns (bug #28, 29). Bug #6 and #30 happen at arbitrary curves. Notably, bugs #10 and #11 are the only bugs that require a specific location of the map, and thus can be harder to exploit in the real world.

Summary. All 30 bugs except for two (#10 and #11) have a wide window of exploitation in the input space that an adversary can easily control in the real world to cause safety-critical misbehaviors.

6.4 Comparison with AV-Fuzzer

AV-Fuzzer [52] is a state-of-the-art ADS testing approach that mutates the trajectory of two actor vehicles driving nearby, aiming to detect vehicle-to-vehicle collisions. It uses the longitudinal distance from the ego-vehicle to actor vehicles as a fitness function for the mutation to create scenarios with smaller distances. It detected five buggy scenarios: (1) hitting an overtaking vehicle, (2) hitting another vehicle while trying to cut in, (3) hitting a vehicle that cuts in, (4) rear-ending a suddenly braking vehicle, and (5) interpreting two adjacent vehicles as one and hitting one.

Quantitative comparison. DriveFuzz was able to automatically generate all five crash scenarios AV-Fuzzer found and successfully detected misbehaviors (bugs #12-14, 19, 21, 25-27). On the other hand, AV-Fuzzer is bound to miss 26 out of 34 (76%) bugs DriveFuzz found due to fundamental limitations in the design. We discuss the reasons in the following, referring to the latest source code of AV-Fuzzer[3]. First, the input space of AV-Fuzzer is a subset of DriveFuzz's driving scenarios. AV-Fuzzer divides a scenario into five time-slices (line 10 in `drive_experiment.py` and lines 11-34 in `Chromosome.py`), and randomly mutates the target speed and the maneuver (*e.g.*, go straight, change to the left lane, or change to the right lane) (lines 203-234 in `GeneticAlgorithm.py`) of *two* hard-coded actor vehicles (line 9 in `drive_experiment.py`), which always start driving at fixed positions (lines 180-181 in `simulation.py`). In contrast, DriveFuzz explores a multifaceted input space including the mission, weather, locations, and trajectories of an unbounded number of actor vehicles and/or pedestrians, and puddles. Second, DriveFuzz detects not only collisions (to vehicles, people, and objects), but also safety-critical traffic violations (*e.g.*, running red lights) with the driving test oracles. However, AV-Fuzzer only considers vehicle-to-vehicle collisions (lines 190-215 in `simulation.py`), which is a subset of the misbehaviors DriveFuzz detects.

Qualitative comparison. In addition to the size of the input space and the types of errors a fuzzer handles, the quality of fuzzing feedback is tightly coupled with the quality of bugs a fuzzer can detect. As we discussed in §4.5.2, the feedback engine of DriveFuzz generates a fine-grained feedback of the recklessness by referring to the physical vehicular states that is highly relevant to the targeted misbehavior. For example, when mutating the input scenario that triggered bug #29 in Table 3, DriveFuzz placed puddles at the locations that decreased the driving quality the most due to oversteering and hard acceleration, and could eventually cause a misbehavior. In the case of AV-Fuzzer, it only favors scenarios in which the ego-vehicle gets closer to the actor vehicles, without considering the physical states of the ADS. Unfortunately, merely reducing the vehicular distance is not sufficient to find scenarios (such as the one for bug #29) where no other vehicles are involved.

6.5 Correctness of Driving Test Oracles

The accuracy of misbehavior detection depends on the correctness of the driving test oracles DriveFuzz leverages. We evaluate it by injecting errors that cause the misbehavior that each oracle targets. Table 5 shows each misbehavior and corresponding errors that are injected to synthesize scenarios where each misbehavior must be observed. For example, to test the collision oracle, the input mutator is set to create a high-speed vehicle driving directly towards the ego-vehicle at 100 different locations. After injecting each error, we run DriveFuzz to check whether the intended misbehavior is detected or not from each mutated scenario. Except for the four rare false negatives caused by a known issue in CARLA's lane invasion sensor [18], the oracles never missed any misbehavior.

6.6 Correctness of Driving Quality Metrics

DriveFuzz analyzes the vehicle states to generate driving quality feedback by detecting vehicular events. To ensure that DriveFuzz

[3] https://github.com/cclinus/AV-Fuzzer/tree/4f67868/freeway

correctly implements the detection of each event, we tested the feedback engine under a few synthesized experiments that are designed to trigger the events. Due to space constraints, we show the correctness of detecting the two most complicated events: understeer and oversteer, which require correct implementation of fuzzy logic, and present the figures in Appendix C.

Understeer experiment. When understeer is triggered, a vehicle cannot turn in the direction it desires, as the frontal grip is lost. The situation can be contrived by placing a puddle at an intersection where the vehicle has to make a turn because the steering will not have any effect on turning the vehicle once it starts slipping (see Figure 10). The feedback engine successfully detected such events as shown in Figure 11, spotting the moments of understeer.

Oversteer experiment. If a vehicle with a non-zero yaw speed enters a section of a road with reduced friction, tires easily lose grip and cause the vehicle to oversteer. By synthesizing a scenario where the ego-vehicle diagonally enters a puddle as shown in Figure 12, we triggered oversteer and tested the feedback engine. As shown in Figure 13, the feedback engine reliably detected the oversteers.

6.7 Effectiveness of Driving Quality Feedback

By associating the likelihood of observing misbehaviors with low quality (*e.g.*, reckless or clumsy) driving, the driving quality feedback prevents the mutation engine from over-exploring less interesting (*i.e.*, hardly buggy) driving scenarios. As a result, it contributes to the effectiveness of DriveFuzz in revealing more bugs within a given time frame. To demonstrate this, we run two configurations of fuzzers; one with the driving quality feedback (*i.e.*, the proposed setting) and the other without the feedback to fuzz Autoware starting with the same seed scenario. As shown in Figure 6, DriveFuzz with the feedback found an average of 19 misbehaviors, which are caused by bugs #12–14 (Table 3) in different situations (note that the initial seed scenario was the one that revealed bug #13). Meanwhile, without any guidance, DriveFuzz blindly mutated scenarios and only discovered an average of 10 misbehaviors, showing a significant decline (-47%). The result substantiates the design choice of DriveFuzz that favoring the scenarios with lower driving quality results in a better chance of finding bugs.

6.8 Fuzzing Overhead

The total duration of one fuzzing round varies significantly depending on the length of a scenario and the existence of the bug since buggy scenarios would terminate early. In our experiments, the average throughput of DriveFuzz was 150 seconds per end-to-end execution. Figure 7 presents the breakdown of the average time spent by each module per fuzzing execution. The time required by DriveFuzz-specific modules (white boxes) including mutation engine, misbehavior detector, driving quality feedback engine, and logger, only accounted for **6%** of the total fuzzing time. The mutation time includes time spent for retries to ensure semantically correct scenarios (§4.2.3), where the number of retries ranged from zero to 2K times (in an extreme case) with an average of 300 retries.

This is negligible compared to the simulation overhead (black box), which dominates the overall fuzzing time (**94%**). Thus, employing a GPU with more computing power, or parallelizing the simulations to multiple GPUs can contribute to resolving the inevitable bottleneck. Moreover, although the throughput of DriveFuzz may seem low compared to traditional fuzzing approaches that feature a high fuzzing speed, the use of a driving simulator enables DriveFuzz to scale testing with significantly lower cost than physically testing autonomous vehicles (§4.3.2); we could detect all 34 bugs by running DriveFuzz for a week.

Table 5: Driving test oracles and the injected errors that trigger each misbehavior. 100 different scenarios are created and tested for each error (# TP: misbehavior was detected, # FN: oracle missed the misbehavior). We manually confirmed that there was no false alarm.

Misbehavior	Injected error	# TP	# FN
Collision	Have a vehicle rear-end the ego-vehicle	100	0
Speeding	Set target speed to above limit	100	0
Running red lights	Disable traffic light detection	100	0
Immobility	Disable control module	100	0
Lane invasion	Force steer left	96	4

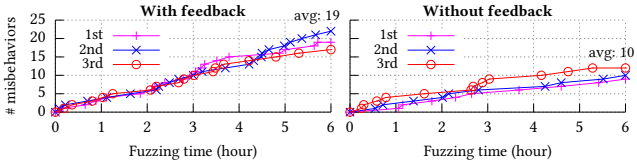


Figure 6: Number of misbehaviors observed while fuzzing Autoware for six hours with (left) and without (right) the driving quality feedback. Each configuration is repeated three times.

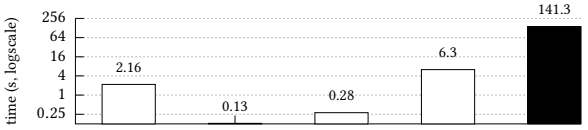


Figure 7: Breakdown of the fuzzing time per round.

7 RELATED WORK

Testing autonomous driving systems. Most existing approaches focus on the white-box testing of individual layers: sensing [15, 33, 42], perception [67, 82, 87], and planning [16, 63, 83]. For example, the series of works on the perception layer [67, 82, 87] tests the robustness of the neural network model with synthetically transformed camera images based on the model's activation patterns. PlanFuzz [83] tries to find denial of service vulnerabilities in the planning layer by introducing physical objects into driving scenes and guiding the input scenario generation based on the code execution that it monitors through instrumentation. Unlike these works, DriveFuzz considers a target ADS as a whole system rather than focusing on a specific layer or problem. This holistic approach allows us to find not only those bugs that individual layer testing covers, but also other types of bugs with *propagating* impacts across multiple layers that cause critical accidents. Besides, our approach does not require the source code, instrumentation, or domain knowledge of the target ADS in contrast to those white-box approaches.

A few testing works take a holistic approach similar to ours [35, 52]. The closest to our work is AV-Fuzzer [52]. AV-Fuzzer mutates the trajectory of nearby vehicles with an objective to find scenarios where the ego-vehicle gets too close to them. Although conceptually similar, its input dimension and the scope of safety violations are a small subset of what DriveFuzz considers, as evaluated in §6.4. Han *et al.* [35] propose an adversarial testing approach, which tests

autonomous vehicles under rather unrealistic test cases (*e.g.*, a static obstacle suddenly appearing and disappearing). This approach does not necessarily focus on the feasibility of exploiting the bugs from the attacker's perspective. In contrast, our approach focuses on generating semantically valid test cases that attackers can exploit.

Fremont *et al.* [31] tackle the testing problem from a different but complementary angle by applying a formal methods-based approach. They focus on generating test cases that will run on a real track based on the formal verification of driving scenarios, rather than finding bugs in ADSes.

Adversarial example attacks. Many existing works focus on finding adversarial attacks that deceive the machine learning model of the perception layer [13, 17, 24, 39, 57, 75, 77, 79]. These attacks input sensor data with carefully crafted perturbations to cause misclassification, such as camera images with a modified traffic sign, or spoofed LiDAR data. Similar to the testing approaches on individual layers, these works target a specific layer and problem; *i.e.*, the lack of robustness of machine learning model in the perception layer. Complementary to these works, the goal of DriveFuzz is finding vulnerabilities in any layer of an ADS regardless of their location.

Coverage-guided fuzzing. Many existing fuzzers are geared towards improving bug detection abilities across various domains. In previous studies, some focus on improving the code coverage feedback [2, 3, 86], while others retrieve more advanced information (*e.g.*, data flow) to handle systems in new domains or platforms (*e.g.*, drone control) [22, 23, 27, 29, 43, 46–48, 65, 70, 76]. Unfortunately, none of these approaches can be directly applied to ADSes as they are designed to find typical software bugs only (*e.g.*, memory safety violation), relying on obvious symptoms of program failures (*e.g.*, segmentation faults) and general code coverage to guide the input mutation. To address this limitation, DriveFuzz is designed specifically for holistically fuzzing ADSes leveraging new test oracles and quality metrics that focus on driving semantics and vehicle states.

8 DISCUSSION AND FUTURE WORK

Fidelity of simulation. Despite a potential gap between the simulated and real environments, the use of high-fidelity simulation brings the quality of test cases in close proximity to that of physical testing and significantly enhances the quality of automated ADS testing over existing methods. This is also demonstrated by the fact that DriveFuzz discovers 33 new ADS and simulator bugs in the corner case driving scenarios that existing testing methods could not attempt to generate. 10 (out of 33 reported) bugs have been acknowledged by the ADS developers, and most are readily exploitable with concrete attacks by an adversary as we demonstrate in §6.3. More importantly, it not only enables a full degree of automation, but also provides other practical benefits, such as low cost and safety of testing, in comparison with physical testing with real vehicles. It is also supported by the fact that major ADS vendors rely heavily on simulators to develop and test their systems before physical testing [37, 84]. In our future work, we plan to reproduce the ADS issues in this paper with a real autonomous vehicle.

Definition of good/bad behaviors. Defining good and bad behaviors is challenging as it is a subjective matter that depends on the circumstance and the intent of the behaviors. For example, Crossing a yellow line at a two-lane expressway is considered an infraction, while it is circumstantially benign if it is to avoid a collision with an object, *e.g.*, a vehicle blocking the road. In light of this, we made the misbehavior oracles configurable so that they can be adjusted per target. In addition, when misbehaviors are detected, DriveFuzz generates detailed reports with all sensor data including the camera feed, so that users can further reconfigure and fine-tune the oracles.

Extensibility of DriveFuzz. DriveFuzz is designed with an extensibility in mind; the mutation engine, misbehavior detector, and driving quality feedback engine are *generic*, operating independently of the ADS under test. In addition, the test executor, which bridges the ADS with the simulator and DriveFuzz, supports ROS to maximize the compatibility with the ROS-based systems. This is showcased by testing a ROS-based system, Autoware in §6.

Limitation. Our driving quality-based feedback directs DriveFuzz to scenarios where an ADS performs unsafe maneuvers. We have proven that such feedback is effective in triggering misbehavior that we target. However, similar to most feedback-driven fuzzers that register a specific fitness function as a feedback, DriveFuzz can have a local optima problem [54], *i.e.*, reaching a local optimum in the search space as a result of feedback guidance, and ends up missing other potential bugs that are less related to the feedback.

In addition, there may exist attacks that do not affect the driving quality score but still cause misbehaviors. For example, if an adversary draws a fake curved lane on a straight road and misleads an ADS to invade a sidewalk, the driving quality score can still be good if an ego-vehicle seamlessly follows the fake lane, while the resulting circumstance is a lane invasion. As a mitigation, we can extend and diversify the driving quality score metrics, *e.g.*, considering the adherence to the original plan, to deal with the bugs that are not necessarily coupled with clumsy driving behaviors.

9 CONCLUSION

This paper presents DriveFuzz, an end-to-end fuzzing framework designed to find bugs in all layers of ADSes that are readily exploitable by attackers. DriveFuzz detects bugs by (1) automatically generating and mutating high-fidelity driving scenarios, (2) checking for safety-critical misbehaviors using driving test oracles contrived by studying real-world traffic rules, and (3) measuring our novel driving quality score by inspecting the vehicle states and using it as feedback to guide the mutation engine towards buggy scenarios. DriveFuzz has found 17 new bugs in Autoware, 13 bugs in Behavior Agent, and three bugs in the simulator, showing that it can discover bugs in all layers of the tested system. Our study shows that the bugs we found can be triggered by only controlling legitimate inputs and cause devastating vehicle accidents.

ACKNOWLEDGMENT

We thank the anonymous reviewers, and our shepherd, Ziming Zhao, for their insightful feedback. This work was supported in part by the University of Texas at Dallas Office of Research through the NFRS program, Texas A&M Engineering Experiment Station on behalf of its SecureAmerica Institute, Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2022-0-00745, The Development of Ransomware Attack Source Identification and Analysis Technology), and a gift from Cisco Systems.

A SEED GENERATION AND VERIFICATION

We present our seed generation and verification process in Algorithm 2, which can be performed without substantial domain knowledge of a particular ADS.

Seed construction. Our seed scenarios are created on top of the maps CARLA provides (https://carla.readthedocs.io/en/0.9.10/core_map/), which have various road components (*e.g.*, lanes, junctions, traffic lights) along with static objects such as trees and buildings. Specifically, we first selected five common components of the road system; urban street, highway, interchange, intersection, and roundabout. For each of the five road components, we select $n_s = 8$ missions, in which the mission (line 10) consisting of the initial position (p_i) and the goal position (p_g) requires either driving within or around the component. Other than having the pre-selected map and mission assigned, each seed driving scenario is in a clean slate, having no other actors (line 11) or puddles (line 12) with sunny weather (line 13).

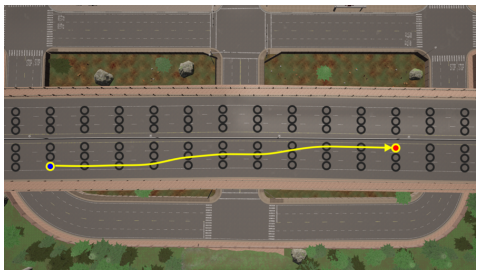
Checking the validity of seeds. The legitimacy of all 40 seeds is verified (line 18) prior to fuzzing by dry-running the target ADS with the seed scenario (line 19) and confirming that it successfully completes the mission (line 20).

Examples. Figure 8 shows concrete examples of the seed scenarios generated and verified through the aforementioned procedure. The black circles indicate the predefined valid waypoints in the CARLA map (please note that the number and density of the waypoints shown are greatly reduced for an illustration purpose). A random waypoint can be retrieved by `get_random_position_near()` function in Algorithm 2. The yellow arrow connecting the blue (initial position, p_i) with the red circles (goal position, p_g) indicates the mission assigned to each seed scenario.

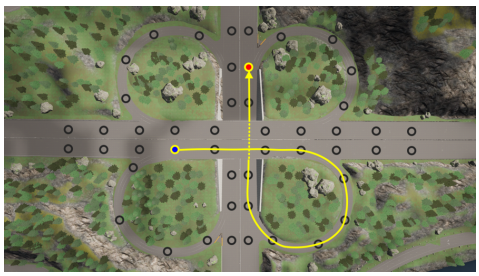
Algorithm 2: Seed generation and verification

```
   Input  : C - a set of road components
            = {urban, highway, interchange, intersection, roundabout},
            n_s - # scenarios per road component
   Output : S - a set of seed scenarios
1  procedure generate_seed()
2      foreach comp ∈ C do
3          for i ← 1 to n_s do
4              while true do
5                  p_i ← 0 // initial position
6                  p_g ← 0 // goal position
7                  while p_i == p_g do
8                      p_i ← get_random_position_near(comp)
9                      p_g ← get_random_position_near(comp)
10                 mission ← {p_i, p_g}
11                 actors ← ∅
12                 puddles ← ∅
13                 weather ← sunny
14                 seed ←
                     init_scenario(mission, actors, puddles, weather)
15                 if verify_scenario{seed} == success then
16                     break
17             S ← S ∪ seed
18 procedure verify_scenario(s)
19     states ← executor.simulate(s)
20     if detector.check_misbehavior(states) == false then
21         return success // seed mission successfully completed
22     else
23         return retry // seed mission failed
```

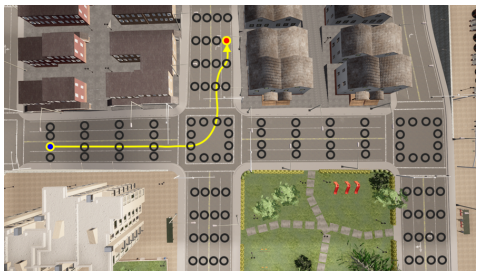
(a) Example seed scenario at urban streets



(b) Example seed scenario at a highway



(c) Example seed scenario at a cloverleaf interchange



(d) Example seed scenario at an intersection

(e) Example seed scenario at a roundabout

Figure 8: Example seed scenario at each road component.

B ON THE EFFECTIVENESS OF CODE COVERAGE-BASED METRICS

As we discuss in §4.5, the coverage-based feedback of traditional grey-box fuzzers (*e.g.*, AFL [86]) is less effective in estimating the effectiveness of test driving scenarios when testing a distributed and stateful system, because the behavior of such system is *dominantly driven by the data* being communicated between distributed nodes, rather than the control flow.

We demonstrate this in Figure 9, which shows the code coverage measured using the `gcov` coverage test program while fuzzing Autoware for six hours. Since AFL cannot be directly applied to testing Autoware, we measured the code coverage while fuzzing with DriveFuzz. Even though the code coverage was quickly saturated at approximately 32%, the ego-vehicle *showed a wide variety of behaviors* driving in the mutated scenarios after it reached the saturation point, *e.g.*, navigating through different parts of the map, and exhibited multiple misbehaviors including collisions and speed limit violations. The uncovered code included unused and irrelevant portions of Autoware, such as visualization, graphic user interface (GUI), and unused modules (*e.g.*, alternative controllers). This substantiates our claim that code coverage is not an effective metric to approximate the test coverage in finding bugs of ADSes.

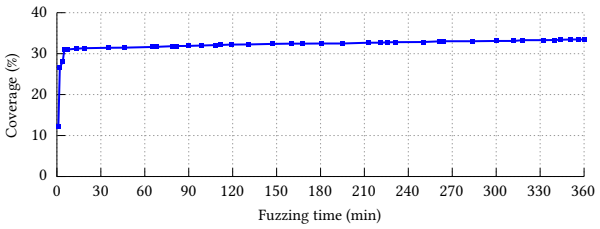
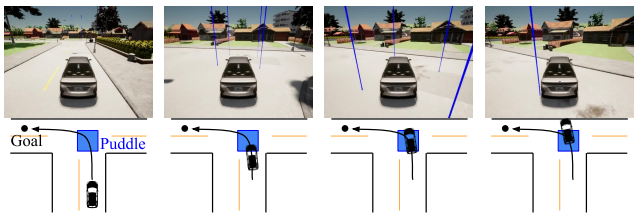


Figure 9: Change of code coverage while fuzzing Autoware for six hours with DriveFuzz.

C FIGURES FOR DRIVING QUALITY METRICS CORRECTNESS EVALUATION

Figure 10~13 show the correctness of driving quality metrics. They are omitted in §6.6 due to the space.



(a) Frame 1 (b) Frame 150 (c) Frame 178 (d) Frame 220

Figure 10: Simulated scenario of the understeer experiment. At the frame 150, the vehicle enters the puddle (the blue box) while turning left to get to the destination. At the frame 178, the vehicle is sliding out to the right even though it tries to turn left, and at the frame 220, the vehicle exits the puddle right before invading the sidewalk.

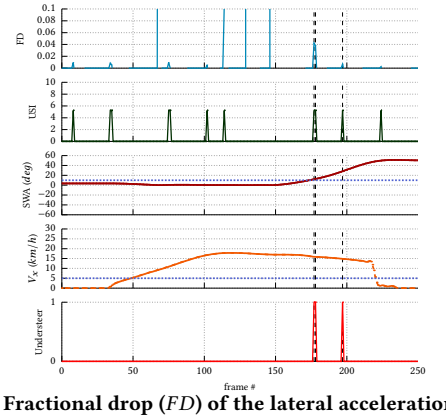
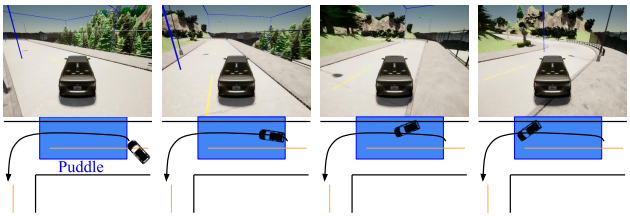


Figure 11: Fractional drop (FD) of the lateral acceleration (A_y), understeer indicator (USI), steering wheel angle (SWA), longitudinal speed (V_x), and final understeer detection throughout the scenario shown in Figure 10. The blue dotted lines shown in the SWA (10 deg) and V_x (5 km/h) graphs are activation thresholds; USI combined with SWA and V_x above thresholds lead to final understeer calls at the frames 177, 178, and 197 (marked by dashed vertical lines).



(a) Frame 1 (b) Frame 58 (c) Frame 108 (d) Frame 185

Figure 12: Simulated scenario of the oversteer experiment. At the frame 58, the vehicle enters the puddle (the blue box) with a non-zero yaw speed (V_x). At the frame 108, the rear end of the vehicle rotates counter-clockwise even though the steering amount small. The rotation stops, but the vehicle continues to slide until it reaches the end of the puddle at the frame 185.

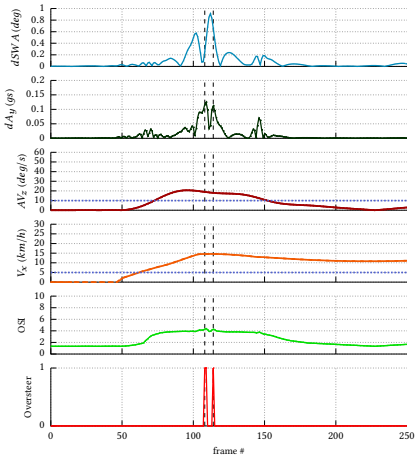


Figure 13: Difference of lightly- and heavily-filtered steering wheel angles ($dSWA$), difference of filtered lateral acceleration (dA_y), yaw speed (AV_z), longitudinal speed (V_x), oversteer indicator (OSI), and final oversteer detected in the scenario shown in Figure 12. When the vehicle starts to oversteer at the frame 108 as the rear-end loses grip and slips out, and when the state exacerbates at the frame 114, the oversteer events are detected (marked by dashed vertical lines).

REFERENCES

[1] 2008. *National Motor Vehicle Crash Causation Survey: Report to Congress.* Technical Report. National Highway Traffic Safety Administration, United States Department of Transportation.

[2] 2017. LibFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

[3] 2018. syzkaller - kernel fuzzer. https://github.com/google/syzkaller.

[4] 2019. Baidu Apollo: An Open Autonomous Driving Platform. http://apollo.auto/.

[5] 2020. The Autoware Foundation. https://www.autoware.org/.

[6] Michael Aeberhard, Thomas Kühbeck, Bernhard Seidl, M Friedl, J Thomas, and O Scheickl. 2015. Automated Driving with ROS at BMW. *ROSCon 2015 Hamburg, Germany* (2015).

[7] Allstate. 2021. Drivewise from Allstate. https://www.allstate.com/drive-wise.aspx.

[8] Jeffery R Anderson and E Harry Law. 2011. Fuzzy Logic Approach to Vehicle Stability Control of Oversteer. *SAE International Journal of Passenger Cars-Mechanical Systems* 4 (2011), 241–250.

[9] Associated Press News. 2020. 3 Crashes, 3 Deaths Raise Questions About Tesla's Autopilot. https://apnews.com/ca5e62255bb87bf1b151f9bf075aaadf.

[10] Association for Standardization of Automation and Measuring Systems. 2021. ASAM OpenDRIVE. https://www.asam.net/standards/detail/opendrive/.

[11] BBC News. 2019. Tesla Model 3: Autopilot Engaged during Fatal Crash. https://www.bbc.com/news/technology-48308852.

[12] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS 2017).*

[13] Adith Boloor, Karthik Garimella, Xin He, Christopher Gill, Yevgeniy Vorobeychik, and Xuan Zhang. 2020. Attacking Vision-Based Perception in End-to-End Autonomous Driving Models. *Journal of Systems Architecture* 110 (2020).

[14] Assaf Botzer, Oren Musicant, and Yaniv Mama. 2019. Relationship between Hazard-perception-test Scores and Proportion of Hard-braking Events during On-Road Driving – An investigation Using a Range of Thresholds for Hard-braking. *Accident Analysis & Prevention* 132 (2019).

[15] Alberto Broggi, Michele Buzzoni, Stefano Debattisti, Paolo Grisleri, Maria Chiara Laghi, Paolo Medici, and Pietro Versari. 2013. Extensive Tests of Autonomous Driving Technologies. *IEEE Transactions on Intelligent Transportation Systems* 14, 3 (2013), 1403–1415.

[16] Alessandro Calò, Paolo Arcaini, Shaukat Ali, Florian Hauer, and Fuyuki Ishikawa. 2020. Generating Avoidable Collision Scenarios for Testing Autonomous Driving Systems. In *Proceedings of the IEEE 13th International Conference on Software Testing, Validation and Verification (ICST 2020).*

[17] Yulong Cao, Chaowei Xiao, Benjamin Cyr, Yimeng Zhou, Won Park, Sara Rampazzi, Qi Alfred Chen, Kevin Fu, and Z Morley Mao. 2019. Adversarial Sensor Attack on LiDAR-Based Perception in Autonomous Driving. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS 2019).*

[18] CARLA simulator. 2021. Lane invasion detector. https://carla.readthedocs.io/en/0.9.11/.

[19] Marco Ceccarelli. 2004. Fundamentals of the mechanics of robots. In *Fundamentals of Mechanics of Robotic Manipulation.* Springer, 73–240.

[20] Ching-Yao Chan. 2017. Advancements, Prospects, and Impacts of Automated Driving Systems. *International Journal of Transportation Science and Technology* 6, 3 (2017), 208–216.

[21] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. 2015. DeepDriving: Learning Affordance for Direct Perception in Autonomous Driving. In *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV 2015).*

[22] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. 2018. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS 2018).*

[23] Yaohui Chen, Dongliang Mu, Jun Xu, Zhichuang Sun, Wenbo Shen, Xinyu Xing, Long Lu, and Bing Mao. 2019. PTrix: Efficient Hardware-Assisted Fuzzing for COTS Binary. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security (ASIACCS 2019).*

[24] Alesia Chernikova, Alina Oprea, Cristina Nita-Rotaru, and BaekGyu Kim. 2019. Are Self-Driving Cars Secure? Evasion Attacks against Deep Neural Networks for Steering Angle Prediction. In *Proceedings of the 2019 IEEE Security and Privacy Workshops (SPW 2019).*

[25] SAE On-Road Automated Vehicle Standards Committee. 2014. *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles.* Technical Report.

[26] Thomas A. Dingus, Sheila G. Klauer, Vicki Lewis Neale, Andy Petersen, Suzanne E. Lee, Jeremy Sudweeks, Miguel A. Perez, Jonathan Hankey, David Ramsey, Santosh Gupta, C. Bucher, Zachary Doerzaph, J. Jermeland, and Ronald Knipling. 2006. *The 100-car Naturalistic Driving Study, Phase II-results of the 100-car Field Experiment.* Technical Report. National Highway Traffic Safety Administration, United States Department of Transportation.

[27] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, and Yan Shoshitaishvili. 2021. Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases. In *Proceedings of the 28th Network and Distributed System Security Symposium (NDSS 2021).*

[28] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. 2017. CARLA: An Open Urban Driving Simulator. In *Proceedings of the 1st Annual Conference on Robot Learning (CoRL 2017).*

[29] Paul Fiterau-Brostean, Bengt Jonsson, Robert Merget, Joeri de Ruiter, Konstantinos Sagonas, and Juraj Somorovsky. 2020. Analysis of DTLS Implementations Using Protocol State Fuzzing. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security 2020).*

[30] McKinsey Center for Future Mobility. 2019. The Future of Mobility Is at Our Doorstep. (2019).

[31] Daniel J. Fremont, Edward Kim, Yash Vardhan Pant, Sanjit A. Seshia, Atul Acharya, Xantha Bruso, Paul Wells, Steve Lemke, Qiang Lu, and Shalin Mehta. 2020. Formal Scenario-Based Testing of Autonomous Vehicles: From Simulation to the Real World. In *In Proceedings of the 2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC 2020).*

[32] Joshua Garcia, Yang Feng, Junjie Shen, Sumaya Almanee, Yuan Xia, and Qi Alfred Chen. 2020. A Comprehensive Study of Autonomous Vehicle Bugs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE 2020).*

[33] Andreas Geiger, Philip Lenz, and Raquel Urtasun. 2012. Are We Ready for Autonomous Driving? The Kitti Vision Benchmark Suite. In *Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2012).*

[34] Bart LJ Gysen, Jeroen LG Janssen, Johannes JH Paulides, and Elena A. Lomonova. 2009. Design Aspects of an Active Electromagnetic Suspension System for Automotive Applications. *IEEE Transactions on Industry Applications* 45, 5 (2009), 1589–1597.

[35] Jia Cheng Han and Zhi Quan Zhou. 2020. Metamorphic Fuzz Testing of Autonomous Vehicles. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW 2020).*

[36] Andrew Hill, Mark S. Horswill, John Whiting, and Marcus O. Watson. 2019. Computer-Based Hazard Perception Test Scores Are Associated with the Frequency of Heavy Braking in Everyday Driving. *Accident Analysis & Prevention* 122 (2019), 207–214.

[37] WuLing Huang, Kunfeng Wang, Yisheng Lv, and FengHua Zhu. 2016. Autonomous Vehicles Testing Methods Review. In *Proceedings of the IEEE 19th International Conference on Intelligent Transportation Systems (ITSC 2016).*

[38] Seok-Hwan Jang, Tong-Jin Park, and Chang-Soo Han. 2003. A Control of Vehicle Using Steer-by-Wire System with Hardware-in-the-Loop Simulation System. In *Proceedings of the 2003 IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM 2003).*

[39] Pengfei Jing, Qiyi Tang, Yuefeng Du, Lei Xue, Xiapu Luo, Ting Wang, Sen Nie, and Shi Wu. 2021. Too Good to Be Safe: Tricking Lane Detection in Autonomous Driving with Crafted Perturbations. (2021).

[40] Kichun Jo, Junsoo Kim, Dongchul Kim, Chulhoon Jang, and Myoungho Sunwoo. 2014. Development of autonomous car—Part I: Distributed system architecture and development process. *IEEE Transactions on Industrial Electronics* 61, 12 (2014), 7131–7140.

[41] Kichun Jo, Junsoo Kim, Dongchul Kim, Chulhoon Jang, and Myoungho Sunwoo. 2015. Development of autonomous car—Part II: A case study on the implementation of an autonomous driving system based on distributed architecture. *IEEE Transactions on Industrial Electronics* 62, 8 (2015), 5119–5132.

[42] Maria Jokela, Matti Kutila, and Pasi Pyykönen. 2019. Testing and Validation of Automotive Point-cloud Sensors in Adverse Weather Conditions. *Applied Sciences* 9, 11 (2019).

[43] Imtiaz Karim, Fabrizio Cicala, Syed Rafiul Hussain, Omar Chowdhury, and Elisa Bertino. 2020. ATFuzzer: Dynamic Analysis Framework of AT Interface for Android Smartphones. *Digital Threats: Research and Practice* 1, 4 (2020).

[44] Shinpei Kato, Eijiro Takeuchi, Yoshio Ishiguro, Yoshiki Ninomiya, Kazuya Takeda, and Tsuyoshi Hamada. 2015. An Open Approach to Autonomous Vehicles. *IEEE Micro* 35, 6 (2015), 60–68.

[45] Shinpei Kato, Shota Tokunaga, Yuya Maruyama, Seiya Maeda, Manato Hirabayashi, Yuki Kitsukawa, Abraham Monrroy, Tomohito Ando, Yusuke Fujii, and Takuya Azumi. 2018. Autoware on Board: Enabling Autonomous Vehicles with Embedded Systems. In *Proceedings of the ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS 2018).*

[46] Hongil Kim, Jiho Lee, Eunkyu Lee, and Yongdae Kim. 2019. Touching the Untouchables: Dynamic Security Analysis of the LTE Control Plane. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy (S&P 2019).*

[47] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. 2019. Finding Semantic Bugs in File Systems with an Extensible Fuzzing Framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles.* 147–161.

[48] Taegyu Kim, Chung Hwan Kim, Junghwan Rhee, Fan Fei, Zhan Tu, Gregory Walkup, Xiangyu Zhang, Xinyan Deng, and Dongyan Xu. 2019. RVFuzzer: Finding Input Validation Bugs in Robotic Vehicles Through Control-Guided Testing. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security 2019)*.

[49] Sheila G. Klauer, Thomas A. Dingus, Vicki L. Neale, Jeremy D. Sudweeks, and David J. Ramsey. 2009. *Comparing Real-world Behaviors of Drivers with High Versus Low Rates of Crashes and Near Crashes.* Technical Report.

[50] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS 2018)*.

[51] Philip Koopman and Michael Wagner. 2016. Challenges in Autonomous Vehicle Testing and Validation. *SAE International Journal of Transportation Safety* 4, 1 (2016), 15–24.

[52] Guanpeng Li, Yiran Li, Saurabh Jha, Timothy Tsai, Michael Sullivan, Siva Kumar Sastry Hari, Zbigniew Kalbarczyk, and Ravishankar Iyer. 2020. AV-FUZZER: Finding Safety Violations in Autonomous Driving Systems. In *Proceedings of the IEEE 31st International Symposium on Software Reliability Engineering (ISSRE 2020)*.

[53] EK Liebemann, K Meder, J Schuh, and G Nenninger. 2004. Safety and performance enhancement: The Bosch electronic stability control (ESP). *SAE Paper* 20004, 2004 (2004), 21–0060.

[54] Valentin JM Manès, Soomin Kim, and Sang Kil Cha. 2020. Ankou: Guiding Grey-box Fuzzing Towards Combinatorial Difference. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1024–1036.

[55] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Transactions on Software Engineering* (2019).

[56] Wassim G Najm, John D Smith, and Mikio Yanagisawa. 2007. *Pre-Crash Scenario Typology for Crash Avoidance Research.* Technical Report. National Highway Traffic Safety Administration, United States Department of Transportation.

[57] Ben Nassi, Dudi Nassi, Raz Ben-Netanel, Yisroel Mirsky, Oleg Drokin, and Yuval Elovici. 2020. Phantom of the ADAS: Phantom Attacks on Driver-Assistance Systems. (2020).

[58] National Highway Traffic Safety Administration (NHTSA). 2018. Traffic Safety Facts 2018 Data: Speeding. https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/812932.

[59] BBC News. 2016. Google Self-driving Car Hits a Bus. https://www.bbc.com/news/technology-35692845.

[60] BBC News. 2016. Uber in Fatal Crash Had Safety Flaws Say US Investigators. https://www.bbc.com/news/business-50312340.

[61] Vilém Novák, Irina Perfilieva, and Jiri Mockor. 2012. *Mathematical Principles of Fuzzy Logic.* Vol. 517. Springer Science & Business Media.

[62] Takashi Ogawa and Kiyokazu Takagi. 2006. Lane Recognition using On-Vehicle LiDAR. In *Proceedings of the 2006 IEEE Intelligent Vehicles Symposium (IV 2006)*.

[63] Hiroki Ohta, Naoki Akai, Eijiro Takeuchi, Shinpei Kato, and Masato Edahiro. 2016. Pure Pursuit Revisited: Field Testing of Autonomous Vehicles in Urban Areas. In *Proceedings of the IEEE 4th International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA 2016)*.

[64] National Committee on Uniform Traffic Laws and Ordinances. 1972. *Traffic Laws Annotated.* National Committee on Uniform Traffic Laws and Ordinances.

[65] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. Parmesan: Sanitizer-Guided Greybox Fuzzing. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security 2020)*.

[66] Chinmay Pandit. 2013. *A Model-Free Approach to Vehicle Stability Control.* Master's thesis. Clemson University.

[67] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP 2017)*.

[68] Progressive. 2021. Snapshot Rewards You for Good Driving. https://www.progressive.com/auto/discounts/snapshot/.

[69] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. 2009. ROS: An Open-Source Robot Operating System. In *ICRA 2009 Workshop on Open Source Software in Robotics.*

[70] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *Proceedings of the 24th Network and Distributed System Security Symposium (NDSS 2017)*.

[71] Guodong Rong, Byung Hyun Shin, Hadi Tabatabaee, Qiang Lu, Steve Lemke, Mārtiņš Možeiko, Eric Boise, Geehoon Uhm, Mark Gerow, Shalin Mehta, Eugene Agafonov, Tae Hyung Kim, Eric Sterner, Keunhae Ushiroda, Michael Reyes, Dmitry Zelenkovsky, and Seonman Kim. 2020. LGSVL Simulator: A High Fidelity Simulator for Autonomous Driving. In *Proceedings of the IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC 2020)*.

[72] Root Insurance. 2021. Test Drive and Save. https://www.joinroot.com/test-drive/.

[73] Young-Woo Seo and Ragunathan Rajkumar. 2014. Tracking and Estimation of Ego-Vehicle's State for Lateral Localization. In *Proceedings of the 17th International IEEE Conference on Intelligent Transportation Systems (ITSC 2014)*.

[74] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. 2017. AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles. In *Proceedings of the 11th Conference on Field and Service Robotics (FSR 2017)*.

[75] Junjie Shen, Jun Yeon Won, Zeyuan Chen, and Qi Alfred Chen. 2020. Drift with Devil: Security of Multi-Sensor Fusion based Localization in High-Level Autonomous Driving under GPS Spoofing. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security 2020)*.

[76] Zisis Sialveras and Nikolaos Naziridis. 2015. Introducing Choronzon: An Approach at Knowledge-Based Evolutionary Fuzzing. *Proceedings of ZeroNights 2015.*

[77] Dawn Song, Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Florian Tramèr, Atul Prakash, and Tadayoshi Kohno. 2018. Physical Adversarial Examples for Object Detectors. In *Proceedings of the 12th USENIX Workshop on Offensive Technologies (WOOT 2018)*.

[78] State Farm. 2021. Drive Safe & Save Program. https://www.statefarm.com/insurance/auto/discounts/drive-safe-save.

[79] Jiachen Sun, Yulong Cao, Qi Alfred Chen, and Z Morley Mao. 2020. Towards Robust LiDAR-Based Perception in Autonomous Driving: General Black-Box Adversarial Sensor Attack and Countermeasures. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security 2020)*.

[80] The Autoware Foundation. 2021. Autoware Story. https://www.autoware.org/visionandmission.

[81] The Mercury News. 2018. Tesla: Autopilot Was On During Deadly Mountain View Crash. https://www.mercurynews.com/2018/03/30/tesla-autopilot-was-on-during-deadly-mountain-view-crash/.

[82] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: Automated Testing of Deep-Neural-Network-driven Autonomous Cars. In *Proceedings of the 40th International Conference on Software Engineering (ICSE 2018)*.

[83] Ziwen Wan, Junjie Shen, Jalen Chuang, Xin Xia, Joshua Garcia, Jiaqi Ma, and Qi Alfred Chen. 2022. Too Afraid to Drive: Systematic Discovery of Semantic DoS Vulnerability in Autonomous Driving Planning under Physical-World Attacks. In *Network and Distributed System Security (NDSS) Symposium, 2022.*

[84] Waymo. 2020. Off road, but not offline: How simulation helps advance our Waymo Driver. https://blog.waymo.com/2020/04/off-road-but-not-offline--simulation27.html.

[85] Lotfi A Zadeh. 1988. Fuzzy Logic. *Computer* 21, 4 (1988), 83–93.

[86] Michal Zalewski. 2014. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl.

[87] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. DeepRoad: GAN-based Metamorphic Testing and Input Validation Framework for Autonomous Driving Systems. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2018)*.