Neural Network Guided Evolutionary Fuzzing for Finding Traffic Violations of Autonomous Vehicles

Ziyuan Zhong, Gail Kaiser, *Senior Member, IEEE*, and Baishakhi Ray, *Member, IEEE*

Abstract—Self-driving cars and trucks, autonomous vehicles (AVs), should not be accepted by regulatory bodies and the public until they have much higher confidence in their safety and reliability — which can most practically and convincingly be achieved by testing. But existing testing methods are inadequate for checking the end-to-end behaviors of AV controllers against complex, real-world corner cases involving interactions with multiple independent agents such as pedestrians and human-driven vehicles. While test-driving AVs on streets and highways fails to capture many rare events, existing simulation-based testing methods mainly focus on simple scenarios and do not scale well for complex driving situations that require sophisticated awareness of the surroundings. To address these limitations, we propose a new fuzz testing technique, called *AutoFuzz*, which can leverage widely-used AV simulators' API grammars to generate semantically and temporally valid complex driving scenarios (sequences of scenes). To efficiently search for traffic violations-inducing scenarios in a large search space, we propose a constrained neural network (NN) evolutionary search method to optimize *AutoFuzz*. Evaluation of our prototype on one state-of-the-art learning-based controller, two rule-based controllers, and one industrial-grade controller in five scenarios shows that *AutoFuzz* efficiently finds hundreds of traffic violationsin high-fidelity simulation environments. For each scenario, *AutoFuzz* can find on average 10-39% more unique traffic violationsthan the best-performing baseline method. Further, fine-tuning the learning-based controller with the traffic violationsfound by *AutoFuzz* successfully reduced the traffic violationsfound in the new version of the AV controller software.

Index Terms—Search-based software engineering, evolutionary algorithms, neural networks, software testing, test generation, autonomous vehicles

1 Introduction

The rapid growth of autonomous driving technologies has made self-driving cars around the corner. As of June 2021, there are 55 autonomous vehicle (AV) companies actively testing self-driving cars on public roads in California [1]. However, the safety of these cars remains a significant concern, undermining wide deployment — there were 43 reported collisions involving self-driving cars in 2020 alone that resulted in property damage, bodily injury, or death [2]. Before mass adoption of AVfor our day-to-day transportation, it is thus imperative to conduct comprehensive testing to improve their safety and reliability.

However, real-world testing (e.g., monitoring an AV on a regular road) is extremely expensive and may fail to test against realistic variations of corner cases. Simulation-based testing is a popular and practical alternative [3], [4], [5], [6]. In a simulated environment, the main AV software, known as the *ego car controller*, receives multi-dimensional inputs from various sensors (e.g., Cameras, LiDAR, Radar, *etc.*) and processes the sensors' information to drive the car.

A good simulation-based testing framework should test the ego car controller by simulating challenging real-life situations — especially the ones that emulate real-world violations made by human drivers that lead to crashes, such as those shown in Table 1. These crash scenarios are rather involved, e.g., a leading car suddenly stopped to avoid a pedestrian and got hit by a following vehicle. However, simulating such involved crash scenarios is non-trivial, especially because the ego car can interact with its surroundings (e.g., driving path, weather, stationery, and moving agents, etc.) in an exponential number of ways. Yet, simulating *some* crash-inducing scenarios, even in this large space, is not so difficult—for example, one can simply place a stationary object on the ego car's path to simulate a crash. Further, many traffic violationscan be reported with slight variations of essentially the same situation (e.g., changing a never seen object's color). Thus one of the requirements for a successful simulation-based testing framework is to simulate scenarios that can lead to many *diverse* violations.

For traditional software, fuzz testing (a.k.a. fuzzing) [8], [9] is a popular way to find diverse bugs by navigating large search spaces. At a high level, fuzzing mutates existing test cases to generate new tests with an objective to discover new bugs. However, incorporating fuzzing into simulation testing of AVis not straightforward, as the test inputs (i.e., driving scenarios in our case) have many features and inter-dependencies, and random mutations of arbitrary features will lead to semantically incorrect scenarios. Although the simulator will eventually reject such inputs, the computational effort on generating and validating these invalid test cases

• *The authors are with the Department of Computer Science, Columbia University, New York, NY 10025 USA. E-mail: ziyuan.zhong@columbia.edu, {kaiser, rayb}@cs.columbia.edu.*

Manuscript received 9 December 2021; revised 22 July 2022; accepted 24 July 2022. Date of publication 1 August 2022; date of current version 18 April 2023.
This work was supported in part by NSF under Grants CCF-1845893, CCF-2107405, IIS-2221943, and CCF-1815494, in part by DARPA and NIWC-Pacific under Grant N66001-21-C-4018, and in part by National Instruments. (Corresponding author: Ziyuan Zhong.)
Recommended for acceptance by P. Pelliccione.
This article has supplementary downloadable material available at https://doi.org/10.1109/TSE.2022.3195640, provided by the authors.
Digital Object Identifier no. 10.1109/TSE.2022.3195640

TABLE 1
Dominant Scenarios Leading to Car Crashes as Per National Highway Traffic Safety Administration (NHTSA) report [7]

Crash Scenario	# Per Year	Economic Cost	Years Lost
A leading vehicle stopped	975 k	$15,388 m	240 k
Vehicle lost control without taking any action	529 k	$15,796 m	478 k
Vehicle(s) Turning at Non-Signalized Junctions	435 k	$7343 m	138 k
A leading vehicle decelerating	428 k	$6390 m	100 k
Vehicle drove off road without taking any action	334 k	$9005 m	270k
Straight Crossing Paths at Non-Signalized Junctions	264 k	$7290 m	174 k

Without taking any action' means the vehicle is going straight or negotiating a curve than explicitly making turns / changing lanes / leaving a parking position.

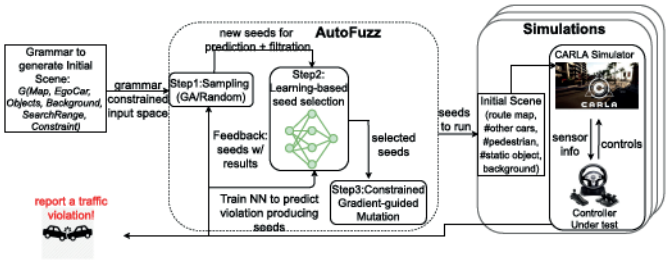


Fig. 1. *AutoFuzz* high-level overview.

will waste a large portion of the testing budget. Thus, each generated scene and sequence of scenes (a scenario consists of a sequence of scenes) should be *semantically correct* as well as triggering *diverse* traffic violations.

Our Approach. We address these challenges by designing a grammar-guided learning-based fuzzer, called *AutoFuzz* (Fig. 1). A self-driving car simulator takes some valid initial scene configuration as input (consisting of: road map; starting position and destination of the ego car; initial locations, directions, and velocities of other cars and pedestrians; *etc.*) and starts the simulation with the initial scene to generate a series of semantically valid consecutive scenes in the constrained driving environment. For initial scene generation, *AutoFuzz* leverages the API grammar provided by the simulator and fuzzes the grammar-constrained input space, treating the simulator as black-box (Section 4). In particular, *AutoFuzz* runs in an evolutionary fuzzing setting where it is optimized to generate test input that the target simulator uses to initiate a scenario, running the ego car through corresponding time steps such that it may lead to a traffic violation. However, if we optimize the search to only find violation-producing inputs (i.e., binary objective), it will be challenging to converge in a sparse space. Instead, following previous work on AV testing [6], [10], [11], [12], we formulate the fuzzing process as a smooth multi-objective search that guides the ego car to the point of interest.

To quantify the notion of traffic violationdiversity, we define the concept of *unique violation*, where the configurations of two violation-producing input scenes should be apart by a user-defined threshold. *AutoFuzz* is optimized towards finding unique violations rather than every possible traffic violation. However, unique violation-producing inputs are sparse, and sparsity increases as the uniqueness threshold becomes more stringent. In such a sparse domain, the success of a fuzzer depends heavily on its initial seed selection and mutation strategy [13], as successful mutants are often limited in a sparse high-dimensional space, and chances of finding them without any guidance are thin. Besides, when a violation has been found, it is not trivial to automatically derive new violations with different parameters since a specific scenario leading to a violation can be very similar to a specific scenario leading to a safe outcome. One example is shown in Fig. 6 where a small change of the leading vehicle's speed can lead to drastically different results. To address these, we propose a novel seed selection and mutation strategy. Our key insight is, we can learn from the success/failure of the past mutants to produce traffic violationsand incorporate that knowledge in our fuzzing strategy. In particular, we devise a novel (i) learning-based seed selection and (ii) a gradient-guided mutation strategy that exploits knowledge learned from previous simulations.

Seed Selection. AutoFuzz learns from previous test-runs' behavior in an incremental learning setting and leverages past knowledge to filter out new test cases (*a.k.a.*seeds) that are unlikely to produce unique traffic violations. In particular, at each generation, we train a Neural Network (NN) classifier [13], [14], [15], [16] on previous runs' results to predict if a new input will lead to a unique traffic violation. The confidence scores of the NN's prediction are then used to rank the candidate inputs from highest to lowest, with the top ones are selected.

Mutation Strategy. The selected seeds are further mutated to increase their likelihood of causing unique traffic violations. Here we leverage a projected gradient descent (PGD) [17] strategy from the ML-based adversarial attack domain. At a high level, a small mutation is added to every relatively lower confident input from the seed selection step to increase the NN's confidence in it, by iteratively back-propagating the NN's gradient. However, naively applying gradient-guided mutation can generate invalid inputs. We resolve this problem by projecting each mutation back into a feasible region. The projection finds a feasible mutation value that obeys the grammar constraints and is also closest to the original mutation value. For this *AutoFuzz* applies a gradient-guided linear regression, where the grammar constraints are expressed as linear equations and the corresponding fields of the mutation values are variables.

Compared with previous works using evolutionary search based methods for AVtesting [6], [12], [18], our proposed seed selection and mutation strategy enable *AutoFuzz* to find more unique traffic violations. Besides, unlike previous works which focus on one particular (mostly proprietary) system in a couple of fixed scenarios running in a particular simulator, we show the effectiveness of our proposed open source fuzzer *AutoFuzz* in the combination of multiple AVcontrollers, scenarios, and simulators.

In summary, we make the following contributions:

- We introduce *AutoFuzz*, a grammar-based fuzzing technique to test AVcontrollers, which leverages the simulator's API specification to generate semantically valid test scenarios.
- We propose a novel learning-based seed selection and mutation strategy to optimize *AutoFuzz* for finding more unique traffic violations.

- We evaluate our *AutoFuzz* prototype on four AVcontrollers [19], [20], [21] in two simulators [22], [23]. On average, *AutoFuzz* can find 10-39% more unique traffic violationsper scenario than the best-performing baseline method.
- We reduce traffic violationsby 75-100% for the learning-based controller by fine-tuning it with the traffic violation-producing test cases.
- We make *AutoFuzz*'s source code and representative traffic violationsavailable at https://github.com/autofuzz2020/AutoFuzz[24].

*Contribution to SE Field.*First, the proposed seed selection and mutation strategy can be potentially applied to other fuzzing areas where inputs take a long time to execute, and one needs to leverage time and effectiveness. Second, *AutoFuzz* is the first open source general framework on fuzz testing for AVs in high fidelity simulators. It allows a user to test a new system under a user-specified scenario in popular, open-source high-fidelity simulators. Besides, it allows a researcher to compare a new AVfuzzing method with existing methods easily. We believe the paper along with *AutoFuzz* can make the research in the field of AVtesting more accessible and efficient to the community.

2 Background

2.1 Definitions

First, we define a few terms based on [25], [26]:

A **Scene** is a frame in the simulation that contains the detailed properties (e.g., location, velocity, acceleration) of the ego-car, other moving objects, the surrounding stationary objects, and road conditions. For example, the ego car is at map location (20, 20) with speed 5m/s facing north on a rainy afternoon.

A **Scenario** is "the temporal development between several scenes in a sequence of scenes" [25]. Two scenes could specify the same initial locations for the ego-car and other objects but different velocities, *etc.*resulting in different scenarios.

A *Functional Scenario* is a natural language description of an abstract scenario, e.g., the ego-car crosses an intersection. The examples in Table 1 belong to this category. Since such an abstract functional scenario cannot be fuzzed directly, we design a corresponding logical scenario as a special implementation of the former.

A *Logical Scenario* is the parameterized space where search during the fuzzing will be bounded. For example, the ego car that is crossing the intersection in the above example will start and end at locations (x_s, y_s) and (x_e, y_e), respectively, where $x_s, y_s \in [0, 20]$ and $x_e, y_e \in [20, 40]$.

A *Specific Scenario* is a concrete instance in the logical search space, e.g., the ego car crossing the intersection will start at $(10, 10)$ and end at $(30, 30)$. A specific scenariousually takes 30-50 seconds—if the simulation runs at 10Hz, this gives around 300-500 consecutive scenes.

2.2 Testing Autonomous Vehicle Controllers

There are three ways to test a controller: real-world, individual component, and simulation.

Real-world testing involves running the controller on the road. However, as per Table 1, many pre-crash functional scenarios may only occur in certain corner cases, i.e., variations in background buildings, weather, the behaviors of other vehicles, *etc.*It is extremely difficult to focus real-world testing towards such rare events.

Single component testing primarily focuses on the perception component or the planning component. The works for the perception component differ on the place perturbed: road sign [27], billboard [28], LiDAR input[29], camera image [30], [31], [32]), LiDAR and camera image [33], and the target they attack: perception[27], [28], [29], [30], motion planing [34], lane following controller [31], [32]. The works for the planning component differ on the characteristics of the scenarios to look for: avoidable collisions [35], patterns satisfaction [36], and requirements violation [37]. However, this line of research tends to miss more involved interactions between different components [38].

Simulator-based end-to-end testing treats the ego-car controller as an end-to-end system and usually uses high-fidelity simulations to find failure cases. Gambi et al. [4] create simulations that reproduce specific scenariosaccording to the functional scenariosleading to real car crashes in police reports. However, their system does not support testing different variations of the constructed specific scenarios, which is important to test for corner case behavior. Most other works study how to efficiently find challenging specific scenarios in a parameterized logical scenario space.

These works usually model the logical scenariowith only one or two agents having relatively simple behavior. However, many real-world crashes involve multiple dynamic agents with involved interaction (e.g., a leading car brakes when the ego car gets close within a certain distance). Further, these works usually focus only on collisions rather than other traffic violationslike going off-road. Furthermore, the search methods used, e.g., adaptive sampling[3] , bayesian optimization [5], topic modeling [20], reinforcement learning [39] , flow-based density estimation [40] tend to be either highly sensitive to hyper-parameters and proposal distributions[3] or not scale well to high-dimensional search space[5], [20], [39], [40] .

Among these, perhaps the closest to our work are evolutionary-based algorithms [10], [18], [41], [42] and their variants (with NN [12] or Decision Tree [6] for seed filtration) on testing AVor Advanced Driver-Assistance Systems (ADAS). These methods can scale to high-dimensional input search spaces. Unfortunately, they are currently only used for testing one particular ADAS system or its component (e.g., Automated Emergency Braking (AEB) [6], Pedestrian Detection Vision based (PeVi) [12], OpenPilot [42], and an integration component [10]) under one particular logical scenario, testing a controller on road networks without any additional elements (e.g., weather, obstacle, and traffic) [41], or focusing on finding collision accidents in a logical scenario with other cars constantly changing lanes [18]. In contrast, our proposed *AutoFuzz* is generalized to different AVsystems and scenarios. Our learning-based seed selection and mutation strategy further enables *AutoFuzz* to disclose more unique traffic violationsthan the existing methods. We adapt the algorithms from [6], [12], [18] in our setting, and compare with *AutoFuzz*.

2.3 Motivating Example

AutoFuzz aims to generate traffic violationsby an ego car controller by fuzzing the input scenes. *AutoFuzz* starts with a

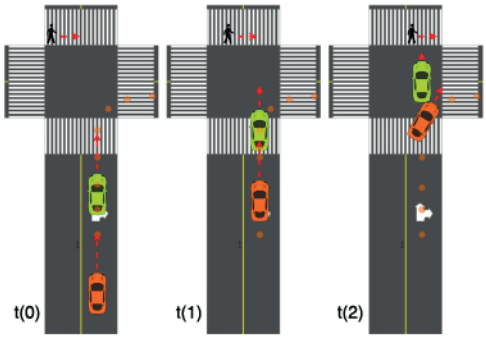


Fig. 2. Example of crash simulation in consecutive time steps.

logical driving scenario that involves traffic violations, designed based on the top pre-crash functional scenariosfrom NHSTA [7] (see Table 1). For instance, "vehicle leading ego car stopped" and "non-signalized junction" are the top causes of manual car crashes, and *AutoFuzz* tests how an AVbehaves in such situations. Fig. 2 presents this scenario. To simulate a crash in such a situation, *AutoFuzz* starts the simulation with a green car leading an orange ego car near a non-signalized junction (Fig. 2-t(0)). From there, with fuzzing, *AutoFuzz* generates the following crash: the ego-car is going to turn right while the leading car suddenly slows down to avoid hitting a pedestrian who is crossing the road (Fig. 2-t(1)). This leads the ego car to collide with the leading car (Fig. 2-t(2)). To simulate the collision, *AutoFuzz* leverages CARLA's APIs related to vehicle, pedestrian, and cross-road in the map. Since the forces that influence collision are mainly the pedestrian's behavior and the leading vehicle's behavior, starting with these agents and starting location in the map, *AutoFuzz* needs to search for valid driving directions for all the agents, their speeds, road condition, etc. to simulate the crash. Exemplary challenging specific scenariosin addition to the collision shown here include the pedestrian gets occluded by the leading vehicle (as shown in Fig. 6), and the background is at night with heavy rain. The detailed search space of this logical scenariois provided in Appendix H in supplementary material, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSE.2022.3195640.

3 API Grammar

Fig. 3 shows a simplified version of the APIs that *AutoFuzz* uses to simulate crashes in our prototype implementation for CARLA. The core of the simulation is an initial driving *Scene* with four main components: a route map, the ego car whose controller is under test, some static and dynamic objects (e.g., other vehicles, pedestrians, *etc.*), and background like weather and road conditions.

CARLAprovides the API specifications as a set of Python APIs [22], [43]. For example, calling *CarlaDataProvider.request_new_actor(pedestrian_model, spawn_point)* creates a pedestrian, where *pedestrian_model* is a pedestrian asset predefined in CARLAand *spawn_point* specifies the pedestrian's initial location and direction. From such specifications we construct a test-generation grammar, $\mathcal{G}$*(Map, Ego Car, Objects, Background)*, shown in Listing 1. Encoding the grammar in JSON format allows us to specify values for each field. We extend the grammar by adding two constraints for restricting the search region (see Listing 1) and additional conditions (e.g., the distance between the ego-car and the leading car must be greater than a certain distance).

After processing CARLA's APIs, we get a Test Grammar, $\mathcal{G}$, as $\mathcal{G}$*(Map, Ego Car, Objects, Background, Search Range, Constraint)*, where the underlined components are optional. The details of search range and constraints are provided in Appendix D in supplementary material, available online.

Listing 1. An example Test Grammar, $\mathcal{G}$, from CARLA's specification. The JSON-encoded grammar snippet is for the pedestrian in the motivating example. The constraints specified at the bottom express one vehicle's target_speed $\leq 0.5\times$ of another vehicle's target_speed

```
pedestrian_0: {
 setup: {
   location: {
     x:[-123, -83, (normal, None, 10)],
     y:[3.5, 43.5, (normal, None, 10)]
   }
   direction: [0, 360],
   type: [0, 12]
 },
 trigger_event: {
   trigger_distance:[2, 50],
   target_speed: [0, 4],
   travel_distance: [0, 50]
   }}

customized_constraints: [{
  coefficients: [1, -0.5],
  labels: [vehicle[0].trigger_event.target_speed,
           vehicle[1].trigger_event.target_speed],
  value: 0
  }]
```

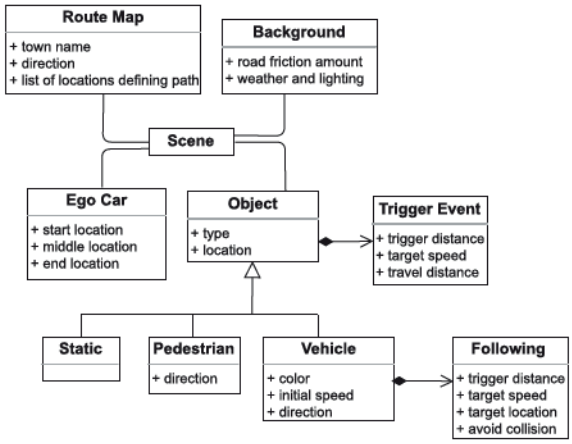
4 Methodology

Leveraging the API grammar as described in Section 4.2, *AutoFuzz* fuzzes inputs to the ego-car's controller in a black-box manner. We make several design decisions to address the following questions: (i) How to define *unique violation* to simulate *diverse* traffic violations? (Section 4.1) (ii) How to generate only semantically *valid* scenes? (Section 4.2) and (iii) How to design the fuzzing algorithm to produce more *valid unique* traffic violations? (Section 4.3)

4.1 Diverse Traffic Violations

We focus on two types of violations: collision and going out-of-road. A *collision* consists of colliding with other moving or stationary objects. An *out-of-road* violation consists of going into a wrong lane (opposite direction traffic), onto the road's shoulder or literally off-road.

The goal of a good fuzzer should be to find diverse bugs. However, defining diversity for traffic violationsis a hard problem. Merely comparing the violation-inducing inputs may lead to infinitely different violations. For example, let's assume that a stationary pedestrian in front of a car results in a crash. By modifying unrelated input parameters (e.g., the position of another pedestrian far from the crash site, the position of another vehicle in a different lane, etc.), possibly outside the vision of the ego-car controller, we can generate an infinite number of *different* violations. But such



API	Description
Route Map	The user selects a route map, identified by *town name*, which the ego-car should drive. A map contains a path consisting of a sequence of 2D locations—the first and last locations in the sequence refer to the start and destination of the ego-car. CARLA comes with eight predefined maps.
Ego Car	The controller of the ego car is under test in this paper.
Background	The user can set up a driving environment with different weather and road conditions. The road conditions are set by different friction values. CARLA has 21 predefined weather and lighting modes.
Objects	The user can choose a range of static (*e.g.*, debris, bus stop, *etc.*) and moving (*e.g.*, vehicles and pedestrians) objects that can appear dynamically around the ego car's route. Each moving object is associated with a *triggering event*, which specifies the triggering condition and behavior after being triggered. Each vehicle is also associated with a behavior, which makes the vehicle follow CARLA's map with a specified speed to a given destination. Users can also choose each vehicle's type (*e.g.*, tesla model 3, nissan patrol, *etc.*), color, and whether to try to drive directly to the destination without regard to other objects.

Fig. 3. A simplified description of CARLA's APIs. We fuzz only over the background and objects.

redundancy is not interesting nor useful. Thus, criteria for precisely defining *unique* traffic violationsis needed.

Abdessalem et al. [6] define that two test specific scenarios are distinct if they differ in "the value of at least one static variable or in the value of at least one dynamic variable with a significant margin." This definition fails in our high-dimensional scenarios, as the example above could be considered different violations by their criteria. We instead count the number of unique violations as:

Unique Violation. For a given type of traffic violation(collision or out-of-road), two violations caused by specific scenarios x and y are *unique*if at least $th_1\%$ of the total number of changeable fields are different between the two, where th_1 is a configurable threshold.

For a discrete field, the corresponding values are different if they are non-identical in x and y (e.g., "color" field is different between a black and a white car). For a continuous field, the corresponding normalized values should be distinguishable by at least $th_2\%$, where th_2 is a user-defined threshold. For instance, if the speed range of a car is [0,10] m/s, and two violations occur at speeds 3m/s and 4m/s, the field is considered to be the same between the two violations since $\frac{4-3}{10-0} = 0.1 < 0.15$, where $th_2\% = 15\%$.

Scalability of the Definition. Compared with the definition in [6], the new definition has two benefits in terms promoting the violation's diversity for higher dimensional logical scenarios. First, since $th_1\%$ is the percentage of the total number of changeable fields that need to be different, given a fixed $th_1\%$, as the number of changeable fields goes up, the number of changeable fields that need to be different for two violations to be considered different also goes up. Second, the current definition enables a user to specify the thresholds $th_1\%$ and $th_2\%$ according to one's need. For scenarios with higher dimensions, larger $th_1\%$ can be used.

Benefits of Finding More Unique Violations. There are two benefits of finding more unique violations for AV testing. First, it enables engineers to better identify the limitation of the AV under test. Different violations can potentially expose different functionality issues and/or with different causes. Compared with the formulation of finding the Pareto front as in [12], our method allows more exploration and thus can find not only the most severe violations but also less severe violations that should be avoided and can be potentially useful for improving the AV under test (e.g., collisions at low speed). Such violations tend to be missed by methods optimized for Pareto front since they usually generate new seeds based on the most extreme violations so far at each generation. Second, by maximizing the number of unique violations found, the "violation coverage" in the user specified logical scenariois maximized. Instead of maximizing the "branch coverage" as in the traditional fuzz testing, we maximize the number of unique specific scenarios(for each logical scenario) that induce violations. This can help a tester to validate if an AV can perform well in the specified logical scenarioas expected.

4.2 Fuzzing With API Grammar

AutoFuzz takes the API grammar as input and fuzzes following the grammar spec. The user first selects a route map where the ego-car controller will drive and a starting initial scene encoded according to the API grammar. Users can optionally specify a customized search region and constraints. *AutoFuzz* uses these pieces of information to sample initial scenes(also called seeds in fuzzing); Each sampled initial scene obeys the constraints enforced by the API grammar.

Fig. 1 shows a high-level overview of the fuzzing process. The objective is to search for initial scenesthat will lead to unique traffic violations. To achieve this, like common black-box fuzzers, *AutoFuzz* runs iteratively: *AutoFuzz* samples the grammatically valid initial scenes (Step-1), and the simulator runs these initial scenes with the controller under test to collect the results as per the objective functions, as detailed in Section 4.3.1. *AutoFuzz* leverages feedback from previous runs to generate new seeds, i.e., favors the ones that have better potential to lead to violations over others (Step-II) and further mutates them (Step-III). The API grammar constraints are followed while incorporating feedback to create new mutants, so all the mutants are also semantically valid. The new seeds are then fed into the simulator to run. The traffic violationsfound are reported, and their corresponding seeds added to the seed pool. This repeats until the budget expires.

4.3 Fuzzing Under Evolutionary Framework

AutoFuzz aims to maximize the number of *unique* traffic violationsfound within a given resource budget (e.g., # simulations). This is an optimization problem, where *AutoFuzz* searches

over the entire input space of grammatically valid initial scenes to maximize unique violations found by simulating from those scenes. More formally, if $\mathcal{X}$ is the space of all possible valid input scenes, *AutoFuzz* searches over $\mathcal{X}$ to maximize traffic violationcount ($\mathcal{Y}$) within a fixed budget, say $\mathcal{T}$. Thus, if B_t is the set of traffic violationsfound by input $x_t \in \mathcal{X}$ at fuzzing step t, then more formally fuzzing is: $\mathcal{Y}_{\mathcal{T}} = \max \| \bigcup_{t=1}^{\mathcal{T}} B_t \|$. Here $\|\cdot\|$ is the norm and $\bigcup(\cdot)$ represents the union of all violations over all possible inputs.

Since the input space $\mathcal{X}$ is prohibitively large, an exhaustive search to optimize the equation is infeasible. Instead, one needs to identify and focus the search on promising regions to optimize the number of unique violations. Fuzzing based on evolutionary algorithms is a common approach for such optimization. Starting with some initial inputs, evolutionary fuzzers tend to select new inputs that find new violations and further mutate those successful inputs to generate further new inputs. Thus, the success of fuzzing depends on careful design of the following three parts:

i) *Objective function (F):* How to design a objective function to maximize unique bugs?
ii) *Seed Selection ($x \in \mathcal{X}$):* Which inputs to mutate [44]? and
iii) *Mutation(m):* How to mutate [16], [45], [46]?

Thus, the next generated input at time t, x_t depends on $(x_{:t-1}, m)$, where $x_{:t-1} := x_1, \ldots, x_{t-1}$. The set of traffic violations B_t found by x_t can be represented as a function (F) of these fuzzing parameters, i.e., $B_t = -F(x_{:t-1}, m)$, such that minimizing F will maximize the unique traffic violations. Thus, more formally, evolutionary fuzzing (with x_0 is an initial seed input) can be written as:

$$\mathcal{Y}_{\mathcal{T}} = \min_{x_{:t-1}, m} \left\| \bigcup_{t=1}^{\mathcal{T}} F(x_{:t-1}, m) \right\| \quad (1)$$

In the following, we discuss the details of the fuzzing.

4.3.1 Objective Function

The ultimate goal of the fuzzing algorithm is to maximize diverse traffic violationsfound. However, as the bug-producing inputs are sparse, we need more violation-specific guidance to help the ego car move towards the violation points. For example, to generate a collision with a pedestrian, we need to guide both the ego car and the pedestrian closer to each other. Thus, we need a *smoother* objective function that helps lead towards the traffic violation. To this end we define the following objective functions:

Violation Type	Objective	Definition
Collision	$F_{collision}$	:= speed of ego-car at collision
	F_{object}	:= minimum distance to other objects
	F_{view}	:= minimum angle from camera's view
Out-of-road	$F_{wronglane}$	:= minimum distance to an opposite lane
	$F_{offroad}$	:= minimum distance to a non-drivable region
	$F_{deviation}$	:= maximum deviation from interpolated route

Collision. We optimize for the weighted sum of the three smooth objective functions: $F_{collision}$, F_{object}, and F_{view}, similar to the objectives used in [6], [10], [12]. $F_{collision}$ and F_{object} promote the severity of collision and the chance of collision, respectively. $F_{collision}$ is set to -1 as per [6] when no collision happens. F_{view} promotes cases where the object(s) involved are within the camera(s) view.

Out-of-road. This is implemented by a weighted sum of the three smooth objectives: $F_{wronglane}$, $F_{offroad}$, and $F_{deviation}$. $F_{deviation}$ is adapted from the objective of "maximum distance deviated from lane center" in [11].

We further define $F_{wronglane}$ and $F_{offroad}$ to strengthen the signals for driving into an incorrect lane or off the road, respectively. Fig. 15 in Appendix E in supplementary material, available online provides an illustration.

For each traffic violationtype, we formulate the fuzzing problem as a constrained multi-objective optimization. Let x be an input, i.e., a specific scenario with all the searchable fields. Denote $F_i(x)$ for $i = 1, \ldots, n$ to be n objective functions, w_i to be some user-provided weights, and $g_j(x)$ for $j = 1, \ldots, p$ to be p constraints, where each constraint is expressed as ≤ 0 form. Then, the objective function $F(x)$ of Eq. (1) can be expressed as a constrained weighted sum: $\min_x \sum_{i=1}^{n} w_i F_i(x)$, s.t. $g_j(x) \leq 0 \; \forall j = 1, \ldots, p$. Unlike [6], [12], we optimize for a weighted sum of objective functions rather than search for a Pareto front of the involved objective functions, because our goal is to find the maximum number of unique traffic violationsrather than traffic violationswith the Pareto front of multiple objectives.

4.3.2 Seed Selection

Common evolutionary fuzzers like AFL [47] maintain a seed queue and tend to favor some seeds over others. Smart seed selection strategies give a significant boost to fuzzing performance to not waste limited resources by running fruitless seeds [48], [49]. In our case, a bad seed may lead to running several scenes without simulating a traffic violation. We devise an incremental learning-based seed selection strategy, as shown in Fig. 1.

For each generation t of our evolutionary search, a Neural Network (NN_{t-1}) is trained with all the seeds executed up to generation $t-1$, such that the NN learns to differentiate between successful versus unsuccessful seeds. NN_{t-1} is used to predict the seeds generated in generation t. It ranks all the candidate seeds of generation t based on its confidence of leading to a unique traffic violation. *AutoFuzz* then selects the top S seeds that are more likely to produce violations, where S is a configurable parameter. Fig. 4 illustrates this process. The top row shows all the seeds generated in a particular generation. The NN ranks them based on their potential to produce unique violations—darker color is more violation prone than lighter. The top S seeds are then selected for future steps (in the second row.)

4.3.3 Mutation

Among the top s seeds selected in the previous step, not all are equally likely to lead to unique violations. In particular, the NN has lower confidence on the bottom seeds of the ranked list (the lighter color seeds in the second row of Fig. 4). *AutoFuzz* further mutates such lower confidence

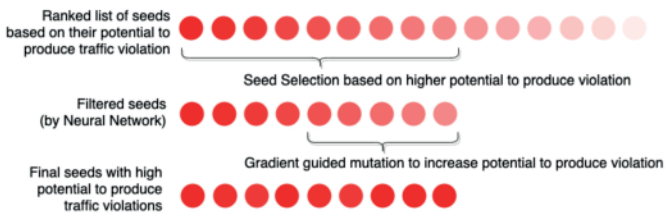


Fig. 4. Seed selection & mutation strategy per generation.

seeds to increase their potential to simulate traffic violations. A constrained gradient-guided perturbation mutates the lower confidence seeds towards higher confidence (the third row in Fig. 4 where all the seeds become dark red). This perturbation is generated by iteratively back-propagating the input's gradient with respect to the NN's prediction. We describe the perturbation algorithm in Section 5.

5 Implementation Details

We realize our evolutionary fuzzing design discussed in Section 4 following the main steps: Sampling, Seed Selection, and Mutation (see Fig. 1). Algorithm 2 in Appendix A in supplementary material, available online gives the detailed algorithm.

Step-I: Sampling: This step samples seed test cases from the entire input space by obeying the constraints enforced by the API grammar. We use two sampling strategies: (i) random and (ii) genetic algorithm (GA). Each field is sampled based on a user-specified distribution, search range, and constraints (see Listing 1). In either strategy, when the specified constraints are not satisfied, each variable will be re-sampled. If the specified constraints and cannot be satisfied after a specified number of attempts, the program will raise an error. We filter out seeds similar to those corresponding to previous relevant traffic violations. In the fuzzing literature, this step is commonly used for *test suite minimization* [10].

At each generation, the GA considers the previous seeds with results, selects from them new parent test cases, and generates new seeds through crossover and mutation.

*Selection:*We adopt binary tournament selection with replacement, like the original NSGA2implementation [50], as well as the variations in [6], [12]. Two duplicates are created for each sample and randomly paired. Each pair's winner is then randomly paired as the parents for this generation's mating process. The rank of two individuals is determined by the objective function in Section 4.3.1.

*Crossover & Mutation:*Simulated Binary Crossover [51], a classical crossover method commonly used for floating point numbers, is adopted, as in [6], [12], [50]. A distribution index (η) is used to control the similarity of the offspring and their parents. The larger η is, the more similar the offspring are *w.r.t.*their parents. We set $\eta = 5$ and probability=0.8 to enable more diversity. If a larger η is used, the offspring will be more similar to their parents, so it takes longer to find distinct offspring for methods with uniqueness filtration and results in fewer unique bugs found for methods without. If a smaller η is used, the offspring will be too distinct from their parents and violation-inducing parents won't be fully leveraged. Polynomial Mutation is applied to each discrete and continuous variable [52]. For discrete variables, we treat the value as continuous during the mutation and round later. We clip the values at specified boundary values. Following [6], mutation rate is set to $\frac{5}{k}$, where k is the number of variables per instance. We further set the mutation magnitude η_m to 5 for larger mutations.

Algorithm 1. Constrained Gradient Guided Mutation

Input: x: test case, f: NN forward function of predicting a test case's likelihood of being a traffic violation, $\mathbf{th_{conf1}}$: threshold of conducting a perturbation, $\mathbf{th_{conf2}}$: threshold of stopping a perturbation, n: maximum number of iterations, λ: step size, c: constraints, ϵ: maximum perturbation bound, $\mathbf{x_{min}}$: minimum allowable values, $\mathbf{x_{max}}$: maximum allowable values
Output: x′: mutated test cases

$x' = x$;
$i = 0$;
if $f(x) > th_{conf1}$ **then**
  return x;
end
while $i < n$ **do**
  $i+ = 1$;
  $dx = \lambda \frac{df(x')}{dx'}$;
  $x' = x' + dx$;
  $x' = clip(x,' x_{min}, x_{max})$;
  $dx = clip(x' - x, -\epsilon, \epsilon)$;
  if *check-constraint-violation* $(c, dx) ==$ *True* **then**
    dx = linear-regression (c, dx);
  end
  if *is-similar* $(X, x + dx)$ **then**
    break;
  end
  $x' = x + dx$;
  if $f(x') > th_{conf2}$ **then**
    break;
  end
end
return x'

Step-II: Seed Selection: As described in Section 4.3.2, we boost fuzzing performance with a learning-based seed selection strategy. We train a shallow neural network (1-hidden layer) using the previous seed test cases to predict if a test case leads to a traffic violation. The NN ranks the next generation seeds based on its confidence of leading to a traffic violationand the most likely tests are selected. Some previous work [12] also leverages an NN for seed selection. There are several major differences. First, we train a single NN for binary classification of traffic violationsrather than several NNs for regressing over all objective values as in [12]. Thus we rank test cases based on the confidence value of finding a traffic violationrather than the Pareto front from multiple NNs. This design choice is motivated by our goal to find maximum number of valid, diverse traffic violationsrather than finding the best set of traffic violationsachieving the optimal trade-off among multiple objectives at the same time. Second, we iteratively train the NN in an active learning setting rather than training fixed ones at the beginning. This active training results in increasingly more training samples than the initial population and, thus, improved NN approximation over time. We show both design choices introduce performance gains in the experiment section.

TABLE 2
Different Driving Scenarios Under Test

Logical Scenarios Names	Corresponding NHTSA functional scenarios*	#Para	Map ID	Road Type	#violations found**
Turning right while leading car slows down/stops	Leading vehicle stopped / deccelerating	26	town05	junction	512
Turning left a non-signalized junction	Vehicle(s) turning at non-signalized junctions	26	town01	non-signalized T-junction	672
Crossing a non-signalized junction	Straight crossing paths at non-signalized junctions	47	town07	non-signalized junction	400
Changing lane	Vehicle(s) changing lanes – same direction	26	town03	straight road	147
Turning left a signalized junction	LTAP/OD at signalized junctions	11	Borregas	signalized	76

*All scenarios involve ego car lost control or drove off-road, without taking any action, by testing if the ego-car goes out-of-road.
**(first four rows) average numbers of collision traffic violations(for town03 and town05) or out-of-road traffic violations(for town01 and town07) found by GA-UN-NN-GRADon the Ibc controller in CARLA. (last row) average number of collision traffic violationsfound by GA-UN-NN-GRADon APOLLO6.0in SVL.

Step-III: Constrained Gradient-Guided Mutation: As per Section 4.3.3, we apply a constrained gradient-guided mutation on the selected top test cases to maximize their likelihood of leading to traffic violations. The procedure, shown in Algorithm 1, is adapted from the constrained adversarial attack in [53].

A test case x is perturbed only when the NN's confidence in its leading to a traffic violation, $f(x)$, is smaller than a threshold th_{conf1}. If a test case is already considered highly likely to lead to a traffic violation, there may be no extra benefit in further perturbing it. Otherwise, an iterative process begins (line 6-21). At each iteration, a small perturbation dx is generated (line 8) via back-propagation from maximizing the test case's NN confidence. The perturbation is then clipped based on allowable input value domains and a user-specified maximum perturbation bound ϵ (line 9-11). Next, the perturbation is checked against grammar constraints (line 12). If necessary, a linear regression projects it back within the constraints. The perturbed test case is then checked against previously found traffic violations(line 15). If a similar test case already found a traffic violation, the perturbation process ends, and the latest perturbation won't be applied. Otherwise, the current perturbation is applied on top of the perturbed test case from the last iteration (line 18). The new perturbed test case is then fed into NN for its confidence of leading a traffic violation. If larger than a specified threshold th_{conf2}, the mutated test case will be returned and the mutation procedure ends. Otherwise, a new iteration begins.

One difficulty here is to make sure the perturbed test case still satisfies the grammar constraints. The simplest solution is to discard the perturbations (and subsequent iterations) that lead to constraint violation. However, as shown in [53], the insight for linear constraints is if an original (unperturbed) test case satisfies the constraints and the perturbation alone satisfies the constraints as well, then the perturbed test case also satisfies the constraints. Thus, only the perturbation needs to be checked against the constraints after each iteration. If some constraints are violated, we apply a linear regression to the perturbation to map it back within the constrained region (motivated by [53]). For the linear regression, the non-constant part of the constraints are weights $\mathbf{W}$ where each row corresponds to the coefficients of one constraint, the constant parts y are the objectives, and the projected perturbation dx_{proj} are the variables to search for. The linear regression starts with the perturbation dx and find the the projection $dx_{proj} = \arg\min_{dx_{proj}} \|\mathbf{W}dx_{proj} - y\|$.

6 Experimental Design

Environment. Our primary evaluation uses the CARLA(version 0.9.9) simulator [22]. To show the generalization of our approach, we further conduct evaluation using the SVL(version 2021.3) simulator [23] in RQ4. All the algorithms are built on top of pymoo [54], an open-source Python framework for single- and multi-objective algorithms.

Scenarios. We run *AutoFuzz* under five different logical scenarios(Table 2) inspired by the NHTSA report [7].

Selection. The first three logical scenarioscover the top six pre-crash functional scenariosin terms of frequency and incurred economic cost as shown in Table 1. The last two scenarios are also common logical scenariosranked fourth and eighth among the pre-crash scenarios of two-vehicle light-vehicle crashes in terms of occurrence frequency [7]. Note that the other frequent scenarios have been covered by the first three.

Adaptation. To use a NHTSA pre-crash scenario for testing, we let the ego car be a vehicle involved in each crash scenario. To make the scenario searchable, we convert each functional scenario into a logical scenario (defined in Section 2.1) that satisfies the functional scenario's description. For example, in the scenario "A leading vehicle decelerating/stopped", the ego car is the following vehicle. We set the search range of the location of the leading vehicle to be in the same lane and ahead of the ego car. Additionally, we set the search range of the speed of the leading vehicle after being activated to be slower than that of its initial speed. These designs enable every generated specific scenario to satisfy the logical scenario "A leading vehicle decelerating/stopped".

Validity. The scenarios we use are supposed to be within the operation design domain (ODD) of the AVcontrollers under test which are all supposed to handle regular traffic scenarios. To check that they can handle the base scenarios, we conduct a validity test for each scenario. The result shows that, when no other vehicles/pedestrians are present, the corresponding AV controller can successfully reach its destination without incurring traffic violations. When there are other vehicles/pedestrians, the corresponding controller

succeeds with no violations in some cases but not others. Our goal is to find those violations.

AVcontroller. We test two rule-based PID controllers, **pid-1** [20] and **pid-2** [19], one end-to-end controller [19], (**lbc**), and one modular controller [21], (**APOLLO6.0**). lbc is a vision-based, end-to-end controller proposed in [19]. PID controllers assume knowledge of the states of other objects in the environment and the trajectory to follow. They attempt to reach the next planned location with a specified speed by adjusting controls for brake, throttle, steering and try to minimize the mismatch with the desired speed and direction while avoiding collision with other objects. pid-1 is a default rule-based controller in CARLA's official release [22] and has been used as the main system under test in existing literature [20]. pid-2 is a rule-based pid controller implemented by the authors in [19] to collect data to train lbc. APOLLO6.0is an industrial-grade, modular controller [21].

*Hyper-Parameters.*The NN for seed selection has a hidden layer of size 150. We use the Adam optimizer with 30 epochs and batch-size 200. th_{conf1} is set to be the $0.25 \times p$-th highest NN confidence value among training data, where p is the percentage of the training data leading to traffic violations, and th_{conf2} is set to 0.9. ϵ is set to be 1, n is set to 255 and λ is set to $1/255$ so an input seed can be perturbed to any other input seed in the input domain. We collected seeds up to 10 generations (and thus 500 simulations) by default. The default method used for seed collection is GA-UN.

Metrics. When we compare search quality, we use the number of *unique* traffic violationsfound over the corresponding number of simulations run. We use the number of simulations rather than time because the former is platform independent. Moreover, the time costs mainly come from simulations. On average, each simulation takes about 40 seconds, while the generation process only takes about 10 seconds and is only invoked once per generation. A simulation ends if a violation happens, the ego car reaches the destination, or time (50 seconds) runs out. When counting collision traffic violations, for lbc, pid-1, and pid-2, we only count those where the collision happened within the view of the controller's front camera and the controller did not stop to avoid the collision. For APOLLO6.0, since it is equipped with LiDAR (providing 360 degrees view), we count all collision traffic violationswhere it did not stop to avoid them. We further manually checked a set of found collision scenarios and found they can be avoided if the controllers maneuver correctly. For example, in Fig. 11, if APOLLO6.0slows down earlier, both collisions could be avoided. When the baseline method AV-FUZZERis considered (i.e., Figs. 7 and 10), since it does not have a seed collection stage, for a fair comparison, the number of simulations for the seed collection stage of other methods is also included. When the comparison does not involve AV-FUZZER, the number of simulations for seed collection is excluded since all the methods will be set to share the same seed collection stage for a fair comparison. We set uniqueness thresholds $th_1 = 10\%$ and $th_2 = 50\%$ as default values, and explore the sensitivity of different search methods under nine different combinations.

Baseline Comparison. We compare *AutoFuzz* with three baseline methods shown in Table 3's baselines row. To fairly compare the fuzzing strategies on equal footing, we used the same objectives from Section 4.3.1 and the same random sampling with uniqueness filtration to generate the initial populations for all. We also compare *AutoFuzz* with alternative design choices in Table 3's variants row.

TABLE 3
Proposed Methods, Baselines and Variations

Method	Description
AutoFuzz (GA-UN-NN-GRAD) (ϵ=1.0)	GA-UN-NNW/ constrained gradient guided mutation
Baselines	
NSGA2-DT [6]	NSGA2W/ decision tree
NSGA2-SM [12]	NSGA2W/ surrogate model
NSGA2-UN-SM-A	NSGA2-SMW/ duplicate elimination and incrementally learned surrogate model
AV-FUZZER [18]	global GA+ local GA
Variants	
GA-UN-NN-GRAD* (ϵ=0.3)	GA-UN-NN-GRADW/ a smaller (0.3 rather than 1) maximum perturbation bound ϵ
RANDOM-UN-NN-GRAD	RANDOMW/ duplicate elimination, NN filtration and constrained gradient guided mutation
GA-UN-NN	GA-UNW/ NN filtration
GA-UN	GAW/ duplicate elimination
GA	genetic algorithm
RANDOM	random sampling

**GA= Genetic Algorithm, UN = Unique, NN = Neural Network based seed selection, GRAD=Gradient guided mutation.*

Among the baseline methods, NSGA2-DTand NSGA2-SMare two multi-objective GA-based methods and AV-FUZZERis a single-objective GA-based method, all of them are adapted from previous work [6], [12], [18]. NSGA2-DTcalls NSGA2 [50] as a subroutine. After each run of NSGA2, NSGA2-DTfits a decision tree over all instances so far. It uses cases that fall into the leaves with more traffic violationsthan normal cases (a.k.a. "critical regions") as the initial population for NSGA2's next run. During NSGA2, only the generated cases that fall into the critical regions are run. We set search iterations to 5 as in [6]. Since the tree tends to stop splitting very early in our logical scenarios, we decrease the impurity split ratio from 0.01 to 0.0001. We set minimum samples split ratio set to 10%.

NSGA2 -SMtrains regression NNs for every search objective and ranks candidate test cases and traffic violationsfound so far based on the largest Pareto front and crowding distance, as in NSGA2. To further compute the effects of uniqueness and incremental learning as well as the effects of weighted sum objective and gradient-guided mutation, we implement NSGA2-UN-SM-A— a variant of NSGA2-SMwith additional duplication elimination and incremental learning. For both NSGA2-SMand NSGA2-UN-SM-Atraining processes, we first sampled 1000 additional seeds to train three regression NNs. For finding collision violations, the three NNs are trained to predict F_{object}, $F_{collision}$, and F_{view}, respectively; for finding out-of-road violations, the three NNs are trained to predict $F_{wronglane}$, $F_{offroad}$, and $F_{deviation}$, resp. The NNs all have one hidden layer with size 100. The batch-size, training

epoch and optimizer are set to 200, 200, and the Adam optimizer.

AV-FUZZER [18] first runs a global GAfor several iterations and enters a local GAwith the initial population set to the scenario vectors with the highest fitness scores. It also starts a new global GAevery time when the fitness score of the current generation does not increase anymore compared with a running average of the last five generations. We keep the hyper-parameters used as in the original implementation e.g. population size is set to 4.

We did not directly compare with FITEST [10], Asfault [11] or FusionFuzz [42] since they are essentially GAwith specifically designed objectives targeting testing of the integration component of an AV, a controller's performance under different road networks, or the fusion component of an AV, respectively, while we focus on testing a black-box end-to-end system on a predefined map available with different specific scenarios by mutating different elements (e.g., weather, agents, their positions and behaviors).

7 Results

To evaluate how efficiently *AutoFuzz* can find unique traffic violations, we explore the following research questions:

RQ1: Evaluating Performance. How effectively can *AutoFuzz* find unique violations versus baselines?

RQ2: Evaluating Design Choices. What are the impacts of different design choices on *AutoFuzz*?

RQ3: Evaluating Repair Impact. Can we leverage traffic violationsfound by *AutoFuzz* to improve the controller?

RQ4: Evaluating Generalizability. Can *AutoFuzz* generalizes to a different system and simulator combination?

RQ1. Evaluating Performance. We first explore whether *AutoFuzz* can find realistic and unique traffic violationsfor the AVcontrollers under test. Note that all the traffic violationsare generated by valid specific scenario, as they are created using CARLA's API interface (we also randomly spot-checked 1000 of them). We run *AutoFuzz* with GA-UN-NN-GRADon all three controllers for 700 simulations, with the search objective to find collision traffic violationsin the town05 logical scenario. Note that even though the search objective is set to finding collisions, the process might also find a few off-road traffic violations. Overall, *AutoFuzz* found 725 unique traffic violationstotal across the three controllers for this logical scenario. In particular, it found 575 unique traffic violationsfor the **Ibc** controller, 80 for the **pid-1** controller, and 70 for the **pid-2** controller. Since **pid-1** and **pid-2** assume extra knowledge of the states of other environment objects, it is usually harder to find traffic violations. Fig. 5 shows snapshots of example traffic violationsfound by *AutoFuzz*. These examples illustrate that starting from the same logical scenario, different violations can be generated because of the high-dimensional input feature space. Fig. 6 shows an example violation of the motivating logical scenario"Turning right while leading car slows down/stops". A small mutation of the leading vehicle's speed from 3m/s to 4m/s or 2m/s leads to completely different outcomes. When the leading vehicle's speed is 3m/s, the ego car has less time to detect the presence of a pedestrian obstructed by the leading vehicle (b2) and ends up with colliding with the pedestrian (c2). When the leading vehicle's speed is

Fig. 5. RQ1. Example traffic violationsfound by *AutoFuzz*. For each row, the time goes by from left to right. (1st row) pid-1 controller collides with a pedestrian crossing the road. (2nd row) pid-2 controller collides with the stopped leading car. (3rd row) Ibc controller makes a wide turn into the opposing lane (considered "off-road").

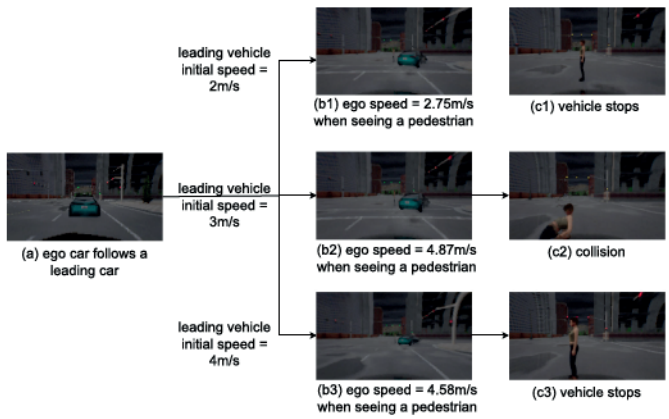


Fig. 6. An example (front camera view) where a small parameter change leads to distinct outcomes for Ibc in CARLA

4m/s, the ego car detects the pedestrian earlier (b3) and avoids the collision by braking on time (c3). When the leading vehicle's speed is 2m/s, although the ego car detects the pedestrian late (b1), the ego car is at low speed (2.75m/s) so it also manages to avoid the collision (c1). It should also be noted that the collision (c2) can be avoided if the ego car brakes the first time it sees the pedestrian (b2). This traffic violationis non-trivial to be found since the change of the leading NPC vehicle's initial speed leads to different reaction of the ego car and it is not clear what value of the initial speed along with other parameters in the search space leads to the collision. In fact, AV-FUZZERfails to find this violation since AV-FUZZERgets stuck at another traffic violationinvolving the ego car's collision with the slowing down leading NPC vehicle.

We compare *AutoFuzz* (i.e., GA-UN-NN-GRAD) with the baseline methods NSGA2-DT, NSGA2-SM, NSGA2-UN-SM-A, and AV-FUZZER under four different logical scenarios. We focus on collision traffic violationsfor two logical scenarios and off-road traffic violationsfor the other two. In each setting, we run each method 6 times and report mean and standard deviation. For AV-FUZZER, we fuzz for 1200 simulations. For other methods, we assume 500 pre-collected seeds and fuzz for 700 simulations. Fig. 7 shows the results.

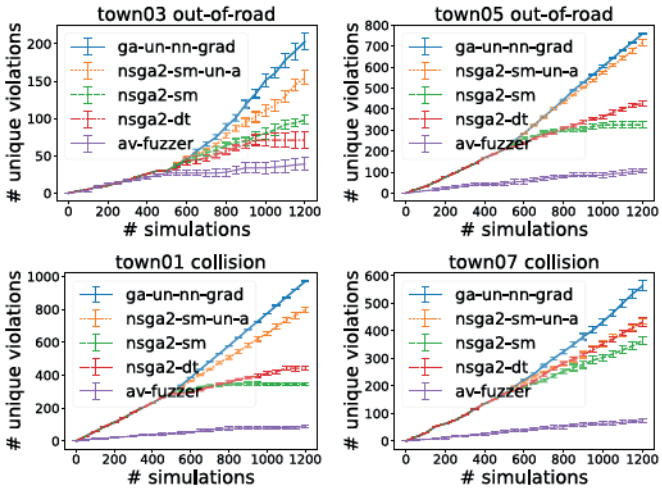


Fig. 7. RQ1. average # unique off-road or collision violations.

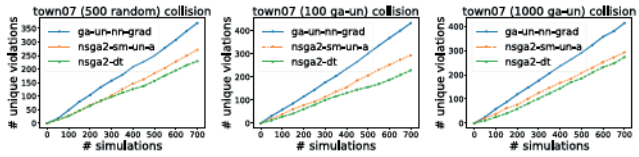


Fig. 8. RQ1. # unique violations under different initial seeds.

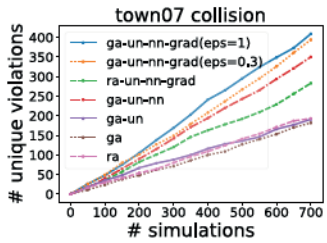


Fig. 9. #unique traffic violationsfound by *AutoFuzz*'s variants.

GA-UN-NN-GRADconsistently finds 10%-39% more than the best-performing baseline method. In particular, GA-UN-NN-GRADfinds 41, 51, 135 and 111 more unique traffic violationsover the second-best method in the four logical scenarios.

We further conduct Wilcoxon rank-sum test [55] and Vargha-Delaney effect size test [56], [57]. For all the settings, the 90% confidence interval of the effect size between GA-UN-NN-GRADand the best baseline is (0.834, 1.166) meaning large effect size, and the p-value is $3.95e^{-3}$ suggesting the gain of the proposed method is statistically significant.

After collecting all the violation-producing specific scenarios, we measure how many are truly unique as per our uniqueness criteria. GA-UN-NN-GRADand NSGA2-UN-SM-Awin by a large margin. For example, for the turning left non-signalized junction logical scenario, GA-UN-NN-GRADand NSGA2-UN-SM-Ahave 100% unique violations while the other three methods have only 42%, 22% and 10%. This is expected since they both have a duplicate elimination component inherent to the search strategy. The results show that the baselines NSGA2-SM, NSGA2-DT, and AV-FUZZER waste many resources by running similar violation-producing specific scenarios.

After introducing duplicate elimination (UN) and incremental learning (A), NSGA2-UN-SM-Afinds more violations than NSGA2-SM. But GA-UN-NN-GRADstill has advantages: (i) Our goal is to maximize the number of unique traffic violationsthan finding traffic violationswith the best Pareto front [6], [12], so a binary classification NN gives a better guide than multiple regression NNs. (ii) The constrained gradient-guided permutation gives a further boost. The second point is shown in the ablation study in RQ 2. Besides, we have observed that AV-FUZZERfinds much fewer traffic violations. It even finds fewer unique traffic violationsthan the seed collection stage (for which GA-UNis used) of other methods. The main reason is that AV-FUZZERhas very limited diversity exploration. In particular, its default mutation rate is small and its local GAstarts with the mutated duplicates of the global best scenario vector so far, both of which limit diversity. If the global best scenario vector does not change after several generations, all the local GAwill start with the same duplicates. Moreover, its resampling process picks the farthest scenario vectors from the existing ones but does not consider the distances among the selected scenario vectors, which results in restarting at a local cluster of scenario vectors with limited diversity.

Next, we study if GA-UN-NN-GRADcan effectively find more unique traffic violationsover baselines under different initial seeds. We compare the number of unique traffic violationsfound by GA-UN-NN-GRADwith NSGA2-UN-SM-Aand NSGA2-DTfor 700 simulations, assuming 500 initial seeds collected by RANDOM, and 100 and 1000 initial seeds collected by GA-UN, resp. As shown in Fig. 8, GA-UN-NN-GRADfinds 99, 139, and 121 more unique traffic violationsthan the baselines.

Result 1: *AutoFuzz finds hundreds of unique traffic violationsacross all three controllers. On average, it finds 9%-41% more unique violations over the second-best baseline.*

RQ2. Evaluating Design Choices. We study the influence of each component and choice of hyper-parameters on *AutoFuzz*. We present the results for the town07 logical scenario, with finding collisions as the objective. However, the observations also hold in general for other logical scenarios and objectives.

We conduct an ablation study on the impact of each GA-UN-NN-GRADcomponent, comparing the number of unique traffic violationsfound by GA-UN-NN-GRADwith the six variations shown in Table 3. Fig. 9 presents the results.

- GA-UN-NN-GRAD *($\epsilon = 1$ versus 0.3)*. With larger ϵ, slightly more violations are detected. A larger ϵ value can perturb the input with a larger magnitude. Thus, it can have more diverse seeds and reach a better optimum in terms of violations likelihood considered by the NN used for seed-selection and mutation.

- GA-UN-NN-GRAD*versus* RANDOM-UN-NN-GRAD. GA-UN-NN-GRADfinds more violations indicating the importance of the base sampling strategy.

- GA-UN-NN-GRAD*versus* GA-UN-NN*versus* GA-UN. GA-UN-NN-GRADfinds more unique violations than GA-UN-NNand GA-UN-NNbeats GA-UN. These show the necessity of the gradient-guided mutation component (GRAD) and seed selection component (NN). Furthermore, GA-UNfinds slightly more unique traffic violationsthan GA.

TABLE 4
# of Unique Violations Found Under Different th_2, th_1

(th_2, th_1)	GA-UN-NN-GRAD	NSGA2-UN-SM-A	NSGA2-DT
(5, 25)	175	110	138
(10, 25)	168	121	142
(20, 25)	161	109	131
(5, 50)	173	121	146
(10, 50)	169	131	92
(20, 50)	35	31	16
(5, 75)	26	16	1
(10, 75)	0	0	0
(20, 75)	0	0	0

We next explore the sensitivity of different search methods under nine different combinations of uniqueness thresholds, th_1 and th_2, as discussed in Section 5. We compare them for 300 simulations after the initial seed collection stage. The trend also holds for more simulations. Table 4 shows GA-UN-NN-GRADfinds at least 10-30% more unique traffic violationsthan the second-best baseline method under seven settings. For the setting (10, 75) and (20, 75), none of the methods can find new traffic violations. This is because the uniqueness constraint is too stringent, so the sampling component cannot find a valid sample that obeys the constraint.

Result 2: *Each component of* GA-UN-NN-GRAD*contributes to the final superior performance and combined they find more unique traffic violationscompared to all other settings.*

RQ3. Evaluating Impact on Repair. Since the purpose of finding erroneous behavior in any software is to help with removing the errors, we speculated whether we can leverage the traffic violationsfound to improve a controller to reduce future traffic violations. We focus on the collisions found for four logical scenarios. For each one, we randomly select 200 detected traffic violationsby GA-UN-NN-GRADfor **lbc**, and split the corresponding specific scenarios into 100 for retraining and 100 for testing. We use **pid-1** as a teacher model to run the 100 specific scenarios for retraining and collect the camera data where it finishes successfully. The collected camera images are down-sampled to two frames per sec (about 2000 images) and use them to fine-tune the **lbc** model for one epoch. Finally, we test the retrained model on the held-out 100 previously failing specific scenarios. Table 5 shows that the retrained controller succeeds in over 75% of the originally failing specific scenarios.

Result 3: *In our preliminary study, retraining with traffic violationsfound by AutoFuzzimproved the* **lbc***controller's performance on failure cases by 75% to 100%.*

RQ4. Evaluating Generalizability.

In Section 7 we reported experimental results based on a single simulator, CARLA, and three research-oriented controllers. To evaluate the generalizability of *AutoFuzz*, we conduct a preliminary study on APOLLO6.0, an industrial-grade

TABLE 5
# of Violations Fixed in the Held-Out Dataset

logical scenarios names	# retraining data	# violations fixed
turning right while leading car slows down	64	82 / 100
turning left non-signalized	47	76 / 100
crossing non-signalized	91	100 / 100
changing lane	64	75 / 100

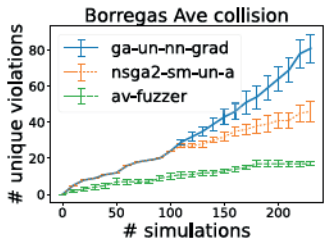


Fig. 10. RQ4. average #unique collision traffic violations.

AV controller [21], using a different simulator, SVL(version 2021.3) [23], [58]. We analyze the SVLAPI similarly to CARLA (Section 4.2) and focus on collision traffic violations(Section 4.3.1). We use a logical scenario where the ego car conducts a left turn at a signalized junction while another vehicle comes from the other side and a pedestrian crosses the street. Since the search space has 11 parameters (we do not consider parameters like weather and lighting since their implementations in SVLdo not influence LiDAR which APOLLO6.0mostly relies on for its perception module) to search for, to speed up the convergence of the search process, we reduce the population size to 10. All other hyperparameters and settings are kept the same as in RQ1. We run *AutoFuzz* and the best performing baseline NSGA2-UN-SM-Afor 14 generations totaling 140 simulations (excluding an initial 100 simulations for the seed collection stage) and run AV-FUZZERfor 240 simulations (since it does not have a seed collection stage). We then compare them over the entire 240 simulations. As shown in Fig. 10, on average of six repetitions, GA-UN-NN-GRADfinds 76 unique traffic violations— which is 49% and 375% more, respectively, than the two baseline methods NSGA2-UN-SM-Aand AV-FUZZER(51 and 16 unique traffic violations, resp.). We further conduct Wilcoxon rank-sum test and Vargha-Delaney effect size test. The 90% confidence interval of the effect size between GA-UN-NN-GRADand the best baseline is (0.807, 1.165) meaning large effect size, and the p-value is $5.07e^{-3}$ suggesting the gain of the proposed method is statistically significant.

Fig. 11 shows two examplary Apollo traffic violationsfound by *AutoFuzz*: the ego car turning left collides with a pedestrian crossing the street and an incoming truck, respectively. They expose different functionality errors: fail to avoid a pedestrian and fail to avoid a truck, respectively. An investigation of the two violations identify their different causes. In the pedestrian collision case, the ego car's detection of the pedestrian is too late and thus the ego car does not have enough time to stop. In the truck collision

Fig. 11. Two traffic violationsfound for APOLLO6.0in SVL. (1st row) The ego-car turning left collides with a pedestrian crossing the street. (2nd row) The ego-car turning left collides with an incoming truck.

case, the ego car detects the truck stably but does not plan its speed properly.

Result 2: *AutoFuzzcan generalize beyond* CARLA. *In particular, it can find more unique traffic violationsthan the baseline methods for* APOLLO6.0*in* SVL.

8 Related Work

Section 2.2 presents the work most related to this paper. This section covers other peripheral works.

Grammar-Based Fuzzing. Fuzzing produces input variations and tries to find failure cases for the software under test [59], [60]. Fuzzing tends to work well with relatively simple input formats such as image [61] or audio [62]. For more complex input formats such as cloud service APIs [63] or language compilers [64], researchers often use grammar-based fuzzing [65], [66]

to obey domain-specific constraints and narrow down the search space for producing effective and valid inputs.

Language Specification and Testing. OpenScenario [67] is an open file format for describing the dynamic contents of driving simulations at a logical level [68], but it is at an early stage. GeoScenario [69] provides a language describing a specific scenario to be simulated; [70] develops a simulation-based testing framework for AV. Neither provides a parametric search space that can be easily fuzzed. In contrast, we parameterize functional scenarios that allows users to specify the range and distributions of parameters and their constraints for automatically finding traffic violations.

9 Discussion & Threats to Validity

Realism. Our evaluation results are limited by the simulator implementations. Some reported traffic violationsmight be due to interactions between the simulator and controller, e.g., message passing delays, rather than the controller itself. To mitigate this threat to internal validity, we experimented with two simulators (CARLAand SVL) and four different controllers (lbc, pid1, pid2, APOLLO6.0). Further, to make the simulated crashes close to the real world, we construct logical scenariosbased on the most frequent pre-crash functional scenariosfrom an NHTSA report. The example shown in Fig. 12 is a complex high-dimensional (328d) scenario with many agents. Since to fully consider the temporal development (e.g., specifying the location of a vehicle at every time step), the search space can grow quickly and makes the searching process intractable, and during most accidents the movements of the involved vehicles/pedestrians can usually be decomposed into a couple of atomic behaviors, we currently consider one behavior development in CARLAand only the initial state (e.g., location, orientation, and speed) in SVL. The integration of more temporal developments into *AutoFuzz* is relatively easy. Besides, given our fuzzing strategy's black-box nature, the additional behavior developments should only have limited influence.

Fig. 12. An example of traffic violationin a high-dimensional scenario: the AV (controlled by lbc) collides with a child crossing street.

Road Infrastructures. The road infrastructures considered in the current work are the default ones in the built-in maps in the simulators, which are mostly modeled based on the current road infrastructures in the United States. Different road infrastructures (e.g., those designed for deploying AVs) can influence the behavior of the AV under test [71], [72]. However, the public road infrastructures with support for connected autonomous vehicles (CAVs) are not yet available and may not be available for quite some time. Nevertheless, there are not-connected AVs on conventional public roads now [73], some of which led to fatal accidents [74]. Thus, it is necessary to study traffic violations by individual AVs on the current road infrastructures. We leave an exploration of road infrastructures with the support for CAVs for future work.

Unique Violations. The uniqueness of traffic violationsis hard to define precisely. We mitigate this threat to construct validity by extending the definition used in [6] with additional configurable parameters th_1 and th_2, enabling users to control uniqueness stringency. A more desirable definition might be based on the internal system fault causing the violation. For example, two traffic violationscan be considered distinct if one is due to a failure of detecting a pedestrian for 2 seconds and the other is due to a sub-optimal tracking for 5 seconds). However, this is not feasible in our black-box testing setting where we assume no knowledge of the system under test. Besides, general methods to locate the root cause for a violation is itself an open question since it is non-trivial to assign the responsibility of a violation to different components of an AV at different time steps and the AV under test can have drastically different sub-components (e.g., lbc is an end-to-end neural network based system while APOLLO6.0is a modular based system). Another desirable definition might be search space causal related, e.g., only variables interacting with the ego car or that have an impact on ego car behavior count. However, efficiently determining the features contributing to a failure behavior is still an open challenge. One idea is to keep all other features fixed while changing the value of one feature and observe whether the failure behavior persists. If so, that feature can be potentially considered

unrelated. This method faces some major limitations: First, as the number of features and the range for each feature become large, it is practically infeasible to conduct such analysis within a given time budget. Second, the features may not be independent and changing them one-by-one will miss the dependencies. Third, there is no consensus on quantifying if the causes of two failure cases are the same. For example, a car may collide with a pedestrian at slightly different locations for two simulations. Should we consider the cause to stay the same? It might be worth looking into the behavior of the controller's internal states, which goes beyond the ability of a black-box testing framework. Because of these challenges, we leave an in-depth study of this topic for future work.

*Policy Implication:*Public officials should be educated on the severe implications of studies like our own, and urged to make a comprehensive AVsafety testing standard. AVs must be shown to satisfy the safety requirements in the necessary testing process (e.g., having acceptably few traffic violations) in order to be allowed to deploy on the public roads.

10 Conclusion

We present *AutoFuzz*, a grammar-based fuzzing technique for finding traffic violationsin AVcontrollers during simulation-based testing. A traffic violationindicates a flaw in the controller that needs to be fixed. *AutoFuzz* leverages the simulator's API specification to generate inputs (seed scenes) from which the simulator will generate semantically and temporally valid specific scenarios. It performs an NN-guided evolutionary search over the API grammar, seeking seeds that lead to distinct traffic violations. Evaluation of our prototype implementation on four AVcontrollers shows that *AutoFuzz* successfully finds hundreds of realistic unique traffic violationsresembling complex real-world crashes and other driving offenses, outperforming the baseline methods. Furthermore, we leverage traffic violationsfound to improve a learning-based controller's behavior on similar cases.

Acknowledgments

The authors thank Suman Jana and Dongdong She from Columbia University for valuable discussions.

References

[1] D. O. M. V. State of California, "Autonomous vehicle testing permit holders," 2020. [Online]. Available: https://www.dmv.ca.gov/portal/vehicle-industry-services/autonomous-vehicles/autonomous-vehicle-testing-permit-holders/

[2] State of California, Department of Motor Vehicles, 2020. [Online]. Available: https://www.dmv.ca.gov/portal/vehicle-industry-services/autonomous-vehicles/autonomous-vehicle-collision-reports/

[3] J. Norden, M. O'Kelly, and A. Sinha, "Efficient black-box assessment of autonomous vehicle safety," in *Proc. Mach. Learn. Auton. Driving Workshop 33rd Conf. Neural Inf. Process. Syst.*, 2019.

[4] A. Gambi, T. Huynh, and G. Fraser, "Generating effective test cases for self-driving cars from police reports," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2019, pp. 257–267.

[5] Y. Abeysirigoonawardena, F. Shkurti, and G. Dudek, "Generating adversarial driving scenarios in high-fidelity simulators," in *Proc. Int. Conf. Robot. Automat.*, 2019, pp. 8271–8277.

[6] R. B. Abdessalem, S. Nejati, L. C. Briand, and T. Stifter, "Testing vision-based control systems using learnable evolutionary algorithms," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 1016–1026.

[7] G. WassimJohn, D. NajmSmith, and M. Yanagisawa, "Pre-crash scenario typology for crash avoidance research," Nat. Highway Trans. Saf. Admin., Washington, DC, USA, Tech. Rep. DOT-HS-810 767, Apr. 2007.

[8] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. Reading, MA, USA: Addison-Wesley, 2007.

[9] B. Miller, M. Zhang, and E. Heymann, "The relevance of classic fuzz testing: Have we solved this one?," *IEEE Trans. Softw. Eng.*, vol. 48, no. 6, pp. 2028–2039, Jun. 2022.

[10] R. B. Abdessalem, A. Panichella, S. Nejati, L. C. Briand, and T. Stifter, "Testing autonomous cars for feature interaction failures using many-objective search," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2018, pp. 143–154.

[11] S. Kuutti, S. Fallah, and R. Bowden, "Training adversarial agents to exploit weaknesses in deep control policies," 2020, *arXiv:2002.12078*.

[12] R. Ben Abdessalem, S. Nejati, L. C. Briand, and T. Stifter, "Testing advanced driver assistance systems using multi-objective search and neural networks," in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2016, pp. 63–74.

[13] D. She, R. Krishna, L. Yan, S. Jana, and B. Ray, "Mtfuzz: Fuzzing with a multi-task neural network," in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Foundations Softw. Eng.*, 2020, pp. 737–749.

[14] S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd ed. Englewood Cliffs, NJ, USA: Prentice Hall, 1998.

[15] I. J. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016, http://www.deeplearningbook.org.

[16] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "NEUZZ: Efficient fuzzing with neural program learning," in *Proc. IEEE Symp. Secur. Privacy*, 2019, pp. 803–817.

[17] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," in *Proc. 6th Int. Conf. Learn. Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=rJzIBfZAb

[18] G. Li et al., "AV-FUZZER: Finding safety violations in autonomous driving systems," in *Proc. IEEE 31st Int. Symp. Softw. Rel. Eng.*, 2020, pp. 25–36.

[19] D. Chen, B. Zhou, V. Koltun, and P. Krähenbühl, "Learning by cheating," in *Proc. 3rd Conf. Robot Learn.*, 2019, pp. 66–75. [Online]. Available: http://proceedings.mlr.press/v100/chen20a.html

[20] W. Ding, B. Chen, M. Xu, and D. Zhao, "Learning to collide: An adaptive safety-critical scenarios generating method," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2020, pp. 2243–2250.

[21] Baidu, "Apollo: 2021. [Online]. Available: An open autonomous driving platform," https://github.com/ApolloAuto/apollo

[22] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," in *Proc. Conf. Mach. Learn.*, 2017, pp. 1–16. [Online]. Available: http://proceedings.mlr.press/v78/dosovitskiy17a.html

[23] LG Electronics, "SVL Simulator: An autonomous vehicle simulator, A ROS/ROS2 multi-robot simulator for autonomous vehicles," 2021. [Online]. Available: https://github.com/lgsvl/simulator

[24] Z. Zhong, G. Kaiser, and B. Ray, "autofuzz2020/autofuzz: v0.0.1," Mar. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.6399383

[25] S. Ulbrich, T. Menzel, A. Reschka, F. Schuldt, and M. Maurer, "Defining and substantiating the terms scene, situation, and scenario for automated driving," in *Proc. IEEE 18th Int. Conf. Intell. Transp. Syst.*, 2015, pp. 982–988.

[26] PEGASUS RESEARCH PROJECT, SCENARIO DESCRIPTION, 2019. [Online]. Available: https://www.pegasusprojekt.de/files/tmpl/PDF-Symposium/04_Scenario-Description.pdf

[27] Y. Zhao, H. Zhu, R. Liang, Q. Shen, S. Zhang, and K. Chen, "Seeing isn't believing: Towards more robust adversarial attack against real world object detectors," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2019, pp. 1989–2004.

[28] H. Zhou et al., "Deepbillboard: Systematic physical-world testing of autonomous driving systems," 2018, *arXiv:1812.10812*.

[29] Y. Jia et al., "Fooling detection alone is not enough: Adversarial attack against multiple object tracking," in *Proc. Int. Conf. Learn. Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=rJl31TNYPr

[30] Y. Cao et al., "Adversarial sensor attack on lidar-based perception in autonomous driving," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2019, pp. 2267–2281. [Online]. Available: https://doi.org/10.1145/3319535.3339815
[31] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 303–314.
[32] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, "Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems," in *Proc. 33rd IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2018, pp. 132–142.
[33] J. Tu et al., "Exploring adversarial robustness of multi-sensor perception systems in self driving," *Proc. 5th Conf. Robot Learn.*, vol. 164, pp. 1013–1024, 2021.
[34] K. Wong et al., "Testing the safety of self-driving vehicles by simulating perception and prediction," in *Proc. Eur. Conf. Comput. Vis.*, 2020, pp. 312–329.
[35] A. Calò, P. Arcaini, S. Ali, F. Hauer, and F. Ishikawa, "Generating avoidable collision scenarios for testing autonomous driving systems," in *Proc. IEEE 13th Int. Conf. Softw. Testing Validation Verification*, 2020, pp. 375–386.
[36] P. Arcaini, X.-Y. Zhang, and F. Ishikawa, "Targeting patterns of driving characteristics in testing autonomous driving systems," in *Proc. 14th IEEE Conf. Softw. Testing Verification Validation*), 2021, pp. 295–305.
[37] Y. Luo et al., "Targeting requirements violations of autonomous driving systems by dynamic evolutionary search," in *Proc. 36th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2021, pp. 279–291.
[38] J. Shen, J. Won, Z. Chen, and Q. A. Chen, "Drift with devil: Security of multi-sensor fusion based localization in high-level autonomous driving under GPS spoofing," in *Proc. 29th USENIX Secur. Symp.*, 2020, pp. 931–948. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/shen
[39] B. Chen, X. Chen, Q. Wu, and L. Li, "Adversarial evaluation of autonomous vehicles in lane-change scenarios," *Adv. Eval. Auton. Veh. Lane-Change Scenarios*, pp. 1–10, 2020, doi: 10.1109/TITS.2021.3091477.
[40] W. Ding, B. Chen, B. Li, K. J. Eun, and D. Zhao, "Multimodal safety-critical scenarios generation for decision-making algorithms evaluation," 2020, *arXiv:2009.08311*.
[41] A. Gambi, M. Mueller, and G. Fraser, "Automatically testing self-driving cars with search-based procedural content generation," in *Proc. 28th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2019, pp. 318–328.
[42] Z. Zhong, Z. Hu, S. Guo, X. Zhang, Z. Zhong, and B. Ray, "Detecting safety problems of multi-sensor fusion in autonomous driving," 2021, *arXiv:2109.06404*.
[43] C. team, "Scenariorunner for carla," 2020. [Online]. Available: https://github.com/carla-simulator/scenario_runner
[44] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," *IEEE Trans. Softw. Eng.*, vol. 45, no. 5, pp. 489–506, May 2019.
[45] C. Lemieux and K. Sen, "FairFuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage," in *Proc. 33rd IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2018, pp. 475–485.
[46] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Proc. IEEE Symp. Secur. Privacy*, 2018, pp. 711–725.
[47] M. Zalewski, "American fuzzy lop," 2017. [Online]. Available: http://lcamtuf. coredump.cx/afl
[48] T. Yue et al., "Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit," in *Proc. 29th {USENIX} Secur. Symp.*, 2020, pp. 2307–2324.
[49] M. Böhme, V. J. Manès, and S. K. Cha, "Boosting fuzzer efficiency: An information theoretic perspective," in *Proc. 28th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2020, pp. 678–689.
[50] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-II," *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, pp. 182–197, Apr. 2002.
[51] R. Agrawal, K. Deb, and R. Agrawal, "Simulated binary crossover for continuous search space," *Complex Syst.*, vol. 9, pp. 115–148, 2000.
[52] K. Deb and S. Agrawal, "A niched-penalty approach for constraint handling in genetic algorithms," in *Artificial Neural Nets and Genetic Algorithms*. Berlin, Germany: Springer, 1999, pp. 235–243
[53] J. Li, J. Lee, Y. Yang, J. Sun, and K. Tomsovic, "Conaml: Constrained adversarial machine learning for cyber-physical systems," 2020, *arXiv:2003.05631*.
[54] J. Blank and K. Deb, "Pymoo: Multi-objective optimization in python," *IEEE Access*, vol. 8, pp. 89497–89509, 2020.
[55] J. A. Capon., *Elementary Statistics for the Social Sciences: Study Guide*. Belmont, CA, USA: Wadsworth Publishing Company, 1991.
[56] A. Vargha and H. D. Delaney, "A critique and improvement of the CL common language effect size statistics of mcgraw and wong," *J. Educ. Behav. Statist.*, vol. 25, no. 2, pp. 101–132, 2000.
[57] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Softw. Testing Verification Rel.*, vol. 24, no. 3, pp. 219–250, 2014.
[58] G. Rong et al., "LGSVL simulator: A high fidelity simulator for autonomous driving," in *Proc. 24th IEEE Int. Conf. Intell. Transp.*, 2021, pp. 1–6.
[59] C. Hutchison et al., "Robustness testing of autonomy software," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng.: Softw. Eng. Pract. Track*, 2018, pp. 276–285.
[60] T. Dreossi et al., "VERIFAI: A toolkit for the design and analysis of artificial intelligence-based systems," *Comput. Aided Verification*, 2019.
[61] Joint Photographic Experts Group, Overview of JPEG 1, 1992. [Online]. Available: https://jpeg.org/jpeg/
[62] Sustainability of Digital Formats, "Planning for library of congress collections. MP3 (MPEG layer III audio encoding)," 1993. [Online]. Available: https://www.loc.gov/preservation/digital/formats/fdd/fdd000012.shtml
[63] V. Atlidakis, R. Geambasu, P. Godefroid, M. Polishchuk, and B. Ray, "Pythia: Grammar-based fuzzing of REST APIs with coverage-guided feedback and learning-based mutations," 2020. [Online]. Available: https://arxiv.org/abs/2005.11498
[64] Free Software Foundation, GCC, the GNU Compiler Collection, 1987. [Online]. Available: https://gcc.gnu.org
[65] M. Eberlein, Y. Noller, T. Vogel, and L. Grunske, "Evolutionary grammar-based fuzzing," in *Search-Based Software Engineering*, A. Aleti and A. Panichella, Eds., vol. 12420. Berlin, Germany: Springer, 2020.
[66] E. Soremekun, E. Pavese, N. Havrikov, L. Grunske, and A. Zeller, "Inputs from hell learning input distributions for grammar-based test generation," *IEEE Trans. Softw. Eng.*, vol. 48, no. 4, pp. 1138–1153, Apr. 2022.
[67] ASAM, "OpenScenario," 2021. [Online]. Available: https://www.asam.net/standards/detail/openscenario
[68] T. Menzel, G. Bagschik, and M. Maurer, "Scenarios for development, test and validation of automated vehicles," in *Proc. IEEE Intell. Veh. Symp.*, 2018, pp. 1821–1827.
[69] R. Queiroz, T. Berger, and K. Czarnecki, "GeoScenario: An open DSL for autonomous driving scenario representation," in *Proc. IEEE Intell. Veh. Symp.*, 2019, pp. 287–294.
[70] T. Duy Son, A. Bhave, and H. Van der Auweraer, "Simulation-based testing framework for autonomous driving development," in *Proc. IEEE Int. Conf. Mechatronics*, 2019, pp. 576–583.
[71] T. U. Saeed, "Road infrastructure readiness for autonomous vehicles," 2019. [Online]. Available: https://hammer.purdue.edu/articles/thesis/Road_Infrastructure_Readiness_for_Autonomous_Vehicles/8949011
[72] H. Lengyel, T. Tettamanti, and Z. Szalay, "Conflicts of automated driving with conventional traffic infrastructure," *IEEE Access*, vol. 8, pp. 163 280–163 297, 2020.
[73] Nhtsa av test initiative - test tracking tool, 2022. [Online]. Available: https://www.nhtsa.gov/automated-vehicle-test-tracking-tool
[74] Tesla deaths, 2022. [Online]. Available: https://www.teslad eaths.com/

Ziyuan Zhong is currently a PhD student in the Department of Computer Science, Columbia University. His current research mainly focuses on testing/improving Autonomous Driving Systems (ADSs) and robustness of deep learning models. Previously, he did undergrads at Reed College and Columbia University.

Gail Kaiser (Senior Member, IEEE) is a professor of Computer Science at Columbia University. She received her ScB in Computer Science and Engineering from Massachusetts Institute of Technology, her MS in Computer Science from Carnegie Mellon University, and her PhD in Computer Science from Carnegie Mellon University.

Baishakhi Ray (Member, IEEE) is an associate professor in the Department of Computer Science, Columbia University, NY, USA. She has received her PhD degree from the University of Texas, Austin. Baishakhi's research interest is in the intersection of Software Engineering and Machine Learning. Baishakhi has received Best Paper awards at FASE 2020, FSE 2017, MSR 2017, IEEE Symposium on Security and Privacy (Oak- land), 2014. Her research has also been published in CACM Research Highlights and has been widely covered in trade media. She is a recipient of the NSF CAREER award, VMware Early Career Faculty Award, IBM Faculty Award, and IEEE CS TCSE Rising Star Award.