

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

Ziyuan Zhong, Gail Kaiser, Baishakhi Ray

提出一种新的模糊测试技术，称为AutoFuzz，它可以利用AV模拟器的API语法，生成语义和时间上有效的复杂假设环境（场景序列）

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), *AutoFuzz* is guided by a constrained Neural Network (NN) evolutionary search over the API grammar to generate scenarios seeking to find unique traffic violations. Evaluation of our prototype on one state-of-the-art learning-based controller, two rule-based controllers, and one industrial-grade controller shows that *AutoFuzz* efficiently finds hundreds of traffic violations in high-fidelity simulation environments. Further, fine-tuning the learning-based controller with the traffic violations found by *AutoFuzz* successfully reduced the traffic violations found in the new version of the AV controller software.

Index Terms—fuzz testing, self-driving cars, test generation

AutoFuzz由对API语法受限神经网络进化搜索引导，以生成寻求发现独特交通违规的场景。

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 AV for 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], [7], [8], [9]. 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 real-life situations that may lead to traffic violations — 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 the ego car from

TABLE 1: Dominant Scenarios Leading to Car Crashes as per National Highway Traffic Safety Administration (NHTSA) report [10].

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

The car controlled by the user (through physical controls or an AV software) is commonly called the '*ego car*'. '*Without taking any action*' here means the ego car is going straight or negotiating a curve rather than explicitly making turns / changing lanes / leaving a parking position.

behind. However, simulating such involved crash scenarios is non-trivial, especially because the ego car can interact with its surroundings (driving path, road condition, weather, stationery, and moving agents, etc.) in an exponentially large number of ways. Yet, simulating *some* crash-inducing scenario, 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 violations can be reported with slight variations of essentially the same situation (*e.g.*, changing an 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) [11], [12], [13] 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 AV is not straightforward, as the

• Z. Zhong, G. Kaiser and B. Ray are with the Department of Computer Science, Columbia University, New York, NY, 10025. E-mail: ziyuan.zhong@columbia.edu, kaiser@cs.columbia.edu, rayb@cs.columbia.edu

因此，成功的基于模拟的测试框架的要求之一是模拟可能导致许多不同违规的场景

模拟器将一些有效的初始场景作为输入（包括：路线图，起始位置，目的地，egoCar，其他汽车和行人的位置、方向和速度）

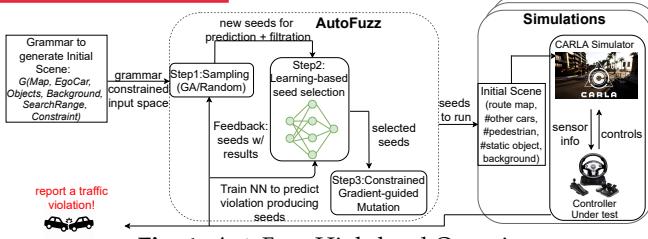


Fig. 1: AutoFuzz High-level Overview

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 spent on generating and validating these invalid test cases 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*, illustrated in Figure 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 testing AV systems [7], [8], [9], [14], we formulate the fuzzing process as a smooth multi-objective search that guides the ego car approach to the point of interest.

To quantify the notion of traffic violation diversity, 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 [15], as successful mutants are often limited in a sparse high-dimensional space, and chances of finding them without any guidance are thin. To address this, 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 violations and 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 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) [16] 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. Essentially, 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.

This paper makes the following contributions:

- We introduce *AutoFuzz*, a grammar-based fuzzing technique to test AV controllers, which leverages the simulator's API specification to generate semantically valid test scenarios.
- We optimize *AutoFuzz* to find unique traffic violations using a novel learning-based seed selection and mutation strategy.
- We evaluate our *AutoFuzz* prototype for the widely used CARLA simulator [17], on one end-to-end learning-based AV controller [18] and two rule-based controllers [6], [18], and report hundreds of traffic violations.
- We reduce traffic violations by 75-100% for the learning-based controller by fine-tuning it with the traffic violation-producing test cases.
- We show that *AutoFuzz* can also find traffic violations using another simulator SVL [19], [20] with an industrial-grade controller APOLLO6.0 [21].

Contribution to SE Field. This paper is core to the software testing field, particularly test generation, in our case for testing self-driving car controllers. We also show the potential of improving the controller software, thus contributing to the automated software repair literature. We hope this paper will overall improve the reliability of AVs.

2 BACKGROUND

2.1 Definitions

First, we define a few terms, most of them taken from [22], [23]:

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 5 m/s facing north on a rainy afternoon. A **Scenario** is “the temporal development between several scenes in a sequence of scenes” [22]. 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 to simulate sampling from the logical search space, *e.g.*, the ego car crossing the intersection will start at (10, 10) and end at (30, 30). A specific scenario usually 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. For example, Waymo has tested its cars on public roads for 20 million miles from 2009 to 2018 [24], which is far less than the average yearly total driving distance in the U.S. (3 trillion miles per year) [25], [26]. However, as per Table 1, many pre-crash functional scenarios may only occur in certain corner cases, *i.e.*, variations in road conditions, background buildings, weather, lighting, the behaviors of other vehicles and pedestrians, *etc.* It is extremely difficult to focus real-world testing towards such rare events.

Single component testing primarily focuses on the perception component [27], [28], [29], [30], [31], [32], [33] or the planning component [34], [35], [36]. The works for the perception component differ on the place perturbed: road sign [27], [28], billboard [29], LiDAR input [30], [37], camera image [31], [32], [33]), LiDAR and camera image [38], and the target they attack: perception [27], [28], [29], [30], [31], motion planing [39], lane following controller [32], [33]. The works for the planning component differ on the characteristics of the scenarios to look for: avoidable collisions [34], patterns satisfaction [35], and requirements violation [36]. However, this line of research tends to miss more involved interactions between different components [40].

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. There are three main ways: (i) constructing a known-hard testing-specific scenario [4], (ii) adding noise to sensor inputs [41], and (iii) searching known-hard specific scenarios in a parameterized logical scenario space [3], [5], [6], [7], [8], [9], [14], [42], [43], [44], [45], [46], [47], [48], [49], [50], [51]. Gambi et al. [4] create simulations that reproduce specific scenarios according to the functional scenarios leading 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. Han et al. [41] apply fuzz testing by randomly adding static boxes into the controller’s sensor. Such tests cannot capture dynamic agents, *e.g.*, pedestrians. Many works of the third category usually model the logical scenario with 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 types of traffic violations like going off-road. Furthermore, the search methods used, *e.g.*, adaptive sampling [3], [44], [45], bayesian optimization [5], topic modeling [6], reinforcement learning [14], [46], [47], flow-based density estimation [48] tend to be either highly sensitive to hyper-parameters and proposal distributions [3], [44], [45] or not scale well to high-dimensional search space [5], [6], [46], [47], [48].

Among these, perhaps the closest to our work are evolutionary-based algorithms [9], [42], [43], [50] and their variants (with NN [8] or Decision Tree [7] for seed filtration) on testing AV or 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) [7], Pedestrian Detection Vision based (PeVi) [8], OpenPilot [42], and an integration component [9]) under one particular logical scenario, testing a controller on road networks without any additional elements (*e.g.*, weather, obstacle, and traffic) [43], or focusing on finding collision accidents in a specific scenario with other cars constantly changing lanes [50]. Nevertheless, we adapt the algorithms from [7], [8], [50] in our setting, and compare with *AutoFuzz*.

2.3 Motivating Example

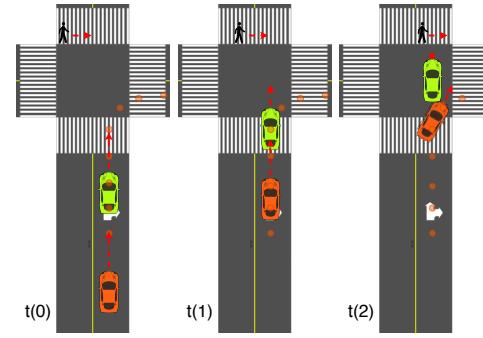


Fig. 2: Example of Crash Simulation in consecutive time steps.

AutoFuzz aims to generate traffic violations by an ego car controller by fuzzing the input scenes. *AutoFuzz* starts with a logical driving scenario that involves traffic violations, designed based on the top pre-crash functional scenarios from NHSTA [10] (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 AV behaves in such situations. Figure 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 (Figure 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 (Figure 2-t(1)). This leads the ego car to collide with the leading car (Figure 2-t(2)). To simulate the collision, *AutoFuzz* leverages CARLA’s APIs related to vehicle, pedestrian, and cross-road in the map. 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.

3 API GRAMMAR

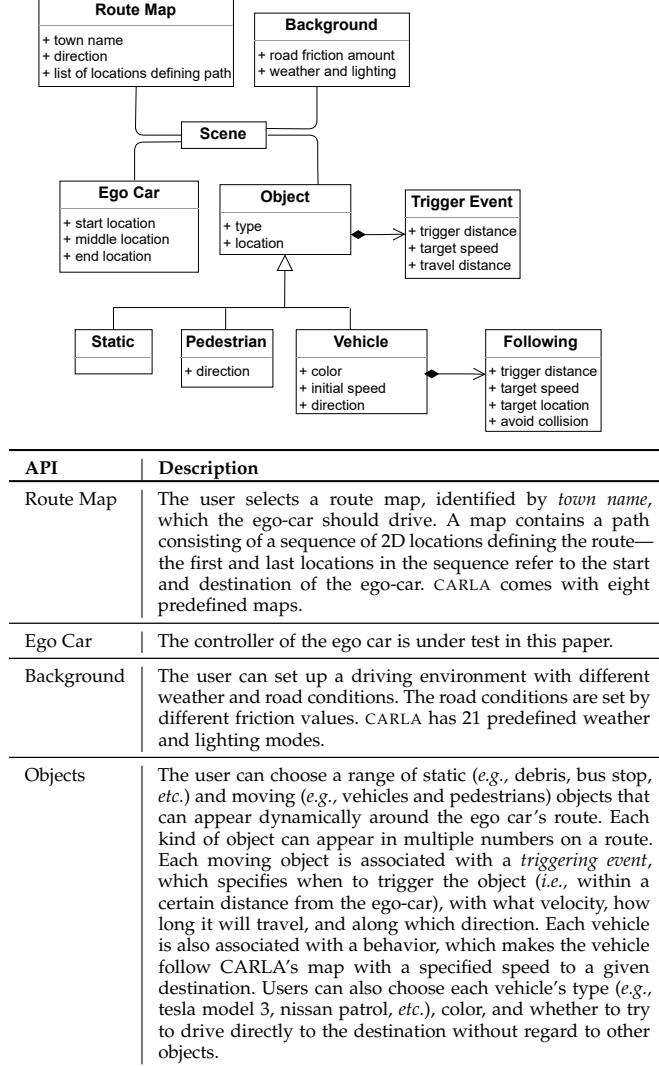


Fig. 3: A simplified description of CARLA’s APIs. We fuzz only over the background and objects.

Figure 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.

CARLA provides the API specifications as a set of Python APIs [17], [52]. For example, calling *CarlaDataProvider.request_new_actor(pedestrian_model, spawn_point)* creates a pedestrian, where *pedestrian_model* is a pedestrian asset predefined in CARLA and *spawn_point* specifies the pedestrian’s initial location and direction. From such specifications we construct a test-generation grammar, $\mathcal{G}(\text{Map}, \text{Ego Car}, \text{Objects}, \text{Background})$,

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
  }
]
```

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 $\mathcal{G}(\text{Map}, \text{Ego Car}, \text{Objects}, \text{Background}, \text{Search Range}, \text{Constraint})$, where the underlined components that facilitate fuzzing are optional. The details of search range and constraints are provided in Appendix D in Supplementary Material.

4 METHODOLOGY

Leveraging the API grammar as described in section 3, *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 increase the potential of producing 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 vehicles, cyclists/motorcycles, pedestrians 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 violations is 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 redundancy is not interesting or useful. Thus, criteria for precisely defining *unique* traffic violations is needed.

Abdessalem *et al.* [7] 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 $th_2\% = 15\%$.

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, which is 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 the fuzzing context); Each sampled initial scene obeys the constraints enforced by the API grammar.

Figure 1 shows a high-level overview of the fuzzing process. The objective is to search for initial scenes that will lead to unique traffic violations. To achieve this, like common blackbox fuzzers, *AutoFuzz* runs iteratively: *AutoFuzz* samples the grammatically valid initial scenes (Step-I), 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 violations found 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 violations found 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 violation count (\mathcal{Y}) within a fixed budget,

say \mathcal{T} . Thus, if B_t is the set of traffic violations found by input $x_t \in \mathcal{X}$ at fuzzing step t , then more formally fuzzing is: $\mathcal{Y}_{\mathcal{T}} = \max \|\cup_{t=1}^{\mathcal{T}} B_t\|$. Here $\|\cdot\|$ is the norm and $\cup(\cdot)$ 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 [53]? and
- (iii) *Mutation(m)*: How to mutate [54], [55], [56], [57]?

Thus, the next generated input at time t , x_t depends on $(x_{:t-1}, m)$, where $x_{:t-1} := x_1, \dots, x_{t-1}$. 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:

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

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

Collision. We optimize for the weighted sum of the three smooth objective functions: $F_{\text{collision}}$, F_{object} , and F_{view} , similar to the objectives used in [7], [8], [9]. $F_{\text{collision}}$ and F_{object} promote the severity of collision and the chance of collision, respectively. $F_{\text{collision}}$ is set to -1 as per [7] 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: $F_{\text{wronglane}}$, F_{offroad} , and $F_{\text{deviation}}$. $F_{\text{deviation}}$ is adapted from the objective of "maximum distance deviated from lane center" in [14].

We further define $F_{\text{wronglane}}$ and F_{offroad} to strengthen the signals for driving into an incorrect lane or off the road, respectively. Figure 14 in Section E provides an illustration.

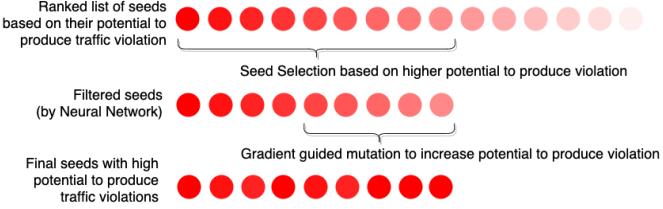


Fig. 4: Seed Selection & Mutation Strategy per Generation

For each traffic violation type, we formulate the fuzzing problem as a constrained multi-objective optimization problem. Let x be an input, *i.e.*, a specific scenarios with all the searchable fields. Denote $F_i(x)$ for $i = 1, \dots, n$ to be n objective functions, w_i to be some user-provided weights, and $g_j(x)$ for $j = 1, \dots, p$ to be p constraints, where each constraint is expressed as ≤ 0 form. Then, the objective function $F(x)$ of Equation (1) can be expressed as a constrained weighted sum: $\min_x \sum_{i=1}^n w_i F_i(x), \text{s.t. } g_j(x) \leq 0 \forall j = 1, \dots, p$. Unlike [7], [8], 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 violations rather than traffic violations with the Pareto front of multiple objectives.

4.3.2 Seed Selection.

Common evolutionary fuzzers like AFL [58] 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 [59], [60]. 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 Figure 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 vs. 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. Figure 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 Figure 4). *AutoFuzz* further mutates such lower confidence 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 Figure 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 Figure 1). Appendix A - Algorithm 2 in Supplementary Material 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 have very small probabilities 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* [9].

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 NSGA2 implementation [61], as well as the variations in [7], [8]. 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 [62], a classical crossover method commonly used for floating point numbers, is adopted, as in [7], [8], [61]. 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 [63]. For discrete variables, we treat the value as continuous during the mutation and round later. We clip the values at specified boundary values. Following

[7], 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.

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 violation and the most likely tests are selected.

Some previous work [8] also leverages an NN for seed selection. There are several major differences. First, we train a single NN for binary classification of traffic violations rather than several NNs for regressing over all objective values as in [8]. Thus we rank test cases based on the confidence value of finding a traffic violation rather than the Pareto front from multiple NNs. This design choice is motivated by our goal to find maximum number of valid, diverse traffic violations rather than finding the best set of traffic violations achieving 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.

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 [64].

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, th_{conf1} : threshold of conducting a perturbation, th_{conf2} : threshold of stopping a perturbation, n : maximum number of iterations, λ : step size, c : constraints, ϵ : maximum perturbation bound, x_{min} : minimum allowable input values, x_{max} : maximum allowable input values

Output: x' : mutated test cases

```

1  $x' = x;$ 
2  $i = 0;$ 
3 if  $f(x) > th_{conf1}$  then
4   | return  $x$ ;
5 end
6 while  $i < n$  do
7   |  $i += 1;$ 
8   |  $df(x')$ ;
9   |  $dx = \lambda \frac{df(x')}{dx};$ 
10  |  $x' = x' + dx;$ 
11  |  $x' = clip(x', x_{min}, x_{max});$ 
12  |  $dx = clip(x' - x, -\epsilon, \epsilon);$ 
13  | if check-constraint-violation ( $c$ ,  $dx$ ) == True then
14    |   |  $dx = linear-regression(c, dx);$ 
15  | if is-similar ( $X$ ,  $x + dx$ ) then
16    |   | break;
17  | end
18  |  $x' = x + dx;$ 
19  | if  $f(x') > th_{conf2}$  then
20    |   | break;
21  | end
22 end
23 return  $x'$ 
```

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.

Enforcing Grammar during Feedback. 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 [64], 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 [64]). For the linear regression, the non-constant part of the constraints are weights 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 projection $dx_{proj} = \arg \min_{dx_{proj}} \|Wdx_{proj} - y\|$.

6 EXPERIMENTAL DESIGN

Environment. Our primary evaluation uses the CARLA version 0.9.9 simulator [17]. All the algorithms are built on top of pymoo [65], an open-source Python framework for single- and multi-objective algorithms.

Scenarios. We run *AutoFuzz* under five different logical scenarios inspired by the NHTSA report [10]. The details are shown in Table 2. The first three logical scenarios cover the top six pre-crash functional scenarios shown in Table 1, and the fourth also occurs frequently. The fifth is also a common logical scenario.

AV controller. We test two rule-based PID controllers, **pid-1** [6] and **pid-2** [18], one end-to-end controller [18], (lbc), and one modular controller [21], (**APOLLO6.0**). lbc is a vision-based, end-to-end controller proposed in [18]. 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 [17] and has been used as the main system under test in existing literature [6]. pid-2 is a rule-based pid controller implemented by the authors in [18] to collect data to train lbc. APOLLO6.0 is 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

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 / decelerating	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	n/a	26	town03	straight road	147
Turning left a signalized junction	n/a	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-GRAD on the lbc controller in CARLA. (last row) average number of collision traffic violations found by GA-UN-NN-GRAD on APOLLO6.0 in SVL.

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 violations found 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. The number of simulations referred to in the following RQs excludes the number of simulations needed for the seed collection stage. 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.

TABLE 3: Proposed methods, baselines and variations

Method	Description
AutoFuzz (GA-UN-NN-GRAD) ($\epsilon=1.0$)	GA-UN-NN w/ constrained gradient guided mutation
Baselines	
NSGA2-DT [7]	NSGA2 w/ decision tree
NSGA2-SM [8]	NSGA2 w/ surrogate model
NSGA2-UN-SM-A	NSGA2-SM w/ duplicate elimination and incrementally learned surrogate model
AV-FUZZER [50]	global GA + local GA
Variants	
GA-UN-NN-GRAD * ($\epsilon=0.3$)	GA-UN-NN-GRAD w/ a smaller (0.3 rather than 1) maximum perturbation bound ϵ
RANDOM-UN-NN-GRAD	RANDOM w/ duplicate elimination, NN filtration and constrained gradient guided mutation
GA-UN-NN	GA-UN w/ NN filtration
GA-UN	GA w/ duplicate elimination
GA	genetic algorithm
RANDOM	random sampling

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

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.

Among the baseline methods, NSGA2-DT and NSGA2-SM are two multi-objective GA-based methods and AV-

FUZZER is a single-objective GA-based method, all of them are adapted from previous work [7], [8], [50]. NSGA2-DT calls NSGA2 [61] as a subroutine. After each run of NSGA2, NSGA2-DT fits a decision tree over all instances so far. It uses cases that fall into the leaves with more traffic violations than 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 [7]. 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-SM trains regression NNs for every search objective and ranks candidate test cases and traffic violations found 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-SM with additional duplication elimination and incremental learning. For both NSGA2-SM and NSGA2-UN-SM-A training 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 [50] first runs a global GA for several iterations and enters a local GA with the initial population set to the scenario vectors with the highest fitness scores. It also starts a new global GA every 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 [9], Asfault [14] or FusionFuzz [42] since they are essentially GA with 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:



Fig. 5: RQ1. Example traffic violations found 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) lbc controller makes a wide turn into the opposing lane (considered "off-road").

RQ1: Evaluating Performance. How effectively can *AutoFuzz* find unique traffic violations in comparison to baselines?

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

RQ3: Evaluating Repair Impact. Can we leverage traffic violations found 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 violations for the AV controllers under test. Note that all the traffic violations are 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-GRAD on all three controllers for 700 simulations, with the search objective to find collision traffic violations in 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 violations total across the three controllers for this logical scenario. In particular, it found 575 unique traffic violations for the **lbc** 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. Figure 5 shows snapshots of example traffic violations found 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.

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

GA-UN-NN-GRAD consistently finds 10%-39% more than the baseline methods. In particular, GA-UN-NN-GRAD finds 41, 51, 135 and 111 more unique traffic violations over the second-best method in the four logical scenarios.

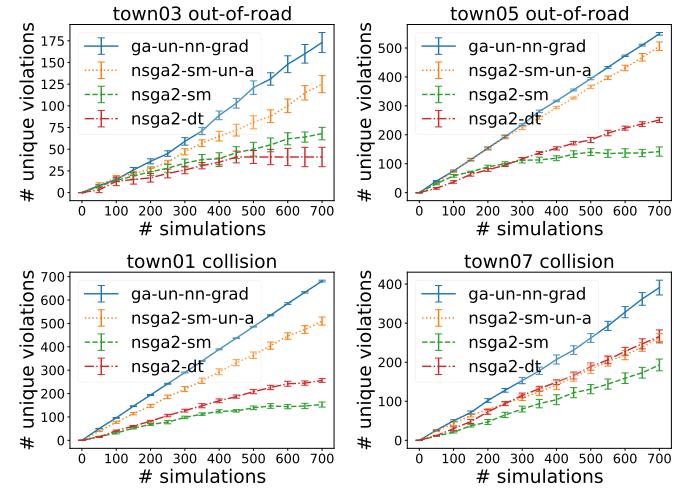


Fig. 6: RQ1. average # unique off-road or collision violations.

We further conduct Wilcoxon rank-sum test [66] and Vargha-Delaney effect size test [67], [68]. For all the settings, the 90% confidence interval of the effect size between GA-UN-NN-GRAD and 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-GRAD and NSGA2-UN-SM-A win by a large margin. For example, for the turning left non-signalized junction logical scenario, GA-UN-NN-GRAD and NSGA2-UN-SM-A have 100% unique violations while the other two methods have only 42% and 22%. This is expected since they both have a duplicate elimination component inherent to the search strategy. The results show that the baselines NSGA2-SM and NSGA2-DT waste many resources by running similar violation-producing specific scenarios.

After introducing duplicate elimination (UN) and incremental learning (A), NSGA2-UN-SM-A finds more violations than NSGA2-SM. But GA-UN-NN-GRAD still has advantages: 1. Our goal is to maximize the number of unique traffic violations than finding traffic violations with the best Pareto front [7], [8], so a binary classification NN gives a better guide than multiple regression NNs. 2. The constrained gradient-guided permutation gives a further boost. The second point is also shown in the ablation study in RQ 2.

Next, we study if GA-UN-NN-GRAD can effectively find more unique traffic violations over baselines under different initial seeds. We compare the number of unique traffic violations found by GA-UN-NN-GRAD with NSGA2-UN-SM-A and NSGA2-DT for 700 simulations, assuming 500 initial seeds collected by RANDOM, and 100 and 1000 initial seeds collected by GA-UN, resp. As shown in Figure 7, GA-UN-NN-GRAD finds 99, 139, and 121 more unique traffic violations than the baselines.

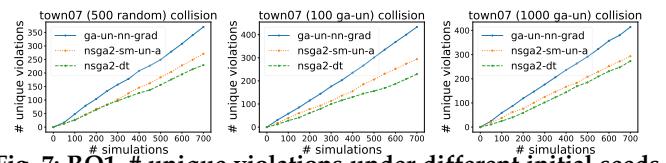


Fig. 7: RQ1. # unique violations under different initial seeds.

Result 1: AutoFuzz finds hundreds of unique traffic violations across 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 with the town07 logical scenario, with finding collisions as the search objective. However, the observations also hold in general for other logical scenarios and objectives.

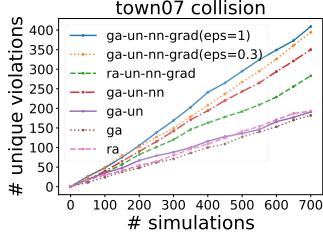


Fig. 8: #unique traffic violations found by AutoFuzz’s variants.

We conduct an ablation study on the impact of each GA-UN-NN-GRAD component, comparing the number of unique traffic violations found by GA-UN-NN-GRAD with the six variations shown in Table 3. Figure 8 presents the results.

- GA-UN-NN-GRAD ($\epsilon = 1$ vs. 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 vs. RANDOM-UN-NN-GRAD. GA-UN-NN-GRAD finds more violations indicating the importance of the base sampling strategy.
- GA-UN-NN-GRAD vs. GA-UN-NN vs. GA-UN. GA-UN-NN-GRAD finds more unique violations than GA-UN-NN and GA-UN-NN beats GA-UN. These show the necessity of the gradient-guided mutation component (GRAD) and seed selection component (NN). Furthermore, GA-UN finds slightly more unique traffic violations than GA.

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-GRAD finds at least 10-30% more unique traffic violations than 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 violations compared 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 violations found to improve a controller to reduce future traffic violations. We focus on the collisions found

TABLE 4: Number of unique traffic violations found by each search method under different definitions of unique traffic violations.

(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

for four logical scenarios. For each one, we randomly select 200 detected traffic violations by GA-UN-NN-GRAD for **Ibc**, 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 **Ibc** 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 driving through for over 75% of the originally failing specific scenarios.

TABLE 5: Number of traffic 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-signaled	47	76 / 100
crossing non-signaled	91	100 / 100
changing lane	64	75 / 100

Result 3: In our preliminary study, retraining with traffic violations found by AutoFuzz improved the **Ibc** controller’s performance on failure cases by 75% to 100%.

RQ4. Evaluating Generalizability.

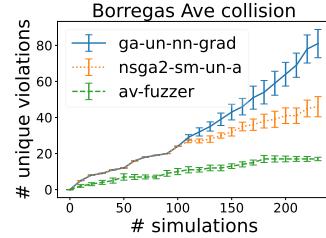


Fig. 9: RQ4. average #unique collision traffic violations.

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 AV controller [21], using a different simulator, SVL (version 2021.3) [19], [20]. We analyze the SVL API similarly to CARLA (Section 3) 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 SVL do not influence LiDAR which APOLLO6.0 mostly 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-A for 14 generations totaling 140 simulations (excluding an initial 100 warm-up simulations) and run AV-FUZZER for 240 simulations (since it does not have warm-up simulations). We then compare them over the entire 240 simulations. As shown in Figure 9, on average of six repetitions, GA-UN-NN-GRAD finds 76 unique traffic violations—which is 49% and 375% more, respectively, than the two baseline methods NSGA2-UN-SM-A and 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-GRAD and 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.

We have observed that AV-FUZZER finds much fewer traffic violations because it has very limited diversity exploration. In particular, its default mutation rate is relatively small and its local GA starts with the mutated duplicates of the global best scenario vector so far, both of which limits diversity. If the global best scenario vector does not change after several generations, all the local GA will start with its duplicates and thus further limit diversity. Moreover, its resampling process picks the scenarios farthest scenario vectors from 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. Figure 10 shows two exemplary Apollo traffic violations found by *AutoFuzz*: the ego car turning left collides with a pedestrian crossing the street and an incoming truck, respectively.



Fig. 10: Two traffic violations found for APOLLO6.0 in 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.

Result 4: *AutoFuzz* can generalize beyond CARLA. In particular, it can find more unique traffic violations than 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



Fig. 11: An example of traffic violation in a high-dimensional scenario: the AV (controlled by lbc) collides with a child crossing street.

test [69], [70]. Fuzzing tends to work well with relatively simple input formats such as image, audio or video [71], [72], [73], [74]. For more complex input formats such as cloud service APIs [75], XML parsers [76] or language compilers [77], researchers often use grammar-based fuzzing [78], [79], [80], [81], [82] to obey domain-specific constraints and narrow down the search space for producing effective and valid inputs.

Language Specification and Testing. OpenScenario [83] is an open file format for describing the dynamic contents of driving simulations at a logical level [84], but it is at an early stage. GeoScenario [85] provides a language describing a specific scenario to be simulated; [86] 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 of parameters, the constraints between them, and their distributions for automatically finding traffic violations.

9 DISCUSSION & THREATS TO VALIDITY

Our evaluation results are limited by the simulator implementations. Some reported traffic violations might 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 (CARLA and SVL) and four different controllers (lbc, pid1, pid2, APOLLO6.0). Further, to make the simulated crashes close to the real world, we construct logical scenarios based on the most frequent pre-crash functional scenarios from an NHTSA report. The example shown in Figure 11 is a complex high-dimensional (328d) scenario with many agents.

The uniqueness of traffic violations is hard to define precisely. We mitigate this threat to construct validity by extending the definition used in [7] with additional configurable parameters th_1 and th_2 , enabling users to control uniqueness stringency. A more desirable definition might be 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.

10 CONCLUSION

We present *AutoFuzz*, a grammar-based fuzzing technique for finding traffic violations in AV controllers during simulation-based testing. A traffic violation indicates 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 three AV controllers shows that *AutoFuzz* successfully finds hundreds of realistic unique traffic violations resembling complex real-world crashes and other driving offenses, outperforming the baseline methods. Furthermore, we capitalize on traffic violations found by *AutoFuzz* to improve a learning-based controller's behavior on similar cases. Finally, we apply *AutoFuzz* on APOLLO6.0 running in SVL to show its generalizability.

REFERENCES

- [1] D. o. M. V. State of California, "Autonomous Vehicle Testing Permit Holders," <https://www.dmv.ca.gov/portal/vehicle-industry-services/autonomous-vehicles/autonomous-vehicle-testing-permit-holders/>, 2020.
- [2] State of California, Department of Motor Vehicles, <https://www.dmv.ca.gov/portal/vehicle-industry-services/autonomous-vehicles/autonomous-vehicle-collision-reports/>, 2020.
- [3] J. Norden, M. O'Kelly, and A. Sinha, "Efficient black-box assessment of autonomous vehicle safety," in *Machine Learning for Autonomous Driving Workshop at the 33rd Conference on Neural Information Processing Systems (NeurIPS 2019)*, 12 2019.
- [4] A. Gambi, T. Huynh, and G. Fraser, "Generating effective test cases for self-driving cars from police reports," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019, 2019, p. 257–267. [Online]. Available: <https://doi.org/10.1145/3338906.3338942>
- [5] Y. Abeyasingheawardena, F. Shkurti, and G. Dudek, "Generating adversarial driving scenarios in high-fidelity simulators," in *2019 International Conference on Robotics and Automation (ICRA)*, 2019, pp. 8271–8277.
- [6] W. Ding, B. Chen, M. Xu, and D. Zhao, "Learning to Collide: An Adaptive Safety-Critical Scenarios Generating Method," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, October 2020, pp. 2243–2250. [Online]. Available: <https://doi.org/10.1109/IROS45743.2020.9340696>
- [7] R. B. Abdessalem, S. Nejati, L. C. Briand, and T. Stifter, "Testing Vision-Based Control Systems Using Learnable Evolutionary Algorithms," in *40th International Conference on Software Engineering*, ser. ICSE '18, May 2018, p. 1016–1026. [Online]. Available: <https://doi.org/10.1145/3180155.3180160>
- [8] R. Ben Abdessalem, S. Nejati, L. C. Briand, and T. Stifter, "Testing advanced driver assistance systems using multi-objective search and neural networks," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 63–74.
- [9] 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 *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018, 2018, p. 143–154. [Online]. Available: <https://doi.org/10.1145/3238147.3238192>
- [10] Wassim G. Najm, John D. Smith, and Mikio Yanagisawa, "Pre-Crash Scenario Typology for Crash Avoidance Research," *National Highway Transportation Safety Administration, Washington, DC, USA, Tech. Rep. DOT-HS-810 767*, April 2007.
- [11] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley, 2007.
- [12] B. P. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities," *Communications of the ACM (CACM)*, vol. 33, no. 12, pp. 32–44, 1990.
- [13] B. Miller, M. Zhang, and E. Heymann, "The Relevance of Classic Fuzz Testing: Have We Solved This One?" *IEEE Transactions on Software Engineering (TSE)*, December 2020. [Online]. Available: <https://doi.org/10.1109/TSE.2020.3047766>
- [14] S. Kuutti, S. Fallah, and R. Bowden, "Training adversarial agents to exploit weaknesses in deep control policies," 02 2020.
- [15] D. She, R. Krishna, L. Yan, S. Jana, and B. Ray, "MtFuzz: fuzzing with a multi-task neural network," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 737–749.
- [16] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. [Online]. Available: <https://openreview.net/forum?id=rJzIBfZAb>
- [17] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," ser. Proceedings of Machine Learning Research, S. Levine, V. Vanhoucke, and K. Goldberg, Eds., vol. 78, 13–15 Nov 2017, pp. 1–16. [Online]. Available: <http://proceedings.mlr.press/v78/dosovitskiy17a.html>
- [18] D. Chen, B. Zhou, V. Koltun, and P. Krähenbühl, "Learning by Cheating," in *3rd Conference on Robot Learning (CoRL)*, October 2019. [Online]. Available: <http://proceedings.mlr.press/v100/chen20a.html>
- [19] LG Electronics, "SVL Simulator: An Autonomous Vehicle Simulator, A ROS/ROS2 Multi-robot Simulator for Autonomous Vehicles," <https://github.com/lgsvl/simulator>, 2021.
- [20] G. Rong, B. H. Shin, H. Tabatabaei, Q. Lu, S. Lemke, M. Možeiko, E. Boise, G. Uhm, M. Gerow, S. Mehta, E. Agafonov, T. H. Kim, E. Sterner, K. Ushiroda, M. Reyes, D. Zelenkovsky, and S. Kim, "LGSVL Simulator: A High Fidelity Simulator for Autonomous Driving," in *24th IEEE International Conference on Intelligent Transportation (ITSC)*, September 2021. [Online]. Available: <https://arxiv.org/abs/2005.03778>
- [21] Baidu, "Apollo: An open autonomous driving platform," <https://github.com/ApolloAuto/apollo>, 2021.
- [22] S. Ulbrich, T. Menzel, A. Reschka, F. Schuldt, and M. Maurer, "Defining and substantiating the terms scene, situation, and scenario for automated driving," in *2015 IEEE 18th International Conference on Intelligent Transportation Systems*, 2015, pp. 982–988.
- [23] PEGASUS RESEARCH PROJECT, "SCENARIO DESCRIPTION," https://www.pegasusprojekt.de/files/tmpl/PDF-Symposium/04_Scenario-Description.pdf, 2019.
- [24] Waymo, "Waymo Safety Report," <https://storage.googleapis.com/waymo-uploads/files/documents/safety/2021-03-waymo-safety-report.pdf>, February 2021.
- [25] US Department of Transportation, Federal Highway Administration, "Average Annual Miles per Driver by Age Group," <https://www.fhwa.dot.gov/ohim/onh00/bar8.htm>, 2018.
- [26] U. D. of Transportation; Federal Highway Administration, "Licensed drivers, by state, 1949 - 2018," 2020. [Online]. Available: <https://www.fhwa.dot.gov/policyinformation/statistics/2018/dl201.cfm>
- [27] K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song, "Robust physical-world attacks on deep learning visual classification," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 1625–1634.
- [28] 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 *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications*

- Security*, ser. CCS '19, 2019, p. 1989–2004. [Online]. Available: <https://doi.org/10.1145/3319535.3354259>
- [29] H. Zhou, W. Li, Y. Zhu, Y. Zhang, B. Yu, L. Zhang, and C. Liu, “Deepbillboard: Systematic physical-world testing of autonomous driving systems,” *CoRR*, vol. abs/1812.10812, 2018. [Online]. Available: <http://arxiv.org/abs/1812.10812>
- [30] Y. Jia, Y. Lu, J. Shen, Q. A. Chen, H. Chen, Z. Zhong, and T. Wei, “Fooling detection alone is not enough: Adversarial attack against multiple object tracking,” in *International Conference on Learning Representations*, 2020. [Online]. Available: <https://openreview.net/forum?id=rJl31TNYPr>
- [31] Y. Cao, C. Xiao, B. Cyr, Y. Zhou, W. Park, S. Rampazzi, Q. A. Chen, K. Fu, and Z. M. Mao, “Adversarial sensor attack on lidar-based perception in autonomous driving,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19, 2019, p. 2267–2281. [Online]. Available: <https://doi.org/10.1145/3319535.3339815>
- [32] Y. Tian, K. Pei, S. Jana, and B. Ray, “Deeptest: Automated testing of deep-neural-network-driven autonomous cars,” in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18, 2018, p. 303–314. [Online]. Available: <https://doi.org/10.1145/3180155.3180220>
- [33] 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 *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018, pp. 132–142.
- [34] A. Calò, P. Arcaini, S. Ali, F. Hauer, and F. Ishikawa, “Generating avoidable collision scenarios for testing autonomous driving systems,” in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020, pp. 375–386.
- [35] P. Arcaini, X.-Y. Zhang, and F. Ishikawa, “Targeting patterns of driving characteristics in testing autonomous driving systems,” in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2021, pp. 295–305.
- [36] Y. Luo, X.-Y. Zhang, P. Arcaini, Z. Jin, H. Zhao, F. Ishikawa, R. Wu, and T. Xie, “Targeting requirements violations of autonomous driving systems by dynamic evolutionary search,” in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021.
- [37] J. Wang, A. Pun, J. Tu, S. Manivasagam, A. Sadat, S. Casas, M. Ren, and R. Urtasun, “Advsim: Generating safety-critical scenarios for self-driving vehicles,” *ArXiv*, vol. abs/2101.06549, 2021.
- [38] J. Tu, H. Li, X. Yan, M. Ren, Y. Chen, M. Liang, E. Bitar, E. Yumer, and R. Urtasun, “Exploring adversarial robustness of multi-sensor perception systems in self driving,” 01 2021.
- [39] K. Wong, Q. Zhang, M. Liang, B. Yang, R. Liao, A. Sadat, and R. Urtasun, “Testing the safety of self-driving vehicles by simulating perception and prediction,” in *ECCV*, 2020.
- [40] 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 *29th USENIX Security Symposium*, August 2020. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity20/presentation/shen>
- [41] J. Han and Z. Q. Zhou, “Metamorphic fuzz testing of autonomous vehicles,” in *5th International Workshop on Metamorphic Testing*, 05 2020.
- [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.
- [43] A. Gambi, M. Mueller, and G. Fraser, “Automatically testing self-driving cars with search-based procedural content generation,” in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019, 2019, p. 318–328. [Online]. Available: <https://doi.org/10.1145/3293882.3330566>
- [44] M. O’ Kelly, A. Sinha, H. Namkoong, R. Tedrake, and J. C. Duchi, “Scalable end-to-end autonomous vehicle testing via rare-event simulation,” in *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31. Curran Associates, Inc., 2018, pp. 9827–9838. [Online]. Available: <https://proceedings.neurips.cc/paper/2018/file/653c579e3f9ba5c03f2f2fc4512b39-Paper.pdf>
- [45] T. A. Wheeler and M. J. Kochenderfer, “Critical factor graph situation clusters for accelerated automotive safety validation,” in *2019 IEEE Intelligent Vehicles Symposium (IV)*, 2019, pp. 2133–2139.
- [46] B. Chen and L. Li, “Adversarial evaluation of autonomous vehicles in lane-change scenarios,” 04 2020.
- [47] M. Koren, S. Alsaif, R. Lee, and M. J. Kochenderfer, “Adaptive stress testing for autonomous vehicles,” in *2018 IEEE Intelligent Vehicles Symposium (IV)*, 2018, pp. 1–7.
- [48] Wenhao Ding, Baimimng Chen, Bo Li, Kim Ji Eun, Ding Zhao, “Multimodal safety-critical scenarios generation for decision-making algorithms evaluation,” in *arxiv*, 2020.
- [49] M. R, M. A, P. M, S. L, and Z. D., “Paracosm: A test framework for autonomous driving simulations,” *Fundamental Approaches to Software Engineering*, 2021.
- [50] G. Li, Y. Li, S. Jha, T. Tsai, M. Sullivan, S. K. S. Hari, Z. Kalbarczyk, and R. Iyer, “AV-FUZZER: Finding Safety Violations in Autonomous Driving Systems,” in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, October 2020, pp. 25–36. [Online]. Available: <https://doi.org/10.1109/ISSRE5003.2020.00012>
- [51] Z. Zhong, Y. Tang, Y. Zhou, V. de Oliveira Neves, Y. Liu, and B. Ray, “A survey on scenario-based testing for automated driving systems in high-fidelity simulation,” 2021.
- [52] C. team, “Scenariorunner for carla,” https://github.com/carla-simulator/scenario_runner, 2020.
- [53] M. Böhme, V.-T. Pham, and A. Roychoudhury, “Coverage-based greybox fuzzing as markov chain,” *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2017.
- [54] C. Lemieux and K. Sen, “Fairfuzz: Targeting rare branches to rapidly increase greybox fuzz testing coverage,” in *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2018.
- [55] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, “VUzzer: Application-aware evolutionary fuzzing,” in *Proceedings of the Network and Distributed System Security Symposium*, 2017.
- [56] P. Chen and H. Chen, “Angora: Efficient fuzzing by principled search,” in *Proceedings of the IEEE Symposium on Security & Privacy*, pp. 711–725, 2018.
- [57] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, “Neuzz: Efficient fuzzing with neural program learning,” in *Proceedings of the IEEE Symposium on Security & Privacy*, 2019.
- [58] M. Zalewski, “American fuzzy lop,” URL: <http://lcamtuf.coredump.cx/afl>, 2017.
- [59] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu, and X. Zhou, “Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit,” in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 2307–2324.
- [60] M. Böhme, V. J. Manès, and S. K. Cha, “Boosting fuzzer efficiency: An information theoretic perspective,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 678–689.
- [61] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, “A fast and elitist multiobjective genetic algorithm: Nsga-ii,” *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.
- [62] R. Agrawal, K. Deb, and R. Agrawal, “Simulated binary crossover for continuous search space,” *Complex Systems*, vol. 9, 06 2000.
- [63] K. Deb and S. Agrawal, ““a niched-penalty approach for constraint handling in genetic algorithms”,” in *Artificial Neural Nets and Genetic Algorithms*. Springer, 1999, pp. 235–243. [Online]. Available: https://doi.org/10.1007/978-3-7091-6384-9_40
- [64] J. Li, J. Lee, Y. Yang, J. Sun, and K. Tomovic, “Conaml: Constrained adversarial machine learning for cyber-physical systems,” 03 2020.
- [65] J. Blank and K. Deb, “Pymoo: Multi-objective optimization in python,” *IEEE Access*, vol. 8, pp. 89 497–89 509, 2020.
- [66] J. A. Capon, “Elementary statistics for the social sciences: Study guide,” 1991.
- [67] A. Vargha and H. D. Delaney, “A critique and improvement of the cl common language effect size statistics of mcgraw and wong,” *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000. [Online]. Available: <https://doi.org/10.3102/10769986025002101>
- [68] A. Arcuri and L. Briand, “A hitchhiker’s guide to statistical tests for assessing randomized algorithms in software engineering,” *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.
- [69] C. Hutchison, M. Zizyte, P. E. Lanigan, D. Guttendorf, M. Wagner, C. Le Goues, and P. Koopman, “Robustness Testing of Autonomy Software,” in *IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, May 2018, pp. 276–285.
- [70] T. Dreossi, D. J. Fremont, S. Ghosh, E. Kim, H. Ravanbakhsh, M. Vazquez-Chanlatte, and S. A. Seshia, “VERIFAI: A Toolkit

- for the Design and Analysis of Artificial Intelligence-Based Systems," *Computer Aided Verification*, 2019. [Online]. Available: https://doi.org/10.1007/978-3-030-25540-4_25
- [71] Joint Photographic Experts Group, "Overview of JPEG 1," <https://jpeg.org/jpeg/>.
- [72] Sustainability of Digital Formats, "Planning for Library of Congress Collections. MP3 (MPEG Layer III Audio Encoding)," <https://www.loc.gov/preservation/digital/formats/fdd/fdd000012.shtml>.
- [73] The Moving Picture Experts Group, "MPEG-4," <https://mpeg.chiariglione.org/standards/mpeg-4>.
- [74] Linux manual page, "elf - format of Executable and Linking Format (ELF) files," <http://man7.org/linux/man-pages/man5/elf.5.html>.
- [75] 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," *arxiv preprint 2005.11498*, May 2020. [Online]. Available: <https://arxiv.org/abs/2005.11498>
- [76] W3C, "Extensible Markup Language (XML) 1.0 (Fifth Edition)," <https://www.w3.org/TR/xml/>.
- [77] Free Software Foundation, "GCC, the GNU Compiler Collection," <https://gcc.gnu.org>.
- [78] M. Eberlein, Y. Noller, T. Vogel, and L. Grunske, "Evolutionary Grammar-Based Fuzzing," in *Search-Based Software Engineering (SSBSE)*, ser. Lecture Notes in Computer Science, A. Aleti and A. Panichella, Eds., vol. 12420. Springer, 2020. [Online]. Available: https://doi.org/10.1007/978-3-030-59762-7_8
- [79] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-Based Whitebox Fuzzing," in *29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08, June 2008, p. 206–215. [Online]. Available: <https://doi.org/10.1145/1375581.1375607>
- [80] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with Code Fragments," in *21st USENIX Conference on Security Symposium*, ser. Security'12, August 2012. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler>
- [81] E. Soremekun, E. Pavese, N. Havrikov, L. Grunske, and A. Zeller, "Inputs from Hell Learning Input Distributions for Grammar-Based Test Generation," *IEEE Transactions on Software Engineering (TSE)*, 2020. [Online]. Available: <https://doi.org/10.1109/TSE.2020.3013716>
- [82] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and Understanding Bugs in C Compilers," in *32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11, vol. 46, no. 6, June 2011, p. 283–294. [Online]. Available: <https://doi.org/10.1145/1993316.1993532>
- [83] ASAM, "OpenScenario," <https://www.asam.net/standards/detail/openscenario>, 2021.
- [84] T. Menzel, G. Bagschik, and M. Maurer, "Scenarios for Development, Test and Validation of Automated Vehicles," in *IEEE Intelligent Vehicles Symposium (IV)*, 2018, pp. 1821–1827. [Online]. Available: <https://doi.org/10.1109/IVS.2018.8500406>
- [85] R. Queiroz, T. Berger, and K. Czarnecki, "GeoScenario: An Open DSL for Autonomous Driving Scenario Representation," in *IEEE Intelligent Vehicles Symposium (IV)*, June 2019, pp. 287–294. [Online]. Available: <https://doi.org/10.1109/IVS.2019.8814107>
- [86] T. Duy Son, A. Bhave, and H. Van der Auweraer, "Simulation-Based Testing Framework for Autonomous Driving Development," in *IEEE International Conference on Mechatronics (ICM)*, vol. 1, March 2019, pp. 576–583. [Online]. Available: <https://doi.org/10.1109/ICMECH.2019.8722847>

APPENDIX A

DETAILS OF *AutoFuzz* GA-BASED FUZZING

Algorithm 2 shows the detailed implementation of our proposed algorithm GA-UN-NN-GRAD. The Neural Network part is highlighted in blue and the gradient-based mutation is highlighted in red.

Algorithm 2: *AutoFuzz* GA-based fuzzing (GA-UN-NN-GRAD)

```

Input : sampling(), evaluate(), max-gen, pop-size,
         generation-to-use-NN, candidate-multiplier,
         max-mating-iter, gradient-mutation-parameters
Output:unique-bugs
1 initial-population = sampling(pop-size);
2 initial-unique-bugs, initial-objectives =
   evaluate(initial-population, []);
3 generations = 1;
4 all-population = initial-population;
5 all-objectives = initial-objectives;
6 unique-bugs = initial-unique-bugs;
7 current-population, current-objectives = initial-population,
   initial-objectives;
8 while generations < max-gen do
9   mating-iter = 0; candidate-offspring = [];
10  remaining-size = pop-size × candidate-multiplier;
11  while mating-iter < max-mating-iter and remaining-num > 0
12    do
13      mating-iter += 1;
14      parents = selection(current-population,
15        remaining-num);
16      new-candidate-offspring = crossover(parents);
17      new-candidate-offspring =
18        mutation(new-candidate-offspring);
19      new-candidate-offspring =
20        filtering(new-candidate-offspring,
21          candidate-offspring, unique-bugs);
22      candidate-offspring = merge(candidate-offspring,
23        new-candidate-offspring);
24      remaining-num = pop-size - len(candidate-offspring);
25    end
26    if remaining-num > 0 then
27      remaining-candidate-offspring =
28        sampling(remaining-num);
29      candidate-offspring = merge(candidate-offspring,
30        remaining-candidate-offspring);
31    end
32    if generations > generation-to-use-NN then
33      f = train-NN(all-population, all-objectives);
34      sorted-candidate-offspring =
35        rank-by-confidence(candidate-offspring, f);
36      offspring = select(sorted-candidate-offspring);
37      offspring =
38        constrained-gradient-guided-mutation(offspring, f,
39          gradient-mutation-parameters);
40    end
41    new-unique-bugs, new-objectives = evaluate(offspring,
42      unique-bugs);
43    all-population = merge(all-population, offspring);
44    all-objectives = merge(all-objectives, new-objectives);
45    unique-bugs = merge(unique-bugs, new-unique-bugs);
46    combined-population = merge(current-population,
47      offspring);
48    combined-objectives = merge(current-objectives,
49      new-objectives);
50    current-population, current-objectives =
51      survival(combined-population, combined-objectives,
52      pop-size);
53    generations += 1;
54  end
55 return unique-bugs

```

APPENDIX B

ADDITIONAL TRAFFIC VIOLATIONS EXAMPLES

Figure 12 shows more examples of found traffic violations in other logical scenarios.

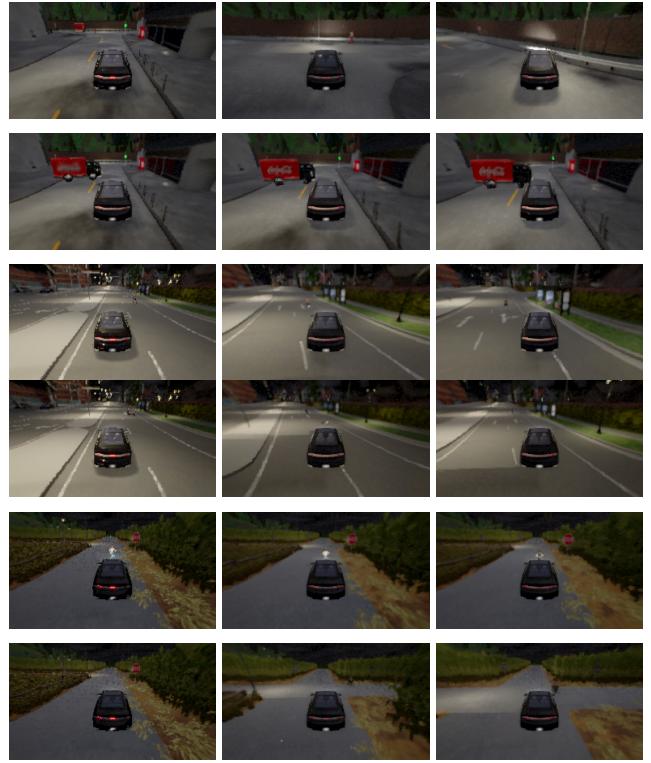


Fig. 12: Additional traffic violations found in other scenarios: (row1) lbc controller hits fencing in town01. (row2) lbc controller hits a truck in town01. (row3) lbc controller goes to the wrong lane in town03. (row4) lbc controller hits a pedestrian lying on the road in town03. (row5) lbc controller hits a pedestrian in front of it in town07. (row5) lbc controller hits a cyclist crossing the street in town07.

APPENDIX C

ADDITIONAL FIGURES

Figure 13 shows the number of all unique traffic violations (including collision traffic violations and out-of-road traffic violations) found by *AutoFuzz* with GA-UN-NN-GRAD in town05 scenario over 700 simulations for each controller as described in RQ1.

APPENDIX D

DETAILS ON PARAMETERIZING THE FUZZABLE FIELDS

Constraining the API Grammar. While fuzzing, each field can explore a large number of possible values, which significantly increases the search space and potentially leads to unrealistic scenarios. To generate more realistic scenarios we enforce following two constraints on the API grammars.

(i) *Search Range*. While fuzzing, each field can explore a large number of possible values, especially for continuous fields, which significantly increases the search space and potentially leads to unrealistic scenarios. We restrict the search space by specifying a minimum and a maximum value of each fuzzable field (represented as $[min, max]$) in

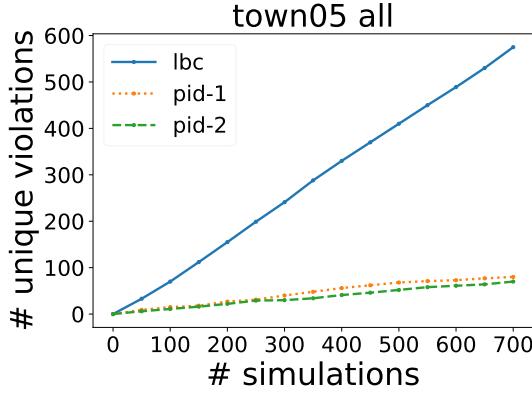


Fig. 13: RQ1. Unique traffic violations (collision and out-of-road) found with three different controllers in a seed scenario where the ego car is approaching a junction while leading car slows down / stops.

Listing 1). A user can also optionally specify a clipped distribution for the field by appending the distribution’s information: $[min, max, (distribution, mean, variance)]$, where the distribution is bounded by $[min, max]$. For example, in Listing 1, a user mutates the pedestrian’s location by sampling from a normal distribution with mean -103 and variance 10, and bounded by -123 to -83 range. With such parameterized field values, AutoFuzz uses CARLA’s built-in search function to look for defined nearby waypoints in the map and update the field values accordingly.

(ii) *Constraints.* AutoFuzz’s input space can be further constrained by providing additional conditions. For example, the distance between the ego-car and a car in front will be larger than a certain distance (see Listing 1). A user can provide a list of constraints. For each constraint, three fields need to be specified. The first field ‘coefficients’ is a list of coefficients to be multiplied before each of the selected field. The field ‘labels’ is a list of corresponding fields where constraints are applied. Finally, a ‘value’ specifies the right-hand side of the inequality. The constraint is always encoded as \leq . Thus, $coefficient[1] \times label[1] - coefficient[2] \times label[2] \leq value$.

APPENDIX E DETAILS AND ILLUSTRATION OF OBJECTIVES FOR OUT-OF-ROAD VIOLATIONS

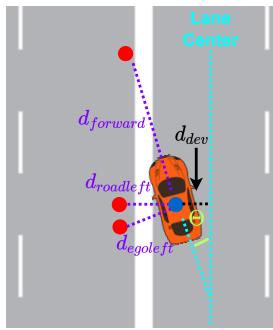


Fig. 14: Parameters computing the wrong lane objective function

Figure 14 shows a car drives on a road. The angle between the ego-car’s forward direction and the lane center is denoted Θ and the ego-car’s center’s distance from the lane center is denoted d_{dev} . $F_{deviation}$ is defined to be the product of the

two i.e. $F_{deviation} = \Theta \times d_{dev}$. $F_{wronglane}$ and $F_{offroad}$ measure the ego-car’s distance from the closest lane of opposite direction or non-drivable region. In particular, $F_{wronglane}$ is the minimum of $d_{forward}$, $d_{roadleft}$, d_{goleft} , $d_{roadright}$ and $d_{goright}$ (the last two are not shown in Figure 14), which denote the ego-car’s distance along different directions w.r.t. the closest point that is on a road with different direction. $F_{offroad}$ is defined similarly except the closest point will be off road.

APPENDIX F

DETAILED RESULTS ON PERCENTAGE OF TRAFFIC VIOLATIONS BEING UNIQUE

TABLE 6: Percentage of found traffic violations that are unique

Map ID	GA-UN-NN-GRAD	NSGA2-DT	NSGA2-SM	NSGA2-UN-SM-A
town05	$97.4 \pm 0.8\%$	$56.8 \pm 1.9\%$	$21.3 \pm 2.8\%$	$96.4 \pm 0.8\%$
town01	100%	$41.2 \pm 1.2\%$	$23.1 \pm 1.5\%$	100%
town07	100%	$67.5 \pm 2.1\%$	$66.0 \pm 4.1\%$	100%
town03	$99.9 \pm 0.3\%$	$79.4 \pm 7.6\%$	$54.0 \pm 7.2\%$	100%

Table 6 shows the detailed results of the percentage of found traffic violations by each method that are unique. Without surprise, GA-UN-NN-GRAD and NSGA2-UN-SM-A have much higher percentages than the other two.

APPENDIX G

MORE DETAILS ON APOLLO6.0 IN SVL

SVL officially only provides APOLLO6.0 (modular testing) in which the perception, camera and traffic light modules of APOLLO6.0 are not activated. Instead, the ground-truth information of other objects as well as traffic light are provided. To test the perception and camera modules as well as other modules, we created a version which has these modules activated. However, the traffic light modules cannot be successfully activated and it is a known issue on SVL github repo. As a workaround, the ground-truth traffic light information are fed into APOLLO6.0. In this way, we can test all modules of APOLLO6.0 except the traffic light module.