# Doppelgänger Test Generation for Revealing Bugs in Autonomous Driving Software

Yuqi Huai [†], Yuntianyi Chen [†], Sumaya Almanee [†], Tuan Ngo [*], Xiang Liao [†], Ziwen Wan [†], Qi Alfred Chen [†], and Joshua Garcia [†]

[†]University of California, Irvine, { yhuai, yuntianc, salmanee, liaox11, ziwenw8, alfchen, joshug4 }@uci.edu
[*]VNU University of Engineering and Technology, tuanngokien@vnu.edu.vn

***Abstract*—Vehicles controlled by autonomous driving software (ADS) are expected to bring many social and economic benefits, but at the current stage not being broadly used due to concerns with regard to their safety. Virtual tests, where autonomous vehicles are tested in software simulation, are common practices because they are more efficient and safer compared to field operational tests. Specifically, search-based approaches are used to find particularly critical situations. These approaches provide an opportunity to automatically generate tests; however, systematically producing *bug-revealing* tests for ADS remains a major challenge. To address this challenge, we introduce DoppelTest, a test generation approach for ADSes that utilizes a genetic algorithm to discover bug-revealing violations by generating scenarios with multiple autonomous vehicles that account for traffic control (e.g., traffic signals and stop signs). Our extensive evaluation shows that DoppelTest can efficiently discover 123 bug-revealing violations for a production-grade ADS (Baidu Apollo) which we then classify into 8 unique bug categories.**

***Index Terms*—cyber-physical systems, autonomous driving systems, search-based software testing**

## I. Introduction

Autonomous vehicles (AVs), a.k.a. self-driving cars, are becoming a pervasive and ubiquitous part of our daily life. More than 50 corporations are actively working on AVs, including large companies such as Google's parent company Alphabet, Ford, and Intel [1]–[3]. Some of these companies (e.g., Alphabet's Waymo, Lyft, ArgoAI, and Baidu) are already serving customers on public roads [4]–[7]. Experts forecast that AVs will drastically impact society, particularly by reducing accidents [8]. However, crashes caused by AVs indicate that achieving this lofty goal remains an open challenge. Despite the fact that companies such as Tesla [9], Waymo [1], or Uber [10] have released prototypes of AVs with a high level of autonomy, they have caused injuries or even fatal accidents to pedestrians. For instance, an AV of Uber killed a pedestrian in Arizona back in 2018 [11]. AVs with lower levels of autonomy have resulted in more fatalities in recent years [11]–[18].

To ensure the safety and quality of an AV, a common practice for testing the autonomous driving system (ADS) that operates it lies in field operational tests, in which AVs are left to drive freely in the physical world. This approach is not only expensive and dangerous but also insufficient since it misses critical testing scenarios [19]. Virtual tests, where AVs are tested in software simulations, offer a far more efficient and safer alternative, precluding the need to obtain necessary permits or licenses from government agencies or the risks associated with potentially crashing the vehicle in the physical world.

While these virtual tests can be automatically generated, it is difficult to systematically generate scenarios with violations that actually reveal ADS software bugs, which we refer to as *bug-revealing violations*. There have been a number of techniques [20]–[22] that focus on generating scenarios for simulation-based testing. Although such testing is capable of finding violations in simulated AV scenarios, these violations do not necessarily reveal a bug in the ADS. Although violations are important and should be minimized, violations that do not correspond to bugs in an ADS are not actually phenomena that software developers can do much to control (e.g., they have little control of obstacles that violate traffic rules or even intentionally try to crash into the AV). The key determining factor as to whether a bug-revealing violation occurs in an AV scenario is if the violation is the responsibility of the AV because responsibility is generally determined based on if the driver had any misbehavior [23]–[28].

Unfortunately, obstacles generated by prior work [20]–[22] lack the intelligence necessary to follow traffic rules that maximize the responsibility of an AV for violations that occur in simulation. For example, vehicles other than the AV in simulation may not follow traffic rules or drive defensively, making violations much more likely to be the responsibility of these non-AV vehicles. As another example, non-AV vehicles may be driving at a constant speed, preventing them from being able to speed up or slow down to prevent an accident (e.g., brake for another obstacle or turn quickly to avoid being hit by another obstacle). Previous AV test generation techniques may produce obstacles that are not aware of other obstacles in the scenario, which makes these obstacles more likely to be responsible for a violation (e.g., a collision). As an example, if an obstacle is traveling (1) fast enough behind an AV to cause a rear-end collision and (2) too fast for the AV to move out of the way, a resulting rear-end collision here is the responsibility of the rear obstacle, not the AV.

To generate bug-revealing violations and address the limitation of traffic rule-ignoring obstacles, we introduce DoppelTest, a novel test-generation approach for AV software testing that (1) utilizes smart obstacles and (2) ensures the ADS is responsible when violations occur by running multiple ADSes in the same scenario, where each ADS controls a different AV. Unlike

1558-1225/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE48619.2023.00216

prior work, which relies on unintelligent obstacles, we use multiple instances of the same ADS to control every vehicle in a scenario, which we refer to as *doppelgänger instances*. As a result, every vehicle in the scenario is an AV. These doppelgänger instances guarantee every vehicle has complex enough logic to consider traffic rules and the presence of other participants. Additionally, when violations happen, at least one of the AVs must have failed to follow traffic rules or made a sub-optimal decision, this ensures that there is a bug in the ADS under test.

The main contributions of this work are as follows:

- We propose Doppelgänger Testing, a novel methodology of testing ADSes. We also design and implement DoppelTest around this novel methodology which automatically discovers bug-revealing violations.
- We evaluate DoppelTest on an industry-grade ADS and discovered 123 bug-revealing violations that reveal 8 unique bug types. We analyzed and submitted a fix addressing 1 bug type, which has been accepted by the ADS developers.

## II. Background

### A. Autonomous Driving Systems

An ADS aims to achieve high automation levels for vehicles to automatically run on the roads. The autonomy levels for self-driving cars depend on various features including adaptive navigation control, environmental detection, and other driver assistance systems. The Society of Automotive Engineers (SAE) defines the levels of autonomous driving from 0 with no assistance systems to Level 5, which represents fully autonomous driving [29]. There are six autonomy levels for autonomous vehicles [30]. The ADS is used to achieve high automation (Level 4) or full automation (Level 5).

Baidu Apollo [31] achieves high automation (Level 4) [32], which means the ADS controls the vehicle for all potential circumstances automatically. The vehicle is able to perform all types of driving tasks in different traffic scenarios and is capable of handling the majority of driving situations without any input from a human driver, leaving a limited number of cases where a human driver may need to intervene.

The ADS is a large software system that consists of different modules with varying functionalities. Our process of testing autonomous vehicles with Apollo in the simulation environment only needs certain modules. A brief introduction of those modules is as below [31]:

- **HD-Map** includes lane geometries and location of traffic control devices, which may be used by other modules.
- **Routing** generates high-level navigation information based on the routing requests. It tells the autonomous vehicle which paths to take to reach its destination.
- **Localization** provides location, heading, velocity, and acceleration information of the AV.
- **Perception** identifies the physical world surrounding the self-driving car by integrating multiple sensors (e.g., camera, radar, and LiDAR) to recognize obstacles.

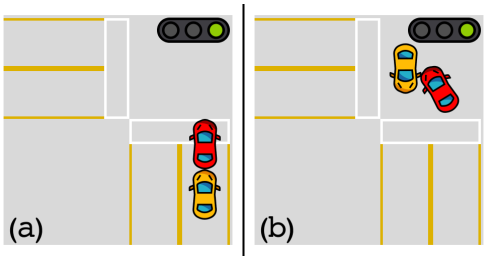


Fig. 1: 2 Illustrations of collisions that may happen, the red vehicle is the AV. (a) shows an obstacle rear-ends the AV that has the right-of-way. (b) shows the AV not yielding the right-of-way and causing a collision.

- **Prediction** receives the obstacle information including position, velocity, and acceleration detected by Perception and predicts the future motion trajectories of the obstacles.
- **Planning** makes decisions for the autonomous vehicle to execute, such as cruising or stopping.

To better understand the nature of AV bugs and how they affect the aforementioned AV modules, Garcia et al. [33] conducted an empirical study on 499 AV bugs from 16,851 commits in the Apollo and Autoware GitHub repositories, both of which are Level 4, open-source ADSes. They found that Planning had the greatest number of bugs (93 bugs in Apollo, 62.14% of total driving bugs in Apollo) and exposed many symptoms associated with driving behaviors of AVs, while previous work focused heavily on Perception and Prediction, which had significantly fewer bugs than Planning (37 bugs in total, 61.48% fewer than Planning in both ADSes studied) [33]. Based on this data, we designed our ADS testing technique to focus on testing the modules that might contain more bugs, especially Planning.

### B. Motivating Example

We introduce the following motivating example to demonstrate the importance of testing for bug-revealing violations, as opposed to just plain traffic violations: Consider a scenario with a collision violation where an AV and a non-AV vehicle collide. A non-AV vehicle is often referred to as a non-player character (NPC). In a scenario, responsibility can be attributed fully to the AV, the NPC vehicle, or even shared by both of them. Cases, where any responsibility can be attributed to the AV, are of particular interest in this paper because they are the types of violations that are bug-revealing.

Figure 1 illustrates how a collision can be bug-revealing or not depending on the situation. As shown in Figure 1(a), the AV is driving in its lane and following traffic rules; the NPC is traveling behind the AV and hits the AV from behind. In California, where an overwhelming majority of AV testing occurs at the production level, rear-end collisions are almost never the fault of the vehicle that is rear-ended [24]–[28]. In traditional scenario-based testing techniques, this type of collision is also regarded as a violation and recorded in the experimental results. As a result, this type of rear-end collision is not caused by the AV's misbehavior and thus is not actually

bug-revealing. Figure 1(b) depicts the NPC passing through the junction while the AV is turning left. From the perspective of the ADS, it should recognize that the NPC has the right-of-way and it should yield until the intersection is clear. However, in this scenario, the AV ignores the NPC and still tried to turn left. As a result, the AV's failure to yield the right-of-way caused the collision. We regard this type of scenario as bug-revealing since it is a sign of functional defects in the ADS. DoppelTest aims to generate scenarios where any violation that occurs will be the responsibility of the AV, which reveals an AV misbehavior and is thus bug-revealing.

## III. State Space Specification

To aid in the generation of effective and valid scenarios, we present a formal specification of the state space in the form of scenarios. DoppelTest uses this formal specification of the state space, along with a genetic algorithm, to generate scenarios that maximize the possibility of the ADS either violating safety or traffic rules.

**Definition 1.** A Scenario $S = \langle \mathbb{A}, \mathbb{P}, \mathbb{T}, \mathbb{V}, t^S \rangle$ is a tuple where:

- $\mathbb{A} = \{A_0, A_1, ..., A_{|\mathbb{A}|}\}$ is a finite, non-empty set of AVs.
- $\mathbb{P} = \{p_0, p_1, ..., p_{|\mathbb{P}|}\}$ is a finite set of pedestrians where $|\mathbb{P}| \geq 0$.
- $\mathbb{T}$ is a traffic signal configuration that specifies the color of each signal at any time in a simulation.
- $\mathbb{V} = \{V(A_i) : A_i \in \mathbb{A}\}$, where $V(A_i) = \langle v_0^{A_i}, v_1^{A_i}, ..., v_n^{A_i} \rangle$ is a set of violations.
- $t^S$ is the maximum allowed duration for the scenario.

**Definition 2.** An AV $A = \langle \mathbb{W}^A, t^A_{start} \rangle$ is a tuple where:

- $\mathbb{W}^A = \langle w_0^A, w_1^A, ..., w_n^A \rangle$ is a sequence of waypoints that the AV $A$ is expected to reach. $w_0^A$ is the vehicle's initial location, and $w_n^A$ is the vehicle's final destination.
- $t^A_{start}$ specifies the timestamp during a scenario at which the AV $A$ begins driving from $w_0^A$ to $w_n^A$ following waypoints specified by $\mathbb{W}^A$ and $0 \leq t^A_{start} < t^S$.

**Definition 3.** A pedestrian $P = \langle \mathbb{W}^P, t^P_{start}, s^P \rangle$ is a tuple where:

- $\mathbb{W}^P$ is a sequence of waypoints $\langle w_0^P, w_1^P, ..., w_m^P \rangle$ that the pedestrian is expected to reach. $w_0^A$ is the pedestrian's initial location, and $w_m^A$ is the pedestrian's final destination.
- $t^P_{start}$ specifies the timestamp during a scenario at which the pedestrian $P$ begins walking from $w_0^P$ to $w_m^P$ following waypoints specified by $\mathbb{W}^P$ and $0 \leq t^P_{start} < t^S$.
- $s^P$ is a finite number specifying the speed $(m/s)$ at which the pedestrian walks, and $0.6 \leq s^P \leq 1.3$, whose values are obtained from the United States Department of Transportation (USDOT) Federal Highway Administration (FHWA) [34].

**Definition 4.** Traffic Control Configuration $\mathbb{T} = \langle \mathbb{M}_{initial}, \mathbb{M}_{final}, d_{initial}, d_{transition}, d_{buffer} \rangle$ is a tuple where:

- $\mathbb{M}_{initial}$ is a mapping between every traffic signal $ts$ and its color $c \in \{\texttt{GREEN}, \texttt{RED}, \texttt{YELLOW}\}$ on a given map.
- $\mathbb{M}_{initial}$ is the starting signal color and $\mathbb{M}_{final}$ is the final signal color through one cycle of color changes.
- $d_{initial}$ is the amount of time during the scenario for which signals will display the color specified by $\mathbb{M}_{initial}$.
- $d_{transition}$ is the duration of the *yellow change interval*, which is an interval where a signal that will change from `GREEN` to `RED` should display `YELLOW` to warn vehicles about an upcoming change in right-of-way status. [34]
- $d_{buffer}$ is the duration of the *red clearance interval*, which is an interval where a signal that will change from `RED` to `GREEN` should remain `RED` to allow vehicles who have entered the intersection during the *yellow change interval* to clear the intersection. [34]

**Definition 5.** A waypoint $w$ is represented by a tuple $\langle x, y \rangle$ which indicates a specific location on the map in the coordinate system, e.g., Universal Transverse Mercator (UTM) system or World Geodetic System (WGS84) in Apollo.

**Definition 6.** A vehicle's decisions $\mathbb{D} = \langle d_0, d_1, ..., d_{|\mathbb{D}|} \rangle$ in a scenario is a finite non-empty set of unique decisions where $d_i \in \mathbb{D}$ is a single decision the vehicle has made at any time during the scenario. $d$ may take on one of the following values:

- `STOP_SS` is a decision to stop for a stop sign.
- `STOP_TS` is a decision to stop for a traffic signal.
- `STOP_OB` is a decision to stop for an obstacle.
- `YIELD_OB` is a decision to yield right-of-way to an obstacle.
- `CRUISE` is a decision to drive forward following its planning trajectory.

**Definition 7.** Road traffic participants $\mathbb{O} = \mathbb{A} \cup \mathbb{P}$ are every vehicle and pedestrian in a scenario.

**Definition 8.** A violation map $vmap : v \rightarrow vtype$ identifies the violation type $vtype = \{collision, routing, stop\_sign, traffic\_signal\}$, which corresponds to the four types of bug-revealing violations we focus on in this paper, to a violation $v \in V(A_i) \in \mathbb{V} \in S$. We elaborate on oracles that detect each of these violation types in Section IV-C.

## IV. Approach

It is often difficult to determine whether the ADS is responsible or not for a violation. To properly assign responsibility, many complex environmental factors such as who had the right-of-way, and any traffic laws one violated before the violation occurred, need to be taken into consideration. Indeed, violations that the ADS is not responsible for cannot be bug-revealing because they are not caused by any ADS misbehavior. Therefore, the key challenge for ensuring ADS responsibility is to determine *how to model obstacles so that (1) their maneuvers are realistic and (2) they will not be responsible for causing violations*. Accomplishing such modeling could be highly challenging because this model needs a highly intelligent representation of road traffic participants that follow all of the traffic rules and react to other road traffic participants appropriately. Our key insight for overcoming this challenge is to use the ADS under test to control all vehicles in the simulation to achieve this required high intelligence, enabling them to follow traffic rules and react to each other. This design

also guarantees that when a violation occurs among the vehicles, the ADS must have misbehaved and is thus responsible for the violation. This ideology of running multiple instances of the same ADS, i.e., doppelgänger instances, and observing interaction among them is what we call *doppelgänger testing*.

Figure 2 shows an overview of DoppelTest. Its main goal is to generate scenarios that may expose the ADS to safety and traffic rule violations. DoppelTest achieves this goal as follows: *Map Parser* uses an HD Map to analyze trajectories that vehicles and pedestrians are allowed to travel on and the necessary constraints that need to be satisfied to produce a valid traffic control configuration at junctions; *Scenario Generator* uses a genetic algorithm that produces scenarios using the constraints created by *Map Parser* and evolves the scenarios with the aim of finding safety and traffic rule violations; *Doppelgänger Orchestrator* converts the scenario representation generated in the previous step to a simulation by coordinating every doppelgänger AV instance available and produces a *Scenario Record* to be analyzed for each instance; *Violation Analyzer* analyzes each *Scenario Record* and evaluates it for safety and traffic rule violations. When the evolution process terminates, every scenario with a bug-revealing violation is saved as output. In the remainder of this section, we discuss each of these elements of DoppelTest in more detail.

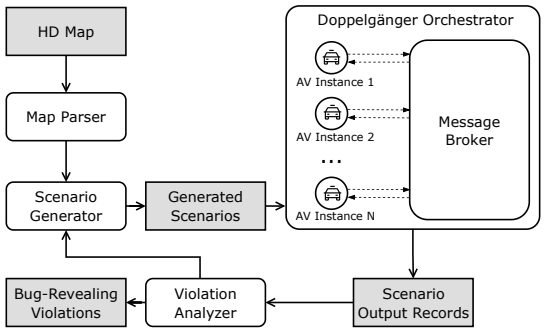


Fig. 2: Overview of DoppelTest

### A. Doppelgänger Orchestrator

One of the main contributions of our work is the design of a testing technique that enables multiple instances of the same ADS to be running in the same scenario without the need for an external simulator. To enable this feature, there needs to be an infrastructure which manages interaction between multiple instances. However, the ADS' architecture does not have an interface that provides native support for communication between other instances of the same software system. Therefore, we design the Doppelgänger Orchestrator (DO), which retrieves and transmits necessary information so that each AV instance is aware of the existence of other instances. To accomplish this goal, DO keeps track of where each instance is positioned by subscribing to the Localization module of each AV instance. Recall from Section II-A that Localization publishes the location, heading, velocity, and other information about the AV. Using the location and heading, DO then constructs a polygon that represents the vehicle's 3D bounding box.

Now that DO recognizes the location of every AV instance, its next goal is to transmit the 3D bounding box of one AV instance to other instances. Recall from Section II-A that Perception is responsible for identifying the physical world surrounding an AV instance, it produces messages that include 3D obstacles with heading and velocity. The information which DO tracks are location, heading, and velocity of every AV instance in a simulation, which is exactly the kind of information the Perception module in an ADS is supposed to track. Therefore, for each AV instance, DO serves as its Perception module and publishes 3D bounding boxes of all other instances. To summarize, DO uses each AV instance's localization information to serve as the perception information for each other AV instance.

### B. Scenario Generator

Prior AV test-generation techniques [20]–[22] produce scenarios that include only one AV controlled by the ADS per scenario with a fixed number of obstacles. In DoppelTest, there may be 2 or more doppelgänger AV instances and 0 or more pedestrians in a scenario; therefore, the genetic representation needs to be designed to allow increasing or decreasing of the number of vehicles and pedestrians. In the remainder of this section, we elaborate on the design of the genetic algorithm and its search operators.

*1) Representation:* DoppelTest uses the definition of a scenario in Section III along with a genetic algorithm to generate bug-revealing scenarios. Figure 3 illustrates a genetic representation (chromosome) of an individual generated by DoppelTest. The population in our approach is a set of scenario representations used by the Doppelgänger Orchestrator to run an actual scenario. Each individual, which is a scenario in our representation, consists of 3 sections: (1) the AV section specifies 2 or more AV instances $\mathbb{A}$ in the scenario; (2) the pedestrian section specifies 0 or more pedestrians $\mathbb{P}$ in the scenario; and (3) the traffic configuration section specifies a traffic signal configuration $\mathbb{T}$. The number of genes in the AV and pedestrian sections varies based on how many AVs or pedestrians there are in a scenario. For example, the size of genes in the AV section will be 6 for 3 AD instances as illustrated in Figure 3 and 8 for 4 AD instances.

Another main contribution of our work is the design of a variable length chromosome (VLC) to represent scenarios that do not have a fixed number of road traffic participants. By allowing variable length in the AV and pedestrian sections, the genetic algorithm has the ability to add or remove road traffic participants to generate more diverse scenarios. The number of genes is bounded by the maximum and minimum allowed AV instances and pedestrians in the scenario.

*2) Search Operators:* Multi-sectional design of the VLC allows each of the sections to be independent of each other, thus modifying one section will neither invalidate any other sections nor the entire chromosome. In turn, the search operator

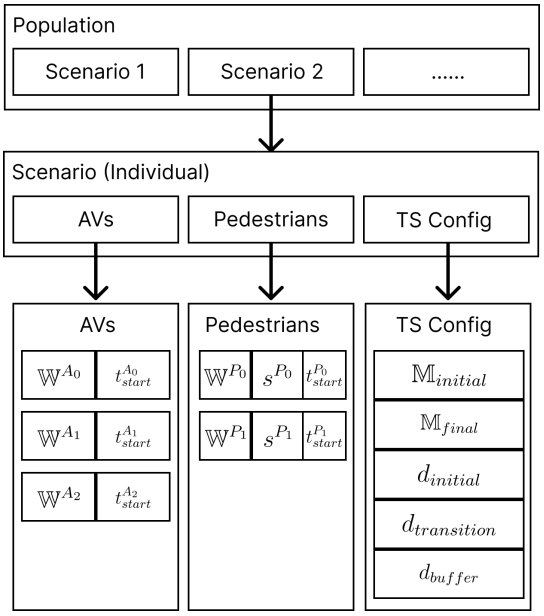


Fig. 3: Genetic representation used by DoppelTest

for such complex chromosomes can be designed independently of each other.

**Mutation**. With equal probability, one of the 3 sections of the chromosome will be mutated to increase diversity in the population. Mutating a specific section implies one of the genes in the section will be mutated. For example, the *yellow change interval* of the traffic configuration section, $d_{transition}$, may be selected for mutation. For sections that allow variable length (e.g., the AV and pedestrian sections), additional operations such as adding or removing a road traffic participant from that section are allowed. For example, after mutating the chromosome illustrated in Figure 3, there may be an additional AV instance $A_3$ represented by 2 genes $\mathbb{W}^{A_3}$ and $t^{A_3}_{start}$.

**Crossover**. With equal probability, one of the 3 sections of the chromosome will be selected for crossover to produce offspring. If the traffic configuration section is selected for crossover, then one of the genes from parent A will be replaced by a gene of the same type from parent B. If the AV Section is selected for crossover, then (1) 1 gene of an AV instance from parent $AVs_{p1}$ is selected to be replaced by the gene of the same type from a random AV instance from parent $AVs_{p2}$, resulting in $AVs_{o1}$ as shown on the left of Figure 4(b); (2) an instance of parent $AVs_{p1}$ is selected to be replaced by a random instance from parent $AVs_{p2}$, resulting in $AVs_{o2}$ as shown on the right of Figure 4(b); (3) an instance of parent $AVs_{p2}$ is randomly selected and added to parent $AVs_{p1}$ if the AV section of parent $AVs_{p1}$ has not reached the maximum allowed AV instances. The crossover operator for the pedestrian section is similar to the one for the AV section.

**Selection**: DoppelTest uses the NSGA-II (**N**on-dominated **S**orting **G**enetic **A**lgorithm selection [35] for breeding the next generation. NSGA-II is an effective algorithm used for solving multi-objective optimization problems (i.e., problems

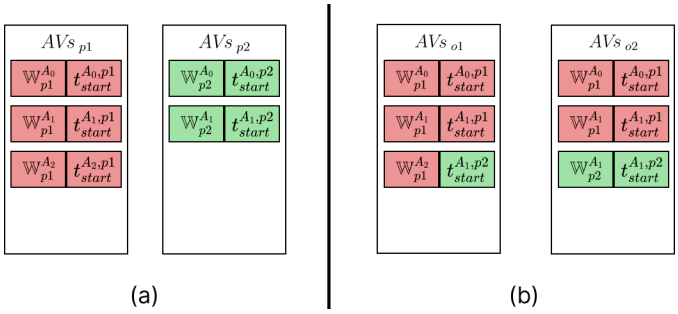


Fig. 4: An example crossover operation of the AV section. (a) 2 parent individuals selected for crossover; (b) 2 examples of offspring that may be produced.

with multiple conflicting fitness functions) and further aims to maintain diversity of individuals.

NSGA-II starts by sorting a set of individuals based on a *non-dominated* order. In a multi-objective problem, an individual $i_1$ is said to *dominate* another individual $i_2$ if (1) $i_1$ is no worse than $i_2$ for **all** objective functions (e.g., collision detection, speeding detection, etc.), and (2) $i_1$ is strictly better than $i_2$ in at least one objective. Once the non-dominated sort is complete, a *crowding distance* is assigned to every individual in a given scenario. A *crowding distance* measures how close individuals are to each other; a large average crowding distance will result in better diversity in the population. Once the crowding distance is assigned, parent individuals are selected to produce offspring based on the fitness and crowding distance; an individual is selected if its order rank is lower than the other, or if the crowding distance is greater than the other. Only the best $N$ individuals are selected, where $N$ is the population size.

The intuition behind using NSGA-II selection is threefold: (1) it uses an elitist principle, i.e., the most elite individuals in a scenario are given the opportunity to be reproduced so their genes can be passed on to the next generation; (2) it uses an explicit diversity-preserving mechanism (i.e., crowding distance), which maintains the diversity of driving scenarios in DoppelTest; and (3) it emphasizes the non-dominated solutions.

*3) Fitness Evaluation:* In each generation, DoppelTest evaluates individuals using the fitness function defined in Equation 1.

$$F(i) = (f_{min\_distance}(i), f_{decision}(i), f_{conflict}(i), f_{violation}(i)) \quad (1)$$

As an overview, the genetic algorithm's 4 objectives are ($f_{min\_distance}$) to encourage AV instances to be in close proximity with other road traffic participants to increase the likelihood of interaction; ($f_{decision}$) to maximize the number of decisions to be made by AV instances so that the scenario can be complex in a way that involves multiple AV instances making multiple decisions with regard to each other; ($f_{conflict}$) to maximize the number of conflicting trajectories between AV instances and other road traffic participants so that a collision violation may occur if an AV instance fails to yield right-of-way; and ($f_{violation}$) to maximize the number of violations that occur in

a scenario. In the remainder of this section, we discuss each objective in more detail.

$f_{min\_distance}(i)$ (Equation 2) evaluates the closest distance between pairs of road traffic participants at any time during a scenario.

$$f_{min\_distance}(i) = min\{D_t(A,O) : 0 \leq t \leq t^S \wedge A \in \mathbb{A} \wedge O \in \mathbb{O} \wedge O \neq A\} \quad (2)$$

where $D_t(A,O)$ is the distance between an AV $A$ and another road traffic participant $O$ at timestamp $t$ during the scenario. $O$ can be either a vehicle or pedestrian in the scenario. $f_{min\_distance}$ captures the intuition that tests with vehicles driving close to other road traffic participants are more likely to lead to interaction and may lead to violations such as collisions.

$f_{decision}(i)$ (Equation 3) evaluates the total number of unique decisions being made by all AV instances for the entire duration of the scenario.

$$f_{decision}(i) = max\{|\bigcup_{i=0}^{|\mathbb{A}|} \mathbb{D}^{A_i}| : A_i \in \mathbb{A}\} \quad (3)$$

Recall from Section III that $|\mathbb{A}|$ is the number of AV instances in the scenario and $\mathbb{D}^{A_i}$ the set of unique decisions made by the *ith* AV instance. The intuition is to encourage scenarios with multiple AV instances to make more decisions with respect to the environment, which creates complex scenarios such as the one we will be discussing later in Case Study 2 of Section V-A.

$f_{conflict}(i)$ (Equation 4) evaluates the total number of pairs of road traffic participants whose trajectory overlaps with each other. The function is defined as follows:

$$f_{conflict}(i) = max\{|\{\langle A,O\rangle : A \in \mathbb{A} \wedge O \in \mathbb{O} \wedge A \neq O \wedge X(A,O)\}|\} \quad (4)$$

where $X(A,O)$ evaluates if $A$ and $O$ have a common waypoint and is defined as follows:

$$X(A,O) = \exists w_i \exists w_j (w_i \in \mathbb{W}^A \wedge w_j \in \mathbb{W}^O \wedge w_i = w_j) \quad (5)$$

The intuition is to encourage scenarios with AV instances that may need to yield the right-of-way during a scenario, since reaching the same location at the same time as another road traffic participant will result in a collision.

$f_{violation}(i)$ (Equation 6) evaluates the total number of violations across all AV instances in a scenario and is defined as follows:

$$f_{violation}(i) = max\{|\bigcup_{i=0}^{|\mathbb{V}|} V(A_i)| : A_i \in \mathbb{A}\} \quad (6)$$

The intuition is to aim for scenarios where multiple AV instances may be involved in multiple violations. We define each of the violations in more detail in the next section.

### C. Violation Analyzer

Prior work [20]–[22] only considered a limited number of oracles, such as collision. The limited number of oracles ignores other types of misbehavior such as violating traffic rules. Unlike prior work, we consider 4 test oracles that assess an ADS' ability to avoid (1) collision violations, (2) traffic signal violations, (3) stop sign violations, and (4) module response failures. The oracles are designed with respect to each AV instance. In the remainder of this section, we describe each oracle in more detail along with its corresponding oracle definition.

The *collision oracle* uses the position of every road traffic participant to check if a vehicle ever collided with another vehicle or pedestrian. The test oracle's failing condition for a single AV instance $A$ is defined as follows:

$$\exists t_i, O : 0 \leq t_i \leq t^S \wedge O \in \mathbb{O} \wedge O \neq A \wedge speed(A,t_i) > 0 \wedge D(position(A,t_i), position(O,t_i)) = 0 \quad (7)$$

where $speed(A,t_i)$ evaluates the AV instance's speed at time frame $t_i$, $position(O,t_i)$ evaluates the location of a road traffic participant, and function $D$ evaluates the distance in meters between the 2 locations. This oracle models the case where a vehicle collides with another road traffic participant at a positive speed.

The *traffic signal oracle* uses stop lines that are associated with traffic signals on a given map and the AV instance's location to check if the instance is following traffic rules regarding traffic signals by stopping when facing a red signal. [36], [37] The test oracle's failing condition for AV instance $A$ is defined as follows:

$$\exists t_i, l_{ts} : 0 \leq t_i \leq t^S \wedge color(ts,t_i) = \texttt{RED} \wedge speed(A,t_i) > 0 \wedge D(position(A,t_i), l_{ts}) = 0 \quad (8)$$

where $color(ts,t_i)$ evaluates the color of traffic signal $ts$ at time frame $t_i$, $D(position(A,t_i), l_{ts})$ evaluates the distance between the AV instance $A$ and a stop line $l_{ts}$ for traffic signal $ts$. When the distance evaluates to 0, it implies the vehicle is attempting to cross the stop line at a positive speed. The oracle models the intuition that the vehicle should not be crossing any stop line whose corresponding traffic signal shows red.

The *stop sign oracle* uses stop lines that are associated with stop signs on a given map and the AV instance to check if the instance made a complete stop before crossing the stop line. [38], [39] The test oracle's failing condition for a single AV instance $A$ is defined as follows:

$$\exists t_i \nexists t_j \exists l_{ss} : 0 \leq t_j < t_i \leq t^S \wedge D(position(A,t_i), l_{ss}) = 0 \wedge D(position(A,t_j), l_{ss}) < th_{stop\_distance} \wedge speed(A,t_j) = 0 \quad (9)$$

where $t_i$ is a time frame in the scenario and $t_j$ is another time frame before $t_i$, $l_{ss}$ is the stop line corresponding to stop sign $ss$; $D(position(A,t_i), l_{ss})$ evaluates if the vehicle is crossing the stop line $l_{ss}$; and $th_{stop\_distance}$ is a threshold that determines how far away the vehicle should be stopped before a stop line. The intuition here is that if there exists a time frame $t_i$ where the AV instance is crossing a stop sign's stop line, then there must exist another time frame $t_j$ which came before $t_i$ at which the AV instance is stopped within $th_{stop\_distance}$ meters away from the stop sign.

The *module response* oracle checks if an AV instance responded to a routing request with a routing response. Failure to produce a routing response directly impacts the AV's ability

to reach its destination, which we consider a kind of violation. The test oracle's failing condition is defined as follows:

$$\nexists t_i : 0 \leq t_i \leq t^S \wedge \mathcal{R}(i) \tag{10}$$

where $t_i$ is a time frame in the scenario and $\mathcal{R}$ is a function checking if a routing response has been produced by the routing module. The intuition is that the AV instance never responded to a routing request and thus will not attempt to travel toward its specified destination. Note that the routing request is valid if there exists a legal path (e.g., it does not require driving against the traffic direction) on the map that allows the vehicle to travel from the initial position to its final position, and a valid routing response is to be expected given a valid request.

### D. Map Parser

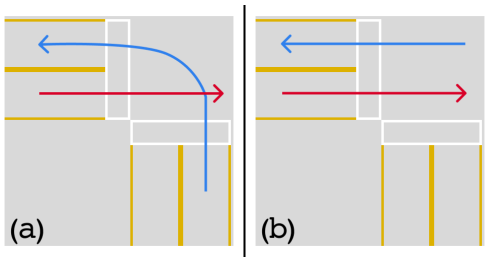


Fig. 5: Examples of lanes that may (a) have conflict, or (b) have no conflict.

The Map Parser's main goal is to analyze constraints between traffic signals that are at the same junction. The intuition is that traffic signals at the same junction controlling lanes that intersect with each other cannot be green at the same time, as shown on the left of Figure 5. If the traffic signals indicate both lanes have right-of-way then a collision may likely happen at the intersecting point. However, if traffic signals at the same junction are controlling lanes that will never intersect with each other, as shown on the right of Figure 5, then there will not exist any constraint between them as they can be in any color without intentionally causing conflicts between vehicles traveling along the lines indicated in the figure.

## V. Evaluation

Our evaluation process involves executing a total of 240 hours of test generation. We conducted our experiments on two machines: 2 AMD EPYC 7551 32-Core Processor (512GB of RAM), 1 AMD EPYC 7551 32-Core Processor (256GB of RAM), 1 Core i9 24-Core Processor (128GB of RAM), and 1 Core i9 24-Core Processor (32GB of RAM) all running Ubuntu 20.04.1. We focus our efforts on testing Baidu Apollo 7.0 [31], an open-source and production-grade AV software system that supports a wide variety of driving scenarios and explicitly aims for both safety and rider's comfort. Our experimentation focused on an real-world HD Map provided as part of Baidu Apollo, which represents Borregas Avenue at Sunnyvale, CA. The map includes multi-lane roads, 1 traffic signal controlled junction, and 1 stop sign junction.

We do not compare DoppelTest against certain approaches due to the fact that they address different research problems. For example, [40] and [41]–[43], are merely concerned with generating road networks. To the best of our knowledge, there is no existing search-based testing approach for production-grade AV software, including [20], [21], [41]–[55] that: (i) uses our novel gene representation, (ii) does not require a manual setup of driving scenarios with fixed scenario attributes (e.g., fixed number of obstacles, fixed obstacle trajectories, limited maneuvers, etc.), (iii) automatically generates a diverse set of scenario types, and (iv) considers traffic signals.

In order to empirically evaluate DoppelTest, we investigate the following research questions:

- **RQ1**: To what extent does DoppelTest find different types of bug-revealing violations?
- **RQ2**: Is DoppelTest more efficient in revealing bugs compared to baseline technique that uses unintelligent obstacles?
- **RQ3**: What is the runtime efficiency of DoppelTest?

### A. RQ1: Bug-Revealing Violations Discovered

DoppelTest generated a total of 165 scenarios that contain violations, each of which we then manually verified to determine (1) if it has at least one AV instance responsible for the violation, (2) the total number of violations found, and (3) the total number of bug-revealing violations. These results are shown in Figure 6. We determine an AV instance is responsible by checking whether the ADS made a sub-optimal decision that directly caused the violation. For example, if a collision occurred because the ADS did not make a yield decision to other vehicles or pedestrians who have right-of-way. 42 collisions (33.6% of total collisions found) were determined to not be bug-revealing because every collision involves the pedestrian walking into a slow moving vehicle who has the right-of-way. This type of violation is not bug-revealing because the pedestrian's unintelligent modeling of following waypoints, ignoring other road traffic participants, and ignoring traffic rules directly caused its collision with a vehicle.

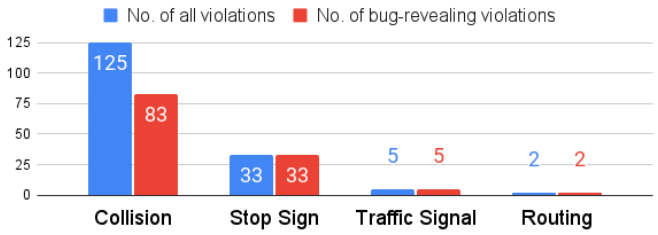


Fig. 6: Number of violations and bug-revealing violations generated by DoppelTest

The remaining 123 bug-revealing scenarios are then categorized into 8 unique types of bugs. 5 of which may result in collision, 2 of which may result in traffic rule violations, and 1 which results in routing module failure. We describe the categories as follows:

TABLE I: Number of scenarios found for each type of bug

| | Category | No. of Viol. |
|---|---|---|
| 1 | Failure to recognize slow dynamic obstacle (FRS) | 19 |
| 2 | Failure to react to obstacle heading change (FRH) | 41 |
| 3 | Improper overtake decision (IOD) | 2 |
| 4 | Unsafe lane borrow decision (ULB) | 9 |
| 5 | Incorrect prediction trajectory (IPT) | 12 |
| 6 | Stop sign rolling stop (SSRS) | 33 |
| 7 | Planning ignores traffic signal (TIS) | 5 |
| 8 | Routing failure (RF) | 2 |

- **Failure to recognize slow dynamic obstacle** (FRS): A dynamic obstacle slowly moves onto the planning trajectory and the AV fails to make a new collision-free trajectory.
- **Failure to react to obstacle heading change** (FRH): A dynamic obstacle suddenly changes its direction, and the AV fails to recognize the obstacle's action in a timely manner.
- **Improper overtake decision** (IOD): The AV attempts to overtake another vehicle coming from a different direction at a junction, when it should yield right-of-way.
- **Unsafe lane borrow decision** (ULB): The AV borrows the lane of opposing traffic to overtake a static obstacle, and collides into another vehicle when attempting to return to its original lane.
- **Incorrect prediction trajectory** (IPT): Prediction produces an incorrect prediction trajectory causing the AV to fail to recognize a conflict with the obstacle's trajectory.
- **Stop sign rolling stop** (SSRS): The AV decelerates to a low speed ($\approx 0.2$ m/s) but does not make a complete stop before crossing a stop sign.
- **Planning ignores traffic signal** (TIS): The AV stops on the stop line and decides to accelerate and clear intersection when facing a steady red signal.
- **Routing failure** (RF): Routing fails to produce a routing response because it cannot recognize that a path exists from one lane to a reachable lane.

For each of the above categories, DoppelTest was able to automatically generate scenarios and detected them through its oracles. The number of occurrences of each bug type is shown in Table I. In the following paragraphs, we discuss 2 specific bug-revealing violations that are found by DoppelTest.

**Case Study 1: Planner Ignores Red Traffic Signal**. Apollo uses a reference line in front of the vehicle to check for traffic control devices (e.g., a traffic signal or stop sign) it should consider when making a planning decision. When the reference line overlaps with a stop line controlled by a traffic signal, it will make a stop decision if the signal is red or yellow.

In the scenario shown in Figure 7, the vehicle is about to reach a junction at which the signal for the lane is green. As the vehicle approaches the stop line, the signal changes to yellow and is about to change to red. Apollo initially makes a stop decision for the signal but was unable to stop with a reasonable deceleration before the stop line, making the vehicle

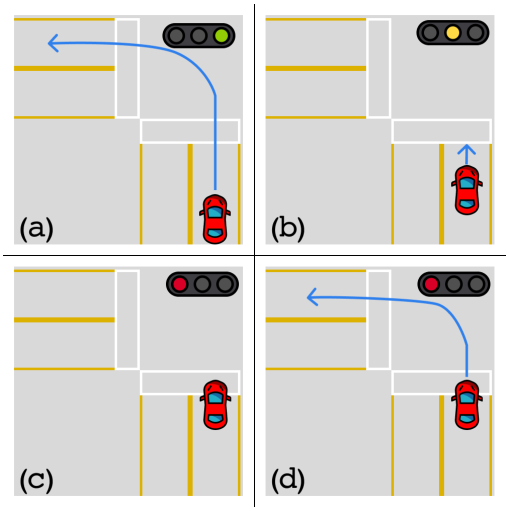


Fig. 7: Planner Ignores Traffic Signal: The blue line indicates AV's planned trajectory. First, (a) the AV is planning to turn left at the junction, then (b) it attempts to stop before the stop line as the signal is yellow. (c) The AV stops on top of the stop line due to limited time and distance and (d) ignores the red signal and resumes turning left.

stop directly on top of the stop line. Notice that in Figure 7(c), the vehicle stops on top of the stop line, and its rear side has not crossed the stop line. However, since the vehicle's head already passed the stop line, Apollo no longer considers the signal ahead and makes a new planning decision to resume a left turn. As a result, Apollo already stopped on the stop line when the signal changed to red, but accelerated and resumed its left turn, causing a red light violation.

The written law does not specify every detail about where the vehicle can be during the yellow change interval. Thus, we confirmed with a police officer that the vehicle's action of stopping on the stop line then accelerating when signal is red will result in a citation due to a red light violation. To fix this bug, Apollo should take the length of the vehicle into consideration when checking for traffic control devices on its path, rather than using a reference line that is only in front of the vehicle.

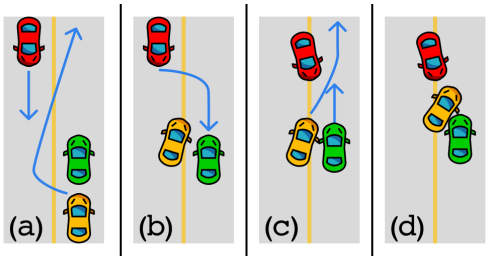


Fig. 8: Unsafe Lane Borrow: The blue line indicates the vehicle's planned maneuver. All 3 vehicles are controlled by three instances of the same ADS software.

**Case Study 2: Unsafe Lane Borrow Decision**. Sometimes a

vehicle has to perform an illegal or risky maneuver to reach its destination, for example, as shown in Figure 8(a), the yellow vehicle plans to go around a stopped vehicle (the green AV in Figure 8) by borrowing the lane with opposing traffic. As the yellow AV proceeds according to its planning trajectory, it recognizes a vehicle (the red AV in Figure 8) that has the right of way and decides to yield—resulting in the yellow AV stopping on top of the yellow line separating traffic going in opposing directions. With the yellow AV stopped, the red AV recognizes there is a static obstacle blocking its path in its original lane and attempts to go around it by borrowing the lane with opposing traffic, as shown in Figure 8(b). The red AV completes half of its planned maneuver and realizes the plan cannot be accomplished because of the existence of the green AV, making the red AV stop on the yellow line as well. Since the red AV, which the yellow AV yielded right-of-way to, is now stopped, the yellow AV resumes its goal to return to its original lane, ignoring the green AV who is now traveling on the lane at 10 m/s, as shown in Figure 8(c). As a result, the yellow AV collides with the green AV, shown in Figure 8(d). This is one of the most complex scenarios that DoppelTest finds which involves 3 AV instances all making decisions with regards to its own goal and its surrounding environment, and it was not discovered prior to DoppelTest.

One partial fix available in Apollo is to simply disable the AV from crossing the yellow line. However, in real-life situations, it may be necessary for the AV to cross the yellow line to overtake the vehicle and may even be legal in some cases. A remedy that can still allow overtaking other vehicles in this case would involve having both the red or yellow AV yield to the green AV. Thus, the ADS can be more cautious toward the end of a lane borrow maneuver to check for vehicles traveling in the lane it is moving into, or the ADS can be more cautious when seeing a vehicle is trying to change lanes in front of it.

**Finding 1**: DoppelTest effectively generates scenarios that reveal 123 bug-revealing violations for 8 types of bugs in an industry-grade autonomous driving software system.

### *B. RQ2: Effectiveness of DoppelTest*

RQ2 evaluates the effectiveness of DoppelTest by comparing our results with a random version that generates scenarios with 1 AD instance and multiple unintelligent obstacles that travel at a constant speed, ignore traffic rules, and ignore other road traffic participants, which we will refer to as `RAND`. Such a technique has similar capabilities as existing state-of-the-art techniques [20]–[22]. We do not compare against implementations of current state-of-the-art techniques because they require the use of an external simulator that is no longer maintained [56]. Moreover, the simulator has a known issue [57] involving obstacle kinematics. For example, specifying an obstacle to travel at 5 $m/s$ and then travel at 10 $m/s$ will result in an obstacle suddenly changing its speed rather than accelerating or decelerating. The simulator's simplification causes scenarios to be physically unrealistic, thus impacting the ADS testing approaches that depend on the simulator, making them not bug-revealing for the ADS. Figure 9 shows the results of our comparison of DoppelTest and `RAND`. Given the same amount of time (240 hours), `RAND` discovered 1109 scenarios with collisions, which is a significantly higher quantity compared to how many DoppelTest was able to find. However, verifying each of the 1109 scenarios generated by `RAND` reveals they are all solely the obstacle's responsibility. 1097 (99% of total) involve the obstacle rear ending the AV, which is either stopped at a stop line or is cruising normally; 12 non-rear-end cases involve obstacles not following traffic rules while Apollo makes a prediction assuming the vehicle will follow traffic rules. We discuss an example in the following case study.

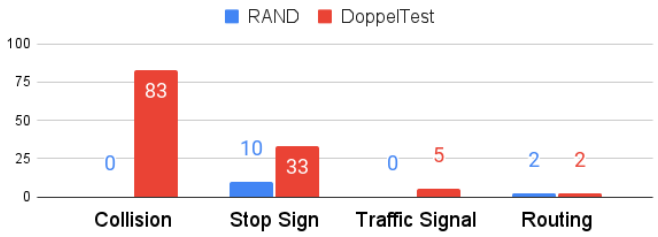


Fig. 9: Number of bug-revealing violations generated by a baseline random approach and DoppelTest

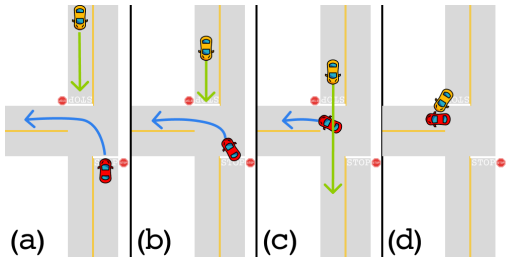


Fig. 10: An illustration of a collision violation that is non-bug-revealing generated by our random baseline. The red vehicle is an AV instance, and the yellow vehicle is an unintelligent obstacle. The blue line is the AV's planned maneuver; the green line is the AV's predicted maneuver for the obstacle. The obstacle is traveling at a constant 10 m/s.

**Case Study 3: Non-Bug-Revealing Collision**. In this case study, we demonstrate how a collision may not be bug-revealing because the AV never misbehaves or violates a traffic law or rule. Figure 10(a) shows the AV executing a complete stop at the stop sign and plans to move forward because it recognizes that the intersection is clear and it is safe to proceed. As it enters the intersection, shown in Figure 10(b), it predicts the obstacle will stop at the stop sign. In Figure 10(c), the AV recognizes that the obstacle does not slow down and, therefore, predicts it will go straight at the intersection. This prediction is reasonable because drivers often decelerate when making a turn. As a result, the obstacle collides with the AV shown in Figure 10(d). The AV did not predict there will be a conflict in trajectory between the vehicles, so the AV could not avoid the

collision, which is solely the obstacle's responsibility because (1) it did not stop for the stop sign and (2) it did not decelerate during any stage of the scenario and completed a right turn at an unrealistic speed.

**Finding 2**: Generating violations is different from generating bug-revealing violations. A random baseline technique can generate thousands of violations with collisions but none of them are actually bug-revealing. In contrast, DoppelTest effectively finds bug-revealing violations that the ADS has responsibility for.

### C. RQ3: Runtime Efficiency of DoppelTest

In RQ3, we study the runtime efficiency of DoppelTest by measuring its execution time. Table II shows that DoppelTest takes 47.48 seconds on average to execute a scenario from Scenario Generator to Violation Analyzer in total (i.e., executing all components together). Map Parser is a one-time initialization that takes on average 2.66 seconds to analyze the given HD Map. The Doppelganger Configuration is another one-time initialization that takes 67.42 seconds in total to configure 5 Apollo instances, on average 13.5 seconds per Apollo instance. Scenario generation takes 8.80 seconds on average to produce each scenario while running the scenario in the Doppelganger Orchestrator phase takes 35.94 seconds to execute. After running the scenario, it takes about 2.74 seconds on average to analyze each scenario.

From the experimental results, we examined the time cost of executing each main component in DoppelTest. The initial map parsing and final violation analysis do not account for much time to run, which is negligible. The most time-consuming phase is to run the scenario with multiple Apollo instances. We also compare our approach with state-of-the-art technique, SCENORITA [22], using the paper's reported results. Overall, DoppelTest can save 15.63%, 13.44%, and 77.11% time in Scenario Generator, Scenario Generator, and Violation Analyzer phase for each scenario, respectively. Note that SCENORITA does not have a Doppelgänger Orchestrator, which is the main contribution of our work. We compare this component with a similar component in SCENORITA whose purpose is to execute a scenario. From the results, we can conclude:

**Finding 3**: Each individual component of DoppelTest is efficient with an average execution time of 47.48 seconds for each scenario, which means DoppelTest can be used to efficiently generate driving scenarios that expose AV systems to bug-revealing violations in practice. DoppelTest can save 77.11% time in Violation Analyzer phase and 25.72% time in total compared with SCENORITA for each scenario to run.

## VI. Threats to Validity

**Internal Threats** One potential threat to internal validity is the selection of scenario duration: Simulation-based tests require the execution of time-consuming computer simulations to produce violations. We determined from our experimentation that our selected scenario duration of 30 seconds finds a significant number and variety of violations without incurring drastically long test execution times. Furthermore, there is no agreed-upon threshold in related work that dictates the correct scenario duration.

Another threat to validity arises from verifying the correctness of generated tests. There are unfortunately no automated strategies nor a ground truth that can be used to assess the accuracy of generated scenarios. For that reason, we had to manually verify the generated tests. We further make DoppelTest's implementation and case study videos available (Section IX) to others to further check our results.

A threat to internal validity is the non-determinism of the ADS: The differing outputs of the ADS modules for each re-execution may impact the outcome of a scenario. To mitigate this threat, DoppelTest attempts to ensure a fresh state for each re-execution of a scenario by precisely controlling the start time and location of each road traffic participant and resetting each of their states.

As a final threat to internal validity, the seed selection may impact the genetic algorithm's effectiveness and efficiency at finding violations. Nevertheless, in our evaluation, initial seeds are random, which can generate a significant variety of bug-revealing violations. At the same time, more ADS instances will produce more diverse and complicated scenarios; an example is discussed in Section V-A Case Study 2. A minimum of 2 ADS instances is enough to produce all types of bug-revealing violations discussed in the paper, either by interacting with each other, interacting with pedestrians, or operating on maps with traffic control devices.

**External Threats** One external threat is that we applied DoppelTest to a single AV software system, Apollo. To mitigate the threat, we selected the only high autonomy (i.e., Level 4), open-source, production-grade AV software system that supports a wide variety of driving scenarios and explicitly aims for both safety and driver comfort. We ran experiments on the Apollo ADS for 240 hours in total. Note that Autoware [58], despite being open-source and widely-used [59], is considered a research-grade and not a production-grade AV software system [60], [61], which we further verified through speaking with Christian John, the Vice Chair and Chief Software Architect of Autoware.

**Construct Validity** The main threat to construct validity is how we measure and evaluate traffic rule violations. To mitigate this threat, we review interpretations of the traffic law from multiple legal experts [36]–[39]. We further spoke with police officers of the state for the map we used to confirm the AV instance's action violates traffic laws.

## VII. Related Work

A wide array of studies focus on applying traditional testing techniques to AVs including adaptive stress testing [62], where noise is injected into the input sensors of an AV to cause accidents; fitness function templates for testing automated and autonomous driving systems with heuristic search [63]; and

TABLE II: Runtime efficiency(s) of DoppelTest and SCENORITA

| | Map Parser | Doppelgänger Configuration | Scenario Generator | (Doppelgänger) Orchestrator | Violation Analyzer | *E2E* |
|---|---|---|---|---|---|---|
| DoppelTest | 2.66 | 67.42 | 8.80 | 35.94 | 2.74 | 47.48 |
| SCENORITA | N/A | N/A | 10.43 | 41.52 | 11.97 | 63.92 |

search-based optimization [64]. These studies provide limited insights into the testing of real-world AVs, since they do not evaluate their techniques on open-source, production-grade AV software. Other related work focuses on the vision, multi-sensor fusion, and machine-learning aspects of AV software [65]–[73] all of which ultimately focus on the Perception and Prediction components of ADS. Unlike these approaches, DoppelTest focuses on AV scenario simulations and Planning, which is known to have more bugs than Perception and Prediction combined [33].

Erbsmehl [74] recreates crashes by replaying the sensory data collected from the physical world. Similarly, *AC3R* [75] generates driving simulations that reproduce car crashes from police reports using natural language processing (NLP). However, *AC3R* requires manual collection of police reports and inherits the accuracy limitations of the underlying NLP used to extract information from police reports.

Gambi et al. proposed a search-based approach, AsFault, that combines procedural content generation to automatically create suitable virtual scenarios for testing AVs [40]. However, AsFault only used one test oracle to test the lane-keeping feature of autonomous driving [76]. Similarly, tools published as part of the Search-Based Software Testing Challenge (SBST) [50], [51] generate challenging road networks for virtual testing of an automated lane keep system such as GABezier [41], Frenetic [42], WOGAN [77], and Deeper [43]. Birchler et al. proposed SDC-Scissor [78], which focuses on predicting whether a particular road structure may cause ADS violations without executing the scenario, and is similar to the focus of the SBST tool competition. However, none of these tools take into account the behavior of other obstacles when testing for safety violations in AVs.

Next we discuss the wide variety of approaches that aim to generate scenarios with violations: Li et. al. [55] generate scenarios that consider the safety and comfort of AVs. Luo et. al. propose a framework (EMOOD) [54], which evolves tests to identify combinations of requirements violations. Chen et al. proposed a reinforcement learning approach to generate challenging scenarios for adversarial evaluation on purpose [79]. AV-Fuzzer is a AV testing framework for generating diverse scenarios and finding safety violations caused by AVs [20]. They classify the safety violations of Baidu Apollo reported by AV-Fuzzer into 5 different types and discuss the major causes of these violations. AutoFuzz [21] uses neural networks in the process of evolutionary search to generate more complex and valid scenarios for AV testing [21]. SCENORITA is a search-based testing framework, which exposes ADS to 5 types of violations in generated scenarios and reduces duplicate scenarios. It supports fully mutable obstacles with valid obstacles trajectories and automates the scenario configuration [22]. Unlike DoppelTest, these techniques do not aim to maximize finding bug-revealing violations, as opposed to just any violation and they do not explicitly account for traffic signals, stop signs, or module response violations.

Calò et al. [80] introduce two search-based approaches to find avoidable collisions for AVs. Unlike DoppelTest, this work was applied to a closed-source ADS, does not specifically handle traffic signal configurations, and does not handle violations other than collisions. Wachi et al. propose a method that uses adversarial reinforcement learning to find failure scenarios for AVs [81]. Unlike DoppelTest, they do not discuss or validate how their failure scenarios may be bug-revealing; they provide no public implementation of their technique, which is implemented on Microsoft AirSim, which is not a production-grade AV system, unlike Baidu Apollo; and do not explain how long they ran their scenarios.

## VIII. Conclusion

We presented DoppelTest, a framework that generates bug-revealing scenarios by making every vehicle an AV and models traffic control (e.g., traffic signals and stop signs). Our work addresses the key challenge of determining responsibility after an ADS is involved in a violation. The takeaway is the insight that, when all vehicles are controlled by an ADS, violations should never occur; if a violation occurs in such a setting, such a violation must be a bug in the ADS. We show that DoppelTest discovered 123 bug-revealing violations from 8 unique bug types in a production-grade ADS, Baidu Apollo 7.0 [31]. We submitted a fix to one type of bug DoppelTest discovered, which was accepted as part of the most recent version of Apollo. For future work, we aim to expand DoppelTest to handle more diverse dynamic obstacles including vehicles of different sizes and bicycles, more oracles for checking module failures, and extending work to other AV software systems.

## IX. Data Availability

The source code of DoppelTest is available at https://doi.org/10.5281/zenodo.7575582. Video recordings of the case studies are available at https://doi.org/10.5281/zenodo.7622399.

## X. Acknowledgments

We thank the anonymous reviewers for their constructive feedback. The authors gratefully acknowledge the support of NSF 1823262, 1929771, 1932464, and 2145493.

## REFERENCES


[1] "Waymo," August 2021. [Online]. Available: https://waymo.com/

[2] "Ford unveils new self-driving test vehicle for 2022 launch," August 2021. [Online]. Available: https://www.cnbc.com/2020/10/20/ford-unveils-new-self-driving-test-vehicle-for-2022-launch.html

[3] "Waymo and intel collaborate on self-driving car technology," August 2021. [Online]. Available: https://tinyurl.com/mvdkd7sz

[4] "Waymo's autonomous cars have driven 8 million miles on public roads," August 2021. [Online]. Available: https://www.theverge.com/2018/7/20/17595968/waymo-self-driving-cars-8-million-miles-testing

[5] "You can take a ride in a self-driving Lyft during CES," August 2021. [Online]. Available: https://bit.ly/3acRidI

[6] "Baidu starts mass production of autonomous buses," August 2021. [Online]. Available: https://bit.ly/3oJXrSy

[7] "Argo is reimagining the human journey. - Argo AI." [Online]. Available: https://www.argo.ai/

[8] M. Bertoncello and D. Wee, "Ten ways autonomous driving could redefine the automotive world," *McKinsey & Company*, vol. 6, 2015.

[9] "Tesla autopilot," August 2021. [Online]. Available: https://www.tesla.com/autopilot

[10] "Uber advanced technolgoy group," August 2021. [Online]. Available: https://www.uber.com/us/en/atg/

[11] "Self-driving uber car kills pedestrian in arizona, where robots roam," August 2021. [Online]. Available: https://www.nytimes.com/2018/03/19/technology/uber-driverless-fatality.html

[12] "There are some scary similarities between tesla's deadly crashes linked to autopilot," August 2021. [Online]. Available: https://qz.com/783009/the-scary-similarities-between-teslas-tsla-deadly-autopilot-crashes/

[13] "Two years on, a father is still fighting tesla over autopilot and his son's fatal crash," August 2021. [Online]. Available: https://jalopnik.com/two-years-on-a-father-is-still-fighting-tesla-over-aut-1823189786

[14] "Tesla driver dies in first fatal crash while using autopilot mode," August 2021. [Online]. Available: https://bit.ly/3nlNavo

[15] "Self-driving tesla was involved in fatal crash, u.s. says," August 2021. [Online]. Available: https://nyti.ms/3abv3Vq

[16] "Tesla: Autopilot was on during deadly mountain view crash," August 2021. [Online]. Available: https://www.mercurynews.com/2018/03/30/tesla-autopilot-was-on-during-deadly-mountain-view-crash/

[17] "Google's self-driving car caused its first accident," August 2021. [Online]. Available: https://bit.ly/3acQwgO

[18] "Tesla autopilot system found probably at fault in 2018 crash," August 2021. [Online]. Available: https://nyti.ms/3qXuioY

[19] N. Kalra and S. M. Paddock, "Driving to safety: How many miles of driving would it take to demonstrate autonomous vehicle reliability?" *Transportation Research Part A: Policy and Practice*, vol. 94, pp. 182–193, 2016.

[20] G. Li, Y. Li, S. Jha, T. Tsai, M. B. Sullivan, S. K. S. Hari, Z. Kalbarczyk, and R. K. Iyer, "AV-FUZZER: finding safety violations in autonomous driving systems," in *31st IEEE International Symposium on Software Reliability Engineering, ISSRE 2020, Coimbra, Portugal, October 12-15, 2020*, M. Vieira, H. Madeira, N. Antunes, and Z. Zheng, Eds. IEEE, 2020, pp. 25–36. [Online]. Available: https://doi.org/10.1109/ISSRE5003.2020.00012

[21] Z. Zhong, G. E. Kaiser, and B. Ray, "Neural network guided evolutionary fuzzing for finding traffic violations of autonomous vehicles," *CoRR*, vol. abs/2109.06126, 2021. [Online]. Available: https://arxiv.org/abs/2109.06126

[22] S. Almanee, X. Wu, Y. Huai, Q. A. Chen, and J. Garcia, "scenorita: Generating less-redundant, safety-critical and motion sickness-inducing scenarios for autonomous vehicles," *arXiv preprint arXiv:2112.09725*, 2021.

[23] E. Leefeldt, "The Tricky Business Of Determining Fault After A Car Accident," Jun. 2022, section: Car Insurance. [Online]. Available: https://www.forbes.com/advisor/car-insurance/determining-fault-after-accident/

[24] "How to determine who is at fault in a car accident in colorado." [Online]. Available: https://www.mcdivittlaw.com/practice-areas/auto-accident/determining-fault

[25] "How is fault determined in a car accident?" [Online]. Available: https://www.gjel.com/car-accident-lawyers/how-is-fault-determined.html

[26] "How to determine who bears liability for a car accident." [Online]. Available: https://www.vilesandbeckman.com/blog/who-bears-liability-car-accident/

[27] "Car accident liability." [Online]. Available: https://www.findlaw.com/injury/car-accidents/car-who-is-liable.html

[28] "How to prove liability in a car accident." [Online]. Available: https://calbizjournal.com/how-to-prove-liability-in-a-car-accident/

[29] C. Rödel, S. Stadler, A. Meschtscherjakov, and M. Tscheligi, "Towards autonomous cars: The effect of autonomy levels on acceptance and user experience," in *Proceedings of the 6th International Conference on Automotive User Interfaces and Interactive Vehicular Applications, Seattle, WA, USA, September 17 - 19, 2014*, L. N. Boyle, G. E. Burnett, P. Fröhlich, S. T. Iqbal, E. Miller, and Y. Wu, Eds. ACM, 2014, pp. 11:1–11:8. [Online]. Available: https://doi.org/10.1145/2667317.2667330

[30] "What are the 6 autonomy levels for autonomous vehicles?" https://www.perforce.com/blog/qac/6-levels-of-autonomous-driving, 2020.

[31] "Baidu apollo: An open autonomous driving platform." [Online]. Available: https://github.com/ApolloAuto/apollo

[32] "Baidu is building level 4 autonomous robotaxis called apollo moon in china," June 2021. [Online]. Available: https://tinyurl.com/39x7xtxc

[33] J. Garcia, Y. Feng, J. Shen, S. Almanee, Y. Xia, and a. Q. A. Chen, "A comprehensive study of autonomous vehicle bugs," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. Seoul South Korea: ACM, Jun. 2020, pp. 385–396. [Online]. Available: https://dl.acm.org/doi/10.1145/3377811.3380397

[34] "Lesson 8 - Federal Highway Administration University Course on Bicycle and Pedestrian Transportation, July 2006 - FHWA-HRT-05-099." [Online]. Available: https://www.fhwa.dot.gov/publications/research/safety/pedbike/05085/chapt8.cfm

[35] E. Zitzler and L. Thiele, "Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach," *IEEE Transactions on Evolutionary Computation*, vol. 3, no. 4, pp. 257–271, 1999.

[36] "What is considered "running a red light" in Calif? (21453 CVC)." [Online]. Available: https://www.shouselaw.com/ca/defense/vehicle-code/21453/

[37] "California VC 21453: Offenses Relating to Traffic Devices." [Online]. Available: https://www.simmrinlawgroup.com/california-vehicle-code/california-vehicle-code-section-21453/

[38] "Vehicle Code 22450 CVC - Running a Stop Sign in California." [Online]. Available: https://www.shouselaw.com/ca/defense/vehicle-code/22450/

[39] "California VC 22450: Special Stops Required | Simmrin Law." [Online]. Available: https://www.simmrinlawgroup.com/california-vehicle-code/california-vehicle-code-section-22450/

[40] A. Gambi, M. Müller, and G. Fraser, "Asfault: testing self-driving car software using search-based procedural content generation," in *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, J. M. Atlee, T. Bultan, and J. Whittle, Eds. IEEE / ACM, 2019, pp. 27–30. [Online]. Available: https://doi.org/10.1109/ICSE-Companion.2019.00030

[41] F. Klück, L. Klampfl, and F. Wotawa, "Gabezier at the sbst 2021 tool competition," in *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, 2021, pp. 38–39.

[42] E. Castellano, A. Cetinkaya, C. H. Thanh, S. Klikovits, X. Zhang, and P. Arcaini, "Frenetic at the sbst 2021 tool competition," in *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, 2021, pp. 36–37.

[43] M. H. Moghadam, M. Borg, and S. J. Mousavirad, "Deeper at the sbst 2021 tool competition: Adas testing using multi-objective search," in *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, 2021, pp. 40–41.

[44] "2021 IEEE International Conference On Artificial Intelligence Testing (AITest)," July 2022. [Online]. Available: http://av-test-challenge.org/

[45] H. Ebadi, M. Moghadam, M. Borg, G. Gay, A. Fontes, and K. Socha, "Efficient and effective generation of test cases for pedestrian detection - search-based software testing of baidu apollo in svl," in *2021 IEEE International Conference On Artificial Intelligence Testing (AITest)*. Los Alamitos, CA, USA: IEEE Computer Society, aug 2021, pp. 103–110. [Online]. Available: http://doi.org/10.1109/AITEST52744.2021.00030

[46] J. Seymour, D. Ho, and Q. Luu, "An empirical testing of autonomous vehicle simulator system for urban driving," in *2021 IEEE International Conference On Artificial Intelligence Testing (AITest)*. Los Alamitos, CA, USA: IEEE Computer Society, aug 2021, pp. 111–117. [Online]. Available: http://doi.org/10.1109/AITEST52744.2021.00031

[47] D. Kaufmann, L. Klampfl, F. Kluck, M. Zimmermann, and J. Tao, "Critical and challenging scenario generation based on automatic action behavior sequence optimization: 2021 ieee autonomous driving ai test challenge group 108," in *2021 IEEE International Conference On Artificial Intelligence Testing (AITest)*. Los Alamitos, CA, USA: IEEE Computer Society, aug 2021, pp. 118–127. [Online]. Available: https://doi.org/10.1109/AITEST52744.2021.00032

[48] V. Nguyen, S. Huber, and A. Gambi, "Salvo: Automated generation of diversified tests for self-driving cars from existing maps," in *2021 IEEE International Conference On Artificial Intelligence Testing (AITest)*. Los Alamitos, CA, USA: IEEE Computer Society, aug 2021, pp. 128–135. [Online]. Available: https://doi.org/10.1109/AITEST52744.2021.00033

[49] K. Viswanadha, F. Indaheng, J. Wong, E. Kim, E. Kalvan, Y. Pant, D. J. Fremont, and S. A. Seshia, "Addressing the ieee av test challenge with scenic and verifai," in *2021 IEEE International Conference On Artificial Intelligence Testing (AITest)*. Los Alamitos, CA, USA: IEEE Computer Society, aug 2021, pp. 136–142. [Online]. Available: https://doi.org/10.1109/AITEST52744.2021.00034

[50] "The 14th Intl. Workshop on Search-Based Software Testing," July 2022. [Online]. Available: https://sbst21.github.io/program/

[51] "The 15th Intl. Workshop on Search-Based Software Testing," July 2022. [Online]. Available: https://sbst22.github.io/program/

[52] I. M. L. Rial and J. P. Galeotti, "Evosuitedse at the sbst 2021 tool competition," in *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, 2021, pp. 30–31.

[53] A. Abdullin, M. Akhin, and M. Belyaev, "Kex at the 2021 sbst tool competition," in *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, 2021, pp. 32–33.

[54] 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, pp. 279–291.

[55] L. Li, W.-L. Huang, Y. Liu, N.-N. Zheng, and F.-Y. Wang, "Intelligence testing for autonomous vehicles: A new approach," *IEEE Transactions on Intelligent Vehicles*, vol. 1, no. 2, pp. 158–166, 2016.

[56] "SVL Simulator by LG - Autonomous and Robotics real-time sensor Simulation, LiDAR, Camera simulation for ROS1, ROS2, Autoware, Baidu Apollo. Perception, Planning, Localization, SIL and HIL Simulation, Open Source and Free." [Online]. Available: https://www.svlsimulator.com/

[57] "How to make NPC accelerate smoothly between waypoints? - Issue #1298 - lgsvl/simulator." [Online]. Available: https://github.com/lgsvl/simulator/issues/1298

[58] "Autoware: Open-source software for urban autonomous driving," August 2021. [Online]. Available: https://github.com/CPFL/Autoware

[59] "Vision and mission of autoware," August 2021. [Online]. Available: https://www.autoware.org/visionandmission/

[60] S. Kato, E. Takeuchi, Y. Ishiguro, Y. Ninomiya, K. Takeda, and T. Hamada, "An open approach to autonomous vehicles," *IEEE Micro*, vol. 35, no. 6, pp. 60–68, 2015.

[61] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi, "Autoware on board: Enabling autonomous vehicles with embedded systems," in *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, 2018, pp. 287–296.

[62] A. Corso, P. Du, K. Driggs-Campbell, and M. J. Kochenderfer, "Adaptive stress testing with reward augmentation for autonomous vehicle validatio," in *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, 2019, pp. 163–168.

[63] F. Hauer, A. Pretschner, and B. Holzmüller, "Fitness functions for testing automated and autonomous driving systems," in *International Conference on Computer Safety, Reliability, and Security*, 2019, pp. 69–84.

[64] F. Kluck, M. Zimmermann, F. Wotawa, and M. Nica, "Genetic algorithm-based test parameter optimization for adas system testing," in *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, 2019, pp. 418–425.

[65] H. Zhou, W. Li, Z. Kong, J. Guo, Y. Zhang, B. Yu, L. Zhang, and C. Liu, "Deepbillboard: Systematic physical-world testing of autonomous driving systems," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 347–358.

[66] A. Stocco, M. Weiss, M. Calzana, and P. Tonella, "Misbehaviour prediction for autonomous driving systems," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 359–371.

[67] 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)*. IEEE, 2018, pp. 132–142.

[68] J. Kim, J. Ju, R. Feldt, and S. Yoo, "Reducing dnn labelling cost using surprise adequacy: an industrial case study for autonomous driving," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1466–1476.

[69] Z. Peng, J. Yang, T.-H. Chen, and L. Ma, "A first look at the integration of machine learning models in complex autonomous driving systems: a case study on apollo," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1240–1250.

[70] R. B. Abdessalem, S. Nejati, L. C. Briand, and T. Stifter, "Testing vision-based control systems using learnable evolutionary algorithms," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 1016–1026.

[71] R. Ben Abdessalem, S. Nejati, L. C. Briand, and T. Stifter, "Testing advanced driver assistance systems using multi-objective search and neural networks," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 63–74.

[72] F. U. Haq, D. Shin, and L. Briand, "Efficient online testing for dnn-enabled systems using surrogate-assisted and many-objective optimization," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 811–822.

[73] Z. Zhong, Z. Hu, S. Guo, X. Zhang, Z. Zhong, and B. Ray, "Detecting multi-sensor fusion errors in advanced driver-assistance systems," in *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, S. Ryu and Y. Smaragdakis, Eds. ACM, 2022, pp. 493–505. [Online]. Available: https://doi.org/10.1145/3533767.3534223

[74] C. Erbsmehl, "Simulation of real crashes as a method for estimating the potential benefits of advanced safety technologies," in *21st International Technical Conference on the Enhanced Safety of Vehicles (ESV) National Highway Traffic Safety*, 2009, pp. 09–0162.

[75] 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*, 2019, pp. 257–267.

[76] A. Gambi, M. Muller, 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, ISSTA 2019, Beijing, China, July 15-19, 2019*, D. Zhang and A. Møller, Eds. ACM, 2019, pp. 318–328. [Online]. Available: https://doi.org/10.1145/3293882.3330566

[77] J. Peltomäki, F. Spencer, and I. Porres, "WOGAN at the SBST 2022 CPS Tool Competition," May 2022, arXiv:2205.11064 [cs]. [Online]. Available: http://arxiv.org/abs/2205.11064

[78] C. Birchler, N. Ganz, S. Khatiri, A. Gambi, and S. Panichella, "Cost-effective Simulation-based Test Selection in Self-driving Cars Software with SDC-Scissor," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Honolulu, HI, USA: IEEE, Mar. 2022, pp. 164–168. [Online]. Available: https://ieeexplore.ieee.org/document/9825849/

[79] B. Chen, X. Chen, Q. Wu, and L. Li, "Adversarial evaluation of autonomous vehicles in lane-change scenarios," *IEEE Trans. Intell. Transp. Syst.*, vol. 23, no. 8, pp. 10 333–10 342, 2022. [Online]. Available: https://doi.org/10.1109/TITS.2021.3091477

[80] A. Calò, P. Arcaini, S. Ali, F. Hauer, and F. Ishikawa, "Generating avoidable collision scenarios for testing autonomous driving systems," in *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*. IEEE, 2020, pp. 375–386. [Online]. Available: https://doi.org/10.1109/ICST46399.2020.00045

[81] A. Wachi, "Failure-scenario maker for rule-based agent using multi-agent adversarial reinforcement learning and its application to autonomous driving," *CoRR*, vol. abs/1903.10654, 2019. [Online]. Available: http://arxiv.org/abs/1903.10654