Misconfiguration Software Testing for Failure Emergence in Autonomous Driving Systems

YUNTIANYI CHEN, University of California, Irvine, USA
YUQI HUAI, University of California, Irvine, USA
SHILONG LI, University of California, Irvine, USA
CHANGNAM HONG, University of California, Irvine, USA
JOSHUA GARCIA, University of California, Irvine, USA

The optimization of a system's configuration options is crucial for determining its performance and functionality, particularly in the case of autonomous driving software (ADS) systems because they possess a multitude of such options. Research efforts in the domain of ADS have prioritized the development of automated testing methods to enhance the safety and security of self-driving cars. Presently, search-based approaches are utilized to test ADS systems in a virtual environment, thereby simulating real-world scenarios. However, such approaches rely on optimizing the waypoints of ego cars and obstacles to generate diverse scenarios that trigger violations, and no prior techniques focus on optimizing the ADS from the perspective of configuration. To address this challenge, we present a framework called ConfVE, which is the first automated configuration testing framework for ADSes. ConfVE's design focuses on the emergence of violations through rerunning scenarios generated by different ADS testing approaches under different configurations, leveraging 9 test oracles to enable previous ADS testing approaches to find more types of violations without modifying their designs or implementations and employing a novel technique to identify bug-revealing violations and eliminate duplicate violations. Our evaluation results demonstrate that ConfVE can discover 1,818 unique violations and reduce 74.19% of duplicate violations.

CCS Concepts: • **Software and its engineering → Search-based software engineering**; • **Computer systems organization** → *Embedded and cyber-physical systems.*

Additional Key Words and Phrases: Autonomous driving systems, Software configuration

ACM Reference Format:
Yuntianyi Chen, Yuqi Huai, Shilong Li, Changnam Hong, and Joshua Garcia. 2024. Misconfiguration Software Testing for Failure Emergence in Autonomous Driving Systems. *Proc. ACM Softw. Eng.* 1, FSE, Article 85 (July 2024), 24 pages. https://doi.org/10.1145/3660792

1 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, Tesla, Ford, GM, and Toyota [24, 25, 31, 33, 34]. Quite a few of these companies are already commercially providing AV products running on public roads, with notable examples of the robo-taxi services from Alphabet's Waymo and GM [18, 24] and also millions of Autopilot-equipped Teslas [34]. Experts forecast that AVs will drastically impact society,

Authors' addresses: Yuntianyi Chen, University of California, Irvine, Irvine, USA, yuntianc@uci.edu; Yuqi Huai, University of California, Irvine, Irvine, USA, yhuai@uci.edu; Shilong Li, University of California, Irvine, Irvine, USA, shilonl2@uci.edu; Changnam Hong, University of California, Irvine, Irvine, USA, changnah@uci.edu; Joshua Garcia, University of California, Irvine, Irvine, USA, joshug4@uci.edu.

ACM 2994-970X/2024/7-ART85
https://doi.org/10.1145/3660792

particularly by reducing accidents [41]. However, crashes caused by AVs indicate that achieving this lofty goal remains an open challenge. Despite the fact that companies such as Tesla [29], Waymo [31], or Uber [30] 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 [26]. AVs with lower levels of autonomy have resulted in more fatalities in recent years [4, 7–10, 17, 22, 26]. In October 2021, an AV operated by Pony.ai hit a street sign on a *median*, i.e., the strip of land between the lanes of opposing traffic on a divided highway, in Fremont, California, prompting California to suspend the company's driverless testing permit.

Autonomous driving software (ADS) that operates these AVs are highly configurable systems. More specifically, we have found open-source versions of high-autonomy (Level 4), production-grade ADSes (i.e., Apollo [28] and Autoware [13]) have 1,943 and 2,475 runtime configuration options in configuration files, respectively, resulting in an exponentially large number of configurations to consider in order to assess and optimize an AV (e.g., to minimize the severity and frequency of errors). ADS configurations can have substantial effects on ADS behaviors, such as altering the functionalities (e.g., driving behavior) or non-functional properties (e.g., performance or passenger comfort) of an ADS. The large number of configuration options of an ADS, along with the dearth of documentation explaining them, prevents engineers from tuning ADS configurations for their own custom needs (e.g., to maximize the performance of a particular AV). Compounding this problem, a recent study of Apollo and Autoware revealed that incorrect configurations cause a large amount of ADS bugs (27.25%) and account for many bug symptoms (e.g., crashes) [54]. Thus, determining whether an ADS prevents improper or invalid values from causing the AV to misbehave, which we refer to as *ADS misconfigurations*, is a highly challenging and safety-critical task. Although many approaches aim to test an ADS by generating driving scenarios [53, 58, 59, 65, 67, 78, 79, 86], we have not found any ADS configuration testing approaches in the existing literature. They have all focused on improving test generation to augment the safety and security of AVs under default configurations, ignoring the myriad of available configuration alternatives.

To test ADSes under varying configurations, a testing approach can execute pre-existing driving scenarios in simulation for every ADS configuration. However, as per discussions among ADS developers and contributors, there are practical challenges associated with speeding up simulations, such as potential inaccuracies or oscillating controls [35]. For this reason, executing driving scenarios is expensive for an ADS since scenario execution time is supposed to be equivalent to the time required to test the scenario in the physical world to obtain realistic simulation results. For example, a driving scenario that takes 30 seconds in the physical world still takes 30 seconds in simulation. Scenario re-execution per configuration and the thousands of configuration options to consider make ADS configuration testing practically impossible to conduct exhaustively. To address this combinatorial problem of testing ADS configurations and re-executing driving scenarios, such testing must minimize the time spent on rerunning scenarios for each alternative configuration tested. To that end, it is particularly critical to ensure that generating such configurations (1) prevents the identification of one failure from masking other failures, which we refer to as *failure masking*, and (2) minimizes the identification of duplicate failures. To prevent masking failures in testing ADS configurations, a testing approach can significantly benefit from determining the ranges of configuration values likely to exhibit a failure. Unfortunately, the valid ranges for options are undocumented and, often, may not be handled by an ADS under test. To minimize the identification of duplicate failures in ADS configuration testing, a testing approach must be able to find unique failures that emerge during testing, which we refer to as *emerged failures*.

To overcome the aforementioned challenges, we introduce ConfVE (Configuration Violation Emerger), a novel framework for ADS configuration testing that leverages a genetic algorithm to produce alternative ADS configurations in a manner that reduces the large configuration space to

identify configuration options and values that can lead to the discovery of bugs not identifiable using the ADS' default configuration and, thus, may not be detected until engineers customize the ADS (e.g., for a particular AV or when customizing the ADS for operation in the physical world). ConfVE could help developers to identify optimal configurations, fix problematic ones, and analyze specific configuration ranges and combinations. For example, by narrowing down the valid range for each option according to the testing results, such a range can help users tailor their needs while avoiding system failures. The main contributions of this work are as follows:

- We propose ConfVE, the first configuration testing approach in the ADS domain, which serves as a testing framework that utilizes scenarios from pre-existing ADS scenario-generation techniques and a genetic algorithm to produce alternative configurations to identify emerged failures in an ADS by preventing the masking of failures and maximizing the possibility of producing bug-revealing violations.
- We design 3 novel module-level oracles that detect bug-revealing violations in ADS scenarios that occur frequently in our ADS configuration-testing experiments.
- We introduce a duplicate elimination process to minimize duplicate failure generation and identify emerged failures, which works by checking the similarity of traffic violations using an unsupervised clustering technique and representing those violations as the key features of driving scenarios with respect to each different violation type.
- We evaluate ConfVE on two open-source versions of production-grade ADSes and discovered 1,818 unique violations from 9 violation types.

2 BACKGROUND

2.1 Autonomous Driving Software

An ADS aims to achieve high automation levels for vehicles to automatically run on 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 six levels of autonomous driving from Level 0, with no assistance systems, to Level 5, which represents fully autonomous driving [73]. An ADS is used to achieve high automation (Level 4) or full automation (Level 5). Baidu Apollo [28] and Autoware [13] achieve high automation, specifically, Level 4 [23], which means they are capable of automatically controlling the vehicle in most potential circumstances and performing 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. An ADS is a large software system consisting of different modules with varying functionalities: **HD Map** includes lane geometries and locations of traffic control devices, which may be used by other modules; **Routing** generates high-level navigation information based on routing requests and tells the autonomous vehicle which routes 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; **Prediction** receives the obstacle information including position, velocity, and acceleration detected by Perception and predicts the future motion of the obstacles; **Planning** makes decisions for the autonomous vehicle to execute, such as cruising or stopping.

2.2 Scenario-generation Approaches

State-of-the-art scenario-generation approaches [58, 59, 65, 67, 86] focus on generating scenarios that can expose ADSes to various violations. These approaches initialize the ADS at a location on the map and send the destination location to the ADS so that it can plan to complete its task; in addition,

such testing approaches also manipulate maneuvers of the obstacles so that complex scenarios involving interactions between the ADS and other road traffic participants can be generated. AV-Fuzzer [65] and AutoFuzz [86] use a number of manually specified variables to represent obstacle positions as well as the initial and final location of the ADS, resulting in generating scenarios where the AV always starts at the same location and drives toward the same destination along with similar obstacles across different scenarios. The required manual specification of these approaches limits the diversity of the scenarios, inspiring fully automated approaches (e.g., SCENORITA [58], DoppelTest [59], DeepCollision [67]). All these aforementioned approaches focus on generating scenarios only under the default configuration but not alternative configurations that may trigger violations or bugs.

2.3 Configuration Testing

Existing configuration testing techniques build upon regression testing and can handle a number of configurable options ranging from tens to thousands [56, 83, 84]. Even the state-of-the-art robotics debugging technique, Swarmbug [61], which conducts experiments on 4 Swarm algorithms, focuses on 6-14 configuration variables, which is much smaller than the 1,943 runtime configuration options in Apollo. Swarmbug takes 25.2 (for Adaptive Swarm), 2.8 (for Swarmlab), 0.4 (for Fly-by-logic), and 0.3 (for Howard's) hours to run 100 tests for this scale of configurable systems under known ranges for each option. Unlike traditional configuration testing techniques where a result of a test case can be obtained instantaneously, ADS test cases usually require executing the simulation of vehicle driving scenarios in real-time (e.g., a single test case can take 30 to 60 seconds).

A large number of options in a software system would result in the problem of *combinatorial explosion* for all possible configurations because every test would ideally be rerun for every configuration change. Even rerunning or recompiling the system for every configuration change to take effect incurs significant time costs. Furthermore, virtual scenario-based ADS testing is already highly expensive because simulations are running in real-time (e.g., 30 minutes of simulation testing is equivalent to 30 minutes of testing in the physical world). Therefore, we cannot test all configuration combinations. For virtual testing of ADS, we leverage the insight that we do not need to fully run every module if testing is not intended to be end-to-end. For instance, prior works [58, 59] replace the perception module with ground truth obstacle information to focus testing on the routing, prediction, and planning modules. This can reduce the need to test all 1,943 or 2,475 configuration options in Apollo or Autoware by focusing on Planning or core modules of a Level-4 ADS, such as Routing and Prediction, because they are the most bug-ridden [54].

2.4 Motivating Example

Prior work [54] conducted a comprehensive study of ADS bugs in 2 open-source ADS repositories (i.e., Apollo [28] and Autoware [13]) and discovered incorrect configurations are the root cause of 27.25% of bugs. Incorrect configuration is also responsible for 97.5% of build failures that may prevent the compilation or building of the ADS. Although previous ADS testing approaches combined have a diverse set of oracles that can detect safety, motion sickness, and traffic law violations, none of them considered different configurations of the ADS and only used the default configuration.

As an example, consider the following bug-revealing violation found by CONFVE but not by previous approaches: In the context of trajectory optimization in the planning module of Baidu Apollo, the `accel_penalty` parameter plays a vital role in ensuring smooth and comfortable driving experiences for passengers. Engineers adjust this parameter to strike a balance between comfort and efficiency based on the specific driving scenario. In a scenario where an AV needs to change lanes on a highway to avoid an obstacle, setting the `accel_penalty` configuration option too high may result in Planning generating trajectories with overly gentle acceleration profiles, leading to

slow and conservative driving. This behavior can increase travel time and reduce efficiency, while also posing safety risks if the vehicle is driving too slowly relative to other traffic. By setting the `accel_penalty` from 1.0 to a large value (e.g., 8988.8), the AV runs at an extremely slow speed when changing lanes, resulting in an *unsafe lane-change* violation, i.e., the AV spends an excessive amount of time driving on lane boundaries.

3 SPECIFICATION OF STATE SPACE

To clarify configuration testing and how it differs from scenario-generation approaches, we present a formal specification of the state space in the form of scenarios. ConfVE uses this formal specification of the state space, along with the search operators in the genetic algorithm, to generate configurations for testing scenarios from previous scenario-generation approaches.

Definition 1. A Scenario $S = \langle E, V, D, \mathbb{O} \rangle$ is a tuple where:
- E represents the ego car (i.e., the autonomous vehicle).
- $V = \{v_1, v_2, \cdots, v_{|V|}\}$ is a set of violations occurring in the scenario. Each violation $v \in V$ has a violation type $v.type$ and $V.types = \{v.type \mid v \in V\}$ is the set of all violation types in V.
- $D = \{d_1, d_2, \cdots, d_{|D|}\}$ is a set of planning decisions produced by the Planning module.
- $\mathbb{O}$ is a finite, non-empty set of obstacles (i.e. non-player characters).

Definition 2. A Test Case $TC = \langle \mathbb{S}, C, Oracles \rangle$ is a tuple where:
- $\mathbb{S} = \{S_1, S_2, \cdots, S_k\}$ is a set of k scenarios to be tested.
- $C = \langle o_1, o_2, \cdots, o_{|C|} \rangle$ is a tuple of configuration options representing a configuration, where $o_i \in C$ is a configuration option. We use C_d to denote the default configuration of the ADS (i.e., the set of values for each configuration option selected by developers for the ADS release under test) and S_d to represent a scenario tested under the default configuration.
- $Oracles$ is a finite, non-empty set of ADS oracles that are used to measure various violations that occurred in the scenario.

4 APPROACH

Figure 1 shows an overview of ConfVE, a novel ADS configuration testing framework. The main goal of ConfVE is to test ADS under different driving scenarios with different configurations to expose ADS failures and violations. ConfVE achieves this goal as follows:

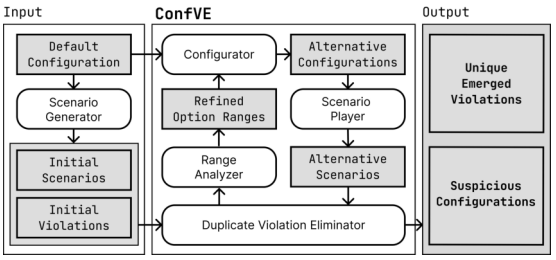

Fig. 1. Overview of ConfVE

Scenario Generator reuses existing ADS scenario-generation approaches to produce test cases, i.e., *initial scenarios* for ConfVE, each of which contains the information necessary to reproduce a scenario. These scenarios are then sorted based on the criteria we discuss in Section 4.1.3, and used as input to ConfVE based on the ordering. *Configurator* analyzes the target configuration file in terms of the option types and uses a genetic algorithm to produce alternative configurations. The genetic algorithm evolves the ADS configurations with the aim of finding configurations that trigger emerged violations, i.e., violations that can be found using an alternative configuration but cannot be found using the default configuration of an ADS. Given an alternative configuration C_a generated by *Configurator*, *Scenario Player* tests scenarios by reproducing them using the routing request, obstacle perception, and traffic signal perception under C_a. *Duplicate Violation Eliminator* evaluates scenarios under alternative configurations and checks the violations arising from each configuration by comparing them with existing violations

to determine if they are emerged violations. For each generation, *Duplicate Violation Eliminator* inspects all accumulated emerged violations to eliminate any duplicate violations, resulting in a set of unique, emerged violations. A dynamic *Range Analyzer* determines value ranges for configuration options to prevent the masking of violations from recurring and reduce duplicate violations.

ConfVE incorporates unique procedures like duplicate elimination among driving scenarios and range analysis, especially concerning options with large potential ranges, e.g., floating-point features, which are common in ADSes and cyber-physical systems but occur less often in configuration systems for other domains. These challenges together exacerbate the need to test configuration options by making testing time longer and the combinatorial search space larger than traditional systems from non-ADS or non-cyber-physical domains. Unlike pre-existing techniques described in Section 7, there is a severe lack of known configuration option ranges to use for testing, requiring a dynamic range analysis like that offered by our approach. We further use domain-specific fitness, such as the score functions of planning decisions and the sinuosity of trajectories, to guide the genetic algorithm in selecting offspring individuals for better alternative configurations.

4.1 Scenario Generator

The objective of ConfVE is to perform comprehensive testing of an ADS across a variety of configurations. *Scenario Generator* serves as a pre-processing step that generates diverse initial scenarios utilizing existing scenario-generation approaches before we start the configuration testing. To prevent redundant testing of the same scenarios across multiple configurations, which tend to produce similar violations, a scenario ranking mechanism has been implemented. This mechanism prioritizes scenarios based on their potential to reveal new or unique violations and selects them for testing in a specific order.

4.1.1 Initial Scenarios Generation. ADS scenario-generation approaches aim to create realistic and effective driving scenarios that expose the ADS to safety and comfort violations [58, 59, 65, 67, 86]. These scenarios typically involve (1) routing requests sent to the ADS, which can be used to extract the initial location of the AV and its destination; (2) obstacle trajectories, which can be used to extract the location, speed, and heading of every obstacle at every timestamp during a scenario; and (3) traffic signal status, which is used to indicate right-of-way status. These approaches use a genetic algorithm to maximize a defined set of fitness functions (representing safety and comfort violations) to guide the search for problematic scenarios that are likely to trigger violations.

Different ADS testing techniques differ in the ability of the ego car to handle different running conditions in terms of the size of maps and the complexity and diversity of scenarios such as obstacle number, obstacle types, and whether the obstacle runs with constant or mutable speed [43, 44, 53, 58, 59, 64, 65, 67, 71, 86]. ConfVE uses different scenario-generation techniques under the default configuration to generate initial scenarios. By analyzing the scenario record file, ConfVE extracts the scenario setup (e.g., perception of obstacles, routing request, module configurations, etc.), uses the extracted information as input, and reproduces the scenario under different configurations to determine how configuration changes influence the ego car and the ADS that operates it to identify bug-revealing violations or other types of bugs (e.g., ADS module crashes).

4.1.2 ADS Oracles. Previous work considers a limited number of test oracles, mainly consisting of one or only a few test oracles (e.g., less than 5) per work [38, 39, 43, 53, 58–60, 65, 86]. The limited use of test oracles found in such techniques ignores important safety and comfort issues (e.g., driving between lanes for too long or causing system failures) and provides significantly less insight into the testing of production-grade ADSes. Unlike previous work, ConfVE considers 9 test oracles, 6 of which are based on grading metrics defined by Apollo's developers [3].

We adopt the following 6 scenario-level oracle types from previous work:

- *Collision* [3] oracle has been defined and used by previous test generation approaches to detect collisions between the AV and other road traffic participants [58, 59, 65, 86]. We apply a bug-revealing checking mechanism to filter out false positives for which an AV is not responsible. Collision violations are safety-critical and can lead directly to severe injuries or loss of life.
- Comfort oracles focus on whether the AV accelerates excessively, i.e., *Fast Acceleration* [3], or decelerates too fast, i.e., *Hard Braking* [3]. An acceleration (or deceleration) is excessive if it exceeds a maximum allowed value of $4m/s^2$, which is a threshold set by Apollo developers and used in prior research [40, 58]. These (de)acceleration violations often cause motion sickness [80], which affects about one-third of the population [36], especially women [55], who are historically underrepresented in healthcare research [69] and technology design [72].
- *Speeding* [3] oracle detects whether the AV is traveling at a speed higher than legally permitted in a given lane. The most common dangers caused by speeding include loss of control as a driver, rollover accidents, and higher severity of crashes [20].
- *Unsafe Lane-change* [58] oracle detects violations in which the AV spends an extended period of time driving on lane boundaries—which may lead to traffic congestion, traffic delays, road rage incidents, or collisions [14, 15, 48].
- *Lane-change in Junction* [3] oracle focuses on traffic law violations in which the AV attempts to change lanes in a junction. This violation might cause dangers such as collisions and road rage incidents [11, 48, 75].

Through experimentation and manual analysis of the ADS under test, we further define 3 novel module-level oracles that aim to detect system failures and invalid configurations:

- A *Module Delay* oracle aims to detect cases in which certain modules are producing output at a lower-than-usual frequency, which makes the AV fail to respond to a decision. For example, if the Control module delays responding for more than two seconds, a collision might occur if an obstacle is in front of the AV at high speed.
- A *Module Malfunction* oracle detects scenarios in which an ADS module fails to be initialized and launched. For example, the Planning module depends on the Routing module in order to make decisions to operate the AV. Failure to initialize either Routing or Planning causes the AV to freeze at its location, making the AV unable to reach its destination or cause a traffic jam if the AV is freezing at a junction. This oracle covers all engaged modules in ADS virtual testing and is a generalization of the Routing test oracle in Apollo Dreamland [3], which is a web-based simulation platform maintained by Apollo. The Pony.ai incident [21], due to module malfunction or delay, led to the system's shutdown and subsequently caused California to suspend its driverless testing permit.
- A *Vehicle Paralysis* oracle detects scenarios in which a module is correctly initialized but is producing output that differs from its expected behavior. For example, Planning is expected to produce planning decisions after Routing has determined a trajectory leading to the AV's destination; however, during experimentation, we observed Planning may produce empty messages while a valid input to Planning is provided. This oracle would also cause the AV to freeze at its location. However, the module is correctly initialized but not running correctly in *Vehicle Paralysis*, unlike the module that is not successfully initialized in *Module Malfunction*.

The module oracles are crucial for detecting crashes and system-level bugs, which is a significant advancement from previous oracles that were primarily focused on driving functionality bugs. These new oracles are particularly relevant in configuration testing, as in our experiments, only configuration-related bugs can trigger specific failures. Although these module-level oracles focus on system states and differ from prior work that mainly checks for driving behavior violations, symptoms of these bugs are still observable during simulation.

Our oracles further aim to account for whether a violation reveals a bug. For all violation types except collision, a violation instance is bug-revealing because only the ego car is involved in these violations, making it always responsible for the violation. For collision violations, we must carefully design a collision oracle to maximize its ability to detect bug-revealing collisions. Specifically, side and rear-end collisions involving the vehicle are highly unlikely to be the fault of the ego car. If the ego car is hit from the back or its side, the responsibility likely lies with the obstacle, making the associated collision not bug-revealing. For example, if an obstacle initiates a lane change maneuver without considering the right of way of other road traffic participants and collides into the AV from the side, the obstacle will be determined as responsible if it has not completed the lane change. As a result, when a collision violation occurs, ConfVE only considers such violations when the collision occurs in front of the ego car as it moves with non-zero velocity by measuring the locations and headings of both the ego car and the obstacle.

4.1.3 Ranking Initial Scenarios. Although ConfVE may obtain numerous scenarios from different ADS scenario-generation approaches, the computational and time expense of executing and simulating the selected scenarios under different configurations prevents ConfVE from re-executing all scenarios generated by an ADS scenario-generation approach. As a result, we need a mechanism to select diverse scenarios that contain different AV running conditions and decide which scenarios should be tested first.

A test case in our configuration testing contains k scenarios with an alternative configuration. We select k scenarios from all initially generated scenarios based on the diversity ranking of scenarios $Ranking_{\mathbb{S}_{init}}(score_{S.V}, score_{S.D}, score_{S.sinuosity})$, where $score_{S.V}$ is the violation rarity, $score_{S.D}$ is the number of planning decisions, and $score_{S.sinuosity}$ is the sinuosity (i.e., curvature or bending) of planning routes. $Ranking_{\mathbb{S}_{init}}$ uses Non-Dominated Sorting [52] to rank initial scenarios based on their violation rarity, the number of planning decisions, and sinuosity of planning routes. Non-Dominated Sorting is a technique used in multi-objective optimization to deal with multiple conflicting objectives, which helps identify Pareto-optimal scenarios by classifying them into different levels of non-dominance. This ensures that the selected scenarios are diverse and representative, covering a wide range of trade-offs among the three criteria.

The intuition of the $score_{S.V}$ ranking scheme is to assign different weights for each test oracle so that a scenario with rarer violations is more likely to be selected compared to a scenario with violations that appear in many other scenarios. More formally,

$$score_{S.V} = \sum_{vt \in S.V.types} W_{vt} * |S.V_{vt}| \tag{1}$$

$$W_{vt} = 1 - \frac{|\mathbb{S}^{vt}_{init}|}{|\mathbb{S}_{init}|} \tag{2}$$

$$S.V_{vt} = \{v \in S.V \mid v.type = vt\} \tag{3}$$

$$\mathbb{S}^{vt}_{init} = \{S \in \mathbb{S}_{init} \mid vt \in S.V.types\} \tag{4}$$

W_{vt} is the rarity weight of a violation type in $\mathbb{S}_{init}$, $S.V_{vt}$ is the set of violations of type vt in scenario S, $\mathbb{S}_{init}$ is the set of all initial scenarios, $\mathbb{S}^{vt}_{init}$ is the set of scenarios containing the violation type vt in $\mathbb{S}_{init}$, and $S.V.types$ is the set of violation types in a scenario.

While scenarios with violations can be compared using the rarity of violations, those that do not have any violations are harder to compare. Instead of only focusing on the outcome of the scenario, we leverage the decisions that the ADS makes during a scenario to evaluate the complexity of the scenario. The intuition is that the more unique decisions the ADS makes (e.g., yield, overtake, stop, etc.), the more complex the scenario is. Hence, the decision ranking scheme is defined as $score_{S.D} = |\{d \in S.D\}|$.

An additional feature that we use for determining whether a scenario is more interesting than another is the sinuosity of the path traversed by the AV. Sinuosity is represented as the ratio of the curvilinear length and the Euclidean distance between the endpoints of the path traversed. The ratio is exactly 1.0 if the AV traverses on a straight line. The intuition behind this feature is that scenarios in which the AV made complex maneuvers (e.g., overtaking an obstacle, turning at junctions) are more interesting than ones only involving AV traversing on a straight line. The sinuosity score is defined as $score_{S.sinuosity} = \frac{dist_{traveled}(E)}{displacement(E)}$, where $dist_{traveled}(E)$ is the total length of the path traversed by the ego car E and $displacement(E)$ is the distance between the initial and final location of E (i.e., straight-line distance) in a scenario.

Through our experiments, we find that constantly testing configurations in fixed initial scenarios tend to produce similar emerged violations. To address this challenge, a scenario substitution mechanism is introduced to change the tested scenarios by replacing initial scenarios if the emerged violations are detected in them using an alternative configuration. More specifically, as the genetic algorithm executes, when ConfVE finds an emerged violation in a scenario, it will replace such a scenario with another one from initially generated scenarios based on the ranking of diversity score (i.e., $Ranking_{\mathbb{S}_{init}}$) at the end of a generation.

4.2 Configurator

Configurator manages the generation of alternative configurations in the genetic algorithm. *Configuration Parser* parses configuration files of the ADS, extracting options, and analyzing their types so that appropriate mutation operators can be applied to individual configuration options. *Configurator* then applies the genetic algorithm and its associated mutation operators to different types of options and initializes the ADS under the alternative configuration.

4.2.1 Configuration Parser. A configuration file under test needs to be analyzed before the configuration testing. *Configuration Parser* identifies options using regular expression and infers the option type and potential range for each option, which would be used in the mutation process of the genetic algorithm. For example, if the type of the default value of an option is floating-point, *Configuration Parser* produces a large range of floating-point values it can take on. From experiments, we found assigning an exceedingly large value generally would cause module violations. However, we do not expect emerged violations to always be these types, which is against one of ConfVE's objectives, i.e., to trigger more types of violations. Since permissible values for an ADS configuration option are seldom specified or documented [42], we also need a way to narrow down the range for options, which we discuss in Section 4.4.

4.2.2 Representation. Figure 2 illustrates the genetic representation of an individual produced by ConfVE. An individual (i.e., chromosome) represents a configuration C, which, in turn, represents a single test. A test suite in ConfVE is a set of test cases that contain different alternative configurations and the same set of scenarios. An individual is represented by a sequence of genes, each corresponding to a configuration option o. When initializing an individual, all genes are assigned their default values. Each gene can change its value through mutation, but it still has to adhere to the ranges determined by *Configuration Parser* and *Range Analyzer*.

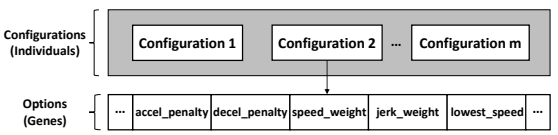

Fig. 2. Genetic Representation of a Configuration

4.2.3 Fitness Evaluation. In each generation, ConfVE evaluates individuals by their fitness with respect to multiple search objectives and determines which individuals should be selected to pass on their genes. ConfVE determines the fitness of an individual by evaluating the diversity and

number of emerged violations and planning decisions. This is measured by calculating an individual i's fitness using a function $F(i) = \left(f_{count}(i), f_{type}(i), score_D(i), score_{sinuosity}(i)\right)$, where $f_{count}(i)$ counts the total number of emerged violations within a scenario; $f_{type}(i)$ refers to the number of emerged violation types; $score_D(i)$ is a function that computes the decision score, $score_{S.D}$; and $score_{sinuosity}(i)$ is a function that computes the sinuosity of planning routes $score_{S.sinuosity}$ for a scenario S produced with configuration i.

4.2.4 Search Operators. In every iteration of the genetic algorithm, ConfVE runs the ADS in the simulation environment and tests scenarios under the configuration. Based on the testing results of this generation, ConfVE updates the configurations to be tested in the next generation. Alternative configurations are generated by applying the mutation and crossover to the configuration individuals of the current generation. *Configurator* aims to focus on four objectives: (1) the number of emerged violations obtained from configuration testing, (2) the number of emerged violation types, (3) the number of planning decisions, and (4) the complexity of the planning route.

Mutation. ConfVE uses a single-point mutation strategy by applying one of the mutation operators shown in Table 1 to a gene from an individual. The mutation operators differ by option type, covering all standard and appropriate operators that could be applied to ADS configuration options. For example, options of numerical types can be mutated by applying the digit type changing or randomly generating a value within the range; options of string type can be mutated by following the mutation operators in state-of-the-art approaches of misconfiguration injection testing [66, 82, 85].

Table 1. Mutation Operators for Different Option Types, and Examples of Option Values Before and After Mutation

Option Type	Mutation Operator	Before	After
Integer	Generate Value	5	10
	Digit type change	5	6.5
Float	Generate Value	15.70	30.42
	Digit type change	15.70	15
E-Number	Generate Value	4e7	4e5
Boolean	Negation	true/yes/max	false/no/min
String	Substitute	"aa/bb"	"aa/cb"
	Add	"aa/bb"	"aa/bcb"
	Delete	"aa/bb"	"aabb"
	Cut	"aa/bb"	"aa"
	Case Conversion	AABB	aabb
	Disorder	AABB	BBAA
	Repeat	AABB	AABBAABB

Multiple mutations can lead to failures, which our approach supports by incrementally mutating configuration options through its single-point mutation. By mutating a single configuration option at a time, our approach enables effective tracking of option tuning, facilitates range analysis, and allows multiple mutations to a single individual. Note that our approach focuses on identifying potential problematic configurations and is not a root-cause analysis or fault-localization approach. Considering the time-consuming nature of scenario testing in the ADS domain, our choice of a single-point mutation strategy makes the relationship between an emerged violation and a tuned option relatively explicit and balances the trade-off between individual diversity and the difficulty of tracking the latest option tuning. If more than one option is tuned for a mutation, it is relatively difficult to judge which option causes a violation because, as the number of simultaneously mutated options increases, the number of required evaluations grows exponentially. For instance, consider a scenario where two options are mutated simultaneously, and a violation emerges. To determine the root cause, it would be necessary to test both options individually and in combination, resulting in four potential evaluations.

Crossover. This operator selects two individuals and creates offspring by mixing the genetic makeup of their parents. ConfVE uses a commonly used single-point crossover strategy, where the crossover point is picked randomly from the mating individuals (i.e., parents), and the genes behind the point are swapped. Figure 3 illustrates the application of the crossover operator on two sample individuals.

Selection. Certain individuals can trigger module violations (e.g., *module malfunction*) and cause the ADS to freeze at its initial location. Once a module violation occurs, tuning other options may

result in the same module violation that is likely triggered by an option mutated in the previous generation as opposed to a newly mutated option in the current generation. We refer to this phenomenon as *violation masking*, i.e., a module violation masks the effect of tuning other options and impairs the effectiveness of the optimization algorithm. ConfVE will not select individuals that are likely to cause violation masking.

ConfVE uses the **N**on-dominated **S**orting **G**enetic **A**lgorithm selection (NSGA-II) for breeding the next generation [47]. NSGA-II is an effective algorithm used for solving multi-objective optimization problems (i.e., problems with multiple conflicting fitness functions) and further aims to maintain the diversity of individuals. NSGA-II starts by sorting a set of individuals based on a *non-dominated* order of fitness. 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., the number of planning decisions), 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 and offspring are selected to produce offspring based on the fitness and crowding distance; an individual is selected if its order rank of fitness is less than the other, or if the crowding distance is greater than the other. Only the best N_{pop} individuals are selected, where N_{pop} 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, which maintains the diversity of driving scenarios in ConfVE; and (3) it emphasizes the non-dominated solutions.

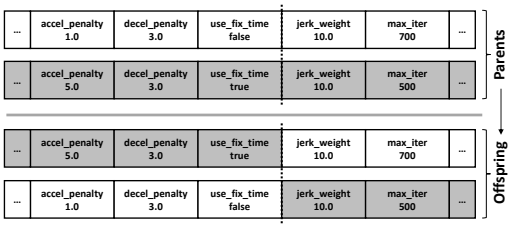

Fig. 3. An Example of a Crossover

4.3 Duplicate Violation Eliminator

Duplicate Violation Eliminator (DVE) identifies violations arising from rerunning scenarios using alternative configurations in *Scenario Player* and filters out duplicate violations, retaining only those that are unique. ConfVE integrates DVE at two distinct stages of its operation. In the first stage, ConfVE assesses the uniqueness of violations that manifest in specific scenarios by rerunning these scenarios under alternative configurations and contrasting the results with those obtained under the default configuration. Violations that are identified as unique under these new configurations are designated as *emerged violations*. As ConfVE progresses, *Duplicate Violation Eliminator* evaluates the violations from these alternative configurations in terms of their emergence and their potential to reveal bugs. In each iteration, emerged violations are aggregated. Notably, configurations that give rise to these emerged violations are classified as *suspicious configurations*. These configurations hold significance as they can be instrumental for range analysis, as referenced in Section 4.4, or even for software debugging purposes. In the second stage, ConfVE employs DVE to examine the accumulated *emerged violations* for duplicates, eliminating them and producing one of ConfVE's final outputs, i.e., unique, emerged violations. This stage is crucial because different configurations could produce the same violation, even for the same scenario.

To distinguish ego behaviors, especially different violations committed by the ego car in a scenario, recent work proposed clustering-based approaches [45, 58], that leverage a distance-based metric to determine the similarity between ego behaviors across two distinct scenarios. For ConfVE, we reused and augmented the duplicate elimination approach in sceneRITA [58], including a set

of general representations of violations and the clustering algorithm, for mitigating the presence of duplicate violations since SCENORITA focuses more on violations while the other approach focuses on ego behaviors. Our approach extends beyond this by introducing *non-strict* and *strict features. Strict features* are domain-specific and any difference between two violations immediately indicates those violations are different while *non-strict features* are used to determine violation similarity in cases where strict features are equal. This dual classification balances adaptability and specificity, ensuring robustness across varying real-world conditions. We further expanded the violation representations to include our newly introduced violation types (i.e., *Module Delay*, *Module Malfunction*, *Vehicle Paralysis*, and *Lane-change in Junction*) according to the principles of applicability across ADSes and the minimum involved components of a violation, which were confirmed by two Apollo contributors. Previous scenario-generation approaches have introduced a feature representation for violations in virtual driving scenarios [58, 59]. To compare violations, CONFVE represents them using the features shown in Table 2. These features represent the key characteristics of violations per type, which we determined by studying each violation type, the information recorded by an ADS, and confirmation from Apollo contributors. For a *collision* violation, the *Duplicate Violation Eliminator* reuses 4 features that are extracted at time t, where t indicates the first timestamp at which the violation occurs. These features include the position p_t^E of the ego car E at time t; E's speed s_t^E at time t; the position p_t^O of obstacle O; obstacle O's speed s_t^O at collision time t. We also add two features h_t^E or h_t^O, which denote the heading of the ego car or obstacle, to represent scenario violations more accurately. For the remaining violations, we extract their respective features. These features include the ego car E's location at a violation time p_t^E, and the speed s_t^E of E. For *speeding*, *unsafe lane-change*, *fast acceleration*, *hard braking*, and *module delay* violations, we also measure the length of time for which it lasts ($duration$) while *fast acceleration* and *hard braking* oracles measure the acceleration value at a violation time ($accel/decel$).

While the aforementioned features can effectively distinguish violations detected in prior work, for module-related violations that are detected in CONFVE, those features cannot correctly determine the uniqueness of a violation. For example, the AV may freeze at the same position with the same heading due to *module malfunction* from 2 different modules. To address this problem, we also use the type of an ADS module (e.g., Routing, Planning, Prediction, or Localization), denoted by $type_{module}$, or a junction's unique identifier, denoted by $id_{junction}$, as *strict features*, which are features that cause any two violations to be identified as different, irrespective of other feature values, if such a feature's values differ for those two violations. For example, if the type of a module $type_{module}$ differs for two violation instances of a *module malfunction*, then the two instances must be different since the malfunction occurs in different modules.

Table 2. Feature Representations of Each Violation Type

Violation Type	Non-strict Features	Strict Features
Collision	$\{p_t^E, s_t^E, h_t^E, p_t^O, s_t^O, h_t^O\}$	–
Fast Acceleration	$\{p_t^E, s_t^E, h_t^E, duration, accel\}$	–
Hard Braking	$\{p_t^E, s_t^E, h_t^E, duration, decel\}$	–
Speeding	$\{p_t^E, s_t^E, h_t^E, duration\}$	–
Unsafe Lane-change	$\{p_t^E, s_t^E, h_t^E, duration\}$	–
Lane-change in Junction	$\{p_t^E, s_t^E, h_t^E\}$	$id_{junction}$
Module Delay	$\{p_t^E, s_t^E, h_t^E, duration\}$	$type_{module}$
Module Malfunction	$\{p_t^E, s_t^E, h_t^E\}$	$type_{module}$
Vehicle Paralysis	$\{p_t^E, s_t^E, h_t^E\}$	$type_{module}$

As a concrete example of duplicate violations, consider the three collisions with partial features shown in Table 3. When comparing Collision-1 and Collision-2, these incidents occurred 5.2572 meters apart from each other, the headings of the AVs differ by -4.0060 degrees, and the headings of the obstacles differ by 0.0001 degrees. These two collisions are considered duplicate collision violations, while Collision-3 is considered to be different from both Collision-1 and Collision-2 given it occurred at a place farther away (181.3857m) and the heading of both the AV and obstacle differ significantly (195.8593 and 237.9755 degrees, respectively). Identifying such duplicates is important

in configuration testing since we need to distinguish misconfiguration-induced violations from ones in the original scenario.

Table 3. Example of Duplicate Collision Violations

ID	ego location x	ego location y	ego heading	obstacle heading	ego speed	obstacle speed
Collision-1	559449.716550803	4157214.07281456	-2.44831086142857	-2.42155592419325	40.1479655443566	68.5231404988578
Collision-2	559445.666180909	4157210.72136067	-2.47056667699013	-2.42155667074278	39.7015265216219	68.5231404988578
Collision-3	559264.568996154	4157220.94727521	-1.38245939298376	-1.09947073870517	29.7268322456541	59.1636072973156

Using the features we defined, ConfVE identifies emerged violations by (1) determining if a violation in an alternative configuration is sufficiently different from one in the default configuration and (2) addressing the inherent non-determinism of an ADS. Due to the inherent non-determinism, reproducing default scenarios may yield slight variations, potentially leading to slightly different violation results for the same scenario. To mitigate this, each initial scenario S_d is rerun under the default configuration C_d 10 times to collect all violations (V_{C_d}) arising from C_d. When ConfVE produces an alternative configuration C_a, ConfVE runs each scenario S_j using C_a to find all violations V_{C_a} arising from C_a. To support diverse and potentially fluctuating scenarios, we employ a clustering algorithm to accommodate variations in scenarios and violations. ConfVE determines if a violation $v^j_{C_a} \in V_{C_a}$ is emergent if (1) the violation type of $v^j_{C_a}$ does not exist in V_{C_d} or (2) the violation type of $v^j_{C_a}$ is in $V_{C_d}.types$ and the similarity between $v^j_{c_a}$ and a violation of this type in V_{C_d} are sufficiently low. For example, if v_{C_a} is assigned to a cluster with more than one violation in it, ConfVE considers v_{C_a} to be sufficiently similar to at least one violation in V_d. In such a case, ConfVE does not consider v_{C_a} an emerged violation. If v_{C_a} is a singleton cluster, ConfVE considers v_{C_a} as an emerged violation since it is an outlier that is sufficiently different from any violation in V_d. For the clustering itself, we chose DBSCAN [50] (i.e., density-based spatial clustering of applications with noise), which was used in scenoRITA [58], since it is distance-based and more suited for spatial data.

4.4 Range Analyzer

From our comprehensive analysis, we observed that program paths of ADS configurations typically do not impose limitations on the value ranges they can assume, indicating that ADSes often bypass range checking for configuration options. After checking the source code of the Apollo planning module in terms of the configuration file, we found only 15 options out of 197 numeric (i.e., integer, floating-point, and Euler's number types) options have constraints or bounds that could be used to infer initial option ranges. However, these initial ranges are somewhat imprecise and broad (e.g., `dense_dimension_s` > 1 or `sparse_unit_s` $\neq 0$), requiring further refinement through dynamic analysis. This lack of range checking by ADS code, combined with a large number of configurations and unknown valid ranges, poses a significant risk of generating unsupported configurations, thus leading to intrinsic crashes, errors, or exceptions within the ADS—especially when it is customized to run on a particular physical AV.

To determine an effective range of values for testing ADS configuration options, ConfVE applies a range analysis to options that cause *violation masking*, i.e., when a violation prevents other likely bug-revealing violations from emerging. From our investigation of ADS configurations, we find that the types of emerged violations are highly related to the values of mutated options and that a value outside of a configuration option's valid range, which is highly unlikely to be documented, tends to mask other failures that occur in an ADS, creating an undesirable increase in time spent rerunning scenarios for each alternative configuration. A common example of this masking phenomenon we have observed is the possibility of an AV freezing at its initial position due to an invalid option value, thus hindering the occurrence of violations such as speeding, fast

acceleration, and hard braking. This masking effect leads to inefficient utilization of time due to the generation of repetitive violations. Through our experimentation and observation, we find that invalid configurations always cause intrinsic crashes, errors, or exceptions that prohibit the modules from normally launching or running. More specifically, we find all types of failures tend to be masked by previously identified module failures (i.e., module failures found by the initial scenarios of a scenario-generation approach or by earlier executions of ConfVE). As an example, once a module failure occurs in the same scenario, changing a configuration option's value to an invalid range (e.g., one that is not even possible in the physical world) would trigger highly similar violations (e.g., the vehicle would always fail to start and stop in the same lane).

To overcome failure masking, we leverage the insights that (1) module violations are often caused by invalid values and (2) the default value is consistently within the valid range to test for option values that are likely to prevent *failure masking*. More specifically, given an option with a range of $[o_{lower}, o_{upper}]$, which can be arbitrarily large or small, and default value o_d, if ConfVE produces a value o_f, we use the value to update the range of values that the option can take on in future executions of ConfVE's genetic algorithm. More specifically, the range for the option is subsequently truncated to $[o_{lower}, o_f]$ if $(o_d < o_f)$ or $[o_f, o_{upper}]$ if $(o_d > o_f)$. In cases where a tested value cannot cause module violations, the range remains unaltered. As a result, the algorithm tends to reduce the initial large range of values, i.e., $[o_{lower}, o_{upper}]$, closer to the default value, which is a value that is unlikely to cause an error. This property of *Range Analyzer* balances between obtaining values closer to a likely valid value (i.e., the default value) while starting with a wide range of values that are likely to be invalid, resulting in the prevention of *failure masking* and the emergence of violations or failures likely to occur, which our evaluation will demonstrate.

5 EVALUATION

In order to empirically evaluate ConfVE, and to understand how configurations affect the scenarios and violations, we investigate the following research questions:

- **RQ1**: How effective is ConfVE at exposing unique emerged violations?
- **RQ2**: How effective is ConfVE at finding unique emerged violations?
- **RQ3**: To what extent are duplicate violations eliminated by ConfVE?
- **RQ4**: What is the runtime efficiency of ConfVE?

5.1 Experimental Setup

We evaluated ConfVE by executing 124,950 virtual tests for a total of 990 hours on Baidu Apollo 7.0 [28] and Autoware v1.0 [19], which are both open-source versions of production-grade or near production-grade ADSes. Although Apollo and Autoware have co-existed for several years, evaluations of prior work [58, 59, 65, 78, 79, 86] were predominantly conducted only on Apollo. To the best of our knowledge, we are the first to evaluate an ADS testing approach on Autoware v1.0. We conducted our experiments on four machines: 2 machines each with 2 AMD EPYC 7551 32-Core Processors (512GB RAM), 2 machines each with 1 Core i9 16-Core Processor (96GB RAM), running Ubuntu 22.04.

For Apollo, we evaluated ConfVE on four real-world HD Maps located in California, including three provided as part of Baidu Apollo [28] and one from the simulator LGSVL [74]. **Sunnyvale Loop** is a large map consisting of 3,061 lanes, with a total length of 107 km; **San Mateo** is a medium map consisting of 1,305 lanes, with a total length of 24 km; **San Francisco** is another medium map consisting of 1,524 lanes, with a total length of 109 km; and **Borregas Ave** at Sunnyvale with 60 lanes and a total length of 3 km. The four maps consist of various types of road curvature (e.g., straight, curved, intersections) and different types of lanes (e.g., highways, city roads, bike lanes, etc.). We selected *planning_config.pb.txt* as the target configuration file to test the planning module.

This file consists of 18 integer options, 167 floating-point options, 29 Boolean options, 51 string options, and 12 options containing Euler's number. We ran 4 scenario-generation approaches to generate initial scenarios, which are used as input for *Scenario Player* in ConfVE. ConfVE uses Apollo's simulation feature, `Sim-Control`, to simulate driving scenarios.

We used different state-of-the-art scenario-generation approaches for ConfVE whose code and dependencies are available for Apollo. To date, various scenario-generation approaches have been proposed and evaluated on an open-source ADS. AV-Fuzzer [65] uses the state-of-the-art simulator LGSVL [74] and applies a genetic algorithm to generate scenarios in which other vehicles may perform various actions such as cutting in during a scenario, testing an ADS' capability of reacting to those vehicles. AutoFuzz [86] uses the same simulator as AV-Fuzzer but applies a neural network-guided fuzzing algorithm to generate scenarios using the simulator's API, which is provided to configure the virtual environment. scenoRITA [58] does not rely on using a specific simulator and, therefore, is not limited by the types of obstacles that the simulator provides to test an ADS. Furthermore, scenoRITA analyzes high-definition maps (HD Maps) so trajectories of the AV and obstacles are automatically generated instead of manually specified. DoppelTest [59] is similar to scenoRITA but uses the ADS to model every vehicle in a scenario as opposed to only a single vehicle. Such a setting guarantees at least one AV is responsible for any violation occurring. *Scenario Generator* uses the default settings and supported maps of these approaches.

Due to ADS scenario-generation approaches [58, 59, 65, 67, 86] being predominantly based on and implemented for Apollo, migrating tools and re-implementing existing scenario-generation approaches from Apollo to Autoware requires overcoming challenges such as map transformation, interface and automation implementations, and the connection between the ADS and simulators. Considering the lack of available scenario-generation approaches implemented for Autoware, we choose to use scenarios provided by the Autoware Evaluator [37], which is an official Autoware Foundation platform that collects datasets and test suites that focus on the Operational Design Domain (ODD) [46], which specifies the operating conditions under which an ADS can operate safely, to test Autoware on different scenarios to enhance safety and optimize functionality. We tested the *behavior velocity planner* of Autoware, which consists of 5 integer options, 181 floating-point options, 55 Boolean options, and 4 string options. We evaluated the approach on three HD Map groups, **LEO-VM**, which are virtual maps created by the AV company Leo Drive; **AWF CICD**, which are HD maps developed by the Autoware Foundation; and HD maps of roads in **Taiwan**.

ConfVE evolved populations of 20 configurations per generation with a maximum scenario duration of 30 seconds for Apollo and 60 seconds for Autoware, which is similar to prior work [58, 59]. In this research, we selected 10 initial scenarios as test cases to start the configuration testing because it balances the ability to find emerged bug-revealing violations and the time it takes to run all scenarios for one alternative configuration. To enhance the speed of ConfVE, we launched five Docker containers to run configuration testing simultaneously. We choose to compare with the baseline approaches, pairwise testing, which is regarded as an efficient and intuitive testing and sampling strategy for highly configurable systems [70, 77], and ConfVD [66], which is a misconfiguration testing approach that utilizes fine-grained constraints of option type classification. To support ADS experiments, we integrate pairwise testing and ConfVD into our configuration testing framework as a different option tuning strategy for *Configurator*. We carefully implemented a version of ConfVD for Apollo and Autoware, as no implementation is publicly available, and we make our implementation of ConfVD publicly available (Section 9). We set a predefined time budget of 10 hours for each experimental group, i.e., either ConfVE or baseline approach conducted on a particular HD Map using the initial scenarios from a scenario-generation approach. We ran each experimental group three times and calculated the averages after rounding to integers as testing results to reduce the influence of bias and non-determinism.

5.2 RQ1: Independent Effectiveness

For RQ1, we evaluated ConfVE's ability to produce unique, emerged violations on different maps and ADS testing approaches. From Table 4, the experiments conducted in this study involved the deployment of scenoRITA and DoppelTest techniques across three distinct maps, while AutoFuzz and AV-Fuzzer approaches were only employed in a single map, due to their manual setup preventing them from being run on other maps. The results obtained from the experiments were organized by violation type and grouped based on the respective ADS testing techniques employed. Notably, the study identified the highest number of violations within the same violation oracle and marked those results in grey, which was highlighted as the most frequent across all groups.

Table 4. ConfVE (CE), Pairwise (Pa.), and ConfVD (CD) in terms of Unique Emerged Violations by Apollo

Violation Type	scenoRITA									DoppelTest									AutoFuzz			AV-Fuzzer		
	Borregas Ave			San Mateo			Sunnyvale			Borregas Ave			San Mateo			Sunnyvale			Borregas Ave			San Francisco		
	CE	Pa.	CD	CE	Pa.	CD	CE	Pa.	CD	CE	Pa.	CD	CE	Pa.	CD	CE	Pa.	CD	CE	Pa.	CD	CE	Pa.	CD
Collision	**1**	0	**1**	0	0	0	0	0	0	0	0	0	**2**	**2**	1	0	0	0	**1**	**1**	0	**1**	**1**	0
Fast Accel.	2	2	**3**	**2**	1	1	**8**	4	1	0	**1**	**1**	**4**	1	1	**8**	5	3	0	0	0	**4**	**4**	1
Hard Brak.	**49**	25	22	**17**	16	5	**101**	35	27	**16**	12	13	**21**	10	13	**31**	26	18	**4**	3	3	**20**	11	9
Speeding	0	0	0	**98**	56	53	**62**	43	39	**16**	12	5	**18**	12	9	**16**	3	3	**13**	7	4	0	0	0
Unsafe LC	**78**	47	30	**52**	44	24	**42**	32	23	**22**	13	7	**23**	8	16	**108**	20	9	**13**	5	9	0	0	0
LC in Junc.	0	0	0	**38**	31	18	**35**	30	25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Delay	**12**	8	3	5	**10**	0	**6**	3	1	5	**9**	6	**2**	**2**	1	**19**	13	7	0	**1**	**1**	**30**	26	6
Malfunc.	53	**56**	34	49	**66**	44	49	**63**	42	31	**33**	22	**30**	29	29	**54**	52	26	**17**	**17**	15	30	**33**	20
Paralysis	27	**30**	19	**21**	14	12	26	**36**	23	24	**26**	19	21	**25**	18	23	**32**	18	**14**	11	13	**18**	17	16
Total	**222**	168	112	**282**	238	157	**329**	246	181	**114**	106	73	**121**	89	88	**259**	151	84	**62**	45	45	**103**	92	52
Improv. (%)	-	32.14	98.21	-	18.49	79.62	-	33.74	81.77	-	7.55	56.16	-	35.96	37.5	-	71.52	208.33	-	37.78	37.78	-	11.96	98.08

As shown in Table 4, Apollo scenarios from scenoRITA, when used with ConfVE, produce more unique, emerged violations (222-329) and more violation types (i.e., the only approach that produces all 9 violation types) than other scenario-generation approaches. DoppelTest, which guarantees any generated collision is bug-revealing by making all vehicles AVs in a scenario, produces 114-259 unique, emerged violations from 8 violation types when used as input to ConfVE. While scenoRITA and DoppelTest produce a wide variety of diverse violations, AutoFuzz and AV-Fuzzer, when used with ConfVE, produce significantly fewer violations and violation types: AutoFuzz only produced 62 violations from 7 violation types; AV-Fuzzer only detected 103 violations from 6 violation types.

We found that for each scenario-generation approach, the large map Sunnyvale Loop has more complex road conditions, and produces more unique, emerged violations than medium or small maps. The greatest number of unique, emerged violations among the three maps was recorded on Sunnyvale Loop, with scenoRITA and DoppelTest generating 329 and 259 violations, respectively.

The bug-revealing checking mechanism of ConfVE identified 196 false positives out of 278 *collision* violations, which means about 71.79% of *collision* violations are filtered out by bug-revealing checking. The amount of *collision* violations is relatively smaller than other violation types because *collision* violations are more dependent on input scenarios. For example, in some initial scenarios, the ego car cannot encounter obstacles throughout the entire process. For Autoware, as shown in Table 5, 8 violation types are detected except *Speeding*, which indicates Autoware's implementation of speed controlling is less likely to violate speed limits on given HD maps.

Finding 1: ConfVE can produce a wide variety of unique, emerged violations (46-329) for all scenario-generation approaches and maps, demonstrating its ability to identify such violations irrespective of a particular scenario-generation approach and across two ADSes.

5.3 RQ2: Comparative Effectiveness

RQ2 aims to compare the number of unique, emerged violations discovered by ConfVE with that of the baseline approaches, i.e., pairwise testing and ConfVD, by setting a predefined timeout (i.e,

10 hours) with a goal of finding emerged violations within a time budget. Table 4 shows the total number of unique, emerged violations discovered by ConfVE and the baseline approach for each scenario-generation approach. The approach that generates more unique, emerged violations than the others on the same map is bolded. ConfVE demonstrated superior performance compared to pairwise testing and ConfVD, achieving an overall improvement of 28.03% and 67.80% in Apollo, respectively. For example, it yields improvements of 27.76%, 42.77%, 37.78%, and 11.96% over pairwise testing in the scenoRITA, DoppelTest, AutoFuzz, and AV-Fuzzer scenarios, respectively. ConfVE also works better on larger maps than the baseline while the testing result in of DoppelTest on Sunnyvale Loop has the highest improvement of 71.52% than the pairwise testing and 208.33% than ConfVD. For Autoware, ConfVE achieves overall improvements of 10.14% and 7.24% than pairwise and ConfVD, respectively, which are significantly less than those in Apollo.

We manually inspected and analyzed some scenarios generated by each of the testing approaches to get an insight as to why ConfVE's performance varies. First, ConfVE is optimized for considering complex road conditions by setting a scenario diversity as objectives in the fitness function. It also reduces occurrences of *violation masking* by dynamic *range analysis* to reduce duplicate module violations. Besides, the complexity and diversity of initial scenarios generated by ADS testing approaches vary. DoppelTest generates the most sophisticated scenarios as it uses the ADS to model every vehicle in the simulation, making the scenario more complex because vehicles are reacting to each other. scenoRITA models automatically generate a considerable number of obstacles and model them as constant speed obstacles, having the highest number of obstacles across all approaches. While DoppelTest and scenoRITA automatically analyze the map and generate different scenarios in terms of the initial and final location of the AV, AutoFuzz and AV-Fuzzer always generate similar scenarios that start at the same position and finish at the same position.

Table 5. ConfVE (CE), Pairwise (Pa.), and ConfVD (CD) in terms of Unique Emerged Violations in Autoware

Violation Type	Autoware Evaluator								
	LEO-VM			AWF CICD			Taiwan		
	CE	Pa.	CD	CE	Pa.	CD	CE	Pa.	CD
Collision	2	3	**4**	**10**	5	4	**2**	1	**2**
Fast Accel.	**2**	1	0	**9**	5	8	0	0	0
Hard Brak.	**9**	6	5	**106**	77	91	**6**	3	5
Speeding	0	0	0	0	0	0	0	0	0
Unsafe LC	**22**	20	**22**	44	44	**52**	11	5	**13**
LC in Junc.	0	0	0	**17**	15	15	0	0	0
Delay	**11**	10	7	3	**5**	3	2	**4**	1
Malfunc.	**29**	23	25	14	**33**	24	19	**23**	16
Paralysis	1	1	**2**	1	1	**2**	6	**7**	3
Total	**76**	64	65	**204**	185	199	**46**	43	40
Improv. (%)	-	18.75	16.92	-	10.27	2.51	-	6.98	15.0

Finding 2: ConfVE managed to generate 27.40%, and 65.88% more unique, emerged violations compared to the pairwise testing and ConfVD within the same time budget. ConfVE performs better in complex and diverse scenarios and produces more violation types than the baselines.

5.4 RQ3: Duplicate Violation Elimination

In RQ3, we study the extent to which ConfVE eliminates similar violations and compare the percentage of duplicate violations generated by 3 configuration testing approaches. To answer this RQ, we use DBSCAN [50] to cluster the scenarios with similar violations into the same group, based on a set of features as described in Table 2.

Table 6 shows all emerged violations (including duplicates) generated by Apollo and Autoware along with the number of unique violations (generated by *Duplicate Violations Eliminator*) and the percentage of eliminated violations. From the results, we observe that ConfVE has a lower elimination ratio (74.19%) than (81.45%) of pairwise testing overall, while ConfVE found fewer emerged violations in total but produced more unique, emerged violations, which indicates that ConfVE is more efficient at finding diverse violations. Although ConfVD has fewer emerged

violations in total and less elimination ratio, the number of unique emerged violations is significantly fewer than ConfVE and pairwise testing.

We also found that *module malfunction* is the most frequent violation type to occur for both ConfVE and pairwise testing. Furthermore, pairwise testing has the highest duplicate elimination ratio, i.e., 90.13% with 4,338 total violations, suggesting that ConfVE outperforms pairwise testing, which spent an excessive amount of time testing *module malfunction* violations, likely due to it lacking a dynamic range analysis to narrow down valid option ranges like that found in ConfVE.

Table 6. Results of Duplication Violation Elimination

Violation Type	ConfVE			Pairwise			ConfVD		
	All	Uniq.	Elim.	All	Uniq.	Elim.	All	Uniq.	Elim.
Collision	29	19	34.48%	29	13	55.17%	24	12	50.00%
Fast Accel.	100	39	61.00%	44	24	45.45%	34	19	44.12%
Hard Brak.	1482	380	74.36%	741	224	69.77%	734	211	71.25%
Speeding	842	223	73.52%	380	133	65.00%	311	113	63.67%
Unsafe LC	1391	415	70.17%	634	238	62.46%	592	205	65.37%
LC in Junc.	314	90	71.34%	225	76	66.22%	162	58	64.20%
Delay	703	95	86.49%	504	91	81.94%	110	36	67.27%
Malfunc.	1743	375	78.49%	4338	428	90.13%	1322	297	77.53%
Paralysis	441	182	58.73%	796	200	74.87%	385	145	62.34%
Total	7045	1818	74.19%	7691	1427	81.45%	3674	1096	70.17%

Finding 3: The *Duplicate Violation Eliminator* eliminated 74.19% duplicate tests in ConfVE, 81.45% in pairwise testing, and 70.17% in ConfVD. Nevertheless, ConfVE had 27.40% more unique, emerged violations (1,818) than pairwise testing (1,427) and 65.88% more unique, emerged violations than ConfVD (1,096).

5.5 RQ4: Runtime Efficiency of ConfVE

To investigate the runtime efficiency of ConfVE, we measure the execution time of scenarios for each combination of a scenario-generation approach and HD Map. *Scenario Player* plays and records every scenario for the same amount of time (i.e., 30 seconds), which previous work [58, 59] has shown to effectively balance the time allocated to find bugs without spending an excessive amount of time executing tests. As a result, every scenario has the same execution time, making *Scenario Player*'s difference in execution time across scenario-generation approaches negligible. At the same time, *Configurator* takes 0.1 seconds or less to execute, on average, making the execution time negligible. Consequently, the major differences in time efficiency arise from *Scenario Generator* and *Duplicate Violation Eliminator*. *Scenario Generator* employs scenario-generation approaches to produce initial scenarios while the *Duplicate Violation Eliminator* measures violations through 9 test oracles and eliminates duplicates. Note that ConfVE and pairwise testing incur the same time spent executing the *Scenario Generator* because it needs to be only executed once and, thereafter, is used to provide initial scenarios as input before ConfVE or pairwise testing runs.

Table 7 shows the average time to execute the *Scenario Generator* and *Duplicate Violation Eliminator* for a scenario. As an example, for scenoRITA, scenario generation takes 34.07 seconds and duplicate violation checking takes 0.86 seconds. The results for Table 7 indicate that the size of the map is correlated with the measurement time of violations. *Duplicate Violation Eliminator* takes 5.25 or 11.95 seconds to analyze one scenario from scenoRITA or DoppelTest in Sunnyvale Loop, which is considerably longer compared with a scenario from Borregas Ave. For Autoware, the scenario generation time ranges from 57.79 to 60.67 seconds and the *Duplicate Violation Eliminator* time ranges from 3.90 to 5.48 seconds.

Table 7. Runtime Efficiency of ConfVE Per Scenario (*Scenario Generator* + *Duplicate Violation Eliminator*)

HD Maps	Execution Time (sec.)				
	scenoRITA	DoppelTest	AutoFuzz	AV-Fuzzer	AutoEva.
Borregas Ave	(34.07+0.86)	(45.55+0.74)	(40.86+0.32)	-	-
San Mateo	(33.33+3.23)	(48.36+3.00)	-	-	-
Sunnyvale Loop	(35.54+5.25)	(48.94+11.95)	-	-	-
San Francisco	-	-	-	(53.86+0.61)	-
LEO-VM	-	-	-	-	(60.67+3.90)
AWF CICD	-	-	-	-	(58.02+5.48)
Taiwan	-	-	-	-	(57.79+4.50)

We also compared different scenario-generation approaches in terms of time efficiency of generating initial scenarios for the same map, Borregas Ave. DoppelTest takes the longest time to generate a scenario, which is consistent with the fact that only DoppelTest considers traffic light signals and manipulates intelligent obstacles thus generating the most complicated scenarios.

Finding 4: Overall, ConfVE adds an additional 0.32 seconds to 11.95 seconds per scenario depending on the utilized scenario-generation approach and HD Map, which is small compared to the time it takes for a scenario-generation approach to produce a scenario (i.e., 33.33 seconds to 60.67 seconds), making the runtime overhead of ConfVE relatively small.

6 THREATS TO VALIDITY

Internal Threats. One potential threat to internal validity is that testing results under specific configurations are not always deterministic. To reduce the non-determinism, we rerun the initial scenarios 10 times to collect the initial violations for checking of emerged violations and rerun ConfVE and pairwise testing on each combination of a scenario-generation approach and HD Map 3 times to get averages for evaluation results. We select the rerunning times mainly for a trade-off between the time budget and the accuracy of results. Another threat arises from our large-scale experiment requiring multiple machines. To mitigate this threat, we selected machines with similar hardware specifications.

External Threats. One external threat is our evaluation of ConfVE on limited ADSes. This threat is mitigated by Apollo and Autoware being high autonomy (i.e., Level 4), and open-source versions of the production-grade AV software systems. Apollo is selected by Udacity to teach state-of-the-art AV technology [27] and can be directly deployed on real-world AVs such as Lincoln MKZ, Lexus RX 450h, and others [6, 16], and has mass production agreements with Volvo and Ford [32] while the Autoware Foundation's membership comprises several industrial entities [12] .

Construct Validity. A threat to construct validity is how we evaluate different violations. To mitigate this threat, we measure these violations using grading metrics defined by Apollo's developers [3]. We utilize thresholds (e.g., speeding or acceleration thresholds) set by Apollo's developers [3], the U.S. Department of Transportation [51], or major AV companies [5].

7 RELATED WORK

ADS Testing Approaches. A variety of ADS testing approaches focus on generating driving scenarios [58, 59, 63, 65, 67, 78, 79, 81, 86], ADS test selection and prioritization [49, 68] and also evaluate in an open source ADS, i.e. Baidu Apollo [28]. Besides the scenario-generation approaches described in Section 5.1, Lu et al. proposed DeepCollision [67], which leverages deep reinforcement learning to configure the simulator using its APIs to construct virtual environments and focuses on generating scenarios that involve collisions. Tian et al. proposed MOSAT [78] and CRISCO [79] that abstract the movements of road traffic participants into a diverse set of maneuvers and find combinations of maneuvers to create scenarios that are more complex than the ones generated by AV-Fuzzer. These two approaches have not been made available and therefore cannot be used as part of ConfVE. Deng et al. proposed STRaP [49], which segments the original ADS scenario to effectively reduce the length of the scenarios but maintain high fault coverage. Lu et al. proposed SPECTRE [68], which extracts attributes from scenarios (e.g., collisions and collision probability) and applies multi-objective evolutionary algorithms to select and prioritize test scenarios.

Unlike previous approaches, ConfVE focuses on testing ADS under different configurations to discover emerged violations. ConfVE also applies a more diverse set of oracles that can detect module failure, safety, and comfort violations.

(Mis)configuration Testing. A traditional way to test configurations is to use a sampling-based testing strategy. Medeiros et al. presented a comparative study [70] between 10 sampling algorithms for finding configuration-related faults. The study showed *t-wise* sampling algorithms detected 92% of configuration-related faults on a corpus with existing faults, with *pair-wise* sampling algorithm being one of the most efficient algorithms. Jung et al. proposed Swarmbug [61], which aims to diagnose the root cause and fix bugs when a robotic system is misconfigured. Sun et al. proposed Ctest [76], which aims to detect failure-inducing configuration changes to prevent production failure, but instruments the software system, which can violate real-time properties of an ADS. Keller et al. proposed ConfERR [62], which models human errors when generating misconfigurations (e.g., typo). Xu et al. proposed SPEX [82], which infers configuration requirements from source code so that *misconfiguration vulnerabilities* (i.e., bad system reactions, such as crashes and hangs) can be exposed. SPEX only generates values out of valid ranges. However, in the ADS testing, some values within a valid range can also lead to AV misbehaviors. Zhang et al. proposed ConfDiagDetector [85], which aims to detect improper diagnostic output produced by the system under misconfiguration. Li et al. proposed ConfVD [66], which utilizes fine-grained constraints of option type classification, so a more diverse set of misconfigurations can be generated.

These approaches require some knowledge about option ranges or techniques to infer possible ranges before testing, which is time-consuming and performs poorly in the ADS domain. They have two key deficiencies: (1) not identifying which options should be changed to reduce search time (e.g., through a genetic algorithm), and (2) not identifying values of configuration options in ADSes that are likely to exhibit failure since ADSes typically do not include this information about valid/invalid option ranges, which are likely to be real-valued/floating-point and undocumented, and pre-existing work ignores by assuming these values can be obtained a priori. Unlike these techniques, ConfVE employs a runtime range analysis to identify option ranges to save testing time and is well-suited for the ADS domain's large potential space of real-valued/floating-point ranges while none of these techniques are designed for the ADS domain.

8 CONCLUSION

We propose ConfVE, the first automated configuration testing framework in the ADS testing domain, which exposes ADS to 3 types of safety-critical, 3 types of motion sickness-inducing, and 3 types of inner module violations in a manner that reduces duplicate violations. We evaluate ConfVE on Baidu Apollo, a high autonomy (Level 4), open-source version of a production-grade ADS that supports a wide variety of driving scenarios, and Autoware, an open-source version of a near production-grade ADS whose foundation's membership comprises several industrial entities, such as Intel, Hitachi, LG, and Xilinx. We compare ConfVE with pairwise testing, which is known to work highly effectively for highly configurable software systems, and ConfVD, a state-of-the-art misconfiguration testing approach. We further compare different state-of-the-art scenario-generation approaches for ADSes under different HD Maps in terms of violation emergence. ConfVE found a total of 1,818 unique, emerged violations and reduced 74.19% of duplicate violations with *module malfunction*, *unsafe lane-change*, and *hard braking* violations occurring most. Moreover, ConfVE generated, on average, 27.40% and 65.88% more unique, emerged violations for different ADS scenario-generation approaches in total compared to pairwise testing and ConfVD within the same time budget. In the future, we aim to (1) use configuration testing knowledge to aid cause analysis and bug localization of ADS bugs, and (2) apply more exotic learning-based approaches, such as surrogate models [57], to support increasingly complex scenarios.

9 DATA AVAILABILITY

The source code of our approach is available at [1] while video recordings are available at [2].

REFERENCES

[1] April 2024. Source Code and Data of ConfVE. https://doi.org/10.5281/zenodo.11406940
[2] April 2024. Video Recordings of ConfVE. https://doi.org/10.5281/zenodo.11051748
[3] August 2021. Dreamland's Grading System. https://bit.ly/3nfe48e
[4] August 2021. Google's Self-Driving Car Caused Its First Accident. https://bit.ly/3acQwgO
[5] August 2021. Look, no hands! Test driving a Google car. https://bit.ly/3kWXPAm
[6] August 2021. Open Vehicles Compatible with Apollo. https://apollo.auto/vehicle/certificate_en.html
[7] August 2021. Self-Driving Tesla Was Involved in Fatal Crash, U.S. Says. https://nyti.ms/3abv3Vq
[8] August 2021. Tesla: Autopilot was on during deadly Mountain View crash. https://bayareane.ws/3U1p4av
[9] August 2021. Tesla driver dies in first fatal crash while using autopilot mode. https://bit.ly/3nlNavo
[10] August 2021. There are some scary similarities between Tesla's deadly crashes linked to Autopilot. https://bit.ly/3Zppqsz
[11] August 2023. 5 Traffic Laws You Might Not Realize You Broke. https://bit.ly/45V7vy7
[12] August 2023. Autoware Foundation Members. https://autoware.org/about/members/
[13] August 2023. Autoware: Open-source software for urban autonomous driving. https://bit.ly/3xPVmuH
[14] August 2023. Contributing Factors To Aggressive Driving. https://bit.ly/48giaF7
[15] August 2023. What You Need to Know About Unsafe Lane Changes. https://bit.ly/3PoMViF
[16] December 2022. Baidu Apollo: An open autonomous driving platform. http://apollo.auto/
[17] February 2018. Two Years On, A Father Is Still Fighting Tesla Over Autopilot And His Son's Fatal Crash. https://bit.ly/42OFMxS
[18] February 2022. Waymo to begin charging for robotaxi rides in San Francisco. https://tcrn.ch/3lUBoNd
[19] February 2024. Autoware release/v1.0. https://github.com/autowarefoundation/autoware/tree/release/v1.0
[20] Feburary 2023. Dangers of Speeding: Facts and Problems | Wilson Kehoe Winingham. bit.ly/3JW0Fzv
[21] July 2022. U.S. agency to review if Pony.ai complied with crash reporting order. https://tinyurl.com/42twsa2b
[22] June 2018. Tesla Autopilot System Found Probably at Fault in 2018 Crash. https://nyti.ms/3qXuioY
[23] June 2021. Baidu is building Level 4 autonomous robotaxis called Apollo Moon in China. https://bit.ly/3HQ46qH
[24] June 2022. GM's Cruise starts charging fares for driverless rides in San Francisco. https://reut.rs/40rtGJu
[25] June 2022. Toyota Follows Tesla In Developing A Vision-Based Self-Driving System. https://bit.ly/3TQQNuw
[26] March 2018. Self-Driving Uber Car Kills Pedestrian in Arizona, Where Robots Roam. https://nyti.ms/40IHWxd
[27] March 2022. Self-Driving Fundamentals: Featuring Apollo | Udacity. https://bit.ly/3RnMVC0
[28] March 2023. Apollo: An open autonomous driving platform. https://bit.ly/3XSQUa5.
[29] March 2023. Tesla Autopilot. https://www.tesla.com/autopilot
[30] March 2023. Uber Advanced Technolgoy Group. https://ubr.to/40tYsBA
[31] March 2023. Waymo. https://waymo.com/
[32] November 2018. Baidu hits the gas on autonomous vehicles with Volvo and Ford deals. https://tcrn.ch/3EL8Prj
[33] October 2020. Ford unveils new self-driving test vehicle for 2022 launch. https://cnb.cx/3zi2AIQ
[34] October 2021. Tesla Sold 2 Million Electric Cars: First Automaker To Reach Milestone. https://bit.ly/3G1zDVm
[35] September 2023. Inquiry about time cost of simulation · autowarefoundation · Discussion #3853 · GitHub. https://bit.ly/432LLiU
[36] September 2023. Motion sickness: MedlinePlus Genetics. https://bit.ly/3sYuoCe.
[37] September 2023. TIER IV Autoware Evaluator. https://bit.ly/49SKfSX
[38] Raja Ben Abdessalem, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. 2016. 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, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 63–74. https://doi.org/10.1145/2970276.2970311
[39] Raja Ben Abdessalem, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. 2018. Testing vision-based control systems using learnable evolutionary algorithms. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 1016–1026. https://doi.org/10.1145/3180155.3180160
[40] Hanna Bellem, Barbara Thiel, Michael Schrauf, and Josef F Krems. 2018. Comfort in automated driving: An analysis of preferences for different automated driving styles and their dependence on personality traits. *Transportation research part F: traffic psychology and behaviour* 55 (2018), 90–100.
[41] Michele Bertoncello and Dominik Wee. 2015. Ten ways autonomous driving could redefine the automotive world. *McKinsey & Company* 6 (2015).
[42] Ranjita Bhagwan, Sonu Mehta, Arjun Radhakrishna, and Sahil Garg. 2021. Learning Patterns in Configuration. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 817–828. https://doi.org/10.1109/ASE51524.2021.9678525

[43] Alessandro Calò, Paolo Arcaini, Shaukat Ali, Florian Hauer, and Fuyuki Ishikawa. 2020. 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, 375–386. https://doi.org/10.1109/ICST46399.2020.00045

[44] Ezequiel Castellano, Ahmet Cetinkaya, Cédric Ho Thanh, Stefan Klikovits, Xiaoyi Zhang, and Paolo Arcaini. 2021. Frenetic at the SBST 2021 Tool Competition. In *14th IEEE/ACM International Workshop on Search-Based Software Testing, SBST 2021, Madrid, Spain, May 31, 2021*. IEEE, 36–37. https://doi.org/10.1109/SBST52555.2021.00016

[45] Mingfei Cheng, Yuan Zhou, and Xiaofei Xie. 2023. BehAVExplor: Behavior Diversity Guided Testing for Autonomous Driving Systems. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 488–500. https://doi.org/10.1145/3597926.3598072

[46] Krzysztof Czarnecki. 2018. Operational design domain for automated driving systems. *Taxonomy of Basic Terms ", Waterloo Intelligent Systems Engineering (WISE) Lab, University of Waterloo, Canada* (2018).

[47] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. 2000. A Fast Elitist Non-dominated Sorting Genetic Algorithm for Multi-objective Optimization: NSGA-II. In *Parallel Problem Solving from Nature PPSN VI*, Marc Schoenauer, Kalyanmoy Deb, Günther Rudolph, Xin Yao, Evelyne Lutton, Juan Julian Merelo, and Hans-Paul Schwefel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 849–858.

[48] Jerry L Deffenbacher, David M Deffenbacher, Rebekah S Lynch, and Tracy L Richards. 2003. Anger, aggression, and risky behavior: a comparison of high and low anger drivers. *Behaviour research and therapy* 41, 6 (2003), 701–718.

[49] Yao Deng, Xi Zheng, Mengshi Zhang, Guannan Lou, and Tianyi Zhang. 2022. Scenario-Based Test Reduction and Prioritization for Multi-Module Autonomous Driving Systems. http://arxiv.org/abs/2209.01546 arXiv:2209.01546 [cs].

[50] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96), Portland, Oregon, USA*, Evangelos Simoudis, Jiawei Han, and Usama M. Fayyad (Eds.). AAAI Press, 226–231. http://www.aaai.org/Library/KDD/1996/kdd96-037.php

[51] Federal Highway Administration, US Department of Transportation. June 2009. *Analysis of Lane-Change Crashes and Near-Crashes*. https://bit.ly/3WUnLtO

[52] Félix-Antoine Fortin, Simon Grenier, and Marc Parizeau. 2013. Generalizing the improved run-time complexity algorithm for non-dominated sorting. In *Genetic and Evolutionary Computation Conference, GECCO '13, Amsterdam, The Netherlands, July 6-10, 2013*, Christian Blum and Enrique Alba (Eds.). ACM, 615–622. https://doi.org/10.1145/2463372.2463454

[53] Alessio Gambi, Marc Müller, and Gordon Fraser. 2019. 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*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 27–30. https://doi.org/10.1109/ICSE-Companion.2019.00030

[54] Joshua Garcia, Yang Feng, Junjie Shen, Sumaya Almanee, Yuan Xia, and Qi Alfred Chen. 2020. A comprehensive study of autonomous vehicle bugs. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 385–396. https://doi.org/10.1145/3377811.3380397

[55] J.F. Golding. 2016. Chapter 27 - Motion sickness. In *Neuro-Otology*, Joseph M. Furman and Thomas Lempert (Eds.). Handbook of Clinical Neurology, Vol. 137. Elsevier, 371–390. https://doi.org/10.1016/B978-0-444-63437-5.00027-3

[56] Xue Han and Tingting Yu. 2016. An Empirical Study on Performance Bugs for Highly Configurable Software Systems. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2016, Ciudad Real, Spain, September 8-9, 2016*. ACM, 23:1–23:10. https://doi.org/10.1145/2961111.2962602

[57] Fitash Ul Haq, Donghwan Shin, and Lionel C. Briand. 2022. Efficient Online Testing for DNN-Enabled Systems using Surrogate-Assisted and Many-Objective Optimization. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 811–822. https://doi.org/10.1145/3510003.3510188

[58] Yuqi Huai, Sumaya Almanee, Yuntianyi Chen, Xiafa Wu, Qi Alfred Chen, and Joshua Garcia. 2023. scenoRITA: Generating Diverse, Fully-Mutable, Test Scenarios for Autonomous Vehicle Planning. *IEEE Transactions on Software Engineering* (2023), 1–21. https://doi.org/10.1109/TSE.2023.3309610

[59] Yuqi Huai, Yuntianyi Chen, Sumaya Almanee, Tuan Ngo, Xiang Liao, Ziwen Wan, Qi Alfred Chen, and Joshua Garcia. 2023. Doppelgänger Test Generation for Revealing Bugs in Autonomous Driving Software. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2591–2603. https://doi.org/10.1109/ICSE48619.2023.00216

[60] Tri Huynh, Alessio Gambi, and Gordon Fraser. 2019. AC3R: automatically reconstructing car crashes from police reports. In *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 31–34. https://doi.org/10.1109/ICSE-Companion.2019.00031

[61] Chijung Jung, Ali Ahad, Jinho Jung, Sebastian G. Elbaum, and Yonghwi Kwon. 2021. Swarmbug: debugging configuration bugs in swarm robotics. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 868–880. https://doi.org/10.1145/3468264.3468601

[62] Lorenzo Keller, Prasang Upadhyaya, and George Candea. 2008. ConfErr: A tool for assessing resilience to human configuration errors. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. IEEE, Anchorage, AK, 157–166. https://doi.org/10.1109/DSN.2008.4630084

[63] Seulbae Kim, Major Liu, Junghwan John Rhee, Yuseok Jeon, Yonghwi Kwon, and Chung Hwan Kim. 2022. DriveFuzz: Discovering Autonomous Driving Bugs through Driving Quality-Guided Fuzzing. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM, 1753–1767. https://doi.org/10.1145/3548606.3560558

[64] Florian Klück, Lorenz Klampfl, and Franz Wotawa. 2021. GABezier at the SBST 2021 Tool Competition. In *14th IEEE/ACM International Workshop on Search-Based Software Testing, SBST 2021, Madrid, Spain, May 31, 2021*. IEEE, 38–39. https://doi.org/10.1109/SBST52555.2021.00017

[65] Guanpeng Li, Yiran Li, Saurabh Jha, Timothy Tsai, Michael B. Sullivan, Siva Kumar Sastry Hari, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. 2020. 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*, Marco Vieira, Henrique Madeira, Nuno Antunes, and Zheng Zheng (Eds.). IEEE, 25–36. https://doi.org/10.1109/ISSRE5003.2020.00012

[66] Shanshan Li, Wang Li, Xiangke Liao, Shaoliang Peng, Shulin Zhou, Zhouyang Jia, and Teng Wang. 2018. ConfVD: System Reactions Analysis and Evaluation Through Misconfiguration Injection. *IEEE Trans. Reliab.* 67, 4 (2018), 1393–1405. https://doi.org/10.1109/TR.2018.2865962

[67] Chengjie Lu, Yize Shi, Huihui Zhang, Man Zhang, Tiexin Wang, Tao Yue, and Shaukat Ali. 2023. Learning Configurations of Operating Environment of Autonomous Vehicles to Maximize their Collisions. *IEEE Transactions on Software Engineering* 49, 1 (Jan. 2023), 384–402. https://doi.org/10.1109/TSE.2022.3150788 Conference Name: IEEE Transactions on Software Engineering.

[68] Chengjie Lu, Huihui Zhang, Tao Yue, and Shaukat Ali. 2021. Search-Based Selection and Prioritization of Test Scenarios for Autonomous Driving Systems. In *Search-Based Software Engineering*, Una-May O'Reilly and Xavier Devroey (Eds.). Vol. 12914. Springer International Publishing, Cham, 41–55. https://doi.org/10.1007/978-3-030-88106-1_4 Series Title: Lecture Notes in Computer Science.

[69] Anna C Mastroianni, Ruth R Faden, and Daniel D Federman. 1994. Women and health research. (1994).

[70] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A comparison of 10 sampling algorithms for configurable systems. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 643–654. https://doi.org/10.1145/2884781.2884793

[71] Mahshid Helali Moghadam, Markus Borg, and Seyed Jalaleddin Mousavirad. 2021. Deeper at the SBST 2021 Tool Competition: ADAS Testing Using Multi-Objective Search. In *14th IEEE/ACM International Workshop on Search-Based Software Testing, SBST 2021, Madrid, Spain, May 31, 2021*. IEEE, 40–41. https://doi.org/10.1109/SBST52555.2021.00018

[72] Caroline Criado Perez. 2019. *Invisible women: Data bias in a world designed for men*. Abrams.

[73] Christina Rödel, Susanne Stadler, Alexander Meschtscherjakov, and Manfred Tscheligi. 2014. 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*, Linda Ng Boyle, Gary E. Burnett, Peter Fröhlich, Shamsi T. Iqbal, Erika Miller, and Yuqing Wu (Eds.). ACM, 11:1–11:8. https://doi.org/10.1145/2667317.2667330

[74] Guodong Rong, Byung Hyun Shin, Hadi Tabatabaee, Qiang Lu, Steve Lemke, Martins Mozeiko, Eric Boise, Geehoon Uhm, Mark Gerow, Shalin Mehta, Eugene Agafonov, Tae Hyung Kim, Eric Sterner, Keunhae Ushiroda, Michael Reyes, Dmitry Zelenkovsky, and Seonman Kim. 2020. LGSVL Simulator: A High Fidelity Simulator for Autonomous Driving. In *23rd IEEE International Conference on Intelligent Transportation Systems, ITSC 2020, Rhodes, Greece, September 20-23, 2020*. IEEE, 1–6. https://doi.org/10.1109/ITSC45102.2020.9294422

[75] Mohamed Shawky. 2020. Factors affecting lane change crashes. *IATSS Research* 44, 2 (2020), 155–161. https://doi.org/10.1016/j.iatssr.2019.12.002

[76] Xudong Sun, Runxiang Cheng, Jianyan Chen, Elaine Ang, Owolabi Legunsen, and Tianyin Xu. 2020. Testing Configuration Changes in Context to Prevent Production Failures. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 735–751. https://www.usenix.org/conference/osdi20/presentation/sun

[77] Kuo-Chung Tai and Yu Lei. 2002. A Test Generation Strategy for Pairwise Testing. *IEEE Trans. Software Eng.* 28, 1 (2002), 109–111. https://doi.org/10.1109/32.979992

[78] Haoxiang Tian, Yan Jiang, Guoquan Wu, Jiren Yan, Jun Wei, Wei Chen, Shuo Li, and Dan Ye. 2022. MOSAT: Finding Safety Violations of Autonomous Driving Systems using Multi-objective Genetic Algorithm. (2022), 13.

[79] Haoxiang Tian, Guoquan Wu, Jiren Yan, Yan Jiang, Jun Wei, Wei Chen, Shuo Li, and Dan Ye. 2022. Generating Critical Test Scenarios for Autonomous Driving Systems via Influential Behavior Patterns. (2022), 13.

[80] Mark Turner and Michael J Griffin. 1999. Motion sickness in public road transport: the effect of driver, route and vehicle. *Ergonomics* 42, 12 (1999), 1646–1664.

[81] Riley Underwood, Quang-Hung Luu, and Huai Liu. 2023. A Metamorphic Testing Framework and Toolkit for Modular Automated Driving Systems. In *8th IEEE/ACM International Workshop on Metamorphic Testing, MET@ICSE 2023, Melbourne, Australia, May 14, 2023*. IEEE, 17–24. https://doi.org/10.1109/MET59151.2023.00010

[82] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. 2013. Do not blame users for misconfigurations. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 244–259. https://doi.org/10.1145/2517349.2522727

[83] Sai Zhang and Michael D. Ernst. 2013. Automated diagnosis of software configuration errors. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 312–321. https://doi.org/10.1109/ICSE.2013.6606577

[84] Sai Zhang and Michael D. Ernst. 2014. Which configuration option should I change?. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 152–163. https://doi.org/10.1145/2568225.2568251

[85] Sai Zhang and Michael D. Ernst. 2015. Proactive detection of inadequate diagnostic messages for software configuration errors. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, Michal Young and Tao Xie (Eds.). ACM, 12–23. https://doi.org/10.1145/2771783.2771817

[86] Ziyuan Zhong, Gail E. Kaiser, and Baishakhi Ray. 2021. Neural Network Guided Evolutionary Fuzzing for Finding Traffic Violations of Autonomous Vehicles. *CoRR* abs/2109.06126 (2021). arXiv:2109.06126 https://arxiv.org/abs/2109.06126

Received 2023-09-28; accepted 2024-04-16