Prunario: Testing Autonomous Driving Systems by Pruning Likely Redundant Scenarios

MINSU KIM, Korea University, Republic of Korea
SUNBEOM SO*, Korea University, Republic of Korea
HAKJOO OH*, Korea University, Republic of Korea

We present Prunario, a novel technique for effectively testing autonomous driving systems (ADS). Ensuring the safety of ADS is critical, as their failures can lead to severe casualties. While ADS testing methods have advanced in recent years, they remain unsatisfactory in generating diverse test scenarios that induce distinct driving behaviors—a key requirement for thoroughly evaluating ADS across different situations. To address this, Prunario employs a novel simulation prediction technique to estimate ADS runtime behavior and prune redundant test scenarios that yield similar driving records. Experimental results demonstrate Prunario's effectiveness: it uncovered 23 previously undetected bugs in an industrial-strength ADS and outperformed three state-of-the-art testing techniques.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**.

Additional Key Words and Phrases: autonomous driving system, simulation prediction

ACM Reference Format:
Minsu Kim, Sunbeom So, and Hakjoo Oh. 2026. Prunario: Testing Autonomous Driving Systems by Pruning Likely Redundant Scenarios. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 154 (April 2026), 30 pages. https://doi.org/10.1145/3798262

1 Introduction

Testing autonomous driving systems (ADS) is critical to ensuring their safety. Since the 1980s, extensive research in industry and academia has advanced the development of practical ADS technologies [32]. Prominent examples include Tesla's Autopilot [8], Tier IV's Autoware [38], Baidu's Apollo [5], Waymo's Waymo Driver [25], and Stanford University's Stanley [13] and Junior [79]. Despite significant progress, current ADSs remain susceptible to critical flaws due to their inherent complexity, which can result in catastrophic accidents involving human casualties [33, 36, 37, 43, 46]. Therefore, developing techniques to ensure ADS safety before real-world deployment is both urgent and essential.

Simulation Testing. In this paper, we propose a novel testing technique to automatically identify critical bugs in ADS. We focus on simulation-based testing, where ADS are tested in virtual environments rather than on real roads, using high-fidelity physical simulators (e.g., CARLA [31], LGSVL [82]). Simulation testing (hereafter referred to as testing) is widely adopted by ADS developers [76], as it complements real-world testing by addressing safety and cost concerns [76, 82].

*Corresponding authors.

Authors' Contact Information: Minsu Kim, Korea University, Seoul, Republic of Korea, minsu@korea.ac.kr; Sunbeom So, Korea University, Seoul, Republic of Korea, sunbeom_so@korea.ac.kr; Hakjoo Oh, Korea University, Seoul, Republic of Korea, hakjoo_oh@korea.ac.kr.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2475-1421/2026/4-ART154
https://doi.org/10.1145/3798262

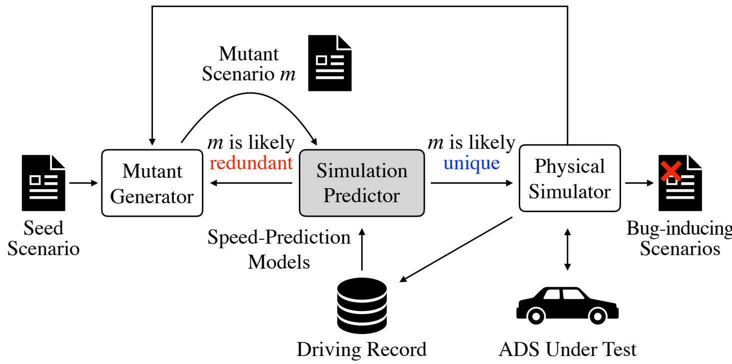


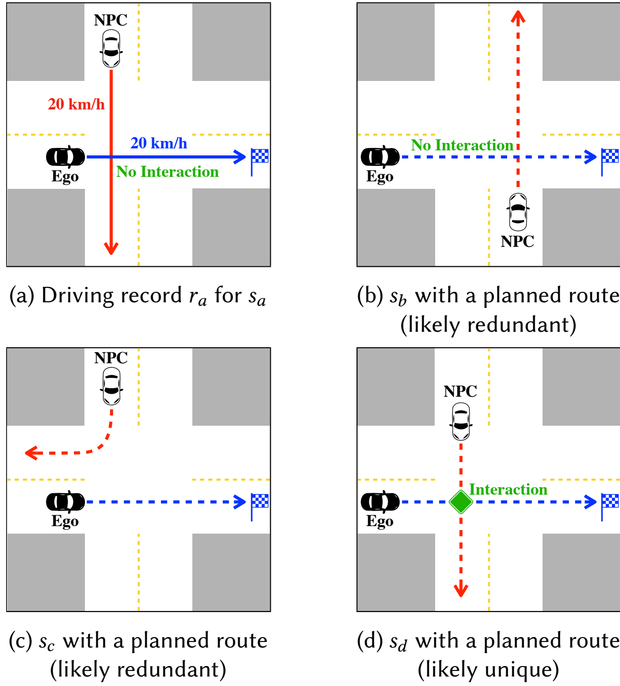
Fig. 1. Overview of PRUNARIO.

Existing Approaches. Various techniques for automated ADS testing have emerged in recent years [48, 49, 51, 55, 56, 61–64, 72–75, 80, 86–89, 92, 97, 99]. However, these methods continue to face performance challenges. In particular, existing techniques struggle to generate diverse test scenarios cost-effectively, a crucial requirement for thoroughly validating ADS across varied conditions. For instance, our study in Section 5 shows that 93% of scenarios produced by three recent tools—ScenarioFuzz [92], DRIVEFUZZ [72], and SAMOTA [61]—are virtually identical to other tested scenarios, significantly reducing testing efficiency and bug-detection capabilities. The issue of redundant scenarios is exacerbated by the time-intensive nature of simulation testing [49, 55, 61]. Specifically, to maintain simulation stability (e.g., avoiding oscillations of control commands), realistic simulations require synchronization between real and simulation time [55]; for example, running a single test scenario in which a vehicle travels 10 km at 10 m/s takes about 17 minutes to complete.

This Work. We propose PRUNARIO, a novel technique to enhance the efficiency of ADS testing. The primary feature of PRUNARIO lies in its ability to detect and prune *likely redundant* scenarios that would generate duplicate driving records when executed. Figure 1 illustrates our approach. The inputs are a seed scenario and an ADS under test, and the output is a set of scenarios that trigger traffic violations in the ADS. Like existing techniques [49, 51, 55, 56, 61–64, 72–75, 86, 88, 89, 92, 97, 99], PRUNARIO employs mutation-based fuzzing, iteratively generating randomly mutated scenarios and evaluating the ADS against them in a simulation environment. However, this baseline approach inherits the performance limitations of prior methods.

To improve testing efficiency, the key innovation of PRUNARIO is to "statically" predict the ADS runtime behavior without executing the computationally expensive physical simulator, pruning scenarios likely to yield redundant driving records. Inspired by static program analysis, which infers a program's runtime behavior without execution, our approach integrates a static simulation predictor into ADS testing (Figure 1). Specifically, we train models from prior simulation runs to estimate the driving record $\hat{r}$ for a mutant scenario m. By comparing $\hat{r}$ with existing records, we determine whether m is likely to expose new behaviors. If m is predicted to produce unique behavior, we execute it in the simulator for accurate validation; otherwise, we prune it to avoid redundant testing. Unlike physical simulations, which comprehensively model real-world dynamics (e.g., real-time synchronization), our predictor is lightweight and fast, as it focuses on capturing partial driving records, such as speeds at waypoints along a planned route.

Experimental results demonstrate that PRUNARIO is highly effective at detecting critical bugs in ADS. We applied PRUNARIO to evaluate recent versions of Autoware [38], a popular open-source ADS with industrial applications such as autonomous valet parking [1] and cargo delivery [2].



(a) Driving record r_a for s_a

(b) s_b with a planned route (likely redundant)

(c) s_c with a planned route (likely redundant)

(d) s_d with a planned route (likely unique)

Fig. 2. Examples for illustrating our goal. (a) shows a schematic driving record of r_a generated by executing a scenario s_a. (b)–(d) present schematic scenarios of s_b–s_d.

Prunario identified 23 previously unknown bugs, of which 20 have been confirmed by the developers. To benchmark performance, we compared Prunario against three state-of-the-art tools: ScenarioFuzz [92], DriveFuzz [72], and SAMOTA [61]. The result shows that Prunario significantly outperforms these tools in detecting unique violations.

Contributions. We summarize our contributions below.

- We propose a novel approach for boosting the efficiency of ADS testing. The main technical contribution is simulation prediction, which enables the efficient detection and pruning of likely redundant scenarios without running physical simulators.
- We prove the practicality of Prunario by reporting 23 new bugs found in Autoware [38] and conducting the comparative experiments with the three state-of-the-art testing tools: DriveFuzz [72], SAMOTA [61], ScenarioFuzz [92].
- We make our tool and data publicly available on Zenodo [71] and GitHub [15].

2 Overview

2.1 Motivating Example

Figure 2a depicts a driving record r_a generated from the simulation of a scenario s_a. The ego vehicle traveled from west to east, while the NPC vehicle (i.e., another vehicle in the scenario) moved from north to south, both at a speed of 20 km/h. Even though the vehicles' trajectories overlap at the junction, they neither encountered nor collided with each other, since the ego vehicle passed the intersecting point much earlier. Figures 2b–2d depict candidate scenarios s_b–s_d with each vehicle's expected trajectory (obtainable from ADS's planning module, such as Autoware's planner [45]).

Given candidate scenarios s_b–s_d, Prunario analyzes their runtime behavior without simulator execution, and determines that scenarios s_b and s_c are likely redundant with s_a, while scenario s_d exposes distinct behavior. To reach this conclusion, Prunario predicts that executing s_b and s_c would generate driving records similar to r_a from the ego vehicle's perspective; assuming each vehicle travels at 20 km/h, the ego vehicle would move along a straight line without vehicle interactions, as in r_a. Specifically, in s_b, the ego vehicle arrives at the intersection much later (Figure 2b), and in s_c, the vehicles' expected trajectories do not overlap (Figure 2c). Conversely, s_d is likely unique, as it would involve an interaction where both vehicles approach the intersection nearly simultaneously, producing a distinct driving record (Figure 2d). As a result, Prunario prunes s_b and s_c from the testing space and executes s_d only.

Comparison to Existing Approaches. Our approach introduces a novel simulation predictor that statically analyzes scenarios' runtime behavior to identify and eliminate redundant scenarios. While some ADS testing tools aim to reduce redundant scenarios [56, 62, 73, 75, 86, 89, 97], their methods are typically limited and fall into two categories. The first category consists of tools that identify redundant scenarios dynamically by executing them on a physical simulator. These tools can either prune [56, 62, 73] or deprioritize [89] subsequent scenarios (i.e., mutant scenarios). However, they are inherently unable to eliminate initial redundant scenarios like s_b and s_c. The second category adopts a static approach, but without analyzing runtime behavior [75, 86, 97]. Instead, they rely on static fields of the scenario–such as the starting and ending points of vehicles–to identify redundancy. This can easily lead to incorrect pruning of scenarios like s_d, which may be considered similar to s_a based on their static fields but actually induce distinct runtime behaviors.

2.2 How Prunario Works

Prunario basically performs random mutation-based fuzzing with two optimization techniques from prior research: dynamically pruning redundant scenarios (Section 3.1.1) and prioritizing scenarios likely to trigger violations (Section 3.1.2). Despite these, its performance remains unsatisfactory (Section 5). To address this, Prunario uses learning-based simulation prediction to detect and prune likely redundant scenarios.

For simplicity, this section assumes a waypoint (a location in a map) is one-dimensional, i.e., a vehicle travels along a straight line. More general descriptions can be found in Section 3. Suppose we have a driving record $r_x = (r_e^x, r_n^x)$ for a scenario s_x, where r_e^x and r_n^x are individual driving records for the ego vehicle (e) and the NPC vehicle (n), respectively:

$$r_e^x = \underbrace{(0,0) \cdot (5,10) \cdot (10,0)}_{(wp_e^1, spd_e^1) \cdot (wp_e^2, spd_e^2) \cdot (wp_e^3, spd_e^3)}, \ r_n^x = \underbrace{(40,0) \cdot (30,20) \cdot (20,0)}_{(wp_n^1, spd_n^1) \cdot (wp_n^2, spd_n^2) \cdot (wp_n^3, spd_n^3)} \tag{1}$$

Here, each vehicle's record is a sequence of waypoint-speed pairs in frame $i \in [1,3]$. r_e^x indicates that the ego vehicle traveled in the right direction by 10 meters from wp_e^1 $(= 0)$ to wp_e^3 $(= 10)$, and the speed in frame 2 was spd_e^2 $(= 10$ m/s). r_n^x represents that the NPC vehicle moved in the left direction by 20 meters, where the speed in frame 2 was spd_n^2 $(= 20$ m/s).

Step 1: Learning Speed-Prediction Models. The goal of simulation prediction is to generate a vehicle's driving prediction, an expected sequence of waypoint-speed pairs, without executing a physical simulator. To do so, we first use the planning module of the ADS to compute a vehicle's expected route (a sequence of waypoints). Given such a planned route, we can construct a driving prediction by assigning an expected speed to each waypoint along the route, forming a sequence of waypoint-speed pairs. Therefore, the problem of simulation prediction essentially boils down to the problem of building speed-prediction models that can forecast the speed at each waypoint along the planned route.

Given training datasets for an ego vehicle (D_e) and an NPC vehicle (D_n), speed-prediction models for the vehicles, denoted $\mathcal{M}_e$ and $\mathcal{M}_n$, can be generated using off-the-shelf supervised learning algorithms (e.g., the ones provided by scikit-learn [54]). We obtain such D_e and D_n from previous driving records. For example, given r_e^x from (1), we may have a training sample $d_e = (v_e^2, spd_e^3) \in D_e$. Here, v_e^2 is the feature vector that describes the ego vehicle's driving status in frame 2, such as the remaining distance to the destination. That is, we build regression models, which predict the speed in the next frame based on the vehicle's driving status in the current frame.

Step 2: Generating Driving Predictions. Suppose we have $\mathcal{M}_e$, a learned speed-prediction model for an ego vehicle. Suppose also a scenario s_y specifies the starting point ($start_e$) and the end point (end_e) of the ego vehicle: $start_e = 10$ and $end_e = 20$. Then, we generate $\hat{r}_e^y$ (the ego vehicle's driving prediction for s_y) of the form:

$$\hat{r}_e^y = (wp_e^1, spd_e^1) \cdot (wp_e^2, spd_e^2) \cdots$$

Initially, $wp_e^1 = start_e = 10$ and $spd_e^1 = 0$ m/s. To construct the final prediction $\hat{r}_e^y$, we iteratively compute and append a waypoint-speed pair (e.g., (wp_e^2, spd_e^2)) to an interim result (e.g., (wp_e^1, spd_e^1)). Specifically, to obtain spd_e^2 using $\mathcal{M}_e$:

(a) Generate a feature vector v_e^1 that represents the driving status in frame 1 (i.e., at the waypoint wp_e^1), following the encoding scheme in Table 1.
(b) Obtain the speed in the next frame with $\mathcal{M}_e$. For example, assume $spd_e^2 = \mathcal{M}_e(v_e^1) = 10$ m/s.

To obtain the next waypoint wp_e^2:

(c) Compute the expected travel distance by $d = \frac{spd_e^1 + spd_e^2}{2} \times \frac{1}{fps}$ where $\frac{1}{fps}$ is the frame time (a fixed unit time per frame). Assuming $\frac{1}{fps} = 1$ s, d is 5 m.
(d) Select a location that is 5 meters (= d) away from wp_e^1 (the current waypoint) within a planned route. For example, if the planned route is $10 \cdot 12 \cdot 14 \cdot 16 \cdot 18 \cdot 20$, wp_e^2 is 15.

The remaining waypoint-speed pairs can be computed by repeating (a)–(d), until we obtain a waypoint close enough to end_e. Given $\mathcal{M}_n$ (a speed predictor for an NPC vehicle), $\hat{r}_n^y$ (an NPC vehicle's driving prediction for scenario s_y) can be obtained in a similar way.

Step 3: Pruning with Prediction Results. Once the driving predictions are computed, we detect redundant scenarios by transforming each prediction into a driving pattern sequence, an abstract form of driving prediction (and driving record) from the perspective of an ego vehicle. Consider the driving predictions for two scenarios s_y and s_z, where $\hat{r}_y = (\hat{r}_e^y, \hat{r}_n^y)$ is the prediction for s_y:

$$\hat{r}_e^y = (10, 0) \cdot (15, 10) \cdot (20, 0), \quad \hat{r}_n^y = (20, 0) \cdot (30, 20) \cdot (40, 0) \tag{2}$$

and $\hat{r}_z = (\hat{r}_e^z, \hat{r}_n^z)$ is the prediction for s_z:

$$\hat{r}_e^z = (15, 0) \cdot (20, 10) \cdot (30, 0), \quad \hat{r}_n^z = (30, 0) \cdot (20, 20) \cdot (10, 0) \tag{3}$$

We first convert the previous driving record r_x from (1) into its driving pattern sequence π_x, following the abstraction method in Section 3.1.1:

$$\pi_x : \mathsf{START} \cdot (\uparrow, \times) \cdot \mathsf{END}$$

Intuitively, π_x represents that the ego vehicle moved straight ($\uparrow$) without interactions ($\times$) with the NPC vehicle; as indicated in (1), the trajectories of the vehicles do not overlap. We also obtain π_y and π_z (the pattern sequences for $\hat{r}_y$ and $\hat{r}_z$) using the same method:

$$\pi_y : \mathsf{START} \cdot (\uparrow, \times) \cdot \mathsf{END}, \quad \pi_z : \mathsf{START} \cdot (\uparrow, \bigcirc) \cdot \mathsf{END}$$

where π_y means that no vehicle interactions ($\times$) are expected; according to (2), no waypoints overlap in the same frame (i.e., at the same time). By contrast, the symbol $\bigcirc$ in π_z indicates that

a vehicle interaction is expected during the simulation of s_z; according to (3), the vehicles are expected to be positioned very closely in frame 2. Observe that, according to the abstraction results that express core driving aspects, the prediction for s_y is a duplicate of the previous record r_x (i.e., $\pi_y = \pi_x$), whereas the prediction for s_z is not (i.e., $\pi_z \neq \pi_x$). Thus, we determine that s_y is likely redundant with s_x, while s_z is likely unique. Consequently, Prunario prunes s_y and tests the ADS against s_z only.

3 Approach

We describe Prunario in detail: basic fuzzing algorithm (Section 3.1) and simulation prediction (Section 3.2).

A simulation map, *map*, is a virtual world in which vehicle objects are simulated. A waypoint $wp = (x, y, z, pitch, yaw)$ is a location on *map*, such as vehicles' starting and end points. x, y, and z are positions at X, Y, and Z-axes in a three-dimensional coordinate system. *pitch* is the vertical rotation angle with respect to Y-axis; if a vehicle is going up (resp., going down), $pitch > 0$ (resp., $pitch < 0$). *yaw* refers to the horizontal rotation angle with respect to Z-axis; if a vehicle is heading left (resp., right), $yaw < 0$ (resp., $yaw > 0$).

Scenario. A scenario consists of each vehicle's specification that must be satisfied when executed. For brevity, we assume exactly two vehicles (an ego vehicle e, and an NPC vehicle n) appear during the simulation. However, our implementation supports more complex forms of scenarios (Section 4), including multiple NPC vehicles. Under the assumption, we define a scenario s as a 3-tuple:

$$s = (map, spec_e, spec_n).$$

$spec_e$ and $spec_n$ are the specifications of an ego vehicle (e) and an NPC vehicle (n). In detail, $spec_e = (start_e, end_e)$ represents that an ego vehicle's objective is to travel the map from the starting point ($start_e$) to the end point (end_e). $spec_n = (start_n, end_n, mode)$ specifies the NPC vehicle's starting point ($start_n$) and end point (end_n), along with its driving mode (*mode*); we aim to find bugs more effectively by exposing the ego vehicle to diverse situations as in prior work [63, 64, 72, 80, 92]. We assume *mode* is either auto or immobile for brevity (see Section 4 for more detail). If *mode* = auto, the NPC vehicle autonomously navigates the map from $start_n$ to end_n, based on the NPC vehicle's ADS (e.g., Behavior Agent [6] supported in the CARLA [31] simulator). If *mode* = immobile, the NPC vehicle remains stationary at the starting point, i.e., $start_n = end_n$.

Simulation Result. A simulation result, (r, b), is the output generated by executing a scenario on a simulator. b is the traffic violation type (Section 4), such as collision. If no violations are found, we write $b = \perp$. $r = (r_e, r_n)$ is a driving record, where r_e and r_n are the individual records for the ego and the NPC vehicle, respectively. Specifically, r_e is a sequence of *frame states*:

$$r_e = (wp_e^1, spd_e^1) \ \cdots \ (wp_e^{|r|}, spd_e^{|r|}),$$
$$r_n = (wp_n^1, spd_n^1) \ \cdots \ (wp_n^{|r|}, spd_n^{|r|})$$

where a frame state is a waypoint-speed pair (wp_e^i, spd_e^i) (resp., (wp_n^i, spd_n^i)) in each *frame* $i \in [1, |r|]$ and $|r|$ represents the length of the record r. A frame is a single logical time step, and its duration is called *frame time* ($= 1/fps$).

3.1 Basic Fuzzing Algorithm

The goal of ADS testing is to generate as many violation-triggering scenarios as possible. We say a scenario triggers a violation (Section 4) if an ego vehicle, controlled by the ADS under test, exhibits unsafe driving behaviors (e.g., collision) during the simulation.

Algorithm 1 PRUNARIO

Input: autonomous driving system *ADS*, initial seed scenario *seed*
Output: a set of violation-triggering scenarios V

```
1: P ← ∅
2: W ← {(seed, 0)}
3: repeat
4:     (s, score) ← argmax_{(s,score)∈W} score                         ▷ s: seed scenario
5:     W ← W \ {(s, score)}
6:     i ← 0                                                            ▷ i: current mutation number
7:     while i < k do                                                   ▷ k: preset mutation bound
8:         s′ ← Mutate(s)                                               ▷ s′: mutant scenario
9: (+)     r̂ ← Predict(s′, P)                                           ▷ §3.2, Alg.2
10: (+)    if r̂ = fail ∨ ¬LikelyRedundant(r̂, P) then
11:            (r, b) ← Simulate(ADS, s′)
12:            if ¬Redundant((r, b), P) then                            ▷ §3.1.1
13:                if b = ⊥ then
14:                    score′ ← Feedback(r)                             ▷ §3.1.2
15:                    W ← {(s′, score′)} ∪ W
16:            P ← {(s′, (r, b))} ∪ P
17:            i ← i + 1
18: until W = ∅ or timeout
19: V ← ExtractV(P)
20: return V
```

Algorithm 1 shows the architecture of PRUNARIO. The inputs are an ADS to test (*ADS*), and a seed scenario (*seed*). The output is a set of violation-triggering scenarios (V). The algorithm maintains a set of previous execution logs (P). An execution log $p \in P$ is an input-output pair of the simulation, i.e., $p = (s, (r, b))$ where s is an executed scenario and (r, b) is the simulation result of s. The algorithm also keeps a workset W that is a set of pairs $(s, score)$. s is a scenario to mutate, and *score* is a number called *risk score* (Section 3.1.2), which quantifies how beneficial mutating s is in terms of finding violations. In our basic algorithm, line 9 is ignored and the whole condition at line 10 is always *true*.

At line 1, we initialize P with the empty set. At line 2, we initialize W with the singleton set of $(seed, 0)$, where 0 is the dummy score of *seed*. We enter the repeat-until loop (lines 3–18). We pick the scenario s with the highest risk score from W (line 4), and remove it from W (line 5). After initializing i with 0 (line 6), we enter the while-loop (lines 7–17) to generate k mutant scenarios ($k = 10$ in the implementation). At line 8, MUTATE(s) outputs a randomly generated mutant scenario s' (will be explained shortly). At line 11, we obtain a simulation outcome (r, b) by executing s' using a simulator (e.g., CARLA [31]). If s' is a unique (line 12) and normal scenario that does not trigger violations (line 13), we compute the risk score of s' (line 14) and add $(s', score')$ to W (line 15). After checking the redundancy of s', we add the input-output pair of the simulation to P (line 16), and increase the current mutation number i (line 17). The repeat-until loop repeats until W becomes empty or the preset time limit expires (line 18). Once the loop terminates, EXTRACTV collects V, the largest set of unique violation-triggering scenarios (line 19): $V = \{s \mid (s, (r, b)) \in \text{argmax}_{P' \subseteq P} |F(P')|\}$ such that

$$F(P') = \{p \mid P' \ni p = (s, (r, b)),\ b \neq \bot,\ \neg\text{Redundant}((r, b), P' \setminus \{p\})\}$$

where the predicate Redundant is defined in Section 3.1.1. At line 20, we finally return V.

Mutation Operators. Given a seed scenario $s = (map, spec_e, spec_n)$ where $spec_e = (start_e, end_e)$ and $spec_n = (start_n, end_n, mode)$, MUTATE (line 8 in Algorithm 1) generates a mutant scenario s' by randomly selecting and modifying one waypoint wp from each vehicle's specification. Specifically, it alters either $start_e$ or end_e in $spec_e$, and $start_n$ or end_n in $spec_n$, using one of the two operations. The first operation, *random relocation*, replaces wp with a randomly chosen $wp' \in VW$, where VW denotes the set of valid waypoints provided by map creators [23]. The second operation, *path-aligned shift*, moves wp forward or backward by a random distance $x \in [2, 10]$ meters along the lane. The mutation is repeated until we produce a mutant scenario where the planned routes [45] of the vehicles overlap, thereby increasing the likelihood of vehicle interactions and violation detection. In Section 4, we describe mutation operators in more detail, considering more complex scenarios.

3.1.1 *Pruning Successors of Redundant Scenarios.*

Scenarios obtained from the mutation of redundant scenarios are more likely to be useless (e.g., due to the mutation operators that do not significantly change the vehicles' missions). To avoid generating such undesirable scenarios, several tools [56, 62, 73] detect redundant scenarios based on their execution results, and do not add those scenarios to a seed pool (W in Algorithm 1). This way, mutant scenarios of redundant scenarios can be excluded from the search space. We use a technique (line 12 in Algorithm 1) for a similar purpose, proceeding in two steps: abstracting driving records into *driving pattern sequences*, and checking redundancy based on the pattern sequences.

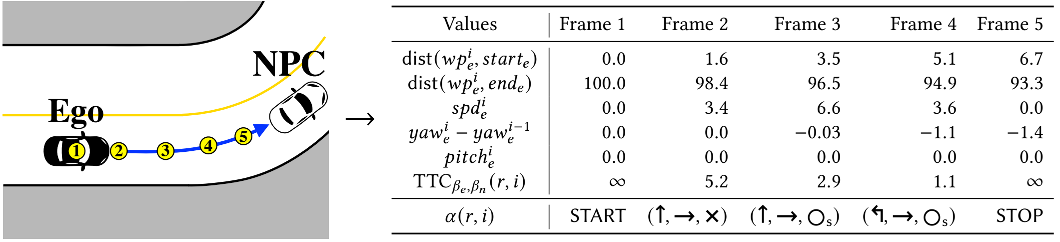
Abstracting Simulation Results. We represent a driving record as an abstract form. The key principle of our abstraction is to omit unnecessary details from an ego vehicle's perspective (e.g., Section 2.1), in order to effectively detect virtually identical scenarios. Our abstraction method is inspired by prior work on ADS testing [56, 57, 73, 77]. We first introduce an abstraction for a frame state (a waypoint-speed pair per frame) and then extend it to an abstraction for a record (a sequence of frame states). Let us assume the record $r = (r_e, r_n)$ for a scenario s is given, where $r_o = (wp_o^1, spd_o^1) \cdots (wp_o^{|r|}, spd_o^{|r|})$ for $o \in \{e, n\}$, and $wp_o^i = (x_o^i, y_o^i, z_o^i, pitch_o^i, yaw_o^i)$ for $i \in [1, |r|]$. We assume all information from s (e.g., $start_e$, end_e) is available when defining the functions.

We define α to transform the i-th frame state into its driving pattern, from the perspective of the ego vehicle:

$$\alpha(r, i) = \begin{cases} \text{START} & \text{if } \mathrm{dist}(wp_e^i, start_e) \leq 0.01 \text{ m} \\ \text{END} & \text{if } \mathrm{dist}(wp_e^i, end_e) \leq 1 \text{ m} \\ \text{STOP} & \text{if } spd_e^i \leq 1 \text{ km/h} \\ (\textsc{Hor}(r,i), \textsc{Vert}(r,i), \textsc{Inter}(r,i)) & \text{if } spd_e^i > 1 \text{ km/h} \end{cases}$$

where dist [22] computes the Euclidean distance between the two waypoints, 0.01 m (resp., 1 m) is the threshold for determining whether the ego vehicle is close enough to the starting point (resp., the end point), and 1 km/h is the threshold for determining whether the ego vehicle (virtually) stops or not in frame i. HOR, VERT, and INTER are functions that generate driving patterns of the moving ego vehicle in frame i. Specifically, $\textsc{Hor}(r, i)$ returns a driving pattern based on the ego vehicle's horizontal rotation in frame i, in comparison with frame $i - 1$, while $\textsc{Vert}(r, i)$ returns a driving pattern based on the ego vehicle's vertical rotation angle in frame i:

$$\textsc{Hor}(r, i) = \begin{cases} \uparrow & \text{if } |yaw_e^i - yaw_e^{i-1}| \leq 0.08^\circ \\ \Lsh & \text{if } yaw_e^i - yaw_e^{i-1} < -0.08^\circ \\ \Rsh & \text{if } yaw_e^i - yaw_e^{i-1} > 0.08^\circ \end{cases} \qquad \textsc{Vert}(r, i) = \begin{cases} \rightarrow & \text{if } |pitch_e^i| \leq 5.0^\circ \\ \nearrow & \text{if } pitch_e^i > 5.0^\circ \\ \searrow & \text{if } pitch_e^i < -5.0^\circ \end{cases}$$



Values	Frame 1	Frame 2	Frame 3	Frame 4	Frame 5
$\text{dist}(wp_e^i, start_e)$	0.0	1.6	3.5	5.1	6.7
$\text{dist}(wp_e^i, end_e)$	100.0	98.4	96.5	94.9	93.3
spd_e^i	0.0	3.4	6.6	3.6	0.0
$yaw_e^i - yaw_e^{i-1}$	0.0	0.0	−0.03	−1.1	−1.4
$pitch_e^i$	0.0	0.0	0.0	0.0	0.0
$\text{TTC}_{\beta_e,\beta_n}(r,i)$	∞	5.2	2.9	1.1	∞
$\alpha(r,i)$	START	(↑, →, ✕)	(↑, →, ○$_s$)	(↰, →, ○$_s$)	STOP

Fig. 3. A visualization of a driving record r (left). The table on the right illustrates the abstraction process: the last row ($\alpha(r,i)$) presents the driving patterns for each frame $i \in [1,5]$ of r, while the remaining rows list the numerical values used in α to derive these driving patterns. Units are omitted for brevity.

where ±0.08° and ±5.0° are the thresholds to handle slight variations. Finally, $\textsc{Inter}(r,i)$ captures whether an interaction between the ego and NPC exists ($\bigcirc_\star$) or not (✕):

$$\textsc{Inter}(r,i) = \begin{cases} \bigcirc_\star & \text{if } \text{TTC}_{\beta_e,\beta_n}(wp_e^i, spd_e^i, wp_n^i, spd_n^i) < 3\text{ s} \\ \times & \text{otherwise} \end{cases}$$

where $\bigcirc_\star$ represents that the ego vehicle has an interaction with a stopped ($\star$ = s) or moving ($\star$ = m) NPC vehicle. β_e and β_n are the predetermined sizes of each vehicle's bounding box [14]. TTC [69] computes the time-to-collision, i.e., the expected remaining time until the ego vehicle collides with the NPC vehicle. That is, the ego and NPC vehicles are considered to influence each other if they are likely to encounter within a few seconds. We set the thresholds for defining α based on a pilot study conducted during development; we qualitatively inspected whether the generated driving patterns were consistent with the behaviors observed in log files (simulation videos), and selected the values that remained robust to small variations.

Example 1. *Figure 3 depicts how the function α derives driving patterns from a driving record r. For instance, the pattern in frame 1 is* START *as the ego vehicle is at the starting point, i.e.,* $\text{dist}(wp_e^i, start_e) = 0$ *m. Also, the pattern in frame 2 (resp., frame 3) is* (↑, →, ✕) *(resp.,* (↑, →, ○$_s$)*) as the ego vehicle is traveling straight on a flat road with no interaction (resp., with interaction). In frame 4 and 5, the ego vehicle turns left ($yaw_e^i - yaw_e^{i-1} < -0.08°$) and stops ($spd_e^i \leq 1$ km/h), respectively.*

On top of α, $\textsc{PatSeq}$ generates the final pattern sequence $\pi_1 \cdots \pi_l$ of a record r:

$$\textsc{PatSeq}(r) = \pi_1 \cdots \pi_l$$

where $\pi_1 \cdots \pi_l = \textsc{DeDup} \circ \textsc{Cons}(p_1 \cdots p_{|r|})$ such that $l \leq |r|$. Here, $p_i = \alpha(r,i)$ indicates a driving pattern in frame $i \in [1, |r|]$. $\textsc{Cons}$ and $\textsc{DeDup}$ are the postprocessing functions that convert driving patterns into their intended forms. Let σ be the *meaningful pattern threshold* – the lower bound on the number of occurrences required for a pattern to be regarded as meaningful; in the implementation, $\sigma = 20$, i.e., we consider a pattern meaningful if it lasts at least 1 second ($= \sigma \times \frac{1}{fps}$). $\textsc{Cons}$ removes noisy patterns (excluding START and END) that do not appear consecutively for at least σ frames. Specifically, given $\pi = \pi_1 \cdots \pi_n$, $\textsc{Cons}(\pi)$ outputs $\textsc{Cons}'(\pi, \pi_1, 0)$ where

$$\textsc{Cons}'(p,x,k) = \begin{cases} \text{START} \cdot \textsc{Cons}'(p_2 \cdots p_n, p_2, 0) & \text{if } p = p_1 \cdot p_2 \cdots p_n \land x = \text{START} \\ \text{END} \cdot \textsc{Cons}'(p_2 \cdots p_n, p_2, 0) & \text{else if } p = p_1 \cdot p_2 \cdots p_n \land x = \text{END} \\ \textsc{Cons}'(p_2 \cdots p_n, x, k+1) & \text{else if } p = p_1 \cdot p_2 \cdots p_n \land x = p_1 \\ \textsc{Cons}'(p, p_1, 0) & \text{else if } p = p_1 \cdot p_2 \cdots p_n \land x \neq p_1 \land k < \sigma \\ x^k \cdot \textsc{Cons}'(p, p_1, 0) & \text{else if } p = p_1 \cdot p_2 \cdots p_n \land x \neq p_1 \land k \geq \sigma \\ x^k & \text{else if } |p| = 0 \land k \geq \sigma \\ \epsilon & \text{else if } |p| = 0 \land k < \sigma \end{cases}$$

Here, x^k indicates the sequence of k-consecutive x patterns. DeDup eliminates consecutive duplicated patterns:

$$\textsc{DeDup}(p) = \begin{cases} p_1 \cdot \textsc{DeDup}(p_2 \cdots p_n) & \text{if } p = p_1 \cdot p_2 \cdots p_n \wedge p_1 \neq p_2 \\ \textsc{DeDup}(p_2 \cdots p_n) & \text{if } p = p_1 \cdot p_2 \cdots p_n \wedge p_1 = p_2 \\ p & \text{if } |p| = 1 \end{cases}$$

Example 2. *Suppose we have a pattern sequence* $p = \mathsf{START} \cdot x \cdot x \cdot y \cdot x \cdot x \cdot \mathsf{END}$, *where* $x \neq \mathsf{START}$, $x \neq \mathsf{END}$, *and* $x \neq y$. *Assuming* $\sigma = 2$, *Cons*(p) *outputs* $p' = \mathsf{START} \cdot x \cdot x \cdot x \cdot x \cdot \mathsf{END}$. *Also, DeDup*$(p') = \mathsf{START} \cdot x \cdot \mathsf{END}$.

Checking Redundancy. We detect and prune redundant scenarios based on their pattern-based abstractions. Let s be a scenario, r be a driving record for s, and b be a violation type (including $\perp$) detected during the simulation of s. We define the predicate Redundant (line 12 of Algorithm 1):

$$\mathsf{Redundant}((r, b), P) \iff \exists (s', (r', b')) \in P.\ b = b' \wedge \textsc{PatSeq}(r) = \textsc{PatSeq}(r').$$

That is, s is *redundant* if there is a previously executed scenario s' with the same driving pattern sequence and violation detection result.

3.1.2 Risk Score-Guided Scenario Prioritization. Like most prior work [48, 49, 51, 55, 56, 61–64, 72–75, 86–89, 92, 97, 99], we take driving records as feedback, and use them to prioritize scenarios that can be beneficial for finding violations. To this end, we use the function Feedback, which computes risk scores (*score′* at line 14 in Algorithm 1) to prioritize scenarios whose execution results reveal more signs of violations.

To define feedback function, we assume a record $r = (r_e, r_n)$ for a scenario is given, where $r_o = (wp_o^1, spd_o^1) \cdots (wp_o^{|r|}, spd_o^{|r|})$ for $o = \{e, n\}$, and $wp_o^i = (x_o^i, y_o^i, z_o^i, pitch_o^i, yaw_o^i)$ for $i \in [1, |r|]$.

We define procedure Feedback that computes risk scores (*score′* at line 14 in Algorithm 1) to prioritize scenarios whose execution results reveal more signs of violations:

$$\textsc{Feedback}(r) = F_t(r) + F_a(r) + F_l(r).$$

F_t assigns higher scores to scenarios with a higher likelihood of collision, based on the time-to-collision (TTC [69]) between the vehicles:

$$F_t(r) = \frac{1}{\min\{\mathrm{TTC}_{\beta_e,\beta_n}(wp_e^i, spd_e^i, wp_n^i, spd_n^i) \mid i \in [1, |r|]\}}$$

where β_e and β_n are the predetermined sizes of each vehicle's bounding box [14]. F_a favors scenarios where the ego vehicle's maximum accelerations are larger:

$$F_a(r) = \frac{\max\{\Delta \mid i \in [2, |r|], \Delta = spd_e^i - spd_e^{i-1}, \Delta > 0\}}{\gamma_a}$$

where γ_a is the normalization factor (5 km/h in the implementation). Lastly, F_l values scenarios that show higher chances of lane invasion:

$$F_l(r) = \frac{\max\{\mathsf{dist}(wp_e^i, c^i) \mid i \in [1, |r|], c^i = \mathsf{center}(wp_e^i)\}}{\gamma_l}$$

Here, dist [22] computes the Euclidean distance between wp_e^i and c^i; center [24] outputs the waypoint c^i projected onto the lane's center line from wp_e^i; and γ_l indicates half of the width of the lane that contains wp_e^i, i.e., $\gamma_l = \frac{\mathsf{LaneWidth}(wp_e^i)}{2}$ where LaneWidth [44] computes the lane width of a given waypoint.

Algorithm 2 Simulation Prediction (PREDICT)

Input: scenario $s = (map, spec_e, spec_n)$, a set of execution logs P
Output: a simulation prediction result for s (or fail)
1: **if** $P = \emptyset$ **return** fail
2: $(\mathcal{M}_e, \mathcal{M}_n) \leftarrow$ LEARN(P) ▷ §3.2.1, Alg.3
3: $\hat{r}_e \leftarrow$ PREDICTEGO$(\mathcal{M}_e, map, spec_e)$ ▷ §3.2.2, Alg.4
4: $\hat{r}_n \leftarrow$ PREDICTNPC$(\mathcal{M}_n, map, |\hat{r}_e|, spec_n)$ ▷ §3.2.2, Alg.5
5: **return** $(\hat{r}_e, \hat{r}_n)$

3.2 Simulation Prediction

Now, we present the key contribution of this paper: simulation prediction to statically estimate the ADS runtime behavior.

Goal. Given a scenario s, our predictor is designed to generate its expected driving record $\hat{r} = (\hat{r}_e, \hat{r}_n)$, where $\hat{r}_e$ and $\hat{r}_n$ are individual driving predictions for the ego vehicle and NPC vehicle. Specifically, we aim to produce a *partial* driving prediction, up to the frame where the first few vehicle interactions or premature stops (i.e., unexpected stops before reaching end points) are expected to occur. This is because making accurate predictions after such frames is extremely challenging; we elaborate on this point in Section 3.2.1. For a similar reason, predicting violation types is also beyond the scope of our method. Despite these restrictions, our predictor is useful for enhancing the violation-finding capability (Section 5).

Algorithm 2 describes the procedure PREDICT, (invoked at line 9 in Algorithm 1). If there are no prior execution logs ($P = \emptyset$), we return fail (line 1). Otherwise (line 2), we construct two *speed-prediction models* from P that predict speeds along a planned route (Section 3.2.2). Using the models, we carry out simulation predictions for each vehicle (lines 3 and 4). Finally, we return the pair of individual predictions (line 5), which will be refined during pruning (Section 3.2.3). Section 3.2.1 and 3.2.2 describe the details of lines 2–4.

3.2.1 Learning Speed-Prediction Models. We formulate the learning problem for speed-prediction models and present the learning procedure LEARN (Algorithm 3).

Learning Problem. Let R be the set of driving records from execution logs P. The learning objective is to solve the following optimization problem:

$$\text{For each } o \in \{e, n\}\text{, find a model } F_o \text{ that minimizes } \sum_{(r_e,r_n)\in R} \sum_{i=1}^{|r_o|-1} (F_o(x_o^i) - spd_o^{i+1})^2 \quad (4)$$

where r_o is the driving record of a vehicle object o, and x_o^i denotes its partial record up to frame i, i.e., $r_o = (wp_o^1, spd_o^1) \cdot (wp_o^2, spd_o^2) \cdots$ and $x_o^i = (wp_o^1, spd_o^1) \cdots (wp_o^i, spd_o^i)$. That is, for each vehicle object $o \in \{e, n\}$, we aim to find a regression model F_o to minimize the mean squared error between the predicted next-frame speeds ($F_o(x_o^i)$) and the actual next-frame speeds (spd_o^{i+1}). The problem (4) can be viewed as an instance of an autoregressive learning problem, in that the next-frame speeds are predicted conditioned on the driving history (x_o^i) up to the current frame.

While F_o in (4) can, in principle, be instantiated with any regression model, we restrict the hypothesis space for F_o to random forest regressors, due to their low training overhead and their robustness to complex (potentially nonlinear) data. Thus, the problem (4) can be restated as follows:

$$\text{For each } o \in \{e, n\}\text{, find a model } \mathcal{M}_o \text{ that minimizes } \sum_{(r_e,r_n)\in R} \sum_{i=1}^{|r_o|-1} (\mathcal{M}_o(\mathbf{v}_o^i) - spd_o^{i+1})^2 \quad (5)$$

Algorithm 3 Learning Speed-Prediction Models (Learn)

Input: P (a set of execution logs)
Output: $(\mathcal{M}_e, \mathcal{M}_n)$ (learned models for ego and NPC)
1: $(D_e, D_n) \leftarrow (\emptyset, \emptyset)$
2: $R \leftarrow \{\text{RefineR}(r) \mid (-, (r, -)) \in P\}$
3: **for each** $(r_e, r_n) \in R$ **do**
4: $\quad D_e \leftarrow \{(\mathbf{v}_e^i, spd_e^{i+1}) \mid 1 \leq i < |r_e|\} \cup D_e$ ▷ $\mathbf{v}_e^i \in \mathbb{R}^m$: feature for the ego (Table 1)
5: $\quad D_n \leftarrow \{(\mathbf{v}_n^i, spd_n^{i+1}) \mid 1 \leq i < |r_n|\} \cup D_n$ ▷ $\mathbf{v}_n^i \in \mathbb{R}^m$: feature for the NPC (Table 1)
6: Train $\mathcal{M}_e$ and $\mathcal{M}_n$ from D_e and D_n respectively.
7: **return** $(\mathcal{M}_e, \mathcal{M}_n)$

where F_o is instantiated using a random forest regressor $\mathcal{M}_o : \mathbb{R}^m \rightarrow \mathbb{R}$, defined over fixed-size m-dimensional feature vectors of x_o^i, i.e., $\mathbf{v}_o^i = \langle v_1, \cdots, v_m \rangle$. The feature representation used to train the models in (5) is described later (Table 1).

Procedure Learn. Algorithm 3 presents Learn to approximate the objective in (5). Note that Learn is invoked during fuzzing (Algorithm 1) rather than in a separate stage. That is, Prunario requires no additional time for training models beyond the testing budget for Algorithm 1. Specifically, Learn retrains models from scratch using all execution logs P collected so far whenever Predict (Algorithm 2) is invoked during fuzzing.

At line 1, we initialize D_e and D_n, the training datasets for the ego and NPC vehicles, with the empty sets. At line 2, we obtain a set of refined driving records R, by preprocessing each prior driving record r (RefineR will be explained shortly). Lines 3–5 iterate over R to construct D_e and D_n. At line 4, for each frame i, we compute a feature vector $\mathbf{v}_e^i \in \mathbb{R}^m$ (following Table 1, $m = 7$) that captures the ego vehicle's driving state in that frame, pair it with spd_e^{i+1} (the speed in frame $i + 1$), and add the resulting sample to D_e. At line 5, we apply the same procedure to r_n for building D_n. Finally, we build and return speed-prediction models (lines 6–7), specifically using the off-the-shelf random forest learning algorithm provided by the scikit-learn library [54]. To prevent the overfitting of random forest regression models, our implementation uses the following parameter values: n_estimator = 35, min_samples_split = 12, min_samples_leaf = 5, max_features = 0.6, max_depth = 20, max_samples = 0.8. The meaning of each parameter can be found in the scikit-learn API documentation [12].

Refining Driving Records (RefineR). We observed the speed information after vehicle interactions or premature stops (before reaching end points) is highly irregular. Specifically, interactions may result in various outcomes (collision, stalling, or deceleration), leading to considerable speed fluctuations in subsequent frames for each run. Also, premature stops sometimes happen for no apparent reason (e.g., even when no obstacles are ahead), and the restarting timing after these abnormal stops was highly irregular too, making it difficult to predict speeds in following frames.

To avoid generating potentially harmful training data resulting from vehicle interactions or premature stops, we define RefineR (line 2 in Algorithm 3). Suppose a driving record $r = (r_e, r_n)$ is given. Recall from Section 3.1.1, σ denotes the meaningful pattern threshold to avoid being regarded as noisy patterns. Let S_e (resp., S_n) be the set of frame indices where the ego vehicle (resp., NPC vehicle) suddenly stops before reaching end points: $S_e = \{i+(\sigma-1) \mid \bigwedge_{j=0}^{\sigma-1} \alpha((r_e, r_n), i+j) = \text{STOP}\}$ and $S_n = \{i + (\sigma - 1) \mid \bigwedge_{j=0}^{\sigma-1} \alpha((r_n, r_e), i + j) = \text{STOP}\}$ where $i \in [1, |r|]$. Let x (resp., y) be the frame index where the first σ-consecutive interactions or premature stops of the ego vehicle (resp., NPC vehicle) end: that is, $x = \min(I \cup S_e)$ and $y = \min(I \cup S_n)$ where $I = \{i + (\sigma - 1) \mid \bigwedge_{j=0}^{\sigma-1} \text{Inter}(r, i + j) = \bigcirc_\star\}$ and $i \in [1, |r|]$. RefineR(r) outputs the refined record $r' = (r'_e, r'_n)$ by

Table 1. Features for training speed-prediction models. h_{start} (resp., h_{end}) denotes the waypoint at which the vehicle's current horizontal driving pattern (↑, ↰, ↱) begins (resp., ends). v_{start} (resp., v_{end}) denotes the waypoint at which the vehicle's current vertical driving pattern (→, ↗, ↘) begins (resp., ends).

ID	Description
f_1	The average speed (in km/h) over at most the past 30 frames
f_2	The travel distance between *start* and *cur*
f_3	The remaining distance between *cur* and *end*
f_4	The travel distance between h_{start} and *cur*
f_5	The remaining distance between *cur* and h_{end}
f_6	The travel distance between v_{start} and *cur*
f_7	The remaining distance between *cur* and v_{end}

removing all frame states in r_e (resp., r_n) after frame x (resp., y), where

$$r'_e = \begin{cases} r_e[1] \cdots r_e[x] & \text{if } x \neq \bot \\ r_e & \text{if } x = \bot \end{cases}, \quad r'_n = \begin{cases} r_n[1] \cdots r_n[y] & \text{if } y \neq \bot \\ r_n & \text{if } y = \bot \end{cases},$$

$r_e[a] = (wp_e^a, spd_e^a)$ for $a \in [1, x]$, and $r_n[b] = (wp_n^b, spd_n^b)$ for $b \in [1, y]$.

Example 3. *Let $\sigma = 2$. Suppose neither the ego vehicle nor the NPC vehicle suddenly stops, but they interact with each other during frames 6–8. Then, since $S_e = \emptyset$, $S_n = \emptyset$, and $I = \{7, 8\}$, we have $x = 7$ and $y = 7$. As a result, Refiner$(r) = (r'_e, r'_n)$ where $r'_e = r_e[1] \cdots r_e[7]$ and $r'_n = r_n[1] \cdots r_n[7]$. That is, we discard the record data after frame 7.*

Frame Features. Table 1 provides the features used to vectorize a vehicle's driving status in each frame. We describe the intuition behind how these features relate to predicting next-frame speeds, thereby helping minimize the mean squared prediction error in (5).

- f_1: It provides a baseline for prediction, helping prevent physically implausible speed fluctuations.
- f_2: As it increases, the vehicle tends to gently accelerate toward the destination.
- f_3: As it gets closer to zero, the vehicle tends to gently decelerate to stop safely at the destination.
- f_4: As it increases (i.e., the vehicle gets farther from a heading-transition point h_{start}), the vehicle tends to gently accelerate, as h_{start} is virtually treated as a new starting waypoint.
- f_5: As it gets closer to zero (i.e., the vehicle gets closer to the end of a heading-transition point h_{end}), the vehicle tends to gently decelerate to turn safely at h_{end}.
- f_6: As it increases (i.e., the vehicle gets farther from a slope-transition point v_{start}), the vehicle tends to gently accelerate, as v_{start} is virtually treated as a new starting waypoint.
- f_7: As it gets closer to zero (i.e., the vehicle gets closer to the end of a slope-transition point v_{end}), the vehicle tends to maintain its speed near v_{end}, avoiding unnecessary changes.

3.2.2 Generating Driving Predictions. We detail lines 3 and 4 in Algorithm 2: simulation predictions using learned models.

Prediction for Ego Vehicle. Algorithm 4 describes the procedure PredictEgo, which forecasts the driving record of the ego vehicle. At line 1, we invoke the planner (e.g., Autoware's planner [45]) of the ego vehicle's ADS, in order to compute an expected route (a sequence of waypoints from a starting point to an end point). We construct the first k frame states (line 2) and initialize the prediction with them (line 3). Here, k stands for the number of *idle* frames during which the ego

Algorithm 4 PREDICTEGO

```
Input: model M, map map, ego vehicle's specification (start, end)
Output: an individual prediction r̂_e
 1: route ← Planner(map, start, end)
 2: (wp^1, spd^1) ··· (wp^k, spd^k) ← (start, 0) ··· (start, 0)
 3: r̂_e ← (wp^1, spd^1) ··· (wp^k, spd^k)                     ▷ k: # of idle frames
 4: i ← k                                                     ▷ i: frame number
 5: while dist(wp^i, end) ≥ 1 m do
 6:     if loop timeout then return r̂_e
 7:     Generate a feature vector ⟨v_1, ··· , v_m⟩             ▷ Table 1
 8:     spd^{i+1} ← CLAMP(x, 0, spd^max)                       ▷ x = M(⟨v_1, ··· , v_m⟩)
 9:     d ← (spd^i + spd^{i+1}) / 2 × 1/fps                    ▷ 1/fps: frame time
10:     wp^{i+1} ← NEXT(wp^i, d, route)
11:     r̂_e ← r̂_e · (wp^{i+1}, spd^{i+1})
12:     i ← i + 1
13: return r̂_e
```

Algorithm 5 PREDICTNPC

```
Input: model M, map map, required length l, NPC vehicle's specification (start, end, mode)
Output: an individual prediction for the NPC vehicle
 1: if mode = auto then
 2:     r̂_n ← PREDICTEGO(M, map, (start, end))
 3:     return ALIGN(r̂_n, l)
 4: else if mode = immobile then
 5:     return (start, 0)^l
```

vehicle remains stationary before moving; we empirically set k to 70 for Autoware [38]. At line 4, we set i, the current frame number, to k. We enter the while-loop (lines 5–12) that iteratively extends the prediction. We skip line 6 for now and revisit it soon. We produce the feature vector that encodes the current driving status (line 7). We calculate the speed in the next frame (line 8), by predicting a speed x with the learned model and clamping x between 0 and spd^{max}. Specifically, CLAMP at line 8 bounds a predicted speed x between 0 and spd^{max}: returns 0 if $x < 0$, returns x if $0 \leq x \leq spd^{max}$, returns spd^{max} if $x > spd^{max}$. Here, spd^{max} is the predefined maximum speed of vehicles; in the implementation, we set spd^{max} to 30 km/h, which is, for rigorous testing, higher than the default value [19] (about 15 km/h) of the subject ADS [38]. At line 9, we compute a movement distance d (in meter) in frame i. At line 10, NEXT determines the next waypoint wp^{i+1} that is at a distance of d away from the current waypoint wp^i along the planned route (NEXT will be explained soon). We append the calculated frame state to $\hat{r}_e$ (line 11) and update the frame number (line 12).

The while-loop repeats until one of the two conditions is met (lines 5-6). First, the algorithm immediately returns the interim prediction result ($\hat{r}_e$) if the loop times out (line 6); in the implementation, the timeout is set to 5 seconds. Second, we exit the loop if the ego vehicle is expected to successfully get close to the end point (line 5), and return the final prediction result (line 13).

NEXT(wp^i, d, $route$) at line 10 outputs wp^{i+1} through four steps. First, we select, from $route$, two waypoints p and q that are closest to wp^i. Second, we compute an interim waypoint wp' located at a distance of d from wp^i, along the direction from p to q: $wp' = wp^i + d \times \frac{q-p}{||q-p||}$. Third, we select,

from *route*, two waypoints p' and q' that are closest to wp'. Last, we interpolate around wp' so that the final waypoint wp^{i+1} is located along the trace of *route*: $wp^{i+1} = \frac{n}{m+n} \times p' + \frac{m}{m+n} \times q'$ where $m = \text{dist}(wp', p')$ and $n = \text{dist}(wp', q')$.

Prediction for NPC Vehicle. Algorithm 5 presents the procedure PredictNPC to generate an expected driving record for an NPC vehicle. The algorithm works differently depending on the NPC vehicle's driving mode. If *mode* = auto (line 1), we reinvoke PredictEgo to obtain an interim result $\hat{r}_n$ (line 2); when Algorithm 4 is executed in this context, we use the planner of the NPC vehicle's ADS (line 1) and a potentially different value of k (lines 2–4), e.g., $k = 30$ for Behavior Agent [6]. Let us revisit Algorithm 5. At line 3, Align adjusts the length of $\hat{r}_n$ to match l: returns $\hat{r}_n[1] \ \cdots \ \hat{r}_n[l]$ if $|\hat{r}_n| > l$, returns $\hat{r}_n \cdot (wp^{|\hat{r}_n|}, 0)^{l-|\hat{r}_n|}$ if $|\hat{r}_n| < l$, returns $\hat{r}_n$ if $|\hat{r}_n| = l$. Here, $\hat{r}_n[i] = (wp^i_n, spd^i_n)$ for $i \in [1, l]$, and the notation $(a, b)^x$ stands for the sequence of x-consecutive (a, b) pairs. If *mode* = immobile (line 4), we return the prediction that expresses a vehicle stationary at the starting point (line 5).

3.2.3 Pruning with Prediction Results. Following the training dataset refinement step (line 2 in Algorithm 3), our predictor aims to forecast driving records, up to the frames where the first vehicle interactions or premature stops are expected to occur. To remove *unreliable* expected frame states (i.e., the frame states after the first vehicle interactions or premature stops) from an interim prediction result produced by Algorithm 2, we define RefineP. Based on partial prediction results deemed reliable by RefineP, we can accurately detect potentially redundant scenarios.

From Section 3.2.1, recall that: I is the set of frame indices where the vehicle interactions end, and S_e (resp., S_n) is the set of frame indices where the ego vehicle (resp., NPC vehicle) stops before reaching destinations. Let z be the frame index where the first (σ-consecutive) interactions or premature stops are expected to end *among all the vehicles*: $z = \min(I \cup S_e \cup S_n)$. Let $\hat{r} = (\hat{r}_e, \hat{r}_n)$ be the interim driving prediction for a scenario s. We define RefineP$(\hat{r})$ that outputs $p = (p_e, p_n)$:

$$p_e = \begin{cases} \hat{r}_e[1] \cdots \hat{r}_e[z] & \text{if } z \neq \bot \\ \hat{r}_e & \text{if } z = \bot \end{cases}, \quad p_n = \begin{cases} \hat{r}_n[1] \cdots \hat{r}_n[z] & \text{if } z \neq \bot \\ \hat{r}_n & \text{if } z = \bot \end{cases},$$

$\hat{r}_e[i] = (wp^i_e, spd^i_e)$, $\hat{r}_n[i] = (wp^i_n, spd^i_n)$, and $i \in [1, z]$. That is, we discard unreliable expected frame states after frame z.

Finally, we define LikelyRedundant (line 10 of Algorithm 1):

$$\text{LikelyRedundant}(\hat{r}, P) \iff \exists (s', (r', -)) \in P.\ (\text{PatSeq} \circ \text{RefineP})(\hat{r}) \sqsubseteq \text{PatSeq}(r')$$

where the order ($\sqsubseteq$) of driving pattern sequences is defined as follows: $\pi_1 \cdots \pi_l \sqsubseteq \pi'_1 \cdots \pi'_m \iff l \leq m \wedge \bigwedge_{1 \leq j \leq l} \pi_j = \pi'_j$. That is, we say s is *likely redundant* iff its predicted pattern sequence $(\pi_1 \cdots \pi_l)$ matches the pattern sequence of some previously executed scenario s', up to the reliable expected frames in $\hat{r}$ (i.e., up to the first few vehicle interactions or premature stops in $\hat{r}$).

Example 4. *Suppose $\pi = (PatSeq \circ RefineP)(\hat{r}) = \text{START} \cdot x \cdot y$. Suppose further, for some r' such that $(-, (r', -)) \in P$, we have $\pi' = PatSeq(r') = \text{START} \cdot x \cdot y \cdot x \cdot \text{END}$. Then, LikelyRedundant$(\hat{r}, P)$ holds since $\pi \sqsubseteq \pi'$.*

4 Implementation

Prunario is implemented in about 17K lines of Python code.

Extending Scenario Definition. Although Section 3 describes our approach based on simple scenarios with a single ego vehicle and a single NPC vehicle, our implementation supports more complex scenarios with richer data fields, comparable to those of state-of-the-art ADS testing tools [61, 72, 92, 97]. Figure 4 shows example scenarios supported by Prunario tool. Multiple NPC objects (vehicles and walkers; lines 8–33) can appear in a single scenario, and more detailed

```
1  {
2    "map": "Town03",
3    "spec_e": {
4   (-)"start": { "x": 7.1, "y": -5.4, "z": 0.2, "pitch": 0.0, "yaw": 90.0 },
5   (+)"start": { "x": 7.1, "y": -2.3, "z": 0.2, "pitch": 0.0, "yaw": 90.0 }, // Mutation: Path-aligned shift
6      "end": { "x": 410.0, "y": 9.8, "z": 0.2, "pitch": 0.0, "yaw": 0.9 }
7    },
8    "spec_n": {
9      "vehicles": [
10       {
11         "mode": "auto",
12         "type": "truck",
13         "start": { "x": -26.0, "y": 134.7, "z": 0.2, "pitch": 0.0, "yaw": -1.3 },
14         "end": { "x": 6.0, "y": 207.5, "z": 0.2, "pitch": 0.0, "yaw": -0.1 }
15       },
16  (+)  { // Mutation: Add a new NPC vehicle
17  (+)    "mode": "linear",
18  (+)    "type": "sedan",
19  (+)    "start": { "x": -113.0, "y": 3.7, "z": 0.2, "pitch": 0.0, "yaw": 0.0 },
20  (+)    "end": { "x": -13.0, "y": 3.7, "z": 0.2, "pitch": 0.0, "yaw": 0.0 },
21  (+)    "speed": 5.3
22  (+)  }
23     ],
24     "walkers": [
25       {
26         "mode": "immobile",
27  (-)    "type": "woman",
28  (+)    "type": "boy", // Mutation: Change the type of an existing NPC walker
29         "start": { "x": 65.5, "y": 7.8, "z": 0.2, "pitch": 0.0, "yaw": 4.0 },
30         "end": { "x": 85.5, "y": 9.2, "z": 0.2, "pitch": 0.0, "yaw": 4.0 }
31       }
32     ]
33   },
34   "puddles": [
35     { "level": 0.4, "x": 49.9, "y": -7.8, "size_x": 479.4, "size_y": 425.1 },
36  (+){ "level": 0.5, "x": 37.9, "y": 6.5, "size_x": 513.4, "size_y": 511.6 } // Mutation: Add a new puddle
37   ],
38   "weather": {
39     "cloudiness": 10.9, "precipitation": 9.0, "precipitation_deposits": 33.1,
40  (-)"wind_intensity": 31.8,
41  (+)"wind_intensity": 75.9, // Mutation: Change wind_intensity
42     "fog_density": 37.1, "wetness": 7.6, "dust_storm": 1.2
43   },
44   "time": {
45  (-)"sun_azimuth_angle": 253.5, "sun_altitude_angle": 60.0
46  (+)"sun_azimuth_angle": 253.5, "sun_altitude_angle": 173.2 // Mutation: Change sun_altitude_angle
47   }
48 }
```

Fig. 4. Example scenario and its mutant produced by PRUNARIO. (-) and (+) indicate the modifications made by the mutation operators.

attributes can be specified: types of NPC vehicles (sedan, van, truck, bicycle, motorcycle; lines 12 and 18) and NPC walkers (boy, girl, man, woman; lines 27–28), and driving environments such as puddles (friction levels of the road; lines 34–37), weather (lines 38–43), and time (lines 44–47). The extended definition also supports one additional driving mode linear^{spd} (line 17), where the NPC object moves along the shortest straight-line route from $start_n$ to end_n at a constant speed (spd).

Extending Methodology. In line with the expanded definition of scenarios, our implementation extends the mutation operators in Section 3. For example, as shown in Figure 4, we either introduce new objects (lines 16–22, and 36) or modify values of various fields (lines 5, 28, 41, and 46). We also ensure that, for the resulting mutant scenarios, the planned routes [45] between the ego vehicle and *every* NPC object overlap to increase the likelihood of interactions and violation detection.

Our implementation also extends other technical components in Section 3. For instance, INTER (Section 3.1.1) distinguishes the ego vehicle's interactions with NPC vehicles and NPC walkers. Also, while INTER (Section 3.1.1) and F_t (Section 3.1.2) rely on the time-to-collision (TTC) [69]

between the two vehicles (an ego vehicle and a single NPC vehicle), we use the minimum TTC between an ego vehicle and every NPC object in the implementation.

Preventing Nonsensical Scenarios. Prunario generates only valid scenarios, in which (1) objects move along lanes (vehicles and walkers) or sidewalks (walkers only), and (2) objects are placed on waypoints without overlapping at the start. The constraint (1) is enforced because Mutate (Section 3.1) samples new waypoints only from the valid waypoints (VW). To enforce the constraint (2), Mutate repeatedly discards mutant scenarios with overlapping starting waypoints and resamples until a scenario with no initial overlap is obtained. We empirically confirmed that Prunario does not generate any nonsensical scenarios.

Test Oracles. Prunario tool detects four kinds of critical violations commonly addressed in prior research [55, 56, 61, 63, 72, 92, 97]: collision (with NPC objects or static obstacles), stalling (stuck before reaching its destination), lane invasion (crossing lane boundaries), and speeding. We detect collision and lane invasion using violation sensors [34, 35] of CARLA [31]. We flag stalling if an ego vehicle drives slower than 1 km/h for 20 seconds. We report speeding if an ego vehicle exceeds 110% of the preset maximum speed ($spd^{\max}$ = 30 km/h).

5 Evaluation

This section aims to answer the three research questions:

- **§5.1**: How does Prunario compare to the state-of-the-art fuzzers in violation detection?
- **§5.2**: How crucial is simulation prediction for enhancing the performance of Prunario?
- **§5.3**: How effective is Prunario at discovering new bugs in industrial-strength ADS?

Test Subject ADS. We focused on testing Autoware [38], a popular ADS that has practical applications [1, 2] and has been actively maintained by Tier IV. In particular, we targeted Autoware Universe [4] (hereafter Autoware); it is a development branch of Autoware project, which is created for implementing experimental and new features to extend a stable branch (Autoware Core [3]). To run Autoware in a simulation environment, we used CARLA [31], a high-fidelity open-source simulator that supports integration with recent versions [26–30] of Autoware. For data communications between Autoware and CARLA, we used different bridge implementations [9, 10], depending on the versions of Autoware. Following the implementations [20, 21] of prior ADS testing work [72, 92] that used CARLA, we set the frame time ($1/fps$) to 0.05 seconds.

We could not test Apollo [5], another popular ADS developed from Baidu, as we were unable to reliably run its latest versions (v. 8.0 and 9.0) with CARLA; Specifically, Apollo exhibited unstable steering in nearly all executions, leading to frequent lane invasions. Similar issues have been reported in a recent paper [61].

5.1 Comparison with Existing Tools

5.1.1 Setup. We used Autoware [38] at commit 4a3de49 [26] (Dec. 2024), and CARLA v. 0.9.15.

Tools for Comparison. We selected three recent ADS testing techniques whose implementations use CARLA: ScenarioFuzz [92], DriveFuzz [72], and SAMOTA [61]. We used their latest implementations [16–18]. These three tools support detecting the four kinds of violations covered by Prunario; collision, stalling, and lane invasion can be detected by all three fuzzers, and speeding is supported by all but SAMOTA. We ran the three fuzzers with default options. We could not consider techniques [56, 64, 86] that use the LGSVL [82] simulator, as LGSVL does not support simulating the recent versions (including 4a3de49) of Autoware.

DriveFuzz [72] is a feedback-driven (similar to Section 3.1.2) fuzzer. Unlike DriveFuzz and Prunario, ScenarioFuzz [92] focuses on testing particular regions (e.g., T-shaped intersections) to

increase interactions among vehicles. SAMOTA [61] is a surrogate-model based fuzzer that predicts the fitness scores (similar to risk score in Section 3.1.2) of candidate scenarios, and based on the scores, selects scenarios likely to trigger violations.

Directly executing SAMOTA [16] on our target ADS (Autoware 4a3de49) was not possible, due to its dependency on the previous version (v. 0.9.10.1) of CARLA related to scenario-mutation strategies (removing buildings or trees in a simulation map). Thus, we modified the code for compatibility with CARLA v. 0.9.15.

Simulation Maps. We evaluated the fuzzers on four different maps supported by CARLA: Town01 [39], Town03 [40], Town04 [41], and Town05 [42]. These maps have been used to evaluate the three competing techniques [61, 72, 92]. Since Prunario does not target traffic light violations (Section 6), we configured all traffic lights in the CARLA-supported maps to stay green, so as to ensure fairness (i.e., to prevent the competing tools from detecting such violations).

Initial Seed Scenarios. To run Prunario, we randomly generated 20 violation-free scenarios without NPC objects (per map), and implemented a routine that runs Algorithm 1 for each input scenario in sequence (though only one was actually used per map, as the seed pool, W in Algorithm 1, remained non-empty). For ScenarioFuzz [92], we provided the seeds generated by itself (using map crawling [92]) at a separate time: 5 for Town01, 10 for Town03, 14 for Town04, and 20 for Town05. For DriveFuzz [72], we supplied the same seeds generated by us (i.e., 20 seeds per map), which were sufficient to keep the seed pool non-empty during the testing. For each map, we configured DriveFuzz to use the scenario employed by Prunario as the first input seed (i.e., we tried to provide opportunities to detect violations found by Prunario). We ran SAMOTA [61] without external input seeds, as it generates seeds internally.

Hardware. All experiments were conducted on two machines with four GPUs in total. One has an Intel i9-10900KF with 64GB RAM and 2 GPUs (NVIDIA GeForce RTX 2070 SUPER with 8GB VRAM and Quadro RTX 6000 with 24GB VRAM). The other has an AMD Ryzen Threadripper PRO 5975WX 32-Core Processor with 256GB RAM, and 2 RTX 4090 GPUs with 24GB VRAM for each.

Testing Budget. For each of the three fuzzers (Prunario, ScenarioFuzz [92], DriveFuzz [72]), we experimented for 8 hours, by assigning one GPU per map (i.e., an experiment for each map is done simultaneously using 4 GPUs). As an exception, we invoked SAMOTA [61] concurrently on 4 GPUs without per-map GPU assignment, since it internally switches simulation maps (Town 01, 03, 05) during a single invocation. We repeated each tool experiment 3 times to account for the randomness of the fuzzers, resulting in 24 hours of testing time and 12 invocations per fuzzer.

5.1.2 Results. The results are shown in Table 2 and 3.

Overall Violation-Detection Results. Table 2 compares the overall violation-detection capabilities. The column Sum reports the aggregated number of violations (i.e., violation-triggering scenarios) found by each fuzzer across all four maps. To account for redundancy among violation-triggering scenarios, the column Dedup provides the aggregated results over the four maps after automatically deduplicating the violations in terms of their associated driving pattern sequences (Section 3.1.1). We omit speeding detection results in Table 2, as no tools reported speeding violations in our experiments. We also note that, Table 2 shows the numbers after manually discarding false positives (i.e., violations for which the ego vehicle is not responsible). Concretely, through our careful manual investigations of all the log files generated by the fuzzers, we identified four false positive cases: (1) collision due to an NPC rushing to the ego vehicle, (2) stalling due to a stationary NPC blocking the ego vehicle (when lane changes are not possible), (3) trivial stalling occurred at the ego vehicle's starting point (considered false positives due to poor reproducibility), (4) sliding of the ego vehicle even in the absence of puddles, caused by an unresolved issue [7] in

Table 2. Violation detection results for Autoware (commit ID: 4a3de49 [26]). C, S, L: the number of collisions, stalling violations, and lane invasions detected by each fuzzer. Sum: the total number of violations across the four maps. Dedup: the number of unique violations in terms of driving pattern sequences (i.e., the number of unique pattern sequences that led to violations). $\hat{A}_{12}$: Vargha-Delaney effect size [91] comparing Prunario vs. each fuzzer. p-value: computed using the Mann-Whitney U test [78] comparing Prunario vs. each fuzzer.

Fuzzer	Town01			Town03			Town04			Town05			Aggregated		Statistical Tests	
	C	S	L	C	S	L	C	S	L	C	S	L	Sum	Dedup	$\hat{A}_{12}$	p-value
Prunario	15	11	0	12	14	6	31	1	0	17	4	0	**111**	**73**	n/a	n/a
ScenarioFuzz [92]	0	0	0	9	5	0	21	6	1	7	9	0	**58**	**23**	0.9653	0.0001
DriveFuzz [72]	1	0	0	4	0	0	0	0	0	1	1	0	**7**	**7**	1.0000	0.0000
SAMOTA [61]	0	0	0	0	0	0	n/a	n/a	n/a	0	0	0	**0**	**0**	1.0000	0.0000

CARLA simulator. To filter out these false positives, we manually inspected all log files, particularly the simulation video recordings. Cases (1)–(3) were straightforward to identify by reviewing a few seconds of video around the reported violation timestamps. For case (4), we checked whether similar sliding behavior occurred under dry-road conditions. In cases where the classification was uncertain, the initial labeling was cross-validated by the second author, and any disagreement was resolved through discussion until consensus was reached. The total number of false positives is as follows: Prunario (77), ScenarioFuzz (143), DriveFuzz (22), SAMOTA (18).

Table 2 shows that Prunario outperforms the three fuzzers in terms of violation-detection counts. For example, Prunario surpassed ScenarioFuzz, the current state-of-the-art in ADS testing, in both total violation counts (111 vs. 58) and unique violation counts (73 vs. 23). The results also show that Prunario is particularly effective in detecting collision that is arguably the most critical type of violation, consistently generating more violation-triggering scenarios across all four maps.

Statistical Analysis. For a more rigorous comparison under the randomness of the fuzzers, we conducted statistical tests [47] using effect size [91] (stochastic superiority) and p-value [78] (statistical significance), based on the per-run unique violation counts (Dedup) from all 12 runs. The effect size, denoted $\hat{A}_{12}$ [91], represents the probability that Prunario (Technique 1) finds more unique violations than a compared fuzzer (Technique 2). The p-value denotes the probability that the observed difference in unique violation counts occurs by chance. Thus, as p-value is smaller, we have a stronger evidence that Prunario surpasses the baseline tool. Following the experimental setup in SAMOTA [61], we used an adjusted significance level $\alpha = \alpha'/N$ by applying Bonferroni correction [93], where $\alpha' = 0.05$ is a typical threshold [47] and $N = 3$ is the number of comparisons. Under this setting, a difference is considered statistically significant if p-value $< \alpha$ (≈ 0.0167).

The statistical tests in Table 2 indicate that Prunario outperforms the compared fuzzers with large effect sizes ($\hat{A}_{12}$) and statistically significant differences (p-value). The effect size against SAMOTA and DriveFuzz is 1.0, indicating that a randomly chosen Prunario run always detects more violations than the baselines. The effect size against ScenarioFuzz is 0.9653, where 0.71 is considered a large effect size [61]. The p-values further confirm that the improvement of Prunario against the three fuzzers is statistically significant, consistently lower than α (≈ 0.0167).

Testing Efficiency. Table 3 compares the testing efficiency using two metrics: unique scenario ratio ($\frac{\text{UniqS}}{\text{ExecS}}$) and unique simulation-time ratio ($\frac{\text{UniqT}}{\text{ExecT}}$). For each tool, the four columns (ExecS, UniqS, ExecT, UniqT) report the numbers averaged over 12 executions (4 runs × 3 repetitions). ExecT and UniqT report the pure simulation time, excluding time spent on all other tasks (e.g., logging, waiting for the activation of Autoware's modules in each simulation).

Table 3. Comparison on testing efficiency. ExecS: # of executed scenarios. UniqS: # of unique scenarios out of the executed ones. ExecT: the wall-clock time spent on executing scenarios. UniqT: the wall-clock time spent on executing unique scenarios. All numbers averaged over 12 runs.

Fuzzer	ExecS	UniqS	$\frac{\text{UniqS}}{\text{ExecS}}$	ExecT	UniqT	$\frac{\text{UniqT}}{\text{ExecT}}$
PRUNARIO	48.7	23.4	48.1 %	256 m	115 m	44.8 %
ScenarioFuzz [92]	127.5	10.6	8.3 %	125 m	18 m	14.5 %
DRIVEFUZZ [72]	65.4	5.1	7.8 %	271 m	19 m	7.2 %
SAMOTA [61]	93.9	4.7	5 %	133 m	8 m	5.7 %

The results confirm our claim in Section 1 on the competing tools; the three existing techniques fall short of efficiently diversifying test scenarios. For example, ScenarioFuzz [92], the best competitor in Table 2, recorded a very low testing efficiency of 8.3% in terms of unique scenario ratio, i.e., 91.7% of the executed scenarios were redundant with respect to other tested scenarios. By contrast, PRUNARIO achieved a far better testing efficiency of 48.1%. Based on these results, we conclude that PRUNARIO's superior violation-finding ability largely lies in its competitive testing efficiency.

Simulation Prediction Overhead. The cost of simulation prediction was almost negligible compared to that of physical simulation [31]. While the average simulation time (per scenario) of PRUNARIO was 315.5 sec, the average simulation-prediction time was 6.4 sec, in addition to 6.2 sec for learning (Algorithm 3).

Simulation Prediction Accuracy. We also evaluated prediction accuracy by comparing each predicted record with its corresponding actual record, both logged during fuzzing. The predicted and actual records can have different lengths–for example, the actual record can be shorter when the simulation terminates early due to a violation–so we aligned each pair of predicted and actual records by truncating the longer record to the length of the shorter one. Across 12 runs, the average numbers of aligned data points were 60,338 for the ego vehicle and 145,130 for the NPC vehicles.

The resulting mean absolute error, computed over all aligned data points across 12 runs, was 2.1 km/h ($\approx$ 0.58 m/s) for the ego vehicle and 1.4 km/h ($\approx$ 0.39 m/s) for NPC vehicles, suggesting that the predictor yields reasonably accurate estimates.

5.2 Importance of Our Pruning

Setup. To evaluate the significance of simulation prediction-based pruning (Section 3.2), we developed two variants of PRUNARIO: Basic and Field. Basic refers to the basic algorithm (Section 3.1) without static pruning. Field extends Basic by performing static pruning based on scenario-field similarity, as in prior work [75, 86, 97]. We implemented Field by redefining LikelyRedundant (line 10 in Algorithm 1) as follows:

$$\mathsf{LikelyRedundant}(s, P) \iff \exists (s', (-,-)) \in P.\ \frac{|\{f \in \mathsf{F}(s) \cup \mathsf{F}(s') \mid \mathsf{Similar}(f, s, s')\}|}{|\mathsf{F}(s) \cup \mathsf{F}(s')|} \geq th_1$$

Intuitively, s is considered likely redundant iff there exists a previously executed scenario s' with a large number of similar field-values. Here, $\mathsf{F}(s)$ and $\mathsf{F}(s')$ return the sets of all flattened field names in s and s': for example, $\mathsf{F}(s)$ includes `spec_n.vehicle1.mode` that represents the field at line 11 in Figure 4. Let us write v and v' to denote the values for the field f in s and s'. $\mathsf{Similar}(f, s, s')$

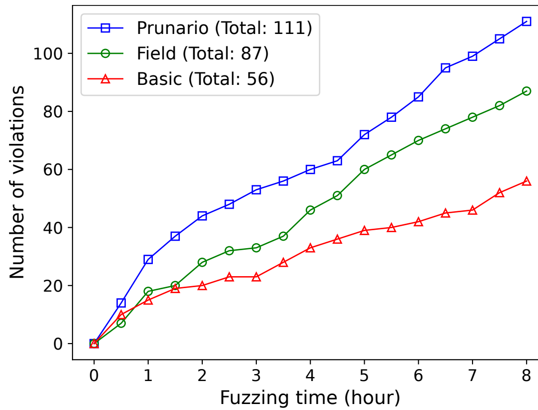


Fig. 5. Comparison on violation detection of PRUNARIO and its variants.

Table 4. Comparison on testing efficiency of PRUNARIO and its variants (same column definitions as in Table 3).

Fuzzer	ExecS	UniqS	$\frac{\text{UniqS}}{\text{ExecS}}$	ExecT	UniqT	$\frac{\text{UniqT}}{\text{ExecT}}$
PRUNARIO	48.7	23.4	48.1 %	256 m	115 m	44.8 %
Field	45.3	19.0	41.9 %	245 m	98 m	39.8 %
Basic	57.4	18.2	31.6 %	301 m	90 m	30.0 %

computes the similarity by comparing the values for the field f in each scenario:

$$\text{Similar}(f, s, s') = \begin{cases} \textit{false} & \text{if } v = \perp \text{ or } v' = \perp \\ \frac{|v - v'|}{\text{Range}(f)} \leq th_2 & \text{if } v \text{ and } v' \text{ are real numbers} \\ v = v' & \text{otherwise} \end{cases}$$

where Range(f) is a field-specific normalization factor: for instance, if f = `spec_e.start.x` (i.e., the x-coordinate of the ego vehicle's starting point), Range(f) denotes the size of the map along the x-axis. Following AutoFuzz [97], we set the thresholds th_1 and th_2 to 0.9 and 0.5, respectively.

We ran the two variants under the same experimental setup (Section 5.1.1) as PRUNARIO. We then created the cactus plots (Figure 5) to compare the number of detected violations (y-axis) over the cumulative testing time (x-axis), aggregated every 30 minutes across 12 runs. We excluded false positives from the violation counts as in Table 2.

Results. Figure 5 shows that our pruning is critical for enhancing the violation-finding ability of PRUNARIO. Compared to Basic, PRUNARIO detects twice as many violations (111 vs. 56). This is a notable improvement given that Basic itself is not a trivial approach; it adopts two optimizations (Section 3.1.1, 3.1.2) from prior work, and also performs mutations to increase vehicle interactions (Section 3.1, 4). By contrast, Field was far less effective, detecting only 87 violations in total. After deduplicating violations by driving-pattern sequences, Field and Basic yield 50 and 41 unique violations, respectively, whereas PRUNARIO identifies 73 unique violations (Table 2). In addition, Table 4 demonstrates the effectiveness of our pruning in improving testing efficiency.

The statistical tests further confirms that our pruning is overall beneficial. Against Basic, the effect size [91] is 0.9132 and the p-value [78] is 0.0006, which is lower than the adjusted significance level $\alpha = \frac{\alpha'}{2} (\approx 0.025)$ (Section 5.1.2). Against Field, the effect size is 0.7812 (considered large [61]),

and the p-value [78] is 0.0198 ($< \alpha$). These consistently large effect sizes and statistically significant p-values provide evidence that our pruning yields a reliable performance improvement.

Precision and Recall of Pruning. To analyze why Prunario detected more violations compared to Field, we compared their precision ($=\frac{|\text{CorrectlyPruned}|}{|\text{Pruned}|}$) and recall ($=\frac{|\text{CorrectlyPruned}|}{|\text{Redundant}|}$). Here, Redundant, Pruned and CorrectlyPruned refer to the sets of redundant, pruned, and correctly pruned (i.e., CorrectlyPruned = Redundant ∩ Pruned) scenarios. For this evaluation, we first built a ground-truth dataset from Basic's execution log, assigning each scenario a label: "redundant" or "unique". We then applied each pruning method to the dataset following the scenario execution order in Basic's run. When assessing Prunario, we averaged the results over three independent runs to account for the randomness in its prediction models (random forest regressors; Section 3.2.1).

The precision and recall are: Prunario (73.1%, 70.0%) and Field (61.3%, 59.5%). These statistics align with the results in Figure 5 and the motivating example in Section 2. Existing static pruning (Field) has lower precision than Prunario and therefore may incorrectly prune scenarios that lead to violations. Prunario mitigates this issue using simulation prediction for precise pruning.

Discussion. Recall that Prunario performs pruning based on predictions up to the first interactions or premature stops (Section 3.2.3). We found that this design choice does not significantly harm Prunario's practicality. On average, out of 164 pruned scenarios where both predictions and records involve interactions or sudden stops, only 13 (7.9%) were incorrectly pruned (i.e., the records for 13 scenarios exhibited distinct driving patterns after interactions or sudden stops).

5.3 Discovering New Bugs

Prunario was also effective at discovering previously unknown bugs in industrial ADS. During the development and evaluation of Prunario (from December 2023 to March 2025), we successfully found several new bugs (i.e., module-level malfunctions that lead to observed violations) in the following versions of Autoware: f6b14ec [30] (July 2023), eeed846 [29] (Jan. 2024), 7877192 [28] (May 2024), 4a3de49 [26] (Dec. 2024), and 75549a6 (Jan. 2025) [27].

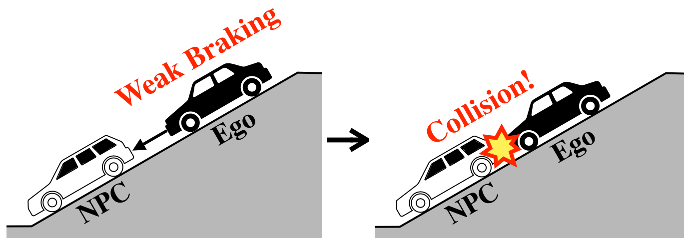
Bug-Reporting Process. To minimize the burden on the Autoware developers, we avoided reporting duplicated issues, by carefully investigating the module-level causes of violations discovered by Prunario. Following the prior work [72], we replayed the violation-triggering scenarios, and manually inspected their log files to see whether each module of Autoware emitted proper commands or accurately interpreted data from other components. After identifying faulty modules in each driving situation, we checked whether similar issues had been reported already on the GitHub repository [4], and reported issues only when we were confident that they are distinct.

Statistics on Newly Discovered Bugs. In Table 5, we summarize the newly discovered bugs. In summary, we reported 23 new bugs, of which 20 are currently confirmed by Autoware developers. 7 bugs (30%) were triggered by collision, the most serious type of violation, demonstrating the ability to detect critical errors. Furthermore, Prunario discovered bugs in the different modules, showing its advantage as a system-level testing technique over module-level approaches [81, 90, 96, 98]. Specifically, the developers identified faulty modules for 13 bugs: localization (3), perception (2), planning (4), and control (4). We introduce two bugs, each caused by a fault in a different module.

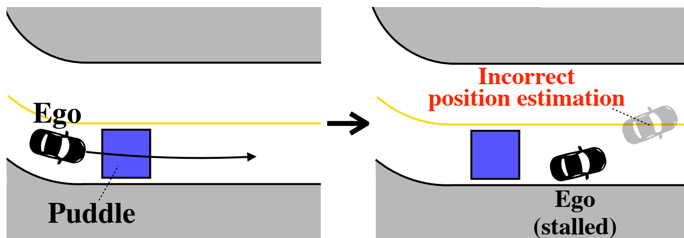
Case Study 1: Fault in Controller. As depicted in Figure 6a, the ego vehicle recognized the NPC vehicle stopped downhill and thus began to slow down. However, the ego vehicle eventually ended up colliding with the NPC vehicle. This issue was caused by a defect in the control module; since the downhill slope was steeper than the maximum slope considered in the controller, the ego vehicle did not apply the brakes sufficiently.

Table 5. The bugs (module-level malfunctions that lead to observed violations) in Autoware [26–30] found by Prunario. **Type**: violation type ("Invasion" indicates lane invasion). **ACK**: whether the bug is confirmed by developers or not (✓ if confirmed). **Module**: the name of the faulty module associated with the violation (identified by Autoware developers (1–4, 9–11, 13, 14, 16, 17, 19, 20) or determined by the authors through manual inspection of execution logs (the rest)). **Description**: the context and cause of the violation.

ID	Commit	Module	Type	Description	ACK
1	f6b14ec	Control	Collision	Produces insufficient brake on a downhill.	✓
2	eeed846	Localization	Stalling	Miscalculates the ego vehicle's position while turning.	✓
3	eeed846	Localization	Stalling	Fails to estimate the ego vehicle's position on a slippery surface.	✓
4	eeed846	Planning	Stalling	The generated trajectory overlaps the curb at a sharp corner.	✓
5	eeed846	Planning	Collision	Perceives the NPC vehicle but does not yield while crossing the intersection.	
6	eeed846	Planning	Stalling	Fails to generate a lane-change trajectory to pass a stopped NPC ahead.	✓
7	eeed846	Planning	Stalling	A submodule in the planning module crashes while turning the corner.	✓
8	eeed846	Control	Stalling	Fails to resume driving after stopping at a downhill slope.	✓
9	eeed846	Control	Collision	Over-accelerates on low-friction surface and fails to decelerate in time.	✓
10	7877192	Perception	Collision	Late detection of a small obstacle leads to failure in stopping in time.	✓
11	7877192	Planning	Stalling	Generates a straight trajectory instead of the required lane change.	✓
12	7877192	Control	Stalling	Fails to accelerate after yielding to the NPC vehicle at the intersection.	
13	4a3de49	Localization	Stalling	Halts unnecessarily due to a localization error at roundabout.	✓
14	4a3de49	Perception	Stalling	Misclassifies a non-blocking object as an obstacle ahead.	✓
15	4a3de49	Perception	Collision	Fails to consistently detect a small obstacle in front.	✓
16	4a3de49	Planning	Collision	Generates trajectory ignoring the slowly approaching NPC vehicle.	✓
17	4a3de49	Planning	Stalling	Incorrectly calculates a drivable area ahead as non-drivable at a corner.	✓
18	4a3de49	Planning	Collision	Fails to yield at the roundabout despite knowing the NPC's expected path.	✓
19	4a3de49	Control	Stalling	Underestimates required acceleration for uphill.	✓
20	4a3de49	Control	Stalling	Continues braking and fails to accelerate near the destination.	✓
21	4a3de49	Control	Stalling	Fails to accelerate despite the obstacle ahead having cleared.	✓
22	4a3de49	Control	Stalling	Erroneously keeps braking at downhill when initialized near the slope start.	
23	75549a6	Planning	Invasion	Produces an invalid backward plan instead of a valid forward route.	✓



(a) A fault in the control module of Autoware [4, 38].



(b) A fault in the localization module of Autoware [4, 38].

Fig. 6. Visualization of the two bugs found by Prunario.

Case Study 2: Fault in Localizer. As shown in Figure 6b, the ego vehicle suddenly stalled after sliding through a low-friction region, even though it was in a position to continue driving. This issue occurred due to a fault in the localization module. Specifically, the localizer malfunctioned during the sliding, leading to a discrepancy between the ego vehicle's actual position and the position estimated by the localizer. As a result, the ego vehicle's ADS incorrectly judged that the ego vehicle crossed into the opposite lane where it cannot continue driving.

6 Limitations and Future Work

We discuss future work based on the limitations of this work.

Generality Across Target ADS. Our evaluation focused on Autoware [38]. The effectiveness of PRUNARIO on other ADS, particularly Apollo [5], remains to be seen. We are continually working to reliably execute the latest version of Apollo (Section 5.1.1).

However, PRUNARIO cannot be directly applied to some ADS that do not provide explicit planning modules (e.g., DAVE-2 [53]) or that are closed-source systems. This is because PRUNARIO is a gray-box approach that, for simulation prediction, relies on internal information (i.e., the vehicles' expected trajectories; refer to line 1 of Algorithm 4) provided by the ADS's planning module [45]. By contrast, the approaches (ScenarioFuzz [92], DRIVEFUZZ [72], and SAMOTA [61]) considered in the comparison experiments can be used to test such ADS, because they are black-box methods that do not require access to internal planning information. Since the underlying assumptions regarding access to planners differ, the comparison in Section 5.1 is not a strictly fair evaluation; rather, the results should be interpreted as illustrating that simulation prediction, enabled by access to planning information, can help improve testing effectiveness.

Enhancing Simulation Prediction. We analyzed the causes of PRUNARIO's failures–averaging 209.3 cases (unsafe or missed pruning) across three repetitions from the study in Section 5.2.

(1) Slight discrepancies between predicted and actual pattern sequences in cornering sections (18%): Such deviations may result in unsafe or missed pruning.
(2) Simulation prediction errors for the presence or absence of interactions (25%): These errors can also lead to unsafe or missed pruning.
(3) Incomplete predictions due to the timeout at line 6 of Algorithm 4 (7%): These situations occur when our predictor underestimates agents' speeds, preventing full prediction generation and possibly leading to unsafe pruning.
(4) Incomplete predictions limited to the first interactions or premature stops (6%): These can also result in unsafe pruning.
(5) Nondeterministic behavior of ADS (44%): Scenarios pruned by PRUNARIO may unexpectedly exhibit unforeseen premature stops or violations even while the simulation proceeds as predicted, due to the inherent nondeterminism of ADS.

We expect that 32% of the failures, corresponding to (2) and (3), could be fixed solely by improving the speed-prediction models (e.g., mitigating autoregressive error accumulation via scheduled sampling [50]). This enhancement could increase precision and recall up to 77.5% and 82.1%, respectively, suggesting that developing more accurate models is a promising research direction.

Supporting Other Violation Types. We plan to extend our approach to support other types of violations not covered in this work. For example, to effectively detect traffic light violations, we aim to enhance simulation prediction to consider traffic lights, once the traffic light recognition issue [11] in CARLA-supported maps is resolved. Specifically, it would be possible to predict whether an ego vehicle will slow down or not before reaching a traffic light, by calculating the remaining time before it turns red.

Measuring Violation Uniqueness. We used the driving-pattern-based scenario-deduplication both in Prunario and in the evaluation to count unique violations (Dedup in Table 2). This is because, unlike scenario-deduplication approaches based on field-values [61, 97], our deduplication can identify redundancy at the behavior level during scenario execution by abstracting away superficial differences in scenario specifications (i.e., field-values). For example, while field-value-based deduplication would treat two scenarios that differ only by an additional non-interacting NPC vehicle as distinct, our approach considers one scenario redundant with respect to the other if the ego vehicle exhibits equivalent driving patterns.

However, since scenario-diversity can be defined in different ways, the unique violation counts in Table 2 should be interpreted under our behavior-level diversity, rather than as a comprehensive comparison across all possible diversity definitions. Defining the notion of violation uniqueness remains an open challenge and an interesting direction for future work.

7 Related Work

Testing ADS. Prunario is distinctive over existing ADS testing approaches [48, 49, 51, 55, 56, 61–64, 72–75, 80, 86–89, 92, 97, 99], in that it proposes a novel technique called simulation prediction, which statically analyzes scenarios' runtime behavior to efficiently detect and prune redundant scenarios. Several existing techniques attempted to reduce redundant scenarios, either through dynamic execution [56, 62, 73, 89] or simple static approaches based on field-similarity [75, 86, 97]. However, as discussed in Section 2.1, they are unable to remove initial redundant scenarios [56, 62, 73, 89] or prone to incorrect pruning [75, 86, 97]. The main goal of this paper is to address these limitations with a novel simulation predictor.

There are also other useful testing techniques for ADS, which are complementary to Prunario. For example, on top of prior work for prioritizing scenarios likely to trigger violations [48, 49, 51, 55, 61, 63, 64, 72, 74, 92, 99], we can design a more sophisticated feedback function (Section 3.1.2) to further enhance the bug-finding ability of Prunario. As other examples, liability analysis techniques [77], causality analysis technique [83], root cause analysis technique [58], and test reduction techniques [52, 57] can help reduce our manual effort required for false-positive elimination (Section 5.1.2) and bug-reporting (Section 5.3), respectively.

Testing Other Systems. Fuzzing has proven to be an effective method for validating the safety of complex systems, including OS kernels [59, 65], cloud systems [60, 85], database management systems (DBMS) [67, 68], compilers [66, 95], and SMT solvers [70, 84, 94]. Although these techniques have been effective for their respective systems, they are not directly applicable to validating ADS. For example, Jiang and Su [68] propose a technique for generating SQL queries to test DBMS, which are not suitable inputs of ADS testing.

8 Conclusion

Improving testing efficiency remains a significant challenge in ADS testing due to the high cost of executing test scenarios in physical simulators. To address this challenge, we presented Prunario, a novel technique to accelerate ADS testing by statically predicting scenario's runtime behavior without invoking physical simulators. Leveraging this *simulation prediction* capability, Prunario effectively prunes scenarios likely to produce redundant driving behaviors. We demonstrated its effectiveness through comprehensive evaluations: Prunario uncovered 23 new bugs in Autoware, an industrial-strength ADS, and outperformed state-of-the-art ADS testing tools.

Acknowledgments

This work was partly supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2020-0-01337,(SW STAR LAB) Research on Highly-Practical Automated Software Repair, No.RS-2024-00440780, Development of Automated SBOM and VEX Verification Technologies for Securing Software Supply Chains, 30%) and by ICT Creative Consilience Program through the Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (IITP-2026-RS-2020-II201819, 5%). This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT)(No. 2021R1A5A1021944, 30%).

Data Availability

The source code of Prunario and the package for reproducing our experimental results are available on both Zenodo [71] and GitHub [15].

References

[1] [n. d.]. A demo video for Autoware's autonomous valet parking. https://www.youtube.com/watch?v=MC7n8vwiLcg. Accessed: October 2025.

[2] [n. d.]. A demo video for Autoware's cargo delivery. https://www.youtube.com/watch?v=JQArZ-z9Zjc. Accessed: October 2025.

[3] [n. d.]. A GitHub repository for Autoware Core. https://github.com/autowarefoundation/autoware_core. Accessed: October 2025.

[4] [n. d.]. A GitHub repository for Autoware Universe. https://github.com/autowarefoundation/autoware_universe. Accessed: October 2025.

[5] [n. d.]. A GitHub repository of Apollo. https://github.com/ApolloAuto/apollo. Accessed: October 2025.

[6] [n. d.]. A URL for the Behavior Agent. https://carla.readthedocs.io/en/0.9.15/adv_agents/. Accessed: October 2025.

[7] [n. d.]. An issue report for the error in CARLA simulator, which leads to false positives. https://github.com/carla-simulator/carla/issues/5693. Accessed: October 2025.

[8] [n. d.]. An URL for Autopilot. https://www.tesla.com/autopilot. Accessed: October 2025.

[9] [n. d.]. An URL for bridge implementation for Autoware 7877192, 4a3de49 and 75549a6. https://github.com/autowarefoundation/autoware_universe/tree/main/simulator/autoware_carla_interface. Accessed: October 2025.

[10] [n. d.]. An URL for bridge implementation for Autoware f6b14ec and eeed846. https://github.com/evshary/autoware_carla_launch. Accessed: October 2025.

[11] [n. d.]. An URL for descrbing the issue related to traffic light recognition of Autoware. https://github.com/autowarefoundation/autoware_universe/tree/0.39.0/simulator/autoware_carla_interface#known-issues-and-future-works. Accessed: October 2025.

[12] [n. d.]. An URL for Random Forest Regressor from scikit-learn library. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html. Accessed: October 2025.

[13] [n. d.]. An URL for Stanley. https://cs.stanford.edu/group/roadrunner//old/index.html. Accessed: October 2025.

[14] [n. d.]. An URL for `carla.BoundingBox`. https://carla.readthedocs.io/en/0.9.15/python_api/#carlaboundingbox. Accessed: October 2025.

[15] [n. d.]. An URL for the artifact of Prunario (GitHub). https://github.com/kupl/Prunario-public. Accessed: February 2026.

[16] [n. d.]. An URL for the artifact of SAMOTA. https://figshare.com/articles/journal_contribution/Replication_Package_For_Efficient_Online_Testing_for_DNN-based_Systems_using_Surrogate-Assisted_and_Many-Objective_Optimization_/16468530?file=34050833. Accessed: October 2025.

[17] [n. d.]. An URL for the artifact of ScenarioFuzz. https://github.com/AtongWang/ScenarioFuzz/tree/49360794846786b133799a62827528bc4644bc62. Accessed: October 2025.

[18] [n. d.]. An URL for the artifact of DriveFuzz. https://gitlab.com/s3lab-code/public/drivefuzz/-/tree/ae08cc66d6fe0f8bc67ff2d3de55ab3f26809b59. Accessed: October 2025.

[19] [n. d.]. An URL for the default max speed of Autoware. https://github.com/autowarefoundation/autoware_launch/blob/c445de532fc373aadb43cdb37d92f7109073320a/autoware_launch/config/planning/scenario_planning/common/common.param.yaml#L3. Accessed: October 2025.

[20] [n. d.]. An URL for the frame time of ScenarioFuzz. https://github.com/AtongWang/ScenarioFuzz/blob/49360794846786b133799a62827528bc4644bc62/src/constants.py#L69. Accessed: October 2025.

[21] [n. d.]. An URL for the frame time of DriveFuzz. https://gitlab.com/s3lab-code/public/drivefuzz/-/blame/master/src/constants.py?ref_type=heads#L35. Accessed: October 2025.
[22] [n. d.]. An URL for the method `distance` in CARLA simulator. https://carla.readthedocs.io/en/0.9.15/python_api/#carla.Location.distance. Accessed: October 2025.
[23] [n. d.]. An URL for the method `get_spawn_points` in CARLA simulator. https://carla.readthedocs.io/en/0.9.15/python_api/#carla.Map.get_spawn_points. Accessed: October 2025.
[24] [n. d.]. An URL for the method `get_waypoint` in CARLA simulator. https://carla.readthedocs.io/en/0.9.15/python_api/#carla.Map.get_waypoint. Accessed: October 2025.
[25] [n. d.]. An URL for Waymo Driver. https://waymo.com/waymo-driver/. Accessed: October 2025.
[26] [n. d.]. autoware.universe 4a3de49. https://github.com/autowarefoundation/autoware.universe/tree/4a3de4915116b5d0e3db50f1976e9dd52edc0e20. Accessed: October 2025.
[27] [n. d.]. autoware.universe 75549a6. https://github.com/autowarefoundation/autoware.universe/tree/75549a6e8b395d36dcc367ba052559bd40832280. Accessed: October 2025.
[28] [n. d.]. autoware.universe 7877192. https://github.com/autowarefoundation/autoware.universe/tree/7877192cb5700c1db15109dcd959785fc381951d. Accessed: October 2025.
[29] [n. d.]. autoware.universe eeed846. https://github.com/autowarefoundation/autoware.universe/tree/eeed8466ab33d316c9e0c2761ccec39f90469e64. Accessed: October 2025.
[30] [n. d.]. autoware.universe f6b14ec. https://github.com/autowarefoundation/autoware.universe/tree/f6b14ecce6c6aabe962a173e6d82afe2333fc0fa. Accessed: October 2025.
[31] [n. d.]. CARLA: Open-source simulator for autonomous driving research. https://carla.org. Accessed: October 2025.
[32] [n. d.]. History of self-driving cars. https://en.wikipedia.org/wiki/History_of_self-driving_cars. Accessed: October 2025.
[33] [n. d.]. How GM's Cruise robotaxi tech failures led it to drag pedestrian 20 feet. https://www.reuters.com/business/autos-transportation/how-gms-cruise-robotaxi-tech-failures-led-it-drag-pedestrian-20-feet-2024-01-26/. Accessed: October 2025.
[34] [n. d.]. Sensors Reference - Collision Detector. https://carla.readthedocs.io/en/0.9.15/ref_sensors/#collision-detector. Accessed: October 2025.
[35] [n. d.]. Sensors Reference - Lane Invasion. https://carla.readthedocs.io/en/0.9.15/ref_sensors/#lane-invasion-detector. Accessed: October 2025.
[36] [n. d.]. Tesla Autopilot feature was involved in 13 fatal crashes, US regulator says. https://www.theguardian.com/technology/2024/apr/26/tesla-autopilot-fatal-crash. Accessed: October 2025.
[37] [n. d.]. Tesla driver dies in first fatal autonomous car crash in US. https://www.newscientist.com/article/2095740-tesla-driver-dies-in-first-fatal-autonomous-car-crash-in-us/. Accessed: October 2025.
[38] [n. d.]. The Autoware Foundation. https://autoware.org. Accessed: October 2025.
[39] [n. d.]. Town 1. https://carla.readthedocs.io/en/0.9.15/map_town01/. Accessed: October 2025.
[40] [n. d.]. Town 3. https://carla.readthedocs.io/en/0.9.15/map_town03/. Accessed: October 2025.
[41] [n. d.]. Town 4. https://carla.readthedocs.io/en/0.9.15/map_town04/. Accessed: October 2025.
[42] [n. d.]. Town 5. https://carla.readthedocs.io/en/0.9.15/map_town05/. Accessed: October 2025.
[43] [n. d.]. Uber self-driving car that killed pedestrian had software flaws. https://www.rac.co.uk/drive/news/motoring-news/uber-self-driving-car-that-killed-pedestrian-had-software-flaws/. Accessed: October 2025.
[44] [n. d.]. An URL for the attribute `lane_width` in CARLA simulator. https://carla.readthedocs.io/en/0.9.15/python_api/#carla.Waypoint.lane_width. Accessed: October 2025.
[45] [n. d.]. An URL for the planning module of Autoware. https://autowarefoundation.github.io/autoware.universe/main/planning/behavior_path_planner/autoware_behavior_path_planner/. Accessed: October 2025.
[46] [n. d.]. Waymo robotaxi accident with San Francisco cyclist draws regulatory review. https://www.reuters.com/world/us/driverless-waymo-car-hits-cyclist-san-francisco-causes-minor-scratches-2024-02-07/. Accessed: October 2025.
[47] Andrea Arcuri and Lionel Briand. 2014. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219–250. doi:10.1002/stvr.1486
[48] Halil Beglerovic, Michael Stolz, and Martin Horn. 2017. Testing of autonomous vehicles using surrogate models and stochastic optimization. In *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 1–6. doi:10.1109/itsc.2017.8317768
[49] 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*. 63–74. doi:10.1145/2970276.2970311
[50] Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. 2015. Scheduled sampling for sequence prediction with recurrent neural networks. *Advances in neural information processing systems* 28 (2015). doi:10.48550/arXiv.1506.03099

[51] Matteo Biagiola and Paolo Tonella. 2024. Testing of deep reinforcement learning agents with surrogate models. *ACM Transactions on Software Engineering and Methodology* 33, 3 (2024), 1–33. doi:10.1145/3631970

[52] Christian Birchler, Sajad Khatiri, Pouria Derakhshanfar, Sebastiano Panichella, and Annibale Panichella. 2023. Single and multi-objective test cases prioritization for self-driving cars in virtual environments. *ACM Transactions on Software Engineering and Methodology* 32, 2 (2023), 1–30. doi:10.1145/3533818

[53] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. 2016. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316* (2016). doi:10.48550/arXiv.1604.07316

[54] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. 2013. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 108–122. doi:10.48550/arXiv.1309.0238

[55] Yuntianyi Chen, Yuqi Huai, Shilong Li, Changnam Hong, and Joshua Garcia. 2024. Misconfiguration Software Testing for Failure Emergence in Autonomous Driving Systems. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1913–1936. doi:10.1145/3660792

[56] 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*. 488–500. doi:10.1145/3597926.3598072

[57] Yao Deng, Xi Zheng, Mengshi Zhang, Guannan Lou, and Tianyi Zhang. 2022. Scenario-based test reduction and prioritization for multi-module autonomous driving systems. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 82–93. doi:10.1145/3540250.3549152

[58] Shiwei Feng, Yapeng Ye, Qingkai Shi, Zhiyuan Cheng, Xiangzhe Xu, Siyuan Cheng, Hongjun Choi, and Xiangyu Zhang. 2024. ROCAS: Root Cause Analysis of Autonomous Driving Accidents via Cyber-Physical Co-mutation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1620–1632. doi:10.1145/3691620.3695530

[59] Sishuai Gong, Dinglan Peng, Deniz Altınbüken, Pedro Fonseca, and Petros Maniatis. 2023. Snowcat: Efficient kernel concurrency testing using a learned coverage predictor. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 35–51. doi:10.1145/3600006.3613148

[60] Jiawei Tyler Gu, Xudong Sun, Wentao Zhang, Yuxuan Jiang, Chen Wang, Mandana Vaziri, Owolabi Legunsen, and Tianyin Xu. 2023. Acto: Automatic end-to-end testing for operation correctness of cloud system management. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 96–112. doi:10.1145/3600006.3613161

[61] Fitash Ul Haq, Donghwan Shin, and Lionel Briand. 2022. Efficient online testing for DNN-enabled systems using surrogate-assisted and many-objective optimization. In *Proceedings of the 44th international conference on software engineering*. 811–822. doi:10.1145/3510003.3510188

[62] Zhisheng Hu, Shengjian Guo, Zhenyu Zhong, and Kang Li. 2021. Coverage-based scene fuzzing for virtual autonomous driving testing. *arXiv preprint arXiv:2106.00873* (2021). doi:10.48550/arXiv.2106.00873

[63] 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). doi:10.1109/tse.2023.3309610

[64] 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 *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2591–2603. doi:10.1109/icse48619.2023.00216

[65] Dae R Jeong, Yewon Choi, Byoungyoung Lee, Insik Shin, and Youngjin Kwon. 2024. OZZ: Identifying Kernel Out-of-Order Concurrency Bugs with In-Vivo Memory Access Reordering. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. 229–248. doi:10.1145/3694715.3695944

[66] Haoxiang Jia, Ming Wen, Zifan Xie, Xiaochen Guo, Rongxin Wu, Maolin Sun, Kang Chen, and Hai Jin. 2023. Detecting JVM JIT Compiler Bugs via Exploring Two-Dimensional Input Spaces. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 43–55. doi:10.1109/ICSE48619.2023.00016

[67] Zu-Ming Jiang, Si Liu, Manuel Rigger, and Zhendong Su. 2023. Detecting transactional bugs in database engines via {graph-based} oracle construction. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 397–417.

[68] Zu-Ming Jiang and Zhendong Su. 2024. Detecting logic bugs in database engines via equivalent expression transformation. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. 821–835. doi:10.5555/3691938.3691982

[69] Yiru Jiao. 2023. A fast calculation of two-dimensional Time-to-Collision. https://github.com/Yiru-Jiao/Two-Dimensional-Time-To-Collision. Accessed: October 2025.

[70] Jongwook Kim, Sunbeom So, and Hakjoo Oh. 2023. Diver: Oracle-Guided SMT Solver Testing with Unrestricted Random Mutations. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2224–2236. doi:10.1109/ICSE48619.2023.00187

[71] Minsu Kim. 2026. Prunario: Testing Autonomous Driving Systems by Pruning Likely Redundant Scenarios. doi:10.5281/zenodo.18810329

[72] 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*. 1753–1767. doi:10.1145/3548606.3560558

[73] Changwen Li, Chih-Hong Cheng, Tiantian Sun, Yuhang Chen, and Rongjie Yan. 2022. ComOpT: Combination and optimization for testing autonomous driving systems. In *2022 International Conference on Robotics and Automation (ICRA)*. IEEE, 7738–7744. doi:10.1109/icra46639.2022.9811794

[74] Guanpeng Li, Yiran Li, Saurabh Jha, Timothy Tsai, Michael Sullivan, Siva Kumar Sastry Hari, Zbigniew Kalbarczyk, and Ravishankar Iyer. 2020. Av-fuzzer: Finding safety violations in autonomous driving systems. In *2020 IEEE 31st international symposium on software reliability engineering (ISSRE)*. IEEE, 25–36. doi:10.1109/issre5003.2020.00012

[75] Lehang Li, Haokuan Wu, Botao Yao, Tianyu He, Shuohan Huang, and Chuanyi Liu. 2024. First-principles Based 3D Virtual Simulation Testing for Discovering SOTIF Corner Cases of Autonomous Driving. *arXiv preprint arXiv:2401.11876* (2024). doi:10.48550/arXiv.2401.11876

[76] Guannan Lou, Yao Deng, Xi Zheng, Mengshi Zhang, and Tianyi Zhang. 2022. Testing of autonomous driving systems: where are we and where should we go?. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 31–43. doi:10.1145/3540250.3549111

[77] You Lu, Yifan Tian, Yuyang Bi, Bihuan Chen, and Xin Peng. 2024. DiaVio: LLM-Empowered Diagnosis of Safety Violations in ADS Simulation Testing. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 376–388. doi:10.1145/3650212.3652135

[78] Henry B Mann and Donald R Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* (1947), 50–60. doi:10.1214/aoms/1177730491

[79] Michael Montemerlo, Jan Becker, Suhrid Bhat, Hendrik Dahlkamp, Dmitri Dolgov, Scott Ettinger, Dirk Haehnel, Tim Hilden, Gabe Hoffmann, Burkhard Huhnke, Doug Johnston, Stefan Klumpp, Dirk Langer, Anthony Levandowski, Jesse Levinson, Julien Marcil, David Orenstein, Johannes Paefgen, Isaac Penny, Anna Petrovskaya, Mike Pflueger, Ganymed Stanek, David Stavens, Antone Vogt, and Sebastian Thrun. 2008. Junior: The Stanford entry in the Urban Challenge. *J. Field Robot.* 25, 9 (Sept. 2008), 569–597.

[80] Vuong Nguyen, Stefan Huber, and Alessio Gambi. 2021. Salvo: Automated generation of diversified tests for self-driving cars from existing maps. In *2021 IEEE international conference on artificial intelligence testing (AITest)*. IEEE, 128–135. doi:10.1109/aitest52744.2021.00033

[81] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*. 1–18. doi:10.1145/3132747.3132785

[82] Guodong Rong, Byung Hyun Shin, Hadi Tabatabaee, Qiang Lu, Steve Lemke, Mārtiņš Možeiko, Eric Boise, Geehoon Uhm, Mark Gerow, Shalin Mehta, et al. 2020. Lgsvl simulator: A high fidelity simulator for autonomous driving. In *2020 IEEE 23rd International conference on intelligent transportation systems (ITSC)*. IEEE, 1–6. doi:10.48550/arXiv.2005.03778

[83] Huijia Sun, Christopher M Poskitt, Yang Sun, Jun Sun, and Yuqi Chen. 2024. ACAV: a framework for automatic causality analysis in autonomous vehicle accident recordings. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3639175

[84] Maolin Sun, Yibiao Yang, Ming Wen, Yongcong Wang, Yuming Zhou, and Hai Jin. 2023. Validating SMT Solvers via Skeleton Enumeration Empowered by Historical Bug-Triggering Inputs. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 69–81. doi:10.1109/ICSE48619.2023.00018

[85] Xudong Sun, Wenqing Luo, Jiawei Tyler Gu, Aishwarya Ganesan, Ramnatthan Alagappan, Michael Gasch, Lalith Suresh, and Tianyin Xu. 2022. Automatic reliability testing for cluster management controllers. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 143–159.

[86] Shuncheng Tang, Zhenya Zhang, Jixiang Zhou, Lei Lei, Yuan Zhou, and Yinxing Xue. 2024. LeGEND: A Top-Down Approach to Scenario Generation of Autonomous Driving Systems Assisted by Large Language Models. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1497–1508. doi:10.1145/3691620.3695520

[87] Yun Tang, Yuan Zhou, Yang Liu, Jun Sun, and Gang Wang. 2021. Collision avoidance testing for autonomous driving systems on complete maps. In *2021 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 179–185. doi:10.1109/iv48863.2021.9575536

[88] Yun Tang, Yuan Zhou, Tianwei Zhang, Fenghua Wu, Yang Liu, and Gang Wang. 2021. Systematic testing of autonomous driving systems using map topology-based scenario classification. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1342–1346. doi:10.1109/ase51524.2021.9678735

[89] 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. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 94–106. doi:10.1145/3540250.3549100

[90] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*. 303–314. doi:10.1145/3180155.3180220

[91] András Vargha and Harold D Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132. doi:10.2307/1165329

[92] Tong Wang, Taotao Gu, Huan Deng, Hu Li, Xiaohui Kuang, and Gang Zhao. 2024. Dance of the ADS: Orchestrating Failures through Historically-Informed Scenario Fuzzing. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1086–1098. doi:10.1145/3650212.3680344

[93] Eric W Weisstein. 2004. Bonferroni correction. *https://mathworld. wolfram. com/* (2004).

[94] Dominik Winterer and Zhendong Su. 2024. Validating SMT Solvers for Correctness and Performance via Grammar-Based Enumeration. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 355 (Oct. 2024), 24 pages. doi:10.1145/3689795

[95] Mingyuan Wu, Minghai Lu, Heming Cui, Junjie Chen, Yuqun Zhang, and Lingming Zhang. 2023. JITfuzz: Coverage-Guided Fuzzing for JVM Just-in-Time Compilers. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 56–68. doi:10.1109/ICSE48619.2023.00017

[96] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*. 132–142. doi:10.1145/3238147.3238187

[97] Ziyuan Zhong, Gail Kaiser, and Baishakhi Ray. 2022. Neural network guided evolutionary fuzzing for finding traffic violations of autonomous vehicles. *IEEE Transactions on Software Engineering* (2022). doi:10.1109/tse.2022.3195640

[98] Husheng Zhou, Wei Li, Zelun Kong, Junfeng Guo, Yuqun Zhang, Bei Yu, Lingming Zhang, and Cong Liu. 2020. Deepbillboard: Systematic physical-world testing of autonomous driving systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 347–358. doi:10.1145/3377811.3380422

[99] Tahereh Zohdinasab, Vincenzo Riccio, and Paolo Tonella. 2024. Focused test generation for autonomous driving systems. *ACM Transactions on Software Engineering and Methodology* 33, 6 (2024), 1–32. doi:10.1145/3664605

Received 2025-10-10; accepted 2026-02-17