DeFT: Maintaining Determinism and Extracting Module Tests for Autonomous Driving Planning

Yuqi Huai
University of California, Irvine
Irvine, CA, USA
yhuai@uci.edu

Yuntianyi Chen
University of California, Irvine
Irvine, CA, USA
yhuai@uci.edu

Ziwen Wan
University of California, Irvine
Irvine, CA, USA
yhuai@uci.edu

Qi Alfred Chen
University of California, Irvine
Irvine, CA, USA
yhuai@uci.edu

Joshua Garcia
University of California, Irvine
Irvine, CA, USA
yhuai@uci.edu

ABSTRACT

An Autonomous Driving System (ADS) is a complex software system often composed of multiple modules, each responsible for its own set of tasks. The ADS planning module is responsible for planning the autonomous vehicle's future driving trajectories and has historically been the most buggy ADS module. In recent years, many approaches have been proposed to test an ADS in complex virtual scenarios through simulation, and these scenarios have been effective in revealing the ADS's suboptimal decisions. However, due to the randomness of events that occur during the real-time execution of an ADS, test scenarios tend to produce varying outcomes and, in turn, make ADS testing non-deterministic, flaky, and unpredictable. To address this challenge, we propose and evaluate DeFT, an approach that extracts deterministic test cases for the ADS planning module from non-deterministic system-level scenario tests. DeFT monitors the messages exchanged by ADS modules during the execution of system-level scenario tests and reconstructs inputs to reproduce the planning module's execution. By using DeFT, we find that planning module tests can (1) more accurately reproduce planning module executions that occurred during system-level scenario tests, (2) be used to deterministically detect the same 658 collision failures revealed by system-level scenario tests, and (3) reduce the time needed to reproduce failures by 43.69% to 77.64%.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Autonomous Driving Systems, Code Coverage, Software Testing

ACM Reference Format:
Yuqi Huai, Yuntianyi Chen, Ziwen Wan, Qi Alfred Chen, and Joshua Garcia. 2026. DeFT: Maintaining Determinism and Extracting Module Tests for Autonomous Driving Planning. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3744916.3773252

ICSE '26, April 12–18, 2026, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3773252

1 INTRODUCTION

Autonomous vehicles (AVs) have gained popularity in recent years, as demonstrated by more than 50 companies developing autonomous driving systems (ADSes) [1, 2]. Examples include Tesla's Autopilot, which aims to achieve full self-driving capability [3]; Waymo, formerly the Google self-driving car project, becoming the first in the world to provide an autonomous ride-hailing service [4] and expanded its service to Los Angeles as of 2024 [5]; and Torc, a self-driving tech company focusing on autonomous trucks [6]. Self-driving cars are expected to bring many benefits: Safety may be improved because 94% of accidents are caused by human error (e.g., due to distraction or intoxication); for those who cannot drive due to age, disability, or lack of experience, self-driving cars can be a convenient traveling method [7]. Despite well-known companies making advances in autonomous driving and the potential benefits that we may have, the technology itself is not as safe as it should be at the current stage. For example, in 2018, a pedestrian died in a crash that involved a self-driving vehicle from Uber [8]; in 2023, a pedestrian was seriously injured after being hit by a human driver and then trapped under an autonomous car operated by GM Cruise [9].

Field-operational tests have been a common method to ensure the safety and quality of ADSes, typically performed by a safety driver sitting in the driver seat of an AV ready to take over in case of emergencies. However, AV companies are required to acquire the necessary permits before conducting this type of testing. In the state of California, where most AV testing occurs at the production level, only 36 companies have permits for testing with a driver [10]. To address some of the limitations of field-operational tests, virtual testing of ADSes in simulations has become a viable option, as such testing can significantly reduce costs by eliminating expenses associated with field-operational testing, including vehicle maintenance, fuel, insurance, and permit acquisition. In addition, virtual testing also allows developers to simulate diverse and potentially hazardous scenarios that are hard to replicate in the physical world without endangering human lives.

Apollo [11] is an open-source version ADS developed by Baidu, who recently deployed its commercial version in Wuhan, China [12], and many ADS testing approaches proposed in recent years

have been predominately validated using this ADS [13–20]. Those approaches focus on automatically generating test cases in the form of complex virtual scenarios, and many of them have been shown to be effective in finding challenging scenarios for the ADS. The workflow of those approaches involves specifying a scenario according to the simulator's specification and then simulating the scenario with the ADS controlling a vehicle in simulation. Based on events that occur during the simulation, different oracles are implemented to detect safety or comfort violations. However, ADSes are observed to be non-deterministic, as they take diverging paths even when repeating the same scenario [21]. While this non-determinism may not impact the key functionality of an ADS (i.e., driving a vehicle), it causes scenario tests to become inherently flaky (i.e., the same version of the ADS non-deterministically passes or fails). For instance, after a violation has been observed in a simulated scenario, there is no guarantee that re-simulating the same scenario test will yield the same violation in the future.

We studied the implementations of the state-of-the-art open-source ADSes (i.e., Apollo [11] and Autoware [22]) to identify if the implementation of ADS modules is the main source of non-determinism. We observed for both systems that the software module responsible for deciding the ADS' behavior (i.e., the planning module) is deterministic, as they produce identical output when processing the same input. However, during real-time simulation of system-level scenario tests, it becomes difficult to precisely control when and what inputs are being processed by the planning module, thus leading to variations in the planning module's outputs when repeating the same scenario test and ultimately leading to the system non-deterministically passing or failing. To obtain a deterministic set of test cases that reproduce failing planning decisions, we introduce **De**terministic **F**rame-based **T**esting (DeFT), a testing approach that divides existing system-level scenario tests into a sequence of frames where each frame corresponds to the input of a planning module test. By isolating ADS testing to a single module, we demonstrate that while scenario tests may not be able to deterministically reproduce violations, DeFT can deterministically reproduce planning decisions and leverage this ability to deterministically detect faulty planning decisions that led to system-level violations. Further, module tests extracted by DeFT can achieve substantial time savings when attempting to reproduce collision violations that are not deterministically reproducible by simply rerunning system-level scenario tests.

The main contributions of this work are as follows:

- We propose DeFT (**De**terministic **F**rame-based **T**esting), an approach that creates high-quality module-level tests for ADS planning from existing system-level scenario tests.
- We demonstrate that DeFT can extract realistic module tests that reproduce planning trajectories from existing system-level tests.
- We demonstrate that planning module tests share the same failure-finding capability as scenario tests by deterministically detecting 658 failures revealed by 8 scenario test generators.
- We provide an implementation of DeFT on a widely-used open-source ADS and a benchmark set of test scenarios exhibiting non-deterministic collisions to facilitate the reproduction of our work [23].

2 BACKGROUND

2.1 Autonomous Driving System

An autonomous driving system (ADS) is a complex software system consisting of multiple modules, each responsible for its sub-tasks and, in the end, achieving the overall goal of driving a vehicle. The Society of Automotive Engineers (SAE) defines different levels of autonomy ranging from level-0 (no driving automation) to level-5 (full driving automation) [24]. Apollo [11] and Autoware [22] are 2 open-source ADSes that aim for level-4 autonomous driving, where a human driver is not required to operate the vehicle when certain conditions are met. Some example features include local driverless taxis and traffic jam chauffeurs [25]. The state-of-the-art ADSes consist of multiple modules and utilize the publish-subscribe architecture, where each module retrieves its input from the message bus by subscribing to one or more topics and publishes its output to the message bus [26]. Each module operates asynchronously at a certain frequency and accomplishes a specific task, for example (1) **Localization** detects the location of the autonomous vehicle itself; (2) **Perception** detects traffic light statuses and road traffic participants; (3) **Prediction** predicts the movement of dynamic obstacles; (4) **Planning** computes the optimal driving trajectory; (5) **Control** realizes the planned trajectory through lateral and longitudinal control. The executions of these modules are managed through high-performance middleware (i.e., CyberRT [27], ROS [28]) during runtime [29]. In our study, we select Apollo [11] as our subject ADS because (1) it is the open-source version of a production-grade level-4 ADS developed by Baidu, (2) it is capable of performing autonomous driving functionality in various scenarios [30], (3) it is selected by Udacity to teach state-of-the-art AV technology [31], (4) it can be directly deployed on real-world AVs such as Lincoln MKZ, Lexus RX 450h, GAC GE3, and others [32], and (5) it has mass production agreements with Volvo and Ford [33]. More recently, Apollo has started serving the general public in cities (e.g., a robo-taxi service in Wuhan, China [12]).

2.2 Simulation-based Testing

Simulation-based ADS testing approaches [13–20] produce scenario tests that are executed through a simulator. During test execution, the simulator controls the actors (e.g., pedestrians, vehicles) based on the specification and generates the corresponding inputs for the ADS. After processing data generated by the simulator, the ADS produces control commands, which the simulator processes to drive the AV in the simulator. The outcome of the scenario can be recorded in different ways. When the simulator provides a data recorder (e.g., Carla recorder [34]), the user can obtain scenario-level information from the perspective of the simulator (e.g., positions of ground-truth road traffic participants and traffic light status at different timestamps). Alternatively, the user can record messages that are being exchanged on the message bus of the ADS (e.g., Apollo's `cyber_recorder` [35], Autoware's `ros2 bag` [36]) to analyze the output of each ADS module. Following test execution, test oracles analyze the driving trajectory of the ADS and determine if any traffic violations occurred (e.g., colliding with another vehicle).

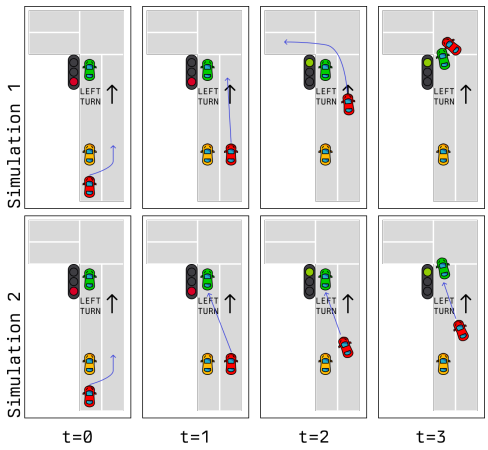

Figure 1: Visualization of two simulations of the same scenario test for the same version of the ADS. The red vehicle is the AV, the yellow vehicle is a static obstacle and the green vehicle is an intelligent obstacle waiting for the traffic signal to turn green. In Simulation 1, the AV cuts off and collides with the green vehicle; in Simulation 2, the AV follows traffic rule and drives behind the green vehicle.

2.3 Motivation

Laurent et al. [21] discussed the inherent non-determinism of ADSes through their observation of an ADS taking diverging paths in repeated simulations of the same scenario, and we observed that ADS taking diverging paths can also lead to variances in test outcomes. Figure 1 depicts 2 simulations of the same scenario test generated by a state-of-the-art ADS test generation technique [18]. In the scenario, the AV is the **red vehicle**, blocked by a static **yellow vehicle**; the **green vehicle** is a dynamic obstacle waiting for the traffic signal to turn green. The **AV** sometimes turns left from a through lane, cuts off the **green vehicle** waiting for the traffic signal to turn green, and eventually collides with that vehicle. This fault occurs non-deterministically, and the ADS avoids the collision by stopping at a far distance behind the **green vehicle** most of the time (16 times in 20 simulations). In both passing and failing simulations, the **AV** drives around the static **yellow obstacle**. However, a critical difference in behavior planning leads to a difference in the overall scenario outcomes for passing and failing simulations. In **Simulation 1**, the **AV** plans to drive side-by-side with the **green vehicle** waiting for the red light and eventually turns left from a through lane while cutting off the **green vehicle**. In **Simulation 2**, the **AV** produces a critical alternative planning trajectory at a similar position and time to follow the **green vehicle** and avoids both traffic rule violations and safety violations. We extended our pilot study to 6,158 scenarios generated by 8 testing approaches, and observed 187 scenarios that reveal collision violations have non-deterministic outcomes, as either violations disappear or new violations arise during repetitions of the same scenario test. We will further elaborate on the details in Section 5.2.

3 PROBLEM STATE SPACE

While the inherent non-determinism of ADS inevitably leads to non-determinism in executions of scenario tests, the module responsible for deciding the ADS's behavior in the state-of-the-art open-source ADS, the planning module, is deterministic (i.e., processing the same input produces the same output) [37]. However, this determinism diminishes during real-time simulation because it is hard to ensure each of the modules is executed under identical conditions, as every simulation can produce a different outcome due to different combinations of software and hardware conditions [38]. To that end, our insight into obtaining a deterministic test suite for ADS planning is to isolate testing a single module and produce module-level tests that capture and reproduce executions that occurred during system-level scenario tests. In the remainder of this section, we present a formal specification of the problem state space, which DeFT utilizes to extract planning module tests.

Definition 1. During the execution of a scenario test, individual modules of the ADS publish their outputs to the message bus at varying times. Messages that can be observed on the message bus are represented by a list $\mathbb{M} = [m_1, m_2, ...m_{|\mathbb{M}|}]$, each m_i is observed at time t_{m_i} and can be one of

- $m_{routing}$ is an output produced by the routing module, representing the start and destination location of the AV in the given scenario at time t_M.
- $m_{localization}$ is an output produced by the localization module, representing the location of the AV observed at time t_L.
- $m_{traffic\_light}$ is an output produced by the perception module, representing the traffic light status of all detected traffic lights at time t_T.
- $m_{detected\_obstacles}$ is another output that the perception module produces, representing the positions of all obstacles that have been detected at time t_D.
- m_{pred} is an output produced by the prediction module, representing the predicted trajectories of detected obstacles at time t_{PD}.
- m_{plan} is an output produced by the planning module, representing the planned future trajectory of ADS at time t_{PL}.

Definition 2. A single execution of the ADS planning module processes a frame F at time t_F, where a frame is defined as the tuple $F = \langle m_{routing}, m_{localization}, m_{traffic\_light}, m_{pred} \rangle$, representing a snapshot of messages from other modules that serve as inputs to the planning module. For each execution of the planning module that occurred during system-level testing, DeFT tries to construct the input frame F that was processed for the execution.

Definition 3. A single execution of an ADS planning module produces output m_{plan}, which we refer to as a *planning trajectory* and later denote as P, at time $t_{m_{plan}}$ where P is a sequence of path points $[p_1, p_2, ..., p_{|P|}]$ and each path point p is a tuple $\langle x, y, t \rangle$ representing the planning module plans to reach position $\langle x, y \rangle$ at timestamp t. Since the planning module is deterministic (i.e., processing the same input yields the same output), DeFT compares the output of the planning module during system-level testing and module-level testing to determine the correctness of the extracted input F.

Given all observed messages $\mathbb{M}$ during the execution of a system-level scenario test where $\mathbb{P}$ is a subset of $\mathbb{M}$ representing all outputs of the planning module (i.e., $\mathbb{P} = [P_1, P_2, ..., P_{|\mathbb{P}|}]$), the goal of DEFT is to find all input frames $\mathbb{F} = [F_1, F_2, ..., F_{|\mathbb{F}|}]$ such that the output produced by the planning module P_i' given input F_i reproduces the original output P_i.

4 APPROACH

DEFT produces module tests for ADS planning by extracting frames from system-level scenarios that can reproduce the module's executions. Elbaum et al. [39] proposed the carving technique that incorporates instrumentation code into the program to record system states or method invocations during system-level testing [40]. However, each of the ADS modules is expected to perform time-critical tasks, and such a system usually does not tolerate instrumentation-induced overhead [41]. To record necessary information that facilitates reproducing executions of the planning module without instrumenting any part of the ADS, our key insight is to leverage the design of the ADS architecture and rely on messages that are being exchanged on the ADS' message bus. This insight is based on the following two intuitions: (1) The message bus is a non-invasive communication channel that can be used to obtain the outputs of each ADS module (e.g., the simulator subscribes to the output of the control module to obtain control commands produced by the ADS). Therefore, a subscriber can be added to the message bus to record additional information without affecting the execution of other ADS modules, and the state-of-the-art ADSes typically provide such a capability [35, 36]. (2) The planning module subscribes to messages that are available on the message bus and processes a combination of them to produce a planning trajectory. As a result, we can reconstruct the necessary conditions to reproduce each execution of the planning module after a real-time simulation of a scenario test by finding the combination of messages recorded from the message bus instead of instrumenting the planning module to record the internal states and arguments for each execution.

Figure 2 shows an overview of how DEFT extracts module tests from system-level scenario tests. DEFT achieves this goal as follows: (1) the *Planning IO Filter* observes messages that are being exchanged on the message bus and identifies candidate messages that could have been processed by the planning module, since during real-time simulation, the planning module is known to subscribe to a set of topics and process messages obtained from those topics; (2) for each planning trajectory produced during real-time simulation, the *Time-Sensitive Input Search Engine* searches for (a) the time at which the planning module processed data and (b) what data was processed, and executes the planning module based on the reconstructed input to reproduce a planning trajectory; (3) the *Trajectory Validator* compares the planning trajectory produced during real-time simulation with the reproduced planning trajectory by isolating the execution of the planning module to validate the result of *Time-Sensitive Input Search Engine*.

4.1 Planning IO Filter

For each planning trajectory produced by the planning module during a system-level scenario test, the goal of the *Planning IO Filter* (PIF) is to identify candidate messages observed from the message bus that the planning module could have processed. To achieve that, PIF leverages the fact that if the planning module produced a planning trajectory P at time t_P, then the input frame for this planning execution cannot consist of messages that are published after t_P.

The planning module tracks the most recently received message from each subscribed topic and creates a snapshot of those messages (i.e., the input frame F) at a certain frequency for processing. Given a planning trajectory P produced at time t_P, the input frame F must have been created at some time $t_F = t_P - \alpha$, where α is the processing time of the planning module. Since the processing time of the planning module is lower bounded by 0 (i.e., the hypothetical case where the planning module instantly produces a result), t_F must be earlier than t_P. Considering the input frame F created at t_F can only consist of messages that have been received before t_F, messages that are published after t_P cannot be received before t_F, and they cannot be part of candidate messages for the input frame F. Figure 3 shows what PIF considers as candidate messages for the input frame of a planning trajectory observed on the message bus. PIF maps each planning trajectory with a set of candidate messages $\mathbb{M}_{cand}$, defined as follows:

$$\mathbb{M}_{cand} \rightarrow P : \{m : m \in \mathbb{M} \wedge t_m <= t_P \wedge subscribesTo(m)\} \quad (1)$$

where $\mathbb{M}$ represents all messages observed on the message bus, t_m is the time at which message m was published, and $subscribesTo(m)$ is a function that determines if the planning module subscribes to message m. For each planning trajectory P, the candidate messages for the input frame of this planning trajectory are messages from the message bus that are published before t_P.

4.2 Time-Sensitive Input Search Engine

The goal of the *Time-Sensitive Input Search Engine* (TISE) is to reconstruct inputs for the planning module to reproduce planning trajectories observed during system-level scenario tests. For each planning trajectory $P_i \in \mathcal{P}$, TISE reconstructs the input of the planning module F_i (recall Definition 2) so that the planning module processing F_i created at time t_{F_i} produces P_i' such that $EQ\_traj(P_i, P_i', \theta)$.

While the upper bound of frame creation time t_F is specified to be t_P by PIF, the first step of TISE is to specify the lower bound of t_F. The state-of-the-art ADS architecture models the execution of each module as callbacks that are triggered by events [42, 43]. Once the triggering event is received by the planning module, it creates a frame consisting of the most recently received messages and computes a planning trajectory based on the frame. Therefore, we have **Heuristic 1:** Given a planning trajectory P observed at time t_P, t_F is upper bounded by t_P, and lower bounded by the planning module's triggering event. Figure 4 shows how TISE determines the bound for t_{F_1} given a planning trajectory P_1 and a prediction message $m3$ that is expected to trigger the execution of the planning module. In this example, message $m3$ is published at $t_{trigger}$, making $(t_{trigger}, t_{P_1})$ the bounds of t_{F_1}.

Now that the range of t_{F_1} has been specified, TISE uniformly assigns a random value to t_{F_1} and begins searching for messages that should be part of the input frame F_1. Based on the value of t_F, we have **Heuristic 2:** Messages published after t_F cannot be part of the input frame F; when multiple messages of the same type are

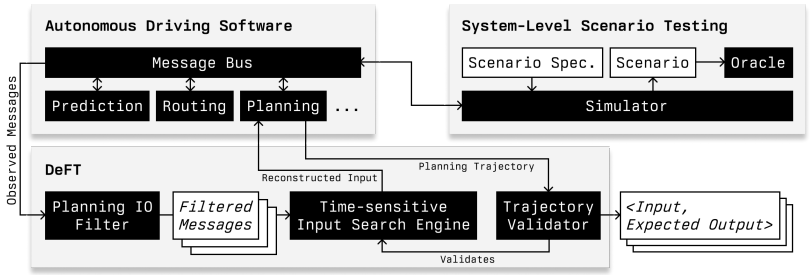

Figure 2: An overview of how DeFT extracts module-level unit tests from system-level scenario test.

published before t_F, more recently published messages are more likely to be part of F.

The planning module subscribes to messages from different topics and keeps track of the most recently received messages from each topic as part of the planning module's internal states. After the triggering event is received, the planning module creates the frame F with the most recently received message from each subscribed topic and processes this frame to produce a planning trajectory. However, due to the message passing delay in publish-subscribe architectures, it is not possible to predict how and when a subscriber receives messages [21], and, therefore, does not guarantee the planning module will always process the most recently published message on the message bus. Hence, TISE assigns a high probability to newer messages but still considers older messages with lower probabilities.

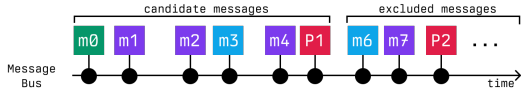

Figure 3: A visualization of messages that are observed on the message bus. P_1 is a planning trajectory produced by the planning module at time t_{P_1}, messages 0 to 4 are published earlier than t_{P_1}, and messages 6 and 7 are published later than t_{P_1}. Messages published later than t_{P_1} cannot be part of the candidate messages for P_1's input frame. Green message $m0$ is produced by the routing module, purple messages $\{m1, m2, m4, m7\}$ are produced by the localization module, blue messages $\{m3, m6\}$ are produced by the prediction module, and red messages $\{P_1, P_2\}$ are planning trajectories produced by the planning module.

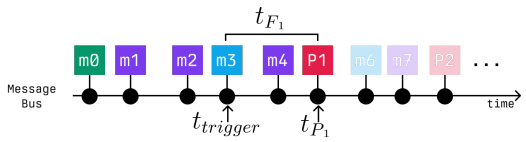

Figure 4: A visualization of messages from the message bus and how TISE determines the range of t_{F_1}. Given the planning module is subscription-triggered by receiving $m3$, which is published at $t_{trigger}$, and the output of the planning module P_1 is published at t_{P_1}, t_{F_1} is lower bounded by $t_{trigger}$ and upper bounded by t_{P_1}.

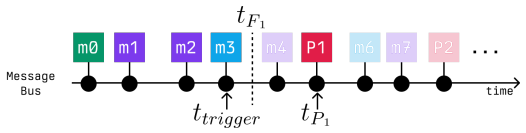

Figure 5: Visualization of messages on the message bus with respect to time t_{F_1}. After TISE estimates the frame creation time t_{F_1}, it then assigns the probability of each message being part of the input frame F_1 based on its relative location from t_{F_1}.

Figure 5 shows a visualization of messages on the message bus with respect to t_{F_1}. Although message $m4$ is published before planning trajectory P_1 was produced, $m4$ cannot be part of the input frame for P_1 because $m4$ was published after the estimated frame creation time t_{F_1}. Multiple messages $\{m1, m2\}$ from the localization module were published before t_{F_1}, and message $m2$ has a higher probability of being on the input frame because it was published at a time closest to t_{F_1}. Message $m0$ appeared the earliest on the message bus, but it must be part of the input frame F_1 because it is the only message from the routing module. Since the execution of the planning module is triggered upon receiving message $m3$, then $m3$ must be part of the input for the planning module that produced P_1.

The planning module subscribes to messages published by other modules and sequentially processes messages that become available on the message bus. Based on this domain-specific knowledge, we can have **Heuristic 3:** Each input frame cannot consist of a message that is older than the message of the same type from the previous frame. Figure 6 shows a visualization of candidate messages for a subsequent input frame. Using the previous heuristics, TISE estimates the routing message $m0$, the prediction message $m3$, and the localization message with the highest probability $m2$ should be part of the first input frame F_1. Hence, localization message $m1$ is excluded from the set of candidate messages for the second input frame F_2 because a newer localization message is expected to be already processed by the previous frame. However, the routing message $m0$ remains in the set of candidate messages for F_2 because no newer routing message has been published before t_{F_2} and the same routing message is reused for a subsequent frame.

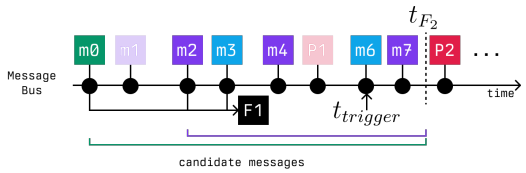

Figure 6: Visualization of candidate messages for the second input frame F_2 after TISE estimates the messages that are expected to be part of the first input frame F_1. Messages in F_2 can either be the same as the previous frame or newer for each message type.

Algorithm 1: Time-sensitive Input Search Engine

```
Input:
    P ← Planning trajectories
    CM ← Map of Candidate Messages
Result: Frames (i.e. inputs) for planning module
F_current ← Empty list;
for i = 0; i < P.length; i = i + 1 do
    f ← estimate_frame(P[i], CM[P[i]], F_current);
    F_current.append(f);
end
// Initialize ADS Planner
Planner ← Initialize Planner;
for i = 0; i < P.length; i = i + 1 do
    traj = Planner(F_current[i]);
    F_current[i].correctness =
      Trajectory_Validator(P[i], traj);
end
return F_current
```

Algorithm 1 shows the main algorithm of DEFT. After the *Planning IO Filter* identifies candidate messages processed by the planning module, the aforementioned heuristics are leveraged to estimate a list of frames $\mathbb{F} = [F_1, F_2, ..., F_{|\mathcal{P}|}]$ and DEFT executes the planning module sequentially with these frames to produce $[P'_1, P'_2, ..., P'_{|\mathcal{P}|}]$. TISE then uses *Trajectory Validator* to evaluate the correctness of each estimated frame based on the similarity between the actual planning trajectory P' is to the expected planning trajectory P.

4.3 Trajectory Validator

The *Trajectory Validator* (TV) is the final step in DEFT's pipeline, and its main function is to assess whether the frames extracted by TISE are accurate enough to be converted into module tests. Recall that each frame extracted by TISE represents the input of the planning module that was processed during the simulation of a scenario test. TV directly feeds the input frames into the planning module to obtain the reproduced outputs and compares the newly generated planning trajectories with the original planning trajectories.

Given an original planning trajectory P and a reproduced planning trajectory P', TV first interpolates P' based on timestamps in P so that two trajectories have the same number of data points, and then uses Equation 2 to compute the lock-step Euclidean distance (LSED) [44] between the two trajectories.

$$Eu(P, P') = \sqrt{\sum_{i=1}^{|P|}((x_i^P - x_i^{P'})^2 + (y_i^P - y_i^{P'})^2)} \quad (2)$$

LSED measures how similar 2 trajectories are by comparing the spatial distance between corresponding locations (e.g., the first 2 points of each trajectory, the second 2 points, etc.). Note that TV can also be configured to use other trajectory similarity measurement algorithms as appropriate. If the distance between them is less than or equal to a threshold θ_M (i.e., $Eu(P, P') \leq \theta_M$), TV considers the input frame to be accurate enough to reproduce a planning module execution and produces a corresponding planning module test. We further elaborate on how we empirically obtain appropriate values for θ_M in Section 5.1.

4.4 Module Tests Execution

In the previous subsections, we discussed how DEFT filters messages based on the input modules specified by the planning module, and reconstructs frames based on a list of heuristics to aggregate messages from different modules into inputs for the planning module. Based on our observation of how the planning module operates during system-level testing, DEFT produces N planning module tests from one run of a system test, each corresponding to a planning module execution. These tests are then executed sequentially to ensure that the internal states of the planning module are restored.

4.5 Oracle Transformation

The difference in semantics between the test output during system-level scenario testing and module-level testing makes directly applying existing scenario oracles challenging. Oracles in the context of system-level testing for ADSes detect violations based on the actual recorded behavior of the system during simulation. For example, the collision oracle checks if the AV has collided with any obstacles by comparing the AV's positions with the obstacles' positions at different time points. In contrast, when testing an ADS at the planning module level, the focus shifts to evaluating the planning trajectories, which outline the planned paths of the AV, including its intended positions and timings.

For planning module tests extracted by DEFT to detect the same failures as system tests, we propose an oracle transformation approach. Given an ADS violation was detected during a simulation of a scenario test, DEFT first determines whether the planning module is the culprit of the violation by checking if the AV's behavior followed the outputs of the planning module. The condition for DEFT to determine the planning module being faulty is defined as $\exists p \in P \in \mathbb{P} : V(p)$, where $V(p)$ evaluates whether a violation is detected when the AV reaches path point p in a planning trajectory P of the output planning modules message $\mathbb{P}$. Morever, DEFT determines if the planning module is faulty by finding a faulty planning trajectory P, later denoted as P_{faulty}, that includes path point p, where a violation is known to occur if AV actually reaches p at the planned time.

With the faulty planning trajectory identified, DEFT module tests for this scenario are considered failing if they accurately reproduce the faulty planning trajectory since the same violation is expected to occur again if the scenario test can be repeated under identical

conditions. The condition for the planning module test to fail is defined as

$$\exists F \in \mathbb{F} : RunTV(RunPlanner(F), P_{\text{faulty}}) \leq \theta_M \quad (3)$$

where F is a single frame from all frames $\mathbb{F}$ extracted by DeFT, $RunPlanner(F)$ represents the planning trajectory reproduced from running the ADS planning module on input F, and $RunTV$ represents running the trajectory validator, which evaluates whether the reproduced planning trajectory is sufficiently similar to the faulty planning trajectory P_{faulty}. When DeFT module tests can no longer reproduce the faulty planning trajectories, this state indicates that the planning module's implementation has changed to avoid the same violation if identical conditions occur when repeating the same scenario test.

5 EVALUATION

To evaluate DeFT, we implemented a prototype for Apollo [11], an open-source ADS that has been widely used in prior work [13–20]. We selected 8 generators (i.e., AV-FUZZER [13], SAMOTA [45], BehAVExplor [20], DriveFuzz [46], scenoRITA [19], DoppelTest [18], and DeepCollision [17]) that focus on finding safety-critical scearios to produce an initial system-level scenario test suite. Note that some of the approaches were originally implemented on another ADS, and some of them used an external simulator that is no longer available from its official maintainer [47]. Given those limitations, we chose the best available implementation to date [48] that ensured the aforementioned approaches generate scenarios for the same ADS using the same built-in simulator that has been recommended by the ADS' official developer [49].

As shown in Table 1, 8 approaches generated a total of 6,158 scenarios, and 658 of them are safety-critical (i.e., a collision occurred). For each generated scenario S, we simulated the test case 10 times to produce $S_0, ..., S_9$, where S_0 represents the initial execution of the scenario test. We then used DeFT to extract a set of module tests U from simulation S_0 and rerun those module tests to determine whether module tests are deterministic and capable of detecting the same failure as S_0.

Table 1: Safety-critical scenarios generated to evaluate.

Approach	Venue	Year	Repo.	No Collision	Collision
AV-FUZZER	ISSRE	2020	[50–52]	255	99
BehAVExplor	ISSTA	2023	[52, 53]	63	214
Decictor	ICSE	2025	[52]	290	11
DeepCollision	TSE	2023	[52, 54]	399	56
DoppelTest	ICSE	2023	[55]	2,824	43
DriveFuzz	CCS	2022	[52, 56]	185	115
SAMOTA	ICSE	2022	[52, 57]	71	93
scenoRITA	TSE	2023	[58]	1,413	27
Total				5,500	658

Given those generated test suites, we focus on studying the following research questions:

> **RQ$_1$ (Planning Reproducibility)** To what extent do scenario tests and module tests reproduce planning-module executions?

Re-executing scenario tests often do not precisely reproduce executions of the planning module as the simulator converts abstractions of scenarios into actual inputs for ADS modules. However, module tests precisely control what inputs are being processed by the planning module, making such tests potentially more accurate at reproducing planning-module executions. At the same time, DeFT aims for the failures that occur in a scenario when rerun at the system level to be reflected by DeFT's resulting module tests, leading us to study the following RQ:

> **RQ$_2$ (Failure Reproducibility)** To what extent can scenario tests and module tests reproduce failures detected by test generation approaches?

Prior work (e.g., from the selected scenario-generation approaches) has shown that system-level scenario testing is effective in revealing ADS failures. While DeFT's module testing aims to maintain the determinism of the planning module by reducing testing to that module, they should also reproduce failures that were revealed by system-level scenario tests as much as possible. Nevertheless, DeFT aims to reproduce planning output and failures efficiently, leading us to our final RQ:

> **RQ$_3$ (Efficiency)** What is the run-time efficiency of DeFT and the produced module tests?

DeFT aims to achieve determinism in testing ADS planning at the cost of ignoring executions of other modules. If less code of the ADS is being tested, we should expect DeFT to take less time to execute. In addition, we should also expect DeFT to produce module tests that require fewer reruns compared to system-level scenario tests when reproducing failures that are non-deterministic.

5.1 RQ1: Planning Reproducibility

In this research question, we study to what extent module tests extracted by DeFT reproduce planning trajectories that were produced during system-level scenario tests. The degree of reproduction serves as an indicator of how realistic the module tests are, reflecting their ability to capture behaviors that occur during system-level tests. To determine a reasonable similarity threshold θ_M for the trajectory validator, we analyze how closely trajectories from reruns that reproduce specific failures match those from the initial execution of the same system-level scenario.

Table 2: Summary statistics of planning trajectory edit distances across reruns of a system-level scenario test.

Measure	FR Planning Edit Distances (m)	FN Planning Edit Distances (m)
1st Quartile	0.10	1.62
2nd Quartile	0.23	17.15
3rd Quartile	6.99	308.71
min	0.0	0.0
mean	29.56	139.51
max	2,320.81	1,453.06

Table 2 shows the summary statistics of lock-step Euclidean distance (recall Section 4.3), measured in meters, between planning trajectories in reruns and the initial execution of system-level scenario tests. Lower edit distances indicate the planning trajectories from a rerun are more similar to the initial run. Since not all

Table 3: Number of planning trajectories reproduced by DEFT module tests compared to rerunning scenario tests under different thresholds θ_M (meters).

θ_M(m)	Rerun 1		Rerun 2		Rerun 3		Rerun 4		Rerun 5		Rerun 6		Rerun 7		Rerun 8		Rerun 9		Average	
	DeFT	S_1	DeFT	S_2	DeFT	S_3	DeFT	S_4	DeFT	S_5	DeFT	S_6	DeFT	S_7	DeFT	S_8	DeFT	S_9	DeFT	S
0	224,888	1,637	224,888	1,814	224,888	922	224,888	660	224,888	620	224,888	1,162	224,888	898	224,888	1,165	224,888	1,166	224,888	1,116
0.1	228,956	42,335	228,956	58,139	228,956	60,758	228,956	56,340	228,956	53,423	228,956	50,780	228,956	54,640	228,956	50,817	228,956	46,040	228,956	52,586
0.23	228,975	95,235	228,975	130,405	228,975	117,932	228,975	106,483	228,975	110,196	228,975	111,036	228,975	108,124	228,975	104,387	228,975	98,794	228,975	109,177
1.62	229,108	127,258	229,108	164,746	229,108	156,662	229,108	138,186	229,108	144,765	229,108	146,757	229,108	145,928	229,108	139,745	229,108	139,209	229,108	144,806
6.99	229,915	154,467	229,915	187,401	229,915	177,544	229,915	157,952	229,915	166,200	229,915	167,281	229,915	165,952	229,915	161,241	229,915	162,522	229,915	166,729
17.15	230,637	173,282	230,637	207,360	230,637	198,619	230,637	175,469	230,637	183,989	230,637	185,747	230,637	184,890	230,637	181,799	230,637	181,156	230,637	185,812
29.56	230,852	184,047	230,852	216,283	230,852	210,330	230,852	188,635	230,852	199,503	230,852	197,119	230,852	195,049	230,852	192,756	230,852	193,055	230,852	197,420
139.51	231,463	199,883	231,463	223,743	231,463	221,096	231,463	210,528	231,463	214,079	231,463	211,353	231,463	209,424	231,463	206,069	231,463	205,251	231,463	211,270
308.71	231,463	210,818	231,463	227,003	231,463	226,841	231,463	219,923	231,463	222,473	231,463	219,285	231,463	217,702	231,463	217,016	231,463	215,246	231,463	219,590
1453.06	231,463	231,463	231,463	231,436	231,463	231,338	231,463	231,174	231,463	231,338	231,463	231,338	231,463	231,463	231,463	231,463	231,463	231,312	231,463	231,369
2320.81	231,463	231,463	231,463	231,463	231,463	231,462	231,463	231,463	231,463	231,463	231,463	231,463	231,463	231,463	231,463	231,463	231,463	231,463	231,463	231,463

collisions were reproduced in the reruns, we distinguish between **F**ailure-**R**eproducing (FR) reruns (i.e., failure from the initial execution is reproduced in a rerun) and **F**ailure-**N**on-Reproducing (FN) reruns (i.e., failure from the initial execution is not reproduced in a rerun). We observe that planning trajectories in FR reruns generally exhibit lower edit distances across all statistical measures, with the interquartile range (IQR) spanning from 0.10 meters to 6.99 meters, indicating they are highly consistent with the planning trajectories in the initial execution. In contrast, FN reruns show higher edit distances, with the interquartile range (IQR) spanning from 1.62 meters to 308.71 meters, suggesting their planning trajectories deviate more from the initial execution. This distinction indicates that failures revealed by system-level scenario tests tend to be reproduced when planning trajectories remain similar to the original (i.e., the initial execution), and significant deviations in planning trajectories lead to non-reproduction of failures.

To assess the effectiveness of our approach in reproducing planning trajectories, we use the statistical measures from Table 2 as thresholds to determine whether one planning trajectory is sufficiently similar to another. As shown in Table 3, which summarizes the number of planning trajectories reproduced by DEFT compared to rerunning the scenario test, we observe that under a very strict threshold (i.e., $\theta_M = 0.0$ meters), DEFT can accurately reproduce 123.97 to 362.72 times more planning trajectories compared to rerunning the scenario test. At a more moderate threshold of $\theta_M = 6.99$ meters, where 75% of planning trajectories from FR reruns reproduced planning trajectories in the initial execution of a system-level scenario test, DEFT can accurately reproduce 1.23 to 1.49 times more planning trajectories.

> **Finding 1:** The results indicate that DEFT generates realistic module tests, as evidenced by their ability to closely reproduce planning trajectories that occurred during system-level tests. With a strict threshold ($\theta_M = 0.0$ meters), DeFT can reproduce 224,888 planning module executions, whereas rerunning the system test only reproduces 1,636 executions.

5.2 RQ2: Failure Reproducibility

In this research question, we focus on evaluating DEFT's effectiveness in terms of its capability to deterministically reproduce failures compared to repeating scenario tests. Figure 7 shows the number of failures (i.e., collisions) detected in each rerun for scenario test and DEFT. We observe that among the 658 collisions that were initially detected, repeating these scenario tests only reproduced 564 to 598 (85.71% to 90.88%) in each rerun, whereas DEFT was capable of reproducing the faulty planning trajectory P_{faulty} for all failing scenarios across all reruns. When considering determinism in scenario testing, only 471 (71.58%) failures were reproducible in all reruns.

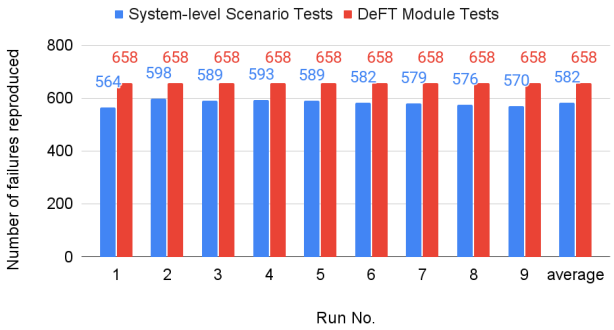

Figure 7: Number of collision violations reproduced in each rerun.

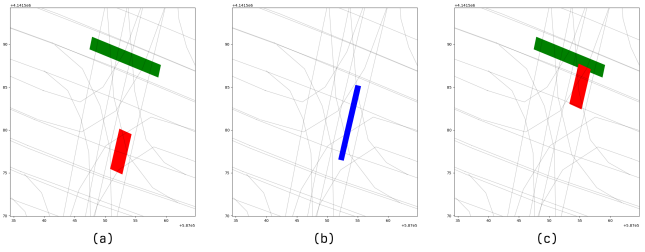

Figure 8: Illustration of (a) the input and (b) the output of a module test for the planning module, and (c) the expected system-level outcome when the planning trajectory reproduced by this module test is executed. The red box illustrates the position of the AV, the green box illustrates the position of an obstacle, and the blue line represents the planned trajectory produced by the planning module.

Figure 8 shows an example of a module test extracted by DeFT that reproduces a faulty planning trajectory. Figure 8(a) visualizes the input frame F extracted by DeFT from the initial execution of a scenario test, and Figure 8(b) visualizes the planning module's output given input frame F. If the ADS executes this planning trajectory, the AV will become involved in a head-on collision with an obstacle as shown in Figure 8(c). The expected outcome of executing this faulty planning decision is consistent with the collision that occurred in the original system-level scenario test. As DeFT module tests accurately reproduce planning trajectories, especially faulty ones, we conclude that module tests extracted from system-level scenario tests can be as effective as those system-level scenario tests in terms of finding the same failure with the additional benefit of being deterministic.

Finding 2: DeFT outperforms simple scenario test reruns by successfully reproducing the faulty planning trajectory for all failing scenarios across all reruns, whereas scenario test reruns only reproduced 85.71% to 90.88% of the 658 initially detected failures per rerun. When considering deterministic failure reproduction, only 471 failures (71.58%) were consistently reproduced in all reruns, whereas DeFT module tests consistently reproduced all failures, resulting in a 28.42 pp gain.

5.3 RQ3: Efficiency

We evaluate the efficiency of DeFT along two key dimensions. First, we measure the time required to extract each module test using DeFT and compare it to a traditional baseline technique, i.e., carving [39]. Second, we assess the runtime performance of DeFT 's extracted module tests by comparing their execution time to that of the original system-level scenario tests.

Table 4: Comparison between DeFT and carving [39].

Dimension	DeFT	Carving [39]
ADS Modification Required	No	Yes
Input Source	Message bus	Instrumented Calls
Granularity	Module	Function
Intrusiveness	Non-intrusive	Intrusive
Runtime Overhead (per frame)	0 ms	49.97 ms
Post-hoc Overhead (per frame)	51.75 ms	0 ms

DeFT shares conceptual similarities with test carving, as both extract concrete inputs from real executions for the purpose of testing or analysis. However, while traditional carving typically targets fine-grained program units—such as individual functions or methods, DeFT operates at the module level by reconstructing message-level inputs from inter-module communication. A further distinction lies in the performance tradeoff. We implemented carving for the planning module by recording input messages to local storage at runtime. As shown in Table 4, traditional carving introduced an average runtime overhead of 49.97 ms per frame. Although this overhead might appear modest, the planning module in our system typically responds in 219.65 ms, meaning carving increased its latency by 22.75%, which is nontrivial in latency-sensitive environments. In contrast, DeFT introduces zero runtime overhead, ensuring the planner's real-time responsiveness remains unaffected. Input reconstruction is instead deferred to an offline phase, with an average cost of 51.75 ms per frame.

Finding 3: DeFT improves upon traditional test carving by reconstructing inputs offline at the module level, avoiding runtime overhead. While traditional carving increased planning latency by 22.75%, DeFT maintains real-time responsiveness with zero impact during execution, making DeFT a more practical and safer option for latency-sensitive, safety-critical systems.

Table 5: Run-time cost of running DeFT compared to system-level scenario tests.

Approach	DeFT			System Test (s)	Reduction
	Extraction	Execution	Total		
AV-FUZZER	21.11	54.23	75.34	86.33	12.73%
BehAVExplor	44.54	86.03	130.57	138.43	5.67%
Decictor	22.50	54.55	77.05	84.85	9.19%
DeepCollision	13.84	39.23	53.07	60.90	12.86%
DoppelTest	12.77	18.69	31.46	35.77	12.04%
DriveFuzz	22.11	57.17	79.27	88.47	10.40%
scenoRITA	14.70	21.88	36.57	34.79	-5.14%

Table 5 compares runtime efficiency of DeFT with system tests. The DeFT columns break down the cost of the approach into extraction and execution times. As shown, DeFT consistently reduces test execution time across most generators, with reductions ranging from 5.67% to 12.86%. Only one case, scenoRITA, resulted in a slight increase in runtime (5.14%). The highest time savings were observed for DeepCollision and AV-FUZZER, with reductions exceeding 12%. Importantly, time savings achieved by DeFT become more substantial as the number of reruns increases, particularly when reproducing non-deterministic failures.

Finding 4: Overall, the results demonstrate that DeFT offers a more efficient alternative to system-level testing in terms of runtime cost while crucially maintaining the deterministic property of the planning module.

Table 6: Total time needed to reproduce failures from each test generator by rerunning system tests and module tests.

Approach	System Test (s)	DeFT Module Test (s)	Time Reduction
AV-FUZZER	104.67	56.54	45.98%
BehAVExplor	152.19	79.33	47.87%
Decictor	334.46	74.78	77.64%
DeepCollision	123.14	58.11	52.81%
DoppelTest	59.68	15.38	74.23%
DriveFuzz	132.09	71.87	45.59%
SAMOTA	94.06	52.97	43.69%
scenoRITA	56.79	20.54	63.83%

To demonstrate these substantial time savings, we evaluate system tests and DeFT module tests in terms of **T**ime-**T**o-**R**eproduce-**F**ailure (TTRF), i.e., the total time in seconds needed to rerun a test case before the expected failure is reproduced. As an example, if a system test always takes precisely 30 seconds to execute, and 3 reruns are needed to reproduce the failure that occurred in the initial execution, then the TTRF of the system test for this scenario

is $3 \times 30 = 90$ seconds; if DeFT module tests extracted from this system test take 20 seconds to execute, and one of these module tests deterministically reproduces the faulty planning trajectory, then the TTRF of module test for this scenario is 20 seconds. As shown in Table 6, rerunning system tests took, on average, 56.79 to 334.46 seconds to reproduce failures that were initially detected, whereas module tests took, on average, 15.38 to 79.33 seconds. DeFT achieves substantial time savings when reproducing failures, especially ones that are non-deterministic, by extracting module tests that reduce TTRF by 43.69% to 77.64% compared to system tests.

> **Finding 5:** Compared to rerunning system-level scenario tests, DeFT can reduce the time needed to reproduce failures (TTRF) by 43.69% to 77.64%.

6 DISCUSSION

6.1 Generalization of the approach

To strengthen the generalizability of our approach beyond a single ADS, we extended our evaluation to Autoware [22] to assess the applicability of DeFT in a different system context. Due to the lack of available test generators for the current Autoware release, we used version 0.45.0 [59] along with the Autoware Evaluator test suite [60], an official Autoware Foundation platform that collects datasets and test suites.

For each scenario, we executed the system test 10 times to capture the effects of non-determinism. We then applied DeFT to extract corresponding module-level tests from the initial system-level execution. Table 7 reports the planning trajectory edit distances across reruns of each scenario in the Autoware test suite. We compare two types of variation: (1) the edit distances between rerun system-level executions and the initial execution (S in Table 7), and (2) the distances between the DeFT-extracted module-level test executions and the same initial execution.

The system-level results show moderate variation, with median edit distances ranging from 0.31 to 0.87 and maximum distances reaching up to 1527.37. These findings indicate the presence of non-determinism in Autoware's planning output during system-level test execution. However, the relatively low median values suggest that Autoware's planning trajectories remain largely consistent across reruns. This consistency is likely influenced by the nature of the evaluated scenarios, many of which involve the ego car traveling in a straight path with minimal interaction or complexity. In contrast, DeFT demonstrates highly accurate reproduction of the planning module's outputs, with edit distances consistently at or near 0.0 and a maximum of only 10.37. This indicates that DeFT effectively captures the execution trace from the initial system-level run with minimal deviation. Moreover, the extracted module-level tests are deterministic, producing consistent outputs across all reruns.

Table 7: Planning trajectory edit distances (meters) across reruns of Autoware test suite.

Rerun	min		1st Quartile		median		3rd Quartile		max	
	S	DeFT	S	DeFT	S	DeFT	S	DeFT	S	DeFT
Rerun 1	0.0	0.0	0.02	0.0	0.53	0.0	6.37	0.0	615.37	10.37
Rerun 2	0.0	0.0	0.02	0.0	0.85	0.0	7.11	0.0	615.38	10.37
Rerun 3	0.0	0.0	0.01	0.0	0.33	0.0	5.81	0.0	363.66	10.37
Rerun 4	0.0	0.0	0.01	0.0	0.77	0.0	5.94	0.0	130.80	10.37
Rerun 5	0.0	0.0	0.02	0.0	0.31	0.0	6.45	0.0	404.41	10.37
Rerun 6	0.0	0.0	0.02	0.0	0.87	0.0	6.17	0.0	539.59	10.37
Rerun 7	0.0	0.0	0.02	0.0	0.40	0.0	6.05	0.0	1527.37	10.37
Rerun 8	0.0	0.0	0.02	0.0	0.84	0.0	6.86	0.0	1121.67	10.37
Rerun 9	0.0	0.0	0.02	0.0	0.86	0.0	7.25	0.0	474.34	10.37

These findings reinforce our three key observations: (1) Autoware exhibits non-deterministic behavior at the system level, consistent with other ADSes; (2) DeFT faithfully reproduces the planning module's execution traces; and (3) the resulting module tests are deterministic and reliable, providing a robust foundation for downstream analysis and debugging.

6.2 Practical Usefulness

While DeFT is not a solution to eliminate non-determinism in system-level testing, it serves as a practical means to harness the value of flaky system tests. Non-determinism in ADSes is largely unavoidable due to real-time simulation and the inherent non-determinism of the ADS itself. Prior work has addressed this challenge by analyzing the impact of flaky simulators [61], using multiple simulation backends to isolate simulator-agnostic failures [62], or discarding non-reproducible results entirely [63]. In contrast, DeFT demonstrates that even flaky system tests can be valuable: They can serve as a foundation for generating deterministic and failure-revealing module tests.

Our evaluation empirically demonstrated two key points: (1) system tests often exhibit non-deterministic outcomes, as repeated executions may not consistently reproduce failures; and (2) DeFT can extract deterministic module tests that reliably reproduce the faulty behavior from a specific failing run. This determinism is especially important for rare or intermittent failures. For instance, the scenario shown in Figure 1 revealed an ADS failure, but the same failure only manifested once after 18 reruns of the same test scenario. As a result, such a test might be perceived as unreliable, leading practitioners to treat it as passing or discard it entirely. DeFT addresses this challenge by capturing the precise failure condition in a deterministic module test—one that fails consistently regardless of the ADS' non-determinism, unless the system itself is modified.

In addition to facilitating failure reproduction and aiding debugging, DeFT provides benefits for regression testing and mutation testing. Since module tests extracted by DeFT maintain the determinism of the planning module, their outputs remain the same as long as the planning module remains unchanged. This deterministic characteristic enables precise identification of which system or module tests need to be rerun following modifications to the planning module. Furthermore, DeFT enhances mutation testing. In this context, a mutant is "killed" if a test case reveals a behavioral difference between the original program and the mutant. While the inherent non-determinism of ADS complicates mutant killing at the system level, DeFT 's deterministic module tests provide a stable and controlled environment for mutation analysis.

7 THREATS TO VALIDITY

External Threats. One external threat to our study is related to whether ADS planning is deterministic in general. Motion planning is PSPACE-hard and a problem belongs to PSPACE complexity if it

can be solved by a deterministic Turing machine [64, 65], and deterministic motion planning is favored in highway motion planning [66]. To mitigate this threat, we have verified with the developers of 2 state-of-the-art open-source ADSes [11, 22] that the planning modules from both ADSes are deterministic.

Another potential threat to the validity of our study is the use of a limited number of autonomous driving software systems. To mitigate this threat, we applied DeFT to two widely used, open-source ADSes, both of which support high levels of autonomy. In both cases, we observed that the extracted module-level tests successfully reproduce planning trajectories and maintain determinism.

Internal Threats. One potential threat to our internal validity is related to whether recording data from the message bus affects the real-time performance of the ADS. The architecture adopted by the state-of-the-art ADSes [27, 28] is highly optimized for performance, latency, and data throughput. Although recording heavy messages, such as raw sensor data, can cause performance issues for the ADS, we further confirm with professional ADS developers that messages subscribed by the planning module are lightweight and recording them, like what DeFT is doing, will not affect the performance of the ADS.

Another potential threat to our internal validity is the selection of the simulator and the implementation of prior testing approaches. Some of the test generators selected in our study [13, 17, 20] were originally implemented in a simulator [67] that is no longer maintained [47]. To mitigate the threat, we selected high-quality reimplementations from the authors of a paper recently accepted to the International Conference on Software Engineering (ICSE) [48]. The simulator used by these implementations (i.e., SimControl) has also been recommended by an official maintainer of the ADS [49].

8 RELATED WORK

ADS Testing. A number of recent studies focus on testing ADSes in simulated environments using simulators. AC3R [68] virtually reconstructed crashes based on police reports. Calo et. al [69] applied search-based testing to discover driving violations (e.g., collision, violation of traffic rules) in ADS. Gambi et al. [70] generated different road networks to discover failure in ADS lane-keeping in simulation. Along the lines of those work, AV-FUZZER [13], AutoFuzz [14], LawBreaker [71], DeepCollision [17], and BehAVExplor [20] use different strategies to generate scenarios but use a common setup of testing the same ADS (i.e., Apollo [11]) with the same simulator (i.e., SVL [67]). Some other works, such as DoppelTest [18] and scenoRITA [19], also choose Apollo [11] but a different simulation environment [49] to focus testing on ADS planning. Because those approaches generate tests that are executed through real-time simulation, non-determinism has been a common threat to validity. PlanFuzz [72] was the first ADS testing approach that focused on mutating the input frame of the planning module to detect high-level behavior planning changes (e.g., cruising, stopping). PlanFuzz achieved this goal by modifying the implementation of the planning module to enable recording and restoring states and manually selecting frames that are suitable for mutation. Different from PlanFuzz, DeFT reconstructs the input frame of the planning module without modification of its implementation and automatically creates unit tests that reproduce each planning trajectory produced by the planning module during a real-time simulation. STRAP [73] is a test reduction and prioritization that discussed the notion of frames by making copies of messages produced by different ADS modules and aligning them with respect to the module that has the highest output frequency. DeFT focuses on reconstructing the input frames for the planning module, while STRAP-aligned frames are not intended to be reused to reproduce the execution of any ADS module.

Unit Test Generation. Fraser et al. [74] proposed EvoSuite, which automatically generates unit tests for Java code. Soltani et al. [75] proposed EvoCrash, which uses a guided genetic algorithm to create unit tests that reproduce crashes based on stack traces. These approaches cannot reproduce the executions of the planning module because of the complexity of the input space (e.g., requiring realistic obstacle trajectories). Orso et al. [75] discussed selective capture and replay of program executions, and Elbaum et al. [39] discussed carving differential unit tests from system tests; both techniques instrument the software under test to collect internal states or method invocation arguments so that they can be replayed later to reproduce the execution of the program. While DeFT is inspired by aforementioned approaches and indeed akin to carving, we demonstrate how carving can be used to solve one of the main challenges in testing robotics systems (i.e., non-determinism) [21, 76, 77], and how we overcome carving's key deficiency of requiring instrumentation since real-time systems are sensitive to instrumentation-induced execution overhead [41, 78, 79]. Alcon et al. [80] discussed ADS unit testing and demonstrated that individual ADS modules are deterministic when their executions are isolated. However, significant modifications to the implementations of the ADS were needed, and despite multiple attempts to contact the authors, we have not received the scripts developed to achieve deterministic execution.

9 ACKNOWLEDGMENT

This work was supported in part by the U.S. National Science Foundation under Grants CNS-2145493, CNS-2413877, CNS-2443763, and CNS-2346561, and by the U.S. Department of Transportation under Grant 69A3552348327 through the CARMEN+ University Transportation Center.

10 CONCLUSION

The inherent non-determinism of ADSes poses challenges for software testing and analysis, as scenario tests yield non-deterministic outcomes due to the nature of the software under test. Instead of generating new scenario tests, our approach treats the execution of a system-level scenario test as a process of generating valid and realistic inputs for the planning module and extracts inputs for module tests to reproduce executions of the planning module that occurred during real-time simulation. Given that the planning module is deterministic in the state-of-the-art open-source ADS, we demonstrated that a deterministic planning module test suite can be obtained from non-deterministic system tests. Further, we demonstrated planning module tests are more effective in terms of deterministically reproducing failures, and taking less time to execute compared to rerunning system tests.

REFERENCES

[1] "The 18 Companies Most Likely to Get Self-driving Dars on the Road First," https://www.businessinsider.com/the-companies-most-likely-to-get-driverless-cars-on-the-road-first-2017-4, August 2019.

[2] "46 Corporations Working On Autonomous Vehicles," https://www.cbinsights.com/research/autonomous-driverless-vehicles-corporations-list/, August 2019.

[3] "Autopilot | Tesla," https://www.tesla.com/autopilot. [Online]. Available: https://www.tesla.com/autopilot

[4] "Waymo One," https://waymo.com/waymo-one/. [Online]. Available: https://waymo.com/waymo-one/

[5] "Scaling Waymo One safely across four cities this year," https://waymo.com/blog/2024/03/scaling-waymo-one-safely-across-four-cities-this-year/.

[6] "Self-Driving Trucks | Torc Robotics," https://torc.ai/trucking/.

[7] X. Rigoulet, "Do we Need Self-Driving Cars?" https://medium.datadriveninvestor.com/do-we-need-self-driving-cars-394c40fe4144. [Online]. Available: https://medium.datadriveninvestor.com/do-we-need-self-driving-cars-394c40fe4144

[8] "Death of Elaine Herzberg," https://en.wikipedia.org/w/index.php?title=Death_of_Elaine_Herzberg&oldid=1066785093. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Death_of_Elaine_Herzberg&oldid=1066785093

[9] "Woman injured after being struck by SF hit-and-run driver, trapped under autonomous car, Cruise says," https://abc7news.com/san-francisco-woman-injured-pedestrian-crash-cruise/13857047/.

[10] "Autonomous Vehicle Testing Permit Holders - California DMV," https://www.dmv.ca.gov/portal/vehicle-industry-services/autonomous-vehicles/autonomous-vehicle-testing-permit-holders/.

[11] "ApolloAuto/apollo," Jan. 2024. [Online]. Available: https://github.com/ApolloAuto/apollo

[12] "Baidu's Apollo Go: Super cheap robotaxi rides spark widespread anxiety in China | CNN Business." [Online]. Available: https://www.cnn.com/2024/07/18/cars/china-baidu-apollo-go-robotaxi-anxiety-intl-hnk/index.html

[13] G. Li, Y. Li, S. Jha, T. Tsai, M. Sullivan, S. K. S. Hari, Z. Kalbarczyk, and R. Iyer, "Av-fuzzer: Finding safety violations in autonomous driving systems," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, 2020, pp. 25–36.

[14] Z. Zhong, G. Kaiser, and B. Ray, "Neural network guided evolutionary fuzzing for finding traffic violations of autonomous vehicles," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1860–1875, 2023.

[15] H. Tian, Y. Jiang, G. Wu, J. Yan, J. Wei, W. Chen, S. Li, and D. Ye, "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*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 94–106. [Online]. Available: https://doi.org/10.1145/3540250.3549100

[16] H. Tian, G. Wu, J. Yan, Y. Jiang, J. Wei, W. Chen, S. Li, and D. Ye, "Generating critical test scenarios for autonomous driving systems via influential behavior patterns," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3560430

[17] C. Lu, Y. Shi, H. Zhang, M. Zhang, T. Wang, T. Yue, and S. Ali, "Learning Configurations of Operating Environment of Autonomous Vehicles to Maximize their Collisions," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 384–402, Jan. 2023, conference Name: IEEE Transactions on Software Engineering.

[18] Y. Huai, Y. Chen, S. Almanee, T. Ngo, X. Liao, Z. Wan, Q. A. Chen, and J. Garcia, "Doppelgänger test generation for revealing bugs in autonomous driving software," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 2591–2603.

[19] Y. Huai, S. Almanee, Y. Chen, X. Wu, Q. A. Chen, and J. Garcia, "scenorita: Generating diverse, fully mutable, test scenarios for autonomous vehicle planning," *IEEE Transactions on Software Engineering*, vol. 49, no. 10, pp. 4656–4676, 2023.

[20] M. Cheng, Y. Zhou, and X. Xie, "BehAVExplor: Behavior Diversity Guided Testing for Autonomous Driving Systems," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. Seattle WA USA: ACM, Jul. 2023, pp. 488–500. [Online]. Available: https://dl.acm.org/doi/10.1145/3597926.3598072

[21] T. Laurent, S. Klikovits, P. Arcaini, F. Ishikawa, and A. Ventresque, "Parameter Coverage for Testing of Autonomous Driving Systems under Uncertainty," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 3, pp. 1–31, Jul. 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3550270

[22] "autowarefoundation/autoware.universe," Jan. 2024. [Online]. Available: https://github.com/autowarefoundation/autoware.universe

[23] "DeFT - Zenodo," https://doi.org/10.5281/zenodo.17978768.

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

[25] A. Tobin, "What are the different levels of self-driving cars?" Jul 2019. [Online]. Available: https://www.forbes.com/sites/annatobin/2019/07/07/what-are-the-different-levels-of-self-driving-cars/?sh=4763b969f475

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

[27] "Apollo Cyber RT framework," https://developer.apollo.auto/cyber.html.

[28] "ROS - Robot Operating System," https://www.ros.org/.

[29] J. Dai, B. Gao, M. Luo, Z. Huang, Z. Li, Y. Zhang, and M. Yang, "SCTrans: Constructing a Large Public Scenario Dataset for Simulation Testing of Autonomous Driving Systems," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. Lisbon Portugal: ACM, Feb. 2024, pp. 1–13. [Online]. Available: https://dl.acm.org/doi/10.1145/3597503.3623350

[30] "Apollo: Overview of the planning module," https://github.com/ApolloAuto/apollo/blob/v10.0.0/docs/%E5%BA%94%E7%94%A8%E5%AE%9E%E8%B7%B5/%E5%BC%80%E5%8F%91%E8%B0%83%E8%AF%95%E6%95%99%E7%A8%8B/Apollo%E8%A7%84%E5%88%92%E5%AE%9E%E8%B7%B5/%E8%A7%84%E5%88%92%E6%A8%A1%E5%9D%97%E7%BB%BC%E8%BF%B0.md.

[31] "Self-Driving Fundamentals: Featuring Apollo | Udacity." [Online]. Available: https://www.udacity.com/course/self-driving-car-fundamentals-featuring-apollo--ud0419

[32] "Apollo Open Vehicle Certificate Platform." [Online]. Available: https://developer.apollo.auto/vehicle/certificate_en.html

[33] "Baidu hits the gas on autonomous vehicles with Volvo and Ford deals | TechCrunch." [Online]. Available: https://techcrunch.com/2018/11/01/baidu-volvo-ford-autonomous-driving/

[34] "Recorder - CARLA Simulator." [Online]. Available: https://carla.readthedocs.io/en/latest/adv_recorder/#recording

[35] "Apollo Cyber RT Developer Tools — Cyber RT Documents documentation." [Online]. Available: https://cyber-rt.readthedocs.io/en/latest/CyberRT_Developer_Tools.html

[36] "Recording and playing back data — ROS 2 Documentation: Iron documentation." [Online]. Available: https://docs.ros.org/en/iron/Tutorials/Beginner-CLI-Tools/Recording-And-Playing-Back-Data/Recording-And-Playing-Back-Data.html

[37] "Testing Oracles · Issue #11268 · ApolloAuto/apollo." [Online]. Available: https://github.com/ApolloAuto/apollo/issues/11268

[38] M. Alcon, H. Tabani, L. Kosmidis, E. Mezzetti, J. Abella, and F. J. Cazorla, "Timing of Autonomous Driving Software: Problem Analysis and Prospects for Future Solutions," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Sydney, NSW, Australia: IEEE, Apr. 2020, pp. 267–280. [Online]. Available: https://ieeexplore.ieee.org/document/9113112/

[39] S. Elbaum, H. N. Chin, M. B. Dwyer, and M. Jorde, "Carving and Replaying Differential Unit Test Cases from System Test Cases," *IEEE Transactions on Software Engineering*, vol. 35, no. 1, pp. 29–45, Jan. 2009, conference Name: IEEE Transactions on Software Engineering.

[40] A. Gambi, H. Gouni, D. Berreiter, V. Tymofyeyev, and M. Fazzini, "Action-Based Test Carving for Android Apps," in *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. Dublin, Ireland: IEEE, Apr. 2023, pp. 107–116. [Online]. Available: https://ieeexplore.ieee.org/document/10132300/

[41] G. M. Tchamgoue and S. Fischmeister, "Lessons learned on assumptions and scalability with time-aware instrumentation," in *Proceedings of the 13th International Conference on Embedded Software*. Pittsburgh Pennsylvania: ACM, Oct. 2016, pp. 1–7. [Online]. Available: https://dl.acm.org/doi/10.1145/2968478.2975584

[42] "roscpp/Overview/Timers," http://wiki.ros.org/roscpp/Overview/Timers.

[43] "roscpp/Overview/Publishers and Subscribers," http://wiki.ros.org/roscpp/Overview/Publishers%20and%20Subscribers.

[44] T. Yaguang, B. Alan, I. S. Rodrigo, B. Kevin, S. Stef, S. P. Ross, L. Patrick, P. Dongliang, T. Kevin, and M. Duckham, "A comparative analysis of trajectory similarity measures," *GIScience & Remote Sensing*, vol. 58, no. 5, pp. 643–669, 2021. [Online]. Available: https://doi.org/10.1080/15481603.2021.1908927

[45] F. U. Haq, D. Shin, and L. Briand, "Efficient online testing for DNN-enabled systems using surrogate-assisted and many-objective optimization," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pp. 811–822, ISSN: 1558-1225.

[46] S. Kim, M. Liu, J. J. Rhee, Y. Jeon, Y. Kwon, and C. H. Kim, "DriveFuzz: Discovering autonomous driving bugs through driving quality-guided fuzzing," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. ACM, pp. 1753–1767. [Online]. Available: https://dl.acm.org/doi/10.1145/3548606.3560558

[47] "Svl simulator sunset," March 2022. [Online]. Available: https://www.svlsimulator.com/news/2022-01-20-svl-simulator-sunset

[48] M. Cheng, Y. Zhou, X. Xie, J. Wang, G. Meng, and K. Yang, "Decictor: Towards evaluating the robustness of decision-making in autonomous driving systems," *arXiv preprint arXiv:2402.18393*, 2024.

[49] "Apollo 7.0 + LGSVL 2021.2 sometimes does not complete a sharp turn," https://github.com/ApolloAuto/apollo/issues/14756#issuecomment-1405921603.

[50] "cclinus/AV-Fuzzer," https://github.com/cclinus/AV-Fuzzer.

[51] "MingfeiCheng/AV-Fuzzer-Reimplement," https://github.com/MingfeiCheng/AV-Fuzzer-Reimplement.

[52] "MingfeiCheng/Decictor," https://github.com/MingfeiCheng/Decictor.

[53] "MingfeiCheng/BehAVExplor," https://github.com/MingfeiCheng/BehAVExplor/.

[54] "simplexity-lab/DeepCollision," https://github.com/simplexity-lab/DeepCollision.

[55] "Software-Aurora-Lab/DoppelTest," https://github.com/Software-Aurora-Lab/DoppelTest.

[56] "S3 Lab/Public/drivefuzz," https://gitlab.com/s3lab-code/public/drivefuzz.

[57] "ADS-Testing/SAMOTA," https://github.com/ADS-Testing/SAMOTA.

[58] "Software-Aurora-Lab/scenoRITA-7.0," https://github.com/Software-Aurora-Lab/scenoRITA-7.0.

[59] "Release 0.45.0 - autowarefoundation/autoware," July 2025. [Online]. Available: https://github.com/autowarefoundation/autoware/releases/tag/0.45.0

[60] "Tier IV Autoware Evaluator," July 2025. [Online]. Available: https://evaluation.tier4.jp/evaluation/

[61] M. H. Amini, S. Naseri, and S. Nejati, "Evaluating the impact of flaky simulators on testing autonomous driving systems," vol. 29, no. 2, p. 47. [Online]. Available: https://link.springer.com/10.1007/s10664-023-10433-5

[62] L. Sorokin, M. Biagiola, and A. Stocco, "Simulator ensembles for trustworthy autonomous driving testing." [Online]. Available: http://arxiv.org/abs/2503.08936

[63] C. Birchler, S. Khatiri, B. Bosshard, A. Gambi, and S. Panichella, "Machine learning-based test selection for simulation-based testing of self-driving cars software," *Empirical Software Engineering*, vol. 28, no. 3, p. 71, Apr. 2023. [Online]. Available: https://doi.org/10.1007/s10664-023-10286-y

[64] S. M. LaValle, *Planning Algorithms*, 1st ed. Cambridge University Press, May 2006. [Online]. Available: https://www.cambridge.org/core/product/identifier/9780511546877/type/book

[65] S. Pendleton, H. Andersen, X. Du, X. Shen, M. Meghjani, Y. Eng, D. Rus, and M. Ang, "Perception, Planning, Control, and Coordination for Autonomous Vehicles," *Machines*, vol. 5, no. 1, p. 6, Feb. 2017. [Online]. Available: http://www.mdpi.com/2075-1702/5/1/6

[66] L. Claussmann, M. Revilloud, D. Gruyer, and S. Glaser, "A review of motion planning for highway autonomous driving," *IEEE Transactions on Intelligent Transportation Systems*, vol. 21, no. 5, pp. 1826–1848, 2020.

[67] "Svl simulator by lg," March 2022. [Online]. Available: https://www.svlsimulator.com/

[68] A. Gambi, T. Huynh, and G. Fraser, "Generating effective test cases for self-driving cars from police reports," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Tallinn Estonia: ACM, Aug. 2019, pp. 257–267. [Online]. Available: https://dl.acm.org/doi/10.1145/3338906.3338942

[69] A. Calo, P. Arcaini, S. Ali, F. Hauer, and F. Ishikawa, "Generating Avoidable Collision Scenarios for Testing Autonomous Driving Systems," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. Porto, Portugal: IEEE, Oct. 2020, pp. 375–386. [Online]. Available: https://ieeexplore.ieee.org/document/9159061/

[70] A. Gambi, M. Mueller, and G. Fraser, "Automatically testing self-driving cars with search-based procedural content generation," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Beijing China: ACM, Jul. 2019, pp. 318–328. [Online]. Available: https://dl.acm.org/doi/10.1145/3293882.3330566

[71] Y. Sun, C. M. Poskitt, J. Sun, Y. Chen, and Z. Yang, "LawBreaker: An Approach for Specifying Traffic Laws and Fuzzing Autonomous Vehicles," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. Rochester MI USA: ACM, Oct. 2022, pp. 1–12. [Online]. Available: https://dl.acm.org/doi/10.1145/3551349.3556897

[72] Z. Wan, J. Shen, J. Chuang, X. Xia, J. Garcia, J. Ma, and Q. A. Chen, "Too Afraid to Drive: Systematic Discovery of Semantic DoS Vulnerability in Autonomous Driving Planning under Physical-World Attacks," in *Network and Distributed System Security (NDSS) Symposium, 2022*, April 2022.

[73] Y. Deng, X. Zheng, M. Zhang, G. Lou, and T. Zhang, "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*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 82–93. [Online]. Available: https://doi.org/10.1145/3540250.3549152

[74] G. Fraser and A. Arcuri, "EvoSuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. Szeged Hungary: ACM, Sep. 2011, pp. 416–419. [Online]. Available: https://dl.acm.org/doi/10.1145/2025113.2025179

[75] A. Orso and B. Kennedy, "Selective capture and replay of program executions," in *Proceedings of the Third International Workshop on Dynamic Analysis*, ser. WODA '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 1–7. [Online]. Available: https://doi.org/10.1145/1083246.1083251

[76] A. Afzal, C. L. Goues, M. Hilton, and C. S. Timperley, "A study on challenges of testing robotic systems," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020, pp. 96–107.

[77] N. Valigi, "Lessons learned building a self-driving car on ROS," https://roscon.ros.org/2018/presentations/ROSCon2018_LessonsLearnedSelfDriving.pdf.

[78] T. Pankumhang and M. Rutherford, "Iterative Instrumentation for Code Coverage in Time-Sensitive Systems," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. Graz, Austria: IEEE, Apr. 2015, pp. 1–10. [Online]. Available: http://ieeexplore.ieee.org/document/7102594/

[79] S. Fischmeister and P. Lam, "Time-Aware Instrumentation of Embedded Software," *IEEE Transactions on Industrial Informatics*, vol. 6, no. 4, pp. 652–663, Nov. 2010. [Online]. Available: http://ieeexplore.ieee.org/document/5559440/

[80] M. Alcon, H. Tabani, J. Abella, and F. J. Cazorla, "Enabling Unit Testing of Already-Integrated AI Software Systems: The Case of Apollo for Autonomous Driving," in *2021 24th Euromicro Conference on Digital System Design (DSD)*. Palermo, Italy: IEEE, Sep. 2021, pp. 426–433. [Online]. Available: https://ieeexplore.ieee.org/document/9556323/