Discovering Adversarial Driving Maneuvers against Autonomous Vehicles

Ruoyu Song, Muslum Ozgur Ozmen, Hyungsub Kim, Raymond Muller,
Z. Berkay Celik, and Antonio Bianchi

Purdue University
{song464, mozmen, kim2956, mullerr, zcelik, antoniob}@purdue.edu

Abstract

Over 33% of vehicles sold in 2021 had integrated autonomous driving (AD) systems. While many adversarial machine learning attacks have been studied against these systems, they all require an adversary to perform specific (and often unrealistic) actions, such as carefully modifying traffic signs or projecting malicious images, which may arouse suspicion if discovered. In this paper, we present ACERO, a robustness-guided framework to discover adversarial maneuver attacks against autonomous vehicles (AVs). These maneuvers look innocent to the outside observer but force the victim vehicle to violate safety rules for AVs, causing physical consequences, e.g., crashing with pedestrians and other vehicles. To optimally find adversarial driving maneuvers, we formalize seven safety requirements for AD systems and use this formalization to guide our search. We also formalize seven physical constraints that ensure the adversary does not place themselves in danger or violate traffic laws while conducting the attack. ACERO then leverages trajectory-similarity metrics to cluster successful attacks into unique groups, enabling AD developers to analyze the root cause of attacks and mitigate them. We evaluated ACERO on two open-source AD software, openpilot and Autoware, running on the CARLA simulator. ACERO discovered 219 attacks against openpilot and 122 attacks against Autoware. 73.3% of these attacks cause the victim to collide with a third-party vehicle, pedestrian, or static object.

1 Introduction

Autonomous driving (AD) is becoming increasingly prevalent throughout the world. AD systems such as Waymo [34], Autoware [23], and openpilot [11] are already deployed on public roads. Unfortunately, previous works have shown that attackers can spoof/disturb sensors of AVs [6, 36, 54, 58] and exploit flaws in AD software's control logic [5, 70].

These attacks, however, may look deliberate to an external observer (e.g., a traffic law enforcer). Thus, attackers are wary of using them to avoid legal liabilities. In contrast, while some maneuvers do not violate any safety or traffic rules, they can still cause AVs to misbehave, putting AVs and nearby traffic in danger. For example, Fig. 1 shows a case of Tesla autopilot steering off the road due to a non-malicious maneuver. An attacker could perform such maneuvers to jeopardize a victim car's safety while minimizing their liability.

Figure 1: Real-world driving behavior causing the Tesla autopilot to steer off the road, forcing the operator to intervene [49]. While this behavior could be defined as "aggressive" or "inconsiderate", it does not look *intentionally* malicious.

In this paper, we systematically discover *low liability* (i.e., low legal responsibility) adversarial maneuvers. We define a maneuver as low liability if the adversary does not violate a set of traffic laws (detailed in Sec. 3). Although such maneuvers are more attractive to attackers, discovering them brings two unique challenges.

The first challenge is the large search space for maneuver-based attacks. An adversary must observe how an AV reacts in a driving scene, which has many associated variables, including traffic, weather, and the behavior of other agents. To this end, previous work on AV safety testing [30, 42, 60, 61, 64] have used high-fidelity simulators, such as CARLA [15] and LGSVL [51]. However, using simulators to effectively search for these attacks in the space of all possible maneuvers requires dedicated search optimizations. Additionally, it requires the implementation of mechanisms to reset a simulator to restore the variables related to an earlier physical state to test different adversarial maneuvers.

A second challenge is that, without proper constraints, maneuver-based attacks may also put the attacker at risk for physical consequences. This can be seen by considering a greedy solution to maneuver-based attacks: the attacker simply hits the victim vehicle. However, by doing so, the attacker

loses *both* low liability and safety, making such an attack impractical. Thus, the search space of the attacks must respect two principles. First, the attacker and the adversarial vehicle should not be damaged. Second, traffic laws, such as driving on the correct side of the road, must be obeyed.

Driving scene generation approaches [18, 30, 42, 60, 61, 64] perturb the environment and traffic behavior to create a safety violation, as opposed to finding the maneuvers that a vehicle can make to cause an AV to violate its safety constraints. A recent work identifies the maneuvers a vehicle can make to cause safety violations for an AV [30]. Yet, this work does not ensure the vehicle obeys traffic laws and self-safety constraints. Further, it only considers traffic conditions that involve vehicle following and lane changing, which restricts its application to AVs operating in specific driving scenarios. Another recent work [53] generates adversarial vehicle trajectories to discover vulnerabilities in collision avoidance systems (CAS) while maximizing the adversarial vehicle's distance from other vehicles to prevent its collisions. Yet, this work focuses on identifying vulnerabilities specific to CAS and does not require the adversary to obey traffic laws.

Motivated by the severity of adversarial maneuvers against AVs and the lack of effective approaches to discover them, we introduce ACERO, an automated system to identify maneuvers that induce safety violations in AD systems while ensuring the adversary's safety and low liability.

Although safety standards have been established for AVs, to verify whether or not they are satisfied, they must be analyzed and modeled into a verifiable form. Thus, we first identify the safety standards for each level of AD capability and formally represent them as "missions" that can be evaluated to determine whether a standard is violated. ACERO then initializes an "attack scene" based on a given real-world scenario pertaining to the target AV's capability. The attack scene initializes an environment with an AV, an adversarial vehicle, traffic, and pedestrians. ACERO next iteratively generates a set of *adversarial control commands* that an adversary can execute to cause the victim AV to violate one or more established safety standards. To find these commands, it combines the established technique of robustness-guidance [25, 45, 46] with novel, formally represented missions, and optimizes the search process via a hill-climbing approach.

Robustness-guidance alone, however, can lead to self-sabotaging or obviously-malicious driving behaviors for the adversary. To address this aspect, we derive seven formal constraints on adversarial control commands that ensure both low liability and safety for the adversary. The system ensures that these constraints are met during the search process. Lastly, ACERO clusters geometrically similar attacks into generalized groups. These groups can be analyzed by AD developers to reproduce the discovered attacks under different traffic and environmental conditions and establish defenses against them.

We evaluate ACERO against two AD systems, openpilot [11] and Autoware [23], running on the CARLA [15] simulator. We create 14 different driving scenes to rigorously test each safety requirement. In these scenes, ACERO discovers 219 successful maneuver-based attacks against openpilot and 122 against Autoware. We cluster them into 28 unique attacks (13 for openpilot and 15 for Autoware) and identify their root causes. These attacks cause the targeted AV to violate a safety standard, putting the victim at risk without harming the attacker. Specifically, 57.8% of the attacks cause the victim to collide with a third-party vehicle, 8.5% with a pedestrian or cyclist, and 7% with a static object (e.g., a road sign). In summary, we make the following contributions:

- **Mission Identification and Formalization.** We analyze AD safety standards and formally represent them using temporal logic. Any testing tool (including ACERO) can leverage these formalized missions to determine how close an AV is to violating its safety standards.

- **Robustness-guided Adversarial Command Generation.** We combine the robustness-guidance approach with novel formulas representative of an AV's safety. These formulas efficiently identify maneuvers a vehicle can follow to force an AV to violate its safety standards.

- **Enforcing Physical Constraints on the Adversarial Vehicle.** We design seven formally verifiable physical constraints to enforce on adversarial maneuvers. These constraints ensure the attacker's safety, maintain low liability, and preserve the practicality of the attack.

- **Evaluation on two AD Systems.** We use ACERO [2] on two popular AD software (openpilot and Autoware) and discover 341 attacks causing the targeted AV to hit other vehicles, pedestrians, cyclists, and static objects.

2 Background

Autonomous Driving (AD) Systems. AD systems consist of sensing, perception, planning, and actuation modules [56]. AD software takes inputs from multiple sensors capturing different aspects of the environment, e.g., the camera outputs RGB video, and the LiDAR outputs 3D point clouds.

The perception module processes the raw sensor data and generates interpretable data. The AD software often extends machine learning (ML) models to process the sensor data in real-time to detect, track, and predict other vehicles and the environment around the vehicle. For instance, the models detect objects and track their 3D locations [73], e.g., traffic signs, surrounding vehicles, and pedestrians, and the moving agents around the vehicle [71]. These models are often trained on sensor data collected from millions of vehicles [11].

The planning module enables a vehicle to find a safe route from a given origin to a destination. It determines the most feasible trajectory by incorporating path search and maneuver planning algorithms (e.g., changing lanes and overtaking) [24] while ensuring the vehicle avoids static obstacles.

Lastly, the AD software issues control commands to the vehicle, e.g., steering angle, throttle, and brake, so that the vehicle achieves its intended missions, such as preventing pedestrian crashes and following the planned travel path.

Levels of Autonomy. The Society of Automotive Engineers (SAE) has defined levels of autonomy [52]. Level 1 provides steering (e.g., lane centering) or accelerating (e.g., adaptive cruise control) support, and Level 2 supports both. Level 3 and 4 autonomous vehicles can drive themselves under certain conditions, but they require different levels of human input. Level 3 vehicles require a human driver to be present and alert at all times, while Level 4 vehicles can operate without human intervention, but only in defined operational locations.

3 Motivation and Threat Model

We consider an adversary that controls a vehicle ($Attack_{car}$) near an autonomous victim vehicle ($Victim_{car}$). We assume the adversary knows the $Victim_{car}$'s control software and has access to its two physical states: ground speed and relative position. The adversary can obtain these physical states through the sensors in the $Attack_{car}$ (e.g., camera and LiDAR).

The adversary's goal is to jeopardize the $Victim_{car}$'s safety by causing it to collide with other agents in the traffic (e.g., static objects, pedestrians, cyclists, and other vehicles), while preserving both the $Attack_{car}$'s safety and its low liability. To preserve the $Attack_{car}$'s safety, we enforce that $Victim_{car}$ does not get physically damaged. To ensure $Attack_{car}$ has low liability, in case of $Victim_{car}$ collides with other agents, we force $Attack_{car}$ to observe traffic laws. Specifically, we focus on ensuring $Attack_{car}$ does not violate the following two traffic laws: (*i*) obeying the traffic signals (e.g., not exceeding the speed limits) and (*ii*) driving in the correct lane. We focus on these two laws as they are consistent across countries and states, and their violation is a major cause of real-world accidents [37].

Concretely, to enforce both the $Attack_{car}$'s safety and low liability, during our simulations, we enforce seven specific physical constraints on the $Attack_{car}$ (Sec. 4.3). We note that by mandating the adversary to respect physical constraints, our threat model is narrow, and it limits the types of attacks that our approach can find. Specifically, we only consider valid attacks in which the $Attack_{car}$ does not violate any traffic rule or crash with any car in the traffic scene.

We allow expanding our threat model by disabling physical constraints, such as by breaking traffic laws, because it may help discover more adversarial maneuvers, though this will jeopardize the attacker's safety and limit the attacker from denying malicious intent or direct responsibility.

To achieve its goal, the adversary aims to find an adversarial trajectory that makes the $Victim_{car}$ violate the safety rules defined in its SAE Level 2-4 control software (Sec. 4.1). We assume the adversary finds such adversarial trajectories on AV simulators that can run diverse AD software. While, in theory, the adversary could try to find the adversarial trajectories on real roads, in practice, such trials are unsafe and impractical compared to the simulated environment. The adversary can collect other agents in the environment (e.g., other cars on the road) and use this information to create maneuvers to make $Victim_{car}$ collide with them opportunistically.

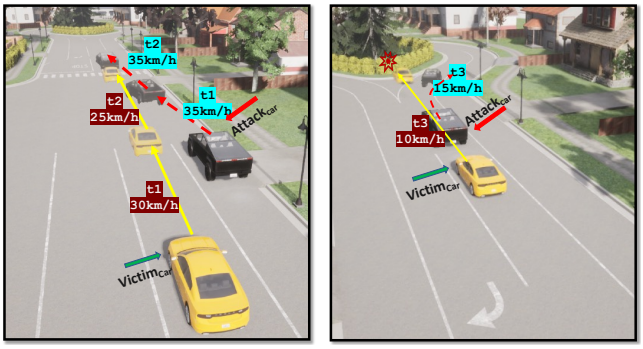

Figure 2: Example adversarial maneuvers against openpilot.

The resulting adversarial trajectory can either be programmed into the autonomous $Attack_{car}$, or the adversary can manually drive a non-autonomous $Attack_{car}$ and follow the trajectory within an acceptable deviation (Sec. 4.5).

Example. We consider a $Victim_{car}$ that travels using the openpilot's [11] adaptive cruise control with lane centering at an initial speed of 36 km/h, as shown in Fig. 2 [2]. The $Victim_{car}$ goes straight at a constant speed until the $Attack_{car}$ brakes in front of it (`t1-t2`). This results in the $Victim_{car}$ also braking (`t2`). However, when the $Attack_{car}$ speeds up and turns right, the $Victim_{car}$'s perception module fails to recognize the road curve and continues moving straight, eventually driving to the sidewalk and colliding with a concrete wall (`t3`).

In these maneuvers, the $Attack_{car}$ (*i*) does not collide with any other agents, and (*ii*) obeys traffic laws. This example shows that the adversarial trajectory impacts the $Victim_{car}$'s perception in a way that its planner fails to keep it centered in the lane. In such a scenario, a human driver would drive more cautiously when the $Attack_{car}$ blocks its vision and also see the road curve before the $Attack_{car}$ blocks its vision and remember it. Yet, this attack exploits the fact that openpilot uses limited historical camera frames and causes a crash by blocking the $Victim_{car}$'s vision for a short period of time ($\approx$1 sec).

4 ACERO System

We introduce ACERO, which systematically discovers the maneuvers an $Attack_{car}$ can make to cause the $Victim_{car}$ to fail its intended operations while ensuring that the adversary remains safe and achieves low liability.

Designing ACERO raises several unique system challenges: (1) Formally identifying the $Victim_{car}$'s missions to ensure its safe driving (Sec. 4.1). (2) Designing and implementing an algorithm to generate realistic attack scenes for the behavior of other traffic agents and weather conditions (Sec. 4.2). (3) Developing a trajectory generation algorithm that identifies the trajectories the $Attack_{car}$ can follow to force the $Victim_{car}$

Table 1: Missions that AVs should adhere to for their correct and safe operation.

ID	Mission Description	SAE Level	LTL Formula†
$\texttt{Stay}_{\texttt{Lane}}$	The $Victim_{car}$ should not move out of its lane.	2	$\square(\texttt{Dist}(\texttt{vv},\texttt{current-lane}) > 0)$
$\texttt{Drive}_{\texttt{Allowed}}$	The $Victim_{car}$ should not enter restricted areas.	3,4	$\square(\texttt{Dist}(\texttt{vv},\texttt{ra}) > 0)$
$\texttt{Follow}_{\texttt{Car}}$	The $Victim_{car}$ should perform car-following.	2	$\square(\texttt{TTC}(\texttt{vv},\texttt{fc}) > \tau)$
$\texttt{React}_{\texttt{SO}}$	The $Victim_{car}$ should respond to static obstacles in the roadway.	3,4	$\square(\texttt{TTC}(\texttt{vv},\texttt{obj}) > \tau)$
$\texttt{React}_{\texttt{LC}}$	The $Victim_{car}$ should respond to intended lane changes or cut-ins.	3,4	$\square(\texttt{TTC}(\texttt{vv},\texttt{cut-in}) > \tau)$
$\texttt{React}_{\texttt{EV}}$	The $Victim_{car}$ should respond to encroaching oncoming vehicles.	3,4	$\square(\texttt{TTC}(\texttt{vv},\texttt{on-coming}) > \tau)$
$\texttt{React}_{\texttt{MO}}$	The $Victim_{car}$ should respond to bicycles, pedestrians, animals, or other moving objects.	3,4	$\square(\texttt{TTC}(\texttt{vv},\texttt{bic}) > \tau \wedge \texttt{TTC}(\texttt{vv},\texttt{ped}) > \tau \wedge \texttt{TTC}(\texttt{vv},\texttt{ani}) > \tau)$

† Dist() and TTC () are detailed in Sec. 4.1.1. ra:= restricted areas $Victim_{car}$ should not enter (e.g., one-way streets), τ:= TTC threshold, cut-in: vehicles that change lanes in front of the victim, on-coming: vehicles coming towards the victim from the opposite direction, obj: objects, bic: bicycles, ped: pedestrians, ani: animals.

to fail its intended missions (Sec. 4.4). (4) Developing a clustering approach to group similar adversarial trajectories for root cause analysis, which can help developers reproduce the attacks and strengthen the AD software (Sec. 4.5.1).

4.1 Identification of Target Driving Missions

Driving missions define the functional requirements that AD software must follow for safe AV operation. To identify the missions, we use the standards developed by National Highway Traffic Safety Administration (NHTSA) [38].

4.1.1 Mission Metrics and Definitions

We converted the relevant NHTSA standards into seven formally verifiable missions that can be used to check whether an AV satisfies the safety requirements of its stated SAE level. As shown in Table 1, position-based missions ($\texttt{Stay}_{\texttt{Lane}}$ and $\texttt{Drive}_{\texttt{Allowed}}$) define the areas that the $Victim_{car}$ must not drive in (e.g., pedestrian sidewalks, one-way streets). Collision-based missions ($\texttt{Follow}_{\texttt{Car}}$, $\texttt{React}_{\texttt{SO}}$, $\texttt{React}_{\texttt{LC}}$, $\texttt{React}_{\texttt{EV}}$, $\texttt{React}_{\texttt{MO}}$) define the conditions where the $Victim_{car}$ must respond and prevent possible collisions.

To formally express the missions, we define two metrics: *distance* (Dist) for position-based missions and *time to collision* (TTC) for collision-based missions.

Distance (Dist). With the Dist metric, we evaluate the correct execution of $\texttt{Stay}_{\texttt{Lane}}$ and $\texttt{Drive}_{\texttt{Allowed}}$ missions. Dist measures the shortest distance from the $Victim_{car}$ to an area that it must not enter. For $\texttt{Stay}_{\texttt{Lane}}$, the area is the locations outside the $Victim_{car}$'s lane. For $\texttt{Drive}_{\texttt{Allowed}}$, the area is any location restricted for the $Victim_{car}$ to enter. These areas include one-way streets and sidewalks that are banned by law enforcement [44]. We compute the distance between the $Victim_{car}$ (VV) and an area as:

$$\texttt{Dist}(\texttt{VV},\texttt{area}) = \min\{\sqrt{(\texttt{VV.x}-\texttt{area.x})^2+(\texttt{VV.y}-\texttt{area.y})^2}\} \quad (1)$$

where $Victim_{car}.\texttt{x}, Victim_{car}.\texttt{y}$ are the coordinates of the $Victim_{car}$, and area.x, area.y represent the coordinates of the position in the restricted area closest to the $Victim_{car}$.

Fig. 3-❶ illustrates our computation of the shortest distance (the solid purple arrow) from the $Victim_{car}$ to the restricted area (sidewalk) and to the lane (blue dashed arrow).

Time to Collision (TTC). TTC defines the remaining time before the $Victim_{car}$ and another object (e.g., vehicle or pedestrian) collide if their current velocities are maintained [42, 69].

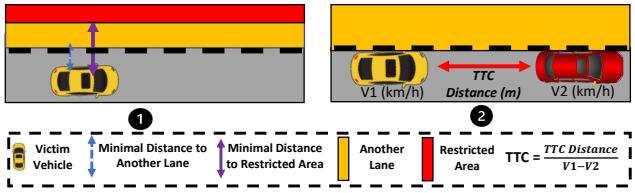

Figure 3: Illustration of computing Distance (Dist) and Time to Collision (TTC) metrics in an attack scene.

We compute the TTC between the $Victim_{car}$ and an agent (a), where pos is the position of the $Victim_{car}$ and the agent, and v is their velocity, as follows:

$$\texttt{TTC}(Victim_{car},\texttt{a}) = \frac{\texttt{pos}(Victim_{car})-\texttt{pos}(\texttt{a})}{\texttt{v}(Victim_{car})-\texttt{v}(\texttt{a})} \quad (2)$$

Fig. 3-❷ shows our TTC computation between $Victim_{car}$ (yellow, on the left) and another vehicle (red, on the right). If TTC becomes lower than a predetermined threshold value, a traffic crash will likely occur, as detailed below.

4.1.2 Formal Representation of Missions

We encode Dist and TTC metrics in the linear temporal logic (LTL) formulas of the missions as conditions, as shown in the "LTL Formula" column of Table 1. We leverage LTL for formalization because it (1) is a well-established specification language [48], (2) enables using various tools for parsing the formulas and checking if they are satisfied or violated on the AV's behavior [66], and (3) allows our formulas to be generalized to other testing frameworks (e.g., [65]).

To illustrate, we define $\square(\texttt{Dist}(\texttt{av},\texttt{current-lane}) > 0)$ for $\texttt{Stay}_{\texttt{Lane}}$ and $\square(\texttt{TTC}(\texttt{vv},\texttt{fc}) > \tau_h + \tau_v)$ for $\texttt{Follow}_{\texttt{Car}}$, where: $\square$ means "always", τ_h is the threshold for human reaction time, and τ_v is the time it takes for a vehicle to stop when it applies full brake. $\texttt{Dist}(\texttt{vv},\texttt{current-lane})$ defines the minimum distance from the $Victim_{car}$ (vv) to its lane marks (current-lane). When the Dist becomes 0, it means the $Victim_{car}$ has moved out of its lane. $\texttt{TTC}(\texttt{vv},\texttt{fc})$ quantifies the time it would take for the $Victim_{car}$ (vv) to collide with the front car (fc) if both vehicles keep their current velocity. If $\texttt{TTC}(\texttt{vv},\texttt{fc})$ is less than $\tau_h + \tau_v$, a collision is likely to happen even if the driver reacts as soon as possible by applying full brake. We define τ_h based on prior work [27, 41, 64], and τ_v based on the vehicle's official documentation on its braking performance (Sec. 6).

Table 2: Physical constraints that the adversary should comply with.

Physical constraint ID	Explanation	LTL Formula
Obey_Signals	Obey the traffic signals and signs.	$\square(\neg \texttt{break}(\texttt{traffic\_signal}))$
NoWrong_Way	Do not drive on the wrong side of the road.	$\square(\texttt{direction}(\texttt{AV}) == \texttt{direction}(\texttt{road}))$
NoTwo_Feet	Do not brake and increase throttle at the same time.	$\square\neg(\texttt{throttle} \wedge \texttt{brake})$
NoExceeding_Operations	Do not issue excessively high commands	$\square(\texttt{throttle} < 0.6 \wedge \texttt{brake} < 0.6 \wedge \lvert\texttt{steer}\rvert < 0.6)$
NoCrash_Traffic	Avoid accidents with the other vehicles in traffic.	$\square(\texttt{TTC}^{\dagger}(\texttt{AV}, \texttt{obj}) > 0)$
SafeDist_FollowVehicle	Keep safe distance while following the front vehicle.	$\square(\texttt{TTC}(\texttt{AV}, \texttt{front-car}) > \tau^{*})$
SafeDist_LaneChange	Keep safe distance while lane changing.	$\square(\texttt{TTC}(\texttt{AV}, \texttt{cut-in-car}) > \tau)$

† as defined TTC threshold in Section 4.1.2, * as defined in Table 1.

4.2 Attack Scene Initialization

We initialize an attack scene that we use to generate adversarial commands and evaluate the $Victim_{car}$'s missions. In each attack scene, $Victim_{car}$ and $Attack_{car}$ travel on a map modeled after a real-world environment (e.g., a specific city street). Initializing different attack scenes allows ACERO to define scenes specific to a mission and assess whether a discovered adversarial trajectory causes mission violations in different weather and traffic conditions.

Traffic Conditions. The traffic conditions and the behavior of other agents, such as vehicles, pedestrians, cyclists, and static objects, are critical factors in creating realistic attack scenes. We create such agents based on the $Victim_{car}$'s missions. If a mission specifies a set of specific agents, we spawn such agents at appropriate locations while evaluating the missions; otherwise, they are omitted. For example, $\texttt{React}_{\texttt{MO}}$ requires the $Victim_{car}$ to respond to pedestrians. Therefore, we spawn pedestrians in reasonable locations, such as on sidewalks or within a marked crosswalk.

Weather Conditions. Weather conditions are critical for the $Victim_{car}$'s missions as they affect the feature space of AD software's perception module. Unfavorable weather conditions may increase the likelihood of specific attack maneuvers to be successful. For example, a lower sun altitude may disturb the vehicle's front camera and cause an accident. Thus, to generate weather conditions, we use preset weather conditions (e.g., rainy, sunny) and data-driven distributions [42] based on four parameters: (*a*) sun altitude, (*b*) cloudiness, (*c*) ground precipitation, and (*d*) air precipitation (See Appendix A).

4.3 Adversarial Commands

Adversarial commands define the control directives ACERO issues to the $Attack_{car}$ in an attack scene. These commands are: (*i*) throttle (th), (*ii*) brake (b), and (*iii*) steering angle (s). We represent each command with $\texttt{Com} = \{\texttt{th}, \texttt{b}, \texttt{s}\}$.

4.3.1 Physical Constraints on Adversarial Commands

Table 2 presents the seven constraints that we formally define to ensure the $Attack_{car}$'s safety and low liability. The Obey_Signals and NoWrong_Way constraints ensure the $Attack_{car}$ obeys common traffic laws and drives at the correct side of the road. The NoTwo_Feet constraint prevents the $Attack_{car}$ from applying brake and throttle simultaneously as it can cause the $Attack_{car}$ to lose control. NoExceeding_Operations ensures the $Attack_{car}$ does not apply excessive commands (e.g., hard brakes and full throttle), preventing the brake-checking behavior and ensuring $Attack_{car}$'s safety. NoCrash_Traffic prevents the $Attack_{car}$ from colliding with the $Victim_{car}$ and other traffic agents. Lastly, SafeDist_FollowVehicle and SafeDist_LaneChange ensure the $Attack_{car}$ keeps a safe distance with nearby vehicles, minimizing the collision risk.

We note we include two universal constraints for traffic rules (Obey_Signals and NoWrong_Way). These constraints can be easily extended based on the local traffic regulations. Additionally, disabling one or more physical constraints can result in discovering more adversarial maneuvers. However, this could jeopardize the attacker's safety and prevent them from denying malicious intent and avoiding responsibility.

4.3.2 Enforcing Physical Constraints

We enforce NoExceeding_Operations by sampling throttle, brake, and steer values from $[-0.6, 0.6]$, where 1 is the highest value one can issue. To enforce NoTwo_Feet, we sample a single value for throttle and brake commands (th/b), where negative values of th/b indicate braking and its positive values indicate increasing throttle. For Obey_Signals, we monitor the $Attack_{car}$'s physical states (i.e., speed, local position, and global position) and ensure that generated commands respect the relevant traffic laws. For example, if the $Attack_{car}$'s speed exceeds the speed limit, we do not increase the throttle. For NoWrong_Way, we monitor the $Attack_{car}$'s position and lane information to prevent constraint violations.

To enforce the remaining physical constraints, we compute the simulation's state after issuing a command and ensure that the constraints are maintained. For example, we check that a safe distance has been maintained and the $Attack_{car}$ has not crashed. To achieve this, one might consider a position-based algorithm to enable $Attack_{car}$ to move to valid locations that comply with the constraints before issuing adversarial commands. Yet, this is a challenging task since it requires complex physical modeling of both the $Attack_{car}$ and all other agents in the scene. Therefore, to address this challenge, we design and implement a rewind function (Detailed in Sec. 4.4.4), which restores the scene to the last prior state before the constraint-violating command was issued. From there, a new command is generated that adheres to the constraints.

Algorithm 1 Adversarial Trajectory Generation

Input: $Attack_{car}$ (av), $Victim_{car}$ (vv), Max number of commands (maxr)
Output: Adversarial Trajectory (AT)

1: **function** AdvTrajGen(av, vv, maxr)
2: AT = [], counter = 0, AGV = (0,0)
3: currRob = calc_rob (current_tick) ▷ current simulation state
4: **while** currRob ≥ 0 **do**
5: AGV, V_{av} = AdvCommandGen(AGV, av, vv)
6: av.Apply_Commands(V_{av}) ▷ Run adv. command
7: currRob = calc_rob (current_tick)
8: AT.append(vv), counter = counter + 1
9: **if** counter > maxr **then** break
10: **end if**
11: **end while**
12: **return** AT
13: **end function**

4.4 Adversarial Trajectory Generation

ACERO generates a set of adversarial commands to guide the $Attack_{car}$'s maneuvers and cause the $Victim_{car}$ violate its missions. We refer to the set of executed adversarial commands as the *adversarial trajectory*. The trajectory includes a set of positions, $P = \{pos_1, \ldots pos_k\}$, where pos_i is the $Attack_{car}$'s position after executing i^{th} command, and k is the number of commands required to cause the $Victim_{car}$ violate its mission.

Algorithm 1 details the adversarial trajectory generation process. It takes three inputs: (1) the $Attack_{car}$'s physical states (speed, position, orientation), (2) the $Victim_{car}$'s physical states, and (3) the max number of commands. It first calculates the $Victim_{car}$'s *robustness value* at the initial scene (Line 3). The robustness quantifies how well the $Victim_{car}$ satisfies its intended missions, as detailed in Sec. 4.4.1. Until the $Victim_{car}$'s robustness value reaches 0, it calls AdvCommandGen (Line 5) to generate the adversarial trajectory. If the number of adversarial commands reaches the user-defined limit before $Victim_{car}$'s robustness becomes 0, ACERO restarts the algorithm (Line 9-10) with different initial positions and velocities for the $Attack_{car}$ and $Victim_{car}$.

4.4.1 Attack Robustness Computation

To guide adversarial trajectory generation, we provide attack robustness metrics that define how well the $Victim_{car}$'s physical states (velocity and location) satisfy its safety missions. The negative robustness values indicate the victim's mission has failed, and positive values indicate the mission is satisfied. We define two different methods for computing robustness metrics, one for collision-based missions and another for position-based missions.

We use TTC - τ to compute the $Victim_{car}$'s robustness in satisfying its collision-based missions. TTC - τ computes if the time to collision is higher than the threshold τ, which defines the time to react to hazards based on AD software's SAE level [27, 41, 64] (as detailed in Sec. 4.1.2). If the TTC value becomes lower than the reaction time, the robustness becomes negative, indicating that a collision will likely happen. Depending on the safety mission, we define τ based on τ_h, the threshold for human reaction time, and τ_v, the amount of time for the $Victim_{car}$ to come to a complete stop when it applies full brake. Thus, for $Follow_{Car}$, $\tau = \tau_h + \tau_v$. For $React_{SO}$, $React_{LC}$, $React_{EV}$, and $React_{MO}$, $\tau = \tau_v$.

We use $Dist(Victim_{car}, area)$ to compute the robustness of position-based missions, where area represents the set of positions the $Victim_{car}$ should avoid. For $Stay_{Lane}$, it is the area outside of the $Victim_{car}$'s lane, while for $Drive_{Allowed}$, it represents the legally forbidden areas (e.g., sidewalks). When Dist becomes 0, it indicates that the $Victim_{car}$ entered an area it should avoid, and thus, the mission is violated.

ACERO assesses $Victim_{car}$'s each mission separately by leveraging guidance from the collision- or position-based robustness metric. This prevents ACERO from creating *conflicting guidance* on the adversarial commands. For instance, if we evaluate multiple missions simultaneously, the TTC metric would attempt to optimize a trajectory for the $Attack_{car}$ to make the $Victim_{car}$ collide (e.g., with a third-party vehicle); while the Dist metric would guide the $Attack_{car}$ to make the $Victim_{car}$ to steer towards restricted areas. Therefore, these goals are mutually exclusive, making it counterproductive to attempt to achieve both of them at once.

Algorithm 2 Adversarial Command Generation

Input: Attack Guidance Vector (AGV), $Attack_{car}$ (av), $Victim_{car}$ (vv)
Output: New AGV (AGV_n), Adversarial Command (V_{av})

1: **function** AdvCommandGen(AGV, av, vv)
2: robs = { }, dists = { }, prev_rob = 0
3: $RP_{prev}(vv, av, t_n)$ = pos(a, t_n) - pos(v, t_n)
4: commands = Candidate_Generation(AGV)
5: **for** command $\in$ commands **do**
6: SIM.execute(av, command)
7: **if** PC_Violation(current_tick) == 0 **then**
8: robs.append(calc_rob(current_tick))
9: dists.append(Dist(av, vv))
10: **end if**
11: RewindScene()
12: **end for**
13: min_rob = Min(robs)
14: **if** Count(min_rob $\in$ robs) = 1 **then**
15: V_{av} = commands[arg min robs]
16: **else**
17: min_rob_commands = commands[arg min robs]
18: min_rob_dists = dists[arg min robs]
19: V_{av} = min_rob_commands[arg min min_rob_dists]
20: **end if**
21: SIM.execute(av, v_{av})
22: $RP_{new}(vv, av, t_{n+1})$ = pos(av, t_{n+1}) - pos(vv, t_{n+1})
23: AGV_n = $RP_{new}(av, vv, t_{n+1})$ - $RP_{prev}(av, vv, t_n)$
24: **return** AGV_n, V_{av}
25: **end function**

4.4.2 Adversarial Command Generation

Algorithm 2 details the AdvCommandGen process. It first generates an initial command for the $Attack_{car}$ by conducting a grid search on an attack region that includes the areas it can reach by executing physically feasible control commands. Fig. 4-❶ shows this process. The left-most region is the result of max left steer, the right-most region is the result of max right steer, the top region is the result of a max throttle, and the bottom region is the result of a max brake. ACERO divides the attack region to $n \times n$ grids, where n is a system parameter.

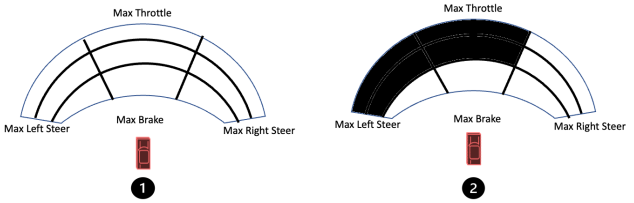

Figure 4: An illustration of adversarial trajectory generation. ❶ shows the candidate area on the reachable region for the initial adversarial command. ❷ shows the candidate area selected with an attack guidance vector guided by robustness.

Higher values of n lead to a fine-grained search but incur a higher computation time and its lower values lead to a more computationally efficient but coarse-grained search.

ACERO then randomly selects a position from each grid and generates a *candidate adversarial commands set* that includes commands steering the $Attack_{car}$ to these positions (Line 4). After sending each command, ACERO checks if $Attack_{car}$'s physical constraints are violated (Line 7). If a constraint is violated, that command is removed from the candidate set. Otherwise, it computes the attack robustness and the distance between the $Attack_{car}$ and $Victim_{car}$ (Lines 8-9). It then restores the simulation to the previous state and sends the next command in the candidate set (Line 11).

After iterating through all commands, ACERO selects the one that minimizes the attack robustness as the $Attack_{car}$'s *initial* command (Line 13). If multiple commands reduce the $Victim_{car}$'s robustness in the same amount, it chooses the command that minimizes the distance between the $Attack_{car}$ and $Victim_{car}$ (Line 17-19) to place the $Attack_{car}$ in a better position for further robustness reduction.

4.4.3 Attack Guidance Vector Generation

ACERO computes an *attack guidance vector* (AGV), which is the relative position change between the $Attack_{car}$ and $Victim_{car}$ that causes the $Victim_{car}$ to have a maximum decrease in robustness (Lines 21-24). The attack guidance vector guides the $Attack_{car}$ to move to the locations that cause similar relative position changes with the $Victim_{car}$ as those that have previously decreased robustness. To represent the AGV, we first define *relative position* (RP) as follows:

$$\texttt{RP}(\texttt{vv},\texttt{av},\texttt{t}) = \texttt{pos}(\texttt{av},\texttt{t}) - \texttt{pos}(\texttt{vv},\texttt{t}) \quad (3)$$

The RP is computed as the $Victim_{car}$'s position vector at time t subtracted from the $Attack_{car}$'s position vector. We next define the AGV as the change between two consequent relative positions (at time t and $\texttt{t}-1$):

$$\texttt{AGV}(\texttt{t}) = \texttt{RP}(\texttt{vv},\texttt{av},\texttt{t}) - \texttt{RP}(\texttt{vv},\texttt{av},\texttt{t}-1) \quad (4)$$

Particularly, ACERO first computes the relative position vectors at time $\texttt{t}-1$ and t. It then subtracts the relative position vector at time $\texttt{t}-1$ from the vector at time t to obtain the attack guidance vector. To generate the next adversarial command, instead of sampling from every grid, ACERO samples its next candidate adversarial commands from the grids that the AGV's direction points to (Line 4). For instance, in Fig. 4-❷, the AGV is (-1, 1), indicating turning left and throttling. Therefore, ACERO samples the candidate adversarial commands only from the grids that include turning left and throttling (marked with black color in Fig. 4-❷).

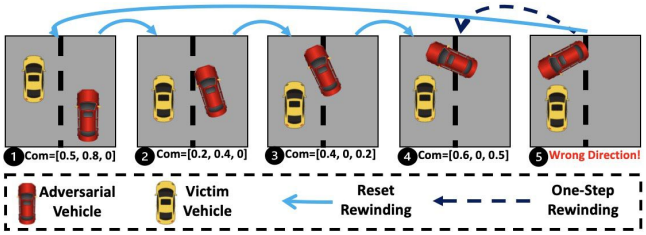

Figure 5: One-step rewinding and reset-rewinding illustration.

4.4.4 Rewinding the Scene

Rewinding is the process of restoring the $Victim_{car}$'s and $Attack_{car}$'s physical states to their condition before the last adversarial command was executed.

ACERO conducts rewinding for two purposes. First, after each adversarial command, it rewinds the scene to send the next command while searching for the one that minimizes the $Victim_{car}$'s robustness. Second, it rewinds the scene to eliminate the cases when one of the $Attack_{car}$'s physical constraints is violated. For example, in Fig. 5, ACERO detects a physical constraint violation on the $Attack_{car}$ at time ❺, and rewinds the scene to time ❹ to sample another command.

One potential method is *one-step rewinding*, where one rewinds directly to the scene when the last command has not been executed (black dashed arrow in Fig. 5). Although simple to implement, this technique causes inconsistencies in the perception modules (sensor buffers) of AD software in specific simulators and disrupts the decision of ML models (e.g., RNNs) that take multiple perception frames as input.

To address this issue, we implement *reset-rewinding*, in which ACERO resets the scene to the initial one and repeats the historical commands until the previous scene (blue solid arrow in Fig. 5). This technique restarts the simulator to clear any prior input frames from the AD software's memory, ensuring consistent AD behavior.

4.5 Attack Guide and Clustering

ACERO outputs an *attack guide*, which contains the complete information to launch an attack (a sample attack guide is given in Appendix B).

4.5.1 Attack Clustering

After each attack guide for a mission is obtained, we cluster adversarial trajectories of successful attacks to group trajectories sharing similar adversarial maneuvers. Clustering the trajectories (1) quantifies the diversity of discovered attacks,

Algorithm 3 Unique Attack Path Discovery

Input: Path 1 (P1), Path 2 (P2), Initial Position of the $Victim_{car}$ (ξ)
Output: Path Similarity (δ)

```
1:  function PathSim(P1, P2, ξ)
2:      windowsize = len(P1) − len(P2), simset = []
3:      for i in range(0, windowsize) do
4:          P1′ = P1[i : i + len(P2)]
5:          δ = CosineSim(P1′, P2, ξ)
6:          simset.append(δ)
7:      end for
8:      return min(simset)
9:  end function
10: function CosineSim(P1, P2, ξ)
11:     T1, T2 = Trajectory(P1,P2, Size(P1)), δ = 0
12:     for i in range(0, size(T1)) do
13:         δ += (cos(T1[i], (P1[i] − ξ)) - cos(T2[i], (P2[i] − ξ))
14:     end for
15:     return δ
16: end function
```

and (2) helps identify the root cause of attacks, which is often the same for attacks in the same cluster (Sec. 6), to guide developers to strengthen the AD software.

Computing Trajectory Similarity. Algorithm 3 shows our approach to computing the path similarity between two successful attacks. We cluster adversarial trajectories into groups based on their similarity score. Our algorithm takes two attack trajectories (P1 and P2) as input. If the trajectories have different lengths, we compare the trajectories in same-length segments. Thus, we first compute a window size as the difference between trajectory lengths (Line 2). We use cosine similarity (CosineSim) to compute the distance between the shorter path and longer trajectory's every sub-path, which has the same length as the shorter trajectory (Line 3-7). Cosine similarity measures the similarity of trajectories irrespective of their lengths [31]. We offset each path based on the relative position between the $Victim_{car}$'s initial position to account for $Attack_{car}$'s different initial positions (Line 11).

We then compute the similarity between the $Victim_{car}$'s initial position and each position of the $Attack_{car}$'s path (Line 12-14). Lastly, we output the minimum similarity value as the trajectory similarity from the set of similarities computed for each sub-path in the sliding window. We group the successful attacks with a given threshold, i.e., paths with a distance lower than the threshold are assigned to the same group.

4.5.2 Determining Attack Reproducibility

To verify an attack's reproducibility, we evaluate whether an attack can be conducted in different weather conditions and whether ACERO's attack guide is successful with a margin of error in the adversarial trajectory.

Weather Conditions. Weather conditions can play a major factor in causing the $Victim_{car}$ to fail its mission when exposed to $Attack_{car}$'s driving patterns. For instance, recent works showed that direct sunlight or heavy rain affects the AV's perception inputs and changes its control decisions [42, 64]. While an attack that is reproducible under certain weather conditions is reasonable, successful attacks in any weather condition are more attractive for adversaries.

Therefore, we replay a successful attack from each cluster using random weather conditions generated from a data-driven model to see whether the attack is successful in each condition. We consider a variety of weather conditions, such as daytime, different rainfall amount, sunshine, and cloud cover, as detailed in Appendix A.

Trajectory Replication Error. While we consider the attacker can control the $Attack_{car}$ with digital inputs (e.g., steering and throttle), a human operator may make mistakes in repeating the exact trajectory outputted by ACERO.

Therefore, we introduce errors to the adversarial trajectory and check whether the attack is still successful. For each attack command that consists of throttle (th), break (b), and steer (s), we add a uniformly distributed error to the command. Particularly, we sample the errors from the ranges $[-\mathtt{th}*\varepsilon, \mathtt{th}*\varepsilon]$, $[-\mathtt{b}*\varepsilon, \mathtt{b}*\varepsilon]$, and $[-\mathtt{s}*\varepsilon, \mathtt{s}*\varepsilon]$, where ε represents the error rate. We set ε to realistic values in our evaluation based on the expected errors of a human driver (See Sec. 6.2.2).

5 Implementation

Simulator. CARLA [15] and LGSVL [29] are the most popular AV simulators. We deployed ACERO into CARLA [15] because LGSVL [29] does not support sending commands to the $Attack_{car}$ during simulation (i.e., requires defining all commands in advance) while running the $Victim_{car}$ with AD software. Thus, it is infeasible to generate adversarial commands and implement an efficient rewinding approach in LGSVL.

Driving Software. We evaluate ACERO on two AD systems, openpilot 0.8.6 [11] (SAE Level 2) officially integrated into CARLA 0.9.11 and Autoware 1.14.0 [23] (SAE Level 4) integrated into CARLA 0.9.1. Apollo [3], however, does not have an ACERO-compatible implementation. In Appendix C, we provide implementation details of openpilot and Autoware, and outline our efforts to integrate Apollo into ACERO.

Attack Scene Initialization and Adversarial Trajectory Generation. We implement ACERO with CARLA's Python APIs [7] that allows us to define attack scenes and directly obtain the physical states of the $Victim_{car}$, $Attack_{car}$, and other agents. To create attack scenes, we create template files for each mission, which can be adapted by any user to suit their testing purposes. The file contains the scene configuration parameters, including weather, traffic conditions, and initial positions/speeds of $Victim_{car}$ and $Attack_{car}$ (Sec. 4.2). We then initialize the attack scenes, configure the $Victim_{car}$ with either openpilot or Autoware, and run the adversarial trajectory generation algorithm with physical constraints to output attack guides. We write 960 lines of code (LoC) for Autoware and 850 LoC for openpilot in Python to generate attack scenes and adversarial trajectories.

Clustering Adversarial Trajectories. For attack guide clustering and verifying the attacks are successful under random weather conditions, we run a worklist-based algorithm to clus-

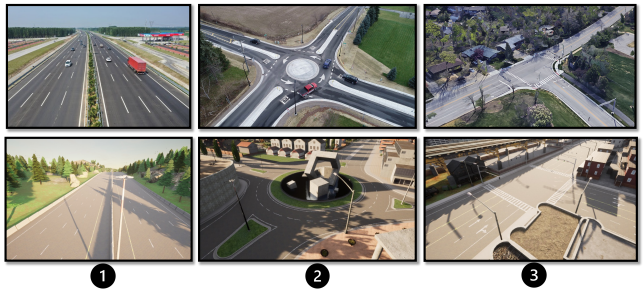

Figure 6: Three types of roads in real life (Top) vs. in CARLA (Bottom) ❶ Highway; ❷ Roundabout; ❸ Intersection.

ter the adversarial trajectories to different groups. We then randomly select an attack from each group and use 14 preset weather conditions (see Appendix A) to determine whether the attack trajectory is successful in causing the vehicle to fail its mission. We write 178 LoC for attack clustering and 124 LoC for verifying weather reproducibility.

Mission Identification. We measure the time required by a knowledgeable user to identify the missions in Table 1 and express them as LTL formulas. Particularly, two authors spent a total of 2.5 hours identifying Level 2-4 AD missions.

6 Evaluation

We evaluate ACERO's effectiveness in identifying attacks against the safety missions (Table 1) of openpilot and Autoware. For openpilot (SAE Level 2 AD), we use ACERO to find attacks against the $\texttt{Stay}_{\texttt{Lane}}$ and $\texttt{Follow}_{\texttt{Car}}$ missions. For Autoware (SAE Level 4 AD), we evaluate its $\texttt{Drive}_{\texttt{Allowed}}$, $\texttt{React}_{\texttt{SO}}$, $\texttt{React}_{\texttt{LC}}$, $\texttt{React}_{\texttt{EV}}$, and $\texttt{React}_{\texttt{MO}}$ missions. We present our results by focusing on the following research questions.

RQ1 What is the attack success rate (i.e., the percentage of the number of mission violations on the number of test cases) for each mission? (Sec. 6.2)

RQ2 What is the percentage of attacks that are reproducible in different weather conditions? (Sec. 6.2.1)

RQ3 What is the reproducibility rate of the attacks when the attack trajectory is not strictly followed? (Sec. 6.2.2)

RQ4 What are the root causes of the attacks? (Sec. 6.3)

RQ5 What is the attack success rate without using the robustness guidance? (Sec. 6.4.1)

RQ6 How does ACERO perform against other AV testing systems? (Sec. 6.4.2)

RQ7 What is ACERO's execution time? (Sec. 6.5)

We perform our experiments on three desktops with Intel i7-10700K CPU, 32 GB RAM, GeForce RTX 2080 Ti GPU, running Ubuntu 20.04.

6.1 Experiment Setup

We generate a set of scenes for the $Attack_{car}$ to conduct attacks and evaluate $Victim_{car}$'s each mission. Creating attack scenes requires: (1) determining the road type, (2) including other traffic agents on the map, and (3) setting weather conditions. We achieve these steps via ACERO's attack scene initialization component introduced in Sec. 4.2.

The first step for attack scene initialization is to configure realistic maps to serve as a location appropriate for evaluating each mission. We use three different maps (provided by CARLA [8]), representative of common real-world scenarios, as shown in Fig. 6. These maps include a highway, a roundabout, and a T-intersection.

We evaluate the position-based missions ($\texttt{Stay}_{\texttt{Lane}}$, $\texttt{Drive}_{\texttt{Allowed}}$) on roundabouts, since lane changes are not allowed in them [21] and a mission violation can cause more severe consequences (e.g., collisions with traffic or the center island) compared to other maps. We evaluate three collision-based missions ($\texttt{React}_{\texttt{SO}}$, $\texttt{React}_{\texttt{EV}}$, $\texttt{React}_{\texttt{MO}}$) on intersections because this road type enables simulating the behavior of a greater variety of traffic agents, e.g., pedestrians, traffic signs, stopped vehicles/obstacles, and encroaching vehicles. Lastly, we evaluate $\texttt{Follow}_{\texttt{Car}}$ and $\texttt{React}_{\texttt{LC}}$ missions on the highway because the missions for car-following and responding to cut-in vehicles are critical in high-speed traffic.

We next configure the behavior of other traffic agents and the weather conditions on each map in two different ways. In Exp-ⓐ, we initialize the scene with only the traffic agents strictly required to evaluate a mission (e.g., another vehicle for $\texttt{Follow}_{\texttt{Car}}$) and the default CARLA weather parameters. In Exp-ⓑ, we initialize the scene with additional traffic agents (i.e., a road sign, a pedestrian, a vehicle, and a cyclist) and sample the weather conditions from a set of real-world data-driven distributions. We present two example attack scenes in Fig. 12 in Appendix D.

6.2 Effectiveness

We run 500 test cases for each mission using the generated attack scenes and evaluate the $Victim_{car}$'s missions (**RQ1**). In each test case, we randomly select the $Attack_{car}$ from 27 different models with 3 car types: sedan, SUV, and truck. We also randomly select the $Attack_{car}$'s and $Victim_{car}$'s initial positions and speeds in each test case to discover different adversarial trajectories. We note that, for the $Victim_{car}$, we use the vehicle configuration that openpilot (Tesla M3) and Autoware (Toyota Prius) official implementations provide. This is because, although it is possible to integrate different $Victim_{car}$ types and models into CARLA, openpilot and Autoware issue commands specifically for their default vehicle, based on the car model's physical properties (e.g., mass, horsepower, height), which are hardcoded in these two software packages.

Table 3 details the number of violations for each mission and the physical consequences the adversary achieves with Exp-ⓐ and Exp-ⓑ. The "Mission Violation" column shows the number of attack cases where the $Attack_{car}$ causes the $Victim_{car}$ to violate its mission listed in the "Mission ID" column. We observe three physical consequences when the $Victim_{car}$ violates its missions, as shown in the "Physical Consequence" column: (1) hitting another vehicle, (2) hit-

Table 3: Percentages of the $Victim_{car}$'s mission violations identified by ACERO and their physical consequences.

Mission ID	Mission Violation	Physical Consequence			Attack Vehicle Type			Number of Attack Clusters
		Hitting Another Vehicle	Hitting a Pedestrian/Cyclist	Hitting Static Objects	Sedan	SUV	Truck	
Exp-ⓐ: *Attack scene without additional traffic and default weather conditions*								
$\texttt{Stay}_{\texttt{Lane}}$	63 (12.6%)*	n/a†	n/a	6 (1.2%)	28 (44.4%)	20 (31.7%)	15 (23.8%)	2
$\texttt{Drive}_{\texttt{Allowed}}$	0 (0%)	n/a	n/a	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0
$\texttt{Follow}_{\texttt{Car}}$	40 (8%)	40 (8%)	n/a	0 (0%)	4 (10%)	3 (7.5%)	33 (82.5%)	4
$\texttt{React}_{\texttt{SO}}$	8 (1.6%)	n/a	n/a	8 (1.6%)	1 (12.5%)	3 (37.5%)	4 (0.5%)	2
$\texttt{React}_{\texttt{LC}}$	42 (8.4%)	42 (8.4%)	n/a	0 (0%)	21 (50%)	17 (40.5%)	4 (9.5%)	2
$\texttt{React}_{\texttt{EV}}$	9 (1.8%)	9 (1.8%)	n/a	0 (0%)	1 (11.1%)	2 (22.2%)	6 (66.7%)	3
$\texttt{React}_{\texttt{MO}}$	17 (3.4%)	n/a	17 (3.4%)	0 (0%)	0 (0%)	0 (0%)	17 (100%)	1
Exp-ⓑ: *Attack scene with additional traffic and sampled weather conditions*								
$\texttt{Stay}_{\texttt{Lane}}$	38 (7.6%)	0 (0%)	0 (0%)	4(0.8%)	11 (29%)	20 (52.6%)	7 (18.4%)	3
$\texttt{Drive}_{\texttt{Allowed}}$	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0
$\texttt{Follow}_{\texttt{Car}}$	78 (15.6%)	78 (15.6%)	0 (0%)	0 (0%)	13 (16.7%)	46 (59%)	19 (24.3%)	4
$\texttt{React}_{\texttt{SO}}$	6 (1.2%)	6 (1.2%)	0 (0%)	0 (0%)	0 (0%)	1 (16.7%)	5 (83.3%)	1
$\texttt{React}_{\texttt{LC}}$	15 (3%)	15 (3%)	0 (0%)	0 (0%)	3 (20%)	5 (33.3%)	7 (46.7%)	2
$\texttt{React}_{\texttt{EV}}$	13 (2.6%)	12 (2.4%)	1 (0.2%)	0 (0%)	2 (15.4%)	3 (23.1%)	8 (61.5%)	2
$\texttt{React}_{\texttt{MO}}$	12 (2.4%)	1 (0.2%)	11 (2.2%)	0 (0%)	8 (66.7%)	0 (0%)	4 (33.3%)	2

† n/a indicates the physical consequence cannot occur as the required traffic agents are not in the attack scene. * Percentages in parentheses are relative to the total number of experiments.

ting a pedestrian or cyclist, and (3) hitting static objects. For instance, in Exp-ⓐ, ACERO discovers 8 cases where the $Attack_{car}$ causes the $Victim_{car}$ to crash to static objects, violating $\texttt{React}_{\texttt{SO}}$. Overall, ACERO discovered 341 safety violations by the $Victim_{car}$ (4.87% out of 7000). Out of these, 57.8% of the violations result in the $Victim_{car}$ colliding with a third-party vehicle, 8.5% cause it to hit a pedestrian or cyclist, 7% cause it to hit a static object (e.g., road sign).

We note that not all mission violations lead to physical consequences. For instance, In Exp-ⓐ, we found 63 $\texttt{Stay}_{\texttt{Lane}}$ mission violations where the $Attack_{car}$ causes the $Victim_{car}$ to move out of its lane. Yet, only 6 of those cases resulted in a physical consequence (hitting a static object).

ACERO could not find any safety violations for Autoware's $\texttt{Drive}_{\texttt{Allowed}}$ mission. This is because Autoware's planning module creates the route (a set of waypoints) to the destination before departure and does not change the route except when a new destination is set.

Impact of Initial Attack Setup on Attack Success Rate. We analyze how the initial attack setup (initial speed, position, and $Attack_{car}$ model) impacts the attack success rate. We find that the attack success rate becomes higher in specific attack setups depending on the mission the $Attack_{car}$ targets. For instance, in $\texttt{React}_{\texttt{LC}}$ Exp-ⓐ, 69.8% of successful attacks (30 out of 42) had $Attack_{car}$'s initial position within 12 and 17 meters from the $Victim_{car}$, and 64.3% of the successful attacks (27 out of 42) have an initial speed difference between the $Attack_{car}$ and $Victim_{car}$ within 0 to 10.8 km/h. This is because the $Attack_{car}$ crashes with the $Victim_{car}$ if they are initialized too close, leading to a `NoCrash_Traffic` constraint violation. In contrast, if the $Attack_{car}$ is initialized too far away from the $Victim_{car}$, it does not impact the $Victim_{car}$'s maneuvers. Similarly, for the initial speed, if the $Attack_{car}$'s initial speed is too high or too low from the $Victim_{car}$'s initial speed, it becomes very difficult for the $Attack_{car}$ to maintain a close distance with the $Victim_{car}$ to impact its movements.

Attack Clustering. The "Number of Attack Clusters" column in Table 3 shows the number of unique adversarial trajectories identified for each mission. ACERO discovered a total of 28 attack clusters. For instance, while assessing the $\texttt{React}_{\texttt{SO}}$ mission in Exp-ⓑ, we identified 4 clusters from the 78 mission violations discovered, indicating there are 4 unique trajectories an adversary can take to cause a violation.

We note that we use different path similarity thresholds for different missions when grouping the attack trajectories, because the attack scene setup is different for each mission. For example, when ACERO runs attacks on $\texttt{React}_{\texttt{LC}}$ in a highway scene, the traffic speed is faster than the attack scene in the city street. Thus, we need to consider the different scale of the adversarial trajectories, by using different thresholds.

Comparison of Attacks in Exp-ⓐ and Exp-ⓑ. We compare successful attacks in Exp-ⓐ and Exp-ⓑ to understand the impact of the additional traffic conditions. We found that the number of violations decreases by 9.4% in Exp-ⓑ. This is because the other vehicles in the attack scenes prevent the $Attack_{car}$'s ability to perform different maneuvers while it complies with the physical constraints.

We additionally observe a wider variety of physical consequences in Exp-ⓑ because there exist road signs, pedestrians, vehicles, and cyclists in the attack scenes that $Victim_{car}$ collides. For instance, while running ACERO on the $\texttt{React}_{\texttt{MO}}$ mission, in Exp-ⓐ, the $Victim_{car}$ only collides with pedestrians and cyclists as only these agents are present in the scene. When we include additional traffic agents for Exp-ⓑ, the $Victim_{car}$ crashes to the other vehicles as well.

6.2.1 Impact of Weather on Attack Reproducibility

We study whether the discovered attacks are reproducible under different weather conditions (**RQ2**). To do so, we randomly choose an attack from each cluster and use its attack guide to replay the same commands in 14 preset weather conditions available in CARLA (see Appendix A for details).

Fig. 7 shows the attack success rate for each attack cluster against openpilot ($\texttt{Follow}_{\texttt{Car}}$, $\texttt{Stay}_{\texttt{Lane}}$) and Autoware ($\texttt{React}_{\texttt{SO}}$, $\texttt{React}_{\texttt{LC}}$, $\texttt{React}_{\texttt{EV}}$, $\texttt{React}_{\texttt{MO}}$). For instance, we show four bars for openpilot's $\texttt{Stay}_{\texttt{Lane}}$ mission, each representing an attack cluster, where the first two are clusters from

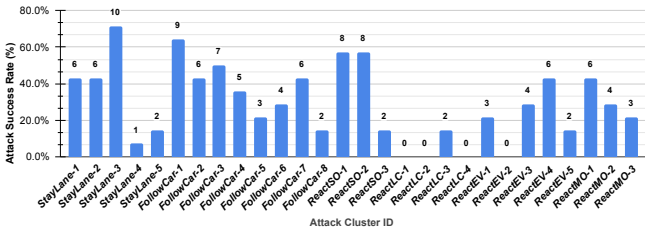

Figure 7: Attack success rate in multiple weather conditions.

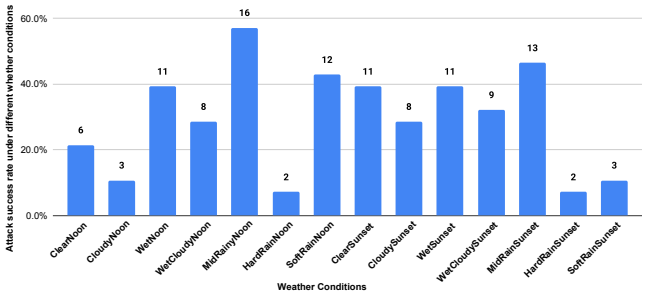

Figure 8: Attack reproducibility of each weather condition.

Exp-ⓐ and the other three are from Exp-ⓑ. We find that, on average, the attacks can be replayed in 36.8% of the weather conditions for openpilot and 22.9% for Autoware. We note that ACERO is able to find violations for missions $\texttt{React}_{\texttt{LC}}\texttt{-1}$, $\texttt{React}_{\texttt{LC}}\texttt{-2}$, $\texttt{React}_{\texttt{LC}}\texttt{-4}$, and $\texttt{React}_{\texttt{EV}}\texttt{-2}$ when the weather parameters are sampled from data-driven distributions. However, ACERO is unable to reproduce the same violations when using the 14 preset weather conditions.

We further analyze the attack reproducibility under specific weather conditions to observe if their impact is consistent on the $Attack_{car}$ and $Victim_{car}$. Fig. 8 presents the number of attack clusters that can be reproduced in each of the 14 preset weather conditions. For instance, the attack reproducibility is 21.4% under rainy weather conditions, and it increases to 29.1% in non-rainy conditions. By analyzing the driving logs, we find that the slight decrease in attack reproducibility under rainy and sunset weather conditions occurs because these conditions disrupt the $Attack_{car}$'s maneuvers, causing it to crash other vehicles and violate its physical constraints. In contrast, under non-rainy and noon weather conditions, the $Attack_{car}$ performs adversarial maneuvers more precisely, driving close to the obstacles and $Victim_{car}$, without hitting them.

6.2.2 Attack Reproducibility with Operator Error

To assess the reproducibility of attacks with an operator error, we randomly choose one attack from each cluster and replay it with a uniformly-distributed deviation added to each command, with a maximum error of 5% ($\varepsilon = 0.05$) of the original command (**RQ3**). We repeat each attack 10 times with random command deviations and verify if the attack is still successful. We consider a case to be reproducible if the commands with deviations applied satisfy the physical constraints and result in a mission failure. We found attacks with deviated commands are successful, on average, 25.38% of the time for openpilot and 26.67% for Autoware. This implies that even human operators unable to precisely follow the exact adversarial trajectories could still successfully conduct these attacks.

Table 4: Root causes of the $Victim_{car}$'s mission violations.

Mission ID	Driving Software	Root Cause		
		Blocking Vision	Perception Module Errors	Planning Module Errors
$\texttt{Stay}_{\texttt{Lane}}$	Openpilot	2*	3	0
$\texttt{Follow}_{\texttt{Car}}$	Openpilot	5	3	0
$\texttt{Drive}_{\texttt{Allowed}}$	Autoware	n/a	n/a	n/a
$\texttt{React}_{\texttt{SO}}$	Autoware	1	1	1†
$\texttt{React}_{\texttt{LC}}$	Autoware	0	2	2†
$\texttt{React}_{\texttt{EV}}$	Autoware	3	0	2
$\texttt{React}_{\texttt{MO}}$	Autoware	2	1	0

* Number of attack clusters † Due to the parameter misconfiguration

6.3 Root Cause Analysis

Four authors of this paper manually investigated the attack guide and driving logs to identify the root cause of the attacks(**RQ4**). As shown in Table 4, we identified three categories of root causes: (1) Blocking vision, (2) perception module error, and (3) planning module error. Although we did not observe the same adversarial trajectory causing a crash in both AV software, we found similar root causes for their mission violations, e.g., both AV software yield perception module errors, such as losing track of an obstacle.

Blocking Vision. We consider an attack's root cause is "Blocking Vision" if the $Victim_{car}$'s perception module output indicates that it failed to detect an obstacle or detected it close to the time of the crash ($\approx$ 1 second TTC). Such blocking of the vision sensor propagates into the $Victim_{car}$'s planning module and causes incorrect decisions, resulting in insufficient time to react to obstacles and eventually causing a crash.

We found seven attack clusters from openpilot and six clusters from Autoware fall into this root cause category. Since these attacks occur due to the $Victim_{car}$'s perception module being unable to detect obstacles, they can be prevented by installing additional vision sensors. For example, it will be harder for the $Attack_{car}$ to block both front and side cameras.

Perception Module Error. We consider an attack's root cause as "Perception Module Error" if the $Victim_{car}$'s perception module (*i*) misclassifies an object, (*ii*) loses track of obstacles, or (*iii*) incorrectly predicts the driving path of another vehicle. In such errors, although the $Victim_{car}$'s vision sensors are not blocked, the $Victim_{car}$'s perception module incorrectly classifies the $Attack_{car}$ when it is driving erratically (See the case studies in Sec. 6.3.1). Such misclassifications and inaccurate tracking pass wrong inputs to the planner module, which triggers the crash.

We found six attack clusters from openpilot and four clusters from Autoware fall into this root cause category. Since these attacks occur due to the ML models used in the $Victim_{car}$'s perception module, they might be prevented by improving their ML models. For example, the object tracker can be retrained with more smoothing trajectories to mitigate

Listing 1 Code for generating deceleration waypoints. The default value of `deceleration_range` is 0.

```
1  if (deceleration_range > 0 && stop_obstacle < 0) {
2      *obstacle = detectDecelerateObstacle(...);
3      if (obstacle < 0) return KEEP;
4      else return DECELERATE;
5  }
```

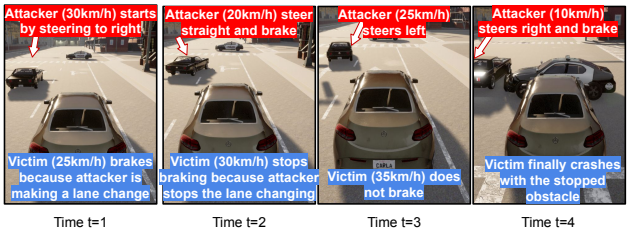

Figure 9: Case Study 1: $Attack_{car}$ causes Autoware to fail at avoiding a static obstacle, causing a collision for $Victim_{car}$.

the impact of $Attack_{car}$'s abrupt movements.

Planning Module Error. We consider the root cause is a "Planning Module Error" when (*i*) the $Victim_{car}$'s vision sensors are not blocked and (*ii*) the perception module outputs are correct, but the $Victim_{car}$ still crashes. In such cases, the $Attack_{car}$'s adversarial trajectory exploits a bug in the $Victim_{car}$'s planning module. This bug makes the $Victim_{car}$'s planning module fail to generate a collision-free trajectory, causing it to crash.

We found five clusters from Autoware fall into this root cause category. For example, the $Victim_{car}$ is not able to react to the hard brake of a front vehicle. While we investigated this case, we noticed that a misconfiguration bug in the planning module is the root cause. In particular, as shown in the simplified code snippet in Listing 1, the Autoware planning module skips adding a deceleration waypoint (Lines 2-4) if the `deceleration_range` parameter is set to zero (Line 1). We then set it to 3 and attack the $Victim_{car}$ against $\texttt{React}_{\texttt{SO}}$ using the exact same attack guide. With the updated parameter value, the $Victim_{car}$ reacts correctly to the $Attack_{car}$'s movements and prevents the crash.

6.3.1 Case Studies

Case Study 1 (Fig. 9) - *Failing to avoid a static object:* This attack [2] makes an Autoware $Victim_{car}$ violate its $\texttt{React}_{\texttt{SO}}$ mission. Particularly, there is a stopped police vehicle directly ahead of the $Victim_{car}$ on a crossroad (Fig. 9-`t1`). The $Attack_{car}$ starts (`t1`) at the $Victim_{car}$'s left front and steers right to make a fake lane cut-in. The $Attack_{car}$ then steers back and brakes at `t2`. At `t3`, the $Attack_{car}$ avoids the potential collision with the police vehicle and steers left. $Victim_{car}$ fails to brake and crashes with the police car (Fig. 9-`t4`).

Root Cause. We conclude this attack's root cause is "perception module error". We first confirmed that the $Victim_{car}$ stops in front of the police vehicle when $Attack_{car}$ does not exist. We next investigated the $Victim_{car}$'s perception module output

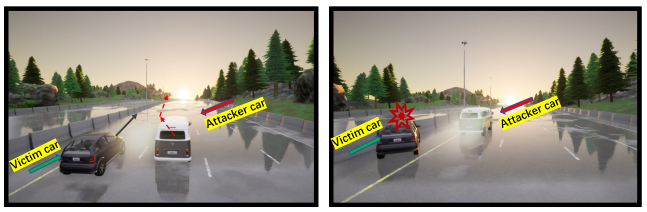

Figure 10: Case Study 2: $Attack_{car}$ exploits blind spots in openpilot's perception to cause the $Victim_{car}$ to crash.

and noticed that it correctly tags both the $Attack_{car}$ and police car. Yet, due to $Attack_{car}$'s movements, the $Victim_{car}$'s prediction module confuses the $Attack_{car}$ with the police vehicle and fails to brake. Particularly, when the $Attack_{car}$ changes its steering angle and throttle, the $Victim_{car}$ presumes the $Attack_{car}$ is going to change lanes, and predicts its path as it will turn right. When the $Attack_{car}$ passes the police vehicle, the $Victim_{car}$'s prediction module transfers the $Attack_{car}$'s predicted driving path to the police car's one and assumes the police vehicle will turn right. Thus, the $Victim_{car}$ eventually crashes with the stopped police vehicle.

Case Study 2 (Fig. 10) - *Failing to respond to a cutting-in vehicle on highway*: This attack [2] makes the $Victim_{car}$ violate the $\texttt{Follow}_{\texttt{Car}}$ mission. The attack is conducted on a highway with a wet road and overhead sun, when the $Victim_{car}$ is travelling the left-most lane, and the $Attack_{car}$ is driving in a rightward lane (Fig. 10-left). The $Attack_{car}$ moves sharply into a reflective patch of the road (such as a puddle) directly in front of the $Victim_{car}$. This causes the openpilot vehicle to steer left and crash at high speeds into the highway barrier (Fig. 10-right). At typical highway speeds between 80 km/h and 110 km/h, this is fatal for the occupants of the $Victim_{car}$.

Root Cause. When replicating the attack under different environmental conditions, we observed that the attack is successful only on a wet and reflective road with the sun shining overhead but not behind the $Victim_{car}$. These environmental conditions create reflections on the ground, off which sunlight bounces directly into the $Victim_{car}$'s camera. These reflections act as "blind spots" to openpilot's perception, registering to the camera as very bright white patches obscuring the view. The attacker can take advantage of this by cutting into these blind spots at the proper moment. To the $Victim_{car}$, the $Attack_{car}$ appears very suddenly at a close range, popping out of an area where the $Victim_{car}$ could not see before. As a result, the $Victim_{car}$ fails to react properly, causing a fatal crash.

6.4 Baseline Comparisons

6.4.1 ACERO without Robustness-Guidance

We implement a baseline approach where we disable the robustness guidance component of ACERO while finding adversarial commands (**RQ5**). In the baseline approach, the $Attack_{car}$ does not obtain any feedback from the $Victim_{car}$'s missions as a guide to select control commands and simply issues commands that maintain all physical constraints. We

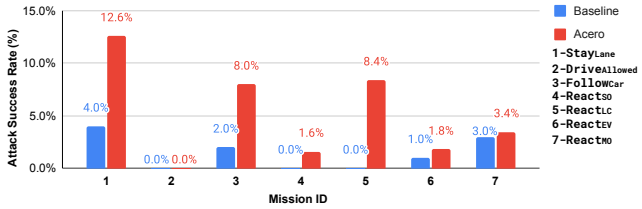

Figure 11: The comparison of the percentage of successful attacks discovered by ACERO vs. baseline approach.

compare the baseline approach with ACERO under 100 test cases for each mission using the same attack scenes.

Fig. 11 shows the percentage of mission violations discovered by ACERO and the baseline approach (without traffic and default weather conditions). The baseline only finds attacks in 1.4% of the test cases (16 out of 700), which is 3.5x less than robustness guidance. We observe ACERO with robustness guidance yields a higher attack success rate in all missions. Only in the $\texttt{React}_{\texttt{MO}}$ mission, the baseline approach gives a similar success rate with ACERO. We investigated the driving logs and found that this stems from Autoware's object-tracking algorithm, which fails to detect small moving objects, such as bicycles and pedestrians. Thus, the $Attack_{car}$'s movements do not affect the behavior of Autoware for such objects, causing it to crash them without robustness guidance.

6.4.2 ACERO vs. AV-Fuzzer

We compare ACERO with AV-Fuzzer [30] because, compared to other works on AV vulnerability discovery [42, 70, 72], it can be extended to search for adversarial maneuvers (**RQ6**). AVFuzzer aims to identify the maneuvers surrounding vehicles can make to induce safety violations for an AV. However, it does not constrain the maneuvers such that the adversary remains unharmed, and obeys traffic and self-safety constraints. It also only considers traffic conditions that involve vehicle-following and lane-changing, which limits its scope to specific AVs, operating in specific scenarios.

ACERO can discover the same attacks as AV-Fuzzer by disabling the `NoCrash_Traffic` physical constraint of $Attack_{car}$; yet, we do not consider such attacks as successful (since they jeopardize the attacker's safety and do not provide low liability.) To compare ACERO with AV-Fuzzer, we disabled physical constraints and conducted 100 additional test cases for $\texttt{Follow}_{\texttt{Car}}$ and $\texttt{React}_{\texttt{LC}}$ (without traffic), which are the only two missions supported by AV-Fuzzer. ACERO found 24 crashes in $\texttt{Follow}_{\texttt{Car}}$ and 27 crashes in $\texttt{React}_{\texttt{LC}}$ between $Attack_{car}$ and $Victim_{car}$, which are similar to the safety violations reported in the AV-Fuzzer. In contrast, AV-Fuzzer identifies seven (three $\texttt{Follow}_{\texttt{Car}}$ and four $\texttt{React}_{\texttt{LC}}$) of the 28 attacks clusters that ACERO discovers. The seven attack clusters can be discovered by targeting $\texttt{Follow}_{\texttt{Car}}$ and $\texttt{React}_{\texttt{LC}}$ missions and giving high-level driving commands to the $Attack_{car}$ that AV-Fuzzer supports (e.g., changing lanes). However, the remaining 21 attack clusters require missions and fine-grained maneuvers that AV-Fuzzer does not support (e.g., steering within the lane).

6.5 Execution Time Analysis

We measure the time spent to run an attack on each AD system and the time to verify their reproducibility in random weather conditions (**RQ7**). The average time to run an attack is 541.9 $\pm$ 46.1 secs for Autoware and 251.4 $\pm$ 31.2 secs for openpilot. The execution time difference between the AD systems is mainly due to attack scene initialization. Autoware initializes sophisticated modules, including localization, perception, object detection & tracking, and motion planning. Thus, when ACERO rewinds the attack scene, restarting the Autoware takes a longer time. The average time to reproduce an attack in random weather conditions takes 164.4 $\pm$ 8.3 secs for openpilot and 325.5 $\pm$ 3.9 secs for Autoware. We next analyze the execution time of ACERO's each component. ACERO's test scenario initialization takes on average 8.3 $\pm$ 0.06 secs for openpilot and 326.2 $\pm$ 18.6 secs for Autoware. Its adversarial trajectory generation takes on average 243.1 $\pm$ 74.7 secs for openpilot and 215.7 $\pm$ 31.1 secs for Autoware.

Lastly, we compare ACERO's execution time with the baseline approach with robustness guidance disabled (Sec. 6.4.1). This adversarial trajectory generation for this baseline takes on average 3.8 $\pm$ 1.3 secs for openpilot and 3.9 $\pm$ 0.05 secs for Autoware. It requires less time than ACERO because it generates the adversarial commands without robustness guidance; thus, the "rewinding phase" is not used to find the attack command with the least robustness, which incurs the highest time overhead.

7 Limitations and Discussion

Multiple Adversarial Agents. A resourceful adversary may leverage multiple adversarial vehicles and/or agents (e.g., pedestrians or cyclists) to attack the $Victim_{car}$. Although we show that even one $Attack_{car}$ is enough to make the $Victim_{car}$ fail its missions, ACERO can be extended to generate adversarial trajectories for multiple collaborating adversarial agents. Particularly, we found that some attacks happen only when certain traffic agents are included in the attack scene. Future work will expand our analysis to robustness guidance that runs on multiple agents to form a collaborative attack.

Attacks against Multiple AVs. ACERO can be extended to identify adversarial trajectories to conduct attacks against multiple cooperative AVs (e.g., in a platoon) to make one or more AVs crash. This requires identifying the missions for cooperative AVs and representing them with LTL to guide ACERO with robustness metrics. As future work, we will expand our formalization on such missions and create scenes to guide the $Attack_{car}$ to attack multiple AVs simultaneously.

Real-world Experiments. Due to physical safety considerations in real-world AV tests, simulations have become the de-facto standard for AV testing. Many approaches from the industry and academia have leveraged state-of-the-art simula-

tors (e.g., CARLA) to identify safety and security violations in AVs [42,54,55]. This is because such simulators accurately reflect the real-world traffic and environmental conditions with complex physical modeling, and enable testing various driving scenes without physical risk [15,29,33]. To ensure our attacks can be transferred to the real world, we use maps from the real-world (Fig. 6) and reproduce the attacks with different weather conditions and operator errors.

To reproduce an attack with a high success rate in the real world, the attacker can determine in the simulation the time (e.g., morning, sunset), location (e.g., crossroad, roundabout, T-intersection), driving software (e.g., Autoware, Openpilot) and the vehicle model of the victim (e.g., Tesla Model 3, Toyota Pirus). The attacker can then perform the maneuvers generated by ACERO within an operator error (Sec. 6.2.2).

Automatic Emergency Braking. All collision cases in Table 3 occur while assuming that $Victim_{car}$ is not equipped with automatic emergency braking (AEB) since none of the simulators support AEB. AEB detects a possible collision through radar and camera, and enables braking from higher speeds to prevent crashes [39]. Therefore, some collision cases in Table 3 might not happen in the real world if the $Victim_{car}$ is equipped with AEB.

AEB, however, cannot prevent all possible collisions for the following reasons. (1) AEB test protocol [17] requires automobile manufacturers to only test forward-collision scenarios, which means that AEB cannot be assured of preventing near-side collision cases. (2) AEB frequently fails to detect small moving agents on roads (e.g., pedestrians or bicyclists) [40]. Although AEB can detect them through cameras, it might be ineffective at night [16]. (3) Even if $Victim_{car}$ avoids a collision with a front vehicle, the sudden braking may cause a collision with another vehicle in the back.

Ethical Implications and Benefits to Industry. The adversarial maneuvers are unconventional regarding the safety and security of AVs. That is, we do not discover traditional software bugs but expose AV mission violations that may harm users, other agents, and the environment. We have thus reported our results to openpilot and Autoware developers and prepared a technical report that summarizes our findings to notify other AD companies, AV organizations, and policymakers. Particularly, we shared our report with 11 different AD companies, three AV organizations, and two policymakers[1].

AD software developers can use ACERO to find buggy behaviors due to the discovered adversarial maneuvers, patch or harden AV software depending on their root causes. For example, knowing that an adversary can exploit an ACERO-discovered bug in the AV's perception module, developers could collect additional data for training or use adversarial trajectories for adversarial training to build more robust models. As another example, developers could set the configurations more conservatively (e.g., set a higher default value for the `deceleration_range` parameter in Listing 1) since ACERO shows that an adversary can exploit misconfigured parameters through specific adversarial maneuvers.

[1] AV Companies: Waymo, Zoox, Ford AV and Mobility, Nuro, May Mobility, Daimler Truck, Motional, Cruies, and Waabi; AV organizations: National Association of City Transportation Officials, The Autonomous Vehicle Computing Consortium and Autonomous Vehicle Industry Association; Policymakers: NHTSA and SAE-ITC.

Table 5: A comparison of ACERO with other AV systems.

System	Physical Constraints	Attack Car	Multiple Traffic Conditions	Random Weather	Trajectory Replication Error	Targets End-to-End AVs	Input to Attack Car
Black-box Testing [42]	N/A	✗	✓	✓	N/A	✓	N/A
Plan-Fuzz [70]	N/A	✗	✓	✗	N/A	✗	N/A
Zhang et al. [72]	✓	✓	✗	✗	✗	✗	Position*
AV-Fuzzer [30]	Partial**	✓	✗	✗	✗	✓	Instructions†
Salgado et al. [53]	✓	✓	✗	✗	✗	✗	Braking
ACERO	✓	✓	✓	✓	✓	✓	Maneuvers‡

* Positions teleport the AV to concrete road positions (e.g., , a coordinate in the map).
** Not crashing with the victim car is not considered.
† Instructions give the AV a concrete driving instruction (e.g., , changing to the right lane)
‡ Maneuvers give the AV concrete throttle, brake, and steer values (e.g., , press throttle by 30% maximum)

8 Related Work

Driving Scene Generation. A line of prior work has generated driving scenes with different weather conditions, road types, and positions of other traffic agents to test the AD software components. To generate such scenes, they leverage diverse methods including reinforcement learning [1,14,28], Monte-Carlo sampling [13,43], deep learning (e.g., autoencoders and RNNs) [50,62], Markov decision processes [10,20], and evolutionary algorithms [26].

Most previous works have focused on generating a single traffic scene in a single map for a specific AV functionality, such as lane changing [1,10,14,28,43,50]. Additionally, these works mostly use traffic scenes to generate test cases only for the AD planners [10,13,14,20,26,28,50]. Furthermore, none of the above works consider the impact of vehicle type while generating traffic scenes. In contrast, ACERO's goal is to generate driving scenes for discovering adversarial maneuvers while integrating physical constraints to ensure both low liability and safety for the adversary. To achieve this goal, we test ACERO within 14 traffic scenes with various weather conditions in three maps specifically designed for seven missions.

Vulnerability Discovery in AVs. In Table 5, we compare ACERO with several recent approaches that differ in focus and scope. These approaches are the most applicable that run with open-source AD software in a simulation with the goal of identifying safety and security violations.

A recent effort uses black-box testing to adapt importance sampling for finding failure cases in AD systems in rare weather conditions [42]. This approach does not consider additional vehicles in the traffic; thus, it cannot generate adversarial driving maneuvers. Another approach, PlanFuzz, finds DoS vulnerabilities in the behavioral planning module of AVs [70]. Although it integrates surrounding vehicles, it does not generate adversarial commands for these vehicles and only inputs fixed trajectories (with planning constraints).

Another line of work targets finding vulnerabilities in vehicle platooning algorithms to degrade the algorithm performance and cause collisions [12, 19]. Yet, they only consider a single mission (car-following) in platoons; therefore, they cannot generate adversarial maneuvers.

A recent work generates adversarial trajectories to maximize the prediction error of AVs with a limited set of physical constraints [72]. However, it only attacks against the trajectory prediction of AVs, whereas ACERO targets discovering adversarial maneuvers against full-stack AD software. Lastly, AV-Fuzzer finds maneuvers that cause safety violations for an AV. Yet, as quantitatively compared in Sec. 6.4.2, it only assesses traffic conditions that involve vehicle following and lane changing and does not ensure the vehicle obeys self-safety constraints and traffic laws, causing $Attack_{car}$ to crash.

A line of recent work leverages falsification, a formal analysis technique that uses optimization algorithms to search for falsifying inputs, to discover safety and security policy violations in AVs [26, 53, 67, 68]. Some systems integrate multi-armed bandit and Halton samplers to search for AV violations with a cost function that considers the distance to other vehicles, time-to-collision, progress towards the destination, and lane violation missions [67, 68]. These approaches do not define physical constraints for the $Attack_{car}$; thus, they usually find violations where $Attack_{car}$ is involved in an accident. A recent work generates adversarial vehicle trajectories to find vulnerabilities in collision avoidance systems while enforcing physical constraints [53]. This work formally defines distance and collision metrics for the $Attack_{car}$ and $Victim_{car}$ to maximize the $Attack_{car}$'s distance to prevent its collisions and minimize the $Victim_{car}$'s distance to other vehicles to cause collisions. Then, it searches for violations through cross-entropy and Bayesian samplers. This work successfully identifies AV vulnerabilities in AV's collision avoidance component; however, it does not consider end-to-end vehicle missions (Table 1), and its input space is limited to braking time, duration, and intensity.

In contrast, we define 7 SAE level 2-4 missions for the end-to-end safe operation of AVs (Table 1) and introduce 7 physical constraints to ensure the $Attack_{car}$'s safety and low liability (Table 2). We further assess each mission with the throttle, brake, and steering commands using a robustness metric to discover adversarial maneuvers and evaluate their reproducibility while enforcing physical constraints.

Attacks against Perception Components. Another line of work conducts sensor spoofing and jamming attacks against AVs [5, 6, 32, 35, 36, 63]. There are also attacks that disturb the classification of AD components related to environment sensing, such as Camera, GPS and LiDAR [22, 36, 47, 54, 58, 59, 74, 75]. These attacks differ from ACERO in scope as they exploit vulnerabilities in specific sensing components of AVs, whereas ACERO aims to discover adversarial maneuvers against the full pipeline of AD software.

9 Conclusions

We introduce ACERO, a robustness-guided framework for discovering adversarial driving maneuvers. ACERO has two key aspects that distinguish it from other methods: (1) applying physical constraints on the adversarial vehicle to ensure the adversary's safety and low liability, and (2) using the robustness of the victim vehicle as guidance to optimize the adversarial command generation. We evaluated ACERO on two popular AD platforms and discovered 341 attack cases against them, and clustered the attacks into 28 unique trajectories.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their valuable suggestions. This work has been partially supported by the National Science Foundation (NSF) under grant CNS-2144645 and Office of Naval Research (ONR) under grant N00014-20-1-2128. Any findings, conclusions and recommendations expressed in this work are those of the authors and do not necessarily reflect the views of the NSF or ONR.

References

[1] Yasasa Abeysirigoonawardena, Florian Shkurti, and Gregory Dudek. Generating adversarial driving scenarios in high-fidelity simulators. In *International Conference on Robotics and Automation (ICRA)*, 2019.

[2] Acero. https://github.com/purseclab/Acero, 2023. [Online; accessed 8-June-2023].

[3] Apollo. https://apollo.auto/, 2023. [Online; accessed 8-May-2022].

[4] AuroAi. Carla apollo bridge. https://github.com/AuroAi/carla_apollo_bridge, 2022. [Online; accessed 4-May-2023].

[5] Yulong Cao, Ningfei Wang, Chaowei Xiao, Dawei Yang, Jin Fang, Ruigang Yang, Qi Alfred Chen, Mingyan Liu, and Bo Li. Invisible for both camera and lidar: Security of multi-sensor fusion based perception in autonomous driving under physical-world attacks. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[6] Yulong Cao, Chaowei Xiao, Benjamin Cyr, Yimeng Zhou, Won Park, Sara Rampazzi, Qi Alfred Chen, Kevin Fu, and Z Morley Mao. Adversarial sensor attack on lidar-based perception in autonomous driving. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[7] Python api reference. https://carla.readthedocs.io/en/latest/python_api/, 2023. [Online; accessed 28-April-2023].

[8] CARLA maps. https://carla.readthedocs.io/en/latest/core_map/#carla-maps, 2023. [Online; accessed 10-May-2023].

[9] CARLA weather. https://carla.readthedocs.io/en/latest/core_map/#carla-maps, 2021. [Online; accessed 10-March-2023].

[10] Baiming Chen, Xiang Chen, Qiong Wu, and Liang Li. Adversarial evaluation of autonomous vehicles in lane-change scenarios. *IEEE Transactions on Intelligent Transportation Systems*, 2021.

[11] Commaai. commaai/openpilot. https://github.com/commaai/openpilot, 2022. [Online; accessed 28-April-2023].

[12] Soodeh Dadras, Ryan M Gerdes, and Rajnikant Sharma. Vehicular platooning in an adversarial environment. In *ACM Symposium on Information, Computer and Communications Security*, 2015.

[13] Wenhao Ding, Baiming Chen, Bo Li, Kim Ji Eun, and Ding Zhao. Multimodal safety-critical scenarios generation for decision-making algorithms evaluation. *IEEE Robotics and Automation Letters*, 2021.

[14] Wenhao Ding, Baiming Chen, Minjun Xu, and Ding Zhao. Learning to collide: An adaptive safety-critical scenarios generating method. In *International Conference on Intelligent Robots and Systems (IROS)*, 2020.

[15] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An Open Urban Driving Simulator. In *Annual Conference on Robot Learning*, 2017.

[16] Ellen Edmonds. AAA warns pedestrian detection systems don't work when needed most. https://newsroom.aaa.com/2019/10/aaa-warns-pedestrian-detection-systems-dont-work-when-needed-most/, 2019. [Online; accessed 28-April-2023].

[17] NCAP Euro. European new car assessment programme (Euro NCAP)—test protocol—AEB systems. https://cdn.euroncap.com/media/26996/euro-ncap-aeb-c2c-test-protocol-v20.pdf, 2017. [Online; accessed 28-April-2023].

[18] Alessio Gambi, Marc Mueller, and Gordon Fraser. Automatically testing self-driving cars with search-based procedural content generation. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019.

[19] Ryan M Gerdes, Chris Winstead, and Kevin Heaslip. CPS: an efficiency-motivated attack against autonomous vehicular transportation. In *Annual Computer Security Applications Conference (ACSAC)*, 2013.

[20] Zahra Ghodsi, Siva Kumar Sastry Hari, Iuri Frosio, Timothy Tsai, Alejandro Troccoli, Stephen W Keckler, Siddharth Garg, and Anima Anandkumar. Generating and characterizing scenarios for safety testing of autonomous vehicles. In *IEEE Intelligent Vehicles Symposium*, 2021.

[21] Iowa Goverment. How do i drive in a roundabout. https://tinyurl.com/3ubtwy48. [Online; accessed 10-March-2023].

[22] Pengfei Jing, Qiyi Tang, Yuefeng Du, Lei Xue, Xiapu Luo, Ting Wang, Sen Nie, and Shi Wu. Too good to be safe: Tricking lane detection in autonomous driving with crafted perturbations. In *USENIX Security Symposium*, 2021.

[23] Shinpei Kato, Shota Tokunaga, Yuya Maruyama, Seiya Maeda, Manato Hirabayashi, Yuki Kitsukawa, Abraham Monrroy, Tomohito Ando, Yusuke Fujii, and Takuya Azumi. Autoware on board: Enabling autonomous vehicles with embedded systems. In *ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, 2018.

[24] Christos Katrakazas, Mohammed Quddus, Wen-Hua Chen, and Lipika Deka. Real-time motion planning methods for autonomous on-road driving: State-of-the-art and future research directions. *Transportation Research*, 2015.

[25] Hyungsub Kim, Muslum Ozgur Ozmen, Antonio Bianchi, Z. Berkay Celik, and Dongyan Xu. PGFUZZ: Policy-Guided Fuzzing for Robotic Vehicles. In *Network and Distributed System Security Symposium (NDSS)*, 2021.

[26] Moritz Klischat and Matthias Althoff. Falsifying motion plans of autonomous vehicles with abstractly specified traffic scenarios. *IEEE Transactions on Intelligent Vehicles*, 2022.

[27] Yongbon Koo, Jinwoo Kim, and Wooyong Han. A method for driving control authority transition for cooperative autonomous vehicle. In *IEEE Intelligent Vehicles Symposium*, 2015.

[28] Mark Koren and Mykel J Kochenderfer. Efficient autonomy validation in simulation with adaptive stress testing. In *IEEE Intelligent Transportation Systems Conference*, 2019.

[29] Instructions - LGSVL Simulator. https://www.svlsimulator.com/docs/archive/2020.06/apollo-master-instructions/, 2023. [Online; accessed 05-March-2023].

[30] Guanpeng Li, Yiran Li, Saurabh Jha, Timothy Tsai, Michael Sullivan, Siva Kumar Sastry Hari, Zbigniew Kalbarczyk, and Ravishankar Iyer. AV-Fuzzer: Finding safety violations in autonomous driving systems. In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2020.

[31] Hechen Liu and Markus Schneider. Similarity measurement of moving object trajectories. In *ACM SIGSPATIAL International Workshop on GeoStreaming*, 2012.

[32] Yanmao Man, Raymond Muller, Ming Li, Z. Berkay Celik, and Ryan Gerdes. That person moves like a car: Misclassification attack detection for autonomous systems using spatiotemporal consistency. In *USENIX Security Symposium*, 2023.

[33] Nvidia drive end-to-end platform for software-defined avs. https://www.nvidia.com/en-us/self-driving-cars/, 2022.

[34] Waymo's official website. https://waymo.com/, 2022. [Online; accessed 8-March-2022].

[35] Raymond Muller, Yanmao Man, Z. Berkay Celik, Ming Li, and Ryan Gerdes. Physical hijacking attacks against object trackers. In *ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[36] Ben Nassi, Yisroel Mirsky, Dudi Nassi, Raz Ben-Netanel, Oleg Drokin, and Yuval Elovici. Phantom of the adas: Securing advanced driver-assistance systems from split-second phantom attacks. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020.

[37] NHTSA. Critical reasons for crashes investigated in the national motor vehicle crash causation survey. https://crashstats.nhtsa.dot.gov/Api/Public/Publication/812115, 2018. [Online; accessed 23-May-2023].

[38] National highway traffic safety administration. https://www.nhtsa.gov/technology-innovation/automated-vehicles-safety, 2020.

[39] NHTSA. Nhtsa proposes automatic emergency braking requirements for new vehicles. https://www.nhtsa.gov/press-releases/automatic-emergency-braking-proposed-rule, 2023. [Online; accessed 12-June-2023].

[40] Nissan's faulty automatic emergency braking and radar system. https://lemonlawexperts.com/nissan-faulty-braking-system/, 2021. [Online; accessed 28-April-2023].

[41] Samyeul Noh and Woo-Yong Han. Collision avoidance in on-road environment for autonomous driving. In *International Conference on Control, Automation and Systems (ICCAS)*, 2014.

[42] Justin Norden, Matthew O'Kelly, and Aman Sinha. Efficient black-box assessment of autonomous vehicle safety. *arXiv preprint arXiv:1912.03618*, 2019.

[43] Matthew O'Kelly, Aman Sinha, Hongseok Namkoong, Russ Tedrake, and John C Duchi. Scalable end-to-end autonomous vehicle testing via rare-event simulation. *Advances in neural information processing systems*, 2018.

[44] National Committee on Uniform Traffic Laws. *Traffic Laws Annotated*. National Committee on Uniform Traffic Laws and Ordinances, 1972.

[45] Muslum Ozgur Ozmen, Xuansong Li, Andrew Chu, Z. Berkay Celik, Bardh Hoxha, and Xiangyu Zhang. Discovering IoT physical channel vulnerabilities. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.

[46] Muslum Ozgur Ozmen, Ruoyu Song, Habiba Farrukh, and Z. Berkay Celik. Evasion attacks and defenses on smart home physical event verification. In *Network and Distributed System Security (NDSS)*, 2023.

[47] Jonathan Petit, Bas Stottelaar, Michael Feiri, and Frank Kargl. Remote attacks on automated vehicles sensors: Experiments on camera and lidar. *Black Hat Europe*, 2015.

[48] Amir Pnueli. The temporal logic of programs. In *Annual Symposium on Foundations of Computer Science*, 1977.

[49] Aggressive driver runs Tesla autopilot off road. https://www.youtube.com/watch?v=ayf4somEq8U&ab_channel=TheTechofTech, 2020. [Online; accessed 23-May-2023].

[50] Davis Rempe, Jonah Philion, Leonidas J Guibas, Sanja Fidler, and Or Litany. Generating useful accident-prone driving scenarios via a learned traffic prior. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022.

[51] Guodong Rong, Byung Hyun Shin, Hadi Tabatabaee, Qiang Lu, Steve Lemke, Mārtiņš Možeiko, Eric Boise, Geehoon Uhm, Mark Gerow, Shalin Mehta, et al. Lgsvl simulator: A high fidelity simulator for autonomous driving. In *IEEE International Conference on Intelligent Transportation Systems*, 2020.

[52] SAE International. Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles. https://www.sae.org/standards/content/j3016_202104/, 2021. [Online; accessed 23-March-2023].

[53] Ivan F Salgado, Nicanor Quijano, Daniel J Fremont, and Alvaro A Cardenas. Fuzzing malicious driving behavior to find vulnerabilities in collision avoidance systems. In *IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2022.

[54] Takami Sato, Junjie Shen, Ningfei Wang, Yunhan Jia, Xue Lin, and Qi Alfred Chen. Dirty Road Can Attack: Security of Deep Learning based Automated Lane Centering under Physical-World Attack. In *USENIX Security Symposium*, 2021.

[55] John M Scanlon, Kristofer D Kusano, Tom Daniel, Christopher Alderson, Alexander Ogle, and Trent Victor. Waymo simulated driving behavior in reconstructed fatal crashes within an autonomous vehicle operating domain. *Accident Analysis & Prevention*, 2021.

[56] Wilko Schwarting, Javier Alonso-Mora, and Daniela Rus. Planning and decision-making for autonomous vehicles. *Annual Review of Control, Robotics, and Autonomous Systems*, 2018.

[57] Chengyao Shen. Decoding comma.ai/openpilot: the driving model. https://medium.com/@chengyao.shen/decoding-comma-ai-openpilot-the-driving-model-a1ad3b4a3612, 2019. [Online; accessed 23-March-2023].

[58] Junjie Shen, Jun Yeon Won, Zeyuan Chen, and Qi Alfred Chen. Drift with Devil: Security of multi-sensor fusion based localization in high-level autonomous driving under GPS spoofing. In *USENIX Security Symposium*, 2020.

[59] Dawn Song, Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Florian Tramer, Atul Prakash, and Tadayoshi Kohno. Physical adversarial examples for object detectors. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2018.

[60] Yun Tang, Yuan Zhou, Fenghua Wu, Yang Liu, Jun Sun, Wuling Huang, and Gang Wang. Route coverage testing for autonomous vehicles via map modeling. In *International Conference on Robotics and Automation (ICRA)*, 2021.

[61] Yun Tang, Yuan Zhou, Tianwei Zhang, Fenghua Wu, Yang Liu, and Gang Wang. Systematic testing of autonomous driving systems using map topology-based scenario classification. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021.

[62] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *International Conference on Software Engineering (ICSE)*, 2018.

[63] Yazhou Tu, Zhiqiang Lin, Insup Lee, and Xiali Hei. Injected and delivered: Fabricating implicit control over actuation systems by spoofing inertial sensors. In *USENIX Security Symposium*, 2018.

[64] Cumhur Erkan Tuncali, Georgios Fainekos, Hisahiro Ito, and James Kapinski. Simulation-based adversarial test generation for autonomous vehicles with machine learning components. In *IEEE Intelligent Vehicles Symposium*, 2018.

[65] Cumhur Erkan Tuncali, Theodore P Pavlic, and Georgios Fainekos. Utilizing s-taliro as an automatic test generation framework for autonomous vehicles. In *IEEE International Conference on Intelligent Transportation Systems (ITSC)*, 2016.

[66] Marcell Vazquez-Chanlatte. mvcisback/py-metric-temporal-logic: v0.1.1. https://github.com/mvcisback/py-metric-temporal-logic, 2019. [Online; accessed 28-April-2023].

[67] Kesav Viswanadha, Francis Indaheng, Justin Wong, Edward Kim, Ellen Kalvan, Yash Pant, Daniel J Fremont, and Sanjit A Seshia. Addressing the ieee av test challenge with scenic and verifai. In *IEEE International Conference on Artificial Intelligence Testing*, 2021.

[68] Kesav Viswanadha, Edward Kim, Francis Indaheng, Daniel J Fremont, and Sanjit A Seshia. Parallel and multi-objective falsification with scenic and verifai. In *Runtime Verification*, 2021.

[69] Katja Vogel. A comparison of headway and time to collision as safety indicators. *Accident analysis & prevention*, 2003.

[70] Ziwen Wan, Junjie Shen, Jalen Chuang, Xin Xia, Joshua Garcia, Jiaqi Ma, and Qi Alfred 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)*, 2022.

[71] Alper Yilmaz, Omar Javed, and Mubarak Shah. Object tracking: A survey. *ACM computing surveys (CSUR)*, 2006.

[72] Qingzhao Zhang, Shengtuo Hu, Jiachen Sun, Qi Alfred Chen, and Z Morley Mao. On adversarial robustness of trajectory prediction for autonomous vehicles. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2022.

[73] Xiangmo Zhao, Pengpeng Sun, Zhigang Xu, Haigen Min, and Hongkai Yu. Fusion of 3d lidar and camera data for object detection in autonomous vehicle applications. *IEEE Sensors Journal*, 2020.

[74] Yue Zhao, Hong Zhu, Ruigang Liang, Qintao Shen, Shengzhi Zhang, and Kai Chen. Seeing isn't believing: Towards more robust adversarial attack against real world object detectors. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[75] Husheng Zhou, Wei Li, Zelun Kong, Junfeng Guo, Yuqun Zhang, Bei Yu, Lingming Zhang, and Cong Liu. Deepbillboard: Systematic physical-world testing of autonomous driving systems. In *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2020.

A Weather Settings

For the weather conditions we use for the exp-❻ in Table 3, we leverage four weather parameters to sample the weather from the previous work [42]: Sun altitude angle (S), Cloudiness (C), Precipitation on the ground ($\mathtt{P_g}$) and Precipitation in the air ($\mathtt{P_a}$). For sun angle altitude, we use $\mathtt{A} \sim \mathtt{90Uniform}(0,1)$; for precipitation on the ground, we use $\mathtt{P_g} \sim \mathtt{50Uniform}(0,1)$;

for cloudiness, we use a mixture distribution: $\mathtt{C} \sim \mathtt{30C_b}\,1\{\mathtt{C_u} < 0.5\} + (\mathtt{40C_b} + \mathtt{60})\,1\{\mathtt{C_u} \geq 0.5\}$ where $\mathtt{C_b} \sim \mathtt{Beta}(2,2)$ and $\mathtt{C_u} \sim \mathtt{Uniform}(0,1)$; The value of $\mathtt{P_a}$ is determined by the cloudiness: $\mathtt{P_a} = C\mathbb{1}\{C \geq 70\}$. All units are CARLA units.

For the weather conditions we use for evaluating the attack reproducibility in Sec. 4.5.2, we use 14 preset weather conditions from CARLA: ClearNoon, CloudyNoon, WetNoon, WetCloudyNoon, MidRainyNoon, HardRainNoon, SoftRainNoon, ClearSunset, CloudySunset, WetSunset, WetCloudySunset, MidRainSunset, HardRainSunset, SoftRainSunset [9].

B Sample Attack Guide

Listing 2 presents an example of a shortened attack guide, including the mission ID, the initial speed of vehicles, a detailed adversarial trajectory, an instance of weather conditions the mission is violated, and the consequence of the attack.

Listing 2 An example of ACERO's attack guide.

```
{Attack Guide : {
  Mission ID: C3,
  With Traffic: True,
  Attack Scene: City Roadway,
  Initial Speed:
      [Victim Vehicle: 25–30 km/h,
       Adversarial Vehicle: 25–30 km/h],
  Example Weather Conditions:
    [Sun Altitude Angle: 0.4.
      Cloudiness: 0,
      Precipitation in the air: 0,
      Precipitation on the ground: 23],
  Adversarial trajectory (relative to the target vehicle
       positions):[
    [Vector3D (x=–12.151794, y=–7.381805)],
    [Vector3D (x=–8.3, y=–5.39)],
    [Vector3D (x=3, y=–5)]],
  Adversarial Command Set: [
    [Vehicle Command (throttle=0.5,steer=0.3,brake=0)],
    [Vehicle Command (throttle=0.2,steer=0.1,brake=0)],
    [Vehicle Command (throttle=0.0,steer=0.3,brake=0.2)],],
   Physical Attack Consequence: Collision with
                                another traffic agent,
   Victim Vehicle speed at accident: 37 km/h}
}
```

C Implementation Details

Openpilot. Openpilot [11] is an open-source SAE Level 2 DA system, including adaptive cruise control, automatic lane centering, forward collision, and lane departure warnings. We test its missions using openpilot 0.8.6, which is officially integrated into CARLA 0.9.11. Openpilot uses a DL-based vision system that takes camera images and feeds the video sequence into the vision model and uses a convolutional neural network (CNN) model with a recurrent neural network (RNN) model for temporal reasoning [57]. Openpilot then taps into a vehicle's CAN bus, and links a car's modules together for control decisions, i.e., steering angle and braking pressure. In our experiments, we use openpilot with the camera to perform its intended missions (as supported in CARLA). However, we note that it may additionally integrate non-camera sensors such as radar when it is deployed to real vehicles.

Autoware. Autoware [23] is an open-source SAE Level 4 software, aiming to provide fully autonomous driving for users. We use Autoware 1.14.0 Docker version officially integrated into CARLA 0.9.10. Autoware uses the LiDAR data for 3D reasoning and camera data to recognize traffic lights and extract additional features. 3D objects are extracted from the point cloud through the Normal Distributions Transform localization algorithm. This data is augmented by Radar, GNSS, and IMU sensors. In our experiments, we use all of these sensors as supported in CARLA in the same way that they are deployed in real vehicles. CNN models are used to perform object detection on 2D images, and the Kalman Filter is for predicting object movements. Given a goal, the positions of the objects, and their trajectories, Autoware then uses a finite state machine (FSM) to determine the best routing path.

Porting ACERO to other AD Systems. While attempting to integrate Apollo into ACERO, we encountered two main challenges. First, Apollo is officially integrated into the LGSVL simulator, which does not support issuing real-time control commands, which ACERO requires. Second, the unofficial bridge [4] allowing Apollo to run in the CARLA simulator does not properly simulate physics and control inputs. Instead, it teleports the $Victim_{car}$ to set waypoints, which makes it an invalid representation of Apollo's actual performance. To be integrated into ACERO, Apollo's CARLA integration should provide a stable control module that accounts for physics and control inputs. Alternatively, the LGSVL implementation should be updated to support sending run-time commands.

In general, porting ACERO to other AD systems requires the following steps: (1) implementing robustness metrics based on the simulator API. (2) implementing the rewinding method based on the structure of the AD system. We believe these tasks are not a burden for developers familiar with AD systems. For instance, when we ported ACERO from openpilot to Autoware, it took about 20 hours of the two authors' manual effort. This includes the time of adding new attack scenes and implementing rewinding techniques.

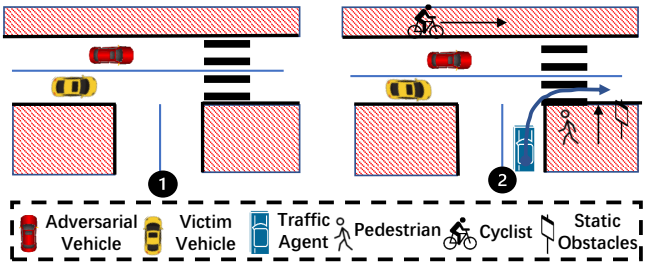

Figure 12: Illustration of two examples of traffic condition setup used in evaluating missions. ❶ No traffic and preset weather conditions. ❷ With traffic agents and weather conditions sampled from parameter distributions.

D Example Traffic Conditions

In Fig. 12, we illustrate two example traffic conditions on the intersection map, which contains the $Attack_{car}$ and $Victim_{car}$ ❶, and other traffic agents, including a road sign, pedestrian, vehicle, and cyclist ❷.