Too Afraid to Drive: Systematic Discovery of Semantic DoS Vulnerability in Autonomous Driving Planning under Physical-World Attacks

Ziwen Wan  Junjie Shen  Jalen Chuang  Xin Xia†  Joshua Garcia  Jiaqi Ma†  Qi Alfred Chen
University of California, Irvine  †University of California, Los Angeles
{ziwenw8, junjies1, jzchuang, joshua.garcia, alfchen}@uci.edu
†{x35xia, jiaqima}@ucla.edu

Abstract—In high-level Autonomous Driving (AD) systems, behavioral planning is in charge of making high-level driving decisions such as cruising and stopping, and thus highly security-critical. In this work, we perform the first systematic study of semantic security vulnerabilities specific to overly-conservative AD behavioral planning behaviors, i.e., those that can cause failed or significantly-degraded mission performance, which can be critical for AD services such as robo-taxi/delivery. We call them semantic Denial-of-Service (DoS) vulnerabilities, which we envision to be most generally exposed in practical AD systems due to the tendency for conservativeness to avoid safety incidents. To achieve high practicality and realism, we assume that the attacker can only introduce seemingly-benign external physical objects to the driving environment, e.g., off-road dumped cardboard boxes.

To systematically discover such vulnerabilities, we design PlanFuzz, a novel dynamic testing approach that addresses various problem-specific design challenges. Specifically, we propose and identify planning invariants as novel testing oracles, and design new input generation to systematically enforce problem-specific constraints for attacker-introduced physical objects. We also design a novel behavioral planning vulnerability distance metric to effectively guide the discovery. We evaluate PlanFuzz on 3 planning implementations from practical open-source AD systems, and find that it can effectively discover 9 previously-unknown semantic DoS vulnerabilities without false positives. We find all our new designs necessary, as without each design, statistically significant performance drops are generally observed. We further perform exploitation case studies using simulation and real-vehicle traces. We discuss root causes and potential fixes.

I. INTRODUCTION

Today, various companies are developing high-level (e.g., Level-4 [1]) Autonomous Driving (AD) vehicles. Some of them, e.g., Google Waymo [2], TuSimple [3], and Pony.ai [4], are already providing services on public roads. To enable such highly-automated driving, after the environmental sensing steps such as perception and localization, the AD systems need to use the sensed information to make high-level driving decisions such as cruising, stopping, lane changing, etc., that not only are safe and efficient, but also conform to driving norms such as traffic rules. Such a decision-making process is commonly referred to as behavioral planning, which is highly security critical as any mistakes made in it can directly lead to undesired driving behaviors such as driving too aggressively to cause collisions, or too conservatively to cause unnecessary emergency stops and road blocking. Various prior works studied security vulnerabilities in AD systems with such semantic consequences [5–9], but they mostly focus on environmental sensing errors (e.g., camera/LiDAR object detection [6–9]) instead of planning. There are recent works able to discover semantic planning errors [10], but they (1) are designed for whole-system testing instead of being planning-specific; (2) only consider overly-aggressive behaviors, leaving the overly-conservative side unexplored; and (3) focus on safety instead of security (i.e., lack explicit threat model considerations).

To fill this research gap, in this paper we perform the first AD planning-specific semantic vulnerability discovery. We specifically choose to focus on the more under-explored overly-conservative behavioral planning behaviors, especially those that can cause the victim AD vehicle to have a failed or significantly-degraded mission performance (e.g., permanent stop and never reach the destination). We refer to these as semantic Denial-of-Service (DoS) vulnerabilities for behavioral planning. We envision that such vulnerabilities can be most generally exposed in practice, based on the hypothesis that behavioral planning in real-world AD systems, especially in production settings, will generally try to be as conservative as possible to avoid any possible safety incidents, which can cause great business reputation and financial damages as in recent fatal accidents for Uber and Tesla [11–14]. In fact, such overly-conservative behaviors have already been observed for many production AD vehicles (e.g., Waymo, Uber, Volvo [15–20]), causing troubles to AD service and traffic flow (§II-B).

Considering the realism and generality of such problems in practice, we set our goal to develop an automated system to systematically discover such vulnerabilities to most generally help address these problems at the AD system development stage. To achieve high practicality and realism, we assume that the attacker can only introduce seemingly-benign external physical objects to the driving environment, e.g., dumped cardboard boxes or parked bikes on the road side, attacker-driven vehicles, etc. Dynamic testing is a promising...
approach to achieve domain-specific vulnerability discovery in general [21–28]. However, none of the existing designs can be directly applied to our problem due to several unique design challenges specific to our problem definition: (C1) Lack of testing oracles to tell whether a change of a planning decision is overly conservative or not. For example, directly putting obstacles ahead of the victim to cause DoS is not a vulnerability for behavioral planning; (C2) Need to systematically generate attacker-introduced physical objects following problem-specific physical constraints, e.g., avoiding road regions directly ahead of the victim as explained above; and (C3) Need to obtain fine-grained code-level feedback from the planning decision-making process to guide our vulnerability discovery, which is highly desired for us as the direct behavioral planning output is usually quite discrete (§III-B).

To achieve our goal, we design PlanFuzz, a novel dynamic testing approach that systematically addresses the aforementioned design challenges in an evolutionary testing framework. To address C1, we propose and identify Planning Invariant (PI) as the problem-specific testing oracle, which defines a set of constraints for the attacker-introduced physical objects based on common driving norms such that if satisfied, the behavior planning should not give up the desired planning decision. To address C2, we design PI-aware physical-object generation, which can systematically enforce the generated testing inputs to conform to both driving norms (e.g., traffic rules) and the PI constraints above, while maintaining diversity and inheritance properties desired for evolutionary testing. To address C3, we design behavioral planning vulnerability distance to measure how close the current planning decision is to violate PI and thus trigger a vulnerability in the run time.

We implement a prototype of PlanFuzz and evaluate it on 3 different behavioral planning implementations from two open-source AD systems, Apollo [29] and Autoware [30], which are both practical AD systems with full-stack implementations [31, 32]. We use LGSVL, an industry-grade AD simulator, to generate diverse driving scenarios, which allows us to obtain 11,912 different initial testing seeds in total for 8 different driving scenarios. Using PlanFuzz with these seeds, all 3 behavioral planning implementations are found vulnerable, with 9 previously-unknown semantic DoS vulnerabilities discovered in total. Among them, 8 can prevent the victim from reaching the destination (7 can cause permanent stop), and the remaining 1 can cause emergency stop. We also perform baseline comparisons by replacing different key components in our design, which shows statistically significant performance drops for almost all of the 9 vulnerabilities, leading to over 3.5× average slow-down or even failure in their discoveries. We also manually verified the discovered vulnerabilities and no false positives are generated.

To concretely understand the end-to-end impacts of the discovered vulnerabilities, we further perform 3 vulnerability exploitation case studies by constructing and evaluating real-world attack scenarios using simulation and real-vehicle sensor traces. The results show that the discovered vulnerabilities can cause the AD vehicle running Apollo or Autoware to (1) permanently stop in an empty road or in front of an empty intersection due to completely off-road cardboard boxes or parked bikes; or (2) give up necessary lane changing purely due to a following vehicle without any intention to change lanes. Demos are at our website https://sites.google.com/view/cav-sec/planfuzz [33]. We also discuss root causes and potential fixes. We also release our code at our website [33].

In summary, this work makes the following contributions:

- To the best of our knowledge, we are the first to perform AD planning-specific semantic vulnerability discovery. We focus on semantic DoS vulnerabilities, which can damage the availability of AD services. We formulate the problem with a domain-specific vulnerability definition and a practical threat model that only allows adding seemingly benign physical objects to the driving environment.
- To systematically discover the vulnerability, we design PlanFuzz, a novel dynamic testing approach that addresses various problem-specific design challenges: (1) propose and identify PIs as the testing oracle, (2) design a novel PI-aware physical object generation to systematically enforce problem-specific input constraints; and (3) design a novel behavioral planning vulnerability distance metric to effectively guide the discovery.
- We evaluate PlanFuzz on 3 planning implementations from two practical open-source AD systems. We find that PlanFuzz can effectively discover DoS vulnerabilities in all 3 implementations without incurring false positives. In total, 9 previously-unknown semantic DoS vulnerabilities are discovered, which can all be exploited to either prevent the victim from reaching its destination or cause an emergency stop. We find all our main designs are necessary, as without each design, statistically significant drops in performance are generally observed.
- We further perform 3 vulnerability exploitation case studies using simulation. The results show that the discovered vulnerabilities can cause the AD vehicle to unnecessarily stop permanently or give up necessary driving decisions. We also discuss root causes and potential mitigations.

II. BACKGROUND & PROBLEM DEFINITION

A. Behavioral Planning (BP) in AD Systems

AD planning. In high-level (e.g., Level-4 [1]) AD systems, planning is a critical module designed to generate safe, efficient, and smooth driving trajectories to reach the destination. In industry-grade AD, such a module typically adopt a 3-layer design: route planning, behavioral planning (BP), and local planning [29, 30, 34–40]. Given the destination, route planning selects a route from the map. To follow the selected route while ensuring safety and correctness (e.g., conform to traffic rules), BP then makes high-level driving decisions such as cruising, stopping, lane changing, etc. based on the real-time driving environment. For example, when the AD vehicle needs to pass a signaled intersection, the BP layer needs to consider both the traffic light and the dynamic behaviors of surrounding vehicles and pedestrians to decide whether and when to proceed. Next,
local planning translates the high-level decisions into concrete low-level driving trajectories (e.g., waypoints), which will then be passed to the vehicle control module to actuate.

**Our focus: AD Behavior Planning (BP).** As described above, BP is at the core of AD decision making, and thus highly security/safety critical: any mistakes made in it can directly lead to undesired driving behaviors such as driving too aggressively, which can cause collisions, or too conservatively, which can cause unnecessary emergency stops and road blocking. In BP, the decisions can be generated from *programmed* or *learned* logic. There are some recent works exploring learning-based planning [41–43], but so far they are all experimental (e.g., designed and evaluated only for simulated racing game setups [44–48]) and generally lack the necessary capability and support for real-world driving (e.g., limited to only cruising without handling intersections, cross-walks, pedestrians [48–51], and no consideration of traffic rules [51–56]). Such a learning-based approach is also generally known to suffer from difficulties with debugging and interpreting [57], and enforcing safety rules/measures [58], while the latter is especially critical for production AD. Thus, the BP in today’s industry-grade AD systems generally adopts programmed logic [29, 30, 59, 60]. Such program-based BP is thus also the main target of our design, which raises design challenges as detailed in §III-B.

### B. Attack Goal and Incentives

**Attack goal: Semantic Denial-of-Service (DoS) of BP.**

In this paper, we target an attack goal of causing semantic Denial-of-Service (DoS) on BP, which we define as causing it to change a normal driving decision to an overly-conservative one so that the victim AD vehicle will have a failed or significantly-degraded mission performance (e.g., never reach the destination). Specifically, we focus on 2 concrete types of such DoS in an AD context: (1) causing an emergency/permanent stop, and (2) causing the victim to give up a mission-critical driving decision, such as necessary left/right turns and lane changing on the route. To achieve this goal, in this paper we target physical-world attack vectors in the AD context (e.g., adding seemingly-benign static/dynamic physical road objects, detailed in §II-C) for high practicality and realism.

We choose to focus on causing overly-conservative driving decisions instead of overly-aggressive ones because we hypothesize that real-world BP, especially those in production settings, will generally try to be as conservative as possible by design to avoid any possible safety incidents. This is based on the fact that one single fatal crash (e.g., the Uber one [11] and increasingly more Tesla ones [12–14]), no matter whether it is mainly due to AD system flaws or not, can cause great reputational damage, lawsuits, and business being paused or even sold [61, 62]. In fact, such overly-conservative behaviors have already been observed in real-world production AD, causing troubles to AD service and traffic flows. For example, there is a video showing the AD vehicle from Waymo, a world-leading AD company, getting stuck by non-road-blocking traffic cones for >10 mins [17], in the parking lot when no other moving objects are around [15], and at an intersection resulting in blocking normal traffic [16]. A snapshot is in Fig. 1; since the environmental perception results are all correct according to the in-vehicle display [17], it is highly likely caused by planning flaws/bugs. There are more issues in the same vein reported for Waymo, Volvo, Uber, etc. [18–20].

### Problem seriousness and incentives.

With the growing deployment of commercial AD services without safety drivers (e.g., by Waymo, Nuro, and Baidu [63–65]), such DoS problems can severely damage the availability of these services and thus ruin their user experience, reputation, and also revenues. It can help if remote operators are available, but such helps are not necessarily effective. For example, in the Waymo video above (Fig. 1), the AD vehicle called remote-operator help twice but it actually made the problem worse, and eventually called road assistance team to physically arrive, causing >10min trip delay in total. Such semantic DoS may also cause safety problems, e.g., by triggering emergency brakes in dangerous road segments like highway exit ramps, or blocking the road and thus forcing other vehicles to borrow the reverse lane like in Fig. 1. Since such consequences can at least damage the reputation of the victim AD company, one potential attack incentive is for business competition (e.g., by a rival AD company to unfairly gain competitive advantages).

**Distinction to traditional software bugs.** As illustrated in Fig. 2, the semantic BP DoS vulnerability targeted in this paper is a type of semantic software vulnerability for BP, which can be caused by either software design flaws or implementation bugs. The key distinction of such semantic vulnerabilities to traditional software bugs is that their symptoms are erroneous behaviors at the BP decision logic level (e.g., keep driving or not, change lane or not) instead of at the generic computer program level (e.g., software crash, memory corruption, hang). In our problem setting, since we target physical-world attack vectors (§II-C), the semantic BP vulnerabilities we target further differ in that the vulnerability triggering is via physical-world realizable perturbations (e.g., by adding attacker-controllable road objects) instead of generic program input changes (e.g., bit-level value changes of BP inputs), which is also illustrated in Fig. 2.

### C. Threat Model

**Attack vector: Attacker-controllable common physical-world road objects.** Fig. 2 illustrates our threat model with a comparison to traditional software vulnerability exploitation. Instead of directly sending malicious inputs to the program, we assume that the attacker can only exploit semantic BP DoS vulnerabilities via introducing common and easily-controllable...
external physical-world objects to the driving environment, e.g., dumped cardboard boxes, parked bikes on the road side, vehicles driving on the roadway, or pedestrians walking on road pavements. We choose such a threat model because the attacker can more realistically launch such an attack in a practical setting since the attacker does not need to compromise or tamper with the internals of the victim AD system and the objects can pretend to be benign once they follow basic traffic laws and driving norms. In §VII-B, we also discuss the capabilities of a stronger threat model that might also be able to compromise the perception sensors. Since this work aims at developing a systematic BP vulnerability discovery system for AD system developers, our system design assumes white-box access to the BP implementation.

**Distinction to safety problems.** Under such a physical-world attack threat model, the attack-targeted unintended BP decision behaviors are also possible to naturally occur in non-adversarial settings, making them also in the scope of general safety or robustness problems. Here, the distinction is that we focus on the set of such unintended behaviors that are (more) triggerable by attackers in the driving environment, e.g., easily-controllable road objects such as cardboard boxes, bikes, and attacker-driven vehicles, instead of weather conditions and road-side building locations/shapes. Such a security focus makes the discovered vulnerabilities arguably more severe, since with an adversary such unintended behaviors can be more frequently, controllably, and strategically triggered to cause more severe real-world consequences, e.g., causing emergency brakes in more dangerous road segments such as highway ramps, and causing traffic blocking in mission-critical roads such as in front of police stations or fire stations.

### III. MOTIVATION AND CHALLENGES

In this section, we use a motivating example to concretely describe the BP DoS vulnerabilities targeted in this paper and the design challenges to systematically discover them.

#### A. Motivating Example

Fig. 3 shows the simplified pseudo code for a BP DoS vulnerability our system discovered from version 5.0 of Apollo, an open-source industry-grade AD system [29] (also confirmed that such vulnerability also exist in 6.0, the latest version). This logic is from path_bounds_decider, one of the BP decision-making steps for the lane following scenario that checks whether the current lane has enough space in the lateral direction (left/right) for the AD vehicle to drive; if not, it considers the lane as blocked.

**Decision logic.** As shown in Fig. 3 and illustrated in Fig. 4 (A), this code maintains 2 variables, left_bound and right_bound, to represent the leftmost and rightmost boundaries of the drivable space for a given longitudinal (forward/backward) position. If the space between these two boundaries is smaller than the AD vehicle width ADC_width (line 13), it means no drivable space laterally and thus the current lane is blocked. The two variables are initialized with the left and right lane boundaries (line 5). Next, it iterates over the list of detected surrounding static obstacles, and use the rightmost lateral boundaries (obs.max_l) of the left-side obstacles to update left_bound (line 8-9), and the leftmost lateral boundaries (obs.min_l) of the right-side obstacles to update right_bound (line 10-11). Such logic can thus avoid causing the vehicle body to hit/touch the static obstacles. Here, a lateral obstacle safety buffer (obstacle_lat_buffer, line 2) is conservatively applied to the leftmost and rightmost boundary calculation (line 9, 11) on top of the detected obstacle boundaries from the perception module, to ensure that the vehicle can keep enough distance from the obstacles.

**BP DoS vulnerability.** In Apollo 5.0, the obstacle safety buffer above is set to a constant value, 0.4m, regardless of obstacle positions in the driving environment. Meanwhile, the minimal lane width is 2.7m in urban areas [66], and thus
static obstacles out of the boundaries of such lanes can in the extreme case reduce the lateral drivable space between \textit{left\_bound} and \textit{right\_bound} to 1.9m $(2.7 - 0.4 \times 2)$. This is actually narrower than most vehicle models popularly used for AD today, e.g., 2.11-2.14m (with mirrors) for Lincoln MKZ, Lexus RX, Jaguar I-Pace, etc. [67–70]. As shown in Fig. 3, the default value in Apollo is set to 2.11m (line 3). This means that with such logic a lane can be considered as blocked \textit{even when its designed drivable space is completely empty with no static obstacles invading its lane boundaries}. For a single-lane road, this means that the whole road is considered blocked and thus the AD vehicle will permanently stop at the current lane, as illustrated in Fig. 4 (B). This is a decision flaw for BP as such driving behavior is \textit{too conservative} to the extent that it \textit{directly} violates common driving norms. For example, the requirement that a vehicle “should never block the normal and reasonable movement of traffic” [71], which can result in the vehicle being cited if this requirement is violated, especially when such a violation occurs inside a tunnel or 15 ft within a fire station’s driveway [72].

Based on the code logic, the root cause for such an overly-conservative BP decision flaw is in the use and setting of the obstacle safety buffer. We found that such a flaw is not specific to Apollo: our system discovers that Autoware, another popular open-source full-stack AD system [30], has similar flawed BP logic even though the concrete implementation is quite different from Apollo’s. Also, Autoware is even more conservative in such buffer settings (§V-B). This thus conforms to our general design hypothesis in §II-B that BP for practical AD systems tends to be as conservative as possible, leading to a general susceptibility to semantic DoS vulnerabilities.

\textbf{Exploitation.} To exploit this vulnerability in the real world, an attacker simply needs to prepare 2 easy-to-carry static objects that are not too uncommon in road regions, e.g., cardboard boxes, and place them close to but still off the lane boundaries on each side of a single-lane road, as illustrated in Fig. 4 (B). This is seemingly benign as these boxes are not blocking road and such randomly-dumped garbage on road side is not entirely uncommon. However, it can cause the AD vehicle with the vulnerable BP logic above to get permanently stuck at this road position and block traffic. Note that these 2 boxes do not have to be at exactly the same longitudinal position; from the code, they can be up to 5m (in the latest Apollo version) apart in longitudinal direction while still causing such a permanent stop decision, which can make such exploitation look more benign and thus stealthier. We did not include such longitudinal direction logic in Fig. 3 for the ease to understand the key vulnerable logic. Fig. 4 (C) shows a snapshot of our end-to-end simulation case study of this exploit for Autoware (detailed in §VI). As shown, the two boxes are clearly off road and far from blocking the road, but the AD vehicle is forced to permanently stop there.

\textbf{B. Design Challenges}

Motivated by the concrete example above and similar problems observed in real-world production AD settings today (§II-B), it is highly desired to develop a systematic approach to discover such BP DoS vulnerabilities at the AD system development stage so that the developers can proactively find and fix them before deployment.

Recently, property-based testing have achieved great success in discovering safety violations in AD software [10, 28, 73–77]. We follow the same general framework to develop a tool which can systematically discover DoS vulnerabilities in AD software. In the motivating example, the property we aim to falsify can be formally expressed as:

\[ \text{IsSafeToDrive} \rightarrow \neg \text{Stop} \]  

The above property indicates that when the current lane is safe to drive, the vehicle should be able to normally follow the lane instead of getting stuck by irrelevant physical objects. Even though the property seems to be simple at the first glance, none of previous works can be directly applied to solve this problem due to the following design challenges:

\textbf{C1. Lack of testing oracles for semantic DoS vulnerability in BP.} For our vulnerability definition (§II-B), a key challenge is how to judge whether the current situation is safe to perform a certain planning behavior. For example, in our motivating example in §III-A, we need to decide the value of predicate IsSafeToDrive in Eq. 1. Previous works [10, 28, 73–75] focus on the safety properties, especially collision, which can be directly extracted from official documentations or easy to define. But when it comes to studying DoS properties, there is no clear boundary of whether it is safe for the vehicle to drive due to the uncertainty and complexity of the environment. The first challenge is that we need to \textit{concretize} the predicate IsSafeToDrive in Eq. 1 into expressions which can be directly computed from planning inputs.

\textbf{C2. Need to systematically generate attack inputs following complicated problem-specific physical constraints.} To discover our BP DoS vulnerability, the dynamic testing process needs to effectively generate attacker-introduced physical-object properties (e.g., positions) that can (1) follow basic traffic rules and driving norms as required in §II-C to achieve high attack practicality and stealthiness, e.g., a moving attack vehicle should drive in a lane following the road direction, instead of on pavements or in the wrong direction; and (2) make sure the value of predicate IsSafeToDrive is true, since we can only find counterexamples when this predicate is true. Both constraints require to resolve complicated problem-specific geometry constraints; the closest solution so far is from Scenic [78], which can generate test inputs within several generic geometric constraints in AD context (e.g., certain distances of a road object to curb). However, it still cannot address the more complicated ones specific to our problem context. For example, since we specifically want to generate road objects that should not affect the ego vehicle driving decision, their geometry constraints are inherently dependent on the planned trajectory of the ego vehicle, e.g., cannot be on or have any intention to move to any lanes that the ego vehicle plans to drive on (PI-C1, C4, C5 in Table I). However, Scenic’s current design does not consider such dependencies.
C3. Need to obtain more fine-grained feedback from the planning decision-making code level (i.e., decision code branches) to guide our vulnerability discovery. For automated software vulnerability discovery in general, existing dynamic testing methods popularly obtain code-level feedback to guide the discovery process, which shows superior effectiveness over treating the software as a black-box [79–83]. In the general CPS testing domain, prior works have used quantitative feedback such as the robustness metrics to guide testing [28, 76], but such a guidance still treats the code-level decision logic (i.e., decision-making code branches) as black-box, which thus cannot provide guidance once the overall planning output stays unchanged. In our problem setting, it is desired if we can improve this with more fine-grained code-level guidance such as the distance between the current inputs and the planning decision boundary at the code branch level. For example, in Fig. 3, the guidance can be more effective if we can leverage the value distance between left_bound and right_bound at the decision code branch at line 13.

IV. DESIGN: PLANFUZZ

In this paper, we are the first to address the 3 challenges in §III-B by designing an automated approach to systematically discover BP DoS vulnerabilities (§II-B), called PlanFuzz.

Design goal. The goal of PlanFuzz is to discover previously-unknown semantic DoS vulnerabilities defined at the BP decision code level (an example is in §III-A). Note that our current focus is not on the comprehensive identification of the triggering scenarios for the discovered vulnerabilities; nevertheless, a few concrete triggering scenarios will come with the discovery to provide the vulnerability triggerability since we adopt a dynamic testing framework (detailed below).

A. Overview of Key Designs

At a high level, PlanFuzz follows an evolutionary testing framework, which is widely adopted by prior works on domain-specific vulnerability discovery with high generality, effectiveness, and efficiency [21, 26, 84–86]. To address the challenges in §III-B, the following key designs are introduced:

Planning Invariant (PI) as testing oracle. To address C1, we propose Planning Invariant (PI) as the problem-specific testing oracle for BP DoS vulnerabilities. PI has 3 components: planning scenario, constraints for physical objects, and desired planning behavior, which together define an invariant property for BP: In a planning scenario, the BP output should always conform to the desired planning behavior as long as the PI constraints for surrounding physical objects are satisfied. For example, for our motivating example in §III-A, the planning scenario is lane following; the PI constraints for physical objects are that the surrounding physical objects are all static and located outside the lane boundaries; the desired planning behavior is that the AD vehicle should keep driving forward. Such invariant properties are derived from common driving norms (e.g., off-road static obstacles should not be considered as blocking the road, detailed in §IV-C) so that its violations can be used to tell an overly-conservative decision.

PI-aware physical-object generation. To address C2, we need to design physical-object generation methods that conform to both the driving norms (e.g., traffic rules) and PI constraints above given a planning scenario and desired planning behavior. A direct solution direction is to directly generate random physical-object properties within these constraints. However, generating them following a random distribution is very difficult since these constraints are quite irregular in the real world (e.g., curvy and zigzag road boundaries), not to mention the problem-specific constraints due to the dependence on the planned trajectory of the ego vehicle (§III-B). To address this, our strategy is to first generate objects without considering these constraints, and then add a PI constraint enforcement step afterwards to adjust its properties (e.g., positions) for conformation. Specifically, here different PI constraint enforcement operators are designed following the diversity and inheritance principles desired for the random input generation step in genetic algorithms [87].

BP vulnerability distance. To overcome C3, we design BP vulnerability distance as the code-level feedback to measure how close the current BP execution is to violate a PI and thus trigger a BP DoS vulnerability. Specifically, such a distance is defined based on the control- and data-flow differences between the current executed code path and its closet path to a set of attack target positions, which are code positions indicating violations of the PI (for example, the callsite of API BuildStopDecision). To facilitate its run-time calculation during the dynamic testing, we first perform an off-line analysis of the BP code to (1) identify the key predicates that the attack target positions control- and data-dependent on, and compute information that can be pre-calculated about their run-time control- and data-flow distances to the target positions, which forms a BP vulnerability distance profile; and (2) instrument these predicates to collect their execution status, called BP vulnerability trace, in the run time. During the testing, based on the execution status of these key predicates collected in BP vulnerability trace, the control- and data-flow distances of the executed key predicates are calculated with the pre-computed information from BP vulnerability distance profile, which are then combined to calculate the final vulnerability distance.

Next, we describe the whole PlanFuzz system design incorporating these key designs, and then provide details for each.

B. PlanFuzz System Design

System input and output. Fig. 5 shows an overview of PlanFuzz system with the key designs above. As shown, the whole PlanFuzz system requires 3 inputs: (1) BP source code (we assume it is available since PlanFuzz is designed for AD developers); (2) BP input traces (including map) for the planning scenarios of interest; and (3) a set of PIs for these scenarios. The output is a set of test inputs that can trigger a BP DoS vulnerability. The AD developers can then utilize them to identify the vulnerable code logic and analyze root causes, with the goal of developing vulnerability fixes.

Manual efforts. Three types of manual efforts are required for using PlanFuzz: (1) Collecting BP input traces for the
targeted scenarios. Since PlanFuzz is designed for AD developers, we assume they have access to such traces from their AD system testing or operations; (2) identifying PIs for these targeted scenarios. Note that such identification is an one-time effort and there are general ones applicable across multiple scenarios identified later in §IV; and (3) identifying and annotating the attack target positions in the source code. For AD developers, we assume they can have high-level knowledge of the BP code (e.g., BP decision APIs and state variable classes) to identify these based on the desired planning behavior of PI and the state variables associated with the scenarios. In our experiments, it took less than 50 lines in total and less than 1 hour of manual efforts for an author who is experienced with the AD planning code.

System framework. With the inputs above, the vulnerability discovery process has 2 phases as shown in Fig. 5: (1) offline analysis and instrumentation, and (2) online BP vulnerability testing. On the left side of Fig. 5, the offline phase takes the BP source code and attack target positions based on PIs, and generate the BP vulnerability profile and instrumented BP code. On the right side of Fig. 5, the online phase follows the evolutionary testing framework in which each single physical object is considered as a “gene”. It customizes each component of a genetic algorithm: (1) fitness, for which we use the run-time BP vulnerability distance to measure how close a BP execution is to trigger a BP DoS vulnerability. It is calculated by executing the instrumented BP code in the planner executor and the fitness calculator in Fig. 5; (2) mutation and crossover, for which we use the PI-aware physical-object generation to mutate and exchange physical objects; (3) seed selection, for which we select test cases with smaller BP vulnerability distances as the new seeds.

To get started, the planner executor extracts the input frames corresponding to each BP decision from the BP input trace, and then feeds these frames to the instrumented BP code.

Table I: General PI (Planning Invariant) constraints across different driving scenarios from the full list of PIs identified and used in this paper (Table IV).

<table>
<thead>
<tr>
<th>Physical object type</th>
<th>PI constraints for physical objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static obstacle (cardboard boxes, parked bikes, etc.)</td>
<td>PI-C1. StaticOffRoad(x)</td>
</tr>
<tr>
<td>Vehicle</td>
<td>PI-C2. IsFollowingVehicle(x) PI-C3. IrrevalentVehicle(x)</td>
</tr>
<tr>
<td>Pedestrian</td>
<td>PI-C4. StaticOffRoad(x) PI-C5. DynamicOffRoad(x)</td>
</tr>
</tbody>
</table>

To maintain the consistency of internal BP states, we record a snapshot of the internal state variables beforehand for the initial seed, and recover it before feeding the testing inputs later. During the testing, the BP DoS vulnerability checker determines whether a certain generated testing case violates the PI; if so, it outputs the discovered vulnerability.

C. Planning Invariant

As introduced in §IV-A and shown in Fig. 5, each PI has 3 components: planning scenario, constraints for physical objects, and desired planning behavior, which together form the BP invariant properties that concretely define the overly-conservative BP decisions. In this paper, we consider 8 different planning scenarios commonly supported by industry-grade AD systems [29,30,88], covering various basic real-world driving scenarios, e.g., lane following, lane changing, intersections with stop signs and traffic signals, lane borrowing, bare intersection, and parking. The full list is in Table IV. Since we target semantic DoS vulnerabilities, the desired planning behavior for each scenario is usually to just keep the intended driving behavior, e.g., keep cruising for lane following, keep moving to pass the intersection, and finish the lane-change/borrow/parking actions.

Given a planning scenario and the desired driving behavior, the next is to identify the physical-world constraints of the attacker-introduced physical objects such that if satisfied, the BP logic should not give up the desired driving
behavior. We derive such constraints conservatively based on common driving norms, e.g., descriptions from the driver’s handbook [71] such as those quoted in §III-A. Specifically, in this paper we focus on the general constraints that are applicable to multiple driving scenarios, which are summarized in Table I and denoted as PI-C. For example, for static obstacles such as cardboard boxes and parked bikes, the ones that are completely off-road and without any violation of the boundaries of the lanes, which the AD vehicle plans to drive on, should generally not cause the BP logic to give up lane following, changing, borrowing, or passing an intersection. For pedestrians and vehicles, the ones moving in their commonly-designated regions (e.g., off-road pavement for pedestrians, and traffic lanes for vehicles) without showing any intention to move towards the AD vehicle or the lane regions the AD vehicles plans to drive one should not give up those desired driving decisions. Here for the simplicity, we define a list of functions StaticOffRoad(x), DynamicOffRoad(x), FollowVehicle(x), IrrelevantVehicle(x) to express the geometry relationship between the road network, the trajectory of a certain physical object x, and the trajectory of the AD vehicle. The complete set of PI-Cs for all planning scenarios and the formal definitions of the functions are in Table IV in Appendix. With the defined PI-Cs, we are able to concretize the high-level property in Eq 1 into the following format to describe the property in lane following scenario:

\[
\text{StaticCons}(x) := (x.\text{type} == \text{Static}) \rightarrow (\text{StaticOffRoad}(x))
\]

\[
\text{VehicleCons}(x) := (x.\text{type} == \text{Vehicle}) \rightarrow (\text{FollowVehicle}(x) \lor \text{IrrelevantVehicle}(x))
\]

\[
\text{PedestrianCons}(x) := (x.\text{type} == \text{Pedestrian}) \rightarrow (\text{DynamicOffRoad}(x) \lor \text{StaticOffRoad}(x))
\]

\[
\land \left( x.\text{type} \in (\text{StaticCons}(x)) \land (\text{VehicleCons}(x)) \right) \land (\text{PedestrianCons}(x)) \rightarrow \neg \text{Stop}
\] (2)

Here O is the set of physical objects and x is a physical object in the set. We define the constraints for each type of objects and merge them in the end to define the availability property.

D. PI-aware Physical-Object Generation

After concretizing the property, the next step is to generate the inputs that can always satisfy the constraints in planning invariant. In other words, we want to make sure that the left side of Eq. 1 is always true. We design PI-aware physical object generation to satisfy this requirement. Due to the page limit, we leave most of the details in our extended version [89]. The input generation contains three main steps:

- **Static property initialization and mutation.** In this step, we first randomly generate the static properties of the objects, e.g., position, type, and size, without considering PI-Cs during the testing input initialization and mutation processes. For each generated physical object, we will assign an appropriate size given the randomly generated object type.

- **PI-constraints enforcement.** The second step is to change the position and heading of each physical object to enforce the position correctness for each object. The high-level idea of this enforcement step is to adjust the property value to the closest one that does not violate PI-C. For example, moving a static obstacle, which violates the lane boundaries, to the closest off-lane position. We also change the lateral position and longitudinal position separately to keep the diversity and inheritance during the evolutionary algorithm.

- **Dynamic property generation.** The last step is to add dynamic properties (e.g., speed and moving trajectory) for dynamic physical objects such as driving vehicles and walking pedestrians. Here, the generated properties need to ensure (1) satisfying the driving norms (e.g., traffic rules) as in §II-C, and (2) conforming to the related PI-Cs such as not showing any intention to move towards the AD vehicle or the lanes it plans to drive on (PI-C3,5 in Table I).

E. BP Vulnerability Distance

The BP vulnerability distance is calculated based on information from both offline BP vulnerability distance profile and runtime BP vulnerability trace. If we recall the example code in Fig. 3, our goal here is to find a set of physical objects, which satisfy the constraints of PI, and make the program execution reach line 14. Inspired by directed greybox fuzzing [90], we design a distance metric to quantify the distance between current execution trace and the attack target positions. The directed greybox fuzzing techniques so far generally only consider control flow distance defined in [90]. In this paper, we further add a data flow distance into our distance metric design. This is motivated by our observation: (1) The BP decision-making is usually based on a list of predicates with floating-point number comparisons. Thus, measuring the difference between floating points operands can help guide the testing in a more fine-grained way. (2) The control-flow changes in our problem setting is not very significant. For example, in the path bounds case shown in §III-A, changing the position of a certain obstacle will not change the overall control flow unless a different decision is made.

Specifically, in the offline analysis and instrumentation phase as shown in Fig. 5, we first build a Program Dependence Graph (PDG) of the BP code and then use it to identify all the predicates that the attack target positions control- and data-dependent on. We further divide the predicates into two parts: critical predicates and non-critical predicates. Critical predicates refer to the predicates that connect themselves with
the attack target position with control dependence edges only. On the other side, non-critical predicates refer to the predicates that must connect with the attack target via data dependence edges. Examples for these two types of predicates based on our motivating example are included in our extended version [89].

For each of these predicates, we calculate their individual control- and data-flow distances, and use the sum as the overall BP vulnerability distance. Our control-flow distance design is similar to that in prior directed grey-box fuzzing methods [90], and thus next we focus on explaining our data-flow distance design using run-time collected BP vulnerability traces. For critical predicates, our offline phase determines which branch is reachable or closer to the target positions. In the runtime, we calculate how close the execution is to triggering such a branch using the differences between the operands and the execution number of each branch. For non-critical predicates, since we do not know which branch is the closer branch towards the target positions, our strategy is to minimize the difference between operands to get more diverse execution traces.

We use the motivating example in §III-A to demonstrate the benefit of this design. As illustrated in Fig. 6, we fix the position of one static object right next to the left lane line and 15m in front of the AD vehicle, and at the same time, we arbitrarily move the other static object and measure the robustness distance metric used in [28,76,77] and BP vulnerability distance proposed by our paper. In Fig. 6 (a)(b), the robustness metric can only give a boolean guidance on whether the decision is changed. However, our distance metric in Fig. 6 (c)(d) can guide the position of the second object into the vulnerable area due to that the operand distance of predicate on Line 8 and Line 13 in Fig. 3 becomes smaller when the object is approaching to the vulnerable area.

V. Evaluation

A. Evaluation Setup

Subject BP implementations. We evaluate PlanFuzz on the BP code from 2 open-source AD systems, Baidu Apollo [29] and Autoware [30]. Both are practical full-stack AD systems that can be readily installed on real vehicles for driving on public roads [91,92], and also have representativeness for industry-grade AD systems as Baidu has been recently ranked among the top 4 leading industrial AD developers with Waymo, Ford, and Cruise [31] and has been providing self-driving taxi services in China for months, while Autoware is adopted by the USDOT in their AD vehicle fleet [32].

Specifically, we select the BP implementations in 2 Apollo versions, 3.0 and 5.0, as their design and implementations are significantly different based on the release log [93]. We did not evaluate on 6.0, the latest Apollo version, as it only made minor changes to 5.0 in BP [93]. We have confirmed that all the discovered vulnerabilities from 5.0 also exist in 6.0. The Autoware BP implementation we evaluate on is from Autoware.AI 1.14.0 [30,94], the latest version with an implementation of the open planner [95]. Both Autoware and Apollo use rule-based logic to make decisions. Their main difference is that Autoware’s BP adopts a single finite state machine-based design where all planning behavior changes are modeled as state transitions, while Apollo’s is more modular, which first decomposes the whole driving decision making into independent primitive tasks (e.g., obstacle avoidance, lane changing, intersection passing, velocity selection, etc.) and then selects the appropriate ones based on different driving scenarios (e.g., lane following, intersection, stop sign).

BP input trace collection. We use LGSVL, a production-grade AD simulator [96], to collect the BP input traces as the initial testing seed (§IV-B). The benefit of using a simulator to generate seeds is that it is easier to (1) create different planning scenarios to increase testing diversity, and (2) control the scenario to prevent any irrelevant physical objects from affecting the generation of desired planning behavior. Note that such simulation-based testing is widely used in the AD industry for flexibility, scenario coverage, and safety [97–99]. LGSVL itself is also designed for performance and safety testing of production AD systems [96].

In total, 40 traces are collected under 8 different driving scenarios (5 per scenario). For each scenario, the 5 traces have diversity in driving tasks (e.g., drive straight or make turns) and road layouts (e.g., width of the local road’s lane width or the highway). Each trace spans up to 47sec with 100-2400 BP decisions, and the input frame for each BP decision is used as an individual testing seed. These traces lead to 28,789 (9,676 for Apollo, 19,113 for Autoware) different initial testing seeds in total (3,598 per scenario on average) used in our evaluation. Details are in our extended version [89]. In these traces, both the AD vehicle itself and other traffic participants are behaving normally/correctly (e.g., follow traffic rules and driving norms, and can correctly execute the designed driving maneuvers).

Test input generation. For each initial testing seed, PlanFuzz generates the attack’s physical objects as described in §IV-D and injects them into the planning input. Here, we directly use their ground-truth physical properties (e.g., type, bounding-box size and shape), which can thus avoid finding violation cases due to errors in upstream modules (e.g., perception) instead of BP. As described in §IV-D, for each attack object, PlanFuzz initializes and mutates their properties within common feasible ranges according to their types. For example, the positions of generated objects is within 80m to the ego vehicle (a common range of AD perception [100]), the sizes for static objects are 0.5-2m each dimension, and those for pedestrians and vehicles are the same as the default ones used in the simulator. Details are in our extended version [89]. Note that consistent with our threat model (§II-C), we do not mutate the non-attacker-controllable planning inputs such as weather conditions and the ego vehicle’s driving speed; all of them just inherit the valid values from original BP input traces.

Evaluation metrics and setup. We consider a vulnerability as discovered when any attack target position is triggered (§IV-B). We consider a discovered BP DoS vulnerability unique if its code-level decision logic (branches) that causes such vulnerability (e.g., those in §III-A) is different from the others, which is similar to unique crashes in traditional fuzzers [101–104]). For each initial seed, we run PlanFuzz
multiple times to increase the chance of finding unique vulnerabilities. For each run, the testing terminates when either a vulnerability is found, or the optimal fitness value is unchanged for 100 generations. We manually verified each discovered vulnerability and did not find any false positives.

B. Vulnerability Discovery Effectiveness

Throughout our experiments, PlanFuzz discovered 9 unique BP DoS vulnerabilities in Apollo and Autoware as shown in Table II. All these vulnerabilities can be exploited to adversely delay the progress of the AD vehicle from reaching its destination; most of them can cause the AD vehicle to permanently stop on the road. We classify the vulnerabilities into three types based on the attack scenarios. In this section, we provide a summary of the attack scenarios, including (1) the driving scenarios and symptoms of the relevant vulnerabilities, (2) the violated PIs, (3) the root causes, and (4) the potential real-world exploitations. Pseudo code and detailed analysis of the vulnerabilities can be found in our extended version [89].

Attack scenario 1: Lane following DoS attack. In this scenario, the AD vehicle keeps cruising in the current lane while static or dynamic obstacles located outside of the current lane boundaries. Leveraging V1/V8/V9, the attacker can cause the AD vehicle to decelerate or permanently stop in the current lane, which effectively prevents the AD vehicle from reaching the destination. Here, since the designed drivable space (i.e., the current lane) is completely empty with no obstacles invading the lane boundaries, such BP decisions thus violate PI1 or PI2 depending on the road structure (i.e., single- or multiple-lane road). As discussed in §III-A, the root cause is the setting and usage of the lateral obstacle safety buffer, which leads to the overly-conservative BP decisions.

To exploit V1 or V8, the attacker needs to find a single-lane road (for V1, the lane width needs to match the corresponding requirement described in §III-A) and place static obstacles close to the lane boundaries. The AD vehicle will then permanently stop in front of the static obstacles. To exploit V9, multiple attackers can coordinate to drive two vehicles in front of the AD vehicle on the lanes other than AD vehicle’s current lane to trigger a deceleration decision.

Attack scenario 2: Intersection passing DoS attack. The third type of attack happens when the AD vehicle is approaching an intersection. V5–7 belong to this type and enable the attacker to cause the AD vehicle to stop in front of the crosswalk or even permanently stop before the stop sign. Since the static objects are all off-road and the dynamic objects’ movement will not affect AD vehicle’s planning behavior, the stopping decisions produced by BP thus violate PI5 and PI6. When passing the crosswalk, the AD vehicle needs to make sure there is no pedestrian inside the crosswalk or with the intention to move into the crosswalk. However, due to the overly-conservative distance checking between the AD vehicle’s driving path and a standing pedestrian (V5) or trajectory collision checking between AD vehicle and a moving pedestrian (V6), the BP decides to stop before the crosswalk despite its driving path is in fact clear. For the intersection with stop signs, the BP maintains a watch list for the objects that arrive earlier such that it can proceed following a first-come first-serve convention. However, due to the overly-conservative distance threshold to the closest lanes when considering which objects it should wait for, the BP mistakenly includes parked bikes off the road into the watch list. Since these bikes are static, the AD vehicle keeps waiting for them and thus permanently stops before the stop line.

By exploiting V5 and V7, the attacker can cause the AD vehicle to permanently stop at the intersection. To achieve that, the attacker only needs to have a standing pedestrian or parked bikes around the intersection. For V6, since the pedestrian must be moving and will eventually leave the intersection, the attacker can thus carefully control the movement of the pedestrian such that the AD vehicle continuously applies a large deceleration, which may pose safety threats to the passengers and other vehicles (§II-B).

Attack scenario 3: Lane-changing DoS attack. This type of attack happens when the AD vehicle is about to change lanes or borrow the reverse lane due to the routing requirement or a blocking static obstacle. By exploiting V2–4, the attacker is able to use non-blocking static obstacles or following vehicles to prevent the AD vehicle from performing the desired lane-changing or lane-borrowing behaviors. As the

<table>
<thead>
<tr>
<th>Vuln #</th>
<th>Driving Scenario</th>
<th>Software</th>
<th>Violated PI #</th>
<th>Attack-influenced Planning Behavior</th>
<th>Triggering Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>Lane following</td>
<td>Apollo 3.0/5.0</td>
<td>PI1 (PI-C1)</td>
<td>Permanent stop</td>
<td>Static obstacle</td>
</tr>
<tr>
<td>V2</td>
<td>Lane changing</td>
<td>Apollo 3.0/5.0</td>
<td>PI3 (PI-C2, 3)</td>
<td>Fail to change lane and never reach destination</td>
<td>Static obstacle</td>
</tr>
<tr>
<td>V3</td>
<td>Lane borrow</td>
<td>Apollo 3.0/5.0</td>
<td>PI4 (SP-PI-C1, 2)</td>
<td>Fail to borrow the lane and permanent stop</td>
<td>Vehicle or static obstacles in front of blocking vehicle</td>
</tr>
<tr>
<td>V4</td>
<td>Lane borrow</td>
<td>Apollo 3.0/5.0</td>
<td>PI4 (PI-C1, 4, 5)</td>
<td>Fail to borrow the lane and permanent stop</td>
<td>Off-road static obstacle</td>
</tr>
<tr>
<td>V5</td>
<td>Intersection w/ traffic signal</td>
<td>Apollo 3.0/5.0</td>
<td>PI6 (PI-C4)</td>
<td>Fail to pass intersection and permanent stop</td>
<td>Static pedestrian</td>
</tr>
<tr>
<td>V6</td>
<td>Intersection w/ traffic signal</td>
<td>Apollo 3.0/5.0</td>
<td>PI6 (PI-C5)</td>
<td>Emergency stop; possible to cause permanent stop and thus fail to pass intersection</td>
<td>Pedestrian who is leaving the intersection</td>
</tr>
<tr>
<td>V7</td>
<td>Intersection w/ stop sign</td>
<td>Apollo 3.0/5.0</td>
<td>PI5 (PI-C1)</td>
<td>Fail to pass intersection and permanent stop</td>
<td>Static bicycle off the road</td>
</tr>
<tr>
<td>V8</td>
<td>Lane following</td>
<td>Autoware</td>
<td>PI1 (PI-C1)</td>
<td>Permanent stop</td>
<td>Static obstacle off the lane</td>
</tr>
<tr>
<td>V9</td>
<td>Lane following</td>
<td>Autoware</td>
<td>PI1 (PI-C3)</td>
<td>Emergency stop</td>
<td>Moving vehicle off the lane</td>
</tr>
</tbody>
</table>
changing and borrowing lanes are clear in such scenarios, the vulnerabilities thus make the BP violate PI3 and PI4 for the lane-changing and borrowing scenarios. Specifically, V2 is caused by overly-conservative design when checking if the AD vehicle’s future driving path overlaps with other vehicles during the lane-changing. V3 is caused by wrong judgement on whether it is necessary to borrow the lane. Although PlanFuzz mainly aims to find BP DoS vulnerabilities introduced by overly-conservative planning decisions, V4 is likely due to an implementation bug when checking whether the perception range is blocked by any obstacle before performing lane borrowing. More details of the vulnerabilities are in our extended version [89].

The attacker can exploit V2 by driving a vehicle tailgating the AD vehicle in the same lane. As long as the attacker’s vehicle is close to the left lane line (but without touching the lane line), the AD vehicle will mistakenly interpret that the changing lane is blocked and give up the lane-changing attempt, which in the worst case can cause significant delays for the AD vehicle to reach its destination if the attacker keeps performing such attack. To exploit V3 and V4, the attacker first needs to find a lane borrowing condition where the road is blocked by a static obstacle (e.g., a truck which is unloading the cargo). Second, the attacker can park another vehicle in front of the truck (V3) or simply place an off-road cardboard box 5m away from the AD vehicle (V4).

C. Baseline Comparison

Since there is no existing alternative fuzzer that directly performs BP DoS vulnerability discovery, we evaluate the benefits of our designs by replacing important design components in PlanFuzz with possible baseline designs. Specifically, through such an evaluation we aim at answering the following methodology-level research questions (RQ):

**RQ1.** Can our BP vulnerability distance design (§IV-E) provide effective guidance to benefit the vulnerability discovery?

**RQ2.** Can our PI-aware physical-object generation design (§IV-D) benefit the vulnerability discovery?

**RQ3.** Can traditional fuzzing techniques without using any of our problem-specific fuzzing designs also effectively discover BP DoS vulnerabilities?

**Baseline setups.** To answer RQ1, we create a baseline setup, PlanFuzz−guide, that replaces the BP vulnerability distance-guided genetic algorithm with random sampling (while still using all other PlanFuzz components such as PI-aware physical-object generation). For RQ2, we create another baseline setup, PlanFuzz−PI, that keeps the BP vulnerability distance design but remove the steps that enforce PI constraints in the generated attack objects’ static and dynamic properties, which are the key problem-specific designs in PI-aware physical-object generation (§IV-D). For RQ3, we remove both BP vulnerability distance and PI-aware physical-object generation designs; we directly use Protobuf-mutator for the entire test input generation process (denoted as PB-M). Protobuf-mutator is a readily-available fuzzer designed specifically for the data structure of protobuffers [105], which is directly compatible with the BP input data structure in both Apollo and Autoware.

**Evaluation setup and metrics.** We use the initial testing seeds that allow PlanFuzz to discover the 9 vulnerabilities in Table II to perform this baseline evaluation. For each seed, we run each of the 4 setups (full PlanFuzz and the 3 baselines) for 10 times, each time for 24 hours, to avoid variance as suggested by prior work [106]. For each fuzzer setup, we measure the discovered unique vulnerabilities (defined in §V-A), and their average Time-to-Exposure (µTTE), i.e., the time taken to discover them. When comparing a given baseline setup with the full PlanFuzz, we also use statistical testing following the suggestions in [90, 106, 107]. Specifically, we use the Vargha-Delaney statistic ̂A12 to measure the effect size, and use Mann-Whitney U [107] to measure the statistical significance of the µTTE performance drop (recommended for assessing randomized algorithms like fuzzers [90, 107]).

**Results.** As shown in the last column of Table III, PB-M, the setup without using any of our main problem-specific designs, fails to detect any of the 9 BP DoS vulnerabilities within 24h, which is thus at least 57× less efficient/effective. This concretely shows that our two main problem-specific fuzzing designs, BP vulnerability distance (§IV-E) and PI-aware physical-object generation (§IV-D), are necessary for effectively discovering BP DoS vulnerabilities (RQ3).

For PlanFuzz−guide and PlanFuzz−PI, the setups that each still retains one of these two problem-specific designs, the same set of unique vulnerabilities (as full PlanFuzz) are still discoverable within 24h. However, their discovery efficiency degrades substantially compared to the full PlanFuzz. Specifically, for almost all vulnerabilities (7/9 for PlanFuzz−guide, and 8/9 for PlanFuzz−PI), the µTTE performance drops are statistically significant (bold ̂A12 values in Table III). From the µTTE values, PlanFuzz−guide and PlanFuzz−PI are on average >4.5× and >3.5× slower respectively; for complicated cases like V1, such degradation can be even over 7.7× for PlanFuzz−guide. Among the 9 vulnerabilities, V4 is the only one without significant µTTE differences for both PlanFuzz−guide and PlanFuzz−PI, likely because it is relatively easier to trigger by nature due to a likely range-checking bug, which can be seen by its much lower µTTE values (1 sec). However, even so, without both of these problem-specific designs (BP vulnerability distance and PI-aware physical-object generation), they still cannot be discovered by traditional fuzzers like PB-M even given 24h (Table III).

VI. EXPLOITATION CASE STUDIES

In this section, we provide three case studies on the BP DoS vulnerabilities discovered by our testing framework and demonstrate how an attacker can exploit the vulnerabilities to disrupt the normal driving behavior of the AD vehicle without raising suspicion. Specifically, we select one vulnerability from each of the attack scenarios categorized in §V-B. The case studies are conducted in an end-to-end way, where we create the concrete driving environments with the attack obstacles in an AD simulator, LGSVL [96], and simulate with
A. Lane Following DoS Attack on Autoware

Vulnerable decision logic. The lane following DoS attack on Autoware (V8) is able to change the AD vehicle’s lane following decision into an overly-conservative decision to permanently stop on a clear road, by exploiting the vulnerable decision logic in trajectory_evaluator in Autoware. This evaluator is designed to decide whether there is a candidate trajectory for AD vehicle to safely pass without crashing into a static obstacle. Similar to Apollo’s design illustrated in §III-A, Autoware predefined an overly-conservative lateral safety buffer of 1.2 meters between the AD vehicle and any obstacles. As described in §III-A, since the minimal urban lane width is 2.7m and typical width of AD vehicle (with mirrors) is larger than 2m, two static obstacles out of lane boundaries can block all the candidate trajectories for the AD vehicle.

Exploitation method. To exploit this vulnerability, the attacker can find a single-lane road up to 4.35m wide (applicable to both common local and highway roads [66]), and put two easy-to-carry objects, e.g., cardboard boxes, on each side of the road. There is no strict relative longitudinal position requirement for them, as long as they are in the same planning range (35m for Autoware). Since Autoware uses a more conservative safety buffer size than Apollo, for a narrow road width such as 2.7m [66], each of such object can be >80cm to the road boundary to make them look more stealthy. An example setup in the simulator is shown in Fig. 4 (C). This can cause an emergency stop and/or permanent stop of the victim AD vehicle and thus damage the AD service, block traffic, and also potentially damage road safety (from emergency stop).

End-to-end attack results. We set up the above exploitation scenario in the simulator. From the simulation, after detecting the cardboard boxes, the AD vehicle immediately starts to decelerate and finally comes to a complete stop at ~10m in front of the boxes. After the full stop, we keep the simulator and AD system running to observe if the AD vehicle will move forward. After waiting for ~30min, the vehicle still stops at the original position. We manually examined the code and confirmed that this will be a permanent stop. A demo video is at our website [33] and the snapshot when the AD vehicle stops in front of the boxes is in Fig. 4 (C). As shown, the cardboard boxes are located very far from the lane boundaries and the lane is completely clear to drive from the driver’s view.

Such a vulnerability can lead to severe congestion and even rear-end collisions if the following vehicle is not able to react in time; video demos are at our website [33]. The attack consequence can be especially severe if launched at critical road segments such as highway exit ramps, or in front of police or fire stations (e.g., to block emergency responder actions).

Physical-world experiment. We further collect attack traces in real world to justify the attack realism. We conduct the experiment with a Lincoln MKZ [108] equipped with Velodyne VLP-32C LiDAR [109] and NovAtel Positioning Kits [110]. We mark a traffic lane with 3.5m width inside the parking lot and use a cardboard box and a trash can to construct the attack scenario. Due to the safety concern, we manually drive the car following the traffic lane to collect the attack trace. We run the Autoware’s LiDAR clustering and tracking nodes to get the object detection results and launch the op_planner nodes to get the planning decisions. For all the frames in our collected traces, the two static obstacles can be detected correctly, and the stop decision is made for every single frame in the traces. Thus, we deemed the Autoware’s lane following vulnerability reconstructable in real world. Fig. 7 shows our experiment setup and result.

B. Intersection Passing DoS Attack on Apollo

Vulnerable decision logic. This DoS vulnerability (V7) appears in Apollo, where the attacker can force an AD vehicle to stop permanently in front of an intersection with a 4-way stop sign. Apollo BP follows the “first come, first serve” traffic principle when the AD vehicle arrives at a 4-way stop sign. The vehicle maintains a watch list containing all vehicles and bicycles which reaches the intersection earlier than itself and waits until all vehicles and bicycles on that list have left the intersection. However, due to an overly-conservative distance threshold (5m), which is much larger than even the typical highway lane width (3.6m [66]), when judging if a vehicle or bicycle is on a lane and waiting for a stop sign, a parked bicycle that is not on the road will be mistakenly added to the watch list, which causes the AD vehicle to unnecessarily wait for off-road objects that are irrelevant to the stop sign-based intersection passing norms.

Exploitation method. To exploit this, the attacker can find any stop sign-based intersections, and place road-side parked bikes as long as they are within 5m from the lane center of any lanes in the intersection. Here, 2 parked bicycles are enough

<table>
<thead>
<tr>
<th>Seed</th>
<th>Uniq.</th>
<th>PlanFuzz</th>
<th>PlanFuzz-\textsuperscript{guide}</th>
<th>PlanFuzz-\textsuperscript{PI}</th>
<th>PB-M</th>
<th>( \mu TTE )</th>
<th>( \mu TTE )</th>
<th>( \mu TTE )</th>
<th>( \mu TTE )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>V1</td>
<td>165s</td>
<td>1278s (7.74\times)</td>
<td>0.97 276s (0.67\times)</td>
<td>0.88 NF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>V2</td>
<td>19s</td>
<td>21s (1.11\times)</td>
<td>0.55 117s (0.65\times)</td>
<td>0.98 NF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>V3</td>
<td>16s</td>
<td>34s (2.12\times)</td>
<td>0.88 53s (3.31\times)</td>
<td>0.88 NF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V4</td>
<td>1s</td>
<td>1s (1.00\times)</td>
<td>0.57 2s (2.00\times)</td>
<td>0.58 NF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>V5</td>
<td>47s</td>
<td>92s (1.95\times)</td>
<td>0.89 167s (3.55\times)</td>
<td>0.98 NF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V6</td>
<td>35s</td>
<td>78s (2.22\times)</td>
<td>0.83 148s (4.22\times)</td>
<td>0.93 NF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>V7</td>
<td>45s</td>
<td>119s (2.64\times)</td>
<td>0.97 208s (4.62\times)</td>
<td>0.96 NF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>V8</td>
<td>53s</td>
<td>193s (3.64\times)</td>
<td>0.96 327s (6.16\times)</td>
<td>0.90 NF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>V9</td>
<td>37s</td>
<td>57s (1.54\times)</td>
<td>0.93 188s (5.08\times)</td>
<td>0.96 NF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>46s</td>
<td>208s (4.52\times)</td>
<td>0.83 165s (3.58\times)</td>
<td>0.89 NF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
to bypass all timeout mechanisms in Apollo BP. This can then force the AD vehicles to stop in front of the stop line forever.

**End-to-end attack result.** We set up the exploitation scenario above in the simulator. Fig. 8 shows the snapshots of the benign and attack demos with and without the 2 parked bicycles. In the benign scenario, the AD vehicle can smoothly proceed and pass the intersection with stop signs. However, in the attack scenario, after the AD vehicle arrives at the intersection, it became stuck in front of the stop sign due to the two roadside parked bicycles despite the intersection being completely empty without any other vehicles.

C. **Lane Changing DoS Attack on Apollo**

**Vulnerable decision logic.** The lane-changing DoS attack on Apollo (V2) is able to force the AD vehicle to give up a lane-changing decision by exploiting decision logic in lane_change_decider of Apollo. As shown by the pseudo code in Fig. 10, the decision logic determines whether the target lane is clear for performing lane changing by checking if any vehicle occupies the target lane and is close to the AD vehicle. In the code, all the position variables (e.g., start_l, start_s) are in the Frenet coordinate relative to the target lane. Due to an overly-conservative lateral distance threshold (2.5m in line 5), a nearby vehicle following the AD vehicle or driving on the other adjacent lane will be considered as occupying the target lane and force the AD vehicle to give up the lane-changing decision.

**Exploitation method.** To exploit this, for lanes with \( \leq 3.55m \) width (applicable to almost all common local and highway lanes (up to 3.6m wide) [66]), the attacker just needs to drive a normal-size car (e.g., 2.11m as in §III-A) to follow a victim AD vehicle with close following distance (up to 8m). This can then prevent the AD vehicle from making lane changes forever, which can make it fail to arrive at the destination (especially critical for AD services such as robo-taxi/delivery). For lanes wider than 3.55m, the attacker just needs to drive with a slight deviation to the lane center (e.g., 5cm for a 3.6m-wide lane, which is far from touching the lane line (>70cm)) towards the victim’s lane changing direction.

**End-to-end attack results.** We set up the exploitation scenario above in the simulator. Fig. 9 shows the snapshots of the benign and attack demo videos. In the benign scenario, the AD vehicle can successfully change lanes since the following vehicle’s lateral position does not satisfy the vulnerability condition. However, in the attack scenario, although the changing lane is completely empty without any vehicles in the front or back, the AD vehicle still gives up the lane-changing decision to stay on the current lane, which causes it to miss the optimal route to the destination. For AD vehicles, such a BP decision often entails a re-routing step to recalculate a new route to reach the destination. However, since the attacker can simply keep following the AD vehicle and perform such attacks on every new route, it is possible for the AD vehicle to never reach the destination in the worst case.

VII. **Discussion and Future Works**

A. **Root Cause and Solution Discussions**

Among the vulnerabilities discovered, only 1 (V4) is likely an implementation bug, while the remaining 8 are due to the overly-conservative planning parameters/logic in judging safety. Specifically, V1, V8 are due to overly-conservative safety buffer configuration to road-side static objects; V9 is due to overly-conservative safe buffer configuration to moving vehicle trajectories in other lanes; V2, V3, V5, V6, and V7 are due to the overly-conservative logic in judging the intention of surrounding vehicles (V2, V3), pedestrians (V5, V6), and parked bikes (V7). More details are in extended version [89].

**Solution discussions.** Based on the causes, V4 can be potentially fixed by a bug patch; the other 8 are harder to fundamentally fix as it is non-trivial to effectively and practically balance the trade-off between safety and availability in an AD context. For example, considering the vulnerability causes are overly-conservative parameters/logic, a direct idea is to just make the design/implementation more aggressive, e.g., reducing the safety buffer configuration to road-side static objects. However, since such existing configurations may already be sufficiently tuned to ensure safety, changing them may compromise safety. One potential design direction for better addressing this trade-off is to consider dynamic configurations instead of fixed ones, e.g., dynamically adjusting the safety margin based on the velocity, since the required safety distance will decrease with a smaller velocity [111] (which is one reason why highway lanes are wider than local ones [112]). However, how to design such dynamic adjustment is non-trivial since various internal and external driving factors need to be systematically considered (e.g., internal ones such as vehicle size/speed, external ones such as the static/dynamic road conditions), which we thus leave as future work.

B. **Limitations and Future Work**

**Tool effectiveness.** PlanFuzz’s results do not have false positives since all the discovered vulnerabilities have been confirmed by the vulnerability checker. However, similar to all dynamic testing approaches, PlanFuzz cannot give any guarantee on the non-existence of the vulnerability and thus can have false negatives (FNs). Specifically, FNs can come from: (1) the evolutionary algorithm gets stuck at a local minimum or terminate too early. We plan to try other generic optimization algorithms in the future; and (2) the vulnerability only exists in a specific road condition (e.g., a specific road layout and/or surrounding traffic pattern) that is not included in our BP input traces. These vulnerabilities may be arguably less important though as their triggering conditions are narrower. To also capture these, one potential future direction is to also mutate the road conditions. However, how to ensure that the PI still holds after such mutations is a challenge.

**Applicability and generality.** Due to the limitation of available code bases, we only evaluate the applicability and generality of PlanFuzz on open-source AD systems. Nevertheless, the design of PlanFuzz is general at both design and
implementation levels. The design of PI and PI-aware physical object mutation does not make any assumption about the AD system designs and implementations under test as long as they are developed for driving on public roads. For the code instrumentation part, we use LLVM [113], which can support a variety of programming languages. Besides, our system should be able to use the gradient to replace BP vulnerability distance if learning-based planners becomes mainstream and easier to interpret, debug, and enforce safety measures in the future.

**Stronger threat models.** To trigger the semantic DoS vulnerabilities, adding road objects is not the only possible threat model. For example, attackers may also attack the perceptional sensors, such as by sensor spoofing [7] or compromising internal AD system components [114], to introduce malicious inputs to BP. However, the downside is that such attack vectors may also introduce new attack requirements/costs, e.g., sensor spoofing equipment [7] and access to internals or the supply chain of AD systems [114], making the vulnerability exploitation potentially less realistic/practical. We leave the systematic exploration of such a direction to future work.

VIII. RELATED WORK

**Autonomous Driving (AD) systems security.** Since AD systems heavily rely on sensors, prior works have studied sensor attacks in AD context such as sensor spoofing/jamming [6, 7, 115–118]. Besides sensor-level attacks, prior works also studied attacks and defenses of AD system components related to environmental sensing, such as object detection and tracking, localization, and lane detection [5, 7–9, 119–129]. However, so far none of them considered security problems specific to downstream modules such as BP like in this paper.

**Vulnerability discovery and property falsification in AD/RV (Robot Vehicle) software.** Recently, increasingly more works consider software vulnerabilities/bugs in AD systems [130, 131]. Some developed methods to discover semantic vulnerabilities in the DNN models in AD [7, 9, 23, 24, 119–122, 125, 126, 132, 133]. These methods assume differentiability of the test subject, which thus cannot be applied to the more industry-representative program-based BP targeted in this paper (§II-A). Previous works also falsify safety properties on AD/RV software [10, 28, 73–77, 134]. They cannot be directly applied since the problem scopes are different and the guidance is only limited to black-box guidance.

Vulnerability discovery in RVs, such as drones or rovers, is a closely-related research domain. Compared to AD, RVs typically follow the control commands sent by a base station without the need to make planning decisions by itself. Thus, existing works concentrate on control-specific vulnerabilities [25, 26, 28] or highly rely on control-specific knowledge [27, 129], which are orthogonal to the design challenges we need to address for BP-specific vulnerabilities in AD.

IX. CONCLUSION

In this paper, we design PlanFuzz, a novel dynamic testing approach to systematically discover BP DoS vulnerabilities under physical-world attacks. We propose and identify PIs as novel testing oracles, and design novel problem-specific fuzzing designs such as PI-aware physical-object generation and BP vulnerability distance. We evaluate PlanFuzz on 3 practical BP implementations, and find that it can effectively discover 9 previously-unknown semantic DoS vulnerabilities without false positives. We further perform exploitation case studies, and discuss root causes and potential vulnerability solution directions. We hope that our findings and insights can inspire effective solution designs in future works.

ACKNOWLEDGMENT

We would like to thank Ningfei Wang, Yunpeng Luo, Takami Sato, Junze Liu, and the anonymous reviewers for valuable feedback on our work. This research was supported in part by the NSF under grants CNS-1850533, CNS-1929771, CNS-2145493, CNS-1823262, and CMMI-2054710.

REFERENCES


“Waymo Self Driving Taxi Fumbles In Construction Zone, Blocks Traffic | JJRicks Rides with Waymo 54,” https://youtu.be/YUy3DuUOLhUE.


“Two People Killed in Fiery Tesla Crash with No One Driving,” https://tinyurl.com/2p9a3dat.


“First self-driving cars will be unmarked so that other drivers don’t try to bully them,” https://www.theguardian.com/technology/2016/oct/30/volvo-self-driving-car-autonomous.


“Simulation-based reinforcement learning for autonomous driving decision module design,” in *CoRL*, 2020.


“Learning to Drive in a Day,” in *ICRA*, IEEE, 2019.


We define such functionality as the following function:

\[ \text{coordinate system (UTM coordinate system in Apollo and transform the position of a physical object from a unified scenario.)} \]

When processing the geometry relationship among objects commonly used among different driving scenarios, note that more customized properties are needed for specific scenarios. When processing the geometry relationship among the objects, AD vehicle, and the road, a necessary step is to transform the position of a physical object from a unified coordinate system (UTM coordinate system in Apollo and Autoware) into a coordinate system relative to a certain lane.

We define such functionality as the following function:

\[ (s, l, \text{laneID}) = \text{transform}(\text{pos}, m) \] (3)

We define the function with input position \( \text{pos} \) in UTM coordinate system and the map \( (m) \). The outputs of this function are longitudinal position \( (s) \) and lateral position \( (l) \) relative to the closest lane, and the ID of the closest lane \( (\text{laneID}) \). We do not expand this function in a formal way since this involves more than one thousand lines of source code in Apollo.

We now introduce the constraints of physical objects in detail. For the static objects, the most common constraint is that the static objects should be off the road and should not intersect with any lane boundary. This can be formally defined based on a specific static physical object \( x \) and a set of polygon points of its boundary \( x.polygon \):

\[ \text{StaticOffRoad}(x) := \forall p \in x.polygon \ | transform(p, m).l > \text{halfLaneWidth} \] (4)

This means every polygon point of the object boundary is not within the range of any lane. The \( \text{halfLaneWidth} \) also needs to be queried from the map. For simplicity, we just describe it as a constant. For a pedestrian \( x \), we want to make sure that it will not touch any lane in its future moving trajectory and does not show any intention to enter the traffic lane or intersection. In this case, for a pedestrian \( x \), the waypoints set \( x.W \) to describe its moving trajectory, \( x.polygon(w) \) for the polygon points at waypoint \( w \), the constraints for the pedestrian can be defined as:

\[ \text{DynamicOffRoad}(x) := \forall w \in x.W, (\forall p \in x.polygon(w) | \text{transform}(p, m).l > \text{halfLaneWidth}) \land (\text{innerProduct}(x.\text{heading}, \text{directionTowardsLane}) < 0) \] (5)

The property \( \text{DynamicOffRoad} \) will check all the waypoints in the future moving trajectory of a physical object \( x \) and make sure that any points on the boundary shall not touch the lane boundary. Besides, it also checks the heading of the moving trajectory is not towards to the lane.

There are two possible constraints for a vehicle. First, a vehicle can simply follow the AD vehicle, and it should not affect the planning behavior. For simplicity, we also use a property \( \text{driveInLane}(x, \text{laneID}) \) to describe that the vehicle \( x \) keeps driving within the lane denoted by the lane ID \( (\text{laneID}) \). This can be modeled as:

\[ \text{FollowVehicle}(x) := \text{transform}(x.pos, m).\text{laneID} = \text{ego.laneID} \land (\text{transform}(x.pos, m).s + \text{safetyFollowingDistance} < \text{ego.s}) \land (x.\text{velocity} \leq \text{ego.velocity}) \land \text{driveInLane}(x, \text{transform}(x.pos, m).\text{laneID}) \] (6)

Another possibility is that the vehicle is driving normally on another lane. We formally define it as:

\[ \text{IrrelevantVehicle}(x) := \text{transform}(x.pos, m).\text{laneID} \neq \text{ego.laneID} \land \text{driveInLane}(x, \text{transform}(x.pos, m).\text{laneID}) \] (7)
<table>
<thead>
<tr>
<th>PI Index</th>
<th>Planning Scenario</th>
<th>Object Type</th>
<th>Constraints on Physical Objects</th>
<th>Desired Planning Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>PI1</td>
<td>Lane following (single-lane road)</td>
<td>Static obstacles</td>
<td>PI-C1. Off-road and w/o any violation of the boundaries of the lanes the AD vehicle plans to drive on</td>
<td>Keep cruising in the current lane</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Vehicles</td>
<td>PI-C2. Follow the AD vehicle</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pedestrians</td>
<td>PI-C3. Drive on reverse lane</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PI-C4+5. Off-road and w/o any intention to move towards to the AD vehicle or the lanes the AD vehicle plans to drive on</td>
<td></td>
</tr>
<tr>
<td>PI2</td>
<td>Lane following (multiple-lane road)</td>
<td>Static obstacles</td>
<td>PI-C1. Off-road and w/o any violation of the boundaries of the lanes the AD vehicle plans to drive on</td>
<td>Keep cruising in the current lane</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Vehicles</td>
<td>PI-C2. Follow the AD vehicle</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pedestrians</td>
<td>PI-C3. Drive on other lanes</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PI-C4+5. Off-road and w/o any intention to move towards to the AD vehicle or the lanes the AD vehicle plans to drive on</td>
<td></td>
</tr>
<tr>
<td>PI3</td>
<td>Lane changing</td>
<td>Static obstacles</td>
<td>PI-C1. Off-road and w/o any violation of the boundaries of the lanes the AD vehicle plans to drive on</td>
<td>Finish changing to the targeted lane</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Vehicles</td>
<td>PI-C2. Follow the AD vehicle</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pedestrians</td>
<td>PI-C3. Drive on other lanes except current and targeted lanes</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PI-C4+5. Off-road and w/o any intention to move towards to the AD vehicle or the lanes the AD vehicle plans to drive on</td>
<td></td>
</tr>
<tr>
<td>PI4</td>
<td>Lane borrow (due to a blocking obstacle)</td>
<td>Static obstacles</td>
<td>PI-C1. Off-road and w/o any violation of the boundaries of the lanes the AD vehicle plans to drive on</td>
<td>Finish borrowing the reverse lane and pass blocking vehicle</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Vehicles</td>
<td>SP-PI-C1. On-lane and in front of the blocking obstacle</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PI-C2. Follow the AD vehicle</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PI-C3. Drive on other lanes except current and targeted lanes</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SP-PI-C1. On-lane and in front of the blocking obstacle</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PI-C4+5. Off-road and w/o any intention to move towards to the AD vehicle or the lanes the AD vehicle plans to drive on</td>
<td></td>
</tr>
<tr>
<td>PI5</td>
<td>Intersection w/ stop sign</td>
<td>Static obstacles</td>
<td>PI-C1. Off-road and w/o any violation of the boundaries of the lanes the AD vehicle plans to drive on and the intersection the AD vehicle is going to pass</td>
<td>Pass intersection w/ stop sign following the traffic rule</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Vehicles</td>
<td>PI-C2. Follow the AD vehicle</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pedestrians</td>
<td>PI-C3. Drive on other lanes except current and targeted lanes</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PI-C4+5. Off-road and w/o any intention to move towards to the AD vehicle or the lanes the AD vehicle plans to drive on</td>
<td></td>
</tr>
<tr>
<td>PI6</td>
<td>Intersection w/ traffic signal</td>
<td>Static obstacles</td>
<td>PI-C1. Off-road and w/o any violation of the boundaries of the lanes the AD vehicle plans to drive on and the intersection the AD vehicle is going to pass</td>
<td>Pass intersection w/ traffic signal following the traffic rule</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Vehicles</td>
<td>PI-C2. Follow the AD vehicle</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pedestrians</td>
<td>PI-C3. Drive on other lanes except current and targeted lanes</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PI-C4+5. Off-road and w/o any intention to move towards to the AD vehicle or the lanes the AD vehicle plans to drive on</td>
<td></td>
</tr>
<tr>
<td>PI7</td>
<td>Bare intersection</td>
<td>Static obstacles</td>
<td>PI-C1. Off-road and w/o any violation of the boundaries of the lanes the AD vehicle plans to drive on and the intersection the AD vehicle is going to pass</td>
<td>Pass the bare intersection</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Vehicles</td>
<td>PI-C2. Follow the AD vehicle</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pedestrians</td>
<td>PI-C3. Drive on other lanes except current and targeted lanes</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PI-C4+5. Off-road and w/o any intention to move towards to the AD vehicle or the lanes the AD vehicle plans to drive on</td>
<td></td>
</tr>
<tr>
<td>PI8</td>
<td>Parking</td>
<td>Static obstacles</td>
<td>SP-PI-C3. Placed on other parking spots</td>
<td>Park into an empty targeted parking spot</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Vehicles</td>
<td>SP-PI-C4. Parked on other parking spots</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pedestrians</td>
<td>SP-PI-C5. Walking pedestrians moving away from AD vehicle</td>
<td></td>
</tr>
</tbody>
</table>